Implementierung und Performanceunterschiede flexibler Schemata in MariaDB und MongoDB

Thomas Steidl

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2017

© Copyright 2017 Thomas Steidl

Diese Arbeit wird unter den Bedingungen der Creative Commons Lizenz *Attribution-NonCommercial-NoDerivatives 4.0 International* (CC BY-NC-ND 4.0) veröffentlicht – siehe https://creativecommons.org/licenses/by-nc-nd/4.0/.

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 26. Juni 2017

Thomas Steidl

Inhaltsverzeichnis

Kurzfassung

In Zeiten von Big Data Anwendungen mit enormen Mengen an strukturierten und unstrukturierten Daten, hat die Wahl des richtigen Datenbanksystems in den letzten Jahren an Bedeutung gewonnen. Relationale Datenbanken waren über mehrere Jahrzehnte sehr dominant und weit verbreitet. Durch die komplexen Anforderungen können viele moderne Anwendungen jedoch nicht mehr mit dem klassischen Relationenmodell umgesetzt werden. Dadurch haben sich in den letzten Jahren immer mehr NoSQL-Datenbanken für bestimmte Anwendungsfälle durchgesetzt. Eine große Stärke dieser Systeme ist die Schema-Flexibilität. Dadurch können Daten mit unterschiedlichsten Strukturen gespeichert werden. *MongoDB* gehört dabei zu den bekanntesten Vertretern unter den NoSQL-Datenbanken. *MariaDB* hat in den letzten Jahren einen starken Aufschwung unter den relationalen Datenbanken erlebt. Durch den Einsatz flexibler Schemata bringt *MariaDB* einen interessanten Ansatz aus den NoSQL-Datenbanken in die Welt der relationalen Datenbanken. Konkrete Zeitmessungen sollen Aufschluss über die Performanceunterschiede flexibler Schemata zwischen *MariaDB* und *MongoDB* geben.

Abstract

Over the past few years, where big data applications have been confronted with vast amounts of structured and unstructured data, choosing the most suitable database system has now become an important factor. For many centuries, relational databases dominated the scene and were duly widespread. However, these classical relational systems can no longer meet the complex requirements of the modern implementation needed. As a result, an increasing number of NoSQL databases have been implemented for particular applications over the past few years. One of the greatest strengths of these systems is the schema flexibility: data of very diverging structures can be stored. *MongoDB* is one of the most well known amongst the NoSQL databases. *MariaDB* has experienced a great upsurge within the relational databases in the last few years. Due to the usage of flexible schemata, *MariaDB* has now introduced an interesting approach for the relational database systems, arising from NoSQL sources. Specific chronometries should provide information about the differences in the performance of the flexible schemata between *MariaDB* and *MongoDB*.

Kapitel 1

Einleitung

Die Speicherung von riesigen Datenmengen hat in den letzten Jahren zunehmend an Bedeutung gewonnen. Relationale Datenbanken waren über Jahrzehnte hinweg die am weitesten verbreitete Gruppe unter den Datenbanksystemen. Mit den Anforderungen heutiger Big Data Anwendungen stoßen relationale Datenbanken jedoch oft an ihre Grenzen. Viele dieser Anforderungen können mit dem klassischen Relationenmodell nicht mehr bewältigt werden.

Daher haben NoSQL-Datenbanken in den letzten Jahren zunehmend an Relevanz für viele Anwendungsfälle gewonnen. NoSQL-Datenbanken verfolgen nicht den relationalen Ansatz, sondern zeichnen sich als schemafreie Datenbanken aus. Durch diese Schema-Flexibilität können riesige Datenbestände in unterschiedlichsten Formen persistiert werden. Weitere große Stärken weisen NoSQL-Datenbanken in den Bereichen Skalierung sowie Sicherheit auf.

Zielsetzung

Ziel dieser Arbeit ist es, einen Vergleich zwischen relationalen Datenbanken und NoSQL-Datenbanken zu ziehen. Dabei wird konkret auf die Systeme *MariaDB* aus der relationalen Welt sowie *MongoDB* aus der NoSQL-Welt eingegangen. *MariaDB* bringt mit der Einführung der Dynamic Columns eine gewisse Flexibilität in das relationale Datenbankmodell. *Mongoose* ist ein interessanter Ansatz um Struktur in die *MongoDB* zu bringen.

Mit auf *Node.js* basierenden Performance Tests sollen in dieser Arbeit Unterschiede zwischen diesen beiden Ansätzen flexibler Schemata aufgezeigt und analysiert werden. Weiters wird auf die Implementierung eines auf *Mongoose* basierenden Administrations Tool für *MongoDB* genauer eingegangen. Dieses Tool war ebenfalls Basis zur Umsetzung und Durchführung der angesprochenen Performance Tests.

Aufbau der Arbeit

Die vorliegende Arbeit ist in einen theoretischen und einen praktischen Teil gegliedert. In Kapitel 2 wird auf technische Grundlagen für das weitere Verständnis der Arbeit eingegangen. Dabei werden allgemeine Begriffe aus dem Bereich Datenbanken erläutert und genauer zwischen relationalen Datenbanken bzw. NoSQL-Datenbanken unterschieden.

In Kapitel 3 werden anschließend konkret die beiden Systeme *MariaDB* und *MongoDB* genauer beleuchtet. Dabei wird auf die für diese Arbeit relevanten Ansätze der Dynamic Columns unter *MariaDB* sowie auf *Mongoose* für *MongoDB* im Detail eingegangen. Außerdem werden Begriffe wie Replikation bzw. Sharding verständlich erklärt.

Kapitel 4 soll danach einen Einblick in die Implementierung des *MongoDB* Administrations Tool sowie in die Umsetzung der Performance Tests geben. Dabei wird im Detail auf die Anforderungen, verwendete Technologien sowie die Architektur der Implementierung eingegangen.

Das Kapitel 5 dient zur kritischen Betrachtung das verwendeten Frameworks bzw. der Ergebnisse der durchgeführten Performance Tests. Zusammenfassend bildet Kapitel 6 mit einem kurzen Fazit den Abschluss dieser Arbeit.

Kapitel 2

Technische Grundlagen

In diesem Kapitel werden die technischen Grundlagen für das weitere Verständnis der Arbeit erklärt. Dabei wird auf Begriffe wie Datenbanksysteme, relationale Datenbanken, Big Data sowie NoSQL-Datenbanken eingegangen. Außerdem gibt es eine kurze Einführung in die Datenbanksprache SQL. Mehr dazu in [2].

2.1 Datenbanksysteme

Ein *Datenbanksystem* (DBS) ist für die Datenspeicherung und Datenorganisation zuständig und besteht im Wesentlichen aus einer Speicherungs- und einer Verwaltungskomponente (Abbildung 2.1). Die *Datenbank* (DB) ist als Speicherungskomponente für die Persistierung der Daten sowie deren Beschreibung in einer organisierten Form verantwortlich. Das *Datenbankmanagementsystem* (DBMS) stellt als Verwaltungskomponente die Benutzerschnittstelle zum Auswerten bzw. Verändern der Daten und Informationen dar. Um den komplexen Anforderungen an eine konsistente und einheitliche Datenhaltung gerecht zu werden, muss ein Datenbanksystem eine Reihe von wichtigen Merkmalen aufweisen:

- Abfragesprache,
- Datenschutz,
- Datensicherheit,
- Datenunabhängigkeit,
- Mehrbenutzerfähigkeit,
- Parallelzugriff,
- u. v. m.

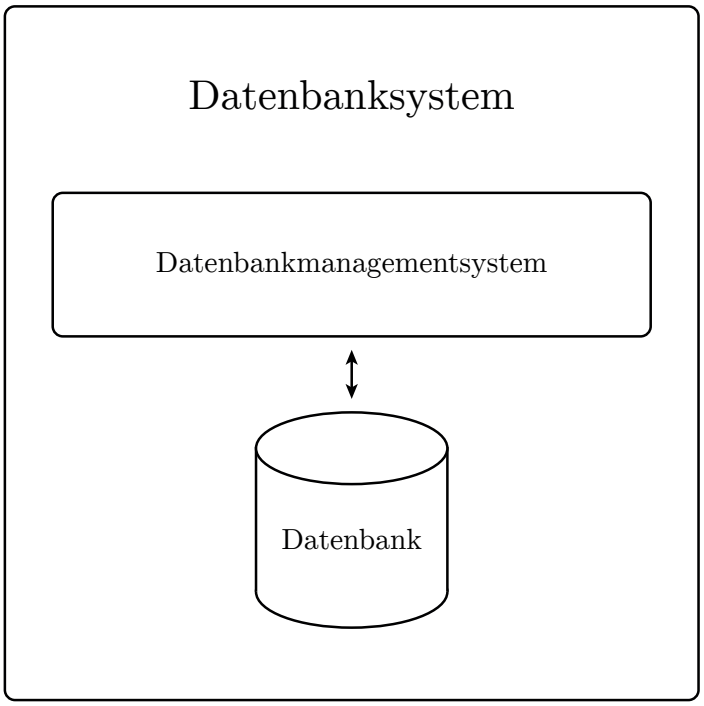


Abbildung 2.1: Aufbau eines Datenbanksystems

2.1.1 Datenbankmanagementsystem

Das *Datenbankmanagementsystem* (DBMS) stellt als Software die Schnittstelle für den Benutzer zur Datenbank dar. Sie wird für jedes Datenbanksystem individuell installiert und konfiguriert. Im DBMS wird darüber hinaus das Datenbankmodell festgelegt, welches maßgeblich über die Funktionalität und die Geschwindigkeit des Datenbanksystems entscheidet.

Wesentliche Funktionen von Datenbankmanagementsystemen sind die Speicherung, Überschreibung und Löschung von Daten, die Verwaltung von Metadaten sowie die Optimierung von Abfragen. Weiters ist es für Vorkehrungen im Bereich Datensicherheit, Datenschutz und Datenintegrität zuständig und ermöglicht den Mehrbenutzerbetrieb durch das Transaktionskonzept.

Bei der Wahl des richtigen Datenbankmanagementsystems gilt es mehrere Faktoren zu berücksichtigen. Es sollte evaluiert werden wie viele Benutzer mit der Datenbank interagieren bzw. welche Art und Menge von Daten in der Datenbank erfasst werden. Ein weiterer wichtiger Aspekt für die richtige Wahl ist das Thema Datenbanksicherheit. Die am meisten verwendeten Datenbankmanagementsysteme sind die relationalen Datenbankmanagementsysteme (RDBMS).

2.1.2 Datenbank

Die *Datenbank* (DB), zum Teil auch „Datenbasis“ genannt, speichert Informationen in Form von einzelnen Datensätzen ab. Sie muss in der Lage sein, mit großen Datenmengen effizient, widerspruchsfrei und dauerhaft umgehen sowie logische Zusammenhänge digital abbilden zu können.

Beim Starten der Datenbank werden die Datenblöcke vom Speichermedium in den Hauptspeicher übertragen. Die laufende Datenbank wird als Datenbankinstanz bezeichnet, wobei es mehrere gleichzeitig laufende Instanzen geben kann.

2.1.3 Datenbankmodell

Das *Datenbankmodell* ist die theoretische Grundlage für eine Datenbank. Es bestimmt, auf welche Art und Weise die Daten im Datenbanksystem gespeichert bzw. bearbeitet werden können. Das Datenbankmodell besteht aus drei wichtigen Faktoren: einer generischen Datenstruktur, einer Menge von generischen Operatoren und einer Menge von Integritätsbedingungen. Die bekanntesten Datenbankmodelle sind:

- Hierarchisches Datenbankmodell,
- Netzwerkdatenbankmodell,
- relationales Datenbankmodell,
- objektrelationales Datenbankmodell,
- objektorientiertes Datenbankmodell,
- dokumentenorientiertes Datenbankmodell.

2.1.4 Datenbankschema

Das *Datenbankschema* beschreibt formal die Struktur der Daten, d. h. es wird festgelegt, in welcher Form welche Daten in der Datenbank persistiert werden und in welchen Beziehungen diese Daten zueinander stehen. Jeder Datensatz muss exakt der Definition im Datenbankschema entsprechen um in die Datenbank aufgenommen zu werden. Das Datenbankschema wird im Data Dictionary einer Datenbank hinterlegt. Änderungen am Datenbankschema sind grundsätzlich möglich, jedoch müssen anschließend alle Datensätze aktualisiert werden, um wieder dem Datenbankschema zu entsprechen. Allgemein gibt es verschiedene Arten von Schemadefinitionen:

- CSV Schema,
- XML Schema,
- Semantic Web Schema,
- Datenbankschema.

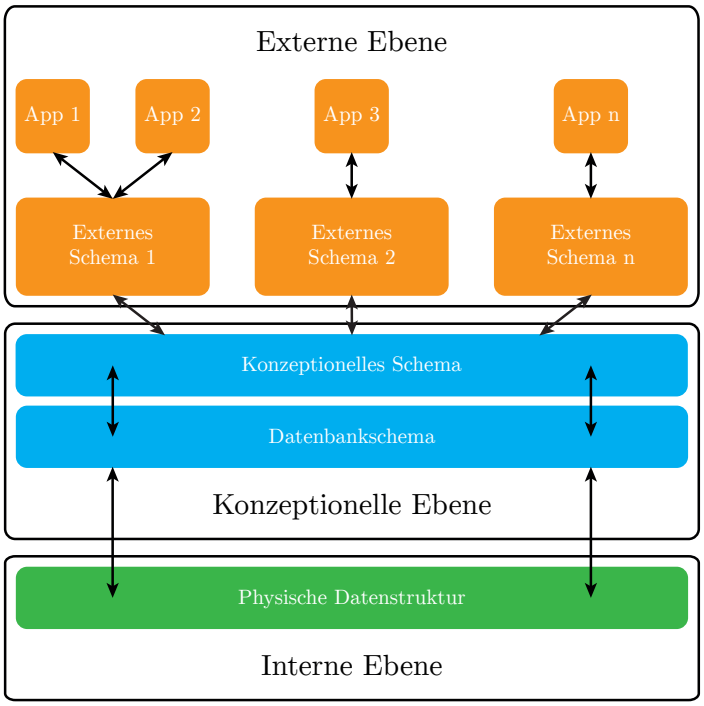


Abbildung 2.2: Aufbau relationaler Datenbanken [13]

2.2 Relationale Datenbanken

Die Grundidee *relationaler Datenbanken* ist es, Daten in Form von Relationen zu beschreiben und in Tabellen zu speichern. Jede Zeile in der Tabelle wird als Datensatz bezeichnet und besteht aus verschiedenen Attributen, den Spalten der Tabelle. Im Relationenschema werden die Attribute für eine Relation genauer festgelegt. Weiters können Verknüpfungen verwendet werden, um Beziehungen zwischen Tabellen auszudrücken.

Der grundlegende Aufbau relationaler Datenbanken ist in der *ANSI-SPARC-Architektur*, auch *Drei-Ebenen-Architektur* genannt, sehr gut beschrieben. Diese Architektur besteht aus drei verschiedenen Datenbankschemata, die in unterschiedliche Ebenen aufgeteilt sind (Abbildung 2.2). Ziel dieser Abstrahierung ist es, eine Unabhängigkeit von der Programmiersprache und der darunterliegenden Hardware zu gewährleisten. Durch die daraus resultierende physische und logische Datenunabhängigkeit kann eine höhere Robustheit gegenüber Änderungen am Datenbanksystem erzielt werden.

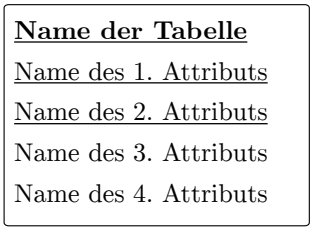


Abbildung 2.3: Diagramm-Notation

Externe Ebene

Die *externe Ebene* stellt den Benutzern und Anwendungen individuelle Sichten zur Verfügung. Jede Sicht beschreibt die Eigenschaften einer Gruppe von Anwendern, die dadurch nur einen Teil der hinterlegten Daten sehen. So wird gewährleistet, dass diese Gruppen nur die Informationen bzw. Daten sehen können, die sie auch sehen dürfen.

Konzeptionelle Ebene

In der *konzeptionellen Ebene* wird beschrieben, welche Daten in der Datenbank zu persistieren sind und in welchen Beziehungen diese zueinander stehen. Ziel dieser Ebene ist die vollständige und redundanzfreie Darstellung sämtlicher zu speichernden Informationen.

Interne Ebene

Die *interne Ebene* stellt die physischen Speicherstrukturen und Zugriffsmechanismen der Datenbank dar. Die Beschreibung enthält die Informationen, wie bzw. wo die Daten in der Datenbank persistiert werden und Zugriffsverfahren, durch die gespeicherte Daten wieder abgerufen werden können.

2.2.1 Datenmodellierung

Die *Datenmodellierung* kann als wichtigster Schritt bei der Entwicklung eines erfolgreichen relationalen Datenbanksystems gesehen werden. Ziel ist es, ein optimales Datenbankschema für die jeweiligen Daten zu finden, damit alle benötigten Daten im Schema abgespeichert werden können, der Datenzugriff effizient erfolgen kann und die Datenkonsistenz gewährleistet ist.

Für die Datenmodellierung von verschiedenen Tabellen hat sich eine übersichtliche Diagramm-Notation bewährt (Abbildung 2.3). In der Kopfzeile des Diagramms steht der Name des Schemas bzw. der Tabelle. Darunter werden die einzelnen Attribute aufgelistet, wobei die Attribute des Primärschlüssels zur Unterscheidung unterstrichen sind.

2.2.2 Beziehungen

In einer relationalen Datenbank können Tabellen bzw. Relationen in unterschiedlichen *Beziehungen* zueinander stehen. Dabei sind grundsätzlich drei Beziehungstypen zu unterscheiden, welche in den nachfolgenden Abschnitten kurz erklärt werden [14].

1:1-Beziehung

In einer *1:1-Beziehung* ist jeder Datensatz in Tabelle A genau einem Datensatz in Tabelle B zugeordnet und umgekehrt. Beispiel:

> Es gibt nur einen Ehemann und eine Ehefrau.

1:n-Beziehung

In einer *1:n-Beziehung* können einem Datensatz in Tabelle A mehrere passende Datensätze in Tabelle B zugeordnet sein, aber einem Datensatz in Tabelle B ist nie mehr als ein Datensatz in Tabelle A zugeordnet. Beispiel:

> Es gibt ein Museum mit mehreren Kunstwerken.

m:n-Beziehung

Bei einer *m:n-Beziehung* können jedem Datensatz in Tabelle A mehrere passende Datensätze in Tabelle B zugeordnet sein und umgekehrt. Diese Beziehungen können nur über eine dritte Tabelle, die Verbindungstabelle C, realisiert werden. Beispiel:

> Ein Student hört sich Vorlesungen mehrerer Professoren an, und
> ein Professor unterrichtet mehrere Studenten.

2.2.3 Normalisierung

Unter *Normalisierung* versteht man in relationalen Datenbanken die Aufteilung von Attributen in mehrere Tabellen. Ziel der Normalisierung ist es, eine völlig redundanzfreie Datenspeicherung zu erzielen. Redundanzfrei bedeutet in diesem Zusammenhang, dass es möglich sein muss, Daten ohne Informationsverlust löschen zu können. Für den Normalisierungsprozess gibt es Normalisierungsregeln und fünf Normalformen. In den nachfolgenden Abschnitten wird anhand eines konkreten Beispiels genauer auf die ersten drei Normalformen eingegangen [23]. Ausgangspunkt für die Überführung in die einzelnen Normalformen ist eine Tabelle, in der Rechnungsdaten gespeichert sind (Tabelle 2.1). Man spricht dabei von der unnormalisierten, nullten Normalform.

Tabelle 2.1: Nullte Normalform

ReNr	*Datum*	*Name*	*Straße*	*Ort*	*Artikel*	*Anzahl*	*Preis*
187	01.01.2017	Max Mustermann	Musterstr 1	12345 Musterort	Stift	2	1,00 €

Tabelle 2.2: Erste Normalform

ReNr	*Datum*	*Name*	*Vorname*	*Straße*	*PLZ*	*Ort*	*Artikel*	*Anzahl*	*Preis*
187	01.01.2017	Mustermann	Max	Musterstr. 1	12345	Musterort	Stift	2	1,00 €

Tabelle 2.3: Zweite Normalform (Rechnungen)

Nr	*Datum*	*KundenNr*
187	01.01.2017	007

Erste Normalform (1NF)

Die *erste Normalform* (1NF) ist dann erreicht, wenn alle Attribute in einer Tabelle atomar vorliegen. Das bedeutet, dass jede Information einer Tabelle eine eigene Spalte bekommt. Im vorliegenden Beispiel werden dadurch zusammenhängende Informationen wie Vor- und Nachname bzw. Postleitzahl (PLZ) und Ort in separate Spalten aufgeteilt (Tabelle 2.2).

Zweite Normalform (2NF)

Die *zweite Normalform* (2NF) ist dann erreicht, wenn sich die Tabelle in der ersten Normalform befindet und jedes Nichtschlüsselattribut von jedem Schlüsselkandidaten voll funktional abhängig ist. Im konkreten Beispiel bedeutet das, dass eine neue Tabelle für Kunden entsteht, da ein Name nicht eindeutig ist und so jeder Kunde nach der Kundennummer identifiziert wird (Tabelle 2.4). Gleiches Prinzip gilt auch für die Artikel (Tabelle 2.6). Weiters wird eine Rechnungspositionstabelle erstellt, da eine Rechnung eines Kunden mehrere Rechnungspositionen mit unterschiedlichen Artikeln enthalten kann (Tabelle 2.5).

Tabelle 2.4: Zweite Normalform (Kunden)

KundeNr	*Name*	*Vorname*	*Straße*	*PLZ*	*Ort*
007	Mustermann	Max	Musterstr. 1	12345	Musterort

Tabelle 2.5: Zweite Normalform (Rechnungspositionen)

RePosNr	*ReNr*	*ArtNr*	*Anzahl*
1	187	69	2

Tabelle 2.6: Zweite Normalform (Artikel)

ArtNr	*Artikel*	*Preis*
69	Stift	1,00 €

Tabelle 2.7: Dritte Normalform (Kunden)

KundenNr	*Name*	*Vorname*	*Straße*	*PLZ*
007	Mustermann	Max	Musterstr. 1	12345

Dritte Normalform (3NF)

Die *dritte Normalform* (3NF) kann als Ziel einer erfolgreichen Normalisierung gesehen werden. Die dritte Normalform ist also dann erreicht, wenn die zweite Normalform erreicht ist und kein Nichtschlüsselattribut transitiv von einem Kandidatenschlüssel abhängt. Im angeführten Beispiel betrifft das die Tabelle mit den Kundendaten. Darin sind der Vorname, die Straße und die Postleitzahl (PLZ) abhängig vom Namen, jedoch nicht vom Primärschlüssel, der Kundennummer. Weiters ist der Ort abhängig von der Postleitzahl (PLZ). Die Aufteilung der Kundentabelle ist in Tabelle 2.7 sowie Tabelle 2.8 ersichtlich. Das Erreichen der dritten Normalform gewährleistet die perfekte Balance aus Redundanz und Flexibilität einer relationalen Datenbank.

2.2.4 Datenbanksprache SQL

Relationale Datenbanken werden sehr oft auch als SQL-Datenbanken bezeichnet. SQL (englisch für Structured Query Language) ist eine strukturierte Abfragesprache zum Bearbeiten (Einfügen, Verändern, Löschen) sowie Abfragen von Datenbeständen in relationalen Datenbanken. In den nachfolgenden Abschnitten wird kurz auf die wichtigsten SQL-Grundlagen eingegangen.

Tabelle 2.8: Dritte Normalform (Postleitzahlen)

PLZ	*Ort*
12345	Musterort

CREATE

Mit dem `CREATE` Statement können Datenbanken bzw. Tabellen erzeugt werden. Nach der eigentlichen Anweisung folgt der Name der zu erzeugenden Datenbank bzw. Tabelle:

```
CREATE DATABASE datenbankname;
CREATE TABLE tabellenname;
```

DROP

Mit dem `DROP` Statement können Datenbanken bzw. Tabellen gelöscht werden. Dazu muss hinter der Anweisung nur der Name der zu löschenden Datenbank bzw. Tabelle angegeben werden:

```
DROP DATABASE datenbankname;
DROP TABLE tabellenname;
```

INSERT

Mit dem `INSERT` Statement können Datensätze in Tabellen eingefügt werden. Dabei ist zu beachten, dass die Spaltennamen nur angegeben werden müssen, wenn nicht alle Spalten der Tabelle befüllt werden:

```
INSERT INTO tabellenname (spalte1, spalte2, spalte3, ...)
VALUES (wert1, wert2, wert3, ...);
```

UPDATE

Mit dem `UPDATE` Statement ist es möglich, Werte bestehender Datensätze einer Tabelle zu modifizieren. Dabei werden mit der `SET` Anweisung alle Werte der angegebenen Spalten unter Berücksichtigung der Bedingung in der `WHERE` Klausel überschrieben:

```
UPDATE tabellenname
SET spalte1 = wert1, spalte2 = wert2, ...
WHERE bedingung;
```

DELETE

Mit dem `DELETE` Statement kann ein bestimmter Datensatz unter Berücksichtigung der Bedingung in der `WHERE` Klausel aus einer Tabelle entfernt werden:

```
DELETE FROM tabellenname
WHERE bedingung;
```

SELECT

Mit dem `SELECT` Statement können Datensätze aus einer Tabelle abgefragt werden. Dabei werden nur bestimmte Spalten unter Berücksichtigung der Bedingung in der `WHERE` Klausel aus der Tabelle selektiert:

```
SELECT spalte1, spalte2, ...
FROM tabellenname
WHERE bedingung;
```

JOIN

Mit dem `JOIN` Statement können Datensätze aus mehreren Tabellen kombiniert werden. Dabei sind Inner Joins, Left Joins, Right Joins, Full Joins sowie Self Joins zu unterscheiden, worauf jedoch im Zuge dieser Arbeit nicht detaillierter eingegangen wird. Mehr dazu in [9].

2.3 Big Data

In Zeiten von sozialen Netzwerken wie *Facebook*, *Twitter* und Co. mit Millionen von Benutzern ist das Thema Big Data wichtiger denn je. Durch die komplexen Anforderungen solcher Anwendungen ist es nicht mehr möglich, die Daten in Form von relationalen Datenbanken abzubilden. Die *Gartner Group* findet in ihrem IT-Glossar dazu folgende Definition [24]:

> Big data is high-volume, high-velocity and/or high-variety information assets that demand cost-effective, innovative forms of information processing that enable enhanced insight, decision making, and process automation.

2.3.1 Volume

Der Aspekt *Volume* bezieht sich auf die enormen Mengen an Daten, die täglich in unterschiedlichster Form generiert werden und mit herkömmlichen Methoden der Datenverarbeitung nicht mehr gespeichert bzw. analysiert werden können. Dabei geht es um Datenbestände im Tera-, Peta-, Exa- und Zettabyte-Bereich.

2.3.2 Velocity

Der Aspekt *Velocity* definiert die Geschwindigkeit, mit der Daten generiert, ausgewertet und weiterverarbeitet werden können. Datenströme müssen rund um die Uhr quer über den Erdball in Echtzeit integriert bzw. analysiert werden können, um auch wichtigen Geschäfts- und Entscheidungsprozessen gerecht zu werden.

2.3.3 Variety

Der Aspekt *Variety* steht für die Vielfalt an unterschiedlichen Datentypen und -strukturen. Grundsätzlich wird zwischen strukturierten, halbstrukturierten und unstrukturierten Daten unterschieden. Strukturierte Daten wie beispielsweise Kundenstammdaten werden großteils in relationalen Datenbanken verarbeitet. Unstrukturierte Daten sind z. B. Bilder, Videos oder Audiodateien. Expertenschätzungen zufolge sind heute nur 15% der Daten strukturiert und ca. 85% der Daten unstrukturiert.

2.4 NoSQL-Datenbanken

Big Data stellt eine gute Überleitung zum Thema NoSQL dar. Zu Anwendungen im Web 2.0-Zeitalter gehören ungeheure Datenmengen, welche mit dem relationalen Datenmodell nicht mehr zu bewältigen sind. NoSQL (englisch für Not only SQL) hat es sich zum Kernziel gemacht, sogenannte Web-Scale-Datenbanken zu entwickeln, um genau mit derart großen Datenmengen effizient und performant umgehen zu können. Unter dem Schlagwort NoSQL wird eine neue Generation von Datenbanksystemen verstanden, mit dem Ziel einige der nachfolgenden Punkte zu berücksichtigen [3, S. 2]:

1. Das zugrundeliegende Datenmodell ist nicht relational.
2. Die Systeme sind von Anbeginn an auf eine verteilte und horizontale Skalierbarkeit ausgerichtet.
3. Das NoSQL-System ist Open Source.
4. Das System ist schemafrei oder hat nur schwächere Schemarestriktionen.
5. Aufgrund der verteilten Architektur unterstützt das System eine einfache Datenreplikation.
6. Das System bietet eine einfache API.
7. Dem System liegt meistens auch ein anderes Konsistenzmodell zugrunde.

Die NoSQL-Welt hat in den letzten Jahren einen regelrechten Boom erlebt. Sogar große Firmen wie *Oracle* oder *IBM* – klassische Hersteller von relationalen Datenbanksystemen – haben bestimmte Produkte in die Welt der NoSQL-Datenbanken einfließen lassen. Grundsätzlich können NoSQL-Datenbanken in zwei Hauptgruppen unterteilt werden: die Gruppe der Core-NoSQL-Systeme und die Gruppe der Soft-NoSQL-Systeme. Eine offizielle Liste aller NoSQL-Datenbanken[1] umfasst derzeit bereits über 225 Einträge aus diesen beiden Gruppen. In den nachfolgenden Abschnitten wird detailliert auf die Untergruppen der Core-NoSQL-Systeme eingegangen.

[1] http://nosql-database.org/

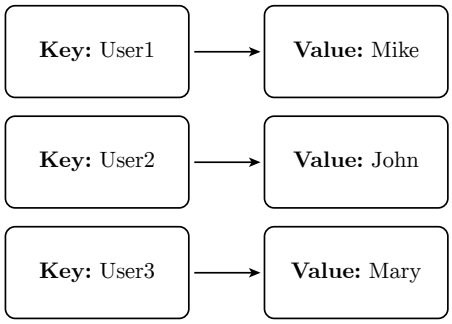


Abbildung 2.4: Key/Value Store [25]

2.4.1 Key/Value Stores

Die einfachste Gruppe der NoSQL-Datenbanken ist die Gruppe der Key/Value Stores. Dabei hat jeder Eintrag in der Datenbank einen Schlüssel (Key) und einen dazugehörigen Wert (Value). Diese Vorgehensweise ist ähnlich wie die Zuweisung eines Wertes zu einer Variable in der klassischen Programmierung. Abbildung 2.4 zeigt dazu drei einfache Beispiele. Vorteile von Key/Value Stores sind die hohe Skalierbarkeit für große Datenmengen über das effiziente Sharding sowie die Flexibilität im Datenschema. Es ist nicht notwendig Tabellen mit Spalten und Datentypen festzulegen. Die bekanntesten Vertreter von Key/Value Stores sind *Riak*, *Voldemort* und *Redis*.

Key/Value Stores haben folgende Eigenschaften [6, S. 224]:

- Es gibt eine Menge von identifizierenden Datenobjekten, die Schlüssel.
- Zu jedem Schlüssel gibt es genau ein assoziiertes deskriptives Datenobjekt, welches den Wert zum zugehörigen Schlüssel darstellt.
- Mit der Angabe des Schlüssels kann der zugehörige Wert aus der Datenbank abgefragt werden.

2.4.2 Column Family Stores

Column Family Stores können als Erweiterung von Key/Value Stores mit etwas mehr Struktur gesehen werden. Es hat sich aus den relationalen Tabellen gezeigt, dass es bei Aggregatfunktionen effizienter ist, die Daten spaltenweise und nicht zeilenweise zu speichern. Es werden nämlich selten alle Spalten einer Zeile benötigt, jedoch werden häufig Gruppen von Spalten zusammen gelesen. Daher werden in Column Family Stores die Daten in solchen Gruppen von Spalten gespeichert. Ein einfaches Beispiel dazu ist in Abbildung 2.5 dargestellt. Die wichtigsten Kandidaten dieser Gruppe sind *Cassandra* und *HBase*.

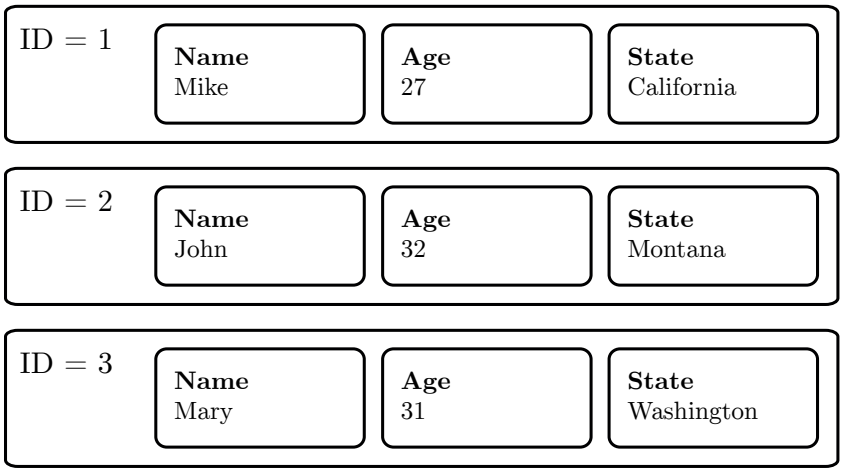


Abbildung 2.5: Column Family Store [25]

Column Family Stores haben folgende Eigenschaften [6, S. 227]:

- Daten werden in mehrdimensionalen Tabellen gespeichert.
- Datenobjekte werden mit Zeilenschlüsseln adressiert.
- Objekteigenschaften werden mit Spaltenschlüsseln adressiert.
- Spalten der Tabelle werden zu Spaltenfamilien zusammengefasst.
- Das Schema einer Tabelle bezieht sich ausschließlich auf Spaltenfamilien; innerhalb einer Spaltenfamilie können beliebige Spaltenschlüssel verwendet werden.
- Bei verteilten, fragmentierten Architekturen werden Daten zu einer Spaltenfamilie physisch möglichst am gleichen Ort gespeichert (Ko-Lokation), um die Antwortzeiten zu optimieren.

2.4.3 Document Stores

Eine weitere Gruppe von NoSQL-Datenbanken sind die Document Stores. Mit Dokumenten sind jedoch keine echten Anwenderdokumente wie Bilder, Videos oder Audiodateien gemeint, sondern strukturierte Daten in Datensätzen, welche Dokumente genannt werden. Abbildung 2.6 zeigt, wie ein einfacher Document Store aussehen kann. Document Stores sind völlig schemafrei, das bedeutet, dass jedes Dokument eine unterschiedliche Struktur haben kann. Dadurch wird eine extreme Flexibilität in der Speicherung unterschiedlicher Daten möglich. Der Big Player unter den Document Stores ist *MongoDB*.

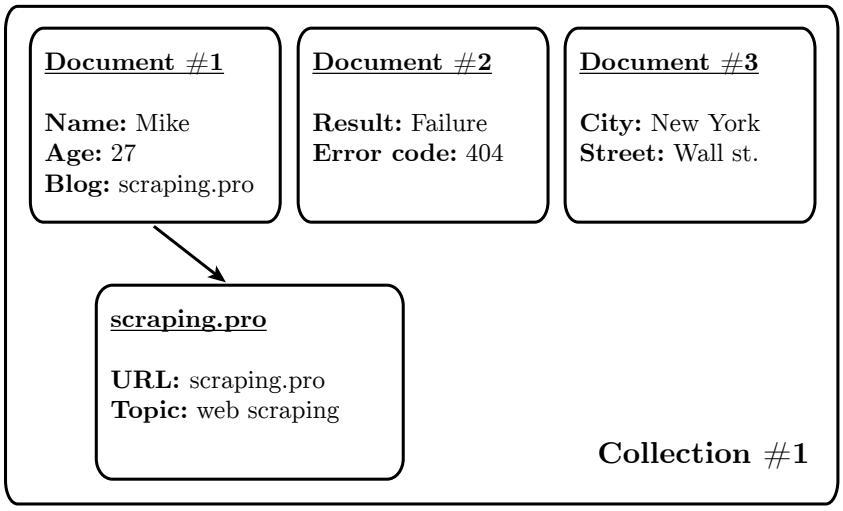


Abbildung 2.6: Document Store [25]

Document Stores haben folgende Eigenschaften [6, S. 230]:

- Sie ist eine Schlüssel-Wert Datenbank.
- Die gespeicherten Datenobjekte als Werte zu den Schlüsseln werden Dokumente genannt; die Schlüssel dienen der Identifikation.
- Die Dokumente enthalten Datenstrukturen in der Form von rekursiv verschachtelten Attribut-Wert-Paaren ohne referenzielle Integrität.
- Diese Datenstrukturen sind schemafrei, d. h. in jedem Dokument können beliebige Attribute verwendet werden, ohne diese zuerst in einem Schema zu definieren.

2.4.4 Graph Stores

Graph Stores unterscheiden sich sehr stark von den bisher besprochenen NoSQL-Datenbanken. Während die anderen Modelle komplett auf ein Schema verzichten, haben Graph Stores sehr wohl ein strukturierendes Schema. Die Daten werden in Form von Knoten, Eigenschaften und Kanten gespeichert. Die Knoten stehen für die eigentlichen Daten, die Eigenschaften repräsentieren die Merkmale der einzelnen Knoten und die Kanten zeigen die Beziehungen der Knoten zueinander. Abbildung 2.7 zeigt dazu ein einfaches Beispiel. Zu erwähnen sind im Bereich Graph Stores vor allem *Neo4J* und *HyperGraphDB*.

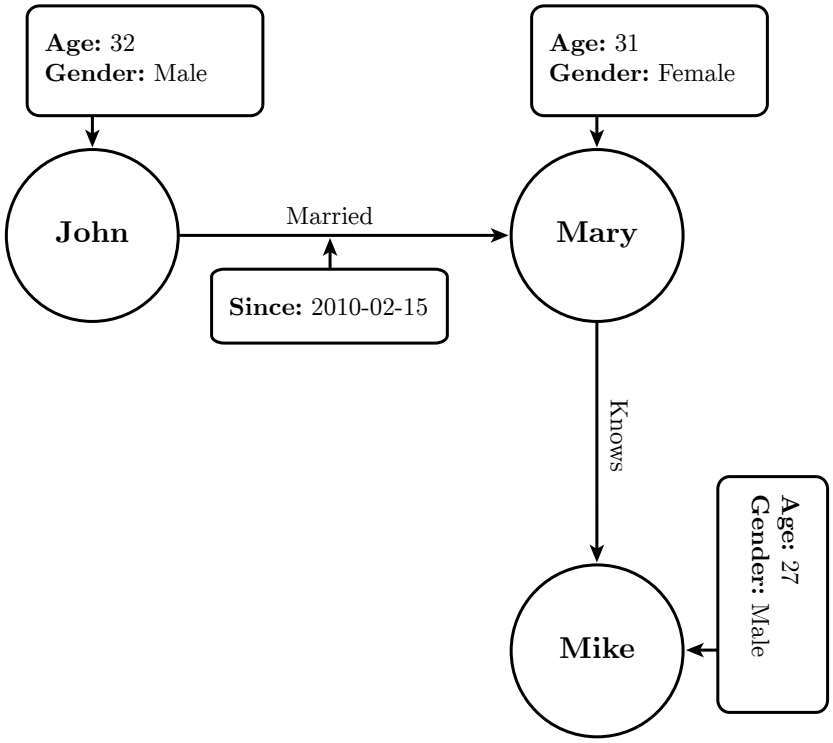


Abbildung 2.7: Graph Store [25]

Graph Stores haben folgende Eigenschaften [6, S. 238]:

- Die Daten und/oder das Schema werden als Graphen oder graphähnlichen Strukturen abgebildet, welche das Konzept von Graphen generalisieren (z. B. Hypergraphen).
- Datenmanipulationen werden als Graph-Transformationen ausgedrückt, oder als Operationen, welche direkt typische Eigenschaften von Graphen ansprechen (z. B. Pfade, Nachbarschaften, Subgraphen, Zusammenhänge, etc.).
- Die Datenbank unterstützt die Prüfung von Integritätsbedingungen, welche die Datenkonsistenz sicherstellt. Die Definition von Konsistenz bezieht sich direkt auf Graph-Strukturen (z. B. Knoten- und Kantentypen, Attribut-Wertebereiche und referenzielle Integrität der Kanten).

Kapitel 3

Stand der Technik

Dieses Kapitel beschäftigt sich mit dem aktuellen Stand der Technik flexibler Schemata. Dabei wird zu Beginn allgemein auf *MariaDB* bzw. *MongoDB* eingangen. Weiters werden grundlegende Konzepte wie Replikation oder Sharding erklärt. Anschließend wird genau erläutert, wie beide Systeme die Verwendung flexibler Schemata ermöglichen – nämlich mit Dynamic Columns für *MariaDB* bzw. *Mongoose* für *MongoDB*.

3.1 MariaDB

MariaDB[1] gehört zu den relationalen Datenbankmanagementsystemen und wird seit 2009 entwickelt. Das Projekt ist Open-Source und wurde durch eine Abspaltung aus *MySQL*[2] ins Leben gerufen. *MariaDB* ist das am schnellsten wachsende RDBMS mit mittlerweile bereits mehr als 12 Millionen Benutzern weltweit. *MariaDB* hat beispielsweise *MySQL* in der beliebten PHP-Entwicklungsumgebung *XAMPP*[3] abgelöst. Der etwas ungewöhnliche Name kommt übrigens von der Tochter des Entwicklers Michael Widenius, die Maria heißt. Mehr dazu in [1, 19]. Teile der nachfolgenden Einführung in *MariaDB* stammen aus dem offiziellen *MariaDB*-Leitfaden [12].

[1] https://mariadb.com/
[2] https://www.mysql.com/
[3] https://www.apachefriends.org/de/index.html

Der erste Schritt, um mit *MariaDB* arbeiten zu können, ist die entsprechende Verbindung mit dem Server herzustellen. Dazu muss sich der Benutzer über folgende Anweisung mit der *MariaDB* verbinden:

```
mysql -u username -p -h host databasename
```

Dabei müssen die Parameter `username`, `host` und `databasename` entsprechend durch den Benutzernamen, den Host und den Namen der Datenbank ersetzt werden. Der Login zu einer lokal laufenden *MariaDB* mit dem Namen `test` könnte dann wie folgt aussehen:

```
mysql -u thomas -p -h localhost test
```

Nach dem erfolgreichen Aufbau der Verbindung befindet sich der Benutzer in der sogenannten *MariaDB* `prompt`. Darin ist es mögich, über sämtliche SQL Statements mit der *MariaDB* zu interagieren. Um nun z. B. eine Auflistung sämtlicher Tabellen zu erhalten, ist folgende Anweisung über die `prompt` erforderlich:

```
MariaDB [test]> show tables;
+----------------+
| Tables_in_test |
+----------------+
| authors        |
| books          |
| series         |
+----------------+
3 rows in set (0.00 sec)
```

In diesem Fall enthält die verbundene *MariaDB* mit dem Namen `test` insgesamt drei Tabellen (`authors`, `books` und `series`). Unter der Rückmeldung wird außerdem noch angezeigt, wie lange die Anfrage gedauert hat. Bei solch kleinen SQL Statements geschieht die Rückmeldung in wenigen Millisekunden. Nun können weitere Informationen zu den Tabellen abgefragt werden. Die erste Möglichkeit ist die Abfrage des Grundgerüstes einer Tabelle mit den einzelnen Spaltennamen und -typen:

```
MariaDB [test]> describe books;
+----------+--------------+------+-----+---------+----------------+
| Field    | Type         | Null | Key | Default | Extra          |
+----------+--------------+------+-----+---------+----------------+
| BookID   | int(11)      | NO   | PRI | NULL    | auto_increment |
| Title    | varchar(100) | NO   |     | NULL    |                |
| SeriesID | int(11)      | YES  |     | NULL    |                |
| AuthorID | int(11)      | YES  |     | NULL    |                |
+----------+--------------+------+-----+---------+----------------+
4 rows in set (0.00 sec)
```

Die zweite Möglichkeit ist die Abfrage der eigentlichen Daten einer Tabelle mit einem einfachen `SELECT` Statement:

```
MariaDB [test]> select * from books;
+--------+----------------------------+----------+----------+
| BookID | Title                      | SeriesID | AuthorID |
+--------+----------------------------+----------+----------+
|      1 | The Fellowship of the Ring |        1 |        1 |
|      2 | The Two Towers             |        1 |        1 |
|      3 | The Return of the King     |        1 |        1 |
|      4 | The Sum of All Men         |        2 |        2 |
|      5 | Brotherhood of the Wolf    |        2 |        2 |
|      6 | Wizardborn                 |        2 |        2 |
|      7 | The Hobbit                 |        0 |        1 |
+--------+----------------------------+----------+----------+
7 rows in set (0.00 sec)
```

3.1.1 Storage Engines

Ein großer Vorteil von *MariaDB* ist die große Palette an *Storage Engines*[4]. Dabei kann für jedes Szenario die optimale Storage Engine ausgewählt und dadurch das Datenbanksystem optimiert werden. Beispiele dafür sind *InnoDB* für Transaktionen, *Spider* mit integrierten Sharding-Funktionen oder *ColumnStore* für parallele Architekturen mit einem spaltenbasierten Modell. *MariaDB* übertrifft dadurch *MySQL* bei Weitem. Im Zuge dieser Arbeit wird auf drei wichtige Storage Engines für *MariaDB* kurz eingegangen.

XtraDB

XtraDB[5] ist die Standard Storage Engine von *MariaDB* bis Version 10.1 und in den meisten Fällen auch die beste Wahl. *XtraDB* ist als Weiterentwicklung von *InnoDB* besser auf moderne Hardware abgestimmt und verfügt über einige Funktionen, die für Hochleistungsdatenbanksysteme nützlich sind.

InnoDB

InnoDB ist eine gute Wahl als Storage Engine, wenn es um Transaktionen geht. *InnoDB* ist die Standard Storage Engine in *MySQL* und auch in *MariaDB* Version 10.2. Meist wird jedoch weiterhin *XtraDB* als Storage Engine bevorzugt.

Aria

Aria[6] wurde direkt von *MariaDB* entwickelt und ist eine moderne Verbesserung der Storage Engine *MyISAM*, die als älteste *MySQL* Storage Engine gilt. Durch besseres Caching konnte in vielen Bereichen eine höhere Performance erzielt werden.

[4] https://mariadb.com/kb/en/mariadb/choosing-the-right-storage-engine/
[5] https://mariadb.com/kb/en/mariadb/about-xtradb/
[6] https://mariadb.com/kb/en/mariadb/aria-storage-engine/

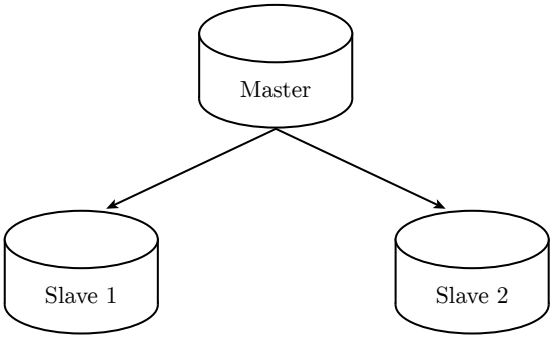


Abbildung 3.1: Replikation Setup

3.1.2 Replikation

Replikation wird unter *MariaDB* in verschiedenen Szenarien eingesetzt. Der wichtigste Bestandteil dabei ist der *Binlog* (Binary Log). Dieser schreibt jedes Event in den Log des Master Servers und anschließend in die Datenbank. Sämtliche Slave Server lesen den Log des Masters und duplizieren die Informationen in die eigene Datenbank. Ziele einer effizienten Replikation unter *MariaDB* sind:

- Skalierbarkeit,
- Datenanalyse,
- Backup,
- Datenverteilung.

Multisource Replikation

Die typische Vorgehensweise bei der Replikation ist die Definition eines Master Servers und mehrerer Slave Server (Abbildung 3.1). Bei der *Multisource Replikation* hingegen gibt es einen Slave Server, der mit mehreren Master Servern verbunden ist (Abbildung 3.2).

Bei einem Ausfall eines Masters würden die Schreiboperationen auf dem anderen Master Server weitergehen. Würde nun der ausgefallene Master wieder laufen, würden sämtliche Änderungen durch Replikation wieder auf diesen Master übertragen werden.

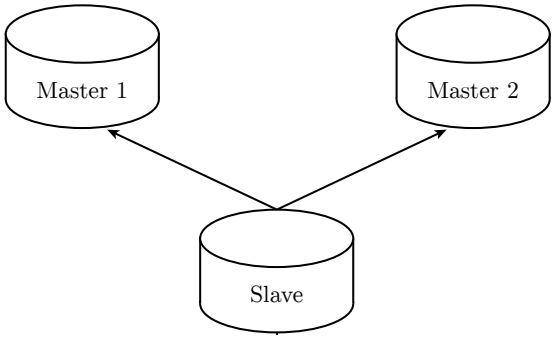


Abbildung 3.2: Multisource Replikation

3.1.3 Dynamic Columns

MariaDB verfolgt einen sehr interessanten Ansatz, um eine gewisse Flexibilität in das statische Datenschema der relationalen Datenbanken zu bringen. Mit Hilfe von Dynamic Columns ist es möglich, innerhalb einzelner Spalten beliebige und dynamisch definierte Attribute bzw. Werte zu vergeben.

Dadurch bringt *MariaDB* eine gewisse Denkweise aus der NoSQL-Welt in die starren Stukturen der relationalen Datenbanken. Dynamic Columns stellen einen wichtigen Teil dieser Arbeit dar, daher wird in diesem Kapitel sehr detailliert darauf eingegangen. Mehr dazu in [15].

Teile der nachfolgenden Einführung in Dynamic Columns wurden aus einem Tutorial des offiziellen *MariaDB*-Blogs übernommen [16, 17]. Der erste Schritt, um Dynamic Columns verwenden zu können, ist die Erstellung eines `BLOB` (Binary Large Object). Dieses binäre Datenobjekt kann eine maximale Größe von 65536 Bytes haben. Dazu wird eine einfache Tabelle erzeugt, die diverse Artikel speichert:

```
CREATE TABLE items
(id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
name varchar(100) NOT NULL, attributes BLOB);
```

Anschließend ist es möglich, verschiedene Artikel mit unterschiedlichsten Eigenschaften mit der Funktion `COLUMN_CREATE` in die Tabelle einzufügen. Lediglich der Name des Artikels muss verpflichtend als Zeichenkette angegeben werden. Innerhalb des BLOB-Objekts gibt es jedoch keine Einschränkungen:

```
INSERT INTO items (name, attributes) VALUES
('MariaDB t-shirt', COLUMN_CREATE('colour','blue', 'size','XXL')),
('MariaDB t-shirt', COLUMN_CREATE('colour','blue', 'size','XL')),
('Samsung Galaxy S5', COLUMN_CREATE('colour','white', 'OS', 'Android',
'type', 'phone')),
('Samsung Galaxy Pro 3', COLUMN_CREATE('colour','white', 'size',8,
'OS', 'Android', 'resolution','1920x1200', 'type','tablet'));
```

In diesem Beispiel werden Artikel eingefügt, die eigentlich nichts gemeinsam

haben und zu unterschiedlichsten Produktfamilien gehören. Über ein einfaches `SELCT` Statement können die Artikel aufgelistet werden. In diesem Fall wird die Ausgabe auf einen Artikel beschränkt und dessen Eigenschaften als `JSON` ausgegeben, was mit der Funktion `COLUMN_JSON` ermöglicht wird:

```
SELECT name AS Item,
COLUMN_JSON(attributes) AS 'Dynamic Columns' FROM items LIMIT 1;
+----------------+------------------------------+
| Item           | Dynamic Columns              |
+----------------+------------------------------+
| MariaDB t-shirt | {"size":"XXL","colour":"blue"} |
+----------------+------------------------------+
```

Weiters ist es möglich, alle Attributnamen der Dynamic Columns aufzulisten. Dazu stellt *MariaDB* die Funktion `COLUMN_LIST` zur Verfügung:

```
SELECT name AS Item,
COLUMN_LIST(attributes) AS 'Attribute Names' FROM items;
+---------------------+-----------------------------------------+
| Item                | Attribute Names                         |
+---------------------+-----------------------------------------+
| MariaDB t-shirt     | `size`,`colour`                         |
| MariaDB t-shirt     | `size`,`colour`                         |
| Samsung Galaxy S5   | `OS`,`type`,`colour`                    |
| Samsung Galaxy Pro 3 | `OS`,`size`,`type`,`colour`,`resolution` |
+---------------------+-----------------------------------------+
```

Der wichtigste Teil der Datenbank sind jedoch die tatsächlichen Werte, die in den Dynamic Columns stehen. Diese können mit der Funktion `COLUMN_GET` abgefragt werden. In diesem Beispiel werden alle Artikel mit deren Farbe selektiert:

```
SELECT name AS Item,
COLUMN_GET(attributes, 'colour' AS CHAR) AS Colour FROM items;
+---------------------+--------+
| Item                | Colour |
+---------------------+--------+
| MariaDB t-shirt     | blue   |
| MariaDB t-shirt     | blue   |
| Samsung Galaxy S5   | white  |
| Samsung Galaxy Pro 3 | white  |
+---------------------+--------+
```

Eine weitere wichtige Funktion ist Funktion `COLUMN_EXISTS`. Mit dieser ist es möglich, nur die Namen jener Artikel abzufragen, die ein bestimmtes Attribut in den Dynamic Columns besitzen:

```
SELECT name FROM items
WHERE COLUMN_EXISTS(attributes, 'OS');
+--------------------------+
| name                     |
+--------------------------+
| Samsung Galaxy S5        |
| Samsung Galaxy Pro 3     |
+--------------------------+
```

3.2 MongoDB

MongoDB[7] gehört zur Gruppe der NoSQL-Datenbanken und ist darin der dominierende Vertreter unter den dokumentenorientierten Systemen. Der Name *MongoDB* kommt vom englischen Wort „humongous“, was übersetzt für „gigantisch“ bzw. „riesig“ steht. *MongoDB* wird seit 2007 entwickelt, ist schemafrei und Open-Source. Die Daten werden in Collections und Dokumente im BSON-Format[8] gespeichert, welches große Ähnlichkeiten mit dem JSON-Format[9] hat. Mehr dazu in [8, 11].

3.2.1 Datenbanken

Nach der Installation kann der *MongoDB*-Server mit der Anweisung `mongod` in der Konsole gestartet werden. Grundsätzlich kann ein laufender *MongoDB*-Server mehrere logische Datenbanken verwalten. Um die vorhandenen Datenbanken abzufragen, muss sich der Benutzer mit der Anweisung `mongo` mit der Mongo Shell verbinden und folgende Anweisung ausführen:

```
> show dbs
local 0.078125GB
```

In einem frisch installierten *MongoDB*-Server gibt es zu Beginn nur eine Datenbank mit dem Namen `local`, welche stets existiert und auch nicht gelöscht werden kann. Um eine neue Datenbank mit dem Namen `test` zu erzeugen und anschließend zu verwenden, ist folgende Anweisung notwendig:

```
> use test
switched to db test
```

3.2.2 Collections

Eine *Collection* ist in *MongoDB* eine Sammlung von Dokumenten innerhalb einer Datenbank und kann mit einer Tabelle in relationalen Datenbanken verglichen werden. Durch die Schema-Flexibilität gibt es keine strukturellen Anforderungen an die Dokumente innerhalb der Collection. Um die Collections der aktuell ausgewählten Datenbank in der Mongo Shell abzufragen, ist folgende Anweisung in der Konsole notwendig:

```
> show collections
system.indexes
```

Weiters kann mit folgender Anweisung eine neue Collection über die Mongo Shell in die verbundene Datenbank eingefügt werden:

```
> db.createCollection("tweets")
{ "ok": 1 }
```

[7] https://www.mongodb.com/
[8] http://bsonspec.org/
[9] http://json.org/

System Collections

Die zuvor erstellte Datenbank `test` ist zu Beginn noch leer und enthält lediglich die Collection mit dem Namen `system.indexes`. Dabei handelt es sich um eine *System Collection*. System Collections haben den reservierten Namensraumpräfix `system.` und besondere Aufgaben. Die genannte Collection `system.indexes` nimmt z. B. alle Indizes der Datenbank auf und entsteht beim Anlegen der ersten Collection. Weiters gibt es System Collections für Namespaces, Profiling, Authentifizierung oder benutzerdefinierte Rollen.

Capped Collections

Bei *Capped Collections* handelt es sich um eine besondere Art von Collections. Diese Collections sind in der Größe in Bytes oder Anzahl an Dokumenten begrenzt, wobei ältere Dokumente nach dem FIFO-Prinzip gelöscht werden, wenn beim Einfügen eine der beiden genannten Grenzen überschritten wird. Um eine Capped Collection anzulegen, müssen beim Erzeugen die Parameter `capped`, `size` und `max` mitgegeben werden:

```
> db.createCollection("capping", {capped: true, size: 1048576, max: 10})
{ "ok": 1 }
```

3.2.3 Dokumente

Ein einzelner Datensatz wird in *MongoDB* als *Dokument* bezeichnet. Diese Dokumente haben eine Größenbeschränkung von 16 MB und bestehen formal aus einer Menge von Feldern, wobei jedes Feld ein Key-Value-Paar darstellt. Die Dokumente werden im BSON-Format abgespeichert, was folgende Vorteile mit sich bringt:

- Bessere Traversierbarkeit,
- kompaktere Speicherung,
- die Bereitstellung weiterer Datentypen.

Jedes Dokument hat zur eindeutigen Identifizierung ein Feld mit dem Schlüssel `_id` und dem Wert vom Typ `ObjectId`. Dieses Feld wird beim Erstellen eines jeden Dokuments automatisch generiert und gilt anschließend als Primärschlüssel.

Subdokumente

Weiters ist es möglich, dass ein Feld als Wert wiederum ein Dokument ist. Dann wird von einem Subdokument oder einem eingebetteten Dokument gesprochen. Die maximale Schachtelungstiefe von Subdokumenten ist auf 100 begrenzt.

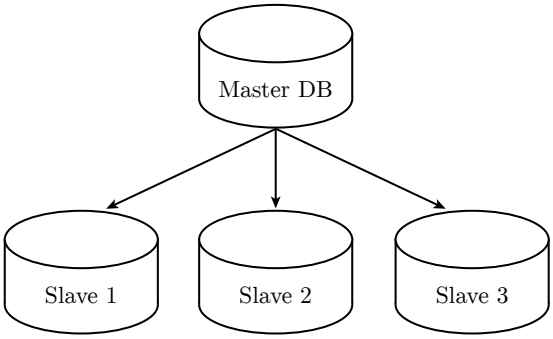


Abbildung 3.3: Master-Slave-Replikation

GridFS

Mit Hilfe von *GridFS* können Daten gespeichert werden, die die Maximalgröße von 16 MB überschreiten. Dabei werden diese größeren Daten in mehrere Dokumente zerlegt und in der *MongoDB* abgelegt. Dazu werden zwei Collections mit den Namen `fs.files` und `fs.chunks` erzeugt. In der Collection `fs.files` sind die Metadaten wie beispielsweise der Dateiname gespeichert. Die Collection `fs.chunks` enthält die eigentlichen Teile (Dokumente) der Datei, die sogenannten Chunks. Jeder Chunk besitzt zusätzlich ein Referenzfeld mit dem Namen `files_id`, damit jeder Chunk auch eindeutig zu dessen Metadaten zugeordnet werden kann.

3.2.4 Replikation

MongoDB setzt die *Replikation* zur Skalierung von Lesezugriffen und Sicherstellung der Ausfallsicherheit ein. Es wird dabei zwischen einer Master-Slave-Replikation und dem Replica Set unterschieden. Grundlage ist das sogenannte *Oplog* (Operations Log). In dieser Capped Collection werden alle schreibenden Operationen festgehalten.

Master-Slave-Replikation

Die *Master-Slave-Replikation* ist mittlerweile veraltet, wird aber hier trotzdem zum weiteren Verständnis kurz erklärt. Dabei nehmen einzelne Server-Instanzen feste Rollen ein (Abbildung 3.3). Ein Server bekommt die Rolle des Masters zugewiesen und ist für alle schreibenden Zugriffe (Inserts, Updates und Removes) zuständig. Der Master repliziert sämtliche Operationen an alle Slaves. Großer Nachteil der Master-Slave-Replikation ist die Tatsache, dass bei einem Ausfall des Master-Servers das gesamte System nur noch eingeschränkt oder gar nicht mehr verwendet werden kann. Daher sollte diese Replikation bei produktiven Systemen nicht mehr eingesetzt werden.

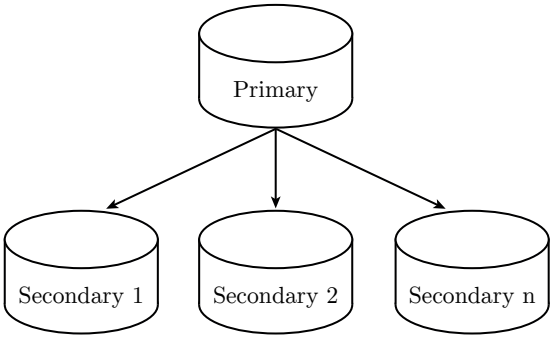


Abbildung 3.4: Replica Set

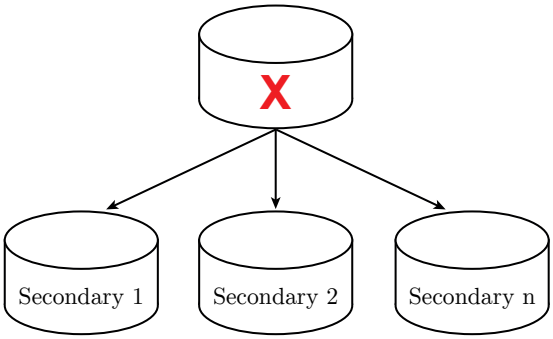


Abbildung 3.5: Ausfall des Primaries

Replica Set

Das *Replica Set* baut auf dem Master-Slave-Konzept auf, bietet jedoch eine wesentlich höhere Ausfallsicherheit. Grund dafür ist die flexiblere Verteilung der einzelnen Rollen. Grundsätzlich gibt es wieder einen Master-Knoten, der in diesem Fall *Primary* genannt wird und mehrere Slave-Knoten, die hier als *Secondaries* bezeichnet werden (Abbildung 3.4).

Auch beim Replica Set führt der Primary die schreibenden Operationen aus und repliziert diese auf die Secondaries. Kommt es nun aber zum Ausfall des Primaries (Abbildung 3.5), entscheiden sich alle Secondaries für einen Secondary als neuen Primary (Abbildung 3.6). Wird der ausgefallene Server wieder aktiv, reiht sich dieser als Secondary ein und wird vom neuen Primary repliziert (Abbildung 3.7).

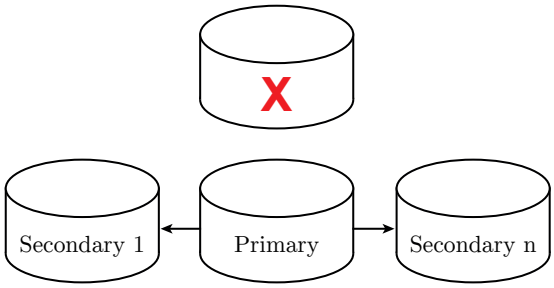


Abbildung 3.6: Festlegen eines neuen Primaries

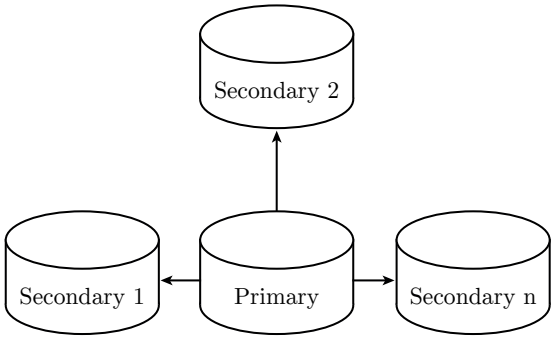


Abbildung 3.7: Einreihung des alten Primaries als Secondary

3.2.5 Sharding

Unter *Sharding* wird die Verteilung von Daten auf mehrere Netzwerkknoten verstanden. *MongoDB* ermöglicht damit die Möglichkeit zur horizontalen Skalierung von Schreib- und Lesezugriffen wodurch Performancesteigerungen erreicht werden können. Sharding wird auf Ebene von Collections gesteuert.

Der wichtigste Bestandteil dabei ist der *Shard Key*, welcher in jedem Dokument vorkommen muss. Grundsätzlich werden beim Sharding die Collections in kleinere Einheiten (Chunks) aufgeteilt. Genau diese Chunks orientieren sich am Shard Key. Somit wird jedes Dokument genau einmal persistiert. Abbildung 3.8 zeigt ein *MongoDB*-Setup bestehend aus drei Shards.

Die einzelnen Shards sind normale `mongod`-Prozesse, worauf die Dokumente gespeichert werden. Die Metadaten für das Sharding werden auf den Config Servern gespeichert. Auch hier handelt es sich um `mongod`-Prozesse. Der wesentliche Teil im Sharding-Prozess ist der Sharding Server. Auf diesem Server läuft ein `mongos`-Prozess, welcher mit den Config Servern kommuniziert und von diesen die Informationen erhält, wo welche Dokumente persistiert sind.

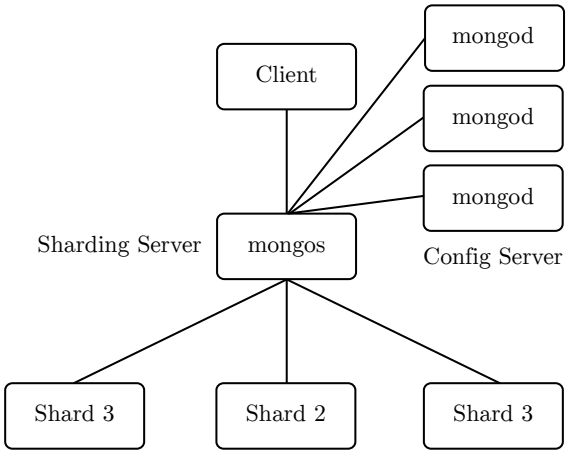


Abbildung 3.8: Sharding Setup

3.2.6 Mongoose

MongoDB ist grundsätzlich eine schemafreie Datenbank. Es gibt jedoch einen interessanten Ansatz, der den Daten der *MongoDB* eine gewisse Struktur gibt und für diese Arbeit einen wichtigen Teil darstellt. Die Rede ist dabei von *Mongoose*. *Mongoose* ist ein Modul für *Node.js*[10] mit dem Ziel, elegante Objektmodellierungen für *MongoDB* zu ermöglichen. Die Entwickler finden dazu auf der offiziellen Website folgende Definition [20]:

> *Mongoose* provides a straight-forward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box.

Schemas

Das *Mongoose Schema* definiert die Form der Dokumente innerhalb einer Collection. Dabei werden für alle Felder entsprechende Typen und Eigenschaften wie Standardwerte oder Pflichtfelder festgelegt. Folgende Typen sind dabei zulässig: `String`, `Number`, `Date`, `Buffer`, `Boolean`, `Mixed`, `ObjectId` und `Array`. Wie nun ein einfaches Schema aussehen kann, ist in Programm 3.1 dargestellt [21].

Dabei handelt es sich um ein Schema für Einträge eines Blogs. Jeder Blog-Eintrag besitzt einen Titel, den Autor und den Text. Diese Felder werden jeweils als `String` gespeichert. Weiters hat jeder Eintrag ein `Array` mit Kommentaren, die jeweils wiederum einen Text und ein Datum besitzen. Außerdem wird das Erstelldatum des Blog-Eintrags selbst als `Date` persistiert,

[10] https://nodejs.org/en/

Programm 3.1: Beispiel Schema

```
var blogSchema = new mongoose.Schema({
    title: String,
    author: String,
    body: String,
    comments: [{
        body: String,
        date: Date
    }],
    date: {
        type: Date,
        default: Date.now
    },
    hidden: Boolean,
    meta: {
        votes: Number,
        favs: Number
    }
});
```

wobei standardmäßig das aktuelle Datum herangezogen wird. Weitere Felder speichern als `Boolean` die Sichtbarkeit des Eintrags sowie als Metadaten die Anzahl an Votes und Favoriten als `Number`.

Models

Um das Schema anschließend verwenden zu können, muss daraus ein *Mongoose Model* erstellt werden. Die folgende Code-Zeile zeigt die Definition des Models aus dem zuvor erstellten Blog-Schema:

```
var Blog = mongoose.model('Blog', blogSchema);
```

Der erste Parameter `Blog` definiert den späteren Namen der Collection. Dabei wird der Plural in Kleinschreibung übernommen. Daher wird die Collection den Namen `blogs` tragen. Der zweite Parameter enthält das eigentliche Schema, aus dem das Model erstellt wird.

Kapitel 4

Implementierung

Im folgenden Kapitel geht es um die für diese Arbeit relevante Implementierung eines *MongoDB* Administrations Tool basierend auf *Mongoose*. Dabei wird genauer auf technische Anforderungen, verwendete Technologien sowie Meilensteine während der Umsetzung eingegangen. Außerdem wird kurz erklärt, wie die Performance Tests implementiert wurden.

4.1 Motivation

Wie bereits im vorherigen Kapitel erwähnt, ist *MongoDB* die Nummer eins unter den dokumentenorientierten Datenbanken und wächst ständig weiter. Umso verwunderlicher ist die Tatsache, dass *MongoDB* keine offizielle graphische Benutzeroberfläche für administrative Aufgaben zur Verfügung stellt. Daher muss grundsätzlich jegliche Interaktion mit der *MongoDB* über die sogenannte Mongo Shell erfolgen, was weder benutzerfreundlich noch übersichtlich ist.

Das war der Denkanstoß, über eine Lösung für ein *MongoDB* Administrations Tool basierend auf *Mongoose* nachzudenken. Es gibt in diesem Bereich bereits mehrere Open-Source Projekte, wie *Robomongo*[1], *Mongoclient*[2] oder *MongoBooster*[3]. Einige Tools sind jedoch mittlerweile veraltet bzw. nicht webbasiert. Außerdem konnte kein einziges Tool mit *Mongoose* umgehen.

[1] https://robomongo.org/
[2] https://www.mongoclient.com/
[3] https://mongobooster.com/

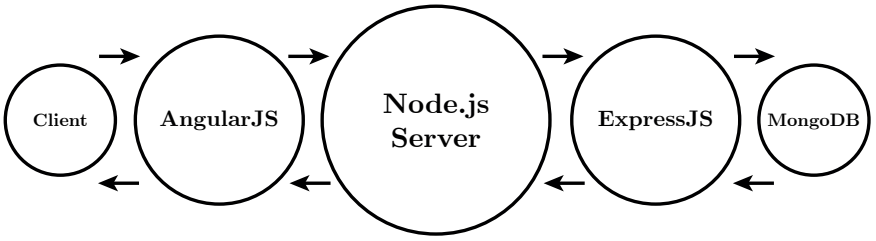


Abbildung 4.1: MEAN-Stack

4.2 Anforderungen

Vor Beginn der Umsetzung stand vor allem die Evaluierung bestehender Tools sowie die Definition von Anforderungen für das eigene Tool im Vordergrund. Manche Tools legen den Fokus auf die Administration der *MongoDB*, andere konzentrieren sich auf die Visualisierung der Daten. Das eigene Tool sollte in erster Linie die Grundfunktionen mit klarem Fokus auf *Mongoose* solide ausführen können. So entstanden folgende Zielsetzungen:

- *MongoDB* Informationen und Statistiken,
- CRUD Funktionalität für Collections und Dokumente,
- Validierung von Dokumenten,
- Suche nach Dokumenten,
- Interpretation von Schemata.

4.3 Technologien

Für die Umsetzung wurde der JavaScript MEAN-Stack verwendet. Dieser besteht grundsätzlich aus den vier Komponenten *MongoDB*, *Express*[4], *AngularJS*[5] und *Node.js*. Wie diese Komponenten bei einem typischen Request Cycle zusammenspielen, ist in Abbildung 4.1 übersichtlich dargestellt.

Der *Node.js*-Server stellt dabei das Herzstück dar. Dieser ermöglicht die Kommunikation zwischen der Datenbank und der Client-Seite. Mit Hilfe von *Express* ist es möglich, jede Anfrage einer bestimmten Route zuzuweisen und so zu den korrekten Daten zu gelangen. *AngularJS* nimmt die Daten anschließend wieder entgegen und gibt diese in entsprechend sauberer Form für den Client aus. Somit bringt der MEAN-Stack folgende Vorteile mit sich:

[4]http://expressjs.com/
[5]https://angularjs.org/

Programm 4.1: Node.js Webserver

```
1 var http = require('http');
2
3 http.createServer(function (req, res) {
4   res.writeHead(200, {
5     'Content-Type': 'text/html'
6   });
7   res.write('Hallo Welt');
8   res.end();
9 }).listen(3000);
```

- JavaScript als Sprache in der kompletten Applikation,
- Model-View-Controller Muster,
- JSON als Datenaustauschformat,
- Große Bibliothek für *Node.js* Module (*npm*).

4.3.1 Node.js

Node.js ermöglicht die Nutzung von JavaScript auf der Server-Seite und baut auf der V8 Engine von *Google* auf, die auch im hauseigenen Browser *Chrome* verwendet wird. Besonders hervorzuheben ist die große Anzahl an verfügbaren Modulen für *Node.js*, die über den *Node Package Manager* (*npm*) eingebunden werden können. *Node.js* findet auf der offiziellen Website folgende Definition [22]:

> *Node.js* is a JavaScript runtime built on *Chrome's* V8 JavaScript engine. *Node.js* uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. *Node.js'* package ecosystem, *npm*, is the largest ecosystem of open source libraries in the world.

Mit *Node.js* ist es sehr schnell möglich, einen einfachen Webserver einzurichten. Programm 4.1 zeigt eine einfache Konfiguration eines Webservers, der mit Hilfe des `http`-Moduls am Port 3000 lauscht und den Text `Hallo Welt` ausgibt.

Express ist als Web-Framework für *Node.js* ein wichtiger Bestandteil des MEAN-Stacks. *Express* ist sehr flexibel und vereinfacht durch viele HTTP-Dienstprogrammmethoden sowie Middlewarefunktionen die Erstellung einer leistungsfähigen API. Das vorherige Beispiel des reinen *Node.js*-Webservers ist natürlich auch mit *Express* möglich (Programm 4.2). Nach der Einbindung des `express`-Moduls wird eine neue `express`-App erzeugt. Unter der Stamm-URL (/) am Port 3000 wird der Text `Hallo Welt` ausgegeben.

Programm 4.2: Express Webserver

```
1 var express = require('express');
2 var app = express();
3
4 app.get('/', function (req, res) {
5   res.send('Hallo Welt');
6 });
7
8 app.listen(3000, function () {
9   console.log('App listening on port 3000!');
10 });
```

4.3.2 AngularJS

AngularJS ist ein JavaScript-Framework, das HTML durch neue Attribute erweitert und perfekt für Single Page Applications (SPAs) geeignet ist. *AngularJS* basiert auf dem Model-View-ViewModel Muster. Dadurch wird die Logik sauber von den eigentlichen Views getrennt. Hilfreich dabei sind in *AngularJS* die sogenannten *Scopes*, welche eine Menge von Variablen und Funktionen für einen bestimmten Kontext enthalten. Mehr dazu in [10].

Zwei-Wege-Datenbindung

Die *Zwei-Wege-Datenbindung* ist ein entscheidendes Konzept von *AngularJS*. Durch diesen Mechanismus ist es möglich, dass bei einer Änderung des Models automatisch die View aktualisiert wird. Auch der umgekehrte Weg, also die automatische Aktualisierung des Models bei einer Änderung der View durch den Benutzer ist gewährleistet. Somit muss nicht auf Änderungen gewartet und reagiert werden, sondern diese Änderungen übernimmt *AngularJS* automatisch.

Controller

Im *Controller* befinden sich die Daten und die Logik für eine View. Jeder Controller besitzt einen eigenen Scope, der die Variablen und Funktionen für die jeweilige View verwaltet. In der View können anschließend die Variablen für den Benutzer ausgegeben oder Funktionen aufgerufen werden.

Models

Unter *Models* werden in *AngularJS* JavaScript-Datentypen verstanden, die sich innerhalb eines Scopes im Controller befinden. Models sind sozusagen das Bindeglied für die Zwei-Wege-Datenbindung zwischen der View und dem Controller.

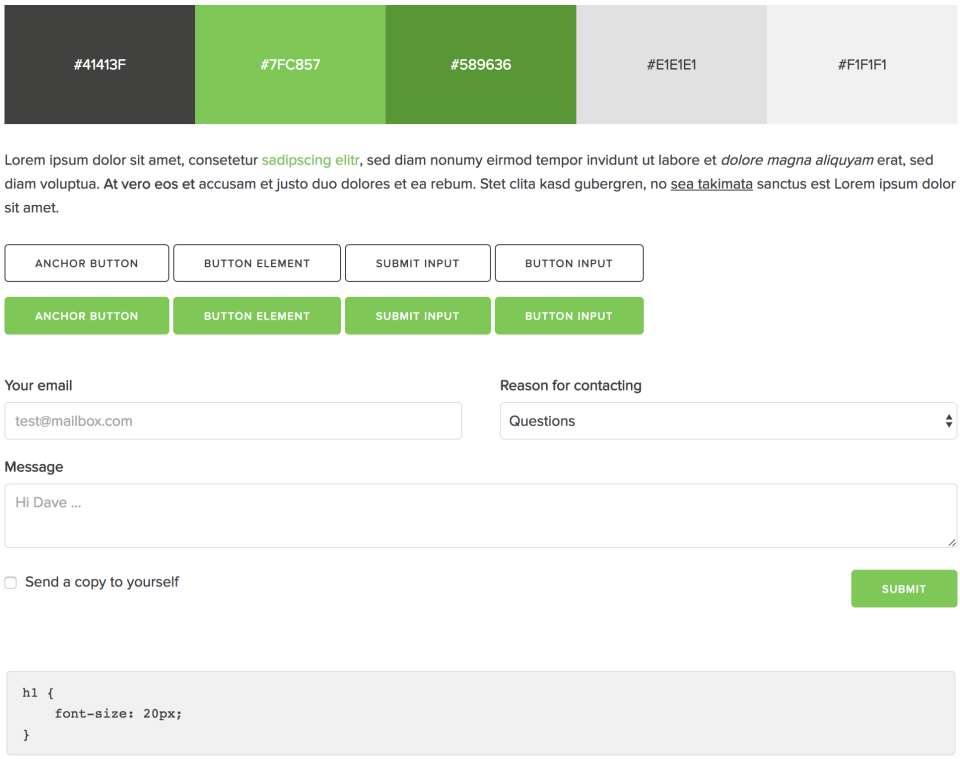


Abbildung 4.2: Styleguide

Routen

Da in Single Page Applications im Prinzip immer nur ein Teil der View ausgetauscht wird, ist die Definition der *Routen* besonders wichtig. Dadurch wird genau festgelegt, welche View bzw. welcher Controller zu welcher Route gehört. Somit wird für die entsprechende Route immer nur der korrekte Controller ausgeführt, was Zeit und Ressourcen spart.

4.3.3 Skeleton

Skeleton[6] ist ein einfacher, responsiver CSS-Boilerplate, der bereits einige hilfreiche Style-Definitionen für HTML-Basiselemente mit sich bringt. Nach Anpassung der offiziellen *MongoDB*-Farben wurde ein kleiner Styleguide (Abbildung 4.2) mit den wichtigsten Elementen erstellt. Nach diesem Schritt konnte mit der eigentlichen Implementierung begonnen werden.

[6] http://getskeleton.com/

Programm 4.3: Verzeichnisstruktur der Implementierung

```
app/
|-- node_modules/
|-- public/
|   |-- css/
|   |-- fonts/
|   |-- images/
|   |-- js/
|   |-- libs/
|   |-- views/
|   |-- index.html
|-- bower.json
|-- package.json
|-- server.js
```

4.4 Architektur

Bei der Umsetzung einer Single Page Application mit dem MEAN-Stack ist eine saubere App Struktur besonders wichtig (Abbildung 4.3). Ein guter Startpunkt war dabei das Projekt *starter-node-angular*[7]. Für die Einbindung sämtlicher Module bzw. Frameworks wurden Package Manager verwendet. Serverseitig wurde dabei *npm*[8] und clientseitig *bower*[9] eingesetzt.

Der Ordner *node_modules* enthält alle Module, die über *npm* installiert werden. Dieser Ordner wird automatisch erstellt sobald das erste Modul installiert wird. Welche Module über das Kommando `npm install` hinzugefügt werden, ist in der Datei *package.json* gespeichert. Dort wird neben sämtlichen Abhängigkeiten auch jene Datei angegeben, die beim Ausführen der App über das Kommando `npm start` gestartet wird.

Im Ordner *public* sind alle Daten für die Client-Seite abgelegt. Dazu gehören CSS-Dateien, Schriften, Bilder sowie JavaScript Dateien. Im Unterordner *js* sind die Controller und Routen über *AngularJS* definiert, welche über die Ansichten im Unterordner *Views* eingebunden werden. Die *bower*-Module werden im Unterordner *libs* abgelegt und anschließend im Grundtemplate, der Datei *index.html* in die App eingebunden. Welche Module über *bower* eingebunden werden, ist in der Datei *bower.json* definiert.

Der Code für die Server-Seite befindet sich in der Datei *server.js*. Diese Datei ist somit auch die auszuführende Datei beim Start der App. Darin wird die *Express*-App erzeugt, die Verbindung zur *MongoDB* aufgebaut und die API zur Verfügung gestellt.

[7] https://github.com/scotch-io/starter-node-angular
[8] https://www.npmjs.com/
[9] https://bower.io/

4.5 Meilensteine der Umsetzung

Zu Beginn der Umsetzungsphase war der Fokus noch nicht konkret auf *Mongoose* gerichtet, sondern eher auf die Validierung der Dokumente. Der Benutzer sollte somit nicht durch ein Schema eingeschränkt werden und alle möglichen Daten in der *MongoDB* verwalten können.

Daher wurde am Anfang mit dem offiziellen *MongoDB*-Treiber[10] für *Node.js* begonnen. Dabei ist es möglich, mit dem *MongoClient*[11] eine Verbindung zur *MongoDB* aufzubauen. Nach dem Verbindungsaufbau kann mit einer *Db*[12] Instanz weitergearbeitet werden. Damit war es möglich folgende Funktionen zu implementieren:

```
GET database statistics
GET list of collections
GET documents of collection
POST collection
DELETE collection
```

Es war also bereits möglich, die Statistiken der verbundenen *MongoDB* über die Funktion `serverStatus` einzusehen. Weiters wurde dem Benutzer eine Liste aller Collections mit Hilfe der Funktion `listCollections` zur Verfügung gestellt und die Dokumente innerhalb einer Collection aufgelistet. Außerdem konnten neue Collections erstellt sowie bestehende Collections gelöscht werden.

Nach den ersten Implementierungen mit dem offiziellen *MongoDB*-Treiber kam eine interessante Alternative ins Spiel. *Mongojs*[13], entwickelt von Mathias Buus, ist ein Modul für *Node.js*, das die wichtigsten Funktionen der offiziellen *MongoDB*-API implementiert. Mit Hilfe von *Mongojs* konnten anschließend folgende Funktionen umgesetzt werden:

```
GET collection statistics
DELETE document of collection
```

Nun konnten auch die Statistiken zu einzelnen Collections angezeigt werden. Auch das Löschen von Dokumenten ist in *Mongojs* etwas einfacher implementiert als im offiziellen *MongoDB*-Treiber. Ab diesem Zeitpunkt ging es um das Einfügen bzw. Editieren von Dokumenten innerhalb der Collections. Es stellte sich die Frage, wie mit Collections mit vordefiniertem Schema umgegangen wird.

Dadurch wurde *Mongoose* Teil der Umsetzung und der komplette Fokus ging in Richtung Administration von *Mongoose* Collections. Diese Entscheidung hatte natürlich auch Auswirkungen auf die bisherige Implementierung.

[10] http://mongodb.github.io/node-mongodb-native/2.2/api/
[11] http://mongodb.github.io/node-mongodb-native/2.2/api/MongoClient.html
[12] http://mongodb.github.io/node-mongodb-native/2.2/api/Db.html
[13] https://github.com/mafintosh/mongojs

Das Erstellen bzw. Löschen von Collections war nicht mehr sinnvoll, weil es keinen Sinn macht, eine *Mongoose* Collection zu erzeugen, ohne ein Schema dafür zu definieren. Es entstanden jedoch neue Herausforderungen für die weitere Umsetzungsphase:

1. Sinnvolle Schachtelungstiefe für Subdokumente,
2. Generische API unabhängig vom *Mongoose* Model,
3. Form Generierung abhängig vom *Mongoose* Model.

4.5.1 Testdaten

Der nächste Schritt war die Suche nach umfangreichen Testdaten. Dazu gibt es auf der offiziellen *MongoDB*-Website einen entsprechenden Datensatz, der Informationen zu Restaurants[14] speichert. Das entsprechende *Mongoose* Schema ist in Programm 4.4 abgebildet.

Dabei wird zu jedem Restaurant der Name als `String` und die Adresse als `Object` gespeichert. Neben der Straße, der Hausnummer, der Postleitzahl und dem Ort enthält die Adresse ein Subdokument für die Koordinaten. Darin werden wiederum der Längen- und Breitengrad als `Number` angegeben. Weiters hat jedes Restaurant einen Typ als `String`, der nur Werte aus einer vordefinierten Liste annehmen darf. Außerdem können Bewertungen zu den Restaurants als `Array` gespeichert werden. Jede Bewertung hat dabei einen Namen und eine Anzahl von Sternen zwischen 1 und 5. Mit dem Zusatz `versionKey: false` wird das Feld für die Version des Dokuments nicht vergeben.

Nach der Umstellung von *Mongojs* auf *Mongoose* mussten die bestehenden Funktionen auf *Mongoose* adaptiert werden. Weiters war es wichtig, ein Modul zu finden, welches eine generische API unabhängig von den *Mongoose* Models zur Verfügung stellt. Dabei wurde auf das *Node.js*-Modul *node-express-crud-router*[15] von Dennis Ahaus zurückgegriffen. Dieses wurde dahingehend erweitert, dass das *Mongoose* Schema an das Frontend weitergeleitet wurde, um später daraus die Form für das Einfügen bzw. Editieren von Dokumenten zu ermöglichen. Somit waren folgende Funktionen verfügbar:

```
GET schema of collection
POST document to collection
UPDATE document of collection
```

Somit waren alle Funktionalitäten auf der Server-Seite implementiert und der Fokus lag nu auf der Client-Seite, nämlich bei der Generierung der Form anhand der *Mongoose* Schemata. Dabei wurde auf das interessante Modul

[14] https://docs.mongodb.com/getting-started/shell/import-data/
[15] https://github.com/DennisAhaus/node-express-crud-router

Programm 4.4: Restaurant Schema

```
var RestaurantSchema = mongoose.Schema({
    name: {
        type: String,
        required: true
    },
    address: {
        street: String,
        number: Number,
        zipcode: Number,
        city: String,
        coordinates: {
            lat: Number,
            lng: Number
        }
    },
    type: {
        type: String,
        enum: ['Pizza', 'Asian', 'Snack', 'Breakfast'],
        required: true
    },
    grades: [{
        _id: false,
        name: String,
        stars: {
            type: String,
            enum: [1, 2, 3, 4, 5]
        }
    }]
}, {
    versionKey: false
});
```

angular-schema-form[16] zurückgegriffen. Das Problem dabei war jedoch, dass dieses Modul nicht direkt mit *Mongoose* Schemata umgehen konnte, sondern das Schema im *JSON-Schema*[17] Format benötigte. Daher musste das *Mongoose* Schema vorher entsprechend in das andere Format gebracht werden, was zu Problemen mit dem *Mongoose* Typ `Array` führte.

Aus diesem Grund wurde schlussendlich das Framework *forms-angular*[18] verwendet. Dieses Framework eignet sich perfekt für die Implementierung, da es selbst auf dem MEAN-Stack aufbaut und somit auch mit *Mongoose* Models sowohl auf Server-Seite als auch auf Client-Seite umgehen kann. Das Framework wird im nächsten Kapitel noch genauer unter die Lupe genommen.

[16] https://github.com/json-schema-form/angular-schema-form
[17] http://json-schema.org/
[18] https://github.com/forms-angular/forms-angular

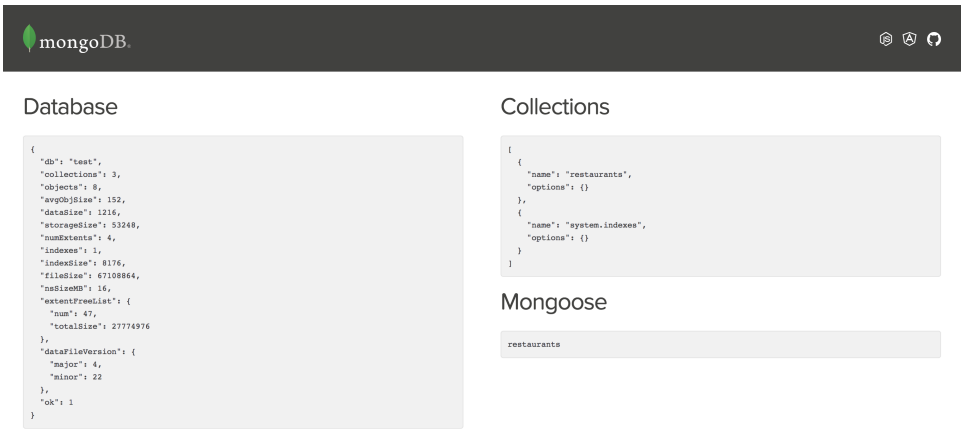


Abbildung 4.3: Dashboard View

4.5.2 Dashboard View

Die Dashboard View (Abbildung 4.3) gibt dem Benutzer einen kurzen Überblick über die aktuell verbundene *MongoDB*. Dazu gehören Informationen und Statistiken wie der Datenbankname, die Anzahl an Collections, die Anzahl an Dokumenten oder die durchschnittliche Dokumentengröße. Weiters werden dem Benutzer sämtliche Collections inkl. den System Collections aufgelistet. Eine weitere Liste inkludiert nur die *Mongoose* Collections, die mit dem Tool bearbeitet werden können.

4.5.3 Collection View

In der Collection View (Abbildung 4.4) hat der Benutzer eine Auflistung aller Dokumente, die sich in der entsprechenden Collection befinden. Die Liste kann außerdem über eine Suche gefiltert werden.

4.5.4 Document View

Die Document View (Abbildung 4.5) ist das Herzstück des Tools. Darin ist es möglich, Dokumente anzulegen, zu editieren oder zu löschen. Die generierte Form entspricht exakt dem *Mongoose* Schema der ausgewählten Collection. Somit eignet sich das Tool optimal als Backend-Modul für eine App, die auf *Mongoose* aufbaut.

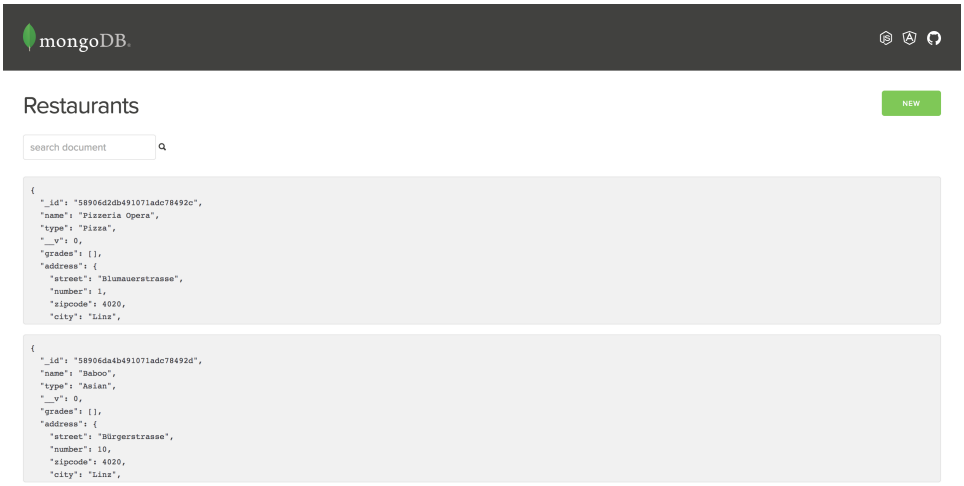


Abbildung 4.4: Collection View

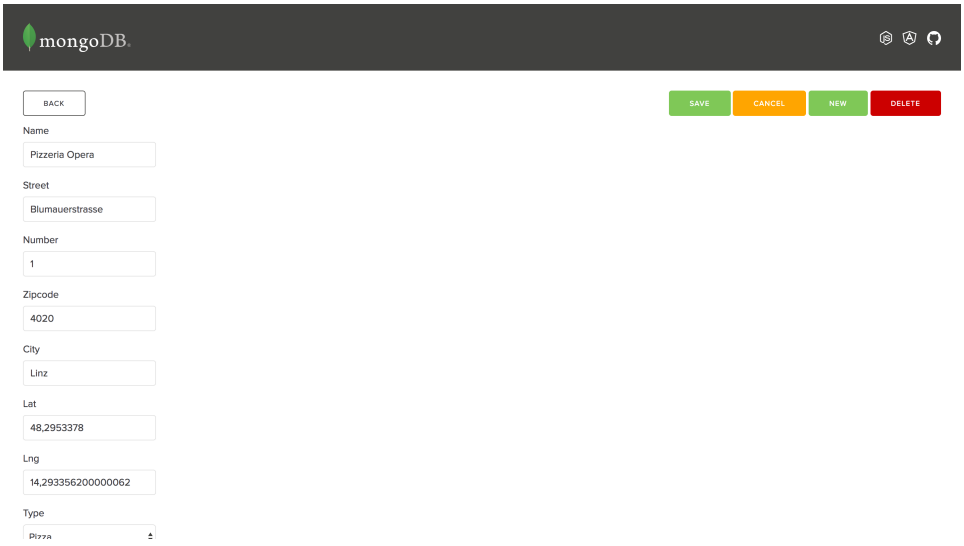


Abbildung 4.5: Document View

4.5.5 Performance

Der zweite wesentliche Teil der Implementierung war die Durchführung von Performance Tests. Dabei wurde *Mongoose* für *MongoDB* mit Dynamic Columns für *MariaDB* verglichen. Es wurden alle CRUD-Operationen auf beiden Seiten getestet und anschließend gegenübergestellt. Auf die verwendeten Testdaten sowie die Ergebnisse wird genauer im nächsten Kapitel eingegangen.

Programm 4.5: Timeout

```
setTimeout(function () {
    console.time('Test');
    // Performance Test
    console.timeEnd('Test');
}, 3000);
```

Wie das Administrations Tool wurden auch die Performance Tests in JavaScript mit *Node.js* implementiert, wobei folgende Module verwendet wurden:

- *mongoose*,[19]
- *mariasql*.[20]

Die Tests wurden dabei etwas verzögert ausgeführt, damit sie von der eigentlichen App nicht beeinflusst werden. Dies ist in JavaScript mit der Funktion `setTimeout` möglich. Programm 4.5 zeigt dies mit einer Verzögerung von 3 Sekunden (3000 Millisekunden). Um die Zeit des eigentlichen Tests zu messen, wurden die Funktionen `console.time` und `console.timeEnd` verwendet. Dadurch wird nach Ausführung des Performance Tests die entsprechend benötigte Zeit in Millisekunden in der Konsole ausgegeben.

[19] https://www.npmjs.com/package/mongoose
[20] https://www.npmjs.com/package/mariasql

Kapitel 5

Evaluierung

Dieses Kapitel soll zur kritische Betrachtung der Implementierung dienen. Dabei wird genau auf das verwendete Framework eingegangen. Außerdem werden die durchgeführten Performance Tests genau interpretiert und damit ein Vergleich zwischen Dynamic Columns für *MariaDB* und *Mongoose* für *MongoDB* gezogen.

5.1 Framework

Das Framework *forms-angular* von Mark Chapman war eine große Hilfe bei der Implementierung des auf *Mongoose* basierenden *MongoDB* Administrations Tool. Das Framework baut selbst auf dem MEAN-Stack auf und wird auf der offiziellen Website mit folgender Definition beschrieben [18]:

> Probably the most opinionated framework in the world.

Ziel des Frameworks ist es, Schemadefinitionen serverseitig zu interpretieren, daraus die entsprechenden REST-Routen zu definieren und clientseitig die entsprechenden CRUD-Operationen und Forms zu generieren. Das Framework wird seit 2015 entwickelt und ständig durch neue Funktionen erweitert. Außerdem kann es relativ leicht über *AngularJS*-Controller erweitert werden. Weiters stehen einige Plugins zur Verfügung. In den nachfolgenden Abschnitten wird genauer auf die beiden Module für das Backend bzw. Frontend eingegangen. Dabei wird konkret die im vorherigen Kapitel beschriebene Implementierung herangezogen.

5.1.1 Backend Modul

Das Backend Modul hat die Aufgabe, anhand von Schemadefinitionen die entsprechende API zur Verfügung zu stellen. Mit Hilfe dieser API ist es möglich, neue Dokumente zu erstellen sowie bestehende Dokumente zu editieren bzw. zu löschen. Außerdem können alle Dokumente einer Collection abgefragt werden. Die Installation ist über *npm* möglich:

```
npm install --save forms-angular
```

Anschließend muss das Modul über *Node.js* eingebunden und konfiguriert werden. Dabei wird definiert, unter welcher URL die API laufen soll und welches *Mongoose* Model verwendet werden soll:

```
var formsAngular = require('forms-angular');
var DataFormHandler = new (formsAngular)(app, {urlPrefix: '/api/'});
DataFormHandler.newResource(Restaurant);
```

Ab diesem Zeitpunkt können über die URL `/api` sämtliche Operationen für die Collection `restaurants` durchgeführt werden.

5.1.2 Frontend Modul

Die Aufgabe des Frontend Moduls ist, aus der API das entsprechende Schema abzufragen und anschließend daraus die korrekte Form zu generieren. Dieses Modul kann über *bower* installiert werden:

```
bower install --save forms-angular
```

Nach Einbinden der entsprechenden JavaScript-Dateien ist noch die Konfiguration über *AngularJS* notwendig. Dabei wird der `routingService` gestartet und über verschiedene Parameter konfiguriert. Die vordefinierten Routen können über die Eigenschaft `fixedRoutes` überschrieben bzw. erweitert werden.

```
formsAngular.config(['routingServiceProvider', function(routingService){
    routingService.start({
        html5Mode: true
    });
}]);
```

Das Framework *forms-angular* erleichterte die Implementierung enorm. Durch die ständige Weiterentwicklung ist das Framework im Umgang mit *Mongoose* bereits sehr ausgereift und kann außerdem über *AngularJS* beliebig erweitert werden. Da es selbst auf die Architektur des MEAN-Stacks aufbaut, war die Einbindung recht einfach und logisch. Im Vergleich zu anderen Frameworks war es die perfekte Lösung, da sowohl die Server-Seite als auch die Client-Seite mit einem Framework abgedeckt werden konnten.

5.2 Performance

Wie bereits im vorherigen Kapitel erwähnt, stellten die Performance Tests einen wichtigen Teil der Implementierung dar. Es gibt zu diesem Thema bereits mehrere interessante Ergebnisse, die sich jedoch vor allem auf den Vergleich zwischen *MySQL* und *MongoDB* beziehen [4, 5, 7]. Dabei war *MongoDB* in den meisten Fällen klar überlegen.

Zwischen *MariaDB* und *MongoDB* sind bis jetzt nicht wirklich Performance Tests zu finden. Umso spannender gestaltete sich die Umsetzung und Durchführung dieser Tests im Zuge der Implementierung. Der klare Fokus lag dabei auf der Implementierung flexibler Schemata, also auf *Mongoose* für *MongoDB* und Dynamic Columns für *MariaDB*.

5.2.1 Testdaten

Der erste Schritt war die Auswahl von passenden Testdaten. Eine wichtige Voraussetzung war dabei ein dynamisches Feld, um die Stärken flexibler Schemata testen zu können.

Schlussendlich wurde dazu ein Teil des Datenmodells der E-Commerce-Plattform *Magento*[1] verwendet. Dabei wurde speziell auf die Tabelle für die Speicherung von Produkten zurückgegriffen, welche unter anderem folgende wichtige Felder enthält:

```
product_id
sku
category_ids
created_at
updated_at
has_options
required_options
```

Die `product_id` stellt den eindeutigen Schlüssel für jedes Produkt dar. Die Stock Keeping Unit (`sku`) ist die Artikelnummer eines lagergeführten Produkts. Das Feld `category_ids` ist besonders wichtig, da dieses eine beliebige Anzahl an Kategorien für ein Produkt enthalten kann. Somit konnte dieses Feld als dynamisches Feld für die Tests herangezogen werden.

Weiters werden noch Zeitstempel beim Erstellen bzw. Aktualisieren eines Produkts gesetzt. Ein Produkt kann außerdem unterschiedliche Optionen haben. Diese Eigenschaften werden in den Feldern `has_options` und `required_options` verwaltet.

5.2.2 Herangehensweise

Der zweite Schritt war die eigentliche Implementierung der Tests. Dazu musste zu Beginn mit Hilfe der beiden *Node.js*-Module eine Verbindung zu den beiden Datenbanken hergestellt werden:

[1] https://magento.com/

Tabelle 5.1: Tabelle Products

Name	*Typ*
product_id	int (11)
sku	text
category_ids	blob
created_at	bigint(20)
updated_at	bigint(20)
has_options	int(1)
required_options	int(1)

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/test');

var mariasql = require('mariasql');
var c = new mariasql({
    host: 'localhost',
    user: 'root',
    password: '',
    db: 'test'
});
```

Es wurde dazu jeweils eine lokale Datenbank mit dem Namen `test` verwendet. Zum Verbindungsaufbau mit *Mongoose* wird der Funktion `connect` der entsprechende Connection-String übergeben. Für *MariaDB* sind der Host (`host`), der Benutzer (`user`), das Passwort (`password`) sowie die Bezeichnung der Datenbank (`db`) erforderlich.

Anschließend wurde aus den entsprechenden Feldern ein *Mongoose* Schema (Programm 5.1) sowie eine *MariaDB* Tabellendefinition (Tabelle 5.1) erstellt. Die dynamischen `category_ids` werden unter *MariaDB* als `blob` gespeichert. In *MongoDB* ist eine Kategorie verpflichtend und alle weiteren Kategorien sind optional. Somit kann dieses Feld in beiden Fällen dynamisch befüllt werden.

Der letzte Schritt vor der Ausführung der Tests war die Generierung der Testdaten. Dazu wurde die `product_id` aufsteigend vergeben. Die `sku` wurde aus einer zufälligen Kombination von drei Buchstaben und drei Ziffern generiert. Für die `category_ids` wurde zu Beginn eine Zufallszahl zwischen 1 und 10 für die Anzahl an Kategorien generiert. Entsprechend dieser Zahl wurden dann die Kategorien für jedes Produkt dynamisch erzeugt. Für die Felder `created_at` und `updated_at` wurde der aktuelle Zeitstempel verwendet. Die Felder für die `options` wurden einfachheitshalber mit 0 oder 1 belegt.

Tabelle 5.2: Durchschnittswerte CREATE (ms)

Datensätze	*MariaDB*	*MongoDB*
1	14,7	25,5
10	23,3	46,8
100	57,8	155,6
1000	233,9	910,1
10000	2209,5	8645,0

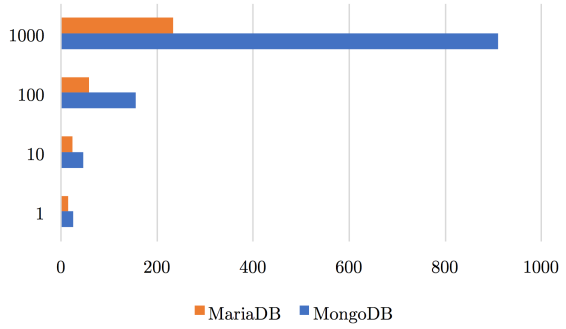


Abbildung 5.1: Diagramm CREATE

5.2.3 Ergebnisse

Für die Tests wurden alle vier CRUD-Operationen herangezogen. Davon wurde jeder Test mit 1, 10, 100, 1000 und 10000 Datensätzen durchgeführt. Innerhalb dieser Gruppen wurde wiederum jeder Test 10 mal ausgeführt, wobei anschließend davon der Durchschnitt berechnet wurde. Das führte insgesamt zu 400 Messergebnissen, die nachfolgend genauer interpretiert werden. Zur besseren Veranschaulichung wurde bei den Diagrammen auf die Tests mit 10000 Datensätzen verzichtet, da sonst die kleineren Messungen nicht erkennbar wären.

CREATE

Die ersten Tests wurden mit dem Einfügen von Produkten durchgeführt. Dabei waren bereits klare Vorteile für *MariaDB* erkennbar (Abbildung 5.1). Tabelle 5.2 zeigt die genauen Ergebnisse der einzelnen Kategorien. Bei 10 Datensätzen war *MariaDB* bereits doppelt so schnell wie *MongoDB*. Besonders deutlich wurde das Ergebnis dann ab 100 Datensätzen. Dabei war *MariaDB* bereits drei mal so schnell, bei 1000 Datensätzen sogar vier mal so schnell wie *MongoDB*. Daraus lässt sich klar schließen, dass die Speicherung des *Mongoose* Schemas für jeden Datensatz klar die Performance beeinflusst.

Tabelle 5.3: Durchschnittswerte READ (ms)

Datensätze	*MariaDB*	*MongoDB*
1	1,3	12,5
10	1,6	15,8
100	2,4	36,7
1000	8,6	166,3
10000	57,6	1368,5

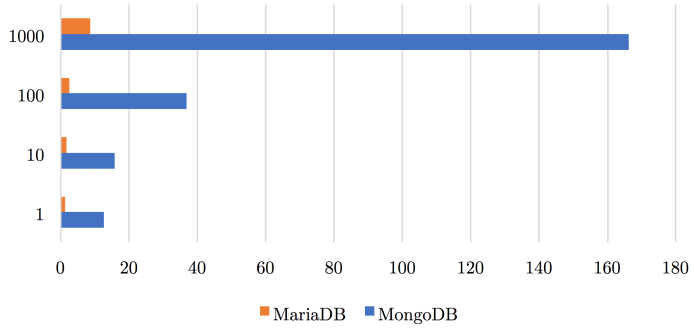


Abbildung 5.2: Diagramm READ

READ

Auch beim lesenden Zugriff auf die Daten war *MariaDB* eindeutig schneller als *MongoDB*. Abbildung 5.2 zeigt, dass *MongoDB* bereits beim Lesen eines einzigen Datensatzes *MariaDB* weit hinterherhinkt. Ähnlich deutlich war das Ergebnis auch bei 10 bzw. 100 Datensätzen.

Weiters ist in Tabelle 5.3 ersichtlich, dass *MariaDB* für das Lesen von 1000 Datensätzen nur 8,6 Millisekunden brauchte – *MongoDB* im Gegensatz dazu jedoch 166,3 Millisekunden. Dies entspricht fast einem Faktor von 20. Bei 10000 Datensätzen war das Ergebnis noch deutlicher, wobei *MariaDB* immer noch schneller war als *MongoDB* bei den 1000 Datensätzen.

Bei der READ-Operation konnte von einem ausgeglicheneren Ergebnis der Tests ausgegangen werden. Gerade beim lesenden Zugriff sollte das Schema eher wenig Einfluss haben. Daher war die klare Überlegenheit von *MariaDB* in dieser Kategorie eher überraschend.

Tabelle 5.4: Durchschnittswerte UPDATE (ms)

Datensätze	*MariaDB*	*MongoDB*
1	3,1	8,8
10	16,3	12,7
100	79,8	49,6
1000	261,3	594,6
10000	2171,6	41190,0

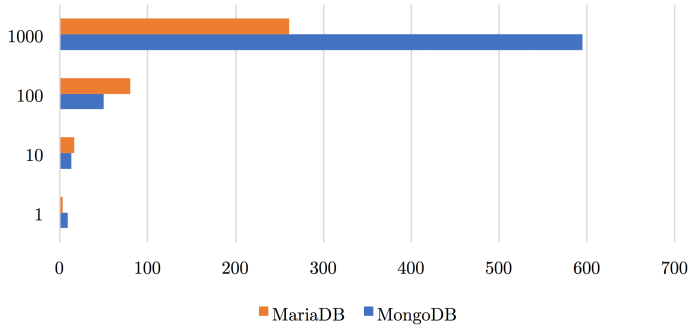


Abbildung 5.3: Diagramm UPDATE

UPDATE

Beim Editieren von Produkten konnte *MongoDB* sehr lange gut mithalten und war bei 10 bzw. 100 Datensätzen sogar schneller als *MariaDB* (Abbildung 5.3). Eher deutlich wurde das Ergebnis dann jedoch ab 1000 Datensätzen. Dafür benötigte *MongoDB* bereits in etwa die doppelte Zeit von *MariaDB*. Besonders eindeutig waren die Ergebnisse dann bei 10000 Datensätzen. Wie in Tabelle 5.4 ersichtlich, benötigte *MariaDB* dafür nur knapp 2 Sekunden. *MongoDB* hingegen brauchte dafür durchschnittlich rund 41 Sekunden, was in dieser Deutlichkeit nicht zu erwarten war.

Auch hier spielt das *Mongoose* Schema wohl eine große Rolle und zehrt eindeutig an der Performance. Warum bei 10000 Datensätzen jedoch ein derart großer Sprung zwischen *MariaDB* und *MongoDB* stattgefunden hat, ist eher schwer nachzuvollziehen.

Tabelle 5.5: Durchschnittswerte DELETE (ms)

Datensätze	*MariaDB*	*MongoDB*
1	2,2	4,3
10	2,6	4,7
100	3,0	5,5
1000	5,7	12,2
10000	28,5	72,2

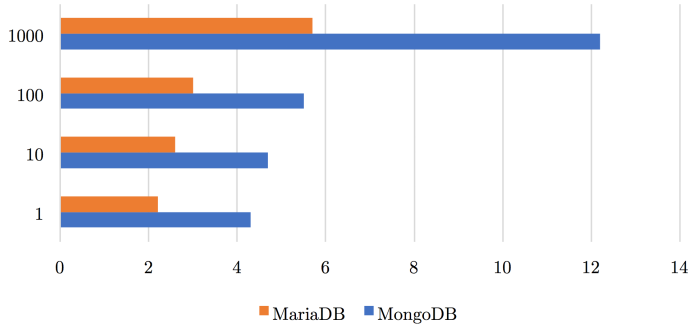


Abbildung 5.4: Diagramm DELETE

DELETE

Das Löschen von Datensätzen kann als relativ zeitsparende Operation gesehen werden. Dabei waren sowohl *MariaDB* als auch *MongoDB* sehr schnell unterwegs, wobei auch hier *MariaDB* eine etwas bessere Performance erzielen konnte (Abbildung 5.4).

Tabelle 5.5 zeigt dazu die genauen Ergebnisse. Das Löschen von 10000 Produkten wurde dabei von *MariaDB* in knapp 28 Millisekunden und von *MongoDB* in etwa 72 Millisekunden erledigt. Das entspricht zwar ca. einem Faktor von 2,5. Dies ist jedoch bei einer derart kurzen Zeit nicht wirklich erwähnenswert.

Programm 5.1: Product Schema

```
var ProductSchema = mongoose.Schema({
    product_id: {
        type: Number,
        required: true
    },
    sku: {
        type: String,
        required: true
    },
    category_ids: {
        category_1: {
            type: Number,
            required: true
        },
        category_2: Number,
        category_3: Number,
        category_4: Number,
        category_5: Number,
        category_6: Number,
        category_7: Number,
        category_8: Number,
        category_9: Number,
        category_10: Number
    },
    created_at: {
        type: Number,
        required: true
    },
    updated_at: {
        type: Number,
        required: true
    },
    has_options: {
        type: Number,
        required: true
    },
    required_options: {
        type: Number,
        required: true
    }
}, {
    versionKey: false
});
```

Kapitel 6

Schlussbetrachtung

Der Einsatz des richtigen Datenbanksystems ist stark vom jeweiligen Anwendungsfall abhängig. Relationale Datenbanken haben in vielen Anwendungen immer noch klare Vorteile und erzielen eine bessere Performance. Anwendungen mit großen Datenmengen und hoher Flexibilität sollten jedoch eher mit NoSQL-Datenbanken umgesetzt werden. Daher ist vor Beginn der Umsetzungsphase eine ausführliche Evaluierung der Daten bzw. der möglichen Datenbanksysteme enorm wichtig.

Die durchgeführten Performance Tests führten zu interessanten Ergebnissen. Die klare Überlegenheit von *MariaDB* war in dieser Deutlichkeit nicht zu erwarten. Anzumerken ist jedoch, dass die große Stärke von *MongoDB*, nämlich die horizontale Skalierung durch Sharding, in diesen Tests keine Anwendung gefunden hat. Die Ergebnisse hätten dadurch bzw. mit größeren Datenmengen vermutlich etwas anders ausgesehen.

Fazit

MariaDB als Vetreter der relationalen Datenbanken setzt mit dem Konzept der Dynamic Columns ein klares Zeichen in Richtung Schema-Flexibilität. Das starre Schema ohne nachträgliche Änderungen hat ausgedient. *MongoDB* unter den NoSQL-Datenbanken geht grundsätzlich den Weg der kompletten Schemafreiheit. *Mongoose* für *Node.js* hat sich zum Ziel gesetzt, eine gewisse Struktur in die Daten der *MongoDB* zu bringen.

Diese Entwicklung der gegenseitigen Annäherung ist in den letzten Jahren durchaus interessant zu beobachten. Auf der einen Seite will man weg vom starren Schema und auf der anderen Seite hin zu etwas mehr Struktur der Daten. Wie diese Entwicklung weitergeht, ist mit Spannung zu erwarten.

Anhang A

Inhalt der CD-ROM

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 PDF-Dateien

Pfad: /

Steidl_Thomas.pdf . .	Masterarbeit (Gesamtdokument)

A.2 LaTeX-Dateien

Pfad: /latex

*.tex	LaTeX-Dateien sämtlicher Kapitel
*.bib	Literatur-Datenbank

A.3 Abbildungen

Pfad: /images

*.pdf	Original Vektorgrafiken
*.png	Original Rasterbilder

A.4 Quellen

Pfad: /sources

*.pdf	Online-Quellen als PDF-Dateien
*.html	Online-Quellen als HTML-Dateien

A.5 Projekt

Pfad: /project

thesis-project.zip	MongoDB Administrations Tool und Performance Tests

Quellenverzeichnis

Literatur

[1] Daniel Bartholomew. *MariaDB Cookbook. Over 95 recipes to unlock the power of MariaDB*. Birmingham: Packt, 2014 (siehe S. 18).

[2] Thomas Connolly und Carolyn Begg. *Database Systems. A Practical Approach to Design, Implementation, and Management*. 6. Aufl. Harlow: Pearson, 2015 (siehe S. 3).

[3] Stefan Edlich u. a. *NoSQL. Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. 2. Aufl. München: Hanser, 2011 (siehe S. 13).

[4] C. Győrödi u. a. „A comparative study: MongoDB vs. MySQL“. In: *Proceedings of the International Conference on Engineering of Modern Electric Systems (EMES)*. Juni 2015, S. 1–6 (siehe S. 45).

[5] Y. Li und S. Manoharan. „A performance comparison of SQL and NoSQL databases“. In: *Proceedings of the Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. Aug. 2013, S. 15–19 (siehe S. 45).

[6] Andreas Meier und Michael Kaufmann. *SQL- & NoSQL-Datenbanken*. 8. Aufl. Heidelberg: Springer, 2016 (siehe S. 14–17).

[7] Zachary Parker, Scott Poe und Susan V. Vrbsky. „Comparing NoSQL MongoDB to an SQL DB“. In: *Proceedings of the 51st ACM Southeast Conference*. ACMSE '13. Savannah, Georgia: ACM, 2013, 5:1–5:6 (siehe S. 45).

[8] Johannes Schildgen. *MongoDB kompakt. Was Sie über die NoSQL-Dokumentendatenbank wissen müssen*. Norderstedt: Books on Demand, 2016 (siehe S. 24).

[9] Thomas Studer. *Relationale Datenbanken. Von den theoretischen Grundlagen zu Anwendungen mit PostgreSQL*. Heidelberg: Springer, 2016 (siehe S. 12).

[10] Philipp Tarasiewicz und Robin Böhm. *AngularJS. Eine praktische Einführung in das JavaScript-Framework*. Heidelberg: dpunkt.verlag, 2014 (siehe S. 34).

[11] Tobias Trelle. *MongoDB. Der praktische Einstieg.* Heidelberg: dpunkt.verlag, 2014 (siehe S. 24).

Online-Quellen

[12] *A MariaDB Primer - MariaDB Knowledge Base.* URL: https://mariadb.com/kb/en/mariadb/a-mariadb-primer/ (siehe S. 18).

[13] *ANSI-3-Ebenenmodell Definition & Erklärung.* URL: http://www.datenbanken-verstehen.de/lexikon/ansi-drei-ebenenmodell (siehe S. 6).

[14] *Beziehungen in Datenbanken.* URL: http://www.datenbanken-verstehen.de/datenmodellierung/beziehungen-datenbanken/ (siehe S. 8).

[15] *Dynamic Columns - MariaDB Knowledge Base.* URL: https://mariadb.com/kb/en/mariadb/dynamic-columns/ (siehe S. 22).

[16] *Dynamic Columns Tutorial – Part 1: Introduction | MariaDB.* URL: https://mariadb.com/resources/blog/dynamic-columns-tutorial-part-1-introduction (siehe S. 22).

[17] *Dynamic Columns Tutorial – Part 2: Searching and Updating | MariaDB.* URL: https://mariadb.com/resources/blog/dynamic-columns-tutorial-part-2-searching-and-updating (siehe S. 22).

[18] *forms-angular.* URL: https://www.forms-angular.org/ (siehe S. 43).

[19] *MariaDB Documentation - MariaDB Knowledge Base.* URL: https://mariadb.com/kb/en/mariadb/documentation/ (siehe S. 18).

[20] *Mongoose ODM.* URL: http://mongoosejs.com/ (siehe S. 29).

[21] *Mongoose Schemas.* URL: http://mongoosejs.com/docs/guide.html (siehe S. 29).

[22] *Node.js.* URL: https://nodejs.org/en/ (siehe S. 33).

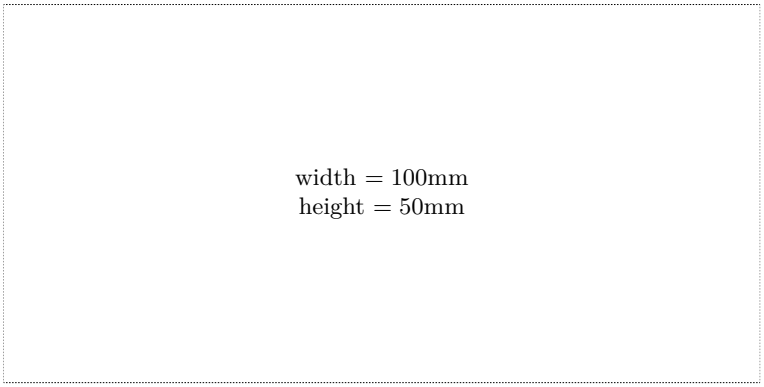
[23] *Normalisierung von Datenbanken.* URL: http://www.datenbanken-verstehen.de/datenmodellierung/normalisierung/ (siehe S. 8).

[24] *What Is Big Data?* URL: http://www.gartner.com/it-glossary/big-data/ (siehe S. 12).

[25] *Where is NoSQL practically used?* URL: http://scraping.pro/where-nosql-practically-used/ (siehe S. 14–17).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —