Hannes Leutloff
5 689 808
Bachelor of Science Informatik
8. Semester
hannes.leutloff@aol.de

Bachelorarbeit

Komparative Analyse von Datenbank- und Indexierungssystemen im Kontext des Natural Language Processing

Hannes Leutloff

Abgabedatum: 14. Mai 2018

Text Technology Lab
Leiter: Prof. Dr. A. Mehler

Erklärung

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

_____

Ort, Datum

_____

Unterschrift

Inhaltsverzeichnis

Abkürzungs- und Begriffsverzeichnis

API Application Program Interface
Die Schnittstelle, die ein Programm(-teil) für die Interaktion mit einem anderen Programm(-teil) verwendet.

AQC ArangoDB Query Language
Die Querysprache, die zur Interaktion mit ArangoDB verwendet wird.

BiGram BiGram
Ein BiGram ist die Komposition zweier aufeinander folgender Begriffe in einem Text.

BSON Binary JSON
BSON ist eine binär-codierte Serialisierung von JSON-artigen Dokumenten [8].

DBIS Datenbank- oder Indexierungssystem
Ein Term, der in dieser Arbeit vereinheitlichend für sowohl Datenbank- als auch Indexierungssysteme verwendet wird.

GB GigaByte
Einheit für Datenmengen.

HTTP Hypertext Transfer Protocol
HTTP ist ein Applikationsprotokoll für die Kommunikation innerhalb verteilter Informationssysteme. [2]

JSON JavaScript Object Notation
„JSON [..] ist ein schlankes Datenaustauschformat, das für Menschen einfach zu lesen und zu schreiben und für Maschinen einfach zu parsen [..] und zu generieren ist.“ [22]

MB MegaByte
Einheit für Datenmengen.

NLP Natural Language Processing
Das Studiengebiet der Verarbeitung natürlicher Sprachen mithilfe von Computern.

NoSQL Not only SQL
Bezeichnet Datenbanken mit nicht-relationalem oder teilweise relationalem Ansatz.

RAM Random Access Memory
Als Arbeitsspeicher moderner Computer verwendeter Datenspeicher.

RDF Resource Description Framework
Ein Standardmodell für Datenaustausch im Internet. [34]

SPARQL SPARQL Protocol And RDF Query Language
Eine Graph-basierte Query-Sprache für RDF Ressourcen. [9]

SQL Structured Query Language
Standardisierte Abfragesprache für relationale Datenbanken.

TFIDF Term-Frequency/Inverse-Document-Frequency
Der Quotient der Term-Frequency durch die Inverse-Document-Frequency eines Terms. Misst die Bedeutsamkeit eines Begriffes in einer Menge von Dokumenten.

TriGram TriGram
Ein TriGram ist - analog zum BiGram - eine Komposition dreier aufeinander folgender Begriffe in einem Text.

TTR Type-Token-Ratio
Das Verhältnis von Tokens zu Types in einem Text.

UIMA Unstructured Information Management Architecture
Eine open-source Bibliothek für die Koordination von NLP-Prozessen.

W3C World Wide Web Consortium
Eine internationale Community für die Entwicklung von Web-Standards.

XMI XML Metadata Interchange
Ein auf XML basierendes Format für Metadaten.

XML eXtensible Markup Language
Eine Markupsprache für den Austausch strukturierter Daten.

1 Einleitung

1.1 Motivation

NLP (Natural Language Processing) [37] ist ein schnell wachsendes und vielseitiges Feld. Seine Anwendungen reichen von E-Mail-Spamerkennung über Wissensaufbereitung in Medien wie Wikipedia bis hin zur Unterstützung von psychologischer Arbeit [6].

Prof. Mehler arbeitet mit seinem Team am TextImager Projekt. [7] Der TextImager bietet Textverarbeitungs- und Visualisierungstools im Webbrowser an. Eigens eingefügte Texte können analysiert werden.

Die angebotenen Tools nutzen im Hintergrund UIMA-Services (Unstructured Information Management Architecture), die Berechnungen auf den analysierten Texten anstellen und Ergebnisse zur Visualisierung zur Verfügung stellen.

Um in diesen Berechnungen und in der Reaktionszeit des Frontends international kompetitiv sein zu können, ist es nötig, verschiedene Data-Storage Optionen zu testen und die effizienteste Wahl zu treffen. Es muss daher ein extensiver Lasttest verschiedener DBIS (Datenbank- oder Indexierungssysteme) mit einem diversen Datenset auf den benötigten Operationen durchgeführt werden, um diese hinsichtlich ihrer Eignung für den TextImager vergleichen zu können.

1.2 Vorgaben

Im Rahmen dieser Bachelorarbeit wurden Analysedaten von Prof. Mehlers Team zur Verfügung gestellt.

Besagte Daten sind gezippte .xmi Dateien. Das XMI (XML Metadata Interchange) Format ist ein Standard für den Metadatenaustausch im XML (Extensible Markup Language) Format [3].

In diesem Falle enthielten die .xmi Dateien voranalysierte Inhalte des Biologie-Bereichs der deutschen Wikipedia. Das Format entspricht dem Ausgabeformat der NLP-Pipeline des Teams um Prof. Mehler und verwendet das DKPro UIMA-Plugin der Technischen Universität Darmstadt [19].

Die verwendeten Metriken und zu überprüfenden Methoden, welche in 2.1 beschrieben werden, basieren auf Vorarbeiten einer Praktikumsgruppe unter Prof. Mehler und auf Konversationen mit dem wissenschaftlichen Mitarbeiter, Herrn Wahed Hemati.

Für die Durchführung der Lasttests wurde vom TextImager-Team ein Server zur Verfügung gestellt. Aufgrund des eingeschränkten Zeitrahmens einer Bachelorarbeit und der Ein-Server-Testumgebung wird bei der Verwendung der Datenbanken horizontale Skalierung (d.h. Verteilung der Daten über mehrere Server) nicht berücksichtigt.

1.3 Ziel

Das Ziel dieser Arbeit ist es, eine Empfehlung für die Wahl eines oder mehrerer DBIS für den TextImager aussprechen zu können.

Zu diesem Zweck wird analysiert, welche Operationen auf den Datenbanken relevant für die Verwendung im TextImager sind.

Anhand dieser Operationen werden die in Abschnitt 1.2 erwähnten Metriken zum Vergleichen der gewählten Datenbanken in Form von `EvaluationCase`s in Java implementiert. Dazu mehr in Abschnitt 2.

Zuletzt werden die Ergebnisse der Evaluationen in JSON-Format (JavaScript Object Notation) und als Graphen für die Interpretation zur Verfügung gestellt.

1.4 Struktur

In dieser Arbeit werden die analysierten DBIS im Detail vorgestellt. Es werden die verwendeten Konzepte der DBIS dargestellt und analysiert, inwiefern diese relevant für den Anwendungsfall sind. Die praktische Umsetzung wird erläutert und welche Unterschiede zwischen Datenbanken existieren.

Zur Strukturierung dieser Informationen gibt es zwei Optionen. Eine Option ist die Gruppierung nach Inhalt - d.h. die Abschnitte „Konzepte der DBIS“, „Praktische Umsetzung“ und „Besonderheiten“ o.ä. sind top-level Kapitel.

Stattdessen wird in dieser Bachelorarbeit nach den DBIS selbst gruppiert. Jeder Abschnitt enthält in dieser Reihenfolge Informationen über die Funktionsweise des DBIS, die praktische Umsetzung und Strukturierung des Codes für dieses DBIS und die Besonderheiten. So kann jedes DBIS für sich genommen betrachtet und extrahiert werden.

2 Architektur der Analysesoftware

Das Evaluationssystem ist in Java geschrieben und verwendet Docker und Docker-Compose zum Ausführen der benötigten DBIS.

Im folgenden werden Kernkonzepte der entwickelten Analysesoftware vorgestellt. Diese sind notwendig um die Zusammenhänge der Software zu verstehen und sowohl ihre Ergebnisse zu interpretieren, als auch um sie zu verwenden und zu erweitern.

2.1 QueryHandlerInterface

Das `QueryHandlerInterface` (siehe Abbildung 2.1) ist die API (Application Program Interface), die für jedes DBIS implementiert wird.

Es enthält Definitionen für die Interaktion mit den DBIS. Dazu gehören:

- Strukturoperationen. Vorbereiten, öffnen und leeren des Inhalts
- Schreiben eines Dokumentes in die Datenbank
- Leseoperationen auf besagten Dokumenten
- Durchsuchen der Daten; hauptsachlich Zählen der enthaltenen Elemente
- Berechnungen von z.B. TTR (Type-Token-Ratio), TFIDF (Term-Frequency/Inverse-Document-Frequency)
- Lesen/Erstellen von BiGrams und TriGrams

Diese Operationen wurden von Prof. Mehler und seinem Team als bedeutsam für die Entwicklung des TextImagers identifiziert (Wahed Hemati, persönliche Kommunikation, 27.10.2017) und bilden daher die Grundlage für den Vergleich von DBIS.

2.2 EvaluationRunner

Der `EvaluationRunner` ist das Verbindungsstück, das einzelne Evaluationen auf die besagten DBIS anwendet. Zu seinen Aufgaben gehört zum einen das Erfragen einer Verbindung zur benötigten Datenbank mithilfe des `ConnectionManager`s und zum anderen Das Ausführen einer der fünf Implementationen des `EvaluationCase` Interfaces.

Die `EvaluationCase` Implementationen sind in fünf Aspekte geteilt:

- Write
  Liest mithilfe einer UIMA-Pipeline Inputdateien ein und schreibt diese in das DBIS.

```java
public interface QueryHandlerInterface
{
    Connections.DBName forConnection ();
    void setUpDatabase() throws IOException;
    void openDatabase() throws IOException;
    void clearDatabase() throws IOException;

    void storeDocumentHierarchy(...);
    String storeJCasDocument(...);
    Iterable<String> storeJCasDocuments(...);
    String storeParagraph(...);
    String storeParagraph(...);
    String storeSentence(...);
    String storeSentence(...);
    String storeToken(...);
    String storeToken(...);

    void checkIfDocumentExists(...);
    Iterable<String> getDocumentIds();
    Set<String> getLemmataForDocument(...);

    void populateCasWithDocument(...);

    int countDocumentsContainingLemma(...);
    int countElementsOfType(...);
    int countElementsInDocumentOfType(...);
    int countElementsOfTypeWithValue(...);
    int countElementsInDocumentOfTypeWithValue(...);
    Map<String, Integer> countOccurencesForEachLemmaInAllDocuments();

    Map<String, Double> calculateTTRForAllDocuments();
    Double calculateTTRForDocument(...);
    Map<String, Double> calculateTTRForCollectionOfDocuments(...);
    Map<String, Integer> calculateRawTermFrequenciesInDocument(...);
    Integer calculateRawTermFrequencyForLemmaInDocument(...);
    double calculateTermFrequencyWithDoubleNormForLemmaInDocument(...);
    double calculateTermFrequencyWithLogNormForLemmaInDocument(...);
    Map<String, Double> calculateTermFrequenciesForLemmataInDocument(...);
    double calculateInverseDocumentFrequency(...);
    Map<String, Double> calculateInverseDocumentFrequenciesForLemmataInDocument(...);
    double calculateTFIDFForLemmaInDocument(...);
    Map<String, Double> calculateTFIDFForLemmataInDocument(...);
    Map<String, Map<String, Double>> calculateTFIDFForLemmataInAllDocuments();

    Iterable<String> getBiGramsFromDocument(...);
    Iterable<String> getBiGramsFromAllDocuments();
    Iterable<String> getBiGramsFromDocumentsInCollection(...);
    Iterable<String> getTriGramsFromDocument(...);
    Iterable<String> getTriGramsFromAllDocuments();
    Iterable<String> getTriGramsFromDocumentsInCollection(...);
}
```

Abbildung 2.1: Ausschnitt aus QueryHandlerInterface.java

```java
public void run()
{
    ...

    Collection<QueryHandlerInterface> queryHandlers = new ArrayList<>();
    for (Connection connection : connectionResponse.getConnections()) {
        queryHandlers.add(QueryHandlerInterface.createQueryHandlerForConnection(connection));
    }

    for (EvaluationCase evaluationCase : this.evaluations) {
        boolean success = false;
        while (!success) {
            try {
                evaluationCase.run(queryHandlers, this.outputProvider);
                success = true;
            } catch ...
        }
    }
}
```

Abbildung 2.2: Gekürzter Inhalt der run Methode des EvaluationRunner.java

- Read
  Liest Dokumente aus dem DBIS aus und stellt diese in einer UIMA-Pipeline zur Verfügung.
- Query
  Führt Operationen aus dem Bereich Auslesen und Zählen aus.
- Calculate
  Führt Operationen aus dem Bereich Berechnungen aus.
- ComplexQuery
  Liest die BiGrams und TriGrams aus dem DBIS aus.

Diese können unabhängig voneinander ausgeführt werden, was die Effizienz der Java-Garbage-Collection erhöht und Memory-Leaks vermeidet.

2.3 BenchmarkQueryHandler

Der `BenchmarkQueryHandler` ist eine Implementation des `QueryHandlerInterfaces`, welche eine weitere Instanz des selbigen verwendet und dessen Performance analysiert.

Um die Ausführungen einzelner Operationen auf der Datenbank möglichst präzise messen zu können, werden alle Methoden der `QueryHandlerInterface` Instanz durch eine Instanz des `BenchmarkQueryHandlers` aufgerufen. Dieser protokolliert die exakte Zeit, die der Aufruf benötigt und speichert diese in einer Liste der Aufrufe.

Auf diese Weise können im Nachhinein alle protokollierten Aufrufe nachvollzogen und analysiert werden.

```java
@Override
public void storeDocumentHierarchy(JCas document) throws QHException
{
    long start = System.currentTimeMillis();
    this.subjectQueryHandler.storeDocumentHierarchy(document);
    long end = System.currentTimeMillis();
    MethodBenchmark mb = this.methodBenchmarks.get("storeDocumentHierarchy");
    mb.increaseCallCount();
    mb.addCallTime(end - start);
}
```

Abbildung 2.3: Beispiel eines Benchmarks an einer Methode des BenchmarkQuery-Handlers

Siehe dazu beispielsweise Abbildung 2.3. Hier umfasst die Zeitmessung die Methode `storeDocumentHierarchy`, welche mit denselben Parametern aufgerufen wird, die der `BenchmarkQueryHandler` erhalten hat.

3 Datenbanken und Indexierungssysteme

3.1 ArangoDB

3.1.1 Konzepte und Fähigkeiten

ArangoDB ist eine „highly available Multi-Model NoSQL database“ [14] geschrieben in C++. Es ist konzipiert als native Multi-Model-Datenbank und unterstützt Key/Value-Store, Document-Store und Graph Modelle. Es ermöglich daher eine sehr flexible Umsetzung von Datenmodellen und effizientes Durchsuchen des Modells nach sowohl relationalen als auch graphbasierten Kriterien [35].

ArangoDB wird von der ArangoDB GmbH entwickelt und supportet.

Um mit ArangoDB zu interagieren, wird AQL (ArangoDB Query Language) benutzt. AQL kann alle unterstützten Datenmodelle in seinen Abfragen vereinen. Sie wurde entworfen, um den Umgang mit einem Multi-Model-Store intuitiv und eindeutig zu machen und alle Bedürfnisse an ArangoDB zusammenzufassen.

Auch unterstützt ArangoDB selbstorganisiertes Clustering für horizontale Skalierung. Die Effizienz dessen schwankt nach den verwendeten Features. Key/Value-Stores sind effizienter skalierbar als z.B. Graphen [15].

3.1.2 Praktische Verwendung

3.1.2.1 Docker

```
FROM arangodb:3.2
```

Abbildung 3.1: ArangoDB Dockerfile

Zur Erstellung des Docker Containers wurde im lokalen Dockerfile (Abbildung 3.1) das offizielle Docker Image verwendet [16].

3.1.2.2 Konfiguration

```
arangodb:
  build: ./dbs/arangodb
  environment:
  - ARANGO_ROOT_PASSWORD=root
```

Abbildung 3.2: ArongoDB Konfiguration in docker-compose

```
# See README.txt in dbs/arangodb for details on:
- ARANGODB_HOST=arangodb
- ARANGODB_PORT=8529
- ARANGODB_USER=root
- ARANGODB_PASS=root
- ARANGODB_DB=uimadatabase
```

Abbildung 3.3: ArongoDB Umgebungsvariablen

Die ArangoDB Instanz wurde über die offizielle Konfiguration im Dockerfile hinaus nur mit einem Password für den Rootnutzer konfiguriert (siehe Abbildung 3.2). Es wird daher in der Anwendung der Standarduser `root` verwendet (siehe Abbildung 3.3).

3.1.2.3 Datenmodell

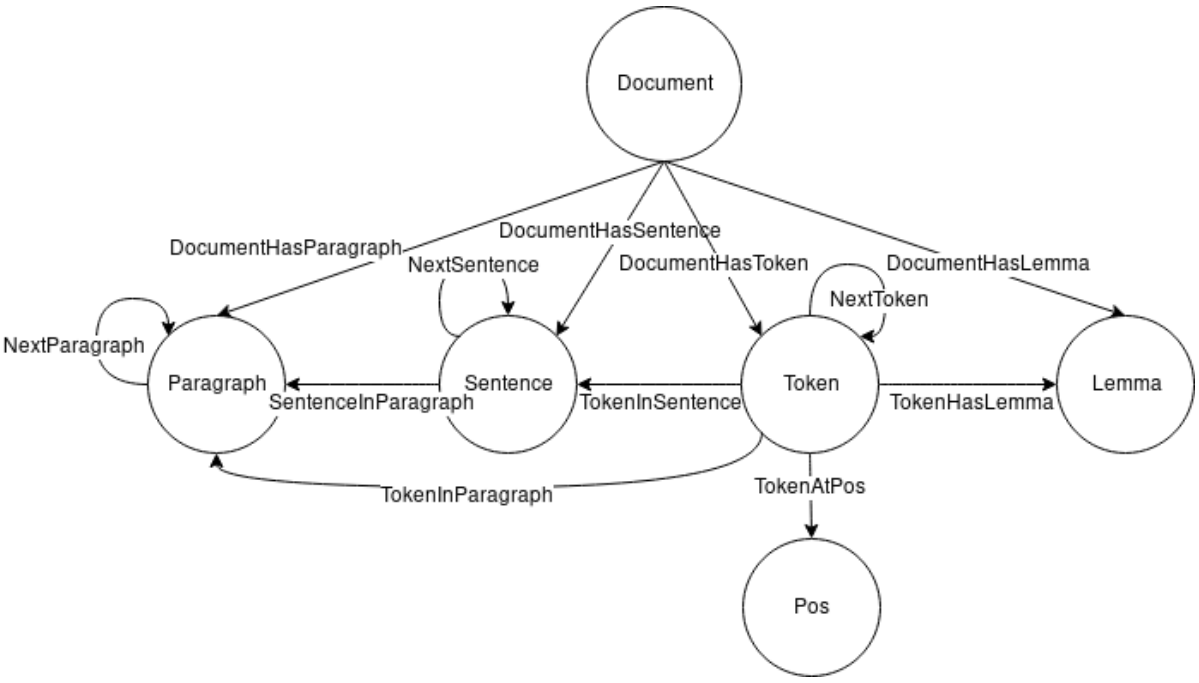

Abbildung 3.4: Struktur des Modells in ArangoDB

Wie in Abbildung 3.4 zu sehen folgt das in ArangoDB verwendete Modell einem simplen hierarchischen Muster mit zusätzlichen Querverbindungen für mehr Queryability.

`Lemma`ta werden als Singletons behandelt. D.h. es wird nie zwei `Lemma` Objekte in der Datenbank geben, die dieselbe `value` haben. Stattdessen werden die vorhandenen Objekte wiederverwendet. Daher lassen sich leicht Verbindungen zwischen Dokumenten erkennen, die dieselben `Lemma`ta enthalten.

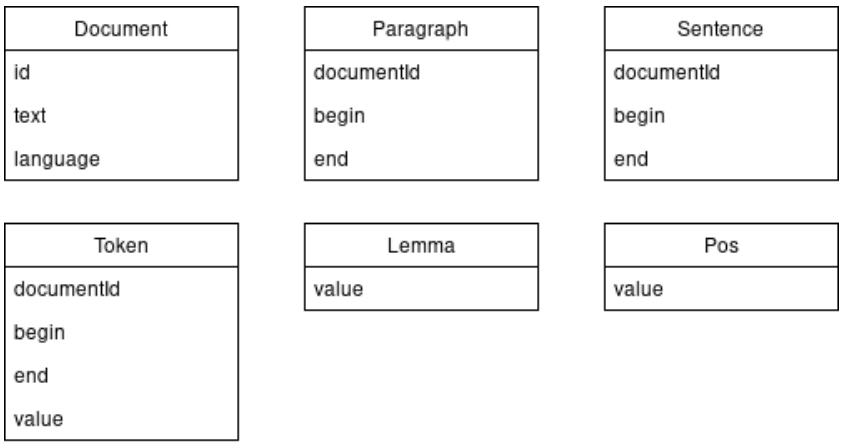

Abbildung 3.5: Felder der Modells in ArangoDB

Die Verbindungen von `Document` zu allen anderen Models (außer `Pos`) machen es trivial, die Anzahl der mit einem Dokument Verknüpften Elemente zu finden.

Die Verbindungen der Arten `NextParagraph`, `NextSentence` und `NextToken` reduzieren stark die benötigte Rechenzeit, um Bi- und TriGramme zu finden.

3.1.2.4 Besonderheiten

Durch die Kombination eines Collection-Stores mit einem Graph-Modell kann ArangoDB sowohl schnell Verbindungen zwischen Elementen abschreiten, als auch die Datenfelder einzelner Objekte abfragen.

Dies kommt unseres Anwendungsfall zugute, da sowohl die Verbindungen z.B. von `Document`s über `Token`s zu `Lemma`ta als auch die `value`s einzelner `Token`s effizient abfragbar sein müssen.

3.2 BaseX

3.2.1 Konzepte und Fähigkeiten

BaseX ist eine hochperformante und robuste XML Datenbank Engine und implementiert den XQuery 3.1 Prozessor mit vollem Support für die W3C (World Wide Web Consortium) Spezifikation. Es ist open source und wird von der BaseX GmbH entwickelt und supportet.

BaseX nutzt das Dateisystem als Grundlage um XML Dateien abzuspeichern. Diese werden mithilfe des XQuery Prozessors indiziert und durchsucht bzw. bearbeitet.

Zum Vorteil dieser Arbeit hat BaseX Java Bindings für alle Funktionen und ist daher in eine Java Anwendung leicht zu integrieren.

Die Dokumentation [30] ist leider lückenhaft und nur teilweise hilfreich.

3.2.2 Praktische Verwendung

3.2.2.1 Docker

```
FROM ubuntu:xenial

RUN apt-get update
RUN apt-get install -y basex

RUN rm -rf /srv
RUN adduser --home /srv --disabled-password --disabled-login \
  --uid 1984 --gecos "" basex && chown -R basex /srv
USER basex

EXPOSE 1984
CMD ["basexserver"]
```

Abbildung 3.6: BaseX Dockerfile

Da das offizielle Dockerfile von BaseX [17] sich bei Tests als nicht produktionsfähig erwiesen hat, musste eines entwickelt werden (siehe Abbildung 3.6).

Dieses basiert nun auf einem generischen Ubuntu Xenial Image [28], auf welchem ein BaseX Server installiert und konfiguriert wird.

3.2.2.2 Konfiguration

```
basex:
  build: ./dbs/basex
```

Abbildung 3.7: BaseX Konfiguration in docker-compose

```
# See README.txt in dbs/basex for details on:
- BASEX_HOST=basex
- BASEX_PORT=1984
- BASEX_USER=admin
- BASEX_PASS=admin
- BASEX_DBNAME=uimadatabase
```

Abbildung 3.8: BaseX Umgebungsvariablen

Da die Konfiguration des BaseX Servers bereits im Dockerfile (Abbildung 3.6) vorgenommen wird, ist in docker-compose (Abbildung 3.7) keine weitere Konfiguration notwendig.

Die in der Anwendung verwendeten Umgebungsvariablen für BaseX sind in Abbildung 3.8 zu finden.

3.2.2.3 Datenmodell

Das BaseX Datenmodell ist identisch zum vorgegebenen Format der Daten, da die XML Dateien direkt importiert werden.

Jegliche Queries setzen daher auf der Struktur der Inputfiles auf.

Besagte Inputstruktur ist lose verbunden und sehr verbos. Verbindungen zwischen Elementen werden hauptsächlich über `xmi:id` Attribute hergestellt.

Da BaseX bzw. XQuery auf das Durchsuchen von Dateien nach Mustern ausgelegt ist und nicht auf das Folgen von Verbindungen in Dateien, führt die Struktur der Dateien zu erhöhter Komplexität von Abfragen, die mehrere Elemente betreffen.

3.2.2.4 Besonderheiten

Da BaseX auf dem Dateisystem arbeitet und XML Dateien direkt importiert werden können, sind initiale Schreibeprozesse sehr schnell.

3.3 Blazegraph

3.3.1 Konzepte und Fähigkeiten

Blazegraph ist eine hochskalierbare und hochperformante Graphdatenbank, mit Unterstützung für Blueprints [21] und RDF (Resource Description Framework) [34]/SPARQL (SPARQL Protocol And RDF Query Language) [9] APIs. Blazegraph nutzt HTTP (Hypertext Transfer Protocol) als sein Hauptkommunikationsmittel. Es kann sowohl als einzelner Server als auch embedded in einer Anwendung verwendet werden. Im Rahmen dieser Bachelorarbeit wurde ein Standalone-Server verwendet.

Blazegraph wird seit 2006 kontinuierlich von Systap, LLC entwickelt und ist unter GPLv2 und einer kommerziellen Lizenz verfügbar. [18]

3.3.2 Praktische Verwendung

3.3.2.1 Docker

```
FROM lyrasis/blazegraph:2.1.4
```

Abbildung 3.9: Blazegraph Dockerfile

Zum Erstellen des Docker Containers wurde im lokalen Dockerfile (Abbildung 3.9) das offizielle Docker Image verwendet [29].

3.3.2.2 Konfiguration

```
blazegraph:
  build: ./dbs/blazegraph
  ports:
    - "9999:8080"
```

Abbildung 3.10: Blazegraph Konfiguration in docker-compose

```
# See README.txt in dbs/blazegraph for details on:
- BLAZEGRAPH_HOST=blazegraph
- BLAZEGRAPH_PORT=8080
- BLAZEGRAPH_USER=
- BLAZEGRAPH_PASS=
```

Abbildung 3.11: Blazegraph Umgebungsvariablen

Für Blazegraph ist über die Konfiguration im Dockerimage hinaus keine Änderung nötig. Über die Umgebungsvariablen in Abbildung 3.11 müssen daher nur die Domain und der Port angegeben werden.

Die Portöffnung im docker-compose Konfigurationsfile in Abbildung 3.11 dient der manuellen Nutzung des Webinterfaces außerhalb der Docker-Container.

3.3.2.3 Datenmodell

Das in Blazegraph verwendete Datenmodell gleicht dem in ArangoDB (siehe Abschnitt 3.1.2.3).

3.3.2.4 Besonderheiten

Blazegraph ist die einzige verwendete Datenbank, mit der manuell über HTTP kommuniziert wird. Mit Blazegraph kommuniziert wird via SPARQL. Daher wurde für Blazegraph keine Java-Bibliothek verwendet und es muss für Evaluationen keine Serververbindung aktiv gehalten werden. Stattdessen wird jede Anfrage an den Server einzeln versendet.

3.3.3 Kommentare

Blazegraph bringt ein intuitives und bei der Entwicklung hilfreiches Web-Interface mit sich. Dieses wurde beim prototypen intensiv verwendet, um Queries zu testen und Daten in der Datenbank zu visualisieren.

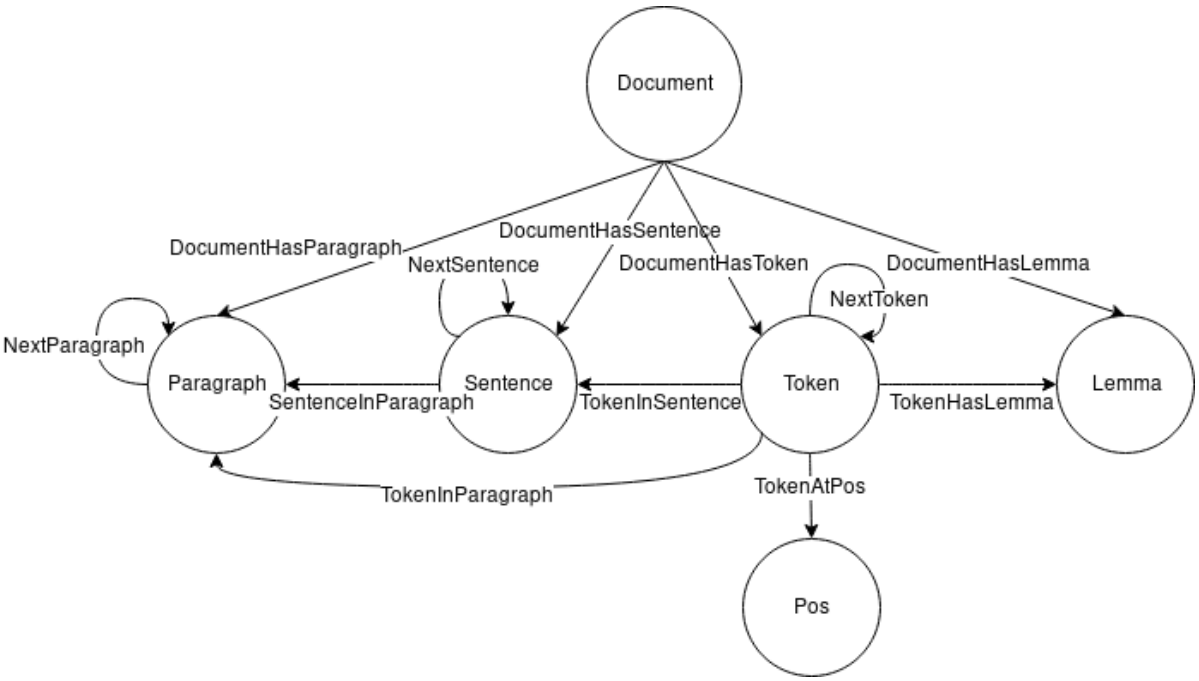

Abbildung 3.12: Struktur des Models in Blazegraph

Die Kommunikation über HTTP macht außerdem das Debuggen leichter, da jedes generierte Query nachvollzogen und separat im Webinterface manuell getestet werden kann.

3.4 Cassandra

3.4.1 Konzepte und Fähigkeiten

Cassandra ist eine NoSQL (Not only SQL) Datenbank mit hoher Scalability und Fehlertoleranz, da Daten unter den richtigen Umständen auf eine vielzahe von selbst-regulierenden Serven verteilt werden können. Die Clusterstruktur von Cassandra ist dezentralisiert, was Ausfälle einzelner Nodes kaschiert.

Cassandra nutzt für Schreib- und Lese-Prozesse ein stark relationales Konzept. Tabellen haben ein vorgegebenes Schema, das aber auf Zeilenbases überschrieben werden kann. Inhalte von Tabellen können nur anhand der vorher festgelegten Primary Keys durchsucht werden, um unvorhersehbare Queryzeiten zu vermeiden.

Dies führt zu starker Redundanz der Daten, um Abfragen nach mehreren Kriterien durchführen zu können; Senkt aber gleichzeitig die Abfragezeit im Gegensatz zu anderen relationalen Datenbanken [10].

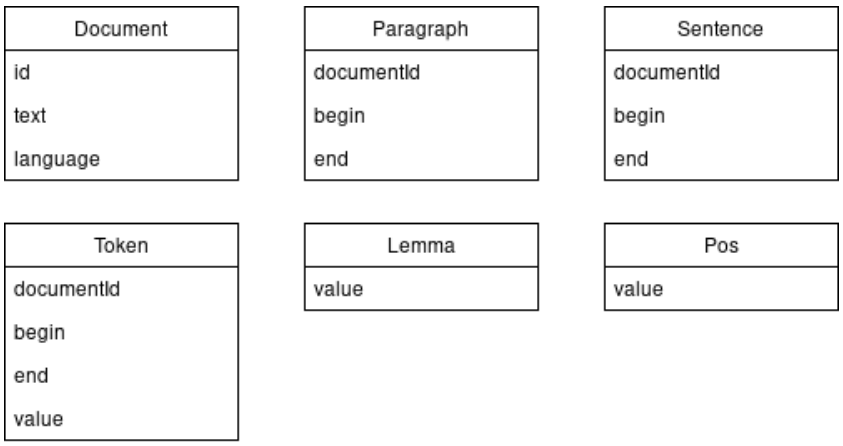

Abbildung 3.13: Felder der Models in Blazegraph

3.4.2 Praktische Verwendung

3.4.2.1 Docker

```
FROM cassandra:3
```

Abbildung 3.14: Cassandra Dockerfile

Zur Erstellung des Docker Containers wurde im lokalen Dockerfile (Abbildung 3.14) das offizielle Docker Image verwendet [23].

3.4.2.2 Konfiguration

```
cassandra:
  build: ./dbs/cassandra
```

Abbildung 3.15: Cassandra Konfiguration in docker-compose

```
# See README.txt in dbs/cassandra for details on:
- CASSANDRA_HOST=cassandra
- CASSANDRA_PORT=9042
- CASSANDRA_USER=cassandra
- CASSANDRA_PASS=cassandra
- CASSANDRA_DB=uimadatabase
```

Abbildung 3.16: Cassandra Umgebungsvariablen

Über die default Konfiguration im verwendeten Dockerfile hinaus waren keine Änderungen erforderlich. Daher werden in der Anwendung die Defaultwerte (zu sehen in Abbildung 3.16) verwendet.

3.4.2.3 Datenmodell

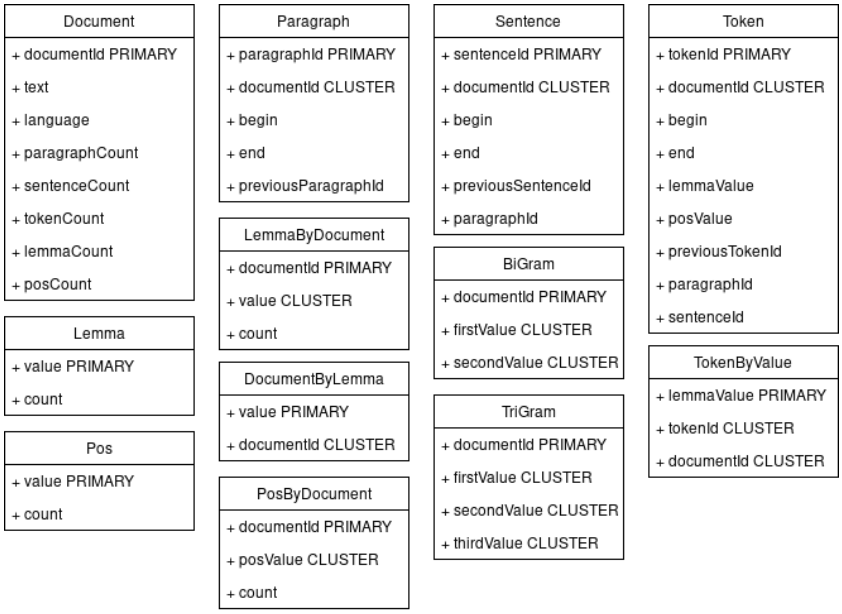

Abbildung 3.17: Felder der Models in Cassandra

Das Cassandra-Model enthält viele Redundanzen. So werden in der `Documents`-Tabelle schon die Anzahlen der enthaltenen `Pargraphs` und anderen Arten Elementen gespeichert, damit diese ohne Relationsverfolgung zugreifbar sind.

Viele Relationen werden beidseitig gespeichert. Es gibt die Tabellen `LemmaByDocument` und `DocumentByLemma`. Erstere ermöglicht das Zählen von Vorkommnissen eines Lemmas in einem Dokumenten oder das Finden aller Lemmata in einem Dokument. Letztere ermöglicht das Zählen aller Dokumente, die ein bestimmtes Lemma enthalten.

Bi- und TriGramme werden schon beim Speichern der Daten erzeugt und separat abgelegt, da sie so nicht im Nachhinein konstruiert werden müssen. Das erhöht die Schreibzeiten und senkt den Aufwand beim Finden der Bi- und TriGramme erheblich.

3.4.2.4 Besonderheiten

Da Cassandra's Struktur keine Joins (siehe Abschnitt 3.7.2.4) und beliebigen Queries erlaubt, mussten viele Daten mehrfach gespeichert werden. Da im Anwendungsfall dieser Arbeit nach dem initialen Einfügen der Dokumente keine weiteren Write-Prozesse nötig sind, erzeugt dieser Umstand keine unbändigbare Komplexität. Es erfordert allerdings erhöhte Aufmerksamkeit beim Einfügen der Daten und bei der Auswahl der Struktur, nach der Daten abgefragt werden sollen.

Es musste daher von Anfang an ein durchdachtes Datenmodell verwendet werden, dass alle möglichen Abfragen befriedigt.

3.4.3 Kommentare

Die hohe Skalierbarkeit und Fehlertoleranz von Cassandra kann im Rahmen dieser Arbeit leider nicht genutzt werden, da für jede Datenbank nur eine Instanz verwendet wird.

Die Ergebnisse des Benchmarks auf Cassandra sollten im Rahmen dessen gesehen werden und können wahrscheinlich im richtigen Setup weiter verbessert werden.

3.5 Lucene & Solr

3.5.1 Konzepte und Fähigkeiten

Lucene [12] ist ein javabasiertes Kommandozeilentool für sowohl Indexierungs- und Suchtechnologie, als auch Rechtschreibeprüfung und Textanalyse. Solr [13] ist ein Suchserver aufgebaut auf Lucene mit XML/JSON HTTP API.

Im Rahmen dieser Arbeit wurde Solr verwendet, um die Interaktion mit Lucene zu vereinfachen und die Client-Server-Struktur aufrecht zu erhalten, die für alle anderen Datenbanken verwendet wird.

Lucene kann auf moderner Hardware bis zu 150GB (GigaByte) Daten pro Stunde indexieren und benötigt dabei nur bis zu 1MB (MegaByte) RAM (Random Access Memory). Der fertige Index benötigt dabei etwa 20-30% des Speichers der Originaltexte. [11]

3.5.2 Praktische Verwendung

3.5.2.1 Docker

```
FROM solr:latest

WORKDIR /opt/solr/server/solr/
ADD uimadatabase /opt/solr/server/solr/uimadatabase
USER root
RUN chown -R solr /opt/solr/server/solr
USER solr
```

Abbildung 3.18: Solr Dockerfile

Zur Erstellung des Docker Containers wurde im lokalen Dockerfile (Abbildung 3.18) das offizielle Docker Image verwendet [27] und ein Ordner mit Konfigurationsdateien für den Verwendeten Solr-Kern hinzugefügt.

3.5.2.2 Konfiguration

```
solr:
  build: ./dbs/solr
  ports:
    - "8983:8983"
```

Abbildung 3.19: Solr Konfiguration in docker-compose

```
# See README.txt in dbs/solr for details on:
- SOLR_HOST=solr
- SOLR_PORT=8983
- SOLR_CORE=uimadatabase
```

Abbildung 3.20: Solr Umgebungsvariablen

Die Konfiguration des Solr Servers erfolgt hauptsächlich über den im Dockerfile (Abbildung 3.18) hinzugefügten Ordner mit Konfigurationsdateien. [5]

Darunter die wichtigste ist `solr/uimadatabase/conf/schema.xml`. Diese enthält die Konfiguration für die Index-Felder, die in Lucene angelegt werden.

Dort werden Defaultwerte verwendet und Indexe hinzugefügt, die BiGramme und TriGramme über alle Dokumente in der Datenbank aufbauen. Besagte Indexe nutzen Shinglefilter [1], grob Implementiert nach einer Anleitung von Toby Hobson. [4]

3.5.2.3 Datenmodell

Lucene indexiert Dateien und durchsucht diese nach eigenem Format. Daher ist kein Datenmodell von Seiten dieser Arbeit vonnöten.

3.5.2.4 Besonderheiten

Da Lucene keine Datenbank, sondern ein Indexierungssystem ist, ist es nur für einen Teil der geforderten Operationen geeignet. Genauer ist es nur geeignet für das Finden von Bi- und TriGrammen. Daher wird Lucene nur in den Evaluationen für diese Berücksichtigt werden. Es wird jedoch kurz auf die Zeiten für das Einfügen von Dateien eingegangen werden.

Bei den Ausführungen der Evaluationen war es nicht möglich, mehr als 99 Dokumente in Lucene zu Indexieren. Das Einfügen des 100. Dokuments konnte auch nach ausgedehnter Wartezeit nicht vollendet werden und wurde daher abgebrochen. Daher wird Lucene auch in den geeigneten Fällen nur bis zu 99 Dokumenten analysiert.

Warum dies passiert, konnte im Zeitrahmen dieser Bachelorarbeit nicht erarbeitet werden.

3.6 MongoDB

3.6.1 Konzepte und Fähigkeiten

MongoDB ist ein Document-Store auf der Basis von BSON (Binary JSON). Es speichert schemafrei flexible Daten in JSON-artigen Dokumenten. Die Struktur und der Inhalt dieser Dokumente kann im Laufe der Zeit frei verändert werden.

Diese Datenstruktur ermöglich das direkte Abbilden von Datenbankinhalten auf Objekte der verwendeten Programmiersprache.

MongoDB unterstützt Queries nach „Muster“. D.h. ein Musterdokument wird angegeben, das verschiedene Filteroptionen enthält und es werden alle Dokumente gefunden, die der Struktur des Musters entsprechen. Auch kann auf diese Weise Echtzeitaggregation und -indexierung durchgeführt werden.

MongoDB ist open source und wird von MongoDB, Inc. entwickelt und supportet.

3.6.2 Praktische Verwendung

3.6.2.1 Docker

```
FROM mongo:3.6
```

Abbildung 3.21: MongoDB Dockerfile

Zur Erstellung des Docker Containers wurde im lokalen Dockerfile (Abbildung 3.21) das offizielle Docker Image verwendet [24].

3.6.2.2 Konfiguration

```
mongodb:
  build: ./dbs/mongodb
```

Abbildung 3.22: MongoDB konfiguration in docker-compose

```
# See README.txt in dbs/mongodb for details on:
- MONGODB_HOST=mongodb
- MONGODB_PORT=27017
- MONGODB_USER=
- MONGODB_PASS=
- MONGODB_DB=uimadatabase
```

Abbildung 3.23: MongoDB Umgebungsvariablen

Über die default Konfiguration im verwendeten Dockerfile hinaus waren keine Änderungen erforderlich. Daher werden in der Anwendung die Defaultwerte (zu sehen in Abbildung 3.23) verwendet.

3.6.2.3 Datenmodell

```
Document:
{
  _id,
  text,
  language,
  sentenceCount,
  tokenCount,
  lemmata: [
    {
      value,
      count
    }
  ],
  pos: [
    {
      value,
      count
    }
  ]
}
```

```
Paragraph:
{
  _id,
  documentId,
  begin,
  end,
  previousParagraphId
}

Sentence:
{
  _id,
  documentId,
  begin,
  end,
  previousSentenceId,
  paragraphId
}
```

```
Token:
{
  _id,
  documentId,
  begin,
  end,
  lemmaValue,
  posValue,
  previousTokenId,
  paragraphId,
  sentenceId
}
```

```
Lemma:
{
  value,
  count
}

Pos:
{
  value,
  count
}
```

```
BiGram:
{
  documentId,
  firstValue,
  secondValue
}

TriGram:
{
  documentId,
  firstValue,
  secondValue
}
```

Abbildung 3.24: Felder der Models in MongoDB

Da MongoDB ähnlich wie Cassandra keine Joins (siehe Abschnitt 3.7.2.4) unterstützt (und daher nicht verschiedene Objekte während einer Abfrage miteinander verknüpfen kann), ist das Schema ähnlich dem Schema für Cassandra (siehe 3.4.2.3) gewählt.

Durch die flexibleren Abfragen, die MongoDB erlaubt, enthält das verwendete Schema jedoch weniger Redundanzen und ist leichter zu verwalten.

3.6.2.4 Besonderheiten

MongoDB - als reiner Documentstore - erlaubt verschachtelte Objekte. Dadurch kann in diesem Schema in jedes `Document` eine Liste der enthaltenen `Lemma-Value`s und `Pos-Value`s eingefügt werden. Dies erleichtert das Durchsuchen nach z.B. Lemmata innerhalb eines spezifischen Dokumentes und ersetzt weitgehend Relationen zwischen Elementen.

3.7 MySQL

3.7.1 Konzepte und Fähigkeiten

MySQL Server ist eine Datenbank mit Unterstützung für Multithreading, Multi-User-Setups und robustes SQL (Structured Query Language) [31], geschrieben in C++ [32]. Es implementiert Transaktionsprotokolle für konsistente Datenbearbeitung und beliebige Indexe auf beliebigen Daten.

MySQL speichert Daten stark schematisiert. Ein Schema muss vor dem ersten Speichern von Daten festgelegt werden und von diesem kann nicht abgewichen werden. Das

Konzept der MySQL-Schemata basiert auf klassischer Tabellenverarbeitung. Ein Schema legt Spalten und deren Typen fest und jeder Datensatz stellt eine Zeile dar.

Diese Zeilen können effizient indexiert, durchsucht, gruppiert und manipuliert werden.

Statische Schemata erfordern gute Planung des Datenmodels. Im Gegensatz zu Cassandra (3.4.2.3) können jedoch Daten auch abgefragt werden, ohne dass die Vewendung eines Indexes erzwungen wird. Daher können Daten mit weniger Redundanz und weniger manueller Aufmerksamkeit beim Schreiben gepflegt werden.

MySQL gehört der Oracle Corporation und wird von dieser entwickelt [31].

3.7.2 Praktische Verwendung

3.7.2.1 Docker

```
FROM mysql:5
```

Abbildung 3.25: MySQL Dockerfile

Zur Erstellung des Docker Containers wurde im lokalen Dockerfile (Abbildung 3.25) das offizielle Docker Image verwendet [25].

3.7.2.2 Konfiguration

```
mysql:
  build: ./dbs/mysql
  environment:
    - MYSQL_ROOT_PASSWORD=root
    - MYSQL_DATABASE=uimadatabase
```

Abbildung 3.26: MySQL Konfiguration in docker-compose

```
# See README.txt in dbs/mysql for details on:
- MYSQL_HOST=mysql
- MYSQL_PORT=3306
- MYSQL_USER=root
- MYSQL_PASS=root
- MYSQL_DBNAME=uimadatabase
```

Abbildung 3.27: MySQL Umgebungsvariablen

Für MySQL müssen sowohl ein Rootpasswort als auch ein Datenbankname gesetzt werden (siehe Abbildung 3.26).

Diese werden entsprechend in der Anwendung verwendet (siehe Abbildung 3.27).

3.7.2.3 Datenmodell

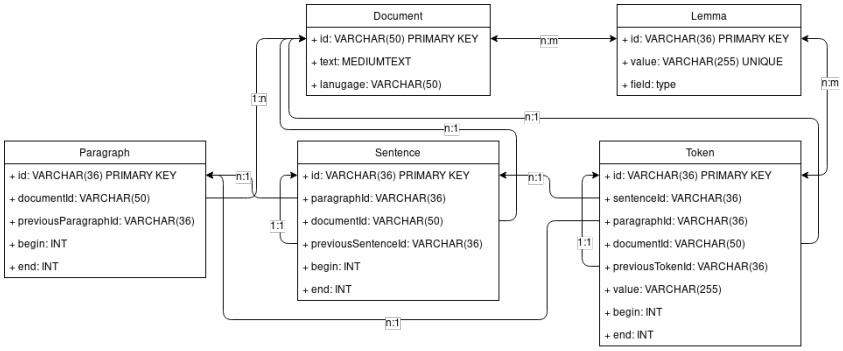

Abbildung 3.28: MySQL Model

Jeder Typ Element hat hier eine eigene Tabelle. POSs sind dabei die Ausnahme und werden implizit in Tokens mitverwaltet.

Nicht im Model zu sehen sind n:m Beziehungstabellen zwischen je `Document` - `Lemma` und `Token` - `Lemma`.

Das Model spiegelt die UIMA-geparste hierarchische Struktur der Inputdokumente wider. Dies senkt die Komplexität des Codes und vermeidet Fehler in der Handhabung des Models. Beziehungen stehen in dieser Implementation im Vordergrund und für nahezu jede Operation ist mindestens ein Join notwendig. So müssen zum Beispiel zum Zählen der `Sentence`s in einem `Document` die beiden Tabellen miteinander verbunden werden.

Es können durch die Eigenreferenzen von `Paragraph`, `Sentence` und `Token` mithilfe von mehreren Joins BiGramme und TriGramme ausgelesen werden.

3.7.2.4 Besonderheiten

MySQL beherrscht im Gegensatz zu z.B. Cassandra `Joins`. Ein Join verbindet die Daten zweier Tabellen basierend auf Spalten mit korrelierenden Inhalten.

Durch Joins können Daten über mehrere Tabellen verteilt und miteinandern in Verbindung gebracht werden. So können Teile von Graphen oder hierarchischen Strukturen simuliert werden.

Während dies für die Konzeption des Models von Vorteil ist, bringt es Performancenachteile mit sich. Ein Join bildet ein (optimiertes) Kreuzprodukt über zwei Tabellen und hat daher eine Laufzeit von `O(n*m)`, wobei n die Anzahl Datensätze in der ersten und m die Anzahl Datensätze in der zweiten Tabelle ist. D.h. je mehr Daten in der Datenbank sind, desto ineffizienter werden Beziehungen abgefragt [33].

3.8 Neo4j

3.8.1 Konzepte und Fähigkeiten

Neo4j ist eine Graphdatenbank, bei der Beziehungen im Vordergrund stehen. Es nutzt die Abfragesprache Cypher, welche für Graphoperationen optimiert ist.

Knoten in Neo4j Graphen haben die Form von Dokumenten, mit beliebigen Key/Value Paaren ohne Schema - vergleichbar mit MongoDB- oder ArangoDB-Dokumenten. Knoten werden über Kanten miteinander verbunden, welche ebenfalls beliebige Key/Value Paare an sich haben können. Sowohl Knoten als auch Kanten können Label haben und nach diesen gefiltert werden.

Neo4j ist darauf optimiert, Beziehungen von Knoten abzufragen. Während in Neo4j ein Join über vier Tabellen hinweg schnell Server überlastet, kann Neo4j beliebige Distanzen zwischen Nodes in vergleichbar kurzer Zeit überwinden [36].

3.8.2 Praktische Verwendung

3.8.2.1 Docker

```
FROM neo4j:3.3
```

Abbildung 3.29: Neo4j Dockerfile

Zur Erstellung des Docker Containers wurde im lokalen Dockerfile (Abbildung 3.29) das offizielle Docker Image verwendet [26].

3.8.2.2 Konfiguration

```
neo4j:
  build: ./dbs/neo4j
  environment:
    - NEO4J_AUTH=none
```

Abbildung 3.30: Neo4j Konfiguration in docker-compose

```
# See README.txt in dbs/neo4j for details on:
- NEO4J_HOST=neo4j
- NEO4J_PORT=7687
- NEO4J_USER=
- NEO4J_PASS=
```

Abbildung 3.31: Neo4j Umgebungsvariablen

Im offiziellen Neo4j Dockerfile ist das Setzen von Credentials umständlich [20]. Daher wird über die Environment Variable `NEO4J_AUTH` der Authentifizierungsmechanismus deaktiviert und in der Anwendung eine unsichere Verbindung verwendet. Dies ist im Rahmen einer Offline-Evaluationen auf einem abgetrennten Server kein Risiko, da ohnehin nur autorisierte Personen Zugriff auf besagten Server haben.

3.8.2.3 Datenmodell

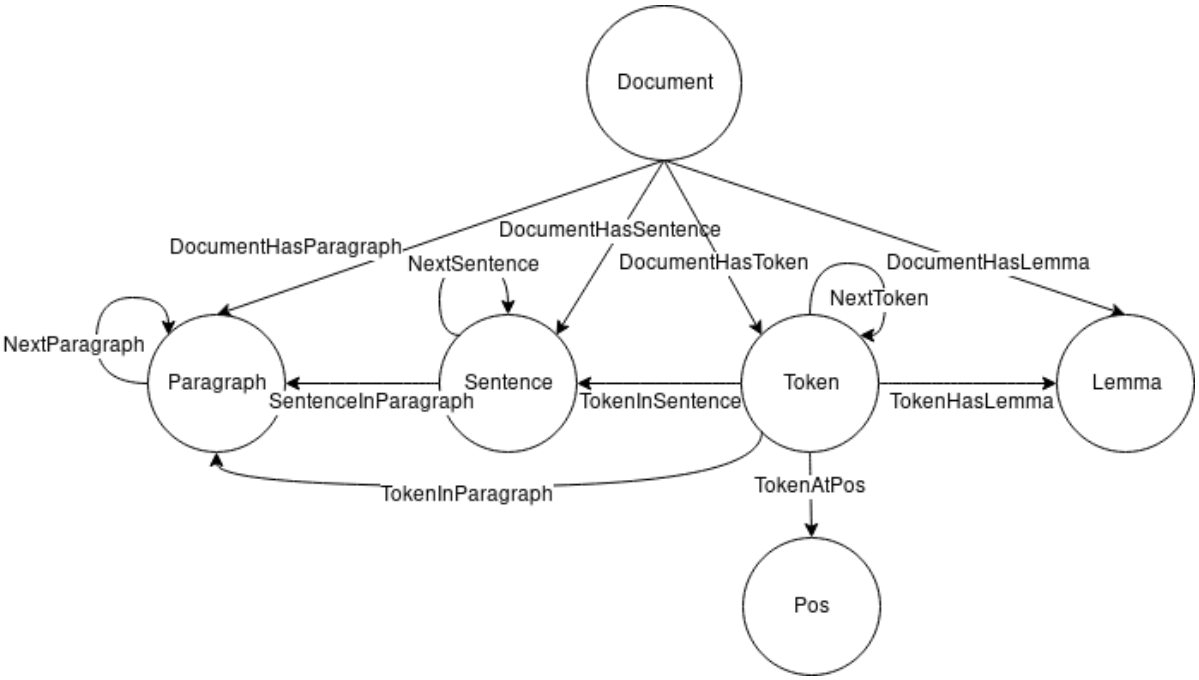

Abbildung 3.32: Struktur des Models in Neo4j

Da die Struktur der Daten in Neo4j der Struktur in ArangoDB gleicht, siehe für eine Erläuterung der Grafiken Abschnitt 3.1.2.3.

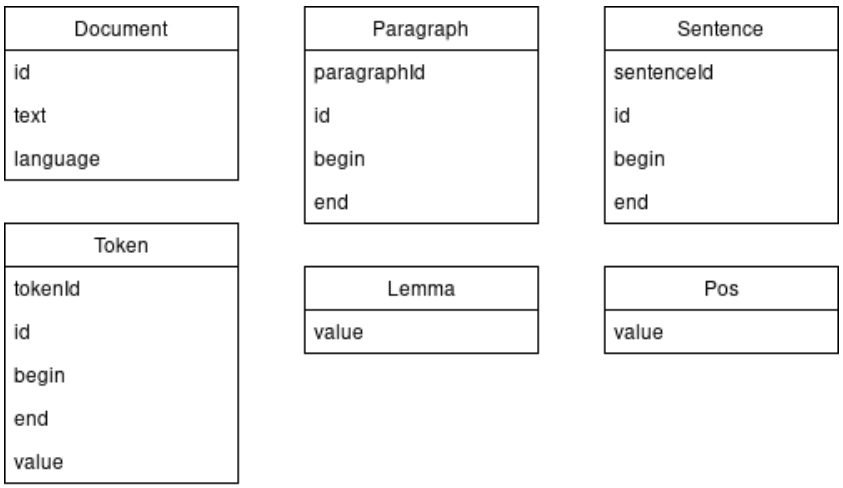

Abbildung 3.33: Felder der Models in Neo4j

3.8.2.4 Besonderheiten

In seiner Graphstruktur weist Neo4j zwar starke Ähnlichkeiten zu ArangoDB auf, ist aber grundsätzlich verschieden zu benutzen. Während ArangoDBs AQL imperativ ist, ist Neo4js Cypher näher an einer Ausdruckssprache.

Daten werden gefunden, indem ein Muster vorgegeben wird, dem die gesuchten Daten entsprechen. Hierbei ist es nicht nötig, speziell einzelne Collections oder Tabellen anzusprechen. Es ist gar möglich, alle Daten einer Anwendung in einem einzigen Graphen zu speichern, da auch große Graphen durch die Verwendung von Labels und Mustern leicht nach kontextspezifischen Datenmustern durchsucht werden können.

Insofern ist Neo4j gut geeignet für die Konzipierung einer Anwendung, in der die Struktur der Daten im Vordergrund steht.

Im Falle dieser Bachelorarbeit ist das von Nutzen, da Beziehungen zwischen Dokumenten, Tokens und Lemmata leicht gefunden und analysiert werden können.

4 Ergebnisse

Die Ergebnisse der Evaluationen liegen vor als JSON-Dateien. In diesen sind für jede Datenbank für jede ausgeführte Evaluation Zeitwerte gespeichert für die minimale, maximale und durchschnittliche Laufdauer der ausgeführten Operationen.

Aus diesen JSON-Dateien wurden Graphen generiert, in welchen die jeweils durchschnittliche Laufdauer verwendet wird. Die Graphen zeigen die Anzahl der verwendeten Dokumente an der X-Achse und die benötigte Zeit für die jeweilige Evaluation an der Y-Achse in Millisekunden.

Die exakten Werte in den Tabellen sind auf Millisekunden gerundet.

Alle Ergebnisse sind im Anhang dieser Arbeit zu finden. Für die Interpretation der Ergebnisse wird exemplarisch eine Auswahl der Ergebnisse verwendet, wobei auf jede der fünf Kategorien (siehe Abschnitt 2.1) eingegangen wird.

4.1 Kommentar zum Verlauf der Evaluation

Die gesamte Evaluation wurde durchgeführt für je 2, 5, 10, 25, 99, 150 und 250 Dateien. Die Inputsets wurden jeweils zufällig aus den zur Verfügungen stehenden Dateien ausgewählt und waren identisch für alle Datenbanken.

In den Graphen sind Höhepunkte zu sehen bei der 10-Dateien-Marke. An dieser Stelle wurden vermutlich besonders große Dateien ausgewäht, die die Statistik verzerren.

Außerdem ist die Evaluation für Neo4j bei großen Dateimengen (150+) und für BaseX und Blazegraph bei 250 Dateien wiederholt abgestürzt, weswegen dort genaue Werte fehlen.

4.2 Schreiben

Wie in Abbildung 4.1 zu sehen ist hat ArangoDB konsistent die schlechteste Leistung und der sichtbare Anfang der Neo4j-Kurve wird mit wachsendem Inputset zunehmend schlechter.

Diese beiden haben daher wohl die längste Einrichtungszeit beim Einpflegen neuer Daten und weisen geringeren Nutzen auf, falls sich das verwendete Datenset oft ändert oder oft neue Daten hinzugefügt werden.

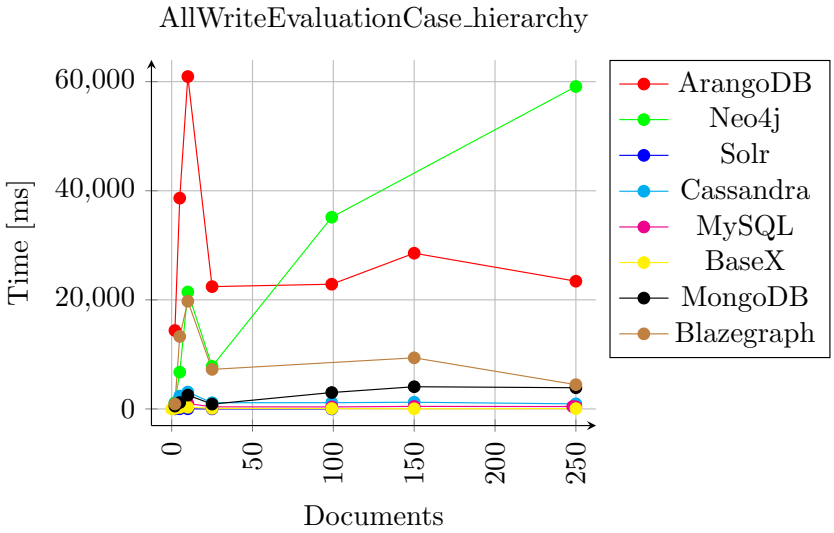

Abbildung 4.1: Schreiben von kompletten Dokumenten in die Datenbank

Die Unterschiede zwischen den restlichen Datenbanken sind aus dem Graphen nicht exakt zu erkennen. Daher ziehen wir die genauen Werte in Abbildung 4.1 zu rate.

ArangoDB	14381	38648	60939	22427	22865	28552	23419
Neo4j	1161	6747	21449	7814	35147	-	-
Solr	16	39	29	14	7	-	-
Cassandra	996	2358	3098	1137	1149	1232	931
MySQL	363	700	996	386	374	467	384
BaseX	104	282	311	106	72	39	66
MongoDB	530	1176	2532	892	3012	4073	3893
Blazegraph	916	13284	19746	7254	9375	4455	-

Tabelle 4.1: Werte zu Schreibprozessen (vgl. Abbildung 4.1)

In den exakten Werten ist nun eindeutig, dass auch bei großen Dateimengen BaseX führt und annähernd konstante Zeit pro Datei benötigt.

Das erklärt sich, wenn man bedenkt, dass BaseX auf XML-Dateien basiert und der Schreibprozess bei BaseX daraus besteht, die vorhandenen XML-Dateien auf den Server zu kopieren. Hierbei ist die Einfügezeit ausschließlich von der Dateigröße und nicht von der Größe des Inputsets abhängig. Vergleichbar nah an der Spitze sind Solr und MySQL.

4.3 Lesen

Abbildung 4.2 entnehmen wir, dass Neo4j sehr bei großen Dateienmengen sehr lange braucht, um `Documents` zu rekonstruieren.

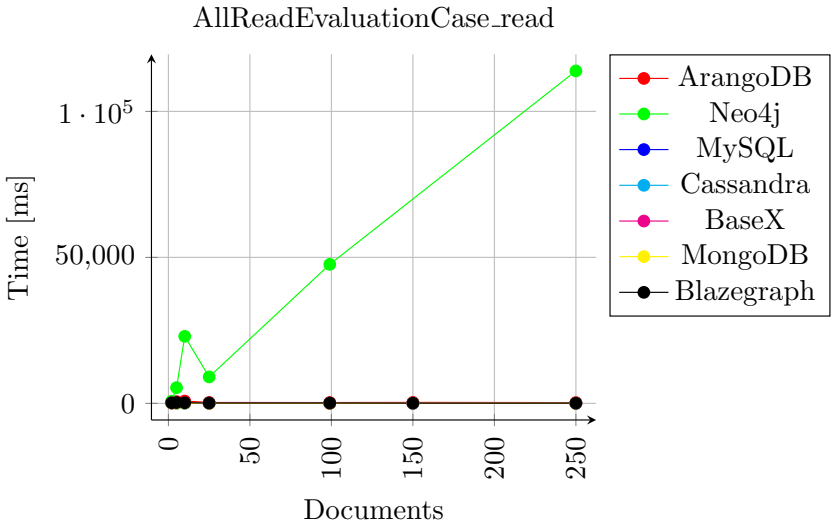

Abbildung 4.2: Lesen von kompletten Dokumenten aus der Datenbank

Auch hier ist die Zeit-Skala so verzerrt, dass wir uns die genauen Werte ansehen müssen.

ArangoDB	389	593	802	302	275	335	266
Neo4j	748	5388	22962	9064	47617	113867	
MySQL	143	72	49	18	7	7	5
Cassandra	139	84	62	24	11	13	7
BaseX	103	225	215	71	54	130	47
MongoDB	143	71	54	24	32	52	65
Blazegraph	200	277	203	103	111	69	68

Tabelle 4.2: Werte zu Leseprozessen (vgl. Abbildung 4.2)

Alle Datenbanken außer Neo4j weisen bei den verwendeten Datensets keine nennenswerte Steigerung der Lesezeiten auf.

Falls es auf jede Millisekunde ankommt, sollte MySQL oder Cassandra verwendet werden; Es ist aber alles außer Neo4j für effizientes Lesen von Dokumenten aus der Datenbank geeignet.

Unter der Annahme, dass der Höhepunkt an der 10-Dateien-Marke aus sehr großen Einzel-Dateien entsteht, ergibt sich aus Abbildung 4.2 für MySQL, Cassandra und MongoDB der vorteilhafte Schluss, dass diese auch sehr große Dokumente schnell konstruieren können.

4.4 Durchsuchen

In Abbildung 4.3 sind die Zeiten für das Finden der IDs alle `Documents` zu finden. Dies wird unter anderem verwendet, um beim Auslesen über die `Documents` zu iterieren, um diese einer UIMA-Pipeline zur Verfügung zu stellen.

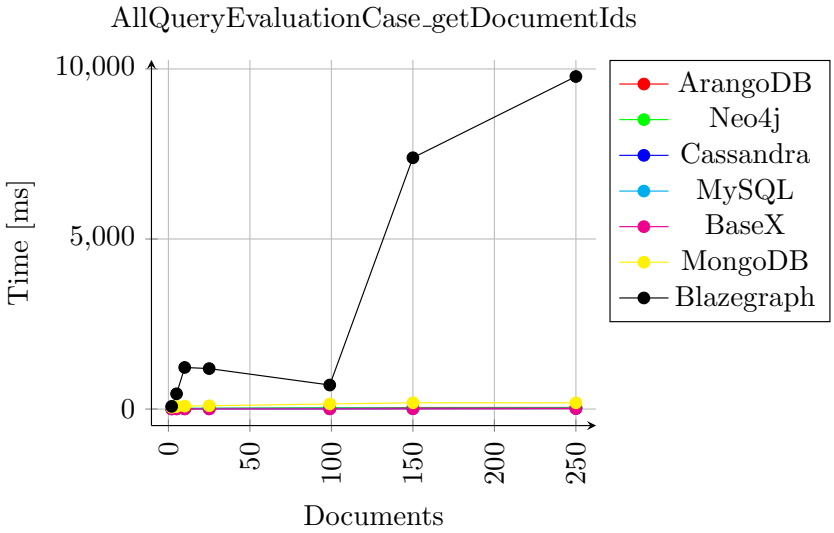

Abbildung 4.3: Finden aller `Document-Ids`

Hier skaliert Blazegraph besonders schlecht, während die restlichen Datenbanken sich gemeinsam am unteren Rand halten. Werfen wir daher wieder einen Blick auf die exakten Werte.

ArangoDB	10	11	12	18	32	45	47
Neo4j	16	18	19	20	38	24	28
Cassandra	7	8	9	9	8	11	15
MySQL	21	21	22	22	25	25	27
BaseX	3	4	4	5	7	8	14
MongoDB	51	73	87	97	150	184	183
Blazegraph	79	449	1223	1190	707	7389	9777

Tabelle 4.3: Werte zum Finden aller `DocumentID`s (vgl. Abbildung 4.3)

MySQL und Neo4j zeigen hier die geringste Steigerung bei wachsenden Datensets.

BaseX und Cassandra zeigen Anzeichen einer hohen Steigung in ihren letzten zwei Werten, sind aber in ihren absoluten Werten an der Spitze. Hier müsste das Datenset weiter vergrößert werden, um diese Trends näher zu beleuchten.

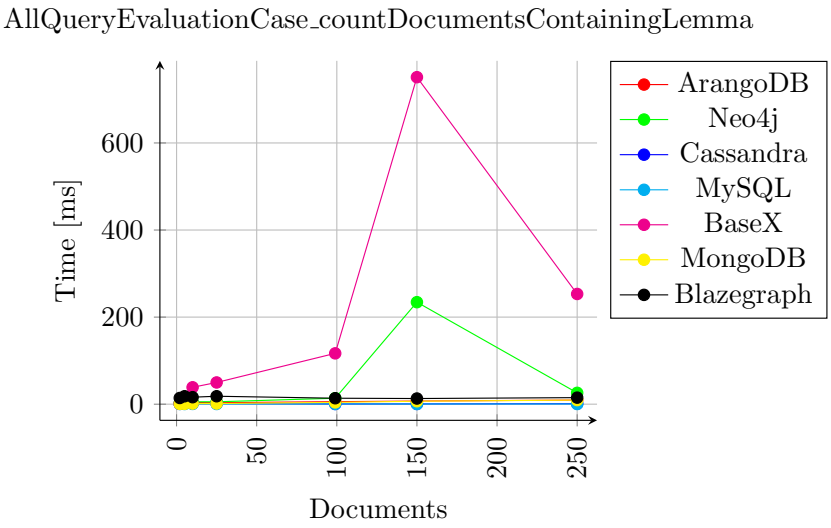

Abbildung 4.4: Zählen aller `Documents`, die ein bestimmtes Lemma enthalten

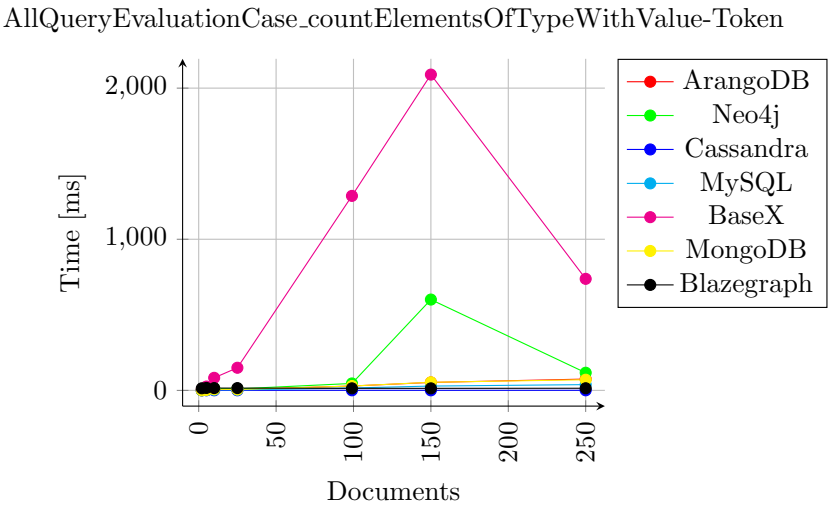

Abbildung 4.5: Zählen aller `Tokens`, die einen bestimmten Begriff enthalten

In Abbildung 4.4 wird die Dauer für das Zählen von `Documents` gezeigt, die ein bestimmtes Lemma enthalten.

In Abbildung 4.5 wird die Dauer für das Zählen von Tokens mit einem bestimmten Wert gezeigt.

Die beiden Graphen zeigen eine sehr ähnliche Form. Insbesondere den Hochpunkt von Neo4j und BaseX bei 150 Dateien und das Absenken bei 250 Dateien. Dies hängt wahrscheinlich mit den Abstürzen der beiden Datenbanken beim Einfügen der Dateien zusammen, weswegen diese Werte für die Analyse als Outlier betrachtet und außer Acht gelassen werden.

Unabhängig von besagten Outliern sind BaseX und Neo4j offensichtlich die langsamsten Datenbanken bei den besagten Operationen, während die restlichen Datenbanken

näher beieinander liegen.

Betrachten wir hierzu die exakten Daten zu Abbildung 4.5, da diese repräsentativ für alle Abfragen sind, in denen Elemente einer bestimmten Art mit einem bestimmten Wert gesucht werden und diese Art Abfrage in verschiedenen Formen durchgeführt wird.

Dazu mehr in den Ergebnissen im Anhang.

ArangoDB	1	2	6	6	29	52	75
Neo4j	1	5	12	10	46	601	117
Cassandra	1	1	1	1	1	1	1
MySQL	1	1	4	4	16	29	38
BaseX	6	25	83	150	1287	2089	738
MongoDB	1	3	7	6	29	53	70
Blazegraph	14	16	15	15	13	14	14

Tabelle 4.4: Werte zum Zählen aller `Token`s mit bestimmtem Wert (vgl. Abbildung 4.5)

Hier ist Cassandra offensichtlich der ungeschlagene Sieger mit konstant 1ms Abfragezeit.

Blazegraph ist in seinen Ergebnissen ebenfalls nahezu konstant, während die restlichen Datenbanken Steigungen bei größeren Datensets aufweisen.

Bei einem Fokus auf Operationen, die Elemente zählen, ist daher Cassandra sehr zu empfehlen.

4.5 Berechnen

In den folgenden Graphen zeigen Neo4j und BaseX erneut Ausreißer vergleichbar mit Abbildungen 4.4 und 4.5 und werden entsprechend behandelt.

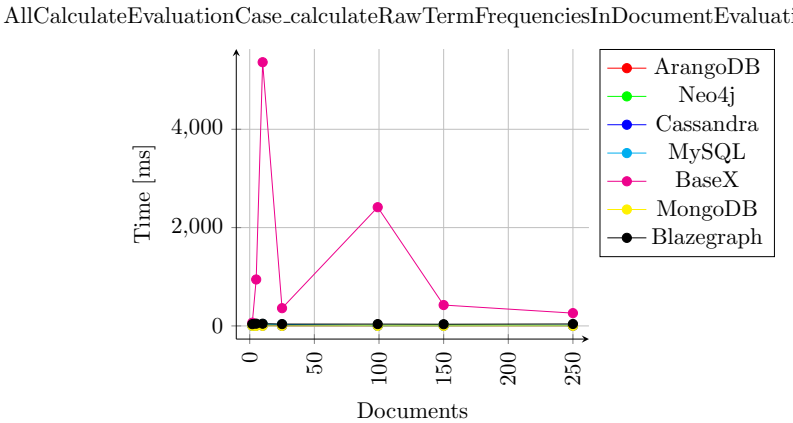

Abbildung 4.6: Finden der Anzahl der Vorkommnisse jedes `Lemma`s in einem Dokument

In Abbildung 4.6 ist BaseX offensichtlich der Verlierer. Die anderen Datenbanken müssen genauer betrachtet werden.

ArangoDB	8	12	16	6	7	6	8
Neo4j	24	27	25	11	14	18	11
Cassandra	5	8	5	3	2	4	2
MySQL	5	15	38	32	5	5	13
BaseX	65	945	5364	361	2415	425	260
MongoDB	2	1	1	1	1	1	1
Blazegraph	36	46	45	38	37	35	40

Tabelle 4.5: Werte zum Berechnen der rohen Term-Frequenzen (vgl. Abbildung 4.6)

Die Betrachtung der genauen Werte in Abbildung 4.5 zeigt nun MongoDB als den klaren Favoriten beim Zählen der rohen Term-Frequenzen, mit Cassandra und ArangoDB knapp im Anschluss.

Es zeigen jedoch alle Datenbanken eine relativ konstante Abfragezeit bei variierender Inputset-Größe. Schwankungen hierbei erklären sich erneut durch den Ausreißer der Dateigrößen an der 10-Dateien-Marke.

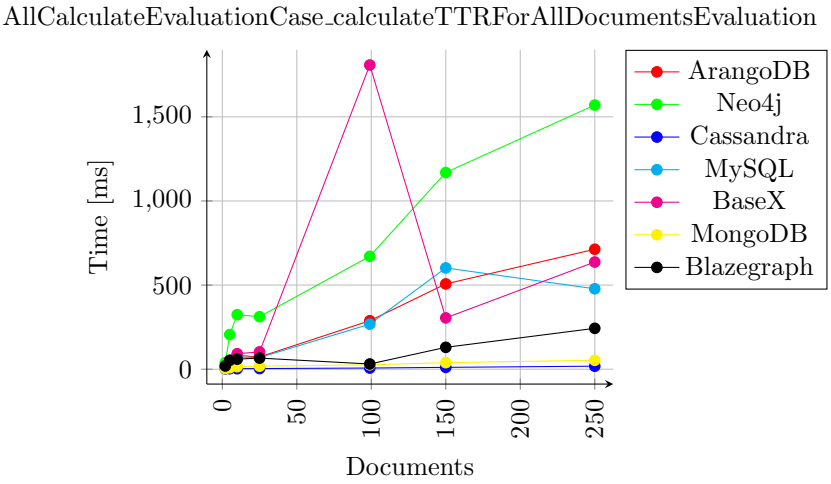

Abbildung 4.7: Berechnen der TTR für alle Dokumente

Abbildung 4.7 zeigt die Abfragezeiten für die Berechnung der Type-Token-Ratio aller vorhandenen `Documents`.

BaseX scheint hier wieder am schlechtesten abzuschneiden. Dies kann aber aufgrund der Abstürze bei großen Inputsets nicht klar eingeschätzt werden.

Darauf folgen Neo4j, ArangoDB und MyQSL mit sowohl den höchsten Werten, als auch signifikanten Steigungen bei wachsendem Inputset.

MongoDB und Cassandra halten sich annähernd konstant, wobei Cassandra die besten absoluten Werte zeigt.

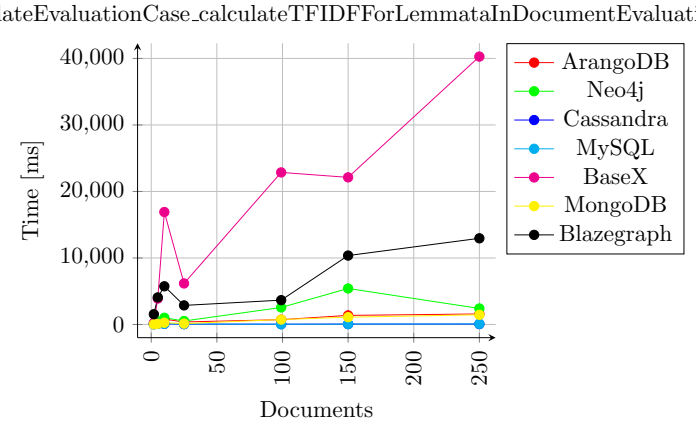

Abbildung 4.8: Berechnen der TFIDF für alle Lemmata in einem bestimmten Dokument

In Abbildung 4.8 sind die Abfragezeiten für die Berechnung der TFIDF (Term-Frequency/Inverse-Document-Frequency) aller Lemmata in einem einzelnen Dokument verbildlicht.

Erneut schneidet BaseX besonders schlecht ab, mit Blazegraph und Neo4j in der Mitte.

Die unteren Werte erfordern wieder Detailbetrachtung.

ArangoDB	89	365	806	380	728	1367	1589
Neo4j	120	435	1014	526	2560	5419	2409
Cassandra	45	111	97	53	50	63	63
MySQL	35	77	111	70	40	59	95
BaseX	227	3905	16909	6172	22864	22121	40283
MongoDB	33	108	264	140	743	1102	1457
Blazegraph	1548	4071	5744	2872	3661	10366	12958

Tabelle 4.6: Werte zum Berechnen der TFIDF der `Lemmata` in einem `Document` (vgl. Abbildung 4.8)

Cassandra und MySQL zeigen wieder konstante Charakteristiken und Cassandra führt in den absoluten Werten. MongoDB und ArangoDB hingegen zeigen Wachstum bei größeren Inputsets.

4.6 Bi- und TriGramme

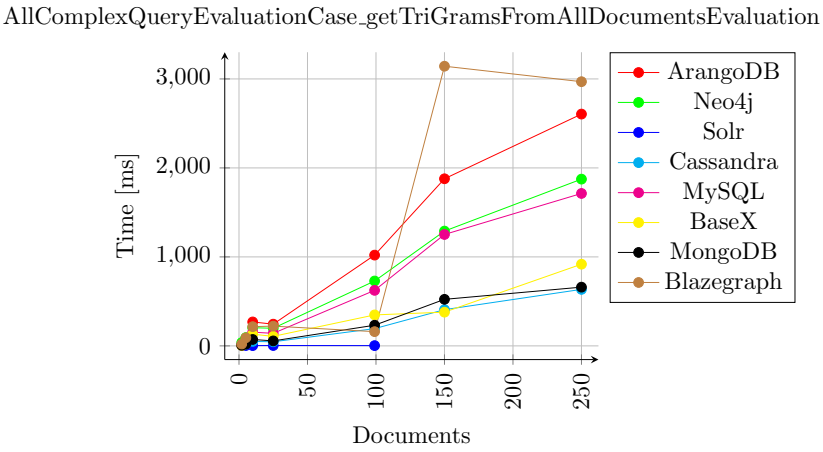

Abbildung 4.9: Finden der TriGramme in allen Dokumenten

Abbildung 4.9 zeigt die Abfragezeiten für TriGramme über alle `Documents`.

Dies ist die einzige Evaluation, in der Solr berücksichtigt werden kann und es ist offensichtlich an der Spitze.

Alle anderen Datenbanken zeigen starkes Wachstum bei wachsenden Inputsets; Solr jedoch bleibt konstant über die fünf messbaren Inputsets.

Abgesehen von Solr sind unter den besseren Datenbanken hier Cassandra, MongoDB und BaseX.

5 Zusammenfassung und Ausblick

In den Abschnitten 4.4 und 4.5 hat sich BaseX konstant als eine der schlechtesten Datenbanken herausgestellt. Beim Schreiben von Dateien in die Datenbank und beim Finden von TriGrammen dagegen hat BaseX gute bis herausragende Ergebnisse gezeigt.

Beim Finden der TriGramme wird es aber von Solr stark unterboten und verliert damit jegliche Kompetitivität.

Solr ist zwar nur für den Zweck des Bi- und TriGram-Findens geeignet, ist in diesen aber ungeschlagen.

Falls sich das Problem des 100-Dateien-Limits beheben lässt, ist Solr mit Abstand die beste Wahl für den speziellen Anwendungsfall des Bi- und TriGram-Findens.

ArangoDB hatte Einstiegsschwierigkeiten, da es zum Einfügen der Inputdateien in die Datenbank außergewöhnlich lange gebraucht hat (siehe Abbildung 4.1). Bei dem Rest der Evaluationen mit Ausnahme der TTR-Berechnung und des Bi- und TriGram-Findens hat sich ArangoDB jedoch stabil im Mittelfeld gehalten und wirkt daher wie ein guter Allrounder.

Die Leichtigkeit der Implementation des Datenmodells im `ArangoDBQueryHandler` spricht für ArangoDB als eine gut wartbare und praxisgeeignete Datenbank.

Neo4j zeigte wie ArangoDB Schwierigkeiten beim Schreiben der Dateien in die Datenbank; Hat diese Probleme aber ebenfalls beim Lesen ganzer Dokumente (siehe Abbildung 4.2).

Dies in Kombination mit den Abstürzen bei hohen Dateimengen und Ausreißern in Abbildungen 4.4, 4.5 und 4.7 macht Neo4j zu einer durchwachsenen und unzuverlässigen Datenbank im Rahmen des Anwendungszweckes dieser Arbeit.

Während sich die Abstürze mit weiterer Forschung vermutlich beheben lassen, wirken die Laufzeiten nicht zukunftsweisend.

MySQL hat einen Ausreißer beim Berechnen der TTRs (siehe Abbildung 4.7) und ist im oberen Mittelfeld bei der Berechnung der Bi- und TriGramme.

Abgesehen davon halten sich MySQL, MongoDB und Cassandra durchgehend unter den Top-Performern. Die absoluten Werte von Cassandra waren mehrfach führend.

MongoDB und Cassandra erreichen ihre schnellen Abfragezeiten unter anderem durch den Planungsaufwand, der in ihre Modelle investiert wurde (siehe Abschnitt 3.4.2.3 über das Datenmodel in Cassandra, welches vergleichbar mit dem in MongoDB ist).

Daher sind MongoDB und Cassandra zwar aus Performance-Sicht Spitzenreiter, erfordern jedoch intensive Forschung am Datenmodel vor Implementation einer Applikation und erfordern höhere Zeitaufwände bei Änderungsnotwendigkeit am Model als andere Datenbanken.

Zusammenfassend sprechen die Befunde der Evaluationen für:

- Einsatz von Lucene & Solr für das Finden von Bi- und TriGram unabhängig vom Rest der Applikation
- Einsatz von Cassandra oder MongoDB, falls Budget und Zeit für intensive Planung zur Verfügung stehen
  - Insbesondere Cassandra über MongoDB, falls die Kompetenz besteht, Cassandra-Indexe zu optimieren
  - Alternativ MongoDB über Cassandra, falls mehr Flexibilität vonnöten ist
- Einsatz von ArangoDB, falls eine schnell implementierte und gut wartbare Lösung gewünscht wird und sich die Inputdateien nicht oft ändern
- Einsatz von MySQL, falls eine schnell implementierte und gut wartbare Lösung gewünscht wird, aber oft neue Dateien importiert werden müssen
- Nicht berücksichtigen von BaseX und Neo4j

In Zukunft können weitere Arbeiten hierauf aufbauen und die horizontale Skalierung, die z.B. Cassandra oder ArangoDB leisten, ebenfalls berücksichtigen.

Auch die Art der getesteten Operationen kann erweitert und an andere Gebiete angepasst werden.

6 Anhang

Im Umfang dieser Arbeit enthalten sind:

- `bachelor_thesis.pdf` - dieses Dokument
- `evaluation_results/` - Ordner mit Evaluationsergebnissen
  - `pdf/` - PDFs mit visualisierten Ergebnissen, wie in dieser Arbeit verwendet
  - `raw/` - Die Rohdaten der Ergebnisse, wie in den Tabellen dieser Arbeit verwendet
- `uimadatabase/` - Der Quellcode für das Evaluationssystem, das für das Benchmarking der DBIS geschrieben wurde

Aufgrund der tiefen Ordnerstruktur des Quellcodes und der vorkommenden überlangen Dateinamen kann nicht garantiert werden, dass die abgegebenen CDs voll kompatibel mit Windows-Systemen sind. Es wird daher dazu geraten, entweder ein Linux/OSX System zu verwenden, oder den Quellcode aus der im nächsten Abschnitt genannten Quelle herunterzuladen.

Abschließender Kommentar

Der Quellcode dieser Arbeit ist ebenfalls verfügbar unter `https://github.com/yeldiRium/uimadatabase`. Dort werden in Zukunft eventuell Änderungen vorgenommen.

Der exakte Stand zum Zeitpunkt dieser Abgabe ist unter `https://github.com/yeldiRium/uimadatabase/releases/tag/Abgabe` zu finden und ist identisch zu den hier abgegebenen Dateien.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] *ShingleFilterFactory (Lucene 4.6.0 API)*, 2013.

[2] *HTTP - Hypertext Transfer Protocol Overview*. https://www.w3.org/Protocols/, 2014.

[3] *About the XML Metadata Interchange Specification Version 2.5.1*. https://www.omg.org/spec/XMI/2.5.1/, 2015.

[4] *n gram - Retrieve Ngram list with frequencies from Solr - Stack Overflow*. https://stackoverflow.com/a/30939948/5541656, 2015.

[5] *How to create_core in Dockerfile or with docker-compose? · Issue #14 · docker-solr/docker-solr*. https://github.com/docker-solr/docker-solr/issues/14, 2016.

[6] *Text2voronoi: An Image-driven Approach to Differential Diagnosis*, 08 2016.

[7] *TextImager: a Distributed UIMA-based System for NLP*, 12 2016.

[8] *BSON (Binary JSON) Serialization*. http://bsonspec.org/, 2017.

[9] *SPARQL 1.1 Overview*. https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/, 2017.

[10] *Apache Cassandra*. https://cassandra.apache.org/, 2018.

[11] *Apache Lucene - Apache Lucene Core*. https://lucene.apache.org/core/, 2018.

[12] *Apache Lucene - Welcome to Apache Lucene*. https://lucene.apache.org/, 2018.

[13] *Apache Solr -*. https://lucene.apache.org/solr/, 2018.

[14] *ArangoDB - highly available multi-model NoSQL database*. https://arangodb.com/, 2018.

[15] *ArangoDB Cluster*. https://www.arangodb.com/why-arangodb/cluster/, 2018.

[16] *arangodb/arangodb - Docker Hub*. https://hub.docker.com/r/arangodb/arangodb/, 2018.

[17] *basex/basexhttp - Docker Hub*. https://hub.docker.com/r/basex/basexhttp/, 2018.

[18] *Blazegraph Licensing Enterprise Licensing.* https://www.blazegraph.com/services/blazegraph-licensing/, 2018.

[19] *Darmstadt Knowledge Processing Repository.* https://www.ukp.tu-darmstadt.de/research/current-projects/dkpro/, 2018.

[20] *docker-neo4j/docker-entrypoint.sh at master · neo4j/docker-neo4j.* https://github.com/neo4j/docker-neo4j/blob/master/src/3.4/docker-entrypoint.sh, 2018.

[21] *Home · tinkerpop/blueprints Wiki.* https://github.com/tinkerpop/blueprints/wiki, 2018.

[22] *JSON.* https://www.json.org/json-de.html, 2018.

[23] *library/cassandra - Docker Hub.* https://hub.docker.com/_/cassandra/, 2018.

[24] *library/mongo - Docker Hub.* https://hub.docker.com/_/mongo/, 2018.

[25] *library/mysql - Docker Hub.* https://hub.docker.com/_/mysql/, 2018.

[26] *library/neo4j - Docker Hub.* https://hub.docker.com/_/neo4j/, 2018.

[27] *library/solr - Docker Hub.* https://hub.docker.com/_/solr/, 2018.

[28] *library/ubuntu - Docker Hub.* https://hub.docker.com/_/ubuntu/, 2018.

[29] *lyrasis/blazegriph - Docker Hub.* https://hub.docker.com/r/lyrasis/blazegraph/, 2018.

[30] *Main Page - BaseX Documentation.* http://docs.basex.org/wiki/Main_Page, 2018.

[31] *MySQL :: MySQL 8.0 Reference Manual :: 1 General Information.* https://dev.mysql.com/doc/refman/8.0/en/introduction.html, 2018.

[32] *MySQL :: MySQL 8.0 Reference Manual :: 1.3.2 The Main Features of MySQL.* https://dev.mysql.com/doc/refman/8.0/en/features.html, 2018.

[33] *MySQL :: MySQL 8.0 Reference Manual :: 13.2.10.2 JOIN Syntax.* https://dev.mysql.com/doc/refman/8.0/en/join.html, 2018.

[34] *RDF - Semantic Web Standards.* https://www.w3.org/RDF/, 2018.

[35] *Why ArangoDB nosql multi-model database that scales.* https://arangodb.com/why-arangodb/, 2018.

[36] Jörg Baach. neo4j performance compared to mysql. http://baach.de/Members/jhb/neo4j-performance-compared-to-mysql, 2015.

[37] Diksha Khurana, Aditya Koli, Kiran Khatter, and Sukhdev Singh. Natural language processing: State of the art, current trends and challenges. *CoRR*, abs/1708.05148, 2017.