# Verfahren zur Ermittlung von Integritätsbedingungen in NoSQL-Datenbanken

Masterarbeit
im Fach Informatik

Eingereicht von

**Hannes Awolin**
Matrikel-Nr.: 215206767

**Universität Rostock**
**Institut für Informatik**

Gutachter
Dr.-Ing.habil. Meike Klettke
Dr.-Ing. Holger Meyer

24.08.2017

# Inhaltsverzeichnis

**Abbildungsverzeichnis** **iii**

**Tabellenverzeichnis** **iv**

**1 Einleitung** **1**

**2 Vorbemerkungen und Grundlagen** **3**

2.1 NoSQL-Datenbanksysteme . . . . . 3

2.2 Grundlegende Definitionen . . . . . 7

2.3 Integritätsbedingungen . . . . . 7

2.4 Integritätsbedingungen für dokumentenorientierte NoSQL-Datenbanksysteme . . . . . 10

2.5 Suche nach funktionalen Abhängigkeiten mittels HyFD Algorithmus . . 14

**3 Konzept** **16**

3.1 Ableitbare Merkmale zur Bestimmung von Integritätsbedingungen . . . 16

3.2 Einschränkung des Suchraums bei der Suche nach Inklusionsabhängigkeiten . . . . . 17

3.2.1 Konkretes Überprüfen von Inklusionsabhängigkeiten . . . . . 22

3.2.2 Multithreading . . . . . 23

3.3 Einschränkung des Suchraums bei der Suche nach funktionalen Abhängigkeiten . . . . . 24

3.4 Ermitteln von Primärschlüsseln . . . . . 24

**4 Prototypische Implementierung** **27**

4.1 Anbindung des Datenbanksystems . . . . . 27

4.2 Erkennen funktionaler Abhängigkeiten . . . . . 28

4.2.1 Erkennen funktionaler Abhängigkeiten in Dokument-Kollektionen 32

4.3 Erkennen von Inklusionsabhängigkeiten . . . . . 36

4.3.1 Überprüfen der Teilmengeneigenschaft . . . . . 38

4.3.2 Erkennen von Inklusionsabhängigkeiten in Dokument-Kollektionen . . . 40
4.3.3 Join-Bedingungen auf Basis von Inklusionsabhängigkeiten ermitteln . . . 42

**5 Evaluation** **44**
5.1 Verwendete Daten . . . 44
5.1.1 Anpassung der Daten . . . 45
5.2 Messung der Performance der Algorithmen . . . 46
5.3 Inklusionsabhängigkeiten . . . 47
5.3.1 Basis-Algorithmus . . . 47
5.3.2 Smart-Iterator-Algorithmus . . . 48
5.3.3 Smart-Iterator-Algorithmus mit Einschränkungen durch Datentypen . . . 49
5.3.4 Smart-Iterator-Algorithmus mit Einschränkungen durch Datentypen und statistische Daten über Attributwerte . . . 50
5.3.5 Vergleich der Suchraumeinschränkungen . . . 51

**6 Zusammenfassung** **57**

**Literaturverzeichnis** **59**

# Abbildungsverzeichnis

2.1 Fälle 1-3 (von links nach rechts) für Primärschlüssel in einer Kollektion . 13
2.2 Fremdschlüssel-Primärschlüssel-Beziehung zwischen Dokumenten zweier Kollektionen . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

4.1 Position List Index Datenstruktur . . . . . . . . . . . . . . . . . . . . . . . . 29
4.2 Position List Index Record Datenstruktur . . . . . . . . . . . . . . . . . . 29
4.3 Beispiel eines gefüllten *FDTrees*, Teilausschnitt . . . . . . . . . . . . . . . 31

5.1 Ergebnisse von 5 Testläufen des Basis-Algorithmus . . . . . . . . . . . . 48
5.2 Ergebnisse von 5 Testläufen des Smart-Iterator-Algorithmus . . . . . . . 48
5.3 Ergebnisse von 5 Testläufen des Smart-Iterator-Algorithmus mit Einschränkung des Suchraums durch Datentypen . . . . . . . . . . . . . . . 49
5.4 Ergebnisse von 5 Testläufen des Smart-Iterator-Algorithmus mit Einschränkung des Suchraums durch Datentypen und statistische Informationen über die Daten . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.5 Vergleich der Algorithmen mit unterschiedlichen Suchraumeinschränkungen . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.6 Vergleich des Algorithmus mit unterschiedlich großen Kollektionen als Eingabedaten . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
5.7 Vergleich des Algorithmus mit unterschiedlich großen Kollektionen als Eingabedaten . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

# Tabellenverzeichnis

3.1 Relation $r_1$ . . . . . . . . . . 17
3.2 Relation $r_2$ . . . . . . . . . . 17

5.1 Konkrete Messwerte der Evaluierten Algorithmen . . . . . . . . . . 51
5.2 Messergebnisse des Algorithmus mit unterschiedlich großen Kollektionen bezogen auf die Anzahl der Attribute . . . . . . . . . . 53
5.3 Messergebnisse des Algorithmus mit unterschiedlich großen Kollektionen bezogen auf die Anzahl der Werte pro Attribut . . . . . . . . . . 55

# Kapitel 1

# Einleitung

Momentan wird die Angabe von Integritätsbedingungen von NoSQL-Datenbanksystemen nicht unterstützt, lediglich die Angabe von IDs ist in einigen Systemen möglich. Jedoch beinhalten viele der Daten, die in NoSQL-Datenbanksystemen gespeichert werden, Integritätsbedingungen. Diese Integritätsbedingungen entstehen, wenn die Daten aus Anwendungen heraus oder durch Object-Mapper erzeugt werden.

Auch zeitkritische Szenarien können zur Entstehung von Integritätsbedingungen, wie Inklusionsabhängigkeiten führen, zum Beispiel in dem Daten zwischen Kollektionen dupliziert werden.

Ziel dieser Arbeit ist es, die Integritätsbedingungen aus bestehenden NoSQL-Datenbanken zu ermitteln und aufzuzeigen. Auf Grundlage der gefundenen Integritätsbedingungen sollen im Rahmen dieser Arbeit zu zwei Entity-Typen Vorschläge für Join-Bedingungen zwischen diesen erzeugt werden.

Dazu müssen vor allem Inklusionsabhängigkeiten zwischen beiden erkannt werden. Anhand dieser können die Attribute ausgewählt werden, welche mittels Join-Operation verknüpft werden sollen.

Damit die Arbeit nachvollziehbar bleibt, werden im nachfolgenden Kapitel die vorhandenen Kategorien von NoSQL-Datenbanksystemen vorgestellt und grundlegende Definitionen erläutert. Es wird ebenfalls ein Algorithmus, auf welchen in dieser Arbeit aufgebaut wird, grundlegend erklärt.

Im Anschließenden Kapitel *Konzept* wird erklärt, welche Merkmale der Daten für die Bestimmung von Integritätsbedingungen wichtig sind. Weiterhin wird das grundlegende Vorgehen bei der Überprüfung der verschiedenen Integritätsbedingungen beschrieben.

Danach wird im folgenden Kapitel auf die prototypische Implementierung eingegangen, wobei auch die Anpassung der Algorithmen bezüglich der JSON-spezifischen

Eigenschaften der Daten erklärt wird.

Anschließend erfolgt die Evaluation der Algorithmen durch konkrete Messungen mit verschiedenen Testdatensätzen. Dabei wird überprüft, ob das Verhalten der Algorithmen mit der Erwartungshaltung übereinstimmt.

# Kapitel 2

# Vorbemerkungen und Grundlagen

In diesem Kapitel sollen die verschiedenen Kategorien von NoSQL Datenbanksystemen vorgestellt werden. Weiterhin werden relevante Definitionen eingeführt. Dabei werden zuerst die Definitionen für relationale Datenbanksysteme behandelt, um darauf aufbauend die Definitionen für NoSQL Datenbanken zu entwickeln. Es werden ebenfalls bereits bestehende Algorithmen erläutert, welche die Grundlage für die, in dieser Arbeit entwickelten, Algorithmen bilden.

## 2.1 NoSQL-Datenbanksysteme

Relationale Datenbanksysteme befinden sich bereits seit vielen Jahren im Einsatz. Im Gegensatz dazu sind NoSQL-Datenbanksysteme noch nicht so alt.

Das Datenmodell bei NoSQL-Datenbanksystemen ist nicht relational, da der Fokus auf horizontaler Skalierbarkeit und der Verteilung der Daten liegt. Diese Eigenschaften erfüllen die Anforderungen moderner Web-Applikationen, welche große Mengen unstrukturierter und semi-strukturierter Daten verarbeiten. Aufgrund der horizontalen Skalierbarkeit werden häufig andere Konsistenzmodelle als ACID verwendet. Das CAP-Theorem beweist, dass es in einem verteilten System unmöglich ist, gleichzeitig Verfügbarkeit, Ausfallsicherheit und Konsistenz zu garantieren (Gilbert und Lynch, 2002).

Das ACID-Prinzip bezieht sich auf Transaktionen in einer Datenbank. Dabei steht ACID für die Anfangsbuchstaben der vier Konzepte *atomicity*, *consistency*, *isolation* und *durability* (Sattler, 2015, S.249). Dabei bedeutet *atomicity*, dass Transaktionen entweder erfolgreich ausgeführt werden, oder gar nicht. *Consistency* bedeutet, dass nach einer Transaktion wieder alle Integritätsbedingungen gelten müssen und die Datenbank wieder in einem konsistenten Zustand ist. Bei der parallelen Ausführung von Trans-

aktionen, sollen diese sich nicht durch Nebeneffekte beeinflussen, dieses Prinzip wird *isolation* genannt. *Durability* bedeutet, dass nach dem Ende einer Transaktion, alle Änderungen der Datenbank, auf einem externen Speichermedium dauerhaft geschrieben sind.

Anstelle des ACID-Prinzips, kann auch das BASE-Prinzip verwendet werden (Pritchett, 2008). Dabei steht BASE für die Anfangsbuchstaben von *basically available*, *soft state* und *eventually consistent*. *Eventually consitent* besagt hierbei, dass die Konsistenz vorübergehen verletzt sein darf, jedoch im fehlerfreien Betrieb wieder herstellbar sein muss (Ritter, 2015, S. 425).

NoSQL-Datenbanksysteme werden, basierend auf ihren Eigenschaften, laut (Störl, 2015) wie folgt klassifiziert.

Key-Value-Datenbanksysteme speichern ihre Daten in einer Hashtabelle als Schlüssel-Wert-Paare. Jeder Schlüsselwert darf dabei nur einmal vorkommen. Die Schlüssel beziehen sich je nach System auf einen Teil der Daten oder auf alle Daten. Als Wert werden die eigentlichen Daten gespeichert. Diese können von einem beliebigen Typ sein, z.B. serialisierte Daten oder XML Dokumente. Daher können auch komplett unterschiedliche Daten „nebeneinander"in der selben Datenbank gespeichert werden. Über die Strukturierung der in den Werten gespeicherten Daten, ist dem System in der Regel nichts bekannt. Da das Speichermodell (Schlüssel-Wert-Paare) recht einfach gehalten ist, ist auch die Schnittstelle des Datenbanksystems auf einfache Operationen begrenzt, beispielsweise das Lesen und Schreiben von Schlüssel-Wert-Paaren. Komplexere Operationen, wie das Aktualisieren einzelner Teile eines Wertes (z.B. XML Dokument), werden nicht unterstützt, abgesehen von Operationen wie Map-Reduce, die die zu Grunde liegende Struktur der Daten ausnutzen können.

Eine weitere Klasse von NoSQL-Datenbanksystemen sind die Column-Family-Datenbanksysteme. Diese zeichnen sich durch ihr Datenmodell aus, welches die Daten in einer oder mehreren Tabellen speichert. Die Datensätze werden in Zeilen organisiert. Diese sind durch einen Schlüsssel eindeutig identifizierbar. Der Schlüssel kann aus einem oder mehreren Werten bestehen. Die Daten werden in Spaltenfamilien (column families) gruppiert. Es kann mehrere Versionen eines Datensatzes geben, diese werden durch Zeitstempel unterschieden.

Die Daten, die zu einer Spaltenfamilie gehören, werden auch zusammen gespeichert, somit ist neben der logischen auch eine physische Gruppierung der Daten möglich. Im Gegensatz zu den Spalten, müssen Spaltenfamilien bereits bei der Erzeugung der Tabelle definiert werden. Die spaltenweise Speicherung ist einer der Unterschie-

de zu relationalen Datenbanksystemen und trägt zur horizontalen Skalierbarkeit bei. Ebenso ermöglicht die spaltenweise Speicherung das dynamische Hinzufügen von neuen Spalten. Sollte ein Datensatz hinzugefügt werden, der eine neue Spalte enthält, so wird diese Spalte zur Laufzeit erzeugt.

Nicht alle Tupel einer Column-Family müssen Werte für alle Spalten der Column-Family enthalten. Es ist daher möglich sehr unterschiedlich strukturierte Daten in einer Column-Family abzuspeichern.

Die dritte Klasse von NoSQL Datenbanksystemen bilden die dokumentenorientierten Datenbanksysteme. Deren Datenmodell besteht aus Key-Value-Paaren, wobei die Werte Dokumente beinhalten. Häufig genutzte Formate für die Dokumente sind JSON und BSON.

JSON ist die Abkürzung von *Java Script Object Notation*. Das Format wurde entwickelt um Java Script Objekte textuell zu repräsentieren und diese zwischen Client und Server auszutauschen. Eine JSON-Datei beinhaltet Objekte. JSON-Objekte haben Eigenschaften, die als Name-Wert-Paar gespeichert werden. Diese Eigenschaften besitzen einen Datentyp.

Die folgenden Datentypen sind Bestandteil von JSON: *string*, *number*, *object*, *array*, *boolean* und *null*. Der Datentyp *object* steht für ein weiteres JSON-Objekt. Objekte werden von geschwungenen Klammern eingeschlossen (siehe listing 2.1, Zeilen 1 und 10 und Zeilen 4 und 9).

Der Name der Name-Wert-Paare wird in doppelten Anführungszeichen geschrieben, gefolgt von einem Doppelpunkt und dem dazugehörigen Wert. Mehrere Name-Wert-Paare werden durch Kommas getrennt (siehe listing 2.1, Zeile 2).

JSON ermöglicht die Verwendung von Arrays, dabei besitzen Arrays einen Namem, gefolgt von einem Doppelpunkt, gefolgt von mehrern Werten in eckigen Klammern (siehe listing 2.1, Zeilen 3 und 7). Arrays können, genau wie Objekte, beliebige Datentypen enthalten, somit ist beliebige Schachtelung von Arrays und Objekten möglich.

```
{
    "name": "Peter Parker",
    "subjects": ["math", "english"],
    "homeroom teacher": {
        {
            "name": "John Smith",
            "teaches subjects": ["math", "english", "chemistry
                "]
        }
    }
}
```

**Listing 2.1:** Beispiel einer JSON-Datei

Bei BSON handelt es sich um binär-codierte JSON-Dateien. Ziel des BSON-Formats ist es, weniger Speicherplatz zu verbrauchen als gleichwertige JSON-Dateien. Das Einlesen von BSON-Dateien soll aufgrund der Größe ebenfalls schneller sein, als das von JSON-Dateien.

Dokumentenbasierte Datenbanksysteme werden im Rahmen dieser Arbeit als Grundlage für die prototypische Implementierung der Algorithmen genutzt. Die Einschränkung auf dokumentenzentrierte Datenbanksysteme erfolgte, da sich die Daten in Column-Family-Datenbanken auf dokumentenzentrierte Datenbanken abbilden lassen.

Die fehlenden Schematas für die NoSQL-Daten ermöglichen eine große Schemaflexibilität, was im Bereich der agilen Softwareentwicklung von Vorteil ist. Auch bei dokumentenbasierten Datenbanksystemen besteht die Möglichkeit der horizontalen Skalierung. Durch die Verteilung der Daten auf mehrere Rechnerknoten kann die Performance erhöht werden und Kosten für Hardware können minimiert werden.

Im Vergleich zu relationalen Datenbanksystemen besitzen dokumentenorientierte Datenbanksysteme eine weniger umfangreiche Benutzerschnittstelle.

Wie auch bei Key-Value-Datenbanksystemen ist das Einfügen und Löschen kompletter Datensätze möglich. Darüber hinaus ist jedoch auch ein partielles Update der Dokumente möglich. Verbundoperationen werden in der Regel von den gegenwärtigen Systemen nicht unterstützt. Daher ist auch das Finden von Inklusionsabhängigkeiten im Rahmen dieser Arbeit von besonderem Interesse. Diese bilden die Grundlage für Verbundoperationen.

Beispiele für NoSQL-Datenbanksysteme sind MongoDB, HBase und Cassandra. Dabei handelt es sich bei MongoDB um ein dokumentenzentriertes Datenbanksystem und bei HBase sowie Cassandra um Key-Value-Datenbanksysteme.

## 2.2 Grundlegende Definitionen

Laut (Heuer und Saake, 2000, S. 108 f.) können Attribute, Wertebereiche, Relationenschemata und Relationen wie folgt definiert werden.

**Attribute und Wertebereiche** Sei $\mathcal{U}$ eine nicht-leere, endliche Menge, das *Universum* der Attribute. Ein Element $A \in \mathcal{U}$ heißt *Attribut*. Sei $\mathcal{D} = \{D_1,...,D_m\}$ eine Menge endlicher, nicht-leerer Mengen mit $m \in \mathbb{N}$. Jedes $D_i$ wird *Wertebereich* oder *Domäne* genannt. Es existiert eine total definierte Funktion dom : $\mathcal{U} \rightarrow \mathcal{D}$. dom($A$) heißt der *Wertebereich* von $A$. Ein $w \in dom(A)$ wird *Attributwert* für $A$ genannt.

**Relationenschemata und Relationen** Eine Menge $R \subseteq \mathcal{U}$ heißt *Relationenschema*. Eine *Relation* $r$ über $R = \{A_1,...,A_n\}$ (kurz: $r(R)$) mit $n \in \mathbb{N}$ ist eine endliche Menge von Abbildungen

$$t : R \longrightarrow \bigcup_{i=1}^{m} \mathcal{D}_i$$

die *Tupel* genannt werden, wobei $t(A) \in dom(A)$ gilt. $t(A)$ ist dabei die Restriktion der Abbildung $t$ auf $A \in R$.

## 2.3 Integritätsbedingungen

Integritätsbedingungen sind ein wichtiger Bestandteil von relationalen Datenbanksystemen. Sie werden bei der Modellierung des Datenbankschemas eingeführt, bilden die Grundlage für einen guten Datenbankentwurf und werden für Verbundanfragen genutzt.

Relationale Datenbanksysteme unterstützen üblicherweise Unique- und Not-Null-Bedingungen, sowie die Vergabe von Primär- und Fremdschlüsseln. Diese Bedingungen gelten für Mengen von Attributen.

Die folgenden Definitionen beziehen sich auf relationale Datenbanksysteme.

**Unique-Bedingung** Die Unique-Bedingung erfordert, dass alle Werte einer Menge von Spalten einmalig sind. Sei X eine nicht-leere Menge von Spalten aus der Relation r, $\nexists t, t' \in X, t \neq t' \wedge t[X] = t'[X]$.

Mit der Unique-Bedingung können neben dem Primärschlüssel, welcher ebenfalls die Unique-Bedingung erfüllt, weitere eindeutige Identifikatoren für Tupel geschaffen werden. Es könnte beispielsweise eine Relation Mitarbeiter geben, in der die Personalnummer Primärschlüssel ist. Jedem Mitarbeiter soll auch ein Arbeitsplatz, in Form einer Arbeitsplatznummer, zugeteilt werden. Die Zuordnung der Arbeitsplatznummer ist ebenfalls eindeutig, da in diesem Beispiel jeder Mitarbeiter nur einen Arbeitsplatz hat und an jedem Arbeitsplatz nur ein Mitarbeiter arbeiten kann. Die Arbeitsplatznummer kann nun der Relation Mitarbeiter unter Verwendung der Unique-Bedingung hinzugefügt werden.

**Not-Null-Bedingung** Die Not-Null-Bedingung verbietet das Auftreten von Null-Werten in einer Menge von Spalten. Sei X eine nicht-leere Menge von Spalten, $\nexists t \in X, t[X] = null$.

Somit kann der Null-Wert in allen Spalten auftauchen, die nicht mit *NOT NULL* gekennzeichnet sind. Dabei können Null-Werte als Platzhalter für später zu ergänzende Werte genutzt werden.

Der Null-Wert gehört zu keinem Wertebereich und symbolisiert, dass ein Wert nicht vorhanden oder nicht bekannt ist. Da der Null-Wert nicht als eigentlicher Wert eines Wertebereichs betrachtet wird, muss dieser auch bei der Verwendung von Aggregatfunktionen besonders behandelt werden. Es ist zum Beispiel denkbar, die Null-Werte bei der *count* Funktion nicht mitzuzählen. Bei der Durchschnittsfunktion müssen Null-Werte ebenfalls ignoriert werden.

**Inklusionsabhängigkeit** Inklusionsabhängigkeiten existieren zwischen Mengen von Spalten verschiedener Relationen. Eine Inklusionsabhängigkeit ist eine allgemeinere Form der Fremdschlüssel-Primärschlüssel-Beziehung und soll daher zuerst erläutert werden. Die folgende Definition stammt aus (Heuer und Saake, 2000, S. 262).

Eine Inklusionsabhängigkeit ist für die Relationen $r_1(R_1), r_2(R_2) \in d(S), X \subseteq R_1, Y \subseteq R_2$ ein Ausdruck der Form $R_1[X] \subseteq R_2[Y]$. $d(S)$ genügt der Inklusionsabhängigkeit $R_1[X] \subseteq R_2[Y]$ genau dann, wenn $\pi_X(r_1) \subseteq \pi_Y(r_2)$ gilt.

**Primärschlüssel** Ein Primärschlüssel besteht aus einer Menge von Spalten einer Relation und identifiziert jeden Tupel der Relation eindeutig, somit müssen Primärschlüssel auch immer die Unique-Bedingung erfüllen. Der Primärschlüssel wird während des Datenbankentwurfs vom Anwender festgelegt. Die Werte des Primärschlüssels dürfen keine keine Null-Werte enthalten und müssen eindeutig sein. Jede Relation darf nur einen Primärschlüssel haben.
In (Heuer und Saake, 2000, S. 111) wird der Primärschlüssel wie folgt definiert:

Eine *identifizierende Attributmenge* für ein Relationenschema $R$ ist eine Menge $K := \{B_1,...,B_k\} \subseteq R$, so dass für jede Realtion $r(R)$ gilt:

$$\forall t_1,t_2 \in r[t_1 \neq t_2 \Longrightarrow \exists B \in K : t_1(B) \neq t_2(B)]$$

Ein *Schlüssel* ist eine bezüglich $\subseteq$ minimale, identifizierende Attributmenge, *Primärattribut* nennt man jedes Attribut eines Schlüssels. Ein *Primärschlüssel* ist ein ausgezeichneter Schlüssel.

Wenn Objekte der realen Welt modelliert werden sollen, besitzen viele bereits Attribute, welche als Primärschlüssel verwendet werden können, z.B. die ISBN für Bücher und die Steueridentifikationsnummer für deutsche Staatsbürger.

**Fremdschlüssel** Fremdschlüssel werden genutzt, um Tabellen miteinander zu verbinden. Ein Fremdschlüssel ist eine Attributmenge, die einen Primärschlüssel einer womöglich anderen Relation referenziert (Sauer, 2015, S. 71).

Die Fremdschlüsselbedingung oder *referenzielle Integrität* besagt, dass zu jedem Wert eines Fremdschlüssels einer Relation $r_1$, ein gleicher Wert des Primärschlüssels in einem Tupel der Relation $r_2$, vorhanden sein muss. $r_1$ und $r_2$ können zusammenfallen (Sauer, 2015, S. 72).

Ist der Fremdschlüssel nicht Teil eines Primärschlüssels, so kann er auch Null-Werte enthalten. Generell ist die Fremdschlüssel-Primärschlüssel-Beziehung eine Spezialisierung der Inklusionsabhängigkeit, da auf der rechten Seite des Teilmengenoperators $\subseteq$ nur Primärschlüssel erlaubt sind.

Die Fremd- und Primärschlüssel werden vor allem verwendet um Abhängigkeiten zwischen Entity-Typen im ER-Modell zu modellieren.

**Funktionale Abhängigkeit** Eine funktionale Abhängigkeit in Bezug auf zwei Attributmengen $X$ und $Y$ einer Relation liegt dann vor, wenn der Attributwert von $X$ den

Attributwert von $Y$ festlegt. $Y$ ist funktional abhängig von $X : X \rightarrow Y$ (Sauer, 2015, S. 82).

Primärschlüssel sind ebenfalls Teil einer funktionalen Abhängigkeit, jedes Nicht-Schlüsselattribut ist funktional vom Primärschlüssel abhängig.

Zwei formale Definitionen für funktionale Abhägigkeiten sind in (Heuer und Saake, 2000, S. 231) zu finden:
Eine funktionale Abhängigkeit ist für eine Relation $r(R)$ und Attributmengen $X,Y \subseteq R$ ein Ausdruck der Form $X \rightarrow Y$. r genügt der FD $X \rightarrow Y$ genau dann, wenn

$$|\pi_Y(\sigma_{X=x}(r))| \leq 1 \text{ für alle X-Werte x}$$

gilt. Diese Definition ist gleichwertig mit der informal eingeführten Bedingung:

$$\forall t,t' \in R : t[X] = t'[X] \Rightarrow t[Y] = t'[Y]$$

Aus funktionalen Abhängigkeiten können Primärschlüssel ermittelt werden. Das ist möglich wenn für ein Relationenschema $R$ eine funktionale Abhängigkeit $X \rightarrow R$ gilt und $X$ minimal ist (Heuer und Saake, 2000, S. 232). Somit spielen funktionale Abhängigkeiten eine wichtige Rolle beim Datenbankentwurf.

## 2.4 Integritätsbedingungen für dokumentenorientierte NoSQL-Datenbanksysteme

Da es für dokumentenorientierte NoSQL-Datenbanksysteme kein Schema für die Daten gibt, können sehr unterschiedlich strukturierte Dokumente zusammen gespeichert werden. Kollektionen werden von vielen Systemen als Strukturierungselement für die Daten angeboten. Dabei wird empfohlen, ähnliche Dokumente in einer Kollektion zusammenzufassen. Kollektionen sind somit das Äquivalent zu den Relationen der relationalen Welt.

**Unique-Bedingung** Die Unique-Bedingung kann für dokumentenorientierte NoSQL-Datenbanksysteme analog zu relationalen Datenbanksystemen übernommen werden.

Die Unique-Bedingung ist für eine Attributmenge erfüllt, wenn alle Attributwerte in ihrer Kombination einmalig sind.

Für reale Datensätze kann angenommen werden, dass die Unique-Bedingung nur für wenige Attributmengen zutrifft, z.B. für Primärschlüssel. Sollte eine Kollektion nur

sehr wenige Dokumente enthalten, ist zu erwarten, dass die Unique-Bedingung für viele der Attributmengen zutrifft. Enthält eine Kollektion beispielsweise nur ein Dokument, so erfüllen alle Attributmengen die Unique-Bedingung. Es ist daher sinnvoll, immer auch die Anzahl der Dokumente einer Kollektion zu betrachten.

**Darstellung des Null-Wertes** In relationalen Datenbanken symbolisiert der Null-Wert im allgemeinen einen unbekannten oder nicht vorhandenen Wert. Dieselbe Semantik kann auch für dokumentenorientierte NoSQL-Datenbanksysteme angewandt werden.

Das generelle Nicht-Vorhanden-Sein eines Werte kann über das Weglassen des entsprechenden Attributes für ein Dokument realisiert werden. Daher ist der Einsatz des Null-Wertes hier nicht sinnvoll.

Um einen nicht vorhandenen Wert darzustellen, kann der Null-Wert jedoch verwendet werden, z.B. als Platzhalter für Werte die zu einem späteren Zeitpunkt hinzugefügt werden oder als Platzhalter für nicht vorhandene Werte zwischen verschiedenen Versionen der Daten.

**Not-Null-Bedingung** Analog zu der relationalen Definition, verbietet die Not-Null-Bedingung das Auftreten von Null-Werten in einer Menge von Attributen. Sei X eine nicht-leere Menge von Attributen, $\nexists t \in X, t[X] = null$.

Sollte ein Attribut in einem Dokument nicht vorkommen, so gibt es zwei Möglichkeiten diesen Fall zu behandeln:

1. Kommt ein Attribut nicht in einem der Dokumente vor, so wird dieser Fall ignoriert. Die Not-Null-Bedingung trifft zu, solange keine Null-Werte in den existierenden Attributen vorkommen.

2. Kommt ein Attribut nicht in einem der Dokumente vor, so kann dieser Fall als Auftreten des Null-Wertes interpretiert werden, die Not-Null-Bedingung ist somit nicht erfüllt.

Da Dokumente sehr unterschiedlich strukturiert sein können, was ein häufiges Fehlen von Attributen in den Dokumenten impliziert, ist der erste Fall der relevantere.

Würde der zweite Fall gewählt werden, so wäre die Not-Null-Bedingung nur in sehr seltenen Fällen gültig.

**Inklusionsabhängigkeit** Analog zu relationalen Datenbanken, werden Inklusionsabhängigkeiten in dokumentenorientierten Datenbanksystemen zwischen Kollektionen untersucht.

Da Dokumente mit sehr unterschiedlicher Struktur in einer Kollektion gespeichert werden können, gelten Inklusionsabhängigkeiten unter Umständen nicht für alle in den Kollektionen enthaltenen Dokumente.

**Primärschlüssel** Da unterschiedlich strukturierte Dokumente in einer Kollektion gespeichert werden können, gibt es mehrere Möglichkeiten den Primärschlüssel für diese Dokumente zu definieren.

1. Ein Primärschlüssel identifiziert alle Dokumente einer Kollektion, dabei müssen alle Schlüsselattribute in allen Dokumenten der Kollektion vorkommen. Die Nicht-Schlüssel-Attribute müssen nicht in allen Dokumenten vorhanden sein. In Abbildung 2.1 stellt die linke Abbildung genau diesen Fall dar. Die Attribute $B$ und $C$ bilden den Primärschlüssel für alle Dokumente ($d_1$,...,$d_6$).

2. Ein Primärschlüssel identifiziert alle Dokumente einer Kollektion, welche die selben Attribute haben, dabei müssen die jeweiligen Schlüsselattribute in allen Dokumenten mit den gleichen Attributen vorkommen. Somit gibt es für unterschiedliche Dokumente der selben Kollektion verschiedene Primärschlüssel.

   Der mittlere Teil von Abbildung 2.1 stellt genau diesen Fall dar. Zwischen den Dokumentenmengen $\{d_1,d_2,d_3\}$ und $\{d_4,d_5,d_6\}$ gibt es keine gemeinsamen Attribute. Der Primärschlüssel für die Dokumente $\{d_1,d_2,d_3\}$ besteht aus den Attributen $B$ und $C$. Der Primärschlüssel für die Dokumente $\{d_4,d_5,d_6\}$ besteht aus dem Attribut $E$.

3. Ein Primärschlüssel identifiziert alle Dokumente einer Kollektion, welche die Primärschlüssel-Attribute haben. Somit gibt es für unterschiedliche Dokumente der selben Kollektion verschiedene Primärschlüssel. Mehrere Primärschlüssel sind hierbei für ein Dokument möglich.

   Dieser Fall wird im rechten Teil von Abbildung 2.1 dargestellt. Die Dokumente $\{d_1,d_2,d_3,d_4\}$ haben die Attribute $B$ und $C$ als Primärschlüssel und die Dokumente $\{d_4,d_5,d_6\}$ haben das Attribut $E$ als Primärschlüssel. Dabei werden dem Dokument $d_4$ zwei verschiedene Primärschlüssel zugeordnet.

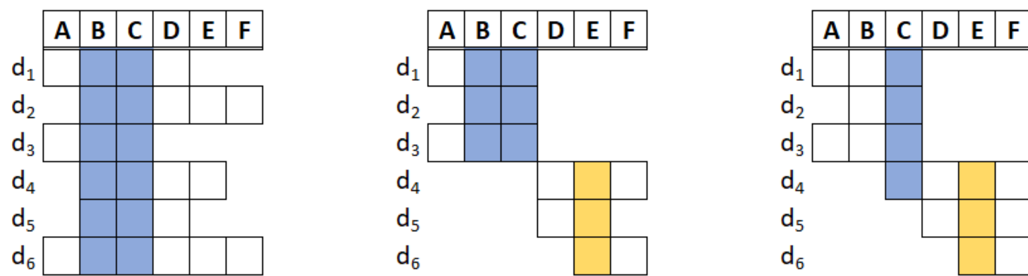


**Abbildung 2.1:** Fälle 1-3 (von links nach rechts) für Primärschlüssel in einer Kollektion

**Fremdschlüssel** Wie auch bei relationalen Daten verweisen Fremdschlüssel auf Primärschlüssel. Da Kollektionen mehrere Primärschlüssel haben können, kann es sein, dass Fremdschlüssel-Primärschlüssel-Beziehungen nicht zwischen allen Dokumenten zweier Kollektionen gelten. In Abbildung 2.2 dargestellt, dass die linke Kollektion

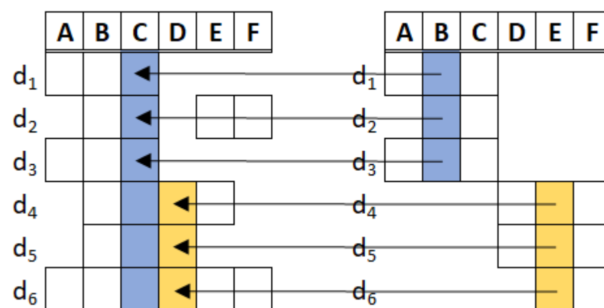


**Abbildung 2.2:** Fremdschlüssel-Primärschlüssel-Beziehung zwischen Dokumenten zweier Kollektionen

mehrere Primärschlüssel hat und es dadurch auch mehrere Fremdschlüssel-Primärschlüssel-Beziehungen geben kann.

**Funktionale Abhängigkeit** Die Definition für funktionale Abhängigkeiten kann von der relationalen Definition übernommen werden.

Durch die Optionalität von Attributen ist es möglich, dass nicht alle für eine Kollektion gefundenen funktionalen Abhängigkeiten auch für alle Dokumente gelten. Jedoch sollte jede funktionale Abhängigkeit für alle Dokumente einer Kollektion gelten, welche alle Attribute der Abhängigkeit beinhalten.

Auch die bereits erwähnte Darstellung des Null-Wertes wirkt sich auch auf die funktionalen Abhängigkeiten aus. Wird der Null-Wert als beliebiger unbekannter Wert aufgefasst, so kann er in keiner funktionalen Abhängigkeit vorkommen.

Somit stellen $null \Rightarrow "value"$ und $"value" \Rightarrow null$ Verletzungen funktionaler Abhängigkeiten dar und können als Gegenbeispiel verwendet werden, ohne dass ein Vergleich mit weiteren Dokumenten erfolgen muss.

## 2.5 Suche nach funktionalen Abhängigkeiten mittels HyFD Algorithmus

Die Grundlage für das Ableiten von funktionalen Abhängigkeiten, bildet der HyFD Algorithmus (Papenbrock und Naumann, 2016).

Dieser Algorithmus ist modular aufgebaut, wobei die Komponenten *Preprocessor*, *Sampler*, *Inductor*, *Validator* und *Guardian* zentrale Aspekte kapseln.

Ziel des Algorithmus ist es sowohl bei vielen Attributen, also auch bei vielen Werten, die Suche nach funktionalen Abhängigkeiten performant durchzuführen.

Die Preprocessor-Komponente ermittelt die Positionen von gleichen Werten für jedes Attribut. Dieser Zusammenhang wird genutzt um Gegenbeispiele für funktionale Abhängigkeiten zu ermitteln. Dabei sind die eigentlichen Werte nicht von Bedeutung, es ist vielmehr wichtig zu wissen, welche Werte gleich und welche ungleich sind.

Dazu werden die Daten in der *position list index* Datenstruktur gespeichert. Die Inverse dieser Datenstruktur wird aus Performance-Gründen ebenfalls berechnet. Die beiden Datenstrukturen werden in Kapitel 4 genauer beschrieben.

Auf den vom *Preprocessor* bereitgestellten Datenstrukturen, sucht die Sampler-Komponente nun nach nicht geltenden funktionalen Abhängigkeiten. Dabei stellt ein Paar von Einträgen, die in einem oder mehr Attributen übereinstimmen, eine Verletzung funktionaler Abhängigkeit dar. Die übereinstimmenden Attribute, können die sich unterscheidenden Attribute nicht funktional bestimmen.

Um alle nicht gültigen funktionalen Abhängigkeiten zu finden, müssen alle Einträge paarweise verglichen werden. Dazu wird der Suchraum eingeschränkt, indem nur ein Teil der Einträge miteinander verglichen wird. Dabei werden die Einträge, die innerhalb eines Fensters fester Größe liegen, miteinander verglichen. Dieses Vorgehen wird genauer in Kapitel 4 erläutert.

Nachdem die Sampler-Komponente eine Liste nicht gültiger funktionaler Abhän-

gigkeiten erzeugt hat, wird diese von der Inductor-Komponente verwendet, um Kandidaten für geltende funktionale Abhängigkeiten abzuleiten.

Es wird angenommen, dass die leere Menge alle Attribute funktional bestimmt. Diese Annahme wird schrittweise, durch die nicht geltenden funktionalen Abhängigkeiten, spezialisiert.

Das Ergebnis dieser Spezialisierungen sind eine Menge von funktionalen Abhängigkeiten die minimal und gültig sind, bezüglich allen Paaren von Einträgen, die in der Sampler-Komponente verglichen wurden.

Diese funktionalen Abhängigkeiten werden nun von der Validator-Komponente auf Gültigkeit, bezüglich des ganzen Datensatzes, überprüft. Die Abhängigkeiten werden in einem Prefix-Baum gespeichert, dieser wird auch als *FDTree* bezeichnet. Dabei sind die Kandidaten die durch die Inductor-Komponente erzeugt wurden, entweder gültige funktionale Abhängigkeiten, oder Verallgemeinerungen von gültigen funktionalen Abhängigkeiten. Sollte ein Kandidat für den kompletten Datensatz nicht gültig sein, so wird er durch alle minimalen, nicht trivialen Spezialisierungen ersetzt. Am Ende erzeugt die Validator-Komponente alle, für den ganzen Datensatz gültigen, funktionalen Abhängigkeiten. Diese sind minimal.

Die letzte Komponente des Algorithmus ist optional und überwacht den Speicherverbrauch. Ziel der Komponente ist, dass Speicherlimit während der Ausführung des Algorithmus nicht zu überschreiten.

# Kapitel 3

# Konzept

Dieses Kapitel behandelt die verschiedenen Konzepte, die zur Bestimmung von Integritätsbedingungen und zur Suchraumeinschränkung verwendet werden.

Es wird mit den Integritätsbedingungen begonnen, die bereits vollständig oder teilweise, während des Einlesens der Daten, ermittelt werden können. Dazu gehören die Unique-Bedingung für einzelne Attribute und die Not-Null-Bedingung.

Anschließend werden die Integritätsbedingungen behandelt, für deren Bestimmung aufwendigere Algorithmen notwendig sind, wie funktionale Abhängigkeiten und Inklusionsabhängigkeiten. Dabei werden auch Konzepte zur Suchraumeinschränkung erläutert.

## 3.1 Ableitbare Merkmale zur Bestimmung von Integritätsbedingungen

Für einzelne Attribute können beim Einlesen der Daten bestimmte Integritätsbedingungen bereits ermittelt werden, dazu gehören die Unique- und die Not-Null-Bedingung.

Ob die Not-Null-Bedingung erfüllt ist, kann für jedes Attribut einzeln bestimmt werden, zum Beispiel, während die Daten eingelesen werden. Sollte ein Attribut Null-Werte enthalten so erfüllt dieses die Bedingung nicht. Jede weitere Attributkombination, bei welcher mindestens ein Attribut die Not-Null-Bedingung nicht erfüllt, erfüllt die Bedingung ebenfalls nicht. Wurde für alle Attribute die Gültigkeit der Not-Null-Bedingung festgestellt, kann daraus ohne viel Rechenaufwand ermittelt werden, ob die Bedingung für weitere Attributkombinationen gültig ist, oder nicht.

Da die Not-Null-Bedingung bei relationalen Datenbanksystemen nur für einzelne Spalten anwendbar ist, ist es naheliegend, zu erwarten, dass auch bei dokumenten-

zentrierten Datenbanken einzelne Attribute die Bedingung erfüllen.

Die Bestimmung der Unique-Bedingung kann ebenfalls beim Einlesen der Daten für einzelne Attribute bestimmt werden. Das Überprüfen der Eigenschaft für Attributkombinationen ist etwas schwieriger als bei der Not-Null-Bedingung. Sei für das Attribut $A$ bereits bekannt, dass die Unique-Bedingung nicht gilt, so muss die Bedingung für die Attributkombination $AB$ trotzdem überprüft werden. Die Kombination mit dem Attribut $B$ kann dafür sorgen, dass die mehrfach in $A$ vorkommenden Werte, in Verbindung mit den Werten aus $B$, einmalig sind.

Ist hingegen bekannt, dass $A$ die Unique-Bedingung erfüllt, so erfüllt auch jede Attributkombination, die $A$ enthält, die Unique-Bedingung, z.B. $ABC$.

Informationen über die Verteilung der Daten ist ebenfalls hilfreich bei der Bestimmung von Integritätsbedingungen. So können gespeicherte Minima und Maxima bei der Suche von Inklusionsabhängigkeiten helfen. Bereits beim Einlesen der Daten können für jedes Attribut Minima und Maxima gespeichert werden.

## 3.2 Einschränkung des Suchraums bei der Suche nach Inklusionsabhängigkeiten

Inklusionsabhängigkeiten bestehen zwischen Mengen von Spalten verschiedener Relationen. Sei $n$ die Anzahl der Attribute in der Relation $r_1$ und $m$ die Anzahl der Attribute in der Relation $r_2$, so gibt es $2^n - 1$ nicht-leere Teilmengen von Attributen für $r_1$ und $2^m - 1$ nicht-leere Teilmengen von Attributen für $r_2$.

| **A** | **B** | **C** |
|---|---|---|
| 1 | a | 6 |
| 2 | ab | 9 |
| 3 | cd | 6 |
| 4 | d | 9 |

**Tabelle 3.1:** Relation $r_1$

| **D** | **E** | **F** |
|---|---|---|
| 6 | a | 6 |
| 9 | ab | 9 |
| 9 | cd | 6 |

**Tabelle 3.2:** Relation $r_2$

Um alle möglichen funktionalen Abhängigkeiten zu ermitteln, werden zuerst alle Attributkombinationen der beiden Relationen gebildet. Anschließend muss überprüft

werden, ob unter den Attributkombinationen mit der gleichen Anzahl an Attributen Teilmengenbeziehungen bestehen. Dabei werden nur Teilmengenbeziehungen zwischen Attributkombinationen verschiedener Relationen betrachtet.

Für die Tabellen 3.1 und 3.2 entstehen die folgenden Attributkombinationen.

$$p_1 = \{\{A\},\{B\},\{C\},\{A,B\},\{A,C\},\{B,C\},\{A,B,C\}\}$$
$$p_2 = \{\{D\},\{E\},\{F\},\{D,E\},\{D,F\},\{E,F\},\{D,E,F\}\}$$

Daraus ergeben sich die folgenden zu überprüfenden Eigenschaften als Kandidaten:

$$\begin{array}{llll}
\{A\} \subseteq \{D\} & \{A,B\} \subseteq \{D,E\} & \{A,B\} \subseteq \{E,D\} & \{A,B,C\} \subseteq \{D,E,F\} \\
\{A\} \subseteq \{E\} & \{A,B\} \subseteq \{D,F\} & \{A,B\} \subseteq \{F,D\} & \{A,B,C\} \subseteq \{D,F,E\} \\
\{A\} \subseteq \{F\} & \{A,B\} \subseteq \{E,F\} & \{A,B\} \subseteq \{F,E\} & \{A,B,C\} \subseteq \{E,D,F\} \\
\{B\} \subseteq \{D\} & \{A,C\} \subseteq \{D,E\} & \{A,C\} \subseteq \{E,D\} & \{A,B,C\} \subseteq \{E,F,D\} \\
\{B\} \subseteq \{E\} & \{A,C\} \subseteq \{D,F\} & \{A,C\} \subseteq \{F,D\} & \{A,B,C\} \subseteq \{F,D,E\} \\
\{B\} \subseteq \{F\} & \{A,C\} \subseteq \{E,F\} & \{A,C\} \subseteq \{F,E\} & \{A,B,C\} \subseteq \{F,E,D\} \\
\{C\} \subseteq \{D\} & \{B,C\} \subseteq \{D,E\} & \{B,C\} \subseteq \{E,D\} & \\
\{C\} \subseteq \{E\} & \{B,C\} \subseteq \{D,F\} & \{B,C\} \subseteq \{F,D\} & \\
\{C\} \subseteq \{F\} & \{B,C\} \subseteq \{E,F\} & \{B,C\} \subseteq \{F,E\} &
\end{array}$$

Da auch die Reihenfolge eine Rolle spielt, müssen nicht nur die einzelnen Teilmengen miteinander verglichen werden, es müssen auf einer Seite der Inklusionsabhängigkeit auch alle Permutationen der Attribute geprüft werden. In dem obigen Beispiel wurden die Attribute der rechten Seite permutiert.

Die Teilmengeneigenschaft muss jedoch auch noch in die andere Richtung überprüft werden. Der Übersichtlichkeit halber, wird hier nur die eine Richtung in den Beispielen dargestellt.

Da hierbei einfach alle Kombinationen überprüft werden, wird der Rechenaufwand bei steigender Anzahl an Attributen schnell sehr groß. Eine Einschränkung des Suchraumes ist daher von Vorteil. Angenommen, dass beide Seiten die selbe Anzahl an Attributen $n$ haben, so gilt für die Anzahl zu prüfender Eigenschaften $a(n)$:

$$\forall n \in \mathbb{N} : a(n) = \sum_{k=1}^{n} \left[ \frac{n!}{(n-k)!} \cdot \frac{n!}{k!(n-k)!} \right]$$

$$
\begin{aligned}
a(2) &= 6 & a(6) &= 13326 \\
a(3) &= 33 & a(7) &= 130921 \\
a(4) &= 208 & a(8) &= 1441728 \\
a(5) &= 1545 & a(9) &= 17572113
\end{aligned}
$$

Die obige Formel beschreibt die Anzahl der zu überprüfenden Eigenschaften für eine Richtung des Teilmengenoperators.

Bei genauerer Betrachtung fällt auf, dass der Bereich für die Suche ebenfalls durch die kleinste Anzahl an Attributen, einer der beteiligten Relationen, eingeschränkt wird. Dabei berechnet sich die Anzahl der zu überprüfenden Eigenschaften $a(n,m)$ wie folgt:

$$\forall n,m \in \mathbb{N}, n \geq m : a(n,m) = \sum_{k=1}^{m} \left[ \frac{n!}{(n-k)!} \cdot \frac{m!}{k!(m-k)!} \right] \tag{3.1}$$

Eine weitere Möglichkeit den Suchraum einzuschränken, ist die Datentypen der Attributmengen in Betracht zu ziehen. Die Teilmengeneigenschaft kann nur zwischen Attributmengen mit den selben Datentypen bestehen.

Mit dieser Einschränkung beschränken sich zu überprüfenden Eigenschaften für das Beispiel auf:

$$
\begin{array}{llll}
\{A\} \subseteq \{D\} & \{A,B\} \subseteq \{D,E\} & \{A,B\} \subseteq \{F,E\} & \{A,B,C\} \subseteq \{D,E,F\} \\
\{A\} \subseteq \{F\} & \{A,C\} \subseteq \{D,F\} & \{A,C\} \subseteq \{F,D\} & \{A,B,C\} \subseteq \{F,E,D\} \\
\{B\} \subseteq \{E\} & \{B,C\} \subseteq \{E,F\} & \{B,C\} \subseteq \{E,D\} & \\
\{C\} \subseteq \{D\} & & & \\
\{C\} \subseteq \{F\} & & &
\end{array}
$$

In relationalen Datenbanken beschränken sich die Inklusionsabhängigkeiten häufig auf die Beziehungen zwischen Fremdschlüssel und Primärschlüssel zwischen zwei Relationen. Wenn die Primärschlüssel der Relationen bekannt sind, könnte die rechte Seite der Inklusionsabhängigkeiten auf die Primärschlüssel eingeschränkt werden. Dadurch können weitere Teilmengenbeziehungen ausgeschlossen werden, bevor es zur eigentlichen Überprüfung kommt.

Gegeben seien die Relationen $r_3(\underline{A},\underline{B},C,D,E,F)$ und $r_4(\underline{G},\underline{H},I,J)$, wobei $AB$ und $GH$ als jeweilige Primärschlüssel ausgezeichnet sind. Wird die Einschränkung auf Primärschlüssel auf die Relationen $r_3$ und $r_4$ angewandt, so reduzieren sich die zu überprüfenden Eigenschaften erheblich. Aus der Formel 3.1 ergeben sich $a(6,4) = 1044$ zu

überprüfende Eigenschaften für jede Richtung des Teilmengenoperators. Durch die Einschränkungen verbleiben für die Richtung $r_3 \subseteq r_4$ mit $a(6,2) = 42$ Eigenschaften und für die Richtung $r_4 \subseteq r_3$ mit $a(4,2) = 20$ Eigenschaften. Somit müssen anstatt von 2088 nur 62 Eigenschaften überprüft werden.

Die Reihenfolge, in der die Abhängigkeiten ausgewertet werden, kann auch einen Einfluss auf den Suchraum haben. Wenn man beginnt, die Abhängigkeiten mit nur einem Attribut auf jeder Seite auszuwerten, können diese Erkenntnisse genutzt werden um weitere Kandidaten auszuschließen.

$$A \nsubseteq D \vee B \nsubseteq E \Rightarrow AB \nsubseteq DE$$

Sollte schon bekannt sein, dass $A \nsubseteq D$ oder $B \nsubseteq E$ erfüllt ist, muss die Eigenschaft $AB \subseteq DE$ nicht mehr überprüft werden, da sie nicht erfüllt sein kann. Genauer gesagt können alle Kandidaten ausgeschlossen werden, deren linke Seite $A$ und deren rechte Seite $D$ enthält. Ebenso können alle Kandidaten ausgeschlossen werden, deren linke Seite $B$ und deren rechte Seite $E$ enthält.

In dem vorherigen Beispiel müssen die Eigenschaften $\{A\} \subseteq \{D\}$, $\{A\} \subseteq \{F\}$, $\{B\} \subseteq \{E\}$, $\{C\} \subseteq \{D\}$, $\{C\} \subseteq \{F\}$ überprüft werden.

$$\begin{array}{lcl}
\{A\} \subseteq \{D\} & \{1,2,3,4\} \subseteq \{6,9\} & \rightarrow false \\
\{A\} \subseteq \{F\} & \{1,2,3,4\} \subseteq \{6,9\} & \rightarrow false \\
\{B\} \subseteq \{E\} & \{a,ab,cd,d,\} \subseteq \{a,ab,cd\} & \rightarrow false \\
\{C\} \subseteq \{D\} & \{6,9\} \subseteq \{6,9\} & \rightarrow true \\
\{C\} \subseteq \{F\} & \{6,9\} \subseteq \{6,9\} & \rightarrow true
\end{array}$$

Die einzig geltenden Inklusionsabhängigkeiten sind somit $\{C\} \subseteq \{D\}$ und $\{C\} \subseteq \{F\}$. Da $\{A\} \nsubseteq \{D\}, \{A\} \nsubseteq \{F\}$ und $\{B\} \nsubseteq \{E\}$ gilt, können alle weiteren Kandidaten ausgeschlossen werden. Die Suche nach Inklusionsabhägigkeiten ist an dieser Stelle beendet.

Ist bereits bekannt, dass $A \subseteq D \wedge B \subseteq E$ gilt (bei anderen Tabellen als 3.1 und 3.2), so muss die Eigenschaft $AB \subseteq DE$ trotzdem überprüft werden. Es sind dann zwar alle Attributwerte aus $A$ in $D$ vertreten und alle aus $B$ in $E$, jedoch kann es trotzdem sein, dass eine Wertekombination von $AB$ nicht in $DE$ enthalten ist. Somit kann $AB \subseteq CD$ nicht erfüllt sein.

Wird mit der Überprüfung der Inklusionsabhängigkeiten zwischen allen einwertigen Attributen begonnen, so können aus den geltenden einwertigen Abhängigkeiten alle noch zu überprüfenden mehrwertigen Abhängigkeiten erzeugt werden. Sind

nicht-geltende einwertige Inklusionsabhängigkeiten, in noch zu überprüfenden mehrwertigen Abhängigkeiten enthalten, so können die mehrwertigen nicht gelten, da schon die enthaltenen einwertigen nicht gültig sind. Somit sind nur noch mehrwertige Abhängigkeiten zu überprüfen, die aus der Kombination bereits gültiger Inklusionsabhängigkeiten hervorgehen.

Angenommen die folgenden Eigenschaften sind erfüllt:

$$A \subseteq D \qquad B \subseteq E \qquad C \subseteq F$$

Aus diesen ergeben sich die folgenden zu überprüfenden Eigenschaften:

$$A \subseteq D \land B \subseteq E \rightarrow AB \subseteq DE$$
$$A \subseteq D \land C \subseteq F \rightarrow AC \subseteq DF$$
$$A \subseteq D \land B \subseteq E \rightarrow BC \subseteq EF$$

Hierbei werden schrittweise Attributkombinationen mit der selben Anzahl an Attributen generiert, wobei die neu generierten Kombinationen genau ein Attribut mehr enthalten als die Ausgangsattribute.

Angenommen $AB \subseteq DE$ und $BC \subseteq EF$ erweisen sich als gültig, so wird im nächsten Schritt daraus $ABC \subseteq DEF$ generiert. Es werden keine neuen Kombinationen mehr generiert, falls keine oder nur noch eine gültige Inklusionsabhängigkeit gefunden wird.

Zu beachten ist dabei, dass sich die zu kombinierenden gültigen Attributkombinationen nur in einem Attribut unterscheiden dürfen.

Dieses Vorgehen ist vergleichbar mit dem Apriori-Schritt des *association rules* Algorithmus (Agrawal u. a., 1994).

Beginnt man mit der Überprüfung der Abhängigkeiten die viele Attribute enthalten, so kann man sich die folgende Eigenschaft zunutze machen:

$$AB \subseteq CD \Rightarrow A \subseteq C \land B \subseteq D$$

Ist also bereits bekannt, dass $AB \subseteq CD$ gilt, so gelten $A \subseteq C$ und $B \subseteq D$ ebenfalls und müssen nicht mehr überprüft werden.

Dieser Ansatz ist jedoch nicht sehr vielversprechend. Am Anfang würden die größtmöglichen Attributkombinationen untersucht werden, dabei ist einerseits der Rechenaufwand größer als bei Inklusionsabhängigkeiten mit nur einem Attribut auf jeder

Seite des Teilmengenoperators und andererseits ist die Wahrscheinlichkeit, dass Inklusionsabhängigkeiten mit so vielen Attributen existieren, gering. Wird jedoch erst nach Abhängigkeiten zwischen einzelnen Attributen gesucht, so ist es sicher, dass dabei Inklusionsabhängigkeiten gefunden werden, sofern diese existieren. Daher ist es sinnvoll mit der Suche von Inklusionsabhängigkeiten zu beginnen, welche zwischen einzelnen Attributen bestehen.

### 3.2.1 Konkretes Überprüfen von Inklusionsabhängigkeiten

Nachdem mit den Methoden, die in Kapitel 3.2 beschrieben wurden, die Menge an Kandidaten für Inklusionsabhängigkeiten eingeschränkt werden kann, muss anschließend trotzdem die Teilmengeneigenschaft für die verbliebenen Kandidaten überprüft werden.

Angenommen die Inklusionsabhängigkeit $A \subseteq B$ ist zu überprüfen, da die Werte in $A$ Teilmenge der Werte in $B$ sein sollen, kann es in $A$ nicht mehr unterschiedliche Werte geben als in $B$. Sollte $A$ mehr unterschiedliche Werte als $B$ enthalten, so kann $A \subseteq B$ nicht gelten. Die Anzahl der unterschiedlichen Werte für ein Attribut, kann ohne Rechenaufwand aus den Datenstrukturen abgelesen werden.

Um die Anzahl unterschiedlicher Werte für Attributkombinationen zu ermitteln, müssen alle Wertekombinationen gebildet werden, daher ist dieser Ansatz für Attributkombinationen nicht von Vorteil.

Neben der Anzahl unterschiedlicher Werte, können auch statistische Daten über die Werte der Attribute helfen. Bei numerischen Werten, kann z.B. der kleinste und der größte Wert abgespeichert werden. Muss dann die Eigenschaft $A \subseteq B$ geprüft werden, so müssen folgende Bedingungen gelten:

$$A.min \geq B.min \qquad A.max \leq B.max$$
$$A.min \in B \qquad A.max \in B$$

Sind alle diese Bedingungen erfüllt, so muss für jeden auftretenden Wert aus $A$ überprüft werden, ob dieser in $B$ enthalten ist. Sind die Werte in einer Art Hash-Tabelle gespeichert, so ist diese Überprüfung effizient realisierbar ($\mathcal{O}(n)$, $n$ sei die Anzahl unterschiedlicher Werte in $A$). Ist einer der Werte aus $A$ nicht in $B$ enthalten, kann die Überprüfung abgebrochen werden, da $A \subseteq B$ nicht gilt.

Da neben dem Datentyp *number* ist in Bezug auf JSON auch noch der Datentyp *string* von Bedeutung. Für Zeichenketten kann ebenfalls ein Minimum und ein Maxi-

mum definiert werden. Dabei soll gelten, dass eine kürzere Zeichenkette kleiner ist als eine längere. Sind zwei Zeichenketten gleich lang, so ist die alphabetische Ordnung entscheidend.

### 3.2.2 Multithreading

Unter Multithreading versteht man im allgemeinen, das Abarbeiten verschiedener Threads zur selben Zeit. Dazu werden Prozessoren mit mehreren Prozessorkernen benötigt. Diese sind beim aktuellen Stand der Technik, jedoch in fast jedem Computer vorhanden.

Die Verwendung mehrerer Threads stellt zwar keine Einschränkung des Suchraums im eigentlichen Sinn dar, jedoch kann dadurch die Performance des Algorithmus verbessert werden.

Da die Laufzeit mit steigender Anzahl an Attributen stark ansteigt (siehe Kapitel 5.3.5), besteht hier großes Potenzial für Multithreading.

Da die Auswertung der Teilmengeneigenschaft nur lesenden Zugriff auf die Datenstruktur, welche die Kollektionen repräsentiert, erfordert, ist es an dieser Stelle nicht nötig die Threads zu synchronisieren.

Es wäre denkbar, mehrere Threads mit einer Warteschlange pro Thread zu erzeugen. Die Warteschlangen können dann im Haupt-Thread mit zu überprüfenden Attributkombinationen gefüllt werden. Damit alle Threads ausgelastet sind, ist auch Loadbalancing zwischen den Warteschlangen denkbar. Die Aufgabe der Arbeiter-Threads besteht dann in der Überprüfung der Teilmengeneigenschaft für die Attributkombination, welche sich in der Warteschlange der Threads befinden.

Da der Iterator für Inklusionsabhängigkeiten erfordert, dass gefundene Abhängigkeiten hinzugefügt werden, ist es erforderlich, dass die Threads schreibenden Zugriff auf diesen haben. Somit muss hier das Prinzip des gegenseitigen Ausschlusses verwendet werden, um den Iterator threadsicher zu machen. Dadurch wird verhindert, dass mehrere Threads gleichzeitig einen kritischen Programmabschnitt betreten und inkonsistente Zustände im Programm auftreten.

Da der Iterator schrittweise Kandidaten für Inklusionsabhängigkeiten mit steigender Anzahl an Attributen erstellt, ist es möglich die Ergebnisse der Arbeiter-Threads schrittweise zu synchronisieren. Immer wenn die Arbeiter-Threads die Ergebnisse für eine Anzahl an Attributen ermittelt haben, werden diese dem Iterator hinzugefügt. Anschließend werden erneut Kandidaten für Inklusionsabhängigkeiten generiert und

den Warteschlangen der Threads hinzugefügt, welche dann wieder überprüft werden. Dieser Vorgang wiederholt sich so lange, bis keine neuen Kandidaten mehr erzeugt werden.

Der Iterator enthält am Ende alle gültigen Inklusionsabhängigkeiten.

## 3.3 Einschränkung des Suchraums bei der Suche nach funktionalen Abhängigkeiten

Der bei der Suche nach funktionalen Abhängigkeiten zugrunde liegende HyFD Algorithmus realisiert bereits mehrere Konzepte der Suchraumeinschränkung, welche in diesem Abschnitt vorgestellt werden.

Die Grundidee des Algorithmus basiert darauf, nicht gültige funktionale Abhängigkeiten, auch nicht-funktionale Abhängigkeiten genannt, zu finden und aus diesen dann die gültigen funktionalen Abhängigkeiten abzuleiten. Die Grundidee auch nicht geltende Abhängigkeiten auszunutzen wurde bereits in (Bell, 1995) erwähnt.

Um alle nicht-funktionalen Abhängigkeiten zu ermitteln, müssen alle Einträge des Datensatzes miteinander verglichen werden, was zu einer quadratischen Komplexität $\mathcal{O}(n^2)$ führt, wobei $n$ die Anzahl der Einträge bzw. Tupel ist.

Die Preprocessor-Komponente gruppiert bereits alle Einträge, die in einem Attribut übereinstimmen, in separaten Listen, auch Cluster genannt. Es werden nur Einträge, die in dem selben Cluster sind, miteinander verglichen. So wird vermieden, dass Einträge verglichen werden, die keine gemeinsamen Werte enthalten, dabei würden keine neuen Erkenntnisse gewonnen.

Eine weitere Einschränkung des Suchraumes wird erreicht, indem nicht alle Einträge eines Clusters miteinander verglichen werden. Stattdessen wird ein Fenster fester Größe über die Einträge in einem Cluster geschoben und nur die Einträge innerhalb des Fensters werden miteinander verglichen.

## 3.4 Ermitteln von Primärschlüsseln

Da durch den HyFD Algorithmus bereits alle minimalen funktionalen Abhängigkeiten ermittelt werden, ist es möglich daraus die Primärschlüssel zu bestimmen.

Es sind diejenigen funktionalen Abhängigkeiten auch Primärschlüssel, welche alle Attribute der Kollektion enthalten.

Eine explizite Suche nach Primärschlüsseln wäre ebenfalls mit dem HyUCC Algorithmus (Papenbrock und Naumann, 2017) möglich. Der Algorithmus ermittelt alle *UCCs* (*unique column combinations*). Dabei enthalten die *UCCs* keinen Eintrag mehrfach und sind somit potentielle Primärschlüssel.

Der HyUCC Algorithmus basiert auf den selben Prinzipien wie der schon vorgestellte HyFD Algorithmus.

Wie auch der HyFD Algorithmus, besteht der HyUCC Algorithmus aus den 5 Komponenten *Preprocessor*, *Sampler*, *Inductor*, *Validator* und *Memory Guardian*.

Für das erkennen von *UCCs* ist ebenfalls wichtig, welche Werte gleich sind. Die Preprocessor-Komponente ist daher dieselbe wie beim HyFD Algorithmus.

Die Sampler Komponente ermittelt nun *non-UCCs*, damit sind nicht gültige *UCCs* gemeint. Dazu werden Einträge bzw. Tupel miteinander verglichen. *Non-UCCs* besitzen dabei gleiche Einträge für mindestens ein Attribut.

Die Gegenbeispiele werden, wie beim HyFD Algorithmus, von der Inductor-Komponente in einen Präfixbaum umgewandelt, der potentielle *UCCs* enthält. Der Präfixbaum wird hier als *UCC tree* bezeichnet. Im Vergleich zum Präfixabum des HyFD Algorithmus ist der *UCC tree* viel kleiner, da es nicht nötig ist, die rechte Seite einer funktionalen Abhängigkeit zu speichern. Laut (Papenbrock und Naumann, 2017) wird der *UCC tree* wie folgt generiert.

Am Anfang enthält der Baum alle individuellen Attribute, unter der Annahme, dass jedes *unique* ist. Mit jeder *non-UCC* wird der Baum weiter angepasst, dabei werden für jede *non-UCC*, die *non-UCC* und alle Teilmengen entfernt, da diese nicht mehr *unique* sein können. Danach werden alle Spezialisierungen der *non-UCC*, durch hinzufügen eines weiteren Attributes, hinzugefügt, da diese noch gültig sein können. Für jede Spezialisierung wird der Baum auf bestehende Teilmengen (Verallgemeinerungen) und Obermengen (Spezialisierungen) durchsucht. Existiert eine Verallgemeinerung, dann ist die erzeugte *UCC* nicht minimal. Wird eine Spezialisierung gefunden, so ist die erzeugte *UCC* ungültig. In beiden Fällen wird die erzeugte *UCC* ignoriert. Ist keiner der beiden Fälle eingetreten, so wird die *UCC* dem Präfixbaum als neuer Kandidat hinzugefügt.

Wie auch beim HyFD Algorithmus, werden die gefundenen Kandidaten durch die Validator-Komponente gegen den kompletten Datensatz validiert. Um eine *UCC* zu validieren, wird die Schnittmenge der *position list indices* gebildet. Dazu werden all cluster miteinander vereinigt (Huhtala u. a., 1999). Das Ergebnis ist wieder ein *postion list index*. Enthält dieser nur Cluster der Größe 1, so ist die *UCC unique*.

Die Validierung ist als Breitensuche realisiert. Dabei entspricht jeder Blattknoten einer *UCC*. Sollte die Validierung positiv ausfallen, so bleibt die *UCC* erhalten. Fällt das Ergebnis negativ aus, wird die *UCC* $X$ entfernt und alle Kandidaten $XA$ mit $A \notin X$, die sowohl minimal (keine Spezialisierung von $XA$ im *UCC tree*) und gültig (keine Verallgemeinerung von $XA$ im *UCC tree*) sind (Papenbrock und Naumann, 2017).

Wie auch beim HyFD Algorithmus, ist die Memory Guardian Komponente optional und für das Speichermanagement des Algorithmus verantwortlich.

Da für die prototypische Implementierung bereits der HyFD Algorithmus gewählt wurde, sollen Primärschlüssel im ersten Schritt aus den funktionalen Abhängigkeiten ermittelt werden. Zu einem späteren Zeitpunkt, können dann die Anpassungen, die am HyFD Algorithmus vorgenommen wurden, auf den HyUCC Algorithmus übertragen werden.

# Kapitel 4

# Prototypische Implementierung

In diesem Kapitel der Arbeit, wird die Implementierung und Anpassung der verwendeten Algorithmen und Datenstrukturen behandelt. Dabei wird zuerst die Implementierung für relationale Daten erklärt. Anschließend folgt eine Erläuterung der Anpassungen der Algorithmen und Datenstrukturen.

Der Begriff relationale Daten bezieht sich in diesem Kapitel auf Daten, die in ein dokumentenorientiertes Datenbanksystem importiert wurden und keine JSON-spezifischen Eigenschaften, wie Arrays und geschachtelte Objekte, enthalten.

Somit ist der Begriff Tupel äquivalent zu einem JSON-Dokument ohne die JSON-spezifischen Eigenschaften.

## 4.1 Anbindung des Datenbanksystems

Als zu Grunde liegendes Datenbanksystem wurde MongoDB verwendet, da es sich laut *DB-Engines Ranking* (solid IT, 2017) um das populärste dokumentenzentrierte Datenbanksystem handelt. Da die Implementierung in Java erfolgt, wird der MongoDB Java-Treiber in der Version 3.4.2 zur Anbindung der Datenbank verwendet.

In der Klasse *MongoDbConnector* wird das der Verbindungsaufbau zur Datenbank, sowie das Auslesen von Kollektionen, gekapselt.

Dabei wird im Konstruktor der Klasse die Verbindung zur MongoDB Datenbank angelegt. Die Methode *getCollection* liefert dann die, über die Parameter spezifizierte, Kollektion zurück.

## 4.2 Erkennen funktionaler Abhängigkeiten

Die Implementierung des Algorithmus zur Erkennung funktionaler Abhängigkeiten basiert auf dem HyFD Algorithmus (Papenbrock und Naumann, 2016). Es handelt sich um eine Reimplementierung auf Basis von Java. Der generelle Ablauf des Algorithmus wurde bereits in Kapitel 2.5 erläutert.

Da der Algorithmus modular aufgebaut ist, folgt die Implementierung ebenfalls dieser Aufteilung. Dabei entstehen die folgenden Komponenten:

- Preprocessor
- Sampler
- Inductor
- Validator

Der originale Algorithmus besitzt noch eine Komponente namens *Memory Guardian*, welche dafür sorgt, dass es während der Ausführung des Algorithmus nicht zu einem Speicherüberlauf kommt. Diese Komponente wird im Rahmen dieser Arbeit nicht betrachtet, da die Implementierung nur prototypisch erfolgen soll.

Die Preprocessor-Komponente importiert die Daten aus einem NoSQL-Datenbanksystem, in diesem Fall MongoDB. Das Datenbanksystem liefert eine Liste von Tupeln bzw. JSON-Dokumenten. Anschließend werden die importierten Daten in zwei Datenstrukturen gespeichert. Dabei trägt die erste Datenstruktur den Namen *position list index*, die zweite Datenstruktur wird als *position list index record* bezeichnet. Um funktionale Abhängigkeiten zu erkennen, ist es wichtig zu wissen, welche Tupel gemeinsame Werte haben. Die Werte selbst sind für das Erkennen der Abhängigkeiten von geringerer Bedeutung. Um zwischen den Tupeln unterscheiden zu können, bekommt jeder Tupel eine Nummer zugewiesen. Diese Nummer ist die Position des Tupels in der Liste der zu importierenden Daten. Fortan wird diese Nummer als *Id* bezeichnet.

Die Datenstruktur *position list index* gruppiert die Werte eines Attributes. Dazu werden die Ids der Tupel, die für ein Attribut den selben Wert haben, in einer Liste gespeichert. Diese Listen werden für jedes Attribut in einem eigenen Wörterbuch gespeichert, wobei der eigentliche Wert als Schlüssel dient. Abbildung 4.1 zeigt beispielhaft wie diese Datenstruktur aussehen kann. Die dargestellte Struktur speichert die Information, dass die Tupel mit den Ids 2, 5 und 6 den selben Wert besitzen, die Zeichenkette *yet another value*.

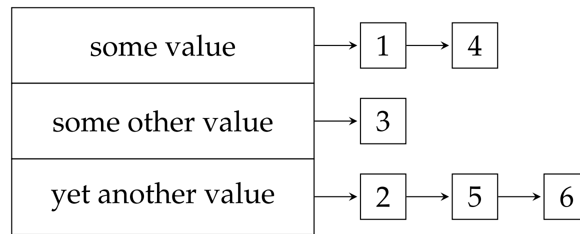


**Abbildung 4.1:** Position List Index Datenstruktur

Jede der gespeicherten Listen stellt somit ein Cluster von Tupeln mit gleichen Werten für das entsprechende Attribut dar. Jedem dieser Cluster wird eine *Cluster-Id* in Form eines Integer-Wertes zugeordnet.

Die zweite Datenstruktur, Position List Index Record, ist ein zweidimensionales Integer-Array. Dieses Array enthält die jeweiligen Cluster-Ids für alle Attribute und Tupel. Abbildung 4.2 zeigt beispielhaft diese Datenstruktur, dabei ist der eigentliche Inhalt des Arrays dick umrandet. Spalte C stellt die Einträge für die Werte aus Abbildung 4.1 dar.

| Id | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 3 | 1 | 3 | 2 | 1 | 1 |
| 3 | 2 | 1 | 2 | 3 | 1 | 2 |
| 4 | 1 | 1 | 1 | 4 | 1 | 2 |
| 5 | 3 | 1 | 3 | 5 | 1 | 3 |
| 6 | 3 | 1 | 3 | 6 | 1 | 4 |

**Abbildung 4.2:** Position List Index Record Datenstruktur

Die Aufgabe der Sampler-Komponente ist es, Verletzungen funktionaler Abhängigkeiten zu finden. Diese nicht gültigen funktionalen Abhängigkeiten, werden dann genutzt, um geltende funktionale Abhängigkeiten abzuleiten. Die vom Preprocessor bereitgestellten Datenstrukturen werden dabei von der Sampler-Komponente genutzt.

Verletzungen von funktionalen Abhängigkeiten können beim Vergleich der Tupel miteinander gefunden werden. Haben zwei Tupel den gleichen Wert für ein Attribut, z.B. das Attribut $A$ und unterschiedliche Werte für das Attribut $B$, so ist die funktionale

Abhängigkeit $A \rightarrow B$ verletzt und es gilt $A \nrightarrow B$. Sind die Werte für $B$ gleich, so kann es sich immer noch um eine funktionale Abhängigkeit handeln und weitere Attribute werden betrachtet.

Um alle Verletzungen funktionaler Abhängigkeiten zu finden, müssen über alle möglichen Paare von Tupeln die jeweiligen Attribute miteinander verglichen werden. Die Komplexität ist hierbei quadratisch über die Anzahl der Tupel. Da der Algorithmus auch für größere Datenmengen performant bleiben soll, ist es sinnvoll die Anzahl an Vergleichen einzuschränken. Da der *Preprocessor* bereits Tupel mit gleichen Werten gruppiert, wird diese Eigenschaft ausgenutzt. Es werden somit nur Tupel miteinander verglichen, bei denen sicher ist, dass sie in einem Attribut übereinstimmen. Um nicht alle diese Tupel miteinander vergleichen zu müssen, wird schrittweise ein Fenster fester Breite über diese Listen geschoben, so dass nur die Tupel miteinander verglichen werden, die sich zeitgleich im Fenster befinden.

Wird eine Verletzung einer funktionalen Abhängigkeit festgestellt, so wird diese in Form eines Bit-Sets gespeichert. Dabei markieren gesetzte Bits gleiche Attributwerte und nicht gesetzte Bits unterschiedliche Attributwerte. Angenommen die Bits des Bit-Sets 1110 stehen für die Attribute $A$-$D$, wobei $A$ durch das *least significant bit* dargestellt wird, so stellt das Bit-Set $B,C,D \nrightarrow A$ dar. Die Bit-Sets werden in einem Hash-Set gespeichert, um Duplikate zu vermeiden.

Anschließend werden die Bit-Sets and die Inductor-Komponente übergeben. Diese benutzt die verletzten funktionalen Abhängigkeiten, um geltenden funktionale Abhängigkeiten abzuleiten.

Die primäre Datenstruktur mit der die Induktor-Komponente arbeitet ist eine Baumstruktur für funktionale Abhängigkeiten, auch *FDTree* genannt. Die Knoten dieser Baumstruktur enthalten zwei Bit-Sets, sowie eine Liste von Kind-Knoten. Die Bit-Sets heißen *marking* und *content*.

Informationen über die linke Seite von funktionalen Abhängigkeiten werden in dem Pfad von der Wurzel zu den Knoten gespeichert. Die rechte Seite der Abhängigkeiten wird durch die Bit-Sets dargestellt. Dabei zeigen die gesetzten Bits in des Bit-Sets *content*, welcher der Kind-Knoten zu einer funktionalen Abhängigkeit führt. Die Gesetzten Bits in dem Bit-Set *marking* entsprechen der rechten Seite einer funktionalen Abhängigkeit. Abbildung 4.3 zeigt einen Ausschnitt aus einem gefüllten *FDTree*. Dabei ist der Wurzelknoten aus Platzgründen nicht mit dargestellt. Der Pfad entlang der grünen Pfeile, könnte beispielsweise die funktionale Abhängigkeit $BDE \rightarrow AC$ darstellen.

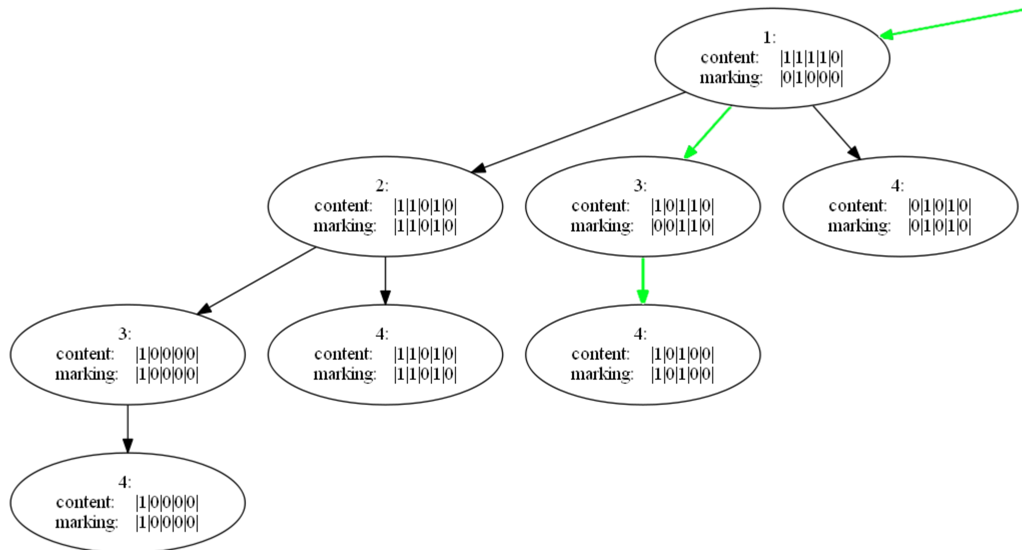


**Abbildung 4.3:** Beispiel eines gefüllten *FDTrees*, Teilausschnitt

Um den *FDTree* zu füllen, werden zuerst alle nicht-funktionalen Abhängigkeiten absteigend nach ihrer Kardinalität sortiert. Somit werden nicht-funktionale Abhängigkeiten, auf deren linker Seite viele Attribute vorhanden sind, zuerst bearbeitet. Dadurch wird sichergestellt, dass der Baum am Anfang wesentlich langsamer wächst, was die Kosten bei der Suche nach Verallgemeinerungen von Abhängigkeiten im Baum reduziert.

Zu Beginn wird der *FDTree* mit der allgemeinsten funktonalen Abhängigkeit $\emptyset \rightarrow R$ initialisiert, wobei $R$ alle Attribute darstellt.

Aus jedem Bit-Set der Sampler-Komponente enstehen so viele nicht-funktionale Abhängigkeiten, wie nicht gesetzte Bits vorhanden sind. Angenommen das Bit-Set 1100 wurde von der Sampler-Komponente erzeugt, daraus entstehen nun die folgenden Bit-Sets als Repräsentation der nicht-funktionalen Abhängigkeiten:

$$1100 \nrightarrow 0010 \qquad AB \nrightarrow C$$

$$1100 \nrightarrow 0001 \qquad AB \nrightarrow D$$

Mit jeder dieser Abhängigkeiten wird nun der *FDTree* spezialisiert. Dabei werden zuerst alle Verallgemeinerungen der Abhängigkeit aus dem *FDTree* entfernt, da diese nicht gültig sein können. Ist beispielsweise $AB \nrightarrow C$ bekannt, so können die allgemeineren Abhängigkeiten $A \rightarrow C$ und $B \rightarrow C$ auch nicht gelten.

Anschließend werden alle Spezialisierungen der Abhängigkeit dem Baum hinzugefügt, da diese noch gültig sein können. Wenn diese hinzugefügt werden, wird sichergestellt, dass diese minimal sind.

Nachdem der *FDTree* mit allen nicht-funktionalen Abhängigkeiten spezialisiert wurde, enthält dieser die Menge aller minimalen und gültigen funktionalen Abhängigkeiten bezüglich dieser.

Die Validator-Komponente überprüft anschließend alle in dem *FDTree* gespeicherten funktionalen Abhängigkeiten nochmals auf ihre Gültigkeit, indem sie sie gegen alle vorhandenen Datensätze validiert.

Die Validator-Komponente wurde im Rahmen dieser Arbeit nicht implementiert. Im folgenden Abschnitt wird konzeptuelle behandelt, wie der HyFD Algorithmus angepasst werden muss, damit die Suche nach funktionalen Abhängigkeiten in Kollektionen von JSON-Dokumenten möglich ist.

### 4.2.1 Erkennen funktionaler Abhängigkeiten in Dokument-Kollektionen

Aufgrund der Merkmale von JSON-Dokumenten im Vergleich zu relationalen Daten, müssen die Algorithmen angepasst werden.

Zuerst wird der Algorithmus so angepasst, dass die Optionalität von JSON-Attributen kein Problem mehr darstellt. Dazu wird eine Komponente genutzt, die eine Schemaextraktion für eine gegebene Kollektion von JSON-Dokumenten durchführt (Klettke u. a., 2015).

Die Schemaextraktionskomponente liefert Daten über extrahierte Schemata in Form eines JSON-Dokumentes zurück. Dabei sind Informationen über Datentyp und Häufigkeit des Auftretens der einzelnen Properties in dem Dokument enthalten. Listing 4.1 zeigt einen Ausschnitt des Ergebnisdokuments, dabei ist erkennbar, dass das Attribut *surgery* nicht in allen Dokumenten enthalten ist.

```
{
 "type" :"object",
 "title" :"horse_divers",
 "properties" :{
  "ts" :{
   "type" :"integer",
   "description" :"Occurence: 100%"
  },
  "surgery" :{
   "type" :"integer",
   "description" :"Occurence: 99%"
  },
  ...
```

**Listing 4.1:** Beispiel für das Ergebnis der Schemaextraktion

Eine Besonderheit des JSON-Formates, ist die Schachtelung von Objekten ineinander. Um damit umzugehen gibt es verschiedene Ansätze.

Es wäre möglich alle geschachtelten Objekte zu entfalten. Dabei würden alle Properties in einem Dokument auf eine Ebene gezogen werden. Aus dem Dokument in Listing 4.1 würde das Dokument in Listing 4.2 werden.

```
{
 "type" :"object",
 "title" :"horse_divers",
 "properties.ts.type" :"integer",
 "properties.ts.description" :"Occurence: 100%",
 "properties.surgery.type" :"integer",
 "properties.surgery.description" :"Occurence: 99%",
 ...
}
```

**Listing 4.2:** Beispiel für das Entfalten der JSON-Objekte

Die Namen der Properties setzen sich dann aus dem Pfad im originalen Dokument zusammen. In diesem Beispiel wurde der Punkt als Trennzeichen zwischen den Pfa-

debenen verwendet.

Diese Transformation der Daten könnte im Datenbanksystem erfolgen und würde bedeuten, dass für die Schachtelung von Objekten keine Anpassung der Algorithmen nötig ist.

Eine weitere Möglichkeit, wäre es, nur nach Abhänngigkeiten auf der gleichen Ebene von Properties zu suchen.

Die Optionalität von Attributen erfordert auch Anpassungen der Datenstrukturen des HyFD Algorithmus. Da die *position list index* Datenstruktur auf einer Hash-Tabelle basiert, spielen fehlende Werte hier erstmal keine Rolle. Es ist jedoch zu beachten, dass nicht für jedes Attribut in dem entsprechenden *position list index* alle Dokumente in Form ihrer Id vorkommen. Besitzt ein Dokument ein Attribut nicht, so fehlt die Id des Dokuments im entsprechenden *position list index*.

Bei der Berechnung des *position list index record* ist zu beachten, dass nicht jeder Eintrag in dem zweidimensionalen Integer-Array auf eine Cluster-Id verweist. Anstelle der Cluster-Id kann hier der Wert -1 verwendet werden, um zu symbolisieren, dass ein Attribut in einem Dokument nicht vorhanden ist.

Die Sampler-Komponente kann weiterhin Eintrgäge vergleichen, die in den selben Clustern gespeichert sind, um nicht-funktionale Abhängigkeiten zu finden. Da nur Ids in einem Cluster gespeichert sind, deren Attribut in den Dokumenten auch existiert, kann der Vergleich für das Attribut erfolgen. Bei dem Vergleich von zwei Einträgen wird festgestellt ob die Werte des selben Attributes gleich oder ungleich sind. Es tritt jedoch, durch die Optionalität bedingt, noch ein weiterer Fall auf, die Werte existieren nicht.

Bisher wurden gleiche Werte mit einem gesetzten Bit in einem Bit-Set gespeichert. Ungleiche Werte wurden durch nicht gesetzte Bits symbolisiert.

Sollte bei einem Vergleich jedoch ein Wert nicht vorhanden sein, so folgt daraus weder die Verletzung, noch die Gültigkeit einer funktionalen Abhängigkeit. Im Bit-Set würde ein dritter Zustand gebraucht, welcher den unbekannten Zustand symbolisiert.

Es wäre möglich ein zweites Bit-Set einzuführen, welches diesen ungewissen Zustand speichert. Eine andere Möglichkeit, wäre es, eine neue Klasse *TriBitSet* als Unterklasse von *BitSet* zu erzeugen und diese um die benötigte Funktionalität zu erweitern. Das hätte den Vorteil, dass der bisherige Code weiterhin ohne Änderungen funktionieren würde.

Angenommen die Klasse TriBitSet wird verwendet und ist in der Lage den unbekannten Zustand ebenfalls abzuspeichern, so resultieren daraus die nächsten Ände-

rungen des Algorithmus in der Inductor-Komponente.

Die Inductor-Komponente erzeugt aus den nicht-funktionalen Abhängigkeiten Kandidaten für die funktionalen Abhängigkeiten. Dazu wird ein Präfixbaum, auch *FDTree* verwendet, um diese zu speichern. Dabei wird der *FDTree* mit jedem TriBitSet spezialisiert. Wie bei dem originalen Algorithmus entstehen jetzt aus jeden TriBitSet mehrere nicht-funktionale Abhängigkeiten.

Vorausgesetzt ein TriBitSet ist gegeben durch 11?00, wobei das Fragezeichen den einen nicht vorhandenen Wert symbolisiert, so entstehen daraus die folgenden nicht-funktionalen Abhängigkeiten:

$$11?00 \nrightarrow 00?10 \qquad AB \nrightarrow D$$
$$11?00 \nrightarrow 00?01 \qquad AB \nrightarrow E$$

Die unbekannten Werte werden bei der Erzeugung der nicht-funktionalen Abhängigkeiten für die Spezialisierung des *FDTree* einfach ignoriert. Die Spezialisierung des *FDTree* verläuft wie im originalen Algorithmus.

Bei der Validierung durch die Validator-Komponente muss im Anschluss beachtet werden, dass nur die Dokumente die funktionalen Abhängigkeiten erfüllen können, die auch alle Attribute besitzen, welche in der Abhängigkeit vorkommen.

Nach diesen Änderungen ist zu erwarten, dass der Algorithmus auch für Kollektionen mit unterschiedlichen Dokumenten funktionale Abhängigkeiten finden kann. Die gefundenen funktionalen Abhängigkeiten gelten nur für die Dokumente der Kollektion, die alle Attribute der Abhängigkeit besitzen.

Eine weiteres offenes Problem ist die Behandlung von Arrays bei der suche nach funktionalen Abhängigkeiten.

Dabei ist es durchaus denkbar, dass normale Properties den Inhalt eines Arrays funktional bestimmen.

Es könnte beispielsweise eine Patientenakte als JSON-Dokument modelliert worden sein. Ein Ausschnitt dieses Dokuments ist in Listing 4.3 zu dargestellt.

```
{
  "case" :1,
  "temperature" :39.9,
  "symptoms" :["fever","headache","dizzyness"],
  ...
}
```

**Listing 4.3:** Beispiel für eine Patientenakte in Form eines JSON-Dokuments

Die Property *temperature* stellt die Körpertemperatur dar und die Property *symptoms* die Symptome. Es ist durchaus logisch, dass die erhöhte Körpertemperatur einen Einfluss auf die Symptome hat.

Da die Array Elemente, durch ihre Position im Array eine Ordnung haben, stellt sich die Frage, ob dies Auswirkungen auf die funktionalen Abhängigkeiten hat. Möglicherweise wird für Arrays eine neue Definition der funktionalen Abhängigkeit benötigt.

## 4.3 Erkennen von Inklusionsabhängigkeiten

Im Gegensatz zu funktionalen Abhängigkeiten, sind bei der Suche nach Inklusionsabhängigkeiten die tatsächlichen Werte der Attribute von Bedeutung.

Diese Attribute werden in der Klasse *DocumentCollectionWrapper* gespeichert.

Der grundlegende Ansatz besteht darin, zwischen allen möglichen Attributkombination zweier Kollektionen die Teilmengeneigenschaft zu überprüfen. Sind alle Wertekombinationen der einen Kollektion, in der anderen enthalten, so besteht eine Inklusionsabhängigkeit.

Listing 4.4 zeigt den Algorithmus für dich Suche nach Inklusionsabhängigkeiten, welcher den vorher als Smart-Iterator, jetzt als *InclusionDependencyIterator* bezeichneten Iterator verwendet.

Wie in Zeile 1 zu erkennen ist, erfolgt die Rückgabe der Inklusionsabhängigkeiten als Liste von Integer-Array Paaren. Die Integer-Arrays enthalten die Indizes der Attribute zwischen denen eine Inklusionsabhängigkeit besteht und repräsentieren somit Kandidaten für mögliche Join-Bedingungen.

Weiterhin bekommt die Methode die zu überprüfenden Kollektionen, in Form von *DocumentCollectionWrapper* Instanzen, sowie zwei boolesche Variablen, als Parameter übergeben.

Die Variablen *dcw1* und *dcw2* enthalten somit die Kollektionen die auf Inklusionsabhängigkeiten überprüft werden sollen. Die boolsche Variable *checkTypes* entscheidet ob auch Vergleiche zwischen den Datentypen stattfinden oder nicht. Ob statistische Daten wie Minimum und Maximum der Attribute genutzt werden, wird durch die Variable *checkMinMax* festgelegt.

```
1  public static List<Pair<int[],int[]>> findInclusionDependenciesSmart(
       DocumentCollectionWrapper dcw1, DocumentCollectionWrapper dcw2,
       boolean checkTypes, boolean checkMinMax){
2   int min = Math.min(dcw1.getNumberOfAttributes(),dcw2.
        getNumberOfAttributes());
3   List<Pair<int[],int[]>> dependencies = new ArrayList<>();
4   for(int i=0; i<dcw1.getNumberOfAttributes(); i++){
5    for(int j =0; j<dcw2.getNumberOfAttributes();j++){
6     int[] left = {i};
7     int[] right = {j};
8     if(dcw1.isSubsetOf(dcw2, left , right, checkTypes, checkMinMax)){
9      dependencies.add(new Pair<>(left, right));
10    }
11   }
12  }
13  InclusionDependencyIterator idi = new InclusionDependencyIterator(
        dependencies,min);
14  while (idi.hasNext()){
15   Pair<int[],int[]> p = idi.next();
16   if (dcw1.isSubsetOf(dcw2,p.getFirst(),p.getSecond(),checkTypes,
         checkMinMax))
17    idi.addKnownDependency(new Pair<int[], int[]>(p));
18  }
19  return idi.getDependencies();
20 }
```

**Listing 4.4:** Algorithmus für dich Suche nach Inklusionsabhängigkeiten

Als erstes wird die kleinste Anzahl von Attributen der beiden Kollektionen ermittelt und in der Variable *min* (Zeile 2) gespeichert. Dieser Wert wird benötigt, damit der *InclusionDependencyIterator* später die richtigen Attributkombinationen erzeugt.

In Zeile 3 bis 12 werden alle einzelnen Attribute der Kollektionen untereinander auf Inklusionsabhängigkeiten überprüft. Die dabei gefundenen Abhängigkeiten werden gespeichert und in Zeile 13 bei der Konstruktion des Iterators zusammen mit der

Variable *min* als Parameter überreicht.

Der Iterator generiert nun aus den bereits bekannten Inklusionsabhängigkeiten die noch zu überprüfenden Kandidaten (Zeile 14 bis 18). Werden dabei neue Inklusionsabhängigkeiten gefunden, so werden sie dem Iterator hinzugefügt (Zeile 17).

Am Ende werden alle gefundenen Inklusionsabhängigkeiten zurückgegeben, die im Iterator gespeichert wurden (Zeile 19).

### 4.3.1 Überprüfen der Teilmengeneigenschaft

Neben dem generellen Ablauf des Algorithmus, ist noch die Überprüfung der Teilmengeneigenschaft von Interesse. Listing 4.5 zeigt den Programmablauf bei der Überprüfung der Teilmengeneigenschaft in Java-ähnlichem Pseudo-Code. Die Methode gehört zur Klasse *DocumentCollectionWrapper*.

```
1  boolean isSubsetOf(DocumentCollectionWrapper left,
       DocumentCollectionWrapper right, int[] leftAttributes, int[]
       rightAttributes,boolean checkTypes, boolean checkMinMax) {
2   if (checkTypes && !typesMatch(...))
3    return false;
4   if (checkMinMax && if(!checkMinMax(...))
5    return false;
6   boolean found;
7   for (int i = 0; i < left.numberOfRecords; i++) {
8   found = false;
9    for (int j = 0; j < right.numberOfRecords; j++) {
10    if(right.values[j] == left.values[i]){
11     found = true;
12     break;
13    }
14   }
15   if(!found)
16    return false;
17  }
18  return true;
19 }
```

**Listing 4.5:** Algorithmus für die Überprüfung der Teilmengeneigenschaft in Java-ähnlichem Pseudocode

Als Parameter werden die zu überprüfenden Kollektionen, die jeweiligen Attribute in

Form von Integer-Arrays und zwei boolsche Variablen übergeben (Zeile 1).

Abhängig von der Variablen *checkTypes*, werden bei der Überprüfung der Teilmengeneigenschaft die Typen der beteiligten Attribute verglichen (Zeilen 2-3). Sollten die Typen nicht übereinstimmen, so kann frühzeitig festgestellt werden, dass die Teilmengeneigenschaft nicht erfüllt ist.

Die Variable *checkMinMax* legt fest, ob Minima und Maxima der jeweiligen Attribute überprüft werden (Zeilen 4-5). Wie schon in Kapitel 3.2.1 beschrieben, müssen dabei die folgenden Eigenschaften zwischen den Attributen $A$ und $B$ gelten, damit $A \subseteq B$ erfüllt sein kann.

$$A.min \geq B.min \qquad A.max \leq B.max$$
$$A.min \in B \qquad A.max \in B$$

Sollte eine dieser Eigenschaften nicht erfüllt sein, so kann hier ebenfalls frühzeitig erkannt werden, dass die Teilmengeneigenschaft nicht gilt.

Sollte keine der vorherigen Methoden zeigen, dass die Teilmengeneigenschaft nicht gilt, so muss überprüft werden, ob alle Attributwert-Kombinationen der linken Seite des $\subseteq$ Operators in den Attributwert-Kombinationen der rechten Seite enthalten sind.

Dabei wird jede Attributwert-Kombination der linken Seite mit allen Attributwert-Kombinationen der rechten Seite verglichen (Zeilen 7-17). Zuvor wird eine boolsche Variable *found* angelegt (Zeile 6). Sollte die Variable nach einem Durchlauf der inneren Schleife in Zeile 9 nicht gesetzt sein, so wurde festgestellt, dass eine Attributwert-Kombination nicht in der rechten Seite enthalten ist. Der Algorithmus wird abgebrochen und es wird *false* zurückgegeben (Zeilen 15-16).

Die Überprüfung auf Gleichheit der Attributwert-Kombinationen in Zeile 10 ist hier, der Übersichtlichkeit wegen, vereinfacht dargestellt. An dieser Stelle spielen natürlich auch die als Array übergebenen Attribute eine Rolle. Generell erfolgt die Speicherung der Werte in einem zweidimensionalen Object-Array.

Sollte die Variable *found* nach dem letzten Durchlauf der äußeren Schleife (Zeile 7) auf *true* gesetzt sein, so wurde die Teilmengeneigenschaft festgestellt und es wird *true* von der Methode zurückgegeben (Zeile 18).

Die Überprüfung, ob die Datentypen der Attributwert-Kombinationen zueinander passen, erfolgt in der Methode *typesMatch(...)* in Zeile 2. Da es in JSON-Dateien möglich ist, dass dasselbe Attribut Werte aus unterschiedlichen Wertebereichen enthält, werden die Datentypen jedes Attributes in einer Liste verwaltet. Die Methode *typesMatch* überprüft dann, ob alle Datentypen der Attribute der linken Seite in den Daten-

typen der Attribute der rechten Seite enthalten sind. Sollte das nicht der Fall sein, so wird von der Methode *false* zurückgegeben.

Die Verwaltung der Datentypen in einer Liste, sowie die Suche in dieser Liste, unterliegt einer quadratischen Komplexität. Da die Anzahl der potentiell in einem Attribut vorkommenden Datentypen stark limitiert ist, stellt die quadratische Komplexität an dieser Stelle kein Problem dar.

Während des Einlesens der Daten aus der Datenbank, wird die Liste mit den Datentypen befüllt.

Da es in JSON-Dateien möglich ist, dass dasselbe Attribut Werte aus unterschiedlichen Wertebereichen enthält, werden Minimum und Maximum für die Datentypen Double, Integer und String vorgehalten. Ein Minimum und Maximum für den Datentyp Boolean zu ermitteln, bietet keine Vorteile.

Da es sein kann, dass nicht alle Minima und Maxima gesetzt wurden, wird pro Paar von Minimum und Maximum eine boolsche Variable vorgehalten, die beim erstmaligen Setzen von Minimum und Maximum auf *true* gesetzt wird. Sowohl die Minima und Maxima, als auch die entsprechenden boolschen Variablen, werden in einem Array vorgehalten, dabei wird durch den Index bestimmt, zu welchem Attribut die Werte gehören.

Da vor allem die numerischen Werte nach der Initialisierung der Arrays auf den Wert 0 gesetzt sind, sind die boolschen Variablen wichtig. Ohne diese, wäre es nach dem Einlesen der Daten nicht unterscheidbar, ob 0 Minimum bzw. Maximum oder nur der Standardwert nach der Initialisierung des Arrays ist.

### 4.3.2 Erkennen von Inklusionsabhängigkeiten in Dokument-Kollektionen

Die Optionalität der Attribute in JSON-Dokumenten stellt ein Problem bei der Suche nach Inklusionsabhängigkeiten dar. Angefangen beim Einlesen der Daten bis zur Überprüfung der Teilmengeneigenschaft, muss beachtet werden, dass nicht alle Dokumente alle Attribute aufweisen.

Während des Einlesens der Daten gilt es unterschiedliche Probleme zu lösen.

Die Speicherung der Werte soll weiterhin in einem zweidimensionalen Object-Array erfolgen. Dabei ist zu beachten, dass dieses bei nicht vorhandenen Attributwerten nun Null-Werte enthält. Um das Array zu initialisieren, muss ermittelt werden, wie viele unterschiedliche Attribute in der Dokument-Kollektion enthalten sind. Dazu wird die Schemaextraktionskomponente (Klettke u. a., 2015) verwendet. Diese liefert alle

Schemata der Kollektion zurück. Daran lässt sich die Anzahl der benötigten Attribute erkennen.

Nachdem nun ein Array entsprechender Größe angelegt werden kann, muss dieses befüllt werden. Dazu muss bekannt sein, welches Attribut mit welchem Index in dem Array gespeichert wird. Der Attributname wird hierbei als Schlüssel benutzt, um den dazugehörigen Index in einer Hashtabelle zu speichern. Diese Datenstruktur wird angelegt und befüllt, während das Schema ausgelesen wird.

Soll nun ein konkreter Attributwert in dem Array gespeichert werden, so wird der entsprechende Index für die Spalte ermittelt, indem mit dem Attributnamen als Schlüssel auf die Hashtabelle zugegriffen wird. Der Index für die Zeile in dem Array ergibt sich aus der Anzahl der Dokumente und der Reihenfolge während des Einlesens.

Die Null-Werte haben weiterhin auch Auswirkungen auf die Überprüfung der Teilmengeneigenschaft. Dabei können folgende Fälle auftreten:

- Ein Null-Wert der linken Seite wird mit einem Wert der rechten Seite verglichen, welcher kein Null-Wert ist.

  $$\{null\} \subseteq \{1\}$$

  Die obige Eigenschaft kann weder als gültig noch als ungültig bewertet werden. Ob eine Teilmengeneigenschaft besteht hängt von den anderen Attributwerten ab, die nicht *null* sind. Somit sind die folgenden Teilmengeneigenschaften erfüllt:

  $$\{null, null, 1\} \subseteq \{1.0, 100, 1\} \rightarrow true$$

  $$\{null, null, 2\} \subseteq \{1.0, 100, 1\} \rightarrow false$$

- Ein Wert der linken Seite, welcher kein Null-Wert ist, wird mit einem Null-Wert der rechten Seite verglichen.

  $$\{1\} \subseteq \{null\}$$

  Auch hier hängt die Teilmengeneigenschaft von den Attributwerten ab, welche nicht *null* sind. Somit sind die folgenden Teilmengeneigenschaften erfüllt:

  $$\{1.0, 100, 1\} \subseteq \{1.0, null, null\} \rightarrow true$$

  $$\{2.0, 100, 1\} \subseteq \{1.0, null, null\} \rightarrow false$$

- Ein Null-Wert der linken Seite wird mit einem Null-Wert der rechten Seite verglichen.

  $$\{null\} \subseteq \{null\}$$

Auch die Überprüfung von zwei Null-Werten beeinflusst die Gültigkeit der Teilmengeneigenschaft nicht. Somit sind die folgenden Teilmengeneigenschaften erfüllt:

$$\{null, null, 1\} \subseteq \{null, null, 1\} \rightarrow true$$

$$\{null, null, 5\} \subseteq \{null, null, 1\} \rightarrow false$$

Viele fehlende Attribute beeinflussen die Gültigkeit der Teilmengeneigenschaft insofern, dass es mehr potenzielle Inklusionsabhängigkeiten gibt. Gibt es beispielsweise für ein Attribut nur 3 ganzzahlige Werte und für ein anderes Attribut mehrere Millionen unterschiedlicher ganzzahliger Werte, so steigt die Wahrscheinlichkeit, dass das Attribut mit den 3 Werten in dem anderen Attribut enthalten ist.

Neben den Attributen, enthält das extrahierte Schema auch eine prozentuale Angabe wie viele der Dokumente ein bestimmtes Attribut besitzen. Dieser Wert kann genutzt werden, um Speicheroptimierungen vorzunehmen. Es wäre zum Beispiel denkbar, alle Attributwerte die in weniger als 50% der Fälle vorkommen, in einer Liste ohne Null-Werte zu speichern.

Ebenfalls ist eine Laufzeitoptimierung eine denkbare Option, z.B. indem alle Indizes der Werte eines Attributes in einer Liste verwaltet werden. Somit könnten Null-Werte bei der Überprüfung der Teilmengeneigenschaft gezielt übersprungen werden.

Den Wert von 50% als Indikator für Optimierungen zu verwenden ist eventuell nicht ausreichend. Ein besserer Wert müsste durch Versuche oder weitere Überlegungen ermittelt werden und kann durchaus wesentlich kleiner als 50% sein, damit die Optimierung effektiv ist.

Bei der Überprüfung der Datentypen, sowie dem Vergleich von Minima und Maxima, müssen für Dokument-Kollektionen keine Anpassungen vorgenommen werden.

Um ineinander geschachtelte JSON-Objekte verarbeiten zu können, sind die gleichen Ansätze wie bei funktionalen Abhängigkeiten denkbar. Vor allem das Entfalten der Objekte scheint eine naheliegende Lösung zu sein.

Bei der Suche nach Inklusionsabhängigkeiten werden Arrays vernachlässigt, da Primärschlüssel-Fremdschlüssel-Beziehungen in Form von Arrays nicht zu erwarten sind.

### 4.3.3 Join-Bedingungen auf Basis von Inklusionsabhängigkeiten ermitteln

Ziel dieser Arbeit ist es, Vorschläge für Join-Bedingungen zwischen zwei Entity-Typen zu erzeugen. Die Entity-Typen werden dabei durch Kollektionen der dokumentorientierten Datenbank dargestellt.

Eine Join-Operation setzt voraus, dass mindestens eine Inklusionsabhängigkeit zwischen den beiden Kollektionen existiert. Diese Inklusionsabhängigkeiten werden durch das Anwenden des dafür entwickelten Algorithmus ermittelt.

Der Algorithmus liefert paarweise die Attribute bzw. Attributkombinationen zurück, zwischen denen Inklusionsabhängigkeiten bestehen. Dabei muss der Algorithmus zweimal angewendet werden, um die Abhängigkeiten für beide Richtungen zu ermitteln. Dazu werden einfach die Parameter, welche die Kollektionen darstellen, beim Aufrufen des Algorithmus, vertauscht.

Vorausgesetzt der Algorithmus liefert zurück, dass es zwischen den Kollektionen $K_1$ und $K_2$ eine Inklusionsabhängigkeit zwischen den Attributen $A_1$ und $B_2$ gibt, wobei $A_1$ zu $K_1$ und $B_2$ zu $K_2$ gehört, so kann eine Join-Anfrage wie folgt aussehen:

$$\text{SELECT * FROM } K_1, K_2 \text{ WHERE } A_1{=}B_2$$

Dabei ist $A_1$=$B_2$ die Join-Bedingung. Nach diesem Schema, können alle Ergebnisse des Algorithmus in Join-Bedingungen umgewandelt werden.

Weiterhin kann anhand der ermittelten Minima und Maxima festgestellt werden, ob statt des $=$-Operators auch andere Operatoren, wie $<$ , $\leq$ , $>$ , $\geq$ in Frage kommen.

# Kapitel 5

# Evaluation

In diesem Kapitel erfolgt der Vergleich der entwickelten Algorithmen. Dabei werden unterschiedlich geartete Datensätze als Input für die Algorithmen verwendet.

Die Evaluation der Algorithmen erfolgt mit konkreten Datensätzen, da sowohl die Verteilung der Datentypen als auch die Verteilung der Daten selbst, starken Einfluss auf die Performance der Algorithmen hat.

Als zu Grunde liegendes dokumentenorientiertes Datenbanksystem, wurde für die Evaluation MongoDB ausgewählt.

## 5.1 Verwendete Daten

Als Grundlage der Evaluation der verschiedenen Konzepte und Algorithmen wurden einige Datensätze ausgewählt. Die Auswahl erfolgte auf Basis der in (Papenbrock und Naumann, 2016) verwendeten Datensätze, da somit ein direkter Vergleich zu der Evaluation bestehender Algorithmen möglich ist.

Die hier verwendeten Datensätze stammen vorwiegend aus dem *UC Irvine Machine Learning Repository*. Dieses Repository verwaltet momentan 379 Datensätze die vor allem für *Machine Learning* Anwendungen gedacht sind.

Daraus wurden der Iris-Datensatz (Fisher, 1988) und der Horse-Colic-Datensatz (McLeish und Cecile, 1989) ausgewählt, um die Implementierung und Evaluation der Algorithmen durchzuführen.

Als weiterer Datensatz wurde der MovieLens 20M Datensatz (Harper und Konstan, 2015) gewählt, da dieser aus mehreren Relationen besteht, zwischen welchen Inklusionsabhängigkeiten bestehen.

### 5.1.1 Anpassung der Daten

Die Daten liegen allgemein im *CSV* Dateiformat vor, dabei ist *CSV* die Abkürzung von *comma seperated values*. Vor dem Import in eine MongoDB Datenbank müssen diverse Anpassungen vorgenommen werden. Fehlende Werte sind in einigen CSV-Dateien durch ein Fragezeichen dargestellt. Der erste Schritt ist es, die Fragezeichen durch den Null-Wert zu ersetzen. Diese Ersetzung erfolgte unter der Zuhilfenahme der Suchen-Ersetzen-Funktion eines Texteditors.

Die Datensätze sind grundlegend für relationale Datenbanksysteme ausgelegt und müssen in JSON-Dokumente umgewandelt werden. Während des Imports erfolgt die Umwandlung der CSV-Dateien durch das Datenbanksystem, in diesem Fall durch *MongoDB*. Dabei wird das Programm *mongoimport* verwendet. Da CSV-Dateien als Quelle für den Import verwendet werden, muss der Importbefehl wie folgt parametrisiert werden.

```
"mongoimport.exe /d Datenbankname /file .\iris.csv /headerline
/type:csv"
```

Beim Import der CSV-Datei ist es nötig den Datenbanknamen, sowie die Quelldatei anzugeben. Weiterhin muss der Dateityp spezifiziert werden und es muss angegeben werden, ob die Attributnamen in der Datei enthalten sind. Das passiert in diesem Fall durch die Angabe von *headerline*.

Die Daten sind nach dem Import in einer Kollektion als JSON-Dokumente gespeichert. Da die Daten ursprünglich für relationale Systeme gedacht waren, fehlen typische Eigenschaften von Dokumenten in einer Kollektion.

Typisch für die Dokumente einer Kollektion ist die Optionalität von Attributen. Damit ist gemeint, dass nicht alle Dokumente der Kollektion dieselben Attribute besitzen. Diese Eigenschaft kann erreicht werden, indem in einigen oder allen Dokumenten die Attribute gelöscht werden, die auf *null* gesetzt sind.

Da die Daten bereits importiert wurden, kann die Update-Funktionalität des Datenbanksystems ausgenutzt werden. Alternativ können auch vor dem Import in die MongoDB Datenbank die entsprechenden Attribute gelöscht werden, so dass nach dem Import keine weiteren Anpassungen mehr nötig sind.

Eine weitere Eigenschaft von JSON-Dokumenten ist die Verschachtelung von Objekten. Diese ist nach dem initialen Import ebenfalls nicht gegeben. Um die Verschachtelung zu erreichen, können bereits bestehende Attribute zu Objekten gruppiert werden.

Die Verwendung von Arrays ist ebenfalls typisch für JSON-Dokumente. Eine Möglichkeit Arrays aus den bestehenden Daten zu Erzeugen, ist vorhandene Attribute des selben Datentyps zu Arrays zusammenzufügen.

## 5.2 Messung der Performance der Algorithmen

Die Performance Messungen werden lokal auf einem Notebook mit den folgenden Spezifikationen durchgeführt:

- Prozessor: Intel Core i5-4210H, 2,9 GHz
- Arbeitsspeicher: 8GB DDR3
- Betriebssystem: Windows 10 64bit
- Festplatte: Modellbezeichnung HTS721010A9E630, SATA 600, 7200 U/min

Die verschiedenen Testfälle werden dabei als JUnit-Tests modelliert. Dabei wird in den einzelnen JUnit-Tests die Zeit für die getesteten Algorithmen gemessen. Die Zeit für das initiale Laden der Daten aus der Datenbank und das Befüllen der Datenstrukturen wird hierbei nicht gemessen.

Die genauen Zeiten werden mit der Methode *System.nanoTime()* ermittelt, da diese die höchste Messgenauigkeit bietet.

Um verlässliche Messergebnisse zu erhalten, werden die einzelnen Messungen wiederholt. Je nach erhaltenen Messwerten, kann es Sinn ergeben, den Durchschnitt oder den Median zu verwenden. Dadurch soll verhindert werden, dass Ausreißer in den Messwerten das Ergebnis verfälschen. Diese Ausreißer können verschiedene Ursachen haben, die nicht immer unterbunden werden können. Es können zum Beispiel Caching-Effekte oder das automatische Hoch-Takten des Prozessors eine Rolle spielen.

Auch die Energieeinstellungen des Rechners wirken sich auf Performance-Messungen aus. Für die hier durchgeführten Messungen wird das Profil *Höchsleistung* verwendet, um möglichst alle Ressourcen des Rechners auszunutzen.

Weiterhin wird die Größe der verwendeten Datensätze variiert um die Skalierbarkeit der Algorithmen zu überprüfen.

## 5.3 Inklusionsabhängigkeiten

Um die verschiedenen Implementierungen für die Suche nach Inklusionsabhängigkeiten zu vergleichen, wird das *MovieLens Dataset* (Harper und Konstan, 2015) verwendet. Dieser Datensatz enthält 20 Millionen Bewertungen für 27000 Filme, sowie 465000 Tags. Die Bewertungen und Tags wurden von 20000 Benutzern vergeben. Der Datensatz enthält noch weitere Informationen, die für die Evaluation jedoch nicht verwendet werden.

Nach dem Import des Datensatzes in MongoDB sind für die Evaluation die Kollektionen *movies* und *links* von Bedeutung. Die Kollektion *movies* enthält ungefähr 27000 Dokumente mit den Attributen *movieId, title* und *genres*. Das Attribut *movieId* ist Primärschlüssel und hat den Datentyp einer Number. In der Kollektion *links* sind ebenfalls um die 27000 Dokumente enthalten. Diese besitzen die Attribute *movieId, imdbId* und *tmdbId*. Hier ist *movieId* ebenfalls der Primärschlüssel und vom Datentyp Number. Zwischen den Attributen *movieID* besteht in beide Richtungen eine Inklusionsabhängigkeit zwischen den Kollektionen. Es gibt keine weiteren Inklusionsabhängigkeiten.

Für das Finden von Inklusionsabhängigkeiten, sollen vier verschiedene Algorithmen miteinander verglichen werden. Dabei bauen diese Algorithmen aufeinander auf und werden schrittweise um Optimierungen bezüglich der Suchraumeinschränkung erweitert.

In den folgenden Messungen überprüfen die Algorithmen ob es Inklusionsabhängigkeiten zwischen den Kollektionen *links* und *movies* gibt. Dabei wird nur in der Richtung $links \subseteq movies$ gesucht.

### 5.3.1 Basis-Algorithmus

Bei diesem Algorithmus werden, unabhängig von Datentypen oder sonstigen Eigenschaften, alle Teilmengeneigenschaften zwischen allen möglichen Attributkombinationen überprüft.

Die Messungen dieses Algorithmus dienen als Vergleichsgrundlage für allen anderen Messungen. Die Erwartung ist, dass jede Einschränkung des Suchraums weitere Verbesserungen der Performance bringt. In Abbildung 5.1 ist erkennbar, dass bis auf den ersten Messwert, alle anderen Messwerte eine relativ kleine Abweichung vom Durchschnittswert (4,918s) und vom Median (5,004s) aufweisen.

Der erste Messwert kann als Ausreißer betrachtet werden und beeinflusst lediglich den Durchschnittswert in negativer Richtung.

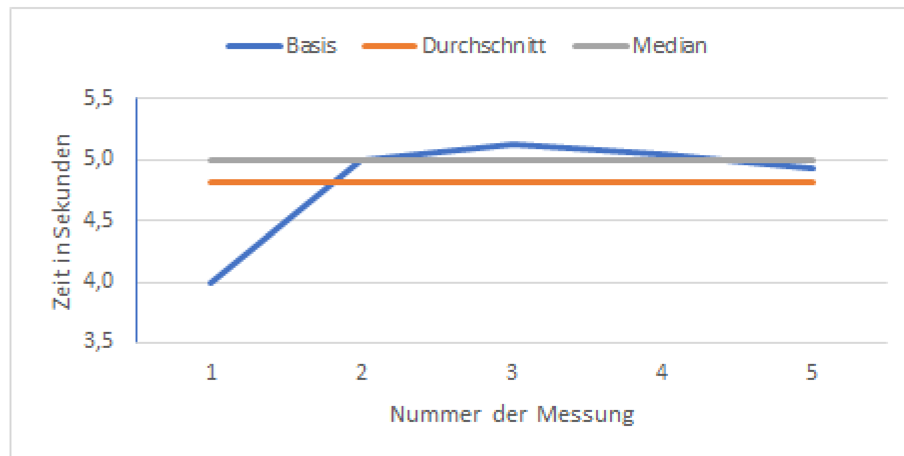


**Abbildung 5.1:** Ergebnisse von 5 Testläufen des Basis-Algorithmus

### 5.3.2 Smart-Iterator-Algorithmus

Der Smart-Iterator-Algorithmus schränkt den Suchraum ein, indem aus bereits bekannten Inklusionsabhängigkeiten, die noch zu überprüfenden generiert werden. Die Komponente, welche die Inklusionsabhängigkeiten generiert, wird hier als Smart-Iterator bezeichnet.

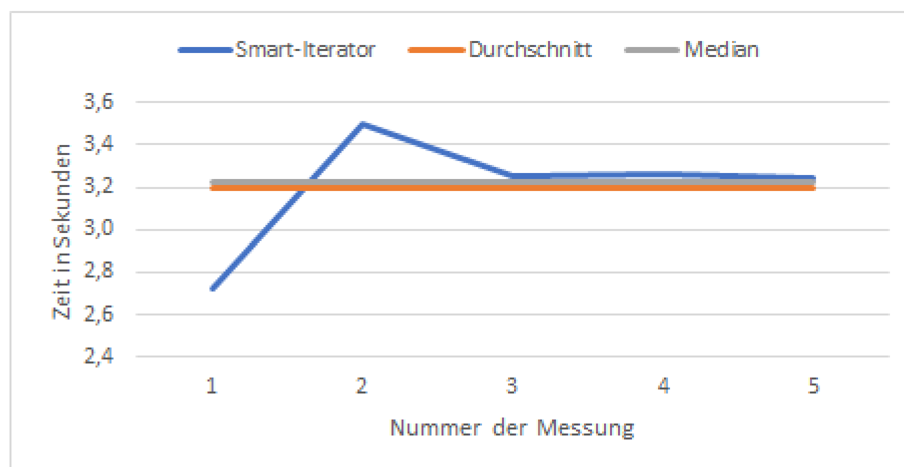


**Abbildung 5.2:** Ergebnisse von 5 Testläufen des Smart-Iterator-Algorithmus

Für den getesten Datensatz bedeutet das konkret, dass erst die Inklusionsabhängigkeiten mit nur einem Attribut geprüft werden. Dabei wird festgestellt, dass es nur eine Inklusionsabhängigkeit gibt, nämlich $links.movieId \subseteq movies.movieId$.

Da aus nur einer bestehenden Inklusionsabhängigkeit keine weiteren Kandidaten für potenzielle Inklusionsabhängigkeiten generiert werden können, endet der Algorithmus hier.

Wie in Abbildung 5.2 zu erkennen ist, sind die ersten beiden Messwerte Ausreißer. Da Median und Durchschnitt mit 3,198s und 3,255s dicht beieinander liegen, heben sich die beiden Ausreißer auf.

Im Vergleich zum Basis-Algorithmus hat sich die Laufzeit reduziert, was der Erwartungshaltung entspricht.

### 5.3.3 Smart-Iterator-Algorithmus mit Einschränkungen durch Datentypen

In diesem Algorithmus wird der Suchraum weiter eingeschränkt. Sollten zwei zu vergleichende Attribute nicht die gleichen Datentypen enthalten, so wird gleich festgestellt, dass keine Inklusionsabhängigkeit vorliegen kann.

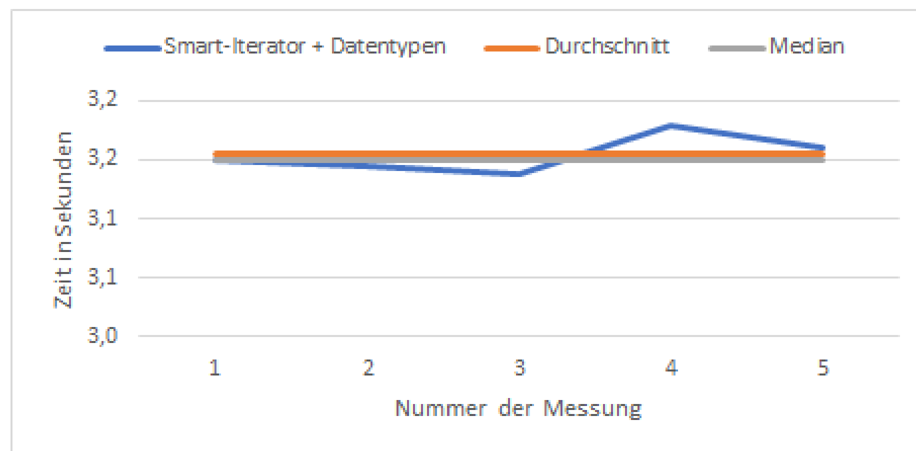


**Abbildung 5.3:** Ergebnisse von 5 Testläufen des Smart-Iterator-Algorithmus mit Einschränkung des Suchraums durch Datentypen

Abbildung 5.1 zeigt, dass alle Messwerte sehr dicht beieinander liegen. Auch der Unterscheid zwischen Median (3,150s) und Durchschnitt (3,155s) ist sehr gering.

Im Vergleich zum vorherigen Algorithmus ist nur eine geringe Verbesserung der Performance zu erkennen. Dieses Ergebnis ist jedoch zu erwarten.

Der Algorithmus bricht die Suche nach Überprüfung der Teilmengeneigenschaft zwischen zwei Attributkombinationen ab, sollten die Datentypen nicht übereinstimmen. Im Fall, das an dieser Stelle nicht abgebrochen wird, wird geprüft, ob die Werte

der linken Seite in der rechten Seite enthalten sind. Nachdem die rechte Seite einmal durchlaufen wurde, wird festgestellt, dass die erste Werte-Kombination nicht enthalten ist. Somit gilt die Teilmengeneigenschaft nicht. Das einmalige Durchlaufen der rechten Seite ist hierbei nicht besonders zeitaufwändig, daher die geringe Verbesserung der Performance.

Da der Smart-Iterator aus den einwertigen Abhängigkeiten nur mehrwertig Kandidaten erzeugt, bei denen die Typen übereinstimmen, greift die Suchraumeinschränkung über Datentypen nur beim Vergleich von einwertigen Attributen.

### 5.3.4 Smart-Iterator-Algorithmus mit Einschränkungen durch Datentypen und statistische Daten über Attributwerte

Dieser Algorithmus erweitert den vorangegangenen. Dabei für jedes Attribut Minimum und Maximum ermittelt und unter Verwendung dieser Werte wird versucht, nicht-gültige Teilmengeneigenschaften schneller festzustellen.

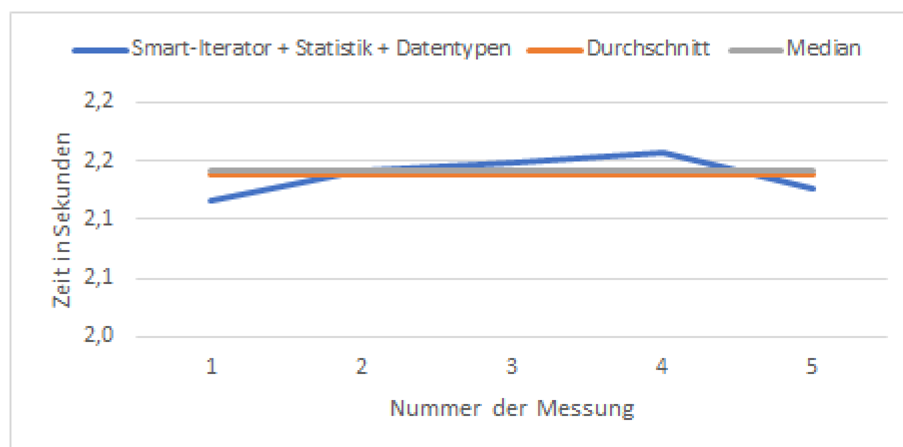


**Abbildung 5.4:** Ergebnisse von 5 Testläufen des Smart-Iterator-Algorithmus mit Einschränkung des Suchraums durch Datentypen und statistische Informationen über die Daten

Wie Abbildung 5.4 zeigt, liegen auch hier die Messwerte sehr nah beieinander. Median und Durchschnitt sind mit einem Unterschied von nur 4ms fast identisch. Die Verbesserung der Performance, im Vergleich zum Vorherigen Algorithmus ist erheblich. Jedoch hängt die Performance-Verbesserung stark von den Daten ab. Sind Minimum und Maximum der linken Seite in der rechten Seite enthalten, so muss der komplette

Suchraum durchmustert werden. In diesem Fall gibt es keine Verbesserung der Performance.

### 5.3.5 Vergleich der Suchraumeinschränkungen

Abbildung 5.5 ermöglicht einen direkten Vergleich der vorgestellten Algorithmen. Dabei wurden nur Median und Durchschnitt miteinander verglichen.

| Testlauf | Basis | Smart-Iterator | Smart-Iterator + Datentypen | Smart-Iterator + Statistik + Datentypen |
|---|---|---|---|---|
| 1 | 3,993 | 2,726 | 3,150 | 2,115 |
| 2 | 5,004 | 3,497 | 3,145 | 2,142 |
| 3 | 5,121 | 3,255 | 3,139 | 2,149 |
| 4 | 5,053 | 3,262 | 3,180 | 2,156 |
| 5 | 4,924 | 3,246 | 3,159 | 2,126 |
| Durchschnitt | 4,819 | 3,198 | 3,155 | 2,138 |
| Median | 5,004 | 3,255 | 3,150 | 2,142 |

**Tabelle 5.1:** Konkrete Messwerte der Evaluierten Algorithmen

In Tabelle 5.1 sind alle Messwerte der vorangegangenen Messungen dargestellt. Aus den Median- und Durchschnittswerten der Abbildung, ergeben sich die Balkendiagramme von Abbildung 5.5.

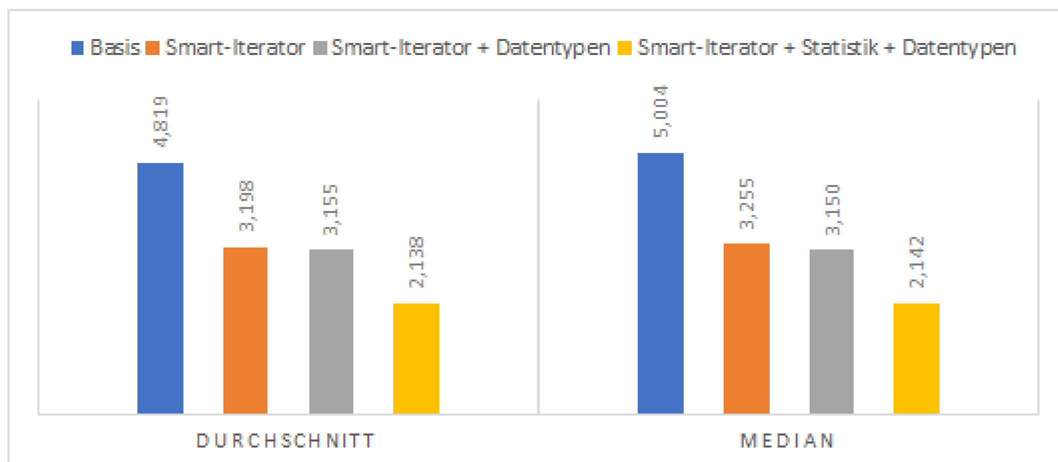


**Abbildung 5.5:** Vergleich der Algorithmen mit unterschiedlichen Suchraumeinschränkungen

Die Diagramme in Abbildung 5.5 lassen erkennen, dass zwei der Optimierungen die

Performance des Algorithmus verbessern und eine Optimierung zu keiner nennenswerten Verbesserung der Performance führt.

Zur Verbesserung der Performance tragen die Einschränkungen der Suche auf relevante Attributkombinationen, durch den Smart-Iterator, und die Nutzung statistischer Daten, bei der Überprüfung der Teilmengeneigenschaft, bei.

Die Komplexität der Suche über die Attributkombinationen ist im Bereich der Fakultät. Bei steigender Anzahl der Attribute, ist zu erwarten, dass die Verwendung des Smart-Iterator einen größeren Beitrag zur Performance-Verbesserung leistet.

Die Verwendung statistischer Daten bei der Überprüfung der Teilmengeneigenschaft, führt ebenfalls zu einer Verbesserung der Performance. Diese Optimierung ist vor allem bei der Überprüfung einwertiger Attributkombinationen effektiv. Da der Smart-Iterator die mehrwertigen Kandidaten nur aus den gültigen einwertigen Inklusionsabhängigkeiten generiert, entsteht bei deren Überprüfung kein Vorteil mehr durch die Optimierung.

Generell kann angenommen werden, dass die Anzahl der Attribute wesentlich kleiner ist als die Anzahl der Dokumente in einer Kollektion. Da der Suchraum jedoch mit steigender Anzahl der Attribute deutlich stärker wächst als mit Steigender Anzahl der Dokumente, sind Suchraumeinschränkungen in beiden Richtungen sinnvoll.

Skalierbarkeit mir der Anzahl der Attribute wurde ebenfalls getestet. Dazu wurde der Iris-Datensatz (Fisher, 1988) verwendet. Dieser beinhaltet 5 Attribute mit jeweils 150 Werten.

Um den Algorithmus möglichst auszulasten, wurden zwei identische Kollektionen, auf Inklusionsabhängigkeiten zwischen diesen, untersucht. Die Anzahl an Attributen wurde ebenfalls erhöht. Die Messungen erfolgten für 5, 8, 10 und 12 Attribute mit jeweils 150 Werten pro Attribut. Dabei wurden die Zeiten von jeweils 10 Ausführungen des Algorithmus gemessen. Die Messergebnisse können in Tabelle 5.2 betrachtet werden.

Da zwei identische Kollektionen auf Inklusionsabhängigkeiten untersucht wurden, ist das Ergebnis, dass zwischen alle gleichen Attributkombinationen eine Inklusionsabhängigkeit besteht.

Der Iris-Datensatz hat 5 Attribute, diese wurden zur Erhöhung der Attributanzahl als neue Spalten bzw. Attribute hinzugefügt. Die Messung mit 12 Attributen wurde als Oberes Limit gewählt, da hier bereits erkennbar ist, dass die Laufzeit des Algorithmus signifikant länger wird.

In Abbildung 5.6 ist die Laufzeit des Algorithmus für Kollektionen mit steigen-

| | Anzahl der Spalten | | | |
|---|---|---|---|---|
| | 5 | 8 | 10 | 12 |
| Testdurchläufe (Zeit in Sekunden) | 0,036 | 0,627 | 5,142 | 48,209 |
| | 0,034 | 0,497 | 4,873 | 47,411 |
| | 0,050 | 0,474 | 4,558 | 57,721 |
| | 0,012 | 0,479 | 4,692 | 57,700 |
| | 0,010 | 0,475 | 4,559 | 58,271 |
| | 0,011 | 0,476 | 4,586 | 57,731 |
| | 0,017 | 0,476 | 4,571 | 58,188 |
| | 0,012 | 0,476 | 4,558 | 57,788 |
| | 0,014 | 0,466 | 4,564 | 57,748 |
| | 0,015 | 0,455 | 4,553 | 57,822 |
| Durchschnitt | 0,021 | 0,490 | 4,666 | 55,859 |
| Median | 0,015 | 0,476 | 4,568 | 57,739 |
| Teilmengenvergleiche | 1.545 | 1.441.728 | 234.662.230 | 1.794.846.864 |

**Tabelle 5.2:** Messergebnisse des Algorithmus mit unterschiedlich großen Kollektionen bezogen auf die Anzahl der Attribute

der Anzahl an Attributen dargestellt. Das Balkendiagramm entstand aus den Durchschnitts- und Medianwerten aus Tabelle 5.2. Da die Median und Durchschnitt für 5 als auch 8 Spalten jeweils unter einer Sekunde liegen, sind die jeweiligen Balken vergleichbar klein. Die Messung mit 10 Attributen lässt erkennen, dass der zu durchmusternde Suchraum beginnt zu wachsen, Median und Durchschnitt liegen dabei etwa im Bereich von 4,5 und 4,7 Sekunden. Bereits bei 12 Attributen steigt die Rechenzeit stark an, der Medianwert liegt bei ungefähr 57,7 Sekunden und der Durchschnittswert liegt bei ungefähr 55,9 Sekunden.

Dieser starke Anstieg entspricht durchaus den Erwartungen, da auch die Anzahl an Teilmengenvergleichen massiv Ansteigt. In Tabelle 5.2 sind für die jeweilige Anzahl der Attribute auch die Anzahl an Teilmengenvergleichen aufgeführt. Sind es bei 10 Attributen noch 234.662.230 Vergleiche, so sind es bei 12 Attributen schon 1.794.846.864 Vergleiche. Die größe der Eingabedaten steigt also um 20% und die Laufzeit um 7649%. Da die Kollektionen identisch sind, greifen auch keine der eingeführten Suchraumeinschränkungen an dieser Stelle.

Die Suche nach Inklusionsabhängigkeiten zwischen identischen Kollektionen stellt bezüglich der Laufzeit einen *worst case* dar. Unter realen Bedingungen ist durchaus anzunehmen, dass es nicht sehr viele Inklusionsabhängigkeiten zwischen zwei unterschiedlichen Kollektionen gibt. Diese Annahme kann getroffen werden, da das für relationale Datenbanken auch nicht der Fall ist und davon ausgegangen wird, dass

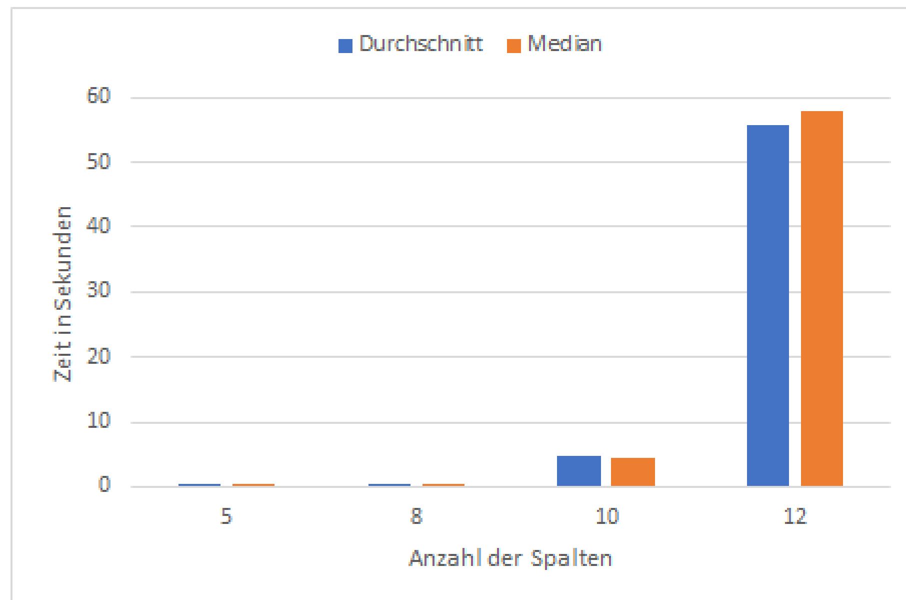


**Abbildung 5.6:** Vergleich des Algorithmus mit unterschiedlich großen Kollektionen als Eingabedaten

NoSQL-Daten mit Integritätsbedingungen durch relationale Systeme erzeugt wurden.

Eine Ausnahme für diese Annahme können jedoch zeitkritische Operationen auf den NoSQL-Systemen sein. Dafür könnten Kollektionen dupliziert werden, wobei zusätzliche Inklusionsabhängigkeiten entstehen würden.

Um zu ermitteln, wie der Algorithmus skaliert, werden weitere Messungen durchgeführt, bei denen die Anzahl der Attribute konstant bleibt, jedoch die Anzahl der Werte pro Attribut erhöht wird. Die Ergebnisse dieser Messungen befinden sich in Tabelle 5.3.

Da die vorherigen Messwerte aus Abbildung 5.6 bereits bei 10 Attributen einen Anstieg der Laufzeit verzeichneten, wurden die folgenden Messungen mit 10 Attributen durchgeführt. Dabei besitzen alle Attribute bei der ersten Messreihe 150 Werte, wie auch der originale Datensatz. Für jede weitere Messung wurden 150 Werte hinzugefügt, wobei vermieden wurde, bereits enthaltene Werte erneut zu verwenden.

Die Erwartung ist, dass der Algorithmus mit steigender Anzahl an Werten quadratisch skaliert. Bei einer Verdopplung der Werte ist also eine Vervierfachung der Laufzeit zu erwarten. In Abbildung 5.7 ist zu erkennen, dass die Laufzeit des Algorithmus stärker als linear, jedoch schwächer als quadratisch ansteigt. Bei der Verdopplung der Werte von 150 auf 300 steigt die Laufzeit, wie zu erwarten war, um ungefähr das Vier-

fache an. Interessant ist, dass bei der Erhöhung der Werte auf 450 die Laufzeit nur minimal ansteigt. Grund dafür könnten Caching-Effekte sein.

Wird die Anzahl der Werte auf 600 erhöht, ist wieder ein signifikanter Anstieg der Laufzeit zu erkennen. Dabei steigt die Laufzeit aber nicht mehr quadratisch mit der Anzahl der Werte. Da von 150 auf 600 Werte die Anzahl vervierfacht wurde, ist davon auszugehen, dass sich die Laufzeit um das 16-fache erhöht. Diese hat sich jedoch nur ungefähr um das 7-fache erhöht. Daraus kann gefolgert werden, dass der Algorithmus nicht voll-quadratisch mit der Anzahl der Werte pro Attribut skaliert.

| | Anzahl der Werte | | | |
|---|---|---|---|---|
| | 150 | 300 | 450 | 600 |
| Testdurchläufe (Zeit in Sekunden) | 5,142 | 18,497 | 20,919 | 35,858 |
| | 4,873 | 16,749 | 18,708 | 32,938 |
| | 4,558 | 16,000 | 32,248 | 33,905 |
| | 4,692 | 20,867 | 20,216 | 33,135 |
| | 4,559 | 20,221 | 20,330 | 33,456 |
| | 4,586 | 19,937 | 20,363 | 33,626 |
| | 4,571 | 19,781 | 20,330 | 32,985 |
| | 4,558 | 19,743 | 20,444 | 32,995 |
| | 4,564 | 19,806 | 20,337 | 33,224 |
| | 4,553 | 19,775 | 20,363 | 33,382 |
| Durchschnitt | 4,666 | 19,138 | 21,426 | 33,550 |
| Median | 4,568 | 19,778 | 20,350 | 33,303 |

**Tabelle 5.3:** Messergebnisse des Algorithmus mit unterschiedlich großen Kollektionen bezogen auf die Anzahl der Werte pro Attribut

Zusammenfassend kann gesagt werden, dass das Verhalten des Algorithmus bezüglich der Laufzeit den Erwartungen entspricht, was den *worst case* betrifft. In realen Szenarien ist zu erwarten, dass es nur einige weinige Inklusionsabhängigkeiten gibt und der Suchraum dadurch stark eingeschränkt wird.

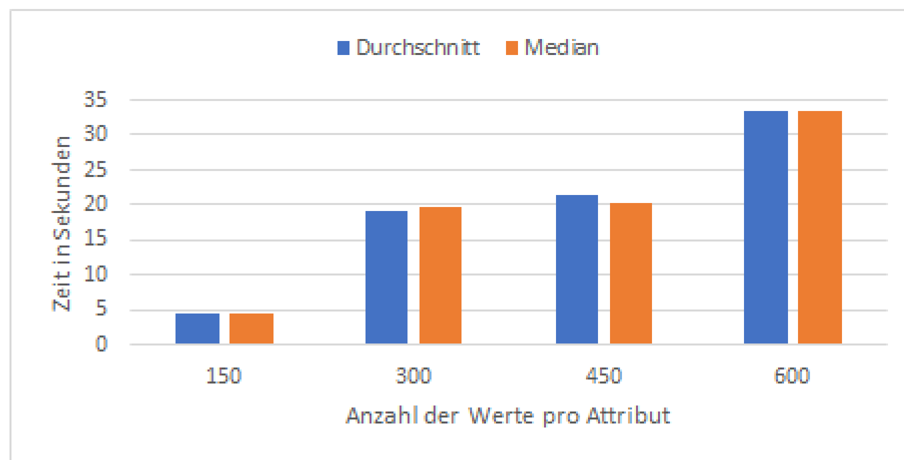


**Abbildung 5.7:** Vergleich des Algorithmus mit unterschiedlich großen Kollektionen als Eingabedaten

# Kapitel 6

# Zusammenfassung

Zweck dieser Arbeit war das Ableiten von Integritätsbedingungen aus Daten in vorhandenen NoSQL-Datenbanken. Dabei wurden dokumentenorientierte NoSQL-Datenbanksysteme als Grundlage verwendet.

Zu den abzuleitenden Integritätsbedingungen gehören: die Not-Null-Bedingung, die Unique-Bedingung, Primärschlüssel, Fremdschlüssel, sowie funktionale Abhängigkeiten und Inklusionsabhängigkeiten.

Für die Bestimmung der Integritätsbedingungen wurden, bereits für relationale Daten entwickelte, Algorithmen als Grundlage verwendet und angepasst, darunter der HyFD Algorithmus (Papenbrock und Naumann, 2016).

Ein Algorithmus für die Suche nach Inklusionsabhängigkeiten wurde selbst entwickelt, dabei wurden Suchraumeinschränkungen über die Datentypen und statistische Eigenschaften der Attribute umgesetzt. Weiterhin wurde ein Iterator entwickelt, welcher aus den gefundenen Inklusionsabhängigkeiten mit einem Attribut weitere Kandidaten generiert. Dazu wurde der *a priori* Schritt des *association rules* Algorithmus als Grundlage verwendet.

Anpassungen bezüglich der Optionalität von Attributen in JSON-Dateien wurden für den Algorithmus zur Suche nach Inklusionsabhängigkeiten entwickelt und umgesetzt.

Der Algorithmus zur Suche von funktionalen Abhängigkeiten wurde teilweise implementiert. Die nötigen Anpassungen bezüglich der Optionalität von Attributen in JSON-Dateien wurden konzeptuell erläutert.

Die Erweiterung der Algorithmen, so dass diese mit geschachtelten Objekten innerhalb eines Dokumentes umgehen können, wurden konzeptuell behandelt.

Weiterhin wurden potenzielle Verbesserungen der Laufzeit durch die Anwendung von Multithreading betrachtet. Dabei wurde festgestellt, dass durchaus Potential für

die Verbesserung der Performance besteht. Insbesondere der Algorithmus zur Suche von Inklusionsabhängigkeiten, kann von Multithreading profitieren.

Wie aus den ermittelten Inklusionsabhängigkeiten Join-Bedingungen ermittelt werden können, wurde ebenfalls erläutert.

Anschließend erfolgte eine Evaluation der Algorithmen, insbesondere des Algorithmus zur Suche von Inklusionsabhängigkeiten. Dabei wurden verschiedene Datensätze aus in MongoDB gespeicherten Kollektionen als Input für den Algorithmus verwendet. Ziel dabei war es, zu zeigen, wie der Algorithmus bei wachsender Anzahl an Attributen bzw. wachsender Anzahl an Werten, skaliert.

Die Evaluation des Algorithmus wurde mit konkreten Daten durchgeführt, da Datentypen und die Verteilung der Werte für Attribute starke Auswirkungen auf die entwickelten Suchraumeinschränkungen haben.

Dabei wurde festgestellt, dass der Algorithmus zum finden von Inklusionsabhängigkeiten mit steigender Anzahl der Attribute stärker an Laufzeit gewinnt als mit steigender Anzahl an Werten pro Attribut. Bezüglich der Eingabedaten verhielt sich der Algorithmus wie erwartet. Ausschlaggebend für die Suchraumeinschränkungen, ist vor allem die Anzahl der Inklusionsabhängigkeiten zwischen einzelnen Attributen der Kollektionen. Sind davon nur wenige vorhanden, kann die weitere Suche stark eingeschränkt werden.

Es gibt weiterhin offene Fragestellungen bezüglich der Behandlung von Arrays und geschachtelten Objekten und Arrays, beispielsweise ob im Zusammenhang mit Arrays eine neue Definition von funktionaler Abhängigkeit benötigt wird, welche Reihenfolge und Mengeneigenschaft der Arrays berücksichtigt. Ebenso ist noch offen, wie Arrays bei der Suche nach Inklusionsabhängigkeiten behandelt werden können und ob auf Ebene der Arrays überhaupt Inklusionsabhängigkeiten zu erwarten sind.

Zusammenfassend wurden in dieser Arbeit viele der Probleme der anfänglichen Fragestellung betrachtet. Dabei wurden sowohl konzeptuelle Lösungsvorschläge als auch prototypische Implementierungen entwickelt. Weiterhin haben sich neue Fragestellungen ergeben, die einer genaueren Betrachtung bedürfen.

# Literaturverzeichnis

[Agrawal u. a. 1994] AGRAWAL, Rakesh ; SRIKANT, Ramakrishnan u. a.: Fast algorithms for mining association rules. In: *Proc. 20th int. conf. very large data bases, VLDB* Bd. 1215, 1994, S. 487–499 21

[Bell 1995] BELL, S: Inferring data independencies. In: *Technical Report16, University Dortmund, Informatik VIII* (1995) 24

[Fisher 1988] FISHER, R.A.: *UCI Machine Learning Repository*. 1988. – URL `https://archive.ics.uci.edu/ml/datasets/iris` 44, 52

[Flach und Savnik 1999] FLACH, Peter A. ; SAVNIK, Iztok: Database dependency discovery: a machine learning approach. In: *AI communications* 12 (1999), Nr. 3, S. 139–160

[Gilbert und Lynch 2002] GILBERT, Seth ; LYNCH, Nancy: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. In: *Acm Sigact News* 33 (2002), Nr. 2, S. 51–59 3

[Harper und Konstan 2015] HARPER, F. M. ; KONSTAN, Joseph A.: The MovieLens Datasets: History and Context. In: *ACM Trans. Interact. Intell. Syst.* 5 (2015), Dezember, Nr. 4, S. 19:1–19:19. – URL `http://doi.acm.org/10.1145/2827872`. – ISSN 2160-6455 44, 47

[Heuer und Saake 2000] HEUER, Andreas ; SAAKE, Gunter: *Datenbanken: Konzepte und Sprachen;[der fundierte Einstieg in Datenbanken; Schwerpunkt: Datenbankentwurf und Datenbanksprachen; inklusive aktueller Trends: SQL-99, JDBC, OLAP, Textsuche]*. mitp, 2000 7, 8, 9, 10

[Huhtala u. a. 1999] HUHTALA, Ykä ; KÄRKKÄINEN, Juha ; PORKKA, Pasi ; TOIVONEN, Hannu: TANE: An efficient algorithm for discovering functional and approximate dependencies. In: *The computer journal* 42 (1999), Nr. 2, S. 100–111 25

[solid IT 2017] IT solid: *DB-Engines Ranking*. `https://db-engines.com/de/ranking`. 2017. – [Online; 22.08.2017] 27

[Klettke 1997] KLETTKE, Meike: Akquisition von Integritätsbedingungen in Datenbanken. (1997)

[Klettke u. a. 2015] KLETTKE, Meike ; STÖRL, Uta ; SCHERZINGER, Stefanie ; REGENSBURG, OTH: Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In: *BTW* Bd. 2105, 2015, S. 425–444 32, 40

[McLeish und Cecile 1989] MCLEISH, Mary ; CECILE, Matt: *UCI Machine Learning Repository*. 1989. – URL `https://archive.ics.uci.edu/ml/datasets/Horse+Colic` 44

[Papenbrock und Naumann 2016] PAPENBROCK, Thorsten ; NAUMANN, Felix: A Hybrid Approach to Functional Dependency Discovery. In: *Proceedings of the 2016 International Conference on Management of Data*. New York, NY, USA : ACM, 2016 (SIGMOD ’16), S. 821–833. – URL `http://doi.acm.org/10.1145/2882903.2915203`. – ISBN 978-1-4503-3531-7 14, 28, 44, 57

[Papenbrock und Naumann 2017] PAPENBROCK, Thorsten ; NAUMANN, Felix: A Hybrid Approach for Efficient Unique Column Combination Discovery. In: *BTW*, 2017, S. 195–204 25, 26

[Pritchett 2008] PRITCHETT, Dan: Base: An acid alternative. In: *Queue* 6 (2008), Nr. 3, S. 48–55 4

[Ritter 2015] RITTER, Norbert: *Taschenbuch Datenbanken, Kapitel 13: Cloud-Datenbanken*. Carl Hanser Verlag GmbH Co KG, 2015 4

[Sattler 2015] SATTLER, Kai-Uwe: *Taschenbuch Datenbanken, Kapitel 7: Komponenten eines Datenbankmanagementsystems*. Carl Hanser Verlag GmbH Co KG, 2015 3

[Sauer 2015] SAUER, Petra: *Taschenbuch Datenbanken, Kapitel 3: Relationales Datenmodell*. Carl Hanser Verlag GmbH Co KG, 2015 9, 10

[Störl 2015] STÖRL, Uta: *Taschenbuch Datenbanken, Kapitel 12: NoSQL-Datenbanksysteme*. Carl Hanser Verlag GmbH Co KG, 2015 4

# Eidesstattliche Versicherung

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.

Die Arbeit ist noch nicht veröffentlicht und ist in gleicher oder ähnlicher Weise noch nicht als Studienleistung zur Anerkennung oder Bewertung vorgelegt worden. Ich weiß, dass bei Abgabe einer falschen Versicherung die Prüfung als nicht bestanden zu gelten hat.

Rostock, 24.08.2017

Hannes Awolin