Advanced-Databases Übungsblatt 01

Aufgabe 2a

Ausgangsdatensatz: Project Gutenberg DVD 07/2006

Vorgehensweise bei der Aufbereitung

1. Entpacken der vielen einzelnen Teilarchive
2. Formatprüfung: nur TXTs
3. Teilauswahl des Datensatzes, um verdaubare Datenverarbeitungszeiten zu ermöglichen
4. Filterung besonders kleiner Dokumente <= 64KB (diese enthielten idR nur Lizenzinformationen)
5. Erkennen und Entfernen von Disclaimer Headern / Footern mittels Regex, da diese die Spracherkennung verfälschen würde (Disclaimer waren immer in Englisch). Da sehr viele unterschiedliche Arten von Header und Footer-Texten existieren, konnten nicht alle zuverlässig mit Regex erfasst werden. Wollte man auf Grundlage dieser Daten Ähnlichkeiten zwischen den Büchern berechnen, könnte dies problematisch werden, da Bücher mit den gleichen Headern bzw. Footer-Texten mehrere gemeinsame Terme, und daher stärkere Ähnlichkeit aufweisen würden. Für unseren Anwendungsfall (Datenbank-Benchmarking) spielt es jedoch nur eine untergeordnete Rolle. Ich wollte es nur mal erwähnt haben.
6. Spracherkennung der Dokumente und Aussortieren von Nicht-Englischen-Büchern. (Filterung nach Sprache, da wir ja später Stemmen wollen) Spracherkennung mittels Language Detection Library for Java by @shuyo (.NET PORT)

Anzahl der Dokumente:

Original Datensatz	24.704
Teilauswahl	6383
Nach Sprachfilterung	5769

Der Source-Code für die Aufbereitung der Rohdaten findet sich in der Klasse `Documents.cs`

Aufgabe 2b

a. **Anzahl an Dokumenten: 5 769**

b. **Anzahl an Termen: 200 949 015**

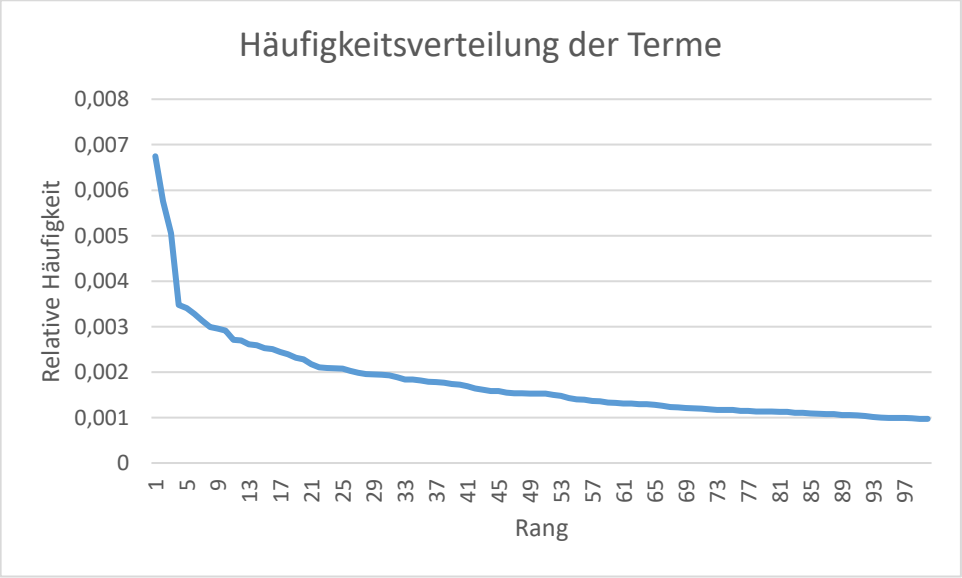
Vorgehensweise bei der Ermittlung der Terme: ToLowerCase()→ Splitting an Hand üblicher Trennzeichen (Whitespace, Komma, Punkt, usw…) → Entfernen aller Wörter, welche nicht-alphanumerischen Zeichen enthalten → Entfernen von Stop-Words (Stop-Word Liste von Google)

c. **Anzahl an eindeutigen (distinct) Termen: 812 120**
Ermittelt mit Hilfe einer HashMap

d. **Anzahl an eindeutigen (distinct) Termen nach Stemming** ~~oder Lemmatization~~: **641 692**
Das Stemming wurde mit Hilfe der .NET Library Annytab Stemmer durchgeführt.

e. **Zipf's Verteilung über die Terme**

Das Zipf'sche Gesetz besagt, dass sich die Häufigkeit einen Terms, umgekehrt proportional zu seinem Rang verhält.
Nachfolgendes Diagramm zeigt die Häufigkeitsverteilung der Terme über unseren Project Gutenberg Korpus.
Aus Darstellungsgründen wurden nur die ersten 100 Ränge dargestellt. Der Datensatz für nachfolgendes Diagramm befindet sich in der count_rank.csv. Siehe auch Source-Code Klasse: `Documents.cs`



Aufgabe 3

a. **Laden Sie die Benchmarking-Daten in MySQL und erzeugen Sie keinen Volltext-Index darauf.**

Siehe Source-Code Klasse: `MySQL.cs`

b. **Laden Sie die Benchmarking-Daten ein weiteres Mal in MySQL und erzeugen Sie einen Volltext-Index darauf.**

```
ALTER TABLE `gutenberg`.`book` ADD FULLTEXT INDEX `content_index` (`content` ASC);
```

c. **Laden Sie die Benchmarking-Daten in Elasticsearch2 .**

Siehe Source-Code in Klasse: `ElasticSearch.cs`

d**. Vergleichen Sie den Speicherbedarf (i) der Rohdaten, (ii) der MySQL-Daten und der (iii) ElasticsearchDaten.**

Speicherbedarf-Ermittlung in MySQL mittels:

```
select table_schema, sum((data_length+index_length)/1024/1024) AS MB from
information_schema.tables group by 1;
```

Speicherbedarf-Ermittlung in ElasticSearch mittels `elasticsearch-head` Plugin.

Persistenzmodell	**Speicherbedarf**
Rohdaten	2399 MB
MySQL	2401 MB (nach Indizierung 2451 MB)
ElasticSearch	3200 MB

e. **Führen Sie diverse Abfragen auf allen drei Systemen aus. Dabei sollen zumindest Vorwärtstrunkierung, Rückwärtstrunkierung und komplexe Abfragen ausgewertet werden. Wie performen die verschiedenen Systeme?**

Bin mir nicht ganz sicher was mit komplexen Abfragen gemeint ist. Da meine Datenbank ja nur eine Tabelle (books) mit den Spalten (Id und Content) besitzt kann ich nicht wirklich besonders komplexe Queries bilden. Im weiteren Verlauf nehme ich daher einfach mal an, dass mit einer komplexen Abfrage, eine postfix-prefix-Wildcard Abfrage gemeint war, also z.B. %house%.

	Vorwärtstrunk.	Rückwärtstrunk.	Komplexe Abfrage
MySQL ohne Index	45s	46s	48s
MySQL mit Index	0.033s	0.021s	0.046s
ElasticSearch	0.031s	0.025s	0.390s

MySQL(ohne Index) Queries: (Die hohen Laufzeiten legen nahe, das stets ein Full-Table Scan durchgeführt wird)

```
SELECT * FROM book WHERE content LIKE 'Constitution%'; // Vorwärtstrunkierung
SELECT * FROM book WHERE content LIKE '%alist'; // Rückwärtstrunkierung
SELECT * FROM book WHERE content LIKE '%sti%';  // Komplexe Abfrage
```

MySQL(mit Volltext-Index) Queries:

```
SELECT * FROM book WHERE MATCH (content) AGAINST ('Constitution*' IN NATURAL LANGUAGE MODE);
SELECT * FROM book WHERE MATCH (content) AGAINST ('*alist IN NATURAL LANGUAGE MODE);
SELECT * FROM book WHERE MATCH (content) AGAINST ('*sti*' IN NATURAL LANGUAGE MODE);
```

ElasticSearch Queries:

```
{"query":{"bool":{"must":[{"query_string":{"default_field":"book.content","query":"Constitution*"}}]}}}
{"query":{"bool":{"must":[{"query_string":{"default_field":"book.content","query":"*alist"}}]}}}
{"query":{"bool":{"must":[{"query_string":{"default_field":"book.content","query":"*sti*"}}]}}}
```

f. **Wie funktioniert Elasticsearch? Wie funktioniert der darunter liegende Volltext-Index?**

- Elasticsearch ist eine aggregatorientierte/dokumentenoriente Datenbank die vor allem als Suchmaschine eingesetzt wird.
- Sie setzt auf Lucene auf und stellt nach außen eine REST-Schnittstelle bereit.

- Die REST-Schnittstelle erwartet Dokumente im JSON-Format. Es stehen die klassischen HTTP-Operationen (GET, POST, PUT, DELETE) zur Verwaltung der Datenbank und des Index zur Verfügung.
- ElasticSearch kann komplett schemafrei betrieben werden. Die Datenbank versucht aus den übergebenen Dokumenten selbstständig so viele Informationen wie möglich über die Struktur des Dokuments abzuleiten.
- Alternativ kann man auch ein Schema (Terminologie in ElasticSearch: Mapping) vorgeben, wie ich es in meinem Source-Code getan habe (siehe Klasse `ElasticSearch.cs`)
- Nachdem man Dokumente dem Index hinzugefügt hat, lassen diese sich über GET-Requests, z.B. http://localhost:9200/project-gutenberg/books/1 abfragen.
- Auf dieselbe Weise lassen sich auch Suchanfragen an den Index durchführen. Die Suchanfrage kann man über Query-Parameter innerhalb der URL formulieren oder in Form einer JSON-strukturierten Message im HTTP-Body (Query DSL).
- Zur Wahl stehen Term-Suche, Query-Suche, Phrasen-Suche, Vektor-Suche, und weitere…
- Der Volltext-Index analysiert neu hinzugekommene Dokumente ehe sie in den Index aufgenommen werden.
- Hierbei werden vom Analyzer mehrere Schritte ausgeführt: Splitting / Erstellen von Tokens, Umwandlung in Kleinbuchstaben, Stopp-Wort-Filterung, ...
- Nach der Analyse des Dokuments aktualisiert die Datenbank den Index dahingehend, dass sie jedem Token, das zugehörige Dokument zuordnet.

g. **Probieren Sie boolesche und auch Vektor-basierte Abfragen in Elasticsearch aus.**

Beispiel für eine ausgeführte boolsche Abfrage:

```
{
   "query":{
      "bool":{
         "must":[
            {
               "query_string":{
                  "default_field":"book.content",
                  "query":"Shakespeare"
               }
            }
         ],
         "must_not":[

         ],
         "should":[
            {
               "query_string":{
                  "default_field":"book.content",
                  "query":"Romeo AND Juliet"
               }
            }
         ]
      }
   }
}
```

i. **Wie verteilt Elasticsearch Daten bzw. den Index?** UND h. **Wie stellt Elasticsearch Performance sicher?**

In ElasticSearch gibt es folgende Architekturbausteine: Cluster → Node → Index → Type → Shard / Replicas

Cluster: Sammlung von Node Servern, über die der Gesamtdatenbestand verteilt ist. Cluster ermöglichen Leistenverteilung und redundante Speicherung über die einzelnen Nodes.

Node: Einzelner Server, der Teil eines Clusters ist und an der Indizierung / Beantwortung von Suchanfragen beteiligt ist.

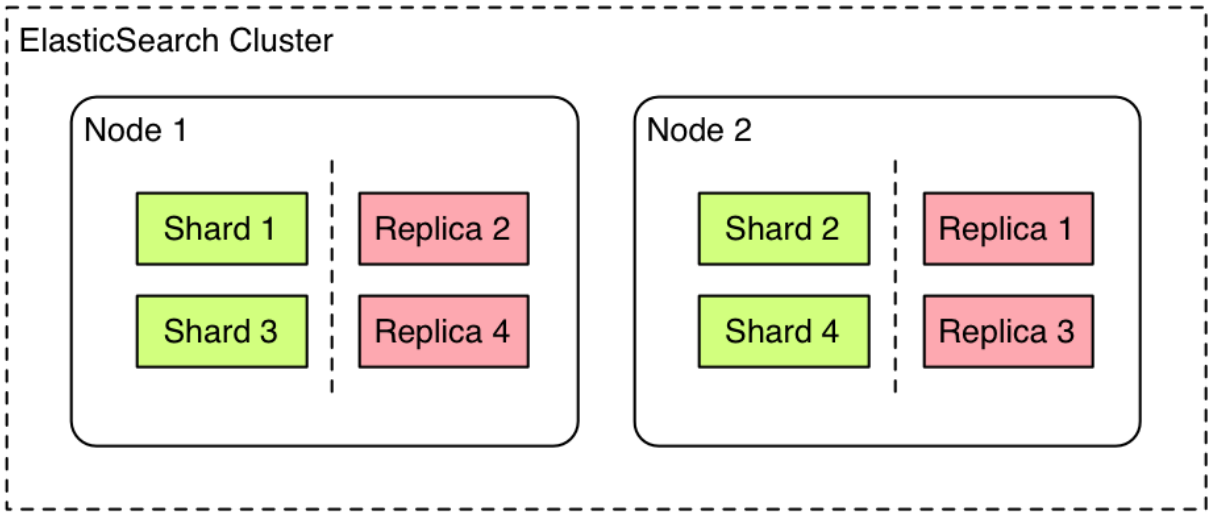
Index: Sammlung von Dokumenten mit ähnlicher Charakteristik (z.B. Büchersammlung, Produktkatalog,..) (in etwa vergleichbar mit einer Datenbank in relationalen Datenbanken)

Type: Logische Untergliederung des Index an Hand gemeinsamer Felder bestimmter Dokumente. (in etwa vergleichbar mit einer Tabelle in relationalen Datenbanken) z.B. Kann der Index „blog“ Subtypen für „posts“ und „comments“ enthalten.

Document: ein einzelner Dateneintrag (in etwa vergleichbar mit einer Zeile in relationalen Datenbanken). Ein Document wird im JSON-Format beschrieben. Jedes indizierte Dokument muss einem Typ angehören.

Primary Shards: Primary Shards sind die Teile eines aufgesplitteten Index. Shard ermöglichen die horizontale Skalierung, d.h. die Verteilung des Index und damit auch der Rechenlast und des Speicherbedarfs, auf zusätzliche Server. ElasticSearch behält selbstständig den Überblick über die Verteilung der Shards auf den einzelnen Nodes und ist daher für den Nutzer komplett transparent.

Replica Shards: Replica Shards sind Kopien von Primary Shards die ebenfalls über die Nodes verteilt werden, um ein Backup / Hochverfügbarkeit zu gewährleisten für den Fall das ein Node ausfällt.



Quelle der Abbildung: https://blog.liip.ch/archive/2013/07/19/on-elasticsearch-performance.html

Performance und Hochverfügbarkeit ergeben sich aus der oben beschriebenen Architektur. Durch die unkomplizierte Art der horizontalen Skalierung kann der Index beliebig groß werden.

j. **Wie kann Lucene Wildcard-Abfragen effizient beantworten?**

- Lucene identifiziert für einen Suchterm alle Vorkommen im Index und verknüpft diese mit dem boolschen OR
- Beispielsweise: Vor* → Vor OR Vordruck OR Vorlage OR Vorgang OR …

k. **Was ist „Fuzzy Search"? Wie funktioniert diese?**

- zu deutsch: unscharfe Suche
- Suchmodus, bei welchem nicht die exakte Zeichenfolge gefunden werden muss
- Ähnliche Zeichenfolgen (nach Schreibweise oder Aussprache) gelten ebenfalls als Treffer
- Anwendungsgebiet: in der Regel im Bereich der natürlichen Sprachsuche z.B. Vertauschte Buchstaben, Tippfehler …
- Ermittelt wird die Ähnlichkeit meist mit Hilfe des Levenshtein-Distanz-Algorithmus
- Dieser errechnet Anzahl an Bearbeitungsschritten (Löschen, Einfügen, Austauschen) um eine Zeichenkette in eine andere zu Überführen. Je weniger Schritte, desto ähnlicher.
- In Lucene kann man beispielsweise über <term>~<0-1> (z.B. Heteroscedasticity~0.1) definieren, wie tolerant das Matching ist: 0 = tolerant <> 1 = nicht tolerant