Hochschule München
Hochschule für angewandte Wissenschaften
Fakultät für Geoinformation

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN MÜNCHEN

Bachelorarbeit

zur Erlangung des akademischen Grades *Bachelor of Engineering (B.Eng.)* im Studiengang Geoinformatik und Satellitenpositionierung

Volltextsuche mit Elasticsearch im Geodaten-Umfeld

vorgelegt von

Mosaab Asli
Matrikelnummer: 61825514

eingereicht am 09. April 2018

Betreuer:
Prof. Dr. Gerhard Joos
Dipl.-Ing. Florian Mückl

Inhaltsverzeichnis

Listingverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AdV	Arbeitsgemeinschaft der Vermessungsverwaltungen der Länder der Bundesrepublik Deutschland
API	Application Programming Interface (Programmierschnittstelle)
BeTA2007	Bundeseinheitliche Transformation für ATKIS
CLI	Command Line Arguments Parser
CODASYL	Conference on Data Systems Language
cURL	Client for URLs
EWKT	Extended Well-Known Text
FTP	File Transfer Protocol (Dateiübertragungsprotokoll)
GUI	Graphical User Interface (Grafische Benutzeroberfläche)
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment (Integrierte Entwicklungsumgebung)
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LDBV	Landesamt für Digitalisierung, Breitband und Vermessung
NDJSON	Newline Delimited JSON
NoSQL	Not only SQL (Nicht nur SQL)
NTv2	National Transformation Version 2
OS	Operating Systems (Betriebssystem)
POI	Point of Interest (interessanter Ort)
RDBMS	Relational Database Management System (Relationales Datenbankmanagementsystem)
RESTful	Representational State Transfer
SQL	Structured Query Language (Strukturierte Abfrage-Sprache)
SRID	Spatial Reference Identifier (Referenz-Identifikationnummer für räumliche Koordinatensysteme)
URL	Uniform Resource Locator
UTM32	Universal Transverse Mercator
WGS84	World Geodetic System 1984
ZAD	Zentraler Adressendienst

Vorwort

Während meines praktischen Semesters, welches ich am Landesamt für Digitalisierung, Breitband und Vermessung in München absolvierte, beschäftigte ich mich mit dem Generieren von Vektor-*Tiles* aus Raster-*Tiles* für den *BayernAtlas*. Im Zuge dieser Arbeit entstand das Thema der vorliegenden Bachelorarbeit.

Mein besonderer Dank für die Betreuung dieser Bachelorarbeit gilt daher Dipl.-Ing Florian Mückl vom LDBV für dessen engagierte Unterstützung. Er stand von Anfang an voll und ganz hinter meiner Arbeit und zeigte mir stets mit wertvollen Ratschlägen den richtigen Weg. Darüber hinaus möchte ich Ihm für die freundliche und kompetente Betreuung während meines Praktikums danken, welches mir viele neue Erfahrungen ermöglicht hat. Ganz herzlich danken möchte ich ebenfalls Professor Dr. Gerhard Joos von der Hochschule München für seine Unterstützung und die nützlichen Hinweise bei der Anfertigung dieser Arbeit.

Danke auch an die Open-Source-Community und die Entwickler der vielen nützlichen Bibliotheken und Tools, die für diese Forschung verwendet wurden.

Abstrakt

Mit dem Eintritt in die Ära des *Cloud-Computing* und *Internet-of-Things* werden immer öfters und immer größere Datenmengen ins Internet hochgeladen, die gespeichert und analysiert werden müssen. In Folge dieser komplexen Datensätze haben *NoSQL*-Datenbanken enorm an Popularität gewonnen, da diese im Vergleich zu relationalen Datenbanken besser mit solch großen Datenmengen umgehen können. Insbesondere dann, wenn es sich um gemischte Datentypen handelt, da *NoSQL*-Datenbanken je nach Datenmodell einfach nur unterschiedlich implementiert werden.

Diese Arbeit befasst sich mit der Übertragung von komplexen und teils unstrukturierten Datensätzen aus der relationalen *PostrgreSQL*- in die *NoSQL*-Datenbank *Elasticsearch* und vergleicht deren Leistung und Funktionsumfang. Dabei wird neben dem allgemeinen Teil zu Aufbau und der Funktionalität von *Elasticsearch* auch das Thema „Implementierung“ und „Anwendbarkeit“ in Bezug auf die Adressen-Suchfunktion des LDBVs behandelt.

1 Einleitung

1.1 Problemstellung

Laut dem amerikanischen Telekommunikationsunternehmen *Cisco* wird der globale Internetverkehr im Jahr 2021 auf ca. 105.800 GB pro Sekunde steigen.[1] Aber schon jetzt gilt es, sich der Herausforderung „Big Data“ zu stellen – d.h. Datensätzen, die mit der konventionellen IT nicht mehr effektiv bearbeitet werden können, da sie zu komplex sind.

Eine wesentliche Funktion von fast allen Webseiten heute ist die Volltextsuche. Besonders bei großen Webseiten gilt es dabei, in komplexen Datenmengen blitzschnell die gesuchten Informationen zu finden. Hierfür wird eine leistungsfähige Programmbibliothek zur Volltextsuche benötigt, wofür wiederum eine produktive Volltextindizierung nötig ist.

Das LDBV mit Sitz in der Landeshauptstadt München fungiert nicht nur als Aufsichtsbehörde für die 51 Ämter für Digitalisierung, Breitband und Vermessung. Es ist zugleich für das Breitbandzentrum Amberg und das IT-Dienstleistungszentrum des Freistaats Bayern verantwortlich und somit auch zuständig für die flächendeckende Bereitstellung von Geobasisdaten.[2] Insbesondere beim Kartenviewer *BayernAtlas*, der verschiedenste Themenkarten sowie Luftbilder und historische Karten zur Verfügung stellt, ist es also entscheidend, wie diese riesigen Datenmengen gespeichert und ausgewertet werden. Bis vor ca. zwei Jahren befanden sich die Daten zum *BayernAtlas* in einer *PostgreSQL*-Datenbank. Diese war für eine effiziente und schnelle Suche nicht geeignet.

Elasticsearch ist hauptsächlich für die Suche nach Daten konzipiert. Dies hat den Vorteil, dass ein Großteil der für ein RDBMS benötigten Funktionen verworfen werden können. *Elasticsearch* beschränkt sich also auf reine Indexdaten, was im Umfeld von *Big Data* ein wichtiger Faktor ist.

Elasticsearch ist somit die Antwort auf die Frage, wie man solche großen Datenmengen, wie sie bei sozialen Medien oder eben auch in der Vermessungsverwaltung entstehen, effizient speichern, verarbeiten und auswerten kann und wie man trotz teils unstrukturierter Daten schnell zu den gesuchten Informationen gelangt.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, *Java*-Applikationen zu programmieren, mit denen folgende Aufgaben erfüllt werden:

- die Adressen-Datenbank (in Form einer *PostgreSQL*-Tabelle) in eine *Elasticsearch*-Datenbank (in der aktuellen Version) zu exportieren
- eine schnelle Adressen-Suchfunktion zu implementieren, mithilfe derer die Suchanfragen und Rückgaben optimiert werden können

1.3 Aufbau der Arbeit

Zur Einführung in die Thematik soll ein kurzer geschichtlicher Abriss zur Entwicklung der Datenbankensysteme und -sprachen dienen (Kapitel 2). Anschließend wird in einem Grundlagen-Kapitel die Architektur und der Funktionsumfang von *Elasticsearch* in der aktuell neuesten Version 6.2.2. beschreiben. Im nachfolgenden Abschnitt (Kapitel 4) soll ein Vergleich verschiedener Datenbankmanagementsysteme die hohe Effizienz und Leistungsfähigkeit von *Elasticsearch* – insbesondere gegenüber *PostgreSQL* – zeigen. Kapitel 5 und 6 befassen sich schließlich mit der Programmierung und Implementierung sowie der REST-Fassade. Anfangs wird die Arbeitsumgebung vorgestellt. Um eine Übersicht zu erlangen, folgt anschließend eine stichpunktartige Auflistung der einzelnen Arbeitsschritte, die nötig sind, um die *PostgreSQL*-Daten auf *Elasticsearch*-Server zu speichern. Danach werden die diese Punkte noch einmal ausführlich erläutert. Abschließend wird der Ablauf bei der Optimierung der Suchergebnisse vorgestellt.

2 Entwicklung der Datenbanksysteme

Heutzutage sind wir in der Lage große Datenmengen zu speichern, abzurufen und zu analysieren. Dabei ist es hilfreich, den Anfang dieser Entwicklung zu kennen. Schon bei den ersten Computerprogrammen in den frühen 1950er Jahren haben Datenbanken eine sehr wichtige Rolle gespielt. Sie beschränkten sich zu diesem Zeitpunkt fast vollständig auf Programmiersprachen und Algorithmen. Als die Computer zunehmend auch für private Zwecke genutzt wurden, standen die Präsentation und Verarbeitung der Daten plötzlich im Vordergrund. 1960 entwarf *Charles W. Bachman* ein integriertes Datenbanksystem. Wenig später hat das Unternehmen IBM ein eigenes Datenbanksystem geschaffen. Beide werden heute als Vorläufer von Navigationsdatenbanken gesehen.

Im Jahr 1971 veröffentlichte die *Database-Task-Group* von CODASYL das Netzwerkdatenbank-modell, welches auch als *CODASYL-Ansatz* oder *CODASYL-Datenbankmodell* bekannt ist und auf der vom eigenen Komitee entwickelten Programmiersprache *COBOL* basiert. Zwei Jahre später haben Michael Stonebraker und Eugene Wong erfolgreich ein relationales Modell demonstriert, das effizient und praktisch war. Es trägt den Namen *INGRES* und arbeitete anfangs mit einer Abfragesprache namens *QUEL*. Diese wurde 1974 durch die von *IBM* entwickelte Datenbank-sprache *SQL* ersetzt.

RDBMS sind eine effiziente Möglichkeit, strukturierte Daten zu speichern und zu verarbeiten. In den letzten Jahren nahm jedoch die Menge an unstrukturierten Daten stark zu, wofür die RDBMS nicht mehr geeignet sind. Als Reaktion darauf hatte *Google* zu Beginn des Jahrtausends mit einem internen Team erfolgreich eine neue Art von proprietärer Datenbank erstellt und im Jahr 2004 zwei Forschungspapiere zu diesem Projekt veröffentlicht. Diese Arbeit fand ihren Weg in die Hände von Doug Cutting und Mike Cafarella, zwei unabhängige Entwickler, die *Yahoo* davon überzeugten, dass diese neue Struktur die Lösung für ihre Such- und Indizierungsaufgaben war. Im Jahr 2006 veröffentlichte *Yahoo* den ersten Prototypen namens *Hadoop* und gab 2008 erstmals seine kommerzielle Implementierung bekannt.

Ab 2008 wurde die allgemeine Terminologie *NoSQL* eingeführt, um diese neuen Datenbanken von ihren RDBMS-Vorgängern zu unterscheiden.[3][4]

Lucene ist eine in Java geschriebene Programmbibliothek zur Volltextsuche. Die erste Version wurde 1999 entwickelt und aktuell gibt es sie in der Version 7.2.1. Da es sich bei *Lucene* lediglich um eine Programmbibliothek handelt, gibt es kein User-Interface. *Lucene* ist die Grundlage einiger weiterer Programmbibliotheken, wie z.B. *Solr* und *Elasticsearch.* Letzteres, welches Thema dieser Arbeit ist, wurde 2010 entwickelt und wuchs binnen kurzer Zeit zu einem ernstzunehmenden Konkurrenten von *Solr* heran. Es gehört heute zu den führenden Open-Source-Suchmaschinen. und wird von der *Apache Software Foundation* geleitet. Die aktuelle Version ist 6.2.2. Die beiden eben genannten Programmbibliotheken speichern ihre Suchergebnisse in einem *NoSQL*-Format. Vorteil von *Elasticsearch* gegenüber *Solr* ist die einfache Skalierbarkeit.[5][6]

3 Elasticsearch

3.1 Hauptmerkmale von Elasticsearch

Um einen Überblick über die Funktionsweise und den Funktionsumfang von *Elasticsearch* zu bekommen, werden nachfolgend die wichtigsten Punkte aufgezählt und beschrieben.[7][8]

- **Dokumentenorientiert und schemafrei:** *Elasticsearch* kann ganze Objekte und komplexe Dokumente speichern und deren Inhalt indexieren. Damit ist es nicht mehr nötig, die Daten vorher aufwendig für ein bestimmtes Schema zu bearbeiten und aufzubereiten oder nach einer Abfrage wiederherzustellen.
- **Volltextsuche**: Aufgrund der dokumentenorientierten und schemafreien Arbeitsweise von *Elasticsearch* ist eine komplexe Volltextsuche möglich – und dies nahezu in Echtzeit.
- **Dynamisches Mapping:** Standardmäßig verwendet *Elasticsearch* das sogenannte dynamische *Mapping*. Das bedeutet, dass beim Hinzufügen neuer Dokumente das Indexieren automatisch erfolgt. Dabei ermittelt *Elasticsearch* auch automatisch den Datentyp, sollte es sich nicht um ein JSON-Format handeln und die Datenstruktur nicht explizit angegeben worden sein. Möchte man dies vermeiden, kann man das dynamische Mapping auch herausnehmen oder anpassen (z.B. Meldung beim Anlegen von neuen Feldern). So können Datenverluste vermieden und die Wahl der Datentypen kontrolliert werden.[9]
- **Horizontale Skalierung:** *Elasticsearch* kann ein System horizontal skalieren, d.h. auf verschiedene Server verteilen. Damit wird die Last einzelner Server verringert und deren Leistung verbessert (z.B. schnellere Abfragen).[10]
- **Verfügbarkeit:** *Elasticsearch* ermöglicht es, von allen Daten Kopien (*Replica*) anzufertigen, welche auf anderen Servern gespeichert werden können. Damit ist auch bei einem Komplett-Ausfall des primären Servers die Verfügbarkeit der Daten gewährleistet.
- **Persistenz:** *Elasticsearch* zeichnet alle Änderungen, die im System vorgenommen werden, in einem sogenannten *Translog* oder Transaktionsprotokoll auf. Das *Translog* enthält alle getätigten Befehle, die ausgeführt aber noch nicht auf die Festplatte geschrieben wurden. Damit kann die Gefahr von Datenverlust durch Systemausfälle zusätzlich minimiert werden. [11]
- **Konfigurierbar und erweiterbar:** Konfigurationen sind beim Erstellen und auch teilweise noch während der Ausführung möglich. Außerdem gibt es mehrere Erweiterungen (*Plugins*), um die Funktionalität von *Elasticserach* noch zu verbessern.[12]
- **RESTful-API:** Über diese leistungsstarke Programmierschnittstelle ist die Programmbibliothek *Lucene*, auf der *Elasticsearch* aufbaut, leicht per JSON und HTTP/S zugänglich. Damit können nicht nur Daten angelegt, bearbeitet durchsucht oder gelöscht werden; es ist auch möglich das System an sich zu prüfen, zu verwalten und Statistiken abzurufen.

3.2 Grundlegendes Konzept

Um die Arbeitsweise von Elasticsearch nachvollziehen zu können, ist es hilfreich, sich dessen Aufbau genauer anzusehen. Grundsätzlich muss man zwischen einem logischen und einem physischen Teil unterscheiden.[13][14][15]

3.2.1 Logisches Layout

Das logische Layout von *Elasticsearch* besteht im Prinzip aus zwei Einheiten: dem Index und den Dokumenten.

Ein **Index** besteht jeweils aus einem Dokument. Er ist der logische Ort, an dem *Elasticsearch* logische Daten zum Dokument speichert. Der Name des Indexes muss eindeutig sein und darf nur aus Kleinbuchstaben bestehen. Bis Version 6.0.0 war es möglich, viele Dokumente mit unterschiedlichen Datentypen innerhalb eines Indexes zu speichern. Nun wird für jedes Dokument ein eigener Index angelegt. Dadurch wird die Speicherstruktur verbessert und die Volltextsuche optimiert.

Ein **Dokument** ist die grundlegende Informationseinheit, die einem Index zugeordnet ist. Die Speicherung erfolgt im JSON-Format. Ein Dokument besteht aus Feldern, welche durch ihre Namen gekennzeichnet sind und einen oder mehrere Werte enthalten können. Jedes Feld hat einen bestimmten Datentyp. *Elasticsearch* unterstützt dabei eine Reihe verschiedener Datentypen: neben den sogenannten Core Datatypes (string, numeric, date, boolean, binary, und range) gibt es auch komplexe Datentypen (array, object), Geo-Datentypen (geo point, geo shape) und sogenannte spezialisierte Datentypen, mit Hilfe derer verschiedene Operationen wie Analyse, Hervorhebung oder Sortierung durchgeführt werden können. So ist es mit dem Datentyp *Completion Suggester* beispielsweise möglich, eine automatische Vervollständigung bzw. eine Suche während der Suchbegriff-Eingabe durchzuführen.[16]

3.2.2 Physisches Layout

Hierunter versteht man die physische Speicherung der Daten. Grundsätzlich werden hier drei Einheiten unterschieden, nämlich *Shard*, *Node* und *Cluster*.

Elasticsearch kann aus vielen Indizes bestehen. Ein Index kann sich entweder auf einem Rechner befinden oder aber – wenn es bei großen Datenmengen die Hardwaregrenzen eines einzelnen Rechners übersteigt – auf mehrere Server verteilt werden. Wird ein Index geteilt, spricht man vom sogenannten *Sharding*. Ein Index besteht grundsätzlich immer aus mindestens einem ***Shard***. Ein *Shard* wiederum kann aus beliebig vielen *Replicas* bestehen. Dabei handelt es sich um zusätzliche Kopien, die wie die primären *Shards* verwendet werden können. Sie stellen sicher, dass beispielsweise bei Überlastung einzelner Server die Serveleistung nicht unterbrochen wird. Die Anzahl der *Shards* und *Replicas* wird beim Erstellen eines Index festgelegt. Während man bei *Replicas* die Konfiguration nachträglich verändern kann, ist die Definition von *Shards* nur einmal beim Erstellen möglich.

Cluster ist die Bezeichnung für eine Gruppe von Servern, die zusammenarbeiten. Er bietet für diese Indizierungs- und Suchfunktionen und ist durch einen eindeutigen Namen gekennzeichnet.

Unter ***Node*** versteht man einen einzelnen Server, also eine Instanz des *Elasticsearch*-Server, auf dem Daten gespeichert sind. Für einfache Anwendungsfälle kann bereits ein einzelner *Node* ausreichend sein. Gibt es mehrere *Nodes* innerhalb eines *Clusters*, so verfügen alle über Informationen zu den übrigen *Nodes* und können *Client*-Anforderungen untereinander versenden. Die Identifikation erfolgt über einen sogenannten *Universally-Unique-Identifier* (UUID)[17], einer zufälligen 16-Byte-Zahl. Je nach Zweck werden die *Nodes* in eine der nachfolgenden Kategorien eingeteilt:[18]

- **Master-Node:** Jeder *Cluster* ist mit einem einzelnen *Master-Node* verbunden, der automatisch ausgewählt wird. Dieser *Master-Node* steuert den *Cluster*.
- **Data-Node:** Dieser *Node* enthält Daten und führt datenbezogene Operationen aus (Erstellen, Lesen, Aktualisieren, Löschen).

3.3 Indexstruktur

Zum besseren Verständnis sollen zunächst noch ein paar Begrifflichkeiten geklärt werden.[19]

- **Index:** Wie bereits erwähnt, entspricht ein Index einer Sammlung an Daten.
- **Indexieren:** Wird ein Dokument indexiert, bedeutet dies, dass man es in einem Index speichert um es später abzurufen. Es ist mit dem *SQL*-Befehl „*INSERT*“ vergleichbar.
- **Invertierter Index:** *Elasticsearch* (wie auch *Lucene)* verwendet eine Struktur, die als invertierter Index bezeichnet wird und eine sehr schnelle Volltextsuche ermöglicht. Unter einem invertierten Index versteht man eine Liste, die jedes einzelne Wort bzw. jeden einzelnen Term (im Englischen auch *Token* genannt) jeglicher Dokumente enthält sowie zu jedem dieser Worte eine Liste der Dokumente, in der sie vorkommen.[20]

3.3.1 Index Settings

Wird ein Index neu angelegt, so gibt es drei wichtige Punkte, die man einstellen kann: die Zahl der *Shards* und *Replicas* sowie den Punkt *Analysis* (vgl. nachfolgenden Listing-Ausschnitt).[21]

```
"settings": { "number_of_shards": "Any_Number", "number_of_replicas": "Any_Number", "analysis": {} }
```

Listing 1: Elasticsearch Index Settings

Die Anzahl der *Shards* (im Englischen *number of shards*) ist standardmäßig auf 5 eingestellt. Möchte man dies ändern, so muss man die Einstellung beim Erstellen des Indexes vornehmen. Im Nachhinein kann sie nicht mehr geändert werden.

Die Anzahl der *Replicas* (im Englischen *number of replicas*) hat als Standardwert 1, was bedeutet, dass für jede einzelne *Shard* eine Kopie angelegt wird. Im Gegensatz zu den primären *Shards* kann die Anzahl auch später noch verändert werden.[22]

Die Einstellung bei *Analysis* legt fest, nach welchem Schema die Indizierung neuer Dokumente erfolgt. Der Analyseprozess erfolgt mittels sogenannter *Built-in* oder durch *Custom Analyzers*. Die *Built-in Analyzers* können bei *Elasticsearch* direkt ausgewählt werden. Der *Standard Analyzer* ist hier voreingestellt. Daneben gibt es noch den *Simple* und den *Whitespace Analyzer* sowie *Language Analyzers*. Alle teilen den Dokumenteninhalt nach unterschiedlichen Kriterien in einzelne Terme bzw. Tokens auf. Mithilfe der *Custom Analyzers* können die Trennung und Filterung der Terme/*Tokens* (wie sie nachfolgend beschrieben werden) nach eigenen Vorgaben festgelegt werden [23].

Grundsätzlich läuft der Analyseprozess bei *Analyzers* immer nach dem gleichen Schema ab [24] [25]:

- ***Tokenizer*** (zum Aufteilen des Texts/Dokuments in einzelne Terme/*Tokens* – nach welchen Kriterien ist von der Wahl des *Analyzer* abhängig)

  Beispiel der Zerlegung eines Satzes in einzelne Begriffe mit dem *Whitespace Analyzer (*er trennt nach Leer- und Satzzeichen) **:**

```
Satz:          My First Name Is „Mosaab“!, My Family Name Is „Asli“!
Begriffe:      My, First, Name, Is, Mosaab, Family, Asli
```

Listing 2: Tokenizer

- ***Character Filters*** (Hinzufügen, Ersetzen/Ändern oder Entfernen einzelner Zeichen oder bestimmte Zeichenfolgen, um einen konsistenten Zeichensatz sicherzustellen).

  Ein HTML-Strip-Char-Filter entfernt beispielsweise HTML-Elemente aus einer Zeichenfolge oder ersetzt HTML-Entitäten durch ihren dekodierten Wert :

HTML-Strip:	** **I'm learning ****Elasticsearch****!** **
Bearbeitungsschritte:	**'** wird zu Apostroph (').
	** ** wird zu Leerzeichen.
	** ** und **** werden gelöscht.
Ergebnis:	I'm learning Elasticsearch!

Listing 3: HTML-Strip-Char-Filter

- ***Token-Filter*** (hier kann mit optionalen Filtern jeder Term nochmals gefiltert/genormt werden – so ist es abhängig vom verwendeten *Analyzer* zum Beispiel möglich, alles in Kleinschreibung umzuwandeln , unnötige „Stoppwörter“ (wie „im“, „ein“ oder „der“) bei der Indexierung zu ignorieren, Umlaute zu ersetzen oder Synonyme hinzuzufügen)[26][27].

Nachfolgendes Beispiel (vgl. Abbildung 1) zeigt, wie ein benutzerdefinierter *Analyzer* aussehen kann. Die einzelnen Komponenten (*Tokenizer, Char-Filter* und *Token-Filter*) können zu einem Analyseprozess zusammengesetzt werden. Der *Tokenizer* trennt zunächst das Dokument in einzelne Terme, welche vom *Character-Filter* bearbeitet werden. Dieser filtert die HTML-Tags, und ersetzt eingebettete Bindestriche in der Zahlenreihe durch Unterstriche. Der *ASCII-Folding-Token-Fi*lter wandelt anschließend alles in Kleinbuchstaben um und konvertiert die alphabetischen, numerischen und symbolischen Unicode-Zeichen entsprechend in ASCII-Zeichen (z.B. Umlaute).

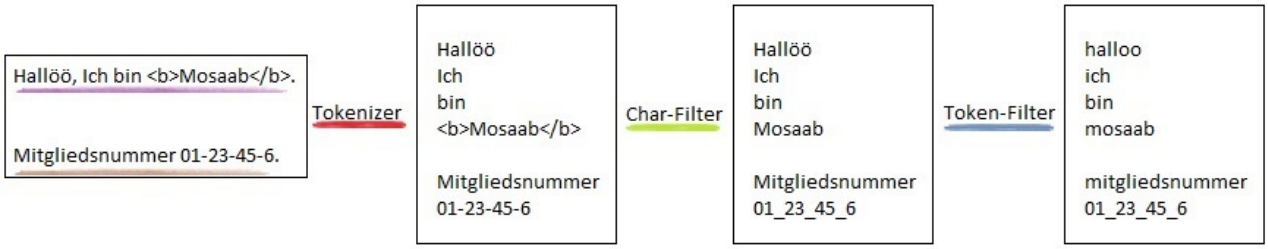

Abbildung 1: Benutzerdefinierter Analyzer aus Tokenizer, Char-Filter und Token-Filter

3.3.2 Mapping Type

Seit Version 6.0.0 besitzt jeder Index nur noch einen *Mapping Type*[28]. Hier ist festgelegt, wie ein Dokument und die darin enthaltenen Felder gespeichert und indiziert werden. Ein *Mapping Type* besteht aus *Meta-Fields* (beinhalten die Metadaten zum Dokument) und *Fields* (beinhalten die Daten). Dank des dynamischen *Mappings* ist es jedoch nicht zwingend erforderlich, Felder oder *Mapping Types* vorher zu definieren. Da *Elasticsearch* in diesem Fall den Datentyp quasi nur rät, empfiehlt es sich aber, die *Mapping*-Regeln anzupassen, wenn man die Daten eines Dokuments kennt. So können Fehler vermieden werden. In diesem Fall spricht man dann von *Explicit Mapping.*

Nach dem Indexieren kann auch ein neues Mapping hinzugefügt werden. Dann versucht Elasticsearch beide zusammenzuführen. Sollte dies nicht gelingen, wird eine Fehlermeldung (*Merge Mapping Exception*) ausgegeben. Dies geschieht beispielsweise, wenn der Datentyp nachträglich geändert werden soll, da dies nicht möglich ist. Die einzige Möglichkeit, diesen Fehler zu umgehen, besteht darin, die Daten zu entfernen, ein neues Mapping einzufügen und anschließend die Daten neu zu indexieren.[29][30]

3.3.3 Aliases

Index *Aliases* sind Pseudonyme oder Ersatznamen für Indizes. Ein Alias kann auf einen oder mehrere Indizes zeigen, je nachdem wie es beim Anlegen eines Indexes definiert wird. Ein Index Alias kann auch nachträglich noch gelöscht und neu angelegt werden. Wird eine Suchoperation für einen Alias ausgeführt, wird diese für alle Indizes, auf die der Alias verweist, ausgeführt. Es wird daher empfohlen, einen Alias anstelle des tatsächlichen Indexnamens in der Anwendung zu gebrauchen. Ein praktisches Beispiel zur Verdeutlichung: bei einer Java-API will man mit einem festen Namen arbeiten und dieser Name soll – falls es eine Aktualisierung gibt – in der Anwendung

nicht immer geändert werden müssen. Indem man hier einen Alias-Namen in der Anwendung benützt, kann man den Index leicht ändern/austauschen, denn der Alias kann auch auf den neuen Index zeigen (vgl. nachfolgendes Listing).[31]

```
POST /_aliases
{
    "actions" : [
        { "remove" : { "index" : "adresse_20180316_1123", "alias" : "adresse" } },
        { "add" : { "index" : "adresse_20180319_1003", "alias" : "adresse" } }
    ]
}
```

Listing 4: Index Aliases

3.4 Arbeitsmechanismus

Um zu verstehen, wie *Elasticsearch* beim Anlegen eines neuen Systems vorgeht, wird hier ein kleines Beispiel angeführt. Die Abbildungen sollen zum besseren Verständnis beitragen.[32]

Wenn ein neuer *Node* ohne Daten gestartet wird, wird ein neuer Cluster mit diesem *Node* als *Master-Node* erstellt (vgl. Abbildung 2). Wie bereits erwähnt, wird der *Master-Node* automatisch ausgewählt – da es in diesem Beispiel nur einen *Node* gibt, wird dieser zwangsläufig auch ein *Master-Node*. Er ist für die Verwaltung von Änderungen im gesamten Cluster verantwortlich.[33]

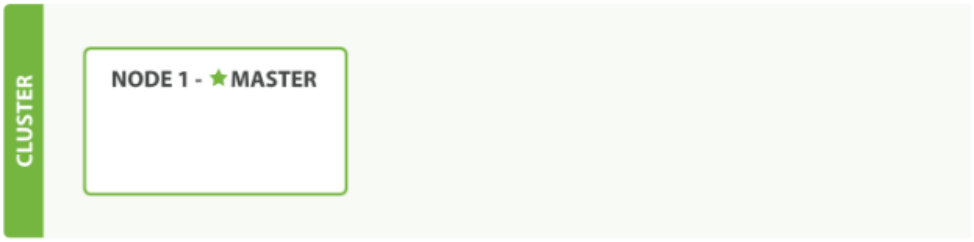

Abbildung 2: Neuer Cluster mit einem leeren Node

Beim Erstellen eines Indexes und dem Einfügen von Daten wird eine genaue Anzahl von primären *Shards* und *Replicas* erstellt, die in den Einstellungen angegeben werden muss. In diesem Beispiel wird ein Index mit drei primären *Shards* sowie eine komplette *Replica* davon, d.h. drei *Replica-Shards*, erstellt (vgl. nachfolgendes Listing).

```
"settings" : { "number_of_shards" : 3, "number_of_replicas" : 1 }
```

Listing 5: Einstellung von Shards und Replicas

Da es keinen Sinn macht, die *Replica-Shards* auf dem gleichen *Node* zu speichern, werden zunächst nur die primären *Shards* erstellt (vgl. Abbildung 3).

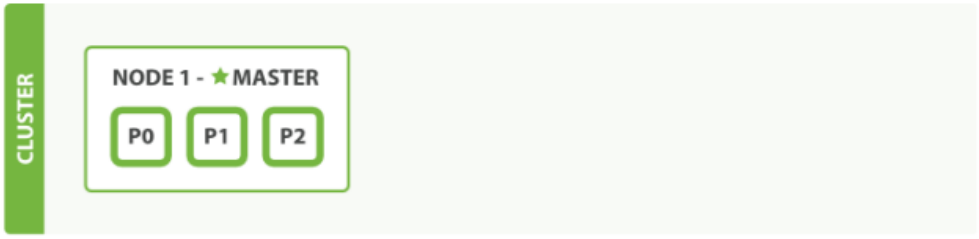

Abbildung 3: Cluster mit einem einzelnen Node

Wenn ein neuer *Node* hinzugefügt wird, werden die *Replica-Shards* automatisch darauf erstellt (vgl. Abbildung 4). Sollte es nun zu einem Ausfall von *Node* 1 kommen, würde *Node* 2 die Aufgabe von *Node* 1 übernehmen und damit einen Komplett-Ausfall der Datenbank verhindern.

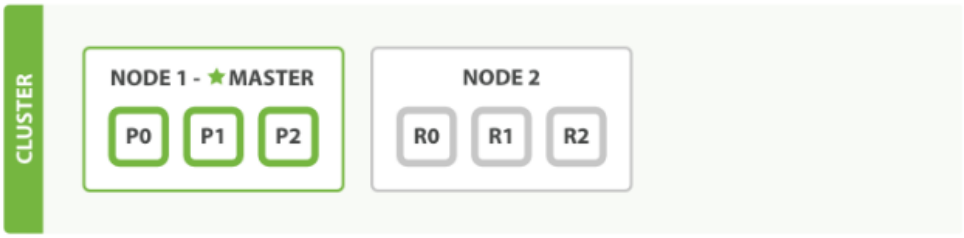

Abbildung 4: Cluster mit zwei Nodes - alle primären und Replica-Shards werden zugewiesen

Wenn weitere *Nodes* hinzugefügt werden, werden die *Shards* und *Replicas* unter den vorhandenen *Nodes* aufgeteilt und neu zugewiesen (vgl. Abbildung 5). So kann die Leistung einzelner Server bedingt durch die geringere Belastung verbessert werden.

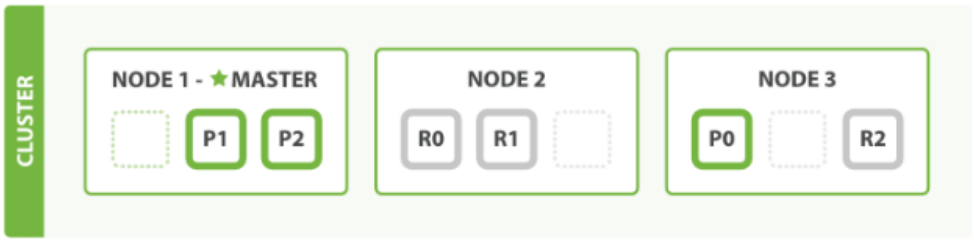

Abbildung 5: Cluster mit drei Nodes, drei primäre Shards und drei Replicas

Für den Fall, dass ein *Node* (im Beispiel *Node* 1) komplett ausfällt, kann *Elasticsearch* die Verteilung dynamisch anpassen ohne dass Daten verloren gehen. So werden die *Shards* vom defekten *Node* auf die anderen *Nodes* im *Cluster* verteilt und *Node* 2 automatisch zum *Master-Node* gemacht (vgl. Abbildung 6).

Abbildung 6: Cluster nach Ausfall eines Nodes

3.5 Suche

Standardmäßig ist jedes Feld in einem Dokument indiziert (besitzt somit einen invertierten Index) und kann durchsucht werden. Ein Feld ohne invertierten Index kann nicht durchsucht werden.

Wie bereits erwähnt, besteht ein invertierter Index aus einer Liste einzelner Terme, die in allen Dokumenten vorkommen, sowie einer Liste zu jedem Term mit den Dokumenten, in dem sie vorkommen. Terme bzw. *Tokens* sind Worte (*strings)* oder Textbestandteile, die in ihrer genauen Form oder in einer Stammform gespeichert werden. Es können auch Synonyme des entsprechenden Terms sein. Nachfolgende Abbildung zeigt, wie ein invertierter Index für drei einfache Dokumente (hier drei Sätze) aussehen kann. Die Terme in dieser Liste werden sortiert, sodass man schnell einen Term finden kann und damit auch sein Vorkommen im jeweiligen Dokument.[34][35][36][37]

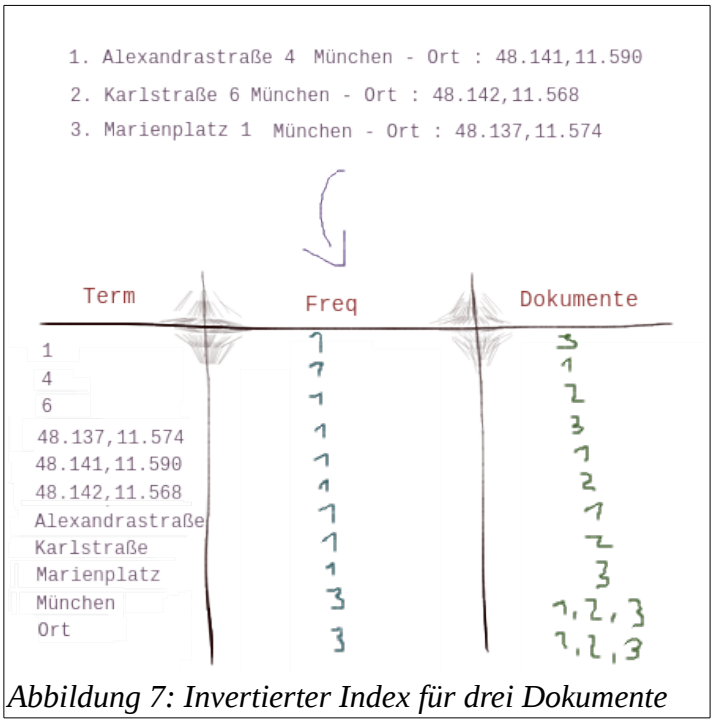

Abbildung 7: Invertierter Index für drei Dokumente

Die Wahl des *Analyzers* bei den Index-Einstellungen bestimmt, was genau im invertierten Index enthalten ist, wenn ein neues Dokument hinzugefügt wird. Der mehrstufige Prozess, bestehend aus *Tokenizer*, *Charakter Filters* und *Token Filters*, wurde in Kapitel 3.3.1 bereits erläutert. Nach dem gleichen Schema wird auch bei Eingabe des Suchbegriffs vorgegangen, damit die Daten auch entsprechend wiedergefunden werden können. Dies geschieht entweder ebenso wie beim Einfügen des Dokuments oder in etwas modifizierter Form.

Elasticsearch bietet eine vollständige *Query DSL* basierend auf JSON zum Definieren von Abfragen. Sie unterstützt neben der Volltextsuche auch die sogenannte Term-Abfrage. Letztere ist wichtig, da es für bestimmte Felder im invertierten Index notwendig sein kann, den Inhalt ohne Analyse unverändert abzuspeichern (wie z.B. ein genaues Datum oder die exakte Zeichenfolge bei einem Benutzernamen oder einer Emailadresse). So ist beispielsweise der exakte Wert „*Admin*“ ist nicht identisch mit „*admin*“ oder der genaue Wert „01_01_*2018*“ ungleich dem Wert „01-01-2018“.

Elasticsearch-Abfragen werden mithilfe der Such-API ausgeführt. Solche Abfragen und ihre Antworten werden im JSON-Format dargestellt. Das Verhalten einer Abfrageklausel hängt davon ab, ob sie im Abfragekontext oder im Filterkontext verwendet wird [38].

- **Abfragekontext (Query):** Diese Query beantwortet die Frage „Wie gut passt dieses Dokument zu dieser Abfrage?“.Dies geschieht mithilfe einer Punktzahl (*score*), die angibt, wie gut das Dokument im Vergleich zu anderen Dokumenten mit den Suchbegriffen übereinstimmt.
- **Filterkontext:** Dieser Filter beantwortet die Frage "Passt dieses Dokument zu dieser Abfrage?". Die Antwort ist ein einfaches Ja oder Nein. Es wird also keine Punktzahl berechnet, was die Leistung verbessert. Daher wird dieser Filter auch häufig verwendet.

3.5.1 Match_All-Abfrage

Hierbei handelt es sich um die einfachste aller Abfragen. Sie gleicht alle Dokumente ab und gibt jedem eine konstante Punktzahl von 1.0. Damit kann man schnell einen Blick auf den Inhalt der Daten werfen. Die Relevanz wird jedoch nicht beachtet. Nachfolgend ist das Listing zu einer *Match_All*-Abfrage dargestellt.[39]

```
GET /_search
{
 "query": {
  "match_all": {}
 }
}
```

Listing 6: Match_All-Abfrage

3.5.2 Volltextsuche

Die Volltextsuche zielt darauf ab, Dokumenteninhalte zu finden, die der Suchanfrage möglichst gut entsprechen, und somit die relevantesten Dokumente auszugeben. Es gibt zwei wichtige Aspekte der Volltextsuche, nämlich die Relevanz und die Analyse.

Es besteht die Möglichkeit, die Ergebnisse der Suchabfrage nach ihrer berechneten **Relevanz** (*Score*) zu ordnen. Die zuvor erwähnte Analyse ist der Prozess der Umwandlung eines Textblocks in unterschiedliche normalisierte *Token*s oder Terme. Bei der Volltextsuche werden zwei verschiedene Arten von Abfragen (*Queries*) unterschieden, nämlich termbasierte *Queries* und echte Volltextsuchen, je nachdem ob und wie eine Analyse stattfindet.[40][41][42][43]

Termbasierte *Queries* sind sogenannte *Low-Level*-Abfragen, die keine Analysephase haben. Sie analysieren also keine Suchanfrage, sondern suchen die invertierte Datei nach dem exakten Suchbegriff (*Exact Value*). So würde für den Begriff "LDBV München" im invertierten Index nach dem genauen Begriff gesucht und die Relevanz für jedes Dokument berechnet werden, das den Begriff enthält (vgl. nachfolgendes Listing). Sollte im invertierten Index die Begriffe „ldbv“ oder „ldbv München“ stehen, würde diese Abfrage kein Ergebnis zurückgeben.

```
POST _search
{
  "query": {
    "term": {
      "firma": "LDBV München"
    }
  }
}
```

Listing 7: Termbasierte Query

Die **Volltextsuche** ist eine High-Level-Abfrage, die das *Mapping* eines Feldes gut versteht. Wenn also zum Beispiel nach Datums- oder Zahlenfeldern abgefragt werden soll, wird die Sucheingabe entsprechend verarbeitet. *Elasticsearch* unterscheidet bei der Abfrage zwischen analysierten und nicht analysierten Feldern. Wenn ein nicht analysiertes Feld durchsucht wird, so wird die gesamte Suchanfrage in ihre einzelnen Terme zerlegt und auf Übereinstimmungen (*Exact-Value*) geprüft. Im Gegensatz dazu wird beim Durchsuchen eines analysierten Feldes die Sucheingabe zuerst mit dem passenden Analyzer zerlegt und behandelt, damit die erzeugten Suchbegriffe dem Feldinhalt entsprechen. Im Fall wie es nachfolgender Listing-Ausschnitt zeigt, werden alle Ergebnisse, die „LDBV“ (in Groß- oder Kleinschreibung) und/oder „München“ enthalten nach ihrer Relevanz sortiert zurückgegeben.

```
POST _search
{
  "query": {
    "match": {
      "firma": "LDBV München"
    }
  }
}
```

Listing 8: Volltextsuche

Eine besondere Form bzw. Erweiterung der Volltextsuche ist die q*uery_string*-Abfrage. Hier wird ein Abfrage-Parser verwendet, der die Eingabe verarbeitet und analysiert. Standardmäßig durchsucht eine q*uery_string*-Abfrage alle Felder. Die Syntax bietet mehr als nur die Suche nach einem einzelnen Feld oder Wort. Es ist möglich, die Suche mithilfe von Operatoren zu steuern. So können z.B. verschiedene Begriffe mit booleschen Operatoren kombiniert und Dokumente aus dem Ergebnis ausgeschlossen oder miteinbezogen werden. Da diese Abfrage viele Parameter/Operatoren haben kann, ist die Anwendung manchmal nicht ganz einfach.[44][45]

Die wichtigsten Parameter der q*uery_string*-Abfrage sind nachfolgend erklärt:

- ***query*:** Gibt den Abfragetext an.
- ***default_field*:** Gibt das Standardfeld an, in dem die Abfrage ausgeführt werden soll.
- ***default_operator*:** Der logische Standardoperator („oder“ bzw. „und“) wird verwendet, wenn kein anderer Operator angegeben ist.

Darüber hinaus gibt es Operatoren der q*uery_string*-Syntax, mit denen die Suche benutzerdefiniert verändert werden kann:

- ***Fields*:** Ein Feld oder mehrere Felder können gleichzeitig durchsucht werden. Die Abfrage kann auf zwei Arten erfolgen: indem man die Feldnamen gefolgt von einem Doppelpunkt und dem gesuchten Begriff eingibt, oder indem man die *fields* verwendet (vgl. nachfolgendes Listing)

```
1-      { "query_string": { "query": "field1 : query_term AND field2 : query_term" }}
2-      { "query_string ": { "query": "query_term", "fields": [ "field1", "field2"] }}
```

Listing 9: Fields Optionen

- ***Wildcards*:** Die *Wildcards*-Suche bietet sich an, wenn man zum Beispiel einzelne oder auch mehrere Zeichen des Suchbegriffs nicht kennt. Wenn ein einzelnes Zeichen ersetzt werden soll, verwendet man das Fragezeichen-Symbol (?); bei keinem oder mehreren Zeichen das Stern-Symbol (*). Wenn man also nur einen Teil einer Adresse kennt, kann man den unbekannten Teil ersetzen (vgl. nachfolgendes Beispiel). Allerdings kann sich die Suche mit *Wildcards* sehr schlecht auf die Performance auswirken (je nachdem wie viele Zeichen unbekannt sind), da unter Umständen sehr viele Kombinationen möglich sind.

```
Admiralbogen 4?     → für genau ein (beliebiges)Zeichen, bei einer Adresse also 40-49 (also 10
verschiedene Ergebnisse)

Admiral*            → für beliebig viele (auch null) Zeichen (also sehr viele Ergebnisse möglich)
```

Listing 10: Wildcard

- ***Fuzziness*:** Dieser Parameter erlaubt die Suche nach ähnlichen Begriffen, die dem Suchbegriff sehr ähnlich sind. Der Standard-Bearbeitungsabstand ist 2 und wird durch die Tilde angegeben (vgl. nachfolgendes Beispiel). Es entspricht der Levenshtein-Distanz, der minimalen Anzahl an Einfüge-, Lösch- und Ersetz-Operationen, um von der einen

Zeichenkette zur anderen zu kommen.[46] In den meisten Fällen sollte eine Bearbeitungsentfernung von 1 ausreichen, um 80% aller menschlichen Rechtschreibfehler zu erfassen und die Suche nicht unnötig zu verlangsamen[47].

Admiralbgoem~2 → wird noch erkannt, da es nur zwei Bearbeitungsschritte zu „Admiralbogen“ sind.
Admiralbgoem~1 → wird nicht mehr erkannt.

Listing 11: Fuzziness

3.5.3 Term-Level-Abfrage

Die Term-Level-Abfrage in *Elasticsearch* ähnelt der Suche in einem RDBMS. Es wird nach Dokumenten gesucht, die der Suchanfrage exakt übereinstimmen. Es gibt also keine teilweise passenden Ergebnisse und es erfolgt keine Bewertung bzw. Berechnung der Relevanz. Diese Anfrage wird eingesetzt, wenn zum Beispiel ein Geo-Punkt mittels Koordinaten gesucht wird oder man alle Postleitzahlen einer Stadt finden möchte. Term-Level-Abfragen können genutzt werden, um die Suchperformanz zu verbessern, weil sie keine Relevanz berechnen und leicht zwischengespeichert werden können.[48][49][36][50]

Die folgenden Schritte zeigen was bei einer gesendeten Term-Level-Abfrage intern ausgeführt wird:

1. Passende Dokumente finden (dazu wird der Suchbegriff mit dem invertierten Index abgeglichen)
2. *Bitset* erstellen (ein Array von 1ern und 0ern, das darstellt, welche Dokumente den gesuchten Term enthalten)
3. *Bitset* zwischenspeichern /*cachen* (bevor die Dokumentenliste für den Nutzer ausgegeben wird, wird das *Bitset* im Speicher abgelegt, um die Leistung zu verbessern)

Es könnte zu unerwünschten Ergebnissen kommen, wenn der Term im ursprünglichen Dokument verfügbar ist, während der Indizierung analysiert (d.h. in veränderter Form aufgenommen) wurde. Ein typisches Beispiel ist die Groß- und Kleinschreibung. Daher ist es oft nützlich, dasselbe Feld für verschiedene Zwecke auf verschiedene Arten zu indexieren. Dies ist der Zweck von sogenannten Multi-Feldern (*multi-fields*). So wird beispielsweise ein Begriff als ein analysiertes Textfeld für die Volltextsuche und als ein nicht analysiertes Schlüsselwortfeld (*keyword*) für die exakte Term-Suche angelegt (vgl. Listing 12).

```
"gemeinde": {
  "type": "text",              → wird analysiert.
  "fields": {
    "keyword": {
      "type": "keyword",       → wird nicht analysiert.
      "ignore_above": 256
    }
  }
}
```

Listing 12: Multifields

So können strukturierte und Volltextsuchanfragen während der Abfrage kombiniert werden.

3.6 Highlighting

Hervorhebungen geben dem Benutzer eine Vorstellung davon, worum es in einem Dokument geht und zeigen seine Beziehung zur Abfrage an. Es ermöglicht dem Benutzer, Teile aus einem Feld oder auch mehreren Feldern in den Suchergebnissen als hervorgehobenen Text mit Hilfe von HTML-Tags anzeigen zu lassen.[51] So wird in nachfolgendem Beispiel das Wort „Admiralbogen“ durch fette oder kursive Schrift hervorgehoben (vgl. Listing 13).

```
"label" : [ "<em>Admiralbogen</em> 45 80939 München" ]
```

Listing 13: Highlighting

3.7 Aggregation

Bei der Entwicklung von Suchlösungen können die Ergebnisse dabei helfen, die Qualität und den Fokus der Suche zu verbessern. Mithilfe von Aggregationen ist es zum Beispiel möglich, die Ergebnisse zu gruppieren und statistisch auszuwerten. Dies ist besonders bei großen, komplexen Datenmengen von Vorteil. Aggregationen in *Elasticsearch* werden wie Suchanfragen ebenfalls im JSON-Format ausgeführt und können miteinander kombiniert werden. [52] Diese Kombinationen sind leistungsfähiger als SQL und können einfach skaliert werden. *Elasticserach*-Aggregationen bestehen immer aus der Kombination von *Buckets* und *Metrics*.[53][54]

Buckets: Ein *Bucket* entspricht einer Sammlung von Dokumenten, die alle eine Gemeinsamkeit haben (z.B. Bayerische Städte). Bei der Ausführung von Aggregationen werden alle Dokumente durchsucht und – wenn sie die Kriterien erfüllen – dem entsprechenden *Bucket* hinzugefügt (so würde ein Dokument über München im *Bucket* „Bayerische Städte“ landen). *Buckets* können auch geschachtelt sein (im Beispiel: der *Bucket* „Bayerische Städte“ befindet sich im *Bucket* „Deutsche Städte“). Das Ergebnis ist eine Liste von *Buckets* mit den jeweils dazugehörigen Dokumenten.

Metrics: Damit zu den Dokumenten eines *Buckets* Statistiken erstellt werden können, benötigt man eine Metrik. Dabei handelt es sich meistens um mathematische Operationen wie beispielsweise Minimum, Maximum oder Mittelwert, die auf Basis der Werte/Inhalte der Dokumente bestimmt werden.

Je nach Wahl der *Bucket*-Kriterien und der Metrik entstehen ganz unterschiedliche Aggregationen. Aus dieser Vielzahl an Möglichkeiten werden drei Typen näher vorgestellt:

***Terms Aggregation*:** Dies ist eine der am häufigsten verwendeten Aggregationsmethoden. Das Ergebnis ähnelt dem "Group By"-Befehl in der *SQL*-Sprache. Hier erfolgt die Zuordnung von Dokumenten zu *Buckets* nach einem eindeutigen Feld-Wert (z.B. nach dem Feld „Regierungs-bezirk“).

Geo Bounds Aggregation: Diese metrische Aggregation berechnet den kleinsten Begrenzungs-rahmen, der alle Geo-Punkte einschließt. Sie ermittelt anhand der Dokumente bzw. der *Buckets* zwei Geo-Punkte, welche den oberen linken und den unteren rechten Koordinaten des Rechtecks entsprechen (vgl. Listing 14).[55]

Top Hits Aggregation: Diese Aggregation wird häufig als Unteraggregation verwendet, damit die relevantesten Dokumente einer Aggregation ein *Bucket* bilden. Diese Aggregation gibt reguläre Suchtreffer zurück, standardmäßig nach ihrer Punktzahl (*score*) sortiert. Daher können auch bestimmte Features kombiniert werden – insbesondere das *Highlighting*, welches für bestimmte Operationen der im Rahmen dieser Bachelorarbeit entwickelten Java API erforderlich ist. (vgl. Listing 15).[56]


```
"bounds": {
"my_bounds": {
  "top_left": {
    "lat": 48.21002135518938,
    "lon": 11.614876063540578
  },
  "bottom_right": {
    "lat": 48.21002135518938,
    "lon": 11.614876063540578
  }}}
```


Listing 14: Abfrage-Ergebnis einer Geo Bounds Aggregation


```
"hits": [{
  "_index": "bayern",
  "_type": "adressen",
  "_id": "n9VzBGIBLMu9bhAv3whf",
  "_score": 18.275902,
  "_source": {
    "agg_strasse": "Admiralbogen (München)
  },
  "highlight": {
    "agg_strasse": [
      "<em>Admiralbogen</em> (München)"
    ]}}]
```


Listing 15: Antwort auf eine Top Hits Aggregation-Abfrage

3.8 Kommunikation mit Elasticsearch

Um mit *Elasticsearch* kommunizieren zu können, gibt es die *Java* API und die *RESTful* API mit JSON über HTTP, die im Folgenden beschrieben werden.

3.8.1 HTTP-Kommunikation

Die Hauptschnittstelle für die Kommunikation mit *Elasticsearch* basiert auf HTTP und einer JSON-*RESTful*-API. Dies bedeutet, dass man sogar einen Webbrowser für einige grundlegende Abfragen und Anforderungen verwenden kann. Für anspruchsvollere Anfragen benötigt man jedoch zusätzliche Software wie z.B. *cURL* oder *Kibana Dev-Tools*. c*URL*, und alle anderen Sprachen kommunizieren mit *Elasticsearch* über ein HTTP-Protokoll. Diese primäre API lässt sich problemlos in nahezu jedes System integrieren, das HTTP-Anfragen senden kann. *Elasticsearch* nutzt als Daten- und Anfragetext-Format (*Request Body*) das JSON-Format.[57][58]

Wie jede HTTP-Anfrage besteht auch die *Elasticsearch*-Anfrage aus den gleichen folgenden Teilen:

```
In cURL geschrieben : curl -X<VERB> '<PROTOCOL>://<HOST:PORT>/<PATH>?<QUERY_STRING>' -d '<BODY>'

curl -XGET 'http://localhost:9200/_search?pretty' -H 'Content-Type: application/json' -d' { "query": { "match" :
{ "message" : "this is a test" } } } '
```

Listing 16: Bestandteile einer HTTP-Anfrage

In der nachfolgenden Tabelle werden die einzelnen Bestandteile der HTTP-Anfrage nochmals aufgelistet und genauer erläutert.[59]

	Beschreibung
Verb	Gibt die HTTP-Methode an: • **Get**: Ruft ein bereits indiziertes Dokument ab. • **Post**: Erstellt ein neues Dokumente (mit eindeutiger ID) oder führt eine Suchoperation durch. • **Put**: Sendet Konfigurationen an *Elasticsearch*. • **Delete**: Löscht ein Dokument aus dem *Elasticsearch*-Index.
Protokoll	http oder https
Host:Port	*Name* oder die *IP* des Computers, auf dem *Elasticsearch* installiert ist, gefolgt von dem Port, auf dem der HTTP-Dienst erreichbar ist.
Query_Path	Index, auf den sich die Anfrage bezieht.
Query_String	Alle optionalen Parameter einer Abfrage
Body	Anfragetext im JSON-Format

Tabelle 1: Bestandteile einer HTTP-Anforderungsnachricht

Die HTTP-Kommunikation kann auch über HTTP-*Clients* (z.B. *HttpComponents* von *Apache)* erfolgen. Ein HTTP-*Client* ist einfach zu erstellen und sehr praktisch, weil er interne Methoden und Aufrufe von Drittanbietern, die z.B. in *Plugins* implementiert sind, ausführen kann.[60][61]

Unter dem Punkten 3.8.3 und 3.8.4 werden noch weitere HTTP-*Clients* vorgestellt, die im Projekt verstärkt verwendet wurden.

3.8.2 Native Kommunikation

Eine zusätzlich Möglichkeit ist das Transport-Protokoll. Dieses wird über die *Java* API zur Verfügung gestellt. Die *Java* API kann von Programmen, die in *Java* oder einer anderen auf JVM basierenden Sprache geschrieben sind, verwendet werden. Vorteil dieser API ist eine integrierte *Cluster*-Erkennung. Diese erkennt automatisch Veränderungen im *Cluster* wie z.B. neue und wegfallende *Nodes*.

Das Transport-Protokoll und die zugehörige *Java* API werden auch intern von *Elasticsearch* selbst verwendet, um die gesamte *Node*-zu-*Node*-Kommunikation durchzuführen. Aus diesem Grund ist der Funktionsumfang der nativen API größer als der von der RESTful-API. Eine dieser zusätzlichen Funktionen ist beispielsweise die Möglichkeit administrativ auf den *Cluster* zuzugreifen. Die *Client*-Objekte beider APIs können die Operationen kumulativ in großen Mengen und asynchron ausführen.

Der *Transport-Client* stellt über das Transport-Protokoll eine Verbindung zum *Elasticsearch-Cluster* her. Der Standard-Port für dieses Protokoll ist 9300. Der *Client* tritt dem *Cluster* nicht als Daten-*Node* bei, sondern erhält nur die Transportadressen und kommuniziert bei jeder Aktion im Rundlauf-Verfahren.[62] Folgendes Listing zeigt die *Transport Client*-Verbindung in *Java*.

```
// on startup
Settings settings = Settings.builder().put("cluster.name","My_Cluster").build();
TransportClient client = new PreBuiltTransportClient(settings);
for(String serverString : serverList){ //serverString z.B. Localhost
    client.addTransportAddress(new TransportAddress(InetAddress.getByName(serverString), 9300))
}
// on shutdown
client.close();
```

Listing 17: Transport-Client in Java

Um den *Client* zu initialisieren, benötigt man den *Cluster*-Name, die IP-Adressen der *Nodes* im *Cluster* und die Port-Nummer, falls *Elasticsearch* nicht auf den Standard-Ports ausgeführt wird.

Die Firma *Elastic.co* hat jedoch angekündigt, dass der *Transport-Client* künftig nicht mehr unterstützt wird. Er soll in Version 7 als "Deprecated" (Veraltet) noch verwendbar sein und in Version 8 nicht mehr vorhanden sein. Die native Kommunikation wird durch einen neu eingeführten *High-Level-REST-Client* ersetzt werden. Es ist deshalb ratsam bei neuen Projekten bereits diesen Client einzusetzen.

3.8.3 cURL

cURL wurde in dieser Arbeit verwendet, um die JSON-Daten auf den *Elasticsearch*-Server hochzuladen. Es ist ein Datenübertragungswerkzeug, das vor allem als HTTP-Client verwendet wird. Es besitzt keine graphische Oberfläche und findet deshalb meist Einsatz in Shell-Skripten. Wie so ein Befehl mit *cURL* aussehen kann, zeigt nachfolgendes Beispiel:

```
curl -XGET 'localhost:9200/bayern/_search?pretty' -H 'Content-Type: application/json' -d' { "query": { "match_all": {}}} '
```

Listing 18: cURL Befehl

cURL sendet eine Anfrage an die angegebene *URL* mit der HTTP-Methode, die mit "-X" definiert ist (hier *GET*). Der Parameter „-d“ oder „--data“ sendet die angegebenen Daten an den HTTP-Server, in diesem Fall der HTTP-Port des *Elasticsearch-Clusters*. Mit dem Parameter „-H“ oder „--header“ fügt man HTTP-Header zu der HTTP-Anfrage hinzu.[63][64]

3.8.4 Kibana Dev-Tools

Kibana ist eine Frontend-GUI zum Anzeigen und Analysieren von Daten in *Elasticsearch*. Es wurde in *Javascript* erstellt und läuft in einem normalen Web-Browser. Auf der *Dev-Tools*-Seite befinden sich die Entwicklungswerkzeuge, die für die Interaktion mit Elasticsearch verwendet werden können. *Kibana Dev-Tools* hat eine sehr einfache Benutzeroberfläche und ist in ein Anfrage- und ein Antwortfenster unterteilt (vgl. nachfolgende Abbildung). Features wie *Syntax-Highlighting*, Autovervollständigung und automatisches Einrücken erleichtern die Bearbeitung von Suchanfragen im Anfrage-Fenster erheblich. Im Antwortfenster wird das jeweilige Ergebnis der Suchanfrage als JSON-Dokument dargestellt. Das Ein- und Ausklappen von Dokumententeilen ist hier ebenfalls möglich.[65]

Abbildung 8: Ausschnitt von der Kibana Dev-Tools Konsole

Abbildung 8 (linke Seite) zeigt den Anfang eines ca. 210 Zeilen langen Codes mit einer Test-Anfrage. Rechts daneben – im Antwortfenster – werden die gesamten zur Anfrage passenden Dokumente von *Elasticsearch* übersichtlich ausgegeben (ebenfalls gekürzt dargestellt). Aufgrund dieser anwendungsfreundlichen Konsole wurde *Kibana Dev-Tools* dazu verwendet, den Code der programmierten *Java*-Applikationen zu testen bzw. auftretende Fehler nachzuvollziehen und beheben zu können.

4 Elasticsearch im Vergleich

4.1 Vergleich mit ausgewählten DBMS

Im Folgenden werden vier ausgewählte Datenbankmanagementsysteme miteinander verglichen: *Elasticsearch*, *PostgreSQL*, *Solr* und *Splunk*. Während *PostgreSQL* als bekanntes Beispiel für ein relationales DBMS vertreten ist, stehen die übrigen drei für die neue Generation an Suchmaschinen, wobei *Solr* und *Elasticsearch* zu den führenden Open-Source-Suchmaschinen gehören. Für eine Übersicht dient Tabelle 2 mit den wesentlichen Merkmalen der vier Suchmaschinen. [66]

	Elasticsearch	**PostgreSQL**	**Solr**	**Splunk**
Beschreibung	Eine moderne Such- und Analysemaschine auf Basis von *Apache Lucene*	Weit verbreitetes Open Source RDBMS	Eine weit verbreitete, auf *Apache Lucene* basierende Unternehmenssuch-maschine	Analyseplattform für *Big Data*
Primäres Datenbankmodell	Suchmaschine	Relationales DBMS	Suchmaschine	Suchmaschine
Server-Betriebssysteme	Alle Betriebssysteme mit einer *Java Virtual Machine*	*FreeBSD, HP-UX, Linux, NetBSD, OpenBSD, OS X, Solaris, Unix, Windows*	Alle Betriebssysteme mit einer *Java Virtual Machine* und einem *Servlet-Container*	*Linux, OS X, Solaris, Windows*
Erstveröffentlichung	2010	1989	2004	2003
Aktuelle Version (im März 2018)	6.2.2	10.3	7.2.1	7.0.2
Lizenz	Open Source	Open Source	Open Source	kommerziell
Datenschema	Schemafrei	Ja	Ja	Ja
API und andere Zugriffskonzepte	*Java API, RESTful HTTP/JSON API*	*native C library, streaming API for large objects, ADO. NET, JDBC, ODBC*	*Java API, RESTful HTTP API*	*HTTP REST*

Tabelle 2: Datenbankmanagementsysteme im Vergleich

In der Kompatibilität unterscheiden sich die vier Datenbankmanagementsysteme fast gar nicht. Alle funktionieren unter den bekannten Betriebssystemen *Linux OS*, *Windows* und *Mac OS*. Auch Suchtools wie *Boosting* (Gewichten von Suchbegriffen nach Relevanz) und Filter bieten alle bis auf *PostgreSQL*. Aufgrund der gemeinsamen *Lucene*-Basis ähneln sich *Solr* und *Elasticsearch* sehr stark. In vielen Anwendungsfällen lässt sich daher kein klarer Sieger ermitteln. Im Bereich *Big Data* gewinnt *Elasticsearch* mit seiner einfachen Skalierbarkeit. Abbildung 9 zeigt, wie sich die

Popularität der vier genannten DBMS in den vergangenen fünf Jahren entwickelt hat (aufgestellt von der solidIT GmbH). Man erkennt schnell, dass *PostgreSQL* im Vergleich zu den vier Genannten noch immer führend ist, *Elasticsearch* aber im gleichen Zeitraum einen deutlich stärkeren Zuwachs hat.[67] Die aktuelle Entwicklung kann auch über *Google Trends* verfolgt werden. [68]

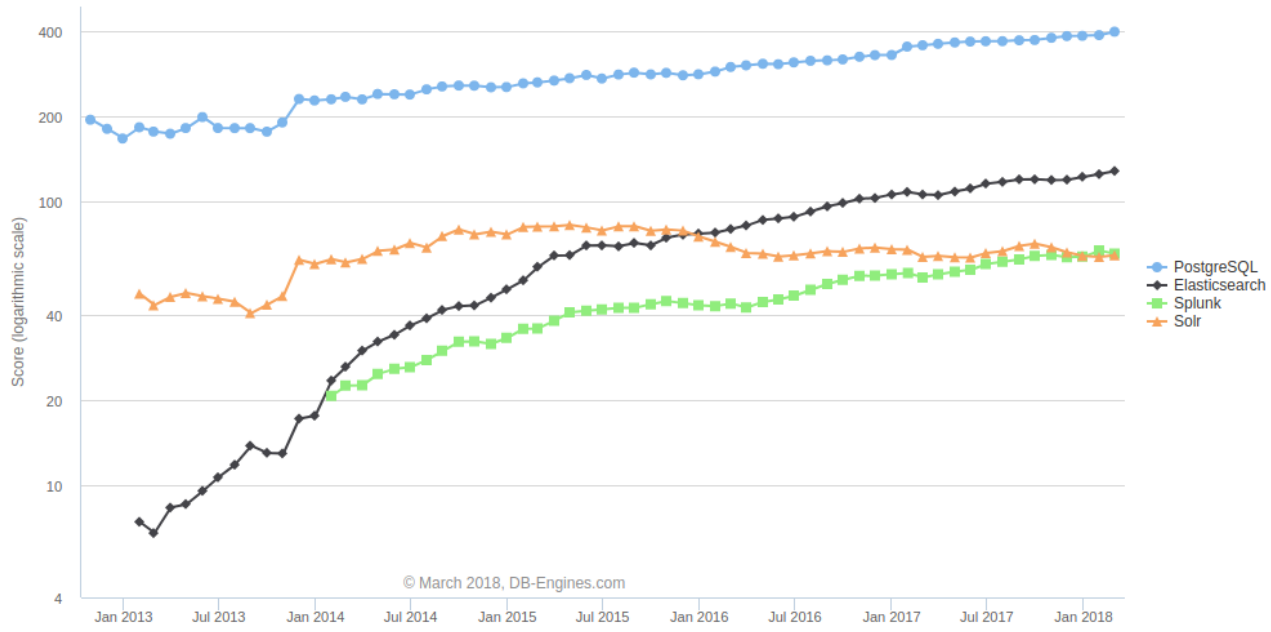

Abbildung 9: Entwicklung der Popularität von vier ausgewählten DBMS

4.2 Leistungsvergleich mit PostgreSQL

Wie einleitend erwähnt, befanden sich die *Adressdaten* ausschließlich in einer *PostgreSQL*-Datenbank bevor sie vor etwa zwei Jahren zusätzlich auf *Elasticsearch-Servern* gespeichert wurden. Aus diesem Grund wird an dieser Stelle der Vergleich dieser beiden DBMS vertieft, um die Beweggründe dieser Entscheidung besser nachvollziehen zu können.

Elasticsearch ist eine hoch skalierbare Open-Source-Suchmaschine. Auch wenn es als einfache Suchmaschine begann, entwickelte es sich schnell hin zu einer analytischen Maschine mit vielen nützlichen Features. Seine Architektur und Anwenderfreundlichkeit ermöglichen einen einfachen Einstieg. Die hohe Skalierbarkeit von *Elasticsearch* ist ebenso ein wichtiger Faktor, wenn es um die Leistungsfähigkeit des Systems geht.

Wie sieht es hingegen bei *PostgreSQL* aus? Auch hier gibt es natürlich die Möglichkeit, mit exakten Suchbegriffen Datensätze schnell zu durchsuchen. Möchte oder kann man nicht nach exakten Begriffen suchen, gibt es bei *PostgreSQL* zwei Möglichkeiten, die Daten zu durchsuchen: entweder über *Like*-Abfragen oder auch hier mittels einer Volltextsuche.[69]

Like-Abfragen werden bei allen modernen relationalen Datenbanken verwendet, um Textwerte mithilfe von *Wildcards* mit einem Muster abzugleichen. Es gibt zwei *Wildcards*, die mit den *Like*-Abfragen verknüpft sind, nämlich das Prozentzeichen "%", welches für eine beliebige Zeichenkette steht, und der Unterstrich "_", welcher ein beliebiges einzelnes Zeichen repräsentiert. *Like*-

Abfragen haben aber den Nachteil, dass durch die Verwendung von *Wildcards* am Anfang eines Datensatzes die Abfrage dazu gezwungen wird, jede einzelne Zeile einer Tabelle zu lesen. Dies kann besonders bei einer hohen Anzahl von Datensätzen sehr lange dauern und es gibt keine Möglichkeit, den Prozess zu beschleunigen.

Die Volltextsuche bei *PostgreSQL* ist inzwischen eine gängige effiziente Methode, direkt die ganze Datenbank zu durchsuchen. Mit einer Erweiterung von *PostgreSQL* kann man den dazu benötigten speziellen Datentyp (*Text Search Vector*) aktivieren. Da die Abfragen direkt in der Primär-Datenbank durchgeführt werden können, ist keine zusätzliche Software nötig.

Auch wenn *PostgreSQL* an dieser Stelle scheinbar auf der gleichen Stufe steht wie *Elasticsearch*, so kann letzteres DBMS mit vielen weiteren Features punkten – beginnend bei der Sucheingabe, wo man mit der automatischen Vervollständigung der möglichen Suchbegriffe Probleme durch Tippfehler schon umgehen kann. Die Stärken von *Elasticsearch* liegen aber v.a. in der Aggregation und der schnellen Verarbeitung von Suchanfragen bei komplexen Datenmengen. Auch die flexiblere und umfangreichere Term-Analyse (z.B. benutzerdefinierte *Analyzer*, Speichern von Synonymen), das dynamische *Mapping* und die Suche über mehrere Indizes sind Pluspunkte für *Elasticsearch*. Was auch bedacht werden muss, ist die die zunehmende Popularität. Bei der Volltextsuche liegt *Elasticsearch* vor *PostgreSQL*.

Zur Verdeutlichung der oben angesprochenen Leistungsunterschiede sollen die Beispiele der nachfolgenden Tabelle dienen. Sie zeigen anhand verschiedener *Like*-Abfragen, wie groß die Zeiteinsparung von *Elasticsearch* gegenüber *PostgreSQL* sein kann.

PostgreSQL	Zeit	Elasticsearch	Zeit	Faktor
`SELECT * FROM "public".hk_index_mat WHERE label LIKE '%Nürnberg%';`	837 ms	`GET bayern/_search { "query": { "query_string": { "default_field": "label", "query": "Nürnberg" } } }`	6 ms	ca. 135 mal schneller
`SELECT * FROM "public".hk_index_mat WHERE label LIKE '%Mühlweg 19 90427 Nürnberg%';`	830 ms	`GET bayern/_search { "query": { "query_string": { "default_field": "label", "query": "Mühlweg 19 90427 Nürnberg" } } }`	5 ms	ca. 160 mal schneller
`SELECT * FROM "public".hk_index_mat WHERE label LIKE '%Regensburg%' ORDER BY label;`	847 ms	`GET bayern/_search { "query": { "query_string": { "default_field": "label", "query": "regensburg" } }, "sort": [ { "label.keyword": { "order": "asc" } } ] }`	5 ms	ca. 160 mal schneller

hk_index_mat ist ein Tabelle, die aus der *hk_toindex_view*-View erstellt wurde, sowie ein Index, der zur Verbesserung der Datenbankleistung erstellt wurde. (vgl. nachfolgenden Code-Ausschnitt):
`CREATE TABLE hk_index_mat AS SELECT * FROM hk_toindex_view;`
`CREATE INDEX idx_hi_index_label ON hk_index_mat(label);`				

Tabelle 3: Leistungsunterschied zwischen Elasticsearch und PostgreSQL bei Like-Abfragen.

5 Programmierung und Implementierung

5.1 Getrennte Datenhaltung

Es soll zwischen zwei Datenbeständen unterscheiden werden, nämlich der Produktions- und der Präsentationsdatenbestand, wie im Abbildung 10 zu sehen ist.

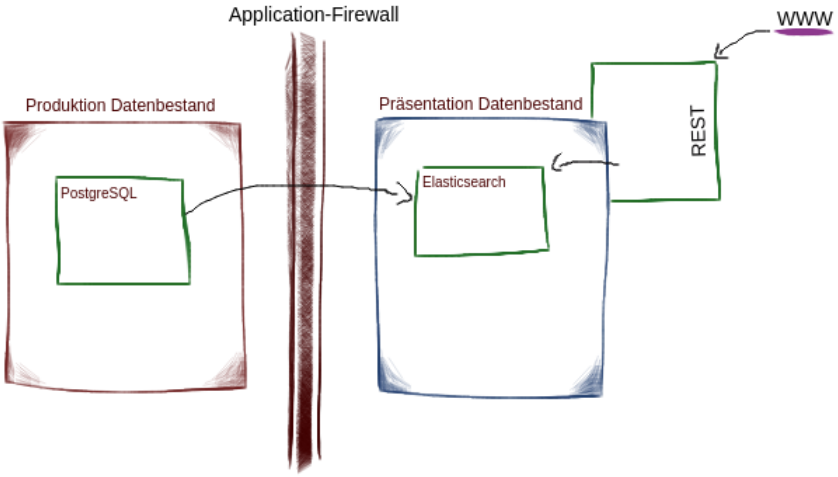

Abbildung 10: Getrennte Datenhaltung (Produktion und Präsentation)

Der Produktionsdatenbestand ist in der relationalen Datenbanken *PostgreSQL* zu finden. Das RDBMS ist für die Produktion bzw. die Datenhaltung konzipiert, da hiermit z.B. Transaktionen, Joins und Konsistenz möglich sind. Für diesem Zweck ist *PostgreSQL* also die beste Wahl.

Bei der Präsentation der Daten gibt es jedoch andere primäre Anforderungen, die nicht mit *PostgreSQL* erfüllt werden können. Für eine schnelle Verarbeitung und die Volltext-Suche ist *Elasticsearch* die bessere Wahl.

Zwischen beiden Datenbeständen besteht eine Entkopplung, was Vor- und Nachteile hat. Zu den Vorteilen zählt, dass beim Import in *Elasticsearch* alle kritischen und sensiblen Daten wie Eigentümer-Informationen ausgeschlossen werden können. Diese sind somit nicht wie die Präsentationdaten über das Internet abrufbar, wodurch sie im Falle eines Hacker-Angriffs geschützt sind. Außerdem ermöglicht die Entkopplung eine problemlose Bearbeitung des Produktionsdatenbestands ohne Beeinträchtigung durch externe Zugriffe/Abfragen. Bei der hohen Anzahl an Zugriffen, die laut LDBV am 20.03.2018 ca. 250.000 betrug, könnten andernfalls Serverüberlastungen oder Systemausfälle passieren. Die Popularität von *Elasticsearch* bezüglich der Volltext-Suche und seine große Community lassen zudem eine schnellere Unterstützung bei Problemen erwarten.

Auf der anderen Seite muss diese neue Technologie mit ihren umfassenden Möglichkeiten und Funktionen auch beherrscht werden können. Um den aktuellen Stand der Adress-Datenbank auch den Nutzern zugänglich zu machen, müssen die beiden Systeme zudem täglich synchronisiert werden.

Da aber diese beiden Punkte leicht zu bewerkstelligen sind und die Vorteile dieser Entkopplung deutlich überwiegen, entschied man sich am LDBV vor ca. zwei Jahren für diesen Schritt.

5.2 Docker

Im Rahmen dieser Arbeit wurde *Elasticsearch* und *PostgreSQL* zu Testzwecken auf einem sogenannten *Docker-Image* installiert, um ihren Funktionsumfang sowie ihre Leistung miteinander vergleichen und die programmierten *Java*-Applikationen testen zu können. *Docker* ist eine Open-Source-Software, die die Bereitstellung von Anwendungen vereinfacht indem sie isolierte Container baut, welche alle zur Laufzeit benötigten Ressourcen enthält. Diese Container können aufeinander aufbauen und miteinander kommunizieren. Jeder Container kann als ein komplettes Betriebssystem betrachtet werden. Er schließt eine einzelne Anwendung und alle ihre Abhängigkeiten wie Bibliotheken und Dienstprogramme in einer portablen *Image*-Datei ein, welche zwischen Nutzern geteilt werden kann (vgl. Listing 19).[70][71]

```
asli_mo@v****72:~$ docker ps -a
CONTAINER ID        IMAGE                                                 NAMES
ae4dfa****d3        postgres_94_postgis_21:latest                         mystifying_easley
c58837****04        postgres_94_postgis_21:latest                         flst_db
ff2957****bc        docker.elastic.co/kibana/kibana-oss:6.2.2             kibana
6f8361****2d        docker.elastic.co/elasticsearch/elasticsearch-oss:6.2.2   asli_Search
4aeb18****b4        postgres_94_postgis_21:latest                         address_db
```

Listing 19: Liste der für diese Arbeit angelegten Container

Beim Start des *Elasticsearch-Containers* werden die Standard-Ports 9200 und 9300 über -p („publish") dem ausführenden Betriebssystem zur Verfügung gestellt (vgl. Listing 20). Die Schreibweise „9300:9300“ bedeutet hier, dass der externe und interne Port 9300 ist. Damit kann leicht ein weiterer *Elasticsearch-Container* gestartet werden, indem der externe Port auf einen anderen Port „gemappt“ wird (z.B.: -p 9301:9300). Damit steht innerhalb von wenigen Sekunden ein weitere *Elasticsearch-Single-Node-Cluster* bereit.

```
docker run -d --restart=always --name bayern_search -p 9200:9200 -p 9300:9300
docker.elastic.co/Elasticsearch/ Elasticsearch-oss:6.2.2

Localhost:9200
{
  "name" : "JpojnaH",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "RgLWzHKGTei9xiMilpGrSw",
  "version" : {
    "number" : "6.2.2",
    "build_hash" : "10b1edd",
    "build_date" : "2018-02-16T19:01:30.685723Z",
    "build_snapshot" : false,
    "lucene_version" : "7.2.1",
    "minimum_wire_compatibility_version" : "5.6.0",
    "minimum_index_compatibility_version" : "5.0.0"
  },
  "tagline" : "You Know, for Search"
}
```

Listing 20: Docker-Container starten

5.3 Vorgehensweise

Jeden Tag kommt es in der Adressen-Datenbank des LDBVs zu Änderungen. Damit der Präsentations-d*atenbestand* immer auf dem neuesten Stand ist, findet daher auch täglich ein Update auf den *Elasticsearch*-Servern statt. Dieses Update soll anhand eines Skripts immer nach dem selben Muster ablaufen. Dafür sind einige Vorarbeiten nötig wie das Programmierung einer PostgreSQL-Funktion und zweier *Java*-Applikationen.

Für das bessere Verständnis soll in diesem Kapitel zunächst der Ablauf der dafür nötigen Schritte stichpunktartig gezeigt werden, bevor in den anschließenden Unterkapiteln diese ausführlich erläutert werden.

1. Adressen-Datenbank (*PostgreSQL*-Tabelle) mit einem sogenannten *View* vorbereiten.
2. *PostgreSQL*-Daten in *JSON*-Objekte mit einem *Elasticsearch-Bulk*-Format umwandeln
3. Generalisieren der *Java*-Applikation
4. Index-Einstellungen festlegen
5. Importieren der Daten auf *Elasticsearch*-Server mittels eines *Bash*-Skripts

5.3.1 View erstellen und anpassen

Die *PostgreSQL*-Tabelle mit dem Namen "ZAD" ist die Abkürzung für den zentralen Adressdienst des LDBV. Um zu vermeiden, dass ungewollte Änderungen an dieser Tabelle vorgenommen werden, wird die Adressen-Datenbank in einem *View* gespeichert. Hierbei handelt es sich um eine virtuelle Tabelle, die entsprechend modifiziert werden kann. Es findet keine physische Speicherung der Daten statt; die Informationen werden aus der Original-Tabelle geholt. Ein *View* kann eine Teilmenge der Tabelle darstellen mit einer Auswahl an bestimmten Spalten; es können aber auch neue hinzugefügt werden. Ein *View* stellt also einen Sicherheitsmechanismus dar, der den Zugriff und die Bearbeitung der Daten ermöglicht ohne dass an den zugrundeliegenden Basistabellen Änderungen stattfinden bzw. eine Berechtigung für den direkten Zugriff auf diese notwendig ist[72].

Die Original-Adressen-Datenbank besteht aus 20 Spalten (siehe Anhang 1), wovon aber nicht alle für die Adressen-Suchfunktion im *BayernAtlas* benötigt werden (z.B. leere Spalten wie "PointWKT"). Diese werden nicht in den *View* übernommen. Andere Spalten werden hingegen hinzugefügt oder bearbeitet, wie nachfolgend beschrieben ist. Das Ziel des *Views* ist es die Präsentationsdaten so aufzubereiten, dass möglichst keine weitere Bearbeitung in nachgeschalteten Programmen nötig ist.

Koordinaten transformieren

Die Spalte „wtk“ enthält die Gauß-Krüger-Koordinaten der räumlichen Bezugspunkte zu den Adressen (Hauskoordinaten), gespeichert im *WKT*-Format (*Well Known Text*) . Damit sie später in *Elasticsearch* als Geo-Punkt importiert werden können, müssen sie zunächst ins WGS84-Koordinatensystem transformiert und die Schreibweise angepasst werden. Dies ist wichtig, damit

später die *Bounding-Box*-Darstellung verwendet werden kann. So würde der Eintrag „Point(4297333.26 5543904.57)“ durch "50.00,9.17" ersetzt werden.

PostgreSQL mit der Erweiterung *PostGIS* beinhaltet für diesen Zweck bereits eigene Funktionen bzw. Methoden (z.B. *ST_Transform*), allerdings sind diese allein zu ungenau. So können sich unter Umständen Abweichungen/Differenzen von bis zu zehn Meter ergeben. Es gibt jedoch die Möglichkeit, diese mit der Erweiterung „NTv2“ zu konfigurieren. Dabei handelt es sich um ein modernes international verwendetes Transformationsverfahren für kartographische Anwendungszwecke. Bei diesem Ansatz wird neben der Basistransformation mithilfe einer Gitterdatei eine Restklaffenverteilung vorgenommen. Dieses Verfahren wird von vielen Geoinformationssystemen unterstützt und bildet die Basis für *BeTA2007* der AdV, das für die Transformation geotopographischer Daten eingesetzt wird. Um *NTv2* in *PostGIS* einzubauen, benötigt man zunächst die *PostgreSQL* interne Tabelle „spatial_ref_sys“, in der die für die Transformation benötigten Parameter aller *OGC*-konformen räumlichen Bezugssysteme aufgelistet sind. Es empfiehlt sich ein *Template* dieser Datenbanktabelle anzulegen. Von diesem können dann später weitere Tabellen abgeleitet werden. Alternativ kann auch die „spatial_ref_sys“-Tabelle selbst bearbeitet werden. Mittels *PROJ4*, einer Standard-Filterfunktion für Koordinaten-Umwandlung, kann das *NTv2* bzw. das *BeTA2007*-Modul auf einfache Art und Weise in *PostGIS* integriert werden. Die Modifikation der Transformations-funktion findet in der Spalte „proj4text“ statt (vgl. Listing 21)[73][74][75][76].

```
UPDATE spatial_ref_sys
SET proj4text = '+proj=tmerc +lat_0=0 +lon_0=9 +k=1 +x_0=3500000 +y_0=0 +ellps=bessel
+units=m +nadgrids=/misc/geo/NTv2/BETA2007.gsb +no_defs'

WHERE srid = 31467;

UPDATE spatial_ref_sys

SET proj4text = '+proj=tmerc +lat_0=0 +lon_0=12 +k=1 +x_0=4500000 +y_0=0 +ellps=bessel
+units=m +nadgrids=/misc/geo/NTv2/BETA2007.gsb +no_defs'

WHERE srid = 31468;
```

Listing 21: Modifikation der Transformationsfunktion

Mit diesem Schritt hat man eine sehr gute Transformationsgenauigkeit, die immer unter 10cm liegt, welche für diesen Anwendungsfall mehr als ausreichend ist.

Spalte „label“ hinzufügen

Um die Volltextsuche zu verbessern, wird eine neue Spalte mit dem Namen „Label“ hinzugefügt. Pro Zeile wird hier eine komplette Anschrift (Straße, Hausnummer (ggf. mit Hausnummer-Zusatz), Postleitzahl und Ort) gespeichert. Da die Adressen-Datenbank eine Unterscheidung macht zwischen Gemeinde bzw. Gemeindeteil und Postort bzw. Postortsteil, wird an dieser Stelle eine Funktion notwendig. Diese gleicht die Einträge der vier genannten Spalten ab und kreiert daraus gegebenenfalls eine Kombination, die dann als Ort in die Spalte Label geschrieben wird. Die Funktion wird *comparefields* genannt.

Warum eine Unterscheidung stattfindet, soll Abbildung 11 verdeutlichen.

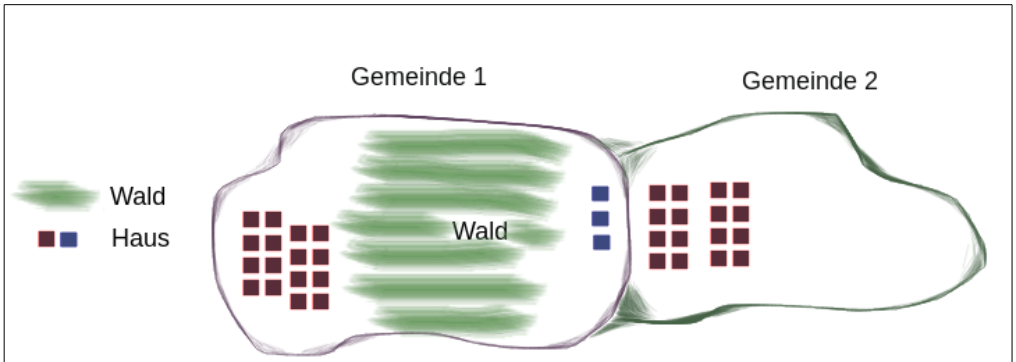

Abbildung 11: Gemeindegrenzen vs. Postortsgrenzen

Die blauen Häuser gehören zu Gemeinde 1, sind aber aufgrund der geographischen Lage näher zur Post, Feuerwehr oder Polizei der Gemeinde 2. Deswegen haben solche Häuser unterschiedliche Angaben in den Spalten „gemeindeteil“ und „postortsteil“ als die übrigen Datensätze in der Gemeinde 1, wie in Abbildung 12 (Zeile 2 oder 9) zu sehen ist.

#	combined	gemeinde	gemeindeteil	postort	postortsteil
1	Abenberg	Abenberg	Abenberg	Abenberg	Abenberg
2	Abenberg , Bechhofen	Abenberg	Bechhofen	Abenberg	Abenberg
3	Abenberg , Bechhofen	Abenberg	Bechhofen	Abenberg	Bechhofen
4	Abenberg , Beerbach	Abenberg	Beerbach	Abenberg	Beerbach
5	Abenberg , Dürrenmungenau	Abenberg	Dürrenmungenau	Abenberg	Dürrenmungenau
6	Abenberg , Ebersbach	Abenberg	Ebersbach	Abenberg	Ebersbach
7	Abenberg , Fischhaus	Abenberg	Fischhaus	Abenberg	Fischhaus
8	Abenberg , Kapsdorf	Abenberg	Kapsdorf	Abenberg	Kapsdorf
9	Abenberg , Marienburg	Abenberg	Abenberg	Abenberg	Marienburg
10	Abenberg , Obersteinbach ob Gmünd	Abenberg	Obersteinbach ob Gmünd	Abenberg	Obersteinbach

Abbildung 12: Gemeinde- und Post-Adressen in der Datenbank

Die Spalte „combined“ enthält das Ergebnis der *PostgreSQL*-Funktion *comparefields*. Hier werden die Einträge in den Spalten „gemeinde“, „gemeindeteil“, „postort“ und „postortsteil“ miteinander verglichen und bei ungleichen Termen so kombiniert, dass sowohl Gemeinde- als auch Postadressen gefunden werden, wenn ein Benutzer einen Suchbegriff eingibt. Nachfolgend der Code zu dieser Funktion:

```
# A, B, C oder D kann eine beliebige Zeichenkette(String) sein.
CREATE OR REPLACE FUNCTION comparefields(A TEXT, B TEXT,C TEXT, D TEXT)
RETURNS TEXT AS
$$
DECLARE
x TEXT; y TEXT; z TEXT;
BEGIN
    x:=' '; y:=' '; z:=' ';
        IF (A~*B) THEN x:=A; ELSEIF (B~*A) THEN x:=B; ELSE x:=(A ||' - '|| B); END IF;
        IF x~*C THEN y:=x; ELSEIF x!~*C THEN y:=(x ||' - '|| C); END IF;
        IF y~D THEN z:=y; ELSEIF y!~D THEN z:=(y ||' - '|| D); END IF;
RETURN z;
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;
```

Listing 22: PostgreSQL-Funktion "comparefields"

Aggregationsspalten anpassen

Auch die Aggregationen „agg_gemeinde“, „agg_postort“ und „agg_strasse“ müssen für die Volltextsuche noch angepasst werden. Auch hier gibt es die Problematik mit den teils unterschiedlichen Bezeichnungen. Die Aggregation „agg_strasse“ besteht zum Beispiel aus den Spalten „strasse“ und „gemeinde“. Für den Fall, dass sich die Einträge in der Gemeinde- und Gemeindeteil-Spalte unterscheiden, wird der Eintrag in „agg_strasse“ entsprechend angepasst (vgl. Listing 23). Außerdem wird die Schreibweise geändert: die Gemeinde wird nicht mehr in Klammern geschrieben, sondern mit einem Komma zum Straßenamen getrennt. Nachfolgend ein paar Beispieleinträge wie sie in der originalen *PostgreSQL*-Tabelle stehen und wie sie nun im *View* gespeichert werden.

```
      agg_strasse (PostgreSQL-Tabelle)          agg_strasse (View)
1-    3C-Ring (Landsberg am Lech)               3C-Ring, Landsberg am Lech
2-    A.-F.-vom-Endt-Straße (Auerbach i.d.Opf.) A.-F.-vom-Endt-Straße, Auerbach i.d.OPf.
3-    A.-J.-Ruckert-Straße (Poppenhausen)       A.-J.-Ruckert-Straße, Poppenhausen - Maibach
4-    A.-Knauer-Straße (Eggolsheim)             A.-Knauer-Straße, Eggolsheim - Kauernhofen
5-    A.-W.-Schultz-Straße (Memmingerberg)      A.-W.-Schultz-Straße, Memmingerberg
```

Listing 23: Anpassen der Aggregationsspalten

Wie man sieht, gab es in den Zeilen 3 und 4 Unterschiede zwischen Gemeinde und Gemeindeteil.

Zusammenfassend hier noch der Code für die Erstellung des *Views* (vgl. Listing 24). Dieser besteht aus den zwanzig Spalten mit den Bezeichnungen „label“, „land“, „bezirk“, „kreis“, „gemeinde“, „gemeindeteil“, „strasse“, „hausnummer“, „hausnumer_zusatz“, „postleitzahl“, „postort“, „postortsteil“, „postort_zusatz“, „agg_gemeinde“, „agg_postort“, „agg_strasse“, „gk_r_h“ (DHDN/Gauß-Krüger-Koordinaten), „wgs84_lat_lon“ (geographische Koordinaten im WGS84), „gdeart“(Stadt- bzw. Gemeindetyp) und „needs_postfix“.

```
CREATE OR REPLACE VIEW hk_toindex_view AS SELECT (strasse ||' '|| TRIM(hausnummer || ' ' ||
hausnummer_zusatz) || ' ' || postleitzahl || ' ' || comparefields(COALESCE(gemeinde,''),
COALESCE(gemeindeteil,''), COALESCE(postort,''), COALESCE(postortsteil,'')))AS label , land, bezirk,
kreis, gemeinde, gemeindeteil, strasse, hausnummer, hausnummer_zusatz, postleitzahl, postort,
postortsteil, postort_zusatz,
gdeart||' '||gemeinde||(CASE WHEN gemeinde!=gemeindeteil THEN COALESCE(' - '||gemeindeteil,'') ELSE
'' END) AS agg_gemeinde,
postleitzahl||' '||postort||COALESCE(postort_zusatz,'')||(CASE WHEN postort!=postortsteil THEN
COALESCE(' - '||postortsteil,'') ELSE '' END) AS agg_postort,
strasse||', '||gemeinde||(CASE WHEN gemeinde!=gemeindeteil THEN COALESCE(' - '||gemeindeteil,'')
ELSE '' END) AS agg_strasse,
'SRID=31468'||';'||wkt AS gk_r_h,
(ST_Y(ST_AsText(ST_Transform(ST_GeomFromText(wkt,31468),4326)))::NUMERIC ||','||
ST_X(ST_AsText(ST_Transform(ST_GeomFromText(wkt,31468),4326)))::NUMERIC) AS wgs84_lat_lon, gdeart,
needs_postfix FROM hk_toindex;
```

Listing 24: PostgreSQL-View

Die Funktion *coalesce* dient dazu, *NULL*-Einträge aufgrund fehlender Informationen zu vermeiden. Sollte also beispielsweise kein Gemeindeteil eingetragen sein, wird statt *NULL* eine leere Zeichenkette (String) übernommen.

5.3.2 Von PostgreSQL zu Elasticsearch

Die Daten im *View* befinden sich noch in einem *PostgreSQL*-Tabelle und müssen im nächsten Schritt in *JSON*-Dateien umgewandelt werden. Dafür ist eine *Java*-Applikation notwendig, denn es handelt sich bei der Adressen-Datenbank um eine *PostgreSQL*-Tabelle mit ca. 3.5 Millionen Einträgen. Die Anwendung mit dem Namen *Postgres2JSON.jar* beinhaltet mehrere Arbeitsschritte, die nachfolgend beschrieben werden.

Umwandlung ins NDJSON-Format

Bei *Elasticsearch* erfolgt der Import großer Datenmengen über die *Bulk-API*, da diese viele Indexoperationen mit einem einzigen *API*-Aufruf durchführen kann. Damit kann die Indizierungsgeschwindigkeit deutlich erhöht werden. Die *Bulk-API* erwartet Daten im *NDJSON*-Format [77]. Die Struktur dieses Formats zeigt Listing 25. Pro Dokument (d.h. pro Zeile der *PostgreSQL*-Tabelle) gibt es zwei Zeilen, wobei jede Zeile mit einem Zeilenvorschubzeichen (**\n**) beendet wird, einschließlich der letzten Zeile.

```
{ action: { metadata }}\n
{ request body }\n
{ action: { metadata }}\n
{ request body }\n
```

Listing 25: Bulk-API-Struktur

Die erste Zeile gibt an, welche Aktion für das darauf folgende Dokument ausgeführt werden soll. Diese Aktionen sind möglich:

- ***Create***: Erstellt ein Dokument nur, wenn es noch nicht existiert.
- ***Index***: Erstellt ein neues Dokument oder ersetzt ein vorhandenes Dokument.
- ***Update***: Führt eine Aktualisierung durch.
- ***Delete*:** Löscht ein Dokument.

Die zweite Zeile besteht aus dem Dokument (*_source*) selbst – also den Feldern (*fields*) und Werten (*values*), die das Dokument enthält. *PostgreSQL* bietet seit der Version 9.2 eine Funktion mit dem Namen *row_to_json()* um Datensätze im JSON-Format auszugeben. Dies vereinfacht im *Java*-Programm die Eingabe des *Request Body*. Dadurch kann JSON direkt vom Datenbankserver zurückgegeben werden. Listing 26 zeigt ein Beispiel eines Adressdatensatzes im NDJSON-Format.


```
{ "index":{} }\n
{"strasse":"Ebersdorfer Straße","label_sort":"Ebersdorfer Straße 00090 96465 Neustadt b.Coburg
-      Ebersdorf","bezirk":"Oberfranken","agg_postort":"96465      Neustadtb.      Coburg      -
Ebersdorf","agg_strasse":"Ebersdorfer Straße, Neustadt b.Coburg - Ebersdorf","gdeart":"Große
Kreisstadt","label":"Ebersdorfer      Straße      90      96465      Neustadt      b.Coburg      -
Ebersdorf","agg_gemeinde":"Große         Kreisstadt         Neustadt         b.Coburg         -
Ebersdorf","wgs84_lat_lon":"50.3266662460062,11.15213379601","kreis":"Coburg","postleitzahl":"
96465","postortsteil":"Ebersdorf","gemeinde":"Neustadt
b.Coburg","gk_r_h":"SRID=31468;POINT(4439730.75
5577084.19)","hausnummer":"90","postort_zusatz":"b.
Coburg","land":"Bayern","needs_postfix":false,"gemeindeteil":"Ebersdorf","hausnummer_zusatz":"
","postort":"Neustadt"}\n
```


Listing 26: NDJSON-Format

Das Exportprogramm wurde so entwickelt, dass die *.json*-Dateien nach Erreichen einer Größe von 99 MB gesplittet werden, da *Elasticsearch* in der unkonfigurierten Installation keine größeren Dateien importieren kann. Die gewünschte Dateigröße kann vom Benutzer beim Export mit der "-s" -Option parametrisiert werden, falls andere Dateigrößen gewünscht sind.

5.3.3 Java-Applikation generalisieren

Die *Java*-Anwendung konnte bisher nur über *NetBeans IDE* genutzt werden. Damit auch andere Benutzer die Applikation verwenden oder verschiedene Variablen nach Bedarf anpassen können ohne den Quellcode zu manipulieren, wird die sogenannte *Apache Commons-CLI* verwendet, eine *Java*-Bibliothek, die die Analyse von Befehlszeilenoptionen ermöglicht. So kann der Benutzer über das *Terminal* (*Shell*) alle Variablen eingeben und die gewünschten *JSON*-Objekte erhalten. Das nächste Beispiel zeigt die Ausführung der *Java*-Anwendung:

```
asli_mo@v****72:~/NetBeansProjects/Postgres2JSON/target$ java -jar postgres2json-1.1-jar-with-dependencies.jar
Missing required options: db, vt
usage: java -jar *.jar -db -vt Optional: [-n] [-p] [-s] [-settings] [-test] [-v]
Postgres2JSON: use the following options to import a PostgreSQL view|table into Elasticsearch-Json-Bulk-Format. Or the
option [-settings] to get an Elasticsearch-DefaultMapping
        -db,--database <args>             enter Datebase url, Database username, Database password. Example: -db
                                          jdbc:postgresql://host/database user pass
        -n,--column <args>                a PostgreSQL-column to be modified for the Naturalsort. Example: -n clmn
                                          clmnNew 3 (11 -> 011), repeate for new column!
        -p,--path <arg>                   locate the save path. default is Jar AbsolutePath:
                                          /home/asli_mo/NetBeansProjects/Postgres2JSON/target
        -s,--size <arg>                   enter size(in mb) of Json output file. Default and MAX 99mb. Example: -s 5
        -settings                         only Defaultmappings with Shell Skript, check the Skript to set the args
        -test <arg>                       try an output from the view|table with a limit. Example: -test 10
        -v,--verbose                      print feedback
        -vt <arg>                         enter postgreSQL view- or table name. Example: -vt myview | -vt mytable
```

Listing 27: Ausführen der JAR-Datei

Wenn die Anwendung gestartet wird, werden die oben gezeigten Anweisungen mit einer detaillierten Erklärung der einzelnen Optionen angezeigt. Im Gegensatz zu den optionalen Angaben (siehe *Optional*, dritte Zeile), die bei Bedarf eingegeben werden können, müssen die erforderlichen Informationen (siehe *usage*, dritte Zeile) in jedem Fall angegeben werden.[78]

5.3.4 Mapping und Settings festlegen

Um ein dynamisches *Mapping* zu vermeiden, muss ein Mapping explizit angegeben werden. Die Idee ist, ein benutzerdefiniertes *Mapping* und *Setting* für die Adressen vorzunehmen. Die Anwendung erstellt zunächst eine Art *Standard Mapping*, das als Orientierung und Grundlage dient. So werden alle Felder als Multi-Felder zugewiesen, das heißt einmal als Text und einmal als *Keyword* (*vgl.* Listing 28*).* Es erleichtert die im weiteren Verlauf die Arbeit, indem der Indexname und der Typname automatisch mit korrekter Formatierung geschrieben wird.

```
PUT bayern
{
 "settings": {},
 "mappings": {
  "adressen": {
   "dynamic": false,        → Verhindert den Dynamischen Mappings
   "properties": {
    "strasse": {
     "type": "text",         → 1. Als Text für die Volltextsuche
     "fields": {
      "keyword": {
       "type": "keyword",    → 2. als Keyword für die Exakt-Value-Suche
       "ignore_above": 256
      }
     }
     }...
```

Listing 28: Standard Mapping

Das *Standard Mapping* wird anschließend erweitert und durch *Settings* mit einem benutzer-definierten *Analyzer* ergänzt. Die ursprünglichen Index-Einstellungen sahen wie folgt aus (vgl. Abbildung 13):

```
1  PUT / My_Index
2  {
3    "settings": {
4      "analysis": {
5        "filter": {
6          "german_stop": {
7            "type": "stop", "stopwords": "_german_"
8          },
9          "german_keywords": {
10           "type": "keyword_marker", "keywords": ["Beispiel"]
11         },
12         "german_stemmer": {
13           "type": "stemmer", "language": "light_german"
14         }
15       },
16       "char_filter": {
17         "my_char_filter": {
18           "type": "html_strip"
19         },
20         "my_mapping_filter": {
21           "type": "mapping",
22           "mappings": ["str. => strasse", "D-=> "]
23         }
24       },
25       "analyzer": {
26         "def_german": {
27           "tokenizer": "standard",
28           "filter": [
29             "lowercase", "german_stop", "german_keywords",
30             "german_normalization", "german_stemmer"
31           ],
32           "char_filter": [
33             "my_char_filter",
34             "my_mapping_filter"
35           ]
36         }
37       }
38     }
39   },
40   "mappings": {
41     "adressen": {
42       "dynamic": false,
43       "properties": {
44         "my_field": {
45           "type": "text",
46           "fields": {
47             "keyword": {
48               "type": "keyword", "ignore_above": 256
49             }
50           }
51         },
52         "geo_field": {
53           "type": "geo_point", "index": false
54         }
55       }
56     }
57   }
```

Abbildung 13: Settings und Mappings

Unter *Settings* (Zeile 3 bis 40) wurde der Standard-*Tokenizer* verwendet (Zeile 27). Dieser bietet Grammatik-basierte Wort-Segmentierung auf Basis des *Unicode* Standards[79]. Ihm folgt eine Sammlung von Filtern (Zeile 28 bis 31), welche aus folgenden Komponenten besteht:

- Deutsche Stoppwörter (*german_stop*): entfernt Stoppwörter aus den Termen (z.B. aber, als, ich, ja) – Liste unter [80].
- Deutsche Schlüsselbegriffe (*german_keywords*): enthält Wörter, die nicht gelöscht werden sollen
- Deutscher Wortstamm (*german_stemmer*): automatisches Zurückführen der Terme auf ihren Wortstamm, ausgelegt für die deutsche Sprache – Liste unter [81].
- Kleinbuchstaben (*lowercase*): Terme in Kleinbuchstaben umwandeln
- Sonderzeichen-Umformung (*german_nor-malization*): wandelt die Sonderzeichen in der deutschen Sprache/Schrift um (z.B. "ß" in "ss") – Liste unter [82].

Der *Char*-Filter (Zeile 32) besteht aus einem *HTML-Strip* und einem Wörter-*Mapping*.

Der benutzerdefinierte Analyzer wird unter dem Namen *def_german* (Zeile 26) gespeichert.

Unter *Mapping* (Zeile 40) wurden allen Felder jeweils zwei Datentypen zugewiesen: Text-Felder werden beim Indexieren analysiert, *Keyword*-Felder werden nicht analysiert/geändert und normalerweise zum Filtern, Aggregieren und Sortieren verwendet. Außerdem gibt es ein Feld des Typs *geo_point* (Zeile 52). „*index : false*“ bedeutet in hier, dass das Feld nicht analysiert wird.

Diese Index-Einstellungen lieferten leider nicht die gewünschten Ergebnisse, sodass sie überarbeitet werden mussten. Grund dafür ist, dass die deutschen Stoppwörter nicht mehr durchsucht werden können, da sie aus den Token-Streams entfernt werden. Man könnte sie natürlich als *Filter-Keywords* markieren und explizit eingeben, bei über 3 Mio. Adressen ist es aber schwer alle zu erfassen. Außerdem ist der Filter mit den deutschen Wortstämmen wenig sinnvoll bei der Adresssuche, da man keine Verben sondern Namen in Volltextsuche eingibt. Aus diesem Grund wurde ein neuer *Analyzer* definiert, der dieses Problem besser löst (siehe Anhang 2). Dieser benutzerdefinierter *Analyzer* hat folgende Spezifikationen:

- *HTML Strip Char Filter*: Dieser *Char*-Filter entfernt *HTML*-Elemente aus dem Text
- *Lowercase Token Filter*: Dieser *Char*-Filter wandelt die Terme in Kleinbuchstaben um
- *Unique Token Filter*: Dieser Filter indiziert eindeutige Terme, d.h er entfernt Duplikate [83]
- *Mapping Char Filter*: Dieser Filter ersetzt eine Liste von Buchstaben in anderen Buchstaben
- *Pattern Tokenizer*: Dieser *Tokenizer* verwendet einen regulären Ausdruck in *Java*, um einen Text in Terme aufzuteilen

5.3.5 Daten-Import

Für den automatischen Import bzw. das Indizieren der *.json*-Dateien auf *Elasticsearch*-Server, generiert die *Java*-Applikation ein *Bash-Skript*, das mit einer *For*-Schleife die *cURL*-Import-Befehle für alle *.json*-Dateien ausführt (vgl. Listing 29). Damit man im Falle eines Fehlers eine Rückmeldung bekommt, kann mit dem JSON-Befehlszeilenprozessor kurz "*jq*" nach bestimmten Begriffen gesucht werden.[84] So wird hier nach dem *Keyword* ".errors" gesucht und mit "*grep*" eine Rückmeldung gegeben für den Fall, dass das *Keyword* den Wert „true“ hat (also ein Fehler vorhanden ist).

```
#!/bin/bash
LISTE=$(ls *.json)
for DATEI in $LISTE
do
echo "Importiere $DATEI .."
curl -s -H "Content-Type: application/json" -XPOST 'localhost:9200/flurstuecke/flurstuecke/_bulk' --data-binary
@"$DATEI" | jq ".errors" | grep true
done
```

Listing 29: Bash-Skript

Bevor die Dateien jedoch importiert werden können, müssen zunächst die Index-Einstellungen (*Mapping* und *Settings*) festgelegt werden. Damit gibt es dann ein fertiges Aktualisierungspaket, das aus *.json-Dateien*, *Mapping.json* und dem *ImportSkript* besteht. Im Echtbetrieb am LDBV wird das Aktualisierungspaket nun in das Zielsystem kopiert. Danach werden die Dateien mithilfe des *ImportSkripts* entsprechend der Index-Einstellungen auf den *Elasticsearch*-Server importiert (vgl. Listing 30). Im Rahmen dieser Arbeit kann auf diesen Zwischenschritt verzichtet werden. Die Daten bleiben auf dem lokalen Server.

```
asli_mo@va****2:~/home/asli_mo/NetBeansProjects/Postgres2Elasticsearch/json_output$ bash import.sh
Importiere elasticsearchJSON0.json ...
Importiere elasticsearchJSON1.json ...
```

Listing 30: Importieren der JSON-Dateien in Elasticsearch

Der gesamte Prozess hängt von der Größe des *PostgreSQL*-Tabelle bzw. des *Views* ab. Das Messen der Ausführungszeit kann mithilfe des Befehls „Time“ durchgeführt werden[85].

5.3.6 Bash-Skripte

Anschließend werden alle Arbeitsschritte in zwei *Bash*-Skripte namens **export_database.sh** und **importJSON.sh** gepackt. Dies sind die Punkte, die abgearbeitet werden:

1. *PostgreSQL*-Daten im *NDJSON*-Format speichern
2. Index erstellen und das vorab gespeicherte *Mapping* anwenden
3. *JSON*-Dateien in *Elasticsearch* indizieren
4. Index-Alias anwenden
5. Alte Alias-Namen/Indizes sowie identische Indizes (falls vorhanden) löschen

In diesem Skript ist der *Postgres2JSON.jar*-Pfad gespeichert. Er wird mit vordefinierten Einstellungen durchgeführt. Mit *cURL*-Befehlen erfolgt die Kommunikation mit *Elasticsearch*. Diese Einstellung ist nur für diese Anwendung geeignet; dies gilt auch für das *Mapping*. Wenn der Benutzer dieses Skript für andere Datenbanken oder einen anderen Server oder Port verwenden möchte, muss er die Variablen im Skript *export_database.sh* ändern. Diese sind in der folgenden Tabelle definiert:

Variable	**Beschreibung**
`INDEXNAME=adresse_$TS`	Index Name
`TMPDIR=tmp/elasticadresse_$TS`	Export Ordner
`SERVER=localhost:9200`	Server und Port
`TYPE=adresse`	Typbezeichnung
`ALIAS=adresse`	AliasName
`JARFILE=/home/asli_mo/NetBeansProjects/Postgres2JSON/target/Postgres2JSON-1.0-jar-with-dependencies.jar`	Java Jar-Archive-Pfad
`$JARFILE -db jdbc:postgresql://va****2.va-lvg.bvv.bayern.de:5432/zad USER PASS -i $INDEXNAME -t $TYPE -v hk_toindex_view -n label label_sort 6 -p $TMPDIR`	Optionen wie Datenbanken-Verbindung, *View* oder Tabellenname usw. der .jar- Archive.

Tabelle 4: Skript-Variablen

Das vollständige Skript befindet sich in Anhang 3. Anhang 3.1 (*export_database.sh*) befasst sich mit der Umwandlung der Daten von PostgreSQL zu JSON. Anhang 3.2 (*importJSON.sh*) führt den Import der *.json*-Dateien auf *Elasticsearch*-Server aus. Ein Anwendungsbeispiel der *Bash*-Skripte mit einer Überprüfung der Dokumentenanzahl in *PostgreSQL* und dem Import der Dokumente in *Elasticsearch* zeigt Anhang 3.3.

6 REST-Fassade

Für den letzten Schritt muss eine zweite Java-Anwendung programmiert werden, um zwischen der Benutzereingabe und der Serverantwort die Ausgabe für den Nutzer zu optimieren. Die REST-Fassade hat bestimmte Vorteile in Bezug auf Schutz und Benutzungsfreundlichkeit, die im Folgenden beschrieben werden und in Abbildung 14 zu sehen ist.

- Einfache Schnittstelle – der Nutzer gibt nur den Suchbegriff ein, statt eine komplette *Elasticsearch*-Abfrage zu erstellen
- Ein Service, der unabhängig vom zugrundeliegenden Produkt und dessen Version nach außen, für den Nutzer immer gleich ist
- Veredelung der Ergebnisse, um sie für den Anwender verständlicher und anschaulicher zu machen als die Ausgabe von Elasticsearch selbst
- Kein direkter Zugriff auf den Elasticsearch-Server, um ungewollte Änderungen zu verhindern.
- Die Möglichkeit der Authentisierung und Protokollierung (wer benutzt den Service, und wie oft)

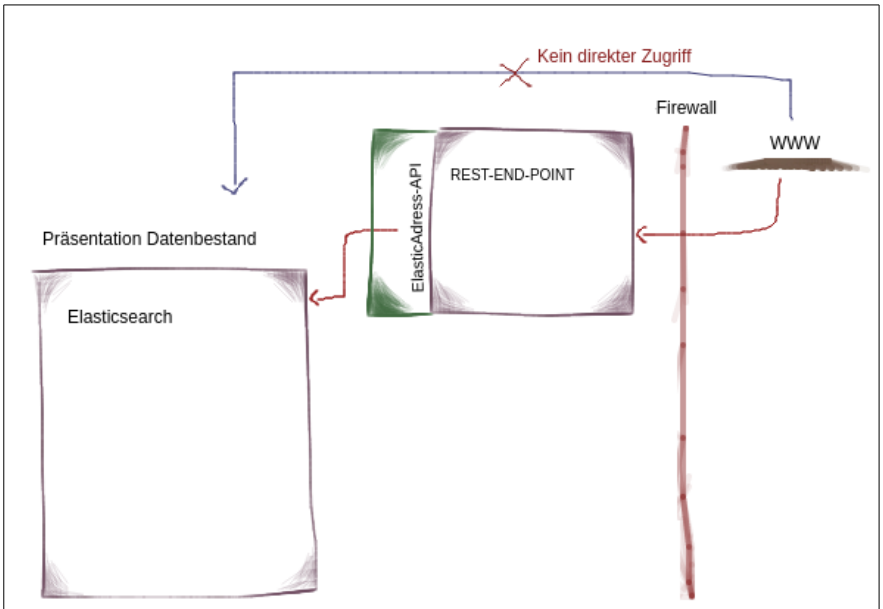

Abbildung 14: REST-Fassade

Für das bessere Verständnis sei hier der Ablauf stichpunktartig aufgezeigt:

1. Vorbereiten der Suchabfrage
2. ElasticAdress-API
3. REST-Endpoint

Bevor die Programmierung in Java erfolgte, wurde die Suche zunächst mittels *Kibana Dev-Tools* vorbereitet und getestet.

6.1 Vorbereiten der Suchabfrage

Wie im Kapitel 3.5 beschrieben, gibt es verschiedene Abfragearten in *Elasticsearch.* Für diese Anwendung wird *query_string*-Abfrage [45] verwendet, da sie die *Lucene-Query*-Syntax unterstützt. Die Suchabfrage soll eine *Terms Aggregation* mit zwei Sub-Aggregationen (*Geo Bounding Aggregation* und *Top Hits Aggregation*) ausgeben, sowie eine Sortierung der Ergebnisse vornehmen und nur gewisse Felder in die Ausgabe aufnehmen (unter *_source* anzugeben). Abbildung 15. zeigt einen Ausschnitt des Eingabefensters von *Kibana Dev-Tools* mit einer Suchabfrage.

```
GET bayern/_search
{
  "size": 20,
  "query": {
    "query_string": {
      "query": "(hauptstraße*) OR (hauptstraße~2)",
      "fields": ["agg_gemeinde", "agg_postort",
        "agg_strasse", "label", "wgs84_lat_lon.keyword"
      ],
      "type": "best_fields",
      "default_operator": "and",
      "fuzziness": "AUTO"
    }
  },
  "_source": {
    "includes": ["label", "wgs84_lat_lon", "gk_r_h"],
    "excludes": []
  },
  "sort": [{
    "label_sort.keyword": {
      "order": "asc"
    }
  }],
  "aggregations": {
    "agg_gemeinde.keyword": {...},
    "agg_postort.keyword": {...},
    "agg_strasse.keyword": {
      "terms": {
        "field": "agg_strasse.keyword",
        "size": 20,
        "order": [{
          "_count": "desc"
        }, {
          "_key": "asc"
        }]
      },
      "aggregations": {
        "bounds": {
          "geo_bounds": {
            "field": "wgs84_lat_lon",
            "wrap_longitude": true
          }
        },
        "Top_Hits": {
          "top_hits": {
            "_source": {
              "includes": ["agg_strasse"],
              "excludes": [""]
            },
            "highlight": {
              "fields": {
                "agg_strasse": {}
              }
            }
          }
        }
      }
    }
  },
  "highlight": {
    "fields": {
      "label": {}
    }
  }
}
```

Abbildung 15: Suchanfrage vorbereiten

Die *query_string*-Abfrage (ab Zeile 5) kann entweder eine *Wildcard* oder eine *Fuzziness* beinhalten, deswegen muss die Such-Anfrage zweimal geschrieben werden – getrennt durch den *OR*-Operator. Die Tilde (~) gibt die Unschärfe der Suche an. Sie beträgt in diesem Fall 2. Das heißt, es erzeugt alle möglichen übereinstimmenden Terme, die innerhalb der maximalen Bearbeitungsentfernung

liegen, und überprüft dann welche dieser erzeugten Terme tatsächlich im Index existieren. Der Stern (*) steht für den Wildcard und bedeutet beliebig viele (auch null) Zeichen folgen auf den Suchbegriff.

Der Parameter *fields* (Zeile 7) ist wichtig, da Die *query_string*-Abfrage auch für mehrere Felder ausgeführt werden kann. Unter dem Parameter mit dem Namen *default_operator* (Zeile 11) ist festgelegt, wie die Such-Anfrage behandelt wird, wenn sie aus mehreren Termen besteht und kein expliziter Operator angegeben ist. Ein Beispiel hierzu ist in Listing 31 zu sehen.

```
(Admiralbogen 45 80939 München) → (Admiralbogen AND 45 AND 80939 AND München)
```

Listing 31: Default Operator "And"

Die *Aggregations*-Parameter ermöglichen die schnelle Berechnung und Zusammenfassung von Daten, die sich aus der Abfrage ergeben. Sie sind für die spätere Bearbeitung der *Java-API* hilfreich. Da es wenig Sinn macht, dem Benutzer hunderte von Ergebnissen für eine bestimmte Abfrage auszugeben, wird nicht die gesamte Anzahl der Abfrageergebnisse angezeigt, sondern Aggregationen davon. Es gibt drei *Aggregations*-Spalten in der *PostgreSQL*-Tabelle, wobei jede einer einfachen Sammlung mehrerer Spalten entspricht. Punkt „Aggregationsspalten anpassen“ in Kapitel 5.3.1 wird zeigt, wie solche *Aggregations*-Spalten aufgebaut sind.

Diese Spalten sind die Grundlage für die *Elasticsearch*-Aggregationen. In Zeile 103 wird nach der Spalte „agg_strasse“ aggregiert. Wichtig ist hier, dass der Feldtyp *Keyword* verwendet wird, da es um den genauen Begriff (*exact value*) geht.

Der *Sort*-Parameter (Zeile 19) hat bei Tests unerwünschte Ergebnisse bzw. eine falsche Sortierung geliefert. Das liegt daran, dass die Standard-Sortierfunktionen in fast jeder Programmiersprache *ASCII*-Werte miteinander vergleichen, was eine Ordnung erzeugt, die nicht mit der menschlichen Logik übereinstimmt. Das bedeutet bei einem Array mit den Werten [2,1,10] würde die Sortierung folgendermaßen aussehen: [1,10,2]. Um dies zu umgehen gibt es in vielen Programmiersprachen den sogenannten alphanumerischen Algorithmus.[86] Dieser Algorithmus sortiert eine Zeichenfolge – bestehend aus Buchstaben und Ziffern – nach folgendem Muster: Ziffern in der Reihenfolge aufsteigend, Nicht-Ziffern in der ASCII-Reihenfolge.[87] Das Endergebnis ist eine natürliche Sortierreihenfolge. Leider ist diese Funktion ist nicht in *Elasticsearch* integriert. Man kann dafür jedoch ein *Plugin* verwenden oder es ebenfalls im Skript programmieren. Folgender Lösungsweg ist aber deutlich einfacher:

Die *Java*-Anwendung *Postgres2JSON.jar* wird aktualisiert und eine neue *Java*-Klasse hinzugefügt. Diese erstellt ein neues Sortierfeld aus einem ausgewählten Feld und verarbeitet den Inhalt. Dabei werden die Felder auf eine benutzerdefinierte Länge normalisiert und gegebenenfalls mit Nullen aufgefüllt, um eine natürliche Sortierung zu erreichen. Nachfolgend ein Beispiel, wie dieses Sortierfeld aussehen kann.

```
[Admiralbogen 45 80939 München → Admiralbogen 000045 080939 München]
```

Listing 32: Natürliche Sortierung (Natural sorting)

Hier wurde eine Länge von sechs Ziffern vorgegeben, d.h. die Hausnummer 45 mit zwei Ziffern wird um vier Nullen ergänzt, die Postleitzahl mit fünf Ziffern bekommt eine Null vorangestellt. Die Sortierung erfolgt zwar nach diesen Sortierfeldern, in den Ergebnissen werden diese jedoch nicht gezeigt.

6.2 ElasticAdress-API

Für die bisherige Arbeit war *Kibana Dev-Tools* sehr gut geeignet, um die Suche vorzubereiten, allgemeine Suchanfragen und die Sortierung zu testen. Nun werden die im vorherigen Kapitel erzeugten Suchanfragen in Java implementiert und als API bereitgestellt. Die *ElasticAdress*-API stellt zwei Aufruf-Optionen bereit:

a) *by-ID-Query* mit zwei Parametern

b) *query_string*-Suche mit vier Parametern

Der Übersichtlichkeit halber wird in dieser Arbeit nur die Abfrage-Methode (2) ausführlich behandelt. Diese wird nachfolgend beschrieben:

Der erste Schritt besteht darin, eine Verbindung zu den *Elasticsearch*-Server herzustellen, indem anfangs der *Elasticsearch-Client* – der so genannte *Transport-Client* – verwendet wird (vgl. Listing 33).

```
final SearchRequestBuilder builder = clientService.getClient().prepareSearch()
        .setIndices(ES_ADDRESS_INDEX)
        .setTypes(ES_ADDRESS_DOCTYPE)
        .addSort(SORT_FIELD, SortOrder.ASC)
        .setQuery(QueryBuilders.queryStringQuery(optimizedQueryString)
                        .fields(SEARCH_FIELDS).defaultOperator(Operator.AND))
        .setFetchSource(FETCH_SOURCE, new String[]{})
        .highlighter(new HighlightBuilder().field(HIGHLIGHT_FIELD))
        .setSize(FETCH_SIZE);

GeoBoundsAggregationBuilder geoSubAggregation = AggregationBuilders
        .geoBounds("bounds").field(BOUNDINGBOX_FIELD);


AGG_LEVEL.stream().forEach((aggregationLevel) -> {
    TermsAggregationBuilder myAggBuilder = AggregationBuilders
                .terms(aggregationLevel).field(aggregationLevel).size(LIMIT)
                .subAggregation(geoSubAggregation);
    builder.addAggregation(myAggBuilder.subAggregation(
            AggregationBuilders.topHits("Top_Hits").size(1)
                                .fetchSource(aggregationLevel.split("\\.", 2)[0], "")
                                .highlighter(new HighlightBuilder()
                                .field(aggregationLevel.split("\\.", 2)[0]))
    ));
});
// Ausführen der Suche
final SearchResponse searchResponse = builder.execute().actionGet();
```

Listing 33: Verbindung über den Transport-Client

Die Anwendung akzeptiert vier Variablen:

- **Suchanfrage** (als *String*): entspricht der Benutzereingabe
- **SRID** (als *Integer*): Angabe des Koordinatensystems, Standard ist das Gauß-Krüger-Koordiantensystem
- **Filter** (als *String*): bestimmt die Rückgabe-Art – entweder nur Adressen oder Adressen und Aggregationen
- ***Fuzziness*** (als *Boolean - Optional*): gibt an, ob die Suchanfrage mit Unschärfe berücksichtigt wird oder nicht

Nach Eingabe der Variablen führt die Anwendung folgenden Schritte aus:

- Optimierung der q*uery_string*
- Zusammensetzen der *Query* in *Java* und Senden an den *Elasticsearch-Clusters*
- Auswertung und Veredelung der *Elasticsearch*-Antwort

Die Suchanfrage sollte optimiert werden, indem man alle Abkürzungen und Fehler löscht, und Wildcard nur zu den Termen addiert, die ausschließlich aus Buchstaben bestehen.

Wenn der Benutzer zusätzlich ein *Fuzziness* haben will, wird die Suchanfrage zweimal bearbeitet. Der Grund für diese Aufteilung ist, dass *Elasticsearch* die Unschärfe und die *Wildcard* nicht zusammen, sondern nur separat verarbeiten kann (vgl. Listing 34, 35 und 36).

```
public static String prepareQueryString(final String queryString, boolean fuzzy) {
    String[] queryParts = queryString.toLowerCase().split(" ");
    StringBuilder sb = new StringBuilder();
    sb.append("(");
    for (String part : queryParts) {
        sb.append(getOptimizedPart(part, "wildcard"));// füge eine Wildcard hinzu
        sb.append(" ");
    }
    sb.setLength(sb.length() - 1);
    if (fuzzy) {// Falls Unschärfe erwünscht
        sb.append(") OR (");
        for (String part : queryParts) {
            sb.append(getOptimizedPart(part, "fuzzy"));// füge eine Unschärfe hinzu.
            sb.append(" ");
        }
        sb.setLength(sb.length() - 1);
    }
    sb.append(")");
    return StringUtils.normalizeSpace(sb.toString());
}
```

Listing 34: Suchabfrageoptimierung – Teil 1

```
private static String getOptimizedPart(String part, String type, boolean ignoreLength) {
   String optimizedPart = part.toLowerCase();
   if (optimizedPart.startsWith("d-")) {
     optimizedPart = optimizedPart.replace("d-", "");
   }
   optimizedPart = optimizedPart.replaceAll("[^A-Za-z0-9äöüß/ .,-]", "");
   optimizedPart = optimizedPart.replace("deutschland", "");
   if (optimizedPart.length() > 4) {
     if (optimizedPart.endsWith("str")) {
       optimizedPart = optimizedPart.replace("str", "straße");
     }
     if (optimizedPart.endsWith("str.")) {
       optimizedPart = optimizedPart.replace("str.", "straße");
     }
   }
   optimizedPart = optimizedPart.trim();
   optimizedPart = QueryParser.escape(optimizedPart);
   if (optimizedPart.matches(".*\\d+.*") && optimizedPart.matches(".*[a-zA-Z].*")) {
     optimizedPart = optimizedPart.replaceAll("(?<=\\D)(?=\\d)|(?<=\\d)(?=\\D)", " ");
   }
   if (optimizedPart.matches(".*[.,-].*")) {
     String[] parts = optimizedPart.split("[.,-]");
     String tempString = "";
     for (String p : parts) {
       tempString += getOptimizedPart(p, type, true) + " ";
     }
     optimizedPart = tempString.trim();
   } else {
     if (optimizedPart.length() >= 2 || ignoreLength) {
       if (!optimizedPart.isEmpty()) {
         if (!optimizedPart.matches(".*\\d+.*")) {
           if (type.equalsIgnoreCase("fuzzy")) {
             optimizedPart = optimizedPart + "~2";
           } else if (type.equalsIgnoreCase("wildcard")) {
             optimizedPart = optimizedPart + "*";
           }
         }
       }
     }
   }
   return optimizedPart;
 }
```

Listing 35: Suchabfrageoptimierung – Teil 2

Danach versucht die Anwendung herauszufinden, welche Terme der Suchanfrage der Postleitzahl bzw. Hausnummer entsprechen. Sie werden ignoriert und nicht mit einer *Wildcard* oder *Fuzziness* verarbeitet, Außerdem werden Hausnummern und der Hausnummern-Zusatz durch Leerzeichen getrennt (vgl. Listing 36 - Test 2) und wenn ein "-" zwischen zwei Terme steht, bekommen beide Terme eine *Wildcard* oder *Fuzziness* (vgl. Listing 36 - Test 3) und der Bindestrich wird gelöscht. Anschließend folgt die Änderung aller Buchstaben der Terme in Kleinbuchstaben.

```
Suchanfrage                                  Optimierte Suchanfrage
Test: 1 - richard w                          -> (richard* w) OR (richard~2 w)
Test: 2 - richard 15a                        -> (richard* 15 a) OR (richard~2 15 a)
Test: 3 - Prälat-Wel                         -> (prälat* wel*) OR (prälat~2 wel~2)
Test: 4 - Admiralbogen 45 80939 München      -> (admiralbogen* 45 80939 münchen*) OR
                                                (admiralbogen~2 45 80939 münchen~2)
```

Listing 36: Beispiel für Suchanfragenoptimierung

Außerdem wird ein bestimmter Grenzwert festgelegt, der die Anzahl der zurückgegebenen Antworten bestimmt. Die Antwort enthält neben den Adressen und Aggregationen auch weitere Informationen wie beispielsweise das Sortierungsfeld, die Bearbeitungszeit oder den *Shard*-Status. Um den Anwender nicht zu verwirren, sollte dieser aber nur die Adressen oder Aggregationen nach dem angegebenen Grenzwert zurückerhalten. Dabei werden folgende Möglichkeiten unterschieden:

- wenn die Filter-Variable auf „Adresse“ gesetzt ist, kommen nur Adressen-Ergebnisse sortiert zurück
- wenn die Filter-Variable leer ist, wird nach der Treffer-Anzahl für Suchergebnisse und dem festgelegten Limit entschieden

Jedes Suchergebnis hat eine Anzahl von Adressen und Aggregationen (Treffer-Anzahl), die zurückgegeben wird. Diese Anzahl wird in der Anwendung mit dem festgelegten Limit verglichen; wenn die Treffer-Anzahl kleiner ist als das Limit, werden Adressenergebnisse sortiert zurückgegeben bis das Limit erreicht ist. Andernfalls – wenn die Treffer-Anzahl größer ist als das Limit – werden die Aggregations-ergebnisse zurückgegeben bis das Limit erreicht ist.

Jede der drei Aggregationen besteht aus *Buckets* und Sub-Aggregationen. In den *Buckets* befinden sich die *keys* und die *doc_counts*. Alle *doc_counts* werden nach der Anzahl ihrer Treffer sortiert und entsprechend dem angegebenen Limit in einem *Array* zwischengespeichert (bei einem Limit von 10, werden nur die erste zehn Aggregate zurückgegeben).

Anschließend wird geschaut, ob die Suchanfrage in einem *Aggregations-Key* auftaucht. Falls ja, werden alle andern aus dem *Array* gelöscht.

Wenn das *Array* nach diesem Schritt leer ist, dann ist folgendes Problem aufgetreten: die Suchanfrage enthält einen Tippfehler und kann damit niemals einem *Aggregations-Key* entsprechen. Wie bereits erwähnt, hat jede Aggregation zwei Sub-Aggregationen. Die Lösung für das genannte Problem ist, bei solchen *Top Hits Aggregations* zu überprüfen, ob sie eine *Highlight*-Option besitzen und diese zurückgeben. Der Vorteil von *Top Hits Aggregations* ist die Unterstützung des *Highlighting* pro Treffer. Das bedeutet, falls die Suchabfrage einen kleinen Fehler hat, wird bei *Elasticsearch* intern alle möglichen Varianten der Suchabfrage nach der *Levenshtein Distance* berechnet und alle mögliche Kombinationen mit *HTML-Strips* markiert. Das folgende Beispiel soll dies verdeutlichen.

```
Die Suchanfrage "Hau<b>tp</b>straße~2" hat ca. 73000 Treffer, hier fällt die Wahl auf die
Aggregationen; in agg_Straße ist die Highlight-Option gesetzt:

"highlight": {
    "agg_strasse": [
        "<em>Hauptstraße</em> (Miltenberg)"
    ]
}
Die Suchanfrage wurde mithilfe der Fuzziness-Funktion korrigiert. Dies wird zurückgegeben. Der
Rest der Highlight-Option ist leer, von daher werden sie ignoriert und nicht zurückgegeben.
```

Listing 37: Highlighting in Top Hits Aggregations

Die Antwort – egal ob von den Adressen oder von den Aggregaten – kommt in der folgenden Form:


```
{
 "results" : [ {
  "id" : "Y9VzBGIBLMu9bhAv3wJf",
  "attrs" : {
   "bbox" : [ 1292962.8701974219, 6141867.423843072, 1292962.8701974219, 6141867.423843072 ],
   "x" : 6141867.423843072,
   "y" : 1292962.8701974219,
   "label" : "<em>Admiralbogen</em> <em>45</em> 80939 München - Schwabing-Freimann"
  }
 } ],
 "otherDocs" : 0
}
```


Listing 38: Antwort Form

In der nachfolgenden Tabelle wird für beide Fälle (Adresse und Aggregat) der Inhalt der Antwort genauer erläutert.

Ausgabe	Adresse	Aggregate
ID	Die Indexoperation wurde ohne Angabe einer ID ausgeführt. In diesem Fall wurde automa--tisch eine ID für die Adresse generiert.	Für Aggregate gibt es keine ID.
Bounding-Box	Dabei handelt es sich nicht um eine echte Bounding-Box, sondern um einen Punkt mit der entsprechenden Adresse.	Aus allen Dokumenten in den Aggregaten wird eine Bounding-Box nach den minimalen Koordinaten und den maximalen Koordinaten gebildet, die darin enthalten sind.
X,Y Koordinaten	X und Y Koordinate der Adresse	der Mittelwert der maximalen X- und Y-Koordinaten
Label	der gesuchte Begriff vervollständigt	der gesuchte Begriff vervollständigt
otherDocs	Anzahl der nicht angezeigten Dokumente (abhängig von der Treffer-Anzahl und dem Limit)	

Tabelle 5: Rückgabe-Form – erläutert für Adresse und Aggregat

Die Suche nach einer ID liefert Detailinformationen zu einer Adresse, die an diese ID gekoppelt ist (vgl. Listing 39).


```
Id: Y9VzBGIBLMu9bhAv3wJf, Gewuenschtes SRID: 4326
{
 "land" : "Bayern",
 "bezirk" : "Oberbayern",
 "kreis" : "München (Stadt)",
 "gemeinde" : "München",
 "gemeindeteil" : "München",
 "strasse" : "Admiralbogen",
 "hausnummer" : "45",
 "hausnummer_zusatz" : "",
 "postleitzahl" : 80939,
 "postort" : "München",
 "postortsteil" : "Schwabing-Freimann",
 "postort_zusatz" : "",
 "label" : "Admiralbogen 45 80939 München - Schwabing-
Freimann",
 "x" : 11.61488308097389,
 "y" : 48.21002809393404
}
```


Listing 39: ID-Suche

Diese ID ändert sich bei jeder Indizierung, da sie von *Elasticsearch* automatisch generiert wird. Deswegen werden normalerweise zuerst die Adressen abgerufen und dann die ID der Antwort für Detailinformationen verwendet.

6.3 REST-Endpoint

Um die *ElasticAddress*-API über HTTP verfügbar zu machen, wird ein *REST-Endpoint* auf Basis von *Jersey* entwickelt.[88] Der *REST-Endpoint* beinhaltet eine termbasierte Authentifizierung, um den Zugriff zu schützen und Zugriffsstatistiken zu gewinnen. Der Nutzerkreis beschränkt sich auf Softwareentwickler, welche den Dienst in ihre Anwendung einbauen. Eine dieser Anwendungen ist der *BayernAtlas*. Für die Dokumentation dieses *Endpoints* wird *Swagger* verwendet.

Der Entwickler benutzt das *Swagger Interface* zum Testen und Ausprobieren des REST-Endpoints.[89] Er stellt die beiden in der *ElasticAdress*-API programmierten Funktionen – Suche nach der ID und Suche nach einer Adresse – bereit. *D*as Suchergebnis kann in den Formaten XML-Text und JSON angefordert werden.

6.4 Weitere Bestandteile der Softwareentwicklung

Neben der reinen Problemlösung wurden zusätzliche Einblicke in weitere Entwicklungsarbeiten ermöglicht, die zwar von außen nicht sichtbar, jedoch sehr wichtig sind.

6.4.1 Logging

Die Anwendung hat zwei zusätzliche wichtige Java-Bibliotheken verwendet, nämlich *SLF4J (Simple Logging Facade for Java*) und *JUnit*. *SLF4J* ist eine generalisierte *Logging*-Schnittstelle. Das *Loggen* oder Protokollieren von Informationen über bestimmte Programmzustände ist sehr wichtig, um künftig den Ablauf modifizieren, analysieren und verstehen zu können. Mit der *Logging*-Schnittstelle lassen sich Nachrichten, die in fünf verschiedene Dringlichkeitsstufen (nämlich *Error*, *Warn*, *Info*, *Debug* und *Trace*) eingeteilt werden können, auf der Konsole oder einem externen Speicher mittels *XML*- oder *TXT*-Dateien protokollieren.[90][91] Das nächste Beispiel zeigt die *Error*-Stufe und die *Debug*-Stufe.

```
        try {
            serverPort = Integer.parseInt(serverString.split(":")[1]);
        } catch (NumberFormatException nfx) {
            Log.error(nfx.getMessage(), nfx);
            throw new ServiceException("Port must be an Integer: " + serverStromg, nfx);
        }

        for (String server : serverList) {
            String serverName = server.split(":")[0];
            int port = Integer.parseInt(server.split(":")[1]);
            Log.debug("adding HttpHost: " + serverName + ":" + port);
            list.add(new HttpHost(serverName, port, "http"));
        }
```

Listing 40: SLF4J – Error-Stufe und Debug-Stufe

6.4.2 Unit-Tests

Bei *JUnit* handelt es sich um ein *Framework* für *Unit*-Tests in *Java*. Diese Tests werden als Methoden in Klassen geschrieben, die automatisch laufen. Sie zeigen, ob bestimmte Programmteile sich so verhalten wie es erwartet wird. Ein Test kennt nur zwei Ergebnisse: entweder er gelingt oder nicht. Im Anschluss liefert das *Framework* den Status aller Tests zurück. Testmethoden werden durch die *JUnit* Annotation *@Test* aufgerufen.[92] Das nächste Beispiel zeigt eine Testmethode.

```
@Test
  public void testById() throws ServiceException, JsonProcessingException, IOException {
    ElasticSearchLocationService esls = new ElasticSearchLocationService();
    LocationSearchResults results = esls.getLocations("Admiralbogen 45", 4326, "address", true);
    mapper.writerWithDefaultPrettyPrinter().writeValue(System.out, results);
    String id = results.getResults().get(0).getId();
    Location l = esls.getLocationById(id, 4326);
    System.out.println("by Id: " + id);
    mapper.writerWithDefaultPrettyPrinter().writeValue(System.out, l);
  }
```

Ergebnisse:

```
{
  "results" : [ {
    "id" : "t7suRGIBu2-sD5tdWsa5",
    "attrs" : {
      "bbox" : [ 11.61488308097389, 48.21002809393404, 11.61488308097389, 48.21002809393404 ],
      "x" : 48.21002809393404,
      "y" : 11.61488308097389,
      "label" : "<b>Admiralbogen</b> <b>45</b> 80939 München , Schwabing-Freimann"
    }
  } ],
  "otherDocs" : 0
}
```

by Id: t7suRGIBu2-sD5tdWsa5

```
{
  "land" : "Bayern",
  "bezirk" : "Oberbayern",
  "kreis" : "München (Stadt)",
  "gemeinde" : "München",
  "gemeindeteil" : "München",
  "strasse" : "Admiralbogen",
  "hausnummer" : "45",
  "hausnummer_zusatz" : "",
  "postleitzahl" : 80939,
  "postort" : "München",
  "postortsteil" : "Schwabing-Freimann",
  "postort_zusatz" : "",
  "label" : "Admiralbogen 45 80939 München , Schwabing-Freimann",
  "x" : 11.61488308097389,
  "y" : 48.21002809393404
}
```

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

Listing 41: JUnit – Test und Ergebnisse.

6.4.3 Upgrade auf neuere Version

Seit *Elasticsearch* 6.0.0-beta1 gibt es den sogenannten *Java-High-Level-REST-Client. Wie schon in Kapitel 3.8.2 erwähnt, hängt dieser Client vom Elasticsearch-Kernobjekt ab, und akzeptiert die gleichen Argumente wie der native Transport-Client.* Er *gibt auch die gleichen Objekte zurück. Der neue Java-High-Level-REST-Client ist vorwärtskompatibel*, was bedeutet, dass er die Kommunikation mit späteren Versionen von *Elasticsearch* unterstützt. Im Vergleich zum nativen *Transport-Client* ist der *Java-High-Level-REST-Client* viel schneller. Er arbeitet genau wie ein *HTML-Client* über den Port 9200.[93][94]. *Das nächste Beispiel zeigt die query_string-Abfrage mit dem High-Level-REST-Client.*

```
    private static SearchRequest buildSearchRequest(String optimizedQueryString) {
        SearchRequest sr = new SearchRequest(ES_ADDRESS_INDEX).types(ES_ADDRESS_DOCTYPE);
        SearchSourceBuilder ssb = new SearchSourceBuilder();
        ssb.query(QueryBuilders.queryStringQuery(optimizedQueryString)
                .fields(SEARCH_FIELDS).defaultOperator(Operator.AND));
        ssb.sort(SORT_FIELD, SortOrder.ASC);
        ssb.fetchSource(FETCH_SOURCE, new String[]{});
        ssb.highlighter(new HighlightBuilder().field(HIGHLIGHT_FIELD)
                .preTags("<b>").postTags("</b>"));
        ssb.size(LIMIT);
        GeoBoundsAggregationBuilder geoSubAggregation = AggregationBuilders
                .geoBounds("bounds").field(BOUNDINGBOX_FIELD);
        AGG_LEVEL.stream().forEach((aggregationLevel) -> {
            TermsAggregationBuilder myAggBuilder = AggregationBuilders.
                    terms(aggregationLevel).field(aggregationLevel).size(LIMIT)
                    .subAggregation(geoSubAggregation);
            ssb.aggregation(myAggBuilder.subAggregation(
                    AggregationBuilders.topHits("Top_Hits").size(1).
                            fetchSource(aggregationLevel.split("\\.", 2)[0], "")
                            .highlighter(new HighlightBuilder().preTags("<b>")
                                    .postTags("</b>")
                                    .field(aggregationLevel.split("\\.", 2)[0]))
            ));
        });
        sr.source(ssb);
        return sr;
    }
     public LocationSearchResults getLocations(final String search,
         final Integer inputSrid, final String filter,
         final boolean fuzzy) throws ServiceException {
       SearchRequest sr = buildSearchRequest(optimizedQueryString);
       final SearchResponse searchResponse;
        try (RestHighLevelClient client = clientService.createClient()) {
            searchResponse = client.search(sr);
        } catch (IOException ex) {
            throw new ServiceException(ex);
        }
       return parseElasticSearchResponse(searchResponse, search, addressOnly, srid);
    }
```

Listing 42: query_string-Abfrage mit dem High-Level-REST-Client.

7 Fazit und Ausblick

Der Hauptbestandteil dieser Forschungsarbeit bestand darin, die Arbeitsweise und den Funktionsumfang von *Elasticsearch* zu verstehen und aufbauend auf diesem Wissen Java-Applikationen zu programmieren, mithilfe derer die tägliche Aktualisierung der *Elasticsearch*-Server sowie die schnelle Adressen-Suchfunktion durchgeführt werden können.

Statistisch gesehen, gab es laut dem LDBV zwischen dem 09.03.2017 und dem 09.03.2018 ca. 58770 Änderungen in der ZAD-Datenbank (Adressen aktualisieren, neue Adressen addieren oder Adressen löschen). Rechnet man dies auf die 73 ADBVs (51 Ämter mit ihren 22 Außenstellen) in Bayern herunter, so kommt man auf ca. drei Änderungen pro Arbeitstag und Amt. Diese Änderungen oder Aktualisierungen werden täglich vom Bash-Skript automatisch übernommen. Der gesamte Prozess dauert ca. 20 Minuten. Die Hälfte der Zeit wird dabei für das Importieren der *JSON*-Dateien benötigt, die anderen zehn Minuten für das Indexieren auf den *Elasticsearch*-Server.

Als Ergebnis dieser Arbeit steht also ein System, welches auf Basis von *Elasticsearch* die Performance des Ortssuchdienst auf Adressen der Bayerischen Vermessungsverwaltung verbessert hat. Auch externe Nutzer außerhalb des LDBVs haben nun die Gelegenheit von dieser Arbeit zu profitieren, da der bisherige Adressensuchdienst hiermit modernisiert worden ist.

Ein weiterer Punkt dieser Arbeit war der Vergleich von *Elasticsearch* mit anderen Datenbankmanagementsystemen. Es ist zu erwarten, dass *Elasticsearch* auch weiterhin an Bedeutung gewinnen und aufgrund seiner vielen Vorteile andere Systeme (insbesondere relationale Datenbankmodelle) ergänzen wird.

Literaturverzeichnis

[1] Cisco. 2017. The Zettabyte Era: Trends and Analysis. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html.

[2] Wikipedia. 2017. Landesamt für Digitalisierung, Breitband und Vermessung Bayern. https://de.wikipedia.org/wiki/Landesamt_f%C3%BCr_Digitalisierung,_Breitband_und_Vermessung_Bayern.

[3] Dataversity Education, LLC. Keith D. Foote. 2017. A Brief History of Database Management. http://www.dataversity.net/brief-history-database-management/.

[4] Bill Vorhies. 2013. A brief history of big data technologies – from Sql to NoSql to hadoop and beyond. http://data-magnum.com/a-brief-history-of-big-data-technologies-from-sql-to-nosql-to-hadoop-and-beyond/.

[5] Wikipedia. 2018. Apache Lucene. https://en.wikipedia.org/wiki/Apache_Lucene.

[6] C.H.Beck. 2015. Ein Vergleich der Open-Source-Giganten Solr und Elasticsearch . https://becksche.de/Meldung?titel=07-01-2015-ein-vergleich-der-open-source-giganten-solr-und-elasticsearch.

[7] Chavi gupta. 2016. Advantages of Elasticsearch. https://www.3pillarglobal.com/insights/advantages-of-elastic-search.

[8] Clinton Gormley; Zachary Tong. 2015. Elasticsearch: The Definitive Guide. Teil 1, Kapitel 1: Document Oriented. Seite 9. O'Reilly Media.

[9] Elastic.co. 2018. Dynamic Mapping. https://www.elastic.co/guide/en/elasticsearch/reference/current/dynamic-mapping.html.

[10] Clinton Gormley; Zachary Tong. 2015. Elasticsearch: The Definitive Guide. Teil 1, Kapitel 2: Scale Horizontally. Seite 30-32. O'Reilly Media.

[11] Clinton Gormley; Zachary Tong. 2015. Elasticsearch: The Definitive Guide. Teil 1, Kapitel 11: Making Changes Persistent. Seite 161-163. O'Reilly Media.

[12] Neil Baker. 2017. Configurable and Extensible elasticsearch - Folie Nr.8. https://de.slideshare.net/NeilBaker18/elasticsearch-for-beginners.

[13] Elastic.co. 2018. Basic Concepts. https://www.elastic.co/guide/en/elasticsearch/reference/current/_basic_concepts.html#_type.

[14] Rafal Kuc; Marek Rogozinski. 2015. Mastering Elasticsearch. Kapitel 1. Seite 16-18. Packt Publishing.

[15] Abhishek Andhavarapu. 2017. Learning Elasticsearch. Kapitel 1 : Introduction to Elasticsearch. Seite 8 - 11. Packt Publishing.

[16] Clinton Gormley; Zachary Tong. 2015. Elasticsearch: The Definitive Guide. Teil 1, Kapitel 6: Mapping and Analysis. Seite 88. O'Reilly Media.

[17] Wikipedia. 2018. Universally Unique Identifier. https://de.wikipedia.org/wiki/Universally_Unique_Identifier.

[18] Elastic.co. 2018. Node types. https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-node.html.

[19] Clinton Gormley; Zachary Tong. 2015. Elasticsearch The Definitive Guide. Teil1, Kapitel 1: Indexing Employee Documents. Seite 11. O'Reilly Media.

[20] Clinton Gormley; Zachary Tong. 2015. Elasticsearch: The Definitive Guide. Teil 1, Kapitel 6: Inverted Index. Seite 81-83. O'Reilly Media.

[21] Imtiaz Ahmed. 2018. Complete Elasticsearch Masterclass with Logstash and Kibana. Teil 3, Vorlesung 8 + 9: Index Settings and Mappings. 41:51 Min. udemy.com.

[22] Clinton Gormley; Zachary Tong. 2015. ElasticsearchThe Definitive Guide. Teil1, Kapitel 10: Index Settings. Seite 132. O'Reilly Media.

[23] Florian Hopf. 2015. Elasticsearch: Ein praktischer Einstieg. Kapitel 3 - Textinhalte auffindbar machen. Seite 38, Abbildung 3.1. dpunkt.verlag GmbH.

[24] Clinton Gormley; Zachary Tong. 2015. Elasticsearch: The Definitive Guide. Kapitel 8 - Analysis and Analyzers. Seite 84. O'Reilly Media.

[25] Clinton Gormley; Zachary Tong. 2015. Elasticsearch: The Definitive Guide. Teil 1, Kapitel 6: Analysis and Analyzers. Seite 84. O'Reilly Media.

[26] Elastic.co. 2018. Lowercase Token Filter. https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-lowercase-tokenfilter.html.

[27] Elastic.co. 2018. Stop Token Filter. https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-stop-tokenfilter.html.

[28] Elastic.co. 2018. Mapping Type. https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping.html#mapping-type.

[29] Clinton Gormley; Zachary Tong. 2015. Elasticsearch: The Definitive Guide. Teil 1, Kapitel 6: Mapping. Seite 88. O'Reilly Media.

[30] Radu Gheorghe; Matthew Lee Hinman; Roy Russo. 2015. Elasticsearch in Action. Teil1, Kapitel 1.3: Indexing, updating, and deleting data. Seite 57 - 58. Manning Publications.

[31] Abhishek Andhavarapu. 2017. Learning Elasticsearch. Kapitel 5 - Organizing your data. Seite 144 - 145. Packt Publishing.

[32] Clinton Gormley; Zachary Tong. 2015. Elasticsearch: The Definitive Guide. Kapitel 2: Life Inside a Cluster. Seite 25 - 33. O'Reilly Media.

[33] Imtiaz Ahmad. 2018. Complete elasticsearch masterclass with logstash and kibana. Teil 2 Vorlesung 6: Distributed Execution of Requests. udemy.com.

[34] Florian Hopf. 2015. Elasticsearch, Ein praktischer Einstieg. Kapitel 2: Der Invertierte Index und Kapitel 3: Textinhalte auffindbar machen. Seite 16 und Seite 37 - 38. dpunkt.verlag GmbH.

[35] Elasticsearch; Alex Brasetvik. 2013. Elasticsearch from the Bottom Up. https://www.elastic.co/blog/found-elasticsearch-from-the-bottom-up.

[36] Abhishek Andhavarapu. 2017. Learning Elasticsearch. Kapitel 6: Different types of queries. Seite 156 - 161. Packt Publishing.

[37] Imtiaz Ahmad. 2018. Complete Elasticsearch Masterclass with Logstash and Kibana. Teil 2, Vorlesung 7: Text Analysis for Indexing and Searching. . udemy.com.

[38] Elastic.co. 2018. Query and filter context. https://www.elastic.co/guide/en/elasticsearch/reference/current/query-filter-context.html.

[39] Elastic.co. 2018. Match All Query. https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-match-all-query.html#query-dsl-match-all-query.

[40] Clinton Gormley; Zachary Tong. 2015. Elasticsearch: The Definitive Guide. Part 2, Kapitel 13: Full-Text Search. 197 - 198. O'Reilly Media.

[41] Elastic.co. 2018. Full text queries. https://www.elastic.co/guide/en/elasticsearch/reference/current/full-text-queries.html.

[42] Abhishek Andhavarapu. 2017. Learning Elasticsearch. Kapitel 6: Term versus Match query. Seite 174. Packt Publishing.

[43] Anna Roes. 2015. Recommendersysteme mit Elasticsearch. 3.2.2 Volltextsuche. Seite 46. inovex GmbH. https://www.inovex.de/fileadmin/files/Fachartikel_Publikationen/Theses/empfehlungsgenerierung-mit-elasicsearch-anna-roes-03-2015.pdf.

[44] Radu Gheorghe; Matthew Lee Hinman; Roy Russo. 2015. Elasticsearch in Action . Part 1, Kapitel 4 - Query_String Query. Seite 96-98. Manning Publications.

[45] Elastic.co. 2018. Query String Query . https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html.

[46] Wikipedia. 2018. Levenshtein Distanz. https://de.wikipedia.org/wiki/Levenshtein-Distanz.

[47] Elastic.co. 2018. Fuzziness. https://www.elastic.co/guide/en/elasticsearch/reference/current/common-options.html#fuzziness.

[48] Clinton Gormley; Zachary Tong. 2015. Elasticsearch: The Definitive Guide. Teil2, Kapitel 12: Structured Search. 173 - 179. O'Reilly Media.

[49] Elastic.co. 2018. Term level queries. https://www.elastic.co/guide/en/elasticsearch/reference/current/term-level-queries.html.

[50] Anna Roes. 2015. Recommendersysteme mit Elasticsearch. 3.2.1 Strukturierte Suche. Seite 45. inovex GmbH. https://www.inovex.de/fileadmin/files/Fachartikel_Publikationen/Theses/empfehlungsgenerierung-mit-elasicsearch-anna-roes-03-2015.pdf.

[51] Radu Gheorghe; Matthew Lee Hinman; Roy Russo. 2015. Elasticsearch in Action. Appendix C: Highlighting. Seite 390. Manning Publications.

[52] Alberto Paro. 2015. Elasticsearch CookBook. Kapitel 6: Aggregations. 195 - 196. Packt Publishing.

[53] Abhishek Andhavarapu. 2017. Learning Elasticsearch. Kapitel 8: Types of aggregations. Seite 287 - 288. Packt Publishing.

[54] Elastic.co. 2018. Aggregations - Buckets and Metrics. https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations.html#search-aggregations.

[55] Elastic.co. 2018. Geo Centroid Aggregation. https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-metrics-geobounds-aggregation.html.

[56] Elastic.co. 2018. Top Hits Aggregation. https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-metrics-top-hits-aggregation.html.

[57] Clinton Gormley; Zachary Tong. 2015. Elasticsearch: The Definitive Guide. Teil1, Kapitel 1: RESTful API with JSON over HTTP. Seite 6 - 8. O'Reilly Media.

[58] Rafal Kuc; Marek Rogozinski. 2015. Mastering Elasticsearch. Kapitel 1: Communicating with Elasticsearch. Seite 21. Packt Publishing.

[59] Vineeth Mohan. 2015. Elasticsearch Blueprints. Kapitel 1: Communicating with the Elasticsearchserver. Seite 3. Packt Publishing.

[60] Elastic.co. 2018. Transport Client. https://www.elastic.co/guide/en/elasticsearch/client/java-api/current/transport-client.html.

[61] Alberto Paro. 2015. Elasticsearch Cookbook. Kapitel 10: Java Integration. Seite 329 - 335. Packt Publishing.

[62] Wikipedia. 2017. Round Robin. https://de.wikipedia.org/wiki/Round_Robin_(Informatik).

[63] Wikipedia. 2018. cURL. https://de.wikipedia.org/wiki/CURL.

[64] Curl.haxx. 2018. cURL Options. https://curl.haxx.se/docs/manpage.html.

[65] Abhishek Andhavarapu. 2017. Learning Elasticsearch. Kapitel 2: Kibana Console. Seite 44 - 45. Packt Publishing.

[66] DB-Engines: solid IT. 2018. System Properties Comparison Elasticsearch vs. PostgreSQL vs. Solr vs. Splunk. https://db-engines.com/en/system/Elasticsearch%3BPostgreSQL%3BSolr%3BSplunk.

[67] DB-Engines: solid IT. 2018. DB-Engines Ranking - Trend of Elasticsearch vs. PostgreSQL vs. Solr vs. Splunk Popularity. https://db-engines.com/en/ranking_trend/system/Elasticsearch%3BPostgreSQL%3BSolr%3BSplunk.

[68] Google. 2004 - Heute. Interesse im zeitlichen Verlauf. https://trends.google.com/trends/explore?date=all&q=solr,lucene,elasticsearch,postgreSQL.

[69] Patrick Oscity. 2015. Wer sucht, der findet - Volltextsuche mit PostgreSQL und Elasticsearch. https://www.zweitag.de/de/blog/technologie/wer-sucht-der-findet-volltextsuche-mit-postgresql-und-elasticsearch.

[70] Wikipedia. 2018. Docker . https://de.wikipedia.org/wiki/Docker_(Software).

[71] LineFeed: Stephan Augsten. 2017. Was sind Docker-Container?. https://www.dev-insider.de/was-sind-docker-container-a-597762/.

[72] Microsoft. 2017. Views. https://docs.microsoft.com/de-de/sql/relational-databases/views/views.

[73] Ldbv. 2018. Transformation von Geofachdaten. https://www.ldbv.bayern.de/vermessung/utm_umstellung/trans_geofach.html.

[74] Postgis. 2010. Using PostGIS: Data Management and Queries. http://postgis.refractions.net/documentation/manual-1.4/ch04.html.

[75] OGC. 2018. OGC. http://www.opengeospatial.org/.

[76] Proj: Frank Warmerdam. 2001. proj.4. http://proj4.org/.

[77] Elastic.co. 2018. Bulk API. https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-bulk.html.

[78] Commons. 2017. Commons CLI. https://commons.apache.org/proper/commons-cli/.

[79] Mark Davis; Laurențiu Iancu. 2017. Basierend auf dem Unicode-Textsegmentierungsalgorithmus, wie in Unicode-Standard, Anhang 29 spezifiziert.. http://unicode.org/reports/tr29/.

[80] Ranks.nl. 2018. German Stopwords . https://www.ranks.nl/stopwords/german.

[81] Snowball. 2018. German stemming algorithm. http://snowball.tartarus.org/algorithms/german/stemmer.html.

[82] Apache Lucene. 2014. GermanNormalizationFilter. http://lucene.apache.org/core/4_9_0/analyzers-common/org/apache/lucene/analysis/de/GermanNormalizationFilter.html.

[83] Elastic.co. 2018. Unique Token Filter. https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-unique-tokenfilter.html.

[84] Stedolan. 2015. JQ, command-line JSON processor. https://stedolan.github.io/jq/.

[85] Computer Hope. 2017. Linux time command. https://www.computerhope.com/unix/utime.htm.

[86] David Koelle. 2010. The Alphanum Algorithm. http://www.davekoelle.com/alphanum.html.

[87] Ascii. 2018. ASCII Tabelle. http://www.asciitable.com/.

[88] Oracle Corporation. 2017. RESTful Web Services in Java. https://jersey.github.io/.

[89] SmartBear Software. 2018. Swagger. https://swagger.io/.

[90] QOS.ch. 2017. Simple Logging Facade for Java. https://www.slf4j.org/.

[91] Christian Ullenboom. 2017. Java 7 - Mehr als eine Insel - Logging und Monitoring. Kapitel 20. Online Buch http://openbook.rheinwerk-verlag.de/java7/1507_20_001.html. Rheinwerk Verlag GmbH.

[92] Fernuni-hagen. 2010. Einführung in JUnit. https://wiki.fernuni-hagen.de/eclipse/index.php/Einf%C3%BChrung_in_JUnit.

[93] Elastic.co. 2018. Java High Level REST Client. https://www.elastic.co/guide/en/elasticsearch/client/java-rest/6.2/java-rest-high.html.

[94] Elastic.co. 2018. Java High Level REST Client Compatibility. https://www.elastic.co/guide/en/elasticsearch/client/java-rest/6.2/java-rest-high-compatibility.html.

Anhänge

Anhang 1: Die ersten 20 Zeilen der PostgreSQL-Tabelle.

#	land	bezirk	kreis	gemeinde	gemeindeteil	strasse	hausnummer	hau	postleitzahl	postort	postortsteil
1	Bayern	Oberbayern	Garmisch-Partenkirchen	Murnau a.Staffelsee	Murnau a.Staffelsee	Kocheler Straße	26		82418	Murnau	Murnau
2	Bayern	Oberbayern	Garmisch-Partenkirchen	Murnau a.Staffelsee	Murnau a.Staffelsee	Kocheler Straße	26	a	82418	Murnau	Murnau
3	Bayern	Oberbayern	Garmisch-Partenkirchen	Murnau a.Staffelsee	Weindorf	Dorfstraße	40		82418	Murnau	Weindorf
4	Bayern	Oberpfalz	Neumarkt i.d.OPf.	Lauterhofen	Ballertshofen	Ballertshofen	78		92283	Lauterhofen	Ballertshofen
5	Bayern	Oberpfalz	Neumarkt i.d.OPf.	Velburg	Lengenfeld	Ostermühlweg	6		92355	Velburg	Lengenfeld
6	Bayern	Oberpfalz	Neumarkt i.d.OPf.	Dietfurt a.d.Altmühl	Dietfurt a.d.Altmühl	Weinbergstraße	6	a	92345	Dietfurt	Dietfurt
7	Bayern	Oberbayern	München	Kirchheim b.München	Heimstetten	Weißenfelder Straße	6		85551	Kirchheim	Heimstetten
8	Bayern	Oberbayern	München	Kirchheim b.München	Heimstetten	Weißenfelder Straße	8		85551	Kirchheim	Heimstetten
9	Bayern	Oberbayern	München	Kirchheim b.München	Heimstetten	Weißenfelder Straße	1		85551	Kirchheim	Heimstetten
10	Bayern	Oberbayern	München	Kirchheim b.München	Heimstetten	Seestraße	1		85551	Kirchheim	Heimstetten
11	Bayern	Oberbayern	München	Kirchheim b.München	Heimstetten	Seestraße	1	a	85551	Kirchheim	Heimstetten
12	Bayern	Oberbayern	Ebersberg	Egmating	Egmating	Münchener Straße	14	a	85658	Egmating	Egmating
13	Bayern	Mittelfranken	Nürnberg (Stadt)	Nürnberg	Nürnberg	Parkstraße	11		90409	Nürnberg	Maxfeld
14	Bayern	Oberbayern	München	Kirchheim b.München	Heimstetten	Hürderstraße	4		85551	Kirchheim	Heimstetten
15	Bayern	Oberbayern	München	Kirchheim b.München	Heimstetten	Sonnenallee	1		85551	Kirchheim	Heimstetten
16	Bayern	Oberbayern	München	Kirchheim b.München	Heimstetten	Hürderstraße	6		85551	Kirchheim	Heimstetten
17	Bayern	Oberbayern	München	Kirchheim b.München	Heimstetten	Hürderstraße	1		85551	Kirchheim	Heimstetten
18	Bayern	Oberbayern	München	Kirchheim b.München	Heimstetten	Sonnenallee	2		85551	Kirchheim	Heimstetten
19	Bayern	Oberbayern	München	Kirchheim b.München	Heimstetten	Hürderstraße	5		85551	Kirchheim	Heimstetten
20	Bayern	Oberbayern	München	Kirchheim b.München	Heimstetten	Marsstraße	1	a	85551	Kirchheim	Heimstetten

Abbildung 16: Anhang 1.1: PostgreSQL-Tabelle

postort_zusatz	kubatur	agg_gemeinde	agg_postort	agg_strasse	wkt	pointwkt	gdeart	needs_postfix
a. Staffelsee	0.0	Markt Murnau a.St...	82418 Murnau...	Kocheler Straße (Murna...	POINT(4440545.5 5281...	<NULL>	Markt	☐
a. Staffelsee	0.0	Markt Murnau a.St...	82418 Murnau...	Kocheler Straße (Murna...	POINT(4440553.5 5281...	<NULL>	Markt	☐
a. Staffelsee	0.0	Markt Murnau a.St...	82418 Murnau...	Dorfstraße (Murnau a.S...	POINT(4440884.5 5282...	<NULL>	Markt	☐
, Oberpf	0.0	Markt Lauterhofen...	92283 Lauter...	Ballertshofen (Lauterho...	POINT(4464626 5472004)	<NULL>	Markt	☐
	0.0	Stadt Velburg	92355 Velburg	Ostermühlweg (Velburg)	POINT(4473253.5 5455...	<NULL>	Stadt	☐
a.d.Altmühl	0.0	Stadt Dietfurt a.d.A...	92345 Dietfurt...	Weinbergstraße (Dietfur...	POINT(4469419 5433732)	<NULL>	Stadt	☐
b. München	0.0	Gemeinde Kirchhei...	85551 Kirchhe...	Weißenfelder Straße (Ki...	POINT(4481736.71 533...	<NULL>	Gemeinde	☐
b. München	0.0	Gemeinde Kirchhei...	85551 Kirchhe...	Weißenfelder Straße (Ki...	POINT(4481779.67 533...	<NULL>	Gemeinde	☐
b. München	0.0	Gemeinde Kirchhei...	85551 Kirchhe...	Weißenfelder Straße (Ki...	POINT(4481817.25 533...	<NULL>	Gemeinde	☐
b. München	0.0	Gemeinde Kirchhei...	85551 Kirchhe...	Seestraße (Kirchheim b...	POINT(4481726.12 533...	<NULL>	Gemeinde	☐
b. München	0.0	Gemeinde Kirchhei...	85551 Kirchhe...	Seestraße (Kirchheim b...	POINT(4481744.92 533...	<NULL>	Gemeinde	☐
	0.0	Gemeinde Egmatin...	85658 Egmating	Münchener Straße (Eg...	POINT(4484717 531866...	<NULL>	Gemeinde	☐
, Mittelfr	0.0	Kreisfreie Stadt Nü...	90409 Nürnbe...	Parkstraße (Nürnberg)	POINT(4433974.12 548...	<NULL>	Kreisfreie Stadt	☐
b. München	0.0	Gemeinde Kirchhei...	85551 Kirchhe...	Hürderstraße (Kirchhei...	POINT(4482103.49 533...	<NULL>	Gemeinde	☐
b. München	0.0	Gemeinde Kirchhei...	85551 Kirchhe...	Sonnenallee (Kirchheim...	POINT(4482168.95 533...	<NULL>	Gemeinde	☐
b. München	0.0	Gemeinde Kirchhei...	85551 Kirchhe...	Hürderstraße (Kirchhei...	POINT(4482242.47 533...	<NULL>	Gemeinde	☐
b. München	0.0	Gemeinde Kirchhei...	85551 Kirchhe...	Hürderstraße (Kirchhei...	POINT(4482247.24 533...	<NULL>	Gemeinde	☐
b. München	0.0	Gemeinde Kirchhei...	85551 Kirchhe...	Sonnenallee (Kirchheim...	POINT(4482276.27 533...	<NULL>	Gemeinde	☐
b. München	0.0	Gemeinde Kirchhei...	85551 Kirchhe...	Hürderstraße (Kirchhei...	POINT(4482373.24 533...	<NULL>	Gemeinde	☐
b. München	0.0	Gemeinde Kirchhei...	85551 Kirchhe...	Marsstraße (Kirchheim ...	POINT(4482872.62 533...	<NULL>	Gemeinde	☐

Abbildung 17: Anhang 1.2: PostgreSQL-Tabelle

Anhang 2: Das verwendete Mapping mit benutzerdefinierten Settings (mit den ersten drei Feldern).

```
PUT My_Index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_custom_analyzer": {
          "type": "custom",
          "tokenizer": "my_tokenizer",
          "char_filter": [
            "html_strip", "my_filter"
          ],
          "filter": [ "lowercase", "unique"]
        }
      },
      "tokenizer": {
        "my_tokenizer": {"type": "pattern", "pattern": "[., -]" }
      },
      "char_filter": {
        "my_filter": {
          "type": "mapping", "mappings": [ "ö=>oe", "ä=>ae", "ü=>ue","ß=>ss"]
        }
      }
    }
  },
  "mappings": {
    "adressen2": {
      "dynamic": false,
      "properties": {
        "label": {
          "type": "text", "analyzer": "my_custom_analyzer",
          "fields": {
            "keyword": {
              "type": "keyword",
              "ignore_above": 256
            }
          }
        },
        "wgs84_lat_lon": {
          "type": "geo_point",
          "fields": {
            "keyword": {
              "type": "keyword",
              "ignore_above": 256
            }
          }
        },
        "kreis": {
          "type": "text",
          "fields": {
            "keyword": {
              "type": "keyword",
              "ignore_above": 256
            }
          }
        }
      }
    }
  },
  "aliases": {
    "actions": [{
      "add": { "index": "bayern_new", "alias": "bayern_db"}
    }]
  }
}
```

Listing 43: Anhang 2, Mapping mit benutzerdefinierte Settings

Anhang 3: Die *Bash*-Skripte

```
#!/bin/bash
A="\e[1m"
E="\e[21m"
R="\e[31m"
D="\e[39m"
set -e
TS=$(date +%Y%m%d_%H%M)
INDEXNAME=adresse_$TS
TMPDIR=tmp/elasticadresse_$TS
SERVER=localhost:9201
TYPE=adresse
ALIAS=adresse
JARFILE=/home/asli_mo/NetBeansProjects/Postgres2JSON/target/Postgres2JSON-1.0-jar-with-dependencies.jar
echo -e "$A~~~~~~~~~~start~~~~~~~~~~$E"
mkdir -p $TMPDIR
time -p java -jar $JARFILE -db jdbc:postgresql://v****72.***.bvv.bayern.de:5***/z** USER PASS -i
$INDEXNAME -t $TYPE -v hk_toindex_view -n label label_sort 6 -v -p $TMPDIR

echo -e "copy Mappings into $TMPDIR"
cp Mappings.json $TMPDIR

echo -e "#!/bin/bash\nINDEX=$INDEXNAME \nSERVER=$SERVER\nTYPE=$TYPE\nTMPDIR=$TMPDIR\nALIAS=$ALIAS" | cat
- importJSON.sh > $TMPDIR/importJSON1.sh # Setzte die Variable auf das nächste Skript

echo -e $A$R"starting importJSON.sh .."$D$E
echo ""

sh ./$TMPDIR/importJSON1.sh
echo -e "$A~~~~~~~~~~finish~~~~~~~~~~$E"
```

Listing 44: Anhang 3.1: Skript 1 - export_database.sh

```
A="\e[1m"; E="\e[21m"; R="\e[31m"; D="\e[39m" # Farbe und Formatierung.

echo -e $A"deleting identical indices if exists"$E

response=$( curl http://$SERVER/$INDEX \ --write-out %{http_code} \ --silent \ --output /dev/null \ )

if [ "$response" == "200" ]; then  curl -XDELETE "$SERVER/$INDEX/?pretty"; fi

echo -e $A$R"creating Index and Setting Mapping"$D$E

curl -XPUT "http://$SERVER/$INDEX?pretty" -H 'Content-Type: application/json' -d @Mappings.json

echo -e $A$R"importing data:"$D$E

time (echo  LISTE=$(ls $TMPDIR/*.json -tr)
for DATEI in $LISTE
do echo "      $DATEI .."
curl -x "" -XPUT -s -H 'Content-Type: application/json' -XPOST "$SERVER/$INDEX/$TYPE/_bulk?pretty"
--data-binary @"$DATEI" | jq ".errors" | grep true
done )

echo -e $A"creating alias"$E

curl -XPUT "$SERVER/$INDEX/_aliases?pretty" -H 'Content-Type: application/json' -d'
{ "actions" : [ { "add" : { "index" : "'$INDEX'", "alias" : "'$ALIAS'" } } ] } '

echo -e $A"deleting old aliases if exists"$E

LISTE_INDEX=$( curl -s "$SERVER/_alias/$ALIAS?pretty" | jq -r 'keys | .[]' | grep -v "$INDEX")

for OLDALIAS in $LISTE_INDEX
do echo "Alias $OLDALIAS -deleted-"
curl -s -XPOST "$SERVER/_aliases?pretty" -H 'Content-Type: application/json' -d"
{\"actions\" : [ { \"remove\" : { \"index\" : \"$OLDALIAS\", \"alias\" : \"$ALIAS\" } } ] } "
done

echo -e $A"deleting old indices if exists"$E

LISTE_INDEX=$(curl -s "$SERVER/_all?pretty" | jq  -r 'keys | .[]' | grep "$ALIAS_" | grep -v "$INDEX"
| sort -r | tail -n +3)

for OLDINDEX  in $LISTE_INDEX
do    echo "Index $OLDINDEX -deleted-"
curl -s -XDELETE "$SERVER/$OLDINDEX?pretty"
done

echo -e $A"Getting documents count"$E

sleep 3

ESCOUNT=$(curl -XGET "$SERVER/$ALIAS/$TYPE/_count?pretty" -s | jq ".count")
JARCOUNT=$(cat $TMPDIR/info.protocol | jq ".count")

if [[ $ESCOUNT -eq $JARCOUNT ]]; then echo "Elasticsearch=$ESCOUNT PostgreSQL=$JARCOUNT OK!"; else
echo "Elasticsearch=$ESCOUNT PostgreSQL=$JARCOUNT Error!"; fi
```

Listing 45: Anhang 3.2: Skript 2 - importJSON.sh

```
asli_mo@v****72:/export/home/asli_mo/ExportDB2ES$ bash export_database.sh
~~~~~~~~~~~start~~~~~~~~~~~
2018/03/22 11:42:41 - Connected
2018/03/22 11:42:42 - Writing files... it can take a while.
2018/03/22 11:42:42 - Path: /export/home/asli_mo/ExportDB2ES/tmp/elasticadresse_20180322_1142/
2018/03/22 11:42:42 - creating new File: elasticsearchJSON0.json
2018/03/22 11:42:57 - creating new File: elasticsearchJSON1.json
                                ...
2018/03/22 11:48:14 - creating new File: elasticsearchJSON24.json
2018/03/22 11:48:27 - Finished!

copy Mappings into tmp/elasticadresse_20180322_1142
starting importJSON.sh ..

deleting identical indices if exists
creating Index and Setting Mapping
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "index" : "adresse_20180322_1142"
}
importing data:

     tmp/elasticadresse_20180322_1142/elasticsearchJSON0.json ..
     tmp/elasticadresse_20180322_1142/elasticsearchJSON1.json ..
     ...
     tmp/elasticadresse_20180322_1142/elasticsearchJSON24.json ..

real    4m26.273s
user    0m5.968s
sys 0m5.102s

creating alias
{
  "acknowledged" : true
}
deleting old aliases if exists
deleting old indices if exists
getting documents count
Elasticsearch=3566334 PostgreSQL=3566334 OK!
~~~~~~~~~~~finish~~~~~~~~~~~
```

Listing 46: Anhang 3.3: Anwendungsbeispiel der Bash-Skripte