Data-Integration Pipeline für die Transformation von NoSQL-Daten (JSON) in relationale Datenbanken

Bachelorarbeit

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik

vorgelegt von:	Felix Beuster
Matrikelnummer:	211207722
geboren am:	18.12.1990 in Bützow
Gutachter:	PD Dr.-Ing. habil. Meike Klettke
Zweitgutachter:	M.Sc. Ilvio Bruder
Betreuer:	PD Dr.-Ing. habil. Meike Klettke
Abgabedatum:	19.08.2016

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Kapitel 1

Einleitung

Im Zuge stetig steigender Nutzerzahlen von Webanwendungen und wachsenden Mengen an zu beobachteten Daten, erfreuen sich NoSQL-Systeme einer großen Beliebtheit. Insbesondere wenn noch keine oder nur grobe Schemadefinitionen vorliegen, lassen sich diese Systeme schneller einrichten und im weiteren Verlauf anpassen, als herkömmliche relationale Datenbanksysteme.
Der „Vorsichtshalber Speichern“ Ansatz vieler Unternehmen, führt zudem zu großen Datenmengen, die bei der Speicherung von einer horizontalen Skalierung stark profitieren. Diese Skalierung ist mit relationalen Datenbanksystem durch das ACID-Prinzip (siehe [SSH13] Kapitel 12.1.1) bedeutend aufwendiger.

Allerdings eignen sich diese Datenbestände vorrangig zur Speicherung der Daten. Anfragen und Suchen in dem Bestand sind wie in Abschnitt 3.4.2 zwar möglich, aber doch aufwendiger als in einer klassischen, relationalen Datenbank. Zudem werden komplexe Anfragen, wie sie in relationalen Modellen bspw. mit Joins möglich sind, meist nicht unterstützt. Für diesen Zweck muss der Datenbestand in ein relationales Schema transformiert werden.

Dieses Schema manuell zu erkennen ist bei hinreichend großen Datenbeständen nicht mehr möglich. Daher gibt es Verfahren, die dieses Schema automatisiert extrahieren können und als strukturiertes JSON-Schema ausgeben. Darüber hinaus ist es möglich auch auf Änderungen des Datenbestandes zu reagieren und das Schema zu aktualisieren. Basierend auf diesem extrahierten Schema lässt sich später ein relationales Datenbankschema ableiten, mit dem die relationale Datenbank aufgebaut werden kann.

Ein solches relationales Schema kann verschiedenen Methoden der Optimierung unterzogen werden, die bereits aus dem Bereich der XML-Verarbeitung bekannt sind. So können Relationen verschmolzen oder eingebettet werden. Optimierungen können aber auch in Hinblick auf die zu erwartende Last und Art der Anfragen erfolgen.

Ist ein passendes relationales Schema gefunden, gibt es auch für den Datentransfer verschiedene Ansätze, insbesondere bei der Behandlung von Datenredundanz. Der finale Datentransfer birgt je nach Größe des Datenbestandes wiederum eigene Herausforderungen. Speicherbedarf für Zwischenergebnisse und Laufzeiten des Transfers sind hier entscheidend und eine verteilte Ausführung oft ebenfalls wünschenswert.

In dieser Arbeit soll neben der Betrachtung existierender Techniken ein Prozess entworfen werden, der eine bestehende NoSQL-Datenbank in eine relationale Datenbank überführt. Darüber hinaus werden auch verschiedene Optimierungsanätze für das erstellte Schema und den Datentransfer untersucht.

Kapitel 2

Datenformate

Schemafreie oder schemaarme Daten können in verschiedenen Formaten vorliegen. Geläufig sind hier unter anderem *JavaScript Object Notation* (JSON) und *Extensible Markup Language* (XML), aber auch *comma-separated values* (CSV) oder *tab-separated values* (TSV). Im Rahmen dieser Arbeit sind vor allem JSON und XML relevant und daher hier näher beschrieben.

2.1 JSON

2.1.1 Beschreibung

JSON ist ein Format zum Strukturieren von Daten. Ecma International beschreibt JSON im Standard als „schlankes, sprach-unabhängiges Datenaustauschformat“ (vgl. [ECM13, Kap. 1]).
Die einfache Syntax (siehe Kapitel 2.1.2) resultiert in einer besseren Lesbarkeit für Anwender, sowie auch schnelleren Verarbeitung in verschiedenen Programmiersprachen als im Vergleich zu Formaten wie bspw. XML.

2.1.2 Struktur

Den Kern von JSON bilden zwei Strukturen: `object` als ungeordnete Liste von Schlüssel-Wert-Paaren und `array` als geordnete Liste von Werten. Beide Strukturen seien hier einmal kurz durch ihre Grammatik in Quelltext 2.1 respektive 2.2 illustriert.

Quelltext 2.1: Grammatik der Struktur `object`

⟨*object*⟩ ::= ‘`{}`’
| ‘`{`’ ⟨*members*⟩ ‘`}`’

⟨*members*⟩ ::= pair
| pair ‘`,`’ members

⟨*pair*⟩ ::= ⟨*string*⟩ ‘`:`’ ⟨*value*⟩

Quelltext 2.2: Grammatik der Struktur `array`

⟨*array*⟩ ::= ‘`[]`’
| ‘`[`’ ⟨*elements*⟩ ‘`]`’

⟨*elements*⟩ ::= ⟨*value*⟩
| ⟨*value*⟩ ',' ⟨*elements*⟩

Zur Ergänzung sei noch `value` in Quelltext 2.3 definiert. Alle weiteren Strukturen entsprechen den allgemein bekannten Datentypen und können ebenfalls im Standard nachgelesen werden.

Quelltext 2.3: Grammatik der Struktur `value`

⟨*value*⟩ ::= ⟨*string*⟩
| ⟨*number*⟩
| ⟨*object*⟩
| ⟨*array*⟩
| 'true'
| 'false'
| 'null'

Anhand der Grammatik von `value` wird deutlich, dass die Strukturen `object` und `array` sich beliebig tief ineinander schachteln lassen. Damit können auch komplexe Datenstrukturen mit verhältnismäßig wenig Syntax dargestellt werden.

Zusammengefasst stellt sich die JSON-Grammatik wie folgt in Quelltext 2.4 dar. Eine Beispiel JSON-Datei ist als Abschluss in Quelltext 2.5.

Quelltext 2.4: Grammatik eines JSON-Dokuments

⟨*document*⟩ ::= ⟨*object*⟩ | ⟨*array*⟩

Quelltext 2.5: Beispielhafte JSON-Datei

```
1 {
2   "name" : "Meier",
3   "alter" : 42,
4   "statusmeldungen" : [
5     { "public" : true, "text" : "Guten Morgen!" },
6     { "public" : false, "text" : "Test des Services" }
7   ],
8 }
```

2.2 XML

2.2.1 Beschreibung

Wie bereits JSON ist auch XML ein Format zur Strukturierung von Daten. Es wurde seit 1996 unter Aufsicht des *World Wide Web Consortium* (W3C) aus der *Standard Generalized Markup Language* (SGML) entwickelt (vgl. [W3C08, Abschnitt 1]).
Eine Gemeinsamkeit zu JSON ist die Zielsetzung ein einfaches Format zu definieren, dass für den universellen Datenaustausch eingesetzt werden kann. Somit ist XML ebenfalls weitestgehend unabhängig von Programmiersprachen und konkreten Programmen.

Große Unterschiede zu JSON gibt es jedoch in der Syntax. Diese ist zwar strukturell ebenfalls einfach gehalten, hat jedoch einen deutlich höheren Aufwand in Bezug auf die Menge der Zeichen.

2.2.2 Struktur

Die Definition des Standards setzt ein XML-Dokument aus bis zu drei Bestandteilen zusammen, wie in Quelltext 2.6 zu entnehmen.
Auch wenn diese Definition einen `prolog` Block zur Festlegung von Dokumenteigenschaften wie der XML-Verison vorsieht, so besteht dieser jedoch aus ausschließlich optionalen Komponenten. Siehe dazu Abschnitt 2.8 in [W3C08]
Der ebenfalls optionale Block `Misc` beinhaltet Kommentare und weitere Verarbeitungsinformationen, die nicht weiter zu den eigentlichen Daten des Dokumentes zählen.

Quelltext 2.6: Grammatik eines XML-Dokuments

⟨*document*⟩ ::= ⟨*prolog*⟩ ⟨*element*⟩ ⟨*misc*⟩*

Als Block für die Daten verbleibt somit nur `element`. Dieses kann zwei Ausprägungen haben. Einerseits gibt es das leere Element, welches einen Namen und eine Liste von Attributen besitzt. Die zweite Variante besteht aus einem beginnendem Tag bestehend aus Namen und Attributliste, sowie einem schließendem Tag der den selben Namen besitzt. Öffnender und schließender Tag umschließen weitere Inhalte und ermöglichen somit eine beliebig tiefe Verschachtlung.

Quelltext 2.7: Grammatik der Struktur `element`

⟨*element*⟩ ::= ⟨*EmptyElemTag*⟩
| ⟨*STag*⟩ ⟨*content*⟩ ⟨*ETag*⟩

Der Block `content` kann sowohl einfachen Text enthalten, als auch weitere Elemente. Eine detailliertere Auflistung findet sich in Abschnitt 3.1 des XML-Standards.

Abschließend sei auch für XML ein Beispiel-Dokument in Quelltext 2.8 angegeben. Für einen besseren Vergleich wird der selbe Datensatz wie im JSON-Abschnitt illustriert.

Quelltext 2.8: Beispielhafte XML-Datei

```
1 <?xml version="1.0"?>
2 <nutzer>
3   <name>Meier</name>
4   <alter>42</alter>
5   <statusmeldungen>
6     <statusmeldung public="true" text="Guten Morgen!" />
7     <statusmeldung public="false" text="Test des Services" />
8   </statusmeldungen>
9 </nutzer>
```

Kapitel 3

Datenbanksysteme

In diesem Abschnitt werden vier unterschiedliche *Datenbanksysteme* (DBS) betrachtet. Neben dem für diese Arbeit wichtigem Bereich der NoSQL-Systeme ist auch ein Einblick in klassische relationale DBS, hierarchische DBS und objektorientierte DBS sinnvoll.

3.1 Hierarchische Datenbanksysteme

3.1.1 Motivation

IBM veröffentlichte 1968 das System IMS. Es ist eines der ersten und erfolgreichsten kommerziellen DBS [Dat90, S. 753]. Mit diesem System hat IBM das *hierarchische Modell* eingeführt, welches eine Einschränkung des Netzwerkmodells ist [SSH13, S.535].

3.1.2 Hierarchisches Modell

Datenstruktur

Eine hierarchische Datenbank besteht aus einer geordneten Liste von Bäumen. Jeder dieser Bäume hat einen Root-Record, der die Datenfelder enthält. Eine Hierarchie ergibt sich dadurch, dass jeder Baum eine Liste von Unterbäumen enthalten kann, die wiederum den selben Aufbau haben.

Datenänderung

Das hierarchische Modell sieht verschiedene Operation zur Datenmanipulation vor, die aus herkömmlicher Softwareentwicklung für die Arbeit mit Baumstrukturen bereits bekannt sind. Dazu gehören unter anderem Operationen zum Finden eines bestimmten Baumes, Operationen zum Navigieren sowohl in einem Baum als in den Records, oder auch für das Hinzufügen, Ändern und Löschen von Records.

Integrität

Durch Befolgen der einfachen Regel, dass kein Kind-Element ohne ein Eltern-Element existieren darf, besitzt das hierarchische Modell bereits eingebaute Integritätsgewährleistung [Dat90, S. 758]. Das Löschen eines Knotens bewirkt ein automatisches Entfernen des gesamten Unterbaums, Einfügen ist nur an existierende Knoten möglich.

In Bezug auf Fremdschlüssel werden Änderungen und Löschungen des Fremdschlüsselknotens an den referenzierenden Knoten weitergegeben, darüber hinaus dürfen Fremdschlüsselverweise nicht leer (also not null) sein [Dat90, S. 758].

3.2 Relationale Datenbanksysteme

3.2.1 Motivation

Relationale Datenbanksysteme sind aktuell die am meisten verbreiteten Systeme. Sie zeichnen sich vor allem durch eine gute Konsistenzwahrung der Daten, Sichtdefinitionen und eine direkte Umsetzung des *Entity-Relationship-Modell* (ERM) nach P. P. Chen [Che76] aus.

3.2.2 Entity-Relationship-Modell

Die Grundlage des ERM bilden die drei Konzepte *Entity*, *Relationship* und *Attribut*. Entities repräsentieren hierbei Modellobjekte aus der realen Welt, wie bspw. Statusmeldungen auf einer Web-Plattform oder deren Nutzer. Relationships stellen Beziehungen zwischen Entities dar, bspw. die Zuordnungen welcher Nutzer welche Statusmeldung geschrieben hat. Attribute sind Eigenschaften von Entities oder Relationships. Im Falle von Statusmeldungen könnten das Inhalt, Datum und Uhrzeit sein.

Erstellt man nun ein ERM für eine Anwendung, so werden Entities mit gleichen Attributen zu einem Entity-Typ zusammengefasst. Gleiches erfolgt analog für Relationships. Dadurch wird das Modell vereinfacht und kann in eine grafische Darstellung wie Abbildung 3.1 übertragen werden. Üblicherweise werden Entities als Rechtecke, Relationships als Rauten dargestellt.

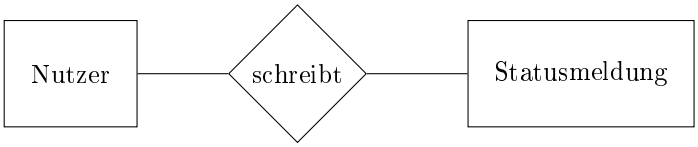

Abbildung 3.1: Entity-Typen und Relationship-Typen in grafischer Darstellung

Bei komplexeren Modellen wird auf die Darstellung der Attribute in der Grafik verzichtet. Diese werden dann meist nur in Ausschnitten des Modells dargestellt, bspw. wenn nur eine Entity dargestellt wird, wie in Abbildung 3.2. Als Darstellungsform wird eine Ellipse verwendet.

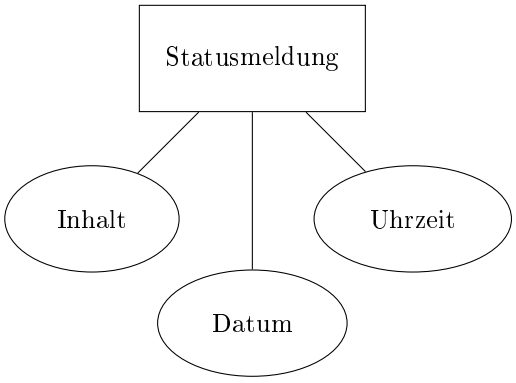

Abbildung 3.2: Entity mit Attributen

Weiterhin relevant sind im ERM noch die Schlüsselattribute und Beziehungseigenschaften. Bei einem Schlüsselattribut handelt es sich um ein Attribut, dass eindeutig ist. Dass heißt in der gesamten Entity

darf es maximal nur einmal vorkommen [SSH13, S. 65f]. Beinhalten die Daten kein Attribut, dass eine eindeutige Zuordnung erlaubt, so wird ein neues Attribut hinzugefügt. Hierbei kann es sich um eine fortlaufende Zahl oder eine zufällige Zeichenkette handeln.
In Bezug auf die Eigenschaften von Beziehungen gibt es im Wesentlichen die Unterscheidung in zwei. Die *Stelligkeit* beschreibt die Anzahl der Entity-Typen in einem Beziehungstyp. Die *Kardinalität* gibt an, wie viele einzelne Entities eines Entity-Typs in eine Beziehung eingehen [SSH13, Abschnitt 3.4].

3.2.3 Relationenmodell

Das Relationmodell wurde 1970 von Codd eingeführt und ist in heutigen System am meisten genutzt [Cod70]. Während das ERM eher im konzeptionellen Entwurf eine Bedeutung hat, wird das Relationenmodell als Modell zur Implementierung genutzt [SSH13].

Struktur

Schemata Im Relationenmodell werden Objekttypen durch Relationschemata dargestellt. Dieses ist eine Menge von Attributen A_1 bis A_n und kann dementsprechend als $R = \{A_1, ..., A_n\}$ geschrieben werden wobei jedes A_i aus der Menge aller Attribute, dem Universum $\mathcal{U}$ stammt [SSH13, S. 88]. Stellt man einen Objekttypen mit seiner Instanz in tabellarischer Form dar, ergibt sich Abbildung 3.3.

A_1	...	A_n
	...	
	...	

Abbildung 3.3: Veranschaulichung des Relationenschemas

Die Instanz eines Objekttyps wird als Relation r bezeichnet und umfasst die konkreten Daten (vgl. mit Abbildung 3.4). Man nennt dies auch r *über* $R = \{A_1, ..., A_n\}$ oder schreibt kurz $r(R)$. Die Daten sind Teil der Wertebereiche der Attribute [SSH13, S. 86ff]. Ein einzelnes Element einer Relation wird Tupel genannt und ist in Abbildung 3.5 veranschaulicht. Bezogen auf das Beispiel entspricht ein Tupel bspw. einer konkreten Statusmeldung.

A_1	...	A_n
	...	
	...	

Abbildung 3.4: Veranschaulichung der Relation

A_1	...	A_n
	...	
	...	

Abbildung 3.5: Veranschaulichung eines Tupels

Die Menge aller Relationschemata $S = \{R_1, ..., R_n\}$ wird als Datenbankschema bezeichnet. Analog bildet die Menge aller Relationen $d = \{r_1, ..., r_n\}$ den Datenbankwert [SSH13, S. 89f].
Bezogen auf das Beispiel ist $S = \{\text{Nutzer}, \text{Statusmeldung}\}$.

Integritätsbedingungen Ein Datenbanksystem muss verschiedene Integritätsbedingungen beobachten und sicherstellen, dass zu jeder Zeit alle Bedingungen erfüllt sind. Die wesentlichen Bedingungen seien hier kurz analog zu [SSH13, S. 90ff] erläutert.

Schlüsselattribut Dieses muss in jeder Relation R enthalten sein. Notwendige Eigenschaft dieses Attributs ist die Eindeutigkeit seiner Werte, d.h. keine zwei Tupel in $r(R)$ dürfen für dies Attribut den gleichen Wert haben.

Lokale Integritätsbedingungen $\mathcal{B}$ Dies sind eine Menge von Abbildungen die eine Relation auf Wahrheitswerte `true` und `false`. Die Bedingung des Schlüsselattributs ist hierbei ein spezielle lokale Integritätsbedingung. Eine weitere Bedingung ist bspw. die Sicherstellung eines Mindestalters eines Nutzers.

Erweitertes Relationenschema Erweitert man ein Relationsschema um die lokalen Integritätsbedingungen, spricht man vom erweiterten Relationsschema $\mathcal{R} = (R, B)$. Dies bedeutet das eine *Relation r über $\mathcal{R}$* ($r(\mathcal{R})$ alle Bedingungen $b \in \mathcal{B}$ erfüllen muss.

Lokal erweitertes Datenbankschema Hierbei handelt es sich um eine Menge von erweiterten Relationenschemata $S = \{\mathcal{R}_1, ..., \mathcal{R}_n\}$.

Fremschlüssel Ein *Fremdschlüssel* in einer Relation $r_1(R_1)$ ist eine Menge von Attributen X als Teilmenge des Relationenschemas R_1, die eine kompatible Attributmenge Y in $r_2(R_2)$ besitzt, welche dort Primärschlüssel ist. Analog ist Y eine Teilmenge von R_2. Kompatibel bedeutet hier, dass die gleiche Anzahl von Attributen vorliegt und die Wertebereich zueinander passen. Die Schreibweise für die Fremdschlüsselbedingung ist

$$X(R_1) \to Y(R_2) \qquad \text{mit} \quad X \subseteq R_1, Y \subseteq R_2$$

Globale Integritätsbedingungen Dies sind Abbildungen einer Datenbank auf Wahrheitswerte. Ein Fremdschlüssel ist eine solche Integritätsbedingung.

Anfragen

Die Relationenalgebra stellt das beliebteste Anfragenmodell für Datenbanken dar und soll hier kurz erläutert werden. Weitere existierende Modelle sind unter anderem das Anfragekalkül [SSH13, Kap. 4.2.3] und Tupelkalkül [SSH13, Kap. 4.2.4].

In der Relationenalgebra wird die mathematische Bedeutung des Wortes *Algebra* umgestetzt. Die Datenbankzustände entsprechen Werten eines Wertebereichs, jede Anfrageoperation kann als Funktion auf diesem Wertebereich betrachtet werden. Komplexe Anfragen werden durch eine Hintereinanderausführung von Operationen umgesetzt[SSH13, S. 95].

In [SSH13, Kap. 4.2.2] werden die folgenden sechs Operationen beschrieben.

- *Projektion*, Attribute ausblenden
- *Selektion*, Tupel heraussuchen
- *Verbund*, Relationen verknüpfen
- *Vereinigung*, Relationen vereinigen
- *Differenz*, Relationen voneinander abziehen

- *Umbenennung*, Attribute umbenennen

Es folgt eine kurze Beschreibung von Projektion, Selektion und Verbund analog zu [SSH13, S. 97ff], da diese für diese Arbeit relevant sind. Die weiteren Operationen finden sich in [SSH13, S. 101ff].

Projektion Bei der Projektion π werden Attribute einer Relation ausgeblendet. Hierzu wird der Operation eine Attributmenge übergeben, welche im Ergebnis der Operation enthalten sein sollen. Die Syntax hierfür ist

$$\pi_{attributmenge}(Relation)$$

Möchte man also bspw. nur die Namen aus einer Nutzerrelation, so lässt sich die als

$$\pi_{\mathsf{Name}}(\mathsf{Nutzer})$$

darstellen.

Selektion Bei der Selektion σ werden anhand einer Bedingung Tupel aus der Relation herausgesucht. Man schreibt

$$\sigma_{Bedingung}(Relation)$$

Diese Bedingung kann mehrere Formen haben:

1. Konstantenselektion mit $Attribut\,\theta\,Konstante$ wobei θ hier einer folgender Vergleichsperatoren ist $=, \neq, \leq, <, >, \geq$ und die $Konstante$ ein bestimmter Wert aus dem Wertebereich des Attributs ist.
2. Attributselektion mit $Attribut_1\,\theta\,Attribut_2$ Für θ gelten die gleichen Bedingungen, allerdings wird das erste Attribut mit einem weiterem Attribut anstatt einer Konstante verglichen.
3. Verknüpfung weiterer Selektionen mit Hilfe der boolschen Operatoren $\wedge, \vee, \neg$.

Im Ergebnis der Operation sind dann nur diejenigen Tupel enthalten, die die Bedingung erfüllen, bspw. kann die Nutzerrelation auf eine bestimmte Altersgruppe reduziert werden.

$$\sigma_{\mathsf{Alter}>12\wedge\mathsf{Alter}<22}(\mathsf{Nutzer})$$

Verbund Der Verbund verwendet den binären Operator $\bowtie$ um zwei Relationen miteinander zu verbinden. Er verknüpft zwei Relationen r_1 und r_2 an gleich benannten Attributen und fügt diejenigen Tupel zusammen, die bei diesen Attributen in beiden Relationen den gleichen Wert haben.
Liegen keine übereinstimmenden Attribute vor, wird aus dem Verbund ein Kreuzprodukt. Hierbei wird dann jedes Tupel aus r_1 mit jedem Tupel aus r_2 verbunden.

In Bezug auf das Beispiel und unter Annahme das die Relationen Nutzer und Statusmeldung über ein gemeinsames Attribut Nutzerid verfügen, lässt sich mit

$$\mathsf{Nutzer} \bowtie \mathsf{Statusmeldung}$$

eine Ergebnistabelle erstellen, die einer Zuordnung der Statusmeldungen zu den jeweiligen Nutzern entspricht.

3.2.4 Datenbankentwurf

Eine relationale Datenbank soll die Daten eines Unternehmens oder Web-Dienstes über mehrere Jahre hinweg speichern und ihre Integrität gewährleisten. Daher kommt dem Datenbankentwurf eine zentrale Bedeutung zu.

Ein verbreitetes Modell für den Datenbankentwurf ist das Phasenmodell nach [SSH13, Kap. 5.2], was mit seinen sieben Phasen im Folgenden beschrieben werden soll. Hierbei müssen zwei Eigenschaften in jeder Phase stets gewährleistet sein:

- *Informationserhalt*: Nach einer Transformation einer Datenbankbeschreibung müssen die selben Informationen gespeichert werden können wie zuvor.
- *Konsistenzerhaltung*: Regeln und Einschränkungen des Eingabedokuments werden im neuen Modell respektiert.

Trotz der recht deutlichen Auftrennung in die verschiedenen Phasen, ist dieses Modell nicht durch einen einmaligen Durchlauf geprägt. Es ist durchaus möglich und oft nötig zu einer vergangenen Phase zurück zu gehen, und das Ergebnis dort zu überarbeiten.

Anforderungsanalyse In dieser Phase werden die Anforderungen an die zukünftige Datenbank gesammelt und ausgewertet.

Konzeptioneller Entwurf Die Datenbank wird mit ihren Anwendungsfunktionen unabhängig von einer späteren Implementierung entworfen.

Verteilungsentwurf Im Falle, dass die Datenbank verteilt umgesetzt werden soll, kann in dieser Phase eine Planung der Verteilung der Daten erfolgen. Dies wird vor der Festlegung bestimmter Systeme durchgeführt, um bei der Umsetzung einzelner Rechnerknoten mehr Freiheit bei der Datenbankmodellwahl zu haben.

Logischer Entwurf Diese Phase ist für die Auswahl des Datenbankmodells vorgesehen. Auf dieses Modell wird das Ergebnis des konzeptionellen Entwurfs abgebildet. Mit Hilfe verschiedener Qualitätskriterien kann eine Optimierung dieser Modelle erfolgen.

Datendefinition Nach dem logischen Entwurf mit der Auswahl der Datenmodelle, müssen diese nun in eine konkrete *Data Definition Language* (DDL) umgesetzt werden.
Diese Umsetzung basiert nun auf einem konkreten DBS. Auch werden hier die festgelegten Benutzersichten aus dem konzeptionellen Entwurf umgesetzt.

Physischer Entwurf Hier erfolgt die Definition der Zugriffsstrukturen mit Hilfe einer *Storage Structure Language* (SSL). Durch eine gute Auswahl der Zugriffsstrukturen können die Leistung eins DBS optimiert werden.

Implementierung und Wartung Diese Phase enthält schlussendlich die Installation des Datenbanksystems basierend auf den vorhergehenden Phasen. Im weiteren Verlauf fallen Wartungsarbeiten, Fehlerbehebung und ANpassung an neue Anforderungen in diese Phase.

3.2.5 Anfragesprache SQL

Structured Query Language (SQL) wurde von der *American National Standards Institute* (ANSI) und *International Standardization Organisation* (ISO) standardisiert und hat sich in den letzten Jahrzehnten zur der Anfragesprache entwickelt, die von allen freien und kommerziellen Systemen unterstützt wird [SSH13, S. 211].
Es gibt drei wesentliche Bestandteile in SQL: Daten- und Schemadefinitionsteil, Anfrageteil, Datenänderungsteil, Definitionsteil für Dateiorganistionsformen und Zugriffspfade, und einen Prozedur- und Funktionsdefinitionsteil [SSH13, S. 211]. An dieser Stelle seien im Folgenden Datendefinitions- und Anfrageteil beschrieben.

Datendefinition

SQL bietet Anweisungen zur Datendefinition auf drei Ebenen:

1. Die *externe Ebene* enthält Sichten, die mit `create view` erstellt und `drop view` gelöscht werden können.
2. Auf der *konzeptuellen Ebene* werden mit den Anweisungen `create table`, `drop table` und `alter table` Relationen, sowie mit `create domain`, `drop domain` und `alter domain` ihre Wertebereiche verwaltet. Ein Beispiel zum Anlegen einer Relation ist in Quelltext 3.1 dargestellt.
3. Die *interne Ebene* ist für Zugriffspfade verantwortlich. Zwar sind die Verwaltungsanweisungen `create index`, `drop index` und `alter index` nicht mehr Teil des SQL-Standards, werden aber dennoch von den meisten Systemen unterstützt [SSH13, S. 212].

Quelltext 3.1: Beispiel zum Anlegen einer Relation mit SQL

```
1 create table Nutzer(
2   NutzerID int not null,
3   Name varchar(40),
4   Alter int)
```

Anfragen

Der Anfrageteil von SQL ist die Umsetzung des Relationenmodells und besteht aus verschiedenen Klauseln. Die wichtigsten von ihnen sind `select`, `from` und `where`. Sie bilden den *SFW*-Block [SSH13, S. 221] wie in Quelltext 3.2 dargestellt.
Darüber hinaus können aber auch weitere Klauseln wie `group by` oder `order by` angehängt werden.

Quelltext 3.2: SFW-Block in SQL

```
1 select projektionsliste
2 from relationenliste
3 where Bedingung
```

select In diesem Block werden die Attribute ausgewählt, die im Ergebnisschema enthalten sein sollen. Dies entspricht der Projektion des Relationenmodells.
Zusätzlich sind hier auch arithmetische Operationen und Aggregatfunktionen wie bspw. `COUNT` oder `MAX` nutzbar.

from Hier werden die zu verwendenden Relationen festgelegt und können bei Bedarf umbenannt werden. Weiterhin kann definiert werden, auf welche Art diese Relationen verbunden werden sollen.

where Bedingungen für die Selektion von Tupeln werden hier angegeben. Darüber hinaus sind auch konkretere Bedingungen für den Verbund von Relationen möglich. Innerhalb der `where`-Klausel können weitere SFW-Blöcke geschachtelt werden.
Quelltext 3.3 zeigt eine Anfrage, die die Namen aller Nutzer einer bestimmten Altersgruppe zurück liefert.

Quelltext 3.3: Beispiel einer SQL-Anfrage

```
1 select Name
2 from Nutzer
3 where Alter >= 18 and Alter <= 21
```

3.3 Objektorientierte Datenbanksysteme

3.3.1 Motivation

Während relationale DBS direkt definiert wurden, unterlagen *Objektorientierte Datenbanksysteme* (ODBS) einer langsameren Entwicklung und wurden näher an objektorientierten Anwendungen heran entwickelt. Hierdurch erhofft man sich eine bessere Verwaltung und Wiederverwertung der Objektstrukturen direkt aus den Anwendungen. Dabei gibt es zwei wesentliche Entwicklungsrichtungen. Eine Kombination beider ist dagegen seltener zu finden [Heu97, S. 24f].

Operationale ODBS Dieser Entwicklungsansatz integriert die Persistenz in objektorientierte Programmiersprachen, erweitert diese also um Datenbankkomponenten. Dadurch können Objekte und Mengen von Objekten gespeichert oder auch Operationen auf ihnen ausgeführt werden. Der Fokus liegt hier auf dem Verhalten der Datenbankobjekte.

Strukturelle ODBS In diesem Ansatz werden herkömmliche DBS mit objektorientierten Konzepten wie Typkonstruktoren und Objektidentitäten angereichert. Dadurch wird der Fokus auf die Struktur von Anwendungsobjekten gelegt.

3.3.2 Datenbankmodelle

Die in [Heu97] Kapitel 6 beschrieben Konzepte objektorientierter Datenmodelle wurden in einer Vielzahl von Modellen umgesetzt, die durchaus verschiedene Ansätze verfolgen. Daher wurde 1991 die *Object Data Management Group* (ODMG) ins Leben gerufen, die versuchte das ODMG-Modell als Standard zu etablieren. Dieses Modell soll hier näher betrachtet werden.

Grundkonzepte

Neben den nachfolgend dargestellten Konzepten *Objekt*, *Literal* und *Schnittstelle* besitzt das ODMG-Modell noch einige weitere Grundkonzepte [Heu97, S. 436f].

- *Typen* sind mehrere Objekte mit gleichen Eigenschaften und Verhalten. Sie sind jedoch keine Klassen.
- Beschrieben wird ein Typ durch eine *Schnittstelle* und mehrere *Implementierungen*
- Eine Schnittstelle und genau eine Implementierung eines Typs werden *Klasse* genannt.
- Eine *Implementierung* eines Typs umfasst die Implementierung der Eigenschaften durch Datenstrukturen oder Methoden, sowie die Implementierung des Verhaltens durch eine Methode für jede Operation.

- Mittels *Typhierarchie* ist die Vererbung von Struktur und Methoden möglich. Das Überschreiben von Methoden und Mehrfachvererbung sind erlaubt.
- Analog gibt es die *Implementierungshierarchie*, die jedoch keine Mehrfachvererbung unterstützt.
- Konkrete Objekte eines Typs werden *Instanzen* genannt.
- Als *Extension* wird die Menge aller Objekte eines Typs bezeichnet. Während ein Objekt nur Instanz eines Typs sein kann, kann es jedoch in mehreren Extensions vorkommen.
- *Schlüssel* sind identifizierbare Eigenschaften eines Objekts.

Objekte

Im ODMG-Modell bestehen Objekte aus *Eigenschaften*, welche die Attribute und Beziehungen umfassen und damit den Zustand eines Objekts bilden. Ist die Eigenschaft eine identifizierbare Eigenschaft, so nennt man diese *Schlüssel*. Auch können Objekte mit *Namen* versehen werden. Sowohl Namen als auch Objektidentifikatoren müssen innerhalb der gesamten Datenbank eindeutig sein.
Weiterhin haben Objekte ein *Verhalten* welches durch Operationen beschrieben wird [Heu97].

Die Erzeugung eines Objekts erfolgt durch einen Konstruktor, welcher in der Schnittstelle definiert wurde. Bei der Erzeugung wird auch festgelegt, ob das Objekt *persistent* (dauerhaft gespeichert und in Datenbankkontrolle) oder *transient* (nicht dauerhaft und in Programmiersprachenkontrolle) ist.

Literale

Der ODMG-Standard umfasst ebenfalls *atomare* und *komplexe Werte*, welche Literale genannt werden.

Atomare Werte Diese Werte umfassen Datentypen, die aus vielen Programmiersprachen bekannt sind.

- `long, short, unsigned long, unsigned long`
- `float, double`
- `boolean`
- `octet`
- `char, string`
- `enum`

Komplexe Werte Dies sind einerseits Datenstrukturen, die auf den atomaren Werten (dargestellt durch `t`) basieren.

- `set<t>`
- `bag<t>`
- `list<t>`
- `array<t>`
- `dictionary<t,w>`

Darüber hinaus können komplexe Werte auch *strukturierte Werte* sein. Hierzu gehören vordefinierte Typen wie `date, time, interval` und `timestamp`, aber auch der Typ `struct`.

Schnittstellen

Zur Beschreibung einer Schnittstelle wird eine *Object Definiton Language* (ODL) genutzt. Eine Schnittstelle umfasst eine `interface`-Klausel, in der Attribute, Beziehungen und Operationen enthalten sind.

In der Schnittstelle werden Attribute mit dem Schlüsselwort `attribute` begonnen, gefolgt von einem Literal oder Typ und dann mit einem bezeichnendem Namen beendet.

Beziehungen beginnen mit dem `relationship` Schlüsselwort. Der weitere Aufbau umfasst einen Komponententyp mit Bezeichner und nach dem Schlüsselwort `inverse` nochmals der selbe Komponententyp gefolgt diesmal von dem Bezeichner für die Rückreferenzierung aus der Zielkomponente. Siehe als Veranschaulichung dazu auch Quelltext 3.4

Quelltext 3.4: Schema einer Beziehungsdefinition

```
1 interface A {
2   relationship B B_Name inverse B::A_Name;
3 }
4
5 interface B {
6   relationship A A_Name inverse A::B_Name;
7 }
```

Ein beispielhafte Schnittstelle ist abschließend in Quelltext 3.5 dargestellt.

Quelltext 3.5: Beispiel einer Schnittstellendefinition

```
1 interface Nutzer {
2   attribute long NutzerID;
3   attribute string Name;
4   attribute short Alter;
5
6   relationship set<Statusmeldung> Verfasst inverse Statusmeldung::
        Verfasser;
7 }
```

3.3.3 Anfragesprachen

Es gibt eine Vielzahl an Anfragesprachen für ODBS, wie in [Heu97, S. 489ff] beschrieben. Die ODMG hat eine *Object Query Language* (OQL) eingeführt, um auch hier einen Standard zu definieren.

Die OQL basiert auf den Konzepten von SQL, erweitert diese aber um weitere Funktionen. So können in Anfragen auch komplexe Werte, Objektidentitäten, Pfadausdrücke, Methoden und auch das Überschreiben von Methoden ausgenutzt werden [Heu97, S. 449].

Einfache Anfragen sind bereits mit den Namen von Extensions oder dem Aufruf von Methoden eines konkreten Objektes möglich. So sind bspw. `Nutzers` (Extension der Objekte von Nutzer) und `Meier.Alter` (Meier ist hierbei der Objektname eines Nutzerobjekts, Alter ein Attribut) bereits gültige Anfragen.

Mit Hilfe des SFW-Blocks lassen sich Mengen von Objekten filtern, sie werden daher auch *objekterhaltende Anfragen* genannt. Diese Anfragen erinnern an SQL, wie im Quelltext 3.6 dargestellt. Ausgewählt werden die Objekte aus der Extension `Nutzers`.

Quelltext 3.6: Beispiel einer Anfrage mit Filter im ODGM

```
1 select n
2 from Nutzers
3 where n.Alter = 21
```

Darüber hinaus gibt es dann auch *objekterzeugende Operationen* zur Konstruktion neuer Objekte.

Quelltext 3.7: Beispiel einer Objekterzeugung im ODGM

```
1 Nutzer(
2   NutzerID: 1337,
3   Name: "Meier",
4   Alter: 42, ...)
```

3.4 NoSQL-Datenbanksysteme

3.4.1 Motivation

In den letzten Jahren hat sich der Anwendungsbereich insbesondere im Web geändert. Die zu bewältigenden Datenmengen sind so rasant angestiegen, dass relationale Systeme mit ihrer vertikalen Skalierung an ihre Grenzen kommen. Auch können sich Anforderungen und Schemadefinitonen schnell ändern. Daher wurden NoSQL-Datenbanksysteme entwickelt, die speziell auf horizontale Skalierung auf vielen Rechnerknoten, sowie auf schemafreie oder schemaarme Daten zugeschnitten sind.

Die Definition des Begriffs *NoSQL-Datenbanksystem* ist durch die schnelle und noch junge Entwicklung in diesem Bereich nur wenig ausgeprägt. Ein mögliche Definition liefert [Kud15, Kap. 12.1]. Demnach zeichnet sich ein NoSQL-Datenbanksystem durch die folgenden Punkte aus:

- nicht-relationales Modell
- schemafrei oder schwache Schemarestriktionen
- horizontale Skalierung
- häufig ein anderes Konsistenzsystem als ACID (vlg. [SSH11, Kap. 9.2])

3.4.2 Ausprägungen

Das Akronym NoSQL (**N**ot **o**nly **SQL**) impliziert bereits, dass es mehrere Ansätze in diesem Bereich gibt. Die drei wichtigsten Datenmodelle nach [Kud15] werden nun kurz beschrieben.

Key-Value-Datenbanksysteme

Bei dieser Art von System werden die Daten als Paar aus einem Schlüssel (key) und einem Wert (value) gespeichert. Während der Wert beliebige Daten und Datenformen annehmen kann, ist der Schlüssel ein einfacher Text- oder Zahlenindikatorder. Dieser Schlüssel ist entweder auf der gesamten oder in einem bestimmten Bereich eindeutig.

Das System hat meist keinen Einblick in die Daten selbst. Somit sind Anfragen an das System nur über den Schlüssel möglich. Selektionen basierend auf Werten oder Joins von Key-Value-Paaren sind ohne eine weiteres, separates System nicht möglich.

Diese Systeme sind demnach auf eine effiziente Speicherung der Daten ausgelegt. Sie finden Anwendung in Bereichen, in denen nur über den Schlüssel Selektionen auf dem Datenbestand durchgeführt werden. Eine Datenmanipulation innerhalb des Dokumentes ist nicht möglich und erfordert ein Überschreiben des gesamten Dokuments.

Dokumentorientierte Datenbanksysteme

Ähnlich zu Key-Value-Datenbanksystemen speichern auch dokumentorientierte Datenbanksystem Paare von Schlüsseln und Werten. Hier sind diese Werte allerdings Dokumente eines bestimmten Datenformats wie bspw. JSON oder XML. Eine Spezifizierung der Struktur der Dokumente ist vorab nicht nötig, allerdings empfiehlt es sich für effizientere Anfragen, sehr unterschiedliche Dokumente auch in verschiedenen Bereichen zu speichern, sogenannten Kollektionen.
Der Wert einer Eigenschaft eines Dokuments, kann ebenfalls wieder in Dokument enthalten, sodass sich hierarchische Strukturen bilden lassen. Im Beispiel können Beispiel die Statusmeldungen eines Nutzers als eine Eigenschaft in dessen Profil-Dokument gespeichert werden.

Bei dokumentorientierten Datenbanksystemen sind auch Anfragen anhand konkreter Eigenschaften in den Dokumenten möglich. Damit dies effizient möglich ist, werden alle Eigenschaften (oder eine Auswahl des Nutzers) automatisch indexiert. Join-Operation werden im Allgemeinen jedoch ebenfalls nicht unterstützt. Dafür können jedoch die Daten des Dokumentes auch im Nachhinein noch manipuliert werden.

Column-Family-Datenbanksysteme

Column-Family-Datenbanksysteme ähneln zunächst dem relationalen Modell. Datensätze werden als Zeilen in einer Tabelle gespeichert, in der die Attribute wiederum die Spalten darstellen. Spalten werden dann zu Spaltengruppen zusammengefasst.

Unterschiede gibt es jedoch beim Schema und dem Speichermodell. So müssen die Spaltenfamilien bei Erstellung einer Tabelle definiert sein, neue Spalten können jedoch dynamisch hinzugefügt werden. Hierzu wird dann einfach ein neuer Datensatz eingefügt, der diese neue Spalte nutzt.
Das Speichermodell sieht für Column-Family-Datenbanksysteme keine zeilenweise sondern eine spaltenweise Speicherung der Daten vor. Dies ist essentiell um Spalten dynamisch hinzufügen zu können.
Darüber hinaus werden Änderungen eines Datensatzes eine neue Version mit einem aktuellen Zeitstempel.

Anfragen an ein solches System sind sowohl mit dem Schlüssel, als auch einzelnen Werten oder sogar der Version durch den Zeitstempel möglich. Allerdings ist auch hier ein Join einzelner Einträge nicht vorgesehen.
Dank des Speichermodells eigenen sich Column-Family-Systeme gut für Bereiche mit spaltenweise Analyse, bspw. Logging- und Tracking-Anwendungen.

3.4.3 Datenmodellierung

Um die horizontale Skalierbarkeit ideal auszunutzen, ist es für NoSQL-Datenbanksysteme wichtig, zusammengehörende Daten auch zusammen zu speichern. Auch sollen aufwendige Join-Operation vermieden werden. Dadurch bieten sich hierarchische Strukturen am ehesten an. Kernaufgabe der Datenmodellierung ist also Zerlegung der Daten in sinnvolle Einheiten, sogenannte Aggregate [Kud15, S. 378f]. Das heißt hier sind Entscheidungen nötig, welche Daten eingebettet werden, und welche referenziert.

Eine Einbettung bietet den Vorteil, dass alle zusammengehörigen Daten auf einmal geladen werden können. Mögliche Redundanz wird hierbei in Kauf genommen. Jedoch sollte geprüft werden, ob wirklich

immer alle Daten benötigt werden oder auch eine Referenzierung genügt. Bei der Einbettung großer Datenmengen kann es zudem zu Peformance-Problemen beim Laden kommen.

Key-Value-Datenbanken

Da die Daten nach außen abgeschlossen sind, benötigen diese Datenbanksysteme keine Datenmodellierung. Es liegt in der Verantwortung der aufrufenden Anwendung diese Daten zu verwalten.

Dokumentorientierte Datenbanken

Durch die Verwendung von Dokumentformaten wie JSON und XML werden Hierarchien nativ bereits unterstützt. Die verbleibende Fragestellung ist die Aufteilung der Daten, wie eingangs erläutert.

Column-Family-Datenbanken

Eingebettete Speicherung ist bei Column-Family-Datenbanken nicht ohne Weiteres möglich. Wie auch im relationalen Modell sind mehrwertige Attribute nicht vorgesehen. Es gibt jedoch zwei Alternativen zur Realisierung der eingebetteten Speicherung.

Fortlaufend nummerierte Spalten Mehrere Werte eines Attributs, bspw. Keywords einer Statusnachricht, werden aufgetrennt in Spalten gespeichert. Die Spaltennamen unterscheiden sich dabei nur in einer fortlaufenden Nummer. Trennen und Zusammensetzten dieser Spalten obliegt der jeweiligen Anwendung, nicht der Datenbank.

Zeitstempel Zusätzliche Werte für ein Attribut werden in einer separaten Zeile gespeichert und einem neuen Zeitstempel versehen. Dadurch können Zeitstempel dann allerdings nur noch zur Versionsreferenzierung genutzt werden.

Kapitel 4

State of the Art der Abbildungsstrategien

4.1 XML-Abbildungsstrategien

Bei den XML-Abbildungsstrategien kann zwischen *schema-obvious* und *schema-aware* unterschieden werden [AYDF04]. Während in schema-obvious ein Allgemeiner Baumansatz verfolgt wird, wie unter anderem von [FK99] beschrieben (Kap. 4.1.1), so verfolgt schema-aware die Einbettungsstrategie, analog zu [STZ+99] (Kap. 4.1.2).

4.1.1 Allgemeiner Baumansatz

XML kann grundsätzlich als Graph betrachtet werden [FK99]. Hierbei bilden die Elemente des XML-Dokumentes die Knoten des Graphen. Die Daten des Dokuments sind in den Blattknoten enthalten. Wesentlicher Teil der für die Speicherung berücksichtigt wird, sind neben Daten des Dokumentes, die Kanten des Graphen, denn diese reflektieren die Eltern-Kind-Hierachie aus dem XML-Dokument

Kantenspeicherung

Für die Speicherung der Kanten des Dokument-Graphen, sieht [FK99] drei verschiedene Variante vor: *Edge Table*, *Binary Table* und *Universal Table*.

Edge Table In dieser Variante sind Attribute für Elternelement (`source`), Kindelement (`target`), Namen des Elementes oder auch XML-Attributs (`name`), sowie eine Ordnungsnummer (`order`) um bei mehreren Kind-Elementen die Reihenfolge bewahren zu können. Hinzu kommen Attribute für die Datenspeicherung, wie im nächsten Abschnitt erläutert.
Während `source` eine einfache, fortlaufende Nummerieung als ID haben kann, so ist es auch möglich, die IDs aller Elternelemnte aneinanderzureihen (Dewey Decimal Classification) [AYDF04]. Eine weitere Mäglichkeit ist eine Intervallkodierung der Form `{start,end}` wobei `start` und `end` hierbei die Anazahl der Level vom root-Element bzw. Blattknoten aus gezählt sein können [AYDF04].

Die Edge Table speichert alle vorhandenen Beziehungen zwischen zwei Elementen, unabhängig von der Art des Eltern- oder Kind-Elements. Als Key werden `source` und `order` gemeinsam genutzt.

Binary Table Die Binary Table verfolgt den selben Ansatz wie die Edge Table, benötigt also die gleichen Attribute. Allerdings wird die einzelne Tabelle in mehrere Tabellen aufgespalten: Für jeden

source	*order*	*name*	*target*	*value*
1	1	name	2	null
1	2	age	null	42
2	1	full_name	null	Max Müller

Abbildung 4.1: Beispiel einer Edge Table

Wert von `name`, also jedes mögliche XML-Element, wird eine Tabelle angelegt [FK99]. Auf das Attribut `name` kann in den Tabellen also verzichtet werden.

Tabelle `name`

source	*order*	*target*	*value*
1	1	2	null

Tabelle `age`

source	*order*	*target*	*value*
1	2	null	*values*

Tabelle `full_name`

source	*order*	*target*	*value*
2	1	null	*values*

Abbildung 4.2: Beispiel einer Binary Table Struktur

Universal Table Die Universal Table erzeugt wiederum nur eine Tabelle für alle Kanten [FK99]. Hierbei wird allerdings nicht nur eine Kante zwischen zwei Knoten, sondern der gesamte Zweig des Baumes in einer Zeile gespeichert. Dies beinhaltet also ein Element, das Kind-Element, so wie alle folgenden Generationen von Enkelelementen. Gibt es auf einer Ebene mehrere Elemente, so bildet diese separate Zweige. Alle vorhergehenden Elemente werden für jeden Zweig erneut in der Tabelle vermerkt. Es ensteht eine Denormalisierung und hohe Datenredundanz [FK99].

Neben dem Attribut `source` muss eine Universal Table die Attribute `target`, `order` und Attribute zur Datenspeicherung für jedes Kind-Element haben [FK99]. Da nicht jeder Zweig des Dokumentes gleich ist, speichert eine Universal Table viele `null` Werte.

In der Abbildung 4.3 sind die Elementnamen wie folgt abgekürzt: a (age), f (full_name) und n (name).

source	$order_n$	$target_n$	$value_n$	$order_f$	$target_f$	$value_f$	$order_a$	$target_a$	$value_a$
1	1	2	null	1	null	Max Müller	null	null	null
1	null	null	null	null	null	null	2	null	42

Abbildung 4.3: Beispiel einer Universal Table

Datenspeicherung

Daten können auf zwei Arten gespeichert werden. Einerseits in separaten Wertetabellen, andererseits aber auch eingebettet in der oder den Tabellen für die Kanten. [FK99]

Im Falle der Einbettung in eine Tabelle wird für jeden möglichen Datentyp eine einzelne Spalte angelegt. Dies ergibt Spalten für boolsche Werte, Zeichenketten und numerische Werte. Sofern eine Kante Daten enthält, werden diese in die entsprechende Spalte geschrieben, alle weiteren Spalten entahlten Nullwerte.

source	*order*	*name*	*target*	$value_{bool}$	$value_{number}$	$value_{string}$
1	2	age	null	null	42	null

Abbildung 4.4: Beispiel zur Speicherung innerhalb einer Edge Table

Bei der Speicherung in separaten Wertetabellen, wird für jeden Datentyp eine Tabelle angelegt. Die Kantentabelle wird um die Spalten `flag` und `value` erweitert, die einerseits den Datentypen zur Tabellenauswahl, andererseits eine Referenz auf den Eintrag in jener Tabelle sind.

Kantentabelle

source	*order*	*name*	*target*	*flag*	$value_{ID}$
1	2	age	null	number	1

Tabelle `number`

ID	*value*
1	42

Abbildung 4.5: Beispiel zur Speicherung in einer separaten Tabelle

4.1.2 Einbettungsstrategie

Da Strategien wie der Baumansatz eine hohe Fragmentierung der Daten und viele Joins bei Anfragen verursachen [AYDF04], sind Einbettungsstrategien entwickelt worden. Der wesentliche Ansatz hierbei ist, dass jedes Element als Relation betrachtet wird. Ausgenommen sind kinderlose Elemente, diese bilden zusammen mit XML-Attributen die Attribute einer Relation.

Bei der Einbettung wird versucht, überflüssige Relationen zu vermeiden, in dem diese in andere Relationen integriert werden. Hierbei gibt es drei wesentliche Ansätze [STZ+99], die im Folgenden mit Hilfe des Beispiels Quelltext 4.1 beschrieben werden.

Quelltext 4.1: Beispielhaftes XML-Dokument für Einbettungsstrategie

```
1  <?xml version="1.0"?>
2  <author>
3    <name>
4      <first>George</first>
5      <last>Martin</last>
6    </name>
7    <age>67</age>
8  </author>
```

In den Betrachtungen von [STZ+99] wird die Reihenfolge der Elemente nciht weiter berücksichtigt. Es wird doch erwähnt, dass die Reihenfolge ohne Weiteres als ein zusätzliches Attribut in den Relationen umgesetzt werden könnte.

Basic Inline

Diese Strategie erzeugt für jedes Element eine separate Relation. Darüber hinaus werden aber so viele Elemente wie möglich in eine Relation eingebettet. Hierbei werden alle Kindelemente eines Elementes ein-

gebettet, mit Ausnahme von Elementen die mehrfach als Kindelement auf einer Ebene auftreten können, sowie Elementen die Rekursion verursachen können [STZ+99].

Quelltext 4.2: Relationen nach Basic Inline

```
1  author(first, last, age)
2  first(id, value)
3  last(id, value)
4  age(id, value)
```

Shared Inline

Shared Inline verfeinert Basic Inline bei der Auswahl der Elmente für separate Relationen. Es wird die Anzahl der Referenzierungen betrachtet. Ist dieser Referenzierungsgrad gleich 1, das Element wird also nur von einem anderen aufgerufen, so kann das Element eingebettet werden. Bei mehr Referenzierungen wird eine separate Relation erstellt, da die Informationen des Elements ggf. von mehreren Elternelementen geteilt werden [STZ+99].
Mehrfache Elemente werden wie beim Basic Inline behandelt. Im Falle einer Rekursion, bei der alle Elemente einen Referenzierungsgrad von 1 haben, muss eines ebenfalls eine separate erhalten [STZ+99].

Quelltext 4.3: Relationen nach Shared Inline

```
1  author(first, last, age)
```

Hybrid Inline

Der letzte Ansatz enspricht im Wesentlichen dem Shared Inline Prinzip. Allerdings können nun auch Elemente eingebettet werden, die mehr als einmal referenziert wurden. Ausnahme sind wiederum Elemente, die in Rekursionen enthalten sind oder mehrfach als Kindelement auftreten können.

Quelltext 4.4: Relationen nach Hybrid Inline

```
1  author(first, last, age)
```

4.1.3 Weitergehende Berücksichtigungen

Neben der Art der Abbildung und Speicherung als solches, gibt es weitere Aspekte, die in disem Prozess berücksichtigt werden können.

So kann einerseits abgewogen werden, inwieweit Optimierungen an einer Abbildungsstrategie für einen konkreten Fall sinnvoll sind. Diese können bspw. auf Anfragen basieren, die hauptsächlich zu erwarten sind [AYDF04]. Ein weiterer Fall wäre es, bestimmte Relationen vorzugeben, die im Ergebnis der Abbildung vorhanden sein sollen.

Weiterhin kann abgewogen werden, ob ein konkreter Abilldungsprozess automatisiert oder manuell stattfinden soll. Zwar bietet ein manueller Prozess viel Flexibilität und Freiheit für die Anpassung eines Schemas, allerdings erfordert dies weitreichende Kenntnisse des Anwenders und kann bei hinreichend großen Schemas unübersichtlich werden [AYDF04].

4.2 JSON-Abbildungsstrategien

4.2.1 Abbildung auf flaches Relationenschema

Eine Möglichkeit JSON auf ein relationales Schema abzubilden ist die Verwendung eines flachen Schemas, dass sich nur durch eine einzelne oder sehr wenige Relationen auszeichnet. **Argo** ist ein solches System. Die Abbildungen erfolgen hier auf eine Relation mittels Argo/1, oder auf drei Relationen mittels Argo/3. [CLP13]

In Argo/1 werden in der Relation insgesamt 5 Attribute angelegt: Eines für die Dokument-ID, eines für den Schlüssel der JSON-Eigenschaft sowie ein Attribut je möglichem Datentyp (String, Number und Boolean)[CLP13].
Argo/3 zeichnet sich dadurch aus, dass die Tabelle aus Argo/1 nach Datentypen aufgeteilt wird [CLP13]. Neben den Attributen für ID und Schlüssel enthält jede Tabelle eine Spalte für den Datenwert.

In beiden Varianten werden Verschachtelungen von Arrays und Objekten direkt auf den Schlüssel der JSON-Eigentschaft abgebildet [CLP13]. Ein Objekt wie in Quelltext 4.5 wird dabei den Schlüssel `user.name` abgebildet. Eine Relation nach Argo/3 könnte dann wie in Tabelle 4.1 dargestellt aussehen.

Quelltext 4.5: Verschachteltes JSON-Objekt

```
1 {
2   "user" : {
3     "name" : "Felix"
4   }
5 }
```

ID	*key*	*value*
1	`user.name`	`Felix`

Tabelle 4.1: Relation nach Argo/3 für das Beispeil aus Quelltext 4.5

Darüber hinaus beinhaltet Argo auch eine an SQL angelehnte Anfragesprache Argo/SQL [CLP13]. Mit dieser ist es möglich relationale `SELECT` Anfragen zu verfassen, die JSON-Strukturen als Resultat zurükliefern. Alternativ können JSON-Objekte mit `INSERT INTO` in Kollektionen eingefügt werden. Das `DELETE` Anfrage wird ebenfalls unterstützt.

Behandlung von Arrays

Arrays als eine Form der Verschachtlung werden ebenfalls im Schlüssel integriert. Hierzu wird auch die Ordnungsnummer des Elements erfasst. Ein Schlüssel kann wie folgt aussehen: `posts[2].author.name`

4.2.2 Abbildung auf verschachteltes Relationenschema

Alternativ zur vorgestellten flachen Abbildung, lassen sich JSON-Daten analog zu den XML-Strategien in ein verschachteltes Schema abbilden. Dieser Ansatz unter anderem von [DA16] verfolgt. In diesem wird ein JSON-Dokument zunächst in eine Universaltabelle abgebilldet. Im Gegensatz zu Argo werden hier allerdings Attribute für jeden Datenwert angelegt. Hieran schließen sich drei Phasen an.

Die erste Phase identifiziert schwache Abhängigkeiten zwischen Attributen. Es wird vorrausgesetzt, dass diese Abhängigkeit nur für die meisten, nicht aber zwingend alle Dateneinträge gilt. Nach Feststellung

dieser Abhängikeiten werden die Attribute gruppiert und in einen Attributbaum übertragen. Die Universaltabelle wird in einige kleinere Tabellen aufgeteilt [DA16].

In der folgenden zweiten Phase werden Gruppen von Attributen gesucht, welche semantisch äquivalent sind, jedoch an unterschiedlichen Orten vorhanden sind oder unterschiedliche Namen habe. Hierzu werden im Attributbaum gleiche Unterbäume gesucht [DA16].

Die abschließende Phase drei erstellt das finale relationale Schema und verschmilzt Tabellen auf Basis der Ergebnisse aus Phase zwei [DA16].

Behandlung von Arrays

Im Verfahren von [DA16] werden Arrays nur bedingt behandelt. Ein Array mit einfachen Werten wird intern als String betrachtet und die Verwaltung des Arrays später der Datenbank überlassen.
Beinhaltet ein Array wiederum Objekte, wird dies als neue Problemstellung für die Schemageneration aufgefasst [DA16]. Der drei-phasige Algorithmus kann also unabhängig auf das Array angewandt werden. Hierbei referenzieren dann die Unterobjekte das Elternelemnt im Falle einer 1:n-Beziehung. Im Falle einer n:m-Beziehung wird eine weitere Hilfstruktur zum Verknüpfen nötig.

Eine Betrachtung der Reihenfolge von Elementen des Arrays oder das Behandeln unterschiedlicher Datentypen erfolgt in [DA16] alelrdings nicht.

Kapitel 5

JSON-Schema Extraktion aus NoSQL-Daten

Um Daten aus einem NoSQL-Datenbestand in ein relationales DBS transformieren zu können, ist es nötig ein Schema für diese Daten zu erstellen. Da dies manuell bei großen Beständen kaum extrahierbar ist, wird im Folgenden ein automatisiertes Verfahren beschrieben.

5.1 Schemastruktur

Ein extrahiertes Schema entspricht nur einer Momentaufnahme des Datenbestandes und kann durchaus Aktualisierungen unterliegen. Auch dient das Schema als Grundlage für die Erstellung neuer Daten. Basierend darauf und auf den Schema-Vorschlägen von [KSSR15] hat [Lan16] die folgenden Angaben als Bestandteile eines extrahierten Schema herausgestellt:

- Name des Elements
- Datentyp(en) des Elements
- Absolute und relative Häufigkeit des Auftretens
- Benötigte sowie optionale Kindelemente
- Hierarchische Elemente
- Reihenfolge der Elemente in Arrays

Diese Informationen wurden auch Schlüsselwörten zugeordnet (siehe Tabelle 5.1), die später in der Schemasprache genutzt werden.

5.1.1 Internes Schema

Die oben beschriebenen Bestandteile reichen jedoch nicht aus, damit ein Schema fortlaufend aktualisiert werden kann. Es fehlen eindeutige Schlüssel für die Elemente und Identifikatoren für die analysierten Dokumente. Daher wird das interne Schema eingeführt, auf dem die Algorithmen arbeiten. Es wird auch als *vollständiges Schema* bezeichnet [Lan16].

Dieses Schema wird intern allerdings nicht mehr als ein JSON-Schema dargestellt, sondern direkt als Objekte und Strukturen in der Sprache der Implementierung. Die Hierarchien aus JSON-Dokumenten

Schlüsselwort	Information
`name`	Name des Elements
`type`	Datentyp(en) des Elements
`description`	Abosulte und relative Häufigkeit, sowie bei Arrays die Elementreihenfolge
`properties`	Kindelemente bei JSON-Objekten
`items`	Kindelemente bei JSON-Arrays
`requiered`	Pflichtelemente
`anyOf`	Auflistung der Array-Elemente, falls ein Array leer ist oder verschiedene Datentypen als Kinder hat

Tabelle 5.1: Zuordnung der Informationen zu Schlüsselworten

werden über Baumstrukturen realisiert [KSSR15]. Ein Knoten dieses Baumes enthält hierbei die folgenden Informationen [Lan16, S. 25]:

- Name des Elements
- Pfad des Elements im Baum
- Ebene des Elements
- Datentyp(en) des Elements und Häufigkeit jedes Typs
- Dokument-IDs der Dokumente mit diesem Element und Häufigkeit der Dokument-ID
- Position, Datentyp, Minima und Maxima von Arrayelementen

5.1.2 Externes Schema

Das externe Schema wird auch als *vereinfachtes Schema* bezeichnet, da die Ausgabe der soeben eingeführten Identifikatoren nicht sinnvoll ist. Das externe Schema soll darüber hinaus den Anforderungen des *JSON-Schemas* [IET13] gerecht werden. Hierbei gibt es drei verschiedene Arten von JSON-Dokumenten, die auftreten können.

JSON-Primitive

Diese Elemente kommen in den Blattknoten des Schemas vor und enthalten die tatsächlich gespeicherten Daten eines Dokuments. Durch die Extraktion können Name und Datentyp des Elements gewonnen werden, zudem werden die Häufigkeiten gespeichert. Quelltext 5.1 zeigt eine passende Darstellung.

Quelltext 5.1: Darstellung eines JSON-Primitives

```
"element_name" : {
  "type" : "Primitiver JSON Datentyp",
  "description" : "absolute und relative Haeufigkeit"
}
```

JSON-Objekte

Ein JSON-Objekt erweitert die vorherige Darstellung um eine Aufzählung der Kindelemente (im Attribut `properties`) sowie einer Liste von Namen von Pflichtfeldern (in `required`), wie Quelltext 5.2 zu entnehmen ist. Die Reihenfolge der Kindeselemente ist bei JSON-Objekten nicht von Bedeutung.

Quelltext 5.2: Darstellung eines JSON-Objekts

```
"objekt_name" : {
  "type" : "JSON Object",
  "description" : "absolute und relative Haeufigkeit",
  "properties" : {
    "kind_element_name" : { JSON Dokument des Kindes },
    ...
  },
  "requiered" : [
    "Namen der Pflichtelemente"
  ]
}
```

JSON-Arrays

Im Gegensatz zu JSON-Objekten ist bei einem JSON-Array die Reihenfolge entscheidend. Daher benötigen wir weitere Eigenschaften in der Darstellung. So erhält die `description` ein zusätzliches Attribut `array order`, in dem die Reihenfolge der Elemente festgehalten wird. Anstelle von `properties` verwenden wir hier das Attribut `items`, welches ein JSON-Dokument des Kindelements enthält, wie in Quelltext 5.3 dargestellt.

Quelltext 5.3: Darstellung eines JSON-Arrays bei gleichem Typ für Kindelemente

```
"array_name" : {
  "type" : "JSON Array",
  "description" : {
    "occurance" : { "absolute und relative Haeufigkeit" },
    "array order" : { "Position, Datentyp, Minima, Maxima" }
  },
  "items" : {
    JSON Dokument des Kindes
  }
}
```

Es ist möglich, dass innerhalb eines Arrays nicht alle Elemente des gleichen Datentyps sind. In diesem Fall wird `array order` um weitere Einträge erweitert. Die Auflistung der Kindelemente in `items` wird in ein Attribut `anyOf` geschaltet. Damit ist dann eine erfolgreiche Validierung möglich und es unterstützt auch leere Arrays. In Quelltext 5.4 ein Schema für ein Array mit zwei Datentypen als Kindelemente angegeben.

Quelltext 5.4: Darstellung eines JSON-Arrays bei unterschiedlichen Typen für Kindelemente

```
"array_name" : {
  "type" : "JSON Array",
  "description" : {
    "occurance" : { "absolute und relative Haeufigkeit" },
    "array order" : {
      "Position, Datentyp, Minima, Maxima",
      "Position, Datentyp, Minima, Maxima"
    }
  },
```

```
10   "items" : {
11     "anyOf" : [
12       JSON Dokument des Kindes,
13       JSON Dokument des Kindes
14     ]
15   }
16 }
```

5.2 Inkrementelle Extraktion

5.2.1 Extraktion

Die Extraktion ist die initiale Erstellung eines Schemas. Bei einem Datenbestand von einem einzelnen Dokument, braucht dieses lediglich durchlaufen und die Schema-Informationen extrahiert werden. Bei der Erweiterung des Bestandes um mehr Dokumenten werden diese auf die gleiche Weise durchlaufen, jedoch muss hier auf bereits aufgenommene Elemente geprüft und diese dann gegebenenfalls aktualisiert werden. So werden unter anderem Elemente, die nicht in allen Dokumenten enthalten sind, als optional deklariert [Lan16, S. 32].

Beim Durchlauf eines Dokuments erfolgt für jedes Element ein dreistufiger Prozess.

1. Es erfolgt ein Test, ob dieses Element bereits in das Schema aufgenommen wurde. Ist dies nicht der Fall, wird es mit Elementname, Pfad, Datentyp und aktueller Dokument-ID gespeichert. Andernfalls brauchen weder Name noch Pfad erneut gespeichert werden.
2. Handelt es sich um einen neuen Datentyp, so wird dieser in einer Hashmap abgelegt und mit einem Zähler versehen. Ist der Datentyp bereits vorhanden, so wird sein Zähler inkrementiert.
3. Analog zum vorherigen Schritt werden die Dokument-IDs behandelt. Neue werden in eine Hashmap aufgenommen, wieder auftretende vorkommen inkrementieren den Zähler.

Das Ergebnis dieser Extraktion ist das interne Schema, das nun weiter verarbeitet oder ausgegeben werden kann.

5.2.2 Aktualisierung

Eine Aktualisierung kann einerseits durch geänderte Dokumente, andererseits auch durch Update-Logs ausgelöst werden. Sind die Update-Informationen, wie im Folgenden beschrieben, ermittelt, werden sie mit dem Schema der Extraktion zusammengeführt.

Aktualisierung durch geänderte Dokumente

Liegen geänderte Dokumente vor, gibt es nach [Lan16] zwei Ansätze zur Aktualisierung.

Im ersten Ansatz werden diese Dokumente zunächst wie bei der Extraktion durchlaufen und behandelt. Für jedes Element werden die gleichen Schritte durchgeführt, ist eine Dokument-ID mit dem selben Datentyp bereits vorhanden, wurde keine Änderung vorgenommen. In diesem Schritt werden neue und geänderte Elemente gefunden.
Der zweite Schritt durchläuft die Elemente des internen Schemas, welche nicht im geänderten Dokument vorhanden waren. Hat eines dieser Elemente die aktuelle Dokument-ID, so wird diese gelöscht. Bei leerer

ID-Liste wird das Element entfernt. Dieser Schritt erkennt also gelöschte Elemente in geänderten Dokumenten.

Um diese vergleichsintensive Variante zu vermeiden, wurde eine zweite Variante der Aktualisierung eingeführt. Hierbei wird im ersten Schritt die aktuelle Dokument-ID aus dem internen Schema gelöscht. Im zweiten Schritt kann das Dokument wie ein unbekanntes Dokument in das Schema aufgenommen werden. Es erfolgen dann die gleichen Schritte wie bei der Extraktion.

Beide Varianten erfassen jedoch keine Dokumente die vollständig gelöscht wurden. Um dies zu umgehen müsste ein leeres Dokument mit lediglich der Dokument-ID in die geänderten Dokumente aufgenommen werden [Lan16, S. 35].

Aktualisierung durch ein Update-Log

Ein Update-Log besteht aus einer Liste von Aktualisierungsschritten. Aus jedem dieser Schritte wird ein Update-Objekt gebildet, welches die folgenden Informationen beinhaltet:

- Elementname
- Pfad im Dokument
- Datentyp vor der Änderung
- Datentyp nach der Änderung
- Dokument-ID
- Art der Aktualisierung

Bei der Übernahme der Update-Objekte werden als gelöscht markierte Dokument-IDs aus dem Schema entfernt und neue Knoten bei Bedarf hinzugefügt. Änderungen von Werten müssen den Datentyp kontrollieren und ggf. anpassen [Lan16, S. 35].

Limitierungen

Die Qualität der Aktualisierung durch die Verfahren hängt unmittelbar von den Update-Informationen ab. Fehlen Dokument-ID, Pfadangabe oder der Name, so kann nicht gewährleistet bleiben, dass das Schema nach dem Update korrekt ist.

Auch fehlende Datentypen können problematisch sein. Zwar kann verhindert werden, dass ein inkorrektes Schema entsteht, jedoch führt dies zu einer Generalisierung des Schemas. Diesem kann man durch (regelmäßiges) Durchlaufen des Datenbestandes entgegenwirken, oder aber auch den Datentyp als identifizierendes Merkmal eines Schema-Knotens betrachten. Letzteres führt allerdings zu einer steigenden Anzahl an Knoten.

5.2.3 Metriken

Ein vereinfachtes Extraktionsverfahren ermöglicht die Herausstellung von einigen Metriken, die nach [Lan16] als relevant erachtet werden. Dies sind *Ausreißerdokumente*, *potentiell gleiche Knoten*, sowie *absolute und relative Häufigkeiten*. In dem vereinfachten Verfahren wird unter anderem auf der Speicherung der Elementidentifikatoren gespeichert, sodass der Speicherbedarf erheblich reduziert wird.

Ausreißerdokumente

Ausreißerdokumente sind in kleinen Datenbeständen schwierig zu ermitteln. Bei großen Datenbeständen mit einer gewissen Homogenität lassen sich diese Dokumente leichter erkennen. Sie unterscheiden sich durch die Struktur einzelner Elemente oder aber der Struktur des gesamten Dokuments.
Durch Zählen der Vorkommen der Elemente, lassen sich diese Ausreißer erkennen. Erwartet wird ein Ein- oder Vielfaches der Dokumentenanzahl. Ist dies nicht der Fall, können gezielt Anfragen gestellt werden.

Potentiell gleiche Elemente

In verschiedenen Dokumenten können Elemente vorkommen, die zwar eine sehr ähnliche oder sogar gleiche Struktur haben, aber an unterschiedlichen Stellen vorkommen. Während die vollständige Extraktion diese als unterschiedliche Elemente auffasst, können die mit der vereinfachten Extraktion und durch Ähnlichkeitsmaße als potentiell gleich erkannt werden [Lan16, S. 41].

Ein Ähnlichkeitsmaß ist die Kombination aus dem Namen des Knotens und dem Namen des Elternknotens. Sind für zwei verschiedene Knoten beide Werte jeweils gleich, die Pfade jedoch unterschiedlich, so werden die Knoten als potentiell gleich angesehen.
Als ein weiteres Maß kann die Kombination aus Kindelementen und Elternelement betrachtet werden. Stimmen auch hier beide Werte überein, liegen potentiell gleiche Elemente vor.

Absolute und relative Häufigkeiten

Aus der Anzahl der Eltern- und Kindelemente lassen sich die absoluten und relative Häufigkeiten ermitteln [Lan16, S. 41]. Das vereinfachte Verfahren kann dies nur auf Elementebene berechnen, da lediglich das vollständige Verfahren auch Dokument-IDs speichert. So können mitunter unterschiedliche Eindrücke entstehen, wenn Ungleichgewichte bei den Häufigkeiten durch eine durchschnittliche Betrachtung wieder relativiert werden.

5.3 Erweiterung des Schemas

Das vorgeschlagene JSON-Schema ist bereits für die Transformation in ein relationales Schema nutzbar. Im Rahmen dieser Arbeit wird das Schema jedoch um zwei Informationen erweitert, die die weitere Verarbeitung in der Transformation erleichtern.

Einerseits um die Eigenschaft `path`, dies ist die Aneinanderreihung der Eigenschaftsnamen bis zu einem bestimmten Datum. In der Schematransformation ist diese Angabe notwendig, um ein bestimmtes Datum aus den JSON-Daten einer Relation und einem Attribut zuordnen zu können. Der Pfad wird dabei wie in Quelltext 5.6 eingebettet.

Quelltext 5.5: Beispielhaftes JSON-Dokument

```
1 {
2   "user" : {
3     "name" : "Felix Beuster"
4   }
5 }
```

Für das Datum `"Felix Beuster"` aus dem Quelltext 5.5 wäre dieser Pfad `/user/name`. Die einzelnen Ebenen werden dabei durch das Trennsymbol `/` getrennt. Diese sollte natürlich nicht in den einzelnen

Namen enthalten und daher anpassbar sein. Bei anfallenden Umbennungen müssen diese natürlich gespeichert und beim Datentransfer berücksichtigt werden.

Bei der Schemaoptimierung werden die Angaben zur absoluten und relativen Häufigkeit eines Elementes oder Datentyps benötigt. Um diese ohne aufwendiges Parsen auslesen zu können, werden sie statt nur in der Beschreibung auch als separate Eigenschaften gespeichert.

Quelltext 5.6: Einbettung der Pfad-Eigenschaft und der Häufigkeiten (Auszug aus dem Schema des Tesla Datensatzes)

```
"$oid": {
  "type": "class com.google.gson.JsonPrimitive.String",
  "description": "Document occurrence: 6/6, 100.0%",
  "path": "/car_orders/_id/$oid"
  "document_occurrence_actual": 6,
  "document_occurrence_total": 6,
  "document_occurrence_relative": 100.0
}
```

Kapitel 6

Transformation eines JSON-Schemas in ein relationales Schema

6.1 Transformation des Schemas

6.1.1 Vorraussetzungen

Für die Erstellung eines relationalen Schemas wird ein JSON-Schema vorrausgesetzt, wie es in [Lan16] erstellt wird. Dieses soll zudem wie in Kapitel 5.3 beschrieben erweitert worden sein. Ein beispielhaftes JSON-Schema befindet sich in Quelltext 6.1.

Quelltext 6.1: Extrahiertes JSON-Schema

```
1  {
2    "title": "car_orders",
3    "description": "Json Schema for collection car_orders of database test
       , created on 2016-07-25 15:53:05.",
4    "$schema": "http://json-schema.org/draft-04/schema#",
5    "properties": {
6      "name": {
7        "type": "class com.google.gson.JsonPrimitive.String",
8        "description": "Document occurrence: 1/1, 100.0%",
9        "path": "/car_orders/name"
10     }
11   },
12   "required properties": [
13     "name"
14   ]
15 }
```

Eine Transformation von Schemas mit Arrays als Wurzelelement erfolgt hier nicht. Systeme wie MongoDB haben stets ein Objekt als Wurzelelement. Werden Arrays importiert, werden diese aufgeteilt und als einzelne Dokumente gespeichert [Mon16].
Darüber hinaus kann ein Objekt als Wurzelelement noch weitere Informationen bereitstellen. Unter anderem wird die `title` Eigenschaft bei der Transformation genutzt.

Als relationales Datenbanksystem wird für das Transformationsverfahren MySQL verwendet. Es zeichnet sich durch eine starke Verbreitung insbesondere bei Webapplikationen aus und ist zudem Open Source. Es sei jedoch angemerkt, dass für das grundsätzliche Vorgehen des Transformationsverfahrens die Wahl des relationalen Systems nicht relavant ist.

6.1.2 Abbildung der Strukturen

Im Allgemeinen kann die Idee zur Abbildung der Strukturen in drei Bereiche gefasst werden:

1. JSON-Objekte werden auf einzelne Relationen abgebildet.
2. JSON-Arrays werden ebenfalls auf einzelne Relationen abgebildet.
   Hierfür wird zudem die Ordnungsnummer der Elemente in ein separates Attribut gespeichert.
3. JSON-Primitive wie Strings, Numbers, Booleans und Null-Werte, werden auf Attribute von Relationen abegebildet.

Über diesen Ansatz hinaus gibt es noch zwei Besonderheiten, die zu beachten sind.

Verschachtelte JSON-Arrays und JSON-Objekte

Das verschachtelte Element wird rekursiv wie das Elternelement behandelt, es erfolgt also wiederum eine Untersuchung auf die drei zuvor erwähnten Bereiche. Das Elternelement speichert anstelle des Objektes oder Arrays einen Verweis auf das verschachtelte Element. Das Attribut wird somit zum Fremdschlüssel.

Datenattribute für Arrays

Arrays verfügen nicht über Namen für ihre Elemente. Daher werden diese Namen für die Attribute generiert. Für die möglichen Datentypen (String, Number, Boolean, Null, sowie Fremdschlüssel auf Arrays und Objekte) wird jeweils ein einzelnes Attribut angelegt. Für Strings wird der Attributname `value_string` generiert. Analog dazu werden die anderen Attribute benannt.

In der Schema-Transformation werden nur Attribute für vorhandene Daten angelegt um so Null-Spalten in einer Relation zu vermeiden. In einer Implementierung sollte dem Anweder jedoch auch eine Einstellungsmöglichkeit gegeben werden, dass Attribute für alle möglichen Typen angelegt werden.

6.1.3 Namen von Relationen und Attributen

Bei der Übertragung von JSON-Strukturen in Relationen kann es vorkommen, dass Relationen mit dem gleichen Namen erstellt werden. Dies kann darauf hindeuten, dass es sich um die gleichen Strukturen handelt, jedoch können sie auch unterschiedlich sein. Diese Betrachtung erfolgt allerdings erst in der Schemaoptimierung.

Bei der Erstellungen des relationalen Schemas wird zunächst darauf geachtet, dass die Namen eindeutig sind, da dies für die Relationen innerhalb einer Datenbank notwendig ist. Um eine Eindeutigkeit zu gewährleisten wird im Falle einer Namenskollision einer der beiden Namen um einen Zähler erweitert. Der genaue Ablauf ist in Abbildung 6.1 dargestellt. In Konflikt stehende Relationen werden darüber hinaus in einer Liste vermerkt, die in der Optimierungsphase genutzt wird.

Die Namen von Attributen einer Relation müssen ebenfalls eindeutig sein. Allerdings erlaubt es JSON unter anderem, in einem Objekt den selben Schlüssel mehrfach zu verwenden. Durch die Erstellung von künstlichen Attributen wie dem Primärschlüssel oder ein Attribut für die Ordnung in Arrays, kann es ebenfalls zu Kollisionen kommen. Es kommt der gleiche Algorithmus aus Abbildung 6.1 zum Einsatz,

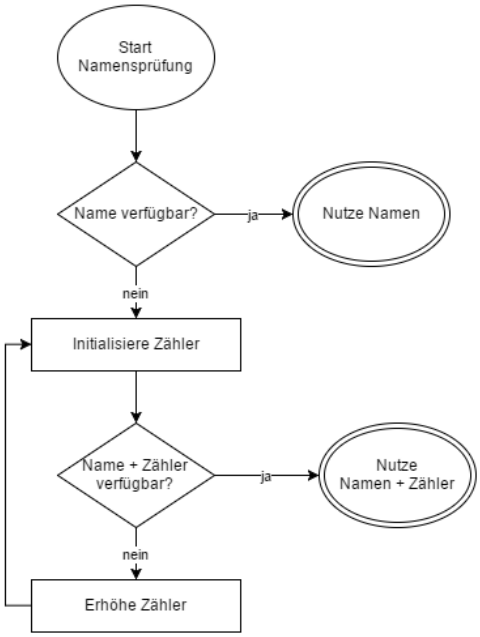

Abbildung 6.1: Ablauf der Namensprüfung

un einen eindeutigen Namen zu finden. Ein freier Name wird dabei nur innerhalb der aktuellen Relation gesucht.

Um beim späteren Datentransfer zu einem JSON-Knoten die passende Relation mit dem passendem Attribut zu finden, werden bei der Schemaerstellung für jeden Knoten bereits der aktuelle Pfad zu dem Knoten, der Name der Relation und des Attributs, sowie der Typ des Attributs in einer Hash-Tabelle gespeichert. Dadurch kann im Datentransfer schnell das Abbildungsziel eines Datums gefunden werden. Mehr dazu in Abschnitt 7.1.

6.1.4 Primärschlüssel und Attributdatentypen

Primärschlüssel

Eine Relation benötigt einen Primärschlüssel, über den die Tupel eindeutig identifiziert werden können. Dieser kann aus einem Attribut oder mehreren Attributen bestehen.
Es kann jedoch nicht davon ausgegangen werden, dass ein bestehender Datenbestand für jedes JSON-Objekt eine passende Eigenschaft hat, die eindeutig ist. Daher wird während der Schematransformation ein künstliches Attribut geschaffen, dass als Primärschlüssel fungiert. Hierfür wird ein `Integer` Wert genommen, welcher durch den Zusatz `AUTO INCREMENT` den Wert selbstständig generiert und so unabhängig vom Datentransfer ist. Dieses Attribut wird hier `ID` genannt.

Jedoch sollten Anwender darüber hinaus die Möglichkeit bekommen, existierende Eigenschaften, die bereits als eine Art Schlüssel in der NoSQL-Datenbank verwendet werden, als Primärschlüssel zu definieren. Dies kann in einer Konfiguration geschehen. Näheres dazu im Abschnitt 8.3.
Eine automatisierte Erkennung solcher Schlüssel ist auch möglich und könnte zur Laufzeit der Schemaextraktion durchgeführt werden. Alternativ wäre während der Schematransformation der Datenbestand erneut zu durchsuchen, was zu einer deutlich erhöhten Laufzeit führen würde.

Beim Einfügen eines Arrays in die entsprechende Relation ist ein einzelner Primärschlüssel jedoch nicht mehr eindeutig, da jedes Element des Arrays den gleichen Array-Schlüssel besitzt. Daher wird ein weiteres Attribut eingeführt, welches `order` gennant wird. Es handelt sich hierbei um die Ordnungsnummer des Elements. Die Kombination von Array-Schlüssel und Ordnungsnummer in einem Primärschlüssel ist wiederum eindeutig für jedes Element des Arrays. In diesem Fall verfügt der ursprüngliche `Integer` Schlüssel auch über kein `AUTO INCREMENT` mehr. Der Wert muss nun manuell vor dem Einfügen eines Datums abgefragt und inkrementiert werden.

Die Namen der künstlichen Attribute führen möglicherweise zu Konflikten mit existierenden Attributen. Eine Behandlung dieser Konflikte wurde bereits besprochen. Darüber hinaus sollte es Anwendern in einer Implementierung möglich sein, die Namen dieser künstlichen Attribute anpassen zu können.

Datentypen

JSON-Dokumente können die folgenen, primitiven Datentypen enthalten, welche in einer Relation dargestellt werden müssen:

- boolean
- null
- number
- string

Die Abbildung dieser Typen auf Datentypen der relationen Datenbank erfolgt gemäß der nachfolgenden Auflistung. Sie wurden gewählt um eine große Flexibilität zu gewährleisten. Ein Anwender sollte jedoch auch hier Anpassungen vornehmen können, um zielgerichteter auf den vorliegenden Datenbestand einzugehen. Idealerweise sollte diese Anpassung auf Datenbank-, Relationen- und Attribut-Ebene möglich sein. Dies gilt insbesondere für Attribute die Zahlen oder Texte enthalten.

boolean Hierfür wird der Datentyp `TINYINT` gewählt, da wir nur eine Abbildung der boolschen Werte auf 0 und 1 benötigen.

null Für `null`-Werte gibt es in MySQL keinen separaten Datentyp. Daher wird der allgemeiner Datentyp `INT` gewählt, welcher dann mit `null`-Werten gefüllt wird.

number JSON selbst unterscheidet nicht zwischen Gleitkommazahlen und ganzen Zahlen. Um daher beide Bereiche und auch große Zahlen abzudecken, wird hier der Datentyp `DOUBLE` gewählt.

string Hier wird der Datentyp `MEDIUMTEXT` gewählt. Dieser erlaubt es bis zu 16 MB Text abzuspeichern.

JSON-Eigenschaften mit mehreren Datentypen

In einem großem Datenbestand ist es wahrscheinlich, dass bedingt durch Update-Prozesse und Schemawechsel, verschiedene Dokumente dieselbe JSON-Eigenschaft in unterschiedlichen Datentypen vorhalten. So kann bspw. eine Bestellnummer in alten Dokumenten als String gespeichert worden sein, während es in neuen Dokumenten bereits als Zahl gespeichert ist. Dies muss während der Schemetransformation mit einbezogen werden.

Als Lösung für dieses Problem wird für jeden vorkommenden Datentyp ein separates Attribut erstellt. Der Attributname ergibt sich in diesem Fall aus dem Namen der JSON-Eigenschaft erweitert um einen

Suffix, um den Datentyp lesbar zu kennzeichnen. Im Beispiel der Bestellung könnten die beiden Attribute dann `bestellnummer_string` und `bestellnummer_number` genannt werden.
Da hier wiederum die Gefahr von Namenskollisionen besteht, wird der bereits beschriebene Mechanismus zur Kollisionsbeseitigung angewandt.

Dieser Fall von mehreren Datentypen macht es notwendig, dass im beschriebenen Data Mapping (Abschnitt 7.1) neben dem Pfad einer JSON-Eigenschaft auch der Datentyp als Schlüssel mit verwendet wird.

6.1.5 Auflösung von n-m-Beziehungen und Redundanz

JSON-Datenbestände können eine hohe Redundanz der Daten aufweisen, wenn Werte an verschiedenen Stellen innerhalb eines Dokumentes oder in verschiedenen Dokumenten verwendet werden. Im Testdatensatz 2 tritt dies unter anderem bei den Adressen auf. In den meisten Fällen ist diese Redundanz eine 1-n-Beziehung.
Verweist eine JSON-Eigenschaft auf ein Array welches wiederum Arrays oder Objekte enthält, kann es sogar zu n-m-Beziehungen kommen. Hier verweist eine JSON-Eigenschaft auf mehrere Werte und diese Werte können könne wiederum in mehreren JSON-Eigenschaften referenziert werden.
Da das Transformationsverfahren direkt Relationen für Arrays erstellt, werden diese n-m-Beziehungen automatisch in 1-n-Beziehungen aufgelöst.

Redundanzvermeidung

Die Auflösung der Redundanz sollte ebenfalls Ziel der Transformation von einer NoSQL-Datenbank zu einer SQL-Datenbank sein. Ein Ansatz hierfür ist die Verwendung eines `UNIQUE INDEX`, der alle Datenattribute umfasst. Dieser stößt jedoch schnell an technische Grenzen von MySQL. Für die Erstellung eines Index für `TEXT` Attribute benötigt MySQL die Angabe einer feste Länge für diesen Index. Diese ist in Standardeinstellung allerdings auf 767 Bytes limitiert und kann auf maximal 3072 Byte erhöht werden [MyS16]. Dies ist jedoch bedeutend kleiner als die mögliche Länge von `TEXT` Attributen, die bei 64 KB liegt. Darüber hinaus ist auch das Limit eines kombinierten Index bei 3072 Bytes und die Länge für einzelne Spalten damit weiter limitiert. Für Datenbestände mit langen Texten würde die Verwendung eines `UNIQUE INDEX` somit einen hohen Datenverlust verursachen können.

Als Alternative zum Index auf dem `TEXT` Attribut kann ein Index auf dem Hash des Textes betrachtet werden. Vorteil des Hashes ist seine feste Länge und er ist zudem bdeutend kürzer als das `TEXT` Attribut. Allerdings macht dies seine Verwendung wiederum problematisch. Beschränkt man den Zeichensatz für das Attribut auf ASCII, so ergeben sich $128^{65636} \approx 10^{100000}$ mögliche Werte für das Attribut. Selbst mit der Verwendung des langen SHA512 Hashes können nur $16^{128} \approx 10^{154}$ verschiedene Hashes existieren. Die Menge der Hashwerte ist also bedeutend kleiner als die Menge möglicher Texte und macht Kollisionen somit wahrscheinlich. Diese Wahrscheinlichkeit erhöht sich zudem, wenn größere Typen wie `MEDIUMTEXT` und Zeichensätze wie UTF-8 verwendet werden. In einem `UNIQUE INDEX` bedeutet eine Kollision die Nichtaufnahme eines Datenfeldes. Auch ein Index auf Hashwerten von Texten kann somit nicht ohne Datenverlust eingesetzt werden.

Eine weitere Möglichkeit der Vermeidung von Datenredundanz ist eine `SELECT` Anfrage vor dem `INSERT` Befehl. Hierbei müssen im `WHERE` Teil der Anfrag alle einzufügenden Attribute verglichen werden. Ist ein Eintrag in der Relation gefunden worden, werden keine neuen Daten eingefügt. Im Falle das die einzufügenden Daten von einem anderen Attribut referenziert werden sollen, wird stattdessen auf das bestehende Tupel verwiesen. Diese Variante unterliegt keinen Limitierungen in Bezug auf eine Schlüssellänge oder Menge von Einträgen, jedoch benötigt sie mit zunehmer Datenmenge auch mehr Zeit in der Ausführung.

Es ist hier ein quadtratischer Aufwand zu erwarten, da schlimmstenfalls für einen Eintrag n insgesamt $n-1$ Vergleiche durchgeführt werden. Damit ergibt sich der Worst-Case-Aufwand von

$$\mathcal{O}(n*(n-1)) = \mathcal{O}(n^2)$$

Der Aufwand der vorherigen Methode lässt sich bedeutend reduzieren, wenn ein regulärer Index auf den `TEXT` Attributen angelegt wird. Im Gegensatz zu einem `UNIQUE INDEX` unterstützt der reguläre `INDEX` auch mehrfache Werte. Da dieser Index lediglich als Zugriffshilfe für die `INSERT` mit `SELECT` Methode verwendet wird, stellen mehrfache Werte hier auch kein Problem dar.
Diese Variante verhindert Redundanz in Relationen, weist keinen Datenverlust auf und hat akzeptable Zugriffszeiten. Daher sollte dieses Verfahren als Standard in einer Implementierung verwendet werden.

Trotz der Nachteile der ersten beiden Verfahren zur Redundanzvermeidung sollte in einer Implementierung der Anwender die Möglichkeit zur Auswahl eines Verfahrens bekommen. Ist bspw. bekannt, dass ein Datenbestand keine oder nur wenige kurze Texte aufweist, so kann ein `UNIQUE INDEX` anstelle des standardmäßigen `SELECT` Verfahrens genutzt und die Laufzeit der Datentransformation reduziert werden.

Ein Vergleich der Laufzeiten der verschiedenen Methoden wird in Abschnitt 8.4.2 vorgenommen.

6.1.6 Mögliche Relation nach der Transformation

Im Folgenden wird beispielhaft die Datei aus Quelltext 6.2 behandelt. Zur besseren Übersicht ist diese einfach gehalten und enthält lediglich eine Objekteigenschaft für den Namen eines Nutzers.
Auf das extrahierte Schema in Quelltext 6.3 wurde eine Transformation ohne weitere Optimierungen zur Redundanz durchgeführt. Das Ergebnis ist ein relationales Schema in Form von `CREATE TABLE` Anweisungen, die in eine Datenbank übernommen werden können. Dieses relationale Schema ist in Quelltext 6.4 dargestellt.

Quelltext 6.2: Beispielhafte JSON-Datei vor Extraktion und Transformation

```
{
  "name" : {
    "realname" : "Felix Beuster",
    "username" : "FBeuster"
  }
}
```

Quelltext 6.3: Extrahiertes Schema aus Quelltext 6.2

```
{
  "title": "user",
  "description": "Json Schema for collection user of database myDB,
     created on 2016-08-05 19:13:29.",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "properties": {
    "name": {
      "type": "class com.google.gson.JsonObject",
      "description": "Document occurrence: 1/1, 100.0%",
      "path": "/car_orders/name",
      "document_occurrence_actual": 1,
      "document_occurrence_total": 1,
      "document_occurrence_relative": 100.0,
```

```
      "properties": {
        "realname": {
          "type": "class com.google.gson.JsonPrimitive.String",
          "description": "Document occurrence: 1/1, 100.0%",
          "path": "/car_orders/name/realname",
          "document_occurrence_actual": 1,
          "document_occurrence_total": 1,
          "document_occurrence_relative": 100.0
        },
        "username": {
          "type": "class com.google.gson.JsonPrimitive.String",
          "description": "Document occurrence: 1/1, 100.0%",
          "path": "/car_orders/name/username",
          "document_occurrence_actual": 1,
          "document_occurrence_total": 1,
          "document_occurrence_relative": 100.0
        }
      },
      "required properties": [
        "realname",
        "username"
      ]
    }
  },
  "required properties": [
    "name"
  ]
}
```

Quelltext 6.4: Transformiertes Schema aus Quelltext 6.3

```
CREATE TABLE `myDb`.`name`(
  `ID` INT NOT NULL AUTO_INCREMENT,
  `realname` MEDIUMTEXT NULL,
  `username` MEDIUMTEXT NULL,
  PRIMARY KEY (`ID`))

CREATE TABLE `myDb`.`car_orders`(
  `ID` INT NOT NULL AUTO_INCREMENT,
  `nameID` INT NULL,
  PRIMARY KEY (`ID`))
```

6.2 Optimierung des relationalen Schemas

Das Ergebnis der Transformation ist ein vollständiges, relationales Schema, dass direkt in eine Datenbank integriert werden kann. Jedoch kann dieses Schema optimiert und die Zahl der Relationen reduziert werden. Dazu werden hier zwei Konzepte betrachtet. Darüber hinaus lassen sich die Metriken, die in der Schemaextraktion gewonnen wurden, für weitere Optimierungen nutzen.

6.2.1 Verschmelzen von Relationen

Der Ansatz Relationen zu verschmelzen ist ähnlich zu dem Ansatz von [DA16], welcher allerdings direkt auf den Daten arbeitet. Im Rahmen dieser Arbeit werden Relationen syntaktisch betrachet, und so nach einer Ähnlichkeit gesucht. Die grundsätzliche Idee bleibt jedoch die gleiche.

Ähnlichkeitsanalyse

Betrachtet werden bei der Ähnlichkeitsanalyse zweier Relationen die folgenden Eigenschaften: Name der Relationen, sowie Anzahl, Namen und Datentypen der Attribute. Konkret untersucht werden die folgenden fünf Punkte:

1. Haben zwei Relationen den gleichen Namen?
2. Stimmen die Namen der Attribute zweier Relationen überein?
3. Stimmen die Typen der Attribute zweier Relationen überein?
4. Ist die Anzahl der Attribute zweier Relationen gleich?
5. Ist die Menge der Attribute einer Relation eine Teilmenge der Attribtue einer anderen Relation?

Bei der Betrachtung der Namen von Relationen und Attributen werden die Namen vor der Erstellung eindeutiger Namen (vgl. 6.1.3) genutzt.

Keines der genannten Merkmale sollte jedoch als alleiniges Kriterium für die Gleichheit zweier Relationen betrachtet werden. So hat bspw. eine Relation *Länder* mit den Attributen *ID*, *Land* und *Landeskürzel* die Attributypen `INTEGER`, `TEXT` und `TEXT`. Dies wäre aber auch der Fall für eine Relation *Nutzer* mit den Attributen *ID*, *Klarname* und *Nutzername*.
Daher sollten immer mehrere Kriterien ausschlaggebend sein. In diesem Verfahren werden drei Kriterien gewählt. Somit wird eine 60%-ige Erfüllung der Kriterien gefordert.

Limitierungen

Die obige Analyse basiert jedoch rein auf der Struktur der JSON-Dokumente und erkannte Optimierungen müssen nicht den tatsächlichen Datenbestand semantisch korrekt abbilden. Daher sollte eine automatische Optimierung nur dann erfolgen, wenn ein Anwender einer Implementierung dies explizit wünscht. Hierzu sollte ein Anwender in der Lage sein, dem Transformationsverfahren eine Liste von Relationsnamen zu geben, welche dann automatisiert verschmolzen werden können. Alternativ sollten vom System vorgeschlagene Optimierung nach einer Bestätigung durch den Nutzer angewandt werden können.

6.2.2 Einbetten von Relationen

Die vorgestellten Einbettungsstrategien nach [STZ+99] werden hier nicht direkt angewendet, jedoch als Anlehung genutzt. Basic Inline verursacht einen hohen Anteil an redundanten Daten, Shared Inline kann in sehr großen Relationen resultieren, wenn es keine oder nur wenige gemeinsame Relationen gibt. Hybrid Inline vereinigt zwar die Vorteile beider Techniken, allerdings auch die genannten Nachteile.

Für das Einbetten von Relationen wird hier daher ein einfacher Ansatz erfolgt. Eine Relation muss die folgenden Bedingungen erfüllen, um eingebettet zu werden:

1. Die Relation ist nicht die Abbildung des JSON-Wurzelobjekts.
2. Die Relation wird nur von einer anderen Relation referenziert.

3. Die Relation ist nicht die Abbildung eines JSON-Arrays.
4. Die Relation enthält maximal n Attribute.

Die Relation des JSON-Wurzelobjekts kann von Natur aus nicht weiter eingebettet werden. Bettet man eine Relation ein, die von mehr als einer anderen Relation referenziert wird, können starke Redundanzen erzeugt werden, die vermieden werden sollten. Dennoch sei erwähnt, dass man auch mehrfach referenzierte Relationen einbetten kann, wenn man als Zielsetzung die Reduktion von Joins hat.

Relationen, die aus der Abbildung von JSON-Arrays entstanden sind, können ebenfalls nicht eingebettet werden. Ein Array besteht aus mehreren Tupeln, eine Einbettung würde daher zu komplexen Attributwerten führen. Alternativ könnten die Tupel separat bleiben, allerdings muss das Tupel der Elternrelation dann auch mehrfach wiederholt werden, sodass auch hier eine große Redundanz entstehen würde.

Der Grenzwert n ist eine natürliche Zahl die angibt, wie viele Attribute eine Relation haben darf, um eingebettet werden zu können. Besitzt eine Relation nur ein Primäschlüssel- und ein Werteattribut, so kann diese generell einfach eingebettet werden. In diesem Falle ist $n = 2$. In einer Implementierung sollte dieser Wert für Anwender anpassbar sein.

6.2.3 Einbezug der Metriken aus der Extraktion

Die bei der Schemaextraktion gewonnenen Metriken können ebenfalls zur Optimierung des Schemas genutzt werden.

Eine dieser Metriken ist die Information über benötigte und optionale Elemente. Ein optionales Element kann von sehr vielen, oder aber auch nur sehr wenigen Dokumenten genutzt werden. Unterschreitet ein JSON-Element eine bestimmte relative Häufigkeit, kann es als Attribut entfernt werden.
Gleiches gilt für die relativen Häufigkeiten von Datentypen eines JSON-Elements. Ist bspw. die Mehrheit eines JSON-Elements vom Typ Number und nur ein kleiner, veralteter Teil der Daten vom Typ String, so kann auch hier auf den wenig genutzten Datentyp als Attribut verzichtet werden.

In beiden Fällen kommt es jedoch zu Datenverlust. Daher sollte ein solches Auslassen der Attribute nicht automatisiert erfolgen, sondern nur auf Anweisungs des Anwenders hin. Zudem ist es in einer Implementierung notwendig die Grenzwerte der relativen Häufigkeit individuell dem vorliegendem Datenbestand anzupassen.

6.2.4 Weitere Optimierungsmöglichkeiten

Eine weitere Möglichkeit das resultierende Schema anzupassen und zu optimieren ist Schema-Matching, also das Vorgeben eines oder mehrerer Teilschemata, die im Ergebnis der Transformation enthalten sein sollen. Dies ist jedoch nicht mehr Teil dieser Arbeit und wird im Fazit (Abschnitt 9.2) noch einmal näher beschrieben.

Ferner kann betrachtet werden, ob alle Daten tatsächlich als Relation oder separates Attribut notwendig sind. Zusammenhängende Daten, bei denen zu erwarten ist, dass sie nur wenig und zumeist zusammen abgefragt werden, kann eine Verschmelzung der Relation ein einzelnes Attribut in Betracht gezogen werden. Diese Verschmelzung erfordert jedoch ein Verständnis der Daten und kann daher hier nicht automatisch vorgenommen werden. Eine Implementierung kann lediglich die Option bieten, vom Anwender definierte Attribute und Relationen zu verschmelzen.

Kapitel 7

Transformation der JSON-Dokumente in das relationale Schema

7.1 Data Mapping Log

Um bei der Transformation der Daten aus den JSON-Dokumenten nicht eine erneute Schematransformation durchführen zu müssen, sollte während der Schematransformation notiert werden, auf welche Relation und welches Attribut ein Datum abgebildet wird.

Hierzu werden zwei Hash-Maps benötigt. Die eine speichert die Abbildung eines Datums auf eine Relation, die andere die Abbildung auf ein Attribut. Als Schlüssel für die Maps wird eine Kombination aus dem Pfad eines Datums und dem Typ des Attributs genutzt. Der Pfad ist hierbei eine Aneinanderreihung der Namen der Eigenschaften bis zu einem bestimmten Datum, welche durch ein Trennsymbol getrennt sind. Dieses sollte nicht in den Namen der Eigenschaften vorkommen und natürlich konifgurierbar sein.
Der Typ eines Attributs als Teil eines Schlüssels ist notwendig, da bspw. die gleiche Eigeschaft eines JSON-Objektes in unterschiedlichen Dokumenten unterschiedliche Typen haben kann. Diese werden wiederum auf unterschiedliche Attribute abgebildet.

Während der Verarbeitung der einzelnen Dokumente kann nun für ein Datum ein effizienter Abruf der Zielrelation und des Zielattributs erfolgen.
Da im Zuge einer Schemaoptimierung sich Zielrelation und -attribut ändern können, muss es möglich sein, das Data Mapping Log im Nachhinein zu ändern. Hierzu gehören Methoden wie das aktualisieren von Einträgen aber auch das Löschen. Dies muss nur anhand der Relation, des Attributs und des Typs möglich sein, da die Schemaoptimierung über keine Pfadinformationen mehr verfügt. Dadurch dass eine Kombination aus Pfad und Typ immer nur auf genau eine Relation und ein Attribut abgebildet wird, ist diese Rückwärtssuche fehlerfrei möglich.

Für das Beispiel aus Quelltext 6.2 könnte eine tabellarische Darstellung des Data Mapping Logs wie in Tabelle 7.1 aussehen.

Pfad - Typ	**Relation**	**Attribut**
/nutzer/name - object	nutzer	name_ID
/nutzer/name/realname - string	name	realname
/nutzer/name/username - string	name	username

Tabelle 7.1: Beispielhafter Data Mapping Log

7.2 Erstellung der SQL-Anweisungen

7.2.1 Einrichtung der Tabellen

Die Transformation der Daten aus den JSON-Doukmenten beginnt mit dem Anlegen der Relationen selbst. Hierzu erfolgt eine Iteration über die Liste von Relationen, die von der Optimierung zurückgegeben wurden. Für jede Relation wird eine SQL-Export Methode aufgerufen, die die `CREATE TABLE` Anweisung, wie in Quelltext 7.1 dargestellt, erzeugt.

Quelltext 7.1: Beispielhafte `CREATE TABLE` Anweisung

```
1 CREATE TABLE `myDb`.`user`(
2   `ID` INT NOT NULL AUTO_INCREMENT,
3   `age` INT NULL,
4   `name` MEDIUMTEXT NULL,
5   PRIMARY KEY (`ID`));
```

Hierbei werden die unterschiedlichen Nutzereinstelungen berücksichtigt und bei Bedarf Indices auf Textattributen oder ein `UNIQUE INDEX` erzeugt. Bei der Erstellung von Indices muss darüber hinaus berücksichtigt werden, dass sowohl die einzelne aber auch kombinierte Größe der Indices nicht die Limitierungen von MySQL überschreitet [MyS16].

Die generierten SQL-Anweisungen werden zunächst in einer Liste zwischengespeichert bevor sie an die Datenbank zur Ausführung übermittelt werden.

7.2.2 Datentransformation mit `INSERT`-Anweisungen

Für die Transformation der konkreten Daten werden `INSERT`-Anweisungen konstruiert. Hierzu wird zunächst eine Verbindung zur der *Not only SQL* (NoSQL)-Datenbank MongoDB aufgebaut und anschließend iterativ in einer Schleife jedes Dokument verarbeitet.

Wie bereits bei der Schemaextraktion werden die Dokumente nach rekursiv verarbeitet. Jedes Objekt oder Array erzeugt hierbei eine separate `INSERT`-Anweisung. Auf jeder Rekursionsebene werden für die Daten der Relationenname und das Attribut abgefragt, wozu der Typ und der Pfad eines Datums notwendig sind. Für jedes Datum einer Ebene, also alle Eigenschaften eines Objekts oder alle Elemente eines Arrays, ist der Relationenname gleich. Nachdem diese Informationen ermittelt wurden, wird eine `INSERT`-Anweisung wie in Quelltext 7.2 erstellt.

Die generierten SQL-Anweisungen werden ebenfalls zunächst in einer Liste zwischengespeichert bevor sie an die Datenbank zur Ausführung übermittelt werden.

Quelltext 7.2: Beispielhafte `INSERT INTO` Anweisung

```
1 INSERT INTO `user`(`age`, `name`) VALUES (25, "Felix Beuster");
```

Bei der Erstellung der `INSERT`-Anweisung wird auch berücksichtigt, ob Hash-Attribute für die Textfelder angelegt wurden. Diese werden dann entsprechend wie in Quelltext 7.3 mit einer MySQL-Funktion befüllt.

Quelltext 7.3: Beispielhafte `INSERT INTO` Anweisung mit Hash

```
1 INSERT INTO `user`(`age`, `name`, `name_hash`) VALUES (25, "Felix
    Beuster", SHA2("Felix Beuster", 512));
```

Ebenfalls wird hier berücksichtigt, ob ein `SELECT` vor dem `INSERT` benötigt wird. Das Resultat ist dann eine komplexere Anweisung wie in Quelltext 7.4 schematisch dargestellt. Die Anweisung wird anschließend näher erläutert.

Quelltext 7.4: Beispielhafte `INSERT INTO` Anweisung mit `SELECT`

```
1  INSERT INTO tabelle (attribut)
2    SELECT *
3    FROM (SELECT wert AS attribut) AS temp_name
4    WHERE NOT EXISTS (
5      SELECT primaer
6      FROM tabelle
7      WHERE
8        attribut = wert
9    )
10   LIMIT 1;
```

`(SELECT wert AS attribut) AS temp_name` Hier wird eine temporäre Relation mit den gegebenen Werten erzeugt. Die Werte werden dabei den passenden Attributen zugeordnet.

`SELECT primaer FROM tabelle WHERE attribut = wert` Dies prüft, ob die gegebenen Daten bereits einen Datensatz in der bestehenden Relation bilden.

`SELECT * FROM ... WHERE NOT EXISTS (...) LIMIT 1` Hierdurch werden alle Attribute und Werte der temporären Relation ausgewählt, sofern sie nicht bereits (wie zuvor getestet) in der bestehenden Relation enthalten sind.

`INSERT INTO tabelle (attribut) ...;` Die Ergebniswerte des vorherigen `SELECT *` werden hier nun in die bestehende Relation eingefügt.

Schlüssel verschachtelter Objekte und Arrays

Relationen können zwei Arten von Primärschlüsseln haben. Für Objektrelationen ist dies ein ganzahliger Wert mit `AUTO_INCREMENT`. Beim Einfügen von Daten braucht dieser also nicht manuell mit einem Wert versehen werden.
Im Falle von Arrays ist der Primärschlüssel eine Kombination aus einer ganzzahligen Array-ID und der Ordnungsnummer. Ein `AUTO_INCREMENT` kann hier nicht angewendet werden. Bevor ein neuer Eintrag erstellt werden kann, wird das aktuelle Maximum der Array-IDs abgefragt, manuell um 1 erhöht und anschließend als Array-ID verwendet. Hier ist es nötig sicherzustellen, dass keine zwei Arrays die gleiche Array-ID zugewiesen bekommen. Dies kann durch den Ausschluss von Parallelität der Datentransformation oder durch entsprechende Sperren auf der Relation erfolgen.

Werden verschachtelte Objekte und Arrays in ihre respektiven Relationen eingefügt, so muss deren Primärschlüssel in die aktuelle Relation als Fremdschlüssel aufgenommen werden. Für Relationen mit einem `AUTO_INCREMENT`-Attribut als Primärschlüssel, kann dafür in MySQL `LAST_INSERT_ID()` genutzt werden. Dieser Wert wird dann wie in in einer Variable gespeichert, die in der nächsten `INSERT`-Anweisung genutzt wird.

Quelltext 7.5: Speichern eines Primärschlüssels

```
1  @FOREIGN_ID = LAST_INSERT_ID();
```

Da der Wert für Arrays manuell gesetzt wird, kann `LAST_INSERT_ID` hier nicht eingesetzt werden. Es kann jedoch die bereits erstellte `MAX`-Variable weiter genutzt werden.
Darüber hinaus liefert `LAST_INSERT_ID` im Falle eines `INSERT` mit `SELECT` bei erkannten Duplikaten keinen oder nur veraltete Werte zurück. Dementsprechend muss die jeweilige ID mit einer separaten `SELECT`-Anfrage und exact-matching auf allen Attributen abgefragt und gespeichert werden.

7.3 Ausführungen der SQL-Anweisungen

Nachdem alle Anweisungen zum Erzeugen der Tabllen und Einfügen der Daten erstellt wurden, werde diese an die Datenbank übermittelt und ausgeführt. Es wird zunächst eine Verbindung zum konfigurierten MySQL-Server aufgebaut. Die Ausführung erfolgt danach in drei Abschnitten.

1. `DROP TABLE IF EXISTS`
   Sofern vom Anwender gewünscht wird zunächst dieser Befehl für jede Relation durchgeführt. Dies ermöglicht den Datentransfer in eine neue Relation ohne veraltete Daten.

2. `CREATE TABLE`
   Es werden nun die einzelnen Relationen erstellt.

3. `INSERT INTO`
   Abschließend werden die `INSERT INTO` Anweisungen zum Einfügen der Daten ausgeführt.

Während der `DROP TABLE` Schritt über die Liste von Relationen iteriert, durchlaufen `CREATE TABLE` und `INSERT INTO` ihre respektiven Listen von SQL-Anweisungen.
Sind alle Anweisungen ausgeführt und damit alle Daten übertragen worden, kann die MySQL-Verbindung wieder getrennt werden.

7.3.1 Alternativer Ausführungsansatz

Durch die Sammlung der `CREATE TABLE` und `INSERT` Anweisungen ist es möglich, diese Listen noch weiter anzupassen und ggf. weiter zu optimieren. Zudem ist es auf diese Weise einfacher möglich, den eigentlichen Datentransfer verteilt auf mehreren Systemen auszuführen. Allerdings resultiert dieser Ansatz in einem größerem Aufwand im Sinne der Laufzeit und Datenmenge, insbesondere für die `INSERT`-Anweisungen. Diese werden nämlich für alle Dokumente in einer Liste gespeichert, bevor sie ausgeführt werden.
Für die sehr kleinen Dokumente des Testdatensatzes 3 (Abschnitt 8.1.3) beläuft sich ein Dokument auf etwa 500 Bytes für die `INSERT`-Anweisungen, Testdatensatz 2 (Abschnitt 8.1.2 umfasst bereits durchschnittlich 4.500 Bytes. Beides entspricht etwa dem dreifachen der Rohdaten. Legt man für Testdatensatz 2 den durch den Eintrag `order_id` implizierten Umfang von 46.000 Dokumenten zugrunde, beläuft sich der erforderliche Speicher auf 197 MB. Im Bereich der Big Data und agilen Webanwendungen sind aber durchaus auch größere und komplexere Datenbestände zu erwarten, welche dann wiederum deutlich mehr Speicher beanspruchen würden.

Dies kann auf unterschiedliche Arten optimiert werden. Einerseits kann die Datentransformation ausgegliedert werden, sodass diese die nötigen Informationen der Schematransformation als Eingabe erhält und dann nur einen bestimmten Teil der Daten verarbeitet. Damit kann die Datentransformation weiterhin verteilt und nun auch in Intervallen durchgeführt werden.
Eine weitere Möglichkeit ist es, die `INSERT`-Anweisungen nicht zwischenzuspeichern, sondern direkt auszuführen. Dies reduziert den benötigten Zwischenspeicher um nahezu 100%.

Kapitel 8

Testdaten und Implementierung

8.1 Testdaten

8.1.1 Datensatz 1: Soziales Netzwerk

Dieser Datensatz beschreibt die Daten eines fiktiven sozialen Netzwerkes. Ein Beispieldokument aus diesem Datensatz kann im Anhang A.1 eingesehen werden.

Ein einzelnes Dokument entspricht hierbei einem Nutzer, über den persönliche Informationen gegeben sind. Weiterhin enthält ein Nutzerdokument auch eine Auflistung an Followern und Auflistung von Statusmeldungen. Der Datensatz zeichnet sich somit durch eine Mischung aus einfachen Datentypen, Objekten und einfachen Arrays aus.

8.1.2 Datensatz 2: Tesla Motors

In diesem fiktiven Datensatz werden Bestellungen eines Autoherstellers erfasst. Ein einzelnes Dokument entspricht einer Bestellung und ein Beispieldokument kann in Anhang A.2 eingesehen werden.

Der Datensatz beinhaltet einerseits eine Konfiguration des Fahrzeugs, die der Kunde in einem Onlineshop vornehmen kann. Im weiteren Produktionsverlauf wird ein Dokument um Informationen aus der Produktion angereichert, bspw. zugeordnete Arbeiter der Arbeitsschritte, der aktuelle Produktionsstatus und Fehlerberichte. Im Zuge fortschreitender Vernetzung von Fabriken und Maschinen ist eine deutlich höhere Datengeneration denkbar, als in diesem Beispiel dargestellt.
Weiterhin enthalten sind Informationen über den Kunden wie Name und Anschrift, sowie auch Referenzen auf vorhergehende Bestellungen und eine Liste von Problemen mit dem Fahrzeug.

Insgesamt zeichnet sich der Datensatz durch eine höhere Komplexität aus, die unter anderem auch verschachtelte Arrays und verschiedene Datentypen für die selbe JSON-Eigenschaft beinhaltet.

8.1.3 Datensatz 3: Personendaten

Der dritte Datensatz umfasst 1.000 Dokumente, die jeweils ein einfaches und kleines Objekt mit fiktiven Personendaten beinhalten. Generiert wurden diese Daten mit Hilfe des Generators [Oma16]. Dieser wählt Daten zufällig aus einem Pool aus und stellt sie nach einem vorgegebenem Schema zusammen.
Ein einzelnes Dokument enthät nur Namen, Email und Adresse einer Person. Ein beispielhaftes Dokument befindet sich in Anhang A.3.

Dieser Testdatensatz eignet sich aufgrund seiner Größe für einfache Performancemessungen, wie bspw. im Abschnitt 8.4.2 zur Redundanzvermeidung.

8.2 Implementierung

Als Basis für die Implementierung für die Schemaextraktion dienen die Arbeit und eine prototypische Implementierung von [Lan16]. Neben einem Refactoring zu einem objektorientierteren Programm, wurde eine Erweiterung vollzogen. Diese wurde in Abschnitt 5.3 beretis erläutert.

8.2.1 Schematransformation

Die Initialisierung der Transformation beginnt mit dem Anlegen benötigter Speicherstrukturen, wie einer Liste für erstellte Relationen und dem Data Mapping Log (vgl. 7.1). Die Transformation benötigt die `title`-Eigenschaft des JSON-Schemas als Namen für das Root-Objekt, sowie die `properties`-Eigenschaft als zu parsendes Root-Objekt selbst. Aufgerufen wird nach der Initialisierung dann eine `makeRelation`-Methode, die eine Relation aus Namen und JSON-Objekt erstellt.

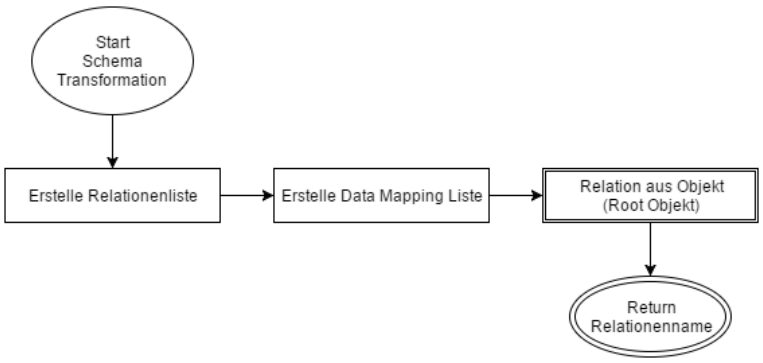

Abbildung 8.1: Initialisierung der Transformation

Diese Methode zur Erstellung einer Relation aus einem Objekt sucht zunächst nach einem eindeutigen Namen, wie bereits in Abschnitt 6.1.3 beschrieben. Des Weiteren wird ein Primärschlüssel erzeugt, bevor eine Methode zur Bearbeitung der einzelnen Objekt-Eigenschaften aufgerufen wird. Diese liefert Attribute zurück, die zur Relation hinzugefügt werden. Abschließend wird die Relation in die Liste der Relationen eingefügt.

Die Bearbeitung der Eigenschaften erfolgt iterativ in einer Schleife. Jeder Durchlauf prüft ob die Eigenschaft einen oder mehrere Typen hat. Für einen einzelnen Typen wird direkt ein Attribut anlegt, in die Liste der Attribute eingetragen. Darüber hinaus wird auch ein Vermerk im Data Mapping Log eingefügt. Im Falle von mehreren Typen für eine Eigenschaft wird eine weitere Schleife gestartet, die diese drei Schritte für jeden Typen wiederholt. Sind alle Eigenschaften abgearbeitet, wird die Liste der Attribute zurückgegeben.

Diese beiden Schleifenprozesse sind gemeinsam mit der Relationenerstellung aus Objekten in Abbildung B.1 dargestellt. Diese befindet sich zur besseren Übersicht mit den Array-Grafiken im Anhang B.

Bei der Erstellung eines Attributes erfolgt die Rekursion auf verschachtelte JSON-Objekte und -Arrays, wie in Abbildung 8.2 dargestellt. Hier erfolgt eine Abfrage, ob es sich um ein weiteres Objekt, Array oder lediglich einen einfachen Datentyp handelt. In den ersten beiden Fällen werden die jeweiligen

`makeRelation`-Methoden für Objekte und Arrays aufgerufen. In die Attributliste wird dann ein Fremdschlüssel aufgenommen, welcher auf die verschachtelte Relation verweist. Für einen einfachen Datentyp wird direkt ein Datenattribut erzeugt.

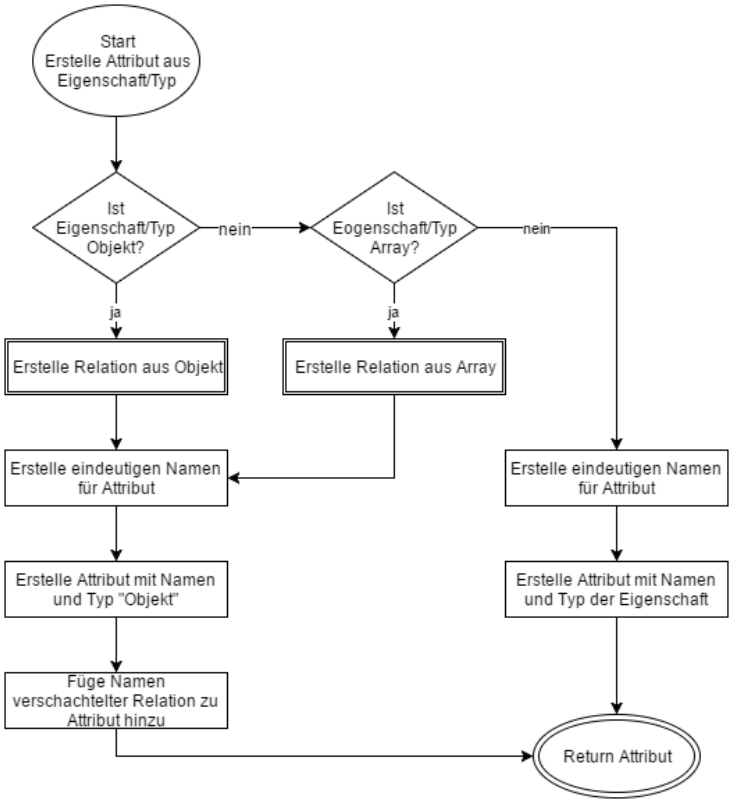

Abbildung 8.2: Attributerstellung und rekursive Aufrufe

Die Methode zur Erstellung einer Relation aus einem Array, in Abbildung B.2, unterscheidet sich konzeptuell nur wenig von der Objektbehandlung. Zunächst wird geprüft, ob es sich um ein eintypisches oder mehrtypisches Array handelt. Bei eintypischen Arrays wird die jeweilige Array-Behandlungsmethode für Objekte, Arrays oder Primitive aufgerufen, wie Abbildung B.3 zu entnehmen ist.
Im Falle von mehreren Typen wird zunächst eine Basis-Relation erstellt. Anschließend werden die einzelnen Typen in einer Schleife iteriert. Jeder Durchlauf erstellt dann ein Attribut für den Datentyp, mit der zuvor besprochenen Behandlung. Die Attribute werden dann der Relation und dem Data Mapping Log hinzugefügt. Sind alle Typen abgearbeitet, wird die Relation zur Relationenliste hinzugefügt.
Unabhängig von der Art des Arrays wird abschließend der Name der Relation zurückgegeben.

Die Erstellung einer Basis-Array-Relation umfasst das Erstellen der Relation, sowie eines ID- und Ordnungsattributs. Beide Attribute werden als gemeinsamer Primärschlüssel definiert.
Für die drei verschiedenen Array-Typen (Array von Objekten, Array von Arrays, Array von einfachen Datentypen) wird jeweils eine Basis-Array-Relation erstellt. Die weitere Behandlung unterscheidet sich wie folgt:

- Bei Objekt-Arrays werden die Eigenschaften des Objekts in die Array-Relation direkt eingebettet. Die einzelnen Eigenschaften werden wie bereits bekannt behandelt.

- Bei Array-Arrays werden für die verschachtelten Arrays neue Relationen angelegt und für das ursprüngliche Array Fremdschlüsselattribute erstellt und der Relation und dem Data Mapping Log hinzugefügt.
- Bei einfachen Arrays wird ein Werte-Attribut erstellt und der Relation sowie dem Data Mapping Log hinzugefügt.

8.2.2 Datentransfer

Der Transfer der Daten beginnt mit einer Initialisierung der Liste für die SQL-Anweisungen sowie dem Verbinden mit MongoDB. Darauf hin werden alle Dokumente der gewählten Kollektion in einer Schleife durchlaufen (Abbildung 8.3). Jeder Durchlauf verarbeitet das aktuelle Dokument als Objekt.

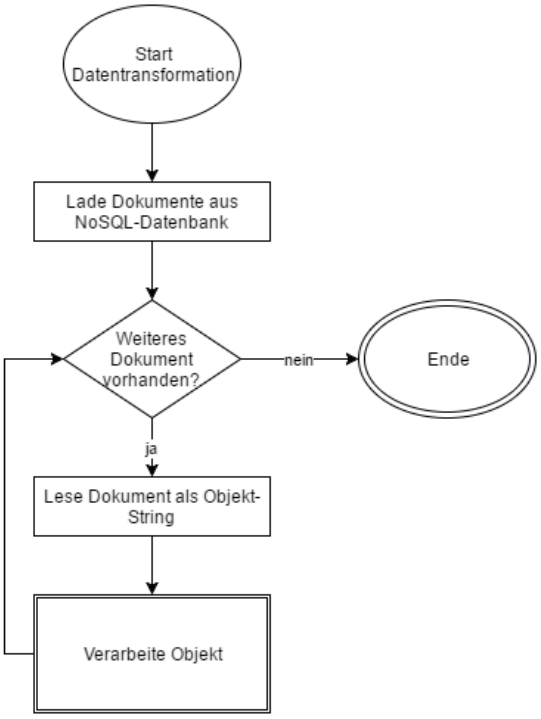

Abbildung 8.3: Verarbeiten der JSON-Dokumente

Für die Verarbeitung eines Objekts werden nun die einzelnen Eigenschaften iteriert und verarbeitet, wie in Abbildung B.4 zu sehen. Während die Daten von primitiven Attributen direkt zwischengespeichert werden können, müssen für die Verarbeitung von Objekt- oder Arrayeigenschaften Subprozesse aufgerufen werden, wie in Abbildung 8.4 verdeutlicht.
Zudem wird hier auch der Relationenname abgerufen und gespeichert. Am Ende der Verarbeitung wird eine `INSERT`-Anweisung in die Liste der SQL-Anweisungen aufgenommen.

Wird versucht ein Objekt im Data Mapping Log zu finden, welches auf eine eingebettete Relation abgebildet wird, so liefert die Suche nach dem Attribut `null` zurück. Wird dies erkannt, wird das verschachtelte Objekt in die aktuelle Relation eingebettet, wie in Abbildung B.5 dargestellt.

Bei der Erstellung von Relationen für Arrays muss zwischen eintypischen und mehrtypischen Arrays unterschieden werden. Allerdings ist dies, wie in Abbildung B.6 dargestellt, erst bei der Behandlung der

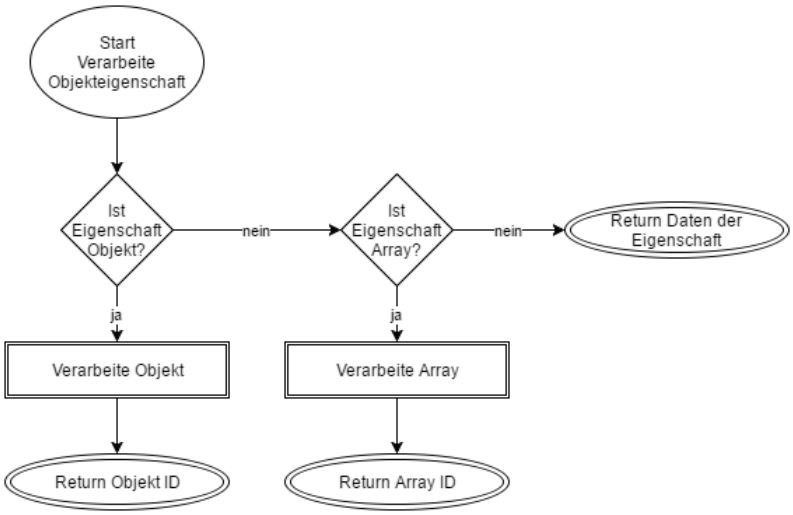

Abbildung 8.4: Verarbeiten einer einzelnen JSON-Objekt-Eigenschaft

Array-Elemente notwendig. Die Abfrage und Speicherung der Array-ID, sowie der Ordnungsnummer, ist in beiden Fällen gleich.
Für Arrays wird bereits nach jedem Element eine `INSERT`-Anweisungen in die Liste der `SQL`-Anweisungen aufgenommen.

Die Verabeitung eines einzelnen Array-Elements unterscheidet sich nur wenig zwischen einem eintypischen (Abbildung B.7) und mehrtypischem Array (Abbildung 8.5). Die Behandlung eines eintypischen Arrays ist in der Lage, Objekte des Arrays direkt einzubetten, während bei mehrtypischen Arrays Objekte nur durch einen Fremdschlüssel in die Relation aufgenommen werden.
Die Behandlung von Arrays und primitiven Daten ist hier in beiden Fällen gleich. Arrays werden nur als Fremdschlüssel aufgenommen, prmitive Daten direkt eingebettet. In beiden Fällen erfolgt sowohl für Objekte als für Arrays ein rekursiver Aufruf zur Verarbeitung.

Die Erstellung einer `INSERT`-Anweisung ist im Wesentlichen eine Konkatenierung der Attributnamen und Werten aus den JSON Dokumenten. Dies wird erweitert durch die notwendige Syntax einer `INSERT`-Anweisung. Das Resultat ist ein String, der zunächst in der Liste von Anweisungen gespeichert wird. Sofern erforderlich wird die Anweisung auf eine `INSERT` mit `SELECT` Anweisung erweitert.

8.3 Konfiguration der Anwendung

Die Datenbestände die in ein relationales Modell überführt werden sollen, können sehr vielfältig sein. Demensprechenend sollte eine Anwendung für diese Transformation zahlreiche Einstellungsmöglichkeiten bieten, um so verschiedenen Szenarios gerecht zu werden.

In meiner Implementierung wird die Konfiguration mittels *YAML Ain't Markup Language* (YAML) vorgenommen. YAML ist ein schlankes Markup-Format, dass sich ideal für Konfigurationen eignet. Durch Einrückung können können Werte gruppiert werden. In der Implementierung werden YAML-Werte generell über eine Aneinanderreihung der Schlüssel abgerufen.
Die beiden Konfigurationsdateien sind einerseits eine `defaults.yaml`, in der alle anpassbaren Werte mit sinnvollen Voreinstellungen erfasst werden. Andererseits kann ein Anwender in einer `config.yaml` diese

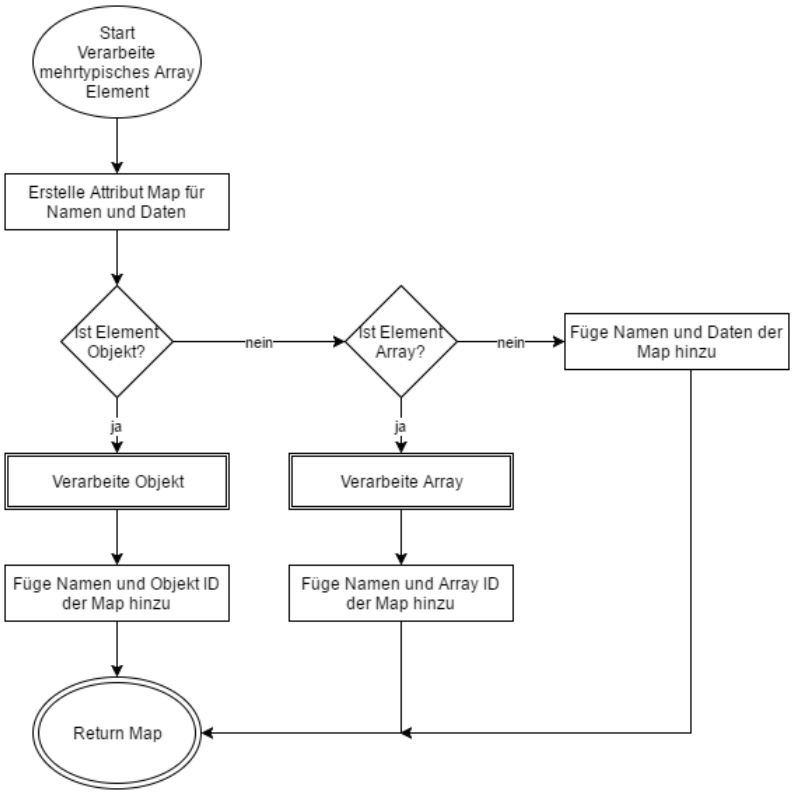

Abbildung 8.5: Verarbeiten eines mehr-typigen JSON-Arrays

Werte überschreiben.

Die Implementierung wurde um eine `Configuration` Klasse erweitert, die einen einfachen Zugriff auf die Einstellungen wie in Quelltext 8.1 ermöglicht. Die `get` Methode sucht hierbei den angeforderten Wert zunächst in der `config.yaml`. Wird dieser dort nicht gefunden, wird der Standardwert aus der `defaults.yaml` geladen.

Quelltext 8.1: Abrufen eines Wertes aus der Konfigurationsdatei

```
1 Configuration config = new Configuration();
2 config.get("transformation.fields.value_field_name")
```

8.4 Laufzeitanalyse

Abschließend seien die Laufzeiten des Transformationsverfahren untersucht. Hierbei ist es interessant zu beobachten, ob es große Unterschiede zwischen den einzelnen Testdatensätzen und/oder Verfahren gibt.

8.4.1 Schematransformation

Die Laufzeit der Transformation eines extrahierten JSON-Schemas in ein relationales Schema ist unabhängig von der Größe eines Datensatzes. Allerdings ist sie stark abhängig von der Komplexität der Dokumente. Tabelle 8.1 verdeutlicht die Laufzeiten für die verschiedenen Testdatensätze. Das Transformationsverfahren wurde insgesamt 100 mal durchgeführt.

Testdatensatz	Durchschnitt (ms)	Minimum (ms)	Maximum (ms)
Soziales Netzwek	1	1	8
Tesla Motors	1	1	8
Personendaten	1	1	5

Tabelle 8.1: Laufzeiten Schematransformation für verschiedene Testdatensätze

8.4.2 Redundanzvermeidung

In Abschnitt 6.1.5 wurden verschiedene Methoden zur Redundanzvermeidung diskutiert. Diese werden im folgenden kurz auf ihre Laufzeiten hin untersucht. Dargestellt sind in Tabelle 8.2 die Zeiten für den Datentransfer. Hierzu wurde der Prozess der Schemaextraktion, -transformation sowie das Transferieren der Daten aus dem Testdatensatz 2 (Abschnitt 8.1.2) 100 Mal durchgeführt.

Wie sich herausstellt besteht für diesen kleinen Datenbestand zwischen keiner Redundanzvermeidung und der Redundanzvermeidung durch ein vorheriges `SELECT` kein gravierender Unterschied in den Laufzeiten des Datentransfers aus der NoSQL- in die MySQL-Datenbank.
Wird beim `INSERT` mit vorherigem `SELECT` jedoch ein Index für die Textattribute erstellt, so beansprucht der Prozess etwas mehr Zeit. Ebenso benötigen die beiden `UNIQUE INDEX` Varianten mehr Zeit als die Variante "keine Redundanzvermeidung".

Redundanzvermeidung	Durchschnitt (s)	Minimum (s)	Maximum (s)
keine	1.8	1.51	2.48
`UNIQUE INDEX`	2.03	1.57	2.53
`UNIQUE INDEX` mit Hash	2.04	1.51	2.72
`INSERT` mit `SELECT`	1.72	1.42	2.07
`INSERT` mit `SELECT` und `INDEX`	2.01	1.71	2.57

Tabelle 8.2: Laufzeiten Redundanzvermeidungsverfahren auf Testdatensatz 2

In Tabelle 8.3 wurde der selbe Test mit dem Testdatensatz 3 (Abschnitt 8.1.3) durchgeführt. Mit dem erheblich größerem Datensatz zeigen sich deutliche Unterschiede in der Laufzeit. Die Variante einer `SELECT` Prüfung vor dem `INSERT` ist sichtbar langsamer als keine Redundanzvermeidung durchzuführen. Bei weiter wachsenden Datenbeständen wird dieser Unterschied aufgrund des quadtratischen Aufwandes der `INSERT` mit `SELECT` Methode schnell größer werden. Eine Einführung eines Indexes mildert diese Verlangsamung deutlich ab und bringt die Laufzeit wieder näher an die Ausgangszeit.
Entscheidet man sich für die Nutzung eines `UNIQUE INDEX`, so ist die Laufzeit ebenfalls im selben Bereich wie ohne jede Optimierung. Wird der `UNIQUE INDEX` jedoch auf einen Hash-Werte anstatt der originalen Textattribute angewendet, verlangsamt sich der Prozess wiederum und ist damit nur noch wenig performanter als die `INSERT` mit `SELECT` Variante.

Redundanzvermeidung	Durchschnitt (s)	Minimum (s)	Maximum (s)
keine	11.21	10.06	13.40
`UNIQUE INDEX`	11.2	9.88	13.5
`UNIQUE INDEX` mit Hash	13.76	12.53	16.25
`INSERT` mit `SELECT`	14.47	13.54	18.17
`INSERT` mit `SELECT` und `INDEX`	11.70	10.65	14.33

Tabelle 8.3: Laufzeiten Redundanzvermeidungsverfahren auf Testdatensatz 3

8.4.3 Einbettung von Relationen

In einem weiteren Testlauf wurden Zeit für die Datentransformation mit und ohne Einbettung von Relationen beobachtet. Zugrunde liegt hier wiederum die 100-fache Ausführung des Transformationsverfahrens auf Testdatensatz 2 in Tabelle 8.4 und Testdatensatz 3 in Tabelle 8.5 Für den Testdatensatz 2 wurde der Schwellwert für die Einbettung auf 3 gesetzt, im Testdatensatz 3 auf 5, was zu einer vollständigen Einbettung und flachen Universalrelation führt.

Einbettung	Durchschnitt (s)	Minimum (s)	Maximum (s)
ohne	2.28	1.86	2.82
mit	1.89	1.59	2.3

Tabelle 8.4: Einfluss der Einbettung auf die Laufzeit der Datentransformation (Testdatensatz 2, Variante `INSERT` mit `SELECT` und Index)

In beiden Fällen wird deutlich, dass der Transfer schneller abläuft, wenn Relationen eingebettet werden. Die Zeitersparnis entsteht dadurch, dass bei einer Einbettung zwei Insert-Anweisungen für zwei Relationen zu einer Anweisung verschmolzen werden. Bei wachsenden Datenbeständen ist dieser Unterschied deutlicher, da hier proportional mehr Insert-Anweisungen eingespart werden.

Einbettung	Durchschnitt (s)	Minimum (s)	Maximum (s)
ohne	8.62	7.80	9.81
mit	4.68	3.93	5.6

Tabelle 8.5: Einfluss der Einbettung auf die Laufzeit der Datentransformation (Testdatensatz 3, Variante `INSERT` mit `SELECT` und Index)

8.4.4 Verschmelzen von Relationen

Ebenfalls kurz untersucht sei der Einfluss des Verschmelzen von Relationen auf die Laufzeit das Datentransfers. Wiederum wurde die gesamte Transformtion 100-mal auf Testdatensatz 2 ausgeführt, die Ergebnisse sind in Tabelle 8.4.

Verschmelzung	Durchschnitt (s)	Minimum (s)	Maximum (s)
ohne	1.68	1.41	1.97
mit	1.59	1.22	1.96

Tabelle 8.6: Einfluss der Verschmelzung von Relationen auf die Laufzeit der Datentransformation (Testdatensatz 2, keine Redundanzoptimierung)

Wie festzustellen ist, besteht kein wesentlicher Unterschied in der Laufzeit zwischen der Transformation ohne Verschmelzen und der Transformtion mit Verschmelzen der Relationen. Dies kann damit begründet werden, dass lediglich die Erstellung einiger Relationen eingespart wird, jedoch keine `INSERT`-Anweisungen. Somit ist die Laufzeit für beide Varianten bei hinreichend großen Datenbeständen nahezu gleich.

Kapitel 9

Fazit

9.1 Fazit

In dieser Arbeit wurde ein Prozess vorgestellt und implementiert, der einen Datenbestand aus NoSQL-Datenbanken in eine relationale Datenbank übertragen kann. Hierbei wurden verschiedene Ansätze der Optimierung vorgestellt und der Implementierung getestet.

Es zeigt sich, dass die Transformation eines JSON-Schemas in ein relationales Schema keinen großen zeitlichen Aufwand im Vergleich zur Schemaextraktion oder zum Datentransfer darstellt.
Beim Datentransfer selbst besteht zwischen den vorgestellten Methoden zur Redundanzvermeidung einer starker Unterschied in der Laufzeit und dem Grad des Datenverlustes bei der Erstellung von `UNIQUE` Indices.

Da die Schemaextraktion die Dokumente einer NoSQL-Datenbank letztendlich nur rein syntaktisch betrachtet, besteht potentiell ein hoher Optimierungsbedarf für ein extrahiertes JSON-Schema (und anschließend relationales Schema). Hier ist oftmals eine manuelle Prüfung durch einen Nutzer nötig, der mit den konkreten Daten vertraut ist. Daher ist es notwendig, einen implementierten Transformationsprozess so anpassbar wie möglich zu halten.

9.2 Ausblick

Weniger betrachtet wurden in dieser Arbeit Optimierungen durch den Einsatz der Metriken, die in der Extraktion gewonnen wurden. Die Nutzung dieser Werte kann ein relationales Schema bedeutend vereinfachen, wenn bspw. Ausreißerdokumente nicht mehr für die Schemaerstellung betrachtet werden.
Auch Schema-Matching oder andere Zielvorgaben für ein Schema wurden wenig behandelt. Diese würden sich allerdings sehr gut eignen, um bestimmte Anwedungsanforderungen besser umzusetzen oder zu erwartenden Anfragearten gerecht zu werden.

In einer weiteren Iteration der Implementierung könnten zudem statt Vorschlägen zur Optrimierung auch direkt mehrere Schemata erstellt werden, die dem Nutzer zur Auswahl gestellt werden. Zusammenen mit einem Bewertungsgrad des jeweiligen Schemas wäre es für Nutzer einfacher das optimale Schema zu finden. Ein solcher Berwertungsgrad könnte auf einem Anforderungsprofil (Relationengröße, Anfragearten, etc.) des Nutzers basieren.

Anhang A

Datensätze

A.1 Beispiel Soziales Netzwek

Quelltext A.1: Einzelnes Dokument aus dem Netzwerkdatensatz

```
{
  "id" : 1,
  "name" : {
    "full_name" : "Felix Beuster",
    "nick_name" : "fixel"
  },
  "date_of_birth" : "1990-12-18",
  "date_of_registration" : 1459517862,
  "verified" : true,
  "follower" : [2, 3],
  "posts" : [
    {
      "id" : 1,
      "timestamp" : 1463251632,
      "content" : "This is my first post. So, hello world?"
    },
    {
      "id" : 2,
      "timestamp" : 1464639223,
      "content" : "Uploaded a new video, check it out! https://youtu.be/
          JDmJsfFzhAU",
      "likes" : 17
    }
  ]
}
```

A.2 Beispiel Tesla Autobestellung

Quelltext A.2: Einzelnes Dokument aus dem Autobestellungsdatensatz

```
{
  "order id" : 26541,
  "status" : "car delivered",
  "model" : {
    "name" : "Model S",
    "year" : 2015
  },
  "fetures" : {
    "color" : "Obsidian Black Metallic",
    "roof" : "All Glass Panoramic Roof",
    "wheels" : "21 Grey Turbine Wheels",
    "interior" : {
      "headliner" : "Black Alcantara Headliner",
      "decor" : "Figured Ash Wood Decor",
      "seats" : "Black Next Generation Seats"
    },
    "variant" : "P90D",
    "ludicrous speed upgrade" : true,
    "carbon fiber spoiler" : true,
    "red breakes" : true
  },
  "options" : {
    "autopilot" : true,
    "premium upgrades" : true,
    "suspension" : true,
    "weather" : false,
    "sound" : true,
    "rear facing seats" : false,
    "charger upgrade" : true
  },
  "price" : 142750,
  "production" : {
    "site" : {
      "address" : {
        "street" : "45500 Fremont Blvd",
        "city" : "Fremont",
        "state" : "CA",
        "zip" : "94538",
        "country" : "United States"
      },
      "production line" : 1
    },
    "status" : "waiting for delivery",
    "production start" : "2015-12-02 9:27am",
    "production finished" : "2016-01-14 9:18am",
    "delivery" : "2016-02-01 9:00am",
    "reports" : [
```

```
      {
        "report id" : 1466,
        "date" : "2016-01-12 12:37pm",
        "related part" : "arm rest",
        "detail" : "wood decor comes off at corner"
      }
    ],
    "assigned workers" : [
      325,
      127,
      [826, 3957],
      1423,
      [171, 17],
      [476, 5224]
    ]
  },
  "owner" : {
    "name" : "Felix Beuster",
    "address" : {
      "street" : "0000 Murray Ave",
      "city" : "Pittsburgh",
      "state" : "PA",
      "zip" : "15217",
      "country" : "United States"
    },
    "orders" : ["742", 26541],
    "complaints" : [
      {
        "complaint id" : 216,
        "related part" : "passenger seat",
        "details" : "1/2 inch sratch in middle of seat"
      }
    ]
  }
}
```

A.3 Beispiel Personendaten

Quelltext A.3: Einzelnes Dokument aus dem Personendatensatz

```
{
  "name": "Macias Bailey",
  "email": "maciasbailey@gonkle.com",
  "address": {
    "street": "950 Locust Avenue",
    "city": "Sharon",
    "state": "Northern Mariana Islands",
    "zip": 4027
  }
}
```

Quelltext A.4: Schema zur Generierung des Personendatensatzes

```
[
  '{{repeat(10000)}}',
  {
    name: '{{firstName()}} {{surname()}}',
    email: '{{email()}}',
    address: {
      street: '{{integer(100, 999)}} {{street()}}',
      city: '{{city()}}',
      state: '{{state()}}',
      zip: '{{integer(100, 10000)}}'
    }
  }
]
```

Anhang B

Programmabläufe

B.1 Schematransformation

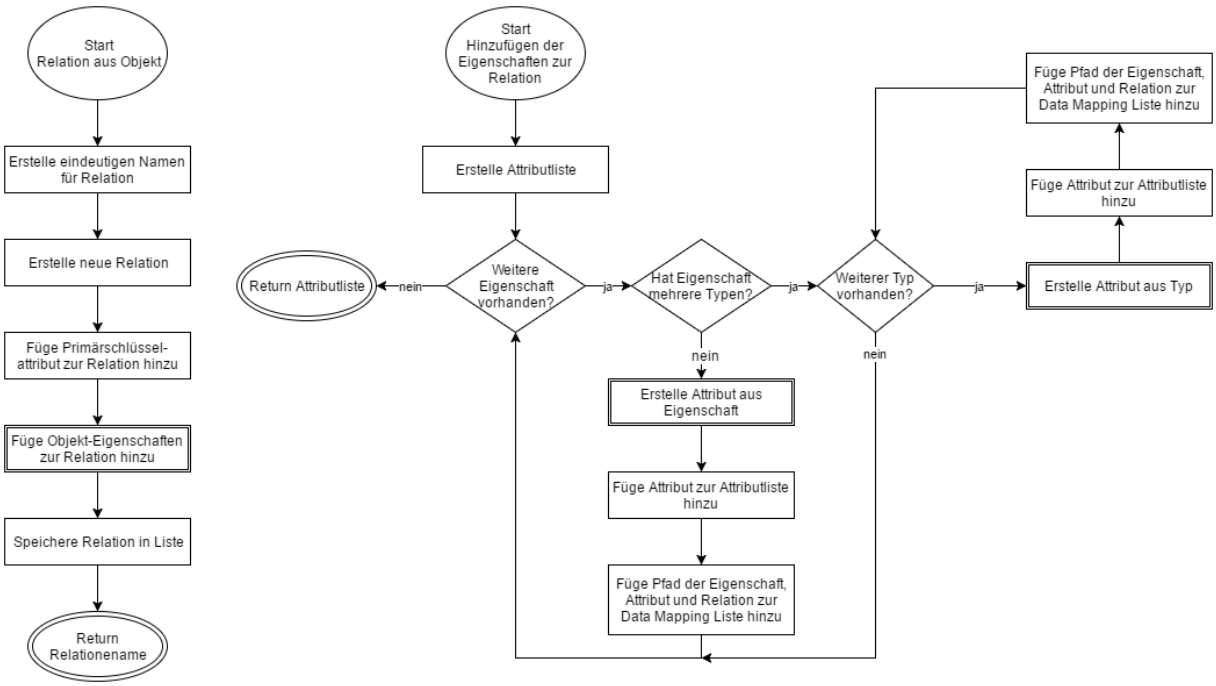

Abbildung B.1: Erstellung einer Relation und Hinzufügen der Objekt-Eigenschaften

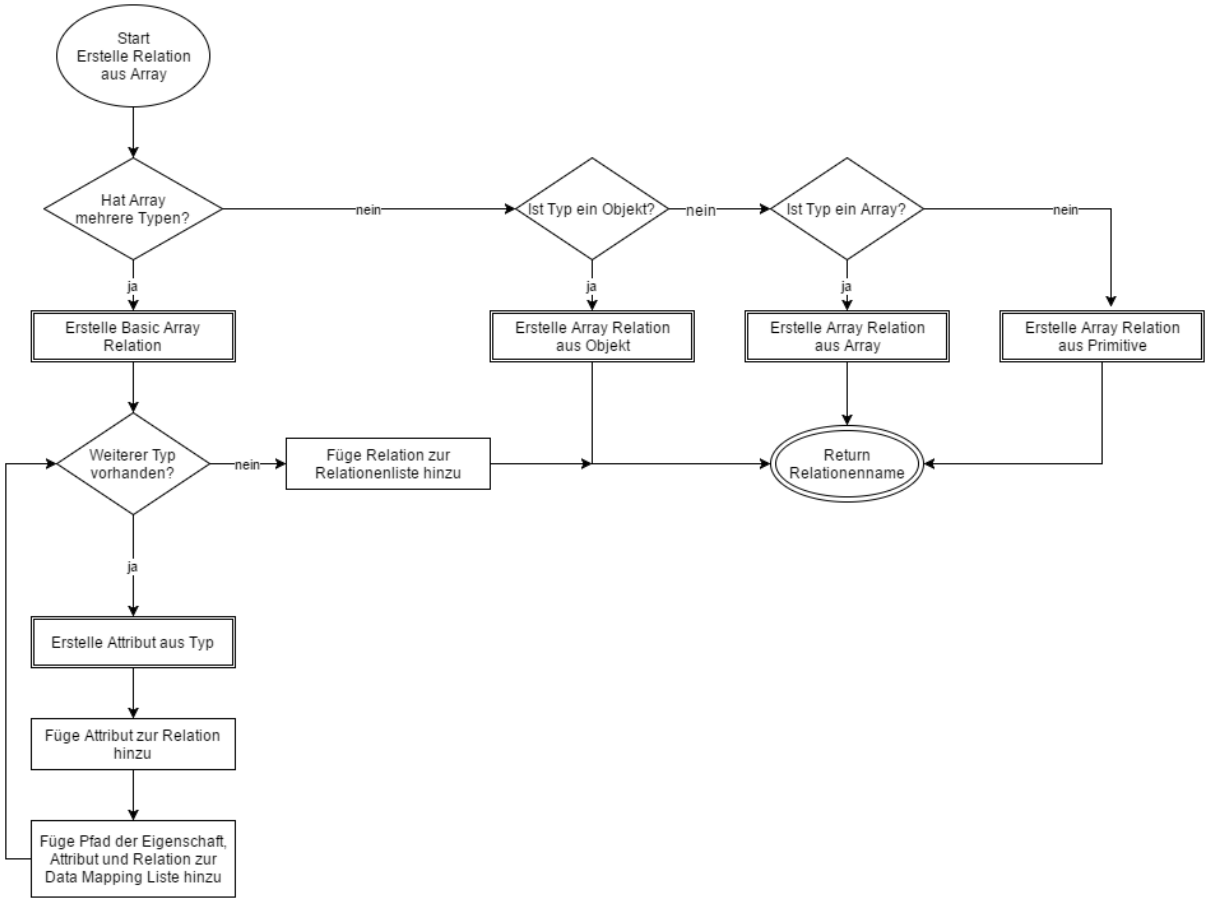

Abbildung B.2: Initialisierung der Schematransformation

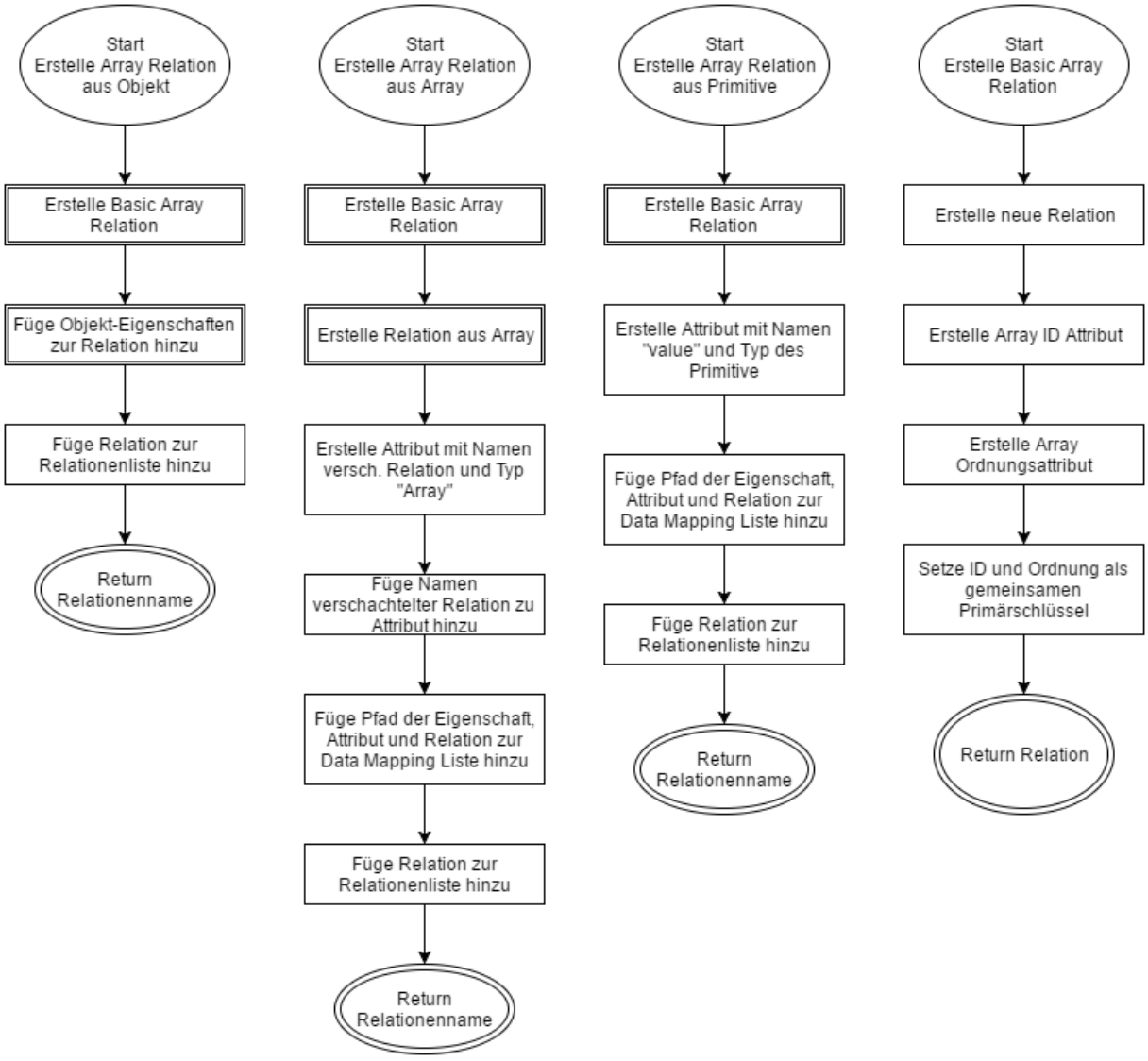

Abbildung B.3: Erstellung verschiedener Array-Relationen

B.2 Datentransformation

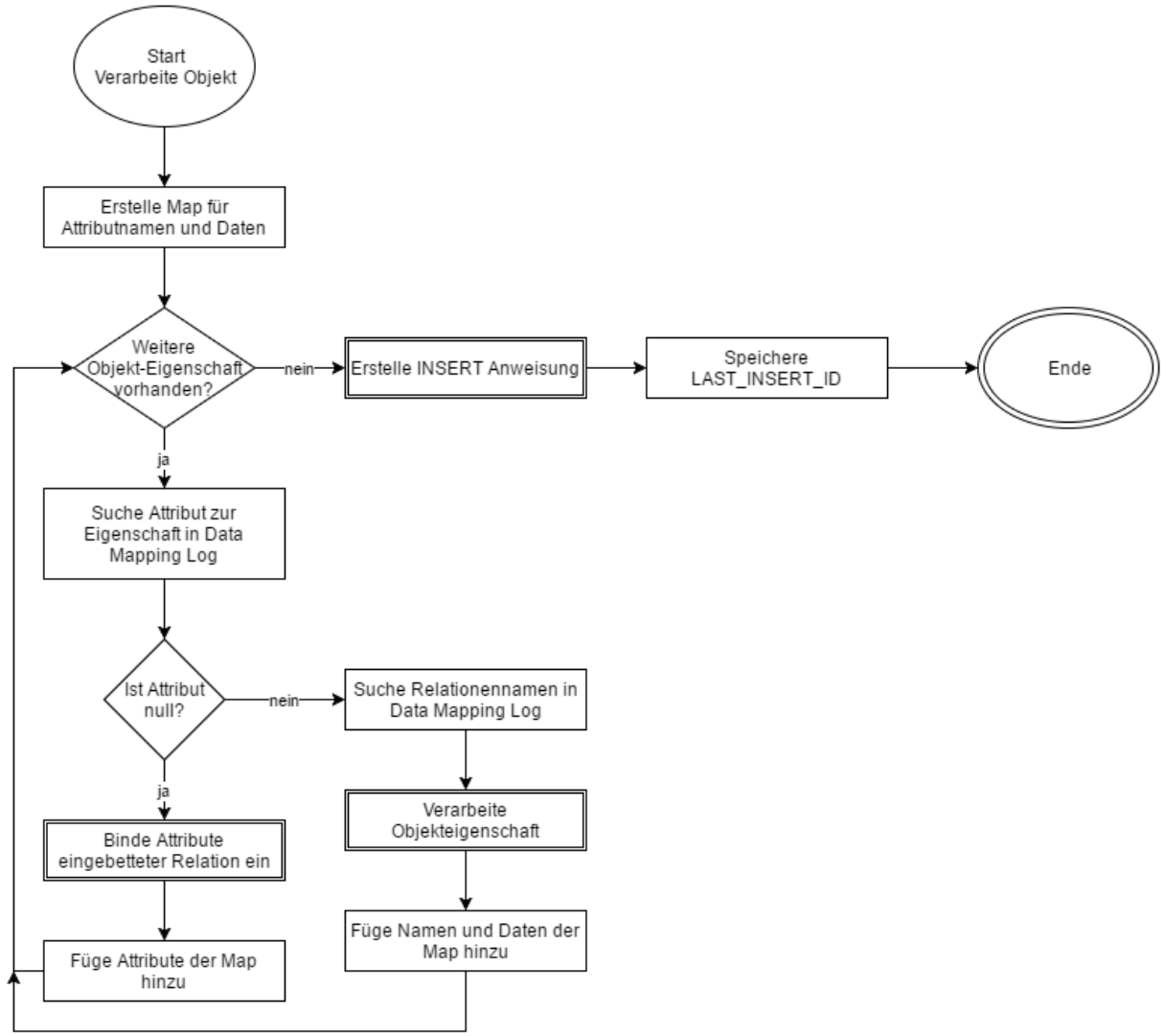

Abbildung B.4: Verarbeiten eines JSON-Objekts

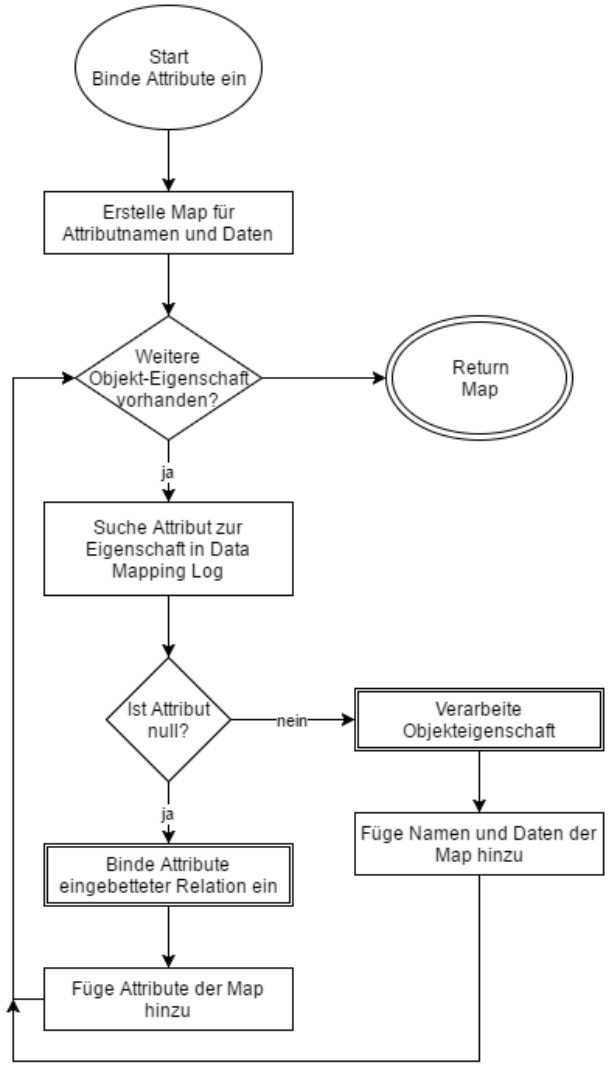

Abbildung B.5: Einbindung der Eigenschaften eines JSON-Objekts

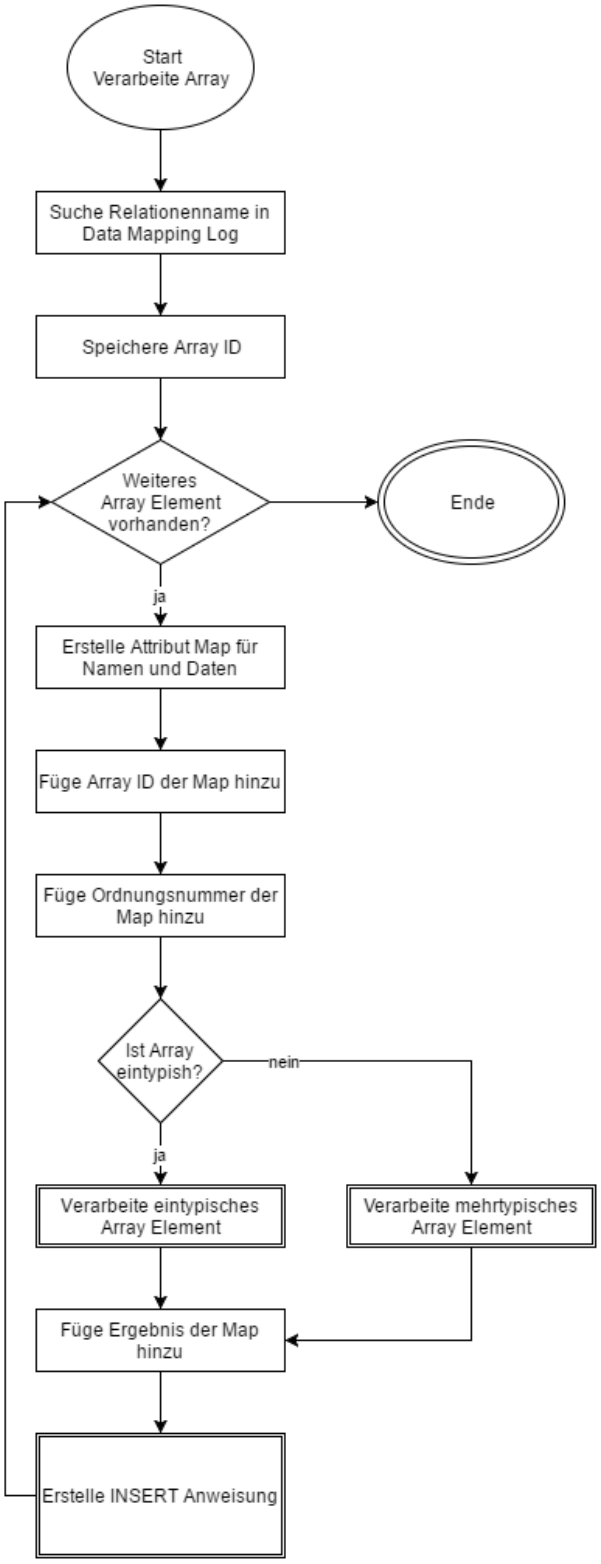

Abbildung B.6: Verarbeiten eines JSON-Arrays

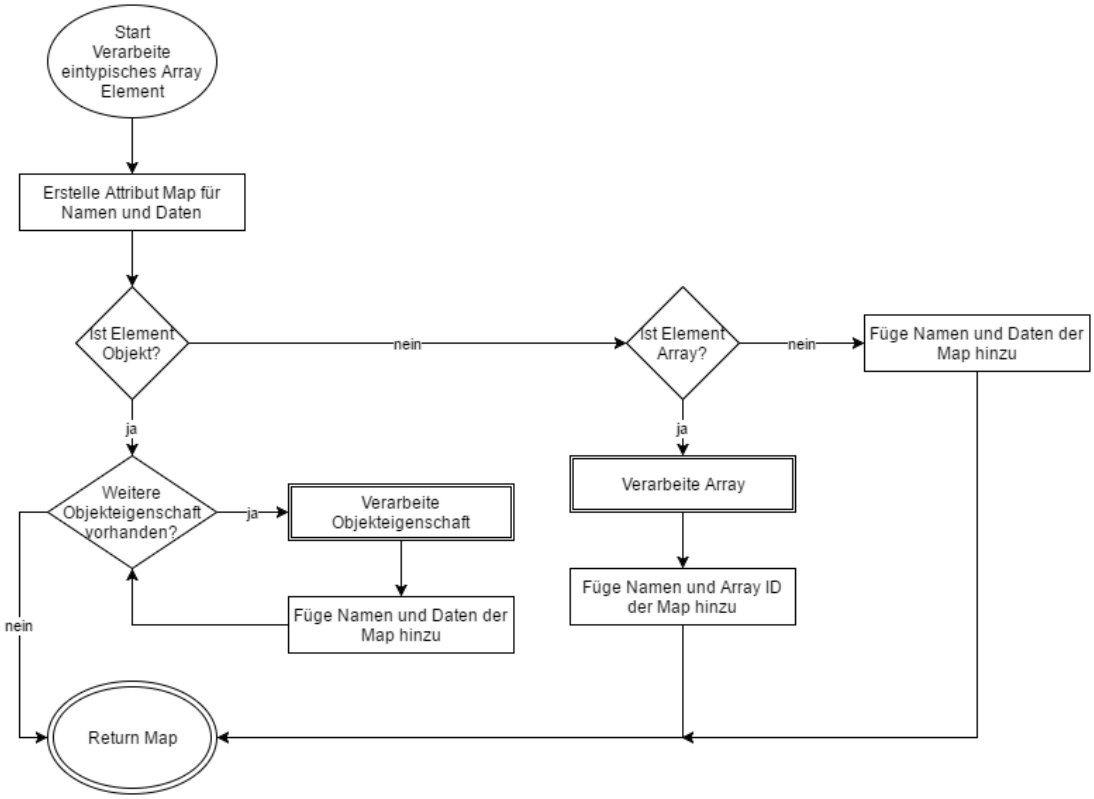

Abbildung B.7: Verarbeiten eines ein-typigen JSON-Array-Elements

Eidesstattliche Erklärung

Ich versichere eidesstattlich durch eigenhändige Unterschrift, dass ich die Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, habe ich als solche kenntlich gemacht. Ich weiß, dass bei Abgabe einer falschen Versicherung die Prüfung als nicht bestanden zu gelten hat.

Felix Beuster

Rostock, den 11.08.2016

Abkürzungsverzeichnis

ANSI	American National Standards Institute
CSV	comma-separated values
DBS	Datenbanksystem
DDL	Data Definition Language
ERM	Entity-Relationship-Modell
ISO	International Standardization Organisation
JSON	JavaScript Object Notation
NoSQL	Not only SQL
ODBS	Objektorientiertes Datenbanksystem
ODL	Object Definiton Language
ODMG	Object Data Management Group
OQL	Object Query Language
SGML	Standard Generalized Markup Language
SQL	Structured Query Language
SSL	Storage Structure Language
TSV	tab-separated values
W3C	World Wide Web Consortium
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

Literaturverzeichnis

[AYDF04] Amer-Yahia, Sihem, Fang Du und Juliana Freire: *A Comprehensive Solution to the XML-to-relational Mapping Problem.* In: *Proceedings of the 6th Annual ACM International Workshop on Web Information and Data Management*, WIDM '04, Seiten 31–38, New York, NY, USA, 2004. ACM.

[Che76] Chen, P. P.: *The entity-relationship model—toward a unified view of data.* ACM Transactions on Database Systems, 1(1):9–36, 1976.

[CLP13] Chasseur, Craig, Yinan Li und Jignesh M Patel: *Enabling JSON Document Stores in Relational Systems.* In: *WebDB*, Band 13, Seiten 14–15, 2013.

[Cod70] Codd, E. F.: *A relational model of data fir large shared data banks.* Communications of the ACM, 13(6):377–387, 1970.

[DA16] DiScala, Michael und Daniel J. Abadi: *Automatic Generation of Normalized Relational Schemas from Nested Key-Value Data.* In: *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, Seiten 295–310, New York, NY, USA, 2016. ACM.

[Dat90] Date, C. J.: *An Introduction Into Database Systems, Volume 1.* Addison Wesley Publishing Company Inc., 5. Auflage, 1990.

[ECM13] ECMA: *The JSON Data Interchange Format Standard*, Oktober 2013. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf` zuletzt abgerufen am 12.05.2016.

[FK99] Florescu, Daniela und Donald Kossmann: *Storing and querying XML data using an RDMBS.* IEEE data engineering bulletin, 22:3, 1999.

[Heu97] Heuer, Andreas: *Objektorientierte Datenbanken Konzepte, Modelle, Standards und Systeme.* Addison Wesley Longman Verlag GmbH, 2. Auflage, 1997.

[IET13] IETF: *JSON Schema: core definitions and terminology*, August 2013. `https://tools.ietf.org/html/draft-zyp-json-schema-04` zuletzt abgerufen am 25.05.2016.

[KSSR15] Klettke, Meike, Uta Störl, Stefanie Scherzinger und OTH Regensburg: *Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores.* In: *BTW*, Band 2105, Seiten 425–444, 2015.

[Kud15] Kudrass, Thomas: *Taschenbuch Datenbanken.* Carl Henser Verlag München, 2. Auflage, 2015.

[Lan16] Langner, Jacob: *Entwicklung und Bewertung von Verfahren zur inkrementellen Schema-Extraktion aus NoSQL-Datenbanken.* Masterarbeit, Universität Rostock, März 2016.

[Mon16] MongoDB, Inc: *mongoimport - MongoDB Manual 3.2*, 2016. `https://docs.mongodb.com/manual/reference/program/mongoimport/#cmdoption--jsonArray` zuletzt abgerufen am 14.06.2016.

[MyS16] MySQL, Oracle Corporation: *MySQL 5.7 Reference Manual :: 15.8.8 Limits on InnoDB Tables*, 2016. `http://dev.mysql.com/doc/refman/5.7/en/innodb-restrictions.html` zuletzt abgerufen am 29.07.2016.

[Oma16] Omanashvili, Vazha: *JSON Generator – Tool for generating random data*, August 2016. `http://www.json-generator.com/` zuletzt abgerufen am 01.08.2016.

[SSH11] Saake, Gunter, Kai-Uwe Sattler und Andreas Heuer: *Datenbanken Implementierungstechniken*. mitp, 3. Auflage, 2011.

[SSH13] Saake, Gunter, Kai-Uwe Sattler und Andreas Heuer: *Datenbanken Konzepte und Sprachen*. mitp, 5. Auflage, 2013.

[STZ+99] Shanmugasundaram, Jayavel, Kristin Tufte, Chun Zhang, Gang He, David J DeWitt und Jeffrey F Naughton: *Relational databases for querying XML documents: Limitations and opportunities*. In: *Proceedings of the 25th International Conference on Very Large Data Bases*, Seiten 302–314. Morgan Kaufmann Publishers Inc., 1999.

[W3C08] W3C: *Extensible Markup Language Standard*, November 2008. `https://www.w3.org/TR/2008/REC-xml-20081126/` zuletzt abgerufen am 16.05.2016.