MongoDB

Andreas Weber
Universität Siegen
andreas.weber@student.uni-siegen.de

ABSTRACT

In diesem Artikel wird das Datenbanksystem MongoDB vorgestellt. Nach einer kurzen Einleitung werden anhand eines Beispiels das Datenmodell sowie die Abfragesprache erläutert. Dabei werden Unterschiede und Gemeinsamkeiten zu relationalen Datenbanksystemen erörtert. Neben Abfrageoptimierungen werden Eigenschaften von MongoDB thematisiert, die beim Schema-Design eine Rolle spielen. Abschließend wird beschrieben, wie MongoDB auf mehreren Servern eingesetzt werden kann. In erster Linie richtet sich der Artikel an Entwickler, die bereits mit relationalen Datenbanksystemen gearbeitet haben, sich jedoch für MongoDB interessieren.

1. EINLEITUNG

Auf dem Markt existieren viele verschiedene Datenbankmanagementsysteme (DBMS). Die meisten davon basieren auf dem relationalen Modell, das die Abfragesprache SQL (*Structured Query Language*) verwendet. Demgegenüber stehen die sogenannten NoSQL-Systeme (*Not only SQL*), die von dem relationalen Modell Abstand nehmen, darunter auch MongoDB.

MongoDB wurde 2007 gegründet und ist damit noch ein recht junges DBMS [2]. Dennoch wird es bereits von vielen großen Unternehmen erfolgreich eingesetzt. MongoDB adressiert eine bestimmte Nutzerschaft: Kunden mit großen Datenmengen. Diese Spezialisierung gibt MongoDB den Namen, der sich von dem englischen Begriff *"humongous"* (gigantisch) ableitet.

2. GRUNDLAGEN

2.1 Datenmodell

Jede nichtleere Datenbank in MongoDB enthält *collections*. Sie entsprechen den Tabellen im relationalen Schema insofern, dass sie Daten zusammenfassen. Allerdings sind die enthaltenen Elemente keine Reihen bzw. Tupel, sondern Dokumente. Man bezeichnet MongoDB daher auch als dokumentenorientiertes DBMS. Abgelegt werden die Dokumente im BSON-Format.

Syntaktisch gibt es große Übereinstimmungen zwischen dem BSON- und JSON-Format, die sich an Figure 1 nachvollziehen lassen. Das Beispiel soll ein Dokument aus einer vereinfachten Datenbank darstellen, die der Verwaltung von öffentlichen Gebäuden dient. Felder werden als Paar nach dem Schema `name:wert` definiert. Mehrere Felder werden durch ein Komma getrennt. Es ist zu erkennen, dass Dokumente strukturiert werden können: Zum einen kann anstelle eines Werts ein Array angegeben werden, das (mehrere) kommaseparierte Werte innerhalb von eckigen Klammern enthält. Zum anderen sind Hierachien möglich, die es ermöglichen Subdokumente zu erstellen.

```
{
  "_id" :ObjectId("566829
     f12bbc8f37511ad19e"),
  "name" : "Adolf Reichwein",
  "address" : {
    "street" : "Hölderlinstraße",
    "number" : 3,
    "city" : "Siegen",
    "location" : [50.906344, 8.028664]
  },
  "rooms" : [
    {"number":1, "type":"auditorium"},
    {"number":2, "type":"laboratory"},
    {"number":2, "type":"administration"}
  ]
}
```

Figure 1: Beispiel für ein Dokument

Im Gegensatz zu JSON verfügt BSON über weitere Datentypen. Für eine effiziente Codierung werden diese durchnummeriert. Besonders interessant sind die Datentypen *ObjectID* und *Timestamp* [8].

Jedes Dokument besitzt das Feld `_id`. Falls es nicht explizit überschrieben wurde, ist es von Typ *ObjectId*. Der 12 Byte große Datentyp setzt sich aus einem Zeitstempel (4 Byte), einem Rechner Indentifizierer (3 Byte), einem Prozess Indentifizierer (2 Byte) und einem Zähler (3 Byte) zusammen [5]. Dadurch kann jedes Dokument nicht nur eindeutig identifiziert, sondern auch genau lokalisert werden.

Der Datentyp *Timestamp* wird hauptsächlich intern verwendet. Seine Gesamtgröße liegt bei 64 Bit. Die ersten 32 Bit geben die *Unixzeit* auf eine Sekunde genau an, während

die weiteren 32 Bit einen Operationszähler auf dem Dokument realisieren. Letzterer wird bei jeden Zugriff auf das Dokument inkrementiert und jede Sekunde zurückgesetzt. Auf diese Weise kann nachvollzogen werden, wie aktuell die Werte in einem Dokument sind, was für das Thema Replikation (Kapitel 5) von Bedeutung ist [8].

2.2 CRUD mit JavaScript

Relationale Datenbanken verwenden die Abfragesprache SQL. Solch eine domänenspezifische Sprache existiert für MongoDB nicht. Stattdessen wurde die Abfragesprache in JavaScript umgesetzt. Das für die Abfragen zentrale Element ist das Objekt `db`, das die aktuell verwendete Datenbank repräsentiert. Auf dem Objekt können mit der in JavaScript üblichen Syntax Methoden aufgerufen werden, die vor allem administrative Funktionen bereitstellen. Eine Auflistung aller verfügbaren Methoden erhält man durch den Aufruf von `db.help()`.

Der Zugriff auf eine *collection* erfolgt nach dem Schema `db.collection.method()`. Im Folgenden werden die wichtigsten Methoden vorgestellt, die auf *collections* angewandt werden. Sie stellen die Basisfunktionalität bereit, die jedes DBMS bieten sollte – auch bekannt unter der Abkürzung *CRUD* (*Create, Read, Update, Delete*).

2.2.1 Create

Neue Dokumente werden mit der Methode `insert()` erzeugt. Als Paramenter erwartet die Funktion ein BSON-Dokument, das in die angegebene *collection* eingefügt wird. Falls die *collection* nicht existiert, wird diese angelegt:

```
db.buildings.insert({
  name: "Adolf Reichwein",
  address : {
    street : "Hölderlinstraße",
    number : 3,
    city : "Siegen",
    location: [ 50.906344, 8.028664 ]
  },
  rooms : [
    {number:1, type: "auditorium"},
    {number:2, type: "laboratory"},
    {number:3, type: "administration"}
  ]
})
```

Figure 2: Einfügen von Dokumenten

Damit unterscheidet sich MongoDB grundlegend von SQL. Hier muss eine Tabelle mit einem eigenen Befehl erzeugt werden, der ein festes Schema definiert.

```
CREATE TABLE buildings (
  id SERIAL PRIMARY KEY,
  name VARCHAR(30),
  ...
)
```

Erst danach ist es möglich, mit dem Befehl `INSERT INTO buildings VALUES (...)` Tupel in die Tabelle einzufügen.

Da eine Schemadefinition entfällt, spricht man bei MongoDB auch von einem schemafreien DBMS. Die Dokumente innerhalb einer *collection* können eine beliebige Struktur haben - solange sie syntaktisch korrekt sind. Lediglich das Feld `_id` muss jedes Dokument besitzen. In der Praxis haben Dokumente innerhalb einer *collection* natürlich gemeinsame Felder, damit sie vergleichbar sind. Dennoch ist MongoDB in dieser Hinsicht nicht so einschränkend wie relationale DBMS und damit wesentlich flexibler. Diese Flexibilität birgt jedoch auch Risiken, da jederzeit falsche Werte eingetragen werden können.

2.2.2 Read

Lesezugriffe auf eine *collection* erfolgen mit der Methode `find()`. Es handelt sich dabei um eine überladene Methode: wird die Methode ohne Parameter angewandt, werden alle Dokumente der *collection* gelistet. Ein Parameter nimmt die Rolle eines Selektors an, während ein zweiter Parameter das Ergebnis projeziert. Die Parameter werden wie bei der `insert()` Methode als BSON-Objekt übergeben. Im Selektor werden Schlüssel und Werte angegeben. Alle Dokumente, die für den jeweiligen Schlüssel den festgelegten Wert besitzen, sind Teil des Ergebnisses. Durch vordefinierte Operanden können komplexe Selektionsbedingungen definiert werden. Mit der Projektion wird das Ergebnis auf die benötigten Felder reduziert. Deutlich wird die Funktionsweise anhand folgenden Beispiels:

```
db.buildings.find(
  // Selektion
  {
    // AND Verknüpfung
    $and: [
      // Gebäude in Siegen...
      {"address.city" : "Siegen"},
      // ...mit drei Räumen
      {rooms : {$size : 3}}
    ]
  },
  // Projektion
  {
    // nur name anzeigen
    name: true
  }
)
```

Figure 3: Eine Abfrage in MongoDB

Im Selektor ist ein Array definiert, das zwei Bedingungen enthält. Die erste Bedingung besagt, dass das Feld `city` im Segment `address` den Wert `Siegen` haben muss. Die zweite Bedingung schränkt die Ergebnisse auf Dokumente ein, bei denen das Array `rooms` eine Größe von drei hat. Dadruch, dass die Bedingunen mit `$and` verknüpft sind, werden nur die Dokumente selektiert, bei denen beide Bedinungen erfüllt sind. Die Projektion im zweiten Paramenter reduziert die Ausgabe auf das Feld `name`.

Bei genauer Betrachtung des Beispiels fallen zwei Details auf, die ungewöhnlich umgesetzt scheinen. Zum einen ist das Feld `address.city` das einzige, das in Anführungszeichen gesetzt ist, zum anderen würde man bei der Überprüfung der Arraygröße intuitiv eine Syntax der Form `room.size==3` verwenden. Der Grund für die Syntax liegt bei JavaScript. Die Punktnotation zum Navigieren auf Objekten sowie Vergleichsoperatoren sind bereits für die Skriptsprache reserviert.

Darum werden verschachtelte Felder als String angegeben bzw. vordefinierte Operanden verwendet, die an dem Dollarzeichen zu erkennen sind [13].

2.2.3 Update

Neben einer Funktion, die neue Dokumente anlegt, benötigt man eine Methode, mit der man Änderungen auf bestehende Dokumente vornehmen kann. In MongoDB heißt diese Methode `update()`. Die Anwendung erfolgt ähnlich zur `find()` Methode: Der erste Parameter ist der Selektor. Auf alle Dokumente, die dem Muster des Selektors entsprechen, werden die im zweiten Parameter definierten Änderungen vorgenommen.

```
db.buildings.update(
  {
    name:"Adolf Reichwein",
    "rooms.number" : 2
  },
  {
    $set:{"rooms.$.renovated":
      ISODate("2015-10-03")}
  }
)
```

Figure 4: Änderung eines bestehenden Dokuments

In dem Beispiel wurde Raum 2 im Gebäude "Adolf Reichwein" renoviert. Um diesen Vorgang im Datenmodell aufzunehmen, wurde dem Raum-Element ein Feld vom Typ *ISODate* hinzugefügt (siehe Figure 4). Mit SQL wäre eine solche Operation nicht möglich, ohne das Schema anzupassen. Dies bedeutet in der Regel, dass die betroffene Tabelle für die Dauer der Anpassung gesperrt wird. Außerdem kann eine Änderung nicht an einer einzelnen Reihe vorgenommen werden, da für die anderen Reihen der Tabelle *Nullfelder* eingetragen werden müssten, die es im Allgemeinen zu vermeiden gilt.

2.2.4 Delete

Die `delete()` Methode funktioniert analog zur `find()` Methode – nur ohne Projektion. D. h. es wird lediglich ein Selektor übergeben. Alle Dokumente, die den Kriterien des Selektors entsprechen, werden vollständig gelöscht. Möchte man lediglich einzelne Felder oder Subdokumente entfernen, verwendet man den Operator `$unset` in der `update()` Methode analog zu Figure 4.

Das Kapitel stellt lediglich einen Bruchteil der in MongoDB verfügbaren Methoden und Operanden vor. Insbersondere wurden jene ausgelassen, die aggregierte Abfragen erlauben. Wie in SQL existieren auch in MongoDB Anweisungen, die diese erlauben. Table 1 gibt einen Überblick über verfügbare Operanden, die in diesem Kontext wichtig sind.

Abgesehen von Verbundoperationen ist die Abfragesprache als ähnlich mächtig wie SQL einzuordnen. Im Gegensatz zu manch anderen NoSQL-DBMS sind die Abfragen *ad-hoc*. Das heißt Abfragen können jederzeit ausgeführt werden, ohne vordefiniert zu sein [13]. Das Datenmodell von MongoDB ist sehr flexibel und lässt sich leicht erweitern. Wer zuvor nur mit SQL gearbeitet hat, mag anfangs von der Syntax verwirrt sein. Den Einstieg erleichtern jedoch gemeinsame Konzepte wie Selektion und Projektion oder etwa Gruppierungen.

SQL	MongoDB
`WHERE`	`$match`
`GROUP BY`	`$group`
`HAVING`	`$match`
`SELECT`	`$project`
`ORDER BY`	`$sort`
`LIMIT`	`$limit`
`SUM()`	`$sum`
`COUNT`	`$sum`
`JOIN`	siehe Kapitel 4

Table 1: Aggregierte Abfragen mit SQL und MongoDB [10]

3. ABFRAGEOPTIMIERUNG

MongoDB ist darauf spezialisiert, große Datenmengen zu verarbeiten. Lässt man sich mit der Methode `explain("executionStats")` die Statisken der Abfrage aus Figure 3 ausgeben, wird man feststellen, dass sämtliche Dokumente der *collection* nach den definierten Kriterien durchsucht werden. Während 100 Dokumente noch schnell durchsucht werden, wird der Vorgang bei 10000 Dokumenten schon deutlich verlangsamt. Beinhaltet die *collection* 1 Millionen Dokumente, wäre die Abfrage kaum noch nutzbar. Damit auch große Datenmengen schnell durchsucht werden, benutzt man Indizes.

3.1 Indexierung

Indexierung ist kein Konzept, das erst seit MongoDB exisiert. Im Grunde wird es von jedem DBMS umgesetzt. Die MongoDB zugrunde liegende Indexstruktur sind B-Bäume.

Bei B-Bäumen handelt es sich um eine graphartige Datenstruktur bestehend aus Knoten und gerichteten Kanten (siehe Figure 5). Ausgangspunkt ist der Wurzelknoten, der einen Schlüsselwert besitzt. Für den Schlüsselwert muss eine Ordnungsrelation definiert sein. In dem Beispiel hat der Wurzelknoten den Schlüsselwert 11 und die Ordnungsrelation $(\mathbb{N}, <)$. Der Schlüsselwert unterteilt den Suchbaum in zwei Unterbäume: im linken stehen alle Schlüsselwerte, die kleiner als 11 sind, im rechten die größeren. Sucht man beispielweise den Schlüsselwert 5, kann man den kompletten rechten Teilbaum ignorieren. Stattdessen untersucht man den nächsten Knoten im linken Teilbaum. Dieser hat zwei Schlüsselwerte, 3 und 7, die wiederum eine Aufteilung in drei Unterbäumen vornehmen. Gemäß der Ordnung wird der nächste Knoten im mittleren Teilbaum traversiert, der den gesuchten Schlüsselwert 5 enthält. Somit muss lediglich ein Pfad im Baum anstelle von allen Werten durchsucht werden [12].

In Figure 5 ist der Suchbaum ausbalanciert, d. h. alle Pfade vom Wurzelknoten zu einem Blattknoten sind gleich lang. Wäre der Suchbaum extrem einseitig aufgebaut oder wären alle Schlüsselwerte in einem Knoten enthalten, hätte man - je nach gesuchtem Schlüsselwert - keine Beschleunigung der Suche, da im Extremfall nach wie vor alle Knoten traversiert werden müssten. Aus diesem Grund wird auf einem B-Baum eine Ordnung definiert. Die Ordnung legt fest, wie viele Schlüsselwerte in einem Knoten enthalten sein dürfen: Bei einer Ordnung von m müssen in jedem Knoten,

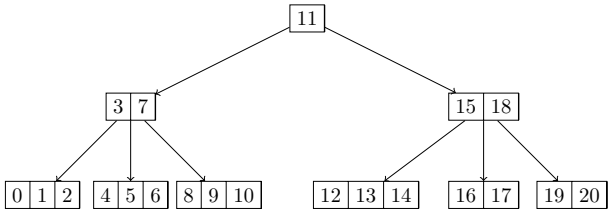

Figure 5: B-Baum der Ordnung 2 [14]

außer dem Wurzelknoten, mindestens m und maximal $2m$ Schlüsselwerte enthalten sein. Droht die Ordnung durch eine Einfüge- bzw. Lösch-Operation verletzt zu werden, lässt sich die Ordnung durch eine Überlauf- bzw. Unterlauf-Behandlung wiederherstellen [12]. Aus diesem Grund sollte ein Index auch nur auf Felder definiert werden, bei denen es nötig ist. Die Verwaltung der Indexstruktur nimmt nämlich Ressourcen in Anspruch, besonders wenn Überlauf- oder Unterlauf-Behandlungen durchgeführt werden müssen.

Ein Index wurde in Kapitel 2 bereits angesprochen: Für das Feld `_id` wird automatisch eine Indexstruktur angelegt. Grundsätzlich lassen sich Indizes für beliebige Felder definieren. Neben Einzelfelder-Indizes werden auch andere Indextypen unterstützt. Darunter Mehrschlüssel-Indizes für die Indexierung von Arrayelementen, aber auch spezielle Geo-Indizes, die eine effektive Distanzberechnung mit Koordinaten erlauben [4]. Bei einem Blick in die *Changelogs* stellt man jedoch fest, dass z. B. partielle Indizes in MongoDB erst seit Dezember 2015 in der Version 3.2 untersützt werden [9]. PostgreSQL hingegen unterstützt partielle Indizes bereits seit Februar 2002 [7].

Anhand dieses kleinen Beispiels werden zwei Dinge deutlich: Zum einen, dass MongoDB noch recht jung ist. Zum anderen, dass die Entwickler von MongoDB bemüht sind, fehlende Features nachzuliefern.

3.2 Mapreduce

Indexierung alleine hebt MongoDB nicht von anderen DBMS ab. Seine wahren Stärken spielt es in der verteilten Andwendung aus (Kapitel 5). Ein wichtiges Konzept, um unter diesen Umständen Abfragen schnell verarbeiten zu können, ist *Mapreduce*.

Das Mapreduce Verfahren wurde 2004 von Google veröffentlicht. Es handelt sich dabei um algorithmisches Konzept, das die parallele Ausführung von Abfragen auf verschiedenen Maschinen erlaubt [13]. Namesgebend für die Abfragen sind deren Hauptbestandteile: die *map* und die *reduce* Funktion. Die Map-Funktion erhält als Eingabe ein Schlüssel/Wert Paar und erzeugt ein Zwischenergebnis, wiederum aus Schlüssel/Wert Paaren bestehend. Anschließend wird das Zwischenergebnis von der Reduce-Funktion zusammengefasst [11].

Die Abfrage in Figure 6 liefert die Gesamtzahl der Räume. Das *mappen* erfolgt durch die Funktion `emit()`, die für jedes Gebäude die Anzahl der Räume ausgibt. Die Reduce-Funktion verarbeitet die Zwischenergebnisse und summiert die Werte.

Im Gegensatz zu normalen aggregierten Abfragen, kann diese auf mehreren Knoten gleichzeitig ausgeführt werden. Bei besonders großen Datenmengen lässt sich dadurch die Rechenlast auf mehrere Maschinen verteilen. Man spricht dabei auch von horizontaler Skalierung.

```
db.buildings.mapReduce(
  // map
  function() { emit('rooms', { count: this
      .rooms.length }); },
  // reduce
  function(key, values) {
    var amount = { count: 0 };
    values.forEach(function(value) {
      amount.count += value.count;
    });
    return amount;
  },
  { out: { inline: 1 }}
)
```

Figure 6: Mapreduce Funktion

4. SCHEMA DESIGN

In Kapitel 2 wurde bereits erläutert, dass MongoDB schemafrei ist. Dennoch ist es wichtig, sich ein Konzept für die Strukturierung der Daten zu überlegen, da diese Entscheidung Auswirkungen auf Performance, Erweiterbarkeit, Konsistenz und Vergleichbarkeit hat.

4.1 Eingebettete vs. normalisierte Datenmodelle

Relationale DBMS verwenden das normalisierte Datenmodell. Das Modell beschreibt Kriterien, unter denen Attribute in Tabellen aufgeteilt werden. Ziel der Aufteilung ist es, Redundanzen zu minimieren. Die Beziehungen zwischen einzelnen Tupeln können durch Identifizierungs- und Fremdschlüssel realisiert werden. Die zentrale Operation, um Daten zu kombinieren, ist der natürliche Verbund. Er kann unter Angabe eines Verbundattributs Tupel zusammenführen, die den gleichen Attributwert haben. Somit lassen sich Daten nahezu beliebig kombinieren. Darüber hinaus sind dadurch neben $1:n$ auch $n:m$ Beziehungen möglich.

Nachteil des natürlichen Verbundes ist der Aufwand. Im Idealfall erfolgt die Verbundberechnung zwischen einem Fremdschlüssel und einem indizierten Primärschlüssel. In diesem Fall beträgt der Aufwand $\mathcal{O}(m \cdot log(n))$. Zu bedenken ist, dass die erste Tabelle komplett in den Hauptspeicher geladen werden muss. Problematisch wird dies bei extrem großen Datenmengen oder einem Mehrwegeverbund [12]. Aus diesem Grund verzichtet MongoDB auf Verbundoperationen.

Neben dem normalisierten Datenmodell lässt sich in MongoDB auch das eingebettete Datenmodell umsetzen. Letzteres bietet sich vor allem bei $1:n$ Beziehung an und wird in dem Beispiel angewandt: Statt eine weitere *collecion* für die Räume anzulegen, werden diese in das Gebäude-Dokument aufgenommen (siehe Figure 1). Der Vorteil des Modells ist einleuchtend: zusammengehörige Daten werden gemeinsam gespeichert. Ist ein Dokument einmal im Hauptspeicher geladen, sind sämtliche Informationen zu dem repräsentieren Objekt ohne Verbundoperationen oder Referenzen verfügbar. Der Nachteil offenbart sich jedoch bei $n:m$ Beziehungen: Objekte werden mehrfach repräsentiert. Diese Redundanz erhöht nicht nur den Speicheraufwand, sondern auch das Risiko für inkonsistente Daten. Es gilt daher genau abzuwägen, ob man Redundanzen in Kauf nehmen möchte.

Werden die Redundanzen durch das eingebette Datenmodell zu groß, sollte es zumindest zu einem gewissen Teil normalisiert werden. Beziehungen zwischen Dokumenten können mit Hilfe von Referenzen auf das Feld `_id` realisiert werden. Im Gegensatz zu Fremdschlüsseln bei relationalen DBMS sichert eine solche Referenz keine Integrität. Darüber hinaus muss für die Auflösung einer Referenz eine weitere Abfrage durchgeführt werden. Unter Umständen verschlechtert sich so die Performance gegenüber dem eingebetteten Datenmodell [3].

4.2 Transaktionen

Eine Transaktion ist eine Änderungsoperation auf einer Datenbank, die atomar, konsistent, isoliert und dauerhaft ist. Bekannt sind diese Eigenschaften auch unter der Abkürzung *ACID (Atomicity, Consistency, Isolation, Durability)*:

- *Atomicity*: Alle Operationen einer Transaktion werden durchgeführt oder gar keine.
- *Consistency*: Wenn sich das System in einem konsitenten Zustand befindet, ist auch der Folgezustand der Transaktion konsistent.
- *Isolation*: Zwei nebenläufige Transaktionen beeinflussen sich nicht gegenseitig, d. h. das Ergebnis ist das gleiche, wie bei einer sequenziellen Ausführung.
- *Durability*: Das Ergebnis einer erfolgreichen Transaktion bleibt dauerhaft gespeichert [15].

Transaktionen sind besonders bei kritischen Änderungsoperationen wichtig. Während relationale DBMS wie PostgreSQL Transaktionen ermöglichen, gibt es bei MongoDB keinen Mechanismus, der die *ACID*-Kriterien erfüllt. MongoDB gewährleistet lediglich Atomarität bei Änderungsoperationen auf Ebene eines einzelnen Dokuments. Aus diesem Grund befindet sich das Thema Transaktionen im Kapitel Schema Design: Verwendet man das eingebettete Datenmodell, kann eine Änderungsoperation auf die Subdokumente atomar durchgeführt werden. Normalisiert man hingegen das Modell, geht dieser Vorteil verloren.

Wer MongoDB verwenden möchte, sollte seinen Anwendungskontext genau kennen. Durch das eingebettete Datenmodell lassen vor allem hierarchische Strukturen gut abbilden. Stehen die Objekte in Beziehung, unterstellt MongoDB keine Integritätsbedingungen, wie es relationale DBMS machen. Bei stark verknüpften Objekten ist ein graphenorientiertes DBMS vielleicht die bessere Wahl. Darüber hinaus ist eine strikte Normalisierung nicht zu empfehlen, sodass man bis zu einem gewissen Grad mit Redundanzen leben muss.

5. VERTEILTE ANWENDUNG

5.1 Replikation

In vielen Anwendungsfällen wünscht man sich eine hohe Verfügbarkeit der Daten. Diese kann durch Replikation erreicht werden. Mit relativ geringen Konfigurationsaufwand lassen sich mehrere Instanzen als repliziertes Set starten. Eine Instanz nimmt die Rolle des primären Servers an, während die anderen als sekundäre Server im Hintergrund laufen. In der Standard-Konfiguration gehen alle Lese- und Schreibzugriffe an den primären Server. Bevor eine Schreiboperation als erfolgreich angesehen wird, muss sie von der Mehrheit der Instanzen übernommen worden sein. Auf diese Weise wird eine hohe Kosistenz der Schreibzugriffe erreicht [1, 13].

Kommt es dazu, dass der primäre Server ausfällt, wird die Rolle von einem sekundären Server übernommen. Dieser muss von der Mehrheit der laufenden Instanzen gewählt werden. Aus diesem Grund verwendet man immer eine ungerade Anzahl an Instanzen, da beispielsweise je zwei durch einen Netzwerkfehler voneinanderer getrennte Server nicht beschlussfähig wären [13]. Aufgrund der Zeitstempel in den Dokumenten (siehe Kapitel 2.1), kennt MongoDB immer den aktuellesten Wert. Somit ist der neue primäre Server auf dem gleichen Stand wie sein Vorgänger.

MongoDB erlaubt für ein repliziertes Set eine Konfiguration, die Lesezugriffe auch auf den Replikaten erlaubt. Schreibzugriffe werden nach wie vor auf dem primären Server durchgeführt. Der Vorteil ist klar: der primäre Server wird entlastet, da er weniger Lesezugriffe erhält. Der Durchsatz kann somit erhöht bzw. die Latenz verringert werden. Der Nachteil ist, dass Clients Änderungen von Dokumenten lesen können, bevor der Schreibvorgang bestätigt wurde. Entsprechend können auch Daten gelesenen werden, die durch ein Rollback wieder verworfen werden. Man kann daher lediglich von einer schwachen Kosistenz sprechen [1].

5.2 Sharding

Relationale DBMS skalieren vertikal. Das heißt, die Größe der Datenbank wird durch die Hardware des einzelnen Servers begrenzt. MongoDB hingegen unterstützt *Sharding*. Dabei handelt es sich um eine Technik, die es erlaubt eine *collection* auf mehrere Server zu verteilen. Dadurch wird die Last auf mehrere Server aufgeteilt. Die Partitionierung wird anhand des Werts oder dem Hashwert eines Feldes vorgenommen [6]. Bei der Stadtverwaltung ließe sich *Sharding* z. B. anhand des Stadtnamens durchführen. Somit wären alle Gebäude einer Stadt auf einem einzelnen Server gespeichert.

Um *Sharding* umzusetzen, bedarf es zusätzlicher Mongo-Instanzen: Zum einen wird ein Konfigurations-Server benötigt, der Metadaten speichert und für die Partitionierung sorgt. Zum anderen wird ein Router `mongos` benötigt, mit dem sich die Clients verbinden können. Er leitet die Abfragen an den jeweiligen Server weiter. Da er nahezu die gleichen Funktionen wie eine `mongod` Instanz bereitstellt, ist er für den Client transparent [13].

6. FAZIT

Eigenschaft	MongoDB	PostgreSQL
Datenmodell	dokumentorientiert	relational
Schema	schemafrei	vordefiniert
Abfragesprache	JavaScript	SQL
Verbundoperation	nein	ja
MapReduce	ja	nein
Transaktionen	nein	ja
Sharding	ja	nein
Replikation	ja	ja
Skalierung	horizontal	vertikal

Table 2: Vergleich MongoDB und PostgreSQL

MongoDB ist in jeglicher Hinsicht darauf optimiert große Datenmengen zu verwalten, ohne Performance einzubüßen.

Das hat seinen Preis: Konzepte wie Normaliserung und Transaktionen werden über Bord geworfen. Durch das schemafreie Design lässt sich das Datenmodell im Projektverlauf zwar leicht anpassen, allerdings erhöht dies auch das Risiko für Anomalien. Die Integrität der Daten muss daher auf Anwendungsseite sichergestellt werden. Da MongoDB für den Einsatz auf mehreren Servern konzipiert ist, existieren zahlreiche Konfigurationsmöglichkeiten für die verteilte Anwendung. Replikation und *Sharding* stellen sicher, dass das System horizontal skaliert und hochverfügbar ist.

7. REFERENCES

[1] Mongodb, 12 2015. https://docs.mongodb.org/v3.0/core/read-preference/.

[2] Mongodb: About, 11 2015. https://www.mongodb.com/company.

[3] Mongodb: Database references, 12 2015. https://docs.mongodb.org/manual/reference/database-references/.

[4] Mongodb: Index introduction, 12 2015. https://docs.mongodb.org/manual/core/indexes-introduction/.

[5] Mongodb: Objectid, 12 2015. https://docs.mongodb.org/manual/reference/object-id/.

[6] Mongodb: Read preference, 12 2015. https://docs.mongodb.org/manual/core/sharding-introduction/.

[7] Postgresql release notes 7.2, 12 2015. http://www.postgresql.org/docs/current/static/release-7-2.html.

[8] Mongodb: Bson types, 1 2016. https://docs.mongodb.org/manual/reference/bson-types/.

[9] Release notes for mongodb 3.2, 1 2016. https://docs.mongodb.org/manual/release-notes/3.2/.

[10] Sql to aggregation mapping chart, 1 2016. https://docs.mongodb.org/manual/reference/sql-aggregation-comparison/.

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. Technical report, Google Inc., 2004. https://static.googleusercontent.com/media/research.google.com/de//archive/mapreduce-osdi04.pdf.

[12] U. Kelter. *Datenbanksysteme I.* Uni Siegen, 12 2013.

[13] E. Redmond and J. R. Wilson. *Sieben Wochen, sieben Datenbanken - Moderne Datenbanken und die NoSQL-Bewegung.* O'Reilly Verlag GmbH & Co. KG, 2012.

[14] M. Thoma. B-baum (angepasst), 12 2015. https://github.com/MartinThoma/LaTeX-examples/blob/master/tikz/b-tree/b-tree-2.tex.

[15] R. Wismüller. Verteilte systeme: Koordination, 2015. http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/ss15/vs/v07.pdf.