# BACHELORARBEIT

zur Erlangung des akademischen Grades
„Bachelor of Science in Engineering“ im Studiengang BIC6

# Performancevergleich von relationalen Datenbanken und NoSQL Document

Ausgeführt von: Markus Hösel
Personenkennzeichen: 121025800

1. BegutachterIn: Priv. Doz. Dipl.-Ing. Dr.techn. Karl M. Göschka

Wien, 19. Mai 2015

FACHHOCHSCHULE
TECHNIKUM WIEN

# Eidesstattliche Erklärung

„Ich, als Autor / als Autorin und Urheber / Urheberin der vorliegenden Arbeit, bestätige mit meiner Unterschrift die Kenntnisnahme der einschlägigen urheber- und hochschulrechtlichen Bestimmungen (vgl. etwa §§ 21, 46 und 57 UrhG idgF sowie § 11 Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien).

Ich erkläre insbesondere korrekt fremde Inhalte, gleich welcher Form, übernommen zu haben und bin mir bei Nachweis fehlender Eigen- und Selbstständigkeit sowie dem Nachweis eines Vorsatzes zur Erschleichung einer positiven Beurteilung dieser Arbeit der Konsequenzen bewusst, die von der Studiengangsleitung ausgesprochen werden können (vgl. § 11 Abs. 1 Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien).

Weiters bestätige ich, dass ich die vorliegende Arbeit bis dato nicht veröffentlicht und weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt habe. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht.“

# Kurzfassung

Um den Anforderungen an massive Skalierbarkeit bezüglich paralleler Lese- und Schreibzugriffe gerecht zu werden, wurden in den letzten Jahren neue Datenbanksysteme entwickelt. Diese Arbeit beleuchtet Document Stores, ein Datenbanktyp welcher immer häufiger für Big Data Applikationen verwendet wird, und vergleicht diese mit den traditionellen relationalen Datenbanken. Es werden die Vor- und Nachteile beider Typen beschrieben und Bezug zu den verwendeten Datenmodellen genommen. Diese sind referenced Documents und embedded Documents für die Document Stores und normalisierte und denormalisierte Relationen für die relationale Datenbank. Es wird weiters auf Transaktionen eingegangen und das Two Phase Commit Protocol clientseitig für all jene Datenbanken implementiert, welche nicht transaktionsorientiert sind. Anschließend wird in einem Testenvironment eine Beispielanwendung genutzt um das Latenzverhalten der einzelnen Datenbanksysteme zu messen. Dabei werden MongoDB und Couchbase als Repräsentanten für die Document Stores und MySQL in Kombination mit InnoDB, MyISAM und NDB als Storage-Engine verwendet. Die Beispielanwendung simuliert eine stark genutzte Blogging Plattform in welcher User Blogs und Kommentare lesen und schreiben und weiters die Kommentare mittels einen "Like" Buttons bewerten können. Die Ergebnisse zeigen, dass das embedded Document Datenmodell als auch die denormalisierte relationale Datenbank bessere Performancewerte bei der Untersuchung von Leseoperationen erzielen als das referenced Document Datenmodell und die normalisierte relationale Datenbank, weisen jedoch wesentlich schlechtere Performancewerte bei Schreiboperationen auf. Die normalisierte relationale Datenbank zeigt das schlechteste Performanceverhalten. Weiters müssen bei der denormalisierten relationalen Datenbank für die Verwaltung jedes einzelnen Blogs für welchen Kommentare existieren, redundante Informationen gespeichert und verwaltet werden, was zu Inkonsistenzen und Datenanomalien führen kann. Da die Blogging Plattform keine unbedingte Notwendigkeit für die Durchführung von Transkationen hat, zeigt sich der Document Store mit einer Kombination des referenced Document Datenmodells und des embedded Document Datenmodells als beste Lösung für die Realisierung einer solchen Big Data Anwendung, wodurch diese Arbeit den positiven Einfluss von Document Stores auf den derzeitigen Datenbankmarkt bestätigt.

**Schlagwörter:** Big Data; Couchbase; Database System; Document Stores; InnoDB; MongoDB; MyISAM; MySQL; NDB; Relational Database

# Abstract

To satisfy the needs of massive scalability regarding to parallel read and write queries new database systems have evolved. This paper introduces document stores and compares them with traditional relational databases. It outlines the pros and cons of both system types and gives a comparison on the most common data models, which are the embedded and referenced document model for the document stores and normalized and denormalized table model for the relational databases. This paper also discusses transactions and how to implement them in the application layer in case the database does not have a support for transactions. A test environment is set up and a sample application is used in order to measure the response time of the systems. MongoDB and Couchbase are used as representatives for document stores and MySQL in combination with InnoDB, MyISAM and NDB as storage engines is used as a representative for relational databases. The sample application mimics a heavily used blogging platform with users writing and reading blog entries and comments and making use of a “Like” button. The results show that the embedded document data model as well as the denormalized table data model are much more responsive to read queries then the referenced document data model and the normalized table model, but with a serious performance bottleneck on write queries. The normalized table data model shows the worst performance. Furthermore the denormalized table model has to store the same information multiple times, which can lead to inconsistent data. Because the blogging platform does not have the requirement to use transactions, the best database solution for such a big data application is the document store with a combination of the embedded and referenced data model. Therefore the paper confirms the positive impact of document stores to the current database market.



**Keywords:** Big Data; Couchbase; Database System; Document Stores; InnoDB; MongoDB; MyISAM; MySQL; NDB; Relational Database

# Danksagung

Bedanken möchte ich mich bei meinem Betreuer, Herrn Karl M. Göschka, für seine thematisch wertvollen Ratschläge als auch für seine engagierte Unterstützung zu kritischen Fragestellungen.

Des Weiteren auch ein Dankeschön an die Couchbase- und Stackoverflow Community, die mir innerhalb kürzester Zeit zu Fragen bezüglich der N1QL Query Language ausführliche Antworten geliefert haben. Für Details dieser Fragen sei auf [35], [36] und [37] verwiesen.

# Inhaltsverzeichnis

# 1 Einleitung

Die Wahl einer Datenbank und des Datenmodells sind einer der ersten Punkte beim Entwerfen einer neuen Applikation. Bei Auswahl einer relationalen Datenbank wird das Datenmodell mittels des Normalisierungsprozesses formuliert, welcher als Ziel hat Datenredundanzen zu eliminieren. Durch diese Eliminierung von Datenredundanzen können Änderungen leicht vollzogen werden, da die Information an nur einer Stelle zu ändern ist. Bei diesem Modell kann es jedoch problematisch werden Informationen aus dem System abzufragen, vor allem wenn Verbundoperationen notwendig werden. Denn abhängig von der Verbundoperation sind multiple Festplattenzugriffe notwendig, wozu bei magnetischen Festplatten sogenannte Seeks notwendig sind. Diese Seeks sind in Verhältnis zu den anderen Operationen, welche die für die Ausführung notwendig sind, extrem zeitaufwendig und sollten vermieden werden. Solid State Disks sind zwar performanter, aber auch hier summieren sich schließlich die einzelnen Zugriffe und können das Ergebnis drastisch verschlechtern. Eine Vermeidung der Festplattenzugriffe kann jedoch nur durch eine Denormalisierung des Datenmodells erreicht werden, womit Redundanzen bewusst in Kauf genommen werden. Im Gegensatz dazu können Document Stores einen zusammenhängenden Datensatz in einem Dokument festhalten ohne Daten redundant speichern zu müssen. Jedes dieser Dokumente wird auf der Festplatte als eine Einheit gespeichert, so dass für eine Lese- bzw. Schreiboperation nur ein Festplattenzugriff notwendig ist. Aber auch das Speichern sämtlicher Informationen zu einem Sachverhalt in nur einem Dokument ist problematisch. Sobald nämlich diese Informationen in einer anderen Struktur benötigt werden, muss diese umgestaltet werden. Des Weiteren können einzelne Dokumente sehr groß werden, was zu einer sehr hohen Belastung des Arbeitsspeichers führen kann. Höhere Flexibilität bieten hier die referenced Documents, bei denen ein Datensatz auf mehrere Dokumente verteilt wird und die Beziehungen mit Referenzen festgehalten werden. Jedoch werden bei den referenced Documents wie bereits beim normalisierten Datenmodell ebenso mehrere Zugriffe auf die Festplatte notwendig, was die Performance verschlechtert. Diese Arbeit hat sich daher als Ziel genommen die vier Datenmodelle zu untersuchen und deren Auswirkungen durch Messungen festzuhalten und zu veranschaulichen.

## 1.1 Problem- und Aufgabenstellung

Diese Arbeit vergleicht die Performance unterschiedlicher Datenmodellierungen von relationalen Datenbanken und Document Stores. Dafür wird ein Testenvironment bestehend aus den drei Datenbanksysteme MySQL, MongoDB und Couchbase sowohl als Standalone Systeme als auch als Cluster aufgesetzt. Diese Datenbanken werden anschließend Messungen unterzogen in welchen die Latenzzeiten der Systeme festgehalten werden. Die Messungen umfassen dabei Schreib- und Leseoperationen. Weiters werden Transaktionen ausgeführt, wobei die Implementierung der Transaktionen entweder in der Datenbank

gegeben ist oder in der Applikation mittels des Two Phase Commit Protokolls erfolgt. Die Ergebnisse werden grafisch und tabellarisch veranschaulicht und diskutiert. Die Wahl der Datenbanken fiel auf MySQL, MongoDB und Couchbase, da diese derzeit nach [11] die populärsten Open Source Vertreter mit Clusterfunktionalität in den Kategorien relationale Datenbanken [34] und Document Stores [38] sind.

## 1.2 Inhalt und Aufbau der Arbeit

Im zweiten Kapitel werden die Aufgaben und die Architektur von Datenbanksystemen betrachtet und auf die beiden Datenbankmodelle Relationale Datenbanken und Document Stores eingegangen. Kapitel 3 beschreibt die für die Messungen herangezogenen Datenbanksysteme MySQL, MongoDB und Couchbase. Anschließend werden im vierten Kapitel die Testapplikation und die Datenmodellierung beschrieben. Im Kapitel 5 werden die Benchmarks durchgeführt und die Messergebnisse grafisch als auch tabellarisch dargestellt und diskutiert. Ähnliche wissenschaftliche Arbeiten werden in Kapitel 6 verglichen. Abschließend erfolgen im Kapitel 7 eine Zusammenfassung und ein Ausblick auf mögliche weitere Tätigkeiten in dem Themengebiet dieser Arbeit.

# 2 Datenbanksysteme

Ein Datenbanksystem (DBS) bezeichnet die Kombination von einer Datenbank mit einem Datenbankmanagementsystem (DBMS). Die Datenbank selbst stellt dabei den Datenbestand dar. Die Zugriffe auf diesen Datenbestand werden von dem DBMS verwaltet.

## 2.1 Aufgaben eines DBMS

Das DBMS stellt die Systemsoftware eines DBS dar und erfüllt eine Reihe an Aufgaben um mit einem Datenbestand in einer Datenbank zu arbeiten.
Die Anforderungen welche bei der Entwicklung von DBS beachtet werden, können aufgrund des Einsatzzwecks sehr unterschiedlich sein. So gibt es unterschiedliche Isolationsebenen bei konkurrierenden Transaktionen um die Verfügbarkeit zu erhöhen, jedoch werden dadurch Konflikte wie Lost Updates, Dirty Read, Non-Repeatable Read oder Phantom Read ermöglicht. Die bewusste redundante Speicherung von Daten zwecks Performancegewinns bei bestimmten Leseoperationen ist ein weiteres Beispiel. Die Anforderungen von datenintensiven Applikationen wie Webseiten mit hohen Lastaufkommen führte weiters zu der Entwicklung der NoSQL Datenbanken, welche im Gegensatz zu den bisher dominierenden relationalen Datenbanksysteme nur schwache Konsistenz und eingeschränkte Transaktionsfähigkeit, dafür aber hohe Skalierbarkeit und geringe Latenzzeiten in einem Rechnerverbund bieten.

## 2.2 Datenbankmodelle

Die Architektur eines DBS wird durch eine externe-, konzeptuelle-, und interne Ebene beschrieben [6]. Dabei ist das auf der konzeptuellen Ebene zu realisierende Datenbankmodell die theoretische Grundlage eines DBS und beschreibt wie die Daten in einer Datenbank gespeichert und verarbeitet werden können. Ein Datenbankmodell wird in [8] durch drei Komponenten beschrieben:

1. Eine Menge an Datenstrukturtypen, die die Struktur einer Datenbank beschreibt.
2. Eine Menge an Operatoren oder Regeln, welche auf die in 1. definierte Struktur angewendet werden können um Daten entweder abzurufen oder abzuleiten.
3. Eine Menge an Integritätsbedingungen, welche entweder implizit oder explizit ein Set an konsistenten Datenbankzuständen oder Änderungen formulieren. Diese Bedingungen werden in einigen Fällen durch Insert-Update-Delete Regeln ausgedrückt.

Diese Arbeit widmet sich bei der Untersuchung der relationalen Datenbanken und der Document Stores zwei unterschiedlichen Datenbankmodellen, den relationalen Datenbankmodell und den dokumentenorientierten Datenbankmodell.

## 2.3 Relationale Datenbanken

Relationale Datenbanken wurden Ende der 1960er Jahre unter den Aspekt der relationalen Algebra entwickelt um die steigenden Datenmengen effizient verwalten zu können [9]. Ziel war es Probleme mit der Datenredundanz, Datenintegration und physischen Abhängigkeiten zur darunterliegenden Hardware zu lösen.
Sie sind heutzutage die meist genutzten Datenbanksysteme. Die populärsten Vertreter sind nach [34] Oracle, MySQL und Microsoft SQL Server. Diese Arbeit wird Bezug zu MySQL nehmen, und Untersuchungen mit den Storage Engines MyISAM, InnoDB und NDB durchführen.

### 2.3.1 Datenmodell

Eine relationale Datenbank verwaltet die Daten in Relationen, welche für den Anwender anschaulich als Tabelle verstanden werden kann. Jede Relation verfügt über ein Relationenschema, welches durch die Attribute (= Spaltendefinition der Tabelle) definiert wird, und beinhaltet eine unbestimmte Anzahl an Tupeln (= Zeilen der Tabelle), die die Datensätze abbilden. Um Datensätze eindeutig identifizieren zu können wird ein Primärschlüssel verwendet. Ein Primärschlüssel setzt sich dabei aus einen oder mehreren Attributen der Relation zusammen und beinhaltet eine Wertkombination welche nur einmalig auftreten darf. Relationen können miteinander in Beziehung stehen. Das relationale Modell kategorisiert diese Beziehungen als 1:1, 1:n und m:n Beziehungen. Die Verbindung

zwischen den Relationen wird durch Wertegleichheit hergestellt. Dabei wird in einer Relation ein Fremdschlüssel definiert, welcher auf den Primärschlüssel einer anderen Relation verweist. Das Prinzip wird in Abbildung 1 veranschaulicht.

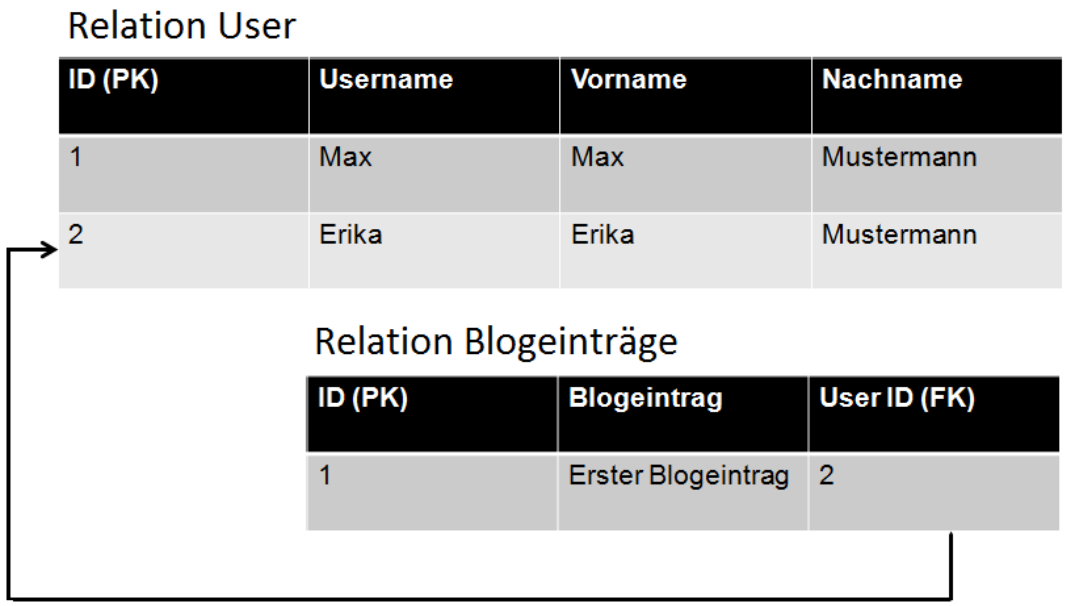



Abbildung 1: Beziehung zwischen Relationen

Den Anwender stehen unter dem Aspekt, dass dieser mit den Beziehungen zwischen den Relationen vertraut ist, zahlreiche Wege offen die Daten abzufragen. So können sowohl Daten abgefragt werden die direkt in Beziehung stehen, als auch Daten die indirekt (durch weitere Relationen) in Beziehung stehen. Die Datenabfrage erfolgt dabei durch eine Datenbanksprache, auf welche im folgenden Unterkapitel eingegangen wird.

### 2.3.2 Datenbanksprache

Eine Datenbanksprache bezeichnet eine formale Sprache für die Interaktion mit einem DBMS und der Datenbank. Bei relationalen Datenbanken kommt die Datenbanksprache SQL (Structured Query Language) zum Einsatz. SQL ist eine standardisierte Datenbanksprache, die letzte Veröffentlichung ist der SQL:2011 Standard [13]. Durch den Einsatz einer standardisierten Sprache wird die Unabhängigkeit vom darunterliegenden System aus Sicht der Anwendung angestrebt. Obwohl SQL standardisiert ist, wird die Sprache von den verschiedenen Datenbankherstellern in voneinander abweichenden Varianten implementiert. Vorliegende Arbeit verwendet nur Sprachelemente, welche im SQL Standard definiert sind.

### 2.3.3 Datenmodellierung

Die Datenstrukturen werden üblicherweise von ER-Modellen oder UML-Modellen gewonnen und unter Berücksichtigung der Normalisierungsregeln auf mehrere Relationen aufgeteilt.

**Normalisierung**

Dieses Kapitel definiert die ersten drei Normalformen. Für den weiteren Verlauf der Arbeit werden Relationen der 1. Normalform als auch Relationen der 3. Normalform verwendet.

**1. Normalform**

Eine Relation ist in erster Normalform, wenn alle Attribute nur atomare Werte besitzen und die Relation frei von Wiederholungsgruppen ist [15].

**2. Normalform**

Eine Relation ist in der zweiten Normalform, wenn sie sich in der erste Normalform befindet und jedes Nichtschlüsselattribut funktional abhängig vom Gesamtschlüssel, jedoch nicht von einer echten Teilmenge der Schlüsselkandidaten ist [14].

**3. Normalform**

Eine Relation ist in der dritten Normalform, wenn sie sich in der zweiten Normalform befindet und kein Nichtschlüsselattribut von einem Schlüsselkandidaten transitiv abhängig ist [14].

#### 2.3.3.1 Denormalisierung

Unter bestimmten Umständen wird eine bewusste Rücknahme der Normalisierung durchgeführt. Die Denormalisierung kann für eine höhere Administrierbarkeit der gespeicherten Daten oder aber auch für die Verringerung der Komplexität eines Systems erfolgen [18]. Meist jedoch wird eine Denormalisierung unter den Aspekt eines verbesserten Laufzeitverhaltens durchgeführt. Dies tritt vor allem bei Datenbanksystemen ein, welche häufig Verbundoperationen für Lesezugriffe durchführen müssen. Aufgrund der redundant gehaltenen Daten besteht jedoch die Gefahr der Datenanomalien, ebenso ist das Laufzeitverhalten eines Schreibvorgangs deutlich schlechter als bei normalisierten Datenbeständen. Es ist daher genau zu prüfen ob die auf der Datenbank operierenden Anwendungen von einer Denormalisierung profitieren können.

## 2.4 Document Stores

### 2.4.1 NoSQL

Document Stores sind eine Unterkategorie der sogenannten NoSQL Datenbanksystemen. NoSQL steht dabei für Not Only SQL, und das Designprinzip dieser DBS basiert auf Technologien die nicht-relationale Strukturen verfolgen. Ein wesentlicher Grund für die

Entwicklung der NoSQL Systeme sind die Skalierungsprobleme von relationalen Datenbanken in verteilten Systemen. Vor allem der Anbruch des Web 2.0 und der vielfältigen Beschaffenheit unstrukturierter Daten als auch die benötigte Flexibilität hinsichtlich Scaling, Design, Kosten und Disaster Recovery lies Ende der 90er Jahre die NoSQL Systeme entstehen, deren Fokus auf Verfügbarkeit, Zuverlässigkeit und (horizontaler) Skalierbarkeit liegt [19].
NoSQL Systeme lassen sich in die vier Gruppen Key Value Stores, Column Family Stores, Document Stores und Graph Databases kategorisieren. Im weiteren Verlauf der Arbeit wird auf die Document Stores und deren populärsten Vertreter MongoDB und Couchbase eingegangen.

### 2.4.2 Allgemeines zu Document Stores

Document Stores unterscheiden sich vor allem durch den Wegfall eines fest definierten Schemas zu relationalen Datenbanken. Im Mittelpunkt steht das Konzept des Dokuments. Die Daten werden dabei in den für den Menschen lesbaren und für den Computer interpretierbaren Standard-Dateiformaten wie JSON (JavaScript Object Notation), XML (Extensible Markup Language) oder YAML (YAML Ain't Markup Language) abgebildet [20].

### 2.4.3 Datenmodell

Dieses Kapitel veranschaulicht das Dokument anhand des Dateiformats JSON, welches das Dateiformat für die in dieser Arbeit genutzten Document Stores ist. Das Konzept des Dokuments ermöglicht es ohne Vorgabe eines starren Schemas Daten zu speichern. Anstatt die einzelnen Datensätze in Tabellen einzufügen, welche immer die gleiche Menge an Spalten haben und in welchen jedes Feld einer Spalte einen festgelegten primitiven Datentyp und Definitionsbereich besitzt, wird bei Document Stores Schemafreiheit verfolgt. Attribute können nach Belieben hinzugefügt oder weggenommen werden. Damit entfällt zwar die Vorgabe eines Schemas nicht, verlagert sich jedoch von der Datenbank hin zur Applikation bzw. zur Middleware [23]. Abbildung 2 zeigt ein Dokument im JSON Dateiformat.

```
// User Document:
{
  "_id": 1,
  “username": “markus",
  “plz”:1200,
  “ort”:”wien”
}
```

Abbildung 2: Einfaches JSON Dokument

Das Dokument zeigt ein einfaches Objekt, welches durch die öffnende Klammer "{" beginnt und durch die schließende Klammer "}" endet. Das Objekt enthält weiters eine durch

Kommata geteilte, ungeordnete Liste von Eigenschaften. Jede dieser Eigenschaften besteht aus einem Schlüssel (den Attribut) und einen Wert. Beim Schlüssel handelt es sich um eine Zeichenkette. Als Wert sind folgende Datentypen zulässig: Nullwert, Boolescher Wert, Zahl, Zeichenkette, Array, Objekt. In obiger Abbildung haben die beiden Eigenschaft mit den Schlüsseln "_id" und "plz" Zahlen als Werte, die beiden Eigenschaften mit den Schlüsseln "username" und "ort" haben Zeichenketten als Werte [24].
In Abbildung 3 wird gezeigt, wie eine Liste als Wert definiert werden kann. Der bei der Eigenschaft mit dem Schlüssel "e-mail" gezeigte Wert besteht aus einer Liste mit einer unbestimmten Anzahl an weiteren Objekten. Jedes dieser Objekte setzt sich wiederum als Schlüssel-Wert Paar zusammen, wie bei den einzelnen Mailadressen in Abbildung 3 ersichtlich ist. In den Dokument mit der "_id":1 beinhaltet die Liste 2 Objekte, im Gegensatz dazu beinhaltet die Liste im Dokument mit der "_id_":2 nur ein Objekt. Es sind auch leere Listen zulässig. Solche verschachtelten Dokumente werden als embedded JSON Documents bezeichnet. Die theoretische Verschachtelungstiefe ist unendlich, wird aber in der Praxis aus Effizienzzwecken und der Wegnahme von Komplexität limitiert.

```
// User Documents
{
  "_id": 1,
  "username": "markus",
  "plz":1200,
  "ort":"wien",
  "e-mail": [
    { "firma": "markus@tech.at"},
    { "privat": "markus@abc.de"}
  ]
}
```

```
{
  "_id": 2,
  "username": "lisa",
  "plz":1200,
  "ort":"wien",
  "e-mail": [
    { "firma": "lisa@tech.at"}
  ]
}
```

Abbildung 3: Embedded JSON Documents

Eine weitere Möglichkeit JSON Dokumente zu modellieren bietet die Referenzierung. Dabei wird in einem Dokument ein Schlüssel-Wert Paar gespeichert, welches auf ein weiteres Dokument verweist. Der grundsätzliche Gedankengang ist dabei derselbe wie bei einer Verbundoperation von Relationen bei relationalen Datenbanken, weist jedoch in der jeweiligen Implementierung starke Unterschiede auf. So ist es zumindest bei den Document Stores welche für diese Arbeit herangezogen werden, nicht möglich transaktionsorientierte Operationen dokumentenübergreifend auszuüben.
Abbildung 4 veranschaulicht das Prinzip von referenzierenden Dokumenten. Dabei werden die Mail Adressen in eigenen Dokumenten festgehalten, welches zusätzlich eine Referenz auf das User Dokument beinhaltet. Aufgrund der Referenz ist es möglich alle Dokumente zu einen logisch zusammengehörenden Datensatz zu finden.

Diese Arbeit wird sowohl Datenbestände bestehend aus embedded Documents als auch Datenbestände bestehend aus referenced Documents untersuchen, und auf die Vor- und Nachteile beider Strukturen eingehen.

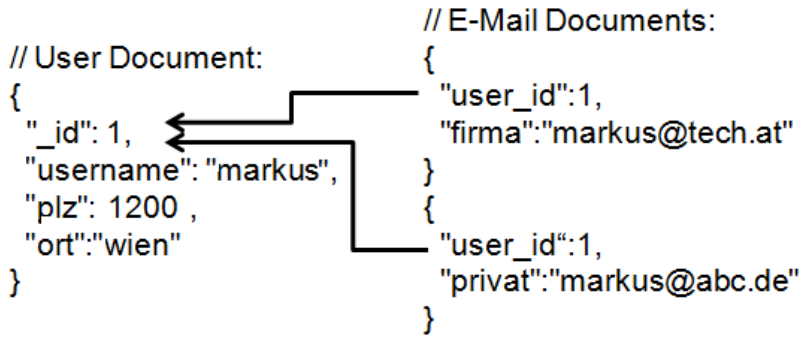



Abbildung 4: Referenced JSON Documents

### 2.4.4 Datenbanksprache

Document Stores bieten im Gegensatz zu relationalen Datenbanken keine standardisierte Datenbanksprache und die Entwicklungen der Datenbanksprachen zwischen den verschiedenen Herstellern weisen große Unterschiede auf. Ein Wechsel zwischen verschiedenen Datenbankimplementationen ist daher meist nur mit großem Aufwand verbunden. Zwar gibt es Ansätze einen gemeinsamen Standard zu entwerfen, zum Zeitpunkt dieser Arbeit befindet sich die Entwicklung dieser Ansätze jedoch nur in theoretischen Stadien. [25] zeigt wie ein gemeinsamer Standard gelingen kann.
Auf die einzelnen Implementierungen der verschiedenen Datenbanksprachen wird in den entsprechenden Unterkapiteln des Kapitels 3 eingegangen.

### 2.4.5 Datenmodellierung

#### Polymorphic Schema

Document Stores bieten die Möglichkeit, Dokumente in verschiedenen Containern zu verwalten. Diese Container haben abhängig von der jeweiligen Implementation verschiedene Namen. So werden diese bei MongoDB Collections und bei Couchbase Buckets genannt. Diese Container können hierarchisch mit den Tabellen einer relationalen Datenbank verglichen werden. Im Gegensatz zu den Tabellen sind die Container jedoch schemalos. Theoretisch ist es daher möglich alle Dokumente in einem Container zu verwalten. Ein gutes Design schreibt jedoch vor, dass ein Container nur Dokumente gleicher Art verwaltet. Für eine einfache Blogging Website könnte beispielsweise ein Container für die User Dokumente und ein weiterer Container für die Blogs angelegt werden. Werden mehrere ähnliche aber nicht idente Dokumente in einem Container verwaltet spricht man auch von einem Polymorphen Schema [26].

### Embedded oder Referenced

Wie im Kapitel 2.6.3 gezeigt, ist es möglich den gleichen Datenbestand in einem Dokument durch Verschachtelung oder durch mehrere referenzierende Dokumente abzubilden. Im Kontrast zu relationalen Datenbanken, in welchen der Weg zu einer normalisierten Datenbank mittels Ausführung der Normalisierungsregeln klar bestimmt ist, bietet die Entscheidung zu verschachteln oder zu referenzieren bei Document Stores mehr Spielraum. Folgende Regeln sollten nach [26] jedoch befolgt werden:

- Einbetten für Lokalität: Dokumente werden zusammenhängend auf der Festplatte gespeichert. Beim Abruf der Informationen ist daher bei Embedded Documents nur ein Zugriff notwendig, in Vergleich dazu müssen bei Referenced Documents mehrere Zugriffe auf die Festplatte ausgeführt werden um denselben Informationsgehalt gewinnen zu können. Abhängig von der Document Store Implementation ist weiters das Zusammenführen der Dokumente aus der Applikation bzw. der Middleware notwendig, was bedeutet, dass für jeden Dokumentenzugriff nicht nur die Zugriffszeit der Festplatte sondern auch die Latenzzeit der Netzwerkkommunikation hinzukommt. Es ist daher sinnvoll Informationen welche häufig gemeinsam abgerufen werden in einem Dokument zu verwalten.
- Einbetten für Atomarität und Isolation: Document Stores unterstützen keine dokumentenübergreifende Transaktionen. Um dennoch auf Datenbankebene Atomarität und Isolation bieten zu können ist die Verwendung von Embedded Documents zwingend. Alternativ kann applikationsseitig mittels eines Commit Protocol bzw. eines Concurrency Control Protocol Atomarität und Isolation realisiert werden.
- Referenzieren für Flexibilität: Obwohl grundsätzlich Embedded Documents besser performen und Datenkonsistenz garantiert wird, gibt es Fälle in denen Referenzierung zu einer besseren und effizienteren Handhabung beitragen. Vor allem bei Abfragen des Datenbestands können referenzierende Dokumente einen Vorteil haben. Wird ein Container nach Dokumenten durchsucht die eine bestimmte Eigenschaft haben, so wird jedes Dokument welches diese Eigenschaft erfüllt mit seinem gesamten Inhalt dem Aufrufer retourniert. Meist jedoch wird nicht der gesamte Inhalt benötigt und so folgt ein Parsen und Filtern dieser Informationen auf die wesentlichen Attribute. Teilt man jedoch solch ein Dokument bereits im Vorfeld in mehrere kleinere, referenzierende Dokumente auf, so können Abfragen dieser Art effizienter erfolgen und die applikationsseitigen Verarbeitungsschritte minimiert werden.
- Referenzieren bei beziehungsintensiven Dokumenten: Bei 1:n oder m:n Beziehungen zwischen Datensätzen mit hoher oder nicht voraussagbarer Stelligkeit eignen sich referenzierende Dokumente aus Gründen der Effizienz besser. Beispielsweise könnten auf einer Blogging Website Blogeinträge mit mehreren tausend

Kommentaren vorhanden sein. Würde es sich bei solch einen Blogeintrag um ein embedded Dokument handeln, so muss bei Zugriff das gesamte Dokument in den Arbeitsspeicher geladen werden. Wenn jedoch nur der Blogeintrag selbst und die ersten fünf Kommentare angezeigt werden, so befinden sich Daten im Arbeitsspeicher die nicht genutzt werden. Der Arbeitsspeicher jedoch ist eine sehr teure und kritische Ressource, und sobald dieser vollständig befüllt ist, werden Seitenfehler (engl. Page Faults) produziert, was zu Random Disk I/O führt. Es ist daher aus effizienzgründen sinnvoll, große Dokumente in mehrere kleinere referenzierende Dokumente aufzuteilen.

# 3 Untersuchung

Zur Durchführung verschiedener Tests werden drei populäre Open Source Datenbanken ausgewählt: MySQL, MongoDB und Couchbase. Diese werden sowohl als Single Systeme als auch als Cluster betrachtet und anschließend verglichen. Dieses Kapitel geht auf die Datenbanken näher ein und erklärt ihre Architektur und den Aufbau für die im anschließenden Kapitel beschriebenen Tests.

## 3.1 MySQL

MySQL ist eines der bekanntesten relationalen DBS und wird nach [34] auf Platz 2 der meist verwendeten DBS eingeordnet. Die Entwicklung von MySQL wurde 1994 im damaligen Unternehmen TcX begonnen. Es handelt sich dabei um ein Projekt, welches mSQL mit dem ISAM Handler des UNIREG DBMS koppelte um so die Funktionalität zu erweitern. 1995 erfolgte die Veröffentlichung der Version 1.0 unter der GPL Lizenz. Das Unternehmen wurde später in MySQL AB umbenannt und 2008 von Sun Microsystem gekauft, welches wiederrum 2010 von Oracle übernommen wurde. Das Softwarepaket wird permanent weiterentwickelt, die aktuelle Version 5.6.23 wurde am 2. Februar 2015 veröffentlicht [27].

### 3.1.1 Struktur

MySQL wurde nach dem Client-Server Pattern entworfen. Dabei nimmt ein MySQL Server bzw. ein Verbund an MySQL Servern Anfragen von MySQL Clients entgegen und beantwortet diese. MySQL bietet die Möglichkeit multiple Datenbanken zu verwalten und in jeder Datenbank können mehrere Tabellen angelegt werden. Beim Erzeugen der Tabelle kann weiters eine Storage-Engine angegeben werden. Wird diese nicht spezifiziert so wird die Default Storage-Engine verwendet, in Version 5.6. ist dies InnoDB. Im Dateisystem wird für jede angelegte Datenbank ein Ordner erstellt, in welchen für jede Tabelle eine Datei angelegt wird deren Struktur abhängig von der gewählten Storage-Engine ist. Diese Arbeit geht auf die zwei populärsten Storage Engines für Singe Node Systeme, namentlich InnoDB und MyISAM, als auch auf die populärste Storage Engine für Clusterarchitekturen, namentlich NDB (Network Database) ein.

### 3.1.2 Storage-Engines

Die Storage-Engine ist jene Softwarekomponente eines DBMS, die sowohl für die CRUD (Create, Read, Update, Delete) Operationen als auch für die Ausführung der Transaktionen und der Verwaltung von Indizes zuständig ist. Die Architektur von MySQL ist dabei so gestaltet, dass verschiedene Storage-Engines unterstützt werden. Je nach Anforderungen der Applikation kann so (auch während der Laufzeit des MySQL Servers) eine neue Storage-Engine eingebunden und verwendet werden, die den Anforderungen am geeignetsten ist [28].

#### InnoDB

InnoDB ist eine hochverfügbare und hochperformante Storage-Engine für verschiedenste Einsatzszenarien. Die Vorteile von InnoDB sind [29]:

- Das Design folgt dem ACID Model
- Transaktionsorientiert mit Commit, Rollback und Crash-Recovery Unterstützung
- Locking auf Zeilenebene
- Die Struktur der Daten auf der Festplatte ist für Abfragen basierend auf den Primärschlüsseln optimiert
- Gewährleistung von Datenintegrität mittels referenzieller Integrität der Fremdschlüssel
- Das Design ist auf eine effiziente CPU Auslastung und maximale Performance für große Datenmengen ausgelegt

#### MyISAM

MyISAM basiert auf der nicht mehr verfügbaren ISAM Storage-Engine und verfügt über viele nützliche Erweiterungen. MyISAM bietet [30]:

- Einfachere Struktur aufgrund fehlender Integrität der Fremdschlüsselbeziehungen
- Locking auf Tabellenebene
- Volltextsuche
- Komprimierte Read Only Tabellen für schnelle Lesezugriffe
- Effiziente Auslastung des Arbeitsspeichers [31]

#### NDB

NDB (auch bekannt als NDBCLUSTER) ist die für den MySQL Cluster standardmäßig verwendete Storage-Engine. Diese Engine befindet sich entweder auf den Datennodes eines MySQL Clusters, auf welche von allen anderen Nodes des Clusterverbunds zugegriffen werden kann oder wird als Standalone Ndb Cluster betrieben [32]. NDB bietet folgende Funktionalität [33]:

- Auto-Sharding der Tabellen über alle im Verbund befindlichen NDB Nodes
- Transaktionsorientiert, jedoch limitiert auf Read Committed
- Gewährleistung von Datenintegrität mittels referenzielle Integrität der Fremdschlüssel
- Synchrone und Asynchrone Replizierung
- Design für Hochverfügbarkeit (99,999% Uptime)
- Automatischer Failover und Failed Node Recovery
- Unterstützt In-Memory Tables
- Online Upgrades und Online Schema Modifizierung

### 3.1.3 Aufbau

Dieses Kapitel beschreibt eine typische Installation und Konfiguration der MySQL Datenbank und entspricht den Aufbau wie er für die Tests in dieser Arbeit verwendet wird. Die Installation und Konfiguration der MySQL Systeme sowie die verwendeten Softwareversionen sind in Anhang B dokumentiert.

#### Standalone Systeme

Für die Standalone Systeme wird MySQL in Kombination mit der InnoDB bzw. mit der MyISAM Storage-Engine verwendet. Der Aufbau ist für beide Kombinationen gleich und wird in Abbildung 5 dargestellt. Auf die Server wird von Clientseite mittels einer Java Applikation zugegriffen. Der Zugriff erfolgt dabei über den in Java 7 inkludierten JDBC Treiber.

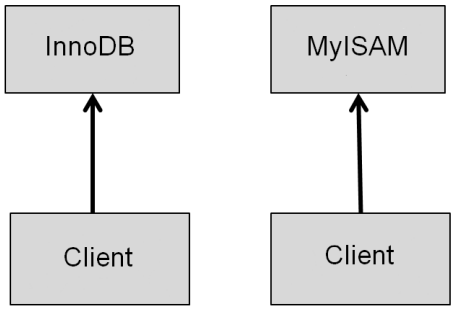



Abbildung 5: Aufbau MySQL Single Systeme

#### Cluster

Für den MySQL Cluster wird auf die NDB Storage-Engine zurückgegriffen. Der Aufbau ist in Abbildung 6 dargestellt. Der MySQL Cluster besteht aus den Data Nodes, dem Management Node und einem Load Balancer [32].

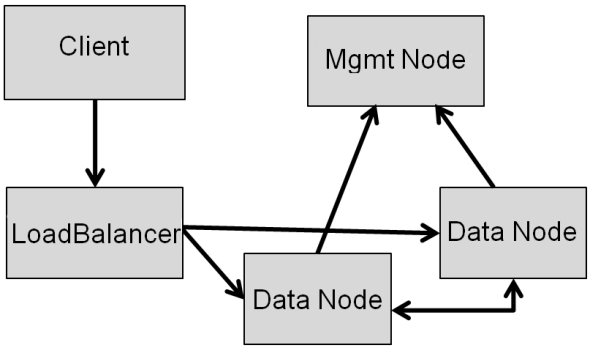


Abbildung 6: Aufbau MySQL Cluster [32]

- Mgmt Node: Aufgabe ist die Verwaltung aller anderen Nodes im Cluster, Verteilung von Konfigurationen und Zuständen, Starten und Stoppen von den Data Nodes, Ausführen von Backups, Protokollieren von Aktivitäten.
- Data Nodes: Jene Nodes auf welchen die NDB Storage-Engine konfiguriert ist. Sie werden auch als Data Shards bezeichnet und verwalten den Datenbestand der Datenbank.
- LoadBalancer: Um eine ausgewogene Last der Data Nodes zu erreichen wird dem MySQL Cluster ein Load Balancer vorgeschalten, welcher die Zugriffe des Clients gleichmäßig auf die Data Nodes verteilt.

In einem produktiven Environment ist es unumgänglich sowohl den Mgmt Node als auch den Load Balancer redundant auszuführen, um in Falle eines Fehlers die Verfügbarkeit nicht einschränken zu müssen. Ebenso ist es notwendig die Daten auf weitere Data Nodes zu replizieren um nicht nur eine höhere Verfügbarkeit zu garantieren, sondern um sich auch von Datenverlust zu schützen. Auf diese Konfigurationen wurde jedoch in dieser Arbeit bewusst verzichtet, um die Komplexität gering zu halten und die vorhandenen physischen Ressourcen (siehe Anhang A) optimal ausnutzen zu können.

### 3.1.4 Datenbanksprache

MySQL hat sich als Ziel gesetzt, bei der Implementierung der Datenbanksprache die im SQL-Standard spezifizierten Funktionalitäten zu erfüllen, ohne dabei jedoch die Geschwindigkeit oder Verfügbarkeit des MySQL Servers einschränken zu müssen. So werden transaktionsorientierte als auch nicht-transaktionsorientierte Storage-Engines supportet. Des Weiteren werden viele Features implementiert welche nicht im SQL Standard enthalten sind [51]. Um garantieren zu können, dass ein MySQL Environment den SQL Standard entspricht kann der ANSI Mode verwendet werden, womit zusätzliche Funktionalitäten nicht genutzt werden können [52]. Zwar wurde in dieser Arbeit nicht explizit der ANSI Mode aktiviert, jedoch wurde keine der von MySQL entwickelten Features in [53] verwendet.

## 3.2 MongoDB

MongoDB ist der bekannteste Vertreter der Document Stores und ist das in dieser Kategorie häufigste genützte DBS [38]. Im gesamten Datenbankmarkt findet sich MongoDB auf Platz 4 der populärsten DBS wieder [34]. Die Entwicklung des Open Source DBS begann 2007 durch das damalige Unternehmen 10gen, welches sich 2013 in MongoDB, Inc. umbenannt hat. Die Erstveröffentlichung fand 2009 statt. Die derzeit aktuelle Version 2.6.8 ist am 25.2.2015 veröffentlicht worden.

### 3.2.1 Struktur

MongoDB bietet die Möglichkeit auf einer Instanz multiple Datenbanken zu verwalten. Jede Datenbank wird dabei auf mehrere Dateien abgebildet, welche bis zu 2GB groß sein können und das Dateiformat BSON verwenden [43]. BSON ist ein Akronym für Binary JSON und ist ein binärkodiertes, serialisiertes, JSON-ähnliches Format, entworfen um den Overhead zu minimieren, leicht durchsuchbar und effizient zu sein. BSON kann als binäre Repräsentation von JSON interpretiert werden [44].
Die Dokumente werden aber nicht direkt in der Datenbank verwaltet, sondern in Containern die bei MongoDB als Collections bezeichnet werden. Eine Collection ist daher das Äquivalent einer Tabelle in RDBMS, besitzt jedoch kein Schema. Daher können Dokumente mit verschiedenen Attributen in derselben Collection verwaltet werden. Aus Gründen des Designs werden Dokumente je nach logischer Zusammengehörigkeit auf mehrere Collections verteilt [45].

### 3.2.2 Aufbau

MongoDB wird für diese Arbeit sowohl als Standalone System als auch als Sharded Cluster implementiert. Auf die Replikation der Daten wird jedoch wie auch in Kapitel 3.1. verzichtet. In einem Produktionsenvironment sollten die einzelnen Data Nodes als Replika Set abgebildet sein. Ein Replika Set besteht aus einem Primary und zumindest einen Secondary und bietet synchrone Replikation um Hochverfügbarkeit zu gewährleisten und Datenverlust zu vermeiden [46]. Installations- und Konfigurationsdetails befinden sich in Anhang C.

#### Standalone System

Abbildung 7 zeigt einen Standalone Server mit Zugriff von einem Client. Die clientseitige Java Applikation verwendet für den Zugriff auf den Datenbankserver den in Java implementierten MongoDB Datenbank Treiber.

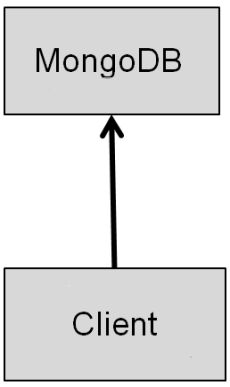



Abbildung 7: MongoDB Standalone Server

**Cluster**

Der Aufbau eines MongoDB Sharded Clusters ist in Abbildung 8 skizziert und besteht aus den Config Server, den Shards und den Router [47].

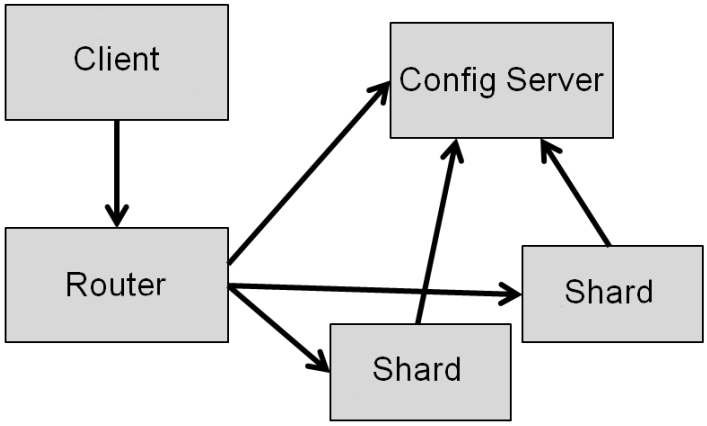


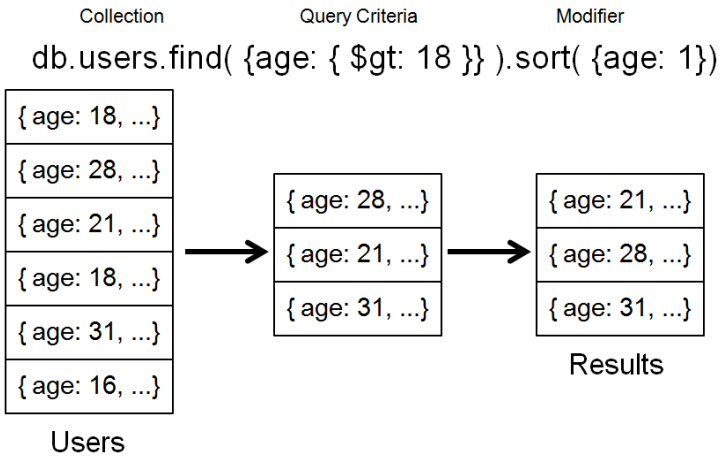

Abbildung 8: MongoDB Sharding Cluster [47]

- Config Server: Speichert die Metadaten des Clusters. In diese Kategorie fallen auch die Informationen auf welchen Shard welche Dokumente gespeichert sind.
- Shards: Speichern die Dokumente.
- Query Router: Sind dafür verantwortlich, die Anfragen vom Client an die richtigen Shards weiterzuleiten und die Ergebnisse zusammengefasst an den Client zurückzusenden.

### 3.2.3 Datenbanksprache

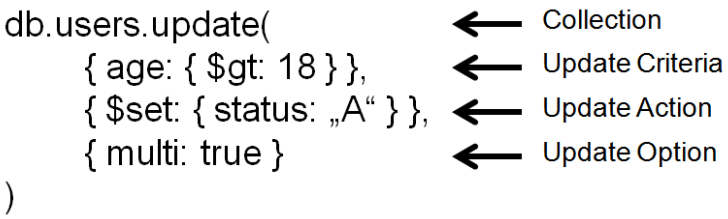
MongoDB verfügt über eine eigene auf dem JSON Format basierende Datenbanksprache. Die Abfragen werden dabei auf einer Collection ausgeführt, was hierarchisch gesehen mit SQL Abfragen auf Tabellenebene gleichgesetzt werden kann. Die Datenbanksprache wird dabei in drei grobe Punkte unterschieden:

- Queries: Abfragen dieser Kategorie spezifizieren Kriterien oder Konditionen welche Dokumente identifizieren. Das Ergebnis der Abfrage beinhaltet alle identifizierten Dokumente. Eine Abfrage besteht dabei aus einer optionalen Projektion, welche dem Select Statement der SQL entspricht, einem oder mehrere durch die logischen Verbundoperatoren "and" und "or" verknüpften Kriterien und optionalen Modifizierern wie Sortierung und Limitierung. Beispiel 1 zeigt eine solche Abfrage. Dabei werden alle Dokumente der Collection "users" durchsucht und geprüft ob das Attribut "age" größer gleich 18 ist. Anschließend folgt für alle Dokumente die dieses Kriterium erfüllen eine aufsteigende Sortierung bevor das Ergebnis zurückgegeben wird.




Beispiel 1: MongoDB Query [48]

- Data Modifications: bezieht sich auf Operationen die Daten erzeugen, updaten oder Löschen. Die Keywords in gleicher Reihenfolge dazu lauten insert, update und remove. Die Update und Löschoperation sind in der Hinsicht gleich wie die Abfrageoperationen aufgebaut, als dass sie auch auf bestimmte Kriterien eingeschränkt werden können welche die Dokumente erfüllen müssen. Beispiel 2 erläutert dies anhand einer Update Operation.

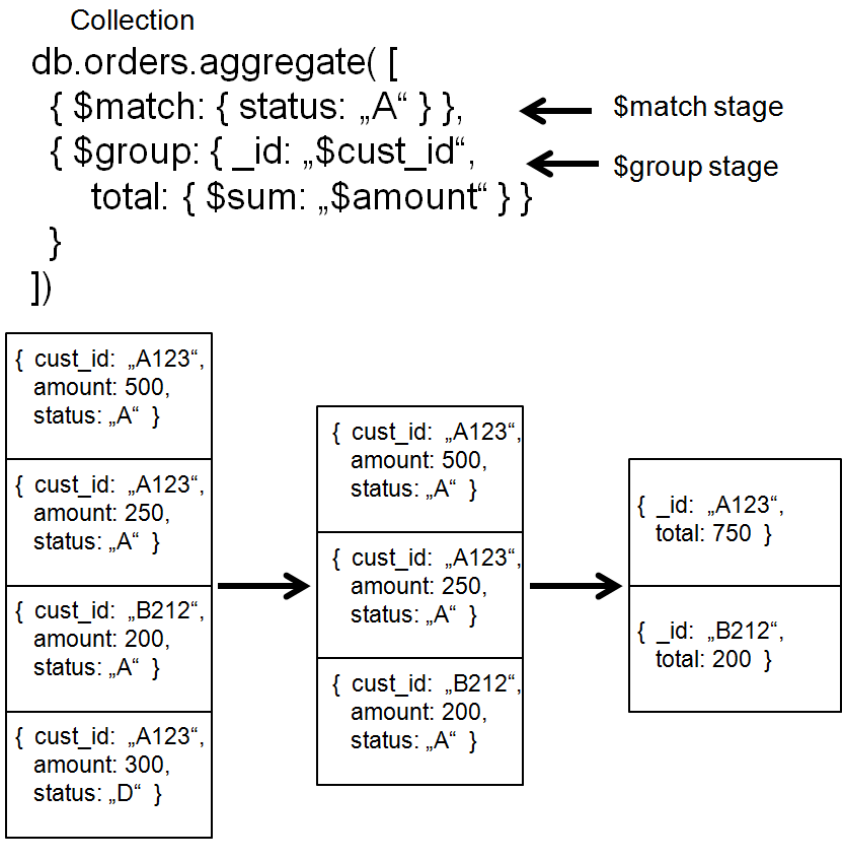



Beispiel 2: MongoDB Update [49]

Zuerst wird die Collection "user" für die Update Operation ausgewählt. Anschließend werden alle Dokumente auf die Eigenschaft "age" größer gleich 18 gefiltert. In weiterer Folge werden allen Dokumenten, die das angegebene Kriterium erfüllen, ein weiteres Attribut "status" mit dem Wert "A" (Adult) hinzugefügt. Die Eigenschaft "multi:true" spezifiziert, dass alle Dokumente die das Suchkriterium erfüllen das Update erhalten sollen.

- Aggregations: Werden verwendet, um Abfragen an Datenbeständen vorzunehmen und diese anschließend auszuwerten. Dabei werden die Werte der einzelnen Dokumente zusammengruppiert und weitere Operationen ausgeführt um das Ergebnis zu minimieren. Für die Aggregation sieht MongoDB drei Möglichkeiten vor: Die Aggregation Pipeline, Die Map-Reduce Funktionalität, und Single Purpose Aggregation Methods. Diese Arbeit beschränkt sich auf die Anwendung der

Aggregation Pipeline. Diese besteht aus mehreren Stages, wobei jede dieser Stages einen dedizierten Verarbeitungsprozess wahrnimmt. Beispiel 3 zeigt den Ablauf einer Aggregation Pipeline:




Beispiel 3: MongoDB Aggregation Pipeline [50]

Die Aggregation Pipeline wird auf der Collection "oders" angewendet, und liefert nach der ersten Stage alle Dokumente zurück, die das Kriterium "Status: A" erfüllen. Diese werden anschließend nach Customern gruppiert, wobei das Feld "Amount" pro Customer summiert wird. Das Ergebnis wird abschließend an den Client zurückgegeben.

## 3.3 Couchbase

Einer der am stärksten wachsenden DBS ist Couchbase. Sie wird nach [34] auf Platz 24 der meist genutzten Datenbanken eingeordnet und ist nach MongoDB und CouchDB der populärste Document Store [38]. Die Entwicklung begann 2010 unter dem Namen Membase, mit dem Ziel die In-Memory Datenbank Memcached um einen persistenten und abfragbaren Layer zu erweitern. 2011 wurde das Projekt mit CouchOne vereint, welche für viele Entwicklungsschritte hinter CouchDB verantwortlich sind. Das daraus entstandene Unternehmen Couchbase, Inc. veröffentlichte die erste Version mit der Versionsnummer

Couchbase-Server 1.8 im Jännern 2012 [39]. Die derzeit aktuellste Version Couchbase Server 3.0.2 wurde im Dezember 2014 released [40]. Parallel zur Couchbase-Server Entwicklung wird seit September 2013 die Abfragesprache N1QL (umgangssprachlich Nickel genannt) entwickelt. Diese ist gegenwärtig in der vierten und letzten Developer Preview verfügbar und beinhaltet alle für die erste Version geplanten Features [41]. N1QL wird in naher Zukunft gebündelt mit Couchbase Server 4.0 als vollständiges Software Packet verfügbar sein und den Namen auf "SQL for Documents" ändern [42]. Diese Arbeit wird primär N1QL in der DP4 für das Abfragen des Dokumentenbestands nutzen, und nur für einfache Lese- und Schreiboperationen auf die in Couchbase-Server eingebetteten Funktionalitäten zurückgreifen.

### 3.3.1 Struktur

Couchbase ist ein clusterbasierendes System in welchen jeder Node die gleichen Aufgaben hat und in welchen die Konfigurationen ident sind. Die Dokumente werden dabei in sogenannten Buckets verwaltet, welche in Couchbase das Synonym für Container darstellen. Im Gegensatz zu MongoDB oder MySQL werden in Couchbase die Buckets automatisch zwischen allen Nodes verteilt, dies muss also nicht explizit konfiguriert werden. Ein weiterer Unterschied ist der hierarchische Aufbau - Couchbase definiert keine Datenbanken. Die maximale Anzahl an konfigurierbaren Buckets ist auf 10 beschränkt [54].
Die sich in den jeweiligen Buckets befindlichen Dokumente werden auf sogenannte vBuckets aufgeteilt. Ein Couchbase Cluster verfügt unabhängig von der Anzahl an Dokumenten oder konfigurierten Buckets über 1024 vBuckets, welche über alle im Cluster befindlichen Nodes verteilt werden. Jedem Dokument wird bei der Speicherung eine ID zugewiesen, welches auf das zuständige vBucket verweist. Mittels einer Mapping Funktion wird bei einem Lookup eines Dokuments das zuständige vBucket ermittelt und der Server, auf welchen sich dieses vBucket befindet, wird für die weiteren Operationen identifiziert. Aufgrund dieser Eigenschaft kann in Couchbase sowohl auf einen Management- als auch auf einen Proxyserver verzichtet werden [55].

### 3.3.2 Aufbau

Der Couchbase Server wird sowohl als Standalone System als auch als Cluster konfiguriert. Ein weiterer Server ist für die N1QL Abfragen notwendig. Auch in Couchbase wird aus den im Kapitel 3.1. genannten Gründen nicht auf die Replikation der Daten eingegangen, sollte jedoch für ein Produktionsenvironment berücksichtigt werden. Anhang D beschreibt die Installation und Konfiguration der Couchbase Systeme.

#### Standalone System

Abbildung 9 zeigt den Aufbau eines Standalone Servers in Kombination mit dem N1QL Query Server und einen Client. Dieser greift auf den Server für CRUD Operationen und Views unter Verwendung des Couchbase Java SDK direkt zu. Für N1QL Abfragen wird der

Request zuerst via des HTTP REST Interface an den N1QL Query Server geleitet, welcher für die Bearbeitung der Anfrage wiederum auf den Couchbase Server zugreift.

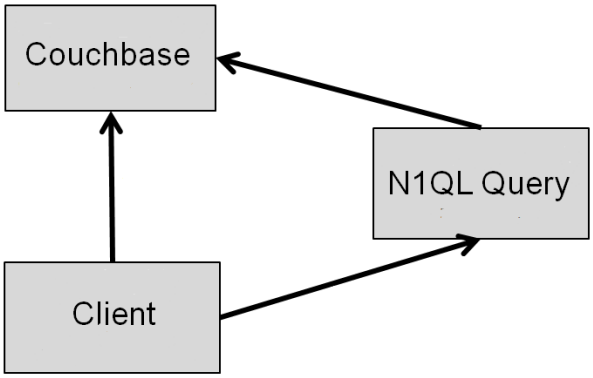



Abbildung 9: Couchbase Standalone und Query Server

**Cluster**

Wie bereits in 3.3.1 beschrieben besteht der Couchbase Cluster aus identen Servern. Dies wird in Abbildung 10 dargelegt. Auf einen Proxy- und Managementserver kann aufgrund der Architektur verzichtet werden. Diese Aufgabe übernimmt der Couchbase SDK, welcher den Status der Nodes, des Clusters und der Information über die Lokationen der einzelnen Dokumente mittels der in 3.3.1 beschriebenen Lookup Funktion abfragen und die Operationen an die entsprechenden Server weiterleiten kann [56].

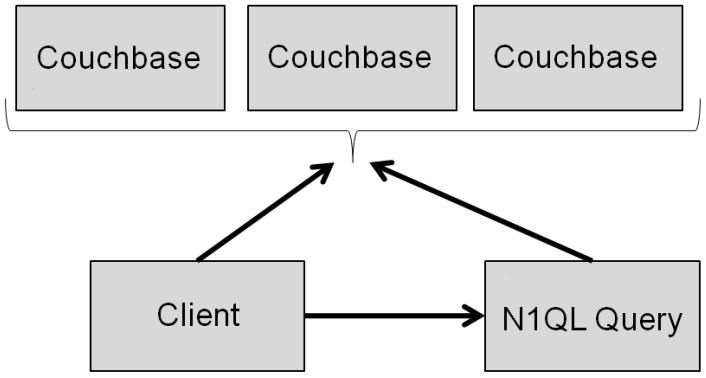



Abbildung 10: Couchbase Cluster und Query Server

### 3.3.3 Datenbanksprache

Couchbase bietet mehrere Möglichkeiten Daten abzufragen und zu manipulieren. Zu den Möglichkeiten gehören Views, SDK CRUD Operations und die experimentelle N1QL Query Language. Der Scope dieser Datenbanksprachen bezieht sich dabei nicht immer auf ein ausgewähltes Bucket, sondern kann im Fall von N1QL auch bucketübergreifend sein. Die Datenbanksprachen haben folgende Spezifikationen:

- SDK CRUD Operations: Repräsentieren die grundlegenden Operationen, welche auf einen Couchbase Cluster durchgeführt werden können. Ihre Verwendung beschränkt sich jedoch nicht nur auf schnelle und einfache Modifikationen, sondern bieten auch Methoden an um Concurrency und Locking zu managen oder Arrays, Strings und Integers mittels "add" und "increment" Methoden zu erweitern [57]. Beispiel 4 zeigt wie ein einfaches JSON Document den Cluster hinzugefügt wird.

```
1 user.insert( 1, {"id":1, "username":"markus"} )
```

Beispiel 4: Couchbase SDK Insert

Im Beispiel wird das Bucket "user" ausgewählt und das JSON Dokument mit der "id: 1" hinzugefügt. Um ein Update des Dokuments zu machen wird anstatt der insert Methode die update Methode verwendet. Zum Löschen eines Dokuments wird die delete Methode verwendet, die Angabe eines JSON Dokuments als zweites Argument entfällt beim Löschen. Beispiel 5 betrachtet ein komplexeres Beispiel, in welchem ein Dokument von der Datenbank abgefragt, geändert und wieder gespeichert wird.

```
1 // Update User Document
2 JsonObject user = userBucket.get("1").content();
3 user.getArray("item").add(item);
4 user.put("transaction", "1");
5 doc = JsonDocument.create("1", user);
6 userBucket.upsert(doc);
```

Beispiel 5: Couchbase SDK Update

Zuerst wird mittels "get" das Dokument des Users mit der "id: 1" angefordert und der Inhalt dieses Dokuments wird in der Variable "user" gespeichert. In Zeile 3 und 4 wird der Inhalt verändert. In Zeile 3 wird ein Array des Dokuments um ein zusätzliches Element erweitert und in Zeile 4 wird ein neues Feld "transaction" mit den Inhalt "1" hinzugefügt. In Zeile 5 wird daraus ein neues JsonDocument generiert, welches in Zeile 6 dem Bucket mit der Methode upsert wieder hinzugefügt wird.

- Views: Views ermöglichen es bestimmte Felder eines Dokuments zu extrahieren und einen Index über diese Felder zu erstellen. Sie finden weiters Verwendung in Map-Reduce Prozeduren. Da auf Views in dieser Arbeit nicht näher eingegangen wird, sei für Interessierte auf [58] verwiesen.
- N1QL: N1QL ist eine neue Datenbanksprache und befindet sich derzeit noch in einer Developer Preview. Um die Sprache schon jetzt testen zu können, muss das Packet auf einen eigenen Server installiert werden. Der Couchbase Server selbst, derzeit in Version 3, wird erst mit Version 4 gebündelt mit N1QL verfügbar sein. Das Ziel von N1QL war die Entwicklung einer Datenbanksprache die an SQL

angelehnt ist. Die Funktionalitäten erstrecken sich von einfachen Select-Statements hin zu Joins, Filter Expressions, Aggregat Expressions und Subqueries, welche auf Embedded Documents, Nested Objects und Multi-Value Attributes angewendet werden können [59].
Beispiel 6 zeigt die Funktionalität anhand eines Queries, welcher den Namen aller User aus den Bucket "user" zurückliefert, welche 18 Jahre oder älter sind.

```
1 SELECT name FROM user WHERE age >= 18;
```

Beispiel 6: Couchbase N1QL Query

N1QL unterstützt auch den Join von Dokumenten aus verschiedenen Buckets. Es können jedoch nicht beliebige Attribute auf Wertegleichheit überprüft werden. Stattdessen werden die Attribute der Dokumente des ersten Buckets auf Wertegleichheit mit der ID der Dokumente des zweiten Buckets geprüft. Die Abbildung 11 stellt dies grafisch dar.

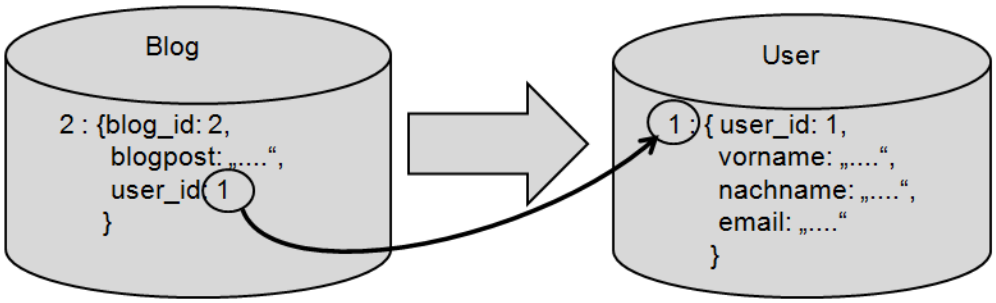



Abbildung 11: Couchbase N1QL Join

Hierbei wird das Attribut "user_id" der Dokumente des Blog Buckets mit der ID der Dokumente des User Buckets auf Wertegleichheit geprüft. Der zugehörige Join ist in Beispiel 7 spezifiziert.

```
1 SELECT Blog.blogpost, User.nachname FROM Blog JOIN User
2 ON KEYS Blog.user_id where Blog.id=2;
```

Beispiel 7: Couchbase N1QL Join Operation

Das Resultat besteht aus den im Select-Statement spezifizierten Attributen "blogpost" und "nachname". In der Where-Klausel findet noch ein Filtern nach dem Blogeintrag mit der ID 2 statt.

Dies führt jedoch auch zu einer Einschränkung. Sobald 1:n oder m:n Beziehungen vorhanden sind kann es je nach Struktur der Dokumente sein, dass gewünschte Informationen nicht verknüpft werden können. So können in Abbildung 12 zu einem Blog nicht alle zugehörigen Kommentare abgefragt werden, da die Information in den Dokumenten des Bucket "Blog" fehlt. Es wäre jedoch möglich zu einem Kommentar über das Attribut "blog_id" den zugehörigen Blog zu finden, allerdings handelt es sich dabei um eine aufwendige Aufgabe, zu dem weiters ein Filtern der Informationen notwendig ist.

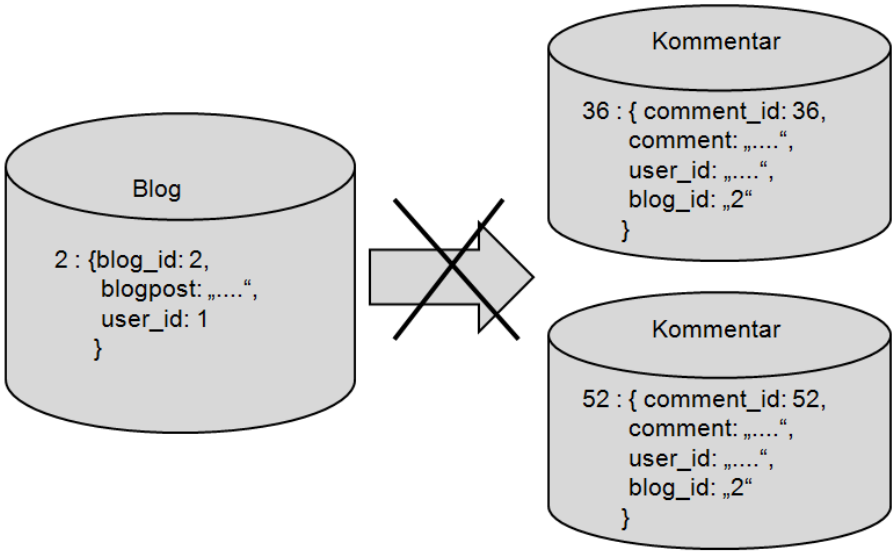



Abbildung 12: Couchbase N1QL Join 1:m Beziehung

Eine alternative Möglichkeit ist, das Design der Dokumente dahingehend zu ändern, dass die Dokumente des Bucket "Blog" diese Information mitführen. Abbildung 13 demonstriert dies.

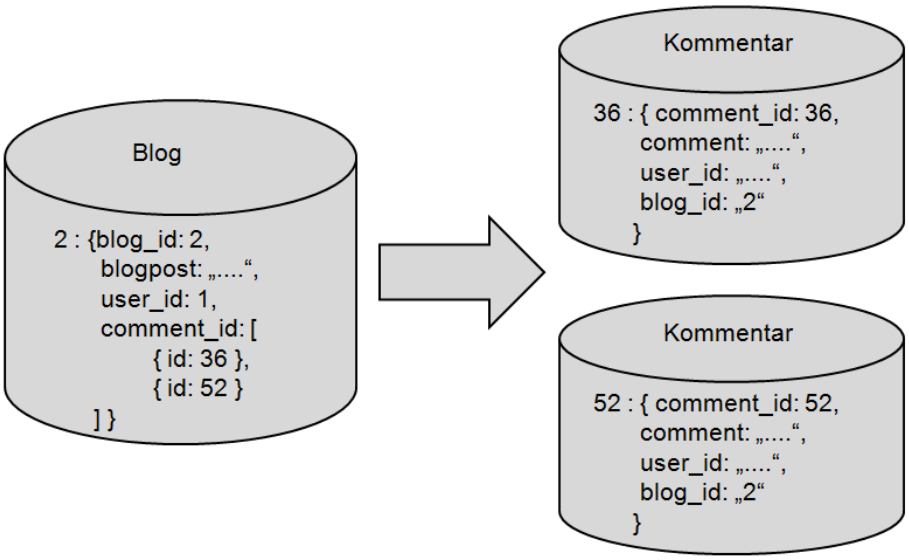



Abbildung 13: Couchbase N1QL Join 1:m Beziehung Couchbase

Um nun den Blog mit der ID 2 und den dazugehörigen Kommentaren zu erhalten kann einfach über das "comment_id" Attribut der Dokumente des Bucket "Blog" iteriert werden, was in Zeile 3 des Beispiel 8 gezeigt wird.

```
1 SELECT * FROM Blog rb
2 LEFT JOIN Comment rc
3 ON KEYS ARRAY c.id FOR c IN rb.comment_id END
4 WHERE rb.blog_id = "2";
```

Beispiel 8: Couchbase N1QL Join Operation 1:m Beziehung

# 4 Test Applikation

Um die Latenzzeiten der Datenbanken hinsichtlich der verschiedenen Modellierungen zu messen wird eine Applikation verwendet welche eine Blogging Website simuliert. Dazu werden die vier Entitäten User, Blog, Kommentar und Like abgebildet und deren Beziehungen zueinander modelliert. Ein User soll dabei nach erfolgter Registrierung sowohl Blogeinträge verfassen können, Kommentare auf existierende Blogeinträge erstellen können und auf existierende Kommentare Likes vergeben können. Die Entitäten und deren Beziehungen sind im Entity-Relationship-Modell in der Abbildung 14 dargestellt.

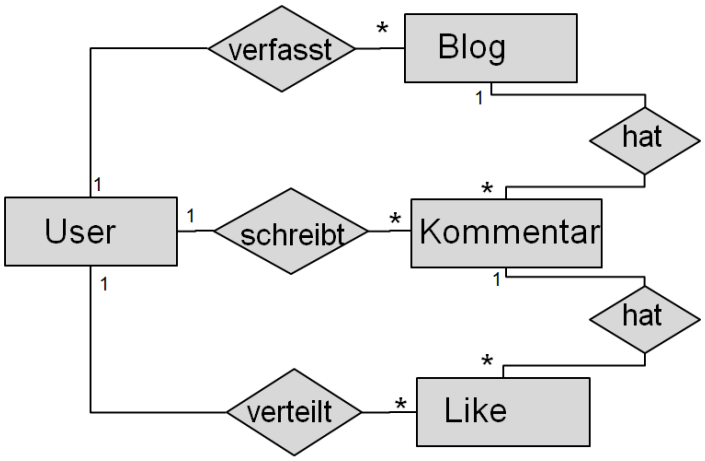



Abbildung 14: Entity-Relationship Diagram

Das Programm selbst ist in Java entworfen und besteht aus zwei Teilen:

- Datengenerierung: Wird verwendet um die Datenbanken mit Daten zu befüllen. Die Inhalte werden dabei mithilfe von Zufallsvariablen erzeugt. Beim normalisierten Datenmodell der relationalen Datenbanken als auch bei den Refrenced Documents der Document Stores werden ebenso die Beziehungen zwischen den Objekten durch Zufallsvariablen bestimmt.

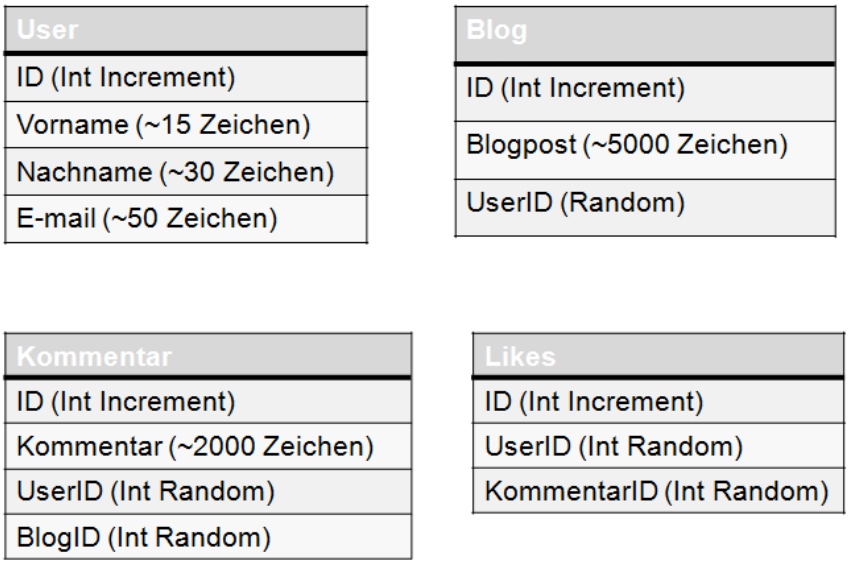



Abbildung 15: Entitäten

Die Entitäten haben die in Abbildung 15 dargestellten Eigenschaften:

- Jeder User ist über eine eindeutige ID identifizierbar. Der Vorname besteht aus einem Zufallsstring mit 15 Zeichen, der Nachname aus einem Zufallsstring mit 30 Zeichen und die E-mail setzt sich zusammen aus den Vornamen, einen Punkt als Trennzeichen, den Nachnamen, den "@" Symbol, einem Zufallsstring aus drei Zeichen welches den Provider repräsentiert, einem Punkt, und einem Zufallsstring aus zwei Zeichen welcher die Länderkennung repräsentiert.
- Der Blog hat eine eindeutige ID, der Blogpost (Blogeintrag) besteht aus einem Zufallsstring bestehend aus 5000 Zeichen und es wird per Zufallsgenerator ein User ermittelt welcher diesen Blog verfasst hat.
- Der Kommentar hat eine eindeutige ID, der Text des Kommentars besteht aus 2000 Zeichen, es wird ein User bestimmt welcher den Kommentar verfasst hat und es wird ein Blog gewählt zu welchem der Kommentar gehört. Alle diese Eigenschaften werden ebenso über den Zufallsgenerator erzeugt.
- Der Like hat eine eindeutige ID und wird von einem User auf einen Kommentar vergeben, sowohl User als auch Kommentar werden vom Zufallsgenerator bestimmt.

- Datenabfrage: Dabei werden verschiedene Queries auf die befüllten Datenbanken abgesetzt und die Latenz (Ausführungszeit [60]) gemessen und aufgezeichnet. Die einzelnen Queries werden im Kapitel 5 beschrieben.

Weitere Details zum Programm befinden sich im Anhang E.

## 4.1 Datenmodellierung

In diesem Kapitel wird gezeigt, wie die vier Entitäten User, Blog, Kommentar und Like modelliert werden. Dabei wird zwischen normalisierten und denormalisierten relationalen Datenbanken und zwischen referenced und embedded Documents unterschieden. Die normalisierte Datenbank entspricht dabei einer Datenbank in 3. Normalform. Die denormalisierte Datenbank stellt eine Datenbank in 1. Normalform dar. Die Modellierung der referenced Documents ist angelehnt an die normalisierte Datenbank, während die embedded Documents denselben Informationsgehalt in einem Dokument zusammenfasst. Im Mittelpunkt der Modellierung steht der Blog. Da sich User theoretisch zuerst registrieren müssen, um in weiterer Folge einen Blog schreiben zu können, ist diese Entität in allen vier Modellierungen vorhanden.

### 4.1.1 Normalisiert

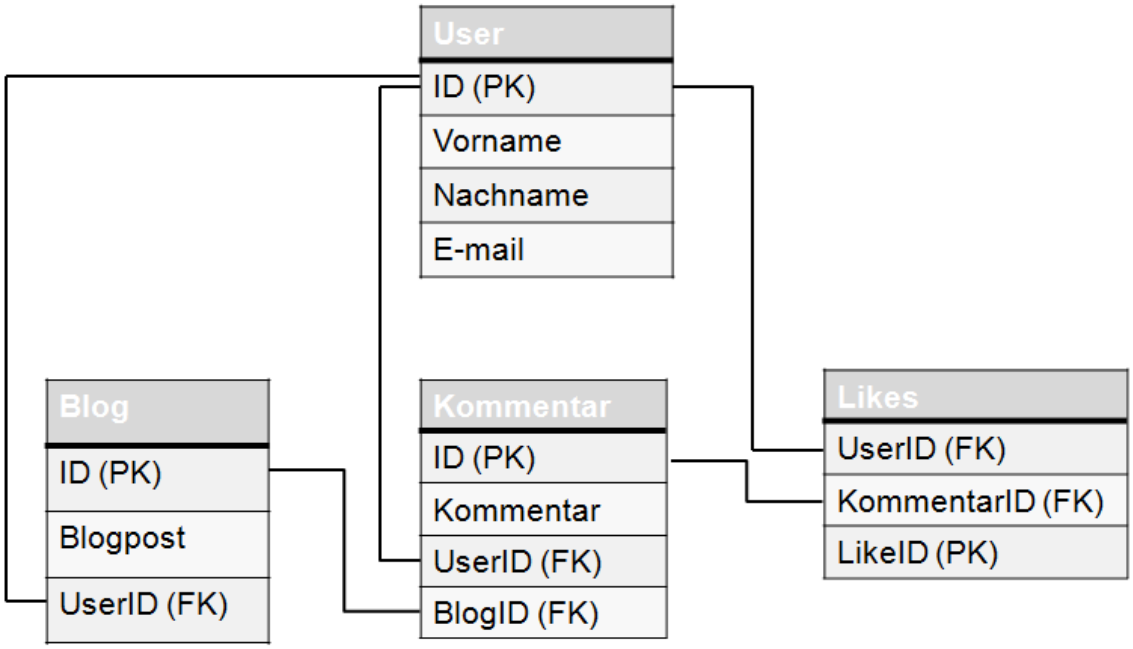



Abbildung 16: Datenmodellierung Normalisiert

Abbildung 16 zeigt die Modellierung der normalisierten relationalen Datenbank. Dabei werden auch die Beziehungen der Entitäten mittels Fremdschlüssel (Foreign Key (FK)) dargestellt. Um diese Modellierung in MySQL abzubilden wird eine Datenbank mit vier Tabellen angelegt. Beispiel 9 zeigt das Anlegen einer Datenbank in MySQL.

```
1 // Anlegen der Datenbank normalized
2 mysql> create database normalized;
```

Beispiel 9: Anlegen der normalisierten Datenbank

Beispiel 10 zeigt das Anlegen der Tabellen anhand der Storage-Engine InnoDB. Dies wurde analog für die Storage-Engine MyISAM und NDB durchgeführt. Da das Wort "Like" ein reserviertes Keyword in SQL ist, wurde hier die Mehrzahl "Likes" im Tabellenname verwendet.

```
1 // Tabelle User
2 mysql> create table User(
3   id_user INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
4   vorname VARCHAR(15), nachname VARCHAR(30), email VARCHAR(50)
5 ) ENGINE = InnoDB ;
6
7 // Tabelle Blog
8 mysql> create table Blog(
9   id_blog INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
10  blogpost VARCHAR(5000), user_id INT NOT NULL,
11  FOREIGN KEY(user_id) REFERENCES User(id_user)
12) ENGINE = InnoDB ;
13
14// Tabelle Comment
15mysql> create table Comment(
16  id_comment INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
17  text VARCHAR(1000),
18  user_id INT NOT NULL,
19  blog_id INT NOT NULL,
20  FOREIGN KEY(user_id) REFERENCES User(id_user),
21  FOREIGN KEY(blog_id) REFERENCES Blog(id_blog)
22) ENGINE = InnoDB ;
23
24// Tabelle Likes
25create table Likes(
26  id_like INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
27  user_id INT NOT NULL,
28  comment_id INT NOT NULL,
29  FOREIGN KEY(user_id) REFERENCES User(id_user),
30  FOREIGN KEY(comment_id) REFERENCES Blog(id_comment)
31) ENGINE = InnoDB ;
```

Beispiel 10: Anlegen der Tabellen der normalisierten Datenbank

Alle Tabellen verfügen über einen eindeutigen Identifier, welcher in den Zeilen 3,11,16 und 26 als Primärschlüssel definiert wird und automatisch für jeden Datensatz inkrementiert wird. Alle Felder welche einen Fremdschlüssel beinhalten wurden weiters mit "NOT NULL" definiert, um die in Abbildung 17 beschriebenen Beziehungen umsetzten zu können.
Obwohl MyISAM keine Fremdschlüsselbeziehungen unterstützt, können diese bei den Tabellendefinitionen angegeben werden. Diese Angaben werden zur Laufzeit jedoch nicht umgesetzt solange MyISAM als Storage-Engine verwendet wird [30].

### 4.1.2 Denormalisiert

Die Modellierung der denormalisierten Tabellen ist in Abbildung 17 gegeben. Wie bereits zu Beginn des Kapitels beschrieben, steht der Blog im Mittelpunkt. Die Tabelle Blog wurde daher so modelliert, dass sie sämtliche Informationen beinhaltet. Dazu zählt der Blog selbst, von wem dieser verfasst wurde, alle zugehörigen Kommentare und von wem diese geschrieben wurden, und alle Likes und vom wem diese vergeben wurden. Der Informationsgehalt der Tabelle entspricht dabei den eines "Full Left Join" der Entität Blog mit den Entitäten User, Kommentar und Likes der normalisierten Datenbank. Weiters wird noch die Tabelle User modelliert, damit sich diese auf der Website registrieren. Es besteht jedoch keine Beziehung zwischen den beiden Tabellen.

| User |
|---|
| ID (PK) |
| Vorname |
| Nachname |
| E-mail |

| Blog | | |
|---|---|---|
| ID (PK) | | |
| Blog_ID (SI) | Kommentar_ID (SI) | Like_ID (SI) |
| Blog_Post | Kommentar_Text | Like_UserID |
| Blog_UserID | Kommentar_UserID | Like_Vorname |
| Blog_Vorname | Kommentar_Vorname | Like_Nachname |
| Blog_Nachname | Kommentar_Nachname | Like_E-mail |
| B_E-mail | Kommentar_E-Mail | |

Abbildung 17: Datenmodellierung Denormalisiert

Beispiel 11 und 12 zeigen das Anlegen der Datenbank und der Tabellen.

```
1 // Anlegen der Datenbank denormalized
2 mysql> create database denormalized;
3
4 // Tabelle User
5 mysql> create table User(
6   id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
7   vorname VARCHAR(50), nachname VARCHAR(50), email VARCHAR(150)
8 ) ENGINE = MYISAM ;
9
```

Beispiel 11: Anlegen der Datenbank und Tabelle User der denormalisierten Datenbank

```
// Tabelle Blog
mysql> create table Blog(
  id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
  b_id INT NOT NULL,
  b_blogpost VARCHAR(5000),
  b_user_id INT NOT NULL,
  b_vorname VARCHAR(15),
  b_nachname VARCHAR(30),
  b_email VARCHAR(50),
  c_id INT NOT NULL,
  c_comment VARCHAR(1000),
  c_user_id INT NOT NULL,
  c_vorname VARCHAR(15),
  c_nachname VARCHAR(30),
  c_email VARCHAR(50),
  l_id INT NOT NULL,
  l_user_id INT NOT NULL,
  l_vorname VARCHAR(15),
  l_nachname VARCHAR(30),
  l_email VARCHAR(50)
) ENGINE = MYISAM ;
```

Beispiel 12: Anlegen der Tabelle Blog der denormalisierten Datenbank

### 4.1.3 Normalisiert vs. Denormalisiert

Kapitel 2.5.3 ist auf die Modellierung von relationalen Datenbanken bereits eingegangen. Hier soll anhand eines Beispiels der Unterschied dargestellt werden. Wie bereits erwähnt ist der Vorteil der denormalisierten Modellierung, dass alle Informationen in einer Tabelle vorhanden sind während diese in der normalisierten Datenbank erst durch Verbundoperationen gewonnen werden müssen. Der Nachteil ist jedoch die redundante Speicherung der einzelnen Einträge. So müssen für jeden neuen Kommentar alle Informationen des Blogs erneut gespeichert werden. Hat nun ein Blog 1000 Kommentare, so wird auch der Blog selbst 1000 Mal gespeichert. Abbildung 18 stellt den Sachverhalt dar (für den Demonstrationszweck wurden nicht alle in der Abbildung 17 spezifizierten Attribute in die Abbildung aufgenommen).

| ID | Blog_ID | Blog | Kommentar_ID | Kommentar |
|---|---|---|---|---|
| 1 | 1 | Blogeintrag1 | 1 | Kommentar1 |
| 2 | 1 | Blogeintrag1 | 2 | Kommentar2 |
| 3 | 2 | Blogeintrag2 | 3 | Kommentar3 |
| 4 | 2 | Blogeintrag2 | 4 | Kommentar4 |
| 5 | 2 | Blogeintrag2 | 5 | Kommentar5 |

Abbildung 18: Beispiel einer denormalisierten Datenbank

Obwohl die beiden Kommentare mit den IDs 1 und 2 auf denselben Blog mit der ID 1 zeigen, muss der Blog für beide Kommentare gespeichert werden. Um alle Informationen bezüglich des ersten Blogeintrags "Blogeintrag1" zu erhalten ist folgendes Statement notwendig:

```
1 SELECT * FROM blog WHERE Blog_ID = 1;
```

Beispiel 13: Abfrage der denormalisierten Tabelle

Im Gegensatz dazu kann der gleiche Datenbestand in einer normalisierten Tabelle ohne Redundanzen modelliert werden, wie die Abbildung 19 veranschaulicht.

| Kommentar_ID | Kommentar | Blog_ID (FK) |
|---|---|---|
| 1 | Kommentar1 | 1 |
| 2 | Kommentar2 | 1 |
| 3 | Kommentar3 | 2 |
| 4 | Kommentar4 | 2 |
| 5 | Kommentar5 | 2 |

| Blog_ID | Blog |
|---|---|
| 1 | Blogeintrag1 |
| 2 | Blogeintrag 2 |

Abbildung 19: Beispiel einer normalisierten Datenbank

Um hier jedoch alle Informationen bezüglich des ersten Blogeintrags abzufragen bedarf es einer Verbundoperation, was zu höherer Komplexität der Abfrage führt.

```
1 SELECT * FROM blog LEFT JOIN comment
2 on blog.Blog_ID = comment.Blog_ID
3 WHERE Blog_ID = 1;
```

Beispiel 14: Abfrage der normalisierten Tabelle

### 4.1.4 Referenced Documents

Wie bei den normalisierten Datenbanken werden auch bei den referenced Documents die Entitäten getrennt verwaltet und durch Referenzen die Beziehungen zwischen ihnen modelliert. Der Zusammenhang wird in Abbildung 20 dargestellt.

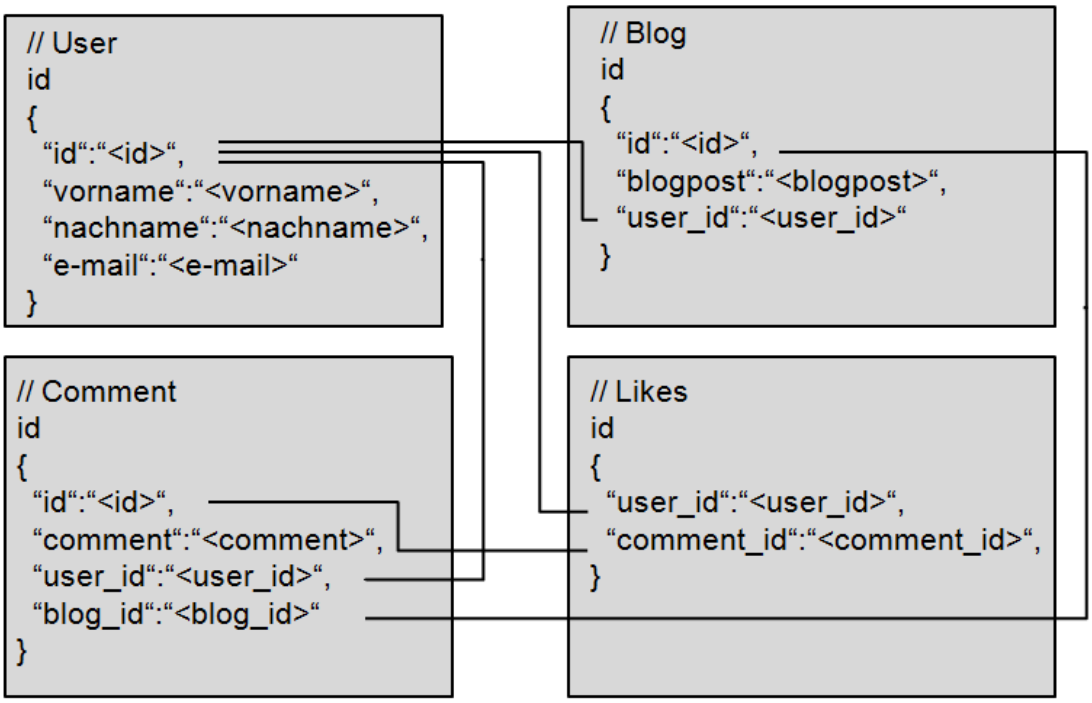



Abbildung 20: Datenmodellierung referenced Documents

Im Gegensatz zu den relationalen Datenbanken wird jedoch bei Document Stores kein Schema vorausgesetzt, was das Anlegen der einzelnen Container erleichtert. Beispiel 15 zeigt das Anlegen der Datenbank und der Collections in MongoDB.

```
> use referenced
switched to db referenced

> db.createCollection("User");
{ "ok" : 1 }

> db.createCollection("Blog");
{ "ok" : 1 }

> db.createCollection("Comment");
{ "ok" : 1 }

db.createCollection("Likes");
{ "ok" : 1 }
```

Beispiel 15: Anlegen der Datenbank und Collections in MongoDB

Der Vorgang erfolgt in Couchbase analog, wird jedoch über ein Webinterface abgewickelt wodurch zugunsten der Übersichtlichkeit auf ein Beispiel verzichtet wird.

### 4.1.5 Embedded Documents

Die embedded Documents können mit der denormalisierten Datenbank verglichen werden. Die für einen Blog vorhandene Information wird innerhalb eines Dokuments verwaltet. Aufgrund der Struktur von JSON Dateien können die meisten Redundanzen vermieden werden. So ergeben sich zum Beispiel bei der Speicherung eines Blogs mit zugehörigen Kommentaren und Likes in einem Dokument im Gegensatz zur denormalisierten Datenbank keine Redundanzen. Wird jedoch zu jedem Blog, Kommentar und Like auch deren Autor festgehalten, so können Redundanzen entstehen. Das ist unter anderem dann der Fall, wenn der Autor des Blogs auch einen Kommentar zu diesem Blog verfasst. Es ist somit nicht immer möglich Redundanzen in embedded Documents zu vermeiden.
Wie in Abbildung 21 festgehalten wird auch hier der User separat vom Blog verwaltet. Zwischen den Dokumenten vom Typ User und den Dokumenten vom Typ Blog bestehen keine Beziehungen.

```
// User
id
{
 "id":"<id>",
 "vorname":"<vorname>",
 "nachname":"<nachname>",
 "e-mail":"<e-mail>"
}
```

```
// Blog
id
{
 „id" : „id",
 "blogpost":"<blogpost>",
 "userid":"<userid>",
 "vorname":"<vorname>",
 "nachname":"<nachname>",
 "e-mail":"<e-mail>",
 "comments": [
  {
  "comment":"<comment>",
  "userid":"<userid>",
  "vorname":"<vorname>",
  "nachname":"<nachname>",
  "e-mail":"<e-mail>",
  "likes": [
    "userid":"<userid>",
    "vorname":"<vorname>",
    "nachname":"<nachname>",
    "e-mail":"<e-mail>",
  ]}
 ]}
```

Abbildung 21: Datenmodellierung embedded Documents

Das Anlegen der Datenbank und der einzelnen Container erfolgt genauso wie in Beispiel 15, wobei nur die beiden Container "User" und "Blog" benötigt und auf die beiden Container "Comment" und "Likes" verzichtet werden kann.

### 4.1.6 Referenced vs. Embedded

Auch hier soll der Unterschied zwischen den beiden Modellierungstypen anhand eines Beispiels gezeigt werden (Um die Übersichtlichkeit zu erhöhen finden wie in Kapitel 3.1.2 nicht alle Attribute Verwendung).

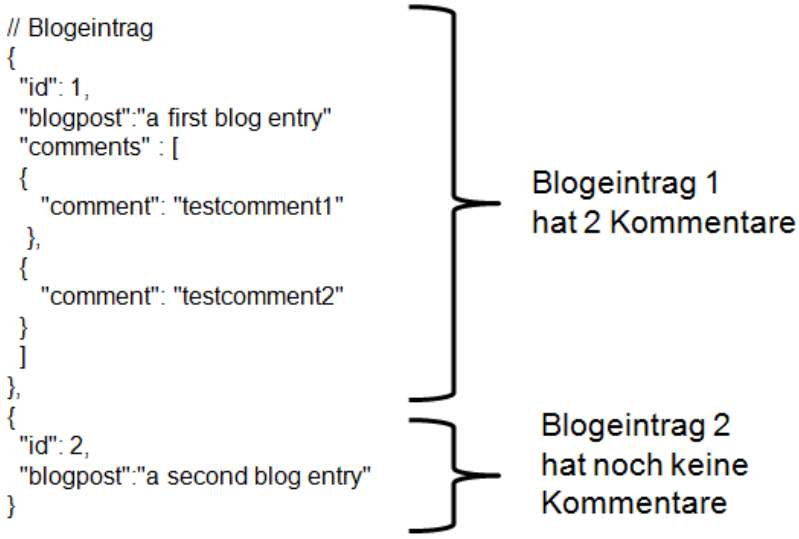



Abbildung 22: Beispiel eines embedded Documents

Abbildung 22 zeigt ein embedded Document, welches einen Blogeintrag modelliert. Die zugehörigen Kommentare werden in einer Liste verwaltet. Die Daten können mit folgendem Statement in MongoDB abgefragt werden:

```
1 db.Blog.find({ id: "1" })
```

Beispiel 16: MongoDB Abfrage eines embedded Documents

Die Abfrage via N1QL erfolgt wie in SQL:

```
1 SELECT * FROM Blog WHERE id = 1;
```

Beispiel 17: Couchbase N1QL Abfrage eines embedded Documents

Abbildung 23 zeigt die Modellierung der referenced Documents. Das Attribut "blog_id" der Dokumente des Bucket "Kommentar" verweist dabei auf das entsprechende Dokument im Bucket "Blogeintrag".


Abbildung 23: Beispiel von referenced Documents

Um in MongoDB die gleiche Information wie im obigen Beispiel gewinnen zu können, müssen die Dokumente der beiden Collections "Blog" und "Kommentar" miteinander verknüpft werden. Da MongoDB keine Verbundoperationen bietet, muss in einer Schleife nach den Kommentaren gesucht werden, welche zu den Blog gehören.

```
1db.Blog.find( { id: "1"} ).forEach(
2  function (newBlog) {
3    newBlog.Comments = db.Comment.find(
4     {"blog_id": newBlog.blog_id }
5   ).toArray();
6  }
7);
```

Beispiel 18: MongoDB Abfrage von referenced Documents

Zeile 1 sucht in der Blog Collection nach Dokumenten mit der ID 1. Da diese ID ein Dokument eindeutig identifizert, wird der Schleifenkörper genau einmal durchlaufen. In diesem wird in der Collection Comment nach allen Kommentaren gesucht, die Wertegleichheit auf dem Attribut "blog_id" haben. Diese werden als Liste (Array) zurückgegeben.

Um die Verbundoperation der N1QL Query Language ausnutzen zu können, wird wie in Kapitel 3.3.3 erläutert, eine Liste der Kommentare in den Dokumenten des Blogs mitgeführt. Abbildung 24 zeigt den Blogeintrag mit dem zusätzlichen Attribut "comments", welches die IDs aller auf diesen Blog referenzierenden Kommentare speichert.

```
// Blogeintrag
{
  "id": 1,
  "blogpost":"a first blog entry„,
  "comments" : [
    { "id" : 1 },
    { "id" : 2 }
  ]
},
{
  "id": 2,
  "blogpost":"a second blog entry"
}
```

```
// Kommentar
{
  "id": "1"
  "comment":"testcomment1",
  "blog_id": 1
},
{
 "id": "2",
  "comment":"testcomment2",
  "blog_id": 1
}
```

Abbildung 24: Beispiel von referenced Documents in Couchbase

Die Abfrage in N1QL wird wie folgt formuliert:

```
1 SELECT * FROM Blog rb LEFT JOIN Comment rc
2 ON KEYS ARRAY c.id FOR c IN rb.comments END;
```

Beispiel 19: Couchbase N1QL Abfrage von referenced Documents

Zeile 1 beschreibt einen Left Join, damit wird ein Blog auch dann zurückgegeben, wenn es keine Kommentare zu diesem Blog gibt. In Zeile 2 erfolgt das Durchlaufen der im Blog stehenden IDs mittels einer Schleife und die Dokumente werden gejoined.

# 5 Benchmarks

Für die Durchführung der Tests wurden die Datenbanken mit 1000, 10.000 und 100.000 Datensätzen pro Entität befüllt. Der gesamte Datenbestand für User, Blog, Kommentar und Likes ergibt somit 4000, 40.000 und 400.000 Datensätze. Die Messung wird am Client durchgeführt. Gemessen werden der Verbindungsaufbau zur Datenbank, die Durchführung des Statements und der Verbindungsabbau. Vor jeder Messung wurden die Caches der Datenbanken geleert. Jeder Messwert stellt einen arithmetischen Mittelwert von 3 Messungen an zwei unterschiedlichen Datensätzen (somit 6 Messungen) dar. Die Graphen wurden, soweit möglich, auf einheitliche Skalen gebracht.

## 5.1 Erster Benchmark

Im ersten Benchmark erfolgt die Abfrage eines Blogs samt Kommentaren, Likes und den jeweiligen Autoren (Usern). In den unten stehenden Beispielen wird der Blog mit der ID 2 abgefragt. Für jede Messung wurde dieser Parameter jedoch gegen eine andere zufällige ID ausgetauscht. Die Abfrage kann dazu verwendet werden, auf der Webseite einen Blog mit allen zugehörigen Informationen anzuzeigen.

### 5.1.1 Formulierung des Statements in MySQL

**normalisiert**

Anmerkung: Zur besseren Übersichtlichkeit wurden die einzelnen Attribute mit dem Platzhalter <attributliste> ersetzt. Die Notwendigkeit der Angabe der Attribute anstatt des *-Operators ergab sich, da Datensätze aus der Tabelle User durch die jeweiligen Joins mit Blog, Kommentar und Likes dreimal vorhanden waren und dadurch die Vergabe von Aliasnamen notwendig war (z.B. musste das Attribut "vorname" durch die Aliases "vorname_author_blog", "vorname_author_kommentar", "vorname_author_like" ersetzt werden, da in einen Statement nicht dreimal das Attribut "vorname" Verwendung finden kann).

```
1  SELECT
2  <attributliste>
3  FROM (
4    SELECT
5    <attributliste>
6    FROM (
7        SELECT
8        <attributliste>
9        FROM Likes JOIN User ON Likes.user_id = User.id_user
10   )
11   AS LU LEFT JOIN Comment ON LU.like_comment_id =
Comment.id_comment
12   LEFT JOIN User ON Comment.user_id = User.id_user
13     )
14AS CLU RIGHT JOIN Blog ON CLU.comment_blog_id = Blog.id_blog
15JOIN User ON Blog.user_id = User.id_user
16WHERE blog_id = 2;
```

Beispiel 20: SQL Statement für 1.Abfrage des normalisierten Datenbestands

**denormalisiert**

Das Statement in der denormalisierten Datenbank weist eine wesentlich geringere Komplexität auf, wie Beispiel 21 zeigt.

```
1  SELECT * FROM Blog WHERE blog_id = 2;
```

Beispiel 21: SQL Statement für 1. Abfrage des denormalisierten Datenbestands

### 5.1.2 Ergebnis in MySQL

**normalisiert**

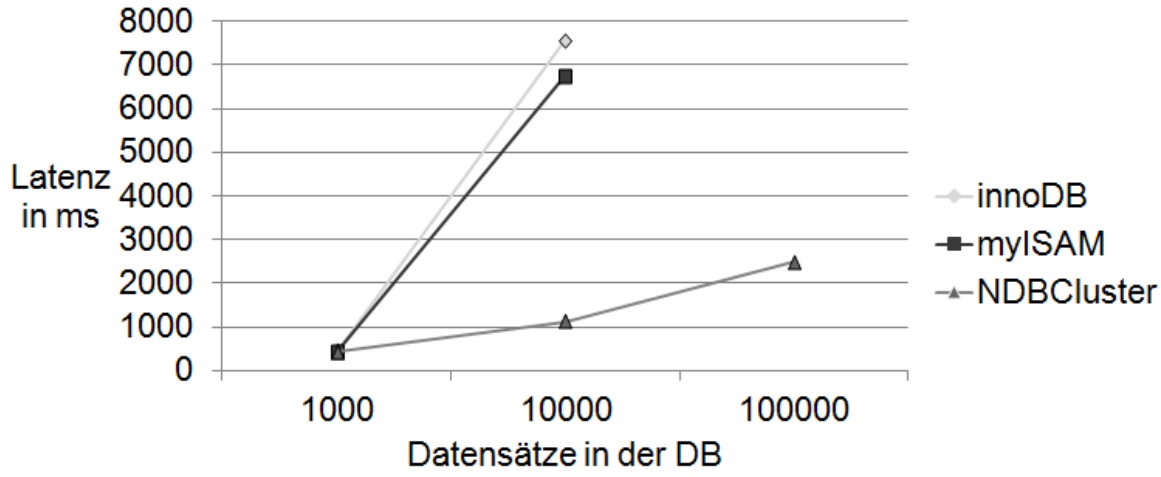



Abbildung 25: Messergebnis 1. Abfrage MySQL normalisierter Datenbestand

| Datensätze | InnoDB | MyISAM | NDBCluster |
|---|---|---|---|
| **1000** | 452 | 429 | 433 |
| **10000** | 7555 | 6728 | 1122 |
| **100000** | 13328759 | 13823153 | 2494 |

Tabelle 1: Messergebnis 1. Abfrage MySQL normalisierter Datenbestand

**denormalisiert**

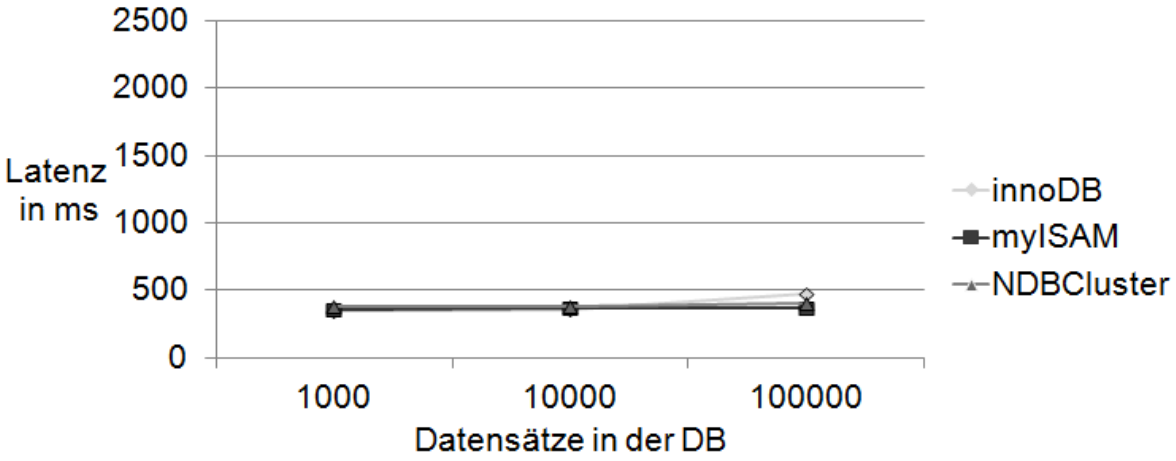



Abbildung 26: Messergebnis 1. Abfrage MySQL denormalisierter Datenbestand

| Datensätze | InnoDB | MyISAM | NDBCluster |
|---|---|---|---|
| **1000** | 347 | 359 | 376 |
| **10000** | 351 | 362 | 385 |
| **100000** | 476 | 363 | 401 |

Tabelle 2: Messergebnis 1. Abfrage MySQL denormalisierter Datenbestand

### Diskussion

Die Messergebnisse der normalisierten Datenbank weisen bereits bei einem Datenbestand > 1000 Datensätzen einen starken Anstieg der Latenz auf. Die Standalone Systeme mit den Storage-Engines InnoDB und MyISAM benötigen für die Durchführung der Verbundoperationen bereits ab 10.000 Einträgen mehrere Sekunden, diese erstreckt sich auf mehrere Stunden bei der Abfrage von 100.000 Datensätzen. Zwar verhält sich der NDB Cluster stabiler, aber auch dessen Antwortzeit weist ein starkes Wachstum auf. Im Gegensatz dazu zeigen die Werte der denormalisierten Datenbank kontinuierlich Werte im Millisekunden Bereich und steigen nur leicht an. Es kann daher deutlich der Vorteil der Denormalisierung für Abfragen dieser Art erkannt werden.

## 5.1.3 Formulierung in MongoDB

Anmerkung: Die Statements sind mittels des MongoDB Java Treibers formuliert worden.

### Embedded

```
1   // Get Blog
2   BasicDBObject blogQuery = new BasicDBObject("id", 2);
3   DBCursor blogCursor = blogCollection.find(blogQuery);
4   try {
5      // Loop through Result and build Result Set
6      HashMap<String, String> resultSet=new HashMap<String, String>();
7      while (blogCursor.hasNext()) {
8          BasicDBObject blog = (BasicDBObject) blogCursor.next();
9          resultSet.put("id", blog.getString("id"));
           < Parse Attributes >
10     }
11
12     result.add(resultSet);
13  }
```

Beispiel 22: MongoDB Statement für 1. Abfrage des Embedded Document Typs

Das Beispiel zeigt, wie in Zeile 2 die Abfrage formuliert wird und in Zeile 3 die Abfrage ausgeführt wird. In Zeile 6 wird eine HashMap für das Ergebnis angelegt und in den Zeilen 7 bis 10 erfolgt das Parsen des Ergebnisses.

### Referenced

```
1   // Get Blog
2   BasicDBObject blogQuery = new BasicDBObject("id", 2);
3   DBCursor blogCursor = blogCollection.find(blogQuery);
4   try {
5
6       // Loop through Result and build Result Set
8       while (blogCursor.hasNext()) {
9           BasicDBObject blog = (BasicDBObject) blogCursor.next();
10          < Parse Blog and Add to Result Set >
11
12          // Get User from Blog
13          BasicDBObject userBlogQery = new BasicDBObject("id",
14                  blog.getInt("user_id"));
15          DBCursor userBlogCursor =userCollection.find(userBlogQery);
16
17          while (userBlogCursor.hasNext()) {
18              BasicDBObject userBlog = (BasicDBObject) userBlogCursor
19                      .next();
20              < Parse User and Add to Result Set >
21          }
22
23          // Get Comments
24          BasicDBObject commentQuery = new BasicDBObject("blog_id",
25                  id);
26          DBCursor commentCursor = commentCollection
27                  .find(commentQuery);
28
29          while (commentCursor.hasNext()) {
30              BasicDBObject comment = (BasicDBObject) commentCursor
31                      .next();
32              < Parse Comment and Add to Result Set >
33
                // Get User from Comments
                // Get Likes
                // Get User From Likes
71
72          }
73          resultSet.put("id", blog.getString("id"));
74
75      }
76
77      result.add(resultSet);
```

Beispiel 23: MongoDB Statement für 1. Abfrage des Referenced Document Typs

Das Abfragen der Informationen ist, wie in Beispiel 23 ersichtlich, für den Referenced Document Typ wesentlich komplexer. Es wird zu Beginn wie in Beispiel 22 eine Abfrage nach dem Blog mit der ID 2 erstellt und anschließend in den Zeilen 9-11 das Ergebnis geparsed. Dieses Dokument enthält aber nur die Information über den Blog, und deswegen wird in der Collection "User" nach dem Ersteller dieses Blogs gesucht. Dies ist in den Zeilen 12-21 spezifiziert. Ab Zeile 23 werden in der Collection "Kommentar" alle Kommentare gesucht, welche zum Blog gehören. Um die Übersichtlichkeit zu bewahren wird ab Zeile 33 nur durch Anmerkungen auf die weiteren Schritte hingewiesen. Diese sind das Suchen der Verfasser der Kommentare, alle Likes welche die Kommentare bekommen haben, und alle User welche diese Likes vergeben haben. Um das Beispiel übersichtlich darstellen zu können, wurden weiters Platzhalter verwendet die mit "<>" gekennzeichnet sind. Für eine vollständige Darstellung der Abfrage sei auf den Source Code in Anhang E verwiesen.

### 5.1.4 Ergebnis in MongoDB

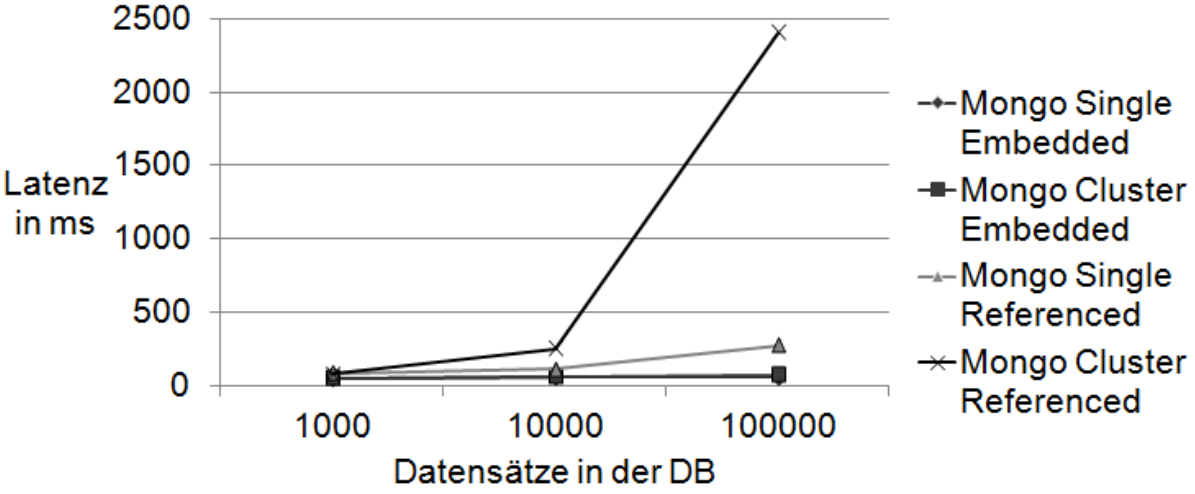



Abbildung 27: Messergebnis 1. Abfrage MongoDB

| Datensätze | MongoDB Standalone Referenced | MongoDB Cluster Referenced | MongoDB Standalone Embedded | MongoDB Cluster Embedded |
|---|---|---|---|---|
| **1000** | 87 | 84 | 51 | 54 |
| **10000** | 118 | 257 | 60 | 66 |
| **100000** | 277 | 2403 | 61 | 75 |

Tabelle 3: Messergebnis 1. Abfrage MongoDB

### Diskussion

Aus den Ergebnissen ist zu erkennen, dass die Latenzzeiten der Referenced Documents höher sind als die der Embedded Documents. Das hat mit den vielfachen Zugriffen auf die verschiedenen Collections und dem eher mühsamen Zusammenführen der Informationen zu tun. Im Vergleich genügt bei dem Embedded Documents ein Zugriff auf die Datenbank um die gewünschten Informationen zu extrahieren. Weiters ist festzustellen, dass der Cluster höhere Latenzwerte als die Standalone Instanz aufweist. Im Gegensatz zur Standalone Instanz müssen beim Cluster zusätzliche Arbeitsschritte abgewickelt werden. So muss der Query Router die Anfragen auf die entsprechenden Shards verteilen und den Response der Shards wieder zu einem Resultat zusammenführen.

## 5.1.1 Formulierung in Couchbase

### Embedded

In Beispiel 24 werden alle Attribute des Dokuments mit der blog_id 2 aus dem Bucket referenced_Blog abgefragt.

```
1 SELECT * from Blog where blog_id = "2";
```

Beispiel 24: Couchbase N1QL Statement für 1. Abfrage des Embedded Document Typs

### Referenced

Um denselben Informationsgehalt wie bei den embedded Documents abzufragen sind in N1QL Verbundoperationen notwendig. Beispiel 25 zeigt einen Join zwischen den Buckets. Aufgrund der Struktur des Dokuments kann hier in Zeile 1 der *-Operator verwendet werden. In den Zeilen 2 und 3 werden die Dokumente der beiden Buckets Blog und Comments verknüpft. In der Zeile 4 wird der Blog mit den User verknüpft. Da im Gegensatz zu den Comments ein User vorhanden sein muss, wird hier kein Left Join durchgeführt. In Zeile 5 werden die Kommentare mit den User verknüpft. Anschließend erfolgt in den Zeilen 6 bis 7 der Join zwischen den Kommentaren und den Likes und in Zeile 8 werden die Likes mit den Usern verknüpft. Zeile 9 schränkt das Ergebnis auf den Blog mit der ID 2 ein.

```
1 SELECT * FROM Blog rb
2 LEFT JOIN Comment rc ON KEYS
3   ARRAY c.id FOR c IN rb.comment_ids END
4 JOIN User rub ON KEYS rb.user_id
5 LEFT JOIN User ruc ON KEYS rc.user_id
6 LEFT JOIN Likes rl ON KEYS
7   ARRAY l.id FOR l IN rc.like_ids END
8 LEFT JOIN User rul ON KEYS rl.user_id
9 WHERE rb.blog_id = "2";
```

Beispiel 25: Couchbase N1QL Statement für 1. Abfrage des Referenced Document Typs

### 5.1.2 Ergebnis in Couchbase

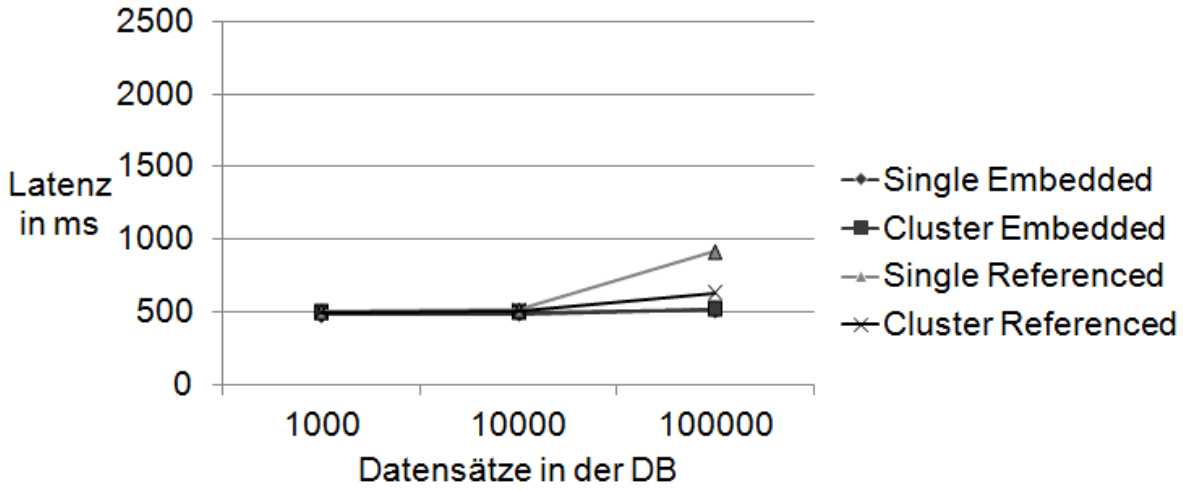



Abbildung 28: Messergebnis 1. Abfrage Couchbase N1QL

| Datensätze | Couchbase Standalone Referenced | Couchbase Cluster Referenced | Couchbase Standalone Embedded | Couchbase Cluster Embedded |
|---|---|---|---|---|
| **1000** | 503 | 501 | 482 | 498 |
| **10000** | 514 | 509 | 489 | 501 |
| **100000** | 918 | 636 | 519 | 522 |

Tabelle 4: Messergebnis 1. Abfrage Couchbase N1QL

**Diskussion**

Ähnlich wie bei MongoDB sind auch bei Couchbase die Latenzzeiten bei der Abfrage der Referenced Documents höher. Dies ist auf das komplexere Statement mit Verbundoperationen zurückzuführen. Im Gegensatz zu MongoDB kann Couchbase mit mehreren Servern im Cluster bessere Latenzzeiten erreichen als mit nur einem Server. Aufgrund der Architektur benötigt Couchbase keinen Proxy Server, sondern verwendet den Couchbase SDK um eine vBucket Map aufzubauen [56]. Dadurch gelingt es Couchbase die vorhandenen Ressourcen effizienter ausnutzen zu können als MongoDB.

## 5.2 Zweiter Benchmark

Mittels des zweiten Statements wird der sogenannte "Top Blogger" ermittelt und mit einem Tag versehen. Der Top Blogger ist dabei jener User, welcher die meisten Blogs verfasst hat. Unabhängig davon sind seine Aktivitäten bezüglich Kommentare oder Likes. Gibt es mehrere User die gleich viele Blogs verfasst haben, erhält derjenige mit der niedrigeren ID den Tag (das wäre in einem echten Environment jener User, der schon länger bei der Community dabei ist).
Um den User als "Top Blogger" kennzeichnen zu können, wurde ein zusätzliches Attribut "rank" bei der Entität User eingefügt. Bei dem User, welcher mit diesem Tag versehen werden soll, wird der Wert auf "Top Blogger" gesetzt. Bei allen anderen Usern ist es entweder auf "null" gesetzt (relationale Datenbank aufgrund der Vorgabe eines Schemas) oder nicht vorhanden (Document Store).

### 5.2.1 Formulierung in MySQL

#### normalisiert

Das Beispiel 26 zeigt zuerst, wie der Top Blogger ermittelt wird. Dafür wird in der Tabelle Blog nach den Usern gruppiert und deren Einträge gezählt. Diese Einträge werden zuerst absteigend sortiert und anschließend wird nach der user_id aufsteigend sortiert, um damit den User mit den meisten Blogeinträgen aber mit der niedrigsten ID zu finden. Anschließend wird das Ergebnis auf den ersten Eintrag des Resultats limitiert.
Im zweiten Statement wird dann das Attribut "rank" dieses Users mit dem Inhalt "Top Blogger" upgedatet.

```
// Find Top Blogger
1 SELECT user_id FROM Blog
2 GROUP BY user_id
3 ORDER BY count(user_id) DESC, user_id
4 limit 1;

// Set Tag
1 UPDATE User SET rank = "Top Blogger"
2 WHERE user_id = ?;
```

Beispiel 26: Statement für 2. Abfrage des normalisierten Datenbestands

#### denormalisiert

Der gleiche Sachverhalt wird in Beispiel 27 für die denormalisierte Datenbank verwendet. Da nun in der Tabelle Blog nicht nur die Blogs selbst, sondern auch alle Kommentare und Likes gespeichert sind, muss zuerst nach den Blogeinträgen gruppiert werden. Dies erfolgt im verschachtelten Select-Statement. Anschließend wird der Top Blogger wie in Beispiel 26 ermittelt.

Da die Information des Users aber redundant an mehreren Stellen vorhanden sein kann, ist es nun notwendig sowohl für alle Blogeinträge in der Tabelle Blog als auch für alle Kommentare und Likes in derselben Tabelle ein Update-Statement auszuführen (Hier sei nochmals auf den Aufbau der Tabelle in Kapitel 4.1.2 hingewiesen).

```
1 // Find Top Blogger
2 SELECT b_user_id FROM (
3   SELECT b_user_id FROM Blog GROUP BY b_id
4 )
5 AS BlogReduced
6 GROUP BY b_user_id
7 ORDER BY count(b_user_id) DESC, b_user_id
8 LIMIT 1;”
9
10 // Set Tag
11 UPDATE Blog SET rank = "Top Blogger"
12 WHERE blog_u_id = ?;
13
14 UPDATE Blog SET rank = "Top Blogger"
15 WHERE comment_u_id = ?;
16
17 UPDATE Blog SET rank = "Top Blogger"
18 WHERE like_u_id = ?;
```

Beispiel 27: Statement für 2. Abfrage des denormalisierten Datenbestands

### 5.2.1 Ergebnis in MySQL

**normalisiert**

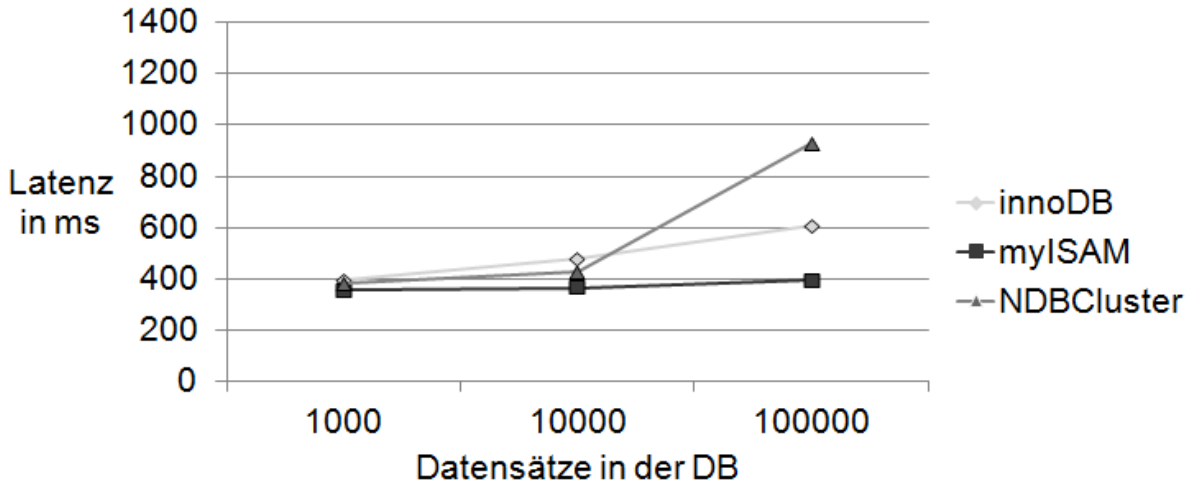



Abbildung 29: Messergebnis 2. Abfrage MySQL normalisierter Datenbestand

| Datensätze | InnoDB | MyISAM | NDBCluster |
|---|---|---|---|
| **1000** | 395 | 356 | 380 |
| **10000** | 478 | 367 | 431 |
| **100000** | 606 | 397 | 927 |

Tabelle 5: Messergebnis 2. Abfrage MySQL normalisierter Datenbestand

**denormalisiert**

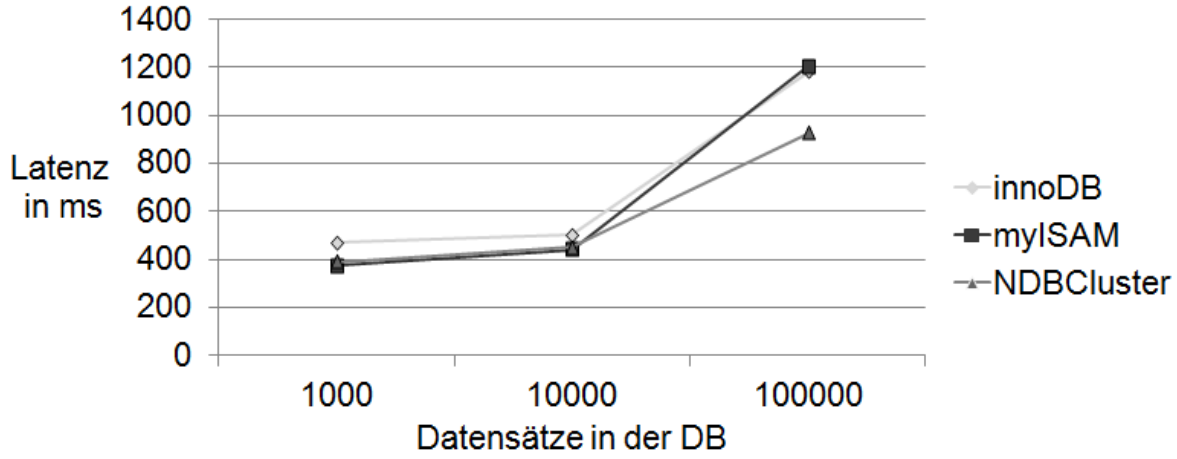



Abbildung 30: Messergebnis 2. Abfrage MySQL denormalisierter Datenbestand

| Datensätze | InnoDB | MyISAM | NDBCluster |
|---|---|---|---|
| **1000** | 468 | 376 | 390 |
| **10000** | 501 | 441 | 450 |
| **100000** | 1182 | 1206 | 927 |

Tabelle 6: Messergebnis 2. Abfrage MySQL denormalisierter Datenbestand

## Diskussion

Entgegen der Ergebnisse der ersten Messung zeigen diese nun bei der zweiten Messung beim denormalisierten Datenbestand starke Anstiege der Latenzzeiten, und beim normalisierten Datenbestand nur leicht steigende Latenzzeiten. Der Grund liegt darin, dass beim denormalisierten Datenbestand die Daten redundant vorkommen und daher an mehreren Stellen geändert werden müssen. Im Gegenzug dazu sind beim normalisierten Datenbestand die Informationen des Users in einer Tabelle abgebildet und es ist nur ein Updatevorgang notwendig.

### 5.2.2 Formulierung in MongoDB

Um den User mit den meisten Blogeinträgen herauszufinden, wird in dieser Abfrage die in Kapitel 3.2.3 beschriebene Aggregation Pipeline verwendet.

#### Referenced

```
1   // Create Group Field Object
2   DBObject groupFields = new BasicDBObject("_id", "$user_id");
3   groupFields.put("sum", new BasicDBObject("$sum", 1));
4   DBObject group = new BasicDBObject("$group", groupFields);
5
6   // Create Sort Field Object
7   DBObject sortFields = new BasicDBObject("sum", -1);
8   sortFields.put("_id", 1);
9   DBObject sort = new BasicDBObject("$sort", sortFields);
10
11  // Create Limit Field Object
12  DBObject limit = new BasicDBObject("$limit", 1);
13
14  // Create Project Field Object
15  DBObject projectFields = new BasicDBObject("_id", 1);
16  DBObject project = new BasicDBObject("$project", projectFields);
17
18  // Execute Aggregation
19  List<DBObject> pipeline = Arrays.
20      asList(group, sort, limit, project);
21  AggregationOutput output = blogCollection.aggregate(pipeline);
22
23  // Loop through Result From Aggregation
24  for (DBObject dbobject : output.results()) {
25      result.put("id", dbobject.get("_id").toString());
26
27      // Update User in User Collection
28      userCollection.update(
29          new BasicDBObject("id", dbobject.get("_id")),
30          new BasicDBObject("$set",
31                  new BasicDBObject("topBlogger", 1)));
```

Beispiel 28: MongoDB Statement für 2. Abfrage des Referenced Document Typs

Das Beispiel zeigt wie sich die Aggregation Pipeline zusammensetzt. In den Zeilen 1-4 wird definiert, dass die Dokumente nach den Feld "user_id" gruppiert und pro User gezählt werden sollen. In den Zeilen 6-9 wird die Sortierung bestimmt, die absteigend für die Summe der Einträge pro User und in zweiter Instanz aufsteigend nach User ID erfolgt. Die Zeilen 11-12 geben an, dass das Resultat auf den ersten Datensatz beschränkt wird. Die Zeilen 14-16 spezifizieren, dass nur das Attribut "_id" des ersten Datensatzes zurückgegeben wird. Zeile 18-21 baut aus den einzelnen Bausteinen die Pipeline zusammen, die Reihenfolge wird in Zeile 20 bestimmt.  Zeile 21 führt die Datenbankabfrage durch.

Ab Zeile 23 wird das Ergebnis ausgewertet und ab Zeile 27 wird das Update des User Dokuments mit der ermittelten ID durchgeführt und das Attribut "topBlogger" mit dem Inhalt 1 hinzugefügt.

### Embedded

```
1   <Aggregation Pipeline>
2
3   // Loop through Result From Aggregation
4   for (DBObject dbobject : output.results()) {
5       result.put("id", dbobject.get("_id").toString());
6
7       // Update User in Blog Dokument
8       blogCollection.updateMulti(
9               new BasicDBObject("user_id", dbobject.get("_id")),
10              new BasicDBObject("$set",
11                    new BasicDBObject("topBlogger", 1)));
12
13      // Update User in Subdokument Comment
14      blogCollection.updateMulti(new BasicDBObject("Comment.user_id",
15              dbobject.get("_id")), new BasicDBObject("$set",
16              new BasicDBObject("Comment.$.topBlogger", 1)));
17
18      // Update User in Subdokument Likes
19      DBCursor userCursor = blogCollection.find(new BasicDBObject(
20              "Comment.Likes.user_id", dbobject.get("_id")));
21      try {
22          while (userCursor.hasNext()) {
23              BasicDBObject user = (BasicDBObject) userCursor.next();
24
25              <get array position and write it into variable pos>
26
27              blogCollection.updateMulti(new BasicDBObject(
28                    "Comment.Likes.user_id", dbobject.get("_id")),
29                    new BasicDBObject("$set", new BasicDBObject(
30                      "Comment." + pos + ".Likes.$.topBlogger", 1)));
31          }
```

Beispiel 29: MongoDB Statement für 2. Abfrage des Embedded Document Typs

Das Suchen des Top Bloggers für den Embedded Document Typ erfolgt wie bei den Referenced Document Typ in Beispiel 28. Es wurde daher in Zeile 1 mit dem Platzhalter <Aggregation Pipeline> ersetzt.
Ab Zeile 3 wird das Ergebnis ausgewertet. Die Information des Users steht nun bei jeden von ihm verfassten Blog, geschriebenen Kommentar und verteilten Like. Es müssen daher alle Dokumente, die Informationen über den User beinhalten upgedatet werden. In Zeile 7-11 wird dies mit der Methode "updateMulti" für alle seine Blogeinträge durchgeführt. In den Zeilen 13-16 wird der User an allen Stellen upgedatet, an denen er Kommentare geschrieben hat. Da es sich bei den Kommentaren jedoch um eine Liste mit null bis n

Einträgen pro Blog handelt, muss identifiziert werden an welcher Stelle in der Liste der Kommentar des betroffenen Users steht. Dies erfolgt mit dem $-Operator in Zeile 16 (Der $-Operator fungiert als Platzhalter für das erste Element bei einer Updateoperation [61] ). In den Zeilen 21-30 erfolgt die Updateoperation für die vergebenen Likes. Da es sich bei den Likes ebenso wie bei den Kommentaren um eine Liste handelt, der $-Operator allerdings in einem Statement nur einmal Verwendung finden kann, muss die Position des Kommentares geparsed werden. Das ist in Zeile 30 ersichtlich, wo einmal die Variable "pos" die Position des Kommentars, und einmal der $-Operator die Position des Likes in den jeweiligen Listen identifiziert.

### 5.2.3 Ergebnis in MongoDB

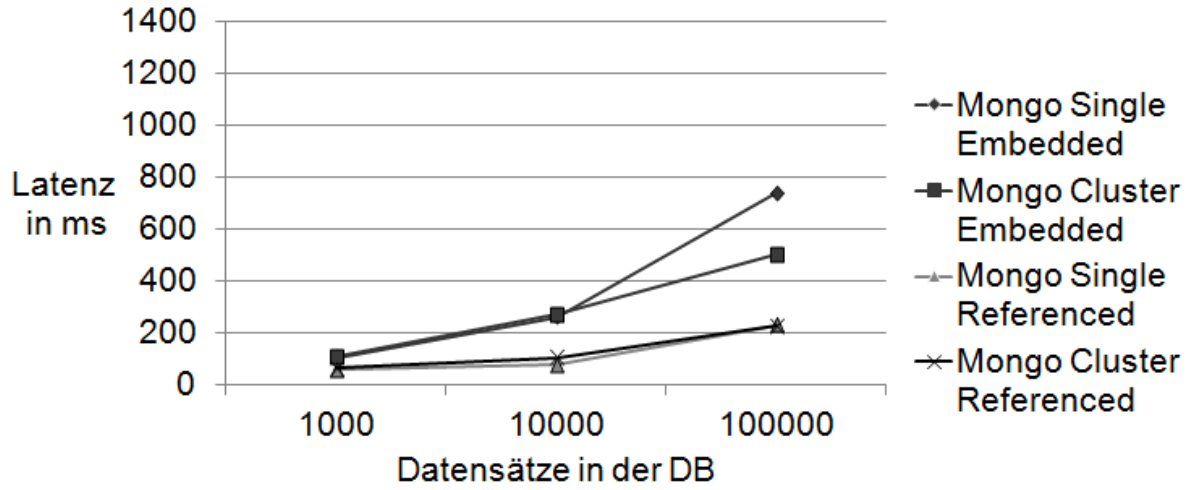



Abbildung 31: Messergebnis 2. Abfrage MongoDB

| Datensätze | MongoDB Standalone Referenced | MongoDB Cluster Referenced | MongoDB Standalone Embedded | MongoDB Cluster Embedded |
|---|---|---|---|---|
| **1000** | 58 | 66 | 104 | 110 |
| **10000** | 76 | 105 | 260 | 270 |
| **100000** | 229 | 227 | 737 | 500 |

Tabelle 7: Messergebnis 2. Abfrage MongoDB

**Diskussion**

Die Latenzzeiten der Embedded Documents zeigen höhere Werte an als die der Referenced Documents. Dies ist auf die Anzahl der Updatevorgänge bei den embedded Documents zurückzuführen (welche aufgrund der Aktivität des Users variiert). Im Gegensatz dazu ist bei den Referenced Documents nur ein Update an zentraler Stelle auszuführen.

### 5.2.4 Formulierung in Couchbase

**Referenced**

```
// Find Top Blogger
SELECT user_id FROM Blog
GROUP BY user_id
ORDER BY count(user_id) DESC, user_id
LIMIT 1;

// Set Tag
UPDATE referenced_User SET rank = ‚Top Blogger‘
WHERE user_id = „ + topBlogger + „;
```

Abbildung 32: Couchbase N1QL Statement für 2. Abfrage des Referenced Document Typs

Die Formulierung des Statements entspricht jener Formulierung, die bei der normalisierten MySQL Datenbank verwendet wurde. Es sei an dieser Stelle auf die Beschreibung des Beispiels 26 verwiesen.

**Embedded**

```
// Find Top Blogger
SELECT user_id FROM Blog
GROUP BY user_id
ORDER BY count(user_id) DESC, user_id
LIMIT 1;

// Update Blog Entries
UPDATE Blog
SET rank = ‚Top Blogger‘
WHERE user_id = „ + topBlogger + „;“

// Update Comment and Like Entries at Once
UPDATE Blog b
SET r.rank = 'TopBlogger',
FOR r WITHIN b.Comment
WHEN r.user_id = „ + topBlogger + „
END;
```

Abbildung 33: Couchbase N1QL Statement für 2. Abfrage des Embedded Document Typs

Auch der erste Teil dieses Statements ist mit dem der denormalisierten MySQL Datenbank gleich. Das Update wird in zwei Schritten durchgeführt. Zuerst wird der Tag "Top Blogger" bei allen Vorkommnissen des Users bei den Blogeinträgen gesetzt. Danach erfolgt das Update bei den Kommentaren und bei den Likes. Dies erfolgt unabhängig von der Tiefe des Dokuments und kann daher in einem Statement abgedeckt werden.

### 5.2.5 Ergebnis in Couchbase

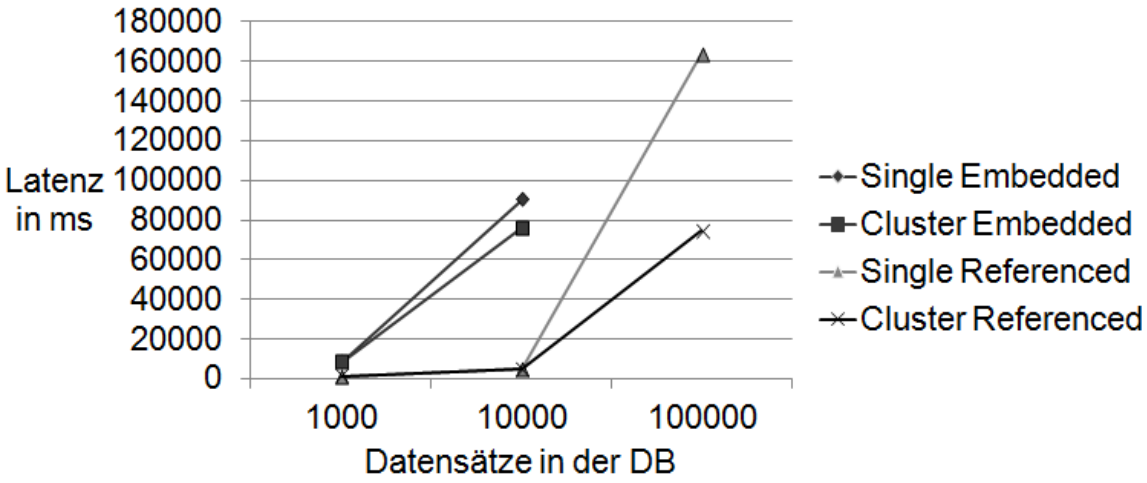


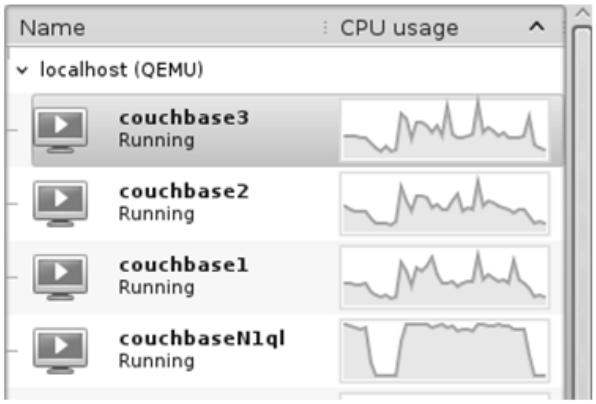

Abbildung 34: Messergebnis 2. Abfrage Couchbase N1QL

| Datensätze | Couchbase Standalone Referenced | Couchbase Cluster Referenced | Couchbase Standalone Embedded | Couchbase Cluster Embedded |
|---|---|---|---|---|
| **1000** | 886 | 1160 | 8263 | 8241 |
| **10000** | 4953 | 4674 | 90192 | 75938 |
| **100000** | 163519 | 74553 | 13951441 | 8951918 |

Tabelle 8: Messergebnis 2. Abfrage Couchbase N1QL

#### Diskussion

Auch die Ergebnisse der Couchbase zeigen das gleiche Muster wie die der MongoDB. Die Latenzzeiten sind jedoch allgemein betrachtet sehr hoch und kann auf die Auslastung des N1QL Query Servers zurückgeführt werden. Abbildung 35 zeigt die CPU Nutzung der einzelnen Server während der Laufzeit. Weiters zeigt die Abbildung einen Ausschnitt des Logfiles. Es ist ersichtlich, dass der N1QL Query Server pro Verarbeitungsschritt in etwa 900 Dokumente verarbeitet, bevor die nächsten 900 Dokumente via REST Zugriff vom Couchbase Cluster angefordert werden. Die einzelnen Verarbeitungsschritte liegen dabei bereits im Sekundenbereich, die schlussendlich aufsummiert die hohen Latenzzeiten in obiger Tabelle und Abbildung ergeben.




_time="2015-03-15T06:11:05+01:00" _level="INFO"
_msg="Request View:
http://192.168.122.120:8092/embedded_Blog_2/_design/ddl_
%23primary/_view/%23primary?limit=1001&startkey=%5B16
0%2C%5B50%2C54%2C49%2C57%2C57%5D%5D&startke
y_docid=26199"

_time="2015-03-15T06:11:06+01:00" _level="INFO"
_msg="Request View:
http://192.168.122.120:8092/embedded_Blog_2/_design/ddl_
%23primary/_view/%23primary?limit=1001&startkey=%5B16
0%2C%5B50%2C55%2C48%2C57%2C57%5D%5D&startke
y_docid=27099"

Abbildung 35: Auslastung des N1QL Query Servers

## 5.3 Dritter Benchmark

Der dritte Benchmark setzt sich mit Transaktionen auseinander. Dabei wird bei der Datenbank MySQL in Kombination mit der InnoDB Storage-Engine (alle im SQL Standard vorgesehenen Isolationsleveln werden unterstützt) und der NDB Storage-Engine (unterstützt nur Read Commited) auf das vorhandene Transaktionsmanagement zurückgegriffen. Bei MySQL in Kombination mit der MyISAM Storage-Engine als auch bei MongoDB und Couchbase wird ein Two Phase Commit Protokoll verwendet, welches in der Software implementiert wurde (Bei der Implementierung wurde zur Reduzierung des Umfangs jedoch nur der Fall betrachtet, dass kein Fehler während der Transaktion auftritt).

Das Protokoll wird notwendig, da Couchbase und MongoDB aufgrund ihres Designs keine dokumentenübergreifende Transaktionen unterstützen. Wie bereits in Kapitel 2.4.1 geschildert liegt der Fokus dieser Datenbanken auf horizontaler Skalierung und der Bewältigung von tausenden von Zugriffen zur gleichen Zeit. Um diese jedoch bewerkstelligen zu können, wird auf die Komplexität verzichtet welche für die Unterstützung von Transaktionen in verteilten Systemen notwendig wird. Auch die Storage Engine MyISAM unterstützt keine Transaktionen. Sie kommt daher oft in Applikationen ohne Anforderungen an Transaktionen zur Anwendung, und kann dort die in Kapitel 3.1.2 beschriebenen Vorteile ausspielen.

Der für diesen Benchmark spezifizierte Test tauscht zwischen zwei Usern ein sogenanntes "Goodie" aus. Dabei kann es sich beispielsweise um Bonuspunkte handeln, welche durch Aktivitäten gesammelt werden. Dieses Goodie wird vom ersten User abgezogen und auf den zweiten User verbucht.

Das Goodie ist dabei ein weiteres Attribut der Entität User. Bei allen nicht-transaktionsorientierten Datenbankmodellen wird weiters ein zusätzliches Attribut Transaktion bei der Entität User als auch eine zusätzliche Tabelle (RDBMS) oder Container (Document Store) für die Protokollierung des Transaktionsfortschrittes benötigt.

Dieser Benchmark wird nur auf der normalisierten relationalen Datenbank bzw. auf den referenced Document Typ durchgeführt. Ziel ist der Vergleich bei Durchführung des Statements mit und ohne Transaktionsunterstützung.

### 5.3.1 Formulierung in MySQL

#### ohne Transaktionsmanagement

```
1 // Update Item Count
2 UPDATE User SET itemcount = itemcount + 5 WHERE id_user = 1
3
4 // Update Item Count
5 UPDATE User SET itemcount = itemcount - 5 WHERE id_user = 2
```

Beispiel 30: SQL Statement 3. Abfrage ohne Transaktionsmanagement

Beispiel 30 zeigt die Durchführung ohne Transaktionsmanagement. Dabei wird vom User mit der ID1 der Itemcount um 5 erhöht und dieser vom User mit der ID 2 um 5 verringert.

#### mit Transaktionsmanagement in InnoDB und NDB

```
1 // Create Connection
2 con = DriverManager.getConnection(url, user, password);
3 con.setAutoCommit(false);
4
5 // Update Item Count
6 UPDATE User SET itemcount = itemcount + 5 WHERE id_user = 1
7
8 // Update Item Count
9 UPDATE User SET itemcount = itemcount - 5 WHERE id_user = 2
10
11con.commit();
```

Beispiel 31: SQL Statement 3. Abfrage mit Transaktionsmanagement InnoDB und NDB

Beispiel 31 zeigt, wie eine Transaktion in der MySQL Datenbank mit den Storage-Engines abgewickelt wird, welche Support für Transaktionen bieten. Dabei wird das Autocommit Feld in Zeile 3 auf "false" gesetzt, und nach der Ausführung aller in der Transaktion involvierten Statements ein "commit" abgesetzt.

**mit Transaktionsmanagement in MyISAM**

Beispiel 32 beschreibt die Durchführung mittels des Two Phase Commit Protocols. Dabei wird in den Zeilen 2-5 die Transaktion erstellt und auf "Init" gesetzt. Damit ist diese Transaktion gereiht, und kann abgearbeitet werden. In den Zeilen 7-9 wird die Transaktion auf "Pending" gesetzt, was darauf hinweist, dass sich die Transaktion in Bearbeitung befindet. In den Zeilen 11-13 und 15-17 wird das Attribut Transaktion der betroffenen User auf 1 gesetzt und gleichzeitig der Itemcount manipuliert. Geschieht dies erfolgreich wird in den Zeilen 19-21 die Transaktion auf "Committed" gesetzt und anschließend in den Zeilen 23-24 und 26-27 das Attribut Transaktion für beide User von 1 auf 0 gesetzt. Damit ist die Transaktion abgeschlossen und wird in den Zeilen 29-31 auf den Status "Done" gesetzt.

```
1  // Create Transaction Object and set state to INIT
2  INSERT INTO Transaction(
3   transaction_id, from_user_id, to_user_id,
4   itemcount, state)
5  VALUES (1,2,1,5,"INIT")
6
7  // Set State to PENDING
8  UPDATE Transaction SET state = "PENDING"
9  WHERE transaction_id = 1
10
11 // Set Transaction to 1 and Update Item Count on first User
12 UPDATE User SET transaction = 1,
13 itemcount = itemcount + 5 WHERE id_user = 1
14
15 // Set Transaction to 1 and Update Item Count on second User
16 UPDATE User SET transaction = 1,
17 itemcount = itemcount - 5 WHERE id_user = 2
18
19 // Set State to COMMITTED
20 UPDATE Transaction SET state = "COMMITTED"
21 WHERE transaction_id = 1
22
23 // Set Transaction to 0 on first User
24 UPDATE User SET transaction = 0 WHERE id_user = 1
25
26 // Set Transaction to 0 on second User
27 UPDATE User SET transaction = 0 WHERE id_user = 2
28
29 // Set State to DONE
30 UPDATE Transaction SET state = "DONE"
31 WHERE transaction_id = 1
```

Beispiel 32: SQL Statement 3. Abfrage mit Transaktionsmanagement MyISAM

### 5.3.2 Ergebnis in MySQL

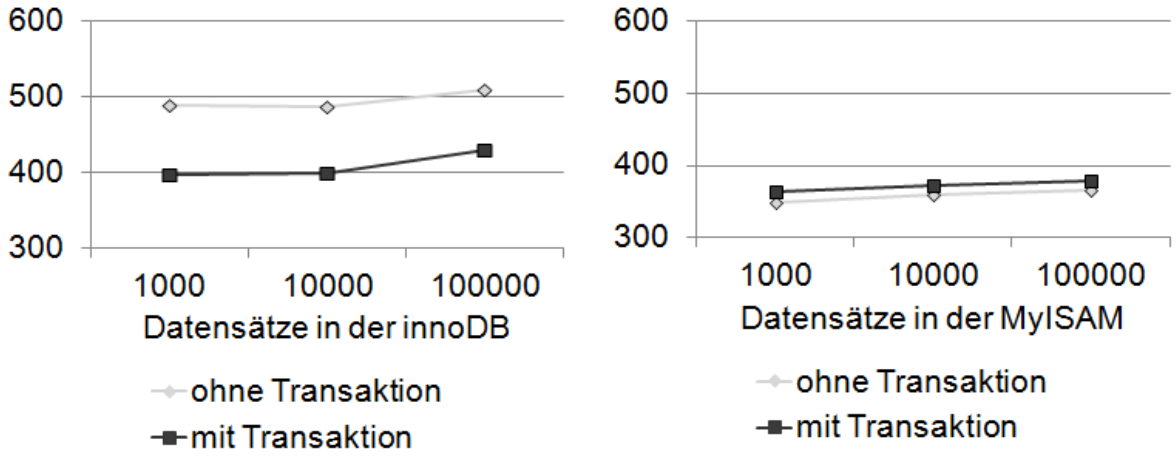



Abbildung 36: MySQL 3. Abfrage Vergleich von InnoDB und MyISAM

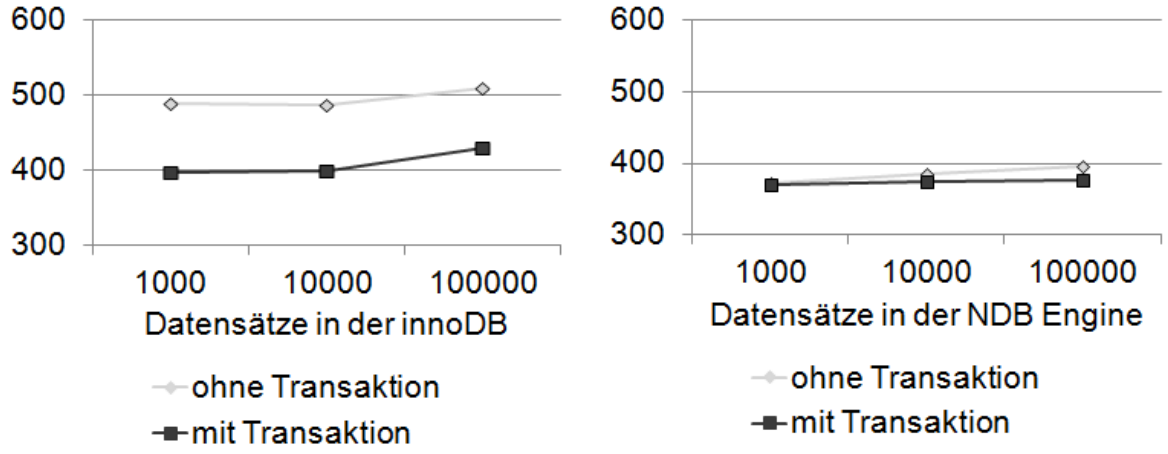



Abbildung 37: MySQL 3. Abfrage Vergleich von InnoDB und NDB

| Datensätze | InnoDB ohne Trans. | InnoDB mit Trans. | MyISAM ohne Trans. | MyISAM mit Trans. | NDB ohne Trans. | NDB mit Trans. |
|---|---|---|---|---|---|---|
| **1000** | 488 | 396 | 349 | 363 | 373 | 371 |
| **10000** | 496 | 399 | 360 | 372 | 384 | 375 |
| **100000** | 510 | 430 | 365 | 378 | 396 | 377 |

Tabelle 9: Messergebnis 3. Abfrage MySQL

**Diskussion**

Abbildung 36 vergleicht die Ergebnisse der beiden Standalone Systeme InnoDB und MyISAM. Es zeigt, dass die transaktionsorientierte Storage Engine InnoDB eine Transaktion bestehend aus mehreren Operationen schneller ausführen kann, als die gleiche Anzahl an Operationen ohne Formulierung einer Transaktion. Im Gegensatz dazu wirkt sich die

Implementierung des Two Phase Commit Protocols bei der MyISAM Engine negativ auf die Latenz aus, da nun mehrere Zugriffe auf die Datenbank notwendig werden als wenn auf die Transaktionssicherheit verzichtet wird.
Abbildung 37 stellt die beiden transaktionsorientierten Storage Engines InnoDB und NDB gegenüber. Bei beiden sind Operationen zusammengefasst zu einer Transaktion effizienter in der Ausführung als ohne dieser Transaktion. Im Gegensatz zur InnoDB, bei welchen der Performanceunterschied auch bei wachsender Anzahl an Datensätzen konstant bleibt, zeigt sich bei NDB bei Zunahme der Datensätze eine höhere Effizienz bei der Transaktion.

### 5.3.3 Formulierung in MongoDB

#### ohne Transaktionsmanagement

```
1   // Update First Document
2   BasicDBObject userQuery = new BasicDBObject("id", 1);
3   DBObject userUpdate = new BasicDBObject();
4   userUpdate.put("$pull", new BasicDBObject("item", item));
5   userCollection.update(userQuery, userUpdate);
6
7   // Update Second Document
8   userQuery = new BasicDBObject("id", 2);
9   userUpdate = new BasicDBObject();
10  userUpdate.put("$push", new BasicDBObject("item", item));
11  userCollection.update(userQuery, userUpdate);
```

Beispiel 33: MongoDB 3. Abfrage ohne Transaktionsmanagement

Beispiel 33 zeigt den Vorgang ohne Transaktionsmanagement. Dabei wird zuerst vom User mit der ID 1 in den Zeilen 1-5 das Item abgezogen und dem User mit der ID 2 in den Zeilen 7-11 verbucht.

#### mit Transaktionsmanagement

Die Vorgehensweise ist grundsätzlich gleich wie bei der Datenbank MySQL in Kombination mit der Storage-Engine MyISAM. Es wird zuerst ein Transaktionsobjekt in der Collection transaction angelegt und auf "Init" gesetzt. Dies ist in den Zeilen 1-8 des Beispiels 34 zu sehen. Damit ist die Transaktion in die Queue aufgenommen. In den Zeilen 10-14 wird der Status dieser Transaktion von "Init" auf "Pending" geändert, womit sich die Transaktion in der Abarbeitung befindet. Die Zeilen 16-21 und 23-28 setzen das Attribut "transaction" der beiden User Dokumente auf 1 und nehmen im selben Schritt das Umbuchen des Items vor. Dies kann im gleichen Prozess geschehen, da es sich um eine atomare Operation handelt. Sobald die Umbuchung erfolgt ist, wird die Transaktion in den Zeilen 30-35 auf den Status "Committed" gesetzt. In den Zeilen 37-41 und 43-47 wird das Attribut "transaction" der beiden User Dokumente wieder auf 0 zurückgesetzt. Die Transaktion ist damit beendet, und das Transaktionsobjekt wird auf den Status "Done" gesetzt.

```
1   // Create Transaction Object and set state to INIT
2   Map<String, Object> transaction = new HashMap<String, Object>();
3   transaction.put("id", 1);
4   transaction.put("from", 1);
5   transaction.put("to", 2);
6   transaction.put("item", "sword");
7   transaction.put("state", "init");
8   transactionCollection.insert(new BasicDBObject(transaction));
9
10  // Set State to PENDING
11  transactionQuery = new BasicDBObject("id", id);
12  DBObject transactionUpdate = new BasicDBObject();
13  transactionUpdate.put("$set", new BasicDBObject("state",
      "pending"));
14  transactionCollection.update(transactionQuery, transactionUpdate);
15
16  // Update First Document and Set Transaction to 1
17  BasicDBObject userQuery = new BasicDBObject("id", from);
18  DBObject userUpdate = new BasicDBObject();
19  userUpdate.put("$pull", new BasicDBObject("item", item));
20  userUpdate.put("$set", new BasicDBObject("transaction", 1));
21  userCollection.update(userQuery, userUpdate);
22
23  // Update Second Document and Set Transaction to 1
24  userQuery = new BasicDBObject("id", to);
25  userUpdate = new BasicDBObject();
26  userUpdate.put("$push", new BasicDBObject("item", item));
27  userUpdate.put("$set", new BasicDBObject("transaction", 1));
28  userCollection.update(userQuery, userUpdate);
29
30  // Set State to COMMITTED
31  transactionQuery = new BasicDBObject("id", id);
32  transactionUpdate = new BasicDBObject();
33  transactionUpdate.put("$set",
34      new BasicDBObject("state", "committed"));
35  transactionCollection.update(transactionQuery, transactionUpdate);
36
37  // Update First Document and Set Transaction to 0
38  userQuery = new BasicDBObject("id", from);
39  userUpdate = new BasicDBObject();
40  userUpdate.put("$set", new BasicDBObject("transaction", 0));
41  userCollection.update(userQuery, userUpdate);
42
43  // Update Second Document and Set Transaction to 0
44  userQuery = new BasicDBObject("id", to);
45  userUpdate = new BasicDBObject();
46  userUpdate.put("$set", new BasicDBObject("transaction", 0));
47  userCollection.update(userQuery, userUpdate);
48
49  // Retrieve Transaction Object and set State to DONE
50  transactionQuery = new BasicDBObject("id", id);
51  transactionUpdate = new BasicDBObject();
52  transactionUpdate.put("$set", new BasicDBObject("state", "done"));
```

Beispiel 34: MongoDB 3. Abfrage mit Transaktionsmanagement

### 5.3.4 Ergebnis in MongoDB

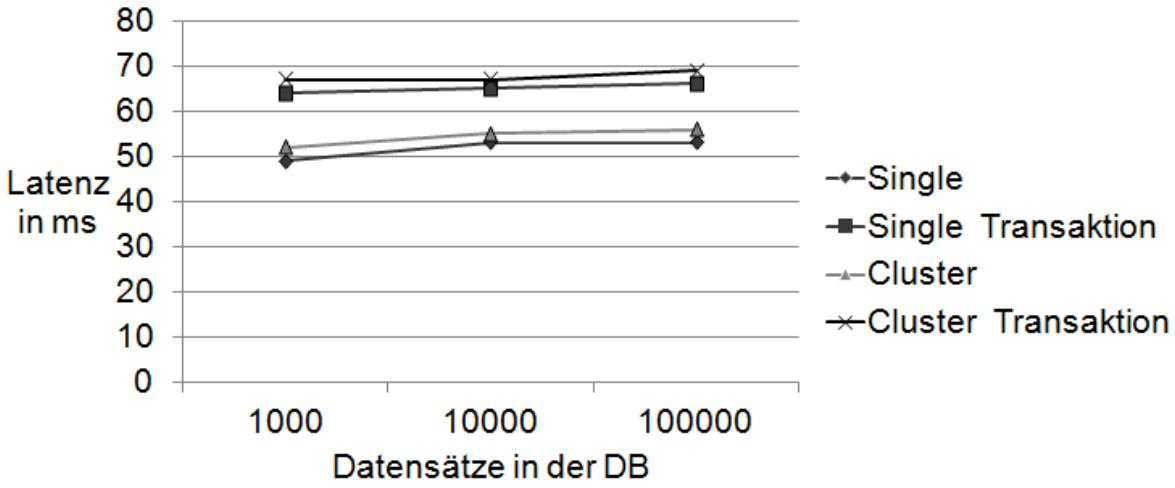



Abbildung 38: MongoDB 3. Abfrage

| Datensätze | MongoDB Singe ohne Trans. | MongoDB Single mit Trans. | MongoDB Cluster ohne Trans. | MongoDB Cluster mit Trans. |
|---|---|---|---|---|
| **1000** | 49 | 64 | 52 | 67 |
| **10000** | 53 | 65 | 55 | 67 |
| **100000** | 53 | 66 | 56 | 69 |

Tabelle 10: Messergebnis 3. Abfrage MongoDB

**Diskussion**

Da MongoDB keine dokumentenübergreifende Transaktionen unterstützt, wurde clientseitig ein Two Phase Commit Protocol verwendet. Mit diesen sind jedoch zusätzliche Anfragen an die Datenbank notwendig, was sich negativ auf die Latenz des Statements auswirkt. So sind die Operationen welche ohne Transaktionsmanagement auf clientseite ausgeführt werden in etwa um 15 - 20% effizienter, als jene welche das 2PC Protokoll verwenden.

### 5.3.5 Formulierung in Couchbase

Dieser Benchmark wurde mit dem Couchbase SDK realisiert.

#### ohne Transaktionsmanagement

Die Vorgangsweise in Beispiel 35 ist wie bei MongoDB in Beispiel 33, jedoch verfügt der Couchbase SDK nicht über einen $pull-Operator wie bei MongoDB. Dadurch ist es notwendig eine neue Liste zu erstellen um das Item vom ersten User zu entfernen. Dies geschieht in den Zeilen 7-11. Das Verbuchen des Items beim zweiten User erfolgt mit dem $add Operator in Zeile 19.

```
1  // Update First Document
2  // Update the Item List -->
3  // Loop through old List and create new one without the item
4  JsonObject fromUser = userBucket.get(from).content();
5  List<String> myList = new ArrayList<String>();
6  List<Object> itemObjects = fromUser.getArray("item").toList();
7  for (Object a : itemObjects) {
8      if (!a.toString().equals(item)) {
9          myList.add(a.toString());
10     }
11 }
12 fromUser.put("item", myList);
13 JsonDocument doc = JsonDocument.create(
14     String.valueOf(from), fromUser);
15 userBucket.upsert(doc);
16
17  // Update Second Document
18 JsonObject toUser = userBucket.get(to).content();
19 toUser.getArray("item").add(item);
20 doc = JsonDocument.create(String.valueOf(to), toUser);
```

Beispiel 35: Couchbase SDK 3. Abfrage ohne Transaktionsmanagement

#### mit Transaktionsmanagement

Auch das Beispiel 36 ist dem Beispiel 34 der MongoDB sehr ähnlich. Es wird zu Beginn wieder das Transaktionsobjekt erzeugt. Anstatt es aber zuerst in den Status "Init" und in einem weiteren Schritt auf "Pending" zu setzen erfolgt dies in Beispiel 36 im gleichen Schritt. Damit muss die Transaktion aber auch gleich vom selben Prozess bearbeitet werden. Es werden anschließend in den Zeilen 9-21 und 23-28 die beiden User Dokumente upgedatet, indem das Attribut "transaction" auf 1 gesetzt und das Item umgebucht wird. Danach wird die Transaktion in den Zeilen 30-34 auf "Committed" gesetzt. In den Zeilen 36-40 und 42-46 wird das Attribut "transaction" der beiden User Dokumente wieder auf 0 zurückgesetzt und in den Zeilen 48-52 die Transaktion auf den Status "Done" gesetzt.

```
// Create Transaction Object and set state to PENDING (not INIT)
JsonObject transaction = JsonObject.empty().put("id", "1")
        .put("from", fromUserId).put("to", toUserId)
        .put("item", transactionItem).put("state", "pending");
JsonDocument doc = JsonDocument.create(
    String.valueOf(1), transaction);
transactionBucket.upsert(doc);

// Update First Document and Set Transaction to 1
JsonObject fromUser = userBucket.get(from).content();
List<String> myList = new ArrayList<String>();
List<Object> itemObjects = fromUser.getArray("item").toList();
for (Object a : itemObjects) {
    if (!a.toString().equals(item)) {
        myList.add(a.toString());
    }
}
fromUser.put("item", myList);
fromUser.put("transaction", "1");
doc = JsonDocument.create(String.valueOf(from), fromUser);
userBucket.upsert(doc);

// Update Second Document and Set Transaction to 1
JsonObject toUser = userBucket.get(to).content();
toUser.getArray("item").add(item);
toUser.put("transaction", "1");
doc = JsonDocument.create(String.valueOf(to), toUser);
userBucket.upsert(doc);

// Retrieve Transaction Object and set State to COMMITTED
transaction = transactionBucket.get("1").content();
transaction.put("state", "committed");
doc = JsonDocument.create(String.valueOf(1), transaction);
transactionBucket.upsert(doc);

// Update First Document and Set Transaction to 0
fromUser = userBucket.get(from).content();
fromUser.put("transaction", "0");
doc = JsonDocument.create(String.valueOf(from), fromUser);
userBucket.upsert(doc);

// Update Second Document and Set Transaction to 0
toUser = userBucket.get(to).content();
toUser.put("transaction", "0");
doc = JsonDocument.create(String.valueOf(to), toUser);
userBucket.upsert(doc);

// Retrieve Transaction Object and set State to DONE
transaction = transactionBucket.get("1").content();
transaction.put("state", "done");
doc = JsonDocument.create(String.valueOf(1), transaction);
transactionBucket.upsert(doc);
```

Beispiel 36: Couchbase SDK 3. Abfrage mit Transaktionsmanagement

### 5.3.6 Ergebnis in Couchbase

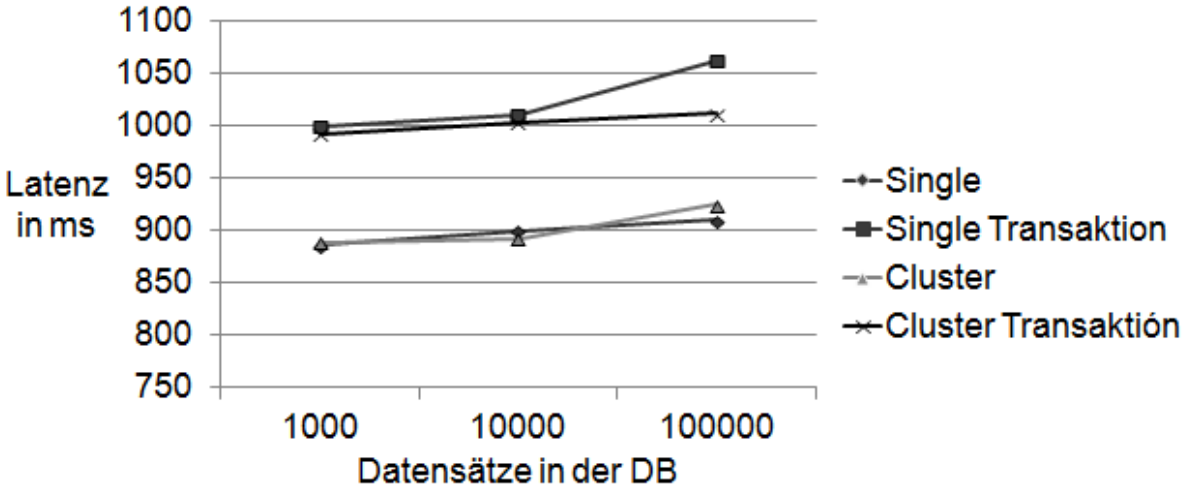



Abbildung 39: Couchbase 3. Abfrage

| Datensätze | Couchbase Single ohne Trans. | Couchbase Single mit Trans. | Couchbase Cluster ohne Trans. | Couchbase Cluster mit Trans. |
|---|---|---|---|---|
| **1000** | 886 | 1000 | 889 | 992 |
| **10000** | 900 | 1011 | 892 | 1003 |
| **100000** | 910 | 1062 | 925 | 1012 |

Tabelle 11: Messergebnis 3. Abfrage Couchbase

**Diskussion**

Auch Couchbase bietet wie MongoDB keine Transaktionssicherheit zwischen dokumentenübergreifenden Operationen und es muss daher ebenso auf eine clientseitige Implementierung des Two Phase Commit Protocol zurückgegriffen werden. Dies resultiert ebenso wie in MongoDB in einer höheren Latenz aufgrund der zusätzlichen Zugriffe auf die Datenbank welche für das clientseitige Transaktionsmanagement notwendig werden. Operationen welche ohne den 2PC Protokolls ausgeführt werden, sind hier in etwa um 10% effizienter.

# 6 Related Work

[62] beschäftigt sich ebenso mit dem Vergleich von relationalen und non-relationalen Datenbanksystemen. Dabei wird nach einer ausführlichen theoretischen Gegenüberstellung von relationalen Datenbanken mit Document Stores ein Benchmark durchgeführt, welcher die beiden Datenbanksysteme MySQL in Kombination mit MyISAM als Storage-Engine und MongoDB beleuchtet. In drei leicht abweichenden Testszenarien wird untersucht, wie schnell die beiden Datenbanken nach Artikeln mit bestimmten Kriterien suchen können. Dabei werden die Felder Titel, Unterüberschrift und Inhalt für die Tests herangezogen. Als Basis für die Tests dient ein Teil des Datenbestands von Wikipedia, der in die Datenbanken importiert wird. Im Unterschied zur vorliegenden Arbeit werden dabei nur Leseoperationen untersucht. Des Weiteren beschränkt sich die Arbeit auf eine Relation in MySQL bzw. auf eine Collection in MongoDB. Es findet somit keine Performancemessung von Verbundoperationen und Transaktionen statt. Im Gegensatz zur vorliegenden Arbeit, welche die unterschiedlichen Datenmodellierungen der einzelnen Datenbanken vergleicht, werden in [62] die Ergebnisse von MySQL und MongoDB direkt gegenübergestellt. Dazu wird einleitend festgehalten, dass aufgrund der verschiedenen APIs, Indexierungsfunktionen, Wildcardsuchfunktionen und Skalierungsmöglichkeiten kein fairer Vergleich möglich ist, jedoch trotzdem erkannt werden kann, wo sich die Systeme bewegen. Die Ergebnisse zeigen das bei einfachen Leseoperationen MySQL performanter ist. Jedoch erweist sich bei komplexeren Leseoperationen die strukturelle Flexibilität des Document Stores als Performancevorteil. Auch die in dieser Arbeit durchgeführten Leseoperationen in Kapitel 5.1 zeigen diesen Vorteil und bestätigen dadurch die Aussagen des Autors.

[63] beschäftigt sich mit der Verwendung von MongoDB als Backend für eine Social Networking Website. Die Struktur der Website ist dabei ähnlich der für diese Arbeit simulierten Blogging Plattform. So können sich User ebenfalls registrieren, Blogs verfassen, Kommentare schreiben und Likes verteilen. Des Weiteren werden noch Profile verwaltet und Freundschaften zwischen den Usern ermöglicht. Die Arbeit geht weiters auf den Punkt API ein, womit eine Interaktion mit anderen Webseiten ermöglich wird. Bei der Beleuchtung von den Vor- und Nachteilen der MongoDB Datenbank kann eine große Übereinstimmung mit vorliegender Arbeit festgestellt werden. Es wird beschrieben, dass Joins von relationalen Datenbanken mittels embedded Documents und referenced Documents in MongoDB abgebildet werden können. Eine tiefere Diskussion findet jedoch nicht statt. Vorliegende Arbeit zeigt in Kapitel 5.2.2 die Schwierigkeit beim Abfragen von embedded Documents, dies wird in [63] ebenfalls beschrieben. Des Weiteren wird festgehalten, dass Verbundoperationen zwischen referenced Documents in MongoDB applikationsseitig im Code erfolgen müssen, was nicht nur für die Entwicklung der Abfragen aufwendig ist, sondern auch die Latenz durch die zusätzliche Round Trip Time erhöht. Auch das fehlende Transaktionsmanagement wird als Nachteil festgehalten. Beide Punkte bestätigen die Aussagen vorliegender Arbeit.

[64] führt einen Benchmarktest auf den beiden Datenbanken MySQL und MongoDB durch. Das Ziel der Arbeit war dabei die Skalierbarkeit der beiden Datenbanken für ein Entertainment Service herauszufinden. Beide Systeme wurden im Standalone Betrieb und MongoDB weiters als Cluster bestehend aus zwei Shards auf handelsüblicher Hardware konfiguriert. MySQL wurde sowohl mit der Option AutoCommit enabled als auch mit der Option AutoCommit disabled durchgeführt (was einer Transaktion gleichkommt). Im Kontrast zu vorliegender Arbeit wurden bei der Durchführung der Messung fünf Messpunkte festgehalten. So wurden das Senden der Abfrage, der Empfang der Abfrage, die Durchführung auf der Datenbank, das Senden der Abfrage und der Empfang der Abfrage festgehalten. Der Zugriff auf die MongoDB Datenbank erfolgte dabei über den MongoDB Java Treiber und der MySQL Datenbankzugriff wurde über den JDBC Treiber realisiert. In Kontrast zur vorliegenden Arbeit wurden nicht nur Single Threaded Tests, sondern auch Multi Threaded Tests durchgeführt. [64] zeigt bei MySQL wesentlich bessere Ergebnisse mit der Option AutoCommit disabled. Auch vorliegende Arbeit zeigt in Kapitel 5.3.2 den Performancegewinn von transaktionsunterstützten Operationen bei den Storage-Engines InnoDB und NDB. Des Weiteren zeigt [64] bei der Untersuchung von MongoDB schlechtere Performancewerte beim Sharded Cluster im Gegenzug zur Standalone Instanz. Auch in vorliegender Arbeit wies der MongoDB Cluster höhere Latenzzeiten als der Standalone Server auf.

# 7 Zusammenfassung

Diese Arbeit ging auf die Vor- und Nachteile der unterschiedlichen Modellierungen von relationalen Datenbanken und Document Stores ein und untersuchte die Latenzzeiten der relationalen Datenbank MySQL in Kombination mit InnoDB, MyISAM und NDB als auch die Document Stores MongoDB und Couchbase, welche sowohl als Standalone Systeme als auch als Cluster in einem Testenvironment installiert wurden. Die Messungen zeigten dabei abhängig von der gewählten Datenmodellierung teilweise erhebliche Unterschiede auf. So wurde ersichtlich, dass Verbundoperationen bei Lesestatements wesentlich schlechter skalieren als die Anwendung des gleichen Statements auf eine denormalisierte Datenbank. Bei der Untersuchung von Updateoperationen konnte der Vorteil von normalisierten Datenbanken unter Beweis gestellt werden, welche nicht nur schneller in der Durchführung der Operationen sind, sondern auch Datenanomalien verhindern. Auch bei den Document Stores konnte der Vorteil von den embedded Documents bei Leseoperationen im Vergleich zu den referenced Documents anhand des Performancevergleichs verifiziert werden. Ebenso war dies für die referenced Documents bei den Schreiboperationen möglich. Aus den vorhandenen Daten kann geschlossen werden, dass der effizienteste Weg zur Modellierung eines Blogs durch eine Kombination von referenced Documents und embedded Documents liegt. Dabei bietet ein embedded Document welches sowohl den Blog, als auch alle zu den Blog gehörigen Kommentare und zu den Kommentaren gehörigen Likes umfasst und auf die jeweiligen User referenziert größtmögliche Flexibilität bei gleichzeitiger Vermeidung von Redundanzen. Diese Vorteile können auf den relationalen Datenbanken nicht in gleichen Maß abgebildet werden. In der dritten Messung konnte jedoch die Nützlichkeit und der Performancevorteil der InnoDB und NDB Storage-Engine aufgrund der Transaktionsunterstützung auf der Datenbankebene gezeigt werden. Ein Vorteil der relationalen Datenbanken aufgrund deren Fokussierung auf das ACID Prinzip vorbehalten bleibt. Abhängig von den Anforderungen sind daher mehrere richtige Entscheidungen möglich. Die Ergebnisse dieser Arbeit unterstreichen jedenfalls die Wichtigkeit des Designprozesses und der damit verbundenen Überlegungen.

## 7.1 Future Work

Aufgrund des exponentiellen Wachstums von Datenbeständen und datenintensiven Applikationen sowie deren Komplexität sind in den letzten Jahren viele neue Datenbanken entstanden. Vorliegende Arbeit untersucht die Datenmodellierung im Bereich der relationalen Datenbanken und Document Stores, für einen weiteren Blickwinkel wäre jedoch auch die Betrachtung weiterer Datenbanken aus dem NoSQL und NewSQL Bereich angebracht. Des Weiteren gibt es unterschiedliche Benchmarks, unter welchen die Untersuchung durchgeführt werden kann. So könnten Stress- und Lasttests neue Erkenntnisse liefern und die vorliegende Arbeit sinnvoll ergänzen.

# Literaturverzeichnis


[1] Gunter Saake, Kai-Uwe Sattler. Datenbanken – Konzepte und Sprachen. 5. Auflage, ISBN: 978-3826694530, mitp (2013).

[2] Roland Gabriel. Datenbankmanagementsystem, http://www.enzyklopaedie-der-wirtschaftsinformatik.de/wi-enzyklopaedie/lexikon/daten-wissen/Datenmanagement/Datenbanksystem/Datenbankmanagementsystem (letzter Zugriff am 4.4.2015).

[3] Tom Jewett. Database Design with UML and SQL. 3. Auflage, http://freecomputerbooks.com/Database-Design-with-UML-and-SQL-3rd-Edition.html (letzter Zugriff am 4.4.2015).

[4] Gunter Saake, Kai-Uwe Sattler, Andreas Heuer. Datenbanken – Implementierungstechniken. 3. Auflage, ISBN: 978-3826691560, mitp (2011)

[5] Elvis Foster, Shripad Godbole. Database Systems – A Pragmatic Approach. 1. Auflage, ISBN: 978-1-484-20878-6, Apress (2014)

[6] Shio Kumar Singh. Database Systems – Concepts, Design and Applications. 2. Auflage, ISBN: 978-81-317-6092-5, Pearson India (2011)

[7] Jose Samos, Felix Saltor, Jaume Sistac, Agusti Bardes. Database Architecture for Data Warehousing – An Evolutionary Approach. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.40.8532&rep=rep1&type=pdf (Letzter Zugriff am 5.4.2015)

[8] Edgar F. Codd. Data Models in Database Management. http://jpkc.yzu.edu.cn/course2/sjkyljyy/dzjc/PART%201%20-%20The%20Relational%20Model%20and%20Relational%20Databases/%28POD-R1-2%29%20Data%20models%20in%20database%20management.pdf (Letzter Zugriff am 5.4.2015)

[9] Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf (Letzter Zugriff am 29.4.2015)

[10] Michael J. Hernandez. Database Design for Mere Mortals - A Hands-On Guide to Relational Database Design. 3. Auflage, ISBN: 978-0321884497, Addison-Wesley Professional (2013)

[11] DB-Engines, http://db-engines.com/ (Letzter Zugriff am 28.4.2015)

[12] Susan Sales Harkins, Martin W.P. Reid. SQL - Acess to SQL Server. http://link.springer.com/chapter/10.1007/978-1-4302-1573-8_1 (Letzter Zugriff am 28.4.2015)

[13] ISO 9075 Standards Catalog, http://www.iso.org/iso/home/search.htm?qt=9075&published=on&active_tab=standards&sort_by=rel (Letzter Zugriff am 28.4.2015)

[14] Toby J. Teorey, Sam S. Lightstone, Tom Nadeau, H.V. Jagadish. Database Modeling and Design - Logical Design. 5. Auflage, ISBN: 978-0123820204. Morgan Kaufmann (2011)

[15] Hermann Sauer. Relationale Datenbanken - Theorie und Praxis. ISBN: 978-3827320605, Addison-Wesley (2002)

[16] Jan L. Harrington. Relational Database Design and Implementation. 3. Auflage. ISBN: 978-0123747303. Morgan Kaufmann (2009)

[17] Normalisierung (Datenbank). http://de.wikipedia.org/wiki/Normalisierung_%28Datenbank%29 (Letzter Zugriff am 28.4.2015)

[18] Denormalisierung. http://de.wikipedia.org/wiki/Denormalisierung (Letzter Zugriff am 28.4.2015)

[19] Charles Roe. The Question of Database Transaction Processing: An ACID, BASE, NoSQL Primer. DataVersity (2013)

[20] Charakteristika und Vergleich von SQL- und NoSQL- Datenbanken. http://dbs.uni-leipzig.de/file/seminar_1112_tran_ausarbeitung.pdf. (Letzter Zugriff am 29.4.2015)

[21] MUMPS Programming Language. http://foldoc.org/MUMPS (Letzter Zugriff am 29.4.2015)

[22] Notes Storage Facility (NSF). http://www.forensicswiki.org/wiki/Notes_Storage_Facility_%28NSF%29 (Letzter Zugriff am 29.4.2015)

[23] Exploring CouchDB. http://www.ibm.com/developerworks/opensource/library/os-couchdb/index.html (Letzter Zugriff am 28.4.2015)

[24] JSON Schema - core definitions and terminology. http://json-schema.org/latest/json-schema-core.html (Letzter Zugriff am 28.4.2015)

[25] Steven van Ombergen. A Comparison of Five Document-Store Query Languages - Finding a Suitable Standard. http://dare.uva.nl/cgi/arno/show.cgi?fid=524062 (Letzter Zugriff am 29.4.2015)

[26] Rick Copeland. MongoDB Applied Design Patterns. ISBN: 978-1449340049. O'Reilly Media (2013)

[27] Marco Emrich. Datenbanken & SQL für Einsteiger - Datenbankdesign und MySQL in der Praxis. ISBN: 978-1492951049. CreateSpace Independent Publishing Platform (2013)

[28] Charles Bell. Expert MySQL. ISBN: 978-1430246596. 2. Auflage. Apress (2012)

[29] The InnoDB Storage Engine. http://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html (Letzter Zugriff am 1.5.2015)

[30] The MyISAM Storage Engine. http://dev.mysql.com/doc/refman/5.7/en/myisam-storage-engine.html (Letzter Zugriff am 1.5.2015)

[31] MySQL Engines: InnoDB vs. MyISAM - A Comparison of Pros and Cons. http://www.kavoir.com/2009/09/mysql-engines-innodb-vs-myisam-a-comparison-of-pros-and-cons.html (Letzter Zugriff am 1.5.2015)

[32] MySQL Cluster Overview: https://dev.mysql.com/doc/refman/5.5/en/mysql-cluster-overview.html (Letzter Zugriff am 1.5.2015)

[33] Differences Between the NDB and InnoDB Storage Engine. https://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-ndb-innodb-engines.html (Letzter Zugriff am 1.5.2015)
[34] DB-Engines Ranking. http://db-engines.com/en/ranking (Letzter Zugriff am 1.5.2015)
[35] Update on nested Documents. https://forums.couchbase.com/t/update-on-nested-documents/3491 (Letzter Zugriff am 1.5.2015)
[36] Couchbase N1QL - Update on nested Documents. http://stackoverflow.com/questions/29581646/couchbase-n1ql-update-on-nested-documents (Letzter Zugriff am 1.5.2015)
[37] Update on deeply nested Documents. https://forums.couchbase.com/t/update-on-deeply-nested-documents/3546/2 (Letzter Zugriff am 1.5.2015)
[38] DB-Engines Ranking of Document Stores. http://db-engines.com/en/ranking/document+store (Letzter Zugriff am 1.5.2015)
[39] Couchbase Server. http://en.wikipedia.org/wiki/Couchbase_Server (Letzter Zugriff am 1.5.2015)
[40] Couchbase Server Relase Notes. http://docs.couchbase.com/admin/admin/rel-notes/rel-notes3.0.html (Letzter Zugriff am 1.5.2015)
[41] N1QL Language Reference, DP4. http://docs.couchbase.com/prebuilt/n1ql/n1ql-dp4/N1QLRef-DP4.pdf (Letzter Zugriff am 1.5.2015)
[42] What's Coming in Couchbase Server 4.0. http://www.couchbase.com/coming-in-couchbase-server-4-0 (Letzter Zugriff am 1.5.2015)
[43] Inside MongoDB - the Internals of an Open-Source Database. http://www.slideshare.net/mdirolf/inside-mongodb-the-internals-of-an-opensource-database (letzter Zugriff am 1.5.2015)
[44] BSON. http://bsonspec.org/ (letzter Zugriff am 1.5.2015)
[45] MongoDB Glossary. http://docs.mongodb.org/manual/reference/glossary/ (letzter Zugriff am 1.5.2015)
[46] Replica Set Secondary Members. http://docs.mongodb.org/manual/core/replica-set-secondary/ (Letzter Zugriff am 1.5.2015)
[47] Sharding Introduction. http://docs.mongodb.org/manual/core/sharding-introduction/ (Letzter Zugriff am 1.5.2015)
[48] Database Operations. http://docs.mongodb.org/manual/core/crud-introduction/ (Letzter Zugriff am 2.5.2015)
[49] Write Operation Overview. http://docs.mongodb.org/manual/core/write-operations-introduction/ (Letzter Zugriff am 2.5.2015)
[50] Aggregation Pipeline. http://docs.mongodb.org/manual/core/aggregation-pipeline/ (Letzter Zugriff am 2.5.2015)
[51] MySQL Standard Compliance. https://dev.mysql.com/doc/refman/5.7/en/compatibility.html (Letzter Zugriff am 2.5.2015)

[52] Selecting SQL Modes. https://dev.mysql.com/doc/refman/4.1/en/sql-mode.html (Letzter Zugriff am 2.5.2015)
[53] MySQL Extensions to Standard SQL. https://dev.mysql.com/doc/refman/5.7/en/extensions-to-ansi.html (Letzter Zugriff am 2.5.2015)
[54] MC Brown. Developing with Couchbase. ISBN: 978-1449331160. O'Reilly Media (2013)
[55] vBuckets: The Core Enabling Mechanism for Couchbase Server Data Distribution. http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/Technical-Whitepaper-Couchbase-Server-vBuckets.pdf (Letzter Zugriff am 2.5.2015)
[56] Couchbase SDKs. http://docs.couchbase.com/couchbase-devguide-2.0/ (Letzter Zugriff am 2.5.2015)
[57] John Zablocki. Couchbase Essentials. ISBN: 978-1784394493. Packt Publishing (2015)
[58] View basics. http://docs.couchbase.com/admin/admin/Views/views-basics.html (Letzter Zugriff am 2.5.2015)
[59] N1QL DP4. http://docs.couchbase.com/developer/n1ql-dp4/n1ql-intro.html (Letzter Zugriff am 2.5.2015)
[60] Lothar Thiele. Technische Informatik 1 - Rechenleistung. http://www.tik.ee.ethz.ch/education/lectures/TI1/materials/5.pdf (Letzter Zugriff am 3.5.2015)
[61] Update Operators. http://docs.mongodb.org/manual/reference/operator/update/ (Letzter Zugriff am 3.5.2015)
[62] Philipp Ständer. Ein Vergleich eines relationalen mit einem non-relationalen Datenbanksystems (MySQL / MongoDB). http://pstaender.com/assets/download/documents/bachelorthesis/bachelorthesis.pdf (Letzter Zugriff am 4.5.2015)
[63] Sumitkumar Kanoje, Varsha Powar, Debajyoti Mukhopadhyay. Using MongoDB for Social Networking Website. http://arxiv.org/ftp/arxiv/papers/1503/1503.06548.pdf (Letzter Zugriff am 16.5.2015)
[64] Pekka Pääkkönen, Daniel Pakkala. Report on Scalability of database technologies for entertainment services. http://virtual.vtt.fi/virtual/nextmedia/Deliverables-2011/D1.2.3.3_MUMUMESE_Report%20on%20Scalability%20of%20database%20technologies%20for%20entertainment%20services.pdf (Letzter Zugriff am 17.5.2015)

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| 2PC | Two Phase Commit |
| BSON | Binary JSON |
| CRUD | Create Read Update Delete |
| DBMS | Datenbankmanagementsystem |
| DBS | Datenbanksystem |
| ER | Entity Relationship |
| FK | Foreign Key (Fremdschlüssel) |
| HTTP | Hypertext Transfer Protocol |
| ISAM | Indexed Sequential Access Method |
| JSON | JavaScript Object Notation |
| MyISAM | My Indexed Sequential Access Method |
| N1QL | Umgangssprachlich: Nickel |
| NDB | Network DataBase |
| NoSQL | Not Only SQL |
| PR | Primary Key (Primärschlüssel) |
| RDBMS | Relational DBMS |
| REST | Representational State Transfer |
| SDK | Software Development Kit |
| SQL | Structured Query Language |
| UML | Unified Modeling Language |
| XML | Extensible Markup Language |
| YAML | YAML Ain't Markup Language |

# Beispielverzeichnis

# Anhang A: Entwicklungsenvironment

Als Testumgebung dient eine virtuelle Serverlandschaft. Das Wirtsystem fungiert zeitgleich als Client, welcher auf den Servern die Last erzeugt.

## Hard- und Softwarekonfiguration

Als Wirtsystem wird ein handelsüblicher Rechner mit folgender Ausstattung verwendet:

- Hauptspeicher: Corsair Vengeance 32GB (4x8GB) DDR3 1600 MHz
- Prozessor: AMD FX-8120 8x3,10GHz
- Festplatte: 2x Western Digital WD30EZRX Green 3TB im RAID 0 Verbund
- Betriebssystem: Fedora 20
- Virtualisierungslayer: QEMU/KVM mit libvirt 1.1.3.9

Die Server werden als virtuellen Systeme realisiert und haben als Ressourcen zugeordnet:

- Hauptspeicher: 1GB
- Prozessor: 1x3,10 GHz
- Festplatte: 50 GB
- Betriebssystem: Ubuntu Server 12.04
    - Ausnahme Couchbase Query Server: CentOS 6.5

# Anhang B: MySQL Setup

## Allgemeines

Das beschriebene Setup wurde auf Servern mit dem Betriebssystem Ubuntu Server 12.04 durchgeführt. Auf diesen wurden nach der Installation ein Update der installierten Packete durchgeführt, das Packet Openssh für Remote Zugänge installiert und die Netzwerkkonfiguration von DHCP auf Static geändert.

Die folgenden Server mit IP Adressen wurden für das Setup benötigt:

- MySql InnoDb mit 192.168.122.244: Single System, welches als Storage Engine die InnoDB Engine verwendet.
- MySql MyIsam mit IP Adresse 192.168.122.15: Single System, welches als Storage Engine die MyIsam Engine verwendet.
- MySQL1 mit IP Adresse 192.168.122.221: Cluster Node, dient als Mgmt. Server
- MySQL4 mit IP Adresse 192.168.122.71: Cluster Node, dient als Proxy Server
- MySQL2 mit IP Adresse 192.168.122.139: Cluster Node, erster Shard des Clusters
- MySQL3 mit IP Adresse 192.168.122.190: Cluster Node, zweiter Shard des Clusters

Liste an verwendeter Software inkl. Versionen:

- MySQL Server Version 5.5
- MySQL Cluster Version 7.4.5
- MySQL Proxy Version 0.8

## Installation des Mgmt. Servers

Die Software wurde von http://www.mysql.com/downloads/cluster/#downloads für die Plattform Linux Generic heruntergeladen, und befand sich zum Zeitpunkt der Installation in Version 7.4.5. Folgende Schritte waren notwendig:

- Die MySQL Cluster Software wurde nach erfolgten Download in das Verzeichnis `/usr/src/mysql-mgm` des Servers kopiert. Für den Download notwendig war die Registrierung bei oracle.com.
- Entpacken der Software im Verzeichnis `/usr/src/mysql-mgm`: `tar xvfz mysql-cluster-gpl-7.2.10-linux2.6-x86_64.tar.gz`
- In das Verzeichnis wechseln: `cd mysql-cluster-gpl-7.4.5-linux2.6-x86_64`
- Kopieren der Binaries in das /usr/bin Directory: `cp bin/ndb_mgm /usr/bin` und `cp bin/ndb_mgmd /usr/bin`
- Ändern der Berechtigung: `chmod 755 /usr/bin/ndb_mg*`
- Anlegen eines Ordners für die Clusterkonfiguration: `mkdir /var/lib/mysql-cluster`

- Anlegen einer Datei für die Clusterkonfiguration: `touch /var/lib/mysql-cluster/config.ini`
- Inhalt der Clusterkonfiguration:

```
[NDBD DEFAULT]

NoOfReplicas=1
DataMemory=80M
IndexMemory=18M
[MYSQLD DEFAULT]

[NDB_MGMD DEFAULT]
DataDir=/var/lib/mysql-cluster
[TCP DEFAULT]

# Section for the cluster management node
[NDB_MGMD]
NodeId=1
# IP address of the first management node (this system)
HostName=192.168.122.221

# Section for the storage nodes
[NDBD]
# IP address of the first storage node
HostName=192.168.122.139
DataDir= /var/lib/mysql-cluster
[NDBD]
# IP address of the second storage node
HostName=192.168.122.190
DataDir=/var/lib/mysql-cluster
# one [MYSQLD] per storage node
```

- Starten des Mgmt. Servers: `ndb_mgmd -f /var/lib/mysql-cluster/config.ini --configdir=/var/lib/mysql-cluster/`
- Eintrag für Automatischen Start beim Booten: `echo "ndb_mgmd -f /var/lib/mysql-cluster/config.ini --configdir=/var/lib/mysql-cluster/" > /etc/init.d/ndb_mgmd`

# Installation der Database Shards

Die folgende Konfiguration ist auf jeden Shard durchzuführen und zeigt das Einrichten einer MySQL Datenbank mit anschließender Clustereinbindung:

- Kopieren der Cluster Software `mysql-cluster-gpl-7.2.10-linux2.6-x86_64.tar.gz` in das Verzeichnis `/usr/local`.
- Anlegen einer Gruppe: `groupadd mysql`
- Anlegen eines Users und Zuweisung zur Gruppe: `useradd -g mysql mysql`
- Wechseln in das Verzeichnis: `cd /usr/local`
- Entpacken der Software im Verzeichnis：`tar xvfz mysql-cluster-gpl-7.4.5-linux-glibc2.5-i686.tar.gz`

- Anlegen eines symbolischen Links: `ln -s mysql-cluster-gpl-7.4.5-linux-glibc2.5-i686 mysql`
- Wechseln in das Verzeichnis: `cd mysql`
- Installation der Datenbank: `scripts/mysql_install_db --user=mysql`
- Anpassen der Berechtigungen: `chown -R mysql data`
- Eintrag für Automatischen Start beim Booten: `cp support-files/mysql.server /etc/init.d/`
- Kopieren der Binaries in das /usr/bin Directory: `cp bin/* /usr/bin`
- Anlegen einer Datei für die Cluster Konfiguration: `touch /etc/my.cnf`
- Erstellen einer MySQL Cluster Konfiguration:

```
[mysqld]
ndbcluster
# IP address of the cluster management node
ndb-connectstring=192.168.122.221
[mysql_cluster]
# IP address of the cluster management node
ndb-connectstring=192.168.122.221
```

- Anlegen eines Datenverzeichnis für MySQL: `mkdir /var/lib/mysql-cluster`
- Wechseln in das Verzeichnis: `cd /var/lib/mysql-cluster`
- Initialisieren des Clusters: `ndbd –initial`
- Starten des Services: `/etc/init.d/mysql.server start`
- Absichern der Installation durch Durchlaufen des Setups unter: `./bin/mysql_secure_installation`
- Eintrag für Automatischen Start beim Booten: `echo ”ndbd” > /etc/init.d/ndbd`

## Verifizieren des Setups

Ausgehend vom Management Node kann nun der Status des Clusters überprüft werden:

- Öffnen der NDB Shell: `ndb_mgm`
- Anzeigen des Clusterstatus: `show`

## Installation des Proxy Servers

Als Software wurde das in den Ubuntu Paketquellen vorhandene Paket mysql-proxy verwendet. Die Konfiguration ist wie folgt:

- Installation des Pakets aus Paketquellen: `apt-get install mysql-proxy`
- Anlegen eines Konfigurationsverzeichnis: `mkdir /etc/mysql-proxy`
- Welcheln in das Konfigurationsverzeichnis: `cd /etc/mysql-proxy`
- Anlegen einer Konfigurationsdatei: `touch mysql-proxy.conf`
- Anpassen der File Permissions: `chmod 660 mysql-proxy.conf`

- Inhalt der Configurationsdatei

```
[mysql-proxy]
daemon=true
keepalive=true
event-threads=50
pid-file=/var/run/mysql-proxy.pid
log-file=/var/log/mysql-proxy.log
log-level=debug
plugins=proxy,admin
proxy-address=192.168.122.71:3306
proxy-skip-profiling=true
proxy-backend-addresses=192.168.122.139:3306,192.168.122.190:3306
proxy-lua-script=/usr/lib/mysql-proxy/lua/proxy/balance.lua
admin-username=root
admin-password=milandra88
admin-lua-script=/usr/lib/mysql-proxy/lua/admin.lua
```

- Anlegen einer Datei für automatischen Start beim Boot: `touch /etc/default/mysql-proxy`
- Inhalt der Datei:

```
ENABLED="true"
OPTIONS="--defaults-file=/etc/mysql-proxy.conf --plugins=proxy"
```

- Starten des Proxyservers: `/etc/init.d/mysql-proxy start`

# Anhang C: MongoDB Setup

## Allgemeines

Das beschriebene Setup wurde auf Servern mit dem Betriebssystem Ubuntu Server 12.04 durchgeführt. Auf diesen wurden nach der Installation ein Update der installierten Packete durchgeführt, das Packet Openssh für Remote Zugänge installiert und die Netzwerkkonfiguration von DHCP auf Static geändert.

Die folgenden Server mit IP Adressen wurden für das Setup benötigt:

- MongoDbS mit IP Adresse 192.168.122.195: MongoDB Single Instance Server
- MongoDb1 mit IP Adresse 192.168.122.243: Cluster Node, MongoDB Configserver
- MongoDb2 mit IP Adresse 192.168.122.159: Cluster Node, MongoDB Routing Server
- MongoDb3 mit IP Adresse 192.168.122.3: Cluster Node, erster Shard des Cluster
- MongoDb4 mit IP Adresse 192.168.122.194: Cluster Node, zweiter Shard des Cluster

Liste an verwendeter Software inkl. Versionen:

- MongoDB Version 3.0.1

Um die Software aus den Paketquellen zu installieren, wurden die offiziellen MongoDB Packages den Paketquellen hinzugefügt und anschießend von diesen deployed. Folgende Punkte sind dazu notwendig:

- Import Public Key: `sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10`
- Erstellen eines list files für MongoDB: `echo "deb http://repo.mongodb.org/apt/ubuntu "$(lsb_release -sc)"/mongodb-org/3.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.0.list`
- Reload der lokalen Packetquellendatenbank: `sudo apt-get update`
- Installation der MongoDB Packages: `sudo apt-get install -y mongodb-org`

## Setup des MongoDB Configservers

MongoDB benötigt drei Instanzen des Configservers, für den Testbetrieb können alle drei Instanzen auf den gleichen Server gestartet werden:

- Starten von drei MongoDB Configserver Instanzen:
    - Erste Instanz: mongod --configsvr --dbpath /data/configdb1 --fork --port 27021 --pidfilepath /data/configdb1/mongod-con1.lock --logpath=/var/log/mongodb/con01.log
    - Zweite Instanz: mongod --configsvr --dbpath /data/configdb2 --fork --port 27022 --pidfilepath /data/configdb2/mongod-con2.lock --logpath=/var/log/mongodb/con02.log
    - Dritte Instanz: mongod --configsvr --dbpath /data/configdb3 --fork --port 27023 --pidfilepath /data/configdb3/mongod-con3.lock --logpath=/var/log/mongodb/con03.log

- Kontrolle mittels netstat tools, ob die Instanzen auf den entsprechenden Ports auf Anfragen warten: netstat –tpl

## Setup des MongoDB Routing Server

- Stoppen des MongoDB Prozesses um Lock Konflikte mit Query Router zu vermeiden: `service mongod stop`
- Starten des MongoDB Query Router Services (MongoDBs Query Router Service nennt sich mongos): `mongos --configdb 192.168.122.243:27021,192.168.122.243:27022,192.168.122.243:27023 --port 27017 --logpath=/var/log/mongodb/mongos.log –fork`

## Setup der Database Shards

- Der beschriebene Prozess ist auf einen der beiden Shards durchzuführen.
- Verbinden zum MongoDB Routing Server: `mongo --host 192.168.122.159 --port 27017`
- Hinzufügen des ersten Shards zum Cluster:

```
mongos> sh.addShard( "192.168.122.3:27017" )
   { "shardAdded" : "shard0000", "ok" : 1 }
```

- Hinzufügen des zweiten Shards zum Cluster:

```
mongos> sh.addShard( "192.168.122.194:27017" )
   { "shardAdded" : "shard0001", "ok" : 1 }
```

- Prüfen des Status:

```
mongos> sh.status()
```

```
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("5503f6ea0d38ee27ea3b8f50")
}
  shards:
        {  "_id" : "shard0000",  "host" : "192.168.122.3:27017" }
        {  "_id" : "shard0001",  "host" : "192.168.122.194:27017" }
  balancer:
        Currently enabled:  yes
        Currently running:  no
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
                No recent migrations
  databases:
        {  "_id" : "admin",  "partitioned" : false,  "primary" :
"config" }
```

# Anhang D: Couchbase Setup

## Allgemeines

Das beschriebene Setup wurde auf Servern mit dem Betriebssystem Ubuntu Server 12.04 durchgeführt, mit Ausnahme des N1QL Query Servers für welchen aufgrund fehlenden Supports auf das Betriebssystem Linux CentOS 6.5 zurückgegriffen wurde. Auf allen Servern wurden nach der Installation ein Update der installierten Packete durchgeführt, das Packet Openssh für Remote Zugänge installiert und die Netzwerkkonfiguration von DHCP auf Static geändert.

Die folgenden Server mit IP Adressen wurden für das Setup benötigt:

- CouchbaseS mit IP Adresse 192.168.122.120: Couchbase Single Instance Server
- Couchbase1 mit IP Adresse 192.168.122.223: erster Shard des Clusters
- Couchbase2 mit IP Adresse 192.168.122.106: zweiter Shard des Clusters
- Couchbase3 mit IP Adresse 192.168.122.121: dritter Shard des Clusters
- CouchbaseN1QL mit IP Adresse 192.168.122.57: N1QL Query Server

Liste an verwendeter Software inkl. Versionen:

- Couchbase-Server Version 3.0.1
- Couchbase-Query-Engine (N1QL) Version DP4

## Setup der Shards

Beschreibung ist auf allen Shards durchzuführen.

- Die Software wurde von http://www.couchbase.com/nosql-databases/downloads als Community Edition für Ubuntu 12.04 als 64-bit Packet heruntergeladen und auf die Server verteilt.
- Installation von Dependencies: `apt-get install libssl0.9.8`
- Installation der Software: `dpkg -i couchbase-server-community_3.0.1-ubuntu12.04_amd64.deb`
- Starten des Servers: `service couchbase-server start`
- Öffnen der Couchbase Web Console unter `http://<server>:8091/`
- Beibehalten der Default-Werte für Database Path und Indices Path
- Als Hostname den Namen des Servers eintragen
- Beim Setup des ersten Shards ist der Punkt Start a new cluster auszuwählen, bei jeden weiteren Shard ist Join a cluster now auszuwählen.
    - Start a new cluster: Vergabe von Administrator als Username und test1234 als Passwort
    - Join a cluster: Angabe der IP Adresse des ersten Shards (192.168.122.121)

## Setup des Query Routers

- Die Software wurde von http://www.couchbase.com/nosql-databases/downloads als Community Edition für Red Hat auf den Server heruntergeladen.
- Entpacken der Software: `tar xvzf couchbase-query_dev_preview4_x86_64_linux.tar.gz`
- Starten der Engine: `./cbq-engine -datastore=http://192.168.122.121:8091/`

# Anhang E: Client Setup

Das Programm zum Testen der Queries wurde vollständig in Java programmiert. Der Code wird auf Github verwaltet.  Es existieren folgende Projekte:

- MySQL: https://github.com/ic12b001/BAC2_MySQL
- MongoDB: https://github.com/ic12b001/BAC2_MongoDB
- Couchbase: https://github.com/ic12b001/BAC2_Couchbase

Folgende Daten umfassen das Programm:

- Java Version 7u75
- MySQL Datenbankzugriff via Default JDBC Driver
- MongoDB Datenbankzugriff via MongoDB Java Driver 2.13
- Couchbase Datenbankzugriff via Couchbase Java SDK 2.2
    - Couchbase N1QL Zugriff über Default Java HTTP Driver

Das Programm wird auf den Client spezifiziert in Anhang A ausgeführt.