Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Schemalose Datenbank

MongoDB als Datenbank für Document Stores

Falls erforderlich: Zur Erlangung des akademischen Grades eines
Master of Science
- Studiengang MCS -

Fachbereich Informatik
Referent: Prof. Dr. Knolle
Falls erforderlich: Korreferent: B. Mager

eingereicht von:
Franz-Dominik Dahmann, Jan Eric Müller, Thomas Tobias Zok
Matr.-Nr.

Sankt Augustin, den 14.01.2016

Inhaltsverzeichnis

1 Vorwort **1**

2 Ausgewählte Persistenzmodelle **2**
2.1 Key-Value Stores . . . 2
2.1.1 Einleitung . . . 2
2.1.2 Eigenschaften . . . 2
2.1.3 Abgrenzung zu Document-Stores . . . 4
2.1.4 Anwendungsfälle . . . 4
2.1.5 Bewertung . . . 4
2.2 Document Stores . . . 5
2.2.1 Einleitung . . . 5
2.2.2 Eigenschaften . . . 5
2.2.3 Datenmodell . . . 6
2.2.4 Anwendungsfälle . . . 8
2.2.5 Bewertung . . . 9

3 Ausgewählte Persistenzsysteme und Big Data Frameworks **10**
3.1 MongoDB . . . 10
3.1.1 Visitenkarte . . . 10
3.1.2 Systemtechnologie . . . 12
3.1.3 Datenmodell . . . 16
3.1.4 Systeminstallation . . . 17
3.1.5 Datenschema . . . 21
3.1.6 Ad-Hoc-Zugriffsmöglichkeiten . . . 21

4 Ausgewählte Anwendungsszenarien **23**
4.1 Web 2.0 - Abschlussarbeiten-DB . . . 23
4.1.1 MongoDB . . . 25

5 Fazit **31**
5.1 MongoDB . . . 31
5.1.1 Zusammenfassung . . . 31
5.1.2 Pros . . . 31
5.1.3 Cons . . . 32
5.1.4 Ausblick . . . 32

Zusammenfassung **33**

6 Anhänge **34**
6.1 Use Cases 34

Literaturverzeichnis **39**

1 Vorwort

Nachfolgend sind die Gruppenmitglieder und ihre Kürzel einzusehen:

- FDD: Franz-Dominik Dahmann
- JEM: Jan Eric Müller
- TTZ: Thomas Tobias Zok

Tabelle 1.1: Arbeitsaufteilung der Gruppenarbeit

	Schriftliche Ausarbeitung	Praktische Umsetzung	Vortrag Vorbereitung	Vortrag Durchführung
Ausgewählte Persistenzmodelle	FDD, JEM	FDD, JEM	FDD, JEM	FDD, JEM
Ausgewählte Persistenzsysteme und Big Data Frameworks	TTZ	TTZ	TTZ	TTZ
Vorstellung des Systems	TTZ	TTZ	TTZ	TTZ
Installation des Systems	TTZ	FDD, JEM, TTZ	TTZ	TTZ
Ad-Hoc-Anwendung des Systems	TTZ	TTZ	TTZ	TTZ
Implementierung der Beispiel-Datenbank	JEM	FDD, JEM, TTZ	TTZ	TTZ
Ausgewählte Anwendungsszenarien	FDD, JEM, TTZ	FDD, JEM, TTZ	FDD, JEM, TTZ	FDD, JEM, TTZ
Spezifikation des Anwendungsszenarios	FDD, JEM, TTZ	FDD, JEM, TTZ	JEM, TTZ	JEM, TTZ
Spezifikation der konkreten Anwendung	FDD, JEM	FDD, JEM	JEM, TTZ	JEM, TTZ
Implementierung der Anwendung	JEM	FDD, JEM	JEM, TTZ	JEM, TTZ

2 Ausgewählte Persistenzmodelle

2.1 Key-Value Stores

2.1.1 Einleitung

Key-Value Stores oder Key-Value Datenbanken sind eine der einfachsten Formen der NoSQL Datenbanken. Sie können beliebige Werte (Englisch: Values) unter einem definierbaren Key persistieren und über diesen selektieren. Dabei können sie mit einem herkömmlichen relationalen Datenbanksystem (RDBS) verglichen werden in welchem die Tabellen nur zwei Attribute besitzen: eine ID- und eine Value-Spalte [Sad13]. Im Unterschied zu dieser relationalen Datenbank erwarten Key-Value Stores keinen Datentyp für den Value. Hier können beliebige Werte als blob gespeichert werden. Auch können Key-Value Datenbanken weder Relationen oder Beziehungen zwischen den Einträgen herstellen. Für die Semantik, Korrektheit der Werte, Beziehungen o.ä., ihre Verwendung, welcher Datentyp vorliegt und etwaiges Fehlerhandling ist allein die Anwendung, welche die Datenbank nutzt, verantwortlich. Dieses sehr grundlegende Verhalten soll eine möglichst hohe Performance und Skalierbarkeit ermöglichen.

2.1.2 Eigenschaften

Einige Key-Value Stores gehen einen Schritt weiter und können das zu speichernde Aggregat differenzieren. So kann Redis beispielsweise Listen, Sets oder Hashes speichern und unterstützt einfache Operationen auf diesen Datenstrukturen. Somit verschwimmt in der Praxis der Unterschied zwischen Key-Value und Document Stores bei einigen Vertretern sehr.

Im Folgenden werden einige Eigenschaften von Key-Value Stores näher betrachtet und aufgezeigt wie diese sich von einem Standard RDBS unterscheiden.

Konsistenz

Da die Key-Value Datenbank Änderungen des Values nicht bestimmen kann, kann die Konsistenz somit nur über den Key und die damit verbundenen Operationen get, put oder delete gewährt werden [Sad13].

In verteilten System wird häufig die *eventually consistent* Eigenschaft angewendet.

Transaktionen

Für Transaktionen gibt es in der NoSQL Welt keine einheitliche Spezifikation. Jeder Anbieter von Key-Value Stores implementiert diese anders, Riak benutzt hier beispielsweise das Quorum-Konzept [Sad13].

Query-Eigenschaften

Da sich Key-Value Datenbanken nicht für den gespeicherten Value interessieren kann man Queries nur auf den Key ausführen. Auf Grund dieses sehr einfachen Handlings sind Key-Value Stores nicht für Anwendungen geeignet, welche über den Inhalt filtern oder suchen müssen. Hier muss die Anwendung wissen welche Daten unter welchem Key zu finden sind und welchen Key sie aufrufen möchte. Da nicht immer der Fall eintritt, dass der Key zu einem Datum bekannt ist, stellen manche Key-Value Datenbanken allerdings eine Suchfunktionalität oder die Möglichkeit der Indexierung bereit, damit nicht jeder Datensatz von der Anwendung geladen und einzeln durchsucht werden muss [Sad13].

Struktur der Daten

Wie bereits erwähnt können in Key-Value Datenbanken verschiedenste Datentypen wie blob, text, JSON, XML oder weitere gespeichert werden, da die Datenbank sich um diese Werte nicht weiter kümmert. Bestimmte Datenbanken, wie beispielsweise Riak können einem Request einen Content-Type übergeben womit diese weiß wie sie die Daten behandeln soll [Sad13].

Skalierbarkeit

Um die Performance zu erhöhen arbeiten die meisten Key-Value Stores mit „Sharding“. Hierbei entscheidet der Wert des Keys auf welchem Knoten bzw. Shard der Datenbank ein Datum gespeichert wird. So werden beispielsweise alle Keys, welche mit g beginnen auf einem anderen Shard gespeichert als solche, die mit a oder einem nummerischen Wert beginnen. Zusätzliche Shards können sehr einfach ergänzt

werden. Um hierbei die Konsistenz nicht zu gefährden, sollte ein Knoten ausfallen, können diese Shards repliziert werden. Die einzelnen Key-Value Datenbanken bieten hierbei verschiedene Möglichkeiten an. So kann bei Riak angegeben werden, wie viele Knoten bei Schreib- und Lesezugriffen mindestens beteiligt sein müssen damit die Transaktion als erfolgreich gilt.

2.1.3 Abgrenzung zu Document-Stores

Sehr ähnlich, Unterschied: Das Aggregat muss eine bestimmte Struktur haben/ von bestimmtem Typ sein Mit Document Stores sind Abfragen auf Attribute möglich. Hier nur per Key. In der Praxis verschwimmt die klare Trennung. Unfertig ...

2.1.4 Anwendungsfälle

Da die Keys in den Key-Value Stores von der Anwendung verwaltet werden, können diese verschiedenen Charakter haben. Eine Möglichkeit ist diese Keys durch einen bestimmten Algorithmus, zum Beispiel eine Hashfunktion, zu generieren. Man kann den Key auch durch Benutzerinformationen, wie Email-Adresse oder User-ID generieren. Alternativ kann dieser auch aus einem Zeitstempel oder anderen Attributen erstellt werden. Dies macht Key-Value Stores attraktiv um Session Informationen, Einkaufswagen im Online-Shopping, Benutzerprofile oder Protokolldateien abzuspeichern, da diese Daten unter einem eindeutigen Key als ein Objekt gespeichert werden [Sad13].

Anwendungsfälle in welchen Key-Value Stores weniger gut geeignet sind:

- bei Abhängigkeiten oder Beziehungen zwischen den Daten wie sie in herkömmlichen RDBS zu finden sind.
- bei *Multioperation Transactions*, in welchen bei einem Fehler die restlichen Operationen zurückgedreht werden müssten.
- bei (Such-)Anfragen, welche den Inhalt der Daten betreffen.
- bei Operationen, die sich auf mehrere Datensätze zum gleichen Zeitpunkt beziehen, da eine Operation in einem Key-Value Store auf einen Key beschränkt ist.

2.1.5 Bewertung

Key-Value Stores ziehen ihren großen Vorteil aus der Einfachheit ihrer Implementierung. Diese ermöglicht ihnen eine hohe Performance. Hierbei verzichten sie ab-

sichtlich auf Funktionen, wie Beziehungen, feste Datentypen und die Möglichkeit Abfragen auf jedes Attribut eines Datensatzes zu stellen, die durch die relationale Datenbankwelt bekannt geworden sind.

2.2 Document Stores

2.2.1 Einleitung

Document-Stores speichern strukturierte Daten in Datensätzen, die Dokumente genannt werden. Grundsätzlich vereinigen sie die Schemafreiheit von Key-Value-Stores mit der Möglichkeit zur Strukturierung der gespeicherten Daten. Somit lassen sich Document-Stores als Erweiterung von Key-Value Stores betrachten. Vorteil hierbei ist, dass es bei Document-Stores im Gegensatz zu Key-Value Stores möglich ist, komplexere auf den Inhalt (die Attribute) der Dokumente gerichtete Abfragen zu stellen. Dies ist bei Key-Value-Stores nicht möglich.

2.2.2 Eigenschaften

Bei der Betrachtung von Document-Stores bezieht sich das Wort “document“auf die hierarchische Struktur der abgespeicherten Daten. Die hier betrachteten Dokumente bestehen jedoch lediglich aus binären Daten oder einfachem Text. Ferner kann es sich um Semi-strukturierte Daten wie JSON oder XML handeln. Während ältere Document-Stores auf XML setzten, ist bei neueren Document-Stores das JSON ähnliche BSON üblich.

In einer Document-Store basierten NoSQL-Datenbank ist es möglich beliebigen Daten zu speichern, ohne das die Datenbank im Vorfeld Wissen über die Struktur der Daten hat oder weiß was die Daten bedeuten. Gegenüber relationalen Datenbanken bietet dies den Vorteil, dass das Datenbankschema bei kleinen Änderungen nicht durch größere Änderungen verändert werden muss [Har15].

In einer Document-Store Datenbank ist folgende Hierarchie typischerweise anzutreffen:

- Ein Dokument ist eine Grundeinheit für das speichern von Daten. Sie ist zu vergleichen mit einer Zeile einer Relationalen Datenbank.
- Ein Dokument besteht aus einem oder mehreren Key-Value Paaren, und kann weitere Dokumente und Arrays enthalten. die Arrays können wiederum ebenfalls Dokumente enthalten, was eine komplexere Strukturierung ermöglicht.

- eine Sammlung oder ein Data Bucket ist ein Set von Dokumenten, die einen Zweck teilen. Dies ist äquivalent zu einer relationalen Tabelle. Hier ist jedoch relevant zu erwähnen, dass die Dokumente hier nicht vom gleichen Typ sein müssen, auch wenn es üblich ist, dass solche Dokumente die gleiche Art von Information darstellen [Har15].

2.2.3 Datenmodell

Im Zentrum des Datenmodells der Document-Stores stehen die Documents, die eine hierarchische Struktur aufweisen. So kann ein Dokument auf viele weitere Dokumente verweisen. Dies ist möglich, da ein Dokument Listen enthalten kann.

Wie bereits erwähnt ist es möglich mithilfe von Document-Stores ein relationales Schema nachzubilden. Dies ist jedoch häufig nicht sinnvoll, da es größere Vorteile bringt die Dokumente in kleineren Sammlungen zusammenzufassen und in eingebetteten Dokumente die größte Detailstufe zu bieten [Har15].

Sei 2.1 die Darstellung einer relationalen Datenbank mit Filmen und Schauspielern.

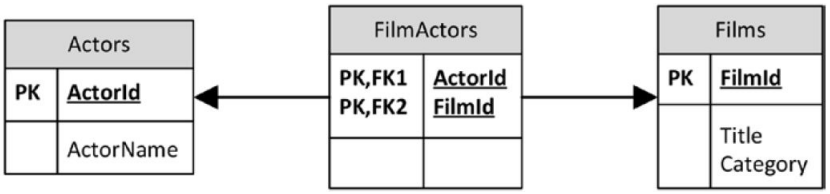

Abbildung 2.1: Beispiel eines JSON-Dokuments

In einer relationalen Datenbank würde es eine Tabelle mit Filmen und eine Tabelle mit Schauspielern geben, die zu einer gemeinsamen Tabelle gejoint werden würden um anzuzeigen welcher Schauspieler bei welchem Film mitgewirkt hat. Mithilfe von Document-Stores wäre dieses Schema so auch möglich, jedoch gibt es hierfür einen besseren Ansatz. Da Document-Stores grundsätzlich keine Join-Operationen ermöglichen und Entwickler es bevorzugen wenn die JSON Strukturen sich nah an dem objektorientierten Programmentwurf orientieren, bietet sich ein Entwurf nach 2.2 eher an [Har15].

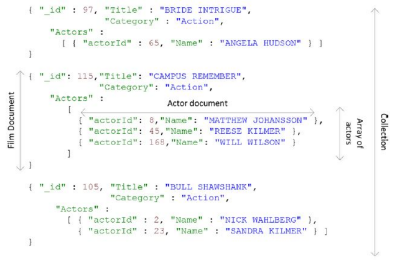

Abbildung 2.2: Beispiel eines JSON-Dokuments

Wie in der Abbildung 2.2 zu sehen, befindet sich ein Array von Schauspieler Dokumenten in dem Film Dokument. Dieser Design-Entwurf, der Document-Embedding genannt wird, hat den Vorteil, dass hierdurch auf aufwendige Join-Operationen verzichtet werden kann. Nachteil hierbei ist jedoch, dass die Schauspieler in vielen Dokumenten auftauchen und somit redundant sind. Sollte zudem eines der Schauspieler Attribute verändert werden müssen kann dies zu Inkonsistenzen führen. Ein weiteres Problem kann auftreten, wenn beispielsweise eine unbegrenzte Anzahl an Schauspielern möglich sein soll, da Dokumente in der Regel begrenzt sind. Um dieses Problem zu lösen besteht jedoch die Möglichkeit Indizes zu verwenden [Har15].

Abbildung 2.3: Beispiel eines JSON-Dokuments

In Abbildung 2.3 ist zu sehen, wie ein solches System aussehen könnte.

Grundsätzlich gilt bei modernen Document-Stores, dass Dokumente üblicherweise in JSON bzw. BSON vorliegen [Har15].

JSON

Die Java-Script-Object-Notation ist ein vom Menschen und Maschinen lesbarer Standard zur Beschreibung von Daten. JSON ist wie XML ein Standard zum Datenaustausch im Web. In 2.2 ist ein Beispiel für ein JSON-Dokument zu sehen. Vorteil von JSON ist, dass JSON-Dokumente einfach zu zerlegen und umzuwandeln sind. Dies verringert den Aufwand, der in der Anwendungsschicht betrieben werden muss.

BSON

Unter BSON sind binär kodierte JSON-Dokumente zu verstehen. BSON erweitert das Modell von JSON um weitere Datentypen und ermöglicht effizientes kodieren und dekodieren innerhalb verschiedener Sprachen.

2.2.4 Anwendungsfälle

JSON basierte Document-Stores haben vor allem in Web-basierten Anwendungen ihre Vorzüge. Hier wird häufig JSON als Datenschicht verwendet, weswegen Document-Stores hier bevorzugt werden.

XML-basierte Document-Stores finden vor allem in Content-Management-Systemen Anwendung, da sie hier ein Management Repository für XML-basierte Textdateien zur Verfügung stellen.

Grundsätzlich lässt sich jedoch feststellen:

- dass Document-Stores vor allem in Umgebungen florieren, in denen über viele Wege Zugang zu den Daten gewährleistet werden soll, die wiederum vielfältiger Natur sein können.
- dass Document-Stores sinnvoll sind, wenn eine große Menge an kleinen, sich wiederholenden Schreib- und Lesevorgängen abgearbeitet werden muss.
- dass Document-Stores sinnvoll sind, wenn die CRUD-Operationen ohne großartige Join-Operationen umgesetzt werden sollen.
- dass Document-Stores sinnvoll sind, wenn eine programmiererfreundliche Datenbank umgesetzt werden soll.

2.2.5 Bewertung

Hieraus lässt sich evaluieren, dass Document-Stores vielfältige Einsatzmöglichkeiten haben. Vor allem im Bereich der Web-Technologien haben Document-Stores wie MongoDB Stärken, die sie zu guten Alternativen zu klassischen relationalen Datenbanken machen. Durch die Freiheiten bei den zu speichernden Daten und der Einfachheit der Abfragen zeigen sich hier vor allem ihre Stärken. Ferner wird deutlich, dass, gemessen an der Geschwindigkeit in der neue Technologien zur Darstellung von Daten entstehen, Document-Stores im Web die notwendige Freiheit bieten, um angemessen vorzugehen.

3 Ausgewählte Persistenzsysteme und Big Data Frameworks

3.1 MongoDB

3.1.1 Visitenkarte

MongoDB ist ein schemaloses NoSQL Datenbanksystem, welches auf Dokumentenspeicherung setzt. Es ist in C++ programmiert[Boi12] und speichert die Dokumente in einem JSON ähnlichen Format, was BSON(Binary JSON) genannt wird. MongoDB wurde im Jahre 2010 zum ersten Mal für die breite Masse veröffentlicht und wurde zuletzt Ende November 2016 auf der Version 3.4 geupdated.[Mon16]

Geschichte

MongoDB wurde von Dwight Merriman, Eliot Horowitz und seinem Team erstellt. Doch dies nicht auf direktem Wege. 2007 plante das MongoDB Team ein Onlineservice für Web-Anwendungen. Dieser Service sollte die Möglichkeit bieten, eine Web-Anwendung zu entwickeln, zu hosten und zu skalieren. Aufgrund von einem nicht geeigneten Datenbanksystem entschloss sich das Team ein eigenes Datenbanksystem zu erstellen und zu nutzen. Diese hatte noch keinen Namen, da dieses System speziell für den Service des Teams erstellt wurde. Im Jahre 2008 wurde dann das Datenbanksystem fertiggestellt. 2009 entschloss sich das Team, dieses Datenbanksystem, was sie erstellt haben, als Open Source Produkt freizugeben. Dieses System wurde als MongoDB veröffentlicht. März 2010 kam die erste Version von MongoDB heraus, die man in einem grösseren Umfang verwenden kann. Über die Jahre wurde MongoDB von dem Team weiterentwickelt und ist zurzeit mit der Version 3.2 veröffentlicht.[Sha15]

Anwendungsfälle

MongoDB wird von verschiedenen Unternehmen zu verschiedensten Anwendungen und Aufgaben verwendet. MTV verwendet MongoDB als Haupt-repository für das MTV-Network. SourceForge verwendet MongoDB als Back-End Speicher. Bit.ly ver-

wendet MongoDB, um den Verlauf von Nutzern zu speichern. New York Times verwendet MongoDB für eine Foto-Abgabe.[Boi12]

Besonderheiten

MongoDB ist ein Datenbanksystem, was darauf ausgelegt wurde, schnell und Flexible mit einer guten Skalierung. Die Entwickler von MongoDB haben MongoDB nicht als die beste Datenbanklösung entwickelt, sondern als ein System, was für Fälle wie komplexe Datensammlungen geeignet ist. Zudem stellt das MongoDB Team eine umfangreiche Dokumentation mit vielen Beispielen und Erklärungen für jede veröffentlichte Version von MongoDB. MongoDB kann keine SQL-Befehle direkt ausführen, besitzt aber Funktionen, die die Funktionen von SQL übernehmen wie ORDER BY und DISTINCT.

Grenzen

MongoDB besitzt in mehrere Bereiche Limitationen, die die Performance und das Nutzen von MongoDB einschränken.

- 32bit Version von MongoDB kann nur eine Datenbank einrichten, die maximal 2 GB grösse sein kann. Dieses Problem kann behoben werden, indem man die 64bit Version von MongoDB verwendet. Zudem wird die 32bit Version von MongoDB nicht mehr seit der 3.0 Version unterstützt.

- BSON Dokumente können maximal 16MB grösse sein, was aber in Vergleich zu anderen Datenbanksystemen sehr grösse ist.

- Indexe in MongoDB können nicht grösser als 1024 Byte sein. Zudem kann eine Collection maximal 64 Indexe besitzen. Auch der Name des Indexes ist auf 128 Byte begrenzt. Diese Limitationen sollten bei einer gut aufgebauten Datenbank nicht erreicht werden.

- Das Sharding sollte wenn möglich so früh wie möglich eingeführt werden, da sonst die Performance von MongoDB stark beeinträchtigt wird, da die Server die Daten auf den unterschiedlichen Shards verteilen muss. Zudem sollte auch geachtet werden, das die Datenbank nicht über 256 GB grösse ist, da MongoDB bei einer Grösse von 256 GB das Sharding nicht mehr erlaubt.

- Die Daten die bei MongoDB versendet werden sind nicht verschlüsselt. Bei Lese und Schreibvorgänge bei der MongoDB muss man auf Grösse- und Kleinschreibung achten, da MongoDB Case-Sensitive ist.

- MongoDB besitzt keine JOIN-Operation. Man muss diese durch verschachtelte Dokumente oder auf einen Verweis auf das jeweilige Dokument machen.

3.1.2 Systemtechnologie

Referenzierung

Die Referenzierung von MongoDB unterscheidet sich von der Referenzierung von den relationalen Datenbanken. Eine Möglichkeit der Referenzierung in MongoDB wäre die Verschachtlung, also ein Dokument in einem Dokument schreiben. Damit kann man die Referenz auf ein anderes Objekt direkt erkennen, was auch für das System einfacher ist, an diese Daten zu gelangen. Das Problem hierbei kann aber eine zu starke Verschachtelungsstruktur sein, was auf Kosten der Übersichtlichkeit geht, zusätzlich wird für ein Dokument mehr Platz gebraucht, da in einem Dokument ein weiteres existiert. Die zweite Möglichkeit für eine Referenzierung, wäre der Verweis auf das jeweilige Dokument. Das verbessert die Übersicht und man kann erkennen, voraus die Daten herkommen. Ein Problem was MongoDB mit sich bringt, ist, dass es keine Direktreferenzierung gibt. In der Relationalen Datenbank kann man mit dem JOIN Befehl die Referenz zu einem anderen Datensatz in Bezug auf einen anderen Datensatz direkt ansprechen und anzeigen lassen. Dies kann MongoDB nicht. Um dies zu ermöglichen, muss man das in einer eigenen Applikation programmieren.[Mon16]

Replikation

Eine Replikation bei MongoDB ist die Kopie von Daten auf weiteren Servern. Dies nennt man replica set. Ein replica set umfasst Server mit derselben Funktionalität und Dient zur Stabilisierung der Aufgabe des jeweiligen replica sets. Die MongoDB Website empfiehlt das Verwenden von drei Servern pro replica set, um somit die Ausfallrate gering zu halten. Die Daten in einem replica set sind auf allen Servern identisch. MongoDB verwendet das Master-Slave System für die Replikation von Daten innerhalb des replica sets. In dem replica set gibt es genau einen Primary und beliebig viele Secondary Server und ggf. einen Arbiter. Datenänderungen wie z.B. Schreiboperationen werden auf dem Primary Server in dem replica set durchgeführt. Nachdem die Operation erfolgreich beendet wurde, übernehmen die Secondary Server die Änderung von dem Primary Server. Sollte der Primary Server ausfallen, so wählen die Secondary Server einen neuen Primary Server aus. Man ist in der Lage, in dem replica set einen Arbiter hinzuzufügen. Ein Arbiter besitzt nicht die Daten, die in dem replica set verteilt werden. Der Arbiter dient lediglich dazu, nur bei einer Abstimmung eine Stimme abzugeben, damit eventuelle Probleme bei einer Abstimmung vermieden werden können. Ein Arbiter selber kann nicht zu einem Primary oder Secondary Server ernannt werden.[Mon16]

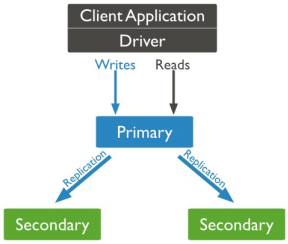

Abbildung 3.1: Replikationsaufbau von MongoDB von [Mon16]

Indexierung

Indizes in MongoDB dienen zur verbesserten Ausführung von Abfragen. Durch Indizies überprüft MongoDB nicht alle Dokumenten innerhalb einer Collection, sondern nur die Dokumente, die mit dem abgefragten Index versehen ist. Ohne Indizes würde MongoDB immer die Collection komplett nach dem Abfragekriterium überprüfen. Ein Index, was MongoDB bei jeden Dokument automatisch erstellt, ist der id Index. Dieser dient dazu, jedes Dokument mit einem eindeutigen Wert zu versehen, das nur auf dieses Dokument zugewiesen werden kann, also wie ein Primärschlüssel in relationalen Datenbanken. Um Indizes zu erstellen, wird folgender Befehl verwendet:

Listing 3.1: Befehl, um eine Collection zu erstellen

```
db.collection.createIndex()
```

Innerhalb der Klammern kann bestimmt werden, welcher Wert als Index gewählt werden soll und welche Art von Index verwendet wird. Indizes sind an einer Collection gebunden und können auch nur in dieser verwendet werden. Wenn man ein Index erstellt, sollte darauf geachtet werden, dass der gewählte Wert nicht bereits ein Index besitzt. Ebenso sollte darauf geachtet werden, dass möglichst viele Dokumente den Wert besitzen, der indexiert wird.[Mon16] MongoDB bietet viele verschiedene Arten von Indizes an, die verwendet werden können.

- Single Field Index die sich nur auf ein Wert beziehen
- Compound Index, womit man ein Index mit mehreren Werten bestimmen kann
- Multikey Index mit einem Wert, der aus mehreren Werten besteht
- Geospatial Index für Geometrische Werte
- Text Indexes um Werte mit Texten zu indexieren
- Hashed Indexes für Hash basierte Verteilung

Zudem kann man indizes mit Eigenschaften versehen:

- Unique Indexes sind Indizes, die nur einmalige Werte akzeptiert

- Partial Indexes sind Indizes nach bestimmten Filteroptionen
- Sparse Indexes indexiert nur die Dokumente, die den angegebenen Wert besitzen
- TTL Indexes sind Indizes, die die indexierte Werte nach einer angegebenen Zeit automatisch löscht.

Sharding

MongoDB ist in der Lage, Daten über mehreren Servern zu verteilen. Diese Methode wird Sharding genannt. Sharding ist in der Lage, in einem Cluster von Servern Daten von einer Datenbank aufzuteilen, sodass dadurch das Abrufen der Daten verbessert und beschleunigt wird. Ein sharded cluster besteht aus drei Komponenten: Den Config Server, den Query Server und Shards. Um ein sharded cluster einzurichten, braucht man mindestens einen Config Server, einen Query Server und 2 Shards. Dies sieht dann wie folgt aus:

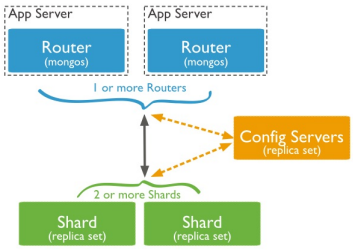

Abbildung 3.2: Aufbau vom Cluster Sharding in MongoDB von [Mon16]

Config Server: Der Config Server in einem sharded Cluster beinhaltet die Metadaten für dieses. Diese Metadaten beinhalten die Informationen zu der Organisation und die Zustände der jeweiligen Komponenten in dem sharded Cluster. Ein Config Server ist in einem replica set. Ein replica set verbessert die Konsistenz über die verschiedenen Config Server. Zusätzlich kann man danke dem replica set mehr als 3 Config Server nutzen. Query Server (Router): Mit dem Query Server ist man in der Lage, Daten auf den verschiedenen Shards zu schreiben und zu Lesen. Dies vollbringt er mit den Metadaten, die der Query Server von dem Config Server bekommt. Der Query Server ist die einzige Schnittstelle in dem sharded Cluster, womit man auf die Daten durch Anwendungen zugreifen kann.

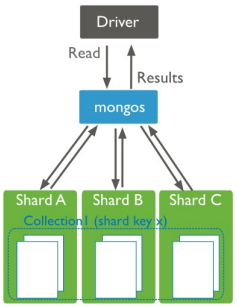

Abbildung 3.3: Datenverteilung von dem Query Server von [Mon16]

Shard: Bei einer Shard handelt es sich um einen Teil der Gesamtdatenbank. Die Daten, die in die Datenbank eingetragen werden, werden nach den Metadaten des Configservers analysiert und dann durch den Query Server auf das jeweilige Shard gespeichert. Zum Beispiel: Eine 3 Shard Datenbank beinhaltet Daten von Personen in einem Unternehmen. Auf Shard 1 werden alle Mitarbeiter gespeichert, die im Nachnamen mit dem Buchstaben A bis H anfangen. Shard 2 hat den Nachnamen von I bis Q und Shard 3 R bis Z. Dadurch, dass die Daten aufgeteilt und auf unterschiedlichen Shards gespeichert werden, ist man in der Lage schneller an die gewünschten Informationen zu kommen. Shards können sich in einem replica set befinden. Dieses dient für Redundanz und hohe Verfügbarkeit der Daten.[Mon16]

3.1.3 Datenmodell

Allgemeiner Aufbau von MongoDB MongoDB ist ein dokumentenbasiertes schemaloses Datenbanksystem. Daten werden dementsprechend in Dokumenten gespeichert. Eine MongoDB-Datenbank besteht aus einer oder mehreren Collections. Und Jede Collection besitzt mindestens ein oder mehrere Dokumente.

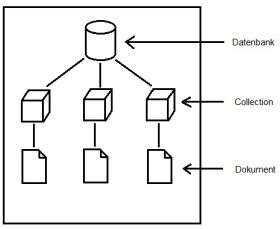

Abbildung 3.4: Datenbankstruktur von MongoDB nachkonstruiert nach [Mon16]

Collections in MongoDB besitzen keine Vorgaben, wie ein Dokument aufgebaut sein sollte. So ist man in der Lage, jede Art von Dokument in Collections einzufügen. Dies sollte aber vermieden werden, da dadurch das Abfragen von den Dokumenten verkompliziert wird. Dokumente in MongoDB können bis zu 16 MB grös sein. So ist es möglich, eine grösse Menge an Daten in ein einziges Dokument zu speichern, was aber auf Kosten der Übersicht und der Performance geht.

3.1.4 Systeminstallation

Installation von MongoDB

Da uns Linux Ubuntu 16.04 Server zur Verfügung stehen für die Installation von MongoDB, müssen wir dementsprechend MongoDB für Linux Ubuntu 16.04 installieren. MongoDB muss auf allen Nodes in dem Cluster installiert werden. Es wurde MongoDB 3.2 installiert, da diese zum Zeitpunkt der Installation die aktuellste Version von MongoDB war. Zurzeit gibt es MongoDB in der Version 3.4. Dementsprechend richtet sich die Installation und Konfiguration von MongoDB auf die Version 3.2
Um MongoDB auf Ubuntu zu installieren, müssen folgede Schritte durchgeführt werden.

1. Um bei Ubuntu mit apt oder dpkg MongoDB zu installieren, wird ein Public key benötigt. Diesen bekommt man durch folgenden Befehl:

Listing 3.2: Befehl, um den Public Key zu erhalten

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv EA312927
```

2. Danach muss eine Listendatei für MongoDB erstellt werden. Diese muss die Version, die installiert wird im Namen beinhalten. Der hier verwendete Befehl lautet:

Listing 3.3: Erstellung der Liste für die Installation

```
sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list
```

3. Bevor man MongoDB installiert, sollte der Ubuntu Server und deren Applikationen aktualisiert werden mit dem Befehl:

Listing 3.4: Aktuallisierung des Servers

```
sudo apt-get update
```

4. Nachdem nun alle Vorbereitungen getroffen worden sind, kann man nun MongoDB installieren. Der Befehl hierfür lautet wie folgt:

Listing 3.5: Installation von MongoDB

```
sudo apt-get install -y mongodb-org
```

Nachdem MongoDB installiert wurde, was einige Minuten in Anspruche nehmen kann, kann man nun MongoDB starten. Und das mit dem Befehl:

Listing 3.6: Starten von MongoDB

```
sudo service mongod start
```

Jetzt da der Mongod Service gestartet wurde, können wir nun unsere Beispieldatenbank erstellen. MongoDB erstellt automatisch eine Datenbank, sollte diese befüllt

werden mit Dokumenten. Um die Beispieldatenbank zu erstellen, müssen wir diese erst mit dem “Use“ Befehl von MongoDB auswählen. Da wir unsere Datenbank abDB nennen, so lautet der “Use“ Befehl:

Listing 3.7: Auswahl der Datenbank

```
use abDB
```

Nun können wir mit “db.createCollection()“ eine Collection für unsere Beispieldatenbank erstellen. Der ganze Befehl denn wir verwendet haben lautet:

Listing 3.8: Erstellung einer Collection

```
db.createCollection(abDBsimpleCSV,{capped:true,size:1024,max:3000})
```

Jetzt wird die Datenbank in MongoDB auch abgespeichert. Wie die Daten von der Datenbank auf unsere Beispieldatenbank übertragen werden, wird in Kapitel 4.1.1 näher erläutert.
Nun da wir MongoDB auf unserem Cluster auf jedem Node gestartet und eine Datenbank angelegt wurde, können wir nun die Konfiguration von MongoDB für das Sharding vornehmen.[Mon16]

Konfiguration von dem Sharding

Wie schon im Kapitel 3.1.2 erläutert, braucht man für das Sharding einen Query-Server, mind. einen Replica Set an Config-Servern und mind. zwei Shards. Um das Sharding für MongoDB zu ermöglichen, muss man zuerst den Replica Set von dem Config Server vornehmen.

Konfiguration Config-Server Replica Set

Da ein Replica Set aus mind. 3 Servern bestehen muss, verwenden wir auch 3 Server um die Stabilität der Server zu gewährleisten. Wir installieren die Config Server auf Node 1, 2 und 3. Die Vorgehensweise sieht wie folgt aus:

1. Verzeichnisse für Dateien und Logs vom Config Server erstellen

Listing 3.9: Erstellung der Verzeichnisse

```
sudo mkdir -p /db/config/log
sudo mkdir -p /db/config/data
```

2. Nun muss der Config Server gestartet werden:

Listing 3.10: Starten von einem Config Server

```
sudo mongod --port 27022 --dbpath /db/config/data --configsvr --replSet abDBcs --
    fork --logpath /db/config/log/mongodb.log
```

Dieser Befehl muss bei allen Config Servern durchgeführt werden, und alle müssen demselben Replica Set zugewiesen werden.

3. Nachdem alle Config Server gestartet wurden, muss nun das Replica Set für die Config Server konfiguriert werden. Dazu müssen wir zuerst in die Mongo Shell gehen. Mit der Mongo Shell ist man in der Lage, die Mongo Instanz zu bedienen. Die Mongo Shell startet man mit:

Listing 3.11: Mongo Shell für den Config Server starten

```
mongo localhost:27022
```

In der Mongo Shell selber kann man dann das Replica Set konfigurieren. Dieser Befehl muss nur auf einen der Config Servern durchgeführt werden:

Listing 3.12: Konfiguration des Replica Sets abDBcs

```
rs.initiate(
{
    _id: "abDBcs",
    configsvr: true,
    members: [
      { _id : 0, host : "10.20.110.43:27022" },
      { _id : 1, host : "10.20.110.41:27022" },
      { _id : 2, host : "10.20.110.39:27022" }
    ]
  }
)
```

Nun sollte in dem Replica Set ein Primary und die Secondarys ausgewählt werden. Damit sollte nun der Config Server Replica Set für das Sharding bereit sein.
Nun werden die Shards konfiguriert. Die Shards werden fast genauso wie die Config Server konfiguriert. Shard 1 wird auf Node 1, 3 und 4 installiert und sieht wie folgt aus:

1. Verzeichnisse für Dateien und Logs vom Shard 1 erstellen:

Listing 3.13: Verzeichnisse für Shard 1 erstellen

```
sudo mkdir -p /db/shard1/log
sudo mkdir -p /db/shard1/data
```

2. Nun müssen die Server in dem Shard gestartet werden:

Listing 3.14: Einen Server für den Shard starten

```
sudo mongod --port 27023 --dbpath /db/shard1/data --shardsvr --replSet abDBrs1 --
    fork --logpath /db/shard1/log/mongodb.log
```

3. Wie auch bei dem Config Server muss nun mit der Mongo Shell auf einen der Servern das Replica Set eingestellt werden:

Listing 3.15: Konfiguration von dem Replica Set abDBrs1

```
rs.initiate(
  {
  _id: "abDBrs1",
    members: [
      { _id : 0, host : "10.20.110.43:27023" },
```

```
      { _id : 1, host : "10.20.110.39:27023" },
      { _id : 2, host : "10.20.110.48:27023" }
    ]
  }
)
```

Shard 2 wird genauso installiert, nur das Shard 2 auf Node 2,3 und 4 installiert wird und ein eigenes Replica Set besitzt.
Nun muss der Query Server eingerichtet werden. Um den Query Server einrichten zu können, müssen die Shards bereits eingerichtet sein. Die Schritte für die Einrichtung von dem Query Server sehen wie folgt aus:

1. Verzeichnis für die Log Dateien erstellen:

Listing 3.16: Verzeichnis für den Query Server erstellen

```
sudo mkdir -p /db/query/log
```

2. Nun muss der Query Server gestartet werden. Hierbei ist es wichtig anzugeben, wo sich der Config Server befindet, da der Query Server die Meta Daten von dem Config Server benötigt, um das Sharding auszuführen:

Listing 3.17: Query Server starten

```
sudo mongos --configdb abDBcs
    /10.20.110.43:27022,10.20.110.41:27022,10.20.110.39:27022 --port 27021 --fork
    --logpath /db/query/log/mongodb.log
```

3. Nachdem der Query Server gestartet wurde, müssen nun die Shards mit der Mongo Shell dem Query Server zugeteilt werden, hierbei muss jeder einzelne Server von einem Shard eingetragen werden:

Listing 3.18: Befehl, um die Shards dem Query Server zuzuweisen

```
sh.addShard("abDBrs1/10.20.110.43:27023")
sh.addShard("abDBrs1/10.20.110.39:27023")
sh.addShard("abDBrs1/10.20.110.48:27023")

sh.addShard("abDBrs2/10.20.110.41:27024")
sh.addShard("abDBrs2/10.20.110.39:27024")
sh.addShard("abDBrs2/10.20.110.48:27024")
```

4. Zum Schluss muss man dem Query Server sagen, welche Datenbank auf den Shards geteilt werden soll:

Listing 3.19: Sharding für die Datenbank aktivieren

```
sh.enableSharding("abDB")
```

Nun wurde MongoDB auf Ubuntu installiert und das Sharding für die Datenbank “abDB“ eingerichtet.[Mon16]

3.1.5 Datenschema

Uns wurde eine CSV Datei bereitgestellt, die von dem Schema der relationellen Abschlussarbeitendatenbank entnommen wurde. Da die Übername von dem relationellen Schema in MongoDB wenig Sinn ergibt, haben wir das Schema angepasst, damit es besser und effektiver von MongoDB verwaltet werden kann. Zuerst stellt sich die Frage: Wie viele Collections verwenden wir, um die Daten der Abschlussarbeitendatenbank in MongoDB darstellen zu können. Wir haben uns dazu entschlossen, dass wir die Abschlussarbeitendatenbank in nur einer einzigen Collection abspeichern werden. Die Gründe für diese Entscheidung lauten wie folgt:
MongoDB verfügt über keine direkte Referenzierung auf andere Dokumente und Collections. Man kann diese zwar nennen in einem Dokument, man muss dann dieses Dokument mit einer weiteren Abfrage heraussuchen. Zudem sind Schemalose Datenbanksysteme darauf aufgebaut, grössere Datenmengen aufzurufen.[Mon16]
Es ist einfacher Indizes auf einer grossen Collection durchzuführen. Indizes können die Abfragezeit um das 10-fache verkürzen.
Die Datenmenge die wir erhalten haben ist gering, sodass eine Aufteilung auf mehrere Collections nur mehr Aufwand als Hilfe darstellen würde.

3.1.6 Ad-Hoc-Zugriffsmöglichkeiten

Um mit MongoDB Daten einzufügen oder auszulesen, sind verschiedene CRUD-Operationen notwendig, wie bei einer relationellen Datenbank. Diese Operationen ähneln wie JavaScript Anweisungen und sind daher leicht anzuwenden. Die wichtigsten Befehle zur Verwendung von MongoDB sind in Tabelle 2.1 zu entnehmen.[Mon16]

Beim Arbeiten von MongoDB sollte stets geachtet werden, dass man auf der richtigen Datenbank arbeitet und dass die Dokumente in die richtige Collections eingetragen werden. Zudem sollte man auf Gross- und Kleinschreibung achten, da MongoDB Case-Sensitive ist.

Befehl	Funktion
Use <Datenbankname>	Setzt die Datenbank zu den benutzten Namen
db.createCollection(„Name“,capped : true, size : <byte>, max : <anzahl>)	Legt eine Collection mit dem angegebenen Namen
db.<name>.insert(Name: „Thomas“, Alter: 25, . . .)	Legt ein Dokument in der Collection <name> ab
db.<name>.save(Name: „Thomas“, Alter: 22)	Einfache Veränderung eines Dokumentes mit dem Namen “Thomas”
db.<name>.update(Name: „Thomas“, Name: „Franz“ , Alter: 24)	Veränderung eines Dokumentes mit dem Namen „Thomas“
db.<name>.find()	Zeigt alle Dokumente in der Collection <name>
db.<name>.count(Name: „Franz“)	Zählt alle Dokumente, die den Namen “Franz” beinhalten
db.<name>.remove(Name: “Franz”)	Löscht alle Dokumente mit dem Namen “Franz”
db.<name>.drop()	Die Collection löschen

Tabelle 3.1: Befehlsliste für MongoDB

4 Ausgewählte Anwendungsszenarien

4.1 Web 2.0 - Abschlussarbeiten-DB

Im Rahmen dieses Projektes ist ein relationales Schema und eine Menge an Datensätzen gegeben. Ziel ist es, dieses in eine NoSQL Lösung zu übertragen und die Datensätze in der Datenbank zu persistieren.

Das semantische Schema In der Abbildung 4.1 ist das Schema der Datenbank zu sehen, die in den gängigen NoSQL Techniken umgesetzt werden sollen. Eine Abschlussarbeit wird von einem “Student“ oder einem “Trainee“ geschrieben. Sowohl der “Student“, als auch der “Trainee“, als auch der “Examiner“, als auch der “Supervisor“ sind Unterklassen der Klasse “Human“. In Auftrag gegeben wird die Arbeit von einer “Organisation“, die eine “University“ oder eine “Company“ sein kann. Die Arbeit wird eventuell von einer Menge von “First_Examiner“ und einer Menge von “Second_Examiner“ begutachtet, wenn es zu einem “Degree“ führt. Das mit dem “Degree_Topic“ verbundene “Degree“ hat einen Titel. Das “Topic“ kann in einer “Topic_Category“ liegen, in der gegebenenfalls auch der “examiner“ oder die “Company“ liegen. Dies bedeutet, die zentrale Klasse ist das “Topic“, dass entweder ein “Degree_Topic“ oder ein “Trainee_Topic“ ist. Hierbei ist das “Degree_Topic“ wiederum entweder die “Bachelor_Thesis“ oder die “Master_Thesis“.

Das Schema deutet also an, dass ein Mensch, der eine Abschlussarbeit schreibt, entweder ein Auszubildender oder ein Student ist. Diese Abschlussarbeit schreibende Person schreibt die Arbeit entweder im Rahmen einer Ausbildung oder im Rahmen eines Studiums, der mit einem Abschluss (“Degree“) belegt ist. Fasst man dies zusammen hätte man eine Tabelle in der Informationen zu Abschlussarbeiten gespeichert sind. Diese enthalten neben dem “Topic“ der Arbeit auch weitere Informationen zu den Umständen. Insgesamt sind 27 Klassen gegeben, auf denen 36 Attribute verteilt sind.

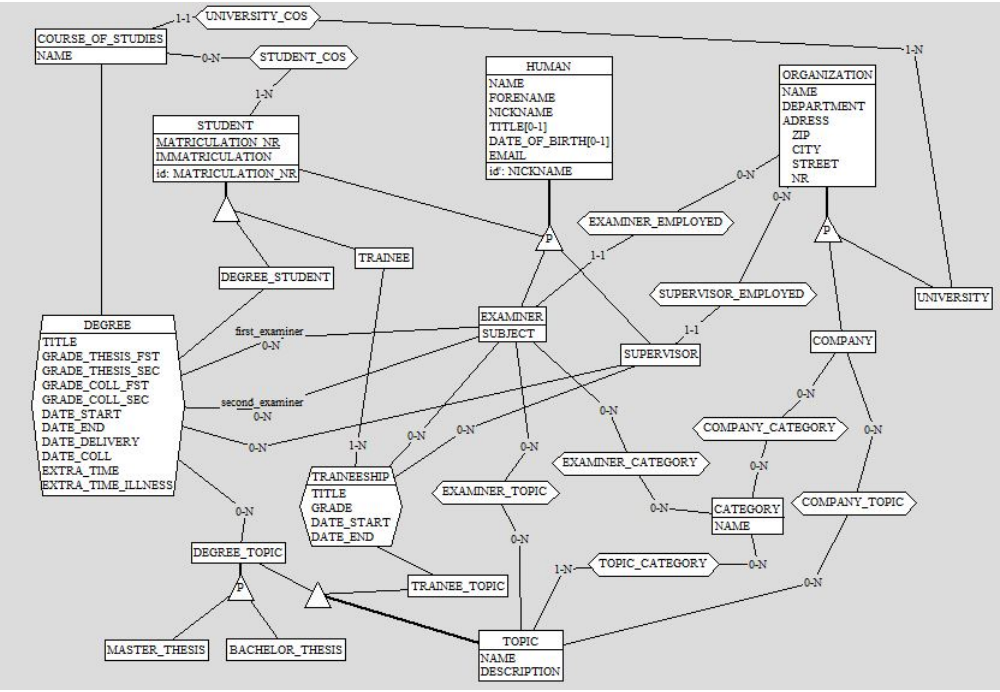

Abbildung 4.1: AbschlussarbeitenDB Schema

Die Datenbestände Die tatsächlichen Daten unterscheiden sich stark von den in dem Schema 4.1 dargestellten Sachverhalten. Wie in Abbildung 4.2 zu sehen gibt es insgesamt 21 Spalten. Insgesamt umfasst die Datenbasis 1637 Datensätze.

Studiengang	POVersion	Semester	Matrikelnr	Nachname	Vorname	Gruppenarbeit
DPL INF	2000	SS 07	121609	Rmyntf	Ugntij	

FristBegin	FristEnd	Verlaengeru	NeuesFristEnd	Abgabe	Versuch	Verfristet
16.04.2007	16.08.2007	0		16.08.2007	1	Nein

TerminKolloquium	Bafoeg	NoteThesis	NoteKolloquium	Gesamtnote	Bemerkung	Zeugnis
26.09.2007	Ja	1,3	1	1,18	Zeugnis am 08.10.07 an FB -Hei	1

ZeugnisUser	ZusatzFaecher	Studiendaue	1. PrÃ¼fer	1. PrÃ¼fer Mailadresse	2. PrÃ¼fer	2. PrÃ¼fer Mailadresse
Heinrich Michaela						

3.PrÃ¼fer	3. PrÃ¼fer Mailadresse	Firma	Ansprechpartner	Thema der Arbeit	Exma	ExmaUser
					1	Johanna Ruwwe

Abbildung 4.2: Beispiel-Datensatz

Im Vergleich mit dem Schema fällt auf, dass es scheinbar einige Probleme gibt. Die erhobenen Daten weisen folgende Probleme auf:

- Die 21 Spalten repräsentieren 21 Attribute, die eine deutliche Differenz zu den 36 Attributen des Schemas darstellen. Dies bedeutet, in den Datenbeständen sind weniger Informationen gegeben, als eigentlich im Schema festgelegt.
- In den Daten kommen Attribute vor, die im Schema nicht erwähnt wurden. Dies umfasst Attribute wie z.B. “Gruppenarbeit“ oder “Bafoeg“.
- Einige Attribute verfügen über andere Bezeichnungen, als in dem Schema genannt werden. So hat ein Student laut Schema eine “Matriculation_NR“. In den Daten ist dieses Attribut jedoch unter der Bezeichnung “Matrikelnr“ aufgeführt.
- In den Spalten der Daten sind einige Attribute mit Sonderzeichen belegt, was bei der Übertragung der Daten in eine NoSQL Datenbank zu Problemen führen kann.
- Einige Spalten enthalten keine Werte, was bei der Übertragung der Daten in die Datenbank gegebenenfalls berücksichtigt werden muss. Besonders kritisch ist dies in den Fällen zu sehen, bei denen das “Thema der Arbeit“ keinen Wert hat.
- Ein Problem der Daten ist, dass ein Professor teilweise unter verschiedenen Bezeichnungen in der Tabelle aufgeführt ist.

Bei der Übertragung der Daten in eine Datenbank, müssen Lösungen für die verschiedenen Probleme gefunden werden. Hier muss vor allem entschieden werden, welches Objekt bei der Migration in die NoSQL Datenbank wichtiger ist. Wird dem Schema größere Beachtung geschenkt, müssen größere Anpassungen an den Daten vorgenommen werden. Sollten die Daten größere Priorität haben, muss die Rolle des Schemas und dessen Gültigkeit überdacht werden.

4.1.1 MongoDB

Für die Abbildung des Anwendungsszenarios wurde sich dafür entschieden die Datensätze, bis auf kleine Korrekturen, unverändert zu übernehmen. Dafür wurde jeder Datensatz, jede Zeile der CSV-Datei, als ein Dokument importiert. Um schnelle Abfragen auf die Daten zu ermöglichen wurden häufig genutzte Attribute indiziert.

Import der Daten

Die Beschreibung des Datenimports ist mit Hilfe des ETL-Prozesses strukturiert:

Extraktion - Da die Daten in Form einer CSV-Datei bereit gestellt wurden, entfällt dieser Schritt. Allerdings müssen zwei kleine Anpassungen an der CSV-Datei vorgenommen werden:

- *Umlaute korrigieren* - Die Umlaute in der CSV-Datei wurden, vermutlich durch den Export aus dem Quellsystem, fehlerhaft dargestellt. Diese wurde so ersetzt, dass sie wieder menschenlesbar sind.

- *Kopfzeile anpassen* - Die Werte in der Kopfzeile listen die Attributnamen auf. Diese werden später die Schlüssel der Werte des Dokuments in der Datenbank sein. Schlüsselnamen dürfen weder Leerzeichen noch Umlaute enthalten. Somit wurden die Leerzeichen entfernt und durch Unterstriche „_“ ersetzt sowie alle Umlaute mit einfachen Vokalen dargestellt. So wurde beispielsweise aus „ä“ ae oder aus „ß“ ss.

Transformation - Da MongoDB Imports primär über JSON erwartet, musste die CSV-Datei in das JSON-Format transformiert werden. Die Transformation der Daten wurde mit Hilfe eines Online Tools durchgeführt. Diese ist unter der URL `http://www.csvjson.com/csv2json` [Dra16] zu finden.

Laden - Der Import in die MongoDB erfolgt über die MongoDB-Konsole. Um die Arbeit hier etwas zu vereinfachen wurde das GUI-Tool „RoboMongo“ verwendet. Die Syntax des Import-Befehls ist Listing 4.1.1 zu entnehmen. Dabei können mehrere Dokumente in einem Array per Komma getrennt gleichzeitig importiert werden.

Listing 4.1: MongoDB Importfunktion

```
db.getCollection('Collection_Name').insert([{Document01},{Document02},
        {"Attribut01":"Wert01", "Attribut02":"Wert02"}])
```

Allerdings sind die 1639 Datensätze mit 32,7 MB Gesamtgröße zu groß für einen Import. Das Maximum hier liegt bei circa 16 MB. Also wurden die Datensätze in drei Teile aufgeteilt und nacheinander importiert.

Anwendungsszenario

Die erstellte Website soll von Professoren und Studenten dafür genutzt werden, um nach Informationen über alle bisherigen Abschlussarbeiten suchen zu können. Dies kann beispielsweise für Professoren statistische oder Auswertungszwecke verfolgen oder für Studenten, welche auf der Suche nach einem Abschlussarbeitsthema sind, als Informationsquelle dienen.

Es wurden acht Use Cases aufgestellt. Darunter sind vier, die für Studenten, zwei die für Professoren und zwei, welche für beide Stakeholder relevant sind. Des Weiteren ist Use Case acht ein allgemeiner Use Case, welcher auf alle vorgestellten Persistenzsysteme gleichermaßen angewandt wurde um als Performance-Vergleich zu dienen.

Die Use Cases bestehen aus einer kurzen textuellen Beschreibung, einer Definition von Output und wahlweise Input, dem Query für die MongoDB-Konsole sowie ihre Umsetzung im Quellcode. Die vollständige Auflistung der Use Cases ist Anhang 6.1 zu entnehmen. Der Aufbau der Queries wird in 4.1.1 näher beschrieben.

Oberflächengestaltung

Die Oberfläche wurde mit Hilfe des Vaadin-Frameworks entworfen. Die Gestaltung der Ein- und Ausgabemaske wurde besonders einfach gehalten. Die Eingabemaske, insofern Vorhanden, besteht lediglich aus einem Label mit einer kurzen Beschreibung der erwarteten Eingabe, einem Eingabefeld für letztere und einem Button um die Abfrage absenden zu können. Die Ausgabemaske beinhaltet nach Betätigung des Buttons die Ergebniswerte und ein Label mit der Ausführszeit des Query. Eine schematische Darstellung ist in Abbildung 4.3 zu sehen.

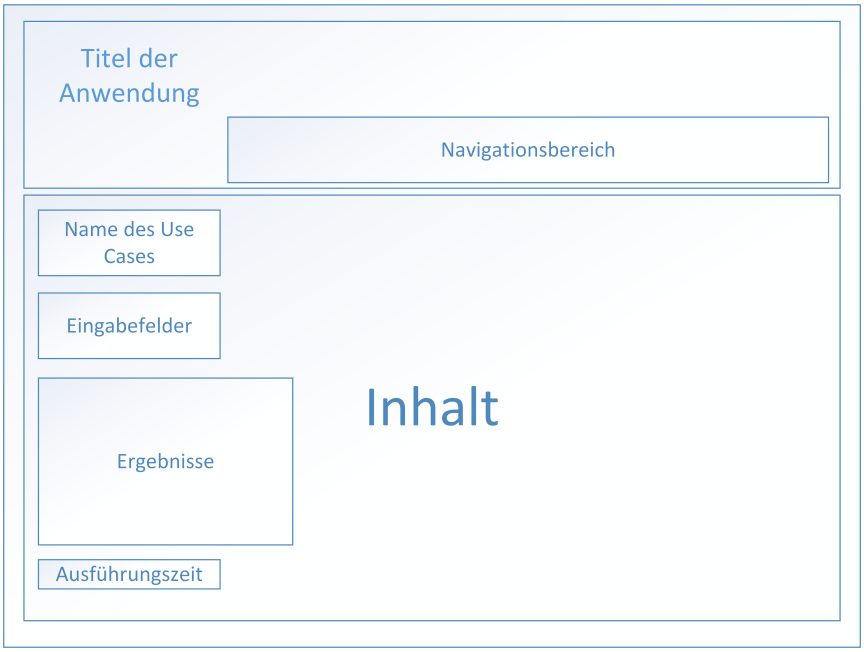

Abbildung 4.3: Schematische Darstellung der Weboberfläche

Dabei wurde darauf geachtet, dass sich der Navigationsbereich dynamisch generiert, sodass dieser nicht jedes mal neu erstellt werden muss wenn ein neuer Use Case hinzu kommt. Der Inhaltsbereich, welcher Ein- und Ausgabemaske darstellt wird abhängig von der Eingabe im Navigationsbereich geladen.

Implementierungskonzept

Bei der Implementierung wurde sich für das Vaadin-Framework entschieden, da es die schnelle Erstellung einer Webseite mittels Java ermöglicht. Dabei müssen

keine HTML- oder JavaScript-Dateien bearbeitet werden. Die Elemente und das Layout werden mit Hilfe von Vaadin-Objekten direkt im Quellcode erzeugt. Die Kommunikation zwischen Webseite und Anwendungsserver wird vollständig durch das Vaadin-Framework gekapselt. Abbildung 4.4 modelliert den Datenfluss der gesamten Anwendung. Hier ist der Datenfluss zwischen Ein- und Ausgabemaske der Anwendung im Browser und der Datenbank besonders relevant. Allerdings kapseln sowohl das Vaadin-Framework als auch die Treiber der MongoDB den größten Teil hiervon. Es kann nur Einfluss auf die Datenverarbeitung im Quellcode auf dem Anwendungsserver genommen werden. Dieser bekommt vom GUI verkörpert durch die Vaadin-Objekte die Eingabewerte in Form von einfachen Java-Datentypen wie String oder Integer (In der Abbildung als „Object“ dargestellt). Die MongoDB erwartet als Eingabe ein Objekt vom Typ DBObject. Aus mehreren ineinander verschachtelten DBObjects wird der Query zusammengestellt. Der Rückgabewert der MongoDB besteht entweder aus einem DBObject, oder einem AggregationOutput, welcher eine Liste von DBObjects verkörpert (In der Abbildung als „Object*“ dargestellt).

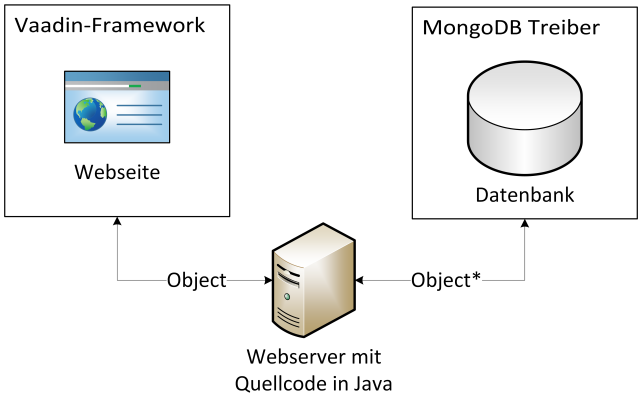

Abbildung 4.4: Exemplarischer Datenfluss zwischen Anwendung und Datenbank

Schnittstelle der Anwendung zur Datenbank

die Implementierung programmiertechnisch unterschiedliche Anfragen vorstellen. Eine Reihe ähnlich gearteter Selects ist da sicher nicht so hilfreich.

Die vorgestellte Java-Anwendung nutzt den von MongoDB bereitgestellten Java Treiber um eine Kommunikation mit der Datenbank herzustellen. Hierbei wird über den Treiber eine Verbindung zum Server hergestellt, auf dem der anzusteuernde Query-Server installiert ist. Auf diesem werden über einfache Funktionsaufrufe die Datenbank und die Collection selektiert.

Ein Query wird, wie bereits erwähnt, durch verschachtelte DBObjects dargestellt. Ein DBObject verlangt im Konstruktor einen Key und einen Value. Als Value kann wiederum ein DBObject eingesetzt werden. Dies wird in Use Case fünf, vergleiche Anhang 6.1, besonders deutlich, wo der *nequery* in den *query* eingesetzt wird. Use Cases sechs und acht stellen dagegen wesentlich komplexere Verschachtelungen dar.

Anschließend wird der fertige Query auf dem Objekt der Collection ausgeführt. Hierfür bietet diese alle Methoden an, die auf der Konsole auf der Collection auch ausgeführt werden können, wie beispielsweise find(), count() und distinct(). Der Rückgabewert hier ist eine Liste von DBObjects wenn das Ergebnis ein Dokument verkörpert oder AggregationOutput wenn es sich um mehrere Dokumente handelt.

Der Vergleich der beiden Queries, wie sie auf der Konsole und im Quellcode konstruiert wurden, zeigt deutlich dass ein DBObject eine BSON-Zuweisung von Key zu Value *{ "Key" : "Value" }* verkörpert.

Test der Anwendng

Die acht vorgestellten Use-Cases haben sich vollständig mit Hilfe von MongoDB abbilden lassen. In Tabelle 4.1 wird aufgezeigt, wie viel Zeit benötigt wurde um die einzelnen Anfragen zu bearbeiten. Hierbei ist jedoch relevant zu erwähnen, dass die Geschwindigkeit der Beantwortung von der Auslastung der Server abhängig ist, die in diesem Fall nicht konstant war. Auffällig ist, dass in einigen Fällen die Anwendung eine schnellere Antwortzeit hatte, als eine Abfrage direkt auf dem Server. Dies verdeutlicht, dass MongoDB ein großes Potenzial in der Performance hat.

Tabelle 4.1: Ausführungszeiten der Use Cases in Sekunden

	Robomongo (in sec)	Anwendung (in sec)
Use Case 1	0,339	0,126
Use Case 2	0,534	0,137
Use Case 3	0,447	0,124
Use Case 4	0,319	0,577
Use Case 5	0,111	0,161
Use Case 6	0,517	0,461
Use Case 7	0,173	0,125
Use Case 8	0,078	0,039

Die für die Anfragen benutzten Eingaben in den Tests lauten wie folgt:

UC1: *"Datenbank"*

UC3: *"Prof. Harm Knolle"*

UC4: *"SS 07"*

UC7: *"Datenbank"*

UC8: *"Datenbank"*

Durch Indizierung von Attributen kann die Ausführungszeit von Queries um den Faktor zehn verbessert werden. Dies wurde während der Tests empirisch bewiesen.

In den gegebenen Use-Cases sind vor allem der Erstprüfer, das Thema der Arbeit, das Zeugnis und die Firma Attribute, die von größerer Relevanz sind. Daher bietet es sich an diese Attribute zu indizieren, um die Performance der Queries zu erhöhen. Die in Tabelle 4.1 aufgelisteten Zeiten wurden bereits durch Indizierung generiert. Folgende Befehle wurden für die Erstellung der Indizes ausgeführt:

Listing 4.2: Befehle zur Erstellung der Indizierung

```
db.abDBsimpleCSV.createIndex({Thema der Arbeit":1},{name:"thema"})
db.abDBsimpleCSV.createIndex({"1. Pruefer":1},{name:"pruefer"})
db.abDBsimpleCSV.createIndex({"Zeugnis":1},{name:"abschluss"})
db.abDBsimpleCSV.createIndex({"Firma":1},{name:"unternehmen"})
```

5 Fazit

5.1 MongoDB

Als NoSQL Datenbanklösung ist MongoDB vor allem für Entwickler eine Datenbank, die eine schnelle Einrichtung ermöglicht.

5.1.1 Zusammenfassung

Da die Befehle für die Interaktion mit der Datenbank JavaScript ähnlich waren, wurde die Arbeit mit der MongoDB vereinfacht. Die bereitgestellten Funktionalitäten und die ausführliche Dokumentation der MongoDB sowie die automatische Verwaltung der Datenbank hat einen schnellen Einstieg ermöglicht. Durch den Java Treiber von MongoDB war es problemlos möglich eine Java-basierte Web-Anwendung an die Datenbank anzubinden. Ferner hat sich gezeigt, dass die Datenbank eine äußerst gute Performance hat.

5.1.2 Pros

Vorteile der MongoDB sind:

- Einfache und schnelle Installation von MongoDB
- Sehr gute Dokumentation über Funktionen und Features
- Skalierbarkeit durch Sharding
- Geschwindigkeit
- Flexibilität
- JSON ähnliche Datenformat

5.1.3 Cons

Nachteile der MongoDB sind:

- CSV-Import-Funktion ist fehlerbehaftet
- MongoDB unterstützt kein JOIN verfahren, Referenzierungen müssen manuell durchgeführt werden
- Concurrency Issues
- Beschränkungen beim Daten-Import

5.1.4 Ausblick

Die entwickelte Web-Anwendung ließe sich weiterentwickeln und verfeinern zu einem Portal, über das Studenten Informationen zu potenziellen Themengebieten, Prüfern und Unternehmen finden können. Professoren könnten hierüber auf einfache Weise prüfen, welche Themen belegt sind und an welchen Stellen Potenzial gegeben ist. Ferner könnten Unternehmen das Portal nutzen, um Studenten für Abschlussarbeitsthemen zu finden. Dafür müsste die Anwendung allerdings um weitere Funktionen, wie beispielsweise eine Login-Komponente erweitert werden um die Sicherheit zu gewährleisten.

Eidesstattliche Erklärung

Ich versichere an Eides Statt durch meine eigenhändige Unterschrift, dass ich die entsprechend im Vorwort gekennzeichneten Abschnitte der vorliegenden Gruppenarbeit selbstständig und ohne fremde Hilfe angefertigt habe. Alle Stellen, die wörtlich oder dem Sinn nach auf Publikationen oder Vorträgen anderer Autoren beruhen, sind als solche kenntlich gemacht. Ich versichere außerdem, dass ich keine andere als die angegebene Literatur verwendet habe. Diese Versicherung bezieht sich auch auf alle in der Arbeit enthaltenen Zeichnungen, Skizzen, bildlichen Darstellungen und dergleichen.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

den 14. Januar 2017
Ort, Datum

Franz-Dominik Dahmann,
Jan Eric Müller, Thomas
Tobias Zok

Zusammenfassung

Moderne Datenbanken sehen sich mit diversen Problemen konfrontiert. Es sollen extrem große Datenmengen verarbeitet werden und die Antwortzeiten sollen niedrig gehalten werden. Ferner soll es für sehr viele Nutzer gleichzeitig möglich sein Abfragen zu stellen. Aus diesen Gründen ist es in modernen Datenbanken notwendig ist, Datenstrukturen möglichst flexibel und performant zu speichern. Diese Probleme sollen mit verschiedenen Ansätzen aus dem Umfeld der NoSQL Datenbanksysteme gelöst werden. Die dokumentenbasierte Speicherung versucht für Probleme, welche sich meist auf große Mengen mit heterogenen Daten beziehen, einen Lösungsansatz zu finden. In diesem Umfeld ist die Funktionsweise der MongoDB äußerst populär, da sie auf hohe Leistung, große Datenmengen, hohe Flexibilität und einfache Skalierbarkeit ausgelegt ist. Die Grundlage für die der MongoDB zugrunde liegenden dokumentenbasierte Speicherung bieten die Key-Value Stores, welche unformatierte Werte unter einem Key persistieren. Document Stores vereinen den Vorteil der Schemafreiheit der Schlüssel-Wert Zuordnungen mit der Möglichkeit zur Strukturierung von Daten. Dadurch sind Datenbanken wie MongoDB in der Lage komplexere JSON ähnliche Datentypen zu verarbeiten und basierend auf ihren Attributen zu indexieren. Die hier umrissenen theoretischen Grundlagen werden detailierter beschrieben und in dem NoSQL Kontext bewertet.

Um die theoretischen Grundlagen zu verdeutlichen werden mithilfe eines Anwendungsszenarios die Vorteile von MongoDB verdeutlicht. Grundlage für das Anwendungsszenario ist das logische Schema einer relationalen Datenbank, in der sowohl Abschlussarbeitsthemen, als auch alle beteiligten Personen, wie beispielsweise Studenten oder Prüfer, sowie Organisationen, gespeichert sind. Ziel des Anwendungsszenarios ist es, dieses Schema mit Hilfe von MongoDB umzusetzen. Ferner soll die Eignung von MongoDB in diesem Kontext analysiert und bewertet werden. Damit MongoDB in diesem Kontext bewertet werden kann, werden unter anderem Performancetests an der Datenbank durchgeführt. Weiter soll eine Web-Anwendung implementiert werden, welche die grundsätzlichen Datenbankoperationen an der MongoDB veranschaulicht.

6 Anhänge

6.1 Use Cases

Use Case 1 Als Student möchte ich herausfinden welche Professoren in der Vergangenheit Themen unterstützt haben, die mich interessieren.
Input: Thema das mich interessiert (als String).
Output: Liste mit Professoren die im Zusammenhang mit dem Thema 1. Prüfer waren.
Query:

Listing 6.1: Query zu Use Case 1

```
db.getCollection('abDBsimpleCSV')
        .distinct('1_Pruefer',{"Thema_der_Arbeit":/.*Thema_der_Arbeit.*/})
```

Quellcode:

Listing 6.2: Quellcode zu Use Case 1

```
1 public List< DBObject > getQuery1( String input )
2 {
3     Pattern pat = Pattern.compile( input, Pattern.CASE_INSENSITIVE );
4     DBObject query;
5     query = new BasicDBObject( "Thema_der_Arbeit", pat );
6     List< DBObject > answer = coll.distinct( "1_Pruefer", query );
7     return answer;
8 }
```

Use Case 2 Als Student möchte ich wissen, welche Professoren Bachelor- und Masterthesen anbieten oder angeboten haben.
Output: Liste von allen Professoren.
Query:

Listing 6.3: Query zu Use Case 2

```
db.getCollection('abDBsimpleCSV')
        .distinct('1_Pruefer',{$or:[{"1_Pruefer":{$ne:""}},
                {"2_Pruefer":{$ne:""}},{"3_Pruefer":{$ne:""}}]})
```

Quellcode:

Listing 6.4: Quellcode zu Use Case 2

```
1 public List< DBObject > getQuery2()
2 {
```

```
3       DBObject nequery = new BasicDBObject( "$ne", "" );
4
5       List< DBObject > obj = new ArrayList<>();
6       obj.add( new BasicDBObject( "1_Pruefer", nequery ) );
7       obj.add( new BasicDBObject( "2_Pruefer", nequery ) );
8       obj.add( new BasicDBObject( "3_Pruefer", nequery ) );
9
10      DBObject orquery = new BasicDBObject( "$or", obj );
11
12      List< DBObject > answer = coll.distinct( "1_Pruefer", orquery );
13      return answer;
14  }
```

Use Case 3 Als Student möchte wissen welche Themen in der Vergangenheit von einem bestimmten Professor angeboten wurden.
Input: Professorname.
Output: Liste mit Themen bei welchem der Professor Prüfer war.
Query:

Listing 6.5: Query zu Use Case 3

```
db.getCollection('abDBsimpleCSV')
        .distinct('Thema_der_Arbeit',{$or:[{"1_Pruefer":"Professorname"},
                {"2_Pruefer":"Professorname"},
                {"3_Pruefer":"Professorname"}]})
```

Quellcode:

Listing 6.6: Quellcode zu Use Case 3

```
1   public List< DBObject > getQuery3( String input )
2   {
3       Pattern pat = Pattern.compile( input, Pattern.CASE_INSENSITIVE );
4       DBObject query;
5       query = new BasicDBObject( "1_Pruefer", pat );
6       List< DBObject > answer = coll.distinct( "Thema_der_Arbeit", query );
7       return answer;
8   }
```

Use Case 4 Als Professor möchte ich mir die Anzahl der der erfolgreichen Abschlüsse für ein bestimmtes Semester ausgeben lassen.
Input: Semester.
Output: Anzahl der Abschlüsse.
Query:

Listing 6.7: Query zu Use Case 4

```
db.abDBsimpleCSV.find({$and:[{Semester: "(Semstername)", Zeugnis: 1}]}).count()
```

Quellcode:

Listing 6.8: Quellcode zu Use Case 4

```
1   public int getQuery4( String input )
2   {
3       List< DBObject > obj = new ArrayList<>();
4       obj.add( new BasicDBObject( "Semester", input ) );
```

```
5       obj.add( new BasicDBObject( "Zeugnis", 1 ) );
6       DBObject andQuery = new BasicDBObject( "$and", obj );
7       int i = coll.find( andQuery ).count();
8       return i;
9   }
```

Use Case 5 Als Student möchte ich wissen, welche Unternehmen Bachelor- und Masterthesen betreut haben.
Output: Liste aller Unternehmen.
Query:

Listing 6.9: Query zu Use Case 5

```
db.abDBsimpleCSV.distinct('Firma',{Firma: {$ne:""}})
```

Quellcode:

Listing 6.10: Quellcode zu Use Case 5

```
1   public List< DBObject > getQuery5()
2   {
3       BasicDBObject nequery = new BasicDBObject( "$ne", "" );
4       BasicDBObject query = new BasicDBObject( "Firma", nequery );
5       List< DBObject > answer = coll.distinct( "Firma", query );
6       return answer;
7   }
```

Use Case 6 Als Professor oder als Student möchte ich mir anzeigen lassen, welcher Professor bisher die meisten Abschlussarbeiten unterstützt hat.
Output: Liste mit Professoren und der Anzahl an unterstützten Abschlussarbeiten.
Query:

Listing 6.11: Query zu Use Case 6

```
db.abDBsimpleCSV
        .aggregate([{$match:{"1_Pruefer":
                {$not: {$size: 0}}}},
                {$unwind: '$1_Pruefer'},
                {$group: {_id: '$1_Pruefer', Anzahl: {$sum: 1}}},
                {$match: {Anzahl: {$gte: 2}}},
                {$sort: {Anzahl: -1}} ,
                {$limit:200}])
```

Quellcode:

Listing 6.12: Quellcode zu Use Case 6

```
1   public AggregationOutput getQuery6()
2   {
3       List< DBObject > obj = new ArrayList<>();
4       //Erstes Element der Queryliste
5       Object sizeQuery = new BasicDBObject( "$size", 0 );
6       DBObject neQuery = new BasicDBObject( "$not", sizeQuery );
7       DBObject matchQuery = new BasicDBObject( "1_Pruefer", neQuery );
8       obj.add( new BasicDBObject( "$match", matchQuery ) );
9       //Zweites Element der Queryliste
10      obj.add( new BasicDBObject( "$unwind", "$1_Pruefer" ) );
```

```
11      //Drittes Element der Queryliste
12      DBObject sumQuery = new BasicDBObject( "$sum", 1 );
13      DBObject auswahlQuery = new BasicDBObject( "_id", "$1_Pruefer" ).append( "
            Anzahl", sumQuery );
14      obj.add( new BasicDBObject( "$group", auswahlQuery ) );
15      //Viertes Element der Queryliste
16      DBObject anzahlQuery, greaterQuery;
17      greaterQuery = new BasicDBObject( "$gte", 2 );
18      anzahlQuery = new BasicDBObject( "Anzahl", greaterQuery );
19      obj.add( new BasicDBObject( "$match", anzahlQuery ) );
20      //Fuenftes Element der Queryliste
21      DBObject sortQuery = new BasicDBObject( "Anzahl", -1 );
22      obj.add( new BasicDBObject( "$sort", sortQuery ) );
23      //Sechstes Element der Queryliste
24      obj.add( new BasicDBObject( "$limit", 200 ) );
25      AggregationOutput answer = coll.aggregate( obj );
26      return answer;
27  }
```

Use Case 7 Als Professor möchte ich wissen, welche Themen bereits vorgeschlagen oder bearbeitet worden sind.
Input: Thema.
Output: Liste aller Themen die dem Input entsprechen.
Query:

Listing 6.13: Query zu Use Case 7

```
db.abDBsimpleCSV.distinct('Thema_der_Arbeit',{"Thema_der_Arbeit":/.*(Thema).*/})
```

Quellcode:

Listing 6.14: Quellcode zu Use Case 7

```
1  public List< DBObject > getQuery7( String input )
2  {
3      Pattern pat = Pattern.compile( input, Pattern.CASE_INSENSITIVE );
4      DBObject query;
5      query = new BasicDBObject( "Thema_der_Arbeit", pat );
6      List< DBObject > answer = coll.distinct( "Thema_der_Arbeit", query );
7      return answer;
8  }
```

Use Case 8 Welche Professoren haben wieviele Arbeiten zu einem Thema betreut.
Input: Stichwort zum Thema (Such-String).
Output: Anzahl Themen mit Such-String gruppiert nach Name des Profs, sortiert nach Anzahl Themen.
Query:

Listing 6.15: Query zu Use Case 8

```
db.abDBsimpleCSV
        .aggregate([
                {$match:{"Thema_der_Arbeit":/.*datenbank*/}},
                {$match:{"1_Pruefer": {$not: {$size: 0}}}},
                {$unwind: '$1_Pruefer'},
                {$group: {_id: '$1_Pruefer', Anzahl: {$sum: 1}}},
                {$sort: {Anzahl: -1}} ,{$limit:200}])
```

Quellcode:

Listing 6.16: Quellcode zu Use Case 8

```
1  public AggregationOutput getQuery8( String input )
2  {
3      List< DBObject > obj = new ArrayList<>();
4      Pattern pat = Pattern.compile( input, Pattern.CASE_INSENSITIVE );
5      DBObject query;
6      query = new BasicDBObject( "Thema_der_Arbeit", pat );
7      BasicDBObject mQuery = new BasicDBObject( "$match", query );
8      obj.add( mQuery );
9      DBObject sizeQuery = new BasicDBObject( "$size", 0 );
10     DBObject neQuery = new BasicDBObject( "$not", sizeQuery );
11     DBObject matchQuery = new BasicDBObject( "1_Pruefer", neQuery );
12     obj.add( new BasicDBObject( "$match", matchQuery ) );
13     obj.add( new BasicDBObject( "$unwind", "$1_Pruefer" ) );
14     DBObject sumQuery = new BasicDBObject( "$sum", 1 );
15     DBObject auswahlQuery = new BasicDBObject( "_id", "$1_Pruefer" ).append( "
           Anzahl", sumQuery );
16     obj.add( new BasicDBObject( "$group", auswahlQuery ) );
17     DBObject sortQuery = new BasicDBObject( "Anzahl", -1 );
18     obj.add( new BasicDBObject( "$sort", sortQuery ) );
19     AggregationOutput answer = coll.aggregate( obj );
20     return answer;
21 }
```

Literaturverzeichnis

[Boi12] Boicea, Alexandru: *MongoDB vs Oracle – Database Comparison.* Researchgate, 2012

[Dra16] Drapeau, Martin: *http://www.csvjson.com/csv2json.* Drapeau, Martin, 2016

[Har15] Harrison, Guy: *Next Generation Databases.* Apress, 2015

[Mon16] MongoDB: *https://docs.mongodb.com/v3.2/.* MongoDB, 2016

[Sad13] Sadalage, Pramod J.: *NoSQL: A Brief Guide to the Emerging World of Polyglot Persistence.* Addison-Wesley, 2013

[Sha15] Shakuntala, Gupta E.: *Practical MongoDB Architecting, Developing, and Administering MongoDB.* APRESS, 2015