Entwicklung einer Schema-Evolutionskomponente für eine NoSQL-Datenbank

Masterarbeit

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik

vorgelegt von:	Marcel Apfelt
Matrikelnummer:	6200253
geboren am:	21.01.1986 in Güstrow
Erstgutachter/Betreuer:	Dr.-Ing. habil. Meike Klettke
Zweitgutachter:	Prof. Dr. rer. nat. Clemens H. Cap
Abgabedatum:	01.10.2014

Zusammenfassung

NoSQL-Datenbanken werden auf Grund ihrer Flexibilität hinsichtlich des Datenbankschemas vermehrt im Zusammenhang mit agilen Anwendungen verwendet, um Daten zu speichern, deren Struktur sich häufig ändert. Da die Datenbanksysteme selbst oftmals nicht über die Möglichkeit verfügen, das Schema der gespeicherten Daten zu verwalten, ist die Existenz einer externen Schema-Management-Komponente wünschenswert, die diese Aufgabe stattdessen übernimmt. Diese Arbeit befasst sich mit einer Teilaufgabe des Schema-Management, der Schema-Evolution. Für ein dokumentorientiertes NoSQL-Datenbanksystem (MongoDB) wird ein Konzept erarbeitet, um aus den Operationen einer gegebenen Evolutionssprache Datenbankupdates zu generieren, durch die sowohl ein gespeichertes JSON-Schema, als auch die dazugehörigen JSON-Dokumente angepasst werden. Als Ergebnis wird eine Schema-Evolutionskomponente in Form einer Bibliothek vorgestellt, mit deren Hilfe ein Evolutionsprozess auf einer MongoDB-Datenbank durchgeführt werden kann.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Für immer mehr Anwendungen stellen NoSQL-Datenbank eine echte Alternative zum klassischen, relationalen Modell dar. Neben dem effizienteren Umgang mit großen Datenmengen zeichnen diese sich häufig auch durch eine besondere Flexibilität hinsichtlich des Schemas der zu speichernden Daten aus. Aus diesem Grund eignen sie sich besonders zum Speichern von Daten agiler Anwendungen, deren Struktur entweder nicht eindeutig festgelegt ist, oder sich häufig verändert. Auf Grund der Tatsache, dass die meisten NoSQL-Datenbanken kein festes Schema besitzen und oftmals auch nicht über die Möglichkeit verfügen, eines zu definieren, liegt die Verantwortung über die Struktur der Daten in der Regel direkt bei der Anwendung.

Da der Umgang mit sehr heterogenen Daten in der Anwendung selbst äußerst aufwändig und ineffizient ist, resultiert daraus die Idee einer zusätzlichen Schema-Management-Schicht außerhalb der NoSQL-Datenbank. Die besondere Herausforderung einer solchen Schicht besteht darin, die Definition und Organisation eines Schemas für eine eigentlich schemafreie Datenbank zu ermöglichen. Des Weiteren zählt die Durchführung von strukturellen Änderungen im Rahmen der Schema-Evolution ebenfalls zum Schema-Management. In diesem Zusammenhang werden Änderungen am Schema durchgeführt, welche wiederum zu entsprechenden Anpassungen an den gespeicherten Datensätzen führen. Im Rahmen dieser Arbeit soll dieser Schema-Evolutionsprozess für eine bestimmte NoSQL-Datenbank realisiert werden.

1.2 Zielsetzung

Ziel dieser Arbeit ist die Entwicklung einer Schema-Evolutionskomponente für eine MongoDB-Datenbank. Auf dieser sind Datensätze in Form von JSON-Dokumenten gespeichert, die jeweils einem, sich ebenfalls auf der Datenbank befindendem JSON-Schema zugeordnet sind, welches ihre Struktur

definiert. Die Evolutionskomponente soll die Operationen einer einfachen, gegebenen Schema-Evolutionssprache akzeptieren und daraus Updateoperationen für die Datenbank generieren. Diese sollen die durch die Evolutionsoperation implizierte Änderung am Schema-Dokument durchführen und die dazugehörigen Datensätze entsprechend der neuen Schema-Version anpassen. Unter der Voraussetzung, dass vor einem Evolutionsschritt die betroffenen Dokumente valide zum zugehörigen Schema sind, sollen die geänderten Dokumente ebenfalls gültig für die neu entstandene Version des Schemas sein.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit ist in sieben Kapitel unterteilt. Nach der Einleitung folgt zunächst die Beschreibung der Grundlagen, auf die diese Arbeit aufbaut. Dazu wird in Kapitel 2 als erstes das Problem der Schema-Evolution näher beleuchtet. Kapitel 3 befasst sich im Anschluss mit den Grundlagen von NoSQL-Datenbanken. In Kapitel 4 werden konkrete Technologien vorgestellt, die im Rahmen dieser Arbeit verwendet wurden. Zu diesen zählen das JSON-Format(4.2), in dem die Dokumente gespeichert werden, das verwendete Datenbanksystem MongoDB (4.3) und die Evolutionssprache(4.4), auf die die entwickelte Komponente aufsetzt. Kapitel 5 erläutert das entworfene Konzept zur Übersetzung der Evolutionsoperationen in ausführbare Datenbankupdates, bevor anschließend die praktische Umsetzung der Evolutionskomponente in Kapitel 6 vorgestellt wird. Abschließend wird in Kapitel 7 ein Fazit und ein Ausblick auf zukünftige Entwicklungen gegeben.

Kapitel 2

Schema-Evolution

2.1 Einführung

Genau wie in der Anwendungsentwicklung ist auch in der Welt der Datenbanken der initiale Entwurf nicht unumstößlich und grundlegende Änderungen sind keine Seltenheit. So muss oftmals auf geänderte Anforderungen und Änderungen in der realen Welt mit einer Anpassung des Datenbankschemas, also der Struktur der zu speichernden Datensätze, reagiert werden. Dieser Prozess der Modifikation eines Schemas und die daraus resultierenden Problemstellungen werden in dem Begriff Schema-Evolution zusammengefasst.

Die Komplexität der Evolution eines Schemas ergibt sich vor allem durch die Notwendigkeit der Datenmigration, also der Anpassung der Datensätze. Vor allem die Sicherung der Datenintegrität, die Anpassung von Datenbankanfragen in der Anwendung und die Vermeidung von langen Ausfallzeiten durch den Evolutionsprozess sind wichtige Aspekte, die es dabei zu beachten gilt [CMZ08].

Die Forschung im Bereich der Schema-Evolution begann bereits in den 80ern, wobei hauptsächlich relationale und objektorientierte Datenbanksysteme im Mittelpunkt standen [Rod92]. Vor allem aber durch die Entwicklung von agilen Webanwendungen in den 2000er Jahren bekam das Problem der Schema-Evolution mehr und mehr an Bedeutung, da dort die Entwicklung und Veröffentlichung von neuen Schema-Versionen viel schneller vorangeht als in traditionellen Datenbanksystemen. Ein Beispiel für eine solche agile Entwicklung ist Wikipedia, dessen Datenbank in nur 4 Jahren über 170 Evolutionsschritte durchlebt hat [CTMZ08].

2.2 Definition

Der Begriff der *Schema-Evolution* bezeichnet im Grunde die Weiterentwicklung eines Schemas ohne den Verlust von bereits existierenden, in diesem Schema gespeicherten Daten. Dabei bilden zum einen die Semantik der

Schemaänderungen und zum anderen die Migration der zum Schema gehörigen Daten (vgl. Abschnitt 2.3.1) die beiden grundlegenden Probleme der Schema-Evolution [FGM00].

In [Rod95] wird in der Definition zwischen den Begriffen der Modifikation und Evolution von Schemata unterschieden. Dabei bezeichnet die Schema-Modifikation die Änderung der Definition des Schemas einer Datenbank, in der bereits Daten gespeichert sind. Von Schema-Evolution wird demnach dann gesprochen, wenn das Datenbanksystem Schema-Modifikationen ermöglicht, ohne dass es zum Verlust von vorhandenen Daten kommt. Grundsätzlich wird dabei zwischen zwei verschiedenen Arten von Evolution unterschieden - der *Domain-/Typ-Evolution* und der *Relation-/Klassen-Evolution.*

Die Domain-/Type-Evolution bezeichnet beispielsweise die Änderung des Datentyps eines einzelnen Attributs und führt unter Umständen zu weiteren Problemen. So ist die Konvertierung der Daten alles andere als trivial und stark abhängig vom Typsystem der verwendeten Datenbank. Oft ist dies auch nur durch manuelles Eingreifen des Datenbankadministrators zu lösen.

Unter Relation-/Klassen-Evolution versteht man das Einfügen, Ändern oder Entfernen von Attributen oder Klassen. Zudem zählen auch Änderungen von Beziehungen zwischen Relationen, also strukturelle Veränderungen eines Schemas, zu dieser Art der Evolution.

Des Weiteren werden in [Rod95] weitere Eigenschaften genannt, die zwar nicht Teil der Definition von Schema-Evolution sind, jedoch oftmals als Anforderungen an praktische Umsetzungen gestellt werden. So ist in der Regel die Steuerung des Evolutionsprozesses durch den Datenbankadministrator erwünscht und ein (möglichst minimales) Eingreifen auf Grund der Komplexität des Problems oftmals auch notwendig.

Eine weitere pragmatische Überlegung ist die Forderung nach Symmetrie der Schema-Modifikationen. Dies bedeutet, dass es nach der Evolution des Schemas möglich sein soll, sowohl bereits bestehende Daten in einer neueren Schema-Version, als auch neu hinzugefügte Daten in einem älteren Schema zu betrachten. Ein interessanter Aspekt ist in diesem Zusammenhang auch die Umkehrbarkeit von Modifikationsoperatoren, die durch Erzeugung von invertierbaren Mappings [Fag06] erreicht werden kann.

Zuletzt sollte noch der Anspruch an eine klar definierte, formale Schema-Modifikationssprache erwähnt werden. Diese ermöglicht eine einfache Prüfung der vorzunehmenden Schemaänderung und stellt die Konsistenz des Schemas sicher. Des Weiteren ist es sinnvoll, diese so zu definieren, dass sie aus wenigen atomaren Operationen besteht, die zu komplexeren Operationen zusammengesetzt werden können.

2.3 Wichtige Aspekte der Schema-Evolution

2.3.1 Datenmigration

Neben der Modifikation des Schemas ist die Datenmigration, also die Anpassung der vorhandenen Daten an die neue Version des Schemas, das zweite grundlegende Problem der Schema-Evolution. Notwendig dafür ist ein Mechanismus, der die Operationen der Schema-Modifikationssprache in die entsprechende Datenbanksprache, beispielsweise SQL, übersetzt. Zusätzlich dazu wird eine Migrationsstrategie benötigt, die festlegt, wann die Daten an die neue Speicherstruktur angepasst werden. In [Rod95] wird dabei zwischen *strict* und *lazy conversion* bzw. *migration* unterschieden.

Unter strict (auch: eager oder early) migration versteht man die Konvertierung aller Daten in einem einzigen Schritt und umgehend nach der Modifikation des Schemas. Der Vorteil an dieser Variante ist, dass nach der Veröffentlichung eines neuen Schemas alle Datensätze die gleiche Form besitzen und somit Anwendungen, die auf diese Daten zurückgreifen, nicht mit multiplen Schema-Versionen umgehen müssen. Nachteilig sind dabei jedoch die langen Ausfallzeiten, die durch die Konvertierung des gesamten Datenbestandes entstehen können.

Im Gegensatz zur strict migration werden bei der lazy migration die Datensätze erst in das neue Schema überführt, wenn diese das nächste mal benötigt und von der Anwendung geladen werden. Dies führt einerseits natürlich zu einer Erhöhung der Zugriffszeit, da die Daten während des Zugriffs gegebenenfalls erst noch an die neue Schema-Version angepasst werden müssen. Andererseits wird die Modifikationszeit und damit die Ausfallzeit der Datenbank im Gegensatz zur strikten Migration stark verkürzt. Ein weiterer Vorteil ist unter anderem, dass nur aktuelle Daten konvertiert werden. Ältere Datensätze werden nicht unnötigerweise in ein neues Schema überführt. Des Weiteren ermöglicht die lazy migration eine einfachere Umkehrung des Evolutionsprozesses, da unter Umständen in der Zwischenzeit nur ein geringer Teil der Daten in das neue Schema konvertiert wurde.

Zusätzlich zur strict und lazy migration wird in [KSS14] noch der *Versionen-Pluralismus* als Ansatz zum Umgang mit neu entstandenen Schema-Versionen auf Grund von Schema-Evolution. Dabei existieren mehrere Schema-Versionen parallel in einer Datenbank und die Datensätze werden jeweils im, zum Zeitpunkt ihrer Speicherung, aktuellsten Schema gespeichert. Die Verantwortung für den Umgang mit den verschiedenen Schemata liegt dabei komplett bei der Anwendung, welche natürlich mit jeder weiteren Schema-Version an Komplexität gewinnt.

2.3.2 Versionierung

Da die Schema-Evolution grundsätzlich nur die Änderungen der Schema-Definition beinhaltet, nicht aber die Historisierung der Änderungen, was in

der Praxis oftmals angebracht ist, ist auch die Schema-Versionierung ein wichtiger Aspekt der Schema-Evolution. Darunter versteht man unter anderem das Speichern der verschiedenen Schema-Versionen, wobei nicht jede Änderung zwingend eine neue Version zur Folge hat. Zudem ist auch die Fähigkeit des Datenbanksystems, den Zugriff auf die Daten über mehrere oder alle Versionen des Schemas zu ermöglichen, Bestandteil der Schema-Versionierung. In diesem Zusammenhang wird zwischen *partieller* und *vollständiger* Versionierung unterschieden [Rod95]. Die partielle Versionierung ermöglicht das Lesen der Daten in allen verfügbaren Schema-Versionen. Das Ausführen von Updates ist dabei jedoch nur über das aktuelle Schema erlaubt. Im Gegensatz dazu wird bei der vollständigen Versionierung sowohl der lesende, als auch der schreibende Zugriff auf die Daten über jedes gespeicherte Schema unterstützt. Auf die Möglichkeiten bei der Formulierung von Anfragen auf Daten, die in verschiedenen Versionen existieren, wird in Abschnitt 2.3.3 näher eingegangen.

2.3.3 Auswirkungen auf Datenbankanfragen

Die strukturellen Änderungen, die im Rahmen der Schema-Evolution vorgenommen werden, betreffen natürlich nicht nur die Datenbank selbst und die dazugehörigen Daten, sondern auch jeden, der mit diesen Daten arbeiten möchte. Da sich die Struktur der Daten nach einem Evolutionsschritt ändert, muss auch der Zugriff auf die entsprechenden Daten, also die Datenbankabfragen, dementsprechend angepasst werden. Dieser Vorgang wird auch als *Query-Rewriting* bezeichnet.

Um durch Query-Rewriting eine Datenbankabfrage, die an ein Schema S_1 gerichtet ist, so anzupassen, dass diese auch für das geänderte Schema S_2 ein äquivalentes Ergebnis liefert, ist es notwendig, dass ein Mapping $S_1 \rightarrow S_2$ existiert. Aus diesem Mapping müssen sich alle Änderungsoperationen erkennen lassen, die notwendig sind, um aus dem Schema S_1 das Schema S_2 zu generieren. Dazu werden die Operationen der Schema-Evolutionssprache, wie auch in [CMZ08], in logische Regeln übersetzt, wie beispielsweise $R(x, y) \rightarrow R(x)$, welche das Löschen des Attributs y aus der Relation R beschreibt. Durch einen Query-Rewriting-Mechanismus können mit Hilfe dieser Regeln entsprechende Ersetzungen durchgeführt werden, beispielsweise durch den *chase-and-backchase*-Algorithmus[DT03], um die Gültigkeit der Abfrage für das geänderte Schema S_2 zu gewährleisten. Das Ergebnis der Abfrage wird durch den Rewriting-Prozess nicht beeinflusst.

Ein weiterer Aspekt bezüglich der Auswirkungen auf Datenbankanfragen ist das Vorhandensein der Daten in verschiedenen Schema-Versionen. Wird in einem Datenbanksystem die Koexistenz mehrerer Schemata erlaubt, so werden auch Mechanismen benötigt, die es ermöglichen, auf bestimmte Versionen der Datensätze zuzugreifen. In [Rod95] werden dazu drei unabhängige zeitliche Dimensionen unterschieden:

- Gültigkeitszeit
- Transaktionszeit
- Schemazeit

Die Gültigkeitszeit wird dafür verwendet, um den Wert von zeitabhängigen Daten genau zu bestimmen. Ein Beispiel dafür sind saisonabhängige Preise von Hotelzimmern, für die, je nachdem für wann ein Zimmer gebucht wird, jeweils unterschiedliche Werte gültig sind. Die Transaktionszeit bezeichnet den Zeitpunkt, an dem ein Datensatz in der Datenbank gespeichert wurde. Dadurch können Daten also anhand ihrer Aktualität selektiert werden. Die Schemazeit bezieht sich wiederum auf die Version des zu verwendenden Schemas. Die Daten werden in dem Schema dargestellt, welches zum angegebenen Zeitpunkt das aktuellste war.

Die Verwendung dieser zeitlichen Dimensionen in den Datenbankanfragen bildet eine Möglichkeit für den Umgang mit verschiedenen Schema-Versionen. Eine Alternative dazu ist die Verwendung eines *vollständigen Schemas*[Rod95]. Ein vollständiges Schema ist die Vereinigung aller bisherigen Schema-Versionen. Es enthält alle Attribute, die für die verschiedenen Versionen jemals definiert wurden, und wird als übergeordnetes Schema genutzt, um versionsübergreifend auf die gespeicherten Daten zugreifen zu können.

2.3.4 Umgang mit Nullwerten

Ein grundsätzliches Problem im Datenbankenbereich bilden Nullwerte. Dabei hängt der Umgang mit Nullwerten vor allem davon ab, wie diese interpretiert werden, also aus welchem Grund die entsprechenden Werte nicht vorhanden sind. Klassischerweise existieren zwei Sichtweisen auf Nullwerte: entweder der Wert existiert, ist aber nicht bekannt (*unknown*), oder der Wert existiert nicht (*does not exist*). In [Zan84] wird zusätzlich die Interpretation *no information* definiert, welche notwendig ist, wenn über die Existenz eines Wertes keine definitive Aussage getroffen werden kann und somit die ersten beiden Sichten auf den Nullwert nicht zutreffend sind. Ein Beispiel dafür bietet das Hinzufügen eines neuen Attributs zu einer Schemadefinition. Bevor die vorhandenen Datensätze nicht (manuell) aktualisiert wurden, liegen keine Informationen darüber vor, ob für dieses Attribut ein Wert existiert und später hinzugefügt werden kann, oder nicht. Die Interpretation dieses initialen Nullwerts nach Hinzufügen des Attributs ist also *no information*.

Um Anfragen auf Datenbanken, die Nullwerte enthalten, zu ermöglichen, wird in der Praxis eine dreiwertige Logik verwendet. Dabei wird neben TRUE und FALSE ein dritter Wert benötigt, der die vorhanden Nullwerte repräsentiert. Die entsprechenden Wahrheitstabellen für die Operatoren AND, OR und NOT sind in Abbildung 2.1 dargestellt.

AND	T	F	ni
T	T	F	ni
F	F	F	F
ni	ni	F	ni

OR	T	F	ni
T	T	T	T
F	T	F	ni
ni	T	ni	ni

NOT	
T	
F	
ni	

Abbildung 2.1: Die Dreiwerte-Logik nach Zaniolo

Da der Umgang mit Nullwerte speziell auch im Bereich der Schema-Evolution ein wichtiger Aspekt ist, kann diese Dreiwerte-Logik um eine weitere Dimension erweitert werden. So richtet sich die Interpretation der Nullwerten neben dem Vorhandensein und der Anwendbarkeit des Wertes auch nach der Tatsache, ob das entsprechende Attribut bereits im Schema definiert ist [Rod95].

Kapitel 3

NoSQL

3.1 Einführung

Mit dem Web 2.0 und dem Ziel, sehr große Mengen agiler Daten zu verarbeiten, veränderten sich Anfang der 2000er die Anforderungen an die verwendeten Datenbanken. Das seit Jahrzehnten als Standard etablierte relationale Datenbankmodell war, unter anderem aufgrund des festen Schemas und der schlechten Skalierbarkeit, weniger für diese Art der Datenverarbeitung geeignet. Aus diesem Grund richtete sich das Augenmerk vermehrt auf alternative Modelle, den *NoSQL-Systemen*.

Auch wenn der Gebrauch neuer Webtechnologien, vor allem im Bereich *Social Media*, den vermehrten Einsatz von NoSQL-Systemen ausgelöst und deren Entwicklung beschleunigt hat, ist die zugrunde liegende Idee nicht neu. Trotz des Erfolges der relationalen Datenbanken, begann schon früh die Suche nach alternativen Modellen. Die Zusammenfassung einiger dieser Konzepte zu NoSQL geschah allerdings erst im Zuge der Entstehung des Web 2.0. Ein wichtiger Meilenstein in diesem Zusammenhang war die Vorstellung des Programmiermodells *MapReduce* durch Google im Jahr 2004[DG04]. Daraufhin folgten auch andere Internetunternehmen, wie zum Beispiel Yahoo, Amazon und Facebook, mit eigenen Entwicklungen. Trotz dessen sind viele NoSQL-Systeme als Open Source-Projekte entstanden. Dies hat dazu geführt, dass NoSQL-Datenbanken zwar nicht zwingend Open Source sein müssen, der Open Source-Gedanke jedoch fest in der NoSQL-Bewegung verankert ist.

Laut [EFHB11] tauchte der Begriff „NoSQL“ das erste Mal 1998 auf und bezeichnete eine relationale Datenbank von Carlo Strozzi, welche jedoch nicht mit SQL als Schnittstelle arbeitete. Die heutige Verwendung des Begriffs existiert demnach seit 2009, soll jedoch eher als „Not only SQL“ verstanden werden. Und obwohl der Begriff irreführend ist, da sich NoSQL nicht auf die Abfragesprache SQL bezieht, sondern auf die Suche nach Alternativen zum klassischen, relationalen Ansatz, hat er sich schnell verbreitet

und als Leitbegriff der Bewegung durchgesetzt.

Im Folgenden wird zunächst eine Definition von NoSQL gegeben 3.2. In Abschnitt 3.3 werden einige grundlegende Konzepte von NoSQL näher erläutert. Abschließend wird in Abschnitt 3.4 eine Kategorisierung der verschiedenen NoSQL-Systeme vorgenommen.

3.2 Definition

Eine klassische Definition für NoSQL zu finden, ist, vor allem durch die verschiedenen Variationen von NoSQL-Systemen, eher schwierig. Vielmehr wird im NoSQL-Archiv [Edl14] eine Reihe von charakteristischen Eigenschaften aufgelistet, von denen zwar nicht zwingend alle, jedoch einige auf ein NoSQL-System zutreffen. Dabei werden zunächst vier grundlegende Eigenschaften genannt:

- nicht-relational
- verteilt
- horizontal skalierbar
- Open Source

Die erste Eigenschaft, *nicht-relational*, bezieht sich auf eine der Grundideen der NoSQL-Bewegung, nämlich der Entwicklung von Alternativen zum relationalen Datenbankmodell. Zwar wird dieses bereits seit mehreren Jahrzehnten als etablierter und zuverlässiger Standard verwendet, jedoch stößt es durch die erhöhten Anforderungen des Web 2.0 Zeitalters an seine Grenzen. Vor allem bei der Verarbeitung von großen Datenmengen im Terabyte- oder gar Petabyte-Bereich treten Probleme auf, die durch relationale Systeme nicht gelöst werden können.

Der zweite und dritte Punkt ergeben sich wiederum aus den gestiegenen Anforderungen des Web 2.0. Um es zu ermöglichen, dass eine große Anzahl von Anwendern auf einer riesigen Datenmenge arbeiten kann, ist es notwendig, dass die Daten auf mehrere Server *verteilt* werden. Die Möglichkeit, auf eine Änderung der Auslastung mit dem Hinzufügen oder Entfernen von Servern reagieren zu können, macht ein verteiltes System *horizontal skalierbar*. Um diesen Anforderungen zu entsprechen, muss bei Design von NoSQL-Systemen also schon auf eine grundsätzliche Ausrichtung auf Skalierbarkeit geachtet werden.

Die *Open Source*-Eigenschaft von NoSQL-Systemen ist laut [EFHB11] weniger als strenges Kriterium, sondern eher symbolisch zu verstehen. So geht es vielmehr darum, auf Open Source-Projekte als sinnvolle Alternative zu kostspielig entwickelten, kommerziellen Produkten hinzuweisen.

Zusätzlich zu diesen vier Eigenschaften werden in [Edl14] noch weitere Charakteristika von NoSQL-Systemen aufgelistet:

- Schemafreiheit
- einfache Unterstützung der Datenreplikation
- einfache Programmierschnittstelle
- alternative Konsitenzmodelle (nicht ACID)
- sehr große Datenmengen

Auch für diese Eigenschaften gilt wiederum, dass sie nicht zwingend auf jedes NoSQL-System zutreffen müssen. Somit kann man wohl sagen, dass es weniger eine klare Definition als eine Art Liste von Richtlinien für NoSQL-Systeme gibt, was wiederum viele verschiedenste Entwicklungen in diesem Bereich ermöglicht.

3.3 Grundlegende Konzepte

In diesem Abschnitt wird auf einige theoretische Grundlagen von NoSQL näher eingegangen. So wird unter anderem der Begriff der Skalierbarkeit erläutert, da dieser eine wichtige Anforderung an NoSQL-Systeme darstellt. Im Anschluss wird der MapReduce-Algorithmus aufgrund seiner essentiellen Bedeutung für die Entwicklung der NoSQL-Bewegung vorgestellt. Abschließend wird das Konsistenzmodell von NoSQL-Systemen näher beleuchtet, da sich gerade darin große Unterschiede zu den traditionellen relationalen Datenbanksystemen befinden.

3.3.1 Skalierbarkeit

Unter dem Begriff *Skalierbarkeit* [DBL12] versteht man die Möglichkeit, die Leistungsfähigkeit eines Datenbanksystems zu erhöhen, um beispielsweise auf steigende Nutzerzahlen oder Datenmengen zu reagieren. Dabei unterscheidet man zwischen *vertikaler Skalierung* (Scale up) und *horizontaler Skalierung* (Scale out).

Vertikale Skalierung ist der traditionelle Weg, ein (verteiltes) Datenbanksystem an gestiegene Anforderungen anzupassen. Dabei werden die vorhandenen Server durch neue Komponenten ergänzt oder komplett durch leistungsfähigere Hardware ersetzt. Von Nachteil ist bei dieser Vorgehensweise jedoch zum einen die Ausfallzeiten der Server, beispielsweise beim Aufrüsten, und zum anderen die Abhängigkeit von wenigen, leistungsfähigen Servern, was im Bezug auf die Ausfallsicherheit des Systems problematisch sein kann.

Durch die Anforderungen, die im Zuge des Web 2.0 an Datenbanksysteme gestellt wurden, ging die Tendenz mehr und mehr in Richtung horizontaler Skalierung. Dies bezeichnet das Anpassen der Leistungsfähigkeit eines verteilten Systems durch Hinzufügen oder Entfernen von einzelnen Servern.

Dabei ist zu erwähnen, dass solche Systeme nicht selten aus mehreren tausend Servern bestehen, die jeweils nicht außergewöhnlich leistungsstark sein müssen, sondern eher aus Standardhardware bestehen. Der Verbund aller Server in einem Netzwerk ermöglicht dann den Umgang mit hohen Nutzerzahlen und großen Datenmengen. Dies führt auch dazu, dass der Ausfall einzelner Server leichter kompensiert werden kann und sichert somit die Verfügbarkeit und Ausfalltoleranz des Systems.

3.3.2 Map/Reduce

Das MapReduce-Verfahren ist ein von Google vorgestelltes Programmiermodell [DG04], welches die Berechnung von großen Datenmengen ermöglicht. Die Grundlage für die Entwicklung des Verfahrens war die Erkenntnis, dass die Verarbeitung von Daten im Terabyte- oder sogar Petabyte-Bereich nur durch nebenläufige Berechnungen umgesetzt werden kann. Als Ausgangspunkt dienten den Entwicklern dafür die funktionalen Programmierung stammenden Funktion *map* und *reduce*.

Die map-Funktion, welche in der ersten Phase des Verfahrens ausgeführt wird, erhält als Argument eine vom Benutzer definierte Funktion, die dann auf alle Elemente eines Datensatzes angewandt wird und daraus eine Reihe von Key/Value-Paaren erstellt. Zum Erzeugen des Zwischenergebnisses werden dann alle Werte mit dem gleichen Schlüssel gruppiert. Die reduce-Funktion, die ebenfalls vom Benutzer definiert wird, übernimmt das Zwischenergebnis als Eingabe und fasst die Werte, wenn möglich, zusammen. Als Endergebnis kann sowohl ein einzelner Wert als auch eine Liste von reduzierten Werten entstehen. Abbildung 3.1 zeigt den Datenfluss zwischen den Phasen des MapReduce-Prozesses.

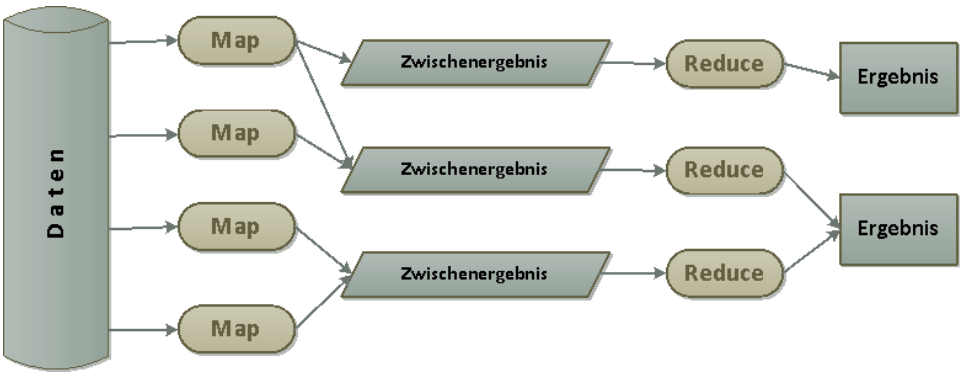

Abbildung 3.1: Datenfluss des MapReduce-Prozesses

Ein anschauliches Beispiel für das MapReduce-Verfahren bietet ein Algorithmus, der die Worthäufigkeiten in einem Text ermittelt. Die map-Funktion erstellt in diesem Fall zunächst Key/Value-Paare der Form *(w, „1“)*, wobei w für ein Wort aus dem Text steht, welches als Schlüssel dient,

und 1 die Häufigkeit darstellt. Bei einem Text mit n Wörtern entstehen also n solcher Paare. Anschließend werden alle Werte mit gleichen Schlüsseln gruppiert, sodass pro Wort eine Zwischenergebnisliste entsteht. In der reduce-Phase werden dann die Häufigkeiten pro Wort addiert und ausgegeben.

Umgesetzt wurde das MapReduce-Verfahren in verschiedensten Frameworks, beispielsweise von Google oder das darauf basierende Open Source-Framework *Hadoop*. Vom Benutzer muss dabei die Anwendungslogik in Form der map- und reduce-Funktionen definiert werden. Das MapReduce-Framework übernimmt alle weiteren Funktionen, wie beispielsweise die Parallelisierung oder die Lastverteilung während der map- und reduce-Phase [EFHB11].

3.3.3 Konsistenzmodell

Konsistenz ist ein wichtiger Aspekt im Datenbankbereich und hat vor allem in relationalen Systemen oberste Priorität. Grund dafür ist der häufige Einsatz von relationalen Datenbanken im geschäftlichen Bereich, für den die Korrektheit von geschäftskritischen Daten unbedingt notwendig ist [EFHB11]. Um die Konsistenz einer relationalen Datenbank zu garantieren, wird sichergestellt, dass alle Transaktionen die *ACID*-Eigenschaften (*Atomicity*, *Consistency*, *Isolation*, *Durability*) erfüllen. Durch Einhalten des ACID-Prinzips, zum Beispiel durch Sperren von gerade verwendeten Daten, wird es ermöglicht, dass auch in verteilten Datenbanken, in denen Daten oft auf mehreren Servern repliziert werden, alle Daten nach Beendigung einer Transaktion konsistent sind.

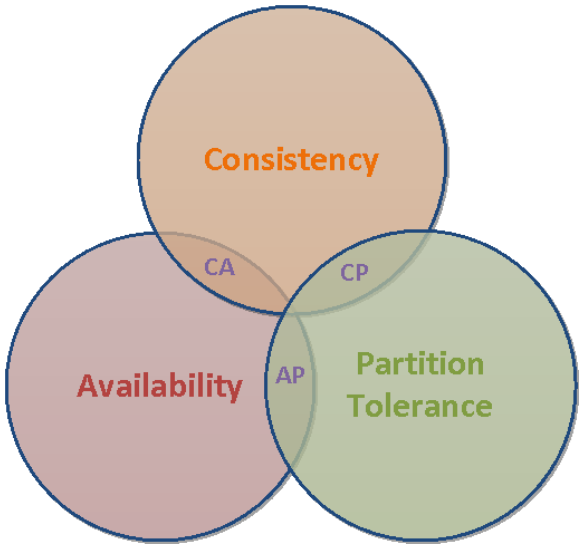

Abbildung 3.2: Veranschaulichung des CAP-Theorems

Durch das Web 2.0 entstanden andere Anforderungen an verteilte Datenbanksysteme. So gewannen vor allem *Verfügbarkeit* und *Ausfalltoleranz* immer mehr an Bedeutung. Diese bilden zusammen mit der *Konsistenz* die drei grundlegenden Eigenschaften von verteilten Datenbanken, welche sich jedoch nicht ohne Weiteres vereinen lassen. In dem von Eric Brewer vorgestelltem *CAP-Theorem* [Bre00] wird dargelegt, dass nur maximal zwei dieser Eigenschaften gleichzeitig garantiert werden können(vgl. Abb. 3.2). So kann beispielsweise die Ausfalltoleranz durch redundante Speicherung und die Konsistenz durch Nutzen von Sperrprotokollen gewährleistet werden. Durch den erhöhten Aufwand beim Ändern von Daten leidet jedoch die Verfügbarkeit des Systems, da auf die bearbeiteten Daten zeitweise nicht zugegriffen werden kann. Dies führte zur Notwendigkeit eines alternativen Konsistenzmodells: *BASE*.

BASE(*Basically Available*, *Soft state*, *Eventually consistent*) ist eine Alternative zu ACID und wurde bereits 1997 in [FGC+97] beschrieben. Dabei wird die Verfügbarkeit in den Vordergrund gestellt, was dazu führt, dass entsprechende Systeme einfacher und schneller sind, jedoch über eine schwache Konsistenz verfügen, da sie in diesem Punkt einen optimistischen Ansatz verfolgen [Bre00].

3.4 Kategorisierung

Aufgrund der Tatsache, dass, wie in Abschnitt 3.2 beschrieben, keine strikte Definition von NoSQL existiert, ging die Entwicklung von NoSQL-Datenbanken in viele verschiedene Richtungen. Die in diesem Abschnitt vorgenommene Kategorisierung basiert dabei auf [EFHB11]. Dort zählen die Autoren *Key/Value-Datenbanken*, *spaltenorientierte Datenbanken*, *dokumentorientierte Datenbanken* und *Graphendatenbanken* zum Kern der NoSQL-Bewegung. Im NoSQL-Archiv [Edl14] findet sich eine Übersicht über weitere Kategorien und jeweilige Implementierungen.

3.4.1 Key/Value-Datenbanken

Eine der ältesten Datenbanksysteme sind Key/Value-Datenbanken, die schon seit den 70er Jahren eingesetzt werden. Sie basieren auf einem einfachen Datenmodell, in dem Daten als Schlüssel-Werte-Paare gespeichert werden. Die Schlüssel werden dabei in der Regel in Namensräume und Datenbanken aufgeteilt. Die Werte besitzen keine bestimmte Form, sondern können, neben einfachen Strukturen (z.B. Zeichenketten), auch komplexere Datenstrukturen, wie Listen oder Sets, besitzen. Die Flexibilität in Bezug auf das Format der gespeicherten Daten beruht auf der Tatsache, dass für Key-Value-Datenbanken der Inhalt der Daten, also deren Struktur, verborgen bleibt. Dadurch werden jedoch die Zugriffsmöglichkeiten auf die Daten beschränkt,

sodass der Zugriff demnach nur über den Schlüssel erfolgt. Ebenso können Änderungen an den Daten nur durch Löschen und Einfügen kompletter Schlüssel-Werte-Paare umgesetzt, da ein partielles Update des Wertes nicht unterstützt wird [Stö14].

Im Zuge des Web 2.0 war es vermehrt notwendig, viele unabhängige Daten zu verarbeiten. Vor allem durch ihre gute Skalierbarkeit eignen sich Key-Value-Datenbanken dafür besonders gut. Ihr einfaches Datenmodell ermöglicht zudem eine einfache und schnelle Datenverwaltung.

In die Kategorie der Key/Value-Datenbanken fallen unter anderem Systeme wie *Redis*, *Riak* und *Amazon Dynamo*.

3.4.2 Spaltenorientierte Datenbanken

Spaltenorientierte Datenbanken sind dadurch charakterisiert, dass sie, im Gegensatz zu relationalen Datenbanken, die Datensätze nicht reihenweise, sondern spaltenweise organisieren. So werden nicht die Attributwerte eines kompletten Datensatzes, also eine Reihe einer Tabelle, zusammen gespeichert, sondern jeweils alle Werte eines Attributs, also eine Spalte einer Tabelle. Dadurch ergibt sich eine effiziente spaltenweise Auswertung von Daten, welche beispielsweise in Data Warehouses häufig benötigt wird. Das Auswerten von kompletten Datensätzen, also eine reihenweise Verarbeitung der Daten, ist hierbei natürlich aufwändiger, da diese auf verschiedene Bereiche des physikalischen Speichers verteilt sind.

Eine besondere Variante von spaltenorientierten Datenbanken bilden die *column family stores* [Stö14]. Hierbei werden mehrere Spalten zu Gruppen (column families) zusammengefasst, welche bereits beim Anlegen der Tabelle definiert werden. Zudem werden die Daten einer column family zusammen gespeichert, was eine physikalische Optimierung ermöglicht.

Ein weiteres wichtiges Merkmal von column family stores ist die Flexibilität bezüglich des Schemas. Im Gegensatz zu den column families können Spalten auch nach dem Anlegen der Tabelle hinzugefügt werden, indem ein Datensatz gespeichert wird, der über eine zusätzliche Spalte verfügt. Das neu entstandene Schema wird dann nur diesem Datensatz zugeordnet, wodurch in der gleichen Datenbank Daten existieren, denen verschiedene Schemata zu Grunde liegen.

Beispiele für spaltenorientierte Datenbanken sind unter anderem *Hadoop/HBase*, *Cassandra* und *Amazon SimpleDB*.

3.4.3 Dokumentorientierte Datenbanken

Obwohl der Name es vielleicht vermuten lassen könnte, sind dokumentorientierte Datenbanken nicht für das Speichern und Verwalten beliebiger Dokumente ausgelegt. Vielmehr speichern sie strukturierte und semistrukturierte Daten in Dokumenten, welche ein festgelegtes Format, wie zum

Beispiel JSON, besitzen. Des Weiteren wird jedem Dokument eine ID zugeordnet, welche es ermöglicht, dieses eindeutig zu identifizieren. Aufgrund dieses Schlüssel-Wert-Prinzips, wobei das gespeicherte Dokument selbst der Wert ist, ergibt sich sogar eine Ähnlichkeit zu den Key/Value-Datenbanken [Stö14]. Ein gravierender Unterschied liegt jedoch darin, dass einer dokumentorientierten Datenbank der Inhalt der Werte nicht verborgen bleibt. So können die Werte nicht nur als Ganzes, sondern auch partiell gelesen und bearbeitet werden. Auch die Selektion von Dokumenten kann dabei nicht nur anhand der ID erfolgen, sondern auch anhand des Inhalts von Dokumenten, wie zum Beispiel der Wert einer Eigenschaft eines JSON-Objekts.

Des Weiteren besitzen auch dokumentorientierte Datenbanken kein festgelegtes Schema, in dem die Daten gespeichert werden. Lediglich das Format der Dokumente ist vorgeschrieben, nicht jedoch deren Struktur. Die Einhaltung eines Schemas beim Einfügen oder Verändern von Daten und gegebenenfalls die Schemaevolution muss also in der Anwendungsschicht realisiert werden.

Zu den dokumentorientierten Datenbanken zählen vor allem *MongoDB* und *CouchDB*, welche für den Autoren von [EFHB11] die wichtigsten Vertreter dieser Kategorie sind.

3.4.4 Graphendatenbanken

Graphendatenbanken sind Datenbanken, die konzipiert wurden, um Graph- oder Baumstrukturen zu verwalten und dadurch das effiziente Lösen von Problemen der Graphentheorie zu ermöglichen. Obwohl bereits in den 80ern und 90ern in diesem Gebiet geforscht wurde, gewannen Graphendatenbanken vor allem auch durch das Aufkommen von *Location Based Services* stark an Bedeutung. Gerade für das Speichern von Geodaten in diesen Bereichen, jedoch auch für die Modellierung anderer vernetzter Informationen, eignen sich Graphen besonders gut, da nicht nur die Informationen selbst von Interesse sind, sondern auch die Art und Weise ihrer Vernetzung. Für die Bearbeitung von solch komplexen Fragestellungen eignen sich speziell modellierte Graphendatenbanken besser als konventionelle relationale Modelle.

Neben dem allgemeinen Graphenmodell, bestehend aus Knoten und gerichteten Kanten, bildet vor allem das *Property-Graph-Modell* eine wichtige Grundlage für Graphendatenbanken. Dabei besitzen Knoten und Kanten eines Graphs jeweils Eigenschaften, in denen weitere Informationen gespeichert werden können. Laut [EFHB11] wird dieses Modell in den meisten aktuellen Implementierungen verwendet.

Beispiele für Graphendatenbanken sind unter anderem *Neo4J*, *InfiniteGraph* und *HyperGraohDB*.

Kapitel 4

Verwendete Technologien

4.1 Einführung

Neben den allgemeinen Grundlagen von Schema-Evolution und NoSQL-Systemen baut diese Arbeit vor allem auch auf verschiedenen konkreten Technologien auf. Diese sollen in diesem Kapitel vorgestellt werden. Dazu wird in Abschnitt 4.2 zunächst das Datenformat JSON und die Möglichkeit der Schemadefinition mittels JSON-Schema näher betrachtet. Abschnitt 4.3 befasst sich mit der, in dieser Arbeit verwendeten, dokumentorientierten NoSQL-Datenbank MongoDB und erläutert grundlegende Merkmale des Datenbanksystems. Abschließend wird in Abschnitt 4.4 die Schema-Evolutionssprache vorgestellt, die als Ausgangspunkt für die im Rahmen dieser Arbeit entwickelten Schema-Evolutionskomponente dient.

4.2 JSON und JSON-Schema

Die *JavaScript Object Notation* [JSO14], kurz JSON, ist ein einfaches Format zum Datenaustausch, welches sowohl als RFC [CB14], als auch von der ECMA (European Computer Manufacturers Association) [ECM13] spezifiziert wird. JSON basiert auf einer Untermenge von JavaScript, ist jedoch grundsätzlich unabhängig von Programmiersprachen. Es gilt im Allgemeinen als leicht lesbar für Menschen und auf Grund einer Vielzahl von Parsern auch gut interpretierbar durch Maschinen. Eingesetzt wird JSON in der Regel im Bereich der Webanwendungen, oftmals auch als einfachere Alternative zu XML. Des Weiteren spielt JSON auch im NoSQL-Bereich eine große Rolle, da es in dokumentorientierten Datenbanken, wie beispielsweise MongoDB (vgl. Abschnitt 4.3), als Datenformat für die zu speichernden Dokumente genutzt wird.

Mit JSON-Schema [Zyp14] existiert ein Entwurf einer Schemasprache, welche während der Entstehung dieser Arbeit lediglich als Internet-Draft in Version 04 [ZG13] spezifiziert ist. JSON-Schema ermöglicht die Definition

eines Schemas, das die Struktur von JSON-Dokumenten beschreibt, und die Validierung von JSON-Dokumenten gegen ein definiertes Schema. Entsprechende Validatoren existieren für verschiedenste Programmiersprachen, wie beispielsweise Java, Python, oder auch als .NET-Framework.

Im Folgenden wird zunächst die grundlegende Datenstruktur von JSON-Dokumenten 4.2.1 erläutert. Im Abschnitt 4.2.2 wird der Aufbau eines JSON-Schemas vorgestellt.

4.2.1 Datenstruktur

Ein JSON-Dokument besteht grundsätzlich aus einem JSON-Objekt, welches von geschweiften Klammern umschlossen wird. Das JSON-Objekt ist entweder leer, oder besteht aus einem oder mehreren, durch Komma voneinander getrennten, Name/Wert-Paaren. Zwischen dem Namen und dem Wert befindet sich ein Doppelpunkt als Trennzeichen. Abbildung 4.1 zeigt den grundsätzlichen Aufbau eines JSON-Dokuments. Dabei ist zu erkennen, dass der Wert eines Name/Wert-Paares verschiedene Formen annehmen kann. Zum einen kann dieser aus einem weiteren Objekt bestehen. Zum anderen kann ein Wert auch aus einem Array, also einer geordneten Liste von Werten, bestehen. Auch ein leeres Array stellt einen gültigen Wert dar.

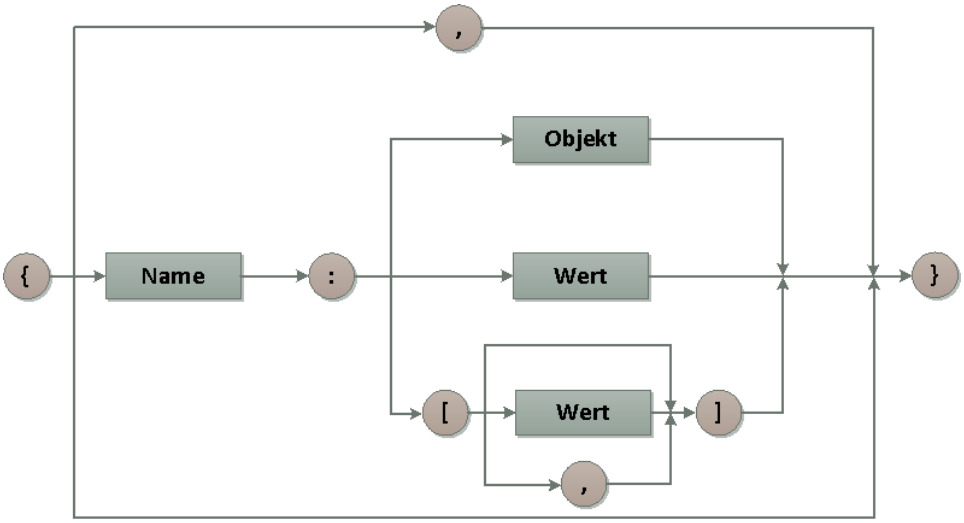

Abbildung 4.1: Struktur eines JSON-Objektes

Neben dem Objekt und dem Array kann ein Wert auch die folgenden Formen haben:

- Zeichenkette
- Zahl
- *true*
- *false*
- *null*

Eine Zahl kann in JSON sowohl ein negatives Vorzeichen enthalten, als auch durch einen Dezimalpunkt unterbrochen sein. Zudem kann eine Zahl um einen Exponenten ergänzt werden. Dies erfolgt unter Angabe von e oder E, gefolgt von + oder - und einer beliebigen Ziffernfolge $0-9$. Führende Nullen bei Zahlen sind nicht erlaubt. Zeichenketten, so wie auch die Namen in den Name/Wert-Paaren, werden in Anführungszeichen geschrieben. Zahlen, so wie die Werte *true*, *false* und *null* stehen nicht in Anführungszeichen.

Im Folgenden findet sich ein Beispiel für ein gültiges JSON-Dokument.

```
{
  "Titel": "The Hitchhiker's Guide to the Galaxy",
  "ISBN": "978-0345391803",
  "Preis": 10.99,
  "Autor": {
    "Name": "Adams",
    "Vorname": "Douglas",
    "Geburtsjahr": 1952,
    "Ehepartner": null,
    "Kinder": []
        },
  "Stichworte": ["42", "Deep Thought"],
  "AufLager": true
}
```

4.2.2 JSON-Schema

Ein JSON-Schema wird in JSON-Syntax definiert. Es besteht aus einem Objekt und bildet somit ein gültiges JSON-Dokument. Für die Definition der Attribute steht eine Reihe von Schlüsselworten zur Verfügung, welche die Namen der Attribute darstellen. Die Definition von anderen Attributen, die keine Schlüsselworte sind, ist jedoch auch möglich.

Grundlegender Aufbau

Zunächst mal stehen verschiedene Schlüsselworte zur Verfügung, mit denen die Metainformationen eines JSON-Schemas definiert werden können. Dazu

zählen unter anderem *title* und *description*, die eine kurze bzw. ausführliche Beschreibung des Schemas enthalten. Zudem existieren die Schlüsselworte *id* und *$schema*, welche beide eine URI enthalten, die auf das definierte Schema bzw. auf das Schema des definierten Schemas verweisen.

Der Datentyp einer Instanz eines Objekts wird durch das Schlüsselwort *type* bestimmt. Dieses kann als Wert *object* oder *array*, sowie alle primitiven JSON-Datentypen enthalten. Dabei ist zu beachten, dass der Zahlentyp unterteilt wird in *number* und *integer*, wobei ein integer ein ganze Zahl darstellt, ohne Bruchteil und Exponententeil.

Um die Attribute eines Objekts bzw. die Elemente eines Arrays zu definieren, werden die Schlüsselworte *properties* bzw. *items* genutzt. Dabei enthält der Wert von *items* die Beschränkungen für die Elemente des Arrays. Der Wert von *properties* enthält eine Liste mit Name/Wert-Paaren, wobei der Name einen Attributnamen darstellt und dazugehörige Wert die Constraints für dieses Attribut, wie beispielsweise den Typ. Um festzulegen, dass bestimmte Attribute in den Instanzen existieren müssen, wird das Schlüsselwort *required* benutzt, dessen Wert ein Arrays mit einer Liste aller benötigten Attribute ist.

```
{
  "title": "Beispielschema",
  "id": "http://abc.de/beispiel-schema#",
  "$schema": "http://json-schema.org/draft-04/schema
     #",
  "description": "Dies ist ein Beispielschema",
  "type": "object",
  "properties": {
    "name": {
      "type": "string"
      },
    "alter": {
      "type": "integer"
    },
    "hobbies": {
      "type": "array",
      "items": "string"
    }
  },
  "required": [ "name", "alter" ]
}
```

Des Weiteren ist es möglich bei der Definition eines Schemas auf andere Schemata zurückzugreifen. So kann man mit Hilfe des Schlüsselwortes *$ref* durch die Angabe einer URI im Wert auf ein bereits vorhandenes Schema verwiesen werden. Dieses kann mit Hilfe des *definitions*-Schlüsselwortes

selbst als Subschema definiert und dann mit einer relativen URI referenziert werden.

```
{
  "type": "object",
  "properties": {
    "name": {
      "$ref": "#/definitions/nameDef"
      }
        },
        "definitions": {
    "nameDef": { "type": "string" }
        }
}
```

Constraints

Neben dem grundlegenden Aufbau eines JSON-Schemas existieren noch viele weitere Möglichkeiten, beispielsweise die Instanzen der definierten Attribute zu beschränken. Einige dieser Möglichkeiten sollen an dieser Stelle kurz vorgestellt werden.

Zum einen existiert die Möglichkeit, mit Hilfe der Schlüsselworte *minProperties* bzw. *maxProperties* die Anzahl der Attribute zu beschränken. Auch zusätzliche Attribute können durch das *additionalProperties*-Schlüsselwort entweder verboten, oder durch die Angabe eines Schemas für diese Attribute ausdrücklich erlaubt werden. Durch Nutzen von *dependencies* können Abhängigkeiten zwischen Attributen definiert werden. So kann dadurch beispielsweise festgelegt werden, dass in einer Instanz ein bisher optionales Attribut benötigt wird, sofern ein anderes optionales Attribut existiert.

In der Definition von Arrays ist es möglich, die Anzahl der Elemente in einer Instanz dieses Arrays durch die Schlüsselworte *minItems* bzw. *maxItems* zu beschränken. Wie bei Objekten könne auch in Arrays zusätzliche Elemente durch *additionalItems* verboten erlaubt und deren Schema bestimmt werden.

Die Typdefinition eines Attributs erlaubt es, nicht nur einen konkreten Typ festzulegen, sondern auch eine Liste von Typen anzugeben, von denen entweder genau einer (*oneOf*), mindestens einer (*anyOf*) oder alle (*allOf*) mit dem Typ des Attributs der Instanz übereinstimmen müssen. Des Weiteren kann durch *enum* eine Liste mit festgelegten Werten angegeben werden, von denen einer dem tatsächlichen Wert in der Instanz entsprechen muss. Für den Fall, dass in der Instanz kein Wert für ein bestimmtes Attribut existiert, lässt sich durch *default* ein Standardwert für ein Attribut festgelegen.

Auch für die primitiven Datentypen, also string und number bzw. integer, existieren verschiedene Möglichkeiten, um den entsprechenden Attri-

buten Constraints hinzuzufügen. So kann unter anderem die Länge eines Strings durch Angabe Werten für *minLength* bzw. *maxLength* beschränkt werden. Ebenso kann ein bestimmtes Muster für einen String festgelegt werden. Dazu wird das Schlüsselwort *pattern* genutzt, dessen Wert einen regulären Ausdruck enthält. Um den Wert einer Zahl zu beschränken, kann unter anderem ein Intervalls durch Angabe eines Minimalwertes (*minimum*) und eines Maximalwertes (*maximum*) erfolgen. Ebenso ist es mögliche, durch den Gebrauch des Schlüsselwortes *multipleOf* nur das Vielfache einer Zahl als Attributwert zuzulassen.

Eine vollständige Liste aller Schlüsselworte ist in [Zyp14] zu finden.

4.3 MongoDB

MongoDB [Mon13] ist eines der bekanntesten Open Source NoSQL-Datenbanksysteme und einer der Hauptvertreter der dokumentorientierten Datenbanken (vgl. Abschnitt 3.4.3). Es bietet die Möglichkeit, JSON-ähnliche Dokumente in verschiedenen Kollektionen zu speichern, zu verwalten und auf diese zuzugreifen. Dabei kann über die verschiedenen Zugriffsoperationen nicht nur auf ganze Dokumente, sondern auf einzelne Felder der Dokumente und deren Attribute zugegriffen werden.

Obwohl ein Grundsatz von MongoDB die schemafreie Speicherung von Dokumenten ist, werden die Kollektionen in der Praxis zur Speicherung strukturell ähnlicher bzw. gleicher Dokumente genutzt.

Als Schnittstelle wird den Nutzern die *Mongo Shell* zur Verfügung gestellt, welche neben den definierten Zugriffsoperationen auch benutzerdefinierte JavaScript-Funktionen interpretieren kann. Als Programmierschnittstelle existieren Treiber für viele gängige Programmiersprachen, wie zum Beispiel Java, C# oder Python.

Nachfolgend wird im Abschnitt 4.3.1 zunächst BSON, das von MongoDB genutzte Format zur Dokumentspeicherung, vorgestellt. In 4.3.2 wird auf verschiedene Merkmale von MongoDB näher eingegangen, bevor in Abschnitt 4.3.3 die CRUD-Operationen (Create, Read, Update, Delete) vorgestellt werden.

4.3.1 BSON

BSON (Binary JSON) [BSO14] ist ein binäres Datenformat, welches von MongoDB für die Speicherung von Dokumenten genutzt wird. Es stellt eine Erweiterung des JSON-Formates dar, wobei für die Elemente eine wesentlich höhere Anzahl an Datentypen zur Verfügung steht. Nachfolgend sind alle für BSON definierten Datentypen aufgelistet:

- Double
- String
- Object
- Array
- Binary Data
- Undefined
- Object Id
- Boolean
- Date
- Null
- Regular Expression
- JavaScript
- Symbol
- JavaScript(with scope)
- 32-Bit Integer
- Timestamp
- 64-Bit Integer
- Min Key
- Max Key

Der Grund für die große Anzahl an Datentypen ist ein wesentliches Ziel der Entwickler von BSON: Effizienz. Durch die speziellen Datentypen und deren Längenbeschränkung (beispielsweise 32- und 64-Bit Integer) können die Dokumente effizienter durchsucht werden. Ein Parser muss nicht jeden Attributwert komplett lesen, um dessen Ende und damit den Anfang des nächsten Attributs zu finden, sondern kann nach Erkennen des Datentyps sofort die Länge des Wertes bestimmen. Dies ermöglicht ein einfacheres Parsen der gespeicherten Dokumente.

4.3.2 Merkmale

Zu den wichtigsten Anforderungen von NoSQL-Datenbanken zählen unter anderem die Skalierbarkeit des Systems und der effiziente Umgang mit großen Datenmengen. Um diesen Anforderungen gerecht zu werden, besitzt MongoDB verschiedene Merkmale, von denen einige in diesem Abschnitt näher beleuchtet werden sollen.

Indexierung

Um Anfragen auf die gespeicherten Dokumente effizient und mit möglichst wenig Zugriffen umsetzen zu können, wird von MongoDB die Indexierung dieser Dokumente unterstützt. Dabei wird standardmäßig ein Index über die Objekt-ID der Dokumente erstellt. Diese ID ist für jedes Dokument einzigartig und wird automatisch von MongoDB beim Speichern des Dokuments erstellt.

Zusätzlich zum Index über die Objekt-ID wird es dem Nutzer ermöglicht, selbst verschiedenste Indexe zu definieren. So können nicht nur Indexe über einzelne Felder, sondern auch ein zusammengesetzter Index (*Compound Index*) über mehrere Felder gebildet werden. Um Arrays zu indexieren, werden *Multikey Indexe* genutzt, welche für jedes Element eines Arrays einen eigenen Index bilden. Des Weiteren existieren auch Hash Indexe, so wie spezielle Indexformen für Geodaten (*Geospatial Index*) und zur Indexierung von Tex-

ten. Bei der Erstellung eines *Text Indexes* werden automatisch sogenannte Stopworte, beispielsweise Konjunktion oder Artikel, aus dem Text entfernt und somit nur Wörter mit möglichst hoher Aussagekraft im Index gespeichert. Aktuell werden dabei 15 verschiedene Sprachen unterstützt.

Indexe in MongoDB besitzen grundsätzlich die Form von B-Bäumen und werden jeweils für eine Kollektion erstellt.

Replikation

Um die Ausfalltoleranz und die Verfügbarkeit eines Datenbanksystems sicherzustellen, ist es notwendig, durch Replikation der vorhandenen Daten die Last auf mehrere Server zu verteilen. In MongoDB wird diese Replikation durch sogenannte *Replica Sets* umgesetzt. Ein Replica Set, dargestellt in Abbildung 4.2, besteht aus mehreren Mongo-Deamon-Prozessen, die zu einer Gruppe zusammengefasst sind. Diese Deamon-Prozesse sind zuständig für die Verwaltung der Daten und die Bearbeitung von Anfragen auf diese Daten. Pro Replica Set existiert genau ein Deamon-Prozess, welcher den *Primary* bildet und der einzige Deamon ist, der Schreiboperationen auf die Daten durchführt. Alle anderen Deamons (*Secondaries*) können nur Leseoperationen durchführen. Sie enthalten eine Kopie der Daten des Primaries und aktualisieren diese anhand seines Operationen-Logs. Die Aktualisierung der Secondaries erfolgt durch einen asynchronen Prozess.

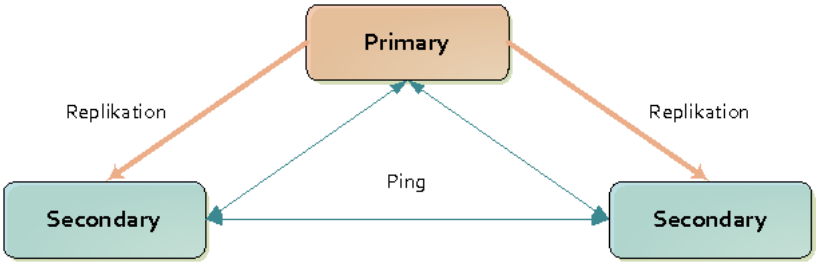

Abbildung 4.2: Beispiel eines Replica Sets

Um die Verbindung zu den anderen Mitgliedern des Replica Sets zu überprüfen, sendet jeder Deamon in regelmäßigen Abständen Pings an alle anderen Deamons. Auf diese Weise kann bei einem Ausfall des Primaries rechtzeitig reagiert werden, indem einer der Secondaries zum neuen Primary gewählt wird. Die Auswahl erfolgt dabei anhand verschiedener Kriterien, wie zum Beispiel der jüngsten Optime, also dem Zeitstempel der letzten Operation des Deamons und natürlich dessen Erreichbarkeit.

Horizontale Skalierung

Skalierbarkeit ist eine der wichtigsten Aspekte von NoSQL-Datenbanken (vgl. Abschnitt 3.3.1). Vor allem die horizontale Skalierung, also die Vertei-

lung der Last auf mehrere Server, um die Leistungsfähigkeit des Systems zu steigern, spielt eine besondere Rolle. MongoDB unterstützt die horizontale Skalierung durch *Sharding*, einem Prozess, bei dem große Datenmengen auf mehrere Knoten, als *Shards* bezeichnet, verteilt werden. Die Partitionierung der Daten geschieht dabei anhand von sogenannten *Shard Keys*, welche entweder aus einem einzigen indexierten Feld oder einem zusammengesetzten Index besteht. Die Aufteilung der Daten in einzelne chunks erfolgt entweder anhand von Intervallen (*range based sharding*), oder durch Nutzen einer Hash-Funktion (*hash based sharding*).

Mit steigender Anzahl von Shards sinkt sowohl die Datenmenge, als auch die Anzahl der Operationen, die jeder Shard verarbeiten muss, was zu Leistungssteigerung des ganzen Systems, genannt *Sharded Cluster*, führt.

Ein Sharded Cluster besteht neben den Shards aus mindestens einem *Query Router* und genau drei *Config Server*. Die Query Router sind dafür zuständig, die Anfragen, die von einer Applikation an die Datenbank gestellt werden, zu verarbeiten, indem diese an die betreffenden Shards weitergeleitet werden. Um die Shards zu finden, die die für die Anfrage benötigten Daten enthalten, wird eine Reihe von Metadaten zurückgegriffen, die wiederum auf den Config Servern in Form von Mappings hinterlegt sind.

Um die Effektivität des Shardings zu erhalten, existieren Prozesse, die während der Laufzeit des Systems die ausgewogene Aufteilung der Daten auf die Shards überwachen und gegebenenfalls anpassen. Zu diesen Prozessen zählt unter anderem das *Splitting*, das auf die Größe der einzelnen chunks achtet. Wird bei einem Insert oder Update die definierte Maximalgröße eines chunks überschritten, wird dieser halbiert und die Metainformationen werden entsprechend angepasst. Um die ausgewogene Verteilung der Daten auf die einzelnen Shards zu garantieren, überwacht ein *Balancing*-Prozess die Anzahl der chunks pro Shard und verschiebt bei ungleicher Verteilung gegebenenfalls chunks zwischen den Shards.

4.3.3 CRUD-Operationen

Um auf die gespeicherten Dokumente und deren Inhalt zugreifen zu können, bietet MongoDB eine Reihe verschiedenster Operationen. Nachfolgend sollen die grundlegenden CRUD-Operationen mitsamt ihrer Parameter und Optionen vorgestellt werden.

Create

Um neue Dokumente in der Datenbank abzulegen, bietet MongoDB die Funktion *Insert*. Als Parameter wird entweder ein einzelnes Dokument, oder eine Reihe von Dokumenten in einem Array übergeben, welche dann in die angegebene Kollektion eingefügt werden.

```
db.collection.insert({beispiel: 1})
```

Sofern keine konkrete Objekt_ID mit dem Feldnamen *_id* angegeben wird, wird vom System automatisch eine ID für jedes Dokument vergeben. Sollte die angegebene Kollektion zur Zeit des Funktionsaufrufes noch nicht existieren, wird sie zunächst erstellt und das übergebene Dokument eingefügt.

Read

Die *Find*-Funktion ermöglicht es, eine Kollektion nach Dokumenten mit bestimmten Eigenschaften zu durchsuchen und diese auszugeben. Als optionale Parameter können Selektionskriterien und eine Projektionsliste übergeben werden. Das Ergebnis dieser Operation ist ein Cursor, der auf alle passenden Dokumente zeigt. Die Funktion hat grundsätzlich die folgende Form:

```
db.collection.find(
    {<Selektionskriterien>},
    {<Projektionsliste>}
)
```

Die Selektionskriterien beinhalten alle Kriterien die an bestimmte Felder der gesuchten Dokumente gestellt werden. Um diese zu spezifizieren können verschiedene Operatoren genutzt werden, wie unter anderem Vergleichsoperatoren (z.B. $gt für „>“) oder logische Operatoren (z.B. $and). Die Projektionsliste enthält die Liste von Feldern, die im Ergebnis ausgegeben werden sollen. Dabei wird die einzigartige Objekt-ID standardmäßig ausgegeben.

Im Folgenden ist ein Beispiel für eine Find-Funktion gegeben, die in der Kollektion „Studenten“ nach allen Dokumenten sucht, bei denen der Wohnort „Rostock“ ist und das Alter größer als 20. Im Ergebnis ausgegeben werden nur der Vorname und der Nachname, so wie standardmäßig die ID des jeweiligen Dokuments.

```
db.studenten.find(
    {wohnort: "Rostock", alter: {$gt: 20}},
    {vorname: 1, nachname: 1}}
)
```

Update

Die *Update*-Funktion ermöglicht es, anhand des Inhalts bestimmte Dokumente zu filtern und in diesen Änderungen durchzuführen, wie unter anderem das Hinzufügen, Entfernen oder Umbenennen von Feldern, so wie das Ändern des Wertes eines Feldes. Die benötigten Parameter der Funktion sind zum einen die Selektionskriterien und zum anderen das gewünschte Update. Optional können auch diverse Optionsparameter übergeben werden.

```
db.collection.uodate(
    {<Selektionskriterien>},
    {<Update>}
)
```

Die Selektionskriterien werden analog wie für die Find-Funktion spezifiziert. Um die vorzunehmende Änderung zu definieren, muss zum einen der Name des Feldes angegeben werden, auf das sich das Update bezieht. Zum anderen existieren diverse Update-Operatoren, durch die die Art der Änderung angegeben wird, wie zum Beispiel *$set* für das Ändern eines konkreten Wertes, oder *$rename* für die Umbenennung des ausgewählten Feldes.

Das nachfolgende Beispiel zeigt das Update des Feldes „Preis“ bei allen Büchern, deren Autor den Vornamen „Douglas“ und den Nachnamen „Adams“ hat. Die *multi*-Option, die hier den Wert *true* bekommt, besagt, dass alle Dokumente, für die die Selektionskriterien gelten, dementsprechend geändert werden. Wird diese Option nicht angegeben, wird nur das erste gefundene Dokument geändert, da der Standardwert von multi *false* ist.

```
db.buecher.update(
    {autor: {
            vorname: "Douglas",
            nachname: "Adams"
            }
    },
    {$set:  {
            preis: 42
            }
    },
    {multi: true}
)
```

Delete

Die *Remove*-Funktion ermöglicht das Löschen von ganzen Dokumenten, indem beim Funktionsaufruf entsprechende Selektionskriterien für die Dokumente angegeben werden. Die Selektionskriterien werden wie in der oben beschriebenen Find-Funktion spezifiziert. Um alle Dokumente einer Kollektion zu löschen, wird als Selektionskriterium das leere Objekt { } übergeben.

```
db.collection.remove(<Selektionskriterium>)
```

4.4 Eine Schema-Evolutionssprache für NoSQL-Systeme

Schema-Evolution ist ein wichtiger Aspekt in der agilen Softwareentwicklung und gewinnt vor allem auch im Bereich der Web-Anwendungen, welche nicht selten auf NoSQL-Datenbanksystemen aufbauen, immer mehr an Bedeutung. In diesem Abschnitt soll eine einfache Evolutionssprache vorgestellt werden, die definiert wurde, um Schema-Evolution in NoSQL-Datenbanken zu ermöglichen[SKS13]. Dazu wird zunächst in Abschnitt 4.4.1 die Motivation für eine solche Sprache dargelegt, bevor dann das zugrundeliegende Speichermodell (4.4.2) und die Sprache selbst (4.4.3) vorgestellt werden.

4.4.1 Motivation

Die Schemafreiheit ist ein wichtiges Merkmal von NoSQL-Datenbanken, welches den Nutzern eine besondere Flexibilität beim Speichern von strukturell unterschiedlichen Daten bietet. Da die Datenbanksysteme in der Regel nicht die Definition eines globalen Schemas ermöglichen, liegt die Verantwortung für die Struktur der gespeicherten Daten dabei komplett bei der Anwendung, die die Daten verwendet. Vor allem wenn Daten in verschiedenen Schema-Versionen vorliegen, steigt der Aufwand beim Umgang mit den Daten, sodass das Schema-Management direkt in der Anwendung eine ungünstige Lösung des Problems bildet. Daraus ergibt sich die Idee einer Schema-Management-Komponente, zu deren Aufgaben, neben der Evolution eines Schemas von einer Version zur nächsten, auch die Migration der zum Schema gehörenden Daten zählt. Die Grundlage für so eine Komponente bildet eine entsprechende Evolutionssprache, die es ermöglicht, Änderungen an Schema und Datensätzen durchzuführen.

4.4.2 Grundlagen zum Daten- und Speichermodell

Die in diesem Abschnitt vorgestellte Evolutionssprache richtet sich vor allem an dokumentorientierte NoSQL-Datenbanken, welche Dokumente in einem festgelegten Datenformat, beispielsweise JSON, speichern. Durch das Speichern von strukturierten bzw. semi-strukturierten Daten ist es überhaupt möglich, ein Schema für die Daten zu definieren und einen Schema-Evolutionsprozess durchzuführen.

Die gespeicherten Instanz-Dokumente (*entities*) besitzen jeweils eine ID und werden einer Gruppe(*kind*) von strukturell ähnlichen Dokumenten zugeordnet. Zudem besitzen die Dokumente verschiedene Eigenschaften (*properties*), welche aus Schlüssel/Wert-Paaren bestehen, wobei auch unter anderem geschachtelte Eigenschaften möglich sind.

Da dokumentorientierten Datenbanken der Inhalt der gespeicherten Dokumente nicht verborgen bleibt, können die zu ändernden Dokumente somit

nicht nur über ihre Gruppe sondern auch anhand des Inhalts adressiert werden. Dazu wird davon ausgegangen, dass die entsprechenden Datenbanksysteme die Möglichkeit bieten, mindestens konjunktive Anfragen zu formulieren und einen einfache Wertevergleich durchzuführen.

4.4.3 Spezifikation einer einfachen Schema-Evolutionssprache

Da NoSQL-Datenbanken in der Regel kein globales Schema besitzen und ebenso nicht die Möglichkeit bieten, eines zu definieren, richtet sich die hier vorgestellte Evolutionssprache primär an die Instanz-Dokumente selbst, an denen die strukturellen Änderungen vorgenommen werden sollen. Zu diesem Zweck wurden die grundlegendsten Änderungsoperationen entsprechend definiert und in einer einfachen Evolutionssprache zusammengefasst, welche in *Erweiterter-Backus-Naur-Form* in Abbildung 4.3 dargestellt ist.

```
evolutionop ::= add | delete | rename | move | copy;
add ::= "add" property "=" value [ selection ];
delete ::= "delete" property [ selection ];
rename ::= "rename" property "to" pname [ selection ];
move ::= "move" property "to" kname [ complexcond ];
copy ::= "copy" property "to" kname [ complexcond ];

selection ::= "where" conds;
complexcond ::= "where" ( joincond | conds | ( joincond "and" conds ) );

joincond ::= property "=" property;
conds ::= cond { "and" cond };
cond ::= property "=" value;

property ::= kname "." pname;
kname ::= identifier;
pname ::= identifier;
```

Abbildung 4.3: Schema-Evolutionssprache in EBNF

Die Sprache beinhaltet insgesamt fünf Operationen, zu denen unter anderem das Hinzufügen, Löschen und Umbenennen einer Eigenschaft zählen. Diese beziehen sich jeweils auf alle Objekte einer Gruppe, welche sich aus dem Bezeichner der entsprechenden Eigenschaft ergibt. Dieser setzt sich zusammen aus dem Namen der Gruppe bzw. des *kind*(kname) und dem Namen der Eigenschaft bzw. des *properties*(pname).

Beim Hinzufügen einer Eigenschaft wird in der Operation zudem noch ein konkreter Wert angegeben, der der Eigenschaft in den Objekten der Gruppe zugewiesen wird. Für die Umbenennung einer Eigenschaft ist die Angabe eines neuen Bezeichners notwendig, wobei dieser in diesem Fall le-

diglich aus dem neuen Namen(pname) besteht und nicht noch zusätzlich den Gruppennamen(kname) enthält.

Darüber hinaus können Eigenschaften von den Objekten einer Gruppe zu den Objekten einer anderen Gruppe verschoben oder kopiert werden. Dabei wird die Zielgruppe, also die Gruppe, zu deren Objekten die Eigenschaft verschoben oder kopiert werden soll, explizit angegeben. Die Ausgangsgruppe, die die Quelle der zu verschiebenden bzw. kopierenden Eigenschaft darstellt, ergibt sich wiederum aus dem Bezeichner der Eigenschaft.

Da die Evolutionssprache nicht ans eigentliche Schema sondern direkt an die gespeicherten Objekte richtet, können jeder Operation optional eine oder mehrere, konjunktiv verknüpfte, Selektionsbedingungen hinzugefügt werden. Dazu wird der Bezeichner der Eigenschaft anhand welcher selektiert werden soll zusammen mit einem Wert angegeben. Beim Verschieben bzw. Kopieren einer Eigenschaft kann auch eine Join-Bedingung angegeben werden, indem anstatt des Wertes einfach ein weiterer Eigenschafts-Bezeichner angegeben wird. Darüber hinaus können weitere Selektionsbedingungen für Ausgangs- und/oder Zielgruppe für diese Operationen definiert werden. Die jeweiligen Änderungen werden dann nur an den Objekten der Gruppe durchgeführt, welche alle der angegebenen Kriterien erfüllen.

Zusätzlich zur Syntax der Evolutionsoperationen wird eine weitere Eigenschaft für die Instanz-Objekte definiert, welches die Versionsnummer des Objekts beinhaltet. Nach jeder durchgeführten Änderungsoperation wird dieser Wert erhöht. Dies ermöglicht die parallele Existenz von Objekten mit unterschiedlichen Schema-Versionen in der selben Gruppe.

Kapitel 5

Konzeption

5.1 Allgemeines

Das Ziel dieser Arbeit ist eine Software-Komponente, mit deren Hilfe ein Schema-Evolutionsprozess auf einer MongoDB-Datenbank durchgeführt werden kann. Da in MongoDB JSON-Dokumente, bzw. JSON-ähnliche Dokumente gespeichert werden, bietet es sich an, die Struktur der zu speichernden Daten mit Hilfe von JSON-Schema zu definieren und diese Schema-Definitionen ebenfalls als Dokumente auf der Datenbank abzulegen. Im Laufe eines Schema-Evolutionsprozesses sollen nun Änderungen an einem Schema und entsprechende Änderungen an den dazugehörigen Daten vorgenommen werden. Gesteuert werden soll dieser Prozess lediglich durch eine Reihe einfacher Evolutionsoperationen, aus denen die entsprechenden Updates auf der verwendeten Datenbank generiert werden müssen. In diesem Kapitel wird nun das Konzept zur Übersetzung der Evolutionsoperationen vorgestellt.

In Abschnitt 5.2 wird zunächst Grundsätzliches zur Struktur der Datenbank und der gespeicherten Dokumente erläutert, bevor die vorgenommenen Erweiterungen an der verwendeten Schema-Evolutionssprache vorgestellt werden (5.3). Anschließend wird dann zunächst der Ablauf des Evolutionsprozesses (5.4), und dann die Generierung der Updates für die Schemata (5.5), die selektierten Dokumente (5.6) und die Datenmigration (5.7) erläutert.

5.2 Aufbau von Datenbank und Dokumenten

Um aus den Evolutionsoperationen der Schemasprache die dazugehörigen Updates für die MongoDB-Datenbank effizient generieren zu können, werden an dieser Stelle zunächst einheitliche Strukturen für die Datenbank und die gespeicherten Dokumente festgelegt. Im Folgenden wird dazu der grundlegende Aufbau der Kollektionen der Datenbank, sowie der gespeicherten

Schema- und Instanz-Dokumente beschrieben.

5.2.1 Kollektionen

Eine Datenbank besteht aus mehreren Kollektionen, welche jeweils strukturell ähnliche Dokumente enthalten. Es existiert eine Kollektion mit dem Namen „schemas“, in der alle Schema-Dokumente gespeichert sind. Jedes dieser Schemata, welches in verschiedenen Versionen in der Schema-Kollektion vorliegen kann, ist genau einer Kollektion zugeordnet, wobei der Name der Kollektion als Referenz genutzt wird und in den entsprechenden Schemata gespeichert ist. Alle Dokumente einer Kollektion sind gültig für eine Version des zugehörigen Schemas. Der Schema-Kollektion selbst ist kein eigenes Schema zugeordnet.

5.2.2 Schema-Dokumente

Die gespeicherten Schema-Dokumente besitzen die JSON-Schema-Syntax, wie in Abschnitt 4.2.2 beschrieben. Zusätzlich dazu werden an dieser Stelle einige Pflichtfelder definiert, die für die einzelnen Evolutionsschritte benötigt werden.

Zum einen muss die Möglichkeit bestehen, die zum Schema gehörige Kollektion zu bestimmen. Zu diesem Zweck wird das Feld *collection* definiert, welches den Namen der Kollektion als String enthält. Dabei ist zu beachten, dass dieser nur aus Buchstaben besteht, die weder durch Punkt, Leerzeichen oder andere Sonderzeichen unterbrochen werden. Des Weiteren soll es möglich sein, verschiedene Versionen eines Schemas zu unterscheiden, was das Feld *version* notwendig macht. Der Wert dieses Feldes ist eine Zahl. Abschließend wird noch das Feld *updateoperation* vom Typ String definiert, welches die Evolutionsoperation enthält, die durchgeführt werden muss, um die nächsthöhere Version des Schemas zu erzeugen. Beinhaltet ein Schema-Dokument die aktuellste Version eines Schemas, so enthält das Feld als Wert lediglich einen leeren String. Alle hier definierten Felder befinden sich im Schema auf Wurzelebene.

Nachfolgend findet sich ein einfaches Beispiel für ein dementsprechendes Schema-Dokument.

```
{
  "collection": "user",
  "version": 2,
  "updateoperation": "delete user.email",
  "type": "object",
  "properties": {
      "name": { "type": "string" },
      "email": { "type": "string" } }
}
```

5.2.3 Instanz-Dokumente

Da in der Schema-Kollektion mehrere Versionen eines Schemas existieren können, ist es notwendig, diejenige ermitteln zu können, für die ein bestimmtes Dokument gültig ist. Zu diesem Zweck muss jedes Instanz-Dokument ein Feld *version* enthalten, welches sich im Dokument auf Wurzelebene befindet und als Wert eine Zahl enthält. Somit kann durch die Kombination des Namens der Kollektion, in der ein Dokument gespeichert ist, und des Wertes des version-Feldes das verwendete Schema eindeutig bestimmt werden.

5.3 Schema-Evolutionssprache

Die Evolutionsoperationen, die als Eingabe für den Schema-Evolutionsprozess dienen sollen, basieren auf der in Kapitel 4.4 vorgestellten Schema-Evolutionssprache. In diesem Abschnitt sollen neben den Änderungen an der Definition der Evolutionsoperationen auch Besonderheiten in der Syntax der Evolutionsoperationen beschrieben werden.

5.3.1 Änderungen der Evolutionssprache

Da die verwendete Evolutionssprache aus den grundlegendsten Änderungsoperationen besteht, die auf ein Schema angewandt werden können, existieren vielseitige Möglichkeiten, diese zu erweitern. In diesem Sinne wurden diverse Änderungen an der Definition der Evolutionssprache vorgenommen, um in dieser einige Funktionalitäten zu ergänzen.

Zum einen wurde in der *ADD*-Operation das optionale Schlüsselwort *required* hinzugefügt. Wird dieses hinter dem Schlüsselwort *add* gesetzt, so wird für das hinzugefügte Feld im entsprechenden JSON-Schema die *required*-Funktion gesetzt. Zum anderen wird beim Hinzufügen eines neuen Feldes das Festlegen eines optionalen Default-Wertes ermöglicht. Dazu kann der entsprechende Wert am Ende der ADD-Operation durch *default = v* angegeben werden, wobei v einem konkreten Wert entspricht.

Weitere Änderungen betreffen die Selektionskriterien, die angegeben werden können, um die Auswahl der Instanz-Dokumente zu bestimmen, auf denen die entsprechende Änderung durchgeführt werden soll. Diese sind, wie auch in der ursprünglichen Version der Evolutionssprache optional. Lediglich für die *MOVE*- und *COPY*-Operation wird die Angabe der Join-Kriterien vorausgesetzt, um die korrekte Ausführung der Änderungen an den Instanz-Dokumenten zu gewährleisten. Die Angabe von weiteren Selektionskriterien, die die Auswahl der Dokumente von Ausgangs- und Zielkollektion für die *MOVE*- bzw. *COPY*-Operation ist wiederum optional.

Des Weiteren wird für die Selektionskriterien die Verwendung von weiteren Vergleichsoperatoren zugelassen. So wird durch die Verwendung von $<$, $>$, $<=$ und $>=$ unter anderem auch die Konstruktion von *range queries*

```
evolutionop ::= add | delete | rename | move | copy;

add ::= "add" [ "required" ] property "=" value [ "where" conds ] [ defaultvalue ];
delete ::= "delete" property [ "where" conds ];
rename ::= "rename" property "to" pname [ "where" conds ];
move ::= "move" property "to" kname "where" complexconds;
copy ::= "copy" property "to" kname "where" complexconds;

conds ::= cond { "and" cond };
complexcond ::= joincond { "and" conds };

joincond ::= property compop property;
cond ::= property compop value;
compop ::= "=" | "<" | ">" | "<=" | ">=" | "<>";

defaultvalue ::= "default" "=" value;

property ::= kname "." pname;
kname ::= identifier;
pname ::= identifier;
```

Abbildung 5.1: Angepasste Schema-Evolutionssprachein EBNF

ermöglicht. Zudem wird auch der Vergleichsoperator <> (not equal) in die Definition mit aufgenommen.

Abbildung 5.1 zeigt die vollständige Definition der Eingabesprache für den Evolutionsprozess als EBNF.

5.3.2 Besonderheiten der Syntax

Ein Feld wird in der verwendeten Evolutionssprache durch einen Ausdruck der Form *kname.pname* adressiert. Der Bezeichner *kname* steht dabei für den Namen der Kollektion, in dem sich das entsprechende Dokument befindet. Der Bezeichner *pname* enthält neben dem Namen des adressierten Feldes auch die Namen aller übergeordneten Felder, sofern es sich um ein geschachteltes Feld handelt. Somit setzt sich die Bezeichnung eines Feldes aus seinem „Pfad“ im Dokument zusammen, welcher in Punktnotation ausgedrückt wird.

Die zweite Besonderheit der Syntax bildet die Schreibweise von konkreten Werten in den Evolutionsoperationen. Diese wird festgelegt, um gegebenenfalls den Typ eines Wertes bestimmen zu können. Dies ist beispielsweise beim Einfügen eines Wertes notwendig, damit der Typ im zugehörigen Schema definiert werden kann. Die Werte können dabei generell die primitiven Datentypen besitzen, die für JSON-Dokumente definiert sind. Dabei werden Strings als einzige in Anführungszeichen gesetzt. Zahlen können sowohl ein negatives Vorzeichen, als auch eine Bruch- und Exponententeil besitzen.

Zudem sind *true*, *false* und *null* gültige Werte und werden anhand dieser Schlüsselworte als Boolesche Werte bzw. als Wert vom Typ *null* erkannt.

Abschließend wird festgelegt, dass Feldern, die durch die ADD-Operation zu einem Dokument hinzugefügt werden, noch zwei weitere Formen von Werten zugeordnet werden können. Zum einen kann ein hinzugefügtes Feld vom Typ *object* sein, was durch den Wert { } bestimmt wird. Dabei ist zu beachten, dass auf diese Weise nur leere Objekte hinzugefügt werden können. Das Einfügen von JSON-Syntax in die geschweiften Klammern wird nicht erlaubt, da auf diese Weise Änderungen am Schema durchgeführt werden können, die nicht über die Evolutionsoperationen gesteuert werden. Zum anderen können durch die ADD-Operation Felder vom Typ *array* hinzugefügt werden, indem der Wert aus einer, von eckigen Klammern umschlossenen Liste der oben beschriebenen primitiven Werte besteht, welche durch Komma voneinander getrennt sind. Ein leeres Array der Form [ ] wird als Wert nicht gestattet.

5.4 Ablauf des Evolutionsprozesses

Die Operationen der verwendeten Schema-Evolutionssprache bewirken in erster Linie Änderungen bei einer Auswahl von Instanz-Dokumenten, welche anhand der angegebenen Selektionskriterien bestimmt wird. Der Gesamtprozess der Schema-Evolution beinhaltet jedoch noch weitere Aufgaben. Dazu zählt zum einen die Anpassung der Instanz-Dokumente einer Kollektion, die nicht in dieser ersten Auswahl enthalten sind, und zum anderen natürlich die entsprechende Modifikation des dazugehörigen Schemas.

Bei der Ausführung einer Evolutionsoperation soll zunächst die Änderung am Schema-Dokument vorgenommen werden. Dazu wird die aktuellste Version des Schemas gesucht und von diesem Dokument eine Kopie angefertigt, welche lediglich eine andere ID und einen um 1 erhöhten Wert im Feld *version* erhält. Zudem wird die aufgerufene Evolutionsoperation im alten Schema-Dokument gespeichert, um den Evolutionsschritt auch später noch nachvollziehen zu können. Anschließend wird das entsprechende Update auf der neuen Schema-Version, wie in Abschnitt 5.5 beschrieben, durchgeführt.

Nach der Anpassung des Schemas folgt die Migration der Dokumente, die den angegebenen Selektionskriterien genügen. Dies erfolgt durch die in Abschnitt 5.6 erläuterten Verfahren.

Der letzte Schritt des Evolutionsprozesses besteht aus der Migration aller übrigen Dokumente einer Kollektion. Abhängig von der Migrationsstrategie, also *eager* oder *lazy*, erfolgt dies als zusammenhängender Prozess zum Abschluss der Evolution, oder schrittweise parallel zur Nutzung der Datenbank. Einzelheiten zur Migration der Instanz-Dokumente werden in Abschnitt 5.7 beschrieben.

Der Ablauf des gesamten Evolutionsprozesses wird in Abbildung 5.2 dar-

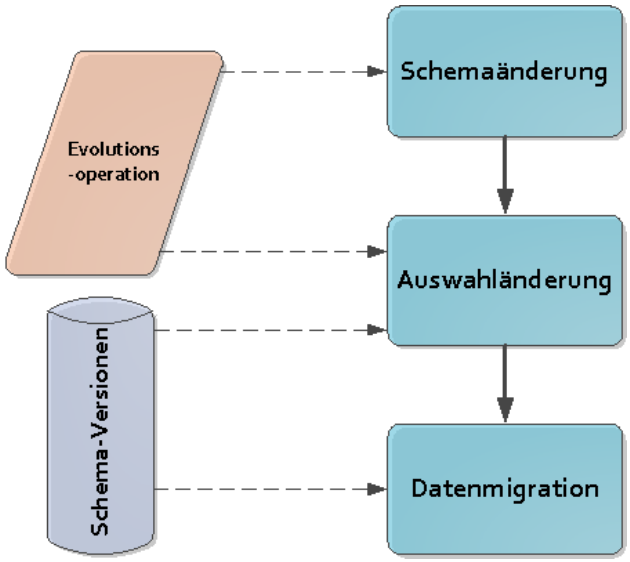

Abbildung 5.2: Ablauf des Schema-Evolutionsprozesses

gestellt.

5.5 Anpassen des Schemas

Die Besonderheit bei der Ausführung von Updates auf die Schema-Dokumente liegt in der Adressierung der einzelnen Felder. Da die Evolutionsoperationen sich auf die Instanz-Dokumente beziehen, werden die Felder dementsprechend adressiert, also beispielsweise in der Form *<parent.child>*. Die Syntax von JSON-Schema legt jedoch fest, dass Felder jeweils als Properties eines JSON-Objects definiert werden. Um das Feld aus dem obigen Beispiel im entsprechenden Schema zu adressieren, muss der Ausdruck zu *<properties.parent.properties.child>* geändert werden. Für die Übersetzung der Evolutionsoperationen müssen die Feldbezeichner jeweils dementsprechend angepasst werden.

Wie bereits erwähnt, wird im ersten Schritt der Schema-Anpassung das aktuellste Schema kopiert und mit einer neuen Versionsnummer gespeichert. Die Änderungen werden dann auf der neuen Version durchgeführt. Nachfolgend wird dieser Vorgang vorausgesetzt und für die neue höchste Versionsnummer in den Anfragen die Variable *max* genutzt, um das zu ändernde Schema auszuwählen.

5.5.1 Add

Beim Hinzufügen eines Feldes zu einem Schema gibt es mehrere Aspekte, die es zu betrachten gilt. Zum einen wird im Schema der Datentyp eines

Feldes definiert. Dazu muss dieser zunächst anhand des in der Evolutionsoperation übergebenen Wertes, wie in Abschnitt 5.3.2 definiert, bestimmt werden. Zum anderen ist es möglich, dass der Bezeichner des hinzuzufügenden Feldes übergeordnete Felder enthält, welche im Schema noch nicht definiert sind. In diesem Fall müssen auch diese Felder erst nacheinander dem Schema hinzugefügt werden.

Ein weiterer Fall, der beachtet werden muss, ist der, dass ein Feld, das hinzugefügt werden soll, bereits existiert. Eine Möglichkeit, damit umzugehen, ist, vor der Ausführung der ADD-Operation zu prüfen, ob ein entsprechendes Feld bereits im Schema vorhanden ist, und dann zu entscheiden, beispielsweise durch Interaktion des Nutzers, ob das vorhandene Feld durch die neue Definition ersetzt werden soll.

Nachfolgend wird anhand von einem allgemeinen Beispiel die Übersetzung der ADD-Operation erläutert. Gegeben ist folgende Operation:

ADD *collection.field* = *value* WHERE <*conditions*>

Um die Änderung in dem entsprechenden Schema-Dokument auf der Datenbank durchzuführen, wird die Operation wie folgt als MongoDB-Operation formuliert:

```
db.schemas.update(
  { collection: "collection", version: max},
  {
    $set: { "properties.field.type": <typeOf(value)> }
  },
  {}
)
```

In dieser Update-Operation wird das Feld selbst und der Typ des Feldes auf einmal definiert. Besteht der Feldbezeichner aus mehreren, geschachtelten Feldern, so werden diese alle in einer set-Operation erstellt, da auch übergeordnete Felder, die noch nicht existieren, automatisch erzeugt werden. Die Typen wiederum müssen einzeln definiert werden, wobei der Typ aller übergeordneten Felder auf *object* gesetzt werden muss.

Für den Fall, dass das hinzugefügte Feld vom Typ *array* ist, muss noch eine weitere set-Operation dem Update hinzugefügt werden, durch die die Items des Arrays und deren Datentyp definiert werden:

```
$set: {"properties.field.items.type": <typeOf(value)>}
```

Abschließend wird die *required*-Option und der Default-Wert für das hinzugefügte Feld gesetzt, sofern diese in der Evolutionsoperation mit angegeben wurden. Das *required*-Feld wird dabei auf dm Level des *properties*-Feldes

gesetzt, in dem sich das hinzugefügte Feld befindet. Als Wert enthält es ein Array, in welches der Name des hinzugefügten Feldes durch folgende Updateoperation eingefügt wird:

```
$push: { "required": "field" }
```

Ein möglicher Default-Wert wird auf dem Level des *type*-Feldes im Schema gespeichert. In dem Feld *default* wird der entsprechende Wert wie folgt hinzugefügt:

```
$set: { "properties.field.default": value }
```

5.5.2 Delete

Bei der Ausführung einer Löschoperation für ein bestimmtes Feld ist es notwendig, zu prüfen, ob sich weitere Bestandteile des Schema-Dokuments auf dieses Feld beziehen. So können beispielsweise Abhängigkeiten zu anderen Feldern definiert sein (*dependencies*), oder die *required*-Option kann für das zu löschende Feld gesetzt sein, welche jedoch an einem anderen Ort im Schema gespeichert wird. Um die Integrität des Schemas zu erhalten, müssen solche Fälle bedacht und die Referenz auf das gelöschte Feld ebenfalls entfernt werden.

Um die Übersetzung der DELETE-Operation in eine entsprechende MongoDB-Funktion zu erläutern, sei folgendes allgemeines Beispiel gegeben:

DELETE *collection.field* WHERE *<conditions>*

Die Löschung des Feldes, ohne Behandlung der oben erwähnten Referenzen, wird auf der Datenbank durch folgende Operation durchgeführt:

```
db.schemas.update(
  {
    collection: "collection",
    version: max},
  {
    $unset: { "properties.field": "" }
  },
  {}
)
```

Dabei wird nur das Feld unter Angabe des entsprechendes Pfades im Bezeichner gelöscht. Anschließend muss durch weitere Operationen im restlichen Schema ein möglicher Bezug auf das gelöschte Feld entfernt werden.

Gezeigt werden soll das am Beispiel des Entfernens der *required*-Option, welche in einem *required*-Feld definiert wird, dessen Wert ein Array ist, das die Namen aller Pflichtfelder enthält. Die nachfolgende Update-Operation prüft durch die Selektionskriterien, ob im Schema ein *required*-Feld an der entsprechenden Position existiert, welches ein Array mit dem Namen des gelöschten Feldes enthält. Falls ja, wird dieses Element im Array gelöscht.

```
db.schemas.update(
  {
    collection: "collection",
    version: max,
    required: { $all: [ "field" ] }
  },
  {
    $pull: { required: "field" }
  },
  {}
)
```

Für andere Referenzen auf ein gelöschtes Feld, also zum Beispiel *dependencies* wird anschließend analog vorgegangen.

5.5.3 Rename

Das Umbenennen eines Feldes in einem Schema ist im Allgemeinen eine unkomplizierte Operation, da keine strukturellen Änderungen am Schema vorgenommen werden. Trotzdem muss, ähnlich wie beim Löschen eines Feldes, geprüft werden, ob im Schema an anderen Stellen Bezug auf das jeweilige Feld genommen wird. Beispiele sind dafür wieder die *required*-Option oder definierte Abhängigkeiten zu anderen Feldern.

Folgendes allgemeines Beispiel für eine RENAME-Evolutionsoperation wird als EIngabe vorausgesetzt:

RENAME *collection.field* TO *newField* WHERE <*conditions*>

Die Übersetzung in eine entsprechende MongoDB-Operation sieht dann wie folgt aus:

```
db.schemas.update(
  {
    collection: "collection",
    version: max},
  {
    $rename: {
        "properties.field": "properties.newField"
    }
  },
  {}
)
```

Dabei ist zu beachten, dass beim Setzen des neuen Namens nicht nur der Name selbst, sondern auch der Pfad zum dazugehörigen Feld mitangegeben wird. Grund dafür ist das Verhalten des *rename*-Operators, der intern erst ein *unset* und dann ein *set* durchführt. Wird nur der Name ohne Pfad angegeben, wird ein neues Feld mit dem Namen auf Wurzelebene erstellt.

Die Änderung des Feldnamens in anderen Zusammenhängen im selben Schema soll wieder anhand der *required*-Option gezeigt werden. Dazu wird in dem entsprechenden Array wiederum zunächst der alte Name entfernt und der neue hinzugefügt:

```
db.schemas.update(
  {
    collection: "collection",
    version: max,
    required: { $all: [ "field" ] }
  },
  {
    $pull: { required: "field" },
    $push: { required: "newField" }
  },
  {}
)
```

5.5.4 Copy

Beim Kopieren eines Feldes von einem Ausgangsschema in ein Zielschema wird nicht nur das eigentliche Feld kopiert, sondern auch der Ort an dem sich das Feld befindet, also alle übergeordneten Elemente. Eventuell definierte Abhängigkeiten in Form von *dependencies* werden nicht mit kopiert, da diese sich auf andere Felder beziehen könnten, welche im Zielschema nicht existieren.

Ausgangspunkt für das Kopieren eines Feldes bildet die Evolutionsoperation COPY, die in der allgemeinen Form wie folgt aussieht:

COPY *collectionA.field* TO *collectionB* WHERE *<complexconditions>*

Die Ausführung der COPY-Operation auf der Datenbank erfolgt durch den folgenden Funktionsaufruf:

```
db.schemas.find({
    collektion: "collectionA",
    version: maxA*}
).forEach(
 function(doc) {
  db.schemas.update(
   {
    collektion: "collectionB",
    version: maxB
   },
   {
    $set: {
     "properties.field": doc.properties.field
                }
   },
    {}
  );
 }
)
```

Dabei wird zunächst über die *find*-Funktion das Ausgangsschema ermittelt. Durch die *forEach*-Funktion wird dann das Ausgangsschema einer Javascript-Funktion übergeben, die wiederum das zu kopierende Feld im Ausgangsschema liest und dieses, durch Aufrufen der Update-Funktion wie sie auch bei der Umsetzung der ADD-Operation definiert wurde, dem Zielschema hinzufügt.

Analog zur DELETE- oder RENAME-Operation wird auch beim Kopieren eines Feldes auf die möglicherweise gesetzt *required*-Option im Ausgangsschema geprüft und diese gegebenenfalls auch im Zielschema gesetzt.

5.5.5 Move

Die MOVE-Operation entnimmt einem Ausgangsschema ein bestimmtes Feld und fügt dieses einem angegebenen Zielschema hinzu. Aufgrund der Ähnlichkeit zur COPY-Operation, wird das Verschieben eines Feldes in ein anderes Schema im Prinzip auf die gleiche Weise umgesetzt. Das entsprechende Feld wird analog kopiert und im Zielschema eingefügt. Der Unterschied besteht jedoch darin, dass das Feld anschließend aus dem Ausgangsschema entfernt

wird. Somit kann die MOVE-Operation also durch die Nacheinanderausführung der bereits definierten COPY- und DELETE-Operationen auf der Datenbank durchgeführt werden.

5.6 Migration der selektierten Instanz-Dokumente

Die Umsetzung der Update-Operationen auf die, durch die angegebenen Selektionskriterien bestimmte, Auswahl von Instanz-Dokumenten stellt den ersten Teil der Datenmigration dar. Die Tatsache, dass die Operationen der Schema-Evolutionssprache diesen Teil des Evolutionsprozesses direkt beschreiben, führt dazu, dass grundsätzlich einige wesentliche Unterschiede zu den Update-Operationen der Schemata existieren. Einer dieser Unterschiede besteht in der Adressierung der zu ändernden Felder. Da die Evolutionsoperationen im Grunde an die Instanz-Dokumente richten, kann der Bezeichner des Feldes fast in der Form in die Datenbank-Operation übernommen werden, wie er in der Evolutionsoperation enthalten ist. Lediglich das *collection*-Präfix wird aus dem Bezeichner entfernt und im Funktionsaufruf zur Adressierung der entsprechenden Kollektion in der Datenbank genutzt. Des weiteren wird für alle Update-Operationen zusätzlich die *multi*-Funktion gesetzt, welche bewirkt, dass die jeweilige Operation auf allen Dokumenten ausgeführt wird, die den angegebenen Selektionskriterien genügen.

Für alle Update-Operationen gilt, dass zu Beginn anhand des *version*-Feldes geprüft wird, ob das Dokument der bis dato aktuellsten Schema-Version entspricht. Sollte dies nicht der Fall sein, muss das Dokument nach und nach an die nächste Version angepasst werden, bis es für die aktuellste Version gültig ist (vgl. Abschnitt 5.7). Nach jeder Anpassung wird dann der Wert des *version*-Feldes um 1 erhöht, um das Dokument der neuen Schema-Version zuordnen zu können.

Im Folgenden wird die Umsetzung der einzelnen Evolutionsoperationen als MongoDB-Funktionen anhand von allgemeinen Beispielen erläutert.

5.6.1 Add

Die ADD-Operation hat im allgemeinen Fall die folgende Form:

ADD *collection.field* = *value*

WHERE *collection.condField* = *condValue*;

Auf die Instanz-Dokumente wird durch die Update-Operation mit Hilfe des *set*-Operators das entsprechende Feld mit dem angegebenen Wert hinzugefügt. Der Funktionsaufruf lautet dann:

```
db.collection.update(
 {
   condField: condValue
 },
 {
  $set: { field: value },
  $inc: { version: 1 }
 },
 { multi: true }
)
```

5.6.2 Delete

Durch die DELETE-Operation wird ein Feld unter Angabe eines Feldbezeichners und einem oder mehrerer Selektionskriterien in einer Auswahl von Dokumenten ein bestimmtes Feld:

DELETE *collection.field*

WHERE *collection.condField* = *condValue*;

Für die Übersetzung in die Update-Operation des Datenbanksystems wird diese mit dem *unset*-Operator aufgerufen, welcher ein bestimmtes Feld eines Dokuments entfernt:

```
db.collection.update(
  {
    condField: condvalue
  },
  {
    $unset: { field: "" },
    $inc: { version: 1 }
  },
  { multi: true }
)
```

5.6.3 Rename

Um den Namen eines Feldes in einer Auswahl von Instanz-Dokumenten zu ändern, wird die RENAME-Operation wie folgt formuliert:

RENAME *collection.field* TO *newField*

WHERE *collection.condField* = *condValue*;

Wie schon bei der Änderung des Schemas muss auch an dieser Stelle beim verwenden des *rename*-Operators darauf geachtet werden, dass der Pfad des Feldes zusätzlich zum neuen Namen mit angegeben werden muss, da sonst das eigentliche Feld samt Wert gelöscht und ein neues Element auf Wurzelebene erzeugt wird. Die entsprechende Update-Operation hat dann folgende Form:

```
db.collection.update(
  {
    condField: condvalue
  },
  {
    $rename: { field: "newField" },
    $inc: { version: 1 }
  },
  { multi: true }
)
```

5.6.4 Copy

Die COPY-Operation bildet zusammen mit MOVE die komplexeste Operation der hier verwendeten Schema-Evolutionssprache. Dabei wird auf Instanz-Dokumente von zwei verschiedenen Kollektionen zugegriffen, wobei sowohl pro Kollektion mehrere Selektionskriterien übergeben werden können, als auch eine Join-Bedingung. Die COPY-Operation hat in der Evolutionssprache die folgende Form:

COPY *collectionA.field* TO *collectionB*

WHERE *collectionA.joinFieldA* = *collectionB.joinFieldB*

AND *collectionA.condFieldA* = *condValueA*

AND *collectionB.condFieldB* = *condValueB*

Die Umsetzung der COPY-Operation auf der Datenbank erfolgt grundsätzlich in zwei Schritten. Zunächst werden mit Hilfe der *find*-Funktion

und den entsprechenden Selektionskriterien alle betreffenden Dokumente des Ausgangsschemas ermittelt, aus denen jeweils ein Feld kopiert werden soll. Durch die *forEach*-Funktion wird dann auf jedes gefundene Dokument eine übergebene Javascript-Funktion angewendet, in der die Selektionskriterien für die Zielkollektion geprüft und gegebenenfalls ein Update ausgeführt werden. Diese Vorgehensweise ist notwendig, da MongoDB eine Join-Operation über mehrere Kollektionen nicht unterstützt. Die Datenbank-Operation zum Ausführen der COPY-Operation sieht demnach folgendermaßen aus:

```
db.collectionA.find(
{ condFieldA: condValueA }
).forEach(

 function(doc) {
  db.collectionB.update(
   {
    condFieldB: condValueB,
    joinFieldB: doc.joinFieldA
   },
   {
    $set: { field: doc.field},
    $inc: { version: 1 }
   },
   {multi: true}
  );
 }
)
```

5.6.5 Move

Wie schon bei der Anpassung des Schemas kann auch bei der Migration der ausgewählten Instanz-Dokumente die MOVE-Operation erzeugt werden, indem zunächst ein COPY und dann ein DELETE auf der Ausgangskollektion durchgeführt werden. Dementsprechend werden dazu die definierten Update-Operationen nacheinander auf der Datenbank aufgerufen.

5.7 Migration der übrigen Instanz-Dokumente

Bei der Migration der Instanz-Dokumente, die nicht zur der initialen Auswahl von Dokumenten gehörten, werden in dieser Arbeit zwei verschiedene Migrationsstrategien verfolgt. Zum einen zählt dazu die *eager*-Migration, bei der nach der Änderung des Schemas direkt alle Instanz-Dokumente and die neue Schema-Version angepasst werden. Nachdem das Update auf die,

durch die Selektionskriterien der Evolutionsoperation bestimmten Instanz-Dokumente durchgeführt wurde, werden alle anderen Dokumente, deren *version*-Feld nicht den aktuellsten Wert besitzt, ebenso dem Update unterzogen. Gegebenenfalls muss die Update-Operation bei älteren Dokumenten dazu aus dem im Schema gespeicherten Evolutionsschritt generiert werden. Nach Abschluss der Migration sind dann alle Dokumente einer Kollektion gültig für das aktuellste Schema. Alle vorherigen Schema-Versionen können somit aus der *schema*-Kollektion entfernt werden.

Die zweite Variante ist die Umsetzung der *lazy*-Migration, wobei Instanz-Dokumente nur dann an das aktuellste Schema angepasst werden, wenn diese das nächste mal verwendet werden. Bei dieser Migrationsstrategie wird der Evolutionsprozess nach dem Update der initialen Auswahl von Instanz-Dokumenten zunächst abgeschlossen, sodass die Datenbank früher wieder verwendet werden kann. Wird dann ein Dokument aufgerufen, sei es durch eine Lese- oder Schreib-Operation, oder durch einen nächsten Evolutionsprozess, muss es zunächst Schritt für Schritt an die aktuellste Version angepasst werden.

Da die Update-Operationen für die Datenmigration im Wesentlichen denen entsprechen, die im Abschnitt 5.6 zur Anpassung der initialen Auswahl von Instanz-Dokumenten definiert wurden, sollen an dieser Stelle lediglich einige Besonderheiten erwähnt werden. Ein wichtiger Unterschied besteht bei einem Update durch die ADD-Operation. Da für die in diesem Teil der Migration zu ändernden Dokumente zum Zeitpunkt des Updates keine konkreten Werte für hinzugefügte Felder vorliegen, kann in diesem Fall zunächst lediglich das neue Feld im Dokument erstellt werden. Als Wert wird dem neuen Feld dabei entweder ein Default-Wert zugewiesen, sofern einer angegeben wurde, oder ein für JSON definierter *null*-Wert. Analog wird für die COPY- und MOVE-Operation verfahren, sofern sich durch die Join-Bedingung aus den Dokumenten der Ausgangskollektion keine konkreten Werte für die Dokumente der Zielkollektion ableiten lassen. Für die RENAME- und DELETE-Operationen ergeben sich keine Unterschiede zu den vorher definierten Update-Funktionen, da diese informationserhaltend bzw. -reduzierend sind.

Kapitel 6

Umsetzung

6.1 Einleitung

Das Ziel dieser Arbeit ist die Umsetzung einer Schema-Evolutionskomponente in Form einer Java-Bibliothek. Diese soll die Operationen einer einfachen Schema-Evolutionssprache als Eingabe akzeptieren und entsprechende Updates an Schema- und Instanz-Dokumenten auf der Datenbank durchführen. Eine wesentliches Merkmal ist dabei die Integrität der gespeicherten Daten, welche sowohl vor, als auch nach der Durchführung der erzeugten Änderungsoperationen vorliegen muss. Dazu zählt neben der Korrektheit der Schema- und Instanz-Dokumente auch die Gültigkeit der Instanz-Dokumente gegenüber dem zugeordneten Schema.

Die Evolutionskomponente wurde für eine MongoDB-Datenbank (Version 2.6.1)[1] entwickelt, welche, wie in Abschnitt 5.2 beschrieben, eingerichtet wurde. Der Zugriff auf die Datenbank erfolgt dabei durch den Java-Treiber für MongoDB (Version 2.12.2)[2].

Nachfolgend werden zunächst die einzelnen Bestandteile der entwickelten Evolutionskomponente und deren Funktionsweise vorgestellt(6.2). Anschließend werden die Schnittstellen der entstandenen Java-Bibliothek beschrieben (6.3) und diverse konzeptionelle Änderungen besprochen, die sich während der Umsetzung ergeben haben (6.4). In Abschnitt 6.5 wird das durchgeführte Verfahren zum Test des Verfahrens beschrieben, bevor in 6.6 einige Erweiterungsmöglichkeiten der vorgestellten Evolutionskomponente diskutiert werden.

6.2 Aufbau und Funktionsweise

Die Evolutionskomponente besteht grundsätzlich aus zwei Teilen, dem Parser und dem Update-Generator (Abb. 6.1). Der Parser prüft die eingehende

[1] `https://www.mongodb.org/downloads`
[2] `http://docs.mongodb.org/ecosystem/drivers/java/`

Evolutionsoperation auf syntaktische Korrektheit und erstellt ein Parser-Ergebnis, welches an den Update-Generator übergeben wird. Dieses enthält die semantischen Informationen der Evolutionsoperation und wird vom Update-Generator verwendet, um die entsprechenden Datenbankoperationen zu erzeugen.

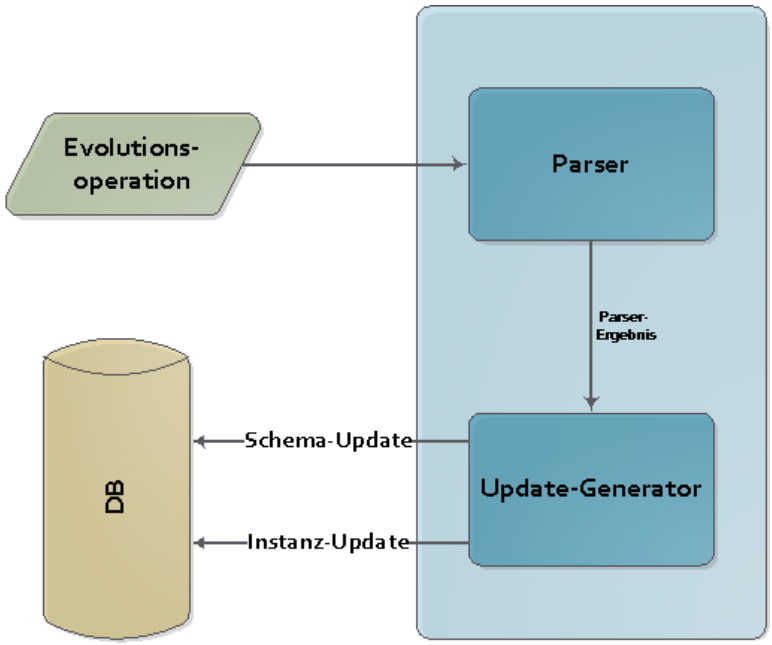

Abbildung 6.1: Schematische Darstellung der Schema-Evolutionskomponente

Im Folgenden werden die Funktionsweise und Besonderheiten der einzelnen Komponenten näher erläutert.

6.2.1 Parser

Die Parserkomponente hat die Aufgabe, eine Operation der Eingabesprache auf Korrektheit zu überprüfen, alle notwendigen Informationen zu filtern und in einem vordefinierten Format an den Update-Generator zu übergeben. Umgesetzt wurde dieser Parser mit Hilfe von JavaCC, einem der gängigsten Java-Parser-Generator, welcher in Form des JavaCC-Plugins (Version 1.5.27)[3] für die Eclipse-Entwicklungsumgebung genutzt wurde. Dabei wurden zunächst mit Hilfe von regulären Ausdrücken alle Token definiert, zu denen unter anderem die Schlüsselwörter der verwendeten Schema-Evolutionssprache, Bezeichner und die verschiedenen Wertetypen (beispielsweise Number oder String) zählen. Mit Hilfe der definierten Token wurde dann die Syntax der Eingabesprache, wie in Abschnitt 5.3 beschrieben, spezifiziert.

[3] `http://eclipse-javacc.sourceforge.net/`

Wird eine Operation an den Parser übergeben, prüft dieser für jedes Token zunächst ob dieses an der jeweiligen Stelle definiert ist. Ist dies nicht der Fall, gibt der Parser eine Fehlermeldung (Abb. 6.2) aus, welche sowohl die Position des Fehlers in der Operation, als auch das gelesene und das erwartete Token enthält. Eine Eingabe-Operation wird vom Parser lediglich bis zum ersten Auftreten eines Zeilenumbruchs, eines *Returns* oder eines Semikolons gelesen.

```
Parser Error:
Encountered " <IDENTIFIER> "test "" at line 1, column 21.
Was expecting one of:
    <STRING> ...
    <NUMBER> ...
    <BOOL> ...
    "null" ...
    <OBJECT> ...
    "[" ...
```

Abbildung 6.2: Beispiel-Fehlermeldung des Parsers

Wenn dem Parser eine syntaktisch korrekte Operation übergeben wird, werden alle Informationen der Operation gefiltert und in einem Objekt der Klasse *ParserResult* gespeichert. Diese wurde definiert, um das Ergebnis des Parsers komplett an andere Funktionen zu übergeben. Alle Details der Evolutionsoperationen, beispielsweise die Art der Operation, das betroffene Feld und die Selektionskriterien, werden dabei in in einem einheitlichen Format gespeichert, sodass die Operation daraus später vollständig rekonstruiert werden kann.

6.2.2 Update-Generator

Der Update-Generator erzeugt anhand der vom Parser gelieferten Informationen die Datenbankupdates und besteht selbst wiederum aus drei Komponenten. Dazu zählen der *Schema-Updater*, welcher die Updates für das jeweilige Schema generiert, der *Selection-Updater*, der die Änderungen an allen Instanz-Dokumenten durchführt, die den Selektionskriterien genügen, und die Migrationskomponente, die alle Instanz-Dokumente gegebenenfalls Schritt für Schritt an die aktuellste Schema-Version anpasst.

Der Schema-Updater

Bei jedem Evolutionsprozess wird zunächst der Schema-Updater aufgerufen, der für jede gültige Evolutionsoperation eine neue Schema-Version erzeugt. Dabei wird zuvor jedoch bei jeder Evolutionsoperationen getestet, ob das betroffene Feld existiert. Wird durch die ADD-Operation versucht, ein Feld hinzuzufügen, welches bereits existiert, oder soll durch eine der anderen Operationen auf ein Feld zugegriffen werden, welches nicht im Schema vorhanden

ist, so wird der Evolutionsprozess mit einer Fehlermeldung abgebrochen. Vor der Ausführung einer COPY- oder MOVE-Operation wird sowohl die Existenz des Feldes im Ausgangsschema geprüft, als auch sichergestellt, dass das Feld im Zielschema noch nicht definiert ist.

Der Selection-Updater

Wurden die Änderungen am Schema fehlerfrei durchgeführt, wird als nächstes der Selection-Updater aufgerufen, welcher die ausgewählten Instanz-Dokumente an die neue Schema-Version anpasst. Dabei wird in der Regel nur eine einzige Update-Operation auf der Datenbank ausgeführt, die alle adressierten Dokumente ändert. Beim Ausführen der MOVE- und COPY-Operation besteht das Update hingegen aus mehreren Datenbankoperationen. Grund dafür ist die Tatsache, dass das entsprechende Update mit Hilfe des Java-Treibers von MongoDB nicht so formuliert werden kann, wie es während der Konzeption in Abschnitt 5.5.4 definiert wurde. Die dort genutzte *forEach*-Funktion, die in der Mongo-Shell verwendet werden kann, wird vom Treiber nicht unterstützt, sodass die Iteration über die Dokumente der Ausgangskollektion in der Evolutionskomponente selbst erfolgen muss. Dazu werden zunächst alle entsprechenden Dokumente bestimmt, aus denen ein Feld kopiert oder verschoben werden soll, und anschließend wird für jedes dieser Dokumente jeweils ein Update auf der Zielkollektion durchgeführt, bei dem alle, der Join-Bedingung genügenden Dokumente entsprechend geändert werden. In diesem Zusammenhang ergab sich während der Implementierung auch die Notwendigkeit einer Join-Bedingung für die MOVE- und COPY-Operation, um das entsprechende Update korrekt ausführen zu können. Ohne diese würde in jedem Iterationsschritt die gleichen Dokumente der Zielkollektion geändert, nämlich die, die den Selektionskriterien für die Zielkollektion genügen. Das würde dazu führen, dass lediglich die Änderungen, die im letzten Iterationsschritt durchgeführt wurden, beibehalten werden und die MOVE- bzw. COPY-Operation ein fehlerhaftes Ergebnis liefert.

Die Migrationskomponente

Den letzten Bestandteil des Update-Generators bildet die Migrationskomponente, welche die Instanz-Dokumente an die aktuellste Schema-Version anpasst. Dabei werden zunächst für alle Versionsnummern, die nicht der aktuellsten entsprechen, alle Instanz-Dokumente gesucht, die unter dieser Schema-Version gespeichert sind. Anschließend werden diese mit Hilfe der im Schema gespeicherten Evolutionsoperation an die nächste Schema-Version angepasst. Diese Anpassung erfolgt schrittweise für alle Dokumente der Versionen 1 bis $M - 1$, wobei M die aktuellste Version darstellt.

Die Operationen, die im Rahmen der Migration durchgeführt werden,

entsprechen prinzipiell denen, die auch im Selection-Updater benutzt werden. Lediglich in den Fällen, in denen ein neues Feld einem Dokument hinzugefügt wird, muss ein Default-Wert (wenn vorhanden) oder ein Nullwert gesetzt werden.

Generell erfolgt die Migration aller Instanz-Dokumente einer Kollektion optional zum Ende eines Evolutionsschrittes, sofern die Variante der *eager*-Migration vom Nutzer gewählt wird. Da jedoch auch die *lazy*-Migration unterstützt werden soll und die parallele Existenz von Dokumenten in verschiedenen Schema-Versionen möglich ist, wird der Migrationsprozess für die vom Selection-Update betroffene Auswahl von Instanz-Dokumenten aufgerufen, bevor weitere Änderungen durchgeführt werden.

6.3 Schnittstellen der Komponente

Die entwickelte Java-Bibliothek bietet als Schnittstelle für den Schema-Evolutionsprozess die Klasse *DBUpdater*, die aus dem Pakte *updategenerator* importiert werden kann. Um ein Objekt dieser Klasse zu erzeugen, wird der Konstruktor mit einer Instanz der Klasse *DB* als Parameter aufgerufen, welche von der MongoDB-API bereitgestellt wird und die Verbindung zur genutzten Datenbank herstellt. Im Folgenden werden alle Methoden vorgestellt, die die *DBUpdater*-Klasse für den Evolutionsprozess bereitstellt.

Um den Evolutionsprozess zu starten und eine Operation der definierten Eingabesprache umzusetzen, bietet der DBUpdater die Methode *processOperation*. Diese kann auf zwei verschiedene Arten aufgerufen werden:

- *processOperation(String operation)*
- *processOperation(String operation, boolean eagerMigration)*

Der Parameter *operation* ist dabei die auszuführende Evolutionsoperation in Form eines Strings. Optional kann ein boolean-Wert als zweiter Parameter übergeben werden, welcher bestimmt, ob die *eager Migration* durchgeführt werden soll, wobei alle Dokumente betreffende Kollektion an die neue Schema-Version angepasst wird. Standardmäßig ist dieser Wert auf *false* gesetzt, sodass die *lazy Migration* als Migrationsstrategie verwendet wird, durch die nur die Dokumente angepasst werden, die den Selektionskriterien der Evolutionsoperation genügen. Für diesen Fall existieren weitere Methoden, durch die der Migrationsprozess separat durchgeführt werden kann:

- *migrateCollection(String collection)*
- *migrateCollection(String collection, BasicDBObject query)*
- *migrateDB()*

Um die Dokumente einer Kollektion an das aktuellste Schema anzupassen, wird die *migrateCollection*-Methode mit dem Bezeichner der Kollektion als Parameter aufgerufen. Dabei kann die Migration wahlweise auf eine bestimmte Auswahl an Dokumenten eingeschränkt werden. Zu diesem Zweck wird ein entsprechendes query in Form eines Objektes der Klasse *BasicDBObject* übergeben, welche wiederum von der MongoDB-API bereitgestellt wird. Da ein solches query auch für den Zugriff einer Anwendung auf die Daten, zum Beispiel durch die *find*- oder *update*-Funktion der MongoDB-API, benötigt wird, kann dieses vorher genutzt werden, um mit Hilfe der Migrations-Methode sicherzustellen, dass die entsprechenden Dokumente vor der Verwendung an die aktuellste Version des Schemas angepasst werden.

Des Weiteren wurde zusätzlich noch die Methode *migrateDB* definiert, welche den Migrationsprozess für alle Kollektionen der verwendeten Datenbank durchführt. Dazu wird zunächst mit Hilfe der gespeicherten Schema-Dokumente eine Liste aller Kollektionen erstellt, welche dann einzeln jeweils komplett an die neueste Schema-Version angepasst werden.

```
Updating Schema:
{ "serverUsed" : "127.0.0.1:27017" , "ok" : 1 , "n" : 1 , "updatedExisting" : true}
Data migration:
{ "serverUsed" : "127.0.0.1:27017" , "ok" : 1 , "n" : 1 , "updatedExisting" : true}
Updating Selection:
{ "serverUsed" : "127.0.0.1:27017" , "ok" : 1 , "n" : 3 , "updatedExisting" : true}
```

Abbildung 6.3: Beispielausgabe des Evolutionsprozesses

Die Ausgabe der vorgestellten Methoden besteht jeweils aus dem durch MongoDB gelieferten *WriteResult* für das Schema-Update, das Selection-Update und der Datenmigration, welche unter Umständen sowohl vor, als auch nach dem Selection-Update durchgeführt wird. Zu erkennen ist daraus unter anderem die Anzahl der Dokumente die von den einzelnen Updates betroffen sind und das eventuelle Auftreten eines Fehlers bei der Ausführung der Datenbankoperation. Abbildung 6.3 zeigt eine beispielhafte Ausgabe der *processOperation*-Methode.

6.4 Konzeptionelle Änderungen während der Umsetzung

Im Laufe der Implementierung der Schema-Evolutionskomponente ergaben sich an einigen Stellen notwendige Modifikationen, aus denen gewisse Unterschiede der Funktionsweise der Komponente zu dem in Kapitel 5 vorgestellten Konzept resultieren. Eine Änderung, die sich während der praktischen Umsetzung ergeben hat, wurde bereits in Abschnitt 6.2.2 erwähnt - die Notwendigkeit einer Join-Bedingung bei der MOVE- und COPY-Operation. Auf Grund der Tatsache, dass diese Operationen ohne eine Join-Bedingung ein

falsches bzw. wahrscheinlich eher ungewolltes Ergebnis liefern würden, wurde diese Änderung auch nachträglich noch im Konzept vorgenommen.

Im Folgenden werden noch weitere Punkte erwähnt, die zwar im Konzept erhalten bleiben, technisch jedoch leicht geändert umgesetzt wurden.

6.4.1 Copy und Move in der Datenmigration

Eine notwendige Modifikation, welche die COPY- und MOVE-Operationen betrifft, ergab sich bei der Migration der Instanz-Dokumente. In der Theorie wurde festgelegt, dass auch beim Migrationsprozess die Join-Bedingung aus der Evolutionsoperation genutzt werden soll, um einzelne Felder (mitsamt Werten) von Dokumenten der Ausgangskollektion in Dokumente der Zielkollektionen zu kopieren bzw. zu verschieben. In der Praxis ergab sich dabei jedoch das Problem, dass Dokumente der Ausgangskollektion zu dem Zeitpunkt, an dem die Dokumente der Zielkollektion an das aktuellste Schema angepasst werden, bereits weitere Evolutionsprozesse durchlaufen haben. Das bedeutet, dass ein zu kopierendes Feld zur Zeit der Migration eventuell nicht mehr existiert, umbenannt oder in der Definition geändert wurde, was ein Problem für die Datenintegrität zur Folge haben kann. Aus diesem Grund wird die Migration für COPY- und MOVE-Operationen behandelt, wie beim Hinzufügen eines Feldes. Das Feld wird in den entsprechenden Dokumenten der Zielkollektion neu erstellt und bekommt einen Default-Wert zugewiesen, sofern dieser in dem zugehörigen Schema für das Feld definiert ist. Fehlt eine entsprechende Definition eines Default-Wertes, wird stattdessen ein Nullwert eingefügt.

Eine andere mögliche Lösung dieses Problems wäre das Hinzufügen weiterer Meta-Informationen zu den Schema- bzw. Instanz-Dokumenten. So könnte beispielsweise die letzte Änderung eines betreffenden Feldes dokumentiert, oder die Schema-Version der Ausgangskollektion mit der Update-Operation gespeichert werden, um bei der Migration festzustellen, ob die zu kopierenden Daten noch unverändert vorliegen. Um den Migrationsprozess nicht zu unübersichtlich zu gestalten und die Schaffung einer Reihe von Sonderfällen zu vermeiden, wurde sich in diesem Fall jedoch dazu entschlossen, alle Dokumente durch das Einfügen von Default- bzw. Nullwerten während der Migration einheitlich behandeln.

6.4.2 Einfügen von Nullwerten

Als Nullwert war ursprünglich vorgesehen, den für JSON und MongoDB gültigen Wert *null* zu verwenden. Während der Entwicklung wurde jedoch festgestellt, dass dieses Vorgehen Probleme bei der Validierung der Instanz-Dokumente gegen das zugehörige Schema erzeugt. Dabei wird *null* nicht als gültiger Wert für ein Feld akzeptiert, dessen Typ im Schema beispielsweise als *string* oder *number* definiert wurde. Der Wert *null* ist nur dann

ein gültiger Wert eines Feldes, wenn dieses als Feld vom Typ *null* definiert wurde.

Zur Lösung dieses Problems boten sich zwei Möglichkeiten an. Die erste bestand daraus, die Typdefinition des Feldes im Schema zu erweitern, indem neben dem eigentlichen Typ des Feldes auch der Typ *null* in der Definition zu verankern. Dadurch könnten die zugehörigen Instanz-Dokumente im entsprechenden Feld sowohl einen konkreten Wert, als auch den *null*-Wert enthalten. Da dieses Vorgehen eine selbstständige, vom Nutzer eventuell unerwünschte Erweiterung des Schemas beinhaltet, wurde dieser Lösungsansatz jedoch wieder verworfen. Stattdessen wurden folgende typabhängige Nullwerte definiert, welche im Falle eines fehlenden Default-Wertes in die Instanz-Dokumente eingefügt werden:

- String: ““(Leerstring)
- Number: 0
- Boolean: *false*
- Array: [ ] (leeres Array)
- Object: { } (leeres Objekt)

Problematisch an dieser Lösung ist jedoch die Tatsache, dass diese Werte unter Umständen von einer Anwendung, die auf die Datenbank zugreift, nicht explizit als Nullwerte erkannt werden. Vor allem für Felder vom Typ *number* oder *boolean* ist diese Entscheidung besonders schwierig, sodass die Definition eines konkreten Default-Wertes in diesem Fall empfehlenswert ist.

6.4.3 Syntax der gespeicherten Schemata

MongoDB ermöglicht grundsätzlich die Speicherung von gültigen JSON-Dokumenten. JSON-Schema-Dokumente sind laut Definition selbst ebenfalls gültige JSON-Dokumente und können somit von MongoDB gespeichert werden. Eine wichtige Besonderheit bilden allerdings einige Schlüsselwörter, welche für JSON-Schema definiert sind und bei der Speicherung in einer MongoDB-Datenbank zu Problemen führen. Dazu zählen die Schlüsselwörter *$schema* und *$ref*.

Grund für die erwähnten Probleme ist das $-Zeichen in den Schlüsselwörtern, welches für MongoDB eine besondere Bedeutung hat. Es wird genutzt, um diverse Operatoren zu kennzeichnen, beispielsweise Vergleichsoperatoren (z.B. *$gt*) oder Updateoperatoren (z.B. *$set*). Der Versuch, ein Schema zu speichern, welches eines dieser Schlüsselwörter enthält, erzeugt demnach eine Fehlermeldung. Um ein Schema dieser Art trotzdem auf der verwendeten Datenbank zu speichern, wäre es also notwendig, entweder die Schema-Definition zu ändern und auf diese Schlüsselwörter zu verzichten, oder das

$-Zeichen jeweils zu ersetzen und das Schema so in eine Form zu bringen, die von MongoDB akzeptiert wird.

Ein weiteres Problem, welches das *$ref*-Schlüsselwort betrifft, ist der Zugriff auf Schemata, die solche Referenzen nutzen. Um auf diese korrekt zugreifen zu können, wäre es notwendig, sie zunächst komplett von der Datenbank zu lesen, die oben beschriebene Umformung rückgängig zu machen und anschließend die enthaltenen Referenzen aufzulösen. Zu diesem Zweck wäre eine Komponente wie das *Jackson JSON Schema Module*[4] notwendig, welches allerdings bisher erst für den JSON-Schema Draft 3 erhältlich ist. Aus diesem Grund und weil es grundsätzlich das Ziel war, alle Änderungsoperationen, sowohl für die Instanz- als auch Schema-Dokumente, als Datenbankoperationen umzusetzen und diese direkt auf der Datenbank auszuführen, werden durch die Evolutionskomponente nur Schema-Definitionen unterstützt, die auf die Nutzung von Referenzen verzichten. Dies schränkt zwar die Art und Weise der Schema-Definition ein, nicht jedoch die Möglichkeiten die JSON-Schema bietet, um die Struktur von Daten zu definieren.

6.5 Test der Evolutionskomponente

Bereits während der Implementierung des Evolutionsprozesses wurden die einzelnen Komponenten mehrfach anhand einfacher Beispiel getestet, um die korrekte Funktionsweise sicherzustellen. Zum Abschluss des Entwicklungsprozesses soll die komplette Evolutionskomponente nochmals getestet werden, indem eine Reihe von Evolutionsoperationen nacheinander auf einer dafür angelegten Testdatenbank ausgeführt werden. Dabei wird zum einen nach jedem abgeschlossenen Evolutionsschritt manuell überprüft, ob die durchgeführten Änderungen an Schema- und Instanz-Dokumenten der eingegebenen Evolutionsoperationen entsprechen. Ebenso wird kontrolliert, ob die Änderungen nur die Instanz-Dokumente betreffen, die den Selektionskriterien der Eingabeoperation genügen. Zum anderen werden sowohl vor dem Test, als auch nach jedem Evolutionsschritt alle Dokumente der betreffenden Kollektion mit Hilfe eines JSON-Schema-Validators(Version 2.2.5)[5] jeweils gegen die entsprechende Schema-Version validiert. Der gesamte Testablauf wurde in einem Eclipse-Projekt implementiert, welches sich zusammen mit der Testdatenbank auf der beigefügten CD-ROM im Anhang A befindet.

Der Test der Evolutionskomponente besteht insgesamt aus zwei Teilen. Im ersten Teil werden nur die *ADD*-, *RENAME*- und *DELETE*-Operationen getestet, da diese sich nur auf ein einzelnes Schema beziehen. Dafür wurde das Schema „person“ (Anhang B.1) erstellt, welches die Attribute Name, Alter und Adresse einer Person definiert. Bei der Definition des Schemas

[4] `https://github.com/FasterXML/jackson-module-jsonSchema`

[5] `https://github.com/fge/json-schema-validator`

wurden nicht nur einzelne Felder als *required* gesetzt, sondern auch *dependencies* für einige Felder definiert, um sicherzustellen, dass die Änderungen durch die RENAME- und DELETE-Operation auch in diesen Bestandteilen des Schemas korrekt vorgenommen werden. Passend zum definierten Schema wurde eine entsprechende Kollektion angelegt, in der zehn Dokumente mit Beispieldaten gespeichert wurden. Auf diese Dokumente und auf das dazugehörige Schema werden die in Abbildung 6.4 dargestellten Evolutionsoperationen in angegebener Reihenfolge durchgeführt.

```
ADD person.telefon = "11834"
        WHERE person.address.country <> "Deutschland"
        DEFAULT = "11833"
```

```
DELETE person.name.middlename
```

```
ADD required person.adult = true
        WHERE person.age >= 18
```

```
RENAME person.address.postalcode TO zipcode
        WHERE person.address.postalcode >= 18059
            AND person.address.postalcode <= 18109
```

```
ADD person.hobbies = ["Fussball", "Musik"]
        WHERE person.name.firstname = "Marcel"
            AND person.name.lastname = "Apfelt"
```

Abbildung 6.4: Testaufrufe der ADD-, RENAME- und DELETE-Operationen

Im Anschluss an diese Evolutionsschritte wird zusätzlich noch die Migrations-Funktion aufgerufen, zunächst für einen Teil der Kollektion, anschließend nochmal für die komplette Kollektion. Auch danach wird zunächst jeweils manuell die korrekte Änderung der Instanz-Dokumente und des Schemas überprüft, um anschließend wiederum die Validierung der Dokumente durchzuführen.

Im zweiten Teil des Tests wird die korrekte Funktionsweise der *COPY*- und *MOVE*-Operation überprüft. Da diese jeweils eine Join-Operation darstellen, wurden zu diesem Zweck zwei einfache Schemata entworfen, die über ein gemeinsames Attribut verfügen. Zum einen wurde das *department*-Schema (Anhang B.2) definiert, welches anhand von Name, Ort und einer ID eine Abteilung beschreibt. In der dazugehörigen Kollektion wurden sechs Abteilungen in entsprechender Form angelegt. Das zweite Schema (*employee*, Anhang B.3) beschreibt die Mitarbeiter anhand von Name, Mitarbeiter-ID und der ID der Abteilung, zu der sie gehören, welche als Join-Bedingung genutzt wird. In der entsprechenden Kollektion wurden zehn Mitarbeiter erstellt, die über die *department_id* jeweils einer Abteilung aus der *department*-Kollektion zugeordnet sind. Abbildung 6.5 zeigt die Operationen, die im Rahmen des Tests der Join-Operationen verarbeitet wurden.

```
COPY department.department_name TO employee
        WHERE department.id = employee.department_id
```

```
MOVE department.location TO employee
        WHERE department.id = employee.department_id
```

```
MOVE employee.location TO department
        WHERE department.id = employee.department_id
```

```
COPY department.location TO employee
        WHERE department.id = employee.department_id
            AND department.department_name = "Softwareentwicklung"
            AND employee.name.lastname = "Fischer"
```

Abbildung 6.5: Testaufrufe der MOVE- und COPY-Operationen

Im Anschluss an die Evolutionsoperationen wird die Migrations-Funktion für die gesamte Datenbank aufgerufen. Abschließend überprüft der Schema-Validator nochmals für alle drei Kollektionen alle Dokumente auf Gültigkeit zum entsprechenden Schema.

6.6 Möglichkeiten zur Erweiterung der Evolutionskomponente

Die in diesem Kapitel vorgestellte Komponente zur Schema-Evolution auf einer MongoDB-Datenbank ermöglicht die Ausführung einfacher Evolutionsoperationen auf einem gewählten Schema und den dazugehörigen Daten. Da die verwendete Evolutionssprache lediglich die grundlegendsten Änderungsoperationen beinhaltet, bietet besonders diese einige Möglichkeiten zur Erweiterung der Evolutionskomponente. Zwar wurden bereits einzelne Erweiterungen der Evolutionssprache durchgeführt (vgl. Abschnitt 5.3), jedoch existieren für JSON-Schema noch zahlreiche weitere Definitionsmöglichkeiten, wie beispielsweise Enumerations oder Patterns für zusätzliche Felder, die ebenfalls durch den Evolutionsprozess unterstützt werden könnten.

Des Weiterhin wurde sich bisher lediglich auf die Nutzung von konjunktiven Selektionsbedingungen beschränkt, wie sie in der genutzten Evolutionssprache definiert wurden. Da MongoDB jedoch auch disjunktive Queries ermöglicht, wäre auch eine entsprechende Änderung der Sprache denkbar, um die Auswahlmöglichkeiten der zu ändernden Instanz-Dokumente zu erweitern.

Kapitel 7

Fazit und Ausblick

7.1 Fazit

Im Rahmen dieser Arbeit wurde eine Komponente entwickelt, die einen Schema-Evolutionsprozess für eine NoSQL-Datenbank ermöglicht, die grundsätzlich nicht über ein festgelegtes Schema verfügt. Dazu wurde zunächst der Aufbau einer entsprechenden Datenbank definiert, in welcher sowohl Schema-Dokumente als auch Instanz-Dokumente in verschiedenen Versionen gespeichert werden, die mittels diverser Metadaten einander zugeordnet werden können. Auf der Grundlage einer gegebenen, teilweise noch erweiterten Evolutionssprache, wurde ein Konzept entwickelt, um eine Reihe von grundlegenden Evolutionsoperationen in entsprechende Datenbankupdates umzuwandeln, durch die die Evolution eines Schemas und die Migration der dazugehörigen Datensätze, sowohl *eager*, als auch *lazy*, realisiert werden kann.

Die Umsetzung des vorgestellten Konzepts resultierte in eine Schema-Evolutionskomponente in Form einer Java-Bibliothek. Diese generiert aus einer eingegebenen Evolutionsoperation die entsprechenden Updates für die Datenbank und führt diese direkt aus. Dabei werden alle Änderungsoperationen direkt auf der Datenbank ausgeführt, sodass die Effizienz eines Evolutionsschrittes zum großen Teil von datenbankinternen Optimierungen, wie zum Beispiel definierten Indexen, beeinflusst werden kann.

Neben diversen Funktionstests während der Entwicklung wurde auch ein abschließender Test aller Evolutionsoperationen erfolgreich durchgeführt, welcher zusätzlich noch eine Schema-Validation nach jedem Evolutionsschritt beinhaltete. Daraus ergibt sich, dass das Ergebnis dieser Arbeit eine Schema-Evolutionskomponente ist, mit der die grundlegendsten Schema-Evolutionsoperationen, inklusive Datenmigration, auf einer MongoDB-Datenbank korrekt durchgeführt werden können, wobei die Integrität der geänderten Daten erhalten wird.

7.2 Ausblick

Die strukturelle Evolution von Daten stellt im Allgemeinen für alle Datenbanksysteme eine Herausforderung dar. Speziell für NoSQL-Datenbanken ist diese besonders groß, da sie in der Regel weder um ein festgelegtes Schema, noch um die Möglichkeit verfügt, ein solches zu definieren oder zu verwalten. Um dies dennoch zu ermöglichen, ist also die Entwicklung von externen Komponenten notwendig, die diese und andere Schema-Management-Aufgaben übernehmen.

Die Realisierbarkeit solcher Komponenten für die verschiedenen NoSQL-Systeme hängt dabei jeweils von den Eigenschaften der einzelnen Systeme ab. In dieser Arbeit wurde die Schema-Evolution für ein bestimmtes dokumentorientiertes Datenbanksystem erfolgreich umgesetzt. Analoge, auf ähnlichen Konzepten basierende Entwicklungen für weitere dokumentorientierte Systeme mit ähnlichen Eigenschaften sind demnach denkbar. Auch spaltenorientierte Datenbanken ermöglichen einen Schema-Evolutionsprozess, da sie grundsätzlich über ein Schema verfügen, welches einzelne Attribute definiert. Auf Grund der Tatsache, dass alle Werte einer Spalte physisch zusammen gespeichert werden, eignen sich spaltenorientierte Datenbanken optimal zum Einfügen, Löschen oder Ändern einer kompletten Spalte, also Operationen, die typischerweise während eines Evolutionsschrittes durchgeführt werden. Für andere NoSQL-Systeme, wie zum Beispiel Key/Value-Datenbanken, ist das Schema-Management wiederum irrelevant, da für diese die Struktur der gespeicherten Daten in der Regel vollständig unbekannt ist.

Anhang A

Inhalt der CD-ROM

Die beigelegte CD-ROM enthält neben einer digitalen Version dieser Arbeit auch die verwendete Literatur, welche im Ordner *Literatur/* in Form von pdf-Dateien oder gespeicherten Webseiten zu finden ist. Des Weiteren befinden sich im Ordner *Implementierung/* sowohl das Eclipse-Projekt der entwickelten Evolutionskomponente, als auch das Projekt, in welche der entworfene Testablauf implementiert wurde. Alle verwendeten Bibliotheken, sowie sonstige Werkzeuge oder Programme sind in dem Ordner *Software/* enthalten. Dazu zählen:

- die MongoDB-Installationsdatei (Version 2.6.1)
- der Java-Treiber für MongoDB (Version 2.12.2)
- die JSON-Schema-Validator-Bibliothek für Java (Version 2.2.5)
- das JavaCC-Plugin für Eclipse (Version 1.5.27)

Des Weiteren befindet sich im Ordner *Testdatenbank/* ein Abbild der für die Tests genutzten Datenbank 'testdb', welches mit Hilfe von *mongodump* erstellt wurde. Um diese Datenbank auf eine aktive *mongod*-Instanz zu schreiben, lautet:

mongorestore –drop –db testdb /../Testdatenbank/testdb

Das Ergebnis der Arbeit, also die Schema-Evolutionskomponente in Form der Java-Bibliothek *MongoEvolution.jar* befindet sich zusammen mit dieser Arbeit in pdf-Form im Wurzelverzeichnis der CD.

Anhang B

Testschemata

B.1 *person*-Schema

```
{
   "type":"object",
   "additionalProperties": false ,
   "description":"A person",
   "collection":"person",
   "version":1,
   "properties":{
      "name":{
         "type":"object",
         "properties":{
            "firstname":{
               "type":"string"
            },
            "middlename":{
               "type":"string"
            },
            "lastname":{
               "type":"string"
            }
         },
         "additionalProperties": false ,
         "dependencies":{
            "middlename":[
               "firstname"
            ]
         }
      },
      "age":{
```

```
            "type":"number"
        },
        "address":{
            "properties":{
                "street":{
                    "type":"string"
                },
                "postalcode":{
                    "type":"number"
                },
                "state":{
                    "type":"string"
                },
                "country":{
                    "type":"string"
                },
                "city":{
                    "type":"string"
                }
            },
            "required":[
                "postalcode",
                "city",
                "country"
            ],
            "additionalProperties": false
        }
    },
    "required":[
        "age"
    ]
}
```

B.2 *department*-Schema

```
{
   "type":"object",
   "additionalProperties": false ,
   "description":"A department",
   "collection":"department",
   "version":1,
   "properties":{
      "department_name":{
         "type":"string"
      },
      "id":{
         "type":"number"
      },
      "location":{
         "type":"string",
         "default":"unknown"
      }
   },
   "required":[
      "department_name",
      "id",
      "location"
   ]
}
```

B.3 *employee*-Schema

```
{
   "type":"object",
   "additionalProperties": false ,
   "description":"An employee",
   "collection":"employee",
   "version":1,
   "properties":{
      "name":{
         "type":"object",
         "properties":{
            "firstname":{
               "type":"string"
            },
            "lastname":{
               "type":"string"
            }
         },
         "required":[
            "firstname",
            "lastname"
         ],
         "additionalProperties": false
      },
      "employee_id":{
         "type":"number"
      },
      "department_id":{
         "type":"number"
      }
   },
   "required":[
      "department_id",
      "employee_id"
   ]
}
```

Literaturverzeichnis

[Bre00] Brewer, Eric A.: *Towards Robust Distributed Systems (Vortrag auf der ACM-PODC-Konferenz).* 2000

[BSO14] *BSON-Spezifikation.* `http://bsonspec.org/`, 2014. – aufgerufen 11.07.2014

[CB14] Crockford, Douglas ; Bray, Tim: *RFC 7159 - The JavaScript Object Notation (JSON) Data Interchange Format.* `http://tools.ietf.org/html/rfc7159`, March 2014. – aufgrufen 11.07.2014

[CMZ08] Curino, Carlo A. ; Moon, Hyun J. ; Zaniolo, Carlo: Graceful Database Schema Evolution: The PRISM Workbench. In: *Proc. VLDB Endow.* 1 (2008), Nr. 1, S. 761–772

[CTMZ08] Curino, Carlo A. ; Tanca, Letizia ; Moon, Hyun J. ; Zaniolo, Carlo: Schema evolution in wikipedia: toward a web information system benchmark. In: *In ICEIS*, 2008

[DBL12] *Datenbanken Online Lexikon der FH Köln.* `http://wikis.gm.fh-koeln.de/wiki_db/Datenbanken/Skalierbarkeit`, 2012. – aufgerufen 22.05.2014

[DG04] Dean, Jeffrey ; Ghemawat, Sanjay: MapReduce: simplified data processing on large clusters. In: *OSDI 04: Proceeding of the 6th conference on symposium on operating systems design and implementation*, USENIX Association, 2004

[DT03] Deutsch, Alin ; Tannen, Val: MARS: A System for Publishing XML from Mixed and Redundant Storage. In: *In VLDB*, 2003, S. 201–212

[ECM13] *ECMA-404 - The JSON Data Interchange Format.* `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`, October 2013. – aufgerufen 11.07.2014

[Edl14] Edlich, Stefan: *NoSQL-Archiv*. `http://www.nosql-database.org/`, 2014. – aufgerufen 19.05.2014

[EFHB11] Edlich, Stefan ; Friedland, Achim ; Hampe, Jens ; Brauer, Benjamin: *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. Hanser Fachbuchverlag, 2011. – ISBN 9783446423558

[Fag06] Fagin, Ronald: Inverting Schema Mappings. In: *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. New York, NY, USA : ACM, 2006 (PODS '06). – ISBN 1–59593–318–2, 50–59

[FGC+97] Fox, Armando ; Gribble, Steven D. ; Chawathe, Yatin ; Brewer, Eric A. ; Gauthier, Paul: Cluster-Based Scalable Network Services. In: *Proceedings of the sixteenth ACM symposium on Operating systems principles*, 1997, S. 78–91

[FGM00] Franconi, Enrico ; Grandi, Fabio ; Mandreoli, Federica: Schema Evolution and Versioning: a Logical and Computational Characterisation. In: *In Workshop on Foundations of Models and Languages for Data and Objects*, Springer-Verlag, 2000, S. 85–99

[JSO14] *JSON-Webseite*. `http://json.org/json-de.html`, 2014. – aufgerufen 11.07.2014

[KSS14] Klettke, Meike ; Scherzinger, Stefanie ; Störl, Uta: Datenbanken ohne Schema? Herausforderungen und Lösungs-Strategien in der agilen Anwendungsentwicklung mit schemaflexiblen NoSQL-Datenbanksystemen. In: *Datenbank-Spektrum* (2014)

[Mon13] *MongoDB-Webseite*. `http://www.mongodb.org/`, 2013. – aufgerufen 11.07.2014

[Rod92] Roddick, John F.: Schema Evolution in Database Systems: An Annotated Bibliography. In: *SIGMOD Rec.* 21 (1992), Nr. 4, S. 35–40. – ISSN 0163–5808

[Rod95] Roddick, John F.: A survey of schema versioning issues for database systems. In: *Information and Software Technology* 37 (1995), S. 383–393

[SKS13] Scherzinger, Stefanie ; Klettke, Meike ; Störl, Uta: Managing Schema Evolution in NoSQL Data Stores. In: *DBPL*, 2013

[Stö14] Störl, Uta: NoSQL-Datenbanksysteme. In: *Taschenbuch Datenbanken, 2. Auflage (noch nicht erschienen)* (2014)

[Zan84] Zaniolo, Carlo: Database Relations with Null Values. In: *Journal of Computer and System Science* 28 (1984), S. 142–166

[ZG13] Zyp, Kris ; Galiegue, Francis: *Internet-Draft - JSON Schema: core definitions and terminology.* `http://tools.ietf.org/html/draft-zyp-json-schema-04`, Januar 2013. – aufgerufen 11.07.2014

[Zyp14] Zyp, Kris: *JSON Schema.* `http://json-schema.org/`, 2014. – aufgerufen 11.07.2014

Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, 01.10.2014