Universität Rostock

Traditio et Innovatio

INSTITUT FÜR INFORMATIK

Masterarbeit

„Entwicklung und Bewertung von Verfahren zur inkrementellen Schema-Extraktion aus NoSQL-Datenbanken“

eingereicht am 29.03.2016

von

Jacob Langner | Wollenweberstraße 20 | 18055 Rostock

Matrikel-Nr.: 8200329

Gutachter:

PD Dr.-Ing. habil. Meike Klettke

Universität Rostock

Albert-Einstein-Str. 22

18059 Rostock

Zweitgutachter:

Dr. Birger Lantow

Universität Rostock

Albert-Einstein-Str. 22

18059 Rostock

Kurzfassung

NoSQL-Datenbanken zeichnen sich durch flexible oder gar schemalose Datenspeicherung aus, was insbesondere in den frühen Phasen der Anwendungsentwicklung nützlich ist, in denen sich die Struktur der Daten noch häufig ändert. In späteren Phasen können sich die schemalosen Datenbanken jedoch schnell zum Nachteil für die Entwickler wenden. Verschiedene Datenversionen müssen in jedem Schritt des Entwicklungsprozesses beachtet werden. Aus diesem Problem heraus sind neue Ideen, wie die Schema-Evolution, entstanden, die die Migration von semi-strukturierten Daten unterstützen. Allerdings basieren diese Ideen auf einem Schema der zugrundeliegenden Daten. Dieses aus hunderten oder tausenden heterogener Datensätze manuell zu erstellen, ist quasi unmöglich. Es werden computergestützte Verfahren zur Schema-Extraktion benötigt. Die derzeitige Tendenz zu Big Data Anwendungen motiviert dabei einen inkrementellen Ansatz zur Aktualisierung des Schemas. In dieser Masterarbeit werden verschiedene Verfahren zur inkrementellen Schema-Extraktion aus NoSQL-Datenbanken konzipiert, implementiert und evaluiert. Die Implementation wird exemplarisch an einer NoSQL-Datenbank durchgeführt, die Einbindung weiterer NoSQL-Datenbanken, die JSON Dokumente speichern, ist jedoch jederzeit möglich. Die entwickelten Verfahren liefern ein JSON Schema, welches als Ausgangspunkt für die Datenmigration dienen kann. Zusätzlich werden ausgewählte Metriken erhoben, die zur alternativen Datenanalyse verwendet werden können.

Schlagwörter: NoSQL-Datenbanken; JSON; JSON Schema; inkrementelle Schema-Extraktion; Schema-Reengineering

Abstract

NoSQL-databases offer data storage with flexible or no schemas at all, which is especially useful in the early stages of application development, where structural data changes happen frequently. In later stages however schemaless databases can become a hazard for those developers. Multiple versions of data have to be managed and considered in every step of the development process. New ideas like schema-evolution emerged, which support data migration of semi-structured data. Nevertheless those ideas are based upon a schema of the underlying data. Creating this manually out of hundreds or thousands of heterogeneous documents is almost impossible. Automated schema extraction processes are necessary. The current trend to big data applications motivates the use of incremental extraction methods. In this master thesis different approaches to incrementally extract schemas from NoSQL-databases are conceptualized, implemented and evaluated. The implementation focusses on one NoSQL-database but can easily be migrated to every other database that supports JSON documents. The developed methods provide a JSON schema, which can be used as a starting point for data migration. Additional metrics shall provide a possibility for alternative data analysis.

Keywords: NoSQL-databases; JSON; JSON schema; incremental schema extraction; schema reengineering

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Nachdem das Thema dieser Arbeit eingehend motiviert werden soll, wird das Ziel der Arbeit im darauffolgenden Abschnitt genauer abgesteckt. Außerdem werden in diesem Kapitel die angewandten Methoden und die Gliederung der Arbeit erläutert.

1.1 Motivation

NoSQL-Datenbanken erfreuen sich großer Beliebtheit in heutigen Software- und Webentwicklungen. Durch die gebotene Flexibilität und den geringen Einrichtungsaufwand stellen sie eine attraktive Alternative zu herkömmlichen, relationalen Datenbanken dar. Der Wegfall einer aufwendigen Schema-Definition zu Beginn des Entwicklungsprozesses ermöglicht einen schnelleren Einstieg in die eigentliche Entwicklungsarbeit. Zudem unterliegt die Datenstruktur während der Entwicklung häufig vielen Änderungen, die sich auch auf den NoSQL-Datenbestand auswirken. Ein anfangs entwickeltes Schema ist daher sehr kurzlebig und muss mit jeder Änderung erneut angepasst werden, deren Umsetzung die Entwickler weiter von ihrer eigentlichen Arbeit abhält. In NoSQL-Datenbanken gibt es oftmals kein Schema, so dass die Daten jederzeit verändert werden können. Dadurch entsteht nicht nur die sehr hohe Flexibilität, sondern es entfällt auch der Aufwand für Schema-Definition und Schema-Änderungen während der Entwicklung. Allerdings wendet sich dieser Vorteil in fortgeschrittenen Entwicklungsphasen oft zum Nachteil, da die Heterogenität der Daten zunimmt und kein einheitliches Format vorliegt. Es entstehen Probleme durch veraltete Daten und komplizierte Versionierungsprozesse. So müssen die Entwickler sicherstellen, dass ihre Anwendung mit allen vorhandenen Datenversionen in der Datenbank kompatibel ist. Eine Möglichkeit bietet die Migration der Daten von den alten auf die aktuelle Version. Dies ist als *„Bulk-Update“*, wobei alle Daten auf einmal auf den neusten Stand gebracht werden, möglich. Insbesondere bei Webanwendungen, die rund um die Uhr verfügbar sein sollen, ist eine so genannte *„Eager Migration“* jedoch oft schwierig. Eine Alternative zum *Bulk-Update* bietet die oftmals ressourcensparendere *„Lazy Migration“*, bei der der Migrationsprozess erst beim Zugriff auf die veralteten Daten gestartet wird. Veraltete Daten werden also erst dann migriert, wenn sie benötigt werden. Nicht mehr benötigte Daten verursachen bei dieser Variante keine Kosten, da sie nicht migriert werden. In [1] wird eine Schema-Evolutionssprache vorgestellt, die die Migration von NoSQL-Datenbeständen ermöglicht. Ausgangspunkt ist jedoch auch dort ein Schema der Daten. Eine manuelle Erstellung würde den durch die Verwendung von schemalosen Datenbanken vermeintlich eingesparten Aufwand der Schema-Definition lediglich nach hinten verschieben. Es bedarf automatisierter Verfahren zur Schema-Extraktion aus bestehenden Datenbeständen. Ansätze solcher Verfahren existieren bereits. [2–5] Aus demselben Grund, aus dem die *Lazy Migration* eingesetzt wird, sind insbesondere bei der Entwicklung von heutigen Big Data- und Webanwendungen jedoch inkrementelle Verfahren zur Schema-Extraktion gefordert. Diese Anwendungen erzeugen schnell sehr große Datenbestände, auf denen dauerhaft viele, geringfügige Änderungen und Aktualisierungen vorgenommen werden. Die inkrementellen Verfahren sollen effizient mit diesen Updates umgehen können und das Schema aktualisieren ohne den kompletten Datenbestand erneut durchlaufen zu müssen. Ziel dieser Arbeit ist es daher, verschiedene Verfahren zur inkrementellen Schema-Extraktion zu entwickeln und zu bewerten. Mit Blick auf die möglichen Einsatzgebiete in der Software- und Webentwicklung sollen zudem hilfreiche Metriken abgeleitet werden, die zur alternativen Datenanalyse verwendet werden können. Dazu

zählen beispielsweise Ausreißerdokumente, also einige wenige Dokumente, die sich in ihrer Struktur von der Masse sonst homogener Dokumente unterscheiden. Im Fall der Anwendungsentwicklung mit *Lazy Migration* könnten diese Ausreißerdokumente veralteten, nicht migrierten Daten entsprechen. Durch die Identifikation solcher Dokumente soll wertvoller Speicherbedarf eingespart werden, indem beispielsweise alte und nicht mehr benötigte Daten archiviert oder endgültig gelöscht werden.
Neben diesen explizit abgeleiteten Metriken ergeben sich aus einem Schema selbst bereits implizite Vorteile. So erleichtert ein Schema zum Beispiel die Definition von Schnittstellen für den Datenimport und -export in Anwendungen. Zudem gibt es den Entwicklern einen Überblick über die Struktur der zur Verfügung stehenden Daten in ihren NoSQL-Datenbanken und ist als Ausgangspunkt der Schema-Evolution einer erster Schritt hin zur nachträglichen Organisation des eigenen Datenbestands.

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist es, Verfahren zur inkrementellen Ableitung von Schemainformationen aus NoSQL-Daten zu entwickeln und anschließend zu bewerten. Zu diesem Zweck soll ein Algorithmus entworfen und anschließend in Java implementiert werden, der aus JSON Daten Schemainformationen ableiten kann. Dieser Algorithmus soll zum einen Schemata aus JSON Datenbeständen extrahieren und zum anderen bereits abgeleitete Schemata inkrementell aktualisieren können. Die Aktualisierung soll sowohl durch die Eingabe geänderter Dokumente als auch durch ein Update-Log durchgeführt werden können. Die Ausgabe soll als wohlgeformtes und valides JSON Schema erfolgen. Zusätzlich sollen Metriken, wie zum Beispiel Ausreißerdokumente, aus den Daten abgeleitet werden.

1.3 Methodik

Die Arbeit gliedert sich in unterschiedliche Teilbereiche. Ein Einstieg soll durch die Sichtung relevanter Literatur erfolgen, die in einem Grundlagen-Kapitel aufgearbeitet darzustellen ist. Anschließend gilt es, die Anforderungen an den zu entwickelnden Algorithmus zu spezifizieren und daraus ein Konzept zur Umsetzung abzuleiten. Dieses Konzept soll anschließend implementiert und die Implementation getestet und bewertet werden. Die Softwareentwicklung ist jedoch in der Regel kein geradliniger Prozess und benötigt oftmals mehrere Iterationen aus Spezifikation, Implementation, Test und Bewertung. Aufgrund der geringen Teamgröße und der Ungewissheit über Frequenz und Umfang auftretender Probleme soll nach dem agilen Vorgehensmodell des „*Extreme Programming*“ [6] gearbeitet werden. Wöchentliche Rücksprachen, zahlreiche Prototypen und frühzeitige Tests sollen das Erreichen der gesetzten Projektziele sicherstellen.

1.4 Aufbau der Arbeit

Die Arbeit ist in sechs Kapitel gegliedert. Folgend auf die in diesem Kapitel gegebene Einleitung zum Thema, soll Kapitel 2 die Grundlagen zum Verständnis der in Kapitel 3 konzipierten, inkrementellen Schema-Extraktionsverfahren schaffen. Das entwickelte Konzept wird in Kapitel 4 implementiert und anschließend in Kapitel 5 getestet und bewertet. Ein abschließendes Fazit der entwickelten Verfahren wird in Kapitel 6 gezogen und durch einen Ausblick ergänzt.

2 Grundlagen

Dieses Kapitel behandelt das Grundlagenwissen und die Vorüberlegungen, die dieser Arbeit vorausgegangen sind. Im ersten Abschnitt wird der Begriff der semi-strukturierten Daten erläutert, deren Struktur eine zentrale Rolle in dieser Arbeit spielt. Semi-strukturierte Daten können beispielsweise in NoSQL-Datenbanken, die im zweiten Abschnitt vorgestellt und klassifiziert werden, im JSON Format gespeichert werden. Die drei darauffolgenden Abschnitte beschäftigen sich entsprechend mit den Begriffen Schema, JSON und JSON Schema. JSON ist ein Datenformat zur Speicherung und zum Austausch von semi-strukturierten Daten. Die in dieser Arbeit exemplarisch verwendete NoSQL-Datenbank speichert Daten in Form von JSON Dokumenten. JSON Schema ist die entsprechende Schemasprache zu JSON. Verwandte Schema-Extraktionsverfahren aus dem XML-Bereich werden im letzten Abschnitt dieses Kapitels vorgestellt.

2.1 Semi-strukturierte Daten

Unter dem Begriff der semi-strukturierten Daten wird die Art von Daten verstanden, die weder gänzlich unstrukturiert noch komplett strukturiert sind. Die Übergänge zwischen den Arten sind dabei fließend und nicht immer eindeutig zu bestimmen. In [7] wird der Begriff der semi-strukturierten Daten durch folgende Charakteristiken näher eingegrenzt:

- Unregelmäßige Struktur der Daten
- Implizites Schema in den Daten
- Unvollständige Struktur der Daten
- Flexible Struktur der Daten
- Keine Schemadefinition im Voraus zwingend
- Das Schema ist groß
- Das Schema ist veränderbar
- Die Struktur der Datenelemente selbst ist veränderbar
- Die Unterscheidung von Daten und Schema ist unscharf

Im Gegensatz zu strukturierten Daten, wie sie zum Beispiel in relationalen Datenbanken zu finden sind, sind semi-strukturierte Daten also flexibler. Ihre Struktur und somit auch ihr Schema können sich im Nachhinein ändern. Weiterhin ist das Schema implizit durch die Daten vorgegeben. Dies kann zum Beispiel durch Kennzeichnungen von Elementen im Dokument geschehen. Diese implizite Struktur grenzt die semi-strukturierten Daten wiederum von den unstrukturierten Daten ab, denen jegliche Strukturinformationen fehlen.

In [7] wird weiter ein Modell zur Repräsentation von semi-strukturierten Daten – das *Object Exchange Model* (OEM) – vorgestellt. Dieses Modell wurde ursprünglich in [8] entwickelt und beschreibt semi-strukturierte Daten als gerichtete Graphen, die aus Knoten, die die Datenobjekte repräsentieren, und beschrifteten Kanten, den Attributen der Objekte, bestehen. Objekte sind durch eine ID eindeutig und stellen entweder atomare Datentypen, wie zum Beispiel Integer oder Strings, oder komplexe Datentypen dar. Komplexe Datentypen sind wiederum Verweise auf weitere Objekte in der Form von Label-Wert-Paaren, die um die Objekt-ID und eine Typdefinition ergänzt werden.

Der so entstehende Graph kann sowohl strukturierte als auch semi-strukturierte Daten repräsentieren. Dabei werden die Label-Wert-Paare so dargestellt, dass die Label die Beschriftungen der Kanten, also die Attribute des Objekts, und die Werte die Knoten des

Graphen, also (Kind)-Objekte, sind. Die Knoten werden durchnummeriert, wodurch sie die eindeutige ID zugewiesen bekommen. Blätter des Graphen stellen atomare Datentypen dar. Die inneren Knoten sind komplexe Datentypen, die per Definition mindestens eine ausgehende Kante, also ein Kindelement, besitzen. Der Graph kann Zyklen enthalten und Kantenbeschriftungen, also die Namen der Elemente, müssen nicht eindeutig sein. Dies ermöglicht die Modellierung von optionalen Elementen und Strukturunterschieden zwischen namentlich gleichen Elementen in den zugrundeliegenden Daten. Abbildung 1 zeigt ein Beispiel eines OEM-Graphen. Es gibt drei Restaurant-Elemente zu denen jeweils unterschiedliche Informationen vorliegen. Zum Beispiel unterscheidet sich der Datentyp der Adressen. So kann die Adresse als komplexer Datentyp aus Straße, Stadt und Postleitzahl bestehen, ein einfacher String sein oder gänzlich fehlen. Die Postleitzahl kann sich auch direkt unter dem Restaurant-Objekt befinden, falls sie nicht in der Adresse aufgeführt ist. Auch die Angabe der Preisinformation ist optional. Dagegen besitzt jedes Restaurant-Objekt eine Kategorie und einen Namen. [7]

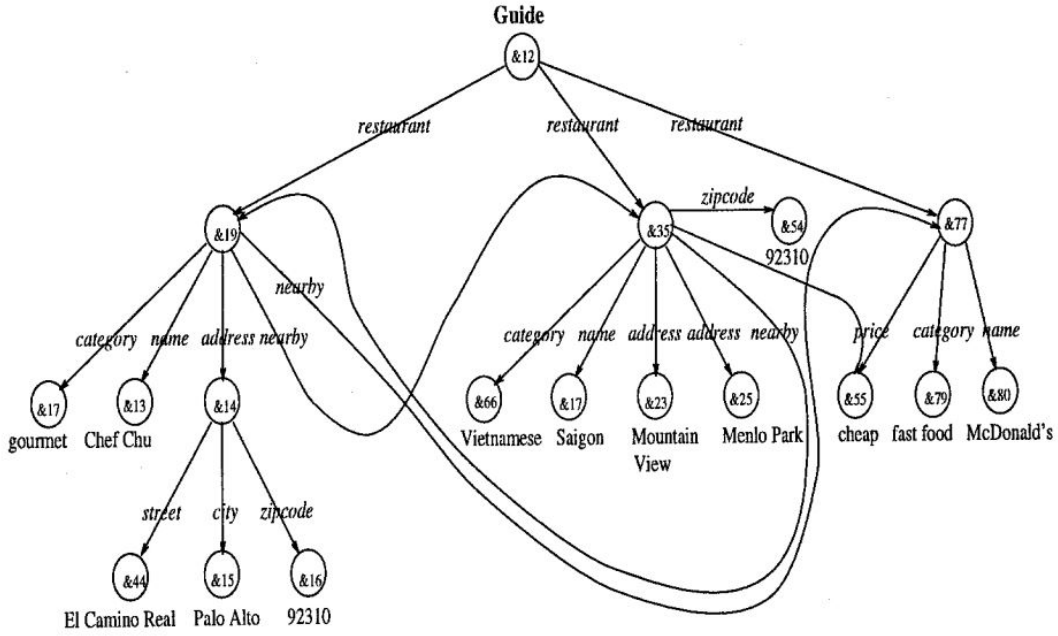

Abbildung 1: Beispiel eines OEM-Graphen ([7] S. 10)

Diese Graph-Repräsentation hat sich weitestgehend als Standard-Modell zur Darstellung von semi-strukturierten Daten etabliert. Insbesondere für die Ableitung von Schema-Informationen aus bestehenden Daten ist die Darstellung als Baum relevant. Formal lässt sich der Graph wie folgt definieren: [9]

$$G = (V, E, r_1, \dots, r_k, v)$$

mit

- $V = V_c \cup V_a$ als Knotenmenge aus der Vereinigung von komplexen Objekten V_c und atomaren Objekten V_a
- $E \subseteq V_c \times A \times V$ als Menge aller Kanten zwischen den Objekten und A als Menge aller Attribute
- r_i als Menge der Wurzelelemente
- $v : V_a \rightarrow D$ als Abbildung der atomaren Objekte auf die Menge aller atomaren Werte

2.2 NoSQL-Datenbanken

NoSQL-Datenbanken sind Datenbanken, die sich vom starren, relationalen Ansatz abgrenzen. Entstanden sind sie aus dem Bedürfnis heraus, große Mengen heterogener Daten zu speichern. Insbesondere durch Webanwendungen, bei denen es oft zu mehreren Millionen Lese- und Schreiboperationen durch verschiedene Nutzer kommt, versagen die relationalen Datenbanken. An dieser Stelle zeichnen sich die NoSQL-Datenbanken durch sehr gute horizontale Skalierbarkeit aus. Durch Aufgabe der ACID Eigenschaften – im Wesentlichen der Datenkonsistenz - erreichen sie ein sehr hohes Maß an Performance, Erreichbarkeit, Flexibilität und Skalierbarkeit. [10, 11] Getrieben wird die Entwicklung der NoSQL-Datenbanken besonders durch die großen Internetdienstleister wie Google und Amazon, die in ihren Anwendungen überall auf der Welt Daten sammeln, verarbeiten und speichern müssen. [10]
NoSQL-Datenbanken sind oft schemalos oder besitzen flexible Schemata. Elemente können somit auf der Datenebene beliebig hinzugefügt, verändert oder gelöscht werden. [10]
NoSQL-Datenbanken lassen sich durch verschiedene Kriterien klassifizieren. So unterscheiden sie sich zum Beispiel durch ihr Datenmodell, den Umfang ihrer Anfragesprache, ihre Skalierbarkeit, ihre Architektur und ihre Persistenz-Implementation. [1] Für diese Arbeit relevant ist die Unterscheidung nach dem Datenmodell. Es lassen sich die folgenden drei Datenmodelle unterscheiden: [11]
Key-Value-Stores speichern Schlüssel-Wert-Paare. Anfragen sind allerdings nur auf den eindeutigen Schlüsseln möglich, da im Wertebereich beliebige Objekte gespeichert werden können, die von den Systemen nicht weiter interpretiert werden. Wichtige Vertreter sind zum Beispiel *Redis, Riak* oder *Scalaris*. [10, 11]
Document Stores speichern ebenfalls Schlüssel-Wert-Paare. Im Gegensatz zu den Schlüssel-Wert-Datenbanken handelt es sich bei den Werten allerdings immer um Dokumente, die eine Verschachtelung von Werten und somit komplexere Datenstrukturen ermöglichen. Ein Dokument kann weitere Dokumente beinhalten. JSON ist ein hierfür verwendetes Datenformat, welches das Konzept der Dokumente unterstützt. In Dokument-Datenbanken kann nicht nur nach Schlüsseln, sondern auch nach Objekten mit gewissen Attributen gesucht werden. *MongoDB* und *CouchDB* sind Beispiele für *Document Stores*. [10, 11]
In ***Extensible Record Stores*** werden die Daten in Form von Reihen und Spalten gespeichert. Skalierbarkeit wird durch das Aufteilen dieser Reihen und Spalten erreicht. Beim Aufteilen der Reihen werden Bereiche nach ihrem Primärschlüssel getrennt und auf verschiedene Server verteilt. Spalten werden zu oft zusammen benötigten Spaltengruppen gruppiert. Googles *Big Table, HBase* und *Cassandra* sind *Extensible Record Stores.* [10, 11]
In dieser Arbeit wird exemplarisch mit *MongoDB* als einem Vertreter der *Document Stores* gearbeitet. *Document Stores* bieten durch das Konzept der Dokumente eine hohe Synergie mit dem verwendeten Datenformat JSON. *MongoDB* bietet zusätzlich den Vorteil, dass eindeutige Dokument-IDs vom System erzwungen werden, die zur Identifikation der Dokumente benötigt werden. Die Anwendung der Verfahren ist allerdings nicht an die *Document Stores* gebunden, da *Extensible Record Stores* gleichmächtig zu den *Document Stores* sind und sogar *Key-Value-Stores* eingesetzt werden können, falls diese JSON Dokumente im Wertebereich speichern. Eine einfache Anbindung weiterer Datenbanken ist durch das datenbankunabhängige Datenformat JSON gewährleistet. [1]

2.3 Schemata

Ein Schema erfasst die Struktur von Wissen oder Daten indem es für einen bestimmten Gegenstandsbereich die wichtigen Merkmale hierarchisch organisiert und auf einer bestimmten Abstraktionsebene darstellt. [12] In der Informatik werden Schemata unter anderem zur Datenorganisation eingesetzt. Dabei wird zu einer Sprache, wie z.B. XML oder JSON, eine Beschreibungs- oder Schemasprache definiert, mit der die Struktur der beschriebenen Daten erfasst werden kann. Schemata können als Vorlage bzw. Muster zur Erstellung neuer Daten verwendet werden, um die durch die Anforderungen definierte Struktur der Daten zu garantieren. Dies spielt insbesondere bei der Anwendungsentwicklung und beim Datenaustausch zwischen verschiedenen Programmen eine Rolle.
In relationalen Datenbanken wird die Struktur der Daten, also das Schema, durch Tabellen und Spaltenköpfe sowie weitere Restriktionen fest vorgegeben. Ohne Festlegung der Spalten, Wertebereiche, etc. können keine Daten in die Tabelle eingefügt werden – ein Schema muss also stets im Voraus definiert werden.
Wie bereits in Abschnitt 2.2 beschrieben, gibt es jedoch auch Datenbanken, die keine Schemadefinition erfordern. Für den Fall, dass im Nachhinein ein Schema aus bereits bestehenden Daten abgeleitet werden muss, gibt es Verfahren, die unter dem Begriff des *Schema-Reengineering* bzw. *Schema-Reverse-Engineering* zusammengefasst werden. Verfahren, die die Übereinstimmung von Schema und Daten überprüfen, werden Schema-Validierungsverfahren genannt.

2.3.1 Schema-Validierung

Schema-Validatoren sind Algorithmen, die eine automatische Schema-Validierung von großen Datenmengen ermöglichen. Die Validator-Algorithmen überprüfen neben der Einhaltung der Syntaxregeln der verwendeten Sprache auch die Vorgaben des Schemas. Vorgaben können dabei zum Beispiel Pflichtelemente, erlaubte, optionale Elemente, bestimmte Datentypen oder andere Beschränkungen und Anforderungen an einzelne Strukturelemente sein. Ein Dokument, welches den Syntaxregeln der Sprache entspricht, wird als wohlgeformt bezeichnet, erfüllt es zudem alle Anforderungen eines Schemas, ist es diesem Schema gegenüber valide. Dokumente, die von einem Validator erfolgreich geprüft werden, sind sowohl wohlgeformt als auch valide. Ein Validator überprüft das Dokument elementweise. Die für jedes Element ermittelten lokalen Validitäten werden anschließend durch Aggregation zur Dokumentvalidität zusammengesetzt. Nur falls jedes Element lokal valide ist, ist auch das Dokument als Ganzes valide. Zur lokalen Überprüfung wird das Element mit allen Attributen gegen die Elementdefinition im Schema geprüft. Entspricht das Element allen spezifizierten Beschränkungen, wird der lokale Validationsprozess erfolgreich abgeschlossen. [13]

2.3.2 Schema-Reengineering

Schemata können nicht nur als Vorlage zur Erzeugung neuer Daten verwendet werden, sondern auch aus bereits bestehenden Daten abgeleitet werden. Dieser Prozess wird als *Schema-Reengineering* bezeichnet. Dabei wird eine festgelegte Dokumentmenge durchlaufen und die Struktur der einzelnen Dokumente im Schema festgehalten. Auf diese Weise können Informationen über die Struktur bereits bestehender Daten extrahiert werden, die bei einem semi-strukturierten Datenbestand sonst nicht ersichtlich wären. Das Schema muss dabei für alle Dokumente valide sein. Demzufolge sind bei heterogenen Daten Verallgemeinerungen im Schema notwendig – dies wird als *Generalisierung* bezeichnet. Die Umkehrung zur

Generalisierung ist die *Spezialisierung*, bei der dem Schema Anforderungen hinzugefügt werden, es also einschränkender wird. Beispiele dafür sind das Hinzufügen von Pflichtelementen, das Verfeinern von Datentypen oder das Setzen von zusätzlichen Restriktionen. Bei der Generalisierung werden unter anderem Datentypen verallgemeinert oder Elemente als optional deklariert. Häufig werden dazu heuristische Verfahren eingesetzt, die entweder auf syntaktischer oder auf semantischer Ebene schlussfolgern. Auf syntaktischer Ebene können neben der direkten Übereinstimmung zweier Elemente auch lexikografische Methoden oder Vergleiche auf der Basis von regulären Ausdrücken durchgeführt werden. Die semantische Ebene ist meist wesentlich komplizierter und benötigt in der Regel fachgebietsabhängiges Spezialwissen, zum Beispiel eine Taxonomie, um solche Verfahren zu implementieren.
Herkömmliche Schemata, insbesondere in der Welt der relationalen Datenbanken, rühmen sich oft durch ihren restriktiven Charakter. Dies liegt darin begründet, dass sie im Voraus definiert werden können und die erwünschten Daten möglichst exakt beschrieben werden sollen. Beim Reverse Engineering aus bestehenden Daten führt eine hohe Exaktheit zu sehr umfangreichen Schemata, die schnell größer als die Daten selbst werden. In der Literatur sind daher andere Qualitätskriterien für die Schema-Extraktion zu finden: [3, 4, 14, 15]
Der Anspruch auf *Korrektheit* ist implizit oder explizit in jedem Ansatz zu finden, da es zum einen das Ziel der Schema-Extraktion ist, die zugrundeliegenden Daten zu repräsentieren. Dies impliziert, dass die Daten in Bezug auf das extrahierte Schema valide sein müssen. Zum anderen verfällt durch ungültige Angaben oder Schlüsse auch die Glaubwürdigkeit eines Schemas.
Der Anspruch auf *Präzision* stammt aus der herkömmlichen Schema-Definition und stellt oft die Motivation zur Erstellung eines Schemas dar. Die zugrundeliegenden Daten sollen möglichst exakt beschrieben werden, damit diese zum einen sehr gut durch das Schema repräsentiert und zum anderen nicht zutreffende Daten durch die Validierung erkannt werden können.
Dagegen wird jedoch bei der Schema-Extraktion der Anspruch auf *Prägnanz* gestellt. Ein erzeugtes Schema soll die zugrundeliegenden Daten möglichst kurz und knapp beschreiben. Dazu werden oftmals Methoden der Generalisierung gewählt, die aus verschiedenen Einzelfällen einen generalisierten Gesamtfall schließen. Dieser ist in der Regel nur eine Annahme, vereinfacht das Schema aber meist sehr stark. Daher werden Präzision und Prägnanz oft in Widerspruch zueinander gestellt und ein geeignetes Mittelmaß zwischen ihnen gesucht.
Weniger frequent ist der Anspruch nach *Lesbarkeit*. Dieser besagt, dass ein extrahiertes Schema gut lesbar und möglichst ähnlich zu dem Schema sein soll, das ein menschlicher Schema-Designer erstellen würde.

2.4 JSON

JSON ist ein maschinenlesbares und programmiersprachenunabhängiges Datenformat, das zum Datenaustausch zwischen Anwendungen konzipiert wurde. Das Akronym steht für *JavaScript Object Notation*. Die erste JSON Version wurde im Jahr 2001 veröffentlicht. JSON zeichnet sich durch ein einfaches, text-basiertes Konzept mit wenigen Regeln und Datentypen aus, was zu einem zeitlich sehr stabilen Standard führt. [16]

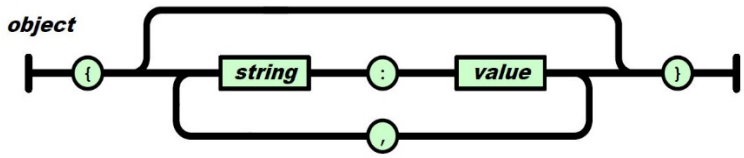

Abbildung 2: JSON Objekt [17] S. 2

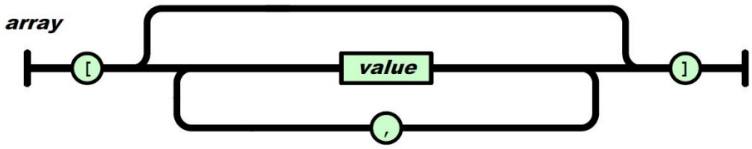

Abbildung 3: JSON Array [17] S. 3

Die JSON Syntax ist sehr einfach gehalten. Neben vier einfachen Datentypen (`String`, `Number`, `Boolean`, `Null`) gibt es nur zwei komplexe Datentypen. Ein `JSON Object` besteht aus keinem, einem oder mehreren ungeordneten Schlüssel-Wert-Paaren. Schlüssel müssen als auf ihrer Ebene eindeutige Strings angegeben werden, Werte können dagegen jeden beliebigen JSON Datentyp annehmen. Ein Wert kann demzufolge wieder ein `JSON Object`, also kein, ein oder mehrere Schlüssel-Wert-Paare sein, wodurch verschachtelte Strukturen ermöglicht werden. Ein `JSON Array` speichert keine, einen oder mehrere geordnete Werte, die unterschiedliche Datentypen besitzen können. [16]

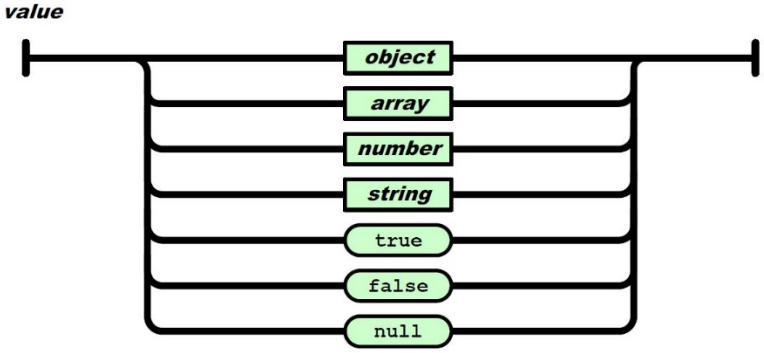

Abbildung 4: JSON Werte [17] S. 2

Weitere Syntaxregeln beschreiben zum Beispiel die korrekte Klammerung von JSON Elementen und die Zusammensetzung von `String`- und `Number`-Elementen. Für JSON gibt es eine Vielzahl von Parsern, die für ihre jeweilige Programmiersprache das Konvertieren von JSON Zeichenketten in die entsprechenden JSON Datentyp-Objekte ermöglichen.

```
{
   "cities" : [ {
      "name" : "Rostock",
      "inhabitants" : 202887,
      "zipcodes" : [18055, 18057, 18059, 18069, 18106, 18107, 18109,
                    18119, 18146, 18147, 18181, 18182],
      "state" : "Mecklenburg-Vorpommern",
      "area" : "181,4 km²",
   }, {
      "name" : "Schwerin",
      "inhabitants" : 102878,
      "zipcodes" : [19053, 19055, 19057, 19059, 19061, 19063],
      "state" : "Mecklenburg-Vorpommern",
      "area" : "130,5 km²",
      "capitol" : true,
   } ]
}
```

Abbildung 5: Beispiel JSON Dokument

Abbildung 5 zeigt ein Beispiel JSON Dokument. Strikte Parser erwarten mindestens ein Schlüssel-Wert-Paar auf der obersten Ebene eines JSON Dokuments. Ein einfacher JSON Primitive wird oft nicht akzeptiert. Ein JSON Dokument sollte also, bedingt durch die geschweiften Klammern, die den Anfang und das Ende des Dokuments kennzeichnen, stets mit einem `JSON Object` beginnen. Die Elemente dieses Wurzel-Objekts sind dann beliebig. Im Beispiel enthält es ein `JSON Array` – `cities` – mit zwei `JSON Objects`, die jeweils mehrere, kommagetrennte Schlüssel-Wert-Paare mit primitiven Datentypen und einem weiteren Array im Wertebereich besitzen. JSON Dokumente können in *Document Stores* als Dokumentkollektionen gespeichert werden.

2.5 JSON Schema

JSON Schema ist, ähnlich wie XML Schema für XML, eine Schemasprache für JSON. Mit JSON Schema kann die Struktur von JSON Daten festgelegt und anschließend validiert werden. Neben der Validierung dient JSON Schema auch zur Dokumentation, Hyperlink-Navigation und Interaktionskontrolle mit JSON Daten. [18]

Insbesondere in der Anwendungsentwicklung finden Schemata eine Anwendung, da sie die Anforderungen an die Struktur der akzeptierten Daten kennzeichnen und anschließend die eingehenden Daten auf Einhaltung der Strukturanforderungen validieren können. Zu diesem Zweck wurde eine Vielzahl von JSON Schema Validatoren entwickelt, die JSON Daten in den verschiedenen Programmiersprachen auf ihre Struktur überprüfen können. [19]

In der JSON Schema Spezifikation [18] werden unter anderem Schlüsselwörter für die Deklaration von Inhalten von `JSON Objects` (`property` oder `member`) und `JSON Arrays` (`item` oder `element`) festgelegt. Weiterhin werden die sieben JSON Datentypen deklariert: `Array`, `Boolean`, `Integer`, `Number`, `Null`, `Object` und `String`. [18]

Abbildung 6 zeigt ein mögliches Schema zu dem in Abbildung 5 gezeigten JSON Dokument. Ein Dokument der `cities collection` muss ein Array mit mindestens zwei Objekten, also einer Liste aus mindestens zwei Städten, besitzen. Für jede Stadt muss ein Name, die Einwohnerzahl, eine Liste mit den Postleitzahlen, das Bundesland und die Fläche angegeben werden. Optional ist der `capitol`-Schlüssel, der darauf verweist, dass die Stadt die Hauptstadt des entsprechenden Bundeslandes ist.

```
{
   "$schema": "http://json-schema.org/draft-04/schema#",
   "title" : "cities collection",
   "description": "JSON schema for the city document collection",
   "type": "object",
   “properties” : {
      “cities” : {
         “type” : “array”,
         "items": {
            "type" : "object",
            "description" : "important facts about the city",
            "properties" : {
               "name" : {
                  "type" : "string"
               },
               "inhabitants" : {
                  "type" : "number"
               },
               "zipcodes" : {
                  "type" : "array",
                  "items" : {
                     "type" : "number"
                  }
               },
               "state" : {
                  "type" : "string"
               },
               "area" : {
                  "type" : "string"
               },
               "capitol" : {
                  "type" : "boolean"
               }
            },
            "required" : ["name", "inhabitants", "zipcodes", "state",
               "area"]
         },
         "minItems" : 2
      }
   },
   “required” : [“cities”]
}
```

Abbildung 6: Beispiel JSON Schema

2.6 Related Work

Für die zu entwickelnden Schema-Extraktionsverfahren wurde in der Literatur nach ähnlichen Verfahren oder Ansätzen gesucht. Für die Schema-Extraktion in JSON wurden keine etablierten Verfahren gefunden. Lediglich ein Prototyp einer vereinfachten Schema-Extraktion in Python, auf den diese Arbeit aufbaut, konnte als Grundlage verwendet werden. Daher wurden Schema-Extraktionsverfahren ähnlicher Sprachen analysiert. Für XML konnten einige Verfahren ausfindig gemacht werden, die hier, nach einer kurzen Gegenüberstellung von XML Schema und JSON Schema, vorgestellt werden sollen.

2.6.1 XML und XML Schema

XML oder *Extensible Markup Language* ist das wohl bekannteste Datenformat für semi-strukturierte Daten. Wie JSON auch, besteht XML aus möglicherweise verschachtelten Elementen. Elemente werden durch sogenannte Tags gekennzeichnet. Weitere Attribute können in den Tags definiert werden.

Für XML gibt es zahlreiche Schemasprachen, die sich mit der Beschreibung der Struktur von XML Dokumenten beschäftigen. Neben dem vom *W3C* entwickelten *XML Schema* [13] gibt es

zum Beispiel die *Document Type Definition (DTD)* [20] oder *RELAX NG* [21]. DTDs können direkt im XML Dokument spezifiziert werden, sind selbst aber nicht in XML geschrieben. Zudem ist ihre Ausdrucksstärke durch fehlende Datentypen und Namensräume geringer als die von XML Schemata.

Die *XML Schemasprache*, auch *XML Schema Definition (XSD)* genannt, ist eine Schemasprache für XML, die sich selbst der XML Syntax bedient. Eine XML Schemadefinition lässt sich in die drei Hauptbestandteile Elemente, Attribute und Restriktionen untergliedern. Im Bereich der Elemente kann zwischen einfachen und komplexen Elementdefinitionen unterschieden werden. Komplexe Elemente sind Containerstrukturen, die weitere Elemente und Attribute beinhalten können. Sie ermöglichen sehr flexible Typdefinitionen. Attribute können Bestandteil eines komplexen Elements sein, sind selbst aber stets einfache Elemente. Sie zeichnen sich durch einen Namen und Datentyp aus, wobei dieser einer der XML Standarddatentypen ist. Attribute können um Restriktionen ergänzt werden. Diese Restriktionen sind sehr vielseitig und tragen wesentlich zur Komplexität und Ausdruckstärke von XML Schema bei. Es können beispielsweise Pflichtelemente, Standardwerte, feste Werte, Minimal- und Höchstwerte oder eine Auswahl möglicher Werte festgelegt werden. Für viele dieser Komponenten lässt sich ein Äquivalent in JSON finden. So sind die einfachen Typdefinitionen mit den `JSON Primitives` vergleichbar, wobei JSON zwischen weniger Datentypen unterscheidet als XML. So werden zum Beispiel fast alle Zahlenformate zum `Number`-Datentyp zusammengefasst, wohingegen in XML zwischen `xs:decimal`, `xs:integer`, `xs:float` und im weiteren Sinne `xs:date` und `xs:time` unterschieden wird. XML Schema unterstützt ein Referenzprinzip zur Verlinkung von Inhalten – insbesondere `ID` und `IDREF`, die vergleichbar mit JSONs `$ref`-Mechanik sind. Neben den einfachen Datentypen gibt es in beiden Schemasprachen die Möglichkeit komplexe Datenstrukturen zu definieren. In XML Schema wird ein `xs:complexType` definiert, dessen Inhalt dann durch `xs:sequence`, `xs:choice` oder `xs:all` spezifiziert werden kann. `JSON Array` und `xs:sequence` sind vergleichbar, da bei diesen Strukturen die Reihenfolge der Kindelemente ein Rolle spielt. Ebenso lassen sich das `JSON Object` und `xs:all` vergleichen, bei denen die Reihenfolge jeweils keine Rolle spielt. Die `xs:choice` kann in JSON durch die Definition eines `oneOf`-Konstrukts dargestellt werden. Allerdings haben diese komplexen Datentypen auch Unterschiede. So erzwingt das `xs:all`-Konstrukt eine Kardinalität der Kindelemente von Null oder Eins, beim `JSON Object` sind alle Elemente optional, es sei denn, sie werden als Pflichtelement im `required`-Array deklariert. Es gibt also in beiden Sprachen verschiedene, aber doch ähnliche Konstrukte, deren Kombination vielseitige Typdefinitionen ermöglicht. XML Schema bietet zudem die Möglichkeit aus bereits definierten Typen durch `xs:extensions` und `xs:restrictions` neue Typen abzuleiten. Neben den Typdefinitionen sind Constraints ein weiteres, wichtiges Mittel zur Schemadefinition, die in beiden Sprachen zur genaueren Einschränkung der definierten Typen dienen. Häufig verwendete Constraints sind in beiden Sprachen zu finden, wie zum Beispiel Kardinalitäten zur Beschränkung der Häufigkeit von Elementen oder Längenangaben zur Einschränkung der Wertebereiche. [13]

Eine Überprüfung der Wohlgeformtheit und Gültigkeit von XML Daten gegen ein XML Schema durch XML Validatoren ist ebenso möglich wie die Überprüfung von JSON Daten gegen ein JSON Schema durch JSON Validatoren. [13]

Abschließend sollen die Spezifikationen im Allgemeinen betrachtet werden. Die XML Schema Spezifikation ist wesentlich umfangreicher als die JSON Schema Spezifikation. Vor allem die Konzepte zur Wiederverwendung von Definitionen oder ganzen Schemata sind in JSON weniger stark ausgeprägt. Die Fragestellungen nach der Mächtigkeit beider Spezifikationen sowie ein Vergleich dieser sollen allerdings nicht in dieser Arbeit geklärt werden. Beide Schemasprachen werden kontinuierlich und bei Bedarf weiterentwickelt, überarbeitet oder ergänzt.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.w3schools.com"
xmlns="http://www.w3schools.com"
elementFormDefault="qualified">

<xs:element name="cities">
   <xs:complexType>
      <xs:sequence>
         <xs:complexType name="city" minOccurs="2"/>
            <xs:all>
               <xs:element name="name" minOccurs="1" type="xs:string"/>
               <xs:element name="inhabitants" minOccurs="1" type="xs:integer"/>
               <xs:complexType name="zipcodes">
                  <xs:sequence>
                     <xs:element name="zipcode" type="xs:integer"/>
                  </xs:sequence>
               </xs:complexType>
               <xs:element name="state" minOccurs="1" type="xs:string"/>
               <xs:element name="area" minOccurs="1" type="xs:string"/>
               <xs:element name="capitol" type="xs:boolean"/>
            </xs:all>
         </xs:complexType>
      </xs:sequence>
   </xs:complexType>
</xs:element>

</xs:schema>
```

Abbildung 7: Beispiel XML Schema

Zum Vergleich stellt Abbildung 7 das in Abbildung 6 dargestellte JSON Schema Beispiel in XML Schema dar.

2.6.2 Extraktionsverfahren

In der Literatur gibt es eine Reihe von Schema-Extraktionsverfahren. Die bekanntesten Verfahren, darunter *DTD-Miner* [2] und *XTRACT [4]*, sowie alle weiteren vorgestellten Verfahren dienen der Schema-Extraktion für XML Dokumente, da XML der mit Abstand meist verbreitete Standard für semi-strukturierte Daten und somit auch Hauptgegenstand der Forschung ist. Aufgrund der Ähnlichkeiten zwischen JSON und XML lassen sich die Verfahren mit einigen Anpassungen jedoch auch auf JSON übertragen.

DTD-Miner

Der *DTD-Miner* [2] ist ein Tool, das aus einer XML Dokumentkollektion eine DTD ableitet, so dass jedes Dokument der Kollektion für die abgeleitete DTD valide ist. Der *DTD-Miner* geht dabei dokumentweise vor und leitet für jedes XML Dokument einen Dokumentbaum ab, der die Struktur des Dokuments darstellt. Anschließend wird dieser Dokumentbaum in einem Spannbaum (*Spanning Graph*) mit den anderen Dokumentbäumen zusammengefügt. Dieser Spannbaum repräsentiert die Struktur der XML Dokumentkollektion.

Im ersten Schritt werden aus den Originaldaten die XML Tags in sogenannte *„intermediate files“* extrahiert, so dass für die Schema-Extraktion irrelevante Textpassagen und Kommentare

nicht mehr durchlaufen werden müssen. Die wesentlich kleineren *intermediate files* werden dann von der eigentlichen Extraktionskomponente, dem *„Document Tree Extraction Sub-Module“*, durchlaufen, das die Struktur in einen Dokumentgraphen extrahiert. Der Dokumentgraph wird dann dem *„Spanning Graph Construction Sub-Module“* übergeben, welches die einzelnen Dokumentgraphen zum Spannbaum, dem allgemeinen Schema, zusammenfügt. Zuletzt wird der erstellte Spannbaum durch das *„DTD Construction Sub-Module“* in eine DTD umgewandelt, wobei heuristische Methoden und ein vom Nutzer gewählter *„Maximum Repetition Factor“* zur Anwendung kommen.
Die Dokumente werden durch eine Dokument-ID, die auch dem entsprechenden Dokumentbaum zugewiesen wird, eindeutig gekennzeichnet. Dabei bekommt jeder Knoten im Dokumentbaum zusätzlich eine eindeutige Knoten-ID. Der Spannbaum ist ein geordneter, gerichteter und azyklischer Graph. Die Ordnung spiegelt dabei die Reihenfolge der Elemente, zum Beispiel die Reihenfolge von Kindelementen bei einer Verschachtelung, wider. Ein Knoten im Spannbaum repräsentiert alle Knoten der Dokumentbäume mit demselben *TagName*, indem alle zugehörigen Dokument-IDs in einer Liste gespeichert werden. Eine Kante stellt die Eltern-Kind-Relation, also die Verschachtelung von Elementen, dar. Dabei besitzt auch jede Kante eine Kanten-ID, die diese eindeutig identifiziert.
Die heuristischen Methoden, die während der DTD Konstruktion zum Einsatz kommen, sind zum einen die Definition von Pflicht- oder optionalen Elementen und zum anderen die Zusammenfassung von gleichartigen, benachbarten Elementen desselben Elternelements. Dabei wird diesen Elementgruppen eine Kardinalität von entweder *One-or-More* oder *Zero-or-More* zugewiesen.

XTRACT

XTRACT [4] ist ein Ansatz, der durch gezielte Generalisierung der abgeleiteten DTD versucht, überschaubarere und aussagekräftigere DTDs zu erzeugen. Dies soll mit Hilfe dreier Techniken, namentlich der *Generalization*, dem sogenannten *Factoring* und einem *„Minimum Description Length (MDL) Principle“*, erfolgen.
Bei der Generalisierung werden die Dokumente als Sequenzen benachbarter Elemente betrachtet. Beim Zutreffen bestimmter Bedingungen werden diese zusammengefasst und durch generalisierte Ausdrücke ersetzt. Der Vergleich geschieht auf Basis von regulären Ausdrücken und verwendet ebenfalls gewisse heuristische Methoden. So wird zum Beispiel durch den Einsatz des Kleene-Sterns ein möglicherweise sehr exakter aber zugleich umfangreicher regulärer Ausdruck stark verkürzt aber gleichzeitig auch generalisiert. Illustriert an der `cities`-Kollektion wäre das zum Beispiel der Fall, wenn die Kollektion die folgenden Dokumente enthält: `cc, ccc, cccc, ccccc`. (In Anlehnung an das Beispiel in [4] S. 168) Ein jedes `c` soll in diesem Fall für eine Stadt des `cities`-Arrays stehen, wobei die vier verschiedenen Fälle in unterschiedlichen Dokumenten der Kollektion auftreten. Der präziseste reguläre Ausdruck, der diese Fälle beschreibt, ist `c(c|c(c|c(c|cc)))`. Intuitiver, wesentlich kürzer dafür aber auch genereller ist des Ausdruck `c*`. Eben solche Formen der Generalisierung sollen automatisch geschlussfolgert werden. Dabei wird im Schritt der Generalisierung zuerst einfach eine Menge möglicher DTDs erzeugt.
Die DTDs der, durch die *Generalization*-Phase erzeugten, *„generalized candidate DTDs“* werden beim *Factoring* durch Techniken der logischen Optimierung durch prägnantere Ausdrücke ersetzt. Es werden stets mindestens zwei Ausdrücke der *Generalization* verknüpft

und durch Umformung und Zusammenfassung zu einem insgesamt kleineren Ausdruck zusammengefasst.
Im letzten Schritt wird das MDL-Prinzip angewendet, um aus den einzelnen *„candidate DTDs“* eine möglichst optimale DTD abzuleiten. Dabei werden die DTDs nach zwei Kennzahlen bewertet. Zum einen sollen die regulären Ausdrücke präzise sein, also nicht zu viele unzutreffende Sequenzen erlauben, zum anderen aber auch prägnant, d.h. im Umfang möglichst gering sein. Die Bewertung basiert auf einem Kostenprinzip mit zwei Summanden. Der erste Summand ergibt sich aus der Länge des regulären Ausdrucks. Der zweite Summand ist selbst wieder eine Summe, die aus der Länge der darzustellenden Sequenzen mit Hilfe des jeweiligen regulären Ausdrucks berechnet wird. Abschließend wird die günstigste DTD als beste DTD ausgewählt.

Schema-Extraktion auf Basis von ECFG

In [5] und [14] wird ein Schema-Extraktionsverfahren vorgeschlagen, das mit Hilfe von *„range extended context-free grammars“ (ECFG)* aus XML Dokumenten ein XML Schema ableitet. Die *ECFG* werden als 5-Tupel $G = (T, N, D, \delta, Start)$ definiert, wobei T eine Menge von Terminalen, N eine Menge von Nichtterminalen und D eine Menge von Datentypen ist. δ ist ein Satz von Produktionsregeln der Form $A \rightarrow \alpha$ mit $A \in N$ und α einer Kombination aus Terminalen, Nichtterminalen oder Datentypen. $Start$ markiert den Ausgangszustand, von dem aus durch Ersetzen der Nichtterminale durch Anwendung der Produktionsregeln gültige Dokumente erzeugt werden können. Durch Sammlung dieser Produktionsregeln beim Durchlaufen der Dokumentkollektion entsteht ein Satz von Regeln, Terminalen, Nichtterminalen und Datentypen, der eine Rekonstruktion eines jeden Dokuments der Kollektion erlaubt. Mit Hilfe geeigneter Äquivalenz- und Ähnlichkeitsmaße wird die *ECFG* vereinfacht, indem beispielsweise Nichtterminale zusammengefasst oder durch reguläre Ausdrücke generalisiert werden. Aus der erstellten *ECFG* wird anschließend das XML Schema abgeleitet.

XSTRUCT

XSTRUCT [3] ist ein Schema-Extraktionsverfahren für XML, das auf XTRACT basiert. Neben der Behebung des Problems der 1-Unzweideutigkeit wird der Fokus auf XML spezifische Gegebenheiten und die Performance des Verfahrens gerichtet. Die Autoren argumentieren, dass durch die Standardisierung von XML die Extraktion von Datentypen und Attributen an Bedeutung gewonnen hat und integrieren diese in ihr Verfahren. Neben dem aus XTRACT bekannten *„Factoring Module“* verwenden sie daher ein *„Datatype Recognition Module“* und ein *„Attribute Extraction Module“*. Die Dokumente werden mit Hilfe eines *„SAX Parsers“* eingelesen und im *„Structure Model“* gespeichert, die maßgeblich für die bessere Performance verantwortlich sein sollen.
Die Adaption eines *„Element Content Models“* $E := (T_1 \ldots T_k)^{<min,max>}$ erlaubt es XSTRUCT, die Struktur von XML Dokumenten über eine Eltern-Kind-Relation zu beschreiben. $T_1 \ldots T_k$ sind Terme, die eine Reihe von XML-Elementen s entweder als Sequenz $T_n := (s_{n1}^{opt} \ldots s_{nj}^{opt})^{<min,max>}$ oder Auswahl $T_n := (s_{n1}^{opt} \mid \ldots \mid s_{nj}^{opt})^{<min,max>}$ darstellen. Der opt-Flag ermöglicht die Definition optionaler Elemente und somit die Integration heterogener Daten in einem Schema. Dies geschieht, indem die einzelnen *Element Content Models* der Dokumente im *Factoring*-Schritt zusammengefügt werden. Dabei werden gleichartige Terme mit dem kleinsten Minima und größten Maxima zusammengefasst.

Das *Datatype Recognition Module* extrahiert für jedes Element den Datentyp. Unter Beachtung der bisher gefundenen Datentypen für ein Element wird nach der Extraktion ggf. ein neuer, allgemeinerer Datentyp durch Generalisierung gebildet. Aus Komplexitätsgründen wurde die Auswahl aller möglichen primitiven XML-Datentypen stark eingeschränkt.
Das *Attribute Extraction Module* zeichnet die Attribute eines jeden Elements auf und trifft am Ende der Extraktion eine Annahme über die Optionalität des Attributs. Kommt es in jedem Dokument vor, so wird es als Pflichtattribut deklariert. Fehlt es in mindestens einem Dokument, wird es als optional markiert. Zusätzlich werden Attribute erkannt, die festgelegte Werte besitzen.
Das *Structure Model* ist zweigeteilt. Zum einen besteht es aus einer Hash-Tabelle, die die einzelnen XML-Elemente indiziert, und zum anderen aus einer Liste, die jedes Element an der indizierten Position speichert und für jedes Element alle benötigten Informationen – darunter auch alle *Element Content Models* – bereitstellt.

3 Konzeption

In diesem Kapitel sollen die zu entwickelnden Verfahren konzipiert werden. Um ein strukturiertes Vorgehen zu garantieren, müssen die einzelnen Verfahren zunächst spezifiziert und die jeweils benötigten Komponenten identifiziert werden. Anschließend kann durch Implementation und Zusammensetzen der Komponenten ein Algorithmus entwickelt werden, der die Ausführung der verschiedenen Verfahren unterstützt.

Grundlage für die Implementation inkrementeller Schema-Extraktionsverfahren ist das Vorhandensein eines initialen Schemas. Dieses wird durch Schema-Extraktion aus dem aktuellen Datenbestand gewonnen. Aufbauend auf dem dafür benötigten Extraktionsverfahren können verschiedene, inkrementelle Verfahren und andere Funktionen implementiert werden. Neben der inkrementellen Aktualisierung des Ausgangsschemas mit Hilfe von Update-Logs und geänderten Dokumenten, sollen Metriken ausgegeben werden. Zu diesen Metriken zählen Ausreißerdokumente, potenziell gleiche Knoten sowie relative und absolute Häufigkeiten der Elemente in der Kollektion. Eine Ausgabe des Schemas in übersichtlicher Form soll ebenfalls erfolgen.

Zur Ableitung der Metriken und Umsetzung aller Funktionen bedarf es verschiedener Voraussetzungen, die im Folgenden näher erläutert werden sollen.

Das Ableiten von Ausreißerdokumenten, das Erkennen von potenziell gleichen Knoten und das Zählen der absoluten und relativen Häufigkeiten sind mit sehr wenigen Informationen möglich. Für eine korrekte Aktualisierung des Schemas sind jedoch exaktere Informationen notwendig. Um für jedes Verfahren eine optimale Geschwindigkeit zu gewährleisten, sollen zwei verschiedene Varianten für die Schema-Extraktion implementiert werden.

Eine schnelle Variante soll nur die Informationen extrahieren, die zur Ableitung der Metriken notwendig sind. Eine zweite, komplexere Variante dient der Aktualisierung. Im Wesentlichen unterscheiden sich die beiden Varianten durch den Umgang mit den Dokument-IDs. Diese sind für die Metriken nicht von Bedeutung und können im einfachen Verfahren durch einen Zähler ersetzt werden, der lediglich der Anzahl der Dokumente und Elemente entspricht. Soll das abgeleitete Schema dagegen korrekt aktualisiert werden, so muss nicht nur jeder Knoten eindeutig identifizierbar sein, sondern auch jedes Dokument. Aufgrund der Größe der Dokumentkollektionen verursacht die Speicherung der Dokument-IDs einen erheblichen Aufwand, der Anlass zur Zweiteilung der Extraktion ist.

Um die Übersicht zu wahren, wurden Konzeption und Algorithmus in folgende Teilkomponenten gegliedert:

- Extraktionskomponente
- Aktualisierungskomponente
- Visualisierungskomponente
- Speicherkomponente
- GUI
- Ableitung der Metriken

Neben dem Algorithmus selbst muss auch die Ausgabe betrachtet werden. Diese stellt durch die benötigten, zu extrahierenden Informationen wesentliche Anforderungen an die Extraktionskomponenten des Algorithmus und wird demzufolge zuerst betrachtet. Anschließend wird das Konzept des Algorithmus zur Schema-Extraktion sowie zur inkrementellen Aktualisierung des Schemas vorgestellt. Es werden weitere Funktionen

eingeführt, die implizit oder explizit benötigt werden oder die Bedienung erleichtern. Der Algorithmus wird dabei bereits in Hinblick auf die Programmiersprache Java konzipiert. Im letzten Abschnitt wird explizit auf die Ableitung der Metriken, die auf den Funktionen des Algorithmus basieren, eingegangen.

3.1 Aufbau des Schemas

Kernbestandteil der Schema-Extraktion ist die Erstellung eines Schemas. Dieses soll die zugrundeliegenden Daten repräsentieren. Daten können jedoch auf sehr unterschiedliche Weise beschrieben werden. Eine Auswahl der darzustellenden Informationen ist folglich unumgänglich und muss im Vorfeld der Implementation erfolgen. In diesem Abschnitt sollen eben dieser Aufbau und die Zusammensetzung des verwendeten Schemas beschrieben werden. Insbesondere ist dabei zu klären, welche Informationen mit welcher Genauigkeit im Schema dargestellt werden sollen. Neben den in der Ausgabe erwünschten Informationen müssen zudem auch alle Informationen extrahiert werden, die zur korrekten Aktualisierung des Schemas benötigt werden.

Bei der Ableitung der Schemainformationen ist zu beachten, dass es sich bei dem Verfahren um Schema-Reengineering handelt. So können einige Schlüsse nicht mit absoluter Sicherheit gezogen werden, da es sich bei dem abgeleiteten Schema immer nur um eine Momentaufnahme des Datenbestandes handelt. Insbesondere bei inkrementellen Verfahren ist von regelmäßigen Änderungen des Datenbestandes auszugehen. Da das Schema nach der Ableitung auch als Muster zur Erzeugung neuer Daten verwendet werden können soll, muss ein sinnvolles Maß zwischen Spezialisierung und Generalisierung gefunden werden.

Folgende Informationen sollen bei der Ausgabe des Schemas angegeben werden:

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

Für die Aufnahme dieser Informationen bietet die Schemasprache vordefinierte Schlüsselwörter. So ergibt sich folgende Abbildung der Informationen auf die Schlüsselwörter:

JSON Schema Schlüsselwort	Gespeicherte Information
`name`	Name des Elements
`type`	Datentyp(en) des Elements
`description`	Beschreibungsfeld, das zur Darstellung der absoluten und relativen Häufigkeit des Auftretens des Elements verwendet werden soll. Bei Arrays wird das Feld außerdem zur Darstellung der Array-Reihenfolge verwendet.
`properties`	Kindelemente eines JSON Objects
`items`	Kindelemente eines JSON Arrays
`required`	Pflichtelemente
`anyOf`	Falls ein Array verschiedene Kindelemente besitzt, werden diese mit Hilfe von `anyOf`

	dargestellt. Die Reihenfolge wird im `description`-Feld beschrieben.

Tabelle 1: Bestandteile des Ausgabeschemas

Die in Tabelle 1 aufgelisteten Informationen reichen jedoch nicht aus, um eine korrekte Aktualisierung des Schemas zu gewährleisten. Zu diesem Zweck müssen weitere Informationen, wie ein eindeutiger Schlüssel für jedes Element und eine eindeutige Identifikationsnummer für jedes Dokument erhoben und gespeichert werden. Diese werden benötigt, um bei der Schema-Aktualisierung korrekte Einfüge-, Aktualisierungs- und Löschvorgänge sicherzustellen. Eine Ausgabe dieser Informationen ist nicht sinnvoll, da sie für den Betrachter wenig Aussagekraft haben und das Schema nur zusätzlich vergrößern würden. Es kommt folglich zur Bildung zweier Schemata: einem folgend als *internes Schema* bezeichneten, vollständigen Schema und einem als *externes Schema* bezeichneten, vereinfachten Schema.

Der Algorithmus soll auf einer als internes Schema bezeichneten Knotenmenge arbeiten und diese nur zur Ausgabe in ein vereinfachtes JSON Schema umwandeln.

3.1.1 Internes Schema

Alle Komponenten des Algorithmus arbeiten mit dem internen Schema. Aus Performance- und Handhabungsgründen wird es nicht im JSON Schema-Format gespeichert, sondern in Form von Java-Objekten. Diese Java-Objekte werden aus einer eigens implementierten Klasse für diese Elemente erzeugt. Die verschachtelte Struktur von JSON Daten lässt sich durch Baumstrukturen repräsentieren. Diese Baumstrukturen bestehen aus Knoten und Kanten. Folglich wurde eine ähnliche Repräsentation für das interne Schema gewählt. Da jedoch für jeden Knoten und jede Kante alle Dokument-IDs der Dokumente mit diesem Element gespeichert werden müssen, wurde zur Reduktion des Speicherverbrauchs auf die Speicherung der Kantenobjekte verzichtet und alle notwendigen Informationen im Knotenobjekt gespeichert. Das interne Schema besteht also aus einer Menge von Knotenobjekten mit den folgenden Attributen:

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

Mit dieser Knotenmenge lassen sich verschachtelte Strukturen darstellen. Name und Pfad des Elements identifizieren dieses eindeutig. Der Datentyp gibt Aufschluss über etwaige Kindelemente, welche mit Hilfe des Pfads und Namens des Elternelements gefunden werden können. Die absolute und relative Häufigkeit lassen sich aus der Anzahl der Dokument-IDs berechnen. Für Arrays muss zudem die Reihenfolge der Elemente gespeichert werden. Dies geschieht gruppiert nach ihrem Datentyp mit jeweiligen minimalen und maximalen

Häufigkeiten der Elementgruppe. Eine eindeutige und korrekte Rekonstruktion eines validen JSON Schemas aus der internen Knotenmenge ist also gegeben.

3.1.2 Externes Schema

Der Algorithmus soll ein derartiges JSON Schema ausgeben, so dass jedes Dokument der eingelesenen Kollektion in Bezug auf dieses Schema wohlgeformt und valide ist. Um eine erfolgreiche Validierung zu garantieren, müssen bei der Erzeugung des externen Schemas die Regeln der JSON Schema Spezifikation befolgt werden. [18] Diese nennt beispielsweise spezielle Anforderungen an das Wurzelelement eines jeden Schemas, welches laut Spezifikation ein `title`-Element, `description`-Element und `$schema`-Element zu beinhalten hat. Im Titel steht der Name des Schemas, welcher vom Algorithmus aus dem Namen der Kollektion bzw. Kollektionen abgeleitet werden soll. Die verwendeten Dokumentkollektionen sowie das Erstellungsdatum des Schemas sind im `description`-Feld zu nennen. Ein Verweis auf die verwendete Schema-Version wird im `$schema`-Element angegeben.
Anschließend folgt das `properties`-Feld und die eigentliche Darstellung der Elemente des Dokuments beginnt. JSON Primitives und JSON Arrays in der obersten Ebene des Dokuments sind aufgrund der Anforderung des Vorhandenseins einer Dokument-ID ausgeschlossen, da auf dieser Ebene das eindeutige Schlüssel-Wert-Paar `_id` vorhanden sein muss, was nur in einem JSON Object möglich ist. Das `_id`-Feld wird durch `MongoDB` automatisch hinzugefügt, falls beim Einfügen des Dokuments keine ID vorhanden ist. `MongoDB` akzeptiert nur Dokumente, die das Hinzufügen der ID ermöglichen.
Im Allgemeinen wird ein Element durch Angabe des Namens, Datentyps, Häufigkeit sowie ggf. der Kindelemente spezifiziert. Das `type`-Feld gibt dabei den JSON Typ des aktuellen Knotens im Schema an. Das `properties`- bzw. `items`-Feld dient der verschachtelten Darstellung der Kindelemente des aktuellen Elements, von denen bei Objekten all diejenigen Elemente, die in allen Dokumenten auftreten, im `required`-Array aufgelistet werden. Das `description`-Feld wird zur Darstellung der relativen und absoluten Häufigkeit jedes Knotens verwendet. Im Folgenden wird die Darstellung der verschiedenen JSON Elemente einzeln erläutert.

Darstellung von JSON Primitives

JSON Primitives kommen in den Blattknoten der Schemastruktur vor. Es sind die Elemente im Dokument, die die tatsächlichen Daten, wie zum Beispiel Zeichenketten oder Zahlen, beinhalten. Bei der Extraktion dieser Elemente können ihr Datentyp sowie der Name des Elements ermittelt werden. Weiterhin lässt sich durch das Speichern und Auswerten der Dokument-IDs die Häufigkeit des Auftretens berechnen. *JSON Primitives* können, wie in Abbildung 8 skizziert, dargestellt werden.

```
"Name des Elements" : {
   "type" : "JSON Primitive Datentyp",
   „description” : “relative und absolute Häufigkeit”
}
```

Abbildung 8: Darstellung von JSON Primitives

Darstellung von JSON Objects

JSON Objects sind einer der zwei komplexen Datentypen in JSON. Sie zeichnen sich durch die beliebige Reihenfolge der Kindelemente aus. Neben dem Namen des *JSON Objects* und dem Datentyp wird also ein Feld für die Kindelemente benötigt. Pflichtelemente unter den

Kindelementen werden in einem `required`-Array angegeben. Abbildung 9 zeigt die Struktur eines *JSON Objects* im Schema.

```
"Name des JSON Objects": {
   "type" : "JSON Object",
   "description" : "relative und absolute Häufigkeit",
   "properties" : {
      Aufzählung der Kindelemente in Form von
      „Name des Kindelements“ : { JSON Dokument }
   },
   "required" : [
      "Namen der Pflichtelemente"
   ]
}
```

Abbildung 9: Darstellung von JSON Objects

Darstellung von JSON Arrays

JSON Arrays erfordern eine spezielle Betrachtung, da sie – neben den bereits betrachteten Objekten – die zweite und letzte Containerklasse darstellen, bei der im Gegensatz zu den Objekten die Reihenfolge der Elemente eine Rolle spielt. Eine weitere Besonderheit ist, dass Elemente in einem Array keinen Namen besitzen. Bei einfachen Elementen, also den JSON Primitives, spielt das keine weitere Rolle, da sie sich durch ihren Datentyp ausreichend unterscheiden. JSON Objects hingegen charakterisieren sich zusätzlich durch ihre Kindelemente und nicht nur ihren Namen und Datentyp. Bei herkömmlichen JSON Objects werden der Name im Schlüssel und die Attribute des Objects im Wert gespeichert. Dieses typische `name`-Feld existiert bei Objects im Array jedoch nicht. Eine automatische, eindeutige Unterscheidung von verschiedenen JSON Objects in einem Array ist also nicht möglich. Jedes Object kann als ein neues Object angesehen werden, was jedoch zu einem sehr spezialisierten Schema führt. Insbesondere gibt es an dieser Stelle Probleme, wenn die Dokumente heterogene Arrays besitzen. Daher wird ein anderer Ansatz verfolgt, bei dem alle JSON Objects eines *JSON Arrays* zu einem so genannten „`ArrayObject`“ zusammengefasst werden. Durch diese Zusammenfassung entstehen bei heterogenen Objects eines Arrays optionale Elemente im `ArrayObject`. Dies stellt eine Generalisierung dar, erlaubt aber die Rekonstruktion aller extrahierten JSON Objects eines Arrays.

Der Schlüssel für alle Kindelemente eines Arrays wird durch JSON Schema mit `items` deklariert. Dieser weist jedoch per Definition einige Besonderheiten auf:

Falls im `items`-Feld ein JSON Schema angegeben ist, muss jedes Element des Arrays gegen dieses Schema valide sein. Falls im `items`-Feld dagegen ein Array steht, muss jedes Element gegen seine Position im `items`-Array validiert werden. Ein `additionalItems`-Schlüssel erlaubt die Behandlung von Elementen außerhalb des `items`-Array-Index. [22]

Die Variante, jedes Arrayelement einzeln im Schema aufzulisten, widerspricht dem Ziel des prägnanten Schemas und wird deshalb nicht angewendet. Im `items`-Feld soll also ein Schema angegeben werden, gegen das jedes Arrayelement validiert werden muss. Bei der Gruppierung der Arrayelemente nach ihrem Datentyp geschieht das bei Elementen mit nur einem primitiven Datentypen automatisch. Objekte werden ebenfalls zu `ArrayObjects` generalisiert (siehe dazu 3.1.3 Objekte in Arrays) Kommen jedoch unterschiedliche Datentypen in einem Array vor, bleibt zunächst nur die Generalisierung des Schemas, so dass alle Elemente für dieses Schema valide sind. Insbesondere beim Auftreten von leeren Arrays müsste im `items`-Feld ein leeres Schema angegeben werden, damit jedes Dokument validiert werden kann.

Durch die Verwendung eines weiteren Schlüsselwortes soll dieser Generalisierung entgegengewirkt werden. JSON Schema bietet drei verschiedene Schlüsselwörter zum Validieren von Elementen gegen Schema-Gruppen: `anyOf`, `oneOf` und `allOf`. `AllOf` erzwingt die erfolgreiche Validierung gegen jedes und `oneOf` gegen genau ein Schema-Element der Schema-Gruppe. Dadurch, dass leere Arrays ausschließlich gegen ein leeres Schema erfolgreich validiert werden können, dieses aber auch jedes beliebige andere Element validiert, kann nur das `anyOf`-Element verwendet werden, bei dem ein Element gegen mindestens ein Schema im Array erfolgreich validiert werden muss.
Eine flexible Angabe der Reihenfolge von Arrayelementen ist in JSON Schema nicht vorgesehen. Die Reihenfolge kann entweder beliebig sein oder über den Index im `items`-Array spezifiziert werden. Die Möglichkeit die Reihenfolge der Arrayelemente in Abhängigkeit von der Häufigkeit des Vorgänger-Elements zu deklarieren, gibt es nicht. Die Ausgabe der Reihenfolge geschieht daher im `description`-Feld in einem selbst definierten „`array order`“-Element. In diesem Element werden die Elemente des Arrays nach Datentypen gruppiert, ihrer Reihenfolge nach aufgelistet und mit ihren jeweiligen Kardinalitäten angegeben. Abbildung 10 zeigt die Darstellung von Arrays mit Elementen eines Datentyps und Elementen verschiedener Datentypen.

```
// falls alle Kindelemente vom selben Datentyp
“Name des JSON Arrays” : {
   “type” : “JSON Array”,
   "description" : {
      „occurences“ : { "relative und absolute Häufigkeit" },
      „array order“ : { „Position, Datentyp, Minima, Maxima” }
   },
   "items": {
      JSON Dokument des Kindelements
   }
},
// falls die Kindelemente mehr als einen Datentyp besitzen
// oder auch leere Arrays vorkommen
“Name des JSON Arrays” : {
   “type” : “JSON Array”,
   "description" : {
      „occurences“ : { "relative und absolute Häufigkeit" },
      „array order“ : {
         „Position, Datentyp, Minima, Maxima”,
         „Position, Datentyp, Minima, Maxima”
      }
   },
   "items": {
      „anyOf“ : [
         { JSON Dokument des Kindelements },
         { JSON Dokument des Kindelements }
      ]
   }
}
```

Abbildung 10: Darstellung von JSON Arrays

3.1.3 Spezialisierung vs. Generalisierung

Bei der Schema-Definition werden die Daten oftmals sehr exakt durch das erstellte Schema dargestellt. Eine große Anzahl verwendeter Beschränkungen soll die Daten möglichst genau beschreiben, damit die Validatoren nur die gewünschten Daten erfolgreich validieren und erzeugte Daten allen Anforderungen entsprechen. Bei der Schema-Extraktion ist dies nicht ohne weiteres möglich, da das erwünschte Ausgangsschema nicht bekannt ist. Es können nur die Beschränkungen gesetzt werden, die sich aus den Daten erschließen lassen. Und auch dort ist

nicht jede Schlussfolgerung sinnvoll, da von Änderungen am Datenbestand auszugehen ist und diese bei Verwendung von schemalosen Datenbanken – insbesondere im Kontext der Webentwicklung – oft neue Strukturen einführen. Auf der anderen Seite darf nicht zu stark verallgemeinert werden, da dann der Sinn des Schemas zunichtegemacht wird. Es gilt ein gutes Maß zwischen Spezialisierung und Generalisierung bei der Extraktion zu finden. In anderen Schema-Extraktionsansätzen wird dieses Problem oft als Widerspruch zwischen exakter Repräsentation der Daten und einem kompakten und prägnanten Schema beschrieben. [3, 4, 14] Wesentliche Generalisierungen und Spezialisierungen im erstellten JSON Schema sollen in diesem Abschnitt vorgestellt und begründet werden.

Pflichtelemente

Ohne die Definition von Pflichtelementen ist die Aussagekraft eines Schemas stark reduziert, da sogar ein leeres Dokument erfolgreich validiert werden würde. Die Definition dieser Pflichtelemente lässt sich allerdings nicht mit Sicherheit aus der Momentaufnahme des Datenbestandes ableiten, da auch ein optionales Element rein zufallsbedingt in jedem Dokument vorkommen kann. Es handelt sich dabei also immer um eine Annahme.

Im Schema werden all die Elemente als Pflichtelemente deklariert, die in jeder Instanz ihres Elternelements mindestens einmal auftreten.

Arrayelemente

Die besondere Eigenschaft von Arrays ist die Reihenfolge der Elemente. Diese kann für ein Dokument problemlos ermittelt werden. Besitzen verschiedene Dokumente nun jedoch eine unterschiedliche Anordnung der Arrayelemente, so ist eine Angabe einer allgemeingültigen Reihenfolge schwieriger. Eine Auflistung aller möglichen Anordnungen wäre eine Lösung. Eine weitere Option ist die Abstraktion über die Datentypen. Dabei werden die Anzahl aneinandergereihter Elemente mit demselben Datentyp zu einem Element im Schema zusammengefasst und mit einem Minimum und Maximum ihres Auftretens in allen Dokumenten versehen. Bei einer in Arrays typischen Indizierung entstehen dadurch jedoch Probleme, da das Minimum und Maximum des ersten Elements die Positionen der folgenden Elemente beeinflussen. Ein Element, das an zweiter Stelle angegeben wird, ist also nicht zwingend auch an der zweiten Indexposition zu finden. Eine dritte Option ist die reine Auflistung aller möglichen Elemente im Array. Dies ist jedoch auch die generellste Lösung.

Arrayelemente sollen im Schema über ihren Datentyp abstrahiert und unter Angabe ihres minimalen und maximalen Auftretens aufgezählt werden.

Objekte in Arrays

Wie bereits beschrieben, können Arrayelemente nach ihrem Datentyp unterschieden werden. Jedoch können verschiedene Objekte in einem Array aufgelistet sein. Objekte identifizieren sich eindeutig über Name und Pfad im Dokument. Objekte in Arrays haben jedoch keinen eigenen Objektnamen. Dies führt dazu, dass diese Objekte nicht mit Sicherheit unterschieden

werden können. Eine Möglichkeit ist, jedes Objekt als einzigartig anzusehen. Eine weitere Option ist es, Objekte nach ihrer Struktur zu unterscheiden. Eine Deklaration von optionalen Elementen ist dabei jedoch nicht möglich, da diese durch ihre unterschiedliche Struktur automatisch ein neues Objekt generieren. Die Anzahl der unterschiedlichen Objekte beeinflusst dabei stark die Größe des Schemas. Eine dritte Option ist es, alle Objekte als ein Objekt anzusehen. Dabei ist die Deklaration von Pflicht- und optionalen Elementen möglich. Es kommt allerdings zur Generalisierung, falls es sich tatsächlich um mehrere Objekte handelt, da dann disjunkte Pflichtelemente der einzelnen Objekte als optional deklariert werden.

Alle Objekte eines Arrays sollen zu einem „ArrayObject“ zusammengefasst werden.

Leere Arrays und anyOf

Wie bereits im Abschnitt Darstellung von JSON Arrays beschrieben, müssen Arrayelemente verschiedener Datentypen mit anyOf gruppiert werden. Im anyOf-Array können verschiedene Schemata angegeben werden, wobei jedes Element des Arrays anschließend mindestens einem dieser Schemata für eine erfolgreiche Validierung entsprechen muss. Sobald ein Dokument ein leeres Array besitzt, muss ein leeres Schema in das anyOf-Array eingefügt werden, so dass auch dieses leere Array validiert werden kann. Das leere Schema führt dann jedoch dazu, dass jedes beliebige Element auch erfolgreich validiert wird. Es kommt zur Generalisierung des kompletten Arrays, da die Angabe der Reihenfolge und Datentypen möglicher Elemente im `description`-Feld von Validatoren nicht interpretiert werden kann.

Verschiedene Datentypen im selben Array werden durch Angabe mehrerer Schemata im `anyOf`-Array dargestellt.

Leere Arrays führen zu einem leeren Schema im `anyOf`-Array, mit dem jedes Arrayelement erfolgreich validiert werden kann.

3.2 Aufbau des Algorithmus

Ziel dieser Arbeit ist es, verschiedene inkrementelle Verfahren zur Ableitung von Schemainformationen zu entwickeln. Diese Verfahren sollen mit Hilfe von Computern ausgeführt werden können. Es bedarf also eines Algorithmus, d.h. einer endlichen, in einer festgelegten Sprache abgefassten Beschreibung der Verfahren. [23] Als Sprache soll die Programmiersprache Java dienen. Neben den, für die Konzeption relevanten, Anforderungen an die Funktionsweise der Verfahren, die sich aus dem Ziel der Arbeit ergeben, sollen, da es sich bei diesen Verfahren um Anwendungsprogramme [24] handelt, auch allgemeine Kriterien zur Sicherstellung der Produktqualität von Software berücksichtigt werden.

Im vorangegangenen Unterkapitel wurde die Struktur des internen und externen Schemas definiert. Nun sollen die verschiedenen Verfahren festgelegt und anschließend näher spezifiziert werden. Verfahren lassen sich beispielsweise durch ihren Input, ihren Output und

ihre innere Funktionsweise unterscheiden. Folgend sollen diese drei Faktoren in Bezug auf die Schema-Extraktion erläutert und daraus Kombinationen gebildet werden, die anschließend untergliedert in Komponenten zu einem Algorithmus kombiniert werden sollen.

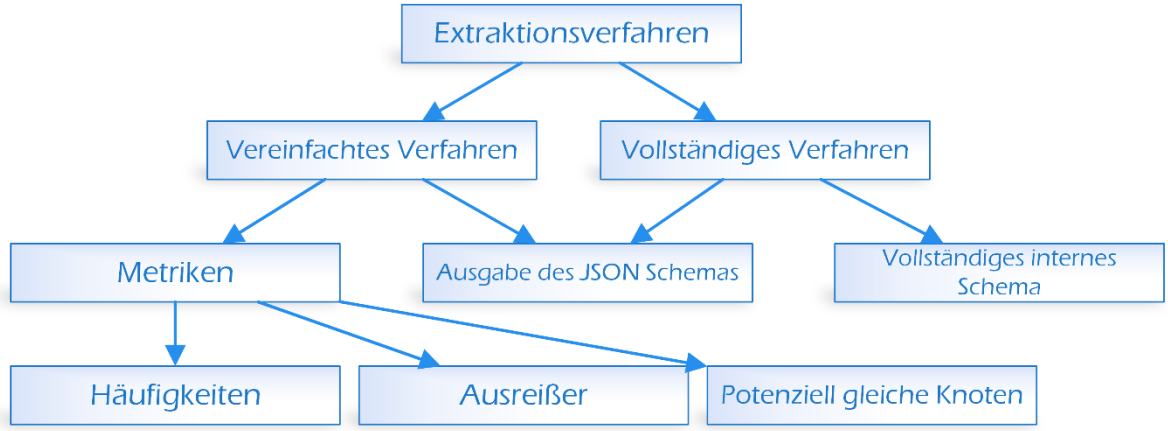

Abbildung 11: Unterteilung der Extraktionsverfahren nach ihrem Output

Der Input für die Schema-Extraktion ist eine zugrundeliegende Dokumentkollektion. Es wurde bereits festgelegt, dass diese JSON Dokumente zu beinhalten hat. Für die inkrementelle Extraktion werden ein Ausgangsschema sowie umzusetzende Änderungen an diesem Schema benötigt. Das Ausgangsschema soll entweder direkt extrahiert oder aber durch eine Importfunktion importiert werden. Die Änderungen sollen zum einen in Form von geänderten Dokumenten und zum anderen durch ein Update-Log zur Verfügung gestellt werden können.

Ein Schema-Extraktionsverfahren soll ein Ausgangsschema bereitstellen.

Aktualisierungen sollen durch ein Update-Log oder in Form von geänderten Dokumenten eingelesen werden.

Zum Output der Verfahren gehört die Ausgabe des Schemas als wohlgeformtes und übersichtliches JSON Dokument. Weiterhin können aus einer Dokumentkollektion Metriken abgeleitet werden. Zu diesen Metriken zählen Häufigkeiten der Elemente oder Ausreißerdokumente. Aber auch das extrahierte interne Schema und die Aktualisierung dieses sind als Output der Verfahren zu betrachten. (siehe Abbildung 11)

Neben der Ausgabe eines wohlgeformten JSON Schemas sollen Metriken über die Dokumentkollektion erfasst werden.

Die interne Funktionsweise wird maßgeblich durch Input und Output bestimmt. Die Bestimmung der bestmöglichen Funktionsweise für jedes Verfahren ist wesentlicher Bestandteil der Implementation. Ziel ist es, für jedes Verfahren eine korrekte und möglichst schnelle Umsetzung zu finden. Zur besseren Handhabung des Aktualisierungsprozesses wird eine Funktion zum Speichern und Laden des internen Schemas benötigt.

Die Verfahren sollen korrekt und mit möglichst geringem Zeitbedarf arbeiten. Insbesondere die Aktualisierung soll inkrementell geschehen.

Der Algorithmus setzt sich entsprechend aus den folgend als *Extraktions-*, *Aktualisierungs-*, *Visualisierungs-* und *Speicherkomponente* bezeichneten Bestandteilen zusammen. (siehe Abbildung 12) Hinzu kommt eine *graphische Benutzeroberfläche*, die zur verbesserten Benutzbarkeit der Software beitragen soll. Abschließend werden in diesem Abschnitt Kriterien festgelegt, die eine hohe allgemeine Softwarequalität des Algorithmus sicherstellen sollen.

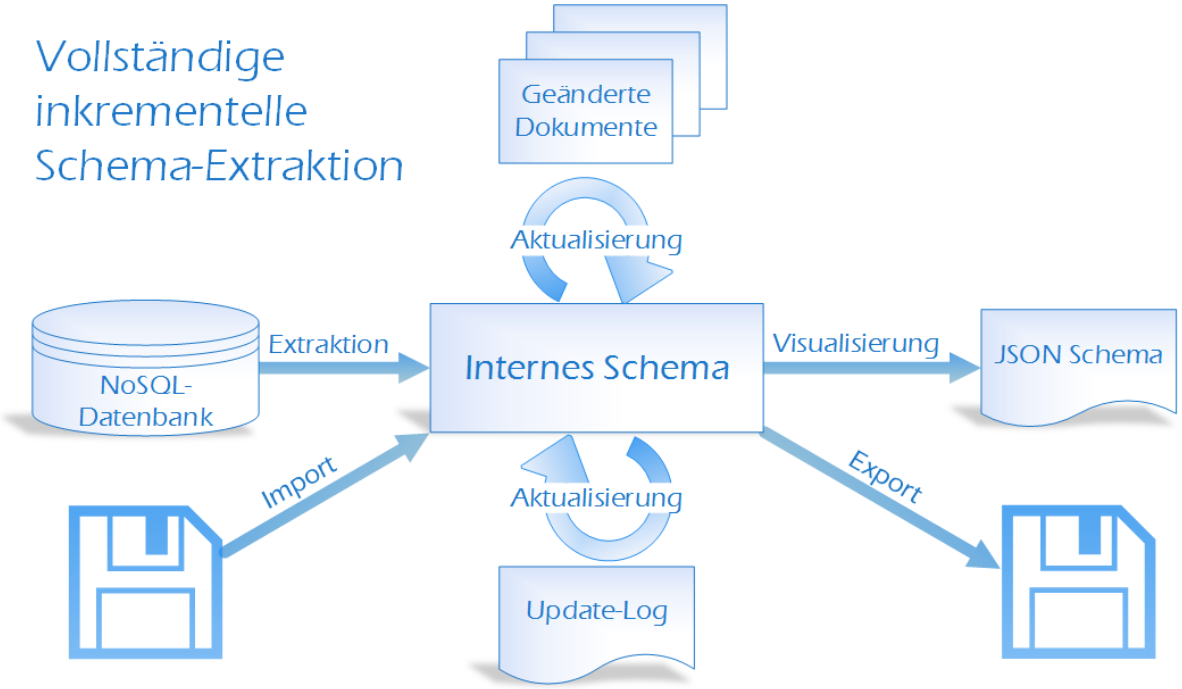

Abbildung 12: Systemarchitektur des vollständigen, inkrementellen Extraktionsalgorithmus

3.2.1 Extraktionskomponente

Bei der Schema-Extraktion soll ein initiales Schema aus den bereits existierenden Daten abgeleitet werden – also der Prozess des Schema-Reengineering auf einem bestimmten Datenbestand durchgeführt werden. Für ein einzelnes Dokument muss dazu lediglich jedes Element durchlaufen und die in Abschnitt 3.1 festgelegten Informationen über jedes Element extrahiert werden. Dieses Vorgehen lässt sich auf eine Dokumentkollektion erweitern, indem diese dokumentweise durchlaufen wird. Bei der Übernahme der Elemente in das Schema muss dann jedoch darauf geachtet werden, ob diese Elemente bereits vorhanden sind. In diesem Fall werden die vorhandenen Elemente aktualisiert und um die Informationen aus dem aktuellen Dokument ergänzt. Unterscheiden sich die Dokumente in ihrem Aufbau, so muss dies im Schema ersichtlich werden. Elemente, die nicht in allen Dokumenten auftreten, müssen als optional deklariert werden. Auch die Datentypen namentlich gleicher Elemente müssen nicht übereinstimmen und daher überprüft und ggf. im Schema angepasst werden. Eine Möglichkeit ist das Generalisieren der Datentypen, so dass ein allgemeinerer Datentyp gewählt wird, der für alle Dokumente zutrifft. Eine zweite Option ist das Auflisten aller vorkommenden Datentypen in einer Datentypliste. Aufgrund der bereits sehr allgemein gehaltenen Datentypen in JSON bietet sich für die Schema-Extraktion aus JSON Daten die Auflistung der Datentypen an.

Im Kapitel 2.6.2 wurden ausgewählte Extraktionsverfahren in der Literatur vorgestellt. Im Rahmen dieser Arbeit war bereits ein erster Entwurf eines Schema-Extraktionsalgorithmus für JSON in der Programmiersprache Python gegeben, welcher sich an dem Prinzip des *DTD-Miners* orientiert. Dieser Algorithmus diente als Grundlage für den Entwurf der Extraktionskomponente. Eine Dokumentkollektion wird dokumentweise eingelesen und für jedes Dokument die eindeutige Dokument-ID extrahiert. Diese Dokument-ID muss für jedes Dokument vorhanden sein, da sie der eindeutigen Zuordnung zum eingelesenen Dokument dient, welche für den Aktualisierungsprozess relevant ist. Beim Einlesen von Dokumenten aus `MongoDB` ist diese Nummer garantiert vorhanden, da sie von `MongoDB` erzeugt wird. Sollten Dokumente aus einer anderen Quelle eingelesen werden, so muss das Vorhandensein dieser Nummer überprüft und sie gegebenenfalls generiert werden.
Das Schema wird jedoch nicht dokumentweise sondern elementweise gespeichert. Um die eindeutige Zuordnung jedes Elements zum Ausgangsdokument zu gewährleisten, müssen die entsprechenden Dokument-IDs in jedem Schema-Element gespeichert werden. Für die Unterscheidung der Elemente werden Pfad und Elementnamen verwendet, da die Namen per JSON Definition auf ihrer Ebene eindeutig sein müssen und somit die Kombination aus Pfad und Name stets zu genau einem Element führt.
Die Extraktionskomponente durchläuft also die Kollektion dokumentweise. Das Lesen eines Dokuments beginnt in der Wurzel und durchläuft rekursiv jeweils alle Kindelemente. Für jedes Element wird der Abgleich mit dem internen Schema durchgeführt. Dies ist ein dreiteiliger Prozess:
Als Erstes wird überprüft, ob das Element bereits im Schema vorhanden ist. Ist das Element noch nicht im Schema enthalten, wird ein neues Element angelegt. Dazu werden Elementname, Pfad, Datentyp und die aktuelle Dokument-ID gespeichert. Ist das Element bereits im Schema vorhanden, müssen Pfad und Knotenname nicht erneut gespeichert werden.
Es kann direkt mit dem zweiten Schritt begonnen werden – dem Abgleich der Datentypen. Die Datentypen werden in einer `HashMap` gespeichert, also als Schlüssel-Wert-Paare. Als Schlüssel wird der Datentyp gespeichert. Im Wert befindet sich ein Zähler, der die Häufigkeit des Auftretens des Datentyps misst. Ist der Datentyp des aktuellen Dokuments bereits im Schema gespeichert, so muss lediglich der Zähler inkrementiert werden. Ist der Datentyp noch nicht vorhanden, wird ein neues Schlüssel-Wert-Paar eingefügt. Der Schlüssel entspricht dem aktuellen Datentyp, der Zähler wird auf Eins gesetzt.
Im letzten Schritt wird die Dokument-ID betrachtet. Diese wird ebenfalls in einer `HashMap` als Schlüssel-Wert-Paar gespeichert, um das Mehrfachauftreten von Elementen in Arrays zu registrieren. Bei Vorhandensein der Dokument-ID wird der Zähler um Eins erhöht. Andernfalls wird die Dokument-ID als Schlüssel mit Wert Eins der `HashMap` hinzugefügt.
Nachdem alle Dokumente durchlaufen wurden, befindet sich eine Repräsentation der Struktur der Dokumentkollektion in Form des internen Schemas im Arbeitsspeicher des Algorithmus. Dieses kann anschließend ausgegeben, gespeichert oder weiter verwendet werden.

3.2.2 Aktualisierungskomponente

Abbildung 13 zeigt die verschiedenen Varianten der Schema-Aktualisierung. Aus dem extrahierten Schema der Datenbank soll zum einen unter Verwendung der *geänderten Datensätze* und zum anderen mit Hilfe des *Update-Logs* ein aktualisiertes Schema erzeugt

werden. Da es sich bei den beiden Varianten um sehr unterschiedliche Prozesse handelt, werden sie folgend getrennt betrachtet.

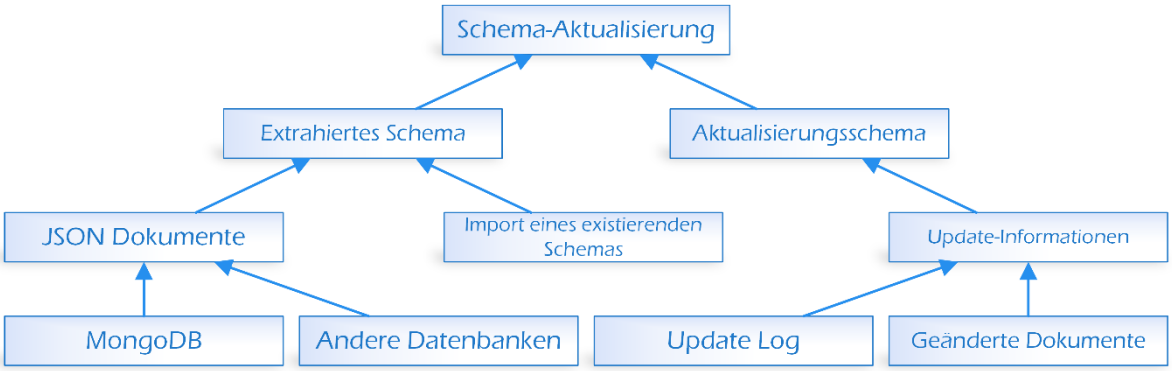

Abbildung 13: Gliederung der Aktualisierungskomponente nach Input-Möglichkeiten

Aktualisierung unter Vorlage der geänderten Dokumente:

Für die Aktualisierung des Schemas mit Hilfe der geänderten Dokumente gibt es zwei verschiedene Ansätze, die je aus zwei Bearbeitungsschritten bestehen. Beide Ansätze gehen dokumentweise vor. Zur eindeutigen Identifikation eines Dokuments dient die Dokument-ID, die zum einen im geänderten Dokument und zum anderen in den Elementen des internen Schemas gespeichert ist, die vor der Änderung im Dokument vorhanden waren.

In der ersten Variante der Aktualisierung mit Hilfe der geänderten Dokumente wird im ersten Schritt das geänderte Dokument wie ein weiteres Dokument der Extraktion behandelt. D.h. jedes Element des Dokuments wird durchlaufen und zum internen Schema durch Aufnahme des Elements oder Hinzufügen der Informationen zum bestehenden Element hinzugefügt. Sollte die Dokument-ID bereits vorhanden sein, wird nach dem Abgleich des Datentyps keine Änderung vorgenommen. In diesem Fall wurde an dem Element keine Änderung vorgenommen. In diesem Schritt werden neu hinzugefügte und geänderte Elemente im geänderten Dokument erkannt.

Nachdem alle Elemente des geänderten Dokuments durchlaufen wurden, müssen anschließend alle Elemente des internen Schemas, die nicht im Dokument vorkommen, durchlaufen und auf Vorhandensein der Dokument-ID des geänderten Dokuments überprüft werden. Gibt es ein solches Element, das die Dokument-ID des geänderten Dokuments enthält, so muss die Dokument-ID aus der Liste der Dokument-IDs entfernt werden und das Element für den Fall, dass die Dokument-ID Liste keine weiteren Einträge enthält, gelöscht werden. Dieser zweite Schritt deckt das Löschen von Elementen im geänderten Dokument ab.

Da bei dieser Variante jedoch sehr viele Vergleichsoperationen von zum Beispiel Häufigkeiten der Elemente, insbesondere in Arrays, durchgeführt werden müssen und im ersten Schritt auch eine zusätzliche Liste mit den bereits durchlaufenen Elementen erstellt werden muss, die dann im zweiten Schritt zum Abgleich der Elemente dient, wurde die zweite Variante der Aktualisierung mit Hilfe der geänderten Dokumente implementiert.

Diese zweite Variante besitzt zwei sehr ähnliche Schritte, wobei im ersten Schritt die Dokument-ID aus dem Aktualisierungsdokument entnommen wird und alle Vorkommen dieser Dokument-ID im internen Schema gelöscht werden. Nach diesem Schritt kommt das Aktualisierungsdokument nicht mehr im internen Schema vor und kann wie ein neues Dokument der Extraktion behandelt werden. Es muss keine Unterscheidung nach bereits

hinzugefügten und neuen Elementen gemacht werden. Das Dokument wird komplett neu in das Schema aufgenommen.
Bei beiden Prozessen ist zu beachten, dass komplett gelöschte Dokumente nicht erkannt werden können, wenn sie nicht als geändertes Dokument vorliegen. D.h. zum korrekten Löschen von Dokumenten müssen diese als leere Dokumente mit der entsprechenden Dokument-ID in die Menge der geänderten Dokumente eingefügt werden. Der Algorithmus behandelt Dokumente, die nur das Dokument-ID-Element besitzen, entsprechend als zu löschende Dokumente.

Gelöschte Dokumente müssen bei der Aktualisierung mit Hilfe der geänderten Dokumente als Dokumente, die nur das Dokument-ID-Element besitzen, aufgelistet werden.

Aktualisierung unter Vorlage des Update-Logs:
Update-Logs können sehr unterschiedlich gestaltet sein. Um eine Einbindung verschiedener Logs zu ermöglichen, soll eine eigene Update-Klasse implementiert werden. Ein auf diese Klasse aufbauendes Verfahren kann dann die Aktualisierung für alle Arten von Update-Logs realisieren. Es bedarf lediglich eines Mappings des jeweiligen Update-Logs mit der Update-Klasse. Aus dieser Klasse wird für jeden Aktualisierungsschritt ein Update-Objekt erzeugt. Dieses enthält alle Informationen, die für die Aktualisierung relevant sind:

- Elementname
- Pfad im Dokument
- Datentyp vor der Änderung
- Datentyp nach der Änderung
- Dokument-ID
- Art der Aktualisierung (Insert, Delete, Update)

Dabei kann es jedoch zur Diskrepanz zwischen vorhandenen und erwünschten Informationen kommen, sollte das Update-Log nicht alle Informationen bereitstellen. Dieser Fall wird im Abschnitt Grenzen der Aktualisierung diskutiert. Nachdem eine homogene Ausgangssituation geschaffen wurde, können die Änderungen in Form der Update-Objekte direkt in das Schema übernommen werden. Wird ein Element in einem Dokument gelöscht, so muss die Dokument-ID im entsprechenden Knoten gelöscht werden. Sollte es sich um die letzte Dokument-ID in diesem Knoten handeln, so muss der Knoten anschließend komplett aus dem Schema entfernt werden. Wird ein Knoten hinzugefügt, muss die Dokument-ID und ggf. der Datentyp zum Knoten im Schema hinzugefügt werden bzw. ein neuer Knoten zum Schema hinzugefügt werden, falls dieser noch nicht existiert. Bei Updates auf Werten muss lediglich der Datentyp kontrolliert und ggf. angepasst werden. Updates auf Schlüsseln können als Kombination aus Inserts und Deletes von Elementen behandelt werden.
Zu beachten ist bei dieser Variante, dass sich die Qualität des Update-Logs, also die verfügbaren Informationen über jeden Schritt, maßgeblich auf die Qualität des Schemas auswirkt. Zwingend erforderlich für eine korrekte Aktualisierung sind das Vorhandensein der Dokument-ID, des Pfads und des Elementnamens für jeden Updateschritt. Informationen zum Datentyp vor und nach der Aktualisierung sind je nach Update-Typ notwendig oder optional, verbessern aber in jedem Fall die Qualität des internen Schemas.

Verschiedene Update-Logs sollen mit Hilfe eines Mappings auf eine Update-Klasse eingebunden werden können.

Der Informationsumfang des Update-Logs schlägt sich maßgeblich auf die Qualität des Schemas nieder.

Grenzen der Aktualisierung

Wie in den einzelnen Varianten bereits angedeutet, kann es zu Problemen bei der Aktualisierung kommen, falls nicht alle benötigten Informationen über die Änderungen vorliegen. Zu unterscheiden sind kritische und optionale Informationen:

Fehlen *kritische Informationen*, so kann die Korrektheit des Schemas nicht mehr garantiert werden. Kritische Informationen sind die Informationen, die zur eindeutigen Identifikation des geänderten Elements beitragen, d.h. die Dokument-ID zur Erkennung des Dokuments in der Dokumentkollektion sowie Name und Pfad des Elements zur Erkennung des korrekten Elements im Schema. Ohne die Dokument-ID kann das geänderte Dokument im Schema nicht gefunden werden. Beim Hinzufügen eines Elements müsste dann eine Platzhalter-Nummer eingefügt werden, die spätestens beim Löschvorgang nicht mehr zugeordnet werden kann. Das Löschen von Elementen ohne Dokument-ID ist von vornherein nicht möglich, da nicht bekannt ist, welche der vorhandenen Dokument-IDs gelöscht werden muss. Ohne die Angabe von Elementname und Pfad kann das geänderte Dokument im Schema nicht eindeutig bestimmt werden. Wird zum Beispiel nur der Name des Elements angegeben, so kann nicht zwischen gleichnamigen Elementen im Schema unterschieden werden, sollte es diese Elemente geben. Dies ist insbesondere bei Änderungen an Datumsangaben in `MongoDB` sehr wahrscheinlich, da diese alle in einem speziellen `$date`-Element bzw. einem `$timestamp`-Element gespeichert werden. Aber auch datensatzspezifische, wiederkehrende Elementnamen, wie zum Beispiel personenbezogene Angaben, Adressen oder Telefonnummern, führen zu diesem Problem. Für eine korrekte Aktualisierung mit Hilfe des Update-Logs ist es zudem unerlässlich, dass atomare Updates angegeben werden. Wird zum Beispiel ein komplettes Array oder ein Objekt eines Arrays in einem einzigen Update-Schritt gelöscht, müssen die Kindelemente ebenfalls entfernt werden. Dazu wurden Mechanismen implementiert, die die Kindelemente automatisch suchen und löschen sollen. Allerdings haben diese Grenzen, bei deren Überschreitung sie die Kindelemente nicht mehr eindeutig zuordnen können. Zur korrekten Aktualisierung müssen also stets die Kindelemente explizit gelöscht werden.

Das Fehlen *optionaler Informationen* führt nicht zu einem ungültigen Schema, sondern lediglich zu einer möglichen Generalisierung des Schemas. Zu diesen optionalen Informationen zählen die Angaben der Datentypen vor der Änderung bei Updates und Deletes unter bestimmten Bedingungen. Fehlt diese Angabe bei einem Update kann und muss der neue Datentyp dem Element hinzugefügt werden, um die Korrektheit zu gewährleisten. Das Vorkommen des alten Datentyps kann jedoch nicht verringert und ggf. gelöscht werden, wodurch dieser womöglich noch im Schema vorkommt aber nicht mehr in der Dokumentkollektion. Dies führt jedoch nicht zu einem inkorrekten Schema, sondern nur zu einem möglicherweise generalisierten Schema, da alle Dokumente der Kollektion noch immer korrekt validiert werden können. Bei Deletes und Updates kann der Datentyp automatisch

ermittelt und gelöscht werden, falls das Element im Schema nur einen Datentyp besitzt. In diesem Fall wird er automatisch auch ohne explizite Angabe im Update-Objekt angepasst. Unter der Bedingung, dass sich nur atomare Update-Schritte im Update-Log befinden, kann der der alte Datentyp zudem vernachlässigt werden, falls er nicht automatisch ermittelt werden konnte. Der Datentyp bleibt im Schema und führt ggf. zur Generalisierung, nicht aber zu einem inkorrekten Schema. Dies gilt nicht für das Löschen oder Ändern von ganzen Objekten oder Arrays, da diese ggf. Kindelemente im Schema hinterlassen und so zu falschen Schlussfolgerungen von Pflichtelementen und Häufigkeiten führen.
Der Generalisierung beim Fehlen optionaler Informationen kann auf verschiedene Weisen entgegengewirkt werden. So kann zum Zeitpunkt der Aktualisierung die Dokumentkollektion durchlaufen werden und die Datentypen und ihre Häufigkeit erneut ermittelt werden. Dabei muss jedoch die komplette Dokumentkollektion durchlaufen werden, was dem Prinzip des inkrementellen Ansatzes widersprechen würde. Möglich wäre auch eine Umstrukturierung des internen Schemas, bei dem Knoten im Schema nicht nur durch Pfad und Name eindeutig bestimmt werden, sondern zusätzlich durch ihren Datentyp. Dies hätte eine Aufspaltung des Schemas in mehr Knoten zur Folge und würde beim Ableiten anderer Metriken, wie zum Beispiel Elementanzahl oder Häufigkeiten zu mehr Aufwand führen. Eine dritte Option ist das Tolerieren der Generalisierung. Da das Schema nicht ungültig wird, kann dieses weiter verwendet werden. Je nach Häufigkeit der Änderungen, Ansprüche an die Exaktheit und Wahrscheinlichkeit der Generalisierung, welche im Einzelfall sehr unterschiedlich sein können, können regelmäßige oder sporadische Neuextraktionen die Datentypen auf den aktuellen Stand bringen. Im Algorithmus soll die dritte Option umgesetzt werden.

Die Generalisierung durch fehlende Update-Informationen soll toleriert und ggf. durch sporadische Extraktionen entfernt werden.

3.2.3 Visualisierungskomponente

Die Visualisierungskomponente dient der Umwandlung des internen Schemas in das externe, JSON Schema konforme Schema. Dabei wird aus der Knotenmenge ein JSON Dokument erzeugt und dieses durch den `Gson-Builder`, einer Java-Klasse aus der `Gson`-Bibliothek, grafisch eingerückt ausgegeben. Zu jedem Knoten werden Name, Datentypen und Kindelemente angezeigt. Zusätzlich wird aus der Menge der Dokument-IDs die absolute und relative Häufigkeit des Auftretens berechnet und der Knoten im Elternknoten als Pflichtelement gekennzeichnet, falls dieser eine relative Häufigkeit von 100% besitzt. Die JSON Schema konformen Schlüsselwörter und Syntaxregeln werden angewendet. Die Ausgabe soll direkt nach der Extraktion bzw. Aktualisierung im Programm geschehen.

3.2.4 Speicherkomponente

Die Speicherkomponente soll es ermöglichen das interne Schema auf einem permanenten Medium abzuspeichern und von diesem wieder laden zu können, so dass das Schema dauerhaft gesichert werden kann und nicht bei jedem Programmstart erneut die gesamte Dokumentkollektion durchlaufen werden muss. Das Laden und Speichern soll problemlos, fehlerfrei und in vertretbarer Zeit erfolgen. Um möglichst unabhängig von anderer Software zu

bleiben, soll der Export in eine einfache Datei möglich sein. Dafür soll die Endung `.sav` verwendet werden, die zudem beim Laden eines Schemas voreingestellt sein soll.

3.2.5 Graphische Benutzeroberfläche

Eine GUI – kurz für *Graphical User Interface* – soll die Interaktion zwischen Mensch und Maschine, also User und Software, erleichtern. Sie soll die Funktionen übersichtlich darstellen und für eine leichte Erlernbarkeit sowie Benutzbarkeit der Software sorgen. Im Falle des Algorithmus sollen mit Hilfe der GUI einfache Bedienelemente zur Auswahl des Inputs, Ausführung der unterschiedlichen Funktionen und Konfiguration des Algorithmus zur Verfügung gestellt werden. Abbildung 14 zeigt die graphische Umsetzung der Extraktionskomponente. Im Anhang A – zusätzliche Abbildungen sind weitere Ausschnitte zu finden. [25]

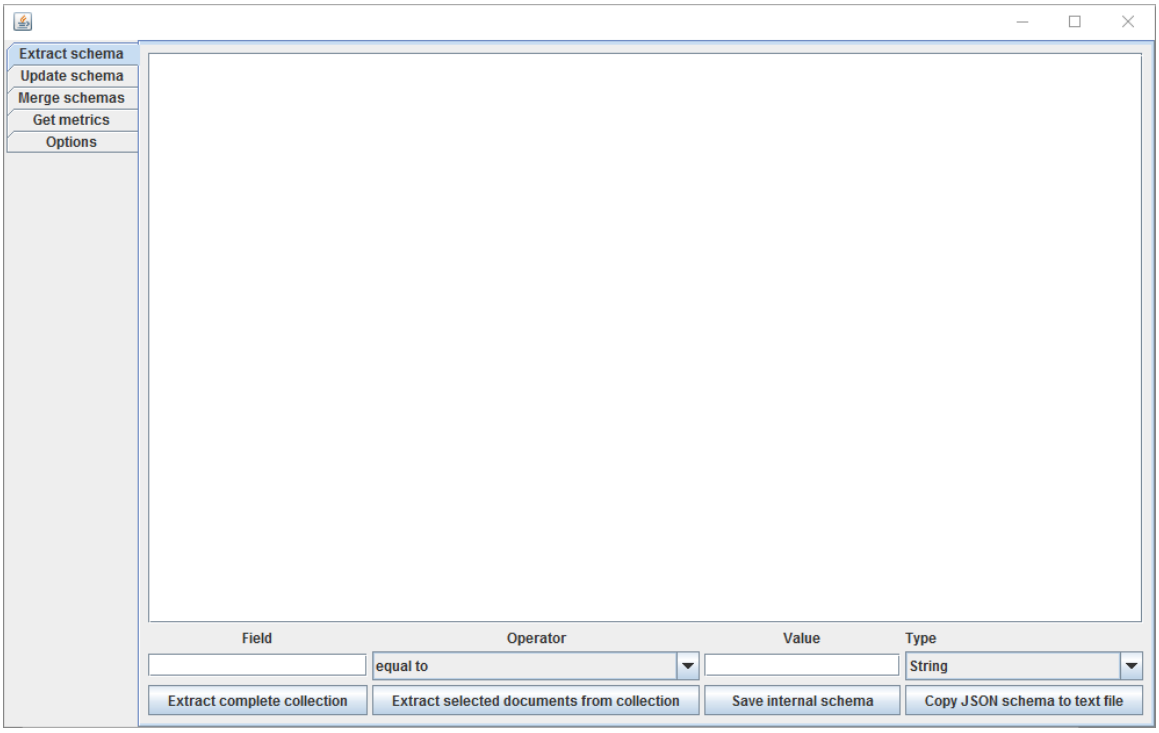

Abbildung 14: Ausschnitt der GUI

Neben der Interaktion soll die GUI auch der Ausgabe der Schemata dienen. Die Darstellung soll im JSON üblichen Format erfolgen und möglichst übersichtlich sein. Im Konfigurationsbereich sollen Einstellungen zur Datenbankverbindung für die Extraktion und Speicherorte für die Schemata festgelegt werden können. Zudem sind Präferenzeinstellungen möglich, die die Ausgabe optionaler Informationen und die Formatierung betreffen. Um die Konfiguration auch über den Programmablauf hinweg speichern zu können, soll diese beim Beenden des Programms in eine Konfigurationsdatei im Programmverzeichnis geschrieben werden. Beim Start des Programms soll entsprechend auf Vorhandensein dieser Datei geprüft und ggf. die Konfiguration geladen werden.

3.2.6 Qualitätssicherung des Algorithmus

Die Produktqualität stellt ein häufig einseitig betrachtetes Merkmal von Software dar. So zählen zu einer guten Produktqualität nicht nur die eigentliche Funktionalität, die oft im Fokus der Entwicklung steht, sondern auch weitere Kriterien, die sich auf den Umgang mit und die Wartung von Software beziehen. Die ISO 25010 [26] stellt ein solches Modell zur

Sicherstellung der Produktqualität dar und nennt die folgenden acht Kriterien, die die Qualität von Software beeinflussen:

- Funktionale Eignung
- Performance
- Kompatibilität
- Benutzbarkeit
- Zuverlässigkeit
- Sicherheit
- Wartbarkeit
- Übertragbarkeit

Diese Kriterien sollen bei der Entwicklung der Software berücksichtigt werden. Folgend werden für jedes Kriterium Anforderungen an die Software definiert, die die Einhaltung eben jenes Kriteriums garantieren sollen.

Unter *funktionaler Eignung* wird der Grad verstanden, zu dem ein Produkt oder System die Funktionen bereitstellt, die implizit oder explizit zur Benutzung notwendig sind. Zu diesem Punkt zählt die Erfüllung der Anforderungen an die Funktionalität der Software. Als implizite Funktionalität werden das Speichern und Laden der Schemata sowie verschiedene Benutzeroberflächen für Schema-Extraktion, Schema-Aktualisierung und Schema-Merging angesehen. Auch die Korrektheit und Angemessenheit der Funktionalität sind für die Erfüllung dieses Kriteriums wichtig. Explizit soll der Algorithmus ein Schema aus einer Dokumentkollektion in `MongoDB` extrahieren können, dieses mit Hilfe von Update-Informationen aus einem Update-Log oder durch geänderte Dokumente aktualisieren können, das Schema in vereinfachter Form ausgeben können und eine vereinfachte Variante der Schema-Extraktion zur schnellen Ableitung von relevanten Metriken zur Verfügung stellen.

Eine gute *Performance* zeichnet sich unter anderem durch einen geringen Speicher- und Zeitbedarf aus. Dies soll durch effiziente Speicherstrukturen und möglichst redundanzfreie Speicherung von Informationen gewährleistet werden. Die Performance wird gesondert in Abschnitt 5.1 betrachtet.

Kompatibilität soll durch die Definition eigener Schnittstellen geschaffen werden, die Unabhängigkeit von bestimmten Systemen und eine einfache Anbindung neuer Systeme durch die Einrichtung eines Mappings garantieren. Der Import aus `MongoDB` soll standardmäßig zur Verfügung gestellt werden. Auch für die Aktualisierung soll eine Schnittstelle eine einfache Anbindung verschiedener Quellen für die Update-Informationen bieten. Der Import von geänderten Dokumenten aus `MongoDB` sowie von Update-Logs im csv-Format und als JSON Objekte soll bereitgestellt werden. Der Output als JSON Schema verwendet gezielt einen unabhängigen Standard.

Eine grafische Benutzeroberfläche (GUI) soll für eine gute *Benutzbarkeit*, insbesondere eine leichte Erlernbarkeit, gutes Verständnis der Funktionen und eine angenehme Bedienung des Programms, sorgen.

Durch den Einsatz ausgereifter Standards, wie JSON als Datenformat, JSON Schema als Beschreibungssprache, Java als Programmiersprache und `Gson` als JSON-Parser, soll eine erhöhte *Zuverlässigkeit* erreicht werden. Zudem soll ein eigenes Fehlermanagement mit aussagekräftigen Fehlermeldungen eine einfache Behebung auftretender Fehler ermöglichen.

Das Thema *Datenschutz* und *Datensicherheit* wird in der Software nicht weiter behandelt, da keine eigenen Daten erzeugt werden, sondern lediglich fremde Daten verarbeitet werden. Somit liegt der Zugriffsschutz in den Quelldaten und außerhalb des Programms. Die Daten selbst werden durch die Software nicht verändert.
Bei der Software soll es sich um ein abgeschlossenes Programm handeln, welches keiner weiteren Änderungen bedarf. Trotzdem ist durch die Untergliederung des Programms in einzelne Komponenten, die weitestgehend unabhängig voneinander agieren, eine gute *Wartbarkeit* garantiert.
Das Programm wird als „`Executable Jar File`“ zur Verfügung gestellt und kann somit auf jeden Java-fähigen Computer *übertragen* werden. Eine Installation ist nicht notwendig. Alle Konfigurationen können im Programm selbst vorgenommen werden. Diese werden beim Beenden des Programms in eine Konfigurationsdatei im Programmverzeichnis gespeichert.

3.3 Ableitung der Metriken

Metriken können übersichtliche Informationen über einen großen Datenbestand liefern. Im Kontext der Schema-Extraktion, Schema-Evolution und Datenmigration werden folgende Metriken als relevant erachtet: *Ausreißerdokumente*, *potenziell gleiche Knoten* und *absolute und relative Häufigkeiten*. Diese aus einem Datenbestand abzuleiten, erfordert dabei nicht alle Informationen, die für die Schema-Aktualisierung benötigt werden. Insbesondere die speicherintensiven Informationen zur eindeutigen Elementidentifikation sind nicht erforderlich. Es soll daher ein vereinfachter und schnellerer Extraktionsprozess implementiert werden, der nur für die Ableitung und Ausgabe dieser Metriken verantwortlich ist. Die Ausgabe soll zusammen mit einem JSON Schema erfolgen, damit die Metriken direkt am Datenbestand nachvollzogen werden können. Abbildung 15 stellt die vereinfachte Architektur zum Ableiten der Metriken dar.

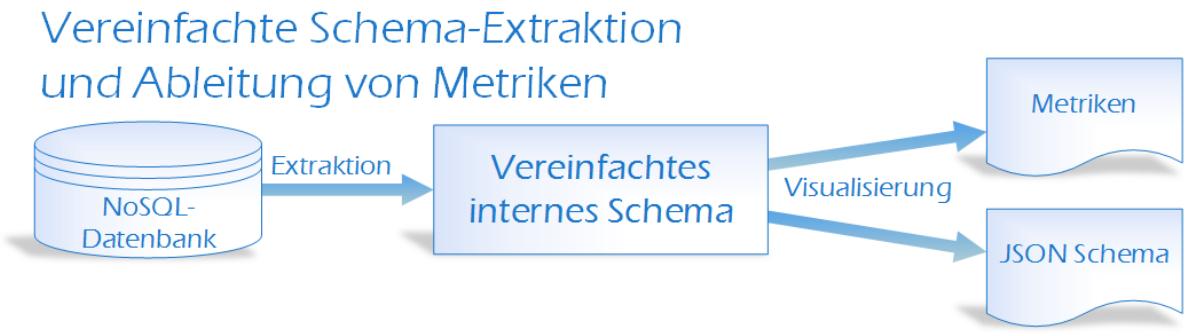

Abbildung 15: Systemarchitektur der vereinfachten Extraktion zur Ableitung von Metriken

3.3.1 Ausreißerdokumente

Ausreißerdokumente sind die Dokumente, die sich aus der homogenen Masse der Dokumente einer Kollektion abheben. Nicht in jeder Kollektion gibt es solche Ausreißer. Es bedarf einer großen Menge gleichartiger Dokumente und einigen wenigen Dokumenten, die sich durch ihre Struktur in einzelnen Elementen oder dem gesamten Aufbau von den anderen Dokumenten abheben.

Ausreißerdokumente sollen mit Hilfe eines prozentualen Schwellenwertes berechnet werden, den der Benutzer festlegen kann.

Erkannt werden können diese Dokumente durch das Zählen der Häufigkeiten eines jeden Elements. Die Gesamtanzahl der Dokumente einer Kollektion ist bekannt. Bei homogenen Daten ist eine Elementanzahl, die einem Ein- oder Vielfachen der Dokumentanzahl entspricht, zu erwarten. Nachdem alle Elementanzahlen ermittelt wurden, können diese mit der Elementanzahl ihres Elternelements verglichen werden und so die Ausreißerelemente ermittelt werden. Die Ausreißerelemente können daraufhin in einer Datenbankanfrage oder direkt im Programm durch Spezifikation einer Query bei der vollständigen Extraktion dazu verwendet werden, die Ausreißerdokumente zu finden. Insbesondere in Hinblick auf NoSQL-Datenbanken in Webanwendungen mit sich entwickelnden Schemata, deren Daten von Version zu Version migriert werden, sollen so veraltete Dokumente erkannt werden können. Bei einer *Lazy Migration* kann zum Beispiel nach einer zu definierenden Zeitspanne von veralteten, nicht mehr benötigten Daten ausgegangen werden, die zur Einsparung von Ressourcen gelöscht werden können.

3.3.2 Potenziell gleiche Elemente

Zu potenziell gleichen Elementen zählen die Elemente, die im Schema nicht an derselben Stelle stehen, aber trotzdem eine sehr ähnliche Struktur besitzen. In der Praxis entsprechen sie nicht migrierten bzw. vergessenen Elementen in Dokumenten oder redundant gespeicherten Daten. Durch die Definition von Ähnlichkeitsmaßen sollen solche Elemente ermittelt werden. Im Rahmen dieser Arbeit soll lediglich auf syntaktischer Ebene nach potenziell gleichen Elementen gesucht werden. Komplexere Verfahren, denen Thesauri oder Taxonomien zugrunde liegen, werden nicht implementiert.

Elemente sind im Schema durch Namen und Pfad eindeutig. Um potenziell gleiche Elemente zu ermitteln sollen zwei verschiedene Kombinationen betrachtet werden. Zum einen wird eine Kombination aus den Namen des Knotens und Elternknotens als Ähnlichkeitsmaß genommen. Stimmen bei zwei Knoten sowohl der Name des Knotens als auch des Elternknotens, nicht aber der vollständige Pfad überein, sollen die beiden Elemente als potenziell gleich markiert werden. Ein zweites Ähnlichkeitsmaß wird durch die Betrachtung des Elementnamens und aller Kindelemente definiert. Stimmen sowohl der Elementname als auch die Namen aller Kindelemente der betrachteten Knoten überein, werden sie als potenziell gleich markiert. Aufgrund der Betrachtung aller Kindelemente und nicht nur eines Elternelements führt dieses Maß zu einem stärkeren Indikator der Ähnlichkeit, denn jedes Elementpaar besitzt per Definition mindestens ein Element-Elternelement-Paar des ersten Ähnlichkeitsmaßes. Gefundene Kandidaten sollen lediglich ausgegeben werden, um nach manueller Überprüfung weiter bearbeitet zu werden.

3.3.3 Absolute und relative Häufigkeiten

Die absoluten und relativen Häufigkeiten lassen sich aus der Anzahl der Elternelemente und der Anzahl der Kindelemente ermitteln. Beim vollständigen Schema kann die Berechnung der Häufigkeiten jedoch noch genauer durchgeführt werden, da zum einen die Anzahl der Dokument-IDs und zum anderen auch die Elementanzahl des Kind- und des Elternelements verglichen werden können. Die Elementanzahl lässt sich als Summe $\sum_{i=1}^{n} H_i$ mit der absoluten Häufigkeit H einer Dokument-ID und n Dokument-IDs des Elements berechnen. Durch diese Unterscheidung der Element- und Dokumenthäufigkeiten kann auch für Elemente in Arrays eine korrekte Häufigkeitsangabe garantiert werden. Für das vereinfachte Schema steht nur die Elementanzahl zur Verfügung, da die Dokument-IDs nicht einzeln gespeichert, sondern direkt

zusammengezählt werden. Abbildung 16 soll den Unterschied der beiden Häufigkeiten verdeutlichen.

```
// JSON Dokumente
{
   „_id“ : „ObjectID(1)“,
   „array“ : [ 1, 2, 3, 4]
},
{
   „_id“ : „ObjectID(2)“,
   „array“ : []
}

// einfache Darstellung der Häufigkeiten der Elemente
„_id“    : „2/2 – 100%“
„array“  : „2/2 – 100%“
„number“ : „4/2 – 200%“

// Darstellung mit Unterscheidung nach Dokumentanzahl und Elementanzahl
„_id“    : „document occurence: 2/2; element occurrence 2/2 – 100%”
„array“  : „document occurence: 2/2; element occurrence 2/2 – 100%”
„number“ : „document occurence: 1/2; element occurrence 4/2 – 200%”
```

Abbildung 16: JSON Beispiel zur Verdeutlichung der unterschiedlichen Häufigkeiten

Das `number`-Element des Arrays erscheint bei der einfachen Darstellung als Pflichtelement, obwohl es in einem der beiden Arrays nicht vorkommt. Jedoch gibt erst die Unterscheidung nach der Dokumentanzahl darüber Aufschluss.
Die Ausgabe erfolgt in beiden Schemavarianten im `description`-Feld der Elemente. Im Schema der vollständigen Extraktion wird die Darstellung mit Unterscheidung nach Dokumentanzahl und Elementanzahl gewählt, falls diese voneinander abweichen.

4 Implementation

Die Implementation beschäftigt sich mit der Umsetzung des Konzepts in die Programmiersprache Java. Auch an dieser Stelle sollen die Komponenten einzeln betrachtet und ihre Arbeitsweise mit Hilfe von Pseudocode erläutert werden.
Die Implementation erfolgte in der *Eclipse IDE for Java Developers (Version: Mars.1 Release(4.5.1))* unter Verwendung von *Java Version 8 Update 73*. Die Anbindung an eine NoSQL-Datenbank wurde am Beispiel von *MongoDB Version 3.0.7* durchgeführt. Zur Integration in Java wurde der *MongoDB Java Driver* in der *Version 3.0.4*. verwendet. Die Bearbeitung der JSON Dokumente und JSON Elemente geschah mit Hilfe der *Gson-Bibliothek* in der *Version 2.4*.

4.1 Internes Schema

In diesem Abschnitt soll auf die Umsetzung des internen Schemas eingegangen werden. Neben den in Abschnitt 3.1.1 spezifizierten Anforderungen an das interne Schema soll hier auch die Performance eine maßgebliche Rolle bei der Eingrenzung der Implementationsvarianten spielen. Zu diesem Zweck wurden verschiedene Implementationen getestet:

Schema als Knoten- und Kantenmenge	Schema als Knotenmenge

`ArrayList Knotenmenge (`
`  Map Knoten (`
`  Map ID (ID, Knotenname),`
`  ArrayList Attribute (`
`  ArrayList Dokument-IDs,`
`  ArrayList Datentypen`
`  )`
`  )`
`)`
`ArrayList Kantenmenge (`
`  Map Kante (`
`  Map ID (Knoten-ID, Knoten-ID),`
`  ArrayList (Dokument-IDs)`
`  )`
`)`
	`ArrayList Knotenmenge (`
	`  Map Knoten (`
	`  Map ID (Knotenname, Pfad),`
	`  ArrayList Attribute (`
	`  HashMap (Dokument-ID, Zähler)`
	`  HashMap (Datentypen, Zähler)`
	`  )`
	`  )`
	`)`

Tabelle 2: Speichervarianten des internen Schemas

Tabelle 2 zeigt zwei Varianten zur Speicherung des internen Schemas. Die linke Variante stellt die erste, intuitive Implementation dar. Es wurden `ArrayLists` und `Maps` zur Speicherung von Knoten und Kanten verwendet. Diese Form der Speicherung erinnert auch an den *Spanning Graph* des *DTD-Miners*. Alle Informationen sind in den beiden Mengen verteilt und können zur erfolgreichen Rekonstruktion eines JSON Schemas kombiniert werden. Jede Dokument-ID wird jedoch redundant sowohl in den Knoten als auch in den Kanten gespeichert. Zudem bricht die Performance von `ArrayLists` ab einer gewissen Größe und bei sehr vielen Einfüge-Operationen stark ein. Mehr dazu im Abschnitt zur Performancebetrachtung. In der zweiten Variante wurden eben diese Probleme behoben, indem der Pfad und die Zählung der Häufigkeit aus den Kanten in die Knoten übernommen und die Kantenmenge gänzlich gestrichen wurde. Die `ArrayLists` für die Dokument-IDs und Datentypen, welche zu den genannten Performanceproblemen führten, wurden durch `HashMaps` ersetzt, die zudem den Zähler integrieren, der davor durch die Anzahl der Kanten dargestellt wurde. Dieser zählt die Häufigkeit des Vorkommens des Datentyps bzw. der Dokument-ID, also die Anzahl des Elements im entsprechenden Dokument, was zur Speicherung von Arrays notwendig ist.

Die dritte und performanteste Umsetzung wird in Tabelle 3 gezeigt. Es handelt sich um eine eigens implementierte Klasse für die Knoten. In dieser werden die Werte, die vorher in dem Konstrukt aus `ArrayLists` und `Maps` gespeichert wurden, als Attribute angelegt und mit Hilfe eigener „`Getter & Setter`“-Methoden gesetzt und abgerufen. Eine Hierarchieebene wurde als Attribut hinzugefügt, um schnell auf das Wurzelelement und Elemente bestimmter Ebenen zugreifen zu können. Zudem bietet die Implementation als Knotenklasse die Möglichkeit das Verhalten gewisser Operationen auf den Attributen der Knotenmenge zu definieren. So wurden einige „`Getter & Setter`“-Methoden angepasst und weitere Methoden implementiert. Eine zusätzlich hinzugefügte `HashMap` zur Speicherung der Reihenfolge der Arrayelemente erlaubt zudem nicht nur die Angabe der Häufigkeit des Auftretens eines jeden Elements, sondern die Angabe der korrekten Reihenfolge.

Knotenklasse node

Attribute

- String nodeName
- Integer level
- ArrayList<String> path
- HashMap<String, Integer> propTypes (propType, counter)
- HashMap<String, Integer> docIds (docId, counter)
- HashMap<Integer, Entry<String, ArrayList<Integer>>> arrayOrder (position, (propType, (minOcc, maxOcc)))

Methoden

- Getter & Setter (nodeName, level, path)
- getPropType()
- setPropType(newPropType, count)
  - if (newPropType exists in propTypes)
    - add count to counter of newPropType in propTypes
  - else
    - add newPropType with count to propTypes
- getDocId()
- setDocId(newDocId, count)
  - if (newDocIc exists in docIds)
    - add count to counter of newDocId on docIds
  - else
    - add newDocId with count to docIds
- setMaximumDocId(newDocId, counter)
  - if (newDocIc exists in docIds)
    - add count to counter of newDocId on docIds
  - else if (count > counter)
    - replace counter of newDocId with count in docIds
- countDocID()
  - return size of docIds
- countMembers()
  - int count = 0
  - for (int i : docIds.values())
    - count = count + i
  - return count
- deleteDocId(docId)
  - if (docId exists in docIds)
    - if (counter of docId == 1)
      - remove docId from docIds
      - if (size of docIds == 0)
        - return „NoDocumentsLeft“
      - else

```
                        o  return „Ok“
            ▪  else
                  •  return „Ok“
      o  else
            ▪  return „Error“
-  deleteCompleteDocId(docId)
      o  if (docId exists in docIds)
            ▪  if (size of propTypes == 1)
                  •  propType = propTypes[0]
                  •  replace counter of proptype with (counter - counter
                     of docId) in propTypes
            ▪  remove docId from docIds
            ▪  if (size of docIds == 0)
                  •  return „NoDocumentsLeft“
            ▪  else
                  •  return „Ok“
      o  else
            ▪  return „Error“
-  deletePropType(propType)
      o  if (propType exists in propTypes)
            ▪  if(counter of propType == 1)
                  •  remove propType from propTypes
            ▪  else
                  •  replace counter of propType with (counter - 1) in
                     propTypes
      o  if (propType equals "JsonArray")
            ▪  // set minOcc of every element in the array to 0
            ▪  if (0 not exists in arrayOrder)
                  •  for (Integer i = 1; i <= arrayOrder.size(); i++)
                        o  minMaxOcc = arrayOrder.get(i).getValue()
                        o  minMaxOcc.set(0, 0)
                        o  arrayOrder.get(i).setValue(minMaxOcc)
                  •  emptyArray = ("empty", new ArrayList<Integer>())
                  •  arrayOrder.put(0, emptyArray)
```

```
- getArrayOrder()
    o return arrayOrder
- setArrayOrder(newArrayOrder)
    o if (0 not exists in arrayOrder)
        ▪ if (arrayOrder is empty)
            • Boolean firstOrder = true
        ▪ Integer j = 1
        ▪ for (Integer i = 1; i <= newArrayOrder.size(); i++)
            • currentEntry = newArrayOrder.get(i)
            • if (j not exists in arrayOrder)
                o create newEntry
                o if firstOrder == true
                    ▪ set minOcc of newEntry to Occ of currentEntry
                o else
                    ▪ set minOcc of newEntry to 0
                o set maxOcc of newEntry to Occ of currentEntry
                o add (j, newEntry) to arrayOrder
            • else if (currentEntry equals arrayOrder.get(j))
                o if (Occ of currentEntry < minOcc of arrayOrder.get(j))
                    ▪ replace minOcc with Occ
                o if (Occ of currentEntry > maxOcc of arrayOrder.get(j))
                    ▪ replace maxOcc with Occ
            • else
                o set minOcc of arrayOrder.get(j) to 0
                o i--
            • j++
        ▪ if (arrayOrder.get(j) exists)
            • while (arrayOrder.get(j) exists)
                o set minOcc of arrayOrder.get(j) to 0
                o j++
    o else // 0 exists in newArrayOrder
        ▪ // newArrayOrder represents empty array
        ▪ // set all minOccurences to 0 and set position 0 flag
        ▪ if (0 not exists in arrayOrder)
            • for (Integer i = 1; i <= arrayOrder.size(); i++)
                o set minOcc of arrayOrder.get(i) zo 0
            • add (0, emptyEntry) to arrayOrder
        ▪ // else not the first empty array -> minOccurences already set to 0
- replaceCollectionName(newName)
    o if level > 0
        ▪ replace path[1] with newName
    o else
        ▪ replace nodeName = newName
- replaceDatabaseName(newName)
    o replace path[0] with newName
```

Tabelle 3: Attribute und Methoden der Knotenklasse

4.2 Extraktionskomponente

Die Extraktionskomponente besteht aus zwei Klassen, die jeweils mehrere Methoden zur Verfügung stellen. Die `extractFromMongoCollection`-Klasse ist für das Abrufen der Dokumente aus der Datenbank und der dokumentweisen Übergabe der Kollektion an die entsprechende Methode der zweiten, `extractFromJsonDocument`-Klasse zuständig. Diese übernimmt alle Schritte der Struktur-Extraktion aus einem einzelnen Dokument und der Speicherung dieser im internen Schema. Weitere Datenbanken können sehr einfach angebunden werden, indem eine neue Klasse für die Datenbank angelegt wird und diese die Dokumente an die `extractFromJsonDocument`-Klasse übergibt, die komplett datenbankunabhängig agiert.
Die Extraktion muss jedoch zuvor gestartet werden. Dies geschieht über die GUI. Dabei wird die in den Optionen festgelegte Kollektion extrahiert. Neben der Extraktion aller Dokumente der Kollektion gibt es die Möglichkeit durch Spezifikation einer Query die Extraktion auf bestimmte Dokumente zu beschränken. Nachdem die Extraktion durch Klick auf einen der Extraktionsbuttons gestartet wurde, werden Datenbank- und Kollektionsname sowie ggf. die Query-Parameter an eine Methode der `extractFromMongoCollection`-Klasse übergeben. Diese beginnt mit dem Aufbau einer Datenbankverbindung und dem Abrufen der Kollektion. Falls eine Query angegeben wurde, wird diese in eine `MongoDB`-Query umgewandelt und nur die entsprechenden Dokumente werden extrahiert. Nachdem die Dokumente aus der Datenbank geholt wurden, muss deren Struktur extrahiert werden. Dies geschieht dokumentweise durch Übergabe eines jeden Dokuments an eine entsprechende Methode der `extractFromJsonDocument`-Klasse. Diese stellt unterschiedliche Methoden für die vereinfachte sowie vollständige Extraktion, für die Aktualisierung, zur Ermittlung des JSON Datentyps sowie zur Speicherung der Elemente im internen Schema bereit. Nach der Übergabe eines Dokuments wird dessen Struktur entsprechend elementweise in das Schema übernommen. Dazu wird im ersten Schritt das gesamte Dokument durch einen `Gson-Parser` als `JsonElement` eingelesen. Ein `JsonElement` ist entweder ein `JsonObject`, `JsonArray` oder `JsonPrimitive`. Der genaue Datentyp wird durch die `getJsonType`-Methode ermittelt und anschließend mitsamt aller übergebenen Informationen durch die `storeNode`-Methode als Element im internen Schema gespeichert. Im Fall von `Objects` und `Arrays` ruft sich die Extraktionsmethode rekursiv für jedes Kindelement genau einmal selbst auf und übergibt das Kindelement in Form eines `JsonElements` sowie weitere Informationen wie Knotenname, Pfad und Dokument-ID. Die hierarchische, baumartige Struktur eines jeden `Json Documents` wird dabei vollständig von der Wurzel bis zu jedem Blatt genau einmal durchlaufen. `Arrays` bedürfen einer speziellen Behandlung, da sie eine Strukturbesonderheit, die Reihenfolge ihrer Elemente, besitzen. Diese wird im Arrayknoten des Schemas gespeichert. Komplexere Methoden garantieren dabei die korrekte Speicherung der Reihenfolge sowie der Minima und Maxima. Die Elemente eines Arrays werden dabei durch ihren Datentyp unterschieden, da, wie im Abschnitt Arrayelemente diskutiert, eine eindeutige Unterscheidung von Objekten mit verschiedenen Kindelementen auf syntaktischer Ebene nicht möglich ist und diese über ihren Datentypen zusammengefasst werden sollen. Tabelle 4 illustriert den groben Ablauf der Schema-Extraktion und Tabelle 10 im Anhang stellt die einzelnen Methoden der Extraktionskomponente im Pseudocode etwas genauer dar.

Ablauf der Extraktion

GUI
- on button event
  - **extractFromMongoCollection**(database, collectionName, query)

extractFromMongoCollection.extractAll/query(database, collectionName)
- connect to database
- retrieve collection (optional with query)
- path = new ArrayList()
- add database to path
- foreach (document in collection)
  - docId = document.get("_id")
  - **extractFromJsonDocument**(parser.parse(document.toJson()), collectionName, path, docId, 0)
- close dbConnection

extractFromJsonDocument.extract(jsonElement, nodeName, path, docId, level)
- propType = **getJsonType**(jsonElement)
- **storeNode**(nodeName, path, propType, docId, level)
- newPath = path
- add nodeName to newPath
- if (node.isJsonobject)
  - foreach (key-value-pair in jsonElement)
    - **extract**(parser.parse(value), key, newPath, docId, level++)
- if (node.isJsonArray)
  - foreach (arrayElement in node)
    - **extract**(parser.parse(arrayElement), "ArrayElement", newPath, docId, level++)
    - save arrayOrder

extractFromJsonDocument.storeNode(name, path, propType, docId, level)
- if (node exists in nodeList)
  - add docId and propType
- else
  - add new node to nodeList

Tabelle 4: grobe Übersicht über den Ablauf der Schema-Extraktion

Die Speicherung eines Elements im internen Schema erfolgt durch Erzeugung eines Knotenobjekts der Knotenklasse. Dabei wird für jedes Element ein neues Objekt erzeugt, wobei sich die Elemente durch Knotenname und Pfad eindeutig unterscheiden – nicht aber durch ihre Dokument-ID. Ist ein Element bereits durch die Extraktion eines früheren Dokuments im internen Schema vorhanden, wird die aktuelle Dokument-ID diesem Knotenobjekt hinzugefügt und der Zähler des entsprechenden Datentyps erhöht. Dies wird durch Methoden, die die Knotenklasse selbst zur Verfügung stellt, verwirklicht. (siehe dazu Tabelle 3)
Sind alle Dokumente durchlaufen worden, ist die Extraktion abgeschlossen und das Schema der Dokumentkollektion befindet sich im internen Schema. Dieses wird im Anschluss an jeden Extraktionsvorgang direkt durch die Visualisierungskomponente im Programm ausgegeben und steht nun den anderen Komponenten, insbesondere der Aktualisierungskomponente, zur Verfügung.
Abbildung 17 veranschaulicht exemplarisch den Übergang von einem Array im JSON Dokument über die Repräsentation als Knoten im internen Schema bis hin zur Ausgabe als JSON Schema.

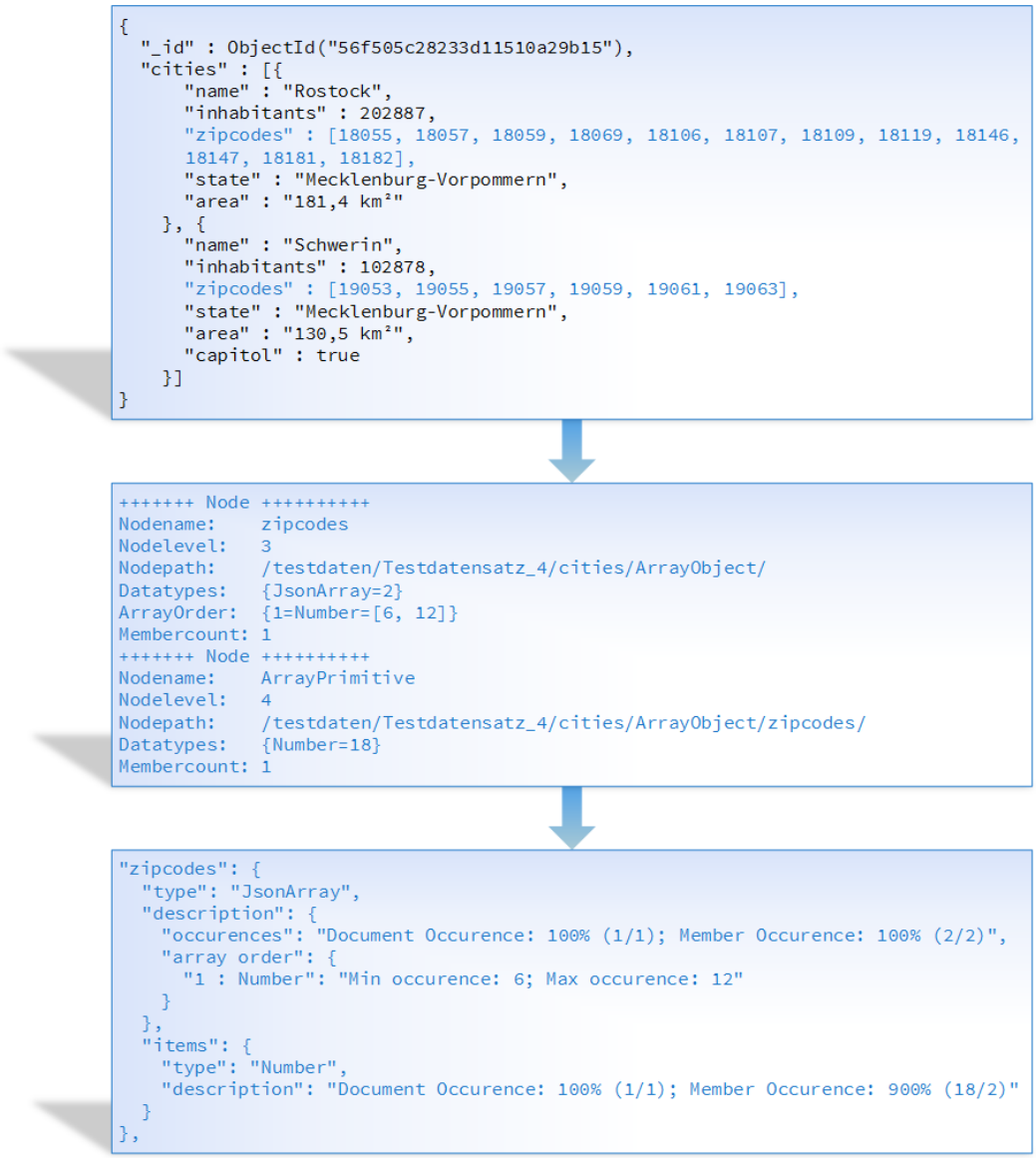

Abbildung 17: Schritte vom JSON Dokument zum JSON Schema

4.3 Aktualisierungskomponente

Die Aktualisierungskomponente soll das interne Schema auf zwei verschiedenen Wegen aktualisieren können. Zum einen ist die Aktualisierung mit Hilfe eines Update-Logs und zum anderen durch das Einlesen von geänderten Dokumenten vorgesehen. Voraussetzung dafür ist das Vorhandensein eines internen Schemas, welches aktualisiert werden kann. Dieses kann über die Benutzeroberfläche (siehe Abbildung 18) ausgewählt werden, indem entweder das durch die Extraktionskomponente erstellte Schema kopiert oder ein gespeichertes Schemas geöffnet wird. Anschließend muss die Aktualisierungsvariante gewählt und entweder das Update-Log oder die aktualisierten Dokumente übergeben werden. Daraufhin kann der eigentliche Aktualisierungsvorgang beginnen.

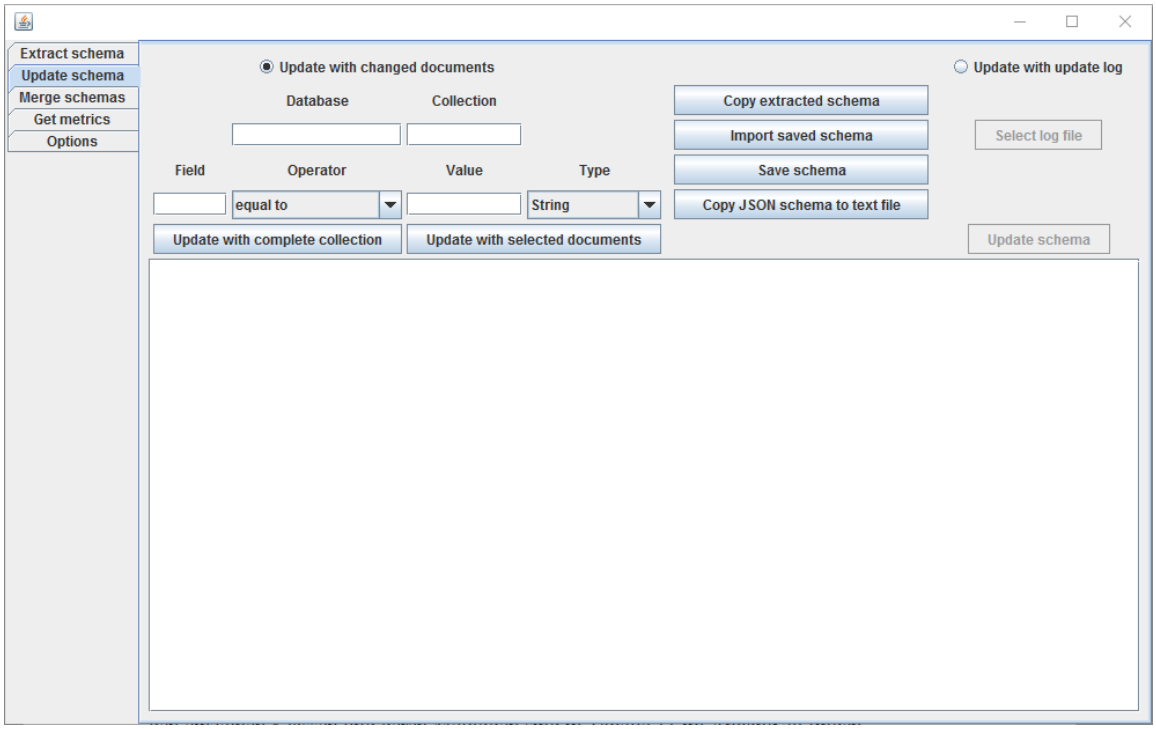

Abbildung 18: Benutzeroberfläche der Aktualisierungskomponente

Die Aktualisierungskomponente besitzt für jedes Verfahren eine Klasse: die `mergeWithChangedDocuments`-Klasse und die `mergeWithUpdateLog`-Klasse. Details zu den einzelnen Klassen und deren Methoden sind in Tabelle 11 im Anhang zu finden.

Updateklasse update
Attribute - String elementName - ArrayList<String> path - String propTypeBeforeChange - String propTypeAfterChange - String docId - String updateType
Methoden - Getter & Setter

Tabelle 5: Attribute und Methoden der Updateklasse

Aktualisierung unter Vorlage der geänderten Dokumente:

Beim Aktualisieren durch die geänderten Dokumente kann mit Hilfe geringfügiger Änderungen die Funktionalität der Extraktionskomponente wiederverwendet werden. Die Aktualisierung wird ebenfalls dokumentweise durchgeführt. Jedes Dokument kann dabei wie ein zusätzliches Dokument der Extraktion behandelt werden, falls vorher sichergestellt werden kann, dass keine Informationen aus diesem Dokument im Schema vorhanden sind.

Zunächst muss die Aktualisierung jedoch gestartet werden, was ebenfalls über die GUI geschieht. Zur Aktualisierung unter Vorlage der geänderten Dokumente müssen die geänderten Dokumente ausgewählt werden, indem eine Kollektion einer Datenbank angegeben wird. Zusätzlich können auch hier Einschränkungen auf der Kollektion mit Hilfe von Query-Parametern gemacht werden. Beim Klick auf einen der Aktualisierungsbuttons werden diese Informationen an entweder die `updateAll`- oder `updateQuery`-Methode der `mergeWithChangedDocuments`-Klasse übergeben. Diese überprüfen zunächst die übergebenen Datenbank- und Kollektionsnamen. Stimmen diese mit den in der Extraktion

verwendeten Namen überein, kann direkt mit dem nächsten Schritt fortgefahren werden. Stimmen sie nicht überein, so müssen die Pfadangaben der Knoten des internen Schemas mit den neuen Datenbank- und Kollektionsnamen überschrieben werden, da sonst der Vergleich der Elemente des internen Schemas mit den geänderten Dokumenten fehlschlagen würde und diese lediglich als neue Dokumente hinzugefügt werden würden. Grund dafür ist, dass die Datenbank und der Kollektionsname Bestandteil des Elementpfades sind und sich die Elemente über Namen und Pfad eindeutig identifizieren und somit zum Abgleich auf bereits vorhandene Elemente verwendet werden.
Nachdem sichergestellt wurde, dass Datenbank- und Kollektionsnamen übereinstimmen, werden diese mitsamt der optionalen Query-Parameter und unter Angabe eines Update-Flags an die entsprechende Methode der `extractFromMongoCollection`-Klasse der Extraktionskomponente übergeben. Diese baut wie bei der Extraktion eine Datenbankverbindung auf und holt die spezifizierten Dokumente aus der angegebenen Kollektion heraus. Anschließend können diese dokumentweise dem internen Schema hinzugefügt werden, wobei durch das Update-Flag vorher ein zusätzlicher Schritt durchgeführt wird, bei dem die Elemente des internen Schemas nach der aktuellen Dokument-ID durchsucht werden. Sollte die Dokument-ID vorhanden sein, wird diese gelöscht. Falls der Datentyp des gelöschten Elements eindeutig bestimmbar ist, wird dessen Zähler um den Zähler der gelöschten Dokument-ID verringert. Das nun bereinigte interne Schema kann anschließend um das aktuelle Dokument erweitert werden. Die Funktionalität dafür wird durch die Methoden der `extractFromJsonDocument`-Klasse bereitgestellt.

Aktualisierung unter Vorlage des Update-Logs:

Die Aktualisierung unter Vorlage des Update-Logs soll die Schritte aus einem Update-Log einer Datenbank nachvollziehen können und in das interne Schema übernehmen. Da Update-Logs jedoch nicht standardisiert sind, sehr unterschiedlich aufgebaut sein können und unterschiedliche Informationen enthalten können, sollen diese zunächst über eine Schnittstelle vereinheitlicht werden. Dazu wurde eine Updateklasse (siehe Tabelle 5) definiert, die alle benötigten und optionalen Informationen zu einem Update-Schritt in Form von Attributen speichert.
Der erste Schritt bei der Aktualisierung durch ein Update-Log ist folglich das Mapping des Update-Logs auf diese Updateklasse. Zwei Schnittstellen wurden beispielhaft implementiert. Zum einen können Updates als Semikolon-getrennte csv-Dateien eingelesen werden und zum anderen im JSON Format aus einer Textdatei. Dies geschieht durch Auswahl des Update-Logs in der GUI. Der ausgewählte Dateipfad wird der `importUpdates`-Methode der `mergeWithUpdateLog`-Klasse beim Betätigen des Aktualisierungsbuttons übergeben. Die `importUpdates`-Methode öffnet die ausgewählte Datei und liest die Updates zeilenweise in eine interne Updateliste ein. Nachdem alle Updates eingelesen wurden, ruft sie die `merge`-Methode derselben Klasse auf und übergibt dieser die Updateliste. In Tabelle 7 und Tabelle 8 ist ein Beispiel-Update-Log im csv-Format und in Abbildung 19 drei Beispiel-Updates im JSON-Format dargestellt. Für einen problemlosen Import müssen Updates als unformatierte JSON Strings in einer Zeile stehen. Zur besseren Darstellung wurden sie hier jedoch in das typische JSON Format gebracht. Beim csv-Format stehen in den ersten fünf Spalten die angegebenen Informationen. Alle weiteren Spalten werden zur Angabe des Pfads verwendet, welcher beim Import wieder zusammengesetzt wird. JSON erlaubt jedes Zeichen im

Elementnamen. Diese Darstellung wurde gewählt, um neben dem durch das csv-Format gesperrte Semikolon, kein weiteres Trennzeichen verbieten zu müssen. Semikolons im Elementnamen sind folglich bei Updates im csv-Format problematisch und führen zu fehlerhaften Update-Objekten. Ein Semikolon im Elementnamen würde die erste Spalte zweiteilen und somit eine Verschiebung aller folgenden Spalten verursachen. Ein Semikolon im Pfad hätte eine falsche Pfadangabe zur Folge.

Elementname	Datentyp vor der Änderung	Datentyp nach der Änderung	Dokument-ID
borough	String	Number	5646d9595d3511bfd4866558
date	JsonObject		5646d9595d3511bfd4866558
kitchen		String	5646d9595d3511bfd4866558

Tabelle 7: Auszug eines Update-Logs im csv-Format (Teil 1)

Update-Typ	Pfad-Element	Pfad-Element	Pfad-Element
update	testdaten	Testdatensatz_1	
delete	testdaten	Testdatensatz_1	grades
insert	testdaten	Testdatensatz_1	

Tabelle 8: Auszug eines Update-Logs im csv-Format (Teil 2)

In der `merge`-Methode werden grundsätzlich drei Update-Typen, angelehnt an die drei grundlegenden Datenbankoperationen, unterschieden: `Inserts`, `Deletes` und `Updates`, welche jeweils gesondert behandelt werden.

Beim `Insert` wird das interne Schema auf Vorhandensein des Elements überprüft und dieses entweder dem existierenden Element oder als neues Element dem Schema hinzugefügt.

Beim `Update` wird ebenfalls das Element im internen Schema gesucht, allerdings wird ein Fehler ausgeworfen, falls dieses nicht gefunden wird. Im zweiten Schritt wird der Datentyp des Update-Elements vor der Änderung mit dem Datentyp nach der Änderung verglichen. Bei herkömmlichen Updates ändert sich oftmals nur der Wert, nicht aber der Datentyp. Stimmen beide Datentypen überein, muss das interne Schema nicht verändert werden. Stimmen die Datentypen jedoch nicht überein oder gibt es keine Angabe zum Datentypen vor der Änderung, muss der neue Datentyp dem Element im Schema hinzugefügt werden. Vorher wird versucht, den alten Datentypen aus dem Schema zu entfernen, um eine mögliche Generalisierung zu verhindern. (näheres dazu im Abschnitt Grenzen der Aktualisierung) Ist der Datentyp vor der Änderung im Update-Objekt angegeben, kann dieser einfach im Schema entfernt werden. Fehlt diese Angabe allerdings, bleibt nur die Betrachtung der Datentypenliste des Schema-Elements. Beinhaltet diese Liste nur einen Datentyp, so kann dessen Zähler um Eins reduziert werden, da der Datentyp vor der Änderung definitiv diesem Datentyp entsprach. Für den Fall, dass die Liste mehr als einen Datentyp enthält, kann nicht mit Sicherheit entschieden werden, welcher Datentyp gelöscht werden muss, und demzufolge wird keiner der Datentypen gelöscht und es kommt ggf. zur Generalisierung des internen Schemas.

Beim letzten Update-Typ, dem `Delete`, wird, wie beim `Update` auch, nach dem Element im internen Schema gesucht und ein Fehler ausgeworfen, falls dieses nicht gefunden werden kann. Falls das passende Element gefunden wurde, wird die im Update-Objekt hinterlegte Dokument-ID im Schema-Element dekrementiert und die Dokument-ID bzw. das komplette Element ggf. gelöscht, falls der Zähler der Dokument-ID auf null gesunken ist bzw. die letzte Dokument-ID des Knotens gelöscht wurde. Anschließend muss der Datentyp des gelöschten Elements betrachtet werden. Häufig wird es keine Angabe zum Datentyp vor der Änderung im Update-

Log geben. Es bleibt folglich nur die Methode, die bereits beim `Update` angewendet wird. Konnte der Datentyp ermittelt werden, so wird, falls es sich beim Datentyp um ein `Json Object` handelt, die `deleteChildren`-Methode und, falls es sich um ein `Json Array` handelt, die `deleteAllChildren`-Methode aufgerufen. Diese Methoden durchlaufen jeweils das interne Schema und löschen die korrekte Anzahl der Dokument-ID in den jeweiligen Kind- und Kindeskindelementen des gelöschten Elements, um die Korrektheit des Schemas sicherzustellen.


```
{
   "nodename" : "borough",
   "proptypebeforechange" : "string",
   "proptypeafterchange" : "Number",
   "documentid" : "5646d9595d3511bfd4866558",
   "updatetype" : "update",
   "path" : ["testdaten", "Testdatensatz_1_small"]
},
{
   "nodename" : "cuisine",
   "proptypebeforechange" : "string",
   "proptypeafterchange" : "",
   "documentid" : "5646d9595d3511bfd4866558",
   "updatetype" : "delete",
   "path" : ["testdaten", "Testdatensatz_1_small"]
},
{
   "nodename" : "kitchen",
   "proptypebeforechange" : "",
   "proptypeafterchange" : "string",
   "documentid" : "5646d9595d3511bfd4866558",
   "updatetype" : "insert",
   "path" : ["testdaten", "Testdatensatz_1_small"]
}
```


Abbildung 19: Drei Beispiel-Updates im JSON-Format

Sowohl beim `Update` als auch beim `Delete` wird ein zweiter Suchvorgang gestartet, falls im ersten Durchlauf kein passendes Element gefunden wurde. Dabei wird dem im Update-Objekt spezifizierten Pfad das `ArrayObject` angehängt und der Suchvorgang erneut gestartet. Der Umgang mit diesem im Schema und Dokument nicht ersichtlichen Element in den Update-Logs ist besonders unklar und soll durch diesen zweiten Suchdurchlauf als potenzielle Fehlerquelle ausgeschlossen werden. Nur, falls auch der zweite Suchdurchlauf erfolglos war, wird eine Fehlermeldung ausgegeben. Beim Insert kann dieser Schritt nicht vollzogen werden, da ein Insert nicht fehlschlägt, falls das Element nicht existiert, sondern dieses neu erstellt. Eine automatische Entscheidung, ob das Insert im Dokument an der angegebenen Stelle oder im `ArrayObject` stattfand, ist aus den Informationen des Update-Logs allein nicht möglich.
Im Anhang D – Schemata sind die zwei extrahierte Schemata sowie deren kombinierte Aktualisierung dargestellt.

4.4 Visualisierungskomponente

Die Visualisierungskomponente dient der Ausgabe, also dem Erstellen und Ausgeben von validen und wohlgeformten JSON Schemata. Wie bereits im Abschnitt Aufbau des Schemas beschrieben, gibt es zwei verschiedene Varianten des Schemas. Der Algorithmus arbeitet intern auf einer entsprechend als internes Schema bezeichneten Knotenmenge. Bei der Ausgabe gilt es nun, aus dieser Knotenmenge ein valides und wohlgeformtes JSON Schema zu

rekonstruieren. Das interne Schema stellt dazu alle benötigten Informationen zur Verfügung. Aufgabe der Visualisierungskomponente ist es, die Baumstruktur wiederherzustellen und jedes Element in seine JSON Repräsentation umzuwandeln. Das daraus entstehende JSON Dokument wird einem `Gson-Builder` aus der `Gson`-Bibliothek übergeben, der es mit Hilfe der `prettyPrinting`-Option in ein formatiertes JSON Schema umwandelt. Das vom Algorithmus abgeleitete JSON Schema der `cities`-Kollektion wird in Abbildung 20 und Abbildung 21 dargestellt. Die Dokument-ID wurde durch das Einfügen des Dokuments in eine `MongoDB`-Kollektion automatisch generiert.

```
{
  "title": "cities",
  "description": "Json Schema for collection: cities from database: testdaten, created on
2016-03-22 16:38:52",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "properties": {
    "_id": {
      "type": "JsonObject",
      "description": "Occurence: 100% (1/1)",
      "properties": {
        "$oid": {
          "type": "String",
          "description": "Occurence: 100% (1/1)"
        }
      },
      "required properties": [
        "$oid"
      ]
    },
    "cities": {
      "type": "JsonArray",
      "description": {
        "occurences": "Occurence: 100% (1/1)",
        "array order": {
          "1 : JsonObject": "Min occurence: 2; Max occurence: 2"
        }
      },
      "items": {
        "type": "JsonObject",
        "description": "Document Occurence: 100% (1/1); Member Occurence: 200% (2/1)",
        "properties": {
          "name": {
            "type": "String",
            "description": "Document Occurence: 100% (1/1); Member Occurence: 100% (2/2)"
          },
          "inhabitants": {
            "type": "Number",
            "description": "Document Occurence: 100% (1/1); Member Occurence: 100% (2/2)"
          },
```

Abbildung 20: Abgeleitetes JSON Schema der cities-Kollektion (Teil 1)

```
          "zipcodes": {
            "type": "JsonArray",
            "description": {
              "occurences": "Document Occurence: 100% (1/1); Member Occurence: 100% (2/2)",
              "array order": {
                "1 : Number": "Min occurence: 6; Max occurence: 12"
              }
            },
            "items": {
              "type": "Number",
              "description": "Document Occurence: 100% (1/1); Member Occurence: 900% (18/2)"
            }
          },
          "state": {
            "type": "String",
            "description": "Document Occurence: 100% (1/1); Member Occurence: 100% (2/2)"
          },
          "area": {
            "type": "String",
            "description": "Document Occurence: 100% (1/1); Member Occurence: 100% (2/2)"
          },
          "capitol": {
            "type": "Boolean",
            "description": "Occurence: 100% (1/1)"
          }
        },
        "required properties": [
          "name",
          "inhabitants",
          "zipcodes",
          "state",
          "area"
        ]
      }
    }
  },
  "required properties": [
    "_id",
    "cities"
  ]
}
```

Abbildung 21: Abgeleitetes JSON Schema der cities-Kollektion (Teil 2)

Das Konzept des Dokuments spielt bei der Erstellung des Schemas eine zentrale Rolle. Ein Dokument ist ein Container-Element, in das Schlüssel-Wert-Paare eingefügt werden können. Die Schlüssel sind Strings – in diesem Fall die JSON Schema Schlüsselwörter, wie `type`, `properties`, `items` oder `description`. Der Wertebereich ist mit dem abstrakten `Java-Object` typisiert und kann neben Arrays oder Strings auch weitere Dokumente beinhalten. So kann die verschachtelte Hierarchie des Schemas durch dieselbe `printElement`-Methode rekursiv zusammengesetzt werden. Zuvor muss jedoch das JSON Schema-Dokument mit allen Schemainformationen erstellt werden. Dazu wird die `print`-Methode der Visualisierungskomponente aufgerufen, die die Knotenmenge nach dem Wurzelknoten durchsucht und daraus das Wurzelelement des JSON Schemas erstellt, welches die geforderten JSON Schema-Elemente `title`, `description`, `$schema`, `type`, `properties` und `required` enthält. Anschließend werden die Elemente der ersten Ebene des Dokuments

gesucht und der `printElement`-Methode übergeben, die diese als Kindelemente der Wurzel unterordnet. Knotenname, Datentypen und Dokument-IDs füllen dabei entsprechend die benötigten Felder im JSON Schema. Abbildung 22 stellt diesen Prozess genauer dar. Der Visualisierungsmethode wird rekursiv Knotenname, Pfad und die Anzahl der Elemente des Elternknotens übergeben. Mit Pfad und Knotenname wird der Knoten in der Knotenmenge eindeutig identifiziert. Der Knotenname des Kindknotens füllt das `name`-Feld und die Datentypen das `type`-Feld, wobei überprüft werden muss, ob unter den Datentypen ein JSON Array oder JSON Object ist. Für Arrays wird ein `items`-Feld angelegt, für Objects ein `properties`-Feld. Diese Felder werden durch den rekursiven Aufruf der Visualisierungsmethode mit den Kindelementen gefüllt. Der Pfad des Kindknotens ergibt sich aus Pfad und Knotenname des Elternknotens. Für jeden gefundenen Kindknoten des Elternknotens wird die Visualisierungsmethode einmal aufgerufen. Zu beachten ist, dass für die Korrektheit des Algorithmus intern die exakten Dokument-IDs erforderlich sind. Im externen Schema werden diese lediglich als absolute und relative Häufigkeit im `description`-Feld ausgegeben, um die Übersicht zu wahren.

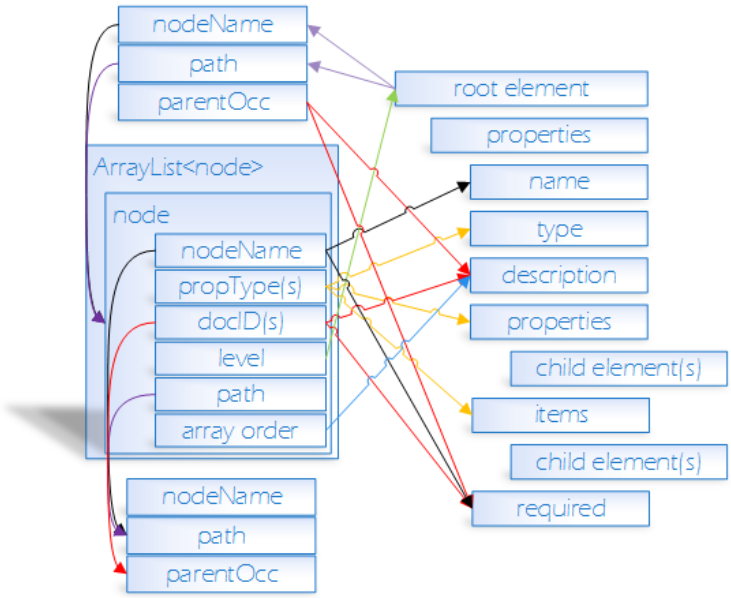

Abbildung 22: Rekonstruktion des externen Schemas aus der internen Knotenmenge

Implementiert wurde die Visualisierungskomponente mit Hilfe von zwei Klassen. Die `toJsonSchema`-Klasse stellt die Methoden zur Umwandlung des internen Schemas in ein JSON Dokument bereit. Dabei wird zwischen einfachem und vollständigem internen Schema unterschieden. Das entstandene JSON Dokument kann der `printSchema`-Methode der `showSchema`-Klasse übergeben werden, die mit Hilfe des `Gson-Builders` das JSON Schema formatiert auf dem Bildschirm ausgibt. Die einzelnen Methoden sind im Anhang in Tabelle 12 dargestellt.

Die Darstellung der Reihenfolge von Arrayelementen soll an folgenden Beispielen verdeutlicht werden.

```
// Array im JSON Dokument
{
   "array" : [1, 2, 3, 4, "a", "b", "c", 1, 2]
}

// Darstellung des Arrays im JSON Schema
"array": {
   "type": "JsonArray",
   “description": {
      "occurences": "Occurence: 100% (1/1)",
      "array order": {
         "1 : Number": "Min occurence: 4; Max occurence: 4",
         “2 : String": "Min occurence: 3; Max occurence: 3",
         "3 : Number": "Min occurence: 2; Max occurence: 2"
      }
   },
   "items": {
      "anyOf": [
         {
            "type": "Number",
            "description": "Document Occurence: 100% (1/1);
             Member Occurence: 600% (6/1)"
         },
         {
            "type": "String",
            "description": "Document Occurence: 100% (1/1);
             Member Occurence: 300% (3/1)"
         }
      ]
   }
}
```

Abbildung 23: Darstellung von Arrays - Beispiel 1

In Abbildung 23 ist die Darstellung eines Arrays mit verschiedenen `Number`- und `String`-Elementen zu sehen. Zu beachten ist die im `description`-Feld angegebene korrekte Reihenfolge der Arrayelemente. Abbildung 24 verdeutlicht diese Darstellung mit Hilfe eines zweiten Arrays, das nicht alle Elemente des ersten Arrays besitzt. Die Heterogenität der beiden Arrays wird sowohl bei der Reihenfolge als auch im `anyOf`-Schema korrekt berücksichtigt. Die minimalen Häufigkeiten der `Strings` und der zweiten `Number`-Gruppe wurden auf null gesetzt, da im zweiten Array diese Elemente nicht auftreten. Die Problematik der Unzweideutigkeit soll im Algorithmus nicht näher behandelt werden. Diese besagt, dass nicht eindeutig geschlussfolgert werden kann, ob die `Number`-Elemente des zweiten Arrays der ersten oder zweiten Gruppe der `Number`-Elemente des ersten Arrays angehören. Der Algorithmus geht stets von einer festen Position aus und beginnt somit immer bei der ersten Gruppe. Sobald eine Übereinstimmung gefunden wurde, bricht er den Schleifendurchlauf für das Element ab. Im `anyOf`-Schema hat das `String`-Element nur noch eine Dokumenthäufigkeit von 50%, obwohl die Elementhäufigkeit mit 150% weiterhin über 100% liegt, da es im ersten von zwei Arrays dreimal vorkommt.

```
// zwei JSON Dokumente mit heterogenen Arrays
{
   "array" : [1, 2, 3, 4, "a", "b", "c", 1, 2]
}
{
   "array" : [1, 2]
}

// Darstellung der Arrays im JSON Schema
"array": {
   "type": "JsonArray",
   "description": {
      "occurences": "Occurence: 100% (2/2)",
      "array order": {
         "1 : Number": "Min occurence: 2; Max occurence: 4",
         "2 : String": "Min occurence: 0; Max occurence: 3",
         "3 : Number": "Min occurence: 0; Max occurence: 2"
      }
   },
   "items": {
      "anyOf": [
         {
            "type": "Number",
            "description": "Document Occurence: 100% (2/2);
             Member Occurence: 400% (8/2)"
         },
         {
            "type": "String",
            "description": "Document Occurence: 50% (1/2);
             Member Occurence: 150% (3/2)"
         }
      ]
   }
}
```

Abbildung 24: Darstellung von Arrays - Beispiel 2

4.5 Speicherkomponente

Aufgrund der Vereinfachungen bei der Ausgabe des JSON Schemas, kann dieses nicht zur Weiterverwendung im Algorithmus gespeichert werden. Es ist lediglich eine Konvertierung vom internen in das externe Schema möglich, die Umkehrung jedoch nicht. Es muss folglich eine Möglichkeit geschaffen werden, das interne Schema zu speichern, damit dieses zur Wiederverwendung geladen werden kann. Beim internen Schema handelt es sich um eine Menge von Knotenobjekten. Java bietet für serialisierbare Klassen eine Bibliothek, die Methoden zur Speicherung und zum Laden von Objekten serialisierbarer Klassen zur Verfügung stellt. Über einen `ObjectOutputStream` werden die einzelnen Knoten des Schemas in eine Datei geschrieben. Diese Datei kann anschließend über einen `ObjectInputStream` geladen werden. Dabei werden zuerst allgemeine `Java-Objects` erstellt, welche anschließend ihrer ursprünglichen Klasse zugewiesen werden müssen. Auf diese Weise kann das interne Schema permanent gespeichert und jederzeit auf jedem beliebigen Computer wieder geöffnet werden.

Zwei zusätzliche Funktionen sollen die Handhabung der Schemata weiter vereinfachen:

Zum einen wird eine Methode angeboten, die auch die ausgegebenen JSON Schemata in eine Textdatei speichern kann, die dann unabhängig vom Programm das extrahierte bzw. aktualisierte Schema enthält.

Zum anderen wird ein getrennter Bereich zum Merging zweier Schemata angeboten. Dort können entweder das extrahierte Schema mit einem importierten Schema oder zwei importierte Schemata zusammengeführt werden. Dabei werden die Schemata nicht wie bei der Aktualisierung als Updates behandelt, das heißt Elemente werden nicht ersetzt, sondern nur hinzugefügt. Sollten dieselben Dokumente in beiden Schemata vorhanden sein, werden keine Duplikate erzeugt.

4.6 Ableitung der Metriken

Die Metriken sollen auf eine andere Art einen Überblick über die repräsentierten Daten geben. Die Ableitung kann mit Hilfe eines vereinfachten internen Schemas geschehen, da die Dokument-IDs für diesen Vorgang nicht benötigt werden. Als Resultat entsteht ein vereinfachter Extraktionsalgorithmus, der schneller ist und mit weniger Speicherbedarf auskommt als die vollständige Extraktion, jedoch Abstriche in der Korrektheit und Verwendbarkeit des Schemas machen muss. Da nicht zwischen Vorkommen des Elements je Dokument-ID unterschieden wird, kann im Nachhinein nicht mit Sicherheit gesagt werden, welchem Dokument ein Element zuzuordnen ist. So kann es zum Beispiel in einer Kollektion 500 Dokumente geben, wobei ein bestimmtes Element bis auf wenige Ausnahmen doppelt vorkommt. In einigen wenigen Dokumenten kommt das Element jedoch nicht vor. Daraus wird im vereinfachten Schema dann trotzdem ein Pflichtelement, da die Gesamtanzahl der Elemente größer ist als die Anzahl der Dokumente. Ein weiterer Nachteil des vereinfachten Schemas ist, dass dieses nicht für die Aktualisierung verwendet werden kann, da diese grundlegend auf die Dokument-IDs angewiesen ist. Es ist jedoch auch nicht der Anspruch des vereinfachten Schemas, für eine korrekte Schema-Aktualisierung im herkömmlichen Sinn zu garantieren. Vielmehr sollen schnell wichtige Metriken bereitgestellt werden, die dann in einem zweiten, manuellen Schritt dazu verwendet werden, gezielt nach Ausreißer-Dokumenten zu suchen, Daten zu migrieren oder zu transformieren.
Umgesetzt wurde dieses Verfahren durch die Definition einer `simpleNode`-Klasse. Diese ähnelt stark der `node`-Klasse, besitzt jedoch keine Speicherstruktur für die Dokument-IDs, sondern lediglich einen Zähler für die Elementanzahl. Die Extraktion und Ausgabe des vereinfachten Schemas geschieht über abgewandelte Methoden der vollständigen Extraktion. Diese wurden in die Extraktions- bzw. Visualisierungskomponente integriert. Die Ableitung der Metriken geschieht im Visualisierungsschritt des vereinfachten Schemas.

4.7 Graphische Benutzeroberfläche

Beim Erstellen der Benutzeroberfläche wurde hauptsächlich die *Java Swing*-Bibliothek verwendet. Diese ermöglicht eine einfache Kombination aus Interface-Elementen. Neben einem Tab für die Extraktion wurde ein Tab für die Aktualisierung, ein Tab für das Schema-Merging, ein Tab für die vereinfachte Extraktion und ein Tab für die Konfiguration erstellt. Dabei bietet jeder Tab die benötigten Schaltflächen, Knöpfe und Ausgabeelemente, die für eine einfache und intuitive Bedienung der jeweiligen Funktion notwendig sind. Entsprechende Operationen des Algorithmus wurden mit den Knöpfen verknüpft. In der Konfiguration können neben der Datenbankverbindung die Ausgabe angepasst und ein permanentes Speicherverzeichnis für die Schemata gewählt werden. Im Anhang befinden sich neben den bereits gezeigten Ausschnitten (Abbildung 14 und Abbildung 18) weitere Abbildungen der einzelnen GUI-Elemente. (Abbildung 46, Abbildung 47, Abbildung 48)

5 Test und Bewertung

In diesem Kapitel sollen die implementierten Verfahren ausführlich getestet und daraufhin bewertet werden. Neben der Korrektheit der Verfahren ist die benötigte Zeit das wichtigste Kriterium. Die Laufzeit der Verfahren soll in Abhängigkeit von der Anzahl der Elemente, die sich aus der Anzahl der Elemente je Dokument und der Anzahl der Dokumente der Kollektion ergibt, ermittelt werden. Zu untersuchen ist ebenfalls, welchen Einfluss die unterschiedlichen Datenstrukturen auf die Laufzeit haben.

Deshalb wurden zwei sehr unterschiedliche Testdatensätze zum Testen der Verfahren verwendet. Eine Dokumentenkollektion mit rund 2,6 Millionen Dokumenten, folgend als `Testdatensatz_2` bezeichnet, wird als Maßstab für alle durchgeführten Tests verwendet, um vergleichbare Ergebnisse zu erzielen. Als Ergänzung wird eine weitere Kollektion – `Testdatensatz_1` – herangezogen, die wesentlich weniger Dokumente beinhaltet, jedoch mehr Elemente und Verschachtelungen je Dokument enthält, als der sehr homogene und flach strukturierte `Testdatensatz_2`. Tabelle 9 stellt relevante Kennzahlen der beiden Testdatensätze gegenüber. Die vollständigen Schemata der beiden Testdatensätze sind im Anhang D – Schemata abgebildet.

Kennzahl	Testdatensatz_1	Testdatensatz_2
Dokumentanzahl	25.359 Dokumente	2.595.243 Dokumente
Art der Elemente	Viele Arrays und Objekte	Wenige Objekte, keine Arrays
Struktur der Dokumente	Stark verschachtelt	Flache Hierarchie
Größe der Kollektion	14.692.880 Byte	290.667.216 Byte
Durchschnittliche Objektgröße	579 Byte	112 Byte

Tabelle 9: Gegenüberstellung der Kennzahlen der beiden Testdatensätze

5.1 Performancebetrachtung

In Hinblick auf die Effizienz, die nicht nur in der ISO 25010 ein Kriterium für die Qualität der Software darstellt, sondern auch für den Einsatz der Software in der Praxis unabdingbar ist, soll eine Performanceuntersuchung und -optimierung des Algorithmus durchgeführt werden. Insbesondere da die erste Implementation der Extraktionskomponente unerwartete Performanceprobleme mit sich zog, ist eine genauere Untersuchung der Ursachen und Lösungen für diese Probleme notwendig. Kritisch war in der ersten Implementation sowohl der Zeitbedarf als auch der Speicherverbrauch des Algorithmus. Die Extraktion wurde beim Durchlauf des `Testdatensatz_2` nach 15 Minuten durch einen `OutOfMemory`-Fehlerzustand abgebrochen. Ein besonderer Fokus der Performancebetrachtung liegt daher in der Optimierung dieser beiden Kriterien.

Betrachtung der Laufzeit

Die Laufzeit des Algorithmus lässt sich durch die Betrachtung der verstrichenen Zeit im Verhältnis zur Anzahl der Schleifendurchläufe bewerten, da nicht nur die Anzahl der Dokumente allein, sondern auch die Anzahl der Elemente je Dokument ausschlaggebend für die Laufzeit sind. Der verwendete `Testdatensatz_2` enthält rund 2,6 Millionen Dokumente mit durchschnittlich neun Elementen je Dokument. Der Algorithmus muss also rund 23,4 Millionen Elemente betrachten, d.h. rund 23,4 Millionen Mal die Extraktionsschleife durchlaufen, um die gesamte Kollektion zu extrahieren. Abbildung 25 stellt den Zeitverlauf der

problematischen ersten Implementation bei der Extraktion des `Testdatensatz_2` dar. Zu Vergleichszwecken wurden zusätzlich die Messungen der einfachen und vollständigen Extraktion des aktuellen Algorithmus eingefügt.

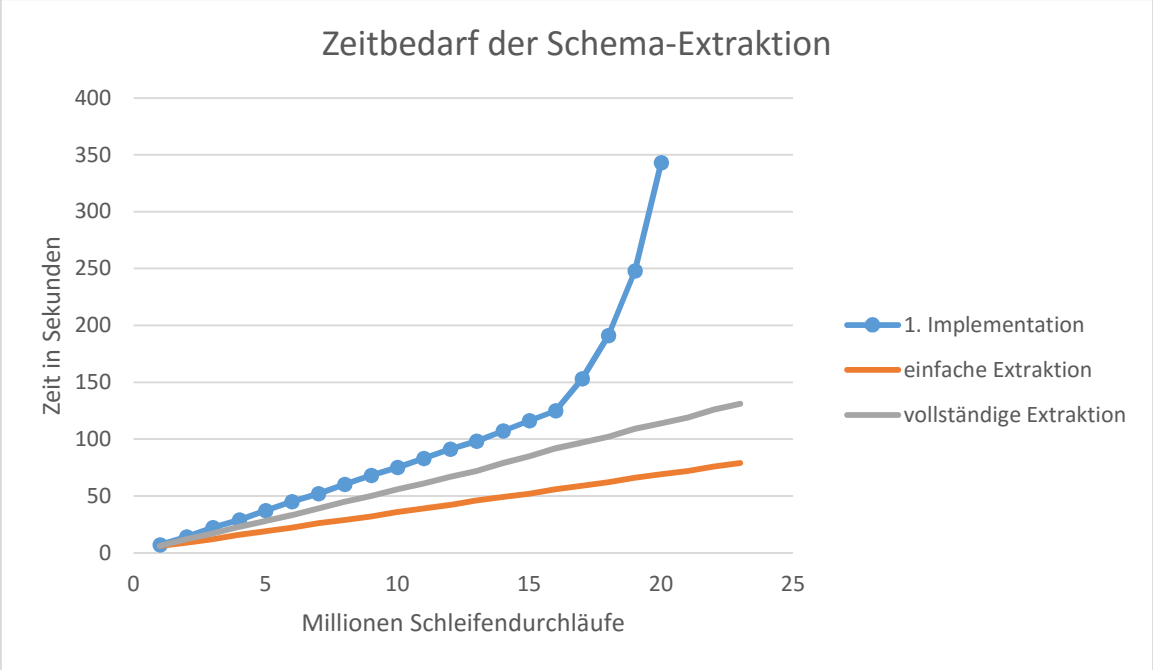

Abbildung 25: Vergleich des Zeitverlaufs der ersten Implementation mit den aktuellen Verfahren

Der Versuch, die Laufzeit des Algorithmus durch Optimierungen an der Anzahl durchzuführender Operationen innerhalb der Schleife zu verkürzen, brachte keine signifikanten Leistungsverbesserungen. Die Anzahl Operationen je Element ist wenig variabel, da diese durch die benötigten Informationen festgelegt werden. Die hohe Anzahl an Operationsschritten entsteht aus der Größe der Datenmenge. Da jedes Element eines jeden Dokuments einmal betrachtet werden muss, kann auch an dieser Stelle wenig optimiert werden.
Auffällig ist bei einer Betrachtung der Laufzeit die abnehmende Performance der ersten Implementation. Die ersten Millionen Schleifendurchläufe benötigen jeweils ca. 7-9 Sekunden. Ab ca. 16 Millionen Durchläufen nimmt die benötigte Zeit je Millionen Durchläufe stark zu. Dieser exponentielle Anstieg führte in Verbindung mit dem anschließenden `OutOfMemory`-Fehler zu der Vermutung, dass eine Überlastung der Hardware Grund für den Leistungseinbruch sein könnte. Auf die Untersuchung der Hardwareauslastung soll im nächsten Abschnitt genauer eingegangen werden.

Betrachtung der Hardware

Nachdem die erste Variante des Extraktionsalgorithmus immer wieder in einen `OutOfMemory`-Fehlerzustand geriet, wurde die Hardwareauslastung während der Testdurchläufe betrachtet. Die Tests wurden an einem Heimcomputer mit einem Intel i7 der ersten Generation und 8 GB Arbeitsspeicher durchgeführt. Abbildung 26 zeigt einen Überblick über die Hardwareauslastung während der Tests. Die CPU-Auslastung lag zu Beginn bei 10-15%. Nachdem überprüft worden war, dass nicht nur ein Kern genutzt wurde, konnte die CPU-Auslastung als erste potenzielle Problemursache ausgeschlossen werden. Der benötigte Arbeitsspeicher steigt abgesehen von den Einbrüchen durch den `Java Garbage Collector` stetig. Eine drastische Veränderung der Hardwareauslastung tritt bei rund 16 Millionen Schleifendurchläufen auf. An dieser Stelle erreicht der Arbeitsspeicher das 2 GB Limit, welches

der Java Laufzeitumgebung standardmäßig als Maximum zugewiesen wird. Gleichzeitig springt die CPU Auslastung auf 100% und die Zeit pro Millionen Schleifendurchläufe steigt von je 7-9 Sekunden für die ersten 15 Millionen auf knapp 100 Sekunden für die letzten Millionen Durchläufe vor dem Fehlerzustand an. (weitere Informationen dazu im Anhang A – zusätzliche Abbildungen: Abbildung 40, Abbildung 41, Abbildung 42 und Abbildung 43)

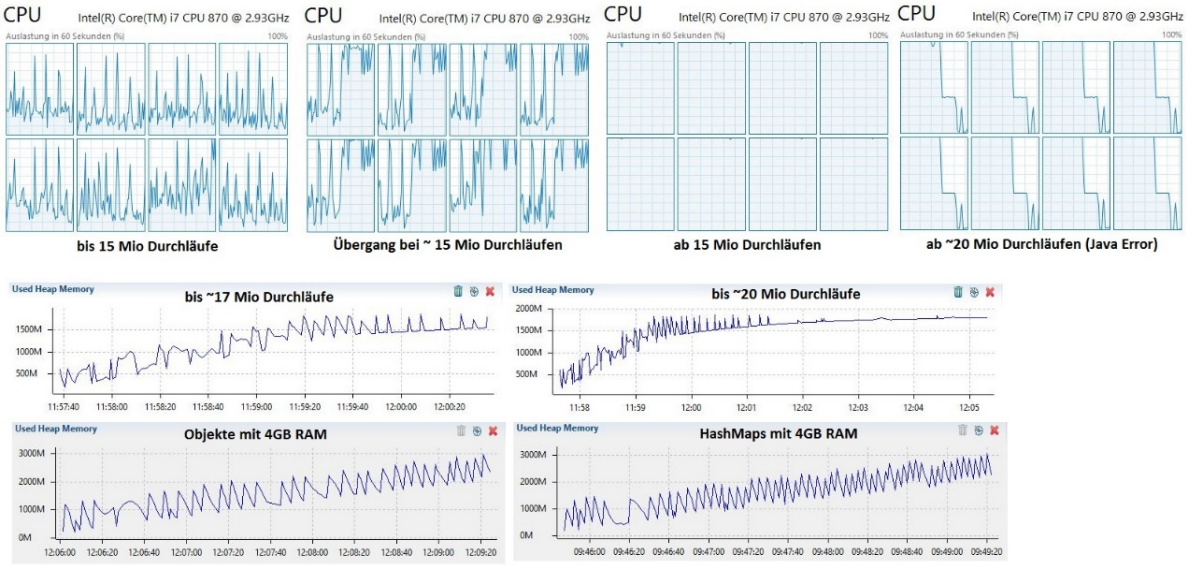

Abbildung 26: Überblick über die Hardwareauslastung während der Tests

Eine Erhöhung des zur Verfügung stehenden Arbeitsspeichers auf 4 GB lieferte dann die ebenfalls in Abbildung 26 zu sehenden Auslastungsdiagramme des Arbeitsspeichers. Die CPU Auslastung blieb bei allen Testdurchläufen mit 4 GB Arbeitsspeicher für die gesamte Laufzeit bei 10-20%.

Betrachtung der Datenstrukturen

Zur Optimierung der Performance müssen auch die verwendeten Speicherstrukturen, insbesondere die Datenstrukturen zur Speicherung der Elemente, Datentypen und Dokument-IDs, betrachtet werden.

Intuitiv wurde das interne Schema als Baumstruktur in einer Knoten- und Kantenmenge gespeichert. Knoten und Kanten enthielten zusammen alle Informationen, die eine Rekonstruktion der Dokumentstruktur ermöglichten. Jedoch wurden dadurch Informationen, namentlich die Dokument-IDs, redundant gespeichert, da sowohl die Knoten als auch die Kanten eindeutig zu ihren jeweiligen Dokumenten zugeordnet werden mussten. Dies führte insbesondere im Fall der Dokument-IDs zu einem enormen Speicherverbrauch, weshalb durch Umstrukturierung der Knotenobjekte die Speicherung der Kantenobjekte überflüssig gemacht wurde. Folgend werden in den Knoten alle benötigten Informationen zur eindeutigen Rekonstruktion gespeichert. Diese Knoten sind als Java-Klasse implementiert und werden entsprechend als Java-Objekte initialisiert. Deren Attribute, darunter die Liste der Datentypen und die Liste der Dokument-IDs wurden anfänglich in einfachen `ArrayLists` [27] gespeichert. Der Versuch, die Java-Objekte gegen `HashMaps` [28] auszutauschen, brachte keine Leistungsverbesserungen ein, da im Testdatensatz lediglich neun verschiedene Knoten gespeichert werden müssen. Der eigentliche Speicherbedarf resultiert aus der Speicherung der Dokument-IDs und der größte Teil des Zeitbedarfs entsteht beim Durchsuchen dieser Dokument-IDs. Der Testdatensatz sollte dabei den Großteil der Dokumentkollektionen

repräsentieren, bei denen die Anzahl unterschiedlicher Elemente wesentlich geringer als die Anzahl der Dokumente ist.
Folglich wurde nach einer effizienten Datenstruktur zur Speicherung großer Datenmengen gesucht, die ebenso effizient nach einem Wert durchsucht werden kann. Die für die Speicherung großer Listen besser geeigneten `TreeSets` [29] führten zu einer merklichen Verbesserung der Performance. Der Speicherverbrauch konnte durch die Verwendung der `TreeSets` nicht reduziert werden, allerdings wurde der Zeitbedarf der Extraktion reduziert.

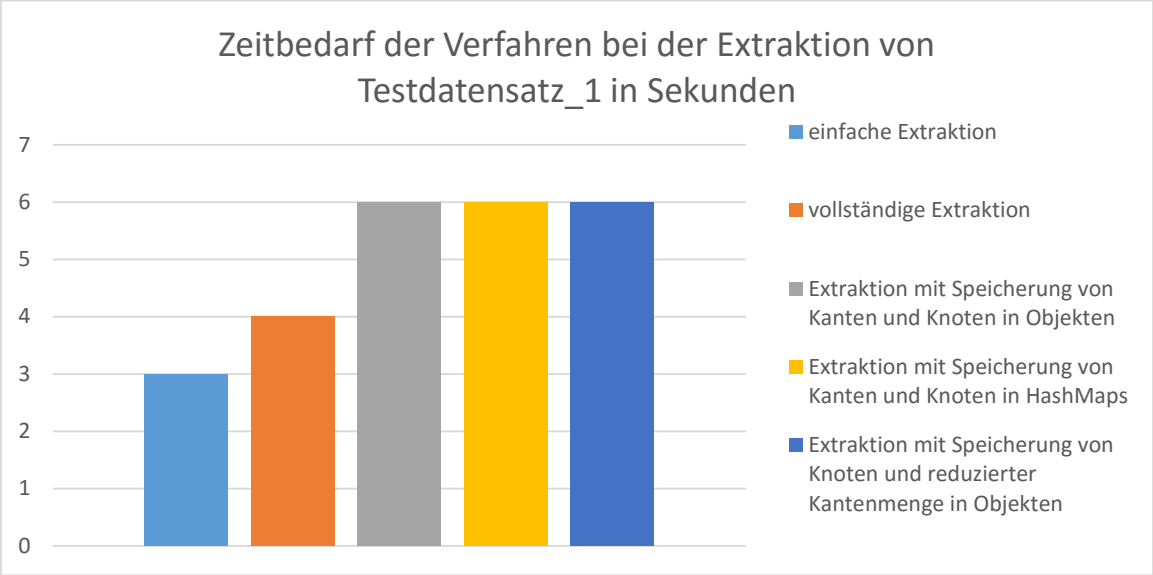

Abbildung 27: Vergleich der Extraktionsverfahren anhand von Testdatensatz_1

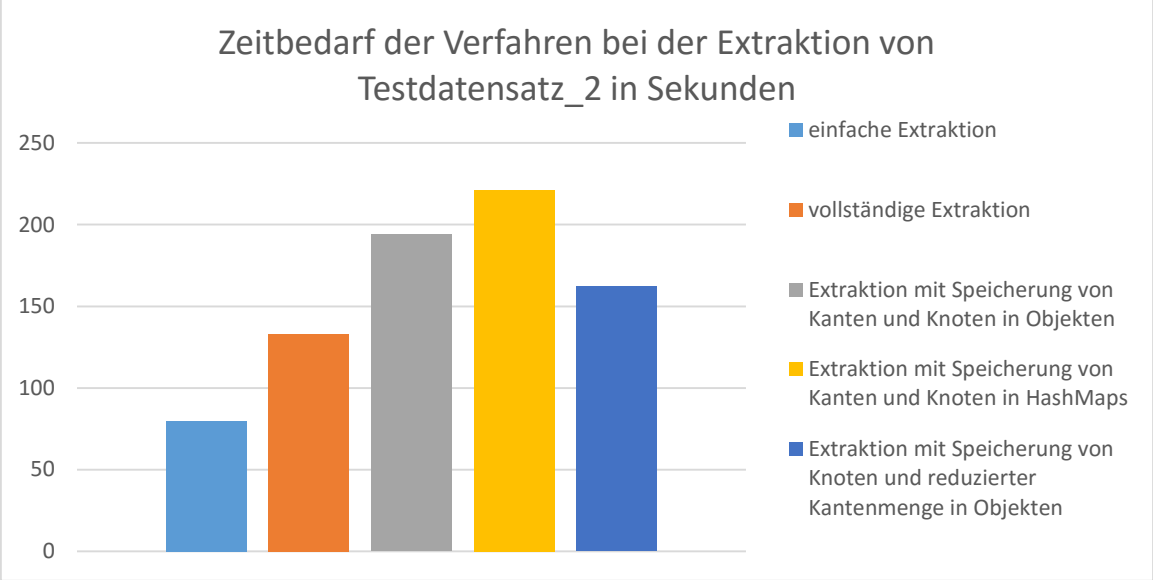

Abbildung 28: Vergleich der Extraktionsverfahren anhand von Testdatensatz_2

Zur Reduktion des Speicherbedarfs wurden die gespeicherten Informationen betrachtet. In einem ersten Schritt wurde die Anzahl zu speichernder Kanten reduziert, indem die Kanten vom Wurzelelement zu den Elementen der ersten Ebene im Dokument entfernt wurden. Diese waren nicht unbedingt für eine korrekte Rekonstruktion erforderlich und konnten problemlos entfernt werden. Dies begünstigte den Speicherverbrauch bei der Schema-Extraktion von `Testdatensatz_2` sehr stark, da durch die flache Hierarchie fast alle Kanten gestrichen werden konnten. Bei stark verschachtelten Datensätzen ist dieser Effekt jedoch wesentlich

geringer. (Vergleich Abbildung 27 und Abbildung 28) Deshalb wurden im zweiten Schritt alle Kanten gestrichen und die benötigten Informationen, die bisher in den Kantenobjekten gespeichert waren, in die Knotenobjekte übernommen. Da die Dokument-IDs fast den kompletten Speicherverbrauch ausmachen, konnte dieser dadurch nahezu halbiert werden.
Die signifikanten Unterschiede des Speicherverbrauchs der einzelnen Implementationen sind in Abbildung 29, Abbildung 30 und Abbildung 31 sowie weiteren Abbildungen im Anhang A – zusätzliche Abbildungen dargestellt. Alle Messungen wurden anhand der Schema-Extraktion von `Testdatensatz_2` durchgeführt.

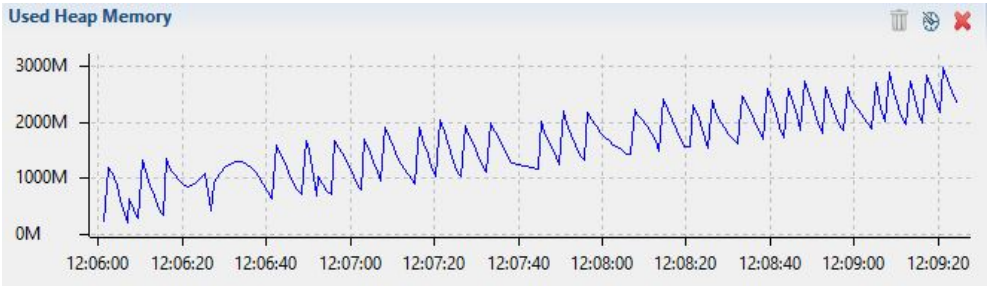

Abbildung 29: Heap Memory Auslastung bei der Extraktion mit Knoten- und Kantenobjekten bei 4GB RAM

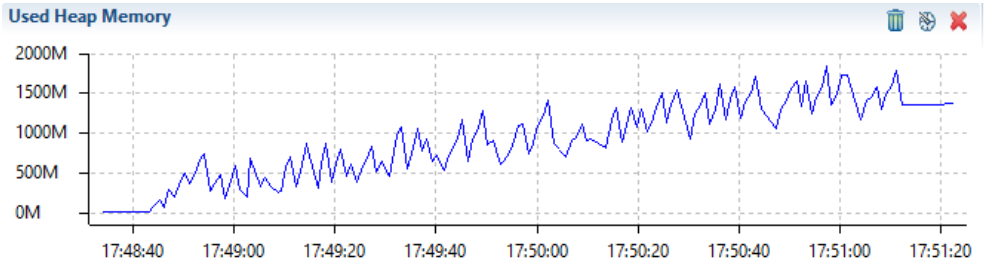

Abbildung 30: Heap Memory Auslastung bei der vollständigen Extraktion mit der endgültigen Implementation

Abbildung 31: Heap Memory Auslastung bei der vereinfachten Extraktion mit der endgültigen Implementation

Im späteren Entwicklungsverlauf wurden die zur Speicherung der Dokument-IDs und Datentypen verwendeten `TreeSets` durch `HashMaps` ersetzt, wobei die Dokument-ID bzw. der Datentyp im Schlüssel und ein Zähler für den jeweiligen Schlüssel im Wert gespeichert wurde. Dies ermöglicht es, die Häufigkeit des Auftretens einzelner Elemente je Dokument und einzelner Datentypen je Element zu speichern, um zum Beispiel Häufigkeiten der Elemente in Arrays zu ermitteln.
Abbildung 32 stellt den Zeitbedarf ausgewählter Verfahren gegenüber. Die Messungen wurden über einen Zähler im Programm durchgeführt, der die verstrichene Zeit jeweils beim Erreichen einer glatten Millionenstelle ausgibt. Tabelle 13 im Anhang stellt die genauen Messergebnisse dar. Bei allen dargestellten Verfahren wurden die Dokument-IDs und Datentypen entweder in `TreeSets` (bei expliziter Speicherung der Kanten) oder in `HashMaps` (zur Speicherung der Kanteninformationen im Knotenobjekt) gespeichert.

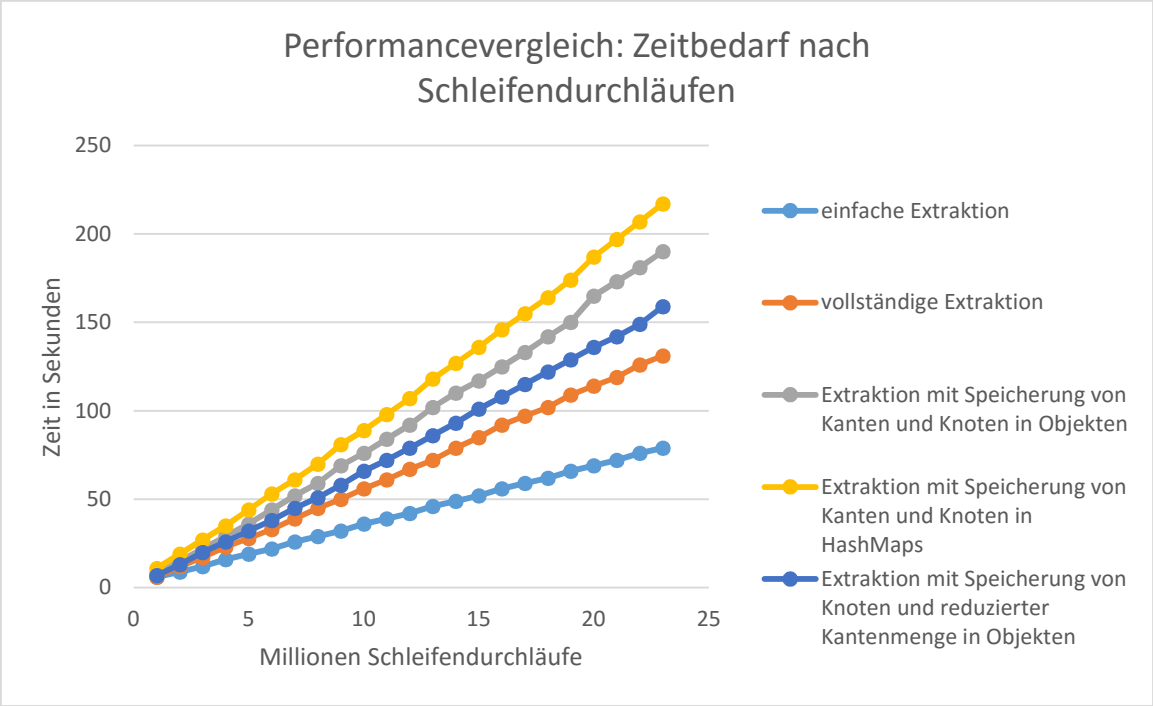

Abbildung 32: Zeitbedarf getesteter Verfahren am Beispiel der Extraktion von Testdatensatz_2

Fazit der Performancebetrachtung

Während der Suche nach den Ursachen des Performanceproblems sind mehrere Varianten des Algorithmus entstanden. Eine Variante implementiert eine eigene Knotenklasse und erzeugt für jeden Knoten ein neues Objekt. Eine andere Variante setzt auf performante Datenstrukturen und verwendet `HashMaps` zur Speicherung der Knoten. Zum Speichern der Dokument-IDs wurden `ArrayLists`, `TreeSets` und `HashMaps` getestet, wobei die Verwendung von `TreeSets` und `HashMaps` einen erheblichen Leistungsschub zur Folge hatte. Dennoch kann das Problem des Speicherverbrauchs durch eine optimale Datenstruktur nicht vollständig gelöst werden, da ab einer gewissen Größe der Listen die Bits des Arbeitsspeichers einfach nicht mehr ausreichen. Eine Erhöhung des der *Java Virtual Machine* zur Verfügung stehenden Arbeitsspeichers verschiebt das Problem dabei lediglich bis die Grenze des Arbeitsspeichers des Computers erreicht ist. So waren für den `Testdatensatz_2` die standardmäßig zugewiesenen 2 GB Arbeitsspeicher nicht ausreichend und dieser musste auf 4 GB erhöht werden, um die Extraktion ohne Speicherüberlauf durchführen zu können. Da der Arbeitsspeicher eines jeden Systems begrenzt ist, die Größe der Dokumentkollektionen jedoch nicht, muss für dieses Problem eine Lösung gefunden werden. Zum einen ist eine Auslagerung des Arbeitsspeichers denkbar, falls dieser an seine Grenzen stößt. Zum anderen kann auch der Extraktionsprozess untergliedert werden, so dass er parallel auf verteilten Systemen durchgeführt werden kann. Die notwendigen Mittel zur Durchführung der zweiten Lösung wurden in das Programm integriert. So können Dokumentkollektionen komplett oder durch Angabe einer Query teilweise extrahiert, das Schema anschließend gespeichert und dieses mit Hilfe des Merging-Tabs in der GUI mit anderen Schemata zusammengefügt werden.

5.2 Korrektheitsbetrachtungen

Für die Korrektheitsbetrachtungen soll das interne Schema als Graph betrachtet werden. Zu Beginn ist das Schema leer. Jedes neue Dokument wird nun mit der Dokumentkollektion als Wurzelelement, die der Wurzel des Graphen des internen Schemas entspricht, diesem Graphen hinzugefügt. Durch die abstrakte Wurzel der Dokumentkollektion werden die möglicherweise mehreren Wurzelelemente eines Dokuments zu Knoten der ersten Ebene, die der Wurzel unterstellt sind. Eine Rekursion kann also bereits in der einzigen Wurzel gestartet werden. Diese garantiert beim jeweiligen Selbstaufruf der Methode für jedes Kindelement, dass jedes Element eines Dokuments durchlaufen und dem Schema hinzugefügt wird. Die Methode wird für jedes Dokument der angegebenen Kollektion aufgerufen. Es ist also garantiert, dass jedes Element einmal im Schema aufgenommen wird. Die Rekursion garantiert zusätzlich, dass, auch wenn bereits zahlreiche andere Dokumente extrahiert wurden, nur Elemente eines Dokuments in das Schema eingefügt werden können, falls dieses Dokument auch mindestens ein Element des Elternknotens im Schema besitzt. Diese Eigenschaft gilt für jedes Element eines jeden Dokuments vom Blatt bis zur Wurzel. Abbildung 33 veranschaulicht dies. Die hellblauen Knoten sind die bereits extrahierten Knoten des aktuellen Dokuments. Die dunkelblauen und dunkelgrünen Knoten sind Knoten des internen Schemas, die aus anderen Dokumenten stammen, wobei die dunkelgrünen Knoten über die Rekursion vom aktuellen Dokument aus erreichbar sind. Nur solche Knoten können im nächsten Rekursionsschritt gefunden werden, da das Dokument selbst hierarchisch aufgebaut ist und im validen JSON Format sein muss. Die hellgrünen Knoten stellen Knoten dar, die noch nicht im internen Schema vorhanden sind, aber von den aus dem aktuellen Dokument bereits hinzugefügten Knoten erreichbar wären. Sie repräsentieren durch den nächsten Rekursionsschritt erreichbare Knoten, die in keinem bisher extrahierten Dokument vorhanden waren.

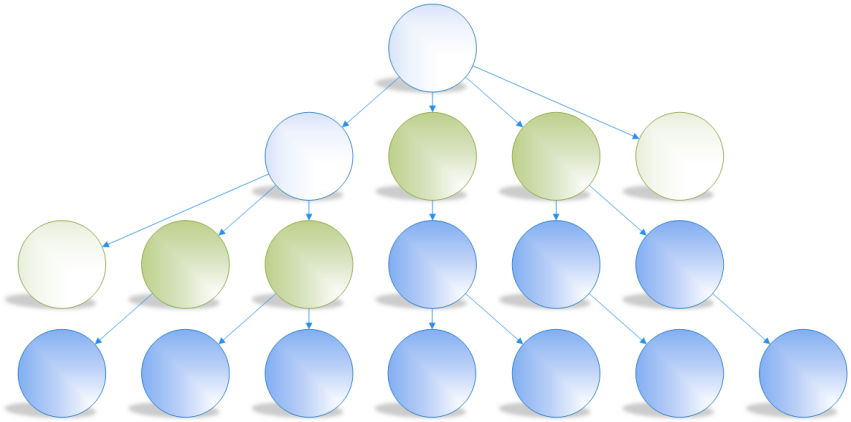

Abbildung 33: Korrektheitsbetrachtung der Extraktion

Im internen Schema befinden sich nach der Extraktion also stets vollständige und zusammenhängende Dokumente. Im nächsten Schritt muss nun gezeigt werden, dass auch die Aktualisierung diese Eigenschaften wahrt.

Für die Aktualisierung durch die geänderten Dokumente ist dieser Schritt trivial. Einem geänderten Dokument wird die Dokument-ID entnommen und diese aus jedem Knoten im internen Schema entfernt. Anschließend leere Knoten werden entsprechend aus dem Schema entfernt. Danach wird das komplette, geänderte Dokument wie ein zusätzliches Dokument der Extraktion behandelt. Es wird also vollständig neu hinzugefügt.

Für die Aktualisierung mit dem Update-Log könnte vorausgesetzt werden, dass nur korrekte, atomare, ausführbare Updates im Update-Log stehen. Der Algorithmus geht jedoch einen Schritt weiter und sichert bestimmte Fälle selbst ab, insbesondere an den Stellen, die durch optionale Angaben im Update-Log nicht abgedeckt sein müssen.

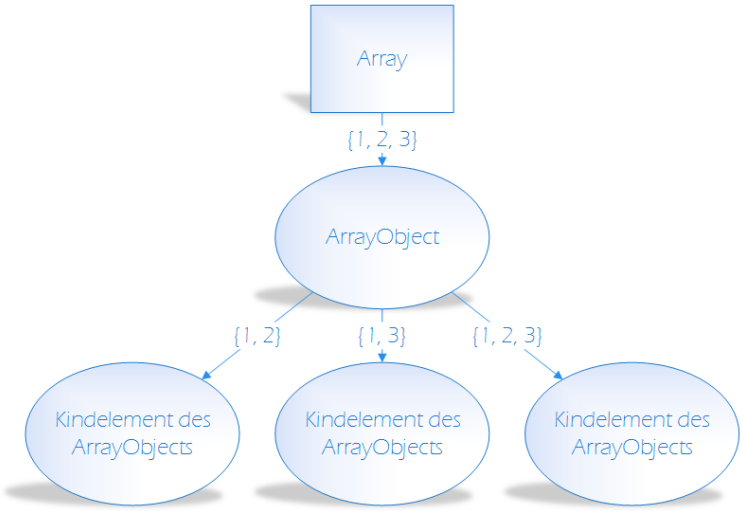

Abbildung 34: Struktur eines Dokuments mit drei unterschiedlichen Objects in einem Array

Bei einem `Insert` wird das eingefügte Element im internen Schema gesucht. Gibt es ein solches Element bereits, wird das Vorkommen der Dokument-ID des Updates hinzugefügt bzw. um Eins erhöht. Zusätzlich wird das Wurzelelement auf Vorhandensein der Dokument-ID überprüft und diese auf mindestens Eins gesetzt, so dass im Falle eines neu hinzugefügten Dokuments dieses im Schema korrekt repräsentiert wird. Bei einem `Delete` wird das Vorkommen der Dokument-ID im gelöschten Element um Eins verringert und der Knoten aus dem Schema gelöscht, falls er keine weiteren Dokument-IDs enthält. Anschließend werden vom Algorithmus automatisch alle Kind- und Kindeskindelemente des gelöschten Elements gesucht, auf Vorhandensein der Dokument-ID überprüft und diese ggf. um Eins verringert. Ausnahme hierbei stellen Arrays dar. Falls ein Array gelöscht wurde, werden anschließend alle Kindelemente mit derselben Dokument-ID gelöscht. Dieser Schritt garantiert, dass ein `Delete` nicht die in Abbildung 33 dargestellte Verkettung der Elemente von der Wurzel zum Blatt durchtrennt. In einem letzten Schritt wird sichergestellt, dass auch die Dokumentanzahl im internen Schema korrekt ist, indem alle Knoten nach der Dokument-ID des `Deletes` durchsucht werden. Ist die Dokument-ID nur noch im Wurzelelement vorhanden, wird sie entfernt und somit die Dokumentanzahl korrekt reduziert. `Updates` verändern den Wert eines Elements und können somit nur Strukturveränderungen über eine Änderung des Datentyps auslösen. Eine Verarbeitung dieser ist jedoch nicht ohne weitere Informationen korrekt und eindeutig möglich. Wird ein JSON Primitive in ein Array oder Object umgewandelt, müssen die Kindelemente explizit angegeben werden. Wird ein Array in ein Object umgewandelt, müssen Namen für die einzelnen Elemente deklariert werden. Bei der Umwandlung eines Objects in ein Array fehlen beispielsweise Informationen über die Reihenfolge der Elemente. Einzig eine Umwandlung eines Objects oder Arrays in einen JSON Primitive könnte unter der Voraussetzung, dass der Datentyp vor der Änderung bekannt ist, verarbeitet werden, wobei selbst unter diesen Voraussetzungen der Algorithmus bei Objekten in Arrays versagen würde.

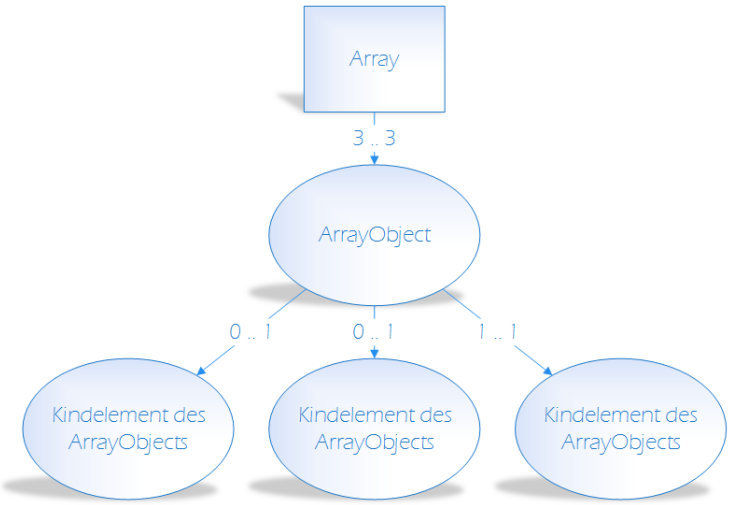

Abbildung 35: Darstellung des Dokuments mit drei unterschiedlichen Objects in einem Array im internen Schema

Wird zum Beispiel eines von drei Objekten in einem Array (dargestellt in Abbildung 34) in einen JSON Primitive umgewandelt und haben diese Objekte im Dokument unterschiedliche Kindelemente, kann aufgrund der vorher durchgeführten Generalisierung – der Zusammenfassung der Objekte zu einem `ArrayObject` (dargestellt in Abbildung 35) – nicht bestimmt werden, welches der Kindelemente zum gelöschten Objekt gehörte. An dieser Stelle muss das Update-Log also korrekte und vollständige Informationen liefern, damit die Korrektheit des internen Schemas garantiert bleibt. Eine automatisierte Bearbeitung der Kindelemente bei Updates wurde daher nicht implementiert, insbesondere da bei vollständigen Update-Logs die Kindelemente doppelt gelöscht würden. Ein ähnliches Problem kann bei den `Deletes` auftreten, falls eines von mehreren Objekten in einem Array gelöscht wird und sowohl das gelöschte als auch mindestens ein nicht gelöschtes Objekt ein gleichnamiges Array als Kind- bzw. Kindeskindelement besitzen. (siehe Abbildung 36)
Aufgrund der aufgezeigten Probleme mit der automatischen Wiederherstellung der Korrektheit nach dem Löschen eines Arrays oder Objects wird dieses Feature nur optional angeboten. In vielen Fällen – falls zum Beispiel nur ein Datentyp im gelöschten Element vorkommt oder nur einfache Objects, Arrays oder Primitives gelöscht werden – können richtige Schlüsse gezogen werden und somit fehlende Angaben eines Update-Logs ausgleichen. Aufgrund der Limitierung durch die Speicherung des internen Schemas allerdings nicht beim Löschen von Objekten in Arrays und Arrays in Arrays.

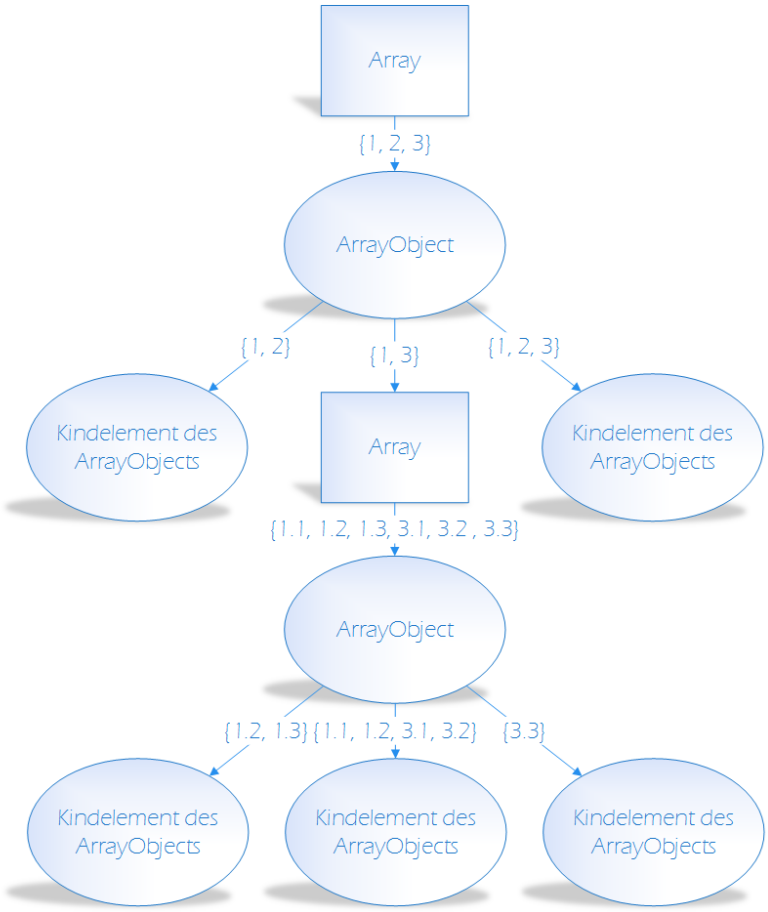

Abbildung 36: Struktur eines Dokuments mit einem Array in einem Array

Im Fall von Objekten in Arrays wird optional für jedes Kindelement des gelöschten Objekts die Dokument-ID um Eins verringert. Sobald der Algorithmus auf ein Array stößt, werden ab diesem Element alle Vorkommen der Dokument-ID gelöscht, um Mehrfachelemente eines Arrays korrekt zu entfernen. Dadurch, dass es durch das erste Array jedoch zwei gleiche Objekte mit derselben Dokument-ID geben kann, sind eindeutige Schlüsse nicht mehr möglich. Wird der beschriebene Löschprozess durchgeführt, wird im Normalfall die Korrektheit des Schemas gewahrt, indem nicht mehr erreichbare Elemente entfernt werden und somit die Validität der Dokumentstruktur sichergestellt wird. Unter Umständen werden jedoch Elemente entfernt, die noch im Dokument enthalten sind. Wird der Löschprozess nicht durchgeführt, bleiben Elemente ggf. als Pflichtelemente deklariert, die nicht mehr in jedem Dokument enthalten sind. Unter Umständen bleiben Elemente im Schema, die in den Dokumenten nicht mehr vorkommen. Sowohl das Löschen als auch der Verbleib der Elemente im Schema können also zu einem inkorrekten Schema führen. Grund ist die festgelegte Zusammenfassung der Arrayobjekte, die die Schemata übersichtlicher und prägnanter machen soll. Abbildung 37 und Abbildung 38 stellen den Unterschied zwischen Generalisierung und vollständigem Schema für nur ein Dokument mit nur drei Elementen dar. Eine interne Nummerierung der Arrayobjekte könnte diese trotzdem eindeutig identifizieren. Abbildung 36 veranschaulicht eine solche Nummerierung. Ein Dokument beinhaltet in einem ersten Array drei Objekte, von denen Objekt Eins und Drei ein weiteres Array mit jeweils drei weiteren Objekten besitzen. Jedoch würde auch bei generalisierter Ausgabe das interne Schema wie in Abbildung 38 stark vergrößert werden. Zudem reicht eine interne Nummerierung noch nicht aus. Auch im Update-Log müsste eben diese Nummerierung eingehalten und angegeben werden.

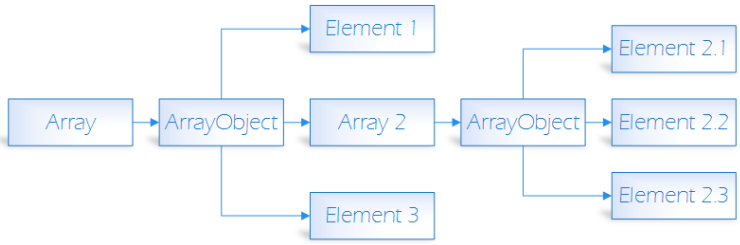

Abbildung 37: Generalisierung der Objekte zu ArrayObjects

Bei der Rekonstruktion wird die Eindeutigkeit der Kombination aus Pfad und Knotenname ausgenutzt. Durch Hinzunahme der ebenfalls eindeutigen Dokument-ID kann jedes Element des internen Schemas eindeutig einem Dokument zugeordnet werden. Ausgenommen sind dabei die explizit nach Datentyp generalisierten Elemente eines Arrays. Insbesondere die Objekte können nur einem allgemeinen `ArrayObject` zugeordnet werden. Aus dem internen Schema kann also zu jedem Zeitpunkt die Struktur aller Dokumente rekonstruiert werden. Diese Beschaffenheit ermöglicht es auch zu jedem Zeitpunkt aus dem internen Schema ein JSON Schema abzuleiten.

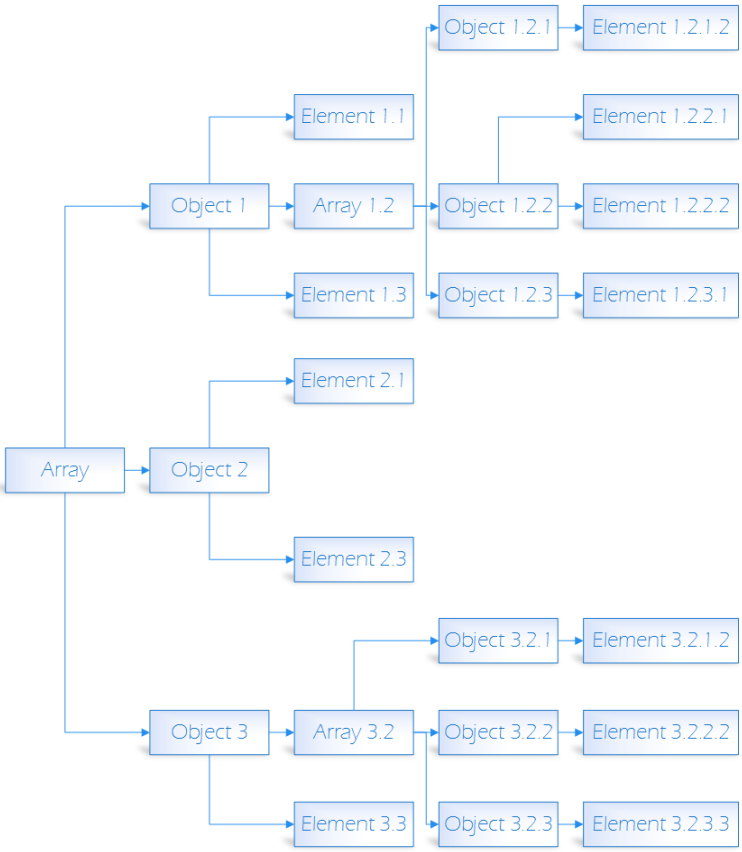

Abbildung 38: vollständige Struktur ohne Generalisierung

5.3 Bewertung der Verfahren

In dieser Arbeit wurden verschiedene Verfahren zur Schema-Extraktion und -Aktualisierung vorgestellt und entwickelt, die nun in diesem Abschnitt bewertet werden sollen.

Eine vollständige Schema-Extraktion, die durch das Speichern von Dokument-IDs, Elementpfaden und Elementnamen eine eindeutige Rekonstruktion der extrahierten Daten (mit Ausnahme der generalisierten `ArrayObjects`) erlaubt, dafür jedoch ein hohen Speicher- und Zeitbedarf hat, bildet den Ausgangspunkt für korrekte Schema-Aktualisierungsverfahren.

Zu den Aktualisierungsverfahren gehören zum einen die Aktualisierung mit Hilfe von geänderten Dokumenten und zum anderen die Aktualisierung mit Hilfe eines Update-Logs. Beide Verfahren besitzen Grenzen, können jedoch beliebig miteinander verknüpft werden. Ausschlaggebend für die Qualität und Korrektheit der Aktualisierung ist der Informationsumfang, der zu jedem Aktualisierungsschritt geliefert wird. Nur unter Vorhandensein von Dokument-IDs und genauen Informationen über die Änderung kann das Schema fehlerfrei aktualisiert werden. Weitere Informationen sind wünschenswert, jedoch nicht für zwingend erforderlich. Entstehende Generalisierungen können durch sporadische Neu-Extraktionen wieder entfernt werden.
Bei der Aktualisierung unter Vorlage der geänderten Dokumente liegen durch das Dokument selbst alle Informationen vor. Eine korrekte und vollständige Aktualisierung ist dadurch jederzeit möglich. Das Verfahren erfordert allerdings, dass sich gelöschte Dokumente unter Angabe ihrer Dokument-ID in der Menge der geänderten Dokumente befinden. Ansonsten können gelöschte Dokumente nicht erkannt werden. Durch die Festlegung, dass die Dokument-ID im dafür vorgesehenen `_id`-Feld stehen muss, können im Schema keine Dokumente, die nur eine Dokument-ID besitzen, vorkommen.
Die Aktualisierung unter Vorlage des Update-Logs stößt beim Löschen von Arrays in Arrays und Objekten in Arrays sowie bei Updates, die die Struktur verändern, an Grenzen. Es wurden Lösungsansätze für diese Probleme diskutiert. Jedoch sind diese nicht alleine durch eine interne Umstrukturierung zu beheben, sondern sind auch vom Informationsumfang der Updates abhängig.
Die Qualität und geringe Fehleranfälligkeit der Aktualisierung mit Hilfe der geänderten Dokumente macht dieses Verfahren zu einer sehr guten Möglichkeit der inkrementellen Schema-Aktualisierung. Allerdings kommt die geringe Fehleranfälligkeit auf Kosten der Performance. Da für jedes Update zuerst das komplette Dokument aus dem Schema gelöscht wird und dieses anschließend inklusive der Änderung wieder hinzugefügt wird, benötigt dieses Verfahren mehr Zeit als die Aktualisierung durch ein Update-Log, bei der explizit nur das geänderte Element bearbeitet wird. Abbildung 39 stellt den Zeitbedarf der beiden inkrementellen Aktualisierungsverfahren und einer Schema-Neuextraktion in Abhängigkeit von der Anzahl der Updates gegenüber. Die Gegenüberstellung ist allerdings vorsichtig zu betrachten, da es sich bei den Aktualisierungen zum einen um generierte Updates handelt und diese nicht den Aktualisierungen in der Praxis entsprechen müssen. Zum anderen können bei der Aktualisierung durch die geänderten Dokumente auch mehrere Update-Schritte eines Update-Logs abgedeckt werden, falls die Änderungen am selben Dokument vorgenommen wurden. So ist die Gleichsetzung eines Schritts des Update-Logs mit einem geänderten Dokument nicht immer zutreffend. Auch spielen bei großen Schemata die Position des Elements in der Knotenmenge und die Position der Dokument-ID in der Dokument-ID-HashMap eine Rolle und können die Geschwindigkeit stark beeinflussen.

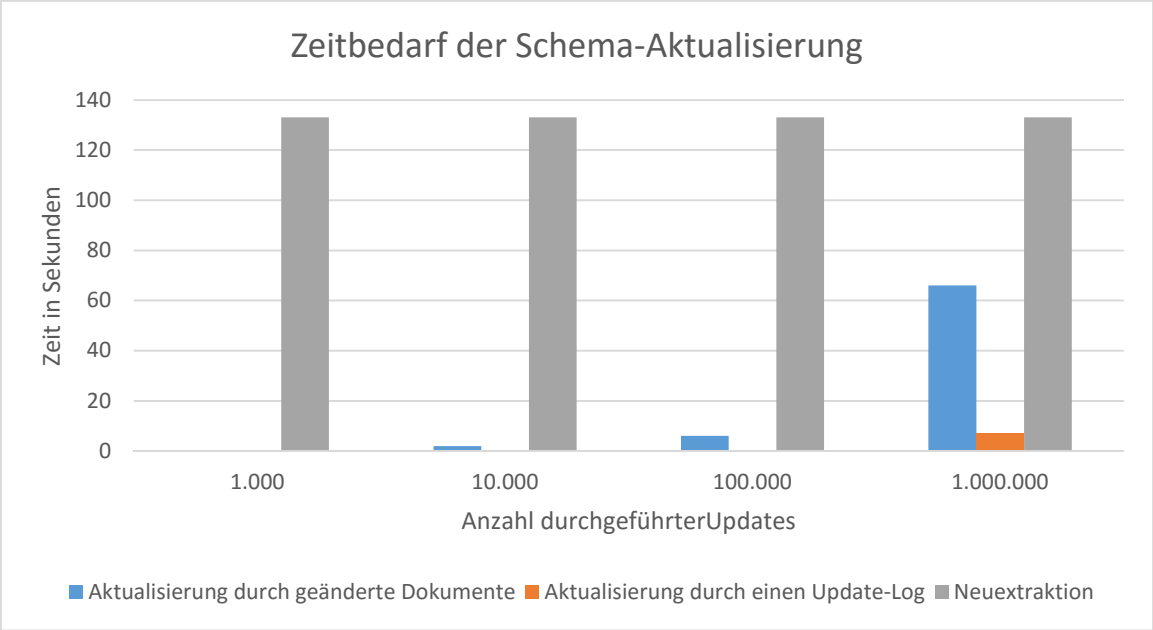

Abbildung 39: Vergleich der Schema-Aktualisierungsvarianten

Neben der vollständigen Extraktion wurden weitere Verfahren diskutiert. Darunter eine vereinfachte Extraktion, die zur Ableitung relevanter Metriken dient. Das Verfahren verzichtet auf die Speicherung von Dokument-IDs und somit auf die Möglichkeit einer korrekten Aktualisierung, spart dadurch aber erheblichen Speicher- und Zeitbedarf ein. Die Metriken können als Ausgangspunkt für weitere Schema-Operationen und Datenanalysen dienen. So kann beispielsweise eine Ausreißer-Analyse als Vorbereitung einer Datenbankbereinigung durchgeführt werden. Das vereinfachte Verfahren bietet sich für solch vorbereitende Analysen an, da das Schema nicht langfristig benötigt wird, sondern eine Momentaufnahme des Datenbestandes ausreicht und diese Momentaufnahme im Vergleich zur vollständigen Extraktion durch die vereinfachte Extraktion fast doppelt so schnell erstellt werden kann. (siehe Abbildung 28)

Ein weiteres Verfahren, das jedoch nicht implementiert wurde, ist ein Extraktionsverfahren, bei dem die Dokument-IDs nicht für jedes Element redundant gespeichert werden, sondern nur einmalig für das komplette Schema eines Dokuments. So wird der Speicherverbrauch für die Dokument-IDs pro Schema wesentlich verringert, allerdings bietet sich dieses Verfahren nur für sehr homogene Datensätze an, da jede Heterogenität zu einem neuen, zu speichernden Schema führt. Zudem leidet die Performance durch die zusätzlich benötigte Rechenzeit, die jeweils durch den Vergleich des gesamten Schemas mit allen vorhandenen Schemata entsteht. Die durchschnittliche Knotenzahl ist in der Regel im Vergleich zur Anzahl der Dokumente eher gering. Somit ist diese Methode vermutlich bereits bei wenigen Ausreißern oder Heterogenitäten in den Dokumenten vom Zeitbedarf schlechter als die vollständige Schema-Extraktion. Bei speicher-kritischen Extraktionen könnte sie dennoch einen entscheidenden Vorteil bieten. Da jedoch auch die verteilte Schema-Extraktion die speicher-kritischen Fälle abdeckt, wurde vorerst auf eine Implementation dieser Variante verzichtet.

Weitere Variationen wurden durch die Implementation des vollständigen Extraktionsverfahrens mit unterschiedlichen Datenstrukturen erprobt. Dabei wurde letztendlich die beste Variante für alle weiteren Verfahren übernommen. Verglichen wurden die Speicherung von Knoten- und Kantenmenge mit der Speicherung einer einzigen Knotenmenge, die Speicherung der Knoten

und Kanten in Objekten mit der Speicherung in Listen und Sets und die Speicherung der Dokument-IDs sowie der Datentypen in einfachen `ArrayLists`, `TreeSets` und `HashMaps`. Die effizienteste Kombination ergab sich aus der alleinigen Speicherung der Knoten in einer Knotenklasse, also mit Hilfe von Objekten, und den Datentypen und Dokument-IDs in `HashMaps`.

6 Zusammenfassung

Ein abschließendes Fazit soll noch einmal den Beitrag dieser Arbeit zusammenfassen. Ein anschließender Ausblick soll mögliche Schritte zur Weiterentwicklung des Algorithmus nennen, diesen aber zugleich auch in den Schema-Evolutionsprozess einordnen und für die Integration notwendige Maßnahmen aufzeigen.

6.1 Fazit

In dieser Arbeit wurden ausgewählte Schema-Extraktionsverfahren aus der Literatur ausgearbeitet und vorgestellt. Anschließend wurde ein Konzept für einen eigenen Algorithmus entwickelt, der mehrere Schema-Extraktionsverfahren in einem Programm integriert. Zu den Verfahren zählt eine vollständige Schema-Extraktion, die durch Reverse Engineering aus einer Dokumentkollektion zuerst ein internes Schema ableitet, welches anschließend in ein JSON Schema umgewandelt werden kann, gleichzeitig aber auch zwei Schema-Aktualisierungsverfahren als Ausgangsschema zur Verfügung steht. Sowohl das interne als auch das JSON Schema können gespeichert und somit beliebig zwischen Systemen transferiert werden. So können Schemata auch dezentral extrahiert und im Anschluss durch einen Schema-Merging-Algorithmus zusammengefügt werden. Bei den Aktualisierungsverfahren handelt es sich um inkrementelle Verfahren, die entweder mit Hilfe von geänderten JSON Dokumenten oder durch die Schritte eines Update-Logs das interne Schema aktualisieren können. Ein anderer Ansatz wurde in Form eines vereinfachten Extraktionsverfahrens implementiert, der sich ausschließlich auf die Extraktion der Informationen konzentriert, die für die Ausgabe im JSON Schema relevant sind und zudem spezielle Metriken ableitet, die im Kontext der Schema-Evolution von Bedeutung sein könnten. Durch diese Konzentration auf die Ausgabe des JSON Schemas und die Ableitung von Metriken konnten zeit- und speicherintensive Teilprozesse der vollständigen Extraktion eingespart werden, die die Geschwindigkeit dieses Verfahrens steigern. Durch gezielte Vorüberlegungen wurden verschiedene Umsetzungsmöglichkeiten diskutiert und für jede Überlegung eine begründete Entscheidung getroffen.
Das Konzept wurde anschließend durch die Implementation in Java in die Praxis umgesetzt. Dabei wurden verschiedene technische Umsetzungen implementiert und daraus auf der Basis von Tests die performanteste Variante für die endgültigen Verfahren ausgewählt. Die Betrachtung der Korrektheit der implementierten Extraktions- und Aktualisierungsverfahren zeigt Stärken und Schwächen der einzelnen Verfahren auf. Auch wenn sich die Verfahren nicht eindeutig miteinander vergleichen lassen, ist dennoch zu erkennen, dass die Aktualisierung mit Hilfe des Update-Logs in den meisten Fällen einen Geschwindigkeitsvorteil auf Kosten der gesicherten Korrektheit des Schemas gegenüber der Aktualisierung durch die geänderten Dokumente aufweisen wird. Die Aktualisierung durch die geänderten Dokumente kann zwar mehrere Schritte des Update-Logs in einem Schritt durchführen, falls diese dasselbe Dokument betreffen, muss allerdings stets alle Elemente des Dokuments durchlaufen. Dies bleibt bei den gezielten Update-Schritten eines Update-Logs erspart. Bei der Aktualisierung mit Hilfe des Update-Logs wurden zwei explizite Fälle aufgezeigt, bei denen das entwickelte Verfahren selbst mit komplexer Fehlerlogik mit dem derzeitigen internen Schema an seine Grenzen stößt. Bei nicht atomaren Operationen, wie dem Löschen von Containerelementen setzt der Algorithmus Methoden zur automatischen Wiederherstellung der Korrektheit ein. Diese Löschen automatisch die Vorkommen der Kindelemente von gelöschten Arrays und Objekten. Beim Löschen von Arrayobjekten und Arrays in Arrays kann jedoch aufgrund der

Generalisierung von Objekten in Arrays im internen Schema nicht mehr zwischen den Kindelementen einzelner Objekte unterschieden werden. Auch beim Aktualisieren von Datentypen, die eine Strukturveränderung mit sich ziehen, kann es bei fehlenden Informationen im Update-Log zu Fehlern kommen. Die Aktualisierung mit Hilfe des Update-Logs muss sich daher auf korrekte, atomare und ausführbare Anweisungen des Update-Logs berufen, das extern eingebunden wird und somit außerhalb der Kontrolle des Algorithmus selbst liegt.
Die Aktualisierung durch die geänderten Dokumente kann dagegen auf den JSON Standard aufbauen und somit eine fehlerfreie Aktualisierung garantieren. Beide Verfahren sind je nach Anzahl durchzuführender Updates wesentlich schneller als eine erneute Extraktion.
Die entwickelten Verfahren wurden zusammen mit weiteren Funktionen, wie dem Import und Export von Schemata und einem Schema-Merging-Bereich in eine GUI integriert. Diese dient neben einer verbesserten Bedienung auch der Ausgabe der JSON Schema, die ebenfalls gespeichert werden können. Ansätze komplexer Automatismen, die die Korrektheit des Schemas jederzeit garantieren sollen, wurden ebenso implementiert wie Präferenzoptionen, die die Ausgabe des Schemas oder weiterer Informationen steuern.

6.2 Ausblick

Die Entwicklung der Schema-Extraktionsverfahren ist soweit abgeschlossen, dass nun ausgiebige Tests und Feedback aus der praktischen Anwendung Anstöße zu Verbesserungen und zur Weiterentwicklung liefern können. Dazu zählt auch die Überprüfung der Zweckmäßigkeit der abgeleiteten Metriken. Bedarfsorientierte Änderungen, insbesondere die Implementation semantischer Verfahren, können ebenso umgesetzt werden. Weiterhin könnte das diskutierte, jedoch nicht umgesetzte, dritte Extraktionsverfahren implementiert und getestet werden, bei dem die Dokument-IDs für komplette Schemata und nicht elementweise in jedem Knoten gespeichert werden. Auch wäre die Implementation der dynamischen Auslagerung des internen Schemas kurz vor dem Speicherüberlauf denkbar. Falls sich die Aktualisierung mit Hilfe der Update-Logs als problematisch erweisen sollte, weil die Update-Logs keine atomaren Updates zur Verfügung stellen, kann auch eine Umstrukturierung des internen Schemas in Erwägung gezogen werden, so dass die entwickelten Automatismen zum Löschen der Kindelemente stets ein korrektes Schema garantieren. An dieser Stelle wäre auch die Implementation von heuristischen Methoden zur automatischen Identifikation von verschiedenen Arrayobjekten denkbar.
Im Rahmen der Schema-Evolution gilt es, das Verfahren in den Prozess zu integrieren. Als Vorbereitungsschritt für die Anwendung von Datenmigrationsschritten müssen die erhobenen Strukturinformationen überprüft werden und diese ggf. an die Bedürfnisse der Schema-Evolution angepasst werden. Durch Anbindung weiterer Datenbanken kann auch die Input-Seite erweitert werden. Zudem ist eine Automatisierung des Aktualisierungsprozesses, zum Beispiel durch eine Art Event-Listener für geänderte Dokumente und neue Updates, denkbar. Anschließend kann das aktualisierte Schema jederzeit ausgegeben werden und sowohl als Ausgangspunkt für die Anwendung von Techniken der Schema-Evolution auf dem Datenbestand verwendet werden als auch zur Kontrolle der durchgeführten Evolutionsschritte durch erneute Aktualisierung des Schemas dienen. Auch die Automatisierung gewisser Evolutionsverfahren auf Basis bestimmter Schema-Gegebenheiten ist denkbar.

Literaturverzeichnis

[1] SCHERZINGER, Stefanie ; KLETTKE, Meike ; STÖRL, Uta: *Managing schema evolution in NoSQL data stores*. In: *arXiv preprint arXiv:1308.0514* (2013)

[2] MOH, Chuang-Hue ; LIM, Ee-Peng ; NG, Wee-Keong: DTD-Miner: A tool for mining DTD from XML documents. In: *Advanced Issues of E-Commerce and Web-Based Information Systems, 2000. WECWIS 2000. Second International Workshop on*, 2000, S. 144–151

[3] HEGEWALD, J. ; NAUMANN, F. ; WEIS, M.: XStruct: Efficient Schema Extraction from Multiple and Large XML Documents. In: *22nd International Conference on Data Engineering Workshops (ICDEW'06)*, S. 81

[4] GAROFALAKIS, Minos ; GIONIS, Aristides ; RASTOGI, Rajeev ; SESHADRI, Sridhar ; SHIM, Kyuseok: XTRACT: a system for extracting document type descriptors from XML documents, Bd. 29. In: *ACM SIGMOD Record*, 2000, S. 165–176

[5] CHIDLOVSKII, Boris: Schema Extraction from XML: A Grammatical Inference Approach, Bd. 45. In: *KRDB*, 2001

[6] *Agile Vorgehensmodelle — Enzyklopaedie der Wirtschaftsinformatik*. URL http://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/is-management/Systementwicklung/Vorgehensmodell/Agile-Vorgehensmodelle – Überprüfungsdatum 2016-03-09

[7] ABITEBOUL, Serge: *Querying semi-structured data* : Springer, 1997

[8] PAPAKONSTANTINOU, Yannis ; GARCIA-MOLINA, Hector ; WIDOM, Jennifer: Object exchange across heterogeneous information sources. In: *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, 1995, S. 251–260

[9] GREINER, Ulrike: *Semistrukturierte Daten*. In: *Seminararbeit Uni Leipzig http://dbs. uni-leipzig. de/seminararbeiten/semSS99/arbeit3/inhalt. html*

[10] GAJENDRAN, Santhosh Kumar: *A survey on nosql databases*. In: *University of Illinois* (2012)

[11] CATTELL, Rick: *Relational Databases, Object Databases, Key-Value Stores, Document Stores, and Extensible Record Stores: A Comparison*. URL http://www.odbms.org/wp-content/uploads/2010/01/Cattell.Dec10.pdf – Überprüfungsdatum 2016-02-04

[12] *Definition » Schema « | Gabler Wirtschaftslexikon*. URL http://wirtschaftslexikon.gabler.de/Definition/schema.html – Überprüfungsdatum 2016-01-15

[13] *XML Schema Part 1: Structures Second Edition*. URL https://www.w3.org/TR/xmlschema-1/. – Aktualisierungsdatum: 2008-08-04 – Überprüfungsdatum 2016-02-02

[14] CHIDLOVSKII, Boris: Schema extraction from XML collections. In: *Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries*, 2002, S. 291–292

[15] WANG, Qiu Yue ; YU, Jeffrey Xu ; WONG, Kam-Fai: Approximate graph schema extraction for semi-structured data. In: *Advances in Database Technology—EDBT 2000* : Springer, 2000, S. 302–316

[16] *RFC 7159 - The JavaScript Object Notation …JSON— Data Interchange Format*. URL http://tools.ietf.org/pdf/rfc7159.pdf – Überprüfungsdatum 2015-12-04

[17] ECMA INTERNATIONAL: *The JSON Data Interchange Format*

[18] *draft-zyp-json-schema-04 - JSON Schema: core definitions and terminology*. URL http://tools.ietf.org/pdf/draft-zyp-json-schema-04.pdf – Überprüfungsdatum 2015-12-04

[19] *JSON Schema Software*. URL http://json-schema.org/implementations.html. – Aktualisierungsdatum: 2015-10-09 – Überprüfungsdatum 2016-01-02

[20] BRAY, Tim ; PAOLI, Jean ; SPERBERG-MCQUEEN, C. M. ; MALER, Eve ; YERGEAU, François: *Extensible Markup Language (XML) 1.0 (Third Edition)*. URL https://www.w3.org/TR/2004/REC-xml-20040204/. – Aktualisierungsdatum: 2004-02-02 – Überprüfungsdatum 2016-03-10

[21] CLARK, James ; MURATA, Makoto: *RELAX NG Specification*. URL http://relaxng.org/spec-20011203.html. – Aktualisierungsdatum: 2010-01-26 – Überprüfungsdatum 2016-03-10

[22] *JSON Schema: interactive and non interactive validation*. URL http://json-schema.org/latest/json-schema-validation.html. – Aktualisierungsdatum: 2016-01-25 – Überprüfungsdatum 2016-03-26

[23] *Springer Gabler Verlag, Gabler Wirtschaftslexikon, Stichwort: Algorithmus*. URL http://wirtschaftslexikon.gabler.de/Archiv/57779/algorithmus-v9.html – Überprüfungsdatum 2016-01-28

[24] *Springer Gabler Verlag, Gabler Wirtschaftslexikon, Stichwort: Anwendungsprogramm*. URL http://wirtschaftslexikon.gabler.de/Archiv/74942/anwendungsprogramm-v9.html – Überprüfungsdatum 2016-01-28

[25] *What is graphical user interface (GUI)? definition and meaning*. URL http://www.businessdictionary.com/definition/graphical-user-interface-GUI.html – Überprüfungsdatum 2016-03-12

[26] *ISO/IEC 25010:2011(en), Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. URL https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en – Überprüfungsdatum 2016-01-28

[27] *ArrayList (Java Platform SE 7)*. URL https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html. – Aktualisierungsdatum: 2015-12-08 – Überprüfungsdatum 2016-01-03

[28] *HashMap (Java Platform SE 7)*. URL https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html. – Aktualisierungsdatum: 2015-12-08 – Überprüfungsdatum 2016-01-03

[29] *TreeSet (Java Platform SE 7)*. URL
https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html. – Aktualisierungsdatum: 2015-12-08 – Überprüfungsdatum 2016-01-03

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. Ich weiß, dass bei Abgabe einer falschen Versicherung die Prüfung als nicht bestanden zu gelten hat.

Ort, Datum

Unterschrift

Anhang

Anhang A – zusätzliche Abbildungen

```
2595243 Documents were found in the collection! The program will begin to iterate through all of them now.
Der 1000000. Durchlauf wurde nach 7 Sekunden beendet. Es sind 10 Knoten und 9 Kanten gespeichert.
Der 2000000. Durchlauf wurde nach 14 Sekunden beendet. Es sind 10 Knoten und 9 Kanten gespeichert.
Der 3000000. Durchlauf wurde nach 22 Sekunden beendet. Es sind 10 Knoten und 9 Kanten gespeichert.
Der 4000000. Durchlauf wurde nach 29 Sekunden beendet. Es sind 10 Knoten und 9 Kanten gespeichert.
Der 5000000. Durchlauf wurde nach 37 Sekunden beendet. Es sind 10 Knoten und 9 Kanten gespeichert.
Der 6000000. Durchlauf wurde nach 45 Sekunden beendet. Es sind 10 Knoten und 9 Kanten gespeichert.
Der 7000000. Durchlauf wurde nach 52 Sekunden beendet. Es sind 10 Knoten und 9 Kanten gespeichert.
Der 8000000. Durchlauf wurde nach 60 Sekunden beendet. Es sind 10 Knoten und 9 Kanten gespeichert.
Der 9000000. Durchlauf wurde nach 68 Sekunden beendet. Es sind 10 Knoten und 9 Kanten gespeichert.
Der 10000000. Durchlauf wurde nach 75 Sekunden beendet. Es sind 10 Knoten und 9 Kanten gespeichert.
Der 11000000. Durchlauf wurde nach 83 Sekunden beendet. Es sind 10 Knoten und 9 Kanten gespeichert.
Der 12000000. Durchlauf wurde nach 91 Sekunden beendet. Es sind 11 Knoten und 11 Kanten gespeichert.
Der 13000000. Durchlauf wurde nach 98 Sekunden beendet. Es sind 11 Knoten und 11 Kanten gespeichert.
Der 14000000. Durchlauf wurde nach 107 Sekunden beendet. Es sind 11 Knoten und 11 Kanten gespeichert.
Der 15000000. Durchlauf wurde nach 116 Sekunden beendet. Es sind 11 Knoten und 11 Kanten gespeichert.
Der 16000000. Durchlauf wurde nach 125 Sekunden beendet. Es sind 11 Knoten und 11 Kanten gespeichert.
Der 17000000. Durchlauf wurde nach 153 Sekunden beendet. Es sind 11 Knoten und 11 Kanten gespeichert.
Der 18000000. Durchlauf wurde nach 191 Sekunden beendet. Es sind 11 Knoten und 11 Kanten gespeichert.
Der 19000000. Durchlauf wurde nach 248 Sekunden beendet. Es sind 11 Knoten und 11 Kanten gespeichert.
Der 20000000. Durchlauf wurde nach 343 Sekunden beendet. Es sind 11 Knoten und 11 Kanten gespeichert.
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
        at sun.nio.cs.UTF_8.newDecoder(UTF_8.java:68)
        at java.lang.StringCoding.decode(StringCoding.java:213)
        at java.lang.String.<init>(String.java:463)
        at java.lang.String.<init>(String.java:515)
        at org.bson.io.ByteBufferBsonInput.readCString(ByteBufferBsonInput.java:124)
        at org.bson.BsonBinaryReader.readBsonType(BsonBinaryReader.java:113)
        at org.bson.codecs.DocumentCodec.decode(DocumentCodec.java:139)
        at org.bson.codecs.DocumentCodec.decode(DocumentCodec.java:45)
        at com.mongodb.connection.ReplyMessage.<init>(ReplyMessage.java:57)
        at com.mongodb.connection.GetMoreProtocol.receiveMessage(GetMoreProtocol.java:124)
        at com.mongodb.connection.GetMoreProtocol.execute(GetMoreProtocol.java:68)
        at com.mongodb.connection.GetMoreProtocol.execute(GetMoreProtocol.java:37)
        at com.mongodb.connection.DefaultServer$DefaultServerProtocolExecutor.execute(DefaultServer.java:155)
        at com.mongodb.connection.DefaultServerConnection.executeProtocol(DefaultServerConnection.java:219)
        at com.mongodb.connection.DefaultServerConnection.getMore(DefaultServerConnection.java:194)
        at com.mongodb.operation.QueryBatchCursor.getMore(QueryBatchCursor.java:197)
        at com.mongodb.operation.QueryBatchCursor.hasNext(QueryBatchCursor.java:93)
        at com.mongodb.MongoBatchCursorAdapter.hasNext(MongoBatchCursorAdapter.java:46)
        at extractSchema.extractFromMongoCollection.extract(extractFromMongoCollection.java:38)
        at Database.Main.main(Main.java:32)
```

Abbildung 40: Ausgabe der Java-Konsole und „OutOfMemory“-Fehlermeldung

Abbildung 41: Heap Memory Auslastung bei der Extraktion mit der 1. Implementation bis ~17Mio Durchläufe

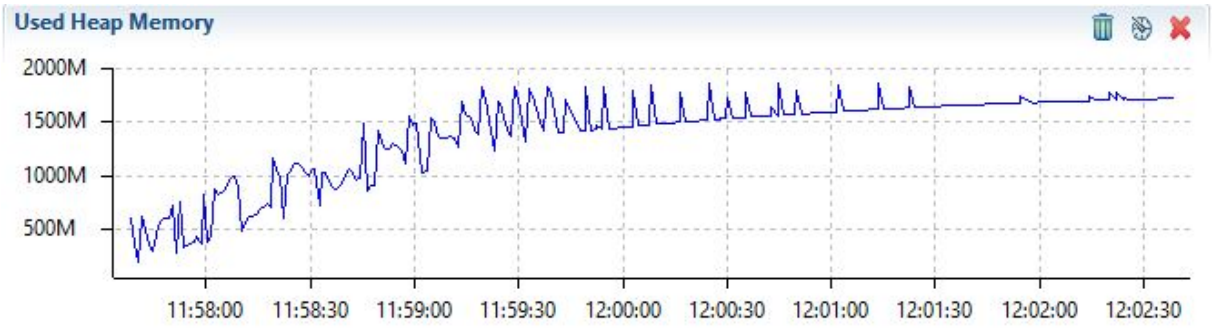

Abbildung 42: Heap Memory Auslastung bei der Extraktion mit der 1. Implementation bis ~19Mio Durchläufe

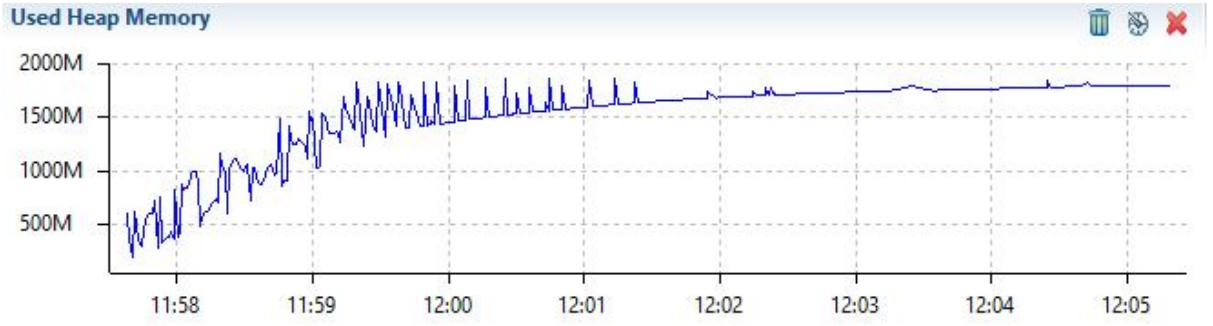

Abbildung 43: Heap Memory Auslastung bei der Extraktion mit der 1. Implementation ~20Mio Durchläufe

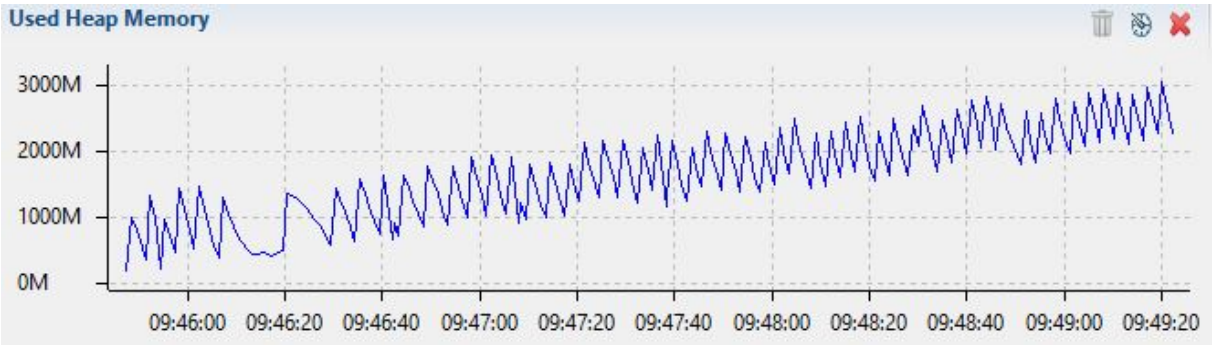

Abbildung 44: Heap Memory Auslastung bei der Extraktion mit Knoten und Kanten in HashMaps bei 4GB RAM

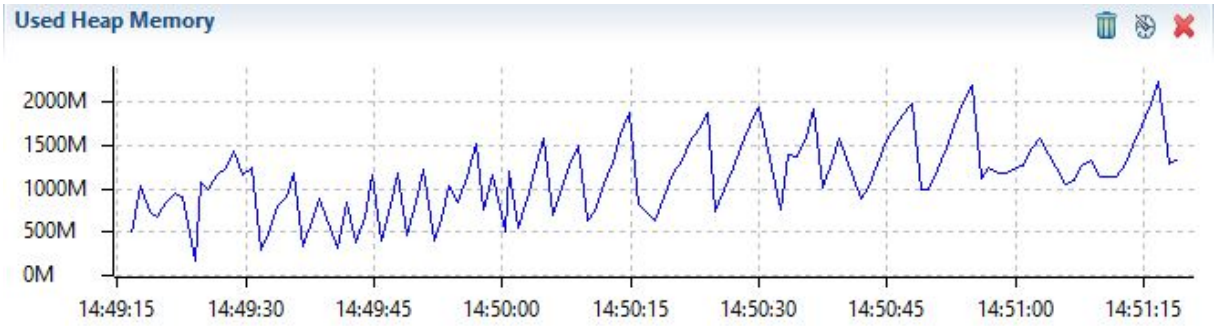

Abbildung 45: Heap Memory Auslastung bei der Extraktion mit nur Knotenobjekten

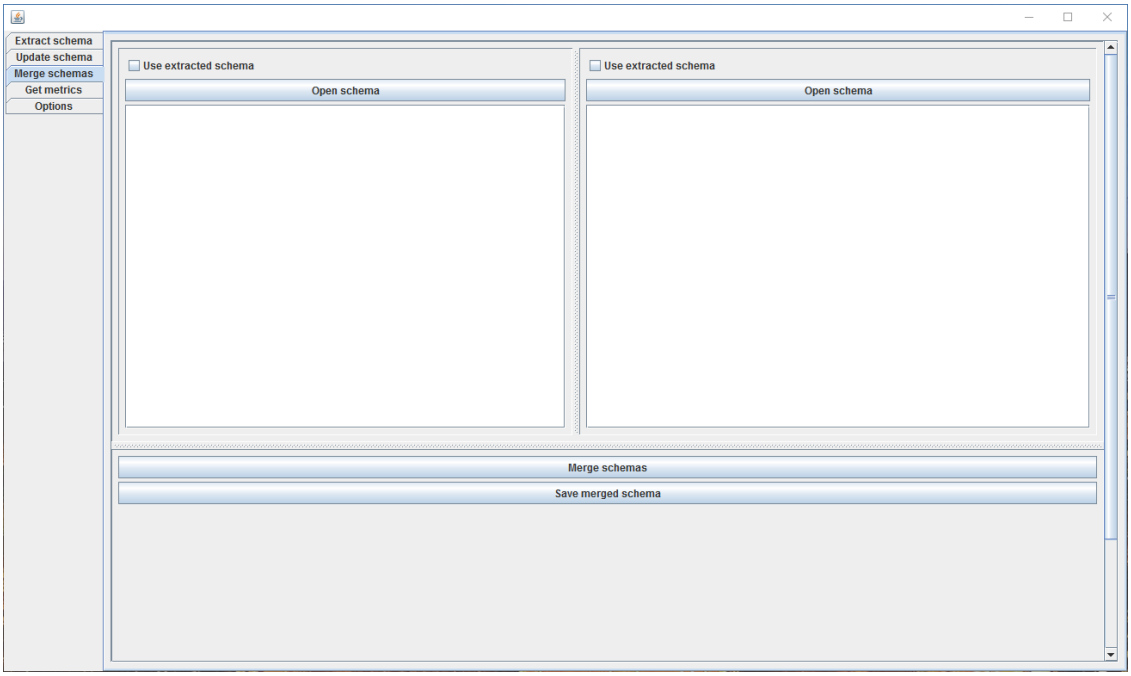

Abbildung 46: GUI zum Merging zweier Schemata

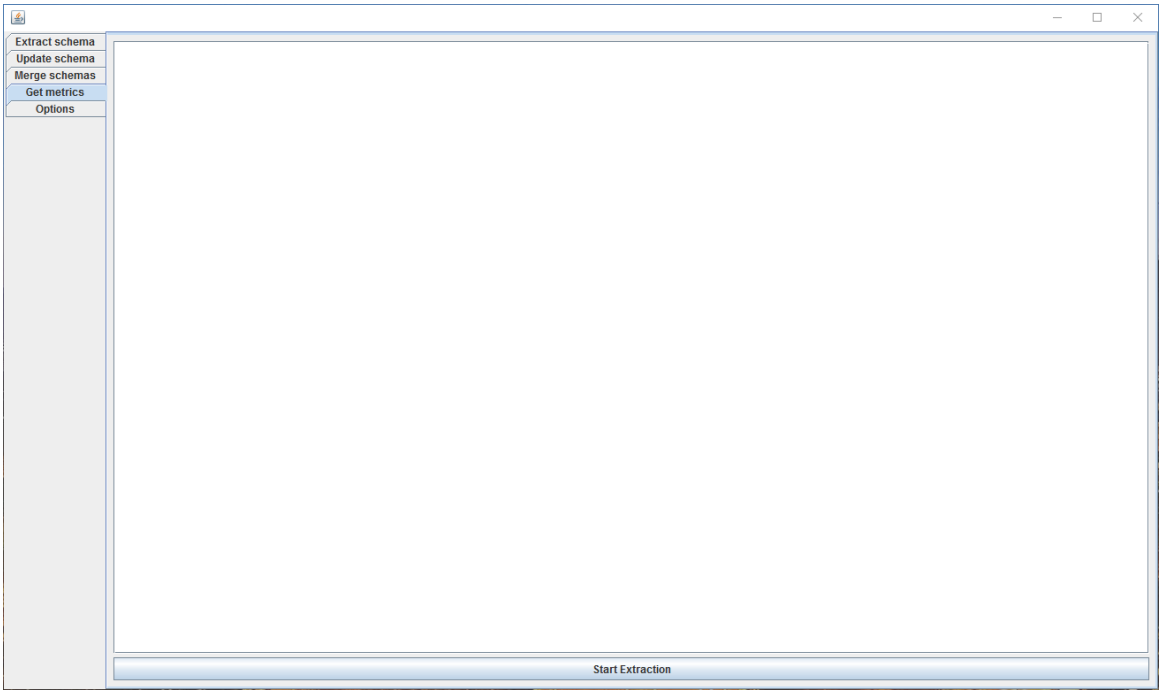

Abbildung 47: GUI der vereinfachten Extraktionskomponente

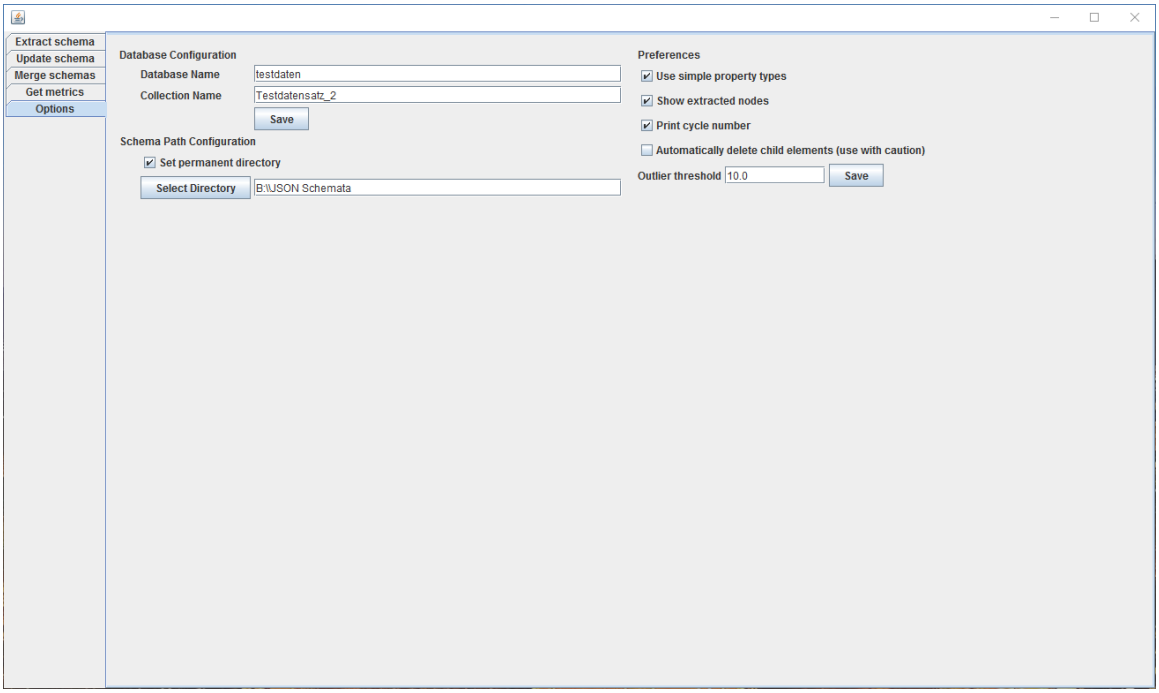

Abbildung 48: GUI zur Konfiguration des Tools

Anhang B – Pseudo-Programmcode

Pseudocode der Bestandteile der Extraktionskomponente

class extractFromMongoCollection

```
extractAll(database, collectionName, operation)
    -   connect to database
    -   retrieve collection
    -   path = new ArrayList()
    -   add database to path
    -   foreach (document in collection)
            o   docId = document.get("_id") or generate new docId
            o   if (operation equals "update"
                    ▪   foreach (node in nodeList)
                            •   delete = node.deleteCompleteDocId(docId)
                            •   if (delete equals "NoDocumentsLeft")
                                    o   remove node from nodeList
                    ▪   if (document.size() > 1) // size == 1 equals document which has only
                        the _id-field and by definition must be deleted from the schema
                            •   extractFromJsonDocument.extract(parser.parse(document.toJson()
                                ), collectionName, path, docId, 0)
            o   else
                    ▪   extractFromJsonDocument.extract(parser.parse(document.toJson()),
                        collectionName, path, docId, 0)
    -   close database connection
```

```
query(database, collectionName, operation, field, operator, value, type)
    -   connect to database
    -   build query from field, operator, value and type
    -   retrieve documents with query
    -   path = new ArrayList()
    -   add database to path
    -   foreach (document in collection)
            o   docId = document.get("_id") or generate new docId
            o   if (operation equals "update"
                    ▪   foreach (node in nodeList)
                            •   delete = node.deleteCompleteDocId(docId)
                            •   if (delete equals "NoDocumentsLeft")
                                    o   remove node from nodeList
                    ▪   if (document.size() > 1) // size == 1 equals document which has only
                        the _id-field and by definition must be deleted from the schema
                            •   extractFromJsonDocument.extract(parser.parse(document.toJson()
                                ), collectionName, path, docId, 0)
            o   else
                    ▪   extractFromJsonDocument.extract(parser.parse(document.toJson()),
                        collectionName, path, docId, 0)
    -   close database connection
```

class extractFromJsonDocument

```
extract(jsonElement, nodeName, path, docId, level)
    •   propType = getJsonType(jsonElement)
    •   storeNode(nodeName, path, propType, docId, level)
    •   newPath = path
    •   add nodeName to newPath
    •   if (node.isJsonobject)
            o   foreach (key-value-pair in jsonElement)
                    ▪   extract(parser.parse(value), key, newPath, docId, level++)
    •   if (node.isJsonArray)
            o   jsonArray = node.getAsJsonArray()
            o   arrayOrder = new HashMap
            o   currentElementType = ""
            o   typeCount = 0
            o   position = 1
            o   if (jsonArray is empty)
```

```
                ▪ add (0, emptyEntry) to arrayOrder
        o arrayIterator = jsonArray.iterator()
        o while (arrayIterator.hasNext())
            ▪ arrayElement = arrayIterator.next()
            ▪ elementType = getJsonType(arrayElement)
            ▪ if (elementType equals currentElementType)
                • typeCount++
            ▪ else
                • if (currentElementType is not empty)
                    o add (position, (currentElementType, typeCount)) to arrayOrder
                    o position++
                • currentElementType = elementType
                • typeCount = 1
            ▪ if (arrayElement.isJsonObject())
                • extract(parser.parse(arrayElement), “ArrayObject”, newPath, docId, level++)
            ▪ else if (arrayElement.isJsonArray)
                • extract(parser.parse(arrayElement), “ArrayArray”, newPath, docId, level++)
            ▪ else
                • extract(parser.parse(arrayElement), “ArrayPrimitive”, newPath, docId, level++)
            ▪ if (currentElementType is not empty)
                • add (position, (currentElementType, typeCount)) to arrayOrder
            ▪ find current node in nodeList
            ▪ add arrayOrder to this node
```

```
getJsonType(JsonElement)
    - test JsonElement against every JsonType
    - return matching type
```

```
storeNode(nodeName, path, propType, docId, level)
    - boolean alreadyExists = false
    - foreach (node in nodeList)
        o if (node.getPath equals path && node.getName equals nodeName)
            ▪ alreadyExists = true
            ▪ node.setPropType(propType, 1)
            ▪ node.setDocId(docId, 1)
            ▪ break
    - if (alreadyExists == false)
        o create new node
        o add all attributes to the new node
        o add node to nodeList
```

Tabelle 10: vereinfachte Pseudocode-Darstellung der Methoden der Extraktionskomponente

Pseudocode der Bestandteile der Aktualisierungskomponente

class mergeWithChangedDocuments

```
updateAll(database, collectionName)
    - if (database not equal to databaseOfExtractedSchema || collectionName not equal to collectionNameOfExtractedSchema)
        o foreach (node in extractedSchema)
            ▪ node.replaceDatabaseName(database)
            ▪ node.replaceCollectionName(collectionName)
    - extractFromMongoCollection.extractAll(database, collectionName, „update“)
```

```
updateQuery(database, collectionName, field, operator, value, type)
    - if (database not equal to databaseOfExtractedSchema || collectionName not equal to collectionNameOfExtractedSchema)
        o foreach (node in extractedSchema)
            ▪ node.replaceDatabaseName(database)
            ▪ node.replaceCollectionName(collectionName)
    - extractFromMongoCollection.query(database, collectionName, „update“, field, operator, value, type)
```

class mergeWithUpdateLog

```
importUpdates(documentPath)
    - updates = new ArrayList
    - if (documentPath.endsWith(„.txt“))
```

```
            o   foreach (BufferedReader.readline())
                    ▪   create new update object
                    ▪   add update information with the help of a JsonParser
                    ▪   add update to updates
    -   else if (documentPath.endsWith(„.csv“))
            o   foreach (BufferedReader.readline())
                    ▪   create new update object
                    ▪   add update information with the help of String.split
                    ▪   add update to updates
    -   if (updates is not empty)
            o   merge(updates)
merge(updates)
    -   foreach (update in updates)
            o   boolean foundMatch = false
            o   if (update.getUpdateType equals „insert“)
                    ▪   foreach (node in extractedSchema)
                            •   if (node.getNodeName == update.getNodeName && node.getPath ==
                                update.getPath)
                                    o   foundMatch = true
                                    o   node.setDocId(update.getDocId, 1)
                                    o   node.setPropType(update.getPropType, 1)
                            •   if (node.getLevel == 0)
                                    o   node.setMaximumDocId(update.getDocId, 1)
                    ▪   if (foundMatch == false)
                            •   create new node
                            •   add information given in update
                            •   add node to extractedSchema
            o   else if (update.getUpdateType equals „delete“)
                    ▪   foreach (node in extractedSchema)
                            •   if (node.getNodeName == update.getNodeName && node.getPath ==
                                update.getPath)
                                    o   foundMatch == true
                                    o   delete = node.deleteDocId(update.getDocId)
                                    o   if (delete equals „NoDocumentsLeft“)
                                            ▪   remove node from extractedSchema
                                    o   else if (delete equals „Ok“)
                                            ▪   if (update.getPropTypeBeforeChange is not empty)
                                                    •   node.deletePropType(update .
                                                        getPropTypeBeforeChange)
                                            ▪   else if (node.getPropType.size() == 1)
                                                    •   propType =
                                                        node.getPropType.keySet.toArray[0]
                                                    •   node.deletePropType(propType)
                                    o   break
                    ▪   if (foundMatch == false)
                            •   add „ArrayObject“ to update.getPath
                            •   try again
                    ▪   if (foundMatch == false)
                            •   throw customException
                    ▪   else if (deleteChildElements == true) // set in options tab
                            •   if (update.getPropTypeBeforeChange == “JsonObject” || propType
                                == “JsonObject”)
                                    o   pathOfChildren = update.getPath
                                    o   add update.getNodeName to pathOfChildren
                                    o   deleteChildren(pathOfChildren, update.getDocId)
                            •   else if (update.getPropTypeBeforeChange == “JsonArray” ||
                                propType == “JsonArray”)
                                    o   pathOfChildren = update.getPath
                                    o   add update.getNodeName to pathOfChildren
                                    o   deleteAllChildren(pathOfChildren, update.getDocId)
                    ▪   boolean foundDocId = false
                    ▪   foreach (node in extractedSchema)
                            •   If (node.getLevel != 0 &&
                                node.getDocId.contains(update.getDocId))
                                    o   foundDocId = true
                                    o   break
                    ▪   if (foundDocId == false)
                            •   foreach (node in extractedSchema)
                                    o   if (node.getLevel == 0)
                                            ▪   node.deleteDocId(update.getDocId)
```

```
                                        ▪ break
            o   else if (update.getUpdateType equals „update“)
                    ▪ foreach (node in extractedSchema)
                        • if (node.getNodeName == update.getNodeName && node.getPath == update.getPath)
                            o foundMatch == true
                            o if (update.getPropType != node.getPropType)
                                ▪ if (update.getPropTypeBeforeChange is not empty)
                                    • node.deletePropType(update.getPropTypeBeforeChange)
                                ▪ else if (node.getPropType.size() == 1)
                                    • propType = node.getPropType.keySet.toArray[0]
                                    • node.deletePropType(propType)
                                ▪ node.setPropType(update.getPropTypeAfterChange)
                            o break
                    ▪ if (foundMatch == false)
                        • throw customException
deleteChildren(path, docId)
    -   foreach (node in extractedSchema)
            o   if (node.getPath equals path)
                    ▪ if (node.getPropType contains „JsonArray“)
                        • newPath = path
                        • add node.getName to newPath
                        • deleteAllChildren(newPath, docId)
                        • delete = node.deleteCompleteDocId(docId)
                        • if (delete equals „NoDocumentsLeft“)
                            o remove node from extractedSchema
                    ▪ else if (node.getPropType contains “JsonObject”)
                        • newPath = path
                        • add node.getName to newPath
                        • deleteChildren(newPath, docId)
                        • Knoten.deleteDocId(Dokument-ID)
                        • delete = node.deleteDocId(docId)
                        • if (delete equals „NoDocumentsLeft“)
                            o remove node from extractedSchema
                        • else if (node.getPropType.size() == 1)
                            o propType = node.getPropType.keySet.toArray[0]
                            o node.deletePropType(propType)
                    ▪ else
                        • delete = node.deleteDocId(docId)
                        • if (delete equals „NoDocumentsLeft“)
                            o remove node from extractedSchema
                        • else if (node.getPropType.size() == 1)
                            o propType = node.getPropType.keySet.toArray[0]
                            o node.deletePropType(propType)
deleteAllChildren(path, docId)
    -   foreach (node in extractedSchema)
            o   if (node.getPath equals path)
                    ▪ if (node.getPropType contains „JsonArray“ or “JsonObject”)
                        • newPath = path
                        • add node.getName to newPath
                        • deleteAllChildren(newPath, docId)
                    ▪ delete = node.deleteCompleteDocId(docId)
                    ▪ if (delete equals „NoDocumentsLeft“)
                        • remove node from extractedSchema
```

Tabelle 11: vereinfachte Pseudocode-Darstellung der Methoden der Aktualisierungskomponente

Pseudocode der Bestandteile der Visualisierungskomponente

class toJsonSchema

```
print(nodeList)
    -   find root element in nodeList (with level == 0)
    -   extract database, collectionName, path, docIdCount, elementCount from root element
    -   add collectionName to path
    -   schema = new Document
    -   add title, description and $schema-fields and content to schema
    -   properties = new Document
    -   required = new JsonArray
    -   foreach (node in nodeList)
```

```
            o   if (node.getLevel == 1)
                    ▪   properties.append(node.getName,            printElement(node.getName,
                        elementCount, docIdCount, path, nodeList)
                    ▪   if (elementCount >= node.countMembers && docIdCount == node.countDocId)
                            •   required.add(node.getName)
    -   schema.append(„propterties“, properties)
    -   schema.append(„required properties“, required)
    -   return schema
printElement(nodeName, parentElementCount, parentDocIdCount, path, nodeList)
    -   schema = new Document
    -   pathOfChildren = path
    -   add nodeName to pathOfChildren
    -   foreach (node in nodeList)
            o   if (node.getName equals nodeName && node.getPath equals path)
                    ▪   schema.append(„type“, node.getPropType)
                    ▪   calculate absolute and relative occurences
                    ▪   String description = „“
                    ▪   if (node.countMembers != node.countDocId)
                            •   description = „Document occurences“ + docCount + „Element
                                occurences“ + elementCount
                    ▪   else
                            •   description = „Occurences“ + docCount
                    ▪   if (propType contains “JsonObject”)
                            •   properties = new Document
                            •   required = new JsonArray
                            •   foreach (node m in nodeList)
                                    o   if (m.getPath equals pathOfChildren)
                                            ▪   properties.append(m.getName,
                                                printElement(m.getName, elementCount, docCount,
                                                pathOfChildren, nodeList))
                                            ▪   if (elementCount >= m.countMembers && docCount
                                                == m.countDocId)
                                                    •   required.add(m.getName)
                            •   schema.append(„properties“, properties)
                            •   schema.append(„required“, required)
                    ▪   if (propType contains „JsonArray“)
                            •   order = new Document
                            •   arrayOrder = node.getArrayOrder
                            •   format and append each element in arrayOrder to order
                            •   extendedDescription = new Document
                            •   extendedDescription.append(„occurences“, description)
                            •   extendedDescription.append(„array order“, order)
                            •   schema.append(“description”, extendedDescription)
                            •   childElements = new ArrayList
                            •   foreach (node m in nodeList)
                                    o   if (m.getPath == pathOfChildren)
                                            ▪   add m to childElements
                            •   if (childElements.size() == 1)
                                    o   schema.append(„items“,
                                        printElement(childElements.get(0).getName(),
                                        elementCount, docCount, pathOfChildren, nodeList))
                            •   else
                                    o   anyOf = new ArrayList
                                    o   foreach (node m in childElements)
                                            ▪   temp = new Document
                                            ▪   temp  =  printElment(m.getName(),  elementCount,
                                                docCount, pathOfChildren, nodeList)
                                            ▪   add temp to anyOf
                                    o   if (arrayOrder contains 0)
                                            ▪   add new Document to anyOf
                                    o   items = new Document
                                    o   items.append(“anyOf”, anyOf)
                                    o   schema.append(“items”, items)
                    ▪   if (schema not contains “description”)
                            •   schema.append(“description”, description)
                    ▪   break
    -   return schema
```

```
printSimpleSchema(simpleNodeList)
    -   find root element in nodeList (with level == 0)
    -   extract database, collectionName, path, elementCount from root element
    -   add collectionName to path
    -   schema = new Document
    -   add title, description and $schema-fields and content to schema
    -   properties = new Document
    -   required = new JsonArray
    -   sameNodes = new Document
    -   outliers = new Document
    -   foreach (simpleNode in simpleNodeList)
            o   if (simpleNode.getLevel == 1)
                    ▪   properties.append(simpleNode.getName,
                        printSimpleElement(simpleNode.getName, elementCount, path, nodeList)
                    ▪   if (elementCount == simpleNode.getDocId)
                            •   required.add(simpleNode.getName)
            o   // metric part
            o   outliersOfN = new Document
            o   sameNames = new Document
            o   sameNames.append(„Nodename:  “  +  simpleNode.getName,  „Parentname:  “  +
                simpleNode.getPath().get(simpleNode.getPath().size() - 1))
            o   sameNames.append(„Path of 1. candidate“, printPath(simpleNode.getPath())
            o   Integer names = 1
            o   sameChildren = new Document
            o   pathOfN = simpleNode.getPath
            o   add simpleNode.getName to pathOfN
            o   Integer children = 1
            o   for (simpleNode m in simpleNodeList)
                    ▪   if(simpleNode.getName  equals  m.getName  &&  simpleNode.getPath  !=
                        m.getPath) // equal names and different paths
                            •   // check for equal parent names
                            •   if (simpleNode.getPath().get(simpleNode.getPath().size() - 1)
                                equals m.getPath().get(m.getPath().size() - 1)))
                                    o   names++
                                    o   sameNames.append(“Path of “ + names + “. candidate“,
                                        printPath(m.getPath()))
                            •   // check for same children
                            •   childrenOfN = new ArrayList
                            •   childrenOfM = new ArrayList
                            •   pathOfM = m.getPath
                            •   add m.getName to path
                            •   foreach (simpleNode o in simpleNodeList)
                                    o   if (o.getPath equals pathOfN)
                                            ▪   add o to childrenOfN
                                    o   if (o.getPath equals pathOfM)
                                            ▪   add o to childrenOfM
                            •   if(childrenOfN equals childrenOfM && childrenOfN is not empty)
                                    o   if (sameChildren is empty)
                                            ▪   sameChildren.append(“Names  of  the  child
                                                elements”, childrenOfN)
                                            ▪   sameChildren.append(“Path  of  1.candidate”,
                                                printPath(simpleNode.getPath)
                                    o   children++
                                    o   sameChildren.append(“Path  of  “  +  children  +  “.
                                        candidate“, printPath(m.getPath)
                    ▪   if (m.getPath equals pathOfN)
                            •   countOfChildOfN = m.countDocId
                            •   if (countOfChildOfN / countOfN <= outlierPercentage) // set in
                                options tab
                                    o   outliersOfN.append(m.getName,  “Relative  and  absolute
                                        occurrences”)
            o   if (names > 1)
                    ▪   sameNodes.append(“Same nodename and parentname”, sameNames)
            o   if (children > 1)
                    ▪   sameNodes.append(“Same child elements”, sameChildren)
            o   if (outliersOfN is not empty)
                    ▪   outliers.append(“Outliers of node “ + printPath(pathOfN), outliersOfN)
    -   schema.append("properties", properties)
    -   schema.append("required properties", required)
    -   schema.append("possibly equal nodes", sameNodes)
```

```
    - schema.append("outliers with relative occurence lower than " + outlierPercentage*100
      + "%", outliers)
    - return schema
printSimpleElement(nodeName, parentOcc, path, simpleNodeList)
    - schema = new Document
    - pathOfChildren = path
    - add nodeName to pathOfChildren
    - foreach (simpleNode in simpleNodeList)
        o if (simpleNode.getName equals nodeName && simpleNode.getPath equals path)
            ▪ schema.append(„type“, simpleNode.getPropType)
            ▪ calculate absolute and relative occurences
            ▪ String description = “”
            ▪ description = „Occurences“ + docCount
            ▪ if (propType contains “JsonObject”)
                • properties = new Document
                • required = new JsonArray
                • foreach (simpleNode m in simpleNodeList)
                    o if (m.getPath equals pathOfChildren)
                        ▪ properties.append(m.getName,
                          printSimpleElement(m.getName, elementCount,
                          pathOfChildren, simpleNodeList))
                        ▪ if (elementCount == m.countDocId)
                            • required.add(m.getName)
                • schema.append(„properties“, properties)
                • schema.append(„required“, required)
            ▪ if (propType contains „JsonArray“)
                • order = new Document
                • arrayOrder = node.getArrayOrder
                • format and append each element in arrayOrder to order
                • extendedDescription = new Document
                • extendedDescription.append(„occurences“, description)
                • extendedDescription.append(„array order“, order)
                • schema.append(“description”, extendedDescription)
                • childElements = new ArrayList
                • foreach (simpleNode m in nodeList)
                    o if (m.getPath == pathOfChildren)
                        ▪ add m to childElements
                • if (childElements.size() == 1)
                    o schema.append(„items“,
                      printSimpleElement(childElements.get(0).getName(),
                      elementCount, pathOfChildren, simpleNodeList))
                • else
                    o anyOf = new ArrayList
                    o foreach (node m in childElements)
                        ▪ temp = new Document
                        ▪ temp = printElment(m.getName(), elementCount,
                          pathOfChildren, simpleNodeList)
                        ▪ add temp to anyOf
                    o if (arrayOrder contains 0)
                        ▪ add new Document to anyOf
                    o items = new Document
                    o items.append(“anyOf”, anyOf)
                    o schema.append(“items”, items)
            ▪ if (schema not contains “description”)
                • schema.append(“description”, description)
            ▪ break
    - return schema
printPath(path)
    - convertedPath = „/“
    - for (String s : path)
        o convertedPath = convertedPath + s + „/“
    - return convertedPath
class showSchema
printSchema(schema, outputArea)
    - gson = new GsonBuilder.setPrettyPrinting().create()
    - print = gson.toJson(schema)
    - outputArea.append(print + “\n”)
```

Tabelle 12: vereinfachte Pseudocode-Darstellung der Methoden der Visualisierungskomponente

Anhang C – Performance-Messungen

Schleifen-durchläufe in Millionen	Extraktion von Knoten und Kanten in Objekte	Extraktion von Knoten und Kanten in HashMaps	Extraktion von Knoten und Kanten in Objekte (reduzierte Kantenanzahl)	Extraktion der Knoten in Objekte	Vereinfachte Extraktion der Knoten in Objekte
1	8	11	7	6	6
2	15	19	13	12	9
3	22	27	20	17	12
4	29	35	26	23	16
5	36	44	32	28	19
6	44	53	38	33	22
7	52	61	45	39	26
8	59	70	51	45	29
9	69	81	58	50	32
10	76	89	66	56	36
11	84	98	72	61	39
12	92	107	79	67	42
13	102	118	86	72	46
14	110	127	93	79	49
15	117	136	101	85	52
16	125	146	108	92	56
17	133	155	115	97	59
18	142	164	122	102	62
19	150	174	129	109	66
20	165	187	136	114	69
21	173	197	142	119	72
22	181	207	149	126	76
23	190	217	159	131	79
23,46	194	221	162	133	80

Tabelle 13: Messung des Zeitbedarfs der Extraktionsverfahren

Anzahl Updates	Aktualisierung durch geänderte Dokumente	Aktualisierung durch einen Update-Log
1	0	0
10	0	0
100	0	0
1000	0	0
10000	2	0
100000	6	0
1000000	66	7

Tabelle 14: Messung des Zeitbedarfs der Aktualisierungsverfahren

Anhang D – Schemata

Vollständiges Schema des Testdatensatz_1


```
{
  "title": "Testdatensatz_1",
  "description": "Json Schema for collection \"Testdatensatz_1\" of database \"testdaten\",
created on 2016-03-16 12:43:53",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "properties": {
    "_id": {
      "type": "JsonObject",
      "description": "Occurence: 100% (25.359/25.359)",
      "properties": {
        "$oid": {
          "type": "String",
          "description": "Occurence: 100% (25.359/25.359)"
        }
      },
      "required properties": [
        "$oid"
      ]
    },
    "address": {
      "type": "JsonObject",
      "description": "Occurence: 100% (25.359/25.359)",
      "properties": {
        "building": {
          "type": "String",
          "description": "Occurence: 100% (25.359/25.359)"
        },
        "coord": {
          "type": "JsonArray",
          "description": "Occurence: 100% (25.359/25.359)",
          "array order": {
            "Number": "Min occurence: 0; Max occurence: 2"
          },
          "items": {
            "type": "Number",
            "description": "Document Occurence: 99,99% (25.357/25.359); Member Occurence:
199,98% (50.714/25.359)"
          }
        },
        "street": {
          "type": "String",
          "description": "Occurence: 100% (25.359/25.359)"
        },
        "zipcode": {
          "type": "String",
          "description": "Occurence: 100% (25.359/25.359)"
        },
        "date": {
          "type": "String",
          "description": "Occurence: 0% (1/25.359)"
        }
      },
```

```
      "required properties": [
        "building",
        "coord",
        "street",
        "zipcode"
      ]
    },
    "borough": {
      "type": "String",
      "description": "Occurence: 100% (25.359/25.359)"
    },
    "cuisine": {
      "type": "String",
      "description": "Occurence: 100% (25.359/25.359)"
    },
    "grades": {
      "type": "JsonArray",
      "description": "Occurence: 100% (25.359/25.359)",
      "array order": {
        "JsonObject": "Min occurence: 0; Max occurence: 9"
      },
      "items": {
        "type": "JsonObject",
        "description": "Document Occurence: 97,09% (24.621/25.359); Member Occurence: 368,56%
(93.463/25.359)",
        "properties": {
          "date": {
            "type": "JsonObject",
            "description": "Document Occurence: 100% (24.621/24.621); Member Occurence: 100%
(93.463/93.463)",
            "properties": {
              "$date": {
                "type": "Number",
                "description": "Document Occurence: 100% (24.621/24.621); Member Occurence:
100% (93.463/93.463)"
              }
            },
            "required properties": [
              "$date"
            ]
          },
          "grade": {
            "type": "String",
            "description": "Document Occurence: 100% (24.621/24.621); Member Occurence: 100%
(93.463/93.463)"
          },
          "score": {
            "type": [
              "Number",
              "JsonNull"
            ],
            "description": "Document Occurence: 100% (24.621/24.621); Member Occurence: 100%
(93.463/93.463)"
          }
        },
        "required properties": [
          "date",
          "grade",
          "score"
        ]
      }
    },
```

```
    "name": {
      "type": "String",
      "description": "Occurence: 100% (25.359/25.359)"
    },
    "restaurant_id": {
      "type": "String",
      "description": "Occurence: 100% (25.359/25.359)"
    }
  },
  "required properties": [
    "_id",
    "address",
    "borough",
    "cuisine",
    "grades",
    "name",
    "restaurant_id"
  ]
}
```

Vollständiges Schema des Testdatensatz_2


```
{
  "title": "Testdatensatz_2",
  "description": "Json Schema for collection \"Testdatensatz_2\" of database \"testdaten\",
created on 2016-03-16 11:55:32",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "properties": {
    "_id": {
      "type": "JsonObject",
      "description": "Occurence: 100% (2.595.243/2.595.243)",
      "properties": {
        "$oid": {
          "type": "String",
          "description": "Occurence: 100% (2.595.243/2.595.243)"
        }
      },
      "required properties": [
        "$oid"
      ]
    },
    "date": {
      "type": "JsonObject",
      "description": "Occurence: 99,77% (2.589.322/2.595.243)",
      "properties": {
        "$date": {
          "type": "Number",
          "description": "Occurence: 100% (2.589.322/2.589.322)"
        }
      },
      "required properties": [
        "$date"
      ]
    },
    "swap": {
      "type": "Number",
      "description": "Occurence: 100% (2.595.243/2.595.243)"
    },
    "timestamp": {
      "type": [
        "Number",
        "JsonObject"
      ],
      "description": "Occurence: 100% (2.595.243/2.595.243)",
      "properties": {
        "$numberLong": {
          "type": "String",
          "description": "Occurence: 4,1% (106.398/2.595.243)"
        }
      }
    },
    "total": {
      "type": "Number",
      "description": "Occurence: 100% (2.595.243/2.595.243)"
    },
    "used": {
      "type": "Number",
      "description": "Occurence: 100% (2.595.243/2.595.243)"
    },
```

```
    "time": {
      "type": "JsonObject",
      "description": "Occurence: 0,23% (5.883/2.595.243)",
      "properties": {
        "$date": {
          "type": "Number",
          "description": "Occurence: 100% (5.883/5.883)"
        }
      },
      "required properties": [
        "$date"
      ]
    }
  },
  "required properties": [
    "_id",
    "swap",
    "timestamp",
    "total",
    "used"
  ]
}
```

Schema-Aktualisierung mit Hilfe geänderter Dokumente (Testdatensatz_1 + Testdatensatz_2)


```
{
  "title": "Merged Collections: Testdatensatz_2, Testdatensatz_1",
  "description": "Json Schema for collection \"Merged Collections: Testdatensatz_2,
Testdatensatz_1\" of database \"testdaten\", created on 2016-03-16 12:08:35",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "properties": {
    "_id": {
      "type": "JsonObject",
      "description": "Occurence: 100% (2.620.602/2.620.602)",
      "properties": {
        "$oid": {
          "type": "String",
          "description": "Occurence: 100% (2.620.602/2.620.602)"
        }
      },
      "required properties": [
        "$oid"
      ]
    },
    "date": {
      "type": "JsonObject",
      "description": "Occurence: 98,81% (2.589.322/2.620.602)",
      "properties": {
        "$date": {
          "type": "Number",
          "description": "Occurence: 100% (2.589.322/2.589.322)"
        }
      },
      "required properties": [
        "$date"
      ]
    },
    "swap": {
      "type": "Number",
      "description": "Occurence: 99,03% (2.595.243/2.620.602)"
    },
    "timestamp": {
      "type": [
        "Number",
        "JsonObject"
      ],
      "description": "Occurence: 99,03% (2.595.243/2.620.602)",
      "properties": {
        "$numberLong": {
          "type": "String",
          "description": "Occurence: 4,1% (106.398/2.595.243)"
        }
      }
    },
    "total": {
      "type": "Number",
      "description": "Occurence: 99,03% (2.595.243/2.620.602)"
    },
    "used": {
      "type": "Number",
      "description": "Occurence: 99,03% (2.595.243/2.620.602)"
    },
```

```
    "time": {
      "type": "JsonObject",
      "description": "Occurence: 0,22% (5.883/2.620.602)",
      "properties": {

        "$date": {
          "type": "Number",
          "description": "Occurence: 100% (5.883/5.883)"
        }
      },
      "required properties": [
        "$date"
      ]
    },
    "address": {
      "type": "JsonObject",
      "description": "Occurence: 0,97% (25.359/2.620.602)",
      "properties": {
        "building": {
          "type": "String",
          "description": "Occurence: 100% (25.359/25.359)"
        },
        "coord": {
          "type": "JsonArray",
          "description": "Occurence: 100% (25.359/25.359)",
          "array order": {
            "Number": "Min occurence: 0; Max occurence: 2"
          },
          "items": {
            "type": "Number",
            "description": "Document Occurence: 99,99% (25.357/25.359); Member Occurence:
199,98% (50.714/25.359)"
          }
        },
        "street": {
          "type": "String",
          "description": "Occurence: 100% (25.359/25.359)"
        },
        "zipcode": {
          "type": "String",
          "description": "Occurence: 100% (25.359/25.359)"
        },
        "date": {
          "type": "String",
          "description": "Occurence: 0% (1/25.359)"
        }
      },
      "required properties": [
        "building",
        "coord",
        "street",
        "zipcode"
      ]
    },
    "borough": {
      "type": "String",
      "description": "Occurence: 0,97% (25.359/2.620.602)"
    },
    "cuisine": {
      "type": "String",
      "description": "Occurence: 0,97% (25.359/2.620.602)"
```

```
    },
    "grades": {
      "type": "JsonArray",
      "description": "Occurence: 0,97% (25.359/2.620.602)",
      "array order": {
        "JsonObject": "Min occurence: 0; Max occurence: 9"
      },
      "items": {
        "type": "JsonObject",
        "description": "Document Occurence: 97,09% (24.621/25.359); Member Occurence: 368,56%
(93.463/25.359)",
        "properties": {
          "date": {
            "type": "JsonObject",
            "description": "Document Occurence: 100% (24.621/24.621); Member Occurence: 100%
(93.463/93.463)",
            "properties": {
              "$date": {
                "type": "Number",
                "description": "Document Occurence: 100% (24.621/24.621); Member Occurence:
100% (93.463/93.463)"
              }
            },
            "required properties": [
              "$date"
            ]
          },
          "grade": {
            "type": "String",
            "description": "Document Occurence: 100% (24.621/24.621); Member Occurence: 100%
(93.463/93.463)"
          },
          "score": {
            "type": [
              "Number",
              "JsonNull"
            ],
            "description": "Document Occurence: 100% (24.621/24.621); Member Occurence: 100%
(93.463/93.463)"
          }
        },
        "required properties": [
          "date",
          "grade",
          "score"
        ]
      }
    },
    "name": {
      "type": "String",
      "description": "Occurence: 0,97% (25.359/2.620.602)"
    },
    "restaurant_id": {
      "type": "String",
      "description": "Occurence: 0,97% (25.359/2.620.602)"
    }
  },
  "required properties": [
    "_id"
  ]
}
```