Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

FAKULTÄT
FÜR MATHEMATIK, INFORMATIK
UND NATURWISSENSCHAFTEN

Inkrementelle Auswertung von MongoDB-Volltextsuchanfragen

Randy Schütt

Randy.Schuett@informatik.uni-hamburg.de
Studiengang Informatik
Matrikelnummer 6209096

Erstgutachter:	Prof. Dr. -Ing Norbert Ritter
Zweitgutachter:	Dr. Dirk Bade
Betreuer:	Wolfram Wingerath

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit behandelt zwei ineinander übergehende und doch separate Themengebiete. Zum einen wird das Thema *Volltextsuche* behandelt. Volltextsuchen gewinnen durch die immensen weltweit aufkommenden Datenmengen (Abb. 1.1) zunehmend an Relevanz[MN]. Nur durch eine Volltextsuchanfrage können ansonsten unüberschaubare Datenmengen effektiv nach bestimmten Begrifflichkeiten durchsucht werden[MN]. Auf Grund der enormen Datenmengen und dem Umstand, dass diese häufig in unstrukturierter und textueller Form vorliegen, wäre eine manuelle Suche nicht effizient möglich[MN]. Die Handhabung und Benutzung von Volltextsuchen beschränkt sich mittlerweile nicht nur auf reines Data-Mining. Inzwischen sind Volltextsuchen auch für den regulären Bedarf von End-Kunden im E-Commerce eine Notwendigkeit geworden. In beinahe jedem E-Commerce System wird die Suchanfrage textuell getätigt. Nur wenn die einzelnen Suchbegriffe (partiell) im Namen des Produkts und/oder in der Beschreibung enthalten sind, ist das Produkt relevant bezüglich der Suchanfrage des Kunden.
Das zweite Thema dieser Arbeit behandelt Echtzeitdatenbanken und Echtzeit-Anfragen. Der Übergang beider Themengebiete wird die inkrementelle Auswertung von MongoDB-Volltextsuchanfragen im Rahmen der von der Universität Hamburg stammenden Projekte *InvaliDB*[WW17] und *Orestes*[FG17b] sein.

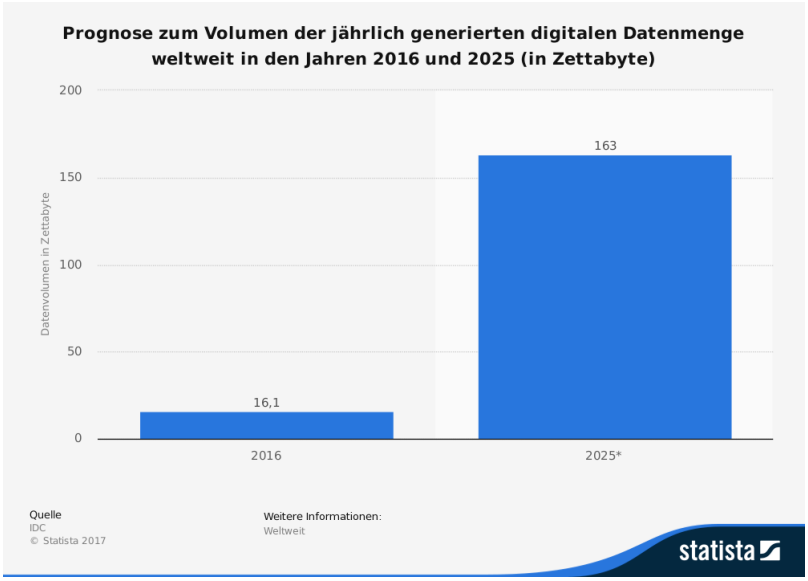


Abbildung 1.1: Prognose zum Volumen der jährlich generierten digitalen Datenmenge weltweit in den Jahren 2016 und 2025 (in Zettabyte) [Sta]

1.1 Motivation

Bevor Netscape im Jahr 1995 Javascript entwickelte, bestand das Web aus statischen Inhalten. Mit dem Aufkommen von Javascript gab es erstmals die Möglichkeit, Inhalte dynamisch nachzuladen, zu manipulieren und zu generieren. Interaktionen mit und von Benutzern sind seitdem ohne erneutes Laden der gesamten Webseite möglich, indem nur partielle Inhalte nachgeladen oder verändert werden. Ab 1998 ermöglichte das Aufkommen von *XMLHttpRequest*[W3S17] Technologien auch asynchrones Nachladen. Spätestens seit dem Aufsatz *A new Approach to Web Applications*[Gar05] von Jesse James Garret im Jahr 2005 wurde diese Technologie unter dem Begriff *Ajax* bekannt. Mit JQuery (Erscheinungsjahr 2006), AngularJS (Erscheinungsjahr 2009) und React (Erscheinungsjahr 2013) entwickeln auch größere Firmen Frameworks, um den Umgang mit dynamischen Inhalten zu vereinfachen. Durch geschicktes Nachladen partieller Inhalte wird versucht, den notwendigen Datendurchsatz auf ein Minimum zu reduzieren. Darstellung, Aktualisierung und Interaktion von Datenbeständen werden dadurch in beinahe Echtzeit möglich.
Für die Aktualisierungen der Daten ist i.d.R. allerdings nach wie vor die Applikation zuständig. Der Datenbestand in der Datenbank muss in bestimmten - für Echtzeit möglichst geringen - Abständen angefragt werden, da die Möglichkeit besteht, dass Änderungen seit dem letzten Abruf erfolgt sind. Die Applikation verfügt zu dem Zeitpunkt der Abfrage nicht über das Wissen, ob sich tatsächlich Daten in der vergangenen Zeitspanne geändert haben oder neue hinzugekommen sind. Das Wissen darüber besitzt einzig und allein die Datenbank. Ein besseres Vorgehen wäre, dass die Applikation die Datenbank darüber in Kenntnis setzt, welche Daten für die momentane Darstellung relevant sind. Die Datenbank ermittelt anschließend bei eingehenden Änderungen, ob diese den angegebenen Kriterien der Applikation entsprechen. Ist dies der Fall, wird die Applikation darüber informiert und der aktualisierte Datenbestand übermittelt.

Der Fachbereich VSIS der Universität Hamburg entwickelt derzeit Konzept und Umsetzung von sogenannten *Real-Time Queries*[Baq17]. *Real-Time Queries* teilen sich in zwei Kategorien: Zum einen gibt es *Self-Maintaining Queries*. Bei eingehenden Schreib-Operationen, die den angegebenen Kriterien entsprechen, wird die Ergebnismenge umgehend aktualisiert und an die Applikation zurückgegeben. Im Gegensatz dazu wird die Applikation bei *Event Stream Queries* über betreffende Schreibvorgänge in Form eines Events informiert. Mittels einer Callback-Funktion kann die Applikation anschließend entscheiden, welche Reaktion auf das Event folgt.

1.2 Struktur der Arbeit

Nach der einführenden Zielsetzung beginnt diese Arbeit mit einem kurzen Überblick über Echtzeitdatenbanken. Das Kapitel „Anwendungskontext: InvaliDB & Orestes“ (Sie-

he Kapitel 2) beschreibt die beiden Komponenten zur Realisierung der Echtzeitdatenbank des Baqend[1]-Systems, *InvaliDB*[WW17] und *Orestes*[FG17b] sowie deren Interaktion. Nach einem kurzen Einblick in den allgemeinen Kontext von Volltextsuchen in dem gleichnamigen Kapitel „Volltextsuche“ folgt eine detaillierte Beschreibung der Volltextsuche in MongoDB[Monh] im Kapitel „Volltextsuche in MongoDB“. Dies ist von besonderer Bedeutung, da die für diese Arbeit implementierte Volltextsuche sich syntaktisch und semantisch an die Volltextsuche von MongoDB anlehnt. Der Schwerpunkt der Beschreibung liegt dabei auf der Funktionsweise der Volltextsuche und der Wirkungsweise der verschiedenen Modi. Neben MongoDB existieren verschiedene andere und größere Systeme, die eine Volltextsuche über eine Vielzahl Dokumente ermöglichen. Um eine Übersicht der Vor- und Nachteile dieser Systeme darzustellen, wird nach der MongoDB-Volltextsuche ein Vergleich zwischen dieser, SolR[Fouo] und Elasticsearch[BVd] im Kapitel „Gegenüberstellung und Vergleich anderer Volltextsuchen“ durchgeführt. Dies ist besonders interessant, da die in dieser Arbeit implementierte Volltextsuche Apache-Lucene [Fouc] verwendet - genau wie SolR und Elasticsearch. Der Abschluss jeder Gegenüberstellung wird eine kritische Auseinandersetzung, inwiefern die betrachtete Volltextsuche eine Alternative zu MongoDB im Kontext von InvaliDB darstellen könnte. Im Kapitel „Implementation der Volltextsuchanfrage in InvaliDB“ wird wird auf die eigentliche Implementierung und Entwicklung der Volltextsuche eingegangen. Die einzelnen Komponenten der Architektur werden ausführlich erklärt und deren Interaktion anhand eines Beispiels demonstriert. Das Kapitel „Evaluation der Volltextsuchanfrage“ bildet den Abschluss dieser Arbeit. Die Volltextsuche wird anhand einer *Proof of Concept*-Applikation evaluiert. Abschließend wird überprüft, ob die Performance von InvaliDB durch den Einsatz der Volltextsuche negativ beeinflusst wird.

1.3 Zielsetzung

Das Ziel dieser Arbeit ist die Konzeption und Implementierung einer skalierbaren Volltextsuche in InvaliDB, die für den Einsatz von *Realtime*-Queries geeignet ist. Während reguläre Volltextsuchen einmalig ein Suchergebnis zurück liefern, soll die zu implementierende Volltextsuche imstande sein, eine inkrementelle Auswertung der Suchanfrage vornehmen zu können. Daraus folgt, dass sobald neue und passende Daten vorliegen, diese direkt als Suchergebnis identifiziert werden. Allerdings resultiert daraus nicht, dass die Suchergebnisse stetig anwachsen. Es kann ebenso der Fall eintreten, dass passende Daten gelöscht und somit im nächsten Suchzyklus nicht mehr als Suchergebnis aufgelistet werden. Idealerweise soll sich die resultierende Volltextsuche sowohl syntaktisch als auch semantisch äquivalent zur MongoDB Volltextsuche verhalten.
Zunächst gilt es die notwendigen Technologien, die für die MongoDB-Volltextsuche verwendet werden, zu identifizieren und Alternativen für die Implementierung in InvaliDB

[1] https://www.baqend.com/

zu finden. Dies stellt eine Herausforderung dar, da InvaliDB eine reine Java Implementierung ist, während MongoDB in C und C++ implementiert wurde. Anschließend müssen die Schritte zur Auswertung einer Volltextsuchanfrage analysiert und entsprechend adaptiert werden. Nach der Implementation wird eine Evaluation folgen, die sicherstellt, dass die Volltextsuche funktioniert und die Performance von InvaliDB nicht negativ beeinflusst.

2 Anwendungskontext: InvaliDB & Orestes

In den letzten Jahren ist eine neue Form der Datenbank-Interaktion als Komplement zu den bisherigen pull-basierten Datenbanksystemen wie MySQL oder Postgre in Erscheinung getreten: push-basierte Datenbanksysteme. Bei pull-basierten Datenbanksystemen ist der Benutzer für einen Abruf der Datensätze verantwortlich. Werden neuen Datensätze benötigt, weil z.B. der Verdacht besteht, dass neue oder geänderte Datensätze vorliegen, muss manuell eine Datenbankanfrage getätigt werden. Liegen keine neuen Datensätze vor, wird die ursprüngliche Ergebnismenge mit einer neuen aber äquivalenten Ergebnismenge überschrieben. Je nach Datenbankzugriff und der Menge an zu durchsuchenden und resultierenden Daten kann dies einen erheblichen Ressourcenaufwand darstellen. Bei push-basierten Datenbank-Systemen hingegen registriert der Benutzer einmalig einen Query und erhält fortan neue Ergebnismengen oder Benachrichtigungen, sobald sich der Datenbestand betreffend des registrierten Queries ändert. Da sich somit die Ergebnismenge des Benutzers stets in einem synchronen Zustand mit der Datenbank befindet, werden derartige Datenbanksysteme als *Echtzeitdatenbanken* bezeichnet. In Abbildung 2.1 ist eine Einordnung verschiedener Datenbanksysteme zu sehen.

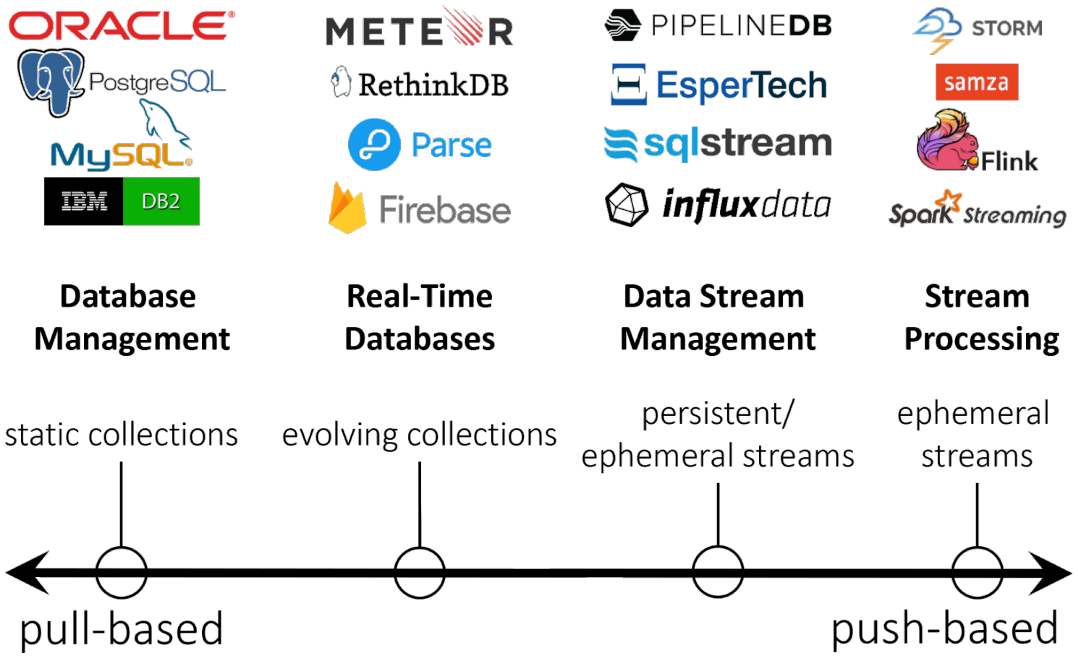


Abbildung 2.1: Einordnung von pull- und push-basierten Datenbanksystemen [Win17]

Echtzeitdatenbanken müssen in der Lage sein, jede eintreffende Schreiboperation für alle registrierten Queries unverzüglich auszuwerten. Ein minimalistisches Beispiel zum besseren Verständnis [Win17]: Angenommen es gibt eine Applikation mit 1000 gleichzeitigen Benutzern und jeder Benutzer hat eine Echtzeitabfrage registriert - eine sogenannte *Query Subscription*. Außerdem gibt es durchschnittlich 1000 Schreiboperationen pro Sekunde. Obwohl jeder Benutzer nur eine aktive Echtzeitabfrage hat, muss die Echtzeitdatenbank bereits eine Million Überprüfungen durchführen - pro Sekunde. Ist das System somit nicht in der Lage, sowohl für eine wachsende Anzahl an Schreiboperationen als auch für eine wachsende Anzahl an zu verwaltender Queries zu skalieren, existiert an dieser Stelle ein Flaschenhals für die gesamte Applikation. Zusätzlich sollte das System zur Verwaltung der Realtime-Queries und das System zur Datenbank-Interaktion autonom funktionieren und austauschbar sein. Sollte z.B. durch eine Überlast das System zur Verwaltung der Realtime-Queries ausfallen, würde dies andernfalls den Ausfall der gesamten Applikation bedeuten. Sind hingegen beide Systeme autonom, können auch in diesem Szenario weiterhin pull-basierte Anfragen vorgenommen werden, bis sich das System zur Verwaltung der Realtime-Queries erholt hat oder ausgetauscht wurde.
Der Fachbereich VSIS der Universität Hamburg und die daraus entsprungene Firma Baqend entwickeln derzeit ein System für Echtzeitdatenbanken und Realtime-Queries. Die innerhalb des Baqend-Systems verwendeten und notwendigen Komponenten heißen InvaliDB und Orestes.

2.1 InvaliDB

InvaliDB[WW17] ist die Kurzschreibweise für **Invali**date **D**ata**b**ase & Queries. Die Namensherkunft ergibt sich aus dem Umstand, dass in bestimmten Situationen die Notwendigkeit besteht, gespeicherte Query-Anfragen oder Ergebnisse zu *invalidieren*.
InvaliDB verwendet den verteilten Streamingprozessor *Apache Storm*[Foud] und sorgt für eine Partitionierung der verwalteten Queries und der eingehenden Schreiboperationen. Das System kann durch hinzufügen weiterer Partitionen sowohl mit Lese- als auch Schreiblast skalieren (Abb. 2.2), wodurch auch anspruchsvolle *Workloads* bewältigt werden können. Muss das System mehr Schreiboperationen bewältigen, werden weitere Objekt-Partitionen hinzugefügt. Müssen mehr Queries verwaltet werden, werden weitere Query-Partitionen hinzugefügt. Das hinzufügen oder entfernen beliebiger Partitionen beider Kategorien ist vollkommen unabhängig voneinander, womit maximale Flexibilität und Skalierbarkeit erreicht wird.
InvaliDB ist der Teil des Baqend-Systems, der die Verwendung von Realtime-Queries ermöglicht. Innerhalb der Quaestor-Architektur[FG17a] stellt InvaliDB die Streaming-Schicht dar und kann durch eine asynchrone und autonome Nachrichten-Schicht Informationen mit Orestes austauschen.

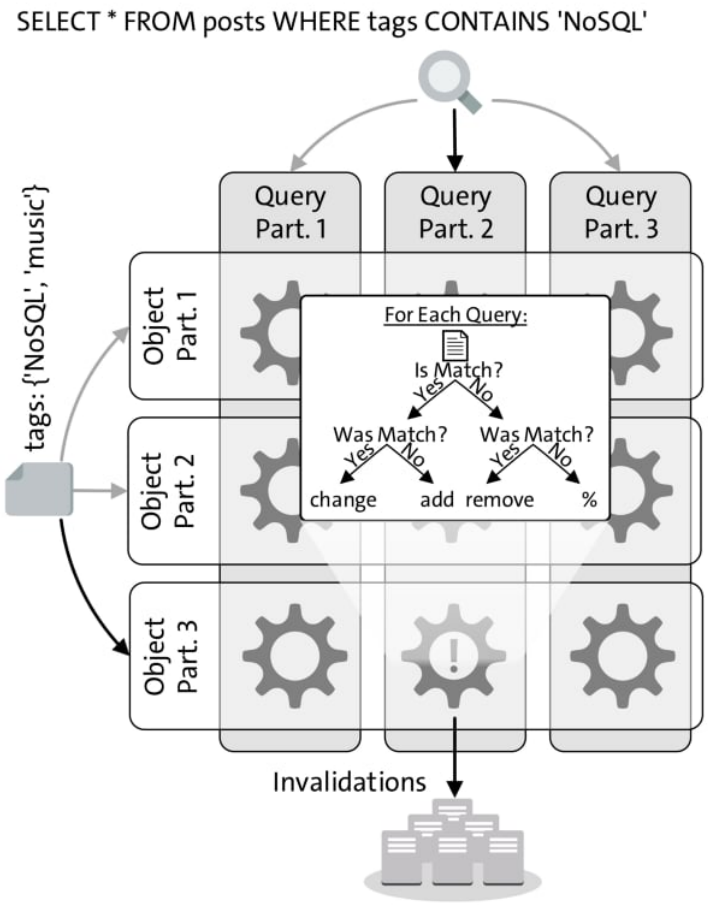


Abbildung 2.2: Query- und Objekt-Partitionierung in InvaliDB [Win17]

2.2 Orestes

Orestes (**O**bjects **REST**fully **e**ncapsulated in **s**tandard formats)[FG14] ist ein skalierbares Database- und Backend-as-a-Service System. Orestes wird wie folgt beschrieben[FG17b]:

> „Orestes ist eine skalierbares Database- und Backend-as-a-Service System. Es reichert NoSQL-Datenbanksysteme mit funktionalen und nicht-funktionalen Eigenschaften an. Durch automatisierte polyglotte Persistenz werden Anfragen an ein zu der Anwendung passendes Datenbanksystem weitergeleitet. Durch kohärentes Web-Caching mit Cache Sketches werden Datenbank-Objekte automatisch gecacht und durch ein Bloom filter-basiertes Cache-Kohärenz-Protokoll aktuell gehalten, um die Vorteile von geringer Latenz und hoher Skalierbarkeit mit definierbaren Konsistenz-Kriterien zu vereinen. Scalable Cache-Aware Optimistic Transactions bieten skalierbare Transaktionen für unveränderte NoSQL-Systeme. Des weiteren werden systemübergreifende Backend- und Datenbank Abstraktionen geschaffen, u.a. feingranulare Zugriffskontrolle durch ein Authentifizierungs- und Autorisierungenssystem, ein reichhaltiges objektorientiertes Schema-System, eine Infrastruktur für JavaScript-basierte

serverseitige Stored Procedures und Trigger, sowie Autoskalierungsstrategien für Caching, Replikation und Sharding.“

Innerhalb der Quaestor-Architektur[FG17a] repräsentiert Orestes die Daten-Schicht, welche direkt mit der Datenbank interagiert und damit die Voraussetzungen schafft, pull-basierte Datenbank-Zugriffe vorzunehmen. Die Verwendung von Realtime-Queries wird erst in Verbindung mit InvaliDB ermöglicht, welches die registrierten Queries verwaltet und ankommende Schreiboperationen auswertet. Zum Austausch etwaiger Informationen dient eine asynchrone und autonome Nachrichten-Schicht.

2.3 Funktionsweise und Interaktion

Eingehende Schreiboperationen werden von Orestes entgegen genommen und an das gekapselte Datenbanksystem delegiert. Anschließend wird das daraus resultierende Abbild - auch *After Image* genannt - von Orestes an InvaliDB in komprimierter Form per Nachrichten-Schicht weitergegeben. Das übermittelte Abbild wird partitionsweise für alle registrierten Queries auf Übereinstimmungen - sogenannte *matches* - untersucht. Die Überprüfung kann als Binärbaum betrachtet werden, der an seinen vier äußersten *Blättern* verschiedene Event-Typen besitzt (Abb. 2.3).

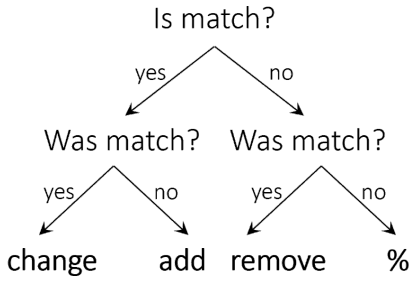


Abbildung 2.3: *match* Auswertung innerhalb von InvaliDB [Win17]

Für jeden verwalteten Query wird anschließend geprüft, ob ein *match* vorliegt. Sofern dies zutrifft, wird überprüft, ob es sich auch zuvor bereits um ein *match* handelte. Ist dies der Fall, resultiert die Untersuchung im *Change*-Blatt: ein Element der nicht leeren Ergebnismenge hat sich verändert. Lag zuvor kein *match* vor, resultiert der Abgleich im *add*-Blatt: die vorliegende Ergebnismenge wird um ein Element ergänzt. Gibt es keine Übereinstimmung mit dem Query und gab es davor auch keine, passiert nichts. Existierten für den Query allerdings vormals Übereinstimmungen, endet die Untersuchung im *remove*-Blatt: das Element wird aus der nicht leeren Ergebnismenge entfernt.
Je nach Verwendungsart wird die verwaltete Ergebnismenge anhand der resultierenden Event-Art modifiziert und an den Klient zurückgegeben, oder der Klient wird anhand eines passenden Events über die Änderungen informiert.

3 Volltextsuche

Eine Volltextsuche ist die Suche und Lokalisierung von Sachverhalten in einer Ansammlung von Dokumenten. Die Angabe des physischen Speicherorts der Dokumente ist dabei i.d.R. nicht relevant, ebenso wenig die Struktur der Dokumente. Generell ist für eine Volltextsuche nur relevant, dass die Dokumente ausgelesen und durchsucht werden können. Eine Volltextsuche dient damit dem Auffinden und der Extraktion von Informationen aus einer großen Anzahl von zumeist unstrukturierten Daten, die manuell nicht (mehr) effizient durchsucht werden können. Volltextsuchen werden besonders in einem Teilbereich des *Data-Mining*, dem sogenannten *Text-Mining*[CB09][Cie05], eingesetzt und wird als *Information Retrieval* bezeichnet[MN].

3.1 Stopwörter

Natürlichsprachliche Texte weisen i.d.R. Worte auf, die zwar eine semantische Aufgabe erfüllen, jedoch für eine Volltextsuche irrelevant sind. Diese als *Stopwörter* bezeichneten Worte sind z.B. Artikel oder Verbundworte und werden generell bei einer Überprüfung der Signifikanz eines Textes gefiltert. Zur Durchführung der Filterung müssen die einzelnen Worte jedoch auch als einzelne Einheiten erkannt werden, da ansonsten keine Differenzierung möglich ist. Dazu findet zunächst eine lexikalische Analyse statt.

3.2 Lexikalische Analyse

Eine lexikalische Analyse findet häufig im Zusammenhang mit einem *Compiler* oder einem *Interpreter* Verwendung. Die Komponente, die die lexikalische Analyse durchführt, wird als *Lexer* oder *Tokenizer* bezeichnet.
Bei einer lexikalischen Analyse wird ein vorliegender Text in lexikalisch zusammengehörige Einheiten, sogenannte *Token*, zerlegt. Ein Token ist eine logisch zusammengehörende Entität innerhalb eines Kontextes die nicht weiter zerteilt werden kann. Für einen natürlichsprachlichen Text verkörpern die einzelnen Worte und Satzzeichen die Token. Ein Token hat i.d.R. einen Typ der spezifiziert um welche Token-Art es sich handelt (z.B. um ein Wort, ein Satzzeichen oder eine Zahl) und optional eine textuelle Repräsentation (z.B. das eigentliche Wort). Des Weiteren ist auch die Angabe des Start- und End-Offsets des Tokens innerhalb des Textes üblich. Darüber kann u.a. die Länge des Token berechnet oder nach Manipulationen am Token der ursprüngliche Begriff ermittelt werden. Ein

Beispiel[1] zur Demonstration:
Angenommen, es liegt der folgende Text vor:

```
I am your father, Luke!
```

Dann würden sich daraus die folgenden Token ergeben:

Token-Art	Textuelle Repräsentation
Wort	*I*
Wort	*am*
Wort	*your*
Wort	*father*
Satzzeichen	*,*
Wort	*Luke*
Satzzeichen	*!*

3.3 Indexierung

Um eine Volltextsuche performant zu gestalten und die erneute Verarbeitung der Dokumente bei jeder eingehenden Volltextsuche zu vermeiden, werden die vorliegenden Dokumente für gewöhnlich indexiert. Dies geschieht i.d.R. einmalig und parallel zur Persistierung des Dokuments. Eine Indexierung stellt eine *Verschlagwortung* des Dokuments dar, indem aus dem vorliegenden Dokument Informationen extrahiert und mittels Deskriptoren (oder *Schlagwörtern*) zugeordnet werden. Durch diese Zuordnung kann der im Dokument vorhandene Sachverhalt anhand des Schlagwortes auch nachträglich erkannt werden, ohne das Dokument erneut zu verarbeiten. Im Gegensatz zu Stichworten sind Schlagworte eine definierte Obermenge, in der zuvor festgelegt wurde, welche Worte oder Phrasen einem Stichwort zugeordnet werden. Zum Beispiel könnte ein Dokument, dass über Abholzung des Regenwaldes handelt, die Schlagworte *Waldsterben*, *Regenwald*, *Rodung* und ggf. *Klimawandel* enthalten. Wird später nach einem dieser Begriffe gesucht, wird dieses Dokument als passendes Suchergebnis identifiziert. Die Verwendung von Indizes ermöglicht i.d.R. einen erheblichen Geschwindigkeitsvorteil bei der Durchsuchung von Datenbeständen. Durch die extrahierten und normalisierten Schlagworte kann die Menge der zu durchsuchenden Daten reduziert und das passende Dokument schneller und ohne erneute Verarbeitung des Dokuments zuverlässig identifiziert werden.

3.4 Stemming

Der Begriff *Stemming* bezeichnet die Rückführung eines Wortes auf den ursprünglichen Wortstamm. In natürlichen Sprachen existieren verschiedene Steigerungs- und gramma-

[1] Beispiel entnommen aus [Trea]

tische Zeitformen, die Wörter beeinflussen können. Bei einer stattfindenden Volltextsuche können sich daher die Suchbegriffe und die vorliegenden Worte im Text signifikant unterscheiden, obwohl der gleiche Wortstamm zugrunde liegt. Zum Beispiel haben die Worte *looking*, *looked* und *looks* denselben Wortstamm *look*. Um diese Differenz zu beseitigen, wird die sogenannte *Stemming*-Prozedur angewendet. Mittels *Stemming* können die Wörter auf einen einheitlichen Wortstamm zurückgeführt werden. Vor der eigentlichen *Stemming*-Prozedur ist es erforderlich, dass die verwendete Sprache des Textes bereits bestimmt wurde. Andernfalls können spezifische linguistische Charakteristika einer Sprache nicht ordnungsgemäß behandelt werden. Beispiele für linguistische Charakteristika einer Sprache sind z.B. die Umlaute der deutschen Sprache oder andere diakritische Symbole (Siehe Abschnitt 4.7.3).
In Abbildung 3.1 können die Schritte der lexikalischen Analyse (das *Tokenizen*), die Filterung von Stopwörtern und das Stemming anhand eines Beispiels grafisch betrachtet werden.

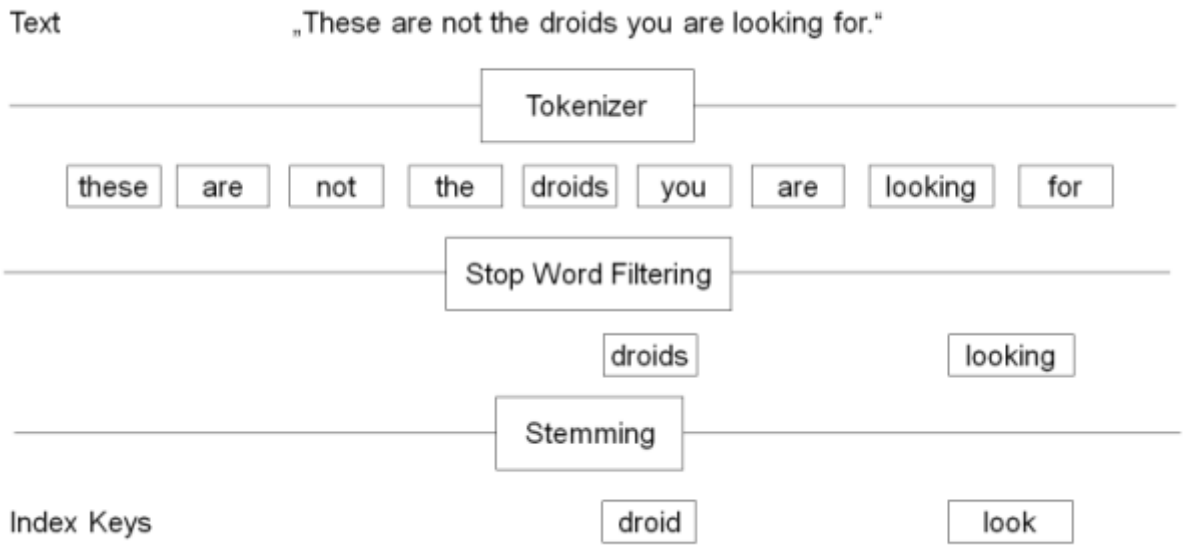


Abbildung 3.1: Filterung von Stopwörtern und Stemming [Trea]

4 Volltextsuche in MongoDB

MongoDB unterstützt Volltextsuchanfragen seit Version 2.4[Mon13]. In diesem Kapitel wird der Ablauf und die Funktionsweise der Volltextsuche von MongoDB detailliert erklärt und an Beispielen demonstriert.

4.1 Text-Indizes

Text-Indizes[Monj] sind eine notwendige Angabe, ohne die eine Volltextsuche nicht stattfinden kann. Für jedes zu durchsuchende Dokument werden jeweils nur die Felder untersucht, für die zuvor ein Text-Index angegeben wurde. Es mag jedoch die Frage aufkommen, warum stattdessen nicht standardmäßig alle Felder durchsucht werden und die Angabe eines Text-Index nur als eine optionale Filtereinstellung dienen kann. Dies ist aus mehreren Gründen nicht zweckmäßig:

- Zum einen wäre ein solches Vorhaben mit einem erheblichen Mehraufwand verbunden. Sämtliche Felder des Dokuments müssten zunächst darauf untersucht werden, ob sie verwertbare Text-Inhalte aufweisen.
- Zum anderen würden auch Inhalte untersucht, die zwar eine textuelle Darstellung aufweisen, allerdings nicht relevant sind. Ein Beispiel dazu wären u.a. Datums-Formate. Text-Indizes sind eine Deklaration des Benutzers, der damit manuell angibt, welche Felder für eine Volltextsuche relevant sind.
- Der wohl wichtigste Grund ist, dass die Abwesenheit eines Text-Index eine effiziente Ausführung einer Volltextsuchanfrage unmöglich machen würde. MongoDB müsste in diesem Fall für jede eingehende Volltextsuchanfrage alle Dokumente der aktuell betrachteten Collection durchsuchen, um die relevanten Dokumente zu bestimmen. Indizes sind in MongoDB - ähnlich wie in relationalen Datenbanken - als B-Baum realisiert[Mond]. Die Implementierung und Funktionsweise von B-Bäumen als Index-Strukturen werden detailliert im Abschnitt 5.3.2 betrachtet.

MongoDB unterstützt allerdings die automatische Erstellung von Text-Indizes für alle Felder, die textuellen Inhalt aufweisen. Dazu werden *Wildcard Text Indexes*[Monw] verwendet:

```
db.collection.createIndex({"$**": "text"})
```

Wildcard Text Indexes werden z.B. eingesetzt, wenn die Struktur der eingehenden Dokumente nicht bekannt ist oder variiert.
Zu Demonstrationszwecken bietet es sich für den weiteren Verlauf dieses Kapitels an, ein übergreifendes und gemeinsames Beispiel zu haben: Angenommen es liegt ein Nachrichtensystem vor, welches einen Titel, eine Kategorie und einen Inhalt aufweist. Für dieses Nachrichtensystem soll mittels der Konsole ein einfacher Text-Index für das Feld *content*, welches den Inhalt der Nachricht repräsentiert, angelegt werden. Für die entsprechend benannte *Collection*[Moni] „news“ kann der Text-Index wie folgt erstellt werden:

```
db.news.createIndex({content: "text"})
```

Listing 4.1: Erstellen eines einfachen Text-Indizes in MongoDB für ein Feld

Die Deklaration *"text"* gibt an, dass es sich um einen Text-Index handelt. Diese Angabe ist nötig, da in MongoDB neben Text-Indizes noch diverse weitere Index-Arten existieren. Weitere Beispiele und Beschreibungen können der MongoDB-Dokumentation[Monj] entnommen werden.

Text-Index Version

Mit MongoDB 3.2[Mon15] wurde die 3. Version des Text-Index eingeführt[Mons]. Während die 1. (eingeführt mit MongoDB 2.4[Mon13]) und 2. Version (eingeführt mit MongoDB 2.6[Mon14] und Standard bis 3.0[Mons]) des Text-Index die Groß- und Kleinschreibung diakritischer Zeichen als unterschiedlich betrachteten, ist dies in der 3. Version nicht der Fall[Mons]. Die 3. Version ist damit standardmäßig sowohl *case insensitive*[Mona] als auch *diacritic insensitive*[Monb]. Mit MongoDB 3.2 wurden ebenfalls entsprechende Optionen zur Steuerung dieses Verhaltens eingeführt. Siehe dafür Abschnitt 4.7.2 und 4.7.3.

4.1.1 Zusammengesetzte Indizes

In MongoDB existiert die Limitierung, das pro *Collection*[Moni] maximal **ein** Text-Index existieren kann. Das heißt allerdings nicht, dass bei einer stattfindenden Volltextsuche nur ein Feld pro Dokument auf seine Inhalte untersucht werden kann. Text-Indizes können auch mehrere Felder umfassen. Um bei dem obigen Beispiel eines Nachrichtensystems zu bleiben, soll diesmal der Text-Index neben dem Feld *content* auch den Titel der Nachricht (Feld *title*) umfassen:

```
db.news.createIndex(
    {
        content: "text",
        title: "text"
    }
)
```

Listing 4.2: Erstellen eines Text-Index in MongoDB für multiple Felder

Wie ersichtlich wird, können beliebige Felder eines Dokuments als Text-Index zusammengefasst werden. Dies wird auch als *Compound Index* bezeichnet. Darüber hinaus kann ein Text-Index auch mit beliebigen anderen Index-Arten interagieren. Auch dazu ein Beispiel: Angenommen es existieren mehrere Nachrichten in verschiedenen Kategorien und es liegt eine Suchanfrage an das Nachrichtensystem vor, indem nach allen Nachrichten einer bestimmten Kategorie gesucht wird. Diese Anfrage könnte wie folgt aussehen:

```
db.news.find(
    {
        category: "Drinks",
        $text: {
            $search: "Tea"
        }
    }
)
```

Es sollen daher alle Nachrichten der Kategorie *Drinks* auf Übereinstimmungen mit dem Suchwort *Tea* untersucht werden. Um die Volltextsuche darauf zu beschränken, nur die Dokumente innerhalb einer bestimmten Kategorie zu durchsuchen, kann der folgende zusammengesetzte Text-Index erstellt werden:

```
db.news.createIndex(
    {
        category: 1,
        content: "text"
    }
)
```

Die Deklaration *category: 1* erstellt einen Aufsteigenden-Index für das Feld *category* (während die Angabe -1 einen Absteigenden Index erstellen würde) und die Deklaration *content: "text"* legt einen Text-Index für das Feld *content* an. Eine nun stattfindende Volltextsuche würde damit die zu durchsuchenden Dokumente auf die Dokumente mit der jeweiligen Kategorie begrenzen. Die obige Anfrage würde daher nur die Dokumente mit der Kategorie *Drinks* untersuchen.

4.1.2 Limitierungen

Ein zusammengesetzter Text-Index ist einigen Limitierungen unterworfen[Mont]:

1. Ein zusammengesetzter Text-Index kann nicht in Verbindung mit speziellen Index-Typen wie Multi-Key-Indizes oder Geospatiale-Indizes erstellt werden
2. Wenn der zusammengesetzte Text-Index vorangestellte Indizes enthält, darf das Abfrageprädikat auf den vorhergehenden Indizes nur eine Gleichheitsbedingung anwenden

3. Sortieroperationen können keine Sortierungsreihenfolge aus einem Text-Index extrahieren. Auch dann nicht, wenn ein zusammengesetzter Text-Index vorliegt. D.h. Sortieroperationen haben keinen Zugriff auf die Sortierung im Text-Index und können diese daher nicht verwenden. Es gibt allerdings die Möglichkeit anhand des *Text-Scores* (Siehe Abschnitt 4.6) absteigend zu sortieren[Mone]:

```
1 db.collection.find(<query>, {
2     score: { $meta: "textScore" }
3 }).sort({
4     score: { $meta: "textScore" }
5 })
```

   In der *sort* Funktion können - parallel zum *Text-Score* - weitere Felder, nach denen sortiert werden soll, angegeben werden.

4.1.3 Aufwand

Die Verwendung eines Text-Index hat im Vergleich zu anderen Index-Arten einen höheren Verwaltungsaufwand und bei einer ausgeführten Volltextsuche auch erhebliche Performance-Kosten. Um nur einige zu nennen:

1. Text-Indizes können sehr groß werden, da sie einen Index-Eintrag für jeden einmaligen, per Stemming hergeleiteten Wortstamm in jedem indexierten Feld für jedes untersuchte Dokument erhalten.

2. Der Aufbau eines Text-Index dauert länger als der Aufbau von geordneten skalaren Indizes, da jedes betreffende Feld wie in Punkt 1. beschrieben verarbeitet werden muss. Skalare Indizes können dagegen vergleichsweise schnell erstellt und geordnet werden.

3. Text-Indizes beeinflussen die eingehenden Schreiboperationen, da Punkt 1. für jede Schreiboperation stattfinden muss, um die Dokumente sachgemäß zu indexieren.

4.2 Angabe der verwendeten Sprache

Um im weiteren Verlauf der Volltextsuche ein korrektes *Stemming* und eine korrekte Eliminierung der Stopwörter der jeweiligen Sprache vollziehen zu können, muss die verwendete Sprache des Dokuments zunächst festgestellt werden. Zum einen kann die verwendete Sprache beim Erstellen des Text-Index mit dem Attribut *default_language* angegeben werden[Monn]:

```
db.news.createIndex(
   { content : "text" },
   { default_language: "de" }
```

```
)
```

Listing 4.3: Erstellen eines Text-Index in MongoDB mit Angabe der verwendeten Sprache (Deutsch)

Wird bei einer Suchanfrage keine Sprache spezifiziert, wird die in *default_language* angegebene Sprache verwendet[Monv]. Wurde *default_language* beim Erstellen des Text-Index nicht explizit definiert, wird Englisch verwendet[Mono]. Die Sprache einer Suchanfrage wird wie folgt angegeben[Monv]:

Listing 4.4: Angabe der Sprache (Deutsch) in einer MongoDB-Suchanfrage

```
db.news.find({
    $text: {
        $search: <string>,
        $language: "de"
    }
});
```

Zum anderen kann die Sprache auch in den Dokumenten selbst spezifiziert werden[Monn]. Im Text-Index existiert dazu im Attribut *language_override* ein Feldname, der die Standardsprache überschreibt, sofern dieses Feld im Dokument enthalten ist[Monp]. Standardmäßig heißt das Feld schlicht *language*:

```
{
    content: "Green Tea is tasty",
    category: "Drinks",
    language: "en"
}
```

Listing 4.5: Angabe der verwendeten Sprache (Englisch) im Dokument

MongoDB hat die folgende Präzedenz für die verwendete Sprache innerhalb von Dokumenten [Monq]:

1. Die lokale Spezifizierung der Sprache innerhalb eines Dokuments hat Vorrang vor der globalen Angabe.

2. Die lokale Spezifizierung der Sprache in eingebetteten Dokumenten hat Vorrang vor der Sprachspezifizierung des umfassenden Dokuments oder der globalen Angabe.

Kann innerhalb eines eingebetteten Dokuments keine Sprachspezifizierung gefunden werden, wird - sofern vorhanden - die Sprache des umschließenden Dokuments oder die Standardsprache des Text-Index verwendet[Monq]. Eine vollständige Übersicht der von MongoDB unterstützten Sprachen kann in der Dokumentation[Monu] betrachtet werden.

4.3 Stopwörter

Nach der erfolgreichen Identifizierung der Sprache eines Dokuments kann noch kein Abgleich zwischen den Inhalten der jeweiligen Text-Index-Felder und der Suchanfrage stattfinden. Zuvor müssen die *Stopwörter* des Textes entfernt werden. Dies dient einerseits dazu, die Menge der zu durchsuchenden Worte einzuschränken und andererseits nur die für die Volltextsuche relevanten Worte zu betrachten. Wie in Abschnitt 3.1 erwähnt, muss dazu zunächst die lexikalische Analyse durchgeführt werden. Der Tokenizer von MongoDB zerlegt zunächst den vorliegenden Text in separate Token und vollzieht anschließend eine Filterung, so dass irrelevante Token direkt entfernt werden. Als irrelevant wird alles betrachtet, was nicht als Wort identifiziert wurde. Zum besseren Verständnis wird das Beispiel des Abschnitts 3.2 betrachtet[Trea]:
Angenommen, es liegt der folgende Text vor:

```
I am your father, Luke!
```

Dann würden sich daraus die folgenden Token ergeben:

Token-Art	**Textuelle Repräsentation**
Wort	*I*
Wort	*am*
Wort	*your*
Wort	*father*
Satzzeichen	*,*
Wort	*Luke*
Satzzeichen	*!*

Der Tokenizer von MongoDB wird alle Token, die nicht der Token-Art *Wort* angehören, filtern.

4.3.1 Ersetzung von Stopwörtern

Um einen Abgleich mit den Stopwörtern einer bestimmten Sprache vorzunehmen gibt es verschiedene Wege. MongoDB hält für jede unterstützte Sprache sämtliche Stopwörter in einer Text-Datei. In dieser Text-Datei stehen die Stopwörter einzeln untereinander. Bei einem Abgleich der Stopwörter wird die entsprechende Text-Datei der verwendeten Sprache ausgelesen und alle darin befindlichen Stopwörter werden aus dem vorliegenden Text entfernt. Die Stopwörter werden für den weiteren Verlauf i.d.R. *gecached*, so dass kein erneutes Auslesen der Datei notwendig ist. Groß- und Kleinschreibung wird bei der Ersetzung der Stopwörter nicht berücksichtigt. Die Stopwörter liegen stets in klein geschriebener Form vor und der gesamte Text muss daher vor der Ersetzung in die Kleinschreibweise überführt werden. Die Entfernung der Stopwörter reduziert die Anzahl der

zu durchsuchenden Worte und wird sowohl auf den textuellen Inhalt des Dokument-Feldes als auch auf die eigentliche Suchanfrage angewendet. Dieser Filterungsschritt ist ohne den zuvor stattfindenden Schritt der Identifizierung der verwendeten Sprache des Dokuments nicht möglich. Zur Demonstration dient erneut das Beispiel des Abschnitts 3.2[Trea]:
Angenommen, es liegt der folgende Text vor:

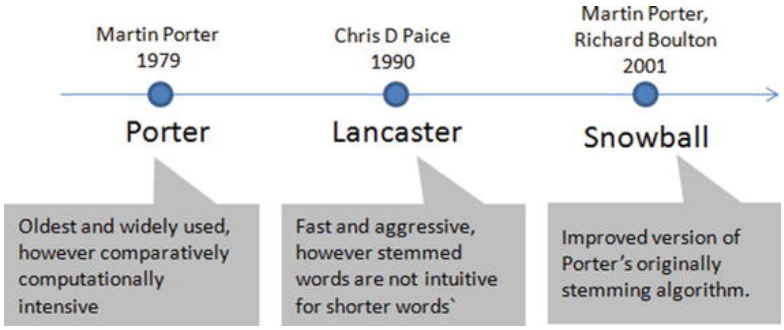
```
I am your father, Luke!
```

Nach der Bestimmung der Sprache (in diesem Fall Englisch), der lexikalischen Analyse und der anschließend stattfindenden Filterung von Stopwörtern würde der obige Text nur noch aus den Wörtern *father* und *luke* bestehen. Die Stopwörter *I*, *am* und *your* sowie die Satzzeichen wurden als irrelevant identifiziert und entfernt. Auf Grund des Stopwörter-Abgleichs ist das Wort *luke* nun kleingeschrieben.

4.4 Stemming: Porter-Stemmer

MongoDB verwendet für die Stemming-Prozedur den weit verbreiteten Porter-Stemmer Algorithmus. Der Porter-Stemmer Algorithmus ist ein Algorithmus aus der Computerlinguistik und wird zum automatischen zurückführen von Wörtern auf ihren Wortstamm (auch „stemming“ genannt) eingesetzt.

Abbildung 4.1: Entwicklung des Porter-Stemming Algorithmus [Swa17]



Der von Martin Porter im Jahr 1979[Por06] entwickelte Algorithmus basiert auf diversen Verkürzungsregeln, die solange auf ein Wort angewandt werden, bis dieses nur noch eine bestimmte Anzahl an Vokal-Konsonant-Sequenzen aufweist. Jedes Wort in jeder natürlichen Sprache besteht aus einer optionalen Anzahl beginnender Konsonanten $[C]$, einer Folge von Vokal-Konsonant Sequenzen $VCVC...$ oder kurz $(VC)^m$ und einer optionalen Folge an Vokalen $[V]$ am Ende. Zusammengesetzt ergibt dies die Form

$$[C](VC)^m[V]$$

Jedes Wort weist also eine Folge von $0-n$ Konsonanten auf, gefolgt von m Vokal-Konsonant

Sequenzen und schließt mit $0-n$ Vokalen ab. Die Anzahl an Vokal-Konsonant-Sequenzen m entscheidet über die Fortsetzung der Anwendung der Verkürzungsregeln. Unterschreitet m ein gewisses Mindestmaß, wird das Stemming für dieses Wort beendet.
Ein Beispiel zum besseren Verständnis[Por80]:

```
c r e p u s c u l a r
    |   |     |   | |
[C] V C V C   V C V C
      1   2     3   4
```

Das Wort lässt sich unter Berücksichtigung der obigen Form wie folgt trennen: Die anfängliche und optionale Folge von Konsonanten **cr** und vier Vokal-Konsonant Sequenzen: **ep** - **usc** - **ul** - **ar**. Für das englische Wort *crepuscular* gilt demnach $m = 4$. Einige weitere Beispiele in deutscher und englischer Sprache:

Wort	**Zerlegung**	m
tree	$[CC] =$ **tr** - $[VV] =$ **ee**	$m = 0$
Baum	$[C] =$ **B** - $VVC =$ **aum**	$m = 1$
between	$[C] =$ **b** - $VCC =$ **etw** - $VVC =$ **een**	$m = 2$
Wikipedia	$[C] =$ **W** - $VC =$ **ik** - $VC =$ **ip** - $VC =$ **ed** - $[VV] =$ **ia**	$m = 3$
crepuscular	$[CC] =$ **cr** - $VC =$ **ep** - $VCC =$ **usc** - $VC =$ **ul** - $VC =$ **ar**	$m = 4$

Wie ersichtlich wird, ist innerhalb der Vokal-Konsonant Sequenzen kein strikter Übergang von Konsonant zu Vokal gefordert. Eine Vokal-Konsonant Sequenz beginnt mit dem ersten Vokal nach einem optionalen Konsonanten und endet mit dem letzten Konsonanten vor einem optionalen Vokal. Als regulärer Ausdruck würde eine Vokal-Konsonant Sequenz daher wie folgt aussehen:

```
V+C+
```

Mindestens ein Vokal gefolgt von mindestens einem Konsonanten.

Martin Porter entwarf den Algorithmus ursprünglich nur für Worte aus der englischen Sprache. Der Algorithmus kann allerdings durch einige Modifizierungen der Regeln relativ leicht für andere Sprachen portiert werden. Im Abschnitt 4.4.1 wird am Beispiel eines Stemmers für die deutsche Sprache auf die Modifizierung dieser Regeln genauer eingegangen.
Die mittels des Porter-Stemmer abgeleiteten Wortstämme entsprechen oft nicht den tatsächlichen linguistisch korrekten Wortstämmen. So wird z.B. das Wort *conspicuously* auf *conspicu* gestemmt und nicht, wie es linguistisch korrekt wäre, auf *conspicuous*. Eine korrekte linguistische Analyse ist jedoch auch nicht Ziel des Algorithmus. Sinn und Zweck ist eine einheitliche Rückführung von verwandten Worten auf einen gemeinsamen Wortstamm. Da bspw. sowohl *conspicuously* als auch *conspicuous* auf *conspicu* gestemmt werden, ist zwar keine linguistisch korrekte, aber eine eindeutige Rückführung gelungen.

Verkürzungsregeln

Die jeweiligen Verkürzungsregeln sind in Gruppen zusammengefasst und mit verschiedenen Prioritäten versehen. Die Regeln selbst bestehen aus Bedingungen und daraus folgernden Ableitungen für bestimmte Wortendungen. Für jede Gruppe gilt, dass nur eine der dort enthaltenen Regeln für den jeweiligen Durchlauf angewendet werden darf. Ein Beispiel einer solchen Gruppe wären die folgenden Regeln:

Endung	**Ersetzung**
-sses	ss
-ies	i
-ss	ss
-s	

Die obigen Regeln sind wie folgt zu verstehen: Endet das untersuchte Wort auf eine der Endungen, wird die entsprechende Endung durch die angegebene Ersetzung ausgetauscht. Die Priorität entspricht der Länge der Endung: Die längste Endung hat die höchste Priorität. Es folgt ein Beispiel für jede der oben genannten Regeln:

Endung	**Ersetzung**	**Vorher**	**Nachher**
-sses	ss	care**sses**	caress
-ies	i	poni**es**	poni
-ss	ss	care**ss**	caress
-s		cat**s**	cat

Zusätzlich zu den obigen Regeln gibt es eine weitere Regel, die den Übergang

$$y \rightarrow i$$

definiert, sofern dem **y** ein Vokal vorausgeht. Der Vokal muss nicht direkt vor dem **y** stehen, es reicht, wenn ein Vokal zuvor aufgetreten ist.
Dadurch würden sich die folgenden zusätzlichen Ersetzungen ergeben:

Wort	**Ersetzung**
happy	happi
sky	sky (Kein Vokal vor dem y)
pony	poni

Fehlerrate

Bei einem Vergleich der beiden Ersetzungstabellen fällt auf, dass sowohl *pony* als auch *ponies* auf den gleichen Wortstamm *poni* reduziert werden. Das Stemming wird also korrekt durchgeführt. Dies ist allerdings nicht immer zutreffend. Wie alle bekannten Stemming-Algorithmen weist auch der Porter-Stemmer-Algorithmus eine gewisse Fehlerrate auf

[Pre08][Bro02][BVq]. Beim Stemming können zwei mögliche Arten von Fehlern auftreten: *Overstemming* und *Understemming*. Werden zwei einzelne und morphologisch[1] nicht verwandte Worte fälschlicherweise auf den gleichen Wortstamm zurückgeführt, wird dies als *Overstemming* bezeichnet. *Understemming* hingegen beschreibt den Fall, wenn zwei morphologisch verwandte Worte nicht auf denselben Wortstamm zurückgeführt werden. Die verschiedenen Stemming-Algorithmen weisen inzwischen häufig Gegenmaßnahmen in Form von Ausnahmen oder weiteren Regeln auf, um *Over-* und *Understemming* zu vermeiden.
Ein Beispiel[2] für *Overstemming* des Porter-Stemmer-Algorithmus ist die Rückführung der morphologisch nicht verwandten Begriffe *universal*, *university* und *universum* auf den gleichen Wortstamm *univers*. Ein Beispiel[3] für *Understemming* des Porter-Stemmer-Algorithmus ist hingegen die nicht stattfindende Rückführung der Begriffe *alumni*, *alumna* und *alumnus* auf einen gemeinsamen Wortstamm. Auf Grund der morphologischen Herkunft der Worte aus dem Lateinischen kann der Porter-Stemmer-Algorithmus mit den Englischen Regeln keine gemeinsame Herleitung ermitteln. Somit bleiben *alumni* und *alumna* unverändert, während bei *alumnus* lediglich das Suffix **s** - wie in der letzten Regel der 1. Ersetzungstabelle definiert - eliminiert wird.
Die Genauigkeit des Porter-Stemmer-Algorithmus ist für die praktische Verwendung jedoch ausreichend gut.

4.4.1 Snowball

Snowball ist eine ursprünglich von Martin Porter entwickelte Verarbeitungssprache für Texte. Sie dient zur Erstellung von Stemming-Algorithmen für verschiedene natürliche Sprachen. *Snowball* verfügt über eine eigene syntaktische und semantische Darstellung, offiziell als *Snowball-Skript* bezeichnet. Dieses *Snowball-Skript* wird durch den *Snowball*-Compiler in eine konkrete Programmiersprache übersetzt. Derzeit werden die Programmiersprachen C, Java, Python, Rust und Go unterstützt[Porc]. Die sprachspezifische Implementierung wird als *Stemmer* bezeichnet. Der Name ist angelehnt an die Sprache *SNOBOL*, die zwischen 1962 und 1967 entwickelt wurde[Por80]:

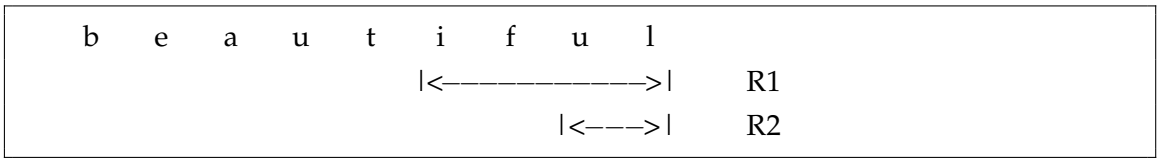
> „'Snowball' named as a tribute to SNOBOL, the excellent string handling language of Messrs Farber, Griswold, Poage and Polonsky from the 1960s."
>
> (Martin Porter)

MongoDB verwendet intern eine C++ Kapselung der C-Portierung des Porter-Stemmer Algorithmus, welcher mit Snowball portiert wurde. SolR und Elasticsearch verwenden beide jeweils Lucene, welches wiederum eine Java Kapselung der Java-Portierung des Porter-Stemmer Algorithmus verwendet, die ebenfalls mittels Snowball portiert wurde.

[1] In der Sprachwissenschaft bezeichnet der Begriff *Morphologie* die Wortbildungs- und Formenlehre
[2] Beispiel entnommen aus [con17]
[3] Beispiel entnommen aus [con17]

Stemmer-Algorithmus in Snowball

Die meisten Stemmer von *Snowball* verwenden nicht den originalen Algorithmus von Porter, sondern eine eigene Variante. Der wesentlichste Unterschied zum Porter-Stemmer Algorithmus ist, dass anstatt *m* stets erneut zu errechnen, zwei feste und initial berechnete Bereiche innerhalb des Wortes verwendet werden[Pora]: Diese Bereiche - im Original auch als Regionen bezeichnet - sind als *R1* (1. Region) und *R2* (2. Region) definiert. Der Bereich *R1* beginnt **nach** dem ersten Nicht-Vokal nach einem Vokal und der Bereich *R2* beginnt **nach** dem ersten Nicht-Vokal nach einem Vokal in *R1*. Sowohl *R1* als auch *R2* können leer sein, falls die angegebenen Bedingungen bis zum Wortende nicht erfüllt wurden. Die Definition der Vokale variiert von Sprache zu Sprache. Als Beispiel diene das Wort *beautiful*:



R1 ist *iful*, da *t* der erste Nicht-Vokal nach dem Vokal *u* ist. In *iful* wiederum ist *f* der erste Nicht-Vokal nach dem Vokal *i*, somit ist *R2 ul*. Im Wort *beauty* ist *R1 y* und *R2* leer und im Wort *beaut* sind sowohl *R1* als auch *R2* leer.

Beispiel: Deutscher Stemming-Algorithmus

Im Folgenden soll der Stemming-Algorithmus aus *Snowball* für die deutsche Sprache im Detail betrachtet werden[4]: Der Stemming-Prozess für ein deutsches Wort gliedert sich in fünf Schritte:

1. Die deutsche Sprache enthält den Buchstaben *ß* der als Alias zu einem scharfen *s* bzw. einem doppelten *s* verstanden wird. Des Weiteren definiert die deutsche Sprache die folgenden Buchstaben als Vokale: *a*, *e*, *i*, *o*, *u*, *y* sowie die Umlaute *ä*, *ü* und *ö*. Zunächst wird das Wort in die Kleinschreibweise überführt. Anschließend wird das *ß* aufgelöst und durch *ss* ersetzt. Zwischen Vokalen vorkommende *u* und *y* werden durch ein großes U bzw. ein großes Y ersetzt, um missliche Vokal-Interpretationen wie *ue*, *oe* und *ae* zu vermeiden. Derartige Vokal-Aliase kommen z.B. aus dem englischsprachigen Raum, da die dort vorhandenen Tastaturen - sprach bedingt - zumeist keine (leichte) Möglichkeit zum Schreiben von deutschen Umlauten besitzen. Dann werden die Bereiche *R1* und *R2* - wie im vorherigen Abschnitt 4.4.1 beschrieben - definiert. Der deutsche Stemming-Algorithmus hat eine Sonderregelung für *R1*: die Region davor muss mindestens drei Buchstaben aufweisen. Ist dies nicht der Fall, wird *R1* entsprechend angepasst. Zum Schluss werden noch die validen Endungen für *s* und *st* festgelegt. Für *s* sind dies die Buchstaben *b*, *d*, *f*, *g*, *h*, *k*, *l*, *m*,

[4]Beispiel entnommen aus [Porb]

n, *r* oder *t*. Die Endungen für *st* sind dieselben wie für *s*, allerdings ohne den Buchstaben *r*. Diese Endungen sind diejenigen Buchstaben, die **vor** einem *s* bzw. *st* am Endes des Wortes stehen können. Ein Beispiel wäre das Wort *Derbst*, welches mit **st** endet und dem der Buchstabe **b** vorausgeht.

2. Zunächst wird nach dem längsten der folgenden Suffixe gesucht und - sofern sie in *R1* enthalten sind - aus dem Wort und *R1* gelöscht:

   a) *em*, *ern*, *er*

   b) *e*, *en*, *es*

   c) *s* (Sofern eine valide Endung vorausgeht)

   Zum Beispiel würde das Wort *armes* nach diesem Schritt zu *arm* verkürzt werden. Ferner gilt, wenn eine Endung aus der Gruppe **b** entfernt wird und der Suffix *niss* bestehen bleibt, dass das doppelte *s* auf ein *s* verkürzt wird. Demnach würde aus dem Wort *bedürfnissen* zunächst *bedürfniss* und dann *bedürfnis* werden.

3. Es wird wiederum nach dem längsten der folgenden Suffixe gesucht und - sofern sie in *R1* enthalten sind - aus dem Wort und *R1* gelöscht:

   a) *en*, *er*, *est*

   b) *st* (Sofern eine valide Endung - der wiederum mindestens drei Buchstaben vorausgehen - vorliegt)

   Zum Beispiel würde das initiale Wort *derbsten* nach Schritt 1 zu *derbst* und nach Schritt 2 zu *derb* verkürzt werden, da der Buchstabe *b* eine valide Endung für *st* darstellt und drei Buchstaben vorausgehen.

4. Es wird wiederum nach dem längsten der folgenden Suffixe gesucht und, sofern die jeweiligen Bedingungen zutreffen, aus dem Wort gelöscht:

   a) *end*, *ung* (Sofern in *R2* enthalten. Wenn *ig* vorausgeht, darf kein *e* davor vorkommen)

   b) *ig*, *ik*, *isch* (Sofern in *R2* enthalten und kein *e* vorausgeht)

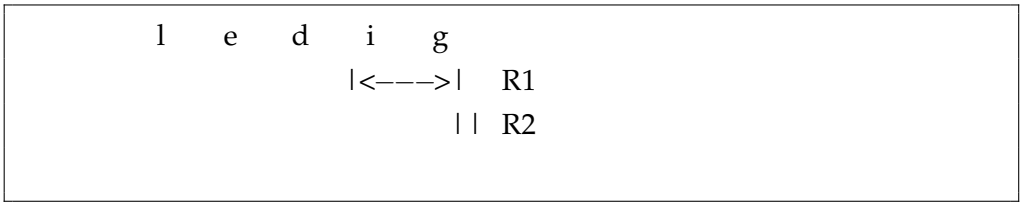
   c) *lich*, *heit* (Sofern in *R2* enthalten. Wenn *er* oder *en* vorausgehen, löschen wenn es in *R1* vorkommt)

   d) *keit* (Sofern in *R2* enthalten. Wenn *lich* oder *ig* vorausgehen, löschen wenn es in *R2* vorkommt)

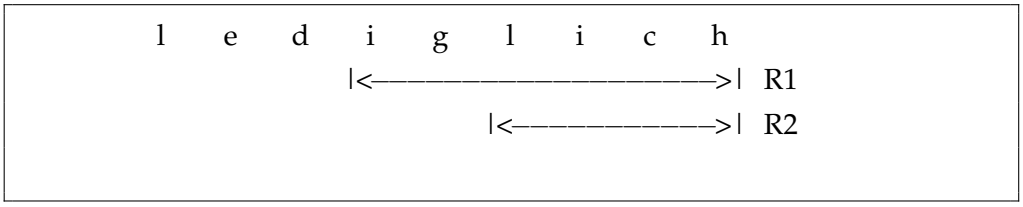
   Im Folgenden zwei Beispiele für die oben beschriebenen Regeln:

   - Auf Grund von Regel **a** würde das Wort *Unterbindung* auf *unterbind* zurückgeführt werden. Das Wort *Steigung* jedoch würde - da ein *e* vor dem *ig* vorkommt - nicht verkürzt werden.

- Auf Grund von Regel **c** würde das Wort *lediglich* auf das Wort *ledig* zurückgeführt werden. Das Wort *ledig* würde nicht weiter verkürzt werden, da Regel **b** besagt, dass *ig* in *R2* enthalten sein muss. *R2* ist für das Wort *ledig* jedoch, wie dem folgenden Beispiel zu entnehmen ist, leer:



  Wie dem Abschnitt 4.4.1 zu entnehmen ist, beginnt *R1* mit dem ersten Nicht-Vokal **nach** einem Vokal. Der erste Nicht-Vokal nach einem Vokal ist der Buchstabe *d*. Damit ist *R1 ig*. *R2* wiederum beginnt **nach** dem ersten Nicht-Vokal nach einem Vokal in *R1*. Der erste Buchstabe aus *R1* ist ein Vokal, der zweite und letzte Buchstabe hingegen nicht. Daher würde *R2* danach beginnen. Da *R1* aber bereits leer ist, ist demzufolge auch *R2* leer. Damit ist die Bedingung für die Regel **b** nicht erfüllt und sie wird nicht angewendet, womit das Wort *ledig* unverändert bleibt.
  Das Wort *lediglich* enthält jedoch, wie dem nachfolgenden Beispiel zu entnehmen ist, den Suffix *lich* in *R2*:



  R1 ist *iglich* und *R2* entspricht *lich*, also in diesem Fall sogar exakt dem entsprechenden Suffix.

5. Abschließend werden die in Schritt 1 in die Großschreibweise transformierten *U* und *Y* zurück in ihre Kleinschreibweise und die deutschen Umlaute in ihre jeweils zugrundeliegenden Buchstaben überführt:
   - ä → a
   - ö → o
   - ü → u

   Das Wort *Feuer* würde nach Schritt 1 in *feUer* transformiert werden, durch Regel **a** von Schritt 2 die Endung *er* verlieren und nach diesem Schritt *feu* lauten.

4.5 Gewichtungen

Bei einer stattfindenden Volltextsuche wird jedem Dokument, dass die Suchanfrage beinhaltet, eine Bewertung - ein sogenannter *Score* - zugewiesen. Dieser *Score* - im weiteren

Verlauf auch als *Text-Score* bezeichnet - bestimmt die Relevanz die ein Dokument bezüglich der Suchanfrage aufweist. Für einen Text-Index kennzeichnet die Gewichtung pro Index-Feld die Signifikanz der Suchanfrage relativ gesehen zu allen anderen Text-Indizes. Für jedes indexierte Feld innerhalb des Dokuments werden die Übereinstimmungen (oder *Matches*) mit den jeweiligen Gewichtungen pro Feld multipliziert und anschließend aufaddiert. MongoDB benutzt diese Summe um einen endgültigen *Text-Score* für das Dokument bezüglich der Suchanfrage zu berechnen. Die standardmäßige Gewichtung für jedes indexierte Feld beträgt 1. Dieser Wert kann allerdings bei der Erstellung des Text-Index angepasst werden. Um bei dem bisherigen Beispiel eines Nachrichtensystems zu bleiben, könnte die Anpassung der Gewichtungen wie folgt aussehen[Monk]:

```
db.news.createIndex(
   {
        content: "text",
        title: "text",
        category: "text"
   },
   {
        weights: {
            content: 10,
            title: 5
        }
   }
 )
```

Der *Compound Text-Index* weist damit die folgenden Gewichtungen für die einzelnen Felder auf:

- *content* hat eine Gewichtung von 10
- *title* hat eine Gewichtung von 5
- *category* hat die Standard-Gewichtung von 1

Diese Gewichtungen kennzeichnen die relative Signifikanz der Index-Felder zueinander: Eine Übereinstimmung mit dem Feld *content* hätte den doppelten Effekt als eine Übereinstimmung mit dem Feld *title* und den zehnfachen Effekt in Relation zum Feld *category*.

4.6 Berechnung des Text-Score

In dem vorherigen Abschnitt 4.5 wurde der Begriff *Text-Score* bereits erwähnt und kurz beschrieben. Der *Text-Score* eines Dokuments definiert die Relevanz bzgl. einer Suchanfrage und setzt sich aus den einzelnen berechneten Text-Scores der jeweiligen Index-

Felder zusammen. Die Zusammensetzung und der Ablauf der Berechnung gliedert sich in vier wesentliche Schritte:

1. Für jeden Text-Index eines Dokuments wird der Inhalt des jeweiligen Feldes ausgelesen. Parallel dazu findet eine Überprüfung statt, ob die angegebene Gewichtung des momentan betrachteten Text-Index innerhalb einer gewissen Größenordnung liegt. Die minimale und zugleich Standardgewichtung - wie dem Abschnitt 4.5 zu entnehmen ist - beträgt 1. Die Obergrenze einer Gewichtung wird MongoDB intern mit dem Wert 99.999 definiert. Anschließend wird der ausgelesene Inhalt des Dokument-Feldes unter Zuhilfenahme der bestimmten Sprache des Dokuments zunächst einer lexikalischen Analyse und darauf folgend einer Stemming-Prozedur unterzogen.

2. Auf jedes der gestemmten Wörter wird nun der folgende Algorithmus angewendet: Zunächst wird der *Score* des angegebenen Wortes betrachtet. Ein *Score* setzt sich aus einem steigenden Exponenten, einer konstant steigenden Zähler-Variable und einer stets neu berechneten Frequenz zusammen. Alle drei Werte besitzen anfänglich den Wert 0. Bei jeder Iteration wird für den betrachteten *Score* zunächst bestimmt, ob dem Exponent bereits ein Wert $>= 1$ zugewiesen wurde. Ist dies der Fall, wird der vorliegende Exponent verdoppelt, andernfalls wird ihm der Wert 1 zugewiesen. Im zweiten Schritt wird die Zähler-Variable um den Wert 1 inkrementiert. Im dritten und letzten Schritt wird das Ergebnis der Division zwischen 1 und dem aktuellen Exponent auf die momentane Frequenz des Wortes aufaddiert.
   Diese drei Schritte werden für alle gefilterten und gestemmten Worte des Feld-Inhalts durchgeführt. Wie ersichtlich wird, steigt die Relevanz eines Wortes, ausgedrückt durch die Frequenz im *Score*, je häufiger es enthalten ist.
   Ein Beispiel zur Demonstration:
   Angenommen, es liegt der folgende Inhalt für ein Dokument Feld vor:
   These droids are looking for danger. These Droids have defence against other droids and danger.
   Womit, nach einer stattgefundenen lexikalischen Analyse und der darauf folgenden Stemming-Prozedur, die folgenden Worte verbleiben:

Wort	**Häufigkeit**
droid	3
look	1
danger	2
defence	1

   Der beschriebene Algorithmus für diesen Schritt würde als Pseudo-Code folgendermaßen aussehen:

```
1 class Score
```

```
2  {
3      exponent = 0.0;
4      count = 0.0;
5      frequency = 0.0;
6  }
7
8  words = ["droid", "look", "danger", "droid", "defence", "
       ↪ droid", "danger"];
9  scores = [];
10 for (word in words) {
11     if (word not in words) {
12         scores[word] = new Score();
13     }
14
15     score = scores[word];
16     if (score.exponent < 1) {
17         score.exponent = 1;
18     } else {
19         score.exponent *= 2;
20     }
21
22     score.count += 1;
23     score.frequency += (1 / score.exponent);
24 }
```

Nach Anwendung des Algorithmus ergeben sich die folgenden Werte:

Wort	**Exponent**	**Count** (Häufigkeit)	**Frequency**
droid	4	3	1.75
look	1	1	1
danger	2	2	1.5
defence	1	1	1

Die Frequenz eines Wortes erhöht sich pro Vorkommnis um den Wert $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$ ... oder kurz gesagt

$$\sum_{n=1}^{\infty}(\frac{1}{2})^n$$

Diese Folge ist auch unter dem Begriff der geometrischen Reihe bekannt. Kommt ein Wort x-mal vor, entspricht die Häufigkeit des Wortes somit

$$1+\frac{1}{2}+...+(\frac{1}{2})^{(x-1)} = (1-(\frac{1}{2})^x)/(1-\frac{1}{2}) = 2(1-(\frac{1}{2})^x)$$

Nachweis: Das Wort *droid* hat eine Häufigkeit von 3. Einsetzen in die obige Formel ergibt:

$$2(1 - (\frac{1}{2})^3) = 1.75$$

Der daraus resultierende Effekt ist, dass weitere Vorkommen desselben Wortes als weniger relevant betrachtet werden. Würde *droid* 5, 7 oder 9 mal vorkommen, würden sich damit die folgenden Werte ergeben:

Häufigkeit	**Wert**
3	$2(1 - (\frac{1}{2})^3) = 1.75$
5	$2(1 - (\frac{1}{2})^5) = 1.9375$
7	$2(1 - (\frac{1}{2})^7) = 1.984375$
9	$2(1 - (\frac{1}{2})^9) = 1.99609375$

3. Nachdem die Scores für die einzelnen Worte berechnet sind, fehlt noch die Gesamtbewertung. Dazu werden nun die *Score*s der einzelnen Worte durchlaufen und für jeden *Score* findet die folgende Berechnung statt:
   Um die Gewichtung als Funktion in Hinsicht auf die Worthäufigkeit zu bestimmen, wird zunächst ein Koeffizient berechnet, indem der finale Wert der Zähler-Variable - also die Häufigkeit des Wortes im Text - mit 0.5 multipliziert und durch die Anzahl der verbleibenden Worte geteilt wird. Um Rundungsfehler zu vermeiden, wird der Wert 0.5 hinzu addiert.
   Unter Zuhilfenahme des berechneten Koeffizienten, der berechneten Frequenz des Wortes, der angegebenen Gewichtung des Text-Index und einer Justierungs-Variable wird der Gesamtwert für dieses Wort berechnet. Dazu werden diese vier Komponenten miteinander multipliziert. Die Justierungs-Variable hat dabei initial pro Berechnung den Wert 1.0. Sollte der ursprüngliche Inhalt des Dokument-Feldes dem aktuell betrachteten Wort entsprechen - sprich: der Inhalt wies nur ein Wort auf, welches noch dazu nicht gestemmt werden musste, weil es bereits in der Grundform vorlag - erhöht sich dieser Wert um 0.1 auf 1.1 um die Signifikanz zu verdeutlichen. Abschließend wird der berechnete Wert auf den ggf. zuvor berechneten Wert desselben Wortes hinzu addiert.
   Auch dieser Algorithmus soll anhand eines Beispiels demonstriert werden. Dafür wird auf dem im vorherigen Schritt beschriebenen Beispiel aufgebaut und zusätzlich angenommen, dass eine Standardgewichtung von 1 für den untersuchten Text-Index vorliegt. Der in diesem Schritt beschriebene Algorithmus würde in Pseudo-Code ausgedrückt wie folgt aussehen:

```
1 numTokens = words.length;
2 weight = 1;
3 frequencies = [];
```

```
4
5  for ((word, score) in scores) {
6      coeff = (0.5 * score.count / numTokens) + 0.5;
7      adjustment = 1.0;
8      if (content.length == word.length && content.toLowerCase
       ↪ () == word.toLowerCase()) {
9          adjustment += 0.1;
10     }
11
12     if (word not in frequencies) {
13         frequencies[word] = 0.0;
14     }
15
16     frequencies[word] += (weight * score.frequency * coeff *
       ↪  adjustment);
17 }
```

Nach der Anwendung ergeben sich die folgenden Werte:

Wort	**Häufigkeit**	**Frequenz**
droid	3	1.25
look	1	0.57142857142857
danger	2	0.96428571428571
defence	1	0.57142857142857

4. Der endgültige*Text-Score* des Dokuments ergibt sich wie folgt:
   Die Worte der Suchanfrage - die ebenfalls einer lexikalischen Analyse sowie einem Stemming-Prozess unterzogen wurden - werden der Reihe nach Wort für Wort durchlaufen. Für jedes Wort wird der im vorherigen Schritt berechnete Wert auf einen mit 0 initialisierten Gesamt-Score aufaddiert. Der *Text-Score* für ein Dokument ergibt sich damit aus den aufaddierten Werten der jeweils gesuchten Worte im gesamten Dokument.
   Auch hier soll ein abschließendes Beispiel den Ablauf verdeutlichen:
   Angenommen, die Sucheingabe würde *look, there are droids* lauten. Nach der ebenfalls stattfindenden lexikalischen Analyse und der Stemming-Prozedur würden lediglich die Worte *look* und *droid* verbleiben. Der in diesem Schritt beschriebene Algorithmus hat die folgende Pseudo-Code Repräsentation:

```
1  search = ["look", "droid"];
2  score = 0.0;
3  for (word in search) {
4      if (word in frequencies) {
5          score += frequencies[word];
```

```
6         }
7 }
```

Nach der Anwendung des Algorithmus würde sich der Wert $1.25+0.57142857142857 \approx 1.8214285714286$ als endgültiger *Text-Score* des Dokuments ergeben, unter der Voraussetzung, dass nur für das eine untersuchte Dokument-Feld ein Text-Index existiert.

4.7 Verfügbare Modi & deren Wirkungsweise

Die MongoDB Volltextsuche kann neben der selbst anzugebenden Gewichtung auch durch spezifische Optionen beeinflusst werden. Die beiden letzten der nun folgenden drei Optionen können optional beim Aufruf der *find*-Operation als boolescher Wert angegeben werden.

4.7.1 Filter Stopwords

Diese Option ist automatisch aktiviert und sorgt dafür, dass vor der Stemming-Prozedur sämtliche Stopwörter der zuvor bestimmten Sprache herausgefiltert werden. Diese Option kann nur durch die explizite Angabe von *none* als zu verwendende Sprache deaktiviert werden. Entweder beim Erstellen des Text-Index, damit nie Stopwörter gefiltert werden (Siehe Abschnitt 4.2):

```
1 { default_language: "none" }
```

Oder direkt im Dokument durch das Attribut das durch *language_override* spezifiziert wurde (Siehe Abschnitt 4.2), damit nur dieses Dokument von der Stopwort-Filterung ausgeschlossen wird. Die Deaktivierung ist u.a. sinnvoll, wenn der Inhalt des Dokuments nicht in einer natürlichen Sprache vorliegt oder wenn die verwendete Sprache derzeit nicht unterstützt wird. Die Deaktivierung hat allerdings auch zur Folge, dass ebenfalls keine Stemming-Prozedur durchgeführt wird[Monr]:

> If you specify a language value of "none", then the text index uses simple tokenization with no list of stop words and no stemming.

4.7.2 Case Sensitive

Wie im Abschnitt 4.3 bereits erwähnt, liegen die Stopwörter in MongoDB vollständig in der Kleinschreibweise vor. Da in MongoDB die lexikalische Analyse und die Stemming-Prozedur zusammenhängende Prozesse sind, werden die einzelnen Worte des Textes zunächst in die Kleinschreibweise überführt. Anschließend wird überprüft, ob es sich bei dem vorliegenden Wort um ein Stopwort handelt. Um den Zusammenhang beider Prozesse zu verdeutlichen, folgt ein Beispiel in Pseudo-Code:

```
while (tokenizer.hasMoreTokens()) {
    token = tokenizer.getNextToken();
    if (!token.isWord()) {
        continue;
    }

    word = token.getWord().toLowerCase();
    if (options.filterStopWords() && isStopWord(word)) {
        continue;
    }
}
```

Handelt es sich bei dem jeweils untersuchten Wort um kein Stopwort, wird das Wort standardmäßig in der nun vorliegenden Kleinschreibweise als Repräsentation des Tokens übernommen. Die MongoDB Volltextsuche ist demnach standardmäßig *Case-Insensitive*. Dies macht in den meisten Fällen keinen Unterschied. Es existieren jedoch Worte und Abkürzungen, die die gleichen Buchstaben besitzen und dennoch unterschiedliche Bedeutungen haben. Die konkrete Bedeutung lässt sich oft nur anhand der Groß- und Kleinschreibung festlegen. Ein passendes Beispiel[5] wäre das Wort *Ram* (auch *Rama*; eine Hindu-Gottheit) und das Wort *RAM* (eine Abkürzung für **R**andom **A**ccess **M**emory). Mit einer *Case-Insensitiven* Suche würden beide Begriffe als gleichwertig eingestuft werden. Mit der Angabe der Option *GenerateCaseSensitiveTokens* wird MongoDB allerdings mitgeteilt, dass eine Berücksichtigung der Groß- und Kleinschreibung - also eine *Case-Sensitive* Betrachtung - stattfinden soll. Bei Angabe dieser Option wird MongoDB, nach der Überprüfung, ob es sich bei dem vorliegenden Wort um ein Stopwort handelt, die ursprüngliche Repräsentation des Wortes wiederherstellen. Um das obige Beispiel entsprechend zu ergänzen, sieht der beschriebene Prozess folgendermaßen aus:

```
while (tokenizer.hasMoreTokens()) {
    token = tokenizer.getNextToken();
    if (!token.isWord()) {
        continue;
    }

    word = token.getWord().toLowerCase();
    if (options.filterStopWords() && isStopWord(word)) {
        continue;
    }

    if (options.generateCaseSensitiveTokens()) {
```

[5] Beispiel entnommen aus [Wik]

```
13          word = token.getWord();
14      }
15
16      stemmedWord = stem(word);
17  }
```

In Zeile 13 ist die Wiederherstellung des ursprünglichen Wortes zu sehen, während in Zeile 16 die Stemming-Prozedur stattfindet.
Diese Option kann wie folgt bei der *find*-Operation angegeben werden:

```
db.news.find({
    $text: {
        $search: <string>,
        $language: <string>,
        $caseSensitive: true
    }
});
```

Die Option *$caseSensitive* wurde mit Version 3.2 eingeführt [Monc] [Monv].

4.7.3 Diacritic Sensitive

Diakritische Zeichen (auch als *Diakritika* bezeichnet) sind linguistische Charakteristika einer Sprache. Diakritika sind erkennbar als besondere Merkmale an Buchstaben, wie Bögen, Punkte oder Striche. Diese Diakritika geben an, dass der Buchstabe eine andere Betonung und/oder Aussprache besitzt. In der Deutschen Sprache sind z.B. die Punkte der Umlaute diakritische Zeichen.
Für gewöhnlich werden diakritische Zeichen in MongoDB bei der Stemming-Prozedur entfernt bzw. durch den zugrunde liegenden Buchstaben ersetzt. So wird aus den deutschen Umlauten **ä**, **ü** und **ö** die Vokale **a**, **u** und **o**. Dieses Verhalten kann durch die Angabe der Option *GenerateDiacriticSensitiveTokens* allerdings deaktiviert werden.
Der Basis-Tokenizer von MongoDB ist allerdings nicht in der Lage, eine Diakritisch-Sensitive lexikalische Analyse durchzuführen, da die C++ Implementierung keine interne Unicode Unterstützung besitzt - anders als etwa Java. Aus diesem Grund existiert ein eigener Tokenizer für Unicode basierte Sprachen, der mittels einer externen Fremdquelle eine Unicode Unterstützung anbietet. Bei der Angabe von *none* als zu verwendende Sprache wird der Basis-Tokenizer eingesetzt.
Diese Option kann wie folgt bei der *find*-Operation angegeben werden:

```
db.news.find({
    $text: {
```

```
        $search: <string>,
        $language: <string>,
        $diacriticSensitive: true
    }
});
```

Die Option *$diacriticSensitive* wurde mit Version 3.2 eingeführt [Monc] [Monv].

4.8 Term-Negationen

Bisher wurde geklärt, inwiefern nach bestimmten Worten gesucht werden kann. Darüber hinaus existiert die Option, Dokumente basierend auf den enthaltenen Worten zu filtern. Diese als negierte Terme bezeichneten Worte werden durch ein unmittelbar vorhergehenden Bindestrich bzw. Minuszeichen gekennzeichnet. Dies gilt allerdings nur dann, wenn eine der beiden folgenden Kriterien erfüllt ist[Monl][Monm]:

- Dem Bindestrich geht mindestens ein Leerzeichen voraus
- Der Bindestrich ist das erstes Zeichen der Suchanfrage

Andernfalls wird er als Trennsymbol interpretiert. Enthält eine Suchanfrage negierte Terme, werden Dokumente, die diese Terme enthalten, ignoriert. Um bei dem Beispiel des Nachrichtensystems zu bleiben, wäre ein passendes Beispiel diejenigen Nachrichten zu filtern, die das Wort „Fake“ beinhalten:

```
db.news.find({
    $text: {
        $search: "-Fake"
    }
});
```

Eine Negation gilt bis zum nächsten Vorkommen eines Leerzeichens. Somit enthält die Suchanfrage

```
-Fake Nachricht
```

den negierten Term *Fake* sowie den positiven Term *Nachricht*. Diese Suchanfrage würde diejenigen Dokumente ausschließen, die entweder das Wort „Fake“ oder nicht das Wort „Nachricht“ beinhalten. Die Suchanfrage

```
- Fake Nachricht
```

enthält dagegen - hervorgerufen durch das Leerzeichen zwischen dem Wort „Fake“ und dem Minus - zwei positive und keine negativen Terme. Enthält ein Term mehrere Negationen, werden die darauf folgenden Worte als einzelne negierten Terme interpretiert. Daher enthält die Suchanfrage

```
-Fake-Nachricht
```

zwei negierte Terme (*Fake* und *Nachricht*) und entspricht damit semantisch der Suchanfrage

```
-Fake -Nachricht
```

4.9 Phrase-Matching

Wird in einer Suchmaschine wie Google nach einem zusammengesetzten Suchbegriff wie *Starwars Episode 6* gesucht, werden die darin vorkommenden Begriffe

- Starwars
- Episode
- 6

mittels einer Oder-Bedingung verknüpft und eine darauf basierende Suchanfrage ausgeführt. Vereinfacht ausgedrückt bedeutet das, dass sämtliche Dokumente die eines dieser drei Worte - aber nicht notwendigerweise alle drei - enthalten, als positives Suchergebnis zurückgeliefert werden. Wenn jedoch nur nach den Dokumenten gesucht werden soll, die auch tatsächlich alle drei Begriffe beinhalten, muss eine auf einer Und-Bedingung basierende Suchanfrage stattfinden. Dazu müssen die Suchbegriffe in Anführungszeichen gesetzte werden: „Starwars Episode 6“. Diese Notation hat sich bei der Mehrheit aller Suchmaschinen bzw. Volltextsuchen durchgesetzt[Phr11] (in manchen Suchmaschinen können auch eckige Klammern verwendet werden[Kar]) und wird allgemein als *Phrase* bezeichnet. Enthält ein Dokument nicht die gesuchte Phrase, wird es nicht als positives Suchergebnis identifiziert. Die Notation von Phrasen hat in MongoDB überdies den Effekt, dass kein Stemming und damit ebenfalls keine Umwandlung in die Kleinschreibweise stattfindet. Ist die Option *GenerateCaseSensitiveToken* aktiviert, wird auch eine Case-Sensitive Überprüfung der Phrase innerhalb des durchsuchten Dokuments durchgeführt. Andernfalls findet eine Case-Insensitive Überprüfung statt.

4.9.1 Phrase-Negationen

Auch Phrasen können negiert werden. Die Notation einer negierten Phrase ist äquivalent zu der von Termen. Ebenso wie bei Termen wird eine negierte Phrase durch ein unmittelbar vorhergehenden Bindestrich bzw. Minuszeichen gekennzeichnet. Dies gilt ebenfalls nur dann, wenn dem Bindestrich mindestens ein Leerzeichen vorausgeht oder er am Anfang der Suchanfrage steht. Andernfalls wird er als Teil der Phrase interpretiert. Sofern innerhalb einer negierten Phrase ein Leerzeichen vorkommt, ist dies Teil der Phrase und die Negation endet nicht. Negative Phrasen stellen wie negierte Terme ein Filterkriterium

dar und signalisieren bei einer positiven Übereinstimmung den Ausschluss des betrachteten Dokuments.

5 Gegenüberstellung und Vergleich anderer Volltextsuchen

Neben MongoDB existieren diverse andere Systeme, die ebenfalls Volltextsuchen in ähnlicher Form unterstützen. Die bekanntesten und größten Vertreter sind u.a. Elasticsearch [BVd] und SolR[Fouo]. In diesem Kapitel werden die Volltextsuchen dieser beiden Vertreter mit der Volltextsuche von MongoDB verglichen und - sofern gegeben - Vor- und Nachteile herausgearbeitet. Den Abschluss bildet eine Gegenüberstellung der Volltextsuchen von klassischen relationalen Datenbanken wie z.B. MySQL.
Um diese Gegenüberstellungen in einer repräsentativen Form darzustellen, werden in den folgenden Abschnitten die verschiedenen Fähigkeiten, Funktionsweisen, Designentscheidungen und Paradigmen der zu den Volltextsuchen gehörenden Komponenten beschrieben. Den Abschluss jeder Gegenüberstellung bildet ein Vergleich daraufhin, ob die beschriebene Volltextsuche eine Alternative Lösung zu MongoDB in Orestes darstellen könnte.

5.1 Elasticsearch

Elasticsearch ist eine vom Unternehmen *Elastic* entwickelte, REST(**Re**presentational **S**tate **T**ransfer)-basierte Such-Engine. Genau wie SolR basiert Elasticsearch auf Apache Lucene und steht unter der Apache Lizenz. Erschienen im Jahr 2010 liegt Elasticsearch momentan (Stand 14.12.2017) in der Version 6.0 vor. Wie die meisten NoSQL-Vertreter ist auch Elasticsearch dokumentorientiert und schemalos. Dokumente, die in der laufenden Elasticsearch Instanz gespeichert werden sollen, werden als JSON übergeben und als solches auch gespeichert - im Unterschied zu MongoDB. MongoDB verwendet ein eigenes, auf JSON aufbauendes Datenformat namens BSON. Die Unterschiede zwischen BSON und JSON werden im Abschnitt 5.1.2 kurz erläutert. Da Elasticsearch einen Schwerpunkt auf verteilte Daten legt, werden eingehende Dokumente auch unter hohen Input/Output Belastungen schnell indexiert[BVn]. Dies hat zur Folge, dass eingehende Dokumente sehr schnell als Suchergebnisse zur Verfügung stehen. Auf Grund dieser Eigenschaft sprechen die Entwickler von Elasticsearch von der Möglichkeit zur Echtzeit-Suche[BVn].

5.1.1 Sharding

Die Persistierung und damit physische Speicherung in Elasticsearch erfolgt in mehreren als *Primary Shards* bezeichneten Lucene-Indizes[BVt]. Das als *Sharding* bezeichnete Verfahren zur Datenverteilung über mehrere separate physische oder virtuelle Maschinen unterstützt auch MongoDB. Wie in MongoDB lässt sich auch in Elasticsearch ein Index auf mehrere verschiedene dieser Maschinen aufteilen. Dies dient einer verbesserten Lastverteilung, da somit die parallele Ausführung und Bearbeitung (komplexer) Suchanfragen ermöglicht wird. Zusätzlich lassen sich die einzelnen *Shards* auf unterschiedliche Knoten eines sogenannten *Clusters* verteilen, damit auch eine verbesserte physische Lastverteilung erreicht werden kann. Die Verteilung auf verschiedenen Knoten eines Clusters ist die fundamentale und ursprüngliche Philosophie von Elasticsearch und ist nach dem Master-Slave Ansatz konzipiert[BVi].

5.1.2 Unterschiede zwischen BSON und JSON

JSON ist eine aus Javascript stammende Objekt-Notation (JSON steht für **J**ava**s**ript **O**bject **N**otation) und hat das Ziel, ein einheitliches und leicht verständliches Format für sowohl Mensch als auch Maschine darzustellen. JSON wurde ursprünglich ab 2002 von Douglas Crockford entwickelt[Ltd17a] und zwischen 2005 und 2007 von Yahoo als eines der ersten Unternehmen zum Austausch von Daten benutzt[Yah07]. JSON zählt heute mit zu den verbreitetsten Kommunikationsformaten zum Austausch von Daten und Informationen im World Wide Web und wird seit 2012 (Siehe Abbildung 5.1) auch häufiger eingesetzt als das etablierte XML. Dokument-orientierte Datenbanken, wie MongoDB oder Elasticsearch, benutzen JSON um Daten in einer ähnlichen Form darzustellen, wie relationale Datenbanken mit Tabellen. MongoDB repräsentiert JSON Dokumente in einem eigenen, binärkodierten Format, genannt BSON - eine Kurzform für Binary JSON. BSON erweitert JSON dahingehend, dass zusätzliche Datentypen, geordnete Felder und eine effiziente Enkodierung und Dekodierung in verschiedenen Programmiersprachen möglich ist[bso].

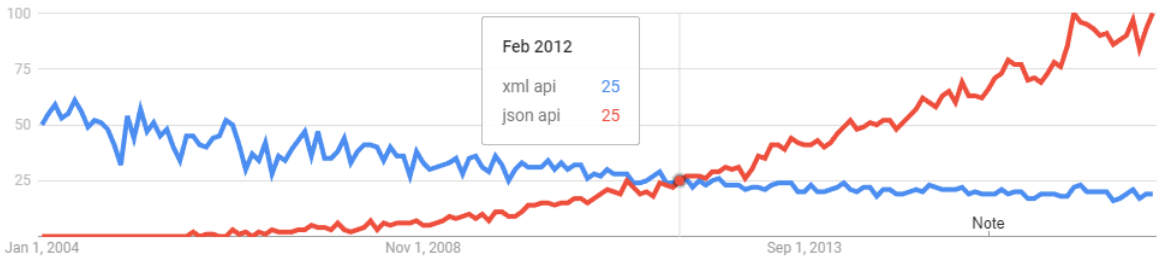


Abbildung 5.1: Entwicklung der Beliebtheit von JSON-API's im Vergleich zu XML [Treb]

5.1.3 Metadaten

Wie MongoDB[Monf][Kva16] verwendet auch Elasticsearch Metadaten, um Zuordnungen und Informationen über Dokumente zu erfassen. Elasticsearch hat drei notwendige Metadaten[BVc]:

1. *_index*: Der Index beschreibt eine Ansammlung gleichartiger Dokumente, die gruppiert werden sollen. Der Index kann entweder manuell oder durch Elasticsearch automatisch vergeben werden. Elasticsearch definiert einige Regeln, wie ein Index auszusehen hat: Er muss kleingeschrieben sein, darf keine Kommata enthalten und darf nicht mit einem Unterstrich beginnen.

2. *_type*: Sofern *_index* als die Kategorie der Dokumente betrachtet wird, so kann *_type* als die Unterkategorie bezeichnet werden. Diese Unterkategorie gruppiert Dokumente, die identische oder sehr ähnliche Schemata aufweisen. Zum Beispiel könnte Amazon alle Produkte in einem gleichnamigen Index zusammenfassen, aber jede spezifische Kategorie der Geräte, wie z.B. Elektronik, wäre ein eigener *_type*. Auch hierfür gibt es von Elasticsearch definierte Regeln: Die Unterkategorie darf groß- oder kleingeschrieben sein, sollte nicht mit einem Unterstrich oder Punkt beginnen und sollte ebenfalls keine Kommata verwenden. Des Weiteren darf *_type* nicht mehr als 256 Zeichen enthalten.

3. *_id*: Die Id identifiziert - zusammen mit *_index* und *_type* - ein Dokument auf eine einzigartige Art und Weise. Bei der Erstellung eines neuen Dokuments kann die Id entweder manuell oder durch Elasticsearch generiert werden.

5.1.4 Dokument-Schema

Obwohl Elasticsearch im Grunde schemalos ist, versucht Elasticsearch jedem Feld einen spezifischen Typen zuzuordnen. Dieses sogenannte *Mapping*[BVh] wird mithilfe einiger Heuristiken durchgeführt. Zum Beispiel kann anhand verschiedener Datumsformate erkannt werden, dass es sich bei einem Feld um ein Datum handelt. Bei einem anderen Feld kann z.B. durch einen nummerischen Abgleich festgestellt werden, dass es sich bei dem enthaltenen Wert um eine Zahl handelt. Das *Mapping* wird automatisch von Elasticsearch durchgeführt, kann aber durch den Benutzer fixiert oder gänzlich eigenständig konfiguriert werden. Felder, die nicht zugeordnet werden können, erhalten den Typ *string*.
Elasticsearch identifiziert durch das stattfindende Mapping, in welche Kategorie ein Feld gehört. Dabei wird zwischen *exakten Werten* und *Volltext* unterschieden[BVe]. In die Kategorie *Volltext* fällt jedes Feld, das nicht zugeordnet werden konnte oder anderweitig den Typ *string* ohne vom Nutzer fixierte zusätzliche Informationen besitzt. Ein Volltext enthält generell eine textuelle Repräsentation, die i.d.R. in einer natürlichen Sprache verfasst ist. Texte in natürlicher Sprache werden zusammen mit anderen textuellen Repräsentationen allgemein als unstrukturierte Daten aufgefasst. Dies ist allerdings eine Fehlbezeich-

nung, denn eine natürliche Sprache hat und muss einer festgelegten Struktur folgen. Das Problem ist vielmehr, dass einige hundert verschiedene Sprachen existieren und die Regeln für diese Sprachen komplex und i.d.R. auch mit speziellen Ausnahmen versehen sind[BVe].
Elasticsearch unterscheidet durch diese Kategorisierung, inwiefern Felder auf Übereinstimmungen überprüft werden. So sind z.B. die exakten Werte „Foo“ und „foo“ vollständig unterschiedlich, während sie als Volltext wiederum eine Übereinstimmung besitzen würden. Bei exakten Werten ist die Prüfung der Übereinstimmung eine Unterscheidung zwischen Ja und Nein, während Volltextsuchen in Relevanz bemessen werden[BVe].

5.1.5 Indizes

Volltext-Indizes werden in Elasticsearch *Inverted-Index*[BVg] genannt und sind für eine effiziente Volltextsuche konzipiert. Ein Inverted-Index besteht aus einer Liste aller eindeutiger Worte eines jeden Feldes. Für jedes dieser Worte gibt es wiederum eine Zuweisungsliste, um die Wörter mit den Dokumenten, in denen sie vorkommen, zu referenzieren. Ein Beispiel[1]: Angenommen, es existieren zwei Dokumente mit den folgenden Inhalten:

1. The quick brown fox jumped over the lazy dog
2. Quick brown foxes leap over lazy dogs in summer

Zunächst trennt Elasticsearch beide Inhalte in einzelne Einheiten, auch Token oder schlicht Term genannt, auf. Dies passiert mithilfe eines Lucene-Analyzers, der die lexikalische Analyse und optional auch die Eliminierung von Stopwörtern sowie das Stemming ausführt. Die Worte, die durch die einzelnen Token repräsentiert werden, werden anschließend in einer Menge zusammengefasst, so dass keine doppelten Einträge für dasselbe Wort existieren. Abschließend wird eine Zuweisungsliste erstellt, die die resultierenden Wörter mit den Dokumenten verknüpft, in denen sie vorkommen. Für das obige Beispiel könnte das folgendermaßen aussehen:

[1] Beispiel entnommen aus [BVg]

Term	1. Dokument	2. Dokument
Quick	✗	✓
The	✓	✗
brown	✓	✓
dog	✓	✗
dogs	✗	✓
fox	✓	✗
foxes	✗	✓
in	✗	✓
jumped	✓	✗
lazy	✓	✓
leap	✗	✓
over	✓	✓
quick	✓	✗
summer	✗	✓
the	✓	✗

Wird nun nach *quick brown* gesucht, werden diese beiden Worte in der obigen Liste nachgeschlagen und alle Dokumente, die eines dieser Worte referenzieren, als Suchergebnis zurückgegeben:

Term	1. Dokument	2. Dokument
brown	✓	✓
quick	✓	✗
Treffer	2	1

Wie zu erkennen ist, beinhalten beide Dokumente die gesuchten Worte. Das 1. Dokument enthält jedoch mehr Treffer bzgl. der Suchanfrage und ist damit als relevanter einzustufen.

Auf Grund der Tatsache, dass kein Stemming vorgenommen wurde und damit auch keine Umwandlung in die Kleinschreibweise stattfand, werden morphologisch verwandte Worte mitunter nicht als das gleiche Worte identifiziert. Wie der ersten Tabelle zu entnehmen ist, werden *Quick* und *quick* als separate Worte aufgeführt, obwohl sie sich nur in der Groß- und Kleinschreibung unterscheiden. Ähnliches gilt für *dog* und *dogs* sowie *fox* und *foxes*, denn diese Worte entspringen demselben Wortstamm. Durch die Auswahl einer anderen Analyzer-Konfiguration kann dieser Umstand allerdings leicht korrigiert werden. Im Gegensatz zu MongoDB, wo die lexikalische Analyse und der Stemming-Prozess standardmäßig durchgeführt und durch Optionen (siehe dazu Abschnitt 4.7) beeinflusst werden können, werden in Elasticsearch die Analyzer selbst spezifiziert. Dazu stellt Elasticsearch einige Built-In Analyzer von Lucene direkt zur Verfügung und erlaubt auch die Kombination und Konfiguration eigener, sogenannter *Custom* Analyzer, z.B. mittels der folgenden Definition[BVb]:

```
PUT /my_index
{
    "settings": {
        "analysis": {
            "char_filter": { ... custom character filters ... },
            "tokenizer":   { ...    custom tokenizers     ... },
            "filter":      { ...   custom token filters   ... },
            "analyzer":    { ...    custom analyzers      ... }
        }
    }
}
```

Durch diese direkte Kontrolle über Lucene lassen sich durch Filter-Konfigurationen auch Synonyme als verwandte Wörter definieren. So kann z.B. das Wort *leap* als Synonym für *jump* auf eben dieses „übersetzt“ werden. Dadurch ist die Volltextsuche von Elasticsearch in der Lage, bei einer Suchanfrage nach *leap* oder nach *jump* auch diejenigen Dokumente als Suchergebnis zu identifizieren, die eines oder beide Worte enthalten. Selbiges ist in MongoDB zum gegenwärtigen Zeitpunkt (Stand 18.12.2017) nicht möglich. Auch verfügt MongoDB über keine derartige fein granulare Kontrolle.
Nach der Auswahl eines entsprechenden Analyzer, der das Stemming und die Eliminierung von Stopwörtern korrekt durchführt, würde die Tabelle den erwarteten Inhalt aufweisen:

Term	**1. Dokument**	**2. Dokument**
quick	✓	✓
brown	✓	✓
dog	✓	✓
fox	✓	✓
jump	✓	✓
lazy	✓	✓
summer	✗	✓

Zu beachten ist die Synonym-Übersetzung: $leap \rightarrow jump$

Wird nun nach *quick brown* gesucht, kommt das folgende unverfälschte Resultat zum Vorschein:

Term	**1. Dokument**	**2. Dokument**
brown	✓	✓
quick	✓	✓
Treffer	2	2

Beide Dokumente weisen die gleiche Relevanz bezüglich der Suchanfrage auf.

5.1.6 Volltextsuche

Elasticsearch unterscheidet zwischen der einfachen *Query-String*[BVs] und der umfassenderen *request body* Such-Methode[BVf]. Letztere erwartet ein komplettes JSON Dokument und benutzt zur Beschreibung der Suchanfrage eine umfangreiche Query-DSL[BVp] (**D**omain **S**pecific **L**anguage). Der offiziellen Elasticsearch Dokumentation zufolge[BVs] wird die *Query-String* Methode für sogenannte *ad hoc queries* empfohlen - also beispielsweise zum Testen oder zu Übungszwecken. Für eine Volltextsuche hingegen bietet es sich an, die weitaus „mächtigere“ *request body* Methode zu bevorzugen. Die Dokumentation sagt dazu folgendes[BVf]:

> „The high-level full text queries are usually used for running full text queries on full text fields like the body of an email. They understand how the field being queried is analyzed and will apply each field's analyzer (or search_analyzer) to the query string before executing.“ (Elasticsearch Dokumentation)

Auf Grund dessen wird auf die *Query-String* Methode nicht weiter eingegangen.
Des Weiteren differenziert Elasticsearch Volltext- und Term-basierte Anfragen. Term-basierte Anfragen werden auf den erstellten *Inverted-Index* ausgeführt und haben nur eine Übereinstimmung mit *exakten* Werten (Siehe Abschnitt 5.1.4). Groß- oder kleingeschriebene Varianten oder morphologisch verwandte Wörter werden nicht als Übereinstimmung betrachtet. Für derartige Unterfangen sind Volltext-Queries zuständig, weswegen auf Term-basierte Queries ebenfalls nicht weiter eingegangen wird.

Volltext-Query Gruppen

Im Gegensatz zu MongoDB, wo Terme, Phrasen und entsprechende Negationen gemixt in einer Query-Anfrage existieren können, differenziert Elasticsearch sehr viel strikter. Elasticsearch teilt die Queries, die für eine Volltextsuche verwendet werden können, in insgesamt sieben verschiedene Gruppen auf:

1. *match*-Query[BVl]: Diese Query-Operation nimmt Texte, nummerische Werte oder Daten entgegen und konstruiert aus dem übergebenen JSON einen Query. Ein Beispiel[2]:

```
{
    "query": {
        "match": {
            "message": "this is a test"
        }
    }
}
```

[2] Beispiel entnommen aus [BVl]

Der enthaltene Text „this is a test“ wird analysiert (Zur Erinnerung: Der Analyse-Prozess in Elasticsearch fasst die lexikalische Analyse, das Filtern von Stopwörtern sowie das Stemming zusammen) und in eine boolesche-Abfrage überführt. D.h. die nicht gefilterten und nunmehr gestemmten Worte *this* und *test* werden miteinander durch einen Operator verknüpft und müssen der Definition des Operators entsprechend entweder beide in dem angegebenen Feld *message* vorkommen (**And**-Operator) oder es reicht das Vorkommen eines der Worte (**Or**-Operator). Der **Or**-Operator ist die Standardeinstellung. Dies kann durch die Angabe von „and“ bzw. „or“ im JSON-Feld *operator* vom Benutzer konfiguriert werden:

```
{
    "query": {
        "match": {
            "message": "this is a test",
            "operator": "and"
        }
    }
}
```

2. *match_phrase*-Query[BVk]: Dieser Query-Operator nimmt einen Text entgegen und konstruiert daraus einen Phrase-Query. Im Gegensatz zu MongoDB, wo das Phrase-Matching strikt ist, kann in Elasticsearch durch die Angabe von $slop$ konfiguriert werden, inwieweit eine Unstimmigkeit der Phrase vorliegen kann. $slop$ definiert diese Unstimmigkeit als nummerischen Wert, der die Länge der Unstimmigkeit repräsentiert. Standardmäßig ist das Verhalten allerdings identisch zu dem von MongoDB, da $slop$ ohne explizite Angabe auf den Wert 0 gesetzt wird.
   Ein Beispiel[3] für einen *match_phrase*-Query könnte wie folgt aussehen:

```
{
    "query": {
        "match_phrase": {
            "message": "this is a test"
        }
    }
}
```

3. *match_phrase_prefix*-Query[BVj]: Dieser Query-Operator agiert beinahe identisch zum *match_phrase*-Query, erlaubt jedoch eine sehr simple Vervollständigung des letzten Wortes des Suchtextes:

```
{
```

[3] Beispiel entnommen aus [BVk]

```
    "query": {
        "match_phrase_prefix" : {
            "message": "quick brown f"
        }
    }
}
```

Der aus der obigen Query-Operation resultierende Query erhält durch die nummerische Konfiguration $max\_expansions$ die angegebene Anzahl an Suffix-Vervollständigungen. Für die obige Angabe „quick brown f“ werden die ersten $max\_expansions$ eindeutigen Terme, die mit einem *f* beginnen, des *Inverted Index* an die angegebene Phrase angehängt. Anschließend werden diese $max\_expansions$ Vervollständigungen als eine Oder-Bedingung miteinander verknüpft. Ein Beispiel: Angenommen $max\_expansions = 2$ und die ersten beiden eindeutigen Terme, die mit *f* beginnen, sind *fox* und *fire*. Dann würde der resultierende Query die Phrase „quick brown fox“ und auch „quick brown fire“ als Übereinstimmungen erkennen, allerdings nicht „quick brown frost“, da *frost* nicht zu den ersten $max\_expansions = 2$ Termen gehört. Standardmäßig hat die Option $max\_expansions$ den Wert 50.

4. *multi_match*-Query[BVm]: Der *multi_match*-Query Operator dient dazu multiple *match*-Operatoren auf verschiedene Felder zu definieren. Zum Beispiel[4] folgendermaßen:

```
{
    "query": {
        "multi_match": {
            "query": "this is a test",
            "fields": ["subject", "message"]
        }
    }
}
```

Der *multi_match*-Query Operator kann durch die Angabe der Option *type* auf verschiedene *match*-Arten angewendet werden, auf die allerdings nicht weiter im Detail eingegangen wird. Für weitere Informationen sei auf die Dokumentation[5] von Elasticsearch verwiesen.

5. *common_terms*-Query[BVa]: Der *common_terms*-Query Operator behebt ein signifikantes Problem der sonstigen *match*-Operatoren in Elasticsearch: Eine Suche nach „The brown fox“ führt - im Gegensatz zu MongoDB - für jedes der angegebenen Worte eine Anfrage auf alle Dokumente aus[BVa]. Eine Lösung, um die Worte in der

[4] Beispiel entnommen aus [BVm]

[5] https://www.elastic.co/guide/en/elasticsearch/reference/master/query-dsl-multi-match-query.html#multi-match-types

Suchanfrage zu reduzieren - die u.a. auch MongoDB benutzt - ist, die Suchanfrage zu analysieren (also u.a. die Stopwörter zu filtern), so dass die Suchanfrage entsprechend transformiert wird: „The brown fox“ $\rightarrow$ „brown fox“. Für Elasticsearch würde sich dadurch die Anzahl an auszuführenden Anfragen reduzieren: von x ursprünglichen Worten zu $z = x - y$, wobei y die Anzahl an Stopwörtern darstellt. Auf den vorliegenden Fall übertragen: $x = 3$, $y = 1$, $z = x - y = 3 - 1 = 2$. Die Entwickler von Elasticsearch stellen jedoch die Behauptung auf, dass die Filterung von Stopwörter Einfluss auf die Relevanz der Suchergebnisse hat. Stopwörtern - so die Meinung der Elasticsearch Entwickler - können eine nicht zu ignorierende Relevanz der Suchanfrage ausmachen[BVa]:

> „The problem with this approach[Der Eliminierung von Stopwörtern] is that, while stopwords have a small impact on relevance, they are still important. If we remove stopwords, we lose precision, (eg we are unable to distinguish between ‚happy‘ and ‚not happy‘) and we lose recall (eg text like ‚The The‘ or ‚To be or not to be‘ would simply not exist in the index).“ (Elasticsearch Dokumentation)

Aus diesem Grund wird bei Verwendung des *common_terms*-Query Operators von einer Stopwort-Filterung gänzlich abgesehen und eine Einteilung der enthaltenen Worte in zwei Gruppen durchgeführt:

a) Häufig vorkommende Worte (im Original *high frequency terms*)

b) Selten vorkommende Worte (im Original *low frequency terms*)

Die *low frequency terms* sind diejenigen Worte, die ansonsten als Stopwörter gefiltert worden wären. Die Einteilung der Worte wird auf Basis ihrer Vorkommen (auch *frequency* genannt) in allen existierenden Dokumenten vorgenommen. Durch die Option $cutoff\_frequency$ kann dieser Wert konfiguriert werden. Ein Beispiel[6]:

```
{
    "query": {
        "common": {
            "body": {
                "query": "this is bonsai cool",
                "cutoff_frequency": 0.001
            }
        }
    }
}
```

Alle existierenden Worte, die eine Häufigkeit von weniger als $0.001 = 0.1\%$ aufweisen, werden in die Kategorie *low frequency terms* eingestuft, alle übrigen in die

[6]Beispiel entnommen aus [BVa]

Kategorie *high frequency terms*[BVu].
Nach dieser Einteilung erfolgen zwei Such-Durchläufe: Zunächst wird nach den Dokumenten gesucht, die Übereinstimmungen mit den *high frequency terms* aufweisen. Anschließend erfolgt ein zweiter Such-Durchlauf mit den *low frequency terms*, in diesem Fall allerdings nur auf den bereits gefundenen Dokumenten. Dadurch kann der Text-Score der Dokumente positiv verändert und die Relevanz dadurch korrigiert werden.

6. *query_string*: Dieser Operator erlaubt die Verwendung der *query_string* Such-Methode als einen einzigen, zusammenhängenden Query, welcher mit *And-*, *Or-* und *Not-*Bedingungen entsprechend angereichert werden kann. Da auf den Query-String in dieser Arbeit nicht näher eingegangen wird, wird auch dieser Operator nicht weiter im Detail beschrieben. Für weiterführende Informationen sei auf die Dokumentation[7] von Elasticsearch verwiesen.

7. *simple_query_string*: Identisch zum *query_string*-Query Operator mit dem Zusatz, dass unter keinen Umständen eine Ausnahme/Fehler auf Grund einer fehlerhaften Angabe generiert wird. Fehlerhafte oder nicht zu verarbeitende Angaben werden schlicht ignoriert.

Query-DSL

Die von Elasticsearch entwickelte und benutzbare Query-DSL[BVp] basiert - wie alle Konfigurationen und Anfragen im Kontext von Elasticsearch - auf JSON. Die Query-DSL wird zur Beschreibung von Anfragen verwendet und ist grundsätzlich eine explizite Definition eines AST (**A**bstract **S**yntax **T**ree)[8] der aus zwei unterschiedlichen Klauseln besteht:

1. *Leaf query clauses*: Blatt-Queries sind Queries auf der untersten Ebene des AST. Blatt-Queries sind und müssen eigenständig ausführbare Query-Ausdrücke sein und suchen in einem bestimmten Feld nach einer Übereinstimmung mit einem festgelegten Wert.

2. *Compound query clauses*: Zusammengesetzte Queries bestehen aus einer beliebigen Anzahl an Blatt-Queries oder wiederum aus anderen zusammengesetzten Queries. Letzteres definiert einen logischen Zusammenhang der geschachtelten Queries, z.B. um eine Bedingung zu formulieren oder um das Verhalten der geschachtelten Queries zu manipulieren.

[7] https://www.elastic.co/guide/en/elasticsearch/reference/master/query-dsl-query-string-query.html
[8] http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html

Realtime Queries: Percolator-Query

Elasticsearch bietet durch die Verwendung von sogenannten *Percolator*-Queries[BVo] die Möglichkeit, Suchanfragen zu speichern. Dadurch kann - nachdem neue Dokumente indexiert wurden – festgestellt werden, ob diese Dokumente Übereinstimmungen mit den zuvor gespeicherten Suchanfragen besitzen. Dadurch lassen sich u.a. News-Feeds, Newsletter und andere Formen von Benachrichtigen elegant umsetzen. *Percolator*-Queries ermöglichen es - vereinfacht ausgedrückt - Queries zu indexieren und anschließend Dokumente gegen diese indexierten Queries zu vergleichen. Dadurch kann bestimmt werden, ob eine Übereinstimmung mit dem Dokument und einem der Queries existiert. Während normalerweise Dokumente indexiert und mit Queries nach Übereinstimmungen in diesen Dokumenten gesucht werden, entsprechen *Percolator*-Queries dem genauen Gegenteil und werden daher auch als eine umgekehrte Suche bezeichnet. Ein Beispiel für die Nutzung eines *Percolator*-Queries ist z.B. ein klassischer News-Feed: Ein Benutzer registriert sich für bestimmte Themen, was der Registrierung eines *Percolator*-Queries entspricht. Sobald ein neuer Beitrag erscheint, findet eine Überprüfung des Beitrags-Dokuments gegen alle registrierten *Percolator*-Queries statt. Werden Übereinstimmungen gefunden, kann der Benutzer umgehend über das Erscheinen eines für ihn interessanten Beitrags informiert werden - in Echtzeit. Das oben beschriebene Szenario eines News-Feeds könnte folgendermaßen umgesetzt werden[BVo]:
Zunächst wird ein entsprechender Text-Index erzeugt:

```
{
    "mappings": {
        "doc": {
            "properties": {
                "message": {
                    "type": "text"
                },
                "query": {
                    "type": "percolator"
                }
            }
        }
    }
}
```

Anschließend wird ein *Percolator*-Query für das Feld *message* wie folgt registriert:

```
{
    "query": {
        "match": {
```

```
            "message": "Elasticsearch"
        }
    }
}
```

Sobald ein Beitrag erscheint, der im Feld *message* das Wort *Elasticsearch* enthält, soll eine Übereinstimmung mit diesem Query gefunden werden. Wird nun ein solches Dokument geprüft bzw. *percolated*:

```
{
    "query": {
        "percolate": {
            "field": "query",
            "document": {
                "message": "A new version of Elasticsearch!"
            }
        }
    }
}
```

resultiert die Überprüfung in dem folgenden *Response*:

```
{
    // ...
    "hits": {
        "total": 1,
        "max_score": ...,
        "hits": [
            {
                "_index": "my-index",
                "_type": "doc",
                "_id": "1",
                "_score": ...,
                "_source": {
                    "query": {
                        "match": {
                            "message": "Elasticsearch"
                        }
                    }
                }
            }
        ]
    }
```

```
}
```

Damit wäre eine positive Übereinstimmung gefunden.

Vergleich mit MongoDB

Elasticsearch besitzt eine feingranulare Volltextsuche, die in nahezu allen Bereichen wesentlich mehr Konfigurationsmöglichkeiten als MongoDB bietet. Die Durchführung von Volltextsuchen ist in Elasticsearch anders umgesetzt als in MongoDB: Die beschriebenen Volltext-Query-Gruppen besitzen eine große Auswahl an Optionen und Konfigurationen und damit eine nicht zu unterschätzende Komplexität. Gleichermaßen wird durch die Auftrennung in einzelne Gruppen eine klare Abgrenzung zwischen den einzelnen Aufgabengebieten erreicht. Die Volltextsuche in MongoDB ist wesentlich einfacher konzipiert aber auch limitierter, womit sie allerdings auch leichter zu erlernen ist. Durch die fehlende Trennung in einzelne Gruppen wie in Elasticsearch werden Phrasen, Terme und die entsprechenden Negationen in MongoDB in einer Abfrage zusammengefasst. Durch diese Zusammenfassung ist die anschließende interne Identifizierung der einzelnen Komponenten mit einem größeren Aufwand verbunden, als es in Elasticsearch der Fall ist, da diese bereits strukturiert vorliegen müssen. *Or*- und *And*-Verknüpfungen setzt MongoDB mit der Differenzierung zwischen einzelnen Termen und ganzen Phrasen um, und *Not*-Operationen durch entsprechende Negationen.
Ein vermeintlicher Vorteil von MongoDB geht aus der Beschreibung der *common_terms*-Query-Gruppe hervor: Elasticsearch durchsucht für jedes der vorkommenden Worte alle betreffenden Dokumente, während MongoDB nur eine Abfrage an alle Dokumente stellt. Dies ist allerdings hinsichtlich der verwendeten *Inverted-Indizes* kein wirklicher Nachteil, da die Suchwörter direkt mit den Dokumenten, die sie enthalten, verknüpft sind (Siehe Abschnitt 5.1.5). Bevor mit Version 2.4[Mon13] ein experimenteller Support für Volltextsuchen in MongoDB eingeführt wurde, riet die MongoDB Dokumentation für derartige Unterfangen manuell ein Array-Feld mit Schlüsselwörtern pro Dokument zu führen, nach denen gesucht werden kann[Mong]. MongoDB verwendet B-Bäume[Mond] für sämtliche Indizes. Die Verwendung von B-Bäumen in Kombination mit Volltextsuchen ist ein Nachteil von MongoDB gegenüber Elasticsearch. Dieser Nachteil beruht auf der Tatsache, dass ein Inverted-Index als ein Assoziatives Array - auch *Map* genannt - dargestellt werden kann (Siehe auch Abschnitt 5.1.5). Daraus geht - sofern perfektes *Hashing* vorausgesetzt wird - eine Zugriffszeit von $\mathcal{O}(1)$ hervor. Die Zugriffszeit eines B-Baums liegt dagegen bei durchschnittlich $\mathcal{O}(log_2(n))$. Daher ist Elasticsearch bezüglich Volltextsuchen unter idealen Umständen die effizientere Wahl.

Eine inkrementelle Volltextsuche ist in Elasticsearch ebenso möglich wie in MongoDB und könnte daher auch in InvaliDB verwendet werden. Vereinfacht ausgedrückt würde der Such-Query nach der Ermittlung der passenden Suchergebnisse gespeichert und

alle nun eingehenden Dokumente - nach der Indexierung - auf eine Übereinstimmung geprüft werden. Durch dieselbe Auswertung wie in InvaliDB (Abb. 2.3) könnte anschließend identifiziert werden, welcher *matching*-Status vorliegt. Durch dieses Vorgehen könnten die ermittelten Suchergebnisse in Echtzeit aktualisiert werden. Durch den Einsatz von *Percolator*-Queries ist dieses Verfahren bereits grundsätzlich implementiert, wobei der Abgleich des *matching*-Status selbständig implementiert werden müsste: Derzeit kann für neu hinzukommende Dokumente geprüft werden, ob diese einer zuvor gespeicherten Suchanfrage entsprechen. Es kann allerdings kein Rückschluss darüber erfolgen, ob es bereits zuvor eine Übereinstimmung mit der Suchanfrage gab. Daher kann nicht bestimmt werden, ob ein *add*-, *change*- oder *remove*-Event vorliegt. Diese Informationen müssten in einer eigenen Implementierung separat gespeichert und abgefragt werden können, um eine Funktionalität wie in InvaliDB zu gewährleisten.

5.2 SolR

SolR (ausgesprochen „Solar“) ist eine als Open-Source vorliegende Suchmaschine die dem Apache Lucene Projekt unterstellt ist[Fouo]. SolR liegt derzeit (Stand 02.01.2018) in Version 7.2.0 vor. Besondere Merkmale von SolR sind Volltextsuchen, Realtime-Indexing und Dynamisches Clustern sowie sogenanntes *rich document handling*[Foup]. Letzteres ist eine Spezialität von SolR und ermöglicht es, den Inhalt von PDF's, Word-Dokumente u.v.w. zu indexieren und in die Datenbank zu übernehmen. Der Fokus von SolR liegt auf Skalierbarkeit und Ausfallsicherheit[Foup]. SolR basiert - wie Elasticsearch - auf Lucene [Foup]. Erstmals im Jahr 2004 von Yonik Seeley als internes Projekt von CNET-Networks implementiert, wurde SolR bereits zwei Jahre später der Apache Software Foundation übergeben [Fou06], die es 2010 mit dem Apache Lucene Projekt zusammenlegten[Fou10]. Seitdem wird es auch als Lucene/SolR bzw. SolR/Lucene Projekt bezeichnet[Fou10]. Während SolR anfangs als die wohl mit Abstand meist verwendete Suchmaschine galt, wurde sie bereits im Jahr 2014 von dem erst 2010 gegründeten Unternehmen *Elastic* mit deren Suchmaschine Elasticsearch überholt (siehe Abb. 5.2).

Abbildung 5.2: Entwicklung der Beliebtheit von SolR im direkten Vergleich zu Elasticsearch [Trec]

Für den damaligen Siegeszug von Elasticsearch gibt es mehrere mögliche Gründe: Elasticsearch war moderner, schemalos und besaß eine wesentlich besser situierte Vertei-

lung. Zusätzlich unterstützte Elasticsearch **N**ear-**R**eal-**T**ime-Suchen. Diese Fähigkeit war bereits seit der ersten Version fester Bestandteil von Elasticsearch, während SolR *NRT*-Suchen erst seit Version 4.0 (Ende 2012) unterstützt[Fou12]. *NRT*-Suchen basieren - vereinfacht ausgedrückt - auf dem Konzept, dass Dokumente unmittelbar nachdem sie indexiert wurden sichtbar sind. Mit anderen Worten: sobald ein Dokument indexiert wurde, kann nach dessen Inhalt gesucht und das Dokument gefunden werden. Es wird dabei zwischen *Soft-* und *Hard-Commits* unterschieden: Ein *Hard-Commit* persistiert das Dokument von flüchtigen in nicht-flüchtigen Speicher, während ein *Soft-Commit* die Änderungen bzw. das Dokument nur sichtbar macht, aber die Daten noch nicht persistiert. SolR und Elasticsearch haben ähnliche, aber auf Grund der verschiedenen Architekturen jeweils eigene Ansätze, zur Realisierung von *NRT*-Suchen, auf die auf Grund ihres Umfangs nicht näher eingegangen wird.

5.2.1 Sharding

Die Dokumentation von SolR empfiehlt seit Version 4 die Verwendung von *SolrCloud*[Fouq]. Intern verwendet *SolrCloud* Apache ZooKeeper[Foun], während Elasticsearch eine eigene Implementierung namens *Zen*[BVv] verwendet.

5.2.2 Dokument-Schema

Im Gegensatz zu Elasticsearch und MongoDB benötigt SolR ein festes Schema für die Daten. Seit Version 4.3[Fou13b] unterstützt SolR allerdings das sogenannte *Managed-Schema*[Foum] - auch *Schemaless Mode* genannt. Wird diese Option aktiviert, versucht SolR die Datentypen der einzelnen Felder dynamisch zu ermitteln. Bis Version 4.2 musste das Schema jedoch manuell angelegt und wenn nötig angepasst werden. Das Schema von SolR deklariert[Tana]:

- welche Felder vorhanden sind und welchen Typ sie aufweisen
- welche Felder als Primär-Schlüssel verwendet werden
- welche Felder benötigt werden
- wie die einzelnen Felder indexiert und wie deren Inhalt bei einer Suche gefunden wird

Als Datentypen enthält SolR u.a.:

- float
- int
- date

- text

Zusätzlich können eigene Datentypen erstellt werden, indem die durch Lucene verfügbaren Filter oder Tokenizer entsprechend konfiguriert bzw. kombiniert werden. Um das Schema für ein bestimmtes Feld anzulegen findet die folgende Notation Anwendung[Fouf]:

```
<field name="price" type="float" default="0.0" indexed="true" />
```

Listing 5.1: Definition eines Schema für ein Preis-Feld

name ist der Name des Feldes, *type* ist der Feld-Typ (z.B. einer der oben aufgeführten) und die Option *indexed* definiert, ob dieses Feld indexiert werden soll. Wird ein Feld nicht indexiert, kann danach auch nicht gesucht werden[Fouf]. Die lexikalische Analyse, das Entfernen von Stopwörtern und das Stemming werden nur für Felder durchgeführt, die indexiert werden sollen[Tana]. Zur Durchführung einer qualitativ hochwertige Volltextsuche ist eine korrekte und konkrete Spezifikation der Daten unumgänglich. Elasticsearch versucht das Daten-Schema weitgehend durch Heuristiken selbst zu bestimmen, um Arbeit zu automatisieren (Siehe Abschnitt 5.1.4). Dies gelingt aber nicht immer und birgt das Risiko der Fehlinterpretation. Das ist der Grund, warum Elasticsearch den Benutzer optional durch das sogenannte *Mapping* das ermittelte Schema manuell fixieren lässt - um etwaige durch die Heuristiken falsch interpretierte Metadaten zu korrigieren. Nur durch ein vollständig valides Daten-Schema ist sichergestellt, dass die ermittelten Relevanzen der Ergebnisse optimal sind und die Indexierung sowie Analyse korrekt durchgeführt werden. Obwohl MongoDB auch als weitgehend schemafrei gilt, wird erst durch die Spezifizierung der Text-Indizes angegeben, welche Felder textuell verwertbaren und durchsuchbaren Inhalt aufweisen. Allerdings kann in MongoDB durch die Verwendung von *Wildcard Text Indexes* die Intention angegeben werden, alle Felder, die textuellen Inhalt aufweisen, als Text-Index zu spezifizieren (siehe Abschnitt 4.1).

5.2.3 Indizes

Wie Elasticsearch verwendet auch SolR *Inverted Indizes*[Tana][Jan14][Fouh]. Weitere Informationen können dem Abschnitt 5.1.5 entnommen werden.

5.2.4 Volltextsuche

Die von SolR unterstützte Anfragesprache bzw. *Query-Language* basiert weitgehend auf Lucene. Die *Query-Language* ist sehr simpel und bietet keine derartige Trennung und Kategorisierung wie etwa Elasticsearch. Im folgenden fünf Beispiele[9] der Query-Syntax in SolR[Tanb]:

[9] Beispiele entnommen aus [Tanb]

Term- und Phrasen-Suche

Nach einem bestimmten Term in einem bestimmten Feld kann gesucht werden, indem der Feldname und der Term durch einen Doppelpunkt getrennt angegeben werden. Um z.B. nach dem Term „SolR“ in dem Feld „titel“ zu suchen, müsste dies wie folgt angegeben werden:

```
titel:SolR
```

Um nach der Phrase „SolR versus Elasticsearch“ zu suchen, muss die Phrase - wie gewohnt - in Anführungszeichen eingebettet werden:

```
titel:"SolR versus Elasticsearch"
```

Sofern auch in anderen Feldern gesucht werden soll, müssen diese Felder separat festgelegt werden. Die einzelnen Spezifikationen werden durch die Angabe der Operatoren *AND* bzw. *OR* miteinander verknüpft. Diese Verknüpfungen können wiederum geklammert werden, um eine entsprechende Zusammenfassung zu ermöglichen. Sollen z.B. nur Dokumente gefunden werden, die im Titel die Phrase „SolR versus Elasticsearch“ **und** in dem Feld *message* den Term „Performance“ aufweisen, wird dies wie folgt definiert:

```
titel:"SolR versus Elasticsearch" AND message:Performance
```

Negationen werden - wie auch in MongoDB und Elasticsearch - durch einen führenden Bindestrich bzw. Minuszeichen gekennzeichnet. Sollen z.B. alle Dokumente gefunden werden, die das Wort „SolR“ aber nicht das Wort „Elasticsearch“ im Titel enthalten, hätte der Query die folgende Form:

```
titel:SolR -titel:Elasticsearch
```

Wildcards

SolR bietet die Möglichkeit, sogenannte *Wildcards* zu verwenden. *Wildcards* dienen als Platzhalter für eine beliebige Anzahl anderer Zeichen. Mittels *Wildcards* wird in SolR die Absicht angegeben, dass eine optionale, nicht näher spezifizierte Sequenz des Wortes am Ende oder mittendrin - allerdings nicht am Anfang[Tanb] - vorkommen kann. Dieses Vorgehen ähnelt der Definition eines Regulären Ausdrucks. Um alle Dokumente aufzulisten, deren Titel mit „SolR“ beginnen, kann der folgende Query verwendet werden:

```
titel:SolR*
```

Das * definiert - wie in MongoDB - den *Wildcard*. Es besagt, dass nach dem Term „SolR“ mehrere beliebige Zeichen vorkommen können, aber nicht müssen.
Ist die Absicht, alle Dokumente zu finden, deren Titel mit „Elastic“ beginnen und mit „search“ enden, kann dies folgendermaßen angegeben werden:

```
titel:Elastic*search
```

Wildcards in SolR ähneln den *match_phrase_prefix* Query-Operator aus Elasticsearch (Siehe Punkt 3 von Abschnitt 5.1.6), unterliegen jedoch nicht der Beschränkung, dass nur die ersten n eindeutigen Terme als Vervollständigung dienen können. Durch die Ähnlichkeit zu dem gleichartigen Regulären Ausdruck unterliegt die Wildcard-Spezifizierung von SolR überhaupt keinen Beschränkungen.

Suche mit Entfernungen

Wie Elasticsearch mit dem *match_phrase_prefix*-Query kann auch SolR Worte finden, die in einer bestimmten Nähe zueinander vorkommen. Dieses als *Proximity matching* bezeichnete Unterfangen kann wie folgt definiert werden:

```
titel:"Elastic search"~2
```

Dieser Query besagt, dass die Worte „Elastic“ und „search“ innerhalb einer Entfernung von 2 Worten zueinander stehen müssen, damit eine positive Übereinstimmung geschlussfolgert wird. Eine Angabe von ~0 ist äquivalent zu der Angabe, dass eine exakte Übereinstimmung vorliegen muss und ~1 ist gleichbedeutend mit einer möglichen Wort-Transposition.

JSON-API

Seit Version 5.0[Kuc15] unterstützt SolR die sogenannte *JSON Request API*[Foui]. Ein Beispiel einer solchen JSON-Anfrage könnte wie folgt aussehen[See15]:

```
{
    query: "titel:SolR"
}
```

Mit diesem *Request* werden alle Dokumente gefunden, die im Feld *titel* das Wort „SolR“ enthalten. Die äquivalente Anfrage bis Version 5.0 würde folgendermaßen aussehen:

```
<url>/solr/query?q=titel:SolR
```

Weitere Parameter müssen URL-kodiert angehängt werden, so dass die URL mit wachsender Anzahl an Parametern sehr lang und unübersichtlich aussieht. Beispielsweise würde die JSON-Abfrage

```
{
    query : "titel:\"SolR\" AND content:\"neue Version\""
}
```

wie folgt als URL-Anfrage aussehen:

```
<url>/solr/query?q=title:%22SolR%22%20and%20content:%22neue
    ↪ Version%22
```

Wobei es sich bei %22 um URL-kodierte Anführungszeichen und bei %20 um URL-kodierte Leerzeichen handelt.
Elasticsearch hingegen bietet bereits seit Version 0.90 eine JSON-API an[BVr].

Vergleich mit MongoDB

SolR ähnelt in vielen Punkten Elasticsearch, bringt aber weniger Flexibilität mit sich. So existiert z.B. keine Gruppierung bzw. Kategorisierung von Suchen. Im Gegensatz zu Elasticsearch konzentriert sich SolR primär auf Volltextsuchen, während Elasticsearch mit Elastic Stack[10] einen Fokus auf Log-Analysen hat und damit mehr als die reine Suche anbietet. Die Schemafreiheit von Elasticsearch im Vergleich zu SolR ist ebenso ein Punkt, der für die Fokussierung auf Log-Analysen spricht. Der dokumentenorientierte und schemafreie Ansatz von Elasticsearch ist sinnvoll, da im Bereich von Log-Analysen und generell in Big Data Szenarien mit vielen unterschiedlichen Informationen zu rechnen ist. Im Gegensatz zu MongoDB und genau wie Elasticsearch verwendet SolR für die Volltextsuche *Inverted Indizes*. Wie bereits in der Gegenüberstellung von Elasticsearch im Abschnitt 5.1.6 beschrieben, sind *Inverted Indizes* für eine Volltextsuche vorteilhafter als eine B-Baum Implementierung wie sie in MongoDB und auch vielen relationalen Datenbanken verwendet wird. Die nicht als Basis-Einstellung vorliegende Schemafreiheit von SolR kann zwiespältig betrachtet werden: Zum einen ist die explizite Angabe eines Schemas notwendig, um eine korrekte Volltextsuche zu ermöglichen. Zum anderen bringt diese Design-Entscheidung manuellen Mehraufwand mit sich. Generell kann die Entscheidung, das Schema explizit und manuell angeben zu müssen, trotz des daraus resultierenden Mehraufwandes, als positiv erachtet werden. Durch diese explizite Einstellung wird garantiert, dass die Volltextsuche nach den Wünschen des Benutzers funktioniert und keine etwaigen Fehler durch eine falsche heuristische Bestimmung des Datentyps eingeführt werden.
Obgleich SolR mit **N**ear-**R**eal**t**ime-Suchen ebenfalls eine Möglichkeit für Suchen in **beinahe** Echtzeit ermöglicht, sind Echtzeit-Queries nicht ohne größeren Aufwand implementierbar. Im Gegensatz zu Elasticsearch, welches durch *Percolator*-Queries (Siehe Abschnitt 5.1.6) bereits eine grundsätzliche Funktionalität für Echtzeit-Queries bietet, besitzt SolR keine derartige Eigenschaft. Dennoch wäre es prinzipiell möglich, Realtime-Queries - wie sie in InvaliDB existieren - auch auf Basis von SolR zu implementieren. Zuzüglich zum Abgleich des *matching*-Status müsste die Eigenschaft der *Percolator*-Queries - das Speichern von Suchanfragen und ein Abgleich dieser mit eingehenden Dokumenten nach der Indexierung - implementiert werden. Eine gleichwertige Funktionalität wie *Percolator*-Queries in Elasticsearch ist für eine zukünftige Version von SolR zwar bereits geplant[Fou13a], jedoch wurden die Arbeiten daran (Stand 07.01.2018) noch nicht begonnen. Da jedoch SolR ein sogenanntes *community-driven* Projekt ist[Foue] - also primär durch die Community entwickelt bzw. die Entwicklung durch die Community beein-

[10] https://www.elastic.co/de/products

flusst wird - finden Änderungen und Erweiterungen in SolR wesentlich schneller Einzug, als es in Elasticsearch der Fall ist. In Elasticsearch werden neue Eigenschaften, Änderungen und Vorschläge nicht durch die Community entschieden, sondern nur durch das Unternehmen *Elastic*. MongoDB als OpenSource Projekt begrüßt, wie SolR, stets Vorschläge für neue Merkmale und Fehlerbehebung[dta].

5.3 Relationale Datenbanken

Relationale Datenbanken wurden erstmals 1970 von Edgar F. Codd vorgeschlagen [Cod70a] [Cod70b] und sind auch gegenwärtig ein weit verbreiteter Standard für Datenbanken. Unter diesen vielen existierenden relationalen Datenbanksystemen gilt das 1994 entwickelte MySQL[Fiv09] als eines der beliebtesten und ebenso als eines der populärsten Open-Source Projekte[DE17]. MySQL zählte alleine bis 2013 50 Millionen Installationen weltweit[Sch13]. NoSQL (**N**ot **o**nly SQL) Datenbanken wie etwa MongoDB sollen Defizite der relationalen Datenbanken bewältigen. Zu diesen Defiziten zählen die Unflexibilität und die schwierige Skalierung.
Relationale Datenbanken bieten, je nach Implementierung, verschiedene Möglichkeiten, Volltextsuchen durchzuführen. Diese Arbeit wird aber nicht auf jede der einzelnen Volltextsuchen konkret eingehen. Stattdessen werden die grundsätzlichen und relevanten Eigenschaften von Volltextsuchen für drei populäre (Siehe Abb. 5.3) relationale Datenbank-Management-Systeme (kurz: DBMS) mit denen von MongoDB verglichen. Bei den ausgewählten DBMS handelt es sich auf Grund ihrer Popularität in den Jahren 2017/2018 um MySQL, PostgreSQL und Oracle.

Rang			DBMS	Datenbankmodell
Jan 2018	Dez 2017	Jan 2017		
1.	1.	1.	Oracle	Relational DBMS
2.	2.	2.	MySQL	Relational DBMS
3.	3.	3.	Microsoft SQL Server	Relational DBMS
4.	4.	4.	PostgreSQL	Relational DBMS

Abbildung 5.3: Die populärsten relationalen DBMS 2017/2018 [DE17]

5.3.1 Dokument-Schema

Relationale Datenbanken besitzen immer ein festgelegtes Schema. Die Daten werden in Form von Tabellen gespeichert, wobei jede Zeile der Tabelle einen eigenen Datenbank-Eintrag darstellt. Die Zellen bzw. Spalten dieser Zeile sind die Felder des Eintrags und haben einen feststehenden und nicht (leicht) zu ändernden Datentyp. Bei der Erstellung einer Tabelle wird das Schema bzw. die Struktur, also u.a. die Feldnamen und deren Datentyp, festgelegt. Das Gegenstück von Tabellen in MongoDB sind *Collections*. Beziehun-

gen bzw. *Relationen* zwischen Tabellen werden, je nach Art der Beziehung, durch eine von zwei Möglichkeiten repräsentiert: eine Fremdschlüssel-Beziehung oder die Verwendung von Relationstabellen. Es gibt die folgenden möglichen Arten von Relationen:

- eine $1:1$ Beziehung
- eine $1:n$ bzw. $n:1$ Beziehung
- eine $n:n$ Beziehung

Relationstabellen stellen einen wesentlichen Nachteil von klassischen relationalen Datenbanken bezüglich der Verteilung von Daten dar. Auf Grund dessen folgt eine kurze Übersicht, wie Relationstabellen funktionieren und welche Alternativen MySQL, Oracle und PostgreSQL aufweisen. Eine detaillierte Erläuterung der Nachteile von Relationstabellen erfolgt im Abschnitt 5.3.3.

Relationstabellen

In relationalen Datenbanken wird eine $1:1$ Beziehung i.d.R. durch eine Fremdschlüssel-Beziehung dargestellt. Liegt keine $1:1$ Beziehung vor, werden üblicherweise Relationstabellen verwendet. Anstatt den Primärschlüssel der einen Tabelle als Fremdschlüssel innerhalb der anderen Tabelle zu speichern, sind Relationstabellen eigenständige Tabellen. Diese Tabellen verweisen durch ein Attribut (z.B. den Primärschlüssel) von Tabelle *a* auf die korrespondierende Zeile in Tabelle *b*. Für jedes der Attribute aus Tabelle *a* können auch multiple Verweise auf Tabelle *b* existieren. Ein Beispiel:
Angenommen es gibt eine Tabelle *Kunden*, die die Namen der Kunden definiert und eine weitere Tabelle *Adressen*, die die zugehörigen Adressen der Kunden enthält. Jeder Kunde hat mindestens eine Adresse, kann aber beliebige weitere Adressen besitzen. Damit liegt vom Standpunkt des Kunden betrachtet eine $1:n$ und vom Standpunkt der Adressen aus eine $n:1$ Beziehung vor. Die Tabellen *Kunden* und *Adressen* haben die folgenden Strukturen:

id	**name**
1	Max Mustermann

Tabelle 5.1: Kunden-Tabelle

id	**plz**	**ort**	**strasse**
1	12345	Hamburg	Hauptstraße A
2	45678	München	Hauptstraße B

Tabelle 5.2: Adressen-Tabelle

Die Relationstabelle könnte wie folgt aussehen:

kunden-id	addressen-id
1	1
1	2

Die Relationstabelle referenziert die Kunden und die Adressen der jeweiligen Kunden. Wird nun nach den Adressen eines beliebigen Kunden gesucht, können diese in der Relationstabelle anhand der *id* des Kunden ermittelt werden. Genauso kann die Zugehörigkeit einer Adresse zu einem Kunden festgestellt werden.
Ein Beispiel für eine $n : n$ Beziehung wäre etwa die Beziehung von verschiedenen Waren und einer Bestellung auf z.B. Amazon: Jede Ware kann zu einer oder mehr Bestellungen gehören und jede Bestellung enthält eine oder mehr Waren.

PostgreSQL[Grod], Oracle[Cor14] und seit Version 5.7.8 auch MySQL[Corg] unterstützen neben dem traditionellen, relationalen Datenmodell in Form von Tabellen auch die Möglichkeit, JSON zu verwenden. Durch diese Begebenheit können Relationen - genau wie in MongoDB - auch als geschachtelte Dokumente bzw. Liste von geschachtelten Dokumenten erstellt werden.

5.3.2 Indizes

Relationale Datenbanksysteme wie MySQL, Oracle, PostgreSQL und viele weitere kombinieren für Indizes zwei Datenstrukturen: eine doppelt verkettete Liste und einen Suchbaum. Die Datenbankeinträge liegen dagegen für gewöhnlich in einer unsortierten Datenstruktur - genannt *Heap*[Fle] - vor.

Verkettete Liste

Die doppelt verkettete Liste dient dem Zweck, die Indizes in einer leicht und schnell traversierbaren und gleichzeitig flexiblen Datenstruktur zu verwalten. Leicht und schnell traversierbar, um möglichst schnell den korrekten Index zu ermitteln und flexibel, um leicht Änderungen an der Index-Struktur vornehmen zu können. Für jede *INSERT* und *DELETE* Operation muss die Index-Struktur angepasst werden. Je mehr Zeit eine derartige Operation benötigt, desto negativer wirkt sich dieser Umstand auf die Zeit der gesamten Schreiboperation aus. Durch die Eigenschaft einer doppelt verketteten Liste (Siehe Abb. 5.4) können Einträge schnell hinzugefügt aber auch entfernt werden, da sich diese Operationen auf nur insgesamt drei Einträge auswirken: auf den betreffenden Knoten sowie der Anpassung des Vorgängers und des Nachfolgers. Auch ermöglicht diese Form der Verknüpfung, dass der Index sowohl vorwärts als auch rückwärts traversiert werden kann.

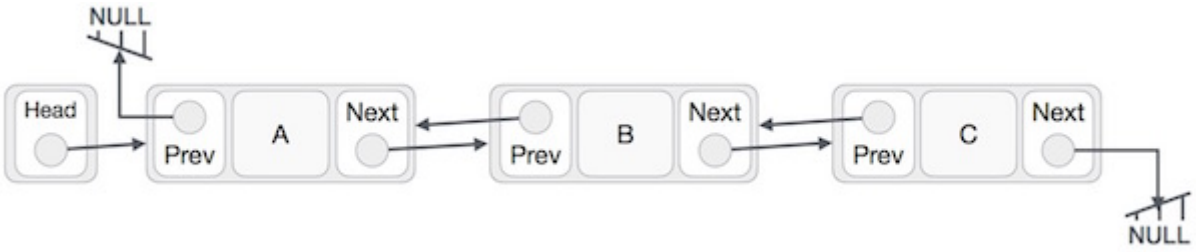


Abbildung 5.4: Eine doppelt verkettete Liste [Poi]

In Abbildung 5.5 ist zu sehen, wie die Blattknoten auf die jeweiligen Tabellen verweisen. Jeder der Blattknoten besteht aus n Index-Einträgen, die jeweils den Wert der indexierten Spalte sowie einen Verweis auf den entsprechenden Tabelleneintrag enthalten[Winb]. Wie zu erkennen ist, sind die Blattknoten sortiert. Werden weitere Index-Einträge hinzugefügt, wird jeder dieser Einträge so eingefügt, dass die Sortierung weiterhin Bestand hat. Durch diesen Umstand wird garantiert, dass die Indizes in einer wohldefinierten Reihenfolge vorliegen. Dadurch ist sichergestellt, dass auf einen bestimmten Index effizient und schnell zugegriffen werden kann, da die Position des Index anhand der Sortierreihenfolge ermittelt wird.

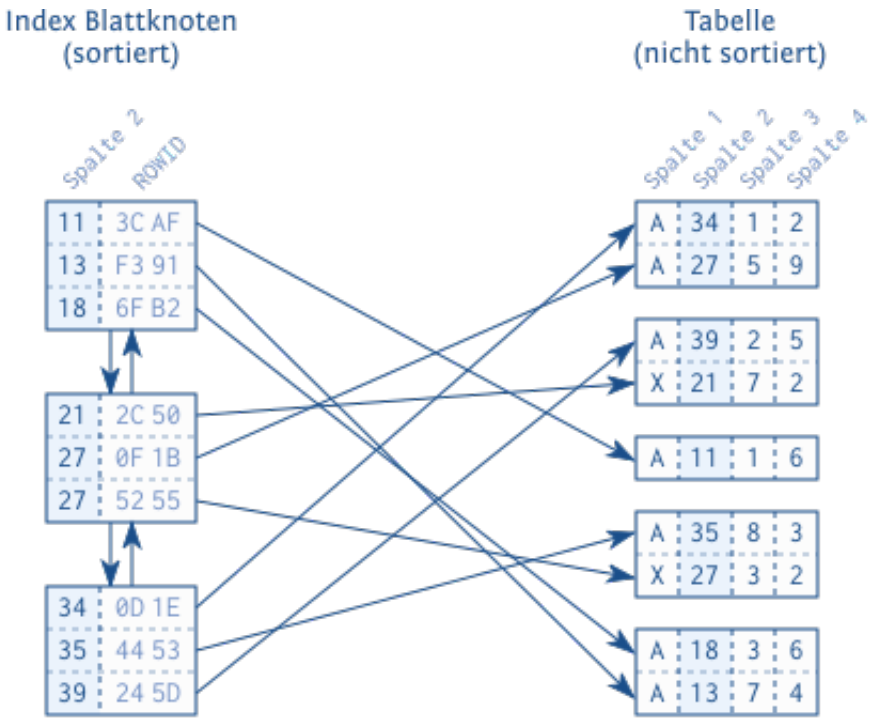


Abbildung 5.5: Die Index-Struktur referenziert die entsprechenden Tabellen [Winb]

Suchbaum

Eine Index-Struktur sollte nach Möglichkeit schnell durchsuchbar sein. In relationalen Datenbanken wird dafür ein **b**alancierter Such**baum** verwendet, kurz auch als *B-Tree*

bzw. B-Baum bezeichnet. Während die verkette Liste die Sortierreihenfolge garantiert [Wina] (siehe Abb. 5.5), dienen die Knoten der höheren Ebenen (siehe Abb. 5.6) der effizienten Suche der korrekten Index-Einträge. Wie in Abbildung 5.6 zu erkennen ist, verweisen die Knoten der höheren Ebene auf die Knoten der niederen Ebene. Der Knoten der höheren Ebene hält eben soviele Einträge, wie er Blattknoten referenziert. Jeder Eintrag ist der jeweils größten Werte der indexierten Spalte der darunter liegenden Blatt-Knoten. Diese Struktur setzt sich rekursiv fort, bis eine Ebene entsteht, die nur noch aus einem einzigen Knoten, dem sogenannten *Root*-Knoten, besteht[Wina]. Die daraus resultierende Struktur kann leicht traversiert werden. Abbildung 5.7 zeigt, wie z.B. der Index 57 gefunden wird[11]: Von dem *Root*-Knoten aus werden jeweils die Einträge der Knoten der niederen Ebene dahingehend verglichen, ob ein Wert der größer oder gleich des gesuchten Index enthalten ist. 57 ist größer als der erste Eintrag 39, aber kleiner als der folgende Eintrag 83. Somit werden die Kind-Knoten für den Eintrag 83 geprüft. Dieses Vorgehen wiederholt sich rekursiv, bis ein Blattknoten und damit der konkrete Index erreicht wurde.

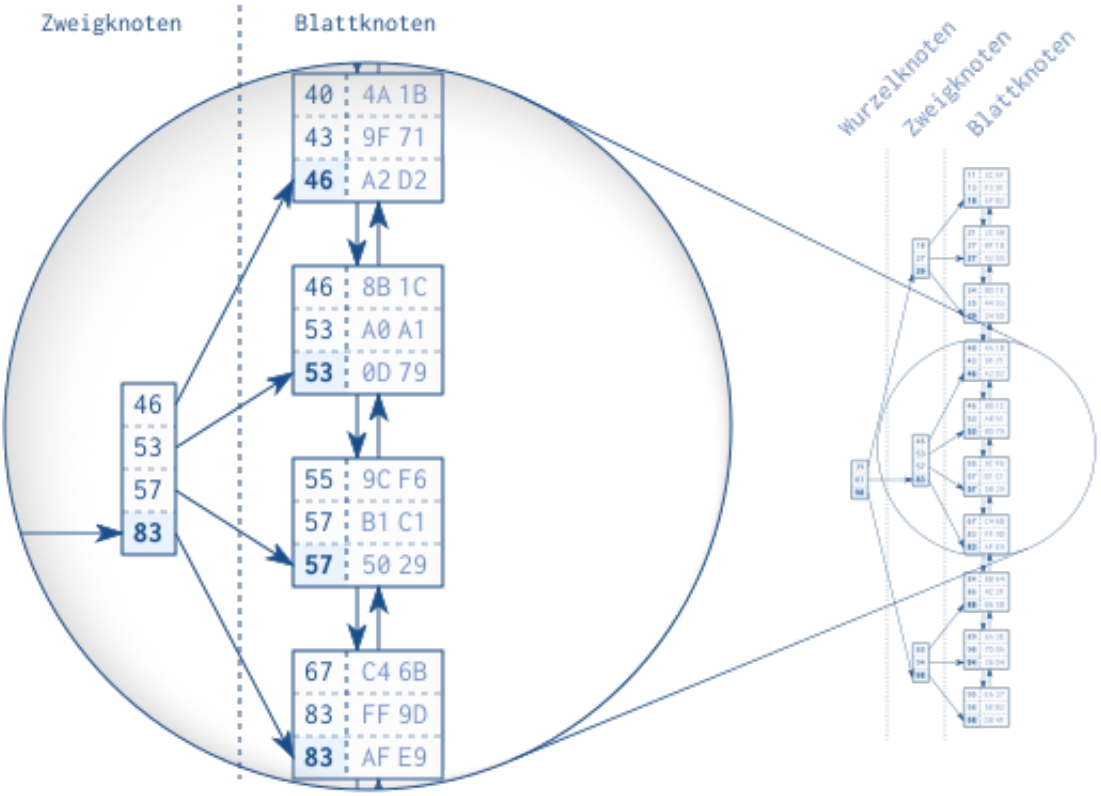


Abbildung 5.6: Die Baum-Struktur des Index [Wina]

[11] Beispiel entnommen aus [Wina]

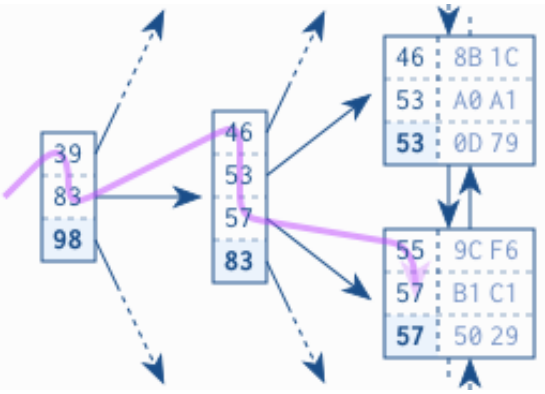


Abbildung 5.7: Suche innerhalb der Baum-Struktur des Index [Wina]

Vorteile

B-Bäume als Index-Strukturen haben den Vorteil, dass die Tiefe des resultierenden Baums nur logarithmisch ansteigt. Zusätzlich zu den Index-Einträgen, die ein Knoten der höheren Ebene speichern kann, wächst die verwaltbare Gesamtanzahl der Index-Einträge mit jeder zusätzlichen Ebene eines B-Baums. Dieses Wachstum ist wiederum exponentiell. Die Anzahl der zu verwaltbaren Index-Einträge ergibt sich aus der Formel $a = b^x$ wobei x die Anzahl der Ebenen des Baumes und b die Anzahl der Einträge der höheren Knoten darstellt[Wina]. Die folgende Tabelle zeigt einen Überblick, wie viele Index-Einträge insgesamt bei einer wachsenden Anzahl der Ebenen x (beginnend bei 3) und bei konstant bleibender Einträge $b = 4$ verwaltet werden können[Wina]:

Anzahl der Ebenen	Verwaltbare Index-Einträge insgesamt
3	$4^3 = 64$
4	$4^4 = 256$
5	$4^5 = 1024$
6	$4^6 = 4096$
7	$4^7 = 16384$
8	$4^8 = 65536$
9	$4^9 = 262144$
10	$4^{10} = 1048576$

5.3.3 Vergleich der Volltextsuche mit MongoDB

Grundsätzlich ist es möglich, Volltextsuchen in relationalen Datenbanken vorzunehmen. Die drei populären relationalen DMBS, Oracle[Cori], PostgreSQL[Grob] und MySQL[Corb] bieten Möglichkeiten, Volltext-Felder anzulegen, diese zu indexieren und anschließend Volltextsuchen auszuführen. Ebenso bieten Oracle[Cork], PostgreSQL[Groe] und MySQL [Corc] die Möglichkeit, Stopwörter (Siehe Abschnitt 3.1) zu filtern. Allerdings sind

die standardmäßigen Stopwörter zumindest in MySQL und Oracle unvollständig: MongoDB verfügt für die englische Sprache über 174 Stopwörter[dtb], Oracle über 114[Corh] und MySQL nur über 36[Cord]. Auf Grund dessen müssen die Stopwörter aus Gründen der Vollständigkeit manuell angegeben oder ergänzt werden. Stemming (Siehe Abschnitt 3.4) wird nur von Oracle[Cora] für Fuzzy-Suchen unterstützt, während MySQL[Corj] dazu bislang keine Möglichkeit bietet. In PostgreSQL gibt es die Möglichkeit, sogenannte *Dictionaries*[Groa] anzulegen - u.a. durch die Verwendung von *Snowball* (Siehe Abschnitt 4.4.1). Mittels dieser *Dictionaries* werden Stopwörter gefiltert und die übrigen Wörter normalisiert (sprich: gestemmt).
Dokumentorientierte und damit schemalose Datenbanken wie MongoDB benötigen keine Relationen in Form von Fremdschlüssel-Beziehungen oder Relationstabellen. In dokumentorientierten Datenbanken werden Relationen im Dokument entweder direkt als geschachteltes Dokument oder in Form einer Liste von geschachtelten Dokumenten gespeichert. Dadurch ist die Datenstruktur wesentlich flexibler und liegt bereits in ihrer endgültigen Fassung vor. In relationalen Datenbanken sind dagegen üblicherweise *Joins* notwendig, um Relationen aufzulösen. Allerdings unterstützen die betrachteten relationalen DBMS-Vertreter, MySQL[Corg], Oracle[Cor14] und PostgreSQL[Grod] mittlerweile ebenfalls JSON.
Sharding ist ein entscheidender Vorteil, um die zu durchsuchende Datenmenge effektiv aufzuteilen und somit u.a. (komplexe) Suchanfragen parallel auszuführen. Relationale Datenbanken lassen sich aus diversen Gründen nur sehr schwer auf mehrere Maschinen aufteilen, u.a. auf Grund der Verwendung von Relationstabellen. Daher tendieren relationale Datenbanken dazu, mit einer wachsenden Datenmenge langsamer zu werden, da die Datenmenge nicht effektiv aufgeteilt und somit auch nicht parallel durchsucht werden kann. Durch die Abwesenheit von Relationstabellen wird das Sharding von relationalen Datenbanken vereinfacht. Ansonsten besteht u.a. die Gefahr, dass Relationen auf anderen Shards gespeichert werden. In diesem Fall müssten *Joins* über Shards hinweg vorgenommen werden, um Daten in der korrekten Form zu ermitteln. Durch die Umstellung von Relationstabellen auf JSON und geschachtelten Dokumenten, um Beziehungen wie in MongoDB zu definieren, könnte dieser Umstand umgangen werden. Allerdings würde dies bedeuten, einen erheblichen Teil des relationalen Datenmodells nicht zu verwenden. Daraus würde sich die Frage ergeben, warum dann überhaupt eine relationale Datenbank verwendet werden sollte. Relationale Datenbanken sollten verwendet werden, sobald strukturierte Daten vorliegen. Der gänzliche Verzicht auf Relationstabellen und die Umstellung auf JSON ist daher ein Widerspruch zum relationalen Datenmodell. Es stimmt zwar, dass MySQL, Oracle und PostgreSQL und noch einige andere relationale DBMS JSON unterstützen. Allerdings ist diese Unterstützung nicht als Ersatz für (Relations-)Tabellen vorgesehen, sondern dient vielmehr dem Umstand, auch unstrukturierte Daten zusammen mit strukturierten Daten speichern zu können.
Relationale Datenbanken verwenden generell einen B-Baum zur Realisierung der Index-

Struktur - ein Umstand den sie mit MongoDB teilen. Allerdings können bei den drei ausgewählten Vertretern relationaler DBMS, MySQL[Corf], PostgreSQL[Groc] und Oracle [Core] - im Gegensatz zu MongoDB - *Inverted Indizes* erstellt werden. Andererseits verfügen relationale Datenbanken über keine Möglichkeit, Suchanfragen zu speichern und darauf folgende indexierte Dokumente auf Übereinstimmungen zu untersuchen. Wegen der erschwerten Skalierbarkeit und den unvollständigen Volltextsuchen ist von der Verwendung relationaler DBMS zur Realisierung einer inkrementellen Volltextsuche abzuraten.

6 Implementation der Volltextsuchanfrage in InvaliDB

Dieses Kapitel wird anschaulich die Auswahl der verwendeten Technologien, die angewendete Architektur sowie deren Komponenten beschreiben. Die Implementation basiert hinsichtlich der Volltextsuche mindestens auf MongoDB 3.2. Mit Version 3.2 wurden die Optionen *$diacriticSensitive* (Siehe Abschnitt 4.7.3) und *$caseSensitive* (Siehe Abschnitt 4.7.2) sowie die 3. Version des Text-Index eingeführt[Monv][Mons]. MongoDB 3.2 war Voraussetzung für die Implementation, um die Benutzung der Volltextsuche durch die angesprochenen Optionen flexibler zu gestalten. Des Weiteren sollte eine zeitgemäße Version des Text-Index verwendet werden (Siehe auch Abschnitt 4.1). Da die 3. Version des Text-Index die aktuellste und der momentane Standard ist, basiert die Implementation auf diesem Verhalten.

6.1 Verwendete Technologien

Zur Realisierung einer MongoDB äquivalenten Volltextsuche mussten zunächst die verwendeten Kern-Technologien der relevanten Bereiche in MongoDB ausfindig gemacht werden. Die relevanten Bereiche waren die lexikalische Analyse zur Zerlegung des vorliegenden Textes, das Filtern der vorkommenden Stopwörter, das Stemming für die Reduktion auf einen einheitlichen Wortstamm und die Berechnung des *Text-Score*. In MongoDB wurden mehrheitlich Eigenimplementierungen der aufgezählten Bereiche vorgenommen, mit zwei Ausnahmen:

- Die Listen der zu filternden Stopwörter werden aus einer Fremdquelle bezogen, um stets über vollständige und aktuelle Listen zu verfügen. Die eigentliche Filterung anhand dieser Liste von Stopwörtern erfolgt allerdings ebenfalls durch eine Eigenimplementierung.
- Die Stemming-Prozedur wird einer für die Sprache *C* generierten Snowball Implementierung überlassen, die durch eine *C++*-Kapselung in MongoDB verwendet wird

Da InvaliDB in Java geschrieben wurde, mussten äquivalente Technologien für oder vorzugsweise in Java gefunden werden. Die Wahl fiel auf Lucene, die sowohl die lexikalische Analyse, das Filtern von Stopwörtern verschiedener Sprachen sowie gleichermaßen das

Stemming verschiedener Sprachen unterstützt. Damit verwendet die im Rahmen dieser Arbeit implementierte Volltextsuche - genau wie Elasticsearch und SolR - Lucene zur Text-Bearbeitung und -Auswertung.

6.1.1 Lucene

Apache Lucene[Fouc] ist eine hoch performante und frei verfügbare Bibliothek mit dem Fokus auf Volltext-Suchen und Text-Auswertungen. Lucene ist vollständig in Java geschrieben und bietet dadurch den Vorteil, Plattform unabhängig eingesetzt werden zu können. Darüber hinaus liegt Lucene als OpenSource vor und kann daher im Rahmen der Apache-Lizenz selbst erweitert und angepasst werden. Lucene liegt zum gegenwärtigen Zeitpunkt in der Version 7.2.1 vor (Stand 23.01.2018). Derzeit wird die im Oktober 2017 erschienene Version 7.1.0 in InvaliDB verwendet. Der immense Funktionsumfang und die performante Ausführung zeigt sich u.a. dadurch, dass mit SolR und Elasticsearch zwei der größten Such-Engines der Welt Lucene bereits seit Anfang an verwenden (Siehe Abschnitt 5.1 und 5.2). Besonders deutlich wird die Leistungsfähigkeit und die Skalierbarkeit von Lucene anhand häufig frequentierter und zwingend auf Skalierbarkeit bauender prominenter Beispiele. Dazu zählen u.a. Wikipedia, die Lucene indirekt durch Elasticsearch verwenden[Fouw], und insbesondere Twitter[Twi], welches täglich $500.000.000$ Beiträge, sogenannte *Tweets*, verarbeiten muss, was ~ 6.000 Beiträgen pro Sekunde entspricht[Asl18].
Lucene wurde ursprünglich von Doug Cutting zwischen 1997 und 1999 entwickelt und wurde Ende 2001 in die Software-Familie *Jakarta*[Foub] der Apache-Foundation eingegliedert [Ltd17b]. Die Software-Familie *Jakarta* repräsentierte diverse populäre Java-Projekte, wurde allerdings 2005 obsolet, da sämtliche Projekte zu vollwertigen Top-Level Projekten aufstiegen[Foub]. Seit Anfang 2005 ist Lucene ein eigenes und unabhängiges Projekt[Ltd17b]. Auf Einzelheiten der verwendeten Lucene-Komponenten wird im Abschnitt 6.2.6 vertieft eingegangen.

6.2 Architektur der Komponenten

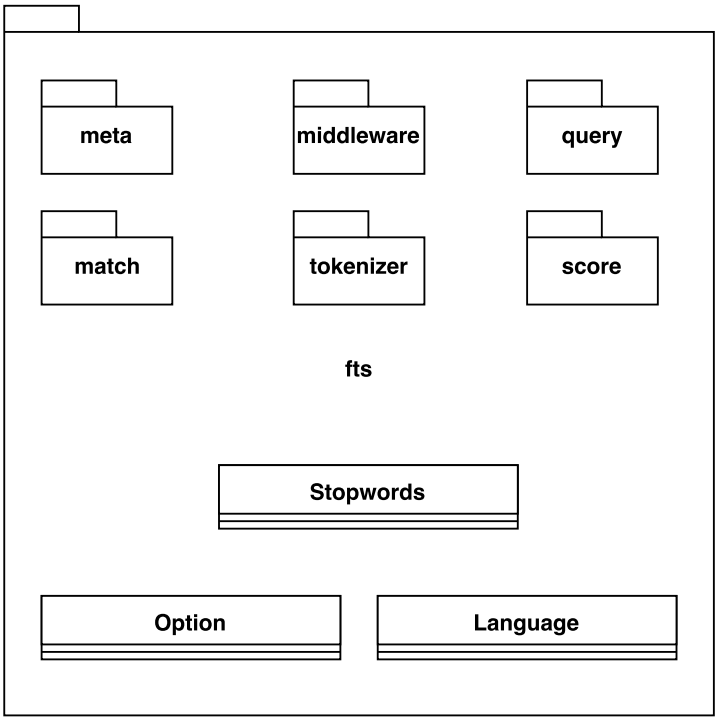


Abbildung 6.1: Übersicht der Komponenten

Die Komponenten, die im Rahmen dieser Arbeit für den Einsatz der Volltextsuche implementiert wurden, bestehen aus vier Klassen auf der obersten Ebene und sechs Java Packages, die ihrerseits weitere Klassen enthalten. In Abbildung 6.1 ist ein Überblick über die Struktur der Architektur zu sehen. Die Klasse *Option* dient der Erfassung und Darstellung der Optionen der Volltextsuchanfrage (Siehe Abschnitt 4.7). *Language* ist ein *enum* und gibt Auskunft darüber, welche Sprachen unterstützt werden. Außerdem kann darüber ein passender Tokenizer erstellt werden. Dem Tokenizer können optional die Optionen (Siehe Abschnitt 4.7), die er für die lexikalische Analyse (Siehe Abschnitt 3.2) verwenden soll, übergeben werden. Die Klasse *Stopwords* verwaltet für jede Sprache die zu verwendenden Stopwörter (Siehe Abschnitt 3.1) und hält diese in einem Cache im Speicher, damit kein erneutes Laden der Stopwörter notwendig ist. Zudem kann *Stopwords* Auskunft darüber geben, ob es sich bei einem übergebenen Wort in der verwendeten Sprache um ein Stopwort handelt, oder nicht.
Die Architektur und die Komponenten der sechs Packages werden, nach einer kurzen Beschreibung der Schnittstelle zu Orestes, im Folgenden als einzelne Abschnitte dargestellt.

6.2.1 Schnittstelle zu Orestes

Um eine Query-Auswertung zu beginnen, müssen die einzelnen verfügbaren Operatoren - im weiteren Verlauf als Prädikate bezeichnet - unterschieden und entsprechend ausgewertet werden. Dazu dient die Klasse *PredicateCache* aus dem Package *orestes-events*. Neben dem im Rahmen dieser Arbeit behandelten Volltextsuch-Prädikat *FullTextSearch* existieren auch Prädikate zur Auswertung von regulären Ausdrücken, Größer/Kleiner Vergleichen, Geo-Daten etc. Jedes Prädikat erbt von der abstrakten Klasse *Predicate* und muss die abstrakte Methode *computeMatchingStatus* implementieren. Diese Methode bekommt das zu überprüfende Dokument im JSON-Format übergeben. In dieser Methode findet die Auswertung des jeweiligen Prädikats statt. Jeder Query, der untersucht werden soll, wird dem Prädikat zusammen mit weiteren Parametern im Konstruktor übergeben. Im Fall des *FullTextSearch*-Prädikats wird der Query anschließend der *parseQuery* Methode übergeben, in der die notwendigen Informationen extrahiert werden. Der genaue Vorgang einer eingehenden Volltextsuchanfrage wird ausführlich im Abschnitt 6.3 beschrieben.

6.2.2 Das Match-Package

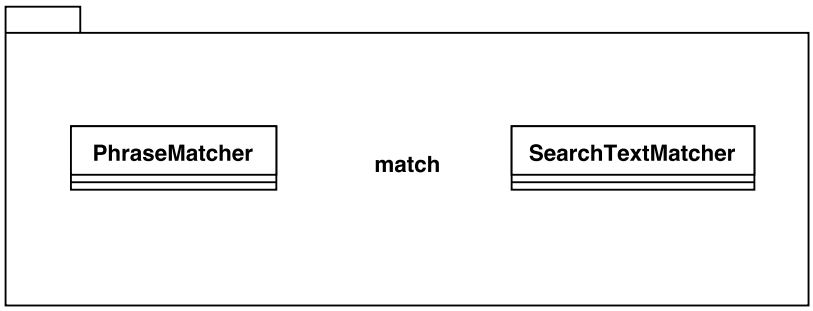


Abbildung 6.2: Übersicht der Komponenten des Match-Package

Die Klassen *SearchTextMatcher* und *PhraseMatcher* werden im Abschnitt 6.3 ausführlicher beschrieben.

6.2.3 Das Middleware-Package

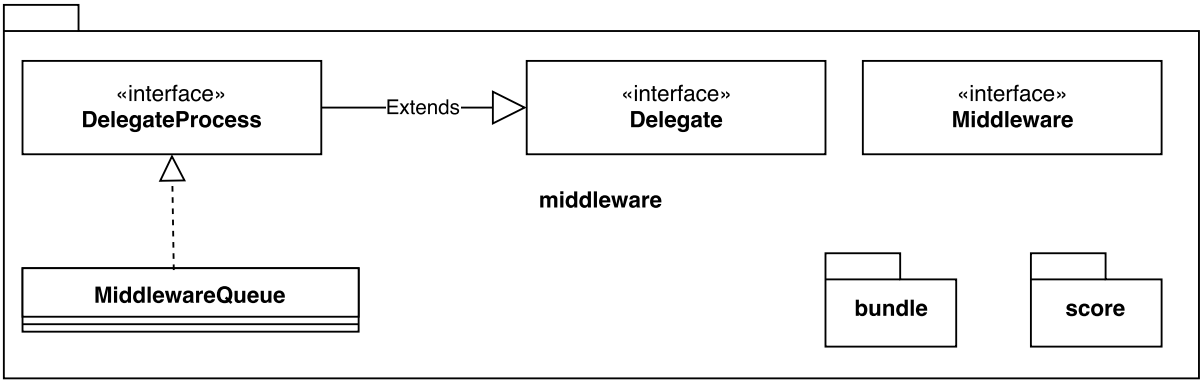


Abbildung 6.3: Übersicht der Komponenten des Middleware-Package

Middlewares (Siehe Abb. 6.4) sind Schichten, durch die Daten geschleust werden. Jede Schicht kann die eingereichten Daten um Informationen ergänzen oder filtern. Die Daten werden in Daten-Bündeln - sogenannten *Bundles* - zusammengefasst und durch die Schichten gereicht. Die *MiddlewareQueue* verwaltet die zu traversierenden *Middlewares* und ist für die Weitergabe und finale Rückgabe des verwendeten *Bundles* zuständig. Die konkrete Implementation der *MiddlewareQueue* ist allerdings irrelevant für die weitere Betrachtung.

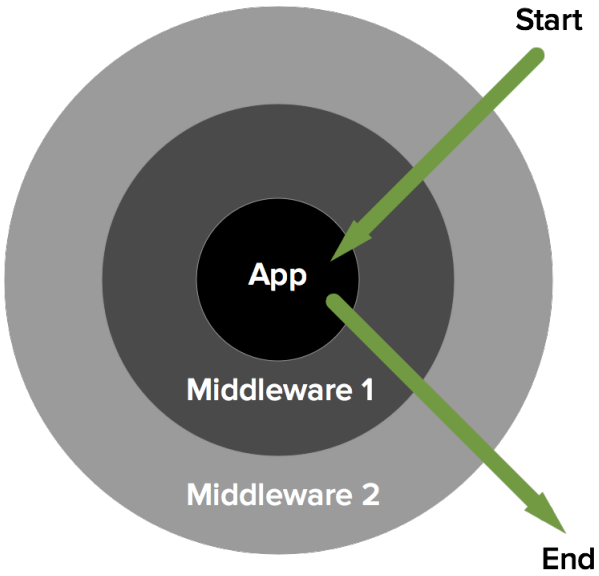


Abbildung 6.4: Middlewares [JL]

Bundles

Das Middleware-Package besitzt wiederum zwei eigene Packages:

- *bundle*:

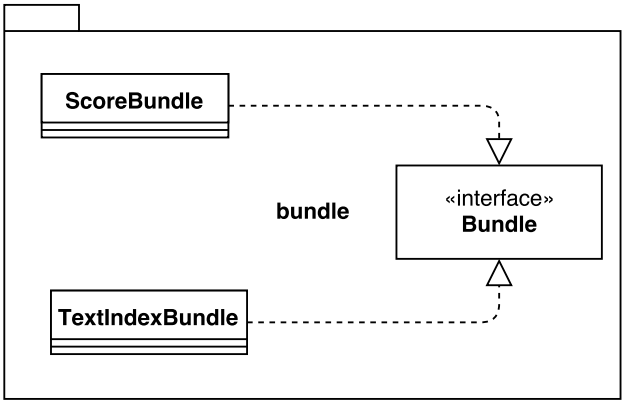


Abbildung 6.5: Übersicht der Komponenten des Bundle-Package

Das *bundle*-Package umfasst die Klassen *ScoreBundle* und *TextIndexBundle*. Die Klasse *TextIndexBundle* findet weitergehende Erwähnung im Abschnitt 6.2.4 „Das Meta-Package“ und wird zusammen mit der Klasse *ScoreBundle* im Abschnitt 6.3 ausführlich beschrieben.

- *score*:

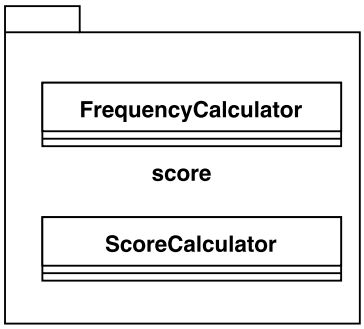


Abbildung 6.6: Übersicht der Komponenten des Score-Package

Das *score*-Package umfasst die Klassen *FrequencyCalculator* und *ScoreCalculator* die zur Berechnung des Text-Score beitragen und ausführlich im Abschnitt 6.3 beschrieben werden.

6.2.4 Das Meta-Package

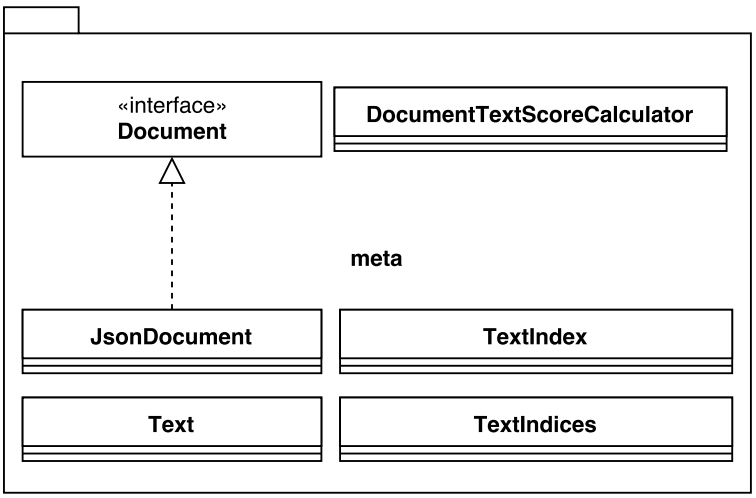


Abbildung 6.7: Übersicht der Komponenten des Meta-Package

Die Klasse *JsonDocument* ummantelt das momentan betrachtete Dokument, welches als Instanz der Klasse *JsonValue* vorliegt. Diese Fassade erleichtert das extrahieren von Informationen aus dem Dokument. Die Klasse *TextIndices* gibt die vorliegenden Text-Indizes in Form einer Liste von *TextIndexBundle*-Instanzen zurück. Die Instanz der Klasse *TextIndices* ist eine zentrale Schnittstelle, die für die Verwaltung und den Abruf der jeweiligen Text-Indizes verantwortlich ist. Ein Text-Index enthält Informationen darüber, für welches Feld der Index angelegt und welche Gewichtung (Siehe Abschnitt 4.5) angegeben wurde. Zusätzlich ist spezifiziert, welche Standard-Sprache (Siehe Abschnitt 4.2) verwendet werden soll. Ein *TextIndexBundle* bündelt diese und weitere Informationen. Zum Beispiel wird die endgültig zu verwendende Sprache gespeichert, bei der es sich entweder um die Sprache des Dokuments oder die im Text-Index angegebene Standard-Sprache handelt. Darüber hinaus wird der Inhalt des Feldes des Text-Index (sofern vorhanden) aus dem Dokument zusammen mit den Optionen (Siehe Abschnitt 4.7) des gespeicherten Volltext-Queries extrahiert. Die Gewichtung wird aus dem Text-Index direkt übernommen. Aus dem unverarbeiteten Inhalt des Dokument-Feldes werden die enthaltenen Worte - abzüglich der Stopwörter (Siehe Abschnitt 4.3) - durch die lexikalische Analyse (siehe Abschnitt 3.2) gewonnen und als Liste gespeichert. Sind keine Text-Indizes angegeben, werden alle Felder des vorliegenden Dokuments verwendet, sofern sie textuell verwertbaren Inhalt aufweisen. Dazu wird geprüft, ob es sich bei dem Feld um eine *JsonPrimitive*-Instanz handelt. Ist dies nicht zutreffend, wird das Feld ignoriert. Andernfalls wird geprüft, ob es sich um einen Text handelt. Nur wenn diese Annahme ebenfalls zutreffend ist, wird der Inhalt des Feldes ausgelesen. Die *DocumentTextScoreCalculator*-Instanz ist für die Berechnung des Text-Score des vorliegenden Dokuments zuständig und bekommt dazu u.a. die Liste der *TextIndexBundle*-Instanzen übergeben. Weitere De-

tails werden im Abschnitt 6.3 beschrieben. Die Klasse *Text* ist lediglich dazu da, um die Parameter der Klasse *TextIndexBundle* zu gruppieren. Die Parameter sind die zu verwendende Sprache, der Inhalt des Feldes und die zugehörige Instanz der Klasse *TextIndex*. Die Klasse *TextIndex* enthält die Standardsprache, den Namen des Dokument-Feldes sowie die Gewichtung. Konnte keine Sprache für das Dokument identifiziert werden, wird die Sprache des Text-Index verwendet (Siehe Abschnitt 4.2). Ist keine Gewichtung angegeben, wird die Standard-Gewichtung verwendet. Außerdem wird darauf geachtet, dass die Gewichtung den minimalen und maximalen Wert nicht überschreitet. Die Werte der minimalen und maximalen Gewichtung in MongoDB können im Abschnitt 4.5 eingesehen werden.

6.2.5 Das Score-Package

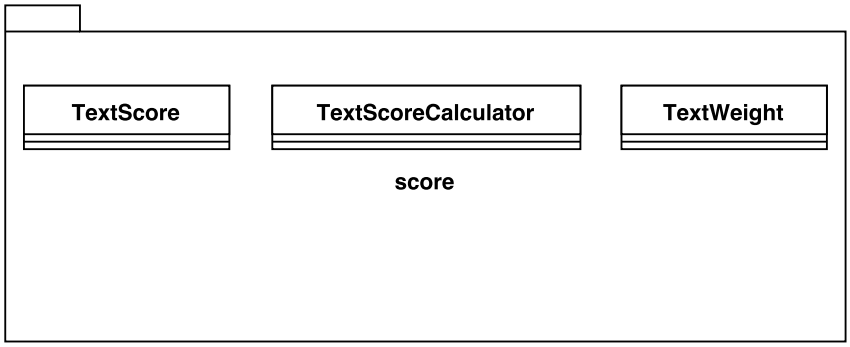


Abbildung 6.8: Übersicht der Komponenten des Score-Package

Die Klassen *TextScore* und *TextScoreCalculator* werden ausführlich im Abschnitt 6.3 beschrieben. Die Klasse *TextWeight* ist ein *enum*, welches den minimalen, maximalen und den Standard-Wert der Text-Index Gewichtung (Siehe Abschnitt 4.5) definiert.

6.2.6 Das Tokenizer-Package

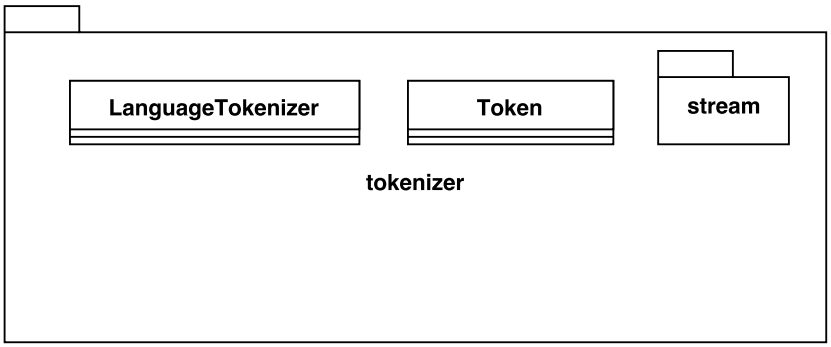


Abbildung 6.9: Übersicht der Komponenten des Tokenizer-Package

Zur Durchführung der lexikalischen Analyse (Siehe Abschnitt 3.2) wird zunächst durch die *Language*-Instanz der verwendeten Sprache sowie den angegebenen Optionen der Volltextsuchanfrage ein *LanguageTokenizer* erstellt. Anschließend werden die positiven und negativen Terme durch ein Leerzeichen getrennt zu einem Text zusammengefügt. Dieser temporäre Text wird der Methode *tokenize* des *LanguageTokenizer* übergeben, die den Text in einzelne Token (repräsentiert durch Instanzen der gleichnamigen Klasse) zerlegt. Die Token entsprechen nicht zwangsläufig den vorherigen einzelnen Termen, da, wie in MongoDB, u.a. Satzzeichen entfernt werden. Es folgt die Filterung der Stopwörter (Siehe Abschnitt 3.1, 4.3 und 4.3.1) und das Stemming (Siehe Abschnitt 3.4 und 4.4). Alle drei Prozesse werden intern durch die Lucene-*Analyzer*[Foua] vorgenommen. Allerdings benötigen die *Analyzer* Informationen darüber, um welche Sprache es sich bei dem vorliegenden Text handelt und welcher *Tokenizer*[Fouv] und welche Filter[Fouu] verwendet werden sollen. Die dazu notwendigen Komponenten befinden sich im *stream*-Package.

Token-Streams

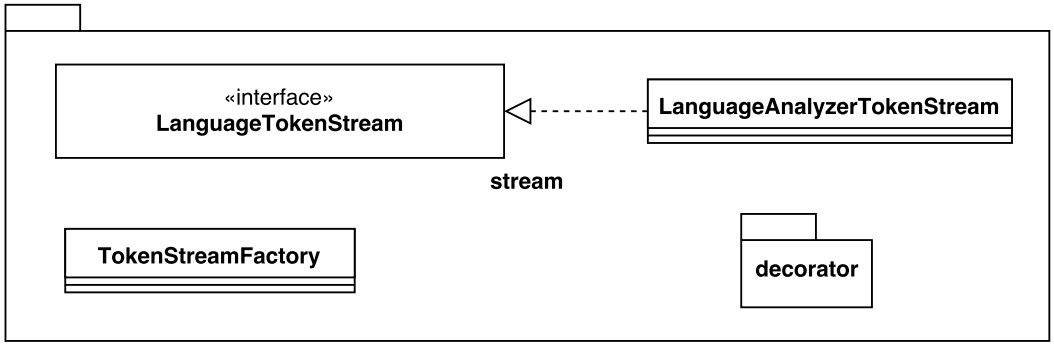


Abbildung 6.10: Übersicht der Komponenten des Stream-Package

Ein *LanguageTokenizer* wird i.d.R. durch eine *Language*-Instanz erstellt. Die *Language*-Instanz erzeugt dazu zunächst mittels einer Instanz der Klasse *TokenStreamFactory* eine *LanguageTokenStream*-Instanz[Fouk], die wiederum die *Language*-Instanz und die zu verwendenden Optionen übergeben bekommt. Die *LanguageTokenStream*-Instanz wird ihrerseits dem *LanguageTokenizer* übergeben und ermittelt intern, welche Komponenten zu verwenden sind. Bei den zu ermittelnden Komponenten handelt es sich um den *Tokenizer* und die entsprechenden Filter zur Bearbeitung des Textes. Als *Tokenizer* wird stets der *StandardTokenizer*[Fous] von Lucene verwendet. Für jede Sprache existieren spezifische Dekoratoren, die angeben, welche weiteren Filter, aufbauend auf dem *StandardFilter*[Four], benutzt werden sollen. Für die Bestimmung der Filter wird dem Dekorator die Optionen und die verwendete Sprache übergeben. Diese Dekoratoren befinden sich im Package *decorator*, deren genaue Implementation allerdings für die weitere Betrachtung nicht relevant ist.
Für jede Sprache müssen i.d.R. unterschiedliche Filter verwendet werden, da jede Sprache andere Zeit- und Steigerungsformen, Ausnahmen wie unregelmäßige Verben, ggf. di-

verse diakritische Zeichen und morphologische Abstammungen besitzt. Allerdings wird immer zumindest der *LowerCaseFilter*[Fouj] und i.d.R. auch ein Normalisierungs-Filter[Foul] (z.B. der *GermanNormalizationFilter*[Foug]) angewendet. Die Normalisierung bereinigt z.B. diakritische Zeichen (Siehe Abschnitt 4.7.3). Der *GermanNormalizationFilter* z.B. ermöglicht eine *Snowball* (Siehe Abschnitt 4.4.1) konforme Text-Basis, indem sämtliche Umlaute in ihre zugrundeliegenden Buchstaben überführt werden:

- ä → a
- ö → o
- ü → u

Gleichermaßen werden Vokal-Aliase, wie *ue*, *oe* und *ae* (Siehe Abschnitt 4.4.1), ebenfalls auf die zugrundeliegenden Buchstaben der korrespondierenden Vokale abgebildet:

- ae → ä → a
- oe → ö → o
- ue → ü → u

Für den regulären Fall, dass Stopwörter gefiltert werden sollen, wird zusätzlich nach dem *LowerCaseFilter* - aber vor dem Normalisierungs-Filter - eine Instanz des *StopFilter*[Fout] eingebunden. Die Reihenfolge ist wichtig, da die Stopwörter in der Kleinschreibweise vorliegen müssen (Siehe Abschnitt 4.3.1), allerdings nach einer Normalisierung nicht mehr als Stopwort erkannt und als solches gefiltert werden könnten.

6.2.7 Das Query-Package

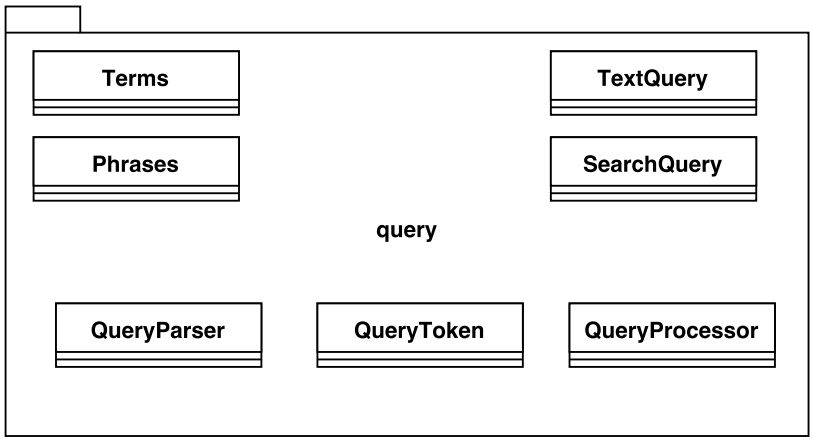


Abbildung 6.11: Übersicht der Komponenten des Query-Package

Sämtliche Komponenten des Query-Packages werden ausführlich im Abschnitt 6.3 beschrieben.

6.3 Ablauf einer Volltextsuchanfrage

Die Auswertung der Volltextsuchanfrage beginnt im *FullTextSearch*-Prädikat von Orestes. Zunächst werden in der Methode *parseQuery* die für die Volltextsuche relevanten Informationen aus dem übergebenen Volltext-Query extrahiert. Zu diesen Informationen zählen:

- die Suchanfrage
- die optional angegebene Sprache (Siehe Abschnitt 4.2)
- die optionale Angabe, ob die Suche *Case-Sensitive* (Siehe Abschnitt 4.7.2) sein soll
- die optionale Angabe, ob die Suche *Diacritic-Sensitive* (Siehe Abschnitt 4.7.3) sein soll

Nach der Extraktion und Zuweisung von möglichen Standardwerten werden die gesammelten Informationen an eine Instanz der Klasse *TextQuery* übergeben. Anschließend wird diese Instanz im Prädikat gespeichert, damit die extrahierten Informationen des Queries in den nachfolgenden Methoden abgerufen werden können. Die Informationen werden z.B. in der Methode *computeScore* benötigt, um für eingehende Dokumente den *Text-Score* (siehe Abschnitt 4.6) zu berechnen. Dazu wird zunächst das momentan betrachtete Dokument, wie im Abschnitt 6.2.4 „Das Meta-Package“ beschrieben, mit einer Instanz der Klasse *JsonDocument* ummantelt. Die Fassade wird, zusammen mit dem Namen der gegenwärtigen MongoDB-*Collection* und den im *TextQuery* gespeicherten Optionen (Siehe Abschnitt 4.7), an die Instanz der Klasse *TextIndices* übergeben. Wie im Abschnitt 6.2.4 beschrieben, werden die vorliegenden Text-Indizes in Form einer Liste von *TextIndexBundle*-Instanzen zurückgegeben. Diese nun vorliegende Liste wird zusammen mit dem *TextQuery* an die Methode *calculate* der *DocumentTextScoreCalculator*-Instanz übergeben. Dort werden alle ermittelten *TextIndexBundle*-Instanzen durchlaufen. Für jedes vorliegende *TextIndexBundle* wird der Text-Score durch eine neue Instanz der Klasse *TextScoreCalculator* berechnet.

6.3.1 Verarbeitung der Suchanfrage

Der *TextScoreCalculator* erstellt zunächst mittels des übergebenen *TextQuery* eine Instanz der Klasse *SearchQuery*. Die Klasse *SearchQuery* ist für die Verarbeitung der Suchanfrage zuständig. Dazu wird neben der eigentlichen Suchanfrage auch der zu verwendende *LanguageTokenizer* übergeben. Sofern keine Sprache für die Suchanfrage angegeben wurde, würde normalerweise die Standardsprache des Text-Index verwendet werden (Siehe Abschnitt 4.2). Der Zugriff auf die Text-Indizes ist allerdings während der Verarbeitung der Volltextsuchanfrage bislang nicht möglich. Die Implementierung des Zugriffs würde weitreichende Eingriffe in die bestehende Implementation von InvaliDB erfordern. Da dies allerdings den Umfang dieser Arbeit übersteigen würde, wurde entschieden, den

Zugriff separat und zu einem späteren Zeitpunkt zu implementieren. Auf Grund dessen wird stets Englisch als Standardsprache verwendet.
Die Klasse *SearchQuery* erzeugt intern eine temporäre Instanz der Klasse *QueryProcessor*, an die die Suchanfrage und der *LanguageTokenizer* weitergegeben werden. Durch einen Aufruf der Methode *process* der *QueryProcessor*-Instanz wird die Suchanfrage in Terme und Phrasen zerlegt, dargestellt durch die Klassen *Terms* und *Phrases*. Die Klasse *Terms* speichert die gefundenen positiven und negativen Terme und die Klasse *Phrases* die ermittelten negativen und positiven Phrasen. Innerhalb des *QueryProcessor* wird wiederum eine Instanz der Klasse *QueryParser* erstellt, der die Suchanfrage in einzelne *QueryToken* aufteilt. Solange weitere *QueryToken* in der Suchanfrage enthalten sind, werden diese wie folgt ausgewertet:

- Entspricht der momentan betrachtete *QueryToken* einem Wort, wird geprüft, ob sich der Verarbeitungsprozess in einer Negation und in einer Phrase befindet. Ist dies der Fall, wird der derzeitige Durchlauf beendet und mit dem nächsten begonnen. Liegt nur eine Negation vor und befindet sich vor dem betrachteten Wort ein Leerzeichen, wird die Negation beendet. Anschließend wird die textuelle Repräsentation des *QueryToken* extrahiert. Je nachdem, ob eine Negation vorliegt, wird das Wort entweder als negativer oder als positiver Term mit einem nachfolgenden Leerzeichen an einen *string* angehängt.

- Handelt es sich bei dem derzeitigen *QueryToken* um einen *Hyphen* - also einem Bindestrich/Minuszeichen - wird mit einer Negation begonnen, sofern die beiden folgenden Kriterien erfüllt sind[Monl][Monm]:
  - Der Verarbeitungsprozess befindet sich nicht in einer Phrase
  - Es lag zuvor ein Leerzeichen vor oder es handelt sich um das erste Zeichen der Suchanfrage

- Ist der betrachtete *QueryToken* ein *quote* - also ein Anführungszeichen - wird geprüft, ob damit eine Phrase eingeleitet oder eine bestehende beendet wird. Befindet sich der Verarbeitungsprozess nicht in einer Phrase, wird damit begonnen. Der derzeitige Offset des *QueryToken* wird gespeichert, um später den Ausschnitt der Suchanfrage, der die Phrase repräsentiert, korrekt ermitteln zu können. Liegt derzeit eine Negation vor und trat vor diesem Token ein Leerzeichen auf, wird die Negation beendet.
  Sofern die Phrase endet, wird der betreffende Ausschnitt der Suchanfrage extrahiert. Je nachdem, ob sich der Verarbeitungsprozess momentan in einer Negation befindet, wird eine negative oder eine positive Phrase gespeichert.

Abschließend werden die durch Leerzeichen getrennten Terme mittels des übergebenen *LanguageTokenizer* einer lexikalischen Analyse sowie einer Stemming-Prozedur unterzogen. Die nun getrennt vorliegenden und normalisierten Terme werden anschließend in einer Instanz der Klasse *Terms* gespeichert.

6.3.2 Berechnung des Text-Score

Der *TextScoreCalculator* verifiziert zunächst mittels der *SearchTextMatcher*-Instanz, ob der *SearchQuery* eine Übereinstimmung mit der *TextIndexBundle*-Instanz aufweist. Dazu werden die folgenden drei Überprüfungen vorgenommen:

1. Die Schnittmenge der positiven Terme des *SearchQuery* und der betrachteten *TextIndexBundle*-Instanz darf nicht leer sein. Ist die Schnittmenge leer, würde das bedeuten, dass die *TextIndexBundle*-Instanz keine relevanten Suchbegriffe aufweist.

2. Die Schnittmenge der negativen Terme muss leer sein. Ist die Schnittmenge nicht leer, enthält die *TextIndexBundle*-Instanz Begriffe, nach denen explizit nicht gesucht werden soll.

3. Der vorliegende, unverarbeitete Text der *TextIndexBundle*-Instanz muss alle gesuchten positiven und keine der angegebenen negativen Phrasen enthalten. Andernfalls gelten die Anmerkungen aus Punkt 1. und Punkt 2.

Sofern die drei Überprüfungen erfolgreich sind, wird die *TextIndexBundle*-Instanz dem *ScoreBundle* als Argument übergeben. Andernfalls wird der Wert 0 zurückgegeben. Das *ScoreBundle* bündelt das *TextIndexBundle* und die für die Berechnung des Text-Score notwendigen Daten. Das *ScoreBundle* wird anschließend durch zwei *Middlewares* aufbereitet:

1. *ScoreCalculator*: Der *ScoreCalculator* übernimmt die Bewertung der einzelnen Worte der momentan betrachteten *TextIndexBundle*-Instanz, wie in Abschnitt 4.6 in Punkt 2. beschrieben.

2. *FrequencyCalculator*: Der *FrequencyCalculator* übernimmt die Berechnung der Häufigkeiten der einzelnen Worte der momentan betrachteten *TextIndexBundle*-Instanz unter Berücksichtigung der zuvor berechneten Bewertung durch den *ScoreCalculator*. Die Berechnung der Häufigkeit bzw. *Frequency* wurde im Abschnitt 4.6 in Punkt 3. detailliert beschrieben.

In der Methode *calculate* des *DocumentTextScoreCalculator* wird der berechnete Text-Score jeder *TextIndexBundle*-Instanz auf den Dokument-Text-Score aufaddiert, so wie im Abschnitt 4.6 in Punkt 4. beschrieben. Ein *match* eines Dokuments mit einer Volltextsuchanfrage liegt dann vor, sofern die Bedingung > 0 für den errechneten Dokument-Text-Score zutrifft.

7 Evaluation der Volltextsuchanfrage

In diesem Kapitel wird eine Applikation für Volltextsuchanfragen erstellt, um die Funktionalität der in dieser Arbeit implementierten Volltextsuche zu überprüfen. Abschließend folgt eine Betrachtung der Skalierbarkeit und Performance, soweit es der Umfang dieser Arbeit ermöglicht.

7.1 Baqend API

Die API von Baqend verfügt über zwei verschiedene Möglichkeiten, Suchanfragen auszuführen: Statische Anfragen und Realtime-Anfragen[Baq17]. Statische-Anfragen sind u.a. aus relationalen Datenbanken bekannt. Sie werden ausgeführt und die Ergebnisse werden einmalig zurückgegeben. Um aktuelle Daten zu erhalten, muss der Query wiederholt ausgeführt werden. Realtime-Queries hingegen werden in InvaliDB gespeichert. Eingehende Dokumente werden anschließend auf Übereinstimmungen mit den registrierten Suchanfragen überprüft. Existiert eine Übereinstimmung, wird der Klient darüber informiert[WWR17]. Ob eine Anfrage statisch oder in Echtzeit ausgeführt werden soll, erfolgt durch den Aufruf einer von drei möglichen Methoden. Zunächst muss der Query allerdings spezifiziert werden. Dies ist in Abbildung 7.1 zu sehen, während die Ausführung in Abbildung 7.2 betrachtet werden kann.

```
// 1.) formulate query
var query = DB.Tweet.find()
              .matches('text', /my filter/)
              .descending('createdAt')
              .offset(20)
              .limit(10);
```

Abbildung 7.1: Erstellung eines Query in einer Baqend-Applikation [Wind]

```
// 2.) execute query:
query.resultList(result => ...); // static
query.resultStream(result => ...); // real-time
```

Abbildung 7.2: Ausführung eines Query in einer Baqend-Applikation [Wind]

Durch die Ausführung des Queries mit *resultList* wird ein einmaliges, statisches Ergebnis zurückgegeben[Wind]. Die Ausführung mittels *resultStream* „streamt“ hingegen stets aktuelle Ergebnisse[Wind]. Dadurch sind die passenden Daten der Anfrage immer in Echtzeit vorhanden. Die dritte Möglichkeit, die ebenfalls in einer Echtzeit-Anfrage resultiert, ist die Ausführung mit *eventStream*. Durch diese Ausführung wird der Klient beständig durch Events über Änderungen der Ergebnisse informiert[Baq17] (Siehe auch Abbildung 2.3).

7.2 Problemstellung

Die Applikation, die zur Gewährleistung der Funktionalität der Implementation erstellt wurde, soll als Lösung für ein realistisches Szenario dienen. Allerdings müssen auf spezifische Details verzichtet werden, um den Umfang dieser Arbeit nicht zu strapazieren. Der Begriff *Proof of Concept* ist daher so zu deuten, dass die Applikation zwar grundlegend funktioniert, allerdings keine Gewährleistung für einen praktischen Einsatz gegeben ist. Dazu müssten durch Feldversuche die Grenzen der bestehenden Applikation ausgetestet und verbessert werden. Zudem gibt es hinsichtlich der Benutzbarkeit sicherlich ebenfalls noch Verbesserungspotential. Dieser Umstand ist damit zu begründen, dass die Applikation mit dem Schwerpunkt der Evaluation der Volltextsuche entwickelt wurde. Die Applikation wird zusammen mit dieser Arbeit vollständig übergeben. Als Szenario wurde die folgende Problemstellung gewählt:
Autofahrer sollen in der Lage sein, in Echtzeit über spezifische Verkehrsmeldungen informiert zu werden. Um die Menge der Verkehrsmeldungen einzugrenzen, können diese nach spezifischen Worten oder Phrasen durchsucht werden. Zur weiteren Reduktion der vorliegenden Ergebnisse können optional sowohl die zu befahrende Autobahn sowie das Bundesland angegeben werden. Ist die Autobahn nicht spezifiziert, werden Verkehrsmeldungen für alle Autobahnen angezeigt. Selbiges gilt für die Angabe des Bundeslandes. Die vorliegenden Verkehrsmeldungen sollen alle 30 Sekunden aktualisiert und der Benutzer in Echtzeit über neue passende Meldungen informiert werden. Um keine Verkehrsmeldungen mehrfach vorliegen zu haben, soll anhand eines spezifischen und eindeutigen Attributs entschieden werden, ob diese Verkehrsmeldung bereits existiert. Ist dies der Fall und unterscheidet sich die Meldung (z.B. weil die Informationen ausgetauscht oder erweitert wurden), soll die Verkehrsmeldung aktualisiert werden. Um keine veralteten Verkehrsmeldungen dauerhaft zu speichern, werden diese regelmäßig gelöscht. Sobald eine Verkehrsmeldung gelöscht wurde, wird sie in Echtzeit von der Übersicht der relevanten Verkehrsmeldungen entfernt.
Da die Volltextsuche erst in dieser Arbeit implementiert wurde, mussten einige Hindernisse umgangen werden. So existiert gegenwärtig keine Möglichkeit, einen Text-Index vom Baqend-Dashboard zu erzeugen. Obwohl, wie in Abschnitt 6.2.4 „Das Meta-Package“ beschrieben, alle textuell verwertbaren Felder eines Dokuments durchsucht werden, wenn

keine Text-Indizes spezifiziert sind, lässt Orestes in diesem Fall keine Volltextsuchanfrage zu. Auf Grund dessen muss mindestens ein Text-Index manuell auf der Konsole in MongoDB (Siehe Listing 2 in Abschnitt 4.1) oder durch die Ausführung des folgenden Codes in Orestes erstellt werden:

```
1 client.createIndex(new Bucket("Tweet", BucketType.db),
2             new OrestesIndex(new OrestesIndexKey("...",
  ↪ IndexType.TEXT)), root);
```

Zudem existiert, wie in Abschnitt 6.3 erwähnt, zum gegenwärtigen Zeitpunkt keine Möglichkeit, auf die Text-Indizes während der Verarbeitung der Suchanfrage zuzugreifen.

7.3 Umsetzung

Für das vorliegende Szenario wurde eine Web-Applikation erstellt, die die Verkehrsmeldungen der Seite `https://www.verkehrsinformation.de/` verwendet. Diese werden alle 30 Sekunden ausgelesen und in eine lokale Datenbank gespeichert. Der Benutzer kann durch ein Eingabefeld die für ihn relevanten Suchbegriffe formulieren und optional ebenfalls ein Bundesland und die Autobahn spezifizieren. Auf der genannten Seite können die Verkehrsmeldungen ebenfalls nach Autobahn und Bundesland gefiltert werden. Die grundlegende Verbesserung dieser Applikation ist die Volltextsuche nach relevanten Informationen innerhalb der vorliegenden Verkehrsmeldungen in Echtzeit. Derartige Informationen könnten z.B. die Frage beantworten, ob etwas blockiert oder gesperrt ist, ein Unfall oder ein Stau vorliegt oder ein defektes Fahrzeug auf der Strecke existiert, das einen möglichen Stau zur Folge haben könnte. Je nachdem, ob eine Meldung geändert wurde oder nicht, werden die Meldungen farblich dargestellt: neue bzw. unveränderte Meldungen in Grün, aktualisierte Meldungen in Gelb. Zu Demonstrationszwecken werden nur die Verkehrsmeldungen der Autobahnen A1 - A24 erfasst.

7.3.1 Die Suche

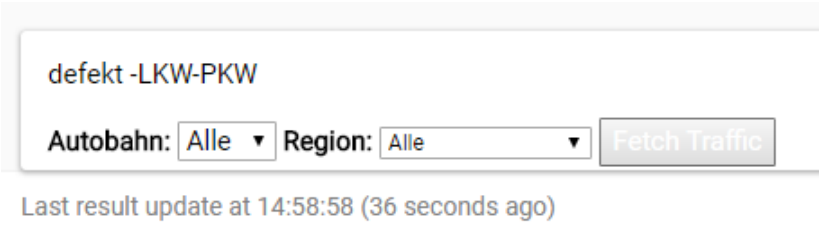


Abbildung 7.3: Die Sucheingabemaske

In der Abbildung 7.3 ist das Eingabefeld mit der Suchanfrage und die beiden optionalen Auswahllisten für die Autobahn und das Bundesland zu sehen. Zudem ist der

Test-Button mit der Aufschrift „Fetch Traffic!“ zu erkennen. Wird dieser betätigt, werden die Verkehrsmeldungen in regelmäßigen Abständen erfasst. Eine erneute Betätigung des Buttons beendet die Erfassung. Unter den Eingabedaten ist die Uhrzeit der letzten Aktualisierung der Suchergebnisse zu erkennen. Die gesamte Oberfläche wurde aus dem Projekt *Twoogle*[Wind] übernommen und nur leicht den Umständen entsprechend angepasst. Die Suchanfrage

```
defekt -LKW -PKW
```

sucht nach allen Verkehrsmeldungen, die das Wort „defekt“, nicht aber die Worte „LKW“ und „PKW“ enthalten. Siehe auch Abschnitt 4.8.

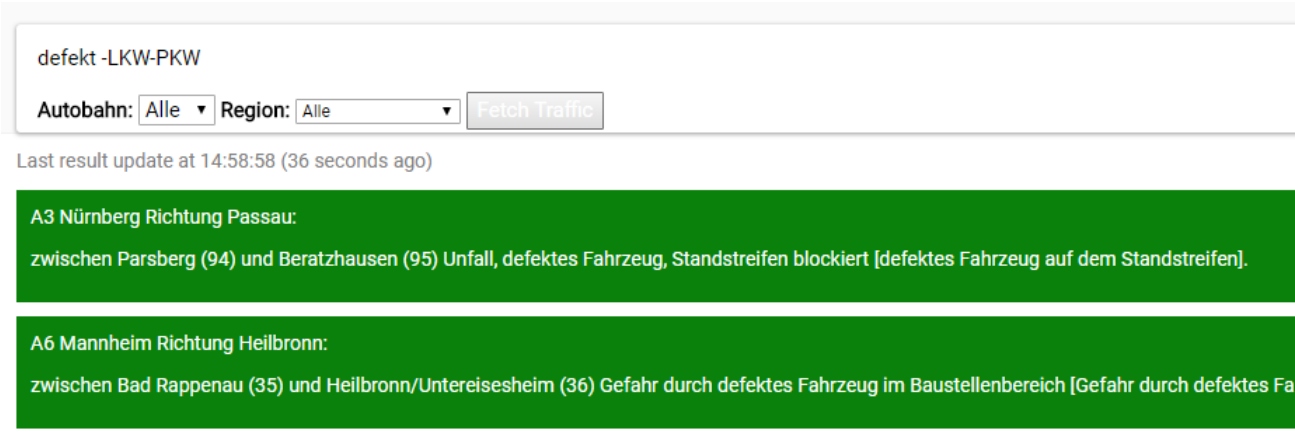


Abbildung 7.4: Übersicht über verschiedene Verkehrsmeldungen

In Abbildung 7.4 sind einige der gefundenen Ergebnisse der Suchanfrage dargestellt. Da diese neu und unverändert vorliegen, sind sie Grün markiert. Der Javascript-Code, der die Suche nach den spezifischen Verkehrsmeldungen ermöglicht, kann in Listing 7.1 betrachtet werden. Der Query wird, sofern vorhanden, mit der Suchanfrage (*filter*), dem Bundesland (*region*) und der Autobahn (*road*) angereichert und ausgeführt. Liegen keine der drei Informationen vor, werden dementsprechend alle Verkehrsmeldungen aller Autobahnen und Bundesländer angezeigt.

Listing 7.1: Suche nach Verkehrsmeldungen

```
1 function search() {
2     $("#searchResult").empty();
3
4     let region = $('#region').val();
5     let road = $('#road').val();
6
7     if (subscription) {
8         subscription.unsubscribe();
```

```
9       }
10
11      query = DB.Traffic.find();
12      if (road) {
13          query = query.equal("road", road);
14      }
15      if (region) {
16          query = query.equal("region", region);
17      }
18      if (filter) {
19          query = query.where({
20              $text: {
21                  $search: filter,
22                  $language: "de"
23              }
24          });
25      }
26  }
```

Da gegenwärtig - wie eingangs erwähnt - kein Zugriff auf die Text-Indizes während der Verarbeitung der Volltextsuchanfrage möglich ist (Siehe auch Abschnitt 6.3), wird die Sprache explizit spezifiziert. Bei Abhandensein der Sprachangabe würde normalerweise die Standardsprache des Text-Index verwendet werden[Monv]. Obwohl der Text-Index mit Deutsch als Standardsprache erstellt wurde, würde ohne diese ausdrückliche Angabe mangels Zugriff dennoch Englisch verwendet werden. Siehe auch Abschnitt 4.2.

7.3.2 Die Erfassung der Verkehrsinformationen

Im Folgenden sind einige Ausschnitte des Javascript-Codes zu sehen, der die Verkehrsmeldungen erfasst und in die lokale Datenbank speichert. Ebenfalls zu sehen ist die Unterscheidung von bereits existierenden Verkehrsmeldungen in der Methode *saveTraffic*:

```
1  function saveTraffic(traffic, info) {
2      if (!traffic) {
3          newTraffic(info);
4      } else {
5          editTraffic(traffic, info);
6      }
7  }
```

Existiert die *report_id* bereits in der Datenbank, wird die dazugehörige Verkehrsinformation mitsamt der erfassten Meldung an die Methode *editTraffic* übergeben. Diese entscheidet dann, ob die Verkehrsmeldung aktualisiert werden muss oder nicht:

```
1  function editTraffic(traffic, info) {
2      let location = getLocation(info.message);
3
4      if (traffic.what != location.what ||
5          traffic.where != location.where)
6      {
7          traffic.what = location.what;
8          traffic.where = location.where;
9          traffic.update();
10     }
11 }
```

Existiert die Verkehrsmeldung laut ihrer *report_id* bislang noch nicht, wird die erfasste Meldung an die Methode *newTraffic* übergeben:

```
1  function newTraffic(info) {
2      let location = getLocation(info.message);
3
4      traffic = new DB.Traffic();
5      traffic.report_id = info.message.id;
6      traffic.what = location.what;
7      traffic.where = location.where;
8      traffic.road = info.road;
9      traffic.region = info.region;
10     traffic.language = "de";
11     traffic.insert();
12 }
```

Wie in Zeile 10 zu erkennen ist, wird explizit angegeben, dass das Dokument in der deutschen Sprache vorliegt. Normalerweise würde die Standardsprache des Text-Index verwendet werden, wenn keine Sprache spezifiziert wurde[Mono] (Siehe auch Abschnitt 4.2 und 6.3). Da gegenwärtig während der Verarbeitung der Volltextsuchanfrage kein Zugriff auf die Text-Indizes möglich ist, muss die Sprache explizit definiert werden.

Die Methode *fetchRoad* wird alle 30 Sekunden für jede der 24 unterstützten Autobahnen und für jedes der 16 Bundesländer aufgerufen:

```
1  const BASE_URL = "https://www.verkehrsinformation.de/";
2
3  async function fetchRoad(road, region) {
4      let url = BASE_URL + "?road=" + road + "&region=" + region;
5
```

```
6       $.get(url, function (data) {
7           $(data)
8               .find(".oldmsg, .newmsg")
9               .each(function (i, msg) {
10                  DB.Traffic.find()
11                      .equal("report_id", msg.id)
12                      .singleResult(traffic => {
13                          saveTraffic(traffic, {
14                              message: msg,
15                              road: road,
16                              region: region
17                          });
18                      });
19              });
20      });
21  }
```

Daraus resultieren $16 * 24 = 384 * 2 = 786$ Aufrufe pro Minute. Die Applikation ist allerdings vollständig konfigurierbar: Sowohl die Zeitangabe von 30 Sekunden, also auch die Anzahl der Autobahnen lässt sich auf Wunsch anpassen.

7.4 Evaluation

Die Funktionalität der entwickelten Applikation kann anhand der Abbildung 2.3 evaluiert werden. Jedes der drei relevanten Blätter des Binär-Baums ist in der Applikation definiert und erzeugt einen visuellen Effekt:

1. *add*: Gibt es eine neue Übereinstimmung mit der Suchanfrage, wird ein *add*-Event von Orestes geliefert. Die im Event enthaltenen Informationen werden zu den bisherigen Suchergebnissen hinzugefügt (Siehe Zeile 3 - 7 in Listing 7.2)

2. *change*: Hat sich die Verkehrsmeldung geändert und wurde daher aktualisiert, wird mit einem *change*-Event darauf reagiert. Die entsprechende Verkehrsmeldung wird anhand der *report_id* herausgesucht und statt Grün nun Gelb markiert (Siehe Zeile 8 - 14 in Listing 7.2)

3. *remove*: Liegt ein *remove*-Event vor, weil die Verkehrsmeldung als veraltet identifiziert und daher gelöscht wurde, wird das entsprechende Ergebnis ebenfalls aus den angezeigten Suchergebnissen entfernt (Siehe Zeile 15 - 17 in Listing 7.2). Eine Verkehrsmeldung wird in der Standardeinstellung der POC nach einer Stunde als veraltet angesehen. Diese Zeitangabe lässt sich allerdings konfigurieren.

Listing 7.2: Events der Volltextsuche

```
var onNext = event => {
    switch (event.matchType) {
        case "add":
            $("#searchResult").append(
                '<div class="item added" id="' + event.data.
    ↪ report_id +'"><p>' + event.data.where + "</p><p>" + event.
    ↪ data.what + "</p></div>"
            );
            break;
        case "change":
            $("#searchResult")
                .find("#" + event.data.report_id)
                .each(function(i, message) {
                    $(message).css("item changed");
                });
            break;
        case "remove":
            $("#searchResult").remove("#" + event.data.report_id
    ↪ );
            break;
    }
};
```

7.4.1 Performance und Skalierbarkeit

In [Winc] wird die horizontale Skalierbarkeit von InvaliDB bewiesen. Dies bedeutet, dass die Überprüfung der Übereinstimmung registrierter Queries mit neuen indexierten Dokumenten - sogenannte *Query Subscriptions* - und eingehende Schreiboperationen linear mit der Anzahl der Maschinen in einem Cluster skalieren. Um die in dieser Arbeit implementierte Volltextsuche in InvaliDB produktiv einsetzen zu können, muss gewährleistet werden, dass diese Charakteristika von InvaliDB immer noch erfüllt ist. Um dies zu überprüfen, muss ein äquivalenter Test unter identischen Bedingungen wie in [Winc] stattfinden. Dazu würden die folgenden Komponenten benötigt:

- Eine Cloud-Umgebung die multiple Rechner enthält, die wiederum mit identischer Hardware ausgestattet sind
- Jeder Rechner benötigt zwei virtuelle Server für die InvaliDB Clients: Einen für die Ausführung der Schreiboperationen sowie einen für die *Query Subscriptions*.

- Einen Redis-Server, der die Kommunikation zwischen InvaliDB und dem Apache-Storm Cluster übernimmt

Im Rahmen dieser Arbeit ist allerdings weder der Versuchsaufbau noch die Durchführung dessen möglich. Es wurde daher beschlossen, dass der Laptop des Autors für eine Überprüfung genügen muss. Auf Grund dessen ist eine Überprüfung der Skalierung auf multiplen Rechnern nicht möglich.
In [Winc] wurden 1000 Schreiboperationen pro Sekunde durchgeführt, mit dem Ziel, nach 60 Sekunden insgesamt 1000 *matches* zu erzielen. Daraus resultieren $1000/60 = 16.66 \approx 17$ *matches* pro Sekunde. In [Winc] wurden Intel Xeon E5-2620 v2 Prozessoren mit jeweils 6 Kernen verwendet. Der Laptop des Autors hingegen verfügt über einen Intel i7 4710HQ mit 4 Kernen und 2.50 Ghz sowie 16 GB RAM. Damit konnten maximal 210 Schreiboperationen pro Sekunde durchgeführt werden. Damit genügend Ressourcen für die Verwaltung der *Query-Subscribtions* zur Verfügung stehen, wurde entschieden, die Schreiboperationen auf 100 pro Sekunde zu limitieren. 100 Schreiboperationen entsprechen $\frac{1}{10}$ der Schreiboperationen in [Winc], womit sich ein Ziel von $\frac{1000}{10} = 100$ *matches* nach 60 Sekunden ergibt. Basierend auf den in [Winc] erzielten 17 *matches* pro Sekunde resultiert ein Gesamtziel von 100 *matches* innerhalb von 60 Sekunden in $100/60 = 1.66 \approx 2$ *matches* pro Sekunde.

Präparierungsphase

Es wurden n *Query-Subscriptions* registriert, wobei 2 dieser *Subscriber match*-Wörter (Alpha und Beta) enthielten. Für jedes eingehende *add*-Event wurde ein Zähler erhöht, der jede Sekunde ausgegeben und anschließend auf 0 zurückgesetzt wurde.

Messphase

Es werden 100 Schreiboperationen pro Sekunde ausgeführt. Jeweils 2 der geschriebenen Datensätze enthalten die in der Präparierungsphase registrierten *match*-Wörter (Alpha und Beta), so das es pro Sekunde auf 100 Datensätze genau 2 Übereinstimmungen gibt. Daraus resultieren $2 * 60 = 120$ *matches* in 60 Sekunden.

Abbruchbedingung

Es wird erwartet, dass pro Sekunde der Zähler der *matches* genau 2 entspricht, da pro Sekunde jeweils 2 übereinstimmende Datensätze geschrieben werden. Variiert diese Anzahl, deutet dies auf eine Überlast seitens InvaliDB hin, da es zu einem Stau der Überprüfungen auf Übereinstimmungen kommt. Tritt dieser Effekt auf, wird die Anzahl der *Query-Subscriptions* um 100 reduziert und der Test wiederholt.

Ergebnisse

In der Startphase mit aktiven Redis, MongoDB und InvaliDB liegt die CPU-Last bei 8%. Bei 100 Schreibvorgängen und 2 *matches* pro Sekunde zeigten sich bei der CPU-Last während der Testläufe deutliche Schwankungen von bis zu 15% (Siehe Abbildung 7.8). Ab 3800 *Query-Subscriptions* lag die CPU-Auslastung bei $83 - 95\%$ (Siehe Abbildung 7.5). Ab 3900 *Query-Subscriptions* erreichte die CPU-Auslastung bereits mitunter 100% (Siehe Abbildung 7.6).

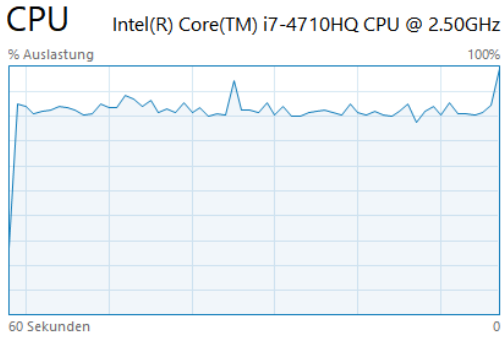


Abbildung 7.5: CPU-Auslastung für 3800 *Query-Subscriptions*

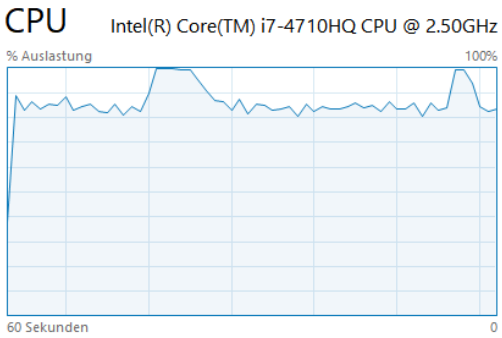


Abbildung 7.6: CPU-Auslastung für 3900 *Query-Subscriptions*

Mit 4400 *Query-Subscriptions* erreichte die CPU-Auslastung $94 - 100\%$ (Siehe Abbildung 7.7).

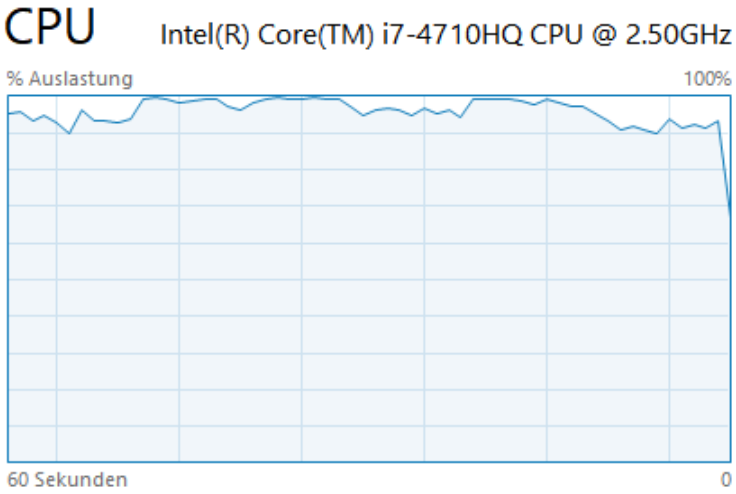


Abbildung 7.7: CPU-Auslastung für 4400 *Query-Subscriptions*

Eine mögliche Ursache, die für derartige Schwankungen verantwortlich sein könnte, sind *Garbage Collection*-Zyklen des Java Garbage Collectors. Anhand des Java Flight Recorder wurden für diverse registrierte *Query-Subscriptions* Profile erstellt. In Abbildung 7.8 ist ein Profil für 3900 *Query-Subscriptions* zu sehen. Es gab in der Tat Zyklen, die mehr als 500ms benötigten und daher zu derartigen CPU-Schwankungen führen könnten. Aus Zeitmangel wurde dieser Verdacht allerdings nicht näher analysiert. Da ab 3800 *Query-Subscriptions* mitunter CPU-Schwankungen von über 90% zu beobachten waren und bei höherer Auslastung gelegentlich Variationen der *matches* pro Sekunde auftraten, wurde diese Anzahl als repräsentativer Wert gewählt.

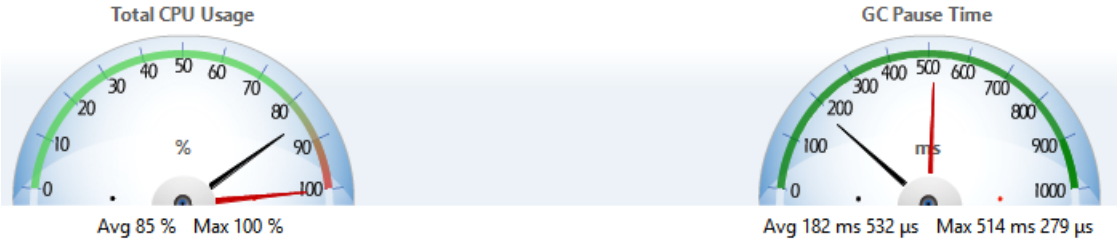


Abbildung 7.8: Profil des Java Fligt-Recorder bei 3900 *Query-Subscriptions*

In [Winc] wurden 1000 Schreiboperationen pro Sekunde ausgeführt mit 1000 registrierten *Query-Subscriptions*. Dies entspricht insgesamt $1000 * 1000 = 1.000.000$ *match*-Operationen pro Sekunde. Auf dem wesentlich leistungsschwächeren Laptop des Autors, auf dem im Gegensatz zu [Winc] InvaliDB, Redis und MongoDB parallel ausgeführt wurden, konnten bis zu $100 * 3800 = 380.000$ *match*-Operationen pro Sekunde ausgeführt werden.

Daraus folgt, dass die Implementation der Volltextsuche die Skalierbarkeit von InvaliDB nicht negativ beeinflusst.

8 Fazit

Das Ziel dieser Arbeit war die inkrementelle Auswertung von MongoDB Volltextsuchanfragen. Dazu musste eine Volltextsuche in Orestes implementiert werden, deren Verhalten und Benutzbarkeit sich an der Volltextsuche von MongoDB orientiert. Dazu wurde zunächst in Kapitel 3 „Volltextsuche“ die Grundbegriffe einer Volltextsuche erläutert. Dazu zählen, was unter einer Volltextsuche zu verstehen ist, welche Komponenten sie umfasst und wie sie funktioniert. Anschließend übernahm das Kapitel 4 „Volltextsuche in MongoDB“ die Aufklärung über den Aufbau und der Funktionsweise der MongoDB Volltextsuche. Zunächst musste erklärt werden, welche Maßnahmen notwendig sind, um eine Volltextsuche auf Dokumenten auszuführen. Dazu wurde zunächst die Erstellung von Text-Indizes, deren Umfang und Limitierungen sowie die Spezifikation der zu verwendenden Sprache beschrieben. Die Angabe der zu verwendenden Sprache war für die weiterführende Erklärung der Begrifflichkeiten *Stopwörter* und *Stemming* essentiell. Diese Begriffe wurden bereits initial im Kapitel „Volltextsuche“ erwähnt und grundlegend beschrieben. Im Kapitel „Volltextsuche in MongoDB“ erhielten diese Begriffe allerdings eine wesentlich detailliertere Beschreibung im Kontext von MongoDB und dessen Volltextsuche. Nachdem diese Grundbegriffe ausführlich beschrieben und anhand von Beispielen demonstrativ erläutert wurden, erfolgte im Kapitel 5 „Gegenüberstellung und Vergleich anderer Volltextsuchen“ eine Gegenüberstellung von MongoDB und anderen großen Vertretern von Volltextsuchen. Diese Vertreter waren Elasticsearch (Siehe Abschnitt 5.1), SolR (Siehe Abschnitt 5.2) und die drei populärsten relationalen Datenbanken des Jahres 2017: MySQL, Oracle und PostgreSQL (Siehe Abschnitt 5.3). In jeder Gegenüberstellung wurde zunächst die notwendigen Begrifflichkeiten vorgestellt. Anschließend erfolgte eine detaillierte Betrachtung der im Kontext von Volltextsuchen relevanten Konzepte, i.d.R. unterstützt durch entsprechende Beispiele. Die Kernfrage, dessen Auseinandersetzung den Abschluss der jeweiligen Gegenüberstellung bildete, war, ob die betrachtete Volltextsuche ebenfalls geeignet wäre, die inkrementelle Auswertung von Echtzeit-Anfragen vorzunehmen und somit eine passable Alternative zu MongoDB darstellt. Diese Frage konnte für die drei betrachteten Volltextsuchen einmal mit „Ja“ (Elasticsearch), einmal mit „gegebenenfalls“ (SolR) und einmal mit „Nein“ (Relationale Datenbanken) beantwortet werden. Auf Grund der hohen Skalierbarkeit, der verwendeten *Inverted Indizes* und nicht zuletzt auf Grund der *Percolator Queries* kann Elasticsearch als eine durchaus konkurrenzfähige Alternative zu MongoDB betrachtet werden. Für SolR kann, mit Ausnahme der fehlenden *Percolator Queries*, eine ähnliche Aussage getroffen werden. Die drei betrachteten Vertreter der relationalen Datenbanken eignen sich,

auf Grund der teils unvollständigen Komponenten der Volltextsuche und der erschwerten und komplexeren Skalierbarkeit, am wenigsten als passable Alternative. Im Kapitel 6 „Implementation der Volltextsuchanfrage in InvaliDB“ wird die eigentliche Implementation der im Rahmen dieser Arbeit konzipierten Volltextsuche vorgestellt. Zunächst wurde ein Überblick der verwendeten Technologien - namentlich Lucene - gegeben, sowie kurz die Beweggründe offenbart, die zu dessen Auswahl führten. Anschließend wurden die einzelnen Komponenten und deren Funktionalität beschrieben. Eine Demonstration der Interaktion dieser Komponenten erfolgte anhand eines beispielhaften Ablaufs einer Volltextsuchanfrage in einem eigenen Unterkapitel. Das Kapitel 7 „Evaluation der Volltextsuchanfrage“ bildet den Abschluss und widmet sich der Evaluation der Implementation. Nach einer einleitenden Beschreibung der verwendeten Baqend-API erfolgte die Demonstration des praktischen Nutzens einer Echtzeit-Volltextsuche, indem eine realistische Problemstellung innerhalb einer *Proof of Concept*-Applikation effektiv gelöst wurde. Anschließend wurde sich der Evaluation der entwickelten Volltextsuche im Kontext von InvaliDB angenommen. Es galt zu beweisen, dass die Skalierbarkeit von InvaliDB auch bei Verwendung der Volltextsuche gewährleistet ist. Dies wurde, soweit es die Mittel dieser Arbeit zuließen, mittels der modifizierten *Proof of Concept*-Applikation gezeigt. Dazu wurde die Beweisführung aus [Winc] auf ein reduziertes Schema übertragen. In mehreren darauf basierenden Versuchsdurchläufen konnte gezeigt werden, dass die Skalierbarkeit auch unter Verwendung der Volltextsuche gewährleistet ist. Damit wurde das angestrebte Ziel, die Entwicklung einer inkrementellen Auswertung von MongoDB-Volltextsuchanfragen, erreicht.

8.1 Ausblick

Neben MongoDB existieren, wie in dieser Arbeit vorgestellt, diverse andere Volltextsuchen, die ebenfalls für eine inkrementelle Auswertung von Volltextsuchanfragen geeignet wären. Es wäre durchaus interessant, die Auswirkungen der Verwendung von z.B. Elasticsearch anstelle von MongoDB in InvaliDB zu begutachten.
Die in dieser Arbeit implementierte Volltextsuche basiert - wie SolR und Elasticsearch - auf Lucene. Auf Grund dessen werden nicht alle Sprachen unterstützt, die MongoDB[1] anbietet. Es fehlen insgesamt drei Sprachen, für dessen Unterstützung eigene Lucene-Komponenten geschrieben werden müssten:

1. *Urdu* (Persisch)
2. Vereinfachtes Chinesisch (*simplified chinese*), auch als *hans* bezeichnet
3. Traditionelles Chinesisch (*traditional chinese*), auch als *hant* bezeichnet

[1] Eine vollständige Übersicht der von MongoDB unterstützten Sprachen kann in der Dokumentation[Monu] betrachtet werden

8.1.1 Fehlender Text-Index Zugriff

Im Abschnitt 6.2.4 „Das Meta-Package“ wurde folgendes geschrieben:

> „Sind keine Text-Indizes angegeben, werden alle Felder des vorliegenden Dokuments verwendet, sofern sie textuell verwertbaren Inhalt aufweisen.“

Dazu heißt es in Abschnitt 4.1:

> „Es mag jedoch die Frage aufkommen, warum stattdessen nicht Standardmäßig alle Felder durchsucht werden und die Angabe eines Text-Index nur als eine optionale Filtereinstellung dienen kann. Dies ist aus mehreren Gründen nicht zweckmäßig. Zum einen wäre ein solches Vorhaben mit einem erheblichen Mehraufwand verbunden, da sämtliche Felder des Dokuments zunächst daraufhin untersucht werden müssten, ob sie verwertbare Text-Inhalte aufweisen. Zum anderen würden dann auch Inhalte untersucht werden, die möglicherweise keine relevanten Inhalte aufweisen.“

Der momentane Zustand ist dem Umstand der fehlenden Möglichkeit geschuldet, innerhalb des Prozesses der Volltextsuche auf bereits registrierte Text-Indizes zuzugreifen. Der Grund dafür wurde in Abschnitt 6.3 erklärt:

> „Der Zugriff auf die Text-Indizes ist allerdings während der Verarbeitung der Volltextsuchanfrage bislang nicht möglich. Die Implementierung des Zugriffs würde weitreichende Eingriffe in die bestehende Implementation von InvaliDB erfordern. Da dies allerdings den Umfang dieser Arbeit übersteigen würde, wurde entschieden, den Zugriff separat und zu einem späteren Zeitpunkt zu implementieren.“

Daher war diese Lösung der einzige Weg, eine Volltextsuche zu ermöglichen, ohne auf registrierte Text-Indizes zuzugreifen. Durch diesen Zustand erreicht die bestehende Implementation womöglich nicht ihr volles Performance-Potential. Sollte zukünftig allerdings eine derartige Lösung implementiert werden, kann dieser Zugriff leicht in der Klasse *TextIndices* implementiert und die derzeitige Lösung deaktiviert werden. In Abschnitt 6.2.4 „Das Meta-Package“ wird zur Klasse *TextIndices* folgende Aussage getroffen:

> „Die Instanz der Klasse *TextIndices* ist eine zentrale Schnittstelle, die für die Verwaltung und dem Abruf der jeweiligen Text-Indizes verantwortlich ist.“

8.1.2 Fehlende Funktionalität im Baqend-Dashboard

In Abschnitt 7.2 wurde bereits erwähnt, dass das Dashboard von Baqend bisher über keine Funktionalität verfügt, Text-Indizes anzulegen. Abhilfe könnte der folgende Ablauf schaffen: sobald die Erstellung eines Index für ein Feld vom Typ *string* angefordert wird,

muss der Benutzer entscheiden, ob ein Text-Index oder ein normaler Index generiert werden soll. Zusätzliche Informationen, wie die Standardsprache des Text-Index (Siehe Abschnitt 4.2) und die Gewichtung (Siehe Abschnitt 4.5), könnten erfasst werden, indem die zu verwendende Sprache aus der Liste von unterstützten Sprachen ausgewählt sowie eine Gewichtung vom Benutzer eingetragen wird. Die Standardauswahl wäre die Sprache Englisch und die Gewichtung 1. Eine Bestätigung würde dann die Erstellung des Text-Index auslösen.

Eidesstattliche Erklärung

Ich, Randy Schütt, Matrikel-Nr. 6209096, versichere hiermit, dass ich meine Masterarbeit mit dem Thema

Inkrementelle Auswertung von MongoDB-Volltextsuchanfragen

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.
Mir ist bekannt, dass ich meine Masterarbeit zusammen mit dieser Erklärung fristgemäß nach Vergabe des Themas in dreifacher Ausfertigung und gebunden im Prüfungsamt der Universität Hamburg abzugeben oder spätestens mit dem Poststempel des Tages, an dem die Frist abläuft, zu senden habe.

Hamburg, den 17. Februar 2018

Literaturverzeichnis

[Asl18] Salman Aslam. Twitter by the numbers: Stats, demographics & fun facts. https://www.omnicoreagency.com/twitter-statistics/, 2018. (Eingesehen am 23.01.2018).

[Baq17] Baqend. Real-time-queries. https://www.baqend.com/guide-next/topics/realtime/, 2017. (Eingesehen am 09.07.2017).

[Bro02] Sonja Brodersen. Reducing words to their root form. https://files.ifi.uzh.ch/cl/ict-open/mue1/porter/stemming/node2.html, 2002. (Eingesehen am 15.02.2018).

[bso] Bson. http://bsonspec.org/. (Eingesehen am 10.02.2018).

[BVa] Elasticsearch BV. Common terms query. https://www.elastic.co/guide/en/elasticsearch/reference/master/query-dsl-common-terms-query.html. (Eingesehen am 18.12.2017).

[BVb] Elasticsearch BV. Custom analyzers. https://www.elastic.co/guide/en/elasticsearch/guide/master/custom-analyzers.html. (Eingesehen am 18.12.2017).

[BVc] Elasticsearch BV. Document metadata. https://www.elastic.co/guide/en/elasticsearch/guide/current/_document_metadata.html. (Eingesehen am 02.01.2018).

[BVd] Elasticsearch BV. Elasticsearch. https://www.elastic.co/. (Eingesehen am 25.10.2017).

[BVe] Elasticsearch BV. Exact values versus full text. https://www.elastic.co/guide/en/elasticsearch/guide/master/_exact_values_versus_full_text.html. (Eingesehen am 18.12.2017).

[BVf] Elasticsearch BV. Full text queries. https://www.elastic.co/guide/en/elasticsearch/reference/master/full-text-queries.html. (Eingesehen am 18.12.2017).

[BVg] Elasticsearch BV. Inverted index. https://www.elastic.co/guide/en/elasticsearch/guide/master/inverted-index.html. (Eingesehen am 18.12.2017).

[BVh] Elasticsearch BV. Mapping and analysis. https://www.elastic.co/guide/en/elasticsearch/guide/master/mapping-analysis.html. (Eingesehen am 18.12.2017).

[BVi] Elasticsearch BV. Master election. https://www.elastic.co/guide/en/elasticsearch/reference/6.x/modules-discovery-zen.html#master-election. (Eingesehen am 18.12.2017).

[BVj] Elasticsearch BV. Match phrase prefix query. https://www.elastic.co/guide/en/elasticsearch/reference/master/query-dsl-match-query-phrase-prefix.html. (Eingesehen am 18.12.2017).

[BVk] Elasticsearch BV. Match phrase query. https://www.elastic.co/guide/en/elasticsearch/reference/master/query-dsl-match-query-phrase.html. (Eingesehen am 18.12.2017).

[BVl] Elasticsearch BV. Match query. https://www.elastic.co/guide/en/elasticsearch/reference/master/query-dsl-match-query.html. (Eingesehen am 18.12.2017).

[BVm] Elasticsearch BV. Multi match query. https://www.elastic.co/guide/en/elasticsearch/reference/master/query-dsl-multi-match-query.html. (Eingesehen am 18.12.2017).

[BVn] Elasticsearch BV. Near real-time search. https://www.elastic.co/guide/en/elasticsearch/guide/current/near-real-time.html. (Eingesehen am 18.12.2017).

[BVo] Elasticsearch BV. Percolate query. https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-percolate-query.html. (Eingesehen am 29.12.2017).

[BVp] Elasticsearch BV. Query dsl. https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html. (Eingesehen am 18.12.2017).

[BVq] Elasticsearch BV. Reducing words to their root form. https://www.elastic.co/guide/en/elasticsearch/guide/current/stemming.html. (Eingesehen am 15.02.2018).

[BVr] Elasticsearch BV. Search apis. https://www.elastic.co/guide/en/elasticsearch/reference/0.90/search.html. (Eingesehen am 12.02.2018).

[BVs] Elasticsearch BV. Search lite. https://www.elastic.co/guide/en/elasticsearch/guide/master/search-lite.html. (Eingesehen am 18.12.2017).

[BVt] Elasticsearch BV. Shards & replicas. https://www.elastic.co/guide/en/elasticsearch/reference/current/_basic_concepts.html#getting-started-shards-and-replicas. (Eingesehen am 16.02.2018).

[BVu] Elasticsearch BV. Stop stopping stop words: a look at common terms query. https://www.elastic.co/blog/stop-stopping-stop-words-a-look-at-common-terms-query. (Eingesehen am 18.12.2017).

[BVv] Elasticsearch BV. Zen discovery. https://www.elastic.co/guide/en/elasticsearch/reference/6.1/modules-discovery-zen.html. (Eingesehen am 12.02.2018).

[CB09] Adriana Kosior Cornelia Baldauf, Valentin Heinz. Volltextsuche und text mining. https://user.phil.hhu.de/~petersen/Einf_CL_0910/material/Volltextsuche_Textmining.pdf, 2009. (Eingesehen am 15.02.2018).

[Cie05] Rafael Cieslik. Volltextsuche und text mining. https://user.phil-fak.uni-duesseldorf.de/~petersen/Einf_CL/Referat_2005-01-20_Volltextsuche.pdf, 2005. (Eingesehen am 15.02.2018).

[Cod70a] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13, 1970.

[Cod70b] Edgar F. Codd. A relational model of data for large shared data banks. Technical report, IBM Research Laboratory, San Jose, California, 1970.

[con17] Wikipedia contributors. Stemming — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Stemming&oldid=816455386#Error_metrics, 2017. (Eingesehen am 8. Februar 2018).

[Cora] Oracle Corporation. Context index and dml. https://docs.oracle.com/cd/B28359_01/text.111/b28303/ind.htm#CIHDJEED. (Eingesehen am 11.01.2018).

[Corb] Oracle Corporation. Full-text search functions. https://dev.mysql.com/doc/refman/5.7/en/fulltext-search.html. (Eingesehen am 11.01.2018).

[Corc] Oracle Corporation. Full-text stopwords. https://dev.mysql.com/doc/refman/5.7/en/fulltext-stopwords.html. (Eingesehen am 11.01.2018).

[Cord] Oracle Corporation. Full-text stopwords. https://dev.mysql.com/doc/refman/5.7/en/fulltext-stopwords.html#idm139807070958192. (Eingesehen am 18.01.2018).

[Core] Oracle Corporation. Indexing with oracle text. https://docs.oracle.com/cd/B19306_01/text.102/b14217/ind.htm#sthref110. (Eingesehen am 23.01.2018).

[Corf] Oracle Corporation. Innodb full-text index design. https://dev.mysql.com/doc/refman/5.6/en/innodb-fulltext-index.html#innodb-fulltext-index-design. (Eingesehen am 23.01.2018).

[Corg] Oracle Corporation. The json data type. https://dev.mysql.com/doc/refman/5.7/en/json.html. (Eingesehen am 18.01.2018).

[Corh] Oracle Corporation. Oracle text supplied stoplists. https://docs.oracle.com/cd/B28359_01/text.111/b28304/astopsup.htm#CCREF1400. (Eingesehen am 18.01.2018).

[Cori] Oracle Corporation. Querying with oracle text. https://docs.oracle.com/cd/B28359_01/text.111/b28303/query.htm#BABGAIHJ. (Eingesehen am 11.01.2018).

[Corj] Oracle Corporation. Stemming for full-text. https://dev.mysql.com/worklog/task/?id=2423. (Eingesehen am 11.01.2018).

[Cork] Oracle Corporation. Stopwords and stopthemes. https://docs.oracle.com/cd/B28359_01/text.111/b28303/ind.htm#CCAPP9074. (Eingesehen am 11.01.2018).

[Cor14] Oracle Corporation. Native json-unterstützung in der datenbank 12.1.0.2. https://apex.oracle.com/pls/apex/germancommunities/apexcommunity/tipp/3161/index.html, 2014. (Eingesehen am 18.01.2018).

[DE17] DB-Engines. Db-engines ranking von relational dbms. https://db-engines.com/de/ranking, 2017.

[dta] MongoDB dev team. Contributing to the mongodb project. https://github.com/mongodb/mongo/blob/master/CONTRIBUTING.rst. (Eingesehen am 07.01.2018).

[dtb] MongoDB dev team. Englische stopwörter in mongodb. https://github.com/mongodb/mongo/blob/master/src/mongo/db/fts/stop_words_english.txt. (Eingesehen am 07.01.2018).

[FG14] Florian Bücklers und Norbert Ritter Felix Gessert. Orestes: A scalable database-as-a-service architecture for low latency. *Data Engineering Workshops (ICDEW)*, März 2014.

[FG17a] Michael Schaarschmidt Eiko Yoneki Wolfram Wingerath Norbert Ritter Felix Gessert, Erik Witt. Quaestor: Scalable and consistent query result caching on the web's infrastructure. http://www.vldb.org/pvldb/vol10/p1670-gessert.pdf, 2017.

[FG17b] Norbert Ritter Felix Gessert. Objects restfully encapsulated in standard formats. https://vsis-www.informatik.uni-hamburg.de/vsis/research/lookproject/55, 2017. (Eingesehen am 25.10.2017).

[Fiv09] Five questions with michael widenius - founder and original developer of mysql. https://web.archive.org/web/20090313160628/http://www.opensourcereleasefeed.com/interview/show/five-questions-with-michael-widenius-founder-and-original-developer-of-mysql, 2009.

[Fle] Hochschule Flensburg. Heapsort. http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/heap/heap.htm.

[Foua] Apache Software Foundation. Analyzer. https://lucene.apache.org/core/7_1_0/core/org/apache/lucene/analysis/Analyzer.html. (Eingesehen am 24.01.2018).

[Foub] Apache Software Foundation. The apache jakarta project. http://jakarta.apache.org/. (Eingesehen am 16.02.2018).

[Fouc] Apache Software Foundation. Apache lucene. https://lucene.apache.org/. (Eingesehen am 25.10.2017).

[Foud] Apache Software Foundation. Apache storm. http://storm.apache.org/. (Eingesehen am 25.10.2017).

[Foue] Apache Software Foundation. Community. http://lucene.apache.org/solr/community.html. (Eingesehen am 16.02.2018).

[Fouf] Apache Software Foundation. Defining fields. http://lucene.apache.org/solr/guide/7_2/defining-fields.html. (Eingesehen am 02.01.2018).

[Foug] Apache Software Foundation. Germannormalizationfilter. https://lucene.apache.org/core/4_2_0/analyzers-common/org/apache/lucene/analysis/de/GermanNormalizationFilter.html. (Eingesehen am 24.01.2018).

[Fouh] Apache Software Foundation. Inverted indexing. https://lucene.apache.org/core/3_0_3/fileformats.html#Inverted%20Indexing. (Eingesehen am 02.01.2018).

[Foui] Apache Software Foundation. Json request api. https://lucene.apache.org/solr/guide/7_1/json-request-api.html. (Eingesehen am 12.02.2018).

[Fouj] Apache Software Foundation. Lowercasefilter. https://lucene.apache.org/core/7_1_0/core/org/apache/lucene/analysis/LowerCaseFilter.html. (Eingesehen am 24.01.2018).

[Fouk] Apache Software Foundation. Lucene token stream. https://lucene.apache.org/core/7_1_0/core/org/apache/lucene/analysis/TokenStream.html. (Eingesehen am 24.01.2018).

[Foul] Apache Software Foundation. Normalization. https://lucene.apache.org/core/7_1_0/core/org/apache/lucene/search/similarities/Normalization.html. (Eingesehen am 24.01.2018).

[Foum] Apache Software Foundation. Schemaless mode. http://lucene.apache.org/solr/guide/7_2/schemaless-mode.html. (Eingesehen am 02.01.2018).

[Foun] Apache Software Foundation. Setting up an external zookeeper ensemble. https://lucene.apache.org/solr/guide/6_6/setting-up-an-external-zookeeper-ensemble.html. (Eingesehen am 12.02.2018).

[Fouo] Apache Software Foundation. Solr. http://lucene.apache.org/solr/. (Eingesehen am 25.10.2017).

[Foup] Apache Software Foundation. Solr feature. http://lucene.apache.org/solr/features.html. (Eingesehen am 16.02.2018).

[Fouq] Apache Software Foundation. Solrcloud. https://lucene.apache.org/solr/guide/6_6/solrcloud.html. (Eingesehen am 12.02.2018).

[Four] Apache Software Foundation. Standardfilter. https://lucene.apache.org/core/7_1_0/core/org/apache/lucene/analysis/standard/StandardFilter.html. (Eingesehen am 24.01.2018).

[Fous] Apache Software Foundation. Standardtokenizer. https://lucene.apache.org/core/7_1_0/core/org/apache/lucene/analysis/standard/StandardTokenizer.html. (Eingesehen am 24.01.2018).

[Fout] Apache Software Foundation. Stopfilter. https://lucene.apache.org/core/7_1_0/core/org/apache/lucene/analysis/StopFilter.html. (Eingesehen am 24.01.2018).

[Fouu] Apache Software Foundation. Tokenfilter. https://lucene.apache.org/core/7_1_0/core/org/apache/lucene/analysis/TokenFilter.html. (Eingesehen am 24.01.2018).

[Fouv] Apache Software Foundation. Tokenizer. https://lucene.apache.org/core/7_1_0/core/org/apache/lucene/analysis/Tokenizer.html. (Eingesehen am 24.01.2018).

[Fouw] Wikimedia Foundation. Cirrus. https://de.wikipedia.org/w/index.php?title=Hilfe:Suche/Cirrus&oldid=173287175. (Eingesehen am 23.01.2018).

[Fou06] Apache Software Foundation. Cnet code contribution. https://issues.apache.org/jira/browse/SOLR-1, 2006. (Eingesehen am 02.01.2018).

[Fou10] Apache Software Foundation. [vote] merge lucene/solr development .. http://lucene.markmail.org/thread/cmbek34vfycb6sfc, 2010. (Eingesehen am 16.02.2018).

[Fou12] Apache Software Foundation. Solr near realtime search. https://wiki.apache.org/solr/NearRealtimeSearch, 2012. (Eingesehen am 12.02.2018).

[Fou13a] Apache Software Foundation. Implement saved searches a la elasticsearch percolator. https://issues.apache.org/jira/browse/SOLR-4587, 2013. (Eingesehen am 07.01.2018).

[Fou13b] Apache Software Foundation. In preparation for dynamic schema modification via rest api, add a managed schema facility. https://issues.apache.org/jira/browse/SOLR-4658, 2013. (Eingesehen am 02.01.2018).

[Gar05] Jesse James Garrett. Ajax: A new approach to web applications. https://pdfs.semanticscholar.org/c440/ae765ff19ddd3deda24a92ac39cef9570f1e.pdf, 2005.

[Groa] PostgreSQL Global Development Group. Dictionaries. https://www.postgresql.org/docs/9.1/static/textsearch-dictionaries.html. (Eingesehen am 11.01.2018).

[Grob] PostgreSQL Global Development Group. Full text search. https://www.postgresql.org/docs/9.5/static/textsearch.html. (Eingesehen am 11.01.2018).

[Groc] PostgreSQL Global Development Group. Gist and gin index types. https://www.postgresql.org/docs/8.3/static/textsearch-indexes.html. (Eingesehen am 23.01.2018).

[Grod] PostgreSQL Global Development Group. Json types. https://www.postgresql.org/docs/10/static/datatype-json.html. (Eingesehen am 18.01.2018).

[Groe] PostgreSQL Global Development Group. Stop words. https://www.postgresql.org/docs/9.1/static/textsearch-dictionaries.html#TEXTSEARCH-STOPWORDS. (Eingesehen am 11.01.2018).

[Jan14] Siva Prasad Rao Janapati. Apache solr – the inverted index. https://smarttechie.org/2014/07/10/apache-solr-the-inverted-index/, 2014. (Eingesehen am 02.01.2018).

[JL] Rob Allen Josh Lockhart, Andrew Smith. Middleware. https://www.slimframework.com/docs/concepts/middleware.html.

[Kar] Stefan Karzauninkat. Phrasensuche. http://www.suchfibel.de/2kunst/phrasen.htm. (Eingesehen am 28.11.2017).

[Kuc15] Rafal Kuc. Cnet code contribution. https://dzone.com/articles/solr-5-json-request-api-part-one, 2015. (Eingesehen am 02.01.2018).

[Kva16] Christian Kvalheim. Metadata. http://learnmongodbthehardway.com/schema/metadata/, 2016. (Eingesehen am 01.02.2018).

[Ltd17a] D4 Software Ltd. A brief history of json. http://blog.sqlizer.io/posts/json-history/, 2017. (Eingesehen am 10.02.2018).

[Ltd17b] Heise Media UK Ltd. Ten years of the lucene search engine at apache. http://www.h-online.com/open/news/item/Ten-years-of-the-Lucene-search-engine-at-Apache-1350761.html, 2017. (Eingesehen am 23.01.2018).

[MN] Steindorfer Jochen Markus Nichterl. Text mining. http://wwwdh.cs.fau.de/IMMD8/Lectures/TextMining/Material/nichterl_steindorfer_text_mining.pdf. (Eingesehen am 15.02.2018).

[Mona] Inc MongoDB. Case insensitivity. https://docs.mongodb.com/manual/core/index-text/#case-insensitivity. (Eingesehen am 16.02.2018).

[Monb] Inc MongoDB. Diacritic insensitivity. https://docs.mongodb.com/manual/core/index-text/#diacritic-insensitivity. (Eingesehen am 16.02.2018).

[Monc] Inc MongoDB. Diacritic insensitivity. https://docs.mongodb.com/manual/core/index-text/#text-index-diacritic-insensitivity. (Eingesehen am 25.10.2017).

[Mond] Inc MongoDB. Indexes. https://docs.mongodb.com/manual/indexes/#b-tree. (Eingesehen am 27.10.2017).

[Mone] Inc MongoDB. $meta: Sort. https://docs.mongodb.com/manual/reference/operator/projection/meta/#proj._S_meta. (Eingesehen am 25.10.2017).

[Monf] Inc MongoDB. Metadata and asset management. https://docs.mongodb.com/ecosystem/use-cases/metadata-and-asset-management/. (Eingesehen am 01.02.2018).

[Mong] Inc MongoDB. Model data to support keyword search. https://docs.mongodb.com/manual/tutorial/model-data-for-keyword-search/. (Eingesehen am 02.01.2018).

[Monh] Inc MongoDB. Mongodb. https://www.mongodb.com/. (Eingesehen am 25.10.2017).

[Moni] Inc MongoDB. Mongodb collections. https://docs.mongodb.com/v3.2/core/databases-and-collections/#collections. (Eingesehen am 27.10.2017).

[Monj] Inc MongoDB. Mongodb text-index. https://docs.mongodb.com/manual/core/index-text/. (Eingesehen am 27.10.2017).

[Monk] Inc MongoDB. Mongodb text-index gewichtungen. https://docs.mongodb.com/manual/tutorial/control-results-of-text-search/. (Eingesehen am 30.10.2017).

[Monl] Inc MongoDB. Negations. https://docs.mongodb.com/manual/reference/operator/query/text/#negations. (Eingesehen am 16.02.2018).

[Monm] Inc MongoDB. $search field. https://docs.mongodb.com/manual/reference/operator/query/text/#search-field. (Eingesehen am 16.02.2018).

[Monn] Inc MongoDB. Specify a language for text index. https://docs.mongodb.com/manual/tutorial/specify-language-for-text-index/. (Eingesehen am 25.10.2017).

[Mono] Inc MongoDB. Specify the default language for a text index. https://docs.mongodb.com/manual/tutorial/specify-language-for-text-index/#specify-the-default-language-for-a-text-index. (Eingesehen am 25.10.2017).

[Monp] Inc MongoDB. Specify the index language within the document. https://docs.mongodb.com/manual/tutorial/specify-language-for-text-index/#use-any-field-to-specify-the-language-for-a-document. (Eingesehen am 25.10.2017).

[Monq] Inc MongoDB. Specify the index language within the document. https://docs.mongodb.com/manual/tutorial/specify-language-for-text-index/#specify-the-index-language-within-the-document. (Eingesehen am 25.10.2017).

[Monr] Inc MongoDB. Supported languages and stop words. https://docs.mongodb.com/manual/core/index-text/#supported-languages-and-stop-words. (Eingesehen am 25.10.2017).

[Mons] Inc MongoDB. $text. https://docs.mongodb.com/manual/core/index-text/#versions. (Eingesehen am 16.02.2018).

[Mont] Inc MongoDB. Text index and sort. https://docs.mongodb.com/manual/core/index-text/#text-index-and-sort. (Eingesehen am 27.10.2017).

[Monu] Inc MongoDB. Text search languages. https://docs.mongodb.com/manual/reference/text-search-languages/#text-search-languages. (Eingesehen am 25.10.2017).

[Monv] Inc MongoDB. Versions. https://docs.mongodb.com/manual/reference/operator/query/text/. (Eingesehen am 16.02.2018).

[Monw] Inc MongoDB. Wildcard text indexes. https://docs.mongodb.com/manual/core/index-text/#wildcard-text-indexes. (Eingesehen am 25.10.2017).

[Mon13] Inc MongoDB. Mongodb 2.4 released. https://docs.mongodb.com/manual/release-notes/2.4/, 2013. (Eingesehen am 10.02.2018).

[Mon14] Inc MongoDB. Mongodb 2.6 released. https://docs.mongodb.com/manual/release-notes/2.6/, 2014. (Eingesehen am 10.02.2018).

[Mon15] Inc MongoDB. Mongodb 3.2 released. https://docs.mongodb.com/manual/release-notes/3.2/, 2015. (Eingesehen am 10.02.2018).

[Phr11] Definition phrase / phrasensuche. https://www.onlinemarketing-praxis.de/glossar/phrase-phrasensuche, 2011. (Eingesehen am 28.11.2017).

[Poi] Tutorials Point. Data structure - doubly linked list. https://www.tutorialspoint.com/data_structures_algorithms/doubly_linked_list_algorithm.htm.

[Pora] Martin Porter. Defining r1 and r2. http://snowball.tartarus.org/texts/r1r2.html. (Eingesehen am 18.11.2017).

[Porb] Martin Porter. German stemming algorithm. http://snowball.tartarus.org/algorithms/german/stemmer.html. (Eingesehen am 18.11.2017).

[Porc] Martin Porter. Snowball. http://snowballstem.org/. (Eingesehen am 18.11.2017).

[Por80] Martin Porter. The porter stemming algorithm. http://snowball.tartarus.org/algorithms/porter/stemmer.html, 1980. (Eingesehen am 18.11.2017).

[Por06] Martin Porter. The porter stemming algorithm. https://tartarus.org/martin/PorterStemmer/, 2006. (Eingesehen am 18.11.2017).

[Pre08] Cambridge University Press. Stemming and lemmatization. https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html, 2008. (Eingesehen am 15.02.2018).

[Sch13] Ludger Schmitz. Die zukunft von mysql gestalten wir. https://www.computerwoche.de/a/die-zukunft-von-mysql-gestalten-wir,2533344,2, 2013.

[See15] Yonik Seeley. Solr 5.1 features. http://yonik.com/solr-5-1/, 2015. (Eingesehen am 02.01.2018).

[Sta] Statista. Prognose zum volumen der jährlich generierten digitalen datenmenge weltweit in den jahren 2016 und 2025 (in zettabyte). https://de.statista.com/statistik/daten/studie/267974/umfrage/prognose-zum-weltweit-generierten-datenvolumen/.

[Swa17] Manohar Swamynathan. Mastering machine learning with python in six steps: A practical implementation guide to predictive data analytics using python. http://apprize.info/python/six/5.html, 2017.

[Tana] Kelvin Tan. Basic solr concepts. http://www.solrtutorial.com/basic-solr-concepts.html. (Eingesehen am 02.01.2018).

[Tanb] Kelvin Tan. Lucene query syntax. http://www.solrtutorial.com/solr-query-syntax.html. (Eingesehen am 02.01.2018).

[Trea] Tobias Trelle. Mongodb text search explained. https://blog.codecentric.de/en/2013/01/text-search-mongodb-stemming/. (Eingesehen am 30.10.2017).

[Treb] Google Trends. Vergleich von json-api's mit xml api's in google trends. https://trends.google.com/trends/explore?date=all&q=xml%20api,json%20api&hl=en-US.

[Trec] Google Trends. Vergleich von solr und elasticsearch in google trends. https://trends.google.com/trends/explore?date=all&q=solr,elasticsearch.

[Twi] Twitter. Twitter. https://blog.twitter.com/engineering/en_us/a/2011/twitter-search-is-now-3x-faster.html. (Eingesehen am 23.01.2018).

[W3S17] W3Schools. Ajax - the xmlhttprequest object. https://www.w3schools.com/xml/ajax_xmlhttprequest_create.asp, 2017. (Eingesehen am 08.07.2017).

[Wik] Ryte Wiki – Digitales Marketing Wiki. Case sensitivity. https://de.ryte.com/wiki/Case_sensitivity. (Eingesehen am 25.10.2017).

[Wina] Markus Winand. Der suchbaum (b-tree) macht den index schnell. http://use-the-index-luke.com/de/sql/anatomie/index-baum. (Eingesehen am 09.01.2018).

[Winb] Markus Winand. Die index-blätter: Eine doppelt verkettete liste. http://use-the-index-luke.com/de/sql/anatomie/index-blaetter. (Eingesehen am 09.01.2018).

[Winc] Wolfram Wingerath. Scalable push-based real-time queries on top of pull-based databases. Noch nicht veröffentlicht.

[Wind] Wolfram Wingerath. Twoogle. https://medium.baqend.com/going-real-time-has-just-become-easy-baqend-real-time-queries-hit-public-beta-3a44a13fde86.

[Win17] Wolfram Wingerath. Real-time databases explained: Why meteor, rethinkdb, parse & firebase don't scale. https://medium.baqend.com/real-

time-databases-explained-why-meteor-rethinkdb-parse-and-firebase-dont-scale-822ff87d2f87, 2017. (Eingesehen am 19.10.2017).

[WW17] Norbert Ritter Wolfram Wingerath, Felix Gessert. Invalidb - scalable query change notifications for pull-based databases. https://vsis-www.informatik.uni-hamburg.de/vsis/research/lookproject/62, 2017. (Eingesehen am 09.07.2017).

[WWR17] Steffen Friedrich Erik Witt2 Wolfram Wingerath, Felix Gessert and Norbert Ritter. The case for change notifications in pull-based databases *Datenbanksysteme für Business, Technologie und Web (BTW). Gesellschaft für Informatik*. https://dl.gi.de/bitstream/handle/20.500.12116/923/paper32.pdf?sequence=1, 2017.

[Yah07] Yahoo. Using json (javascript object notation) with yahoo! web services. https://web.archive.org/web/20071011085815/http://developer.yahoo.com/common/json.html, 2007. (Eingesehen am 10.02.2018).

Abbildungsverzeichnis