Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Implementierung einer RDF-Storage Lösung für MongoDB

Bachelorarbeit

Leipzig, August 2011

vorgelegt von
Nitzschke, Marcus
Studiengang Informatik

Betreuender Hochschullehrer:
Prof. Dr. Klaus-Peter Fähnrich
Institut für Informatik

Zusammenfassung

In dieser Arbeit wird ein System entwickelt, das es erlaubt mittels des JavaScript-Frameworks Node.js RDF-Daten in der dokumentenorientierten Datenbank MongoDB zu speichern. Dafür wird unter anderem ein abgewandeltes Datenformat für RDF vorgestellt werden, durch welches die Speicherung erst ermöglicht wird. Darüber hinaus wird dieses System mit der relationalen Datenbank Virtuoso, hinsichtlich der Performanz bei Einfüge- und Abfrageoperationen verglichen werden.

Stichworte RDF LinkedData MongoDB Node.js Virtuoso

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Diese Arbeit baut auf zwei Kerntechnologien auf, welche in den letzten Monaten und Jahren viel Interesse erfahren haben. Es handelt sich dabei um MongoDB[1] und Node.js[2]. MongoDB ist ein namhafter Vertreter der so genannten NoSQL-Datenbanken. Diese stellen einen komplett neuen Ansatz zur Speicherung von Daten im Vergleich zu den herkömmlichen relationalen Datenbanken dar. Node.js ist ein ereignisgesteuertes Framework für die Programmiersprache JavaScript. Es ermöglicht das serverseitige Erstellen von Programmen mit JavaScript. Nähere Erläuterungen zu den Technologien folgen in Kapitel 2.
Die Motivation dieser Arbeit folgt nun daraus, mit diesen beiden verhältnismäßig jungen, aber nicht weniger erfolgreichen Technologien zu arbeiten und diese zu koppeln. Daraus sollen dann im Speziellen neue Erkenntnisse bezüglich der Speicherung von Daten im Semantic Web erzielt werden. Das Semantic Web stellt dabei ein spannendes Forschungsgebiet dar, welches zunehmend an Bedeutung gewinnt und darüber hinaus allmählich auch außerhalb der Lehre und Forschung Anwendung findet.

1.2 Problemstellung

Möchte man mittels Node.js Daten für das Semantic Web abspeichern, bieten sich zunächst bewährte Datenbanken wie Virtuoso[3] an. Virtuoso ist eine relationale Datenbank die bereits von vielen Projekten zur Speicherung solcher Daten verwendet wird.

[1] `http://www.mongodb.org/` (31.07.2011)
[2] `http://nodejs.org/` (31.07.2011)
[3] `http://virtuoso.openlinksw.com/dataspace/dav/wiki/Main` (21.06.2011)

Virtuoso besitzt eine ODBC-Schnittstelle, anhand dieser externe Programme Daten sowohl Einfügen, als auch Abfragen können. Node.js bietet zum jetzigen Zeitpunkt aber keinerlei Möglichkeiten eine solche Schnittstelle anzusprechen. Darüber hinaus bietet Virtuoso auch eine HTTP-Schnittstelle. Hier ist aber die zu erzielende Performanz auf Grund des Overheads, der durch die Verwendung des HTTP-Protokolls entsteht, fraglich. Somit gibt es derzeit keine praktikable Möglichkeit Virtuoso in Kombination mit Node.js zu verwenden.
Als alternative Datenbank wurde in der Folge MongoDB ausgewählt. Die detailierten Gründe dazu folgen in Kapitel 2.3.
MongoDB speichert seine Dokumente mittels des Datenformates JSON. Dies erfordert es, dass die verwendeten Daten eine spezielle Syntax einhalten. Für RDF-Daten exisitiert mit der in [Ale08] spezifizierten RDF/JSON Serialisierung ein solches Format. Dieses Format sieht es vor *Uniform Resource Identifier* (URI) als Bezeichner der JavaScript-Objekte zu verwenden. URIs, beispielsweise `http://www.example.org`, enthalten in aller Regel Punkte, wie z.B. zur Abtrennung von Top-Level-Domains. MongoDB verbietet es aber in den Objekt-Bezeichnern Punkte zu verwenden. Es ist somit nicht möglich das gegebene RDF/JSON Format in MongoDB zu speichern.
Ein weiteres Problem ist, dass es zum jetzigen Zeitpunkt kaum Vergleiche von relationalen Datenbanken und anderen Konzepten, wie MongoDB, bezüglich der Speicherung von RDF-Daten gibt.

1.3 Ziele

Es soll durch diese Arbeit ermöglicht werden, dass mittels des JavaScript-Frameworks Node.js, RDF-Daten in MongoDB gespeichert werden können.
Darüber hinaus sollen erste wissenschaftliche Vergleichswerte bezüglich der Performanz, bei dem Speichern und Abfragen von RDF-Daten zwischen einer relationalen Datenbank und einer NoSQL-Datenbank entstehen.

2 Grundlagen

2.1 Semantic Web

Das Internet ist innerhalb des letzten Jahrzehnts zu einem festen Bestandteil unserer Gesellschaft geworden. Die Verfügbarkeit wird immer höher und die Kosten für den Endbenutzer immer geringer. Diese Entwicklung verdeutlicht die heutige Tendenz zur Informationsgesellschaft. Fast jeder Lebensbereich wird dabei bereits vom Internet ergänzt. Seien es Behörden in Form des eGovernment, Lehranstalten in Form von eLearning oder nicht zuletzt die sozialen Verbindungen von Menschen in Form von Sozialen Netzwerken. So positiv und wichtig diese Entwicklungen sind, bringt die entstehende Menge an Informationen doch auch einige Probleme mit sich.
Ein Beispiel für so ein Problem ist die Heterogenität der Informationen. Es gibt im derzeitigen World Wide Web praktisch keine Grenzen wie man ein und den selben Sachverhalt darstellen kann. Dies führt dazu, dass man Informationen nur sehr schwer integrieren kann. Ein weiteres großes Problem ist das implizite Wissen was von der Vielzahl an Informationen bereitgestellt wird. Hier ist es dem Menschen möglich logisch zu schlussfolgern und Wissen in Relationen zu setzen. Maschinell ist das jedoch nicht ohne Weiteres möglich. Ein praktisches Beispiel für daraus resultierende Einschränkungen sind die heutigen Suchmaschinen. Diese arbeiten lediglich stichwortbasiert, und verstehen weder die Semantik der gestellten Anfrage, noch der indexierten Dokumente. Hier wären inhaltliche Suchen natürlicher und praktikabler.
Die Idee des Semantic Web ist nun, die Vielzahl von Informationen so zu beschreiben, dass sie sowohl maschinen-lesbar, als auch im Besonderen maschinen-interpretierbar sind. Das Konzept wurde dabei erstmals von Tim Berners-Lee in [BLHL01] vorgestellt. Um dieses Konzept umzusetzen, bedarf es zunächst einiger Voraussetzungen und

Standards, wovon die wichtigsten nachfolgend erläutert werden. Eine Übersicht über alle relevanten Technologien und Standards ist in Abbildung 2.1 gegeben.

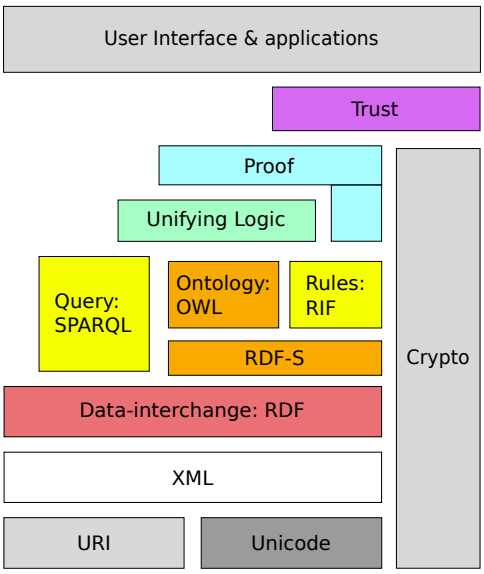

Abbildung 2.1: Semantic Web Stack, Quelle: Wikimedia Commons, eigenes Werk von Mhermans

Im Kontext des Semantic Web spricht man im Allgemeinen davon, dass Ressourcen beschrieben werden. Solche Ressourcen können beliebige Objekte wie Personen, Produkte oder Bauwerke sein. Damit diese Ressourcen eindeutig beschrieben werden können, müssen sie eindeutig identifizierbar sein. Zur Identifizierung von Ressourcen werden URIs eingesetzt. Eine URI kann zum Beispiel, wie in Kapitel 1.2 angerissen, eine URL, ähnlich `http://www.example.org`, zur Identifizierung von Web-Ressourcen sein. Ein weiteres Beispiel ist das System der ISBN-Nummern zur Identifizierung von Büchern. Richtlinien zum Entwurf von URIs werden in [BL98] gegeben.
Beschrieben werden die Ressourcen in der Folge mittels des *Resource Description Framework* (RDF). Einzelne Informationen werden dabei anhand von Tripeln dargestellt. Ein Beispiel für so ein Tripel ist in Abbildung 2.2 visualisiert. Die Bestandteile eines

Abbildung 2.2: Beispiel eines RDF-Tripels

solchen Tripels können dabei als Subjekt, Prädikat und Objekt betrachtet werden. Das Subjekt stellt den Teil der Aussage dar, über welchen etwas ausgesagt werden soll. Das Prädikat spiegelt eine Eigenschaft wider und das Objekt schließlich den Wert der Eigenschaft. Das Beispiel "`Alice-mag-Bob`" verdeutlicht dabei die Parallelen zur deutschen Grammatik. Die Ausdrucksstärke von RDF ist nun darauf zurückzuführen, dass die Objekte in Form von URIs wieder andere Subjekte repräsentieren können. So entstehen beliebig große Graphen. Die Objekte können darüber hinaus auch einfache Literalwerte wie Zeichenketten oder Zahlen sein, um atomare Eigenschaften auszudrücken. Prädikate werden dagegen immer als URI repräsentiert. Für umfangreichere Modellierungen, wie mehrwertige Beziehungen, können Subjekt und Objekt auch Anonyme Knoten (blank nodes) sein. Dadurch ist es möglich, dass nicht jede Ressource explizit über eine URI referenziert werden muss, sondern anonym bleiben kann. So muss beispielsweise bei der Aussage "*Alice hat einen Freund, der 20 Jahre alt ist.*" die Ressource des Freundes nicht explizit durch eine URI beschrieben werden.
Da die Graphdarstellung ungeeignet für die Verwendung innerhalb von Anwendungen ist, existieren verschiedene Serialisierung hierfür. Die RDF-Spezifikation definiert dabei das in [MM04] näher beschriebene RDF/XML. Darüber hinaus gibt es auch übersichtlichere Formate wie Turtle, welches in [tur08] näher beschrieben wird. Listing 2.1 stellt das Tripel aus Abbildung 2.2 in Turtle Notation dar. Hier wurde zusätzlich von einem Präfix als Abkürzung Gebrauch gemacht, da die Domäne bei allen drei URIs dieselbe ist. In dieser Arbeit wird jedoch vorangig die RDF/JSON Notation von Bedeutung sein.

```
@prefix ex: <http://example.org/>.
ex:subject ex:predicate ex:object.
```

Listing 2.1: Beispiel eines RDF-Tripels – Turtle Notation

Mit SPARQL existiert außerdem eine Abfragesprache für RDF, welche es erlaubt gezielte Informationen aus den Graphen zu extrahieren. So ist es auch möglich über gewisse Schnittstellen der Triplestores, also der Speicher der RDF-Graphen, Daten in andere Anwendungen zu integrieren. Es kann dann ein Netzwerk von Informationen über die Grenzen einzelner Graphen hinaus entstehen. Tim Berners-Lee hat dabei

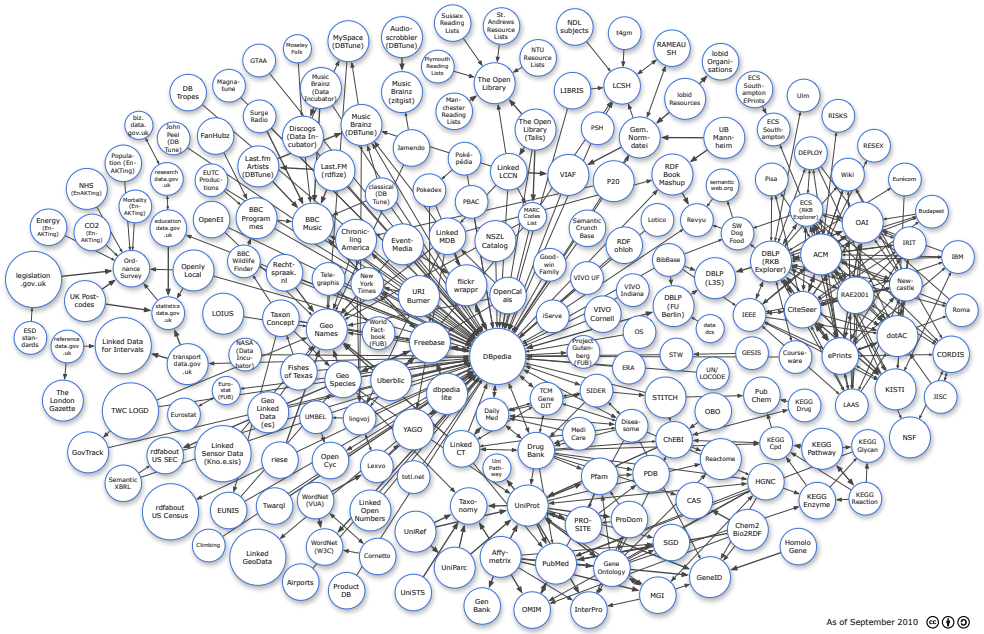

Abbildung 2.3: Linking Open Data cloud diagram, Quelle: Richard Cyganiak und Anja Jentzsch. `http://lod-cloud.net`, Stand: 09/2010

unter dem Begriff *Linked Data* Richtlinien für die Publikation weiterer Quellen für solch ein Netzwerk zur Verfügung gestellt[4]. Sofern die Daten frei zugänglich sind, spricht man auch von *Linked Open Data* (LOD). Abbildung 2.3 zeigt schemahaft die Knoten des LOD-Netzwerkes im September 2010.

Zuletzt sei eine Technik erwähnt, bei der zu einer gegebenen Ressource spezielle Darstellungen eines RDF-Graphen erzeugt werden. Dafür wird durch verschiedene Regeln ein Teilgraph des originalen Graphen erzeugt. Eine solche Technik ist die *Concise Bounded Description*[5] (CBD). Bei dieser Darstellung werden alle verfügbaren Informationen, bei denen die gewählte Ressource als Subjekt auftritt, herausgefiltert. Diese Ressourcen-zentrierte Struktur ähnelt von der Idee dem nativen RDF/JSON-Format stark. Somit liegt es durchaus nahe, RDF-Daten in RDF/JSON persistent abzuspeichern.

[4] `http://www.w3.org/DesignIssues/LinkedData.html` (10.08.2011)

[5] `http://www.w3.org/Submission/CBD/` (15.08.2011)

2.2 Node.js

Node.js ist ein Framework für JavaScript. Es stellt dem Entwickler also verschiedenste Funktionen für die Weiterverwendung zur Verfügung. Das Hauptanwendungsgebiet sind dabei Datei- und Netzwerkoperationen. Node.js an sich basiert auf der von Google entwickelten V8-JavaScript-Engine[6]. Darüber hinaus implementiert Node.js den CommonJS-Standard[7] für Module. Dadurch ist eine Vielzahl an Bibliotheken für Node.js verfügbar, was es zu einem umfangreichen und flexiblen Framework macht. Anwendungen die mit Node.js umgesetzt wurden, werden entgegen der vorwiegenden Nutzung von JavaScript serverseitig ausgeführt. Diese Tatsache stellt einen allgemeinen Trend in der Verwendung von JavaScript dar. So werden mittlerweile ganze Desktopumgebungen, wie die Gnome Shell[8], mit JavaScript umgesetzt.
Das Besondere an Node.js ist, dass es den Großteil der Operationen asynchron verarbeitet. Dafür verwendet Node.js so genannte Ereignisse. Es ist somit also ein ereignisgesteuertes Framework. Um die Vorteile dieser Eigenschaften deutlich zu machen, muss zunächst auf das Thema Nebenläufigkeit eingegangen werden.
Bei Anwendungen mit vielen Eingabe- und Ausgabeoperationen möchte man im Allgemeinen, dass voneinander unabhängige Prozesse parallel ausgeführt werden, um die Performanz zu steigern. Dafür gibt es zum einen den Multithreading-Ansatz. Bei softwareseitigem Multithreading startet eine Anwendung bei aufwendigen Operationen einen neuen Ausführungsstrang (Thread), welcher dann weitere Operationen bearbeiten kann. Bei hardwareseitigem Multithreading werden die einzelnen Threads auf die verschiedenen Kerne des Prozessors verlagert, wo sie dann parallel arbeiten können. Multithreading in diesen Formen hat jedoch auch Nachteile. So birgt es immer die Gefahr von Verklemmungen (Deadlocks). Ein reales Beispiel dafür wäre eine Straßenkreuzung ohne Ampeln, an der von allen vier Seiten Autos warten. Unter Beachtung der Regel *rechts vor links* wartet nun jedes Auto auf das rechts von ihm stehende. Ohne weitere Kommunikation ist diese Situation nicht lösbar.
Eine ereignisgesteuerte Herangehensweise funktioniert dagegen so, dass in bestimm-

[6] `http://code.google.com/p/v8/` (09.08.2011)
[7] `http://www.commonjs.org/` (09.08.2011)
[8] `http://www.gnome.org/gnome-3/` (11.08.2011)

ten Situationen Ereignisse ausgelößt werden, auf welche die Anwendungen dann reagieren können. In dem Straßenkreuzungs-Beispiel könnte nun ein Ereignis ausgelößt werden und in der Folge zum Beispiel entsprechend der Chronologie der Baujahre der Autos die Situation aufgelößt werden. In diesem Zusammenhang ist die Asynchronizität der Operationen von Bedeutung. Wären die Ein-/Ausgabeoperationen nicht asynchron, so würden die Anwendungen häufig blockiert werden. Beispielsweise möchte eine Anwendung in eine Datei schreiben obwohl auf dem entsprechenden Datenträger kein Speicherplatz mehr verfügbar ist. Wäre die Schreiboperation nicht asynchron, so würde die Anwendung nun warten bis ihr Speicherplatz zur Verfügung steht, um in die Datei zu schreiben. Im asynchronen Fall bekommt die Anwendung mitgeteilt, dass kein Speicherplatz mehr verfügbar ist und sie kann entsprechend reagieren und andere Operationen ausführen. Zu einem späteren Zeitpunkt würde die Anwendung dann noch einmal versuchen den Schreibvorgang auszuführen.
Node.js vereint nun die Eigenschaften der ereignisgesteuerten Programmierung und der asynchronen Kommunikation und bildet so eine höchst performante und skalierbare Lösung für Andwendungen mit vielen Ein-/Ausgabeoperationen.

2.3 MongoDB

Relationale Datenbanken sind bis heute die am häufigsten verwendete Datenbanktechnik. Bei diesem Modell werden die Datensätze anhand von Tabellen (Relationen) gespeichert. Ein Datensatz stellt dabei eine Zeile der Tabelle dar. Die Spalten bilden die jeweiligen Eigenschaften des Datensatzes. Dennoch führte der Wunsch nach alternativen Ansätzen schon in den 80er Jahren zu Varianten wie den Objektorientierten Datenbanken (OODB) oder den Objektrelationalen Datenbanken (ORDB). In den vergangenen Jahren hat sich ein Trend entwickelt, der auf Grund der extrem zunehmenden Benutzerzahlen im Internet den Fokus bei Datenbanken auf Faktoren wie Performanz und Skalierbarkeit verschiebt. Dafür wird häufig auf die traditionellen Eigenschaften der Atomarität, Konsistenz, Isolation und Dauerhaftigkeit (ACID-Kriterien) verzichtet.
Aus diesem Trend heraus entstanden verschiedene Implementierungen, welche unter

dem Begriff 'Not only SQL' (NoSQL) Datenbanken zusammengefasst werden. Der Begriff NoSQL wurde erstmals 1998 von Carlo Strozzi verwendet, welcher eine Datenbank entwickelte, die auf eine Zugriffsmöglichkeit über die *Structured Query Language* (SQL) verzichtete. Der ursprüngliche Begriff ist demnach von der heutigen Verwendung abzugrenzen[9]. Der Begriff in der heutigen Bedeutung wurde Anfang 2009 von Johan Oskarsson im Aufruf für eine Konferenz über verteilte, strukturierte Datenspeicher eingeführt[10].

Die Klasse der NoSQL-Datenbanken kann in weitere Gruppen, wie dokumentenorientierte oder graphenbasierte Datenbanken gegliedert werden. MongoDB wird dabei zu den dokumentenorientierten Datenbanken eingeordnet. Dabei werden die Daten in Form von Dokumenten (wie XML oder JSON) gespeichert. Diese Dokumente werden darüber hinaus, analog zu den Tabellen im relationalen Modell, in so genannten Collections gruppiert.

Eine der wichtigsten Eigenschaften von dokumentenorientierten Datenbanken im Vergleich zu relationalen Datenbanken ist der Verzicht auf feste Datenbankschemas. Das bedeutet, dass die gespeicherten Dokumente unterschiedliche Strukturen aufweisen können. Dies ist dahingehend interessant, als dass auch RDF-Daten schemalos sind. Es ist also naheliegend RDF-Daten eher in NoSQL-Datenbanken zu speichern, als in relationalen. Relationale Datenbanken haben außerdem den Nachteil, dass häufig inperformante JOIN-Berechnungen durchgeführt werden müssen.

Die in MongoDB gespeicherten Dokumente werden intern mittels BSON-Datenstrukturen[11] repräsentiert. Dies bringt entgegen der trivialen Speicherung von JSON-Dokumenten Vorteile bei der Performanz mit sich. Im Gegenzug ist der Aufwand der Umwandlung von JSON in BSON und umgekehrt relativ gering, was anhand der spezifizierten Grammatik auf der Projektseite von BSON nachvollzogen werden kann. Die Verwendung von JSON als Syntax der Dokumente resultiert auch darin, dass MongoDB JavaScript als Skriptsprache für Funktionen nutzt.

Diese aufgeführten Eigenschaften legen es nun nahe, MongoDB für die Speicherung von RDF-Daten im RDF/JSON-Format zu verwenden.

[9] siehe auch: `http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page` (10.08.2011)
[10] `http://blog.sym-link.com/2009/05/12/nosql_2009.html` (10.08.2011)
[11] `http://bsonspec.org/` (10.08.2011)

3 Anforderungen

Aus den in Kapitel 1.2 festgestellten Problemen und in Kapitel 1.3 daraus definierten Zielen, lassen sich folgende konkrete Anforderungen ableiten:

1. **Entwurf einer oder mehrerer URI-Transformationstechniken, damit URIs als Schlüsselwerte in MongoDB verwendet werden können.**
   In Kapitel 1.2 wurde bereits herausgearbeitet, dass es MongoDB nativ nicht ermöglicht URIs als Schlüsselwerte zu speichern. Dies verhindert das Abspeichern von RDF/JSON in seinem definierten Format. Aus diesem Grund soll eine oder mehrere Varianten dieses Formates erarbeitet werden, welche die Verwendung von URIs als Schlüsselwerte umgehen. Die äquivalente Semantik der Darstellungen muss dabei gewährleistet sein.
2. **Implementierung einer API, zum Speichern und Abfragen von RDF in MongoDB.**
   Es soll eine Programmierschnittstelle (API) für Node.js bereit gestellt werden, welche es ermöglicht RDF-Tripel in MongoDB zu speichern und abzufragen. Dabei sollen für den Anwender mögliche Transformationen des Datenformats verborgen bleiben, so dass dieser nur mit dem ursprünglichen RDF/JSON arbeitet. Des Weiteren soll sich das Design der entworfenen API an der von MongoDB zur Verfügung gestellten orientieren. Dies betrifft zum Beispiel Bezeichner von Funktionen oder spezielle Rückgabewerte.
3. **Vergleich der Implementierung mit einem herkömmlichen, auf einer relationalen Datenbank aufbauenden RDF-Store**.
   Die Ergebnisse der Implementierung sollen abschließend in einem Vergleich mit einem RDF-Store auf relationaler Basis genauer untersucht werden. Hier bietet sich Virtuoso Open-Source an. Bei dem Vergleich steht die Gegenüberstellung

von benötigter Zeit und benötigtem Speicher bei Einfüge- und Abfrageoperationen im Vordergrund.

4 Gegenwärtiger Stand der Technik

Auf Grund von bislang nicht vorhandenen wissenschaftlichen Publikationen zur Verwendung von NoSQL-Datenbanken in Kombination mit RDF-Daten, exisitieren auch keinerlei fundierte Erfahrungen oder Vergleichswerte zu relationalen Datenbanken.
Was im Speziellen die Verwendung von MongoDB zur Speicherung von RDF-Daten betrifft, gibt es ein paar wenige veröffentlichte, nicht wissenschaftliche, Ansätze. Diese verwenden aber jeweils eine andere Plattform der Implementierung, wie beispielsweise Perl[12], Python[13] oder C#[14]. Besonders der Artikel über die C#-Implementierung bietet gute Vergleichsansätze, was zum einen die URI-Transformationstechniken, aber auch Performanzwerte betrifft.
Der Autor stellt in seinem Artikel zwei mögliche Formate vor, welche es erlauben mit RDF/JSON serialisierte RDF-Tripel in MongoDB zu speichern. Im Folgenden sollen diese Ansätze kurz vorgestellt werden und eventuelle Einschränkungen oder Nachteile dargestellt werden. Sei folgendes Tripel in RDF/JSON Notation gegeben:

```
{
  "http://example.org/subject" : {
    "http://example.org/predicate" : [
      { "type" : "uri" , "value" : "http://example.org/object" }
    ]
  }
}
```

Listing 4.1: Beispiel-Tripel in RDF/JSON Notation

[12] `https://github.com/ant0ine/MongoDB-RDF` (28.06.2011)
[13] `http://wwaites.posterous.com/mongo-as-an-rdf-store` (28.06.2011)
[14] `http://www.dotnetrdf.org/blogitem.asp?blogID=35` (28.06.2011)

Es folgen die Transformations-Ansätze:

```
{
  "name" : "some-name",
  "graph" : [
            {
              "subject" : "http://example.org/subject" ,
              "predicate" : "http://example.org/predicate" ,
              "object" : "http://example.org/object"
            }
          ]
}
```

Listing 4.2: Struktur des Graph-basierten Ansatzes

Dieser 'Graph-basierte' Ansatz erstellt für jeden Graphen genau ein Dokument in der Datenbank. Der Autor erklärt hierbei selber, dass für Anfragen nach häufig vorkommenden Prädikaten, wie `rdf:type`, vermutlich alle Graphen getroffen werden und somit jedes Dokument im Ergebnis enthalten ist. Somit müsste dieses dann nachfolgend weiter gefiltert werden. Dies ist vermutlich äußerst inperformant. Ein weiterer Nachteil dieser Transformationstechnik ist, dass bei der Verwendung eines einzelnen Graphen lediglich ein einziges Dokument in der Datenbank gespeichert ist. Dieser Umstand mündet wiederum in ineffizienten Abfragen, in denen jeweils nur das eine vorhandene Dokument getroffen werden kann.
Der zweite Ansatz setzt den Fokus nicht auf die Graphen sondern auf die vorhandenen Tripel.

```
{
  "name" : "some-name" ,
  "uri" : "http://example.org/graph"
}
```

Listing 4.3: Struktur des Tripel-zentrierten Ansatzes – Graph-URIs

```
{
  "subject" : "http://example.org/subject" ,
  "predicate" : "http://example.org/predicate" ,
  "object" : "http://example.org/object" ,
  "graphuri" : "http://example.org/graph"
}
```

Listing 4.4: Struktur des Tripel-zentrierten Ansatzes – Quadrupel

Hierbei werden die Graphen unabhängig von den Tripeln gespeichert und lediglich über die Eigenschaft `graphuri` referenziert. Somit erfolgt die Darstellung als Quadrupel. Dieser Ansatz repräsentiert damit die Darstellung von RDF-Tripeln in Virtuoso[15]. Hier werden die RDF-Tripel ebenfalls als Quadrupel aus Graph-URI, Subjekt, Prädikat und Objekt gespeichert.
Bei diesem Ansatz gibt es keine größeren Nachteile. Dieses Format wird auch bei den Performanztests des Autors benutzt. Es bleibt aber anzumerken, dass diese Transformation vermutlich nicht die Vorteile der flexiblen Datenstrukturen von dokumentenorientierten Datenbanken ausnutzt und eher in die Kategorie der Schlüssel-Wert-Speicher fällt.
Die Ergebnisse der Performanztests können auf Grund von unterschiedlicher Hardware und unterschiedlichen Implementierungen nur schwer im Detail verglichen werden. Der Gesamteindruck und das Fazit des Autors bestätigen allerdings die später vorgestellten Ergebnisse dieser Arbeit.

[15] siehe auch: `http://docs.openlinksw.com/virtuoso/rdfdatarepresentation.html` (15.08.2011)

5 Spezifikation

URI-Transformationstechniken

Unabhängig von den Ansätzen, die in Kapitel 4 vorgestellt wurden, entstanden in dieser Arbeit zwei weitere Transformationen. Diese sollen nachfolgend vorgestellt werden:

```
{
  "subject": "http://example.org/subject",
  "predicates": [
    {
      "uri": "http://example.org/predicate",
      "objects": [
        {
          "type": "uri",
          "value": "http://example.org/object"
        }
      ]
    }
  ]
}
```

Listing 5.1: Struktur der Transformationstechnik 'nested'

Der erste Ansatz wird auf Grund seiner verschachtelten Form im Weiteren auch mit der Bezeichnung "nested" referenziert werden.
Diese Transformation erstellt pro gegebenem Subjekt genau ein Dokument, welches alle vorhandenen Tripel des Subjekts umfasst. Dafür werden sowohl die Prädikate,

als auch die darin enthaltenen Objekte jeweils als Arrays implementiert. Diese Art der Datenhaltung stellt einen praktikablen Mittelweg in der Anzahl der gespeicherten Dokumente dar. Extreme wie die Speicherung eines Dokumentes pro Graph, oder ein Dokument pro Tripel werden hier vermieden. Dieser Ansatz stellt somit einen Ressourcen-zentrierten Ansatz dar, der besonders auf Anfragen mit gegebenem Subjekt ausgelegt ist. Diese Art des Datenzugriffes ist vorallem im Umfeld von Linked Data von großem Interesse.
Trotz der verhältnismäßig tiefen Schachtelung der Dokumente, lassen sich in MongoDB auf dieser Struktur sehr einfach Abfragen stellen. Dies wird durch die feste Deklaration der Schlüsselbezeichner erreicht, wodurch mit der *Dot Notation*[16] auch die am tiefsten geschachtelten Dokumente mit nativen Abfragefunktionen erreicht werden können.
Des Weiteren können auch ursprünglich nicht vorgesehene Attribute wie die URI des zugehörigen Graphen problemlos ergänzt werden.

Dem zweiten Ansatz liegt die Idee zugrunde, dass das ursprüngliche Format der RDF/JSON-serialisierten Tripel möglichst wenig verändert werden soll. Somit liegt der Fokus auf dem reinen Ersetzen der in den Schlüsseln gespeicherten URIs.

```
{ "id": "foo1", "uri": "http://example.org/subject" },
{ "id": "foo2", "uri": "http://example.org/predicate" }
```

Listing 5.2: Struktur der Transformationstechnik 'dict' – Wörterbuch

```
{
  "foo1": {
    "foo2": [ { "type" : "uri" , "value" : "http://example.org/object" } ]
  }
}
```

Listing 5.3: Struktur der Transformationstechnik 'dict' – Tripel

Die ursprünglichen URIs werden hier in eigene Dokumente ausgelagert und jeweils um einen eindeutigen Bezeichner ergänzt. In den Ursprungsdokumenten werden dann

[16] `http://www.mongodb.org/display/DOCS/Dot+Notation+(Reaching+into+Objects)` (07.07.2011)

die URIs der Subjekte und Prädikate jeweils durch den eindeutigen Bezeichner ersetzt. Es wird somit eine Art Wörterbuch der URIs vorgehalten, weswegen dieser Ansatz auch mit der Bezeichnung "dict" abgekürzt wird.
Die eindeutigen Bezeichner im Wörterbuch werden durch native *Object IDs*[17] realisiert. Geht man nun davon aus, dass eine solche Object ID im Allgemeinen weniger Speicherplatz benötigt als eine durch einen String repräsentierte URI, so stellt das einmalige Speichern des Strings und mehrmalige Verwenden der Object ID eine Verbesserung des Speicherbedarfs dar. Dies wirkt sich jedoch erst dann positiv aus, wenn die entsprechende URI mehrmals in dem jeweiligen Datensatz verwendet wird. Das ist aber insbesondere bei typischen Prädikat-URIs, wie `rdfs:label` häufig gegeben.
In den Objekten vorkommende URIs werden bei diesem Ansatz jedoch nicht ersetzt. Der Grund hierfür ist, dass bei diesem Attribut die Problematik, der in Schlüsselwerten gespeicherten URIs, nicht gegeben ist und somit die Operationen für das Ersetzen und spätere Auflösen der jeweiligen URIs durch ihre Referenzen gespart werden kann.
Ein Nachteil dieser Transformationstechnik ist, dass nur schwer zusätzliche Attribute, wie z.B. die Graph-URI, ergänzt werden können. Es wäre zwar möglich in die Objekte zusätzliche Attribute zu ergänzen, jedoch würde das den eigentlichen Ansatz verletzen, dass die RDF/JSON-Struktur nicht verändert werden soll.

Merging-Algorithmus bei Einfügeoperationen

Damit bei Anfragen an die Datenbank jeweils nur ein Dokument pro Subjekt zurückgeliefert wird, ist es von Nöten entweder beim Einfügen von neuen Dokumenten oder aber beim Verarbeiten von Anfrageergebnissen die Dokumente, deren Subjekte nicht einmalig sind, zu mergen. Um die Abfrageoperationen möglichst performant zu implementieren, wird bei der entwickelten API bereits beim Einfügen eines Dokuments auf dessen Eindeutigkeit geprüft. Für den Fall, dass das Subjekt des einzufügenden Dokumentes bereits existiert, wird ein Merging-Algorithmus ausgeführt, der noch nicht vorhandene Prädikate oder Objekte in dem bereits gespeicherten Doku-

[17] `http://www.mongodb.org/display/DOCS/Object+IDs` (07.07.2011)

ment ergänzt. Nachfolgend ist der Algorithmus als UML Aktivitätsdiagramm nach [Bal05] visualisiert.

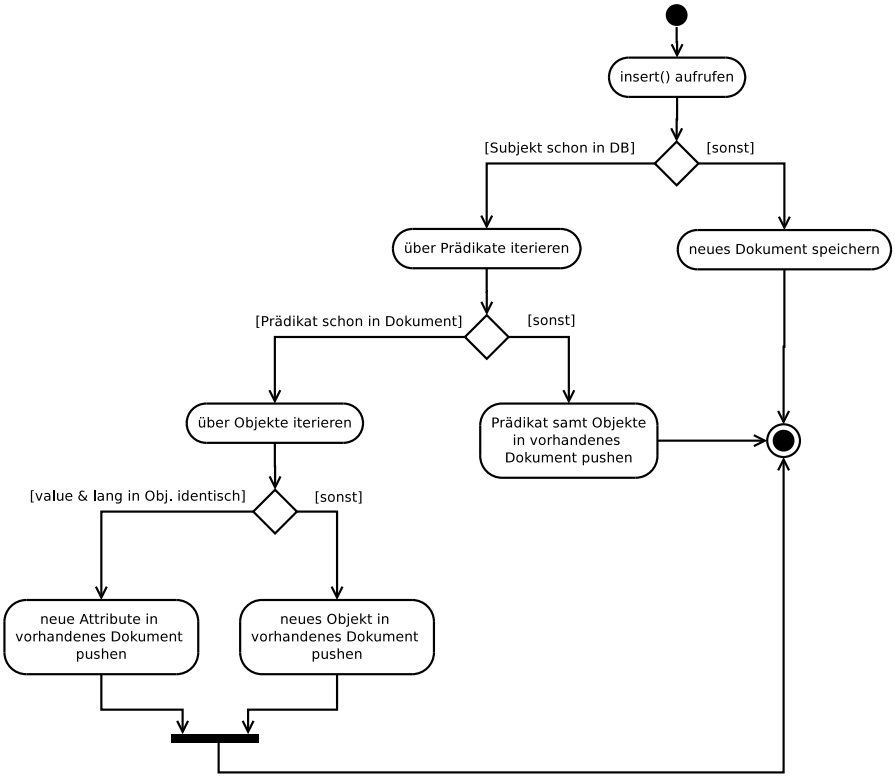

Abbildung 5.1: Aktivitätsdiagramm des Merging-Algorithmus

Bei der chronologisch letzten Entscheidung wird beim Iterieren über die Objekte geprüft ob die jeweiligen Werte der Attribute `value` und `lang` übereinstimmen. Dies soll bei möglichen multilingualen Ambiguitäten, wie "Sauerkraut" oder Eigennamen, wie "Berlin" verhindern, dass noch nicht vorhandene Objekte nicht ergänzt werden.
Seien zwei Dokumente A und B gegeben, wobei A bereits in der Datenbank gespeichert ist und B eingefügt werden soll. Dann kann die maximale Anzahl an Vergleichsoperationen n wiefolgt angegeben werden: p_A bzw. p_B sei die Anzahl der Prädikate in A bzw. B. o_A bzw. o_B sei die Anzahl der Objekte in den beiden gleichnamigen Prädikaten. Da zunächst jeweils jedes einzufügende Prädikat mit jedem vorhandenen Prädikat verglichen werden muss ergeben sich hier $p_A \times p_B$ Vergleiche. Nachdem zwei Prädikate gefunden wurden, welche die selbe URI besitzen, muss nun jedes neue Objekt mit jedem vorhandenen Objekt verglichen werden und es ergeben sich $o_A \times o_B$

Vergleiche. Für den Fall, dass in zwei Objekten `value` und `lang` identisch sind, muss nun noch das optionale Attribut `datatype` auf Vorhandensein geprüft werden und ggf. ergänzt werden. Es ergeben sich somit im Worstcase insgesamt

$$n = (p_A \times p_B) + (o_A \times o_B) + 1 \tag{5.1}$$

Vergleiche.

6 Implementierung

Die entwickelte API besteht im Kern aus sechs Methoden. Diese sind nachfolgend aufgeführt und die entsprechende Funktionalität kurz erläutert. Die Angaben in eckigen Klammern stellen dabei jeweils die Argumente für die Methoden dar.

- `insert(<tripel>)`
  Die Methode `insert()` dient dem Einfügen von RDF-Tripeln. Diese werden dabei in RDF/JSON-Notation entgegengenommen, danach entsprechend Kapitel 5 transformiert und anschliessend in die Datenbank eingefügt.
- `findBySubject(<uri>)`
  Diese Methode erhält als Argument eine bestimmte URI und liefert in der Folge alle Tripel zurück, welche als Subjekt die gewünschte URI besitzen. Die Tripel werden dabei unabhängig von der internen Transformation in RDF/JSON-Syntax zurückgegeben. Dies ist bei allen Abfrageoperationen der Fall.
- `findBySubjectDeep(<uri>)`
  Analog zu `findBySubject()` sucht diese Methode nach Tripeln mit einer gewünschten URI als Subjekt. Zusätzlich werden hier aber Referenzen in den Objekten aufgelößt. Das heißt, besitzt ein gefundenes Tripel als Objekt eine URI, wird geprüft ob diese URI wiederum ein Subjekt in der Datenbasis darstellt. Ist dies der Fall, so wird auch dieses Tripel als Ergebnis geliefert.
- `findBySubjectPredicate(<suri>, <puri>)`
  Auch diese Methode sucht zunächst nach Tripeln mit einem bestimmten Subjekt. Darüber hinaus werden hier die Tripel aber auch durch eine gegebene URI für das Prädikat eingeschränkt.
- `findByPredicateValue(<puri>, <val>)`
  Die Methode `findByPredicateValue()` liefert alle Tripel zurück, die auf eine

gegebene URI für das Prädikat und dem entsprechenden Objekt (value) zutreffen. Das Objekt kann dabei sowohl eine URI, als auch ein Literal sein.

- `findByValue(<val>)`
  Entgegen der Methode `findByPredicateValue()` sucht diese Methode nur nach dem Objekt und schränkt nicht zusätzlich über das Prädikat ein.

Bei den Abfrageoperationen wurde sich dabei gegen die Verwendung der Abfragesyntax der nativen `query()`-Methode[18] von MongoDB entschieden. Hier müsste der Entwickler, welcher die API einsetzt, nämlich Wissen über die transformierte Darstellung der Daten innerhalb der Datenbank besitzen. Bei den implementierten Operationen ist dies nicht notwendig. Es wurden auch Überlegungen angestellt die Abfragesyntax von MongoDB zu verwenden und diese Abfragen dann jeweils auf die nötige Syntax der verwendeten Transformation zu übersetzen. Hier steht der benötigte Aufwand, der im Schreiben eines Parsers bestünde, aber in keinem Verhältnis zu dem erzielten Nutzen.

Um allgemein Daten mit Node.js in MongoDB zu speichern, benötigt es einen Datenbanktreiber. Die beiden implementierten Ansätze wurden dabei mit zwei verschiedenen Treibern umgesetzt. Der Ansatz dict nutzt hier `node-mongo-native`[19]. Dieser Treiber ist sehr Datenbank-nah. Das bedeutet, die bereitgestellten Funktionen orientieren sich alle an den nativ von MongoDB verwendeten Funktionen. Der zweite Ansatz, nested, greift auf das Mongoose-Projekt[20] zurück. Mongoose stellt dabei eher einen ORM-Ansatz[21], als einen reinen Datenbanktreiber, dar. Da dieses Projekt aber letztlich auch `node-mongo-native` für die Kommunikation mit der Datenbank benutzt, sind hier gleiche Bedingungen zwischen den Implementierungen gegeben.
Beide Ansätze haben Vor- und Nachteile, weshalb es sich schwierig gestaltet einen der beiden zu favorisieren. Mongoose kann mit interessanten Merkmalen wie *Model Validation*[22] aufwarten. Dies ist zum Beispiel bei dem Typ eines Objektes hilfreich, wo nur die Eigenschaften `uri, literal` und `bnode` zulässig sind. Dagegen ist das ORM-

[18] `http://www.mongodb.org/display/DOCS/Querying` (09.08.2011)
[19] `https://github.com/christkv/node-mongodb-native` (09.08.2011)
[20] `http://mongoosejs.com/` (09.08.2011)
[21] ORM: Object-Relational Mapping
[22] `http://mongoosejs.com/docs/validation.html` (09.08.2011)

Konzept manchmal hinderlich, wenn man wirklich Datenbank-nah arbeiten möchte. Dieses Datenbank-nahe Arbeiten ermöglicht dagegen `node-mongo-native`.

Zuletzt sei erwähnt, dass zur Verwaltung des Quellcodes der Implementierung das Versionskontrollsystem Git[23] verwendet wurde. Der Quellcode ist über das Portal GitHub unter dem Projektnamen `node_mongo_rdf`[24] verfügbar.

[23] `http://git-scm.com/` (09.08.2011)

[24] `https://github.com/kenda/node_mongo_rdf` (09.08.2011)

7 Evaluation

7.1 Setup

Abkürzungen der Vergleichsobjekte

In den nachfolgenden Auswertungen werden für die untersuchten Vergleichsobjekte Abkürzungen verwendet. Diese werden hiermit eingeführt und kurz erläutert:

- nested: Gegenstand der Arbeit, siehe Kapitel 5
- dict: Gegenstand der Arbeit, siehe Kapitel 5
- virtNative: Standardinstallation Virtuoso Open-Source 6.1.2 über native ODBC-Schnittstelle via ISQL[25]
- virtHttp: Standardinstallation Virtuoso Open-Source 6.1.2 über lokalen HTTP-Endpunkt. Einfügeoperation via Curl über HTTP PUT[26], Abfrageoperationen via `node-sparql-endpoint`[27]

Datensätze

Als Datenbasis diente bei den Auswertungen die Datenbank *caucasus spiders*[28], welche in mehrere unterschiedlich große Datensätze unterteilt wurde. Das Verhältnis zwischen der Anzahl der Subjekte, der Anzahl an URIs und der Gesamtzahl an Tripeln ist dabei in Tabelle 7.1 dargestellt.

[25] `http://docs.openlinksw.com/virtuoso/isql.html` (30.07.2011)
[26] `http://docs.openlinksw.com/virtuoso/rdfinsertmethods.html` (03.08.2011)
[27] `https://github.com/0xfeedface/node-sparql-endpoint` (25.07.2011)
[28] `http://caucasus-spiders.info/` (26.07.2011)

Kürzel	Subjekte	URIs	Tripel
2k	2056	2064	10698
5k	5464	5476	29446
10k	9830	9842	59431
20k	19583	19595	110431
40k	41901	41918	238965

Tabelle 7.1: verwendete Datensätze

Messverfahren

Bei den durchgeführten Messungen der verschiedenen Operationen, wurden jeweils zehn Durchgänge betrachtet. Die aufgeführten Werte ergeben sich durch den Median dieser zehn Durchgänge. Der Median wurde hier gewählt, um eventuelle Ausreißer durch anderweitige Prozesse auf dem Testsystem zu kompensieren. Das Testsystem ist dabei gegeben durch einen AMD Phenom II X2 555, 2.0 GiB RAM und einen Linux Kernel Version 2.6.38.8.
Die Messungen der Abfrageoperationen wurden jeweils "cold" duchgeführt, d.h. der Datenbank-Server wurde vor jedem Testlauf neugestartet um den Zwischenspeicher zu leeren. Im Falle von MongoDB wurden zunächst auch einige Messläufe "warm" durchgeführt. Hier konnten aber keine Unterschiede zu den Ergebnissen aus den cold Läufen festgestellt werden, weswegen im Anschluss nur cold getestet wurde. Bei Virtuoso gibt es dagegen Unterschiede zwischen warm und cold, jedoch würden die Ergebnisse der warm-Läufe die Schere bei den Auswertungen noch weiter öffnen, was für den eigentlichen Vergleich zwischen MongoDB und Virtuoso keinen Mehrwert hat.
Die in Abschnitt 7.2.3 vorgestellten logischen Datenbankgrößen wurden im Fall von MongoDB mittels des `collstats`-Kommandos[29] ermittelt. Im Fall von Virtuoso wurde auf die Statistik-Auswertung der Administrationsoberfläche Conductor[30] zurückgegriffen.

[29] `http://www.mongodb.org/display/DOCS/collStats+Command` (04.08.2011)
[30] `http://docs.openlinksw.com/virtuoso/adminui.html` (04.08.2011

Speichergrößen

Für die Vergleichsobjekte, welche mit Node.js implementiert wurden – also nested, dict und virtHttp – wurde zusätzlich der jeweils genutzte Arbeitsspeicher (RAM) des Prozesses mit gemessen. Hierbei werden zwei Größen unterschieden:

- RSS (Resident Set Size): beschreibt den benötigten Arbeitsspeicher
- VSize (Virtual Memory Size): beschreibt den benötigten Arbeitsspeicher inklusive umgelagerten Speicher auf der Festplatte (Swap)

SPARQL-Queries

Die von der entwickelten API implementierten Abfrageoperationen stellen häufig verwendete Queries dar. Da MongoDB z.B. bei der Suche nach einem bestimmten Subjekt und Prädikat nicht nur die getroffenen Tripel zurückgibt, sondern komplett die jeweiligen Dokumente, wurden die SPARQL-Queries so erweitert, dass sie ebenfalls jeweils alle vorhandenen Tripel eines getroffenen Subjektes zurückliefern. Nachfolgend die verwendeten Queries bei den Messungen von Virtuoso:

- `findBySubject(<uri>)`:

```
SELECT ?p ?o
WHERE {<uri> ?p ?o}
```

- `findBySubjectDeep(<uri>)`:

```
SELECT DISTINCT ?p ?o ?p2 ?o2
WHERE { <uri> ?p ?o. OPTIONAL {?o ?p2 ?o2.} }
```

- `findBySubjectPredicate(<suri>, <puri>)`:

```
SELECT ?p ?o
WHERE {<suri> <puri> ?o2. <suri> ?p ?o.}
```

- `findByPredicateValue(<puri>, <val>)`:

```
SELECT ?s ?p ?o
WHERE {?s <puri> <val>. ?s ?p ?o.}
```

- `findByValue(<val>)`:

```
SELECT ?s ?p ?o
WHERE {?s ?p2 <val>. ?s ?p ?o.}
```

7.2 Ergebnisse

7.2.1 Einfügen

Abbildung 7.1 stellt anhand eines Liniendiagramms den Zusammenhang zwischen der Größe des Datensatzes und der benötigten Zeit bei der Einfügeoperation `insert()` dar. Die x-Achse repräsentiert dabei die Größe des Datensatzes und ist durch die Anzahl der jeweiligen Subjekte gegeben. Die y-Achse repräsentiert die benötigte Zeit in Sekunden.
Anhand der Graphen wird deutlich, wie stark sich die Performanz bei dem Einfügen von Tripeln zwischen den implementierten Ansätzen und virtNative bzw. virtHttp unterscheidet. Während virtNative mit größer werdenden Datensätzen im Schnitt nur linear mehr Zeit benötigt, wachsen nested und dict exponentiell an. Bei dem kleinsten Datensatz entsteht so bei dem Vergleich virtNative – nested eine Differenz von 1.097 s, bei dem Vergleich virtNative – dict eine Differenz von 9.519 s. Bei dem größten Datensatz liegt bei dem Vergleich virtNative – nested eine Differenz von 11.97 Minuten und bei virtNative – dict eine Differenz von 16.39 Minuten vor. Die Ergebnisse von virtHttp sind im Schnitt um Faktor drei schlechter als die von virtNative. Bei dem benötigten Speicher gibt es zwischen nested und dict nur geringe Unterschiede. Beide benötigen bei dem größten Datensatz rund 310 MiB RAM inklusive Swap.

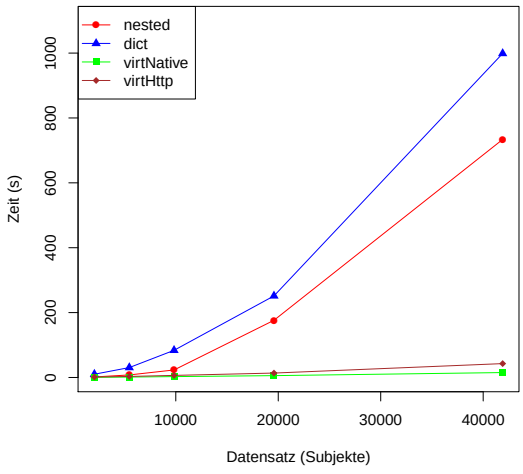

Abbildung 7.1: insert() – Zeit

	nested	dict	virtNative	virtHttp
2k	1.609	10.031	0.512	1.415
5k	7.811	30.522	1.300	3.163
10k	23.148	83.707	2.697	6.368
20k	176.297	251.442	5.674	13.605
40k	733.773	998.763	15.153	42.731

Tabelle 7.2: insert() – Zeit (in s)

	nested	dict
2k	58.28	36.89
5k	56.33	54.56
10k	80.43	73.04
20k	137.23	130.50
40k	261.65	279.80

Tabelle 7.3: insert() – RSS (in MiB)

	nested	dict
2k	90.79	69.07
5k	88.28	86.99
10k	112.71	105.83
20k	175.54	162.84
40k	306.18	312.73

Tabelle 7.4: insert() – VSize (in MiB)

Der zeitliche Unterschied zwischen nested und dict ist darauf zurückzuführen, dass dict mehr datenbankseitige Einfügeoperationen benötigt. Beispielweise benötigt nested bei dem Einfügen eines einzigen Tripels genau eine Einfügeoperation, wobei dict hier drei benötigt. Zwei werden benötigt um die URI des Subjektes und des Prädikats in die URI-Collection einzufügen und danach kann das Tripel an sich in die Tripel-Collection eingefügt werden.
Die enormen zeitlichen Unterschiede zu Virtuoso lassen schon frühzeitig die implementierten Ansätze unbrauchbar erscheinen. Interessanterweise kommt der in Kapitel 4 vorgestellte Autor zu einem ähnlichen Ergebnis bei seinen Tests. Somit kann also der Schluss gezogen werden, dass nicht die Implementierung oder die Plattform der Implementierung für die schlechten Ergebnisse der Einfügeoperation verantwortlich ist, sondern MongoDB an sich.

7.2.2 Abfragen

Die Ergebnisse der Abfrageoperationen sind anhand von Säulendiagrammen visualisiert. Die Säulen sind dabei nach dem jeweiligen Datensatz gruppiert. So entstehen fünf Gruppen mit jeweils vier Balken, für die einzelnen Vergleichsobjekte. Die y-Achse repräsentiert die benötigte Zeit in Sekunden. Die Skala der y-Achse ist total, wurde jedoch zur besseren Wahrnehmung logarithmiert, da sonst teilweise Balken auf Grund der großen Unterschiede in den Ergebnissen nicht mehr erkennbar wären.

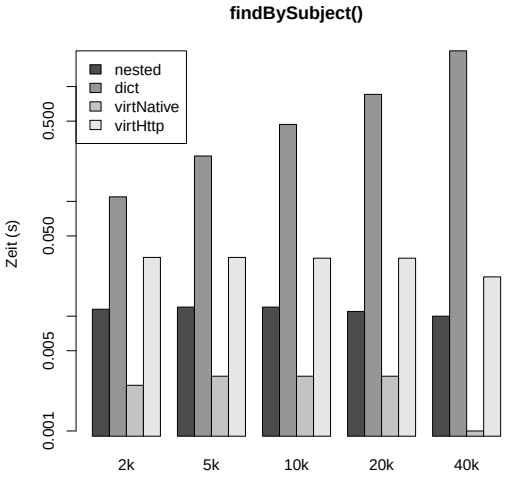

Abbildung 7.2: findBySubject() – Zeit

	nested	dict	virtNative	virtHttp
2k	0.0115	0.1095	0.0025	0.0325
5k	0.0120	0.2485	0.0030	0.0325
10k	0.0120	0.4680	0.0030	0.0320
20k	0.0110	0.8550	0.0030	0.0320
40k	0.0100	2.0470	0.0010	0.0220

Tabelle 7.5: findBySubject() – Zeit (in s)

Die grafische Repräsentation der benötigten Zeit von `findBySubject()` in Abbildung 7.2 zeigt, dass bei allen Vergleichsobjekten außer dict, im Schnitt die benötigte Zeit bei jedem Datensatz konstant bleibt. Im Gegensatz dazu wächst die benötigte Zeit

	nested	dict	virtHttp
2k	14.40	26.36	7.51
5k	14.04	62.15	7.51
10k	14.34	65.34	7.51
20k	14.24	165.50	7.51
40k	14.36	320.45	7.52

Tabelle 7.6: findBySubject() RSS (in MiB)

	nested	dict	virtHttp
2k	59.30	66.18	54.70
5k	59.18	94.25	54.70
10k	59.28	98.11	54.70
20k	59.28	198.36	54.70
40k	59.18	355.89	54.70

Tabelle 7.7: findBySubject() VSize (in MiB)

bei dict linear an. Somit besteht bei dem größten Datensatz zwischen schnellster und langsamster Implementierung ein Unterschied von 2.046 s. Zwischen virtNative und nested besteht im Schnitt ein Unterschied von 0.009 s zugunsten von Virtuoso. Im Vergleich zu virtHttp ist nested durchschnittlich 0.02 s schneller. Auffällig bei den Ergebnissen ist, dass beide Virtuoso-Implementierungen ihre Ergebnisse bei dem größten Datensatz verbessert haben. Bei dem genutzten Arbeitsspeicher bietet sich ein ähnliches Bild: Während die Werte bei nested und virtHttp bei größer werdenden Datensätzen konstant bleiben, wächst der Speicherverbrauch von dict an.
Der Grund hierfür ist, dass bei dieser Implementierung die komplette URI-Collection in einer vereinfachten Form im Speicher gehalten wird. Dies ist nötig, um bei der Ersetzung der referenzierenden IDs in den Tripeln Datenbank-Queries zu sparen. Andernfalls müsste für jede referenzierende ID in der Tripel-Collection eine Abfrage an die URI-Collection gestellt werden. Dies würde zu einer noch größeren Inperformanz führen. Die schlechten Ergebnisse von dict in der Zeitmessung sind darauf zurückzuführen, dass in der Tripel-Collection keine Datenbank-Indizes[31] angelegt werden können. Der Grund ist, dass solche Indizes nur auf festen Schlüsselbezeichnern angelegt werden können. In der Tripel-Collection gibt es diese festen Bezeichner aber nicht, da hier die referenzierenden IDs als Bezeichner verwendet werden.

Die Resultate der Operation `findBySubjectDeep()` fallen durchgängig schlechter aus als bei `findBySubject()`. Jedoch ist aus Abbildung 7.3 erkennbar, dass die Reihenfolge der Vergleichsobjekte identisch ist. So erzielt virtNative die besten Ergebnisse, dict dagegen die schlechtesten. Unerklärlich ist der Ausreißer von dict bei dem

[31] `http://www.mongodb.org/display/DOCS/Indexes` (28.07.2011)

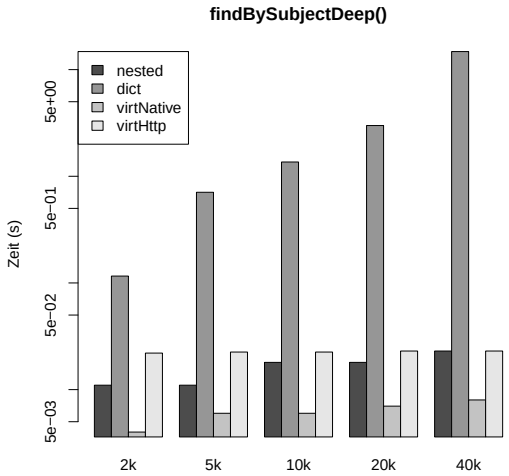

Abbildung 7.3: findBySubjectDeep() – Zeit

	nested	dict	virtNative	virtHttp
2k	0.0110	0.1160	0.0040	0.0220
5k	0.0110	0.7080	0.0060	0.0225
10k	0.0180	1.3610	0.0060	0.0225
20k	0.0180	2.9905	0.0070	0.0230
40k	0.0230	14.7635	0.0080	0.0230

Tabelle 7.8: findBySubjectDeep() – Zeit (in s)

	nested	dict	virtHttp
2k	14.06	21.17	7.52
5k	13.99	42.06	7.52
10k	14.88	54.15	7.53
20k	14.93	79.67	7.52
40k	15.20	227.43	7.53

Tabelle 7.9: findBySubjectDeep() RSS (in MiB)

	nested	dict	virtHttp
2k	59.19	60.89	54.70
5k	59.19	74.02	54.70
10k	59.24	86.37	54.70
20k	59.24	112.53	54.70
40k	59.24	265.78	54.70

Tabelle 7.10: findBySubjectDeep() VSize (in MiB)

größten Datensatz. Hier ist keinerlei Verhältnismäßigkeit zu den anderen Datensätzen erkennbar. Der Speicherverbrauch ist sowohl bei nested, als auch bei virtHttp konstant geblieben und identisch zu den bereits vorgestellten Ergebnissen der Operation

`findBySubject()`. Der benötigte Speicher von dict ist dagegen durchgängig geringer. Die zeitlichen Unterschiede zu `findBySubject()` sind darauf zurückzuführen, dass jedes Dokument der Ergebnismenge auf mögliche Objekte vom Typ `uri` durchsucht wird. Für den Fall, dass solche Objekte gefunden wurden, fallen zusätzliche Abfragen an, welche nach der jeweiligen referenzierten Ressource suchen.

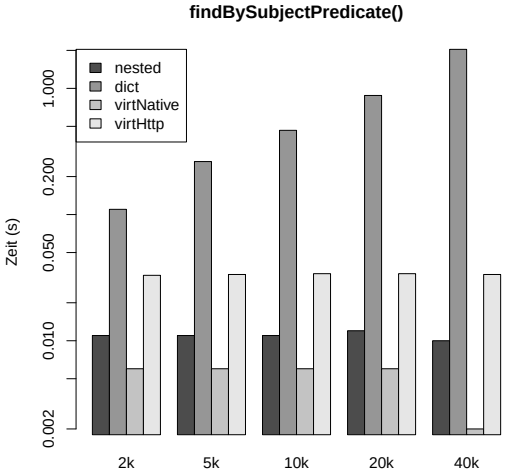

Abbildung 7.4: findBySubjectPredicate() – Zeit

	nested	dict	virtNative	virtHttp
2k	0.0110	0.1100	0.0060	0.0330
5k	0.0110	0.2630	0.0060	0.0335
10k	0.0110	0.4650	0.0060	0.0340
20k	0.0120	0.8785	0.0060	0.0340
40k	0.0100	2.0380	0.0020	0.0335

Tabelle 7.11: findBySubjectPredicate() – Zeit (in s)

Abbildung 7.4 beschreibt das Verhalten der Vergleichsobjekte bei der Abfrageoperation `findBySubjectPredicate()`. Dabei sind die erzielten Ergebnisse im Vergleich zu der Operation `findBySubject()` nahezu identisch. Sowohl die benötigte Zeit, als auch der benötigte Arbeitsspeicher unterscheiden sich kaum. Lediglich virtNative benötigt doppelt so viel Zeit wie bei `findBySubject()`, ist aber dennoch nachwievor die schnellste Implementierung. Bei dem kleinsten Datensatz besteht eine Differenz von 0.005 s

	nested	dict	virtHttp
2k	14.37	30.02	7.52
5k	13.97	62.17	7.51
10k	14.17	65.29	7.51
20k	14.25	165.52	7.52
40k	14.21	320.57	7.51

Tabelle 7.12: findBySubjectPredicate() RSS (in MiB)

	nested	dict	virtHttp
2k	59.28	66.17	54.70
5k	59.26	94.25	54.70
10k	59.23	98.43	54.70
20k	59.26	198.48	54.70
40k	59.26	354.45	54.70

Tabelle 7.13: findBySubjectPredicate() VSize (in MiB)

zu nested, bei dem größten Datensatz eine Differenz von 0.008 s.

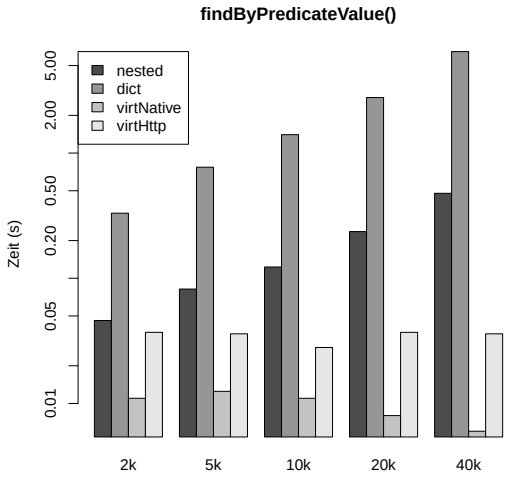

Abbildung 7.5: findByPredicateValue() – Zeit

	nested	dict	virtNative	virtHttp
2k	0.0460	0.3310	0.0110	0.0370
5k	0.0820	0.7710	0.0125	0.0360
10k	0.1230	1.4025	0.0110	0.0280
20k	0.2360	2.7760	0.0080	0.0370
40k	0.4770	6.4630	0.0060	0.0360

Tabelle 7.14: findByPredicateValue() – Zeit (in s)

Die Ergebnisse der Operation `findByPredicateValue()` fallen im Schnitt schlechter aus, als bei den bisherigen Operationen. Während die benötigte Zeit von nested bei

	nested	dict	virtHttp
2k	13.95	30.02	7.50
5k	13.97	62.19	7.51
10k	14.16	65.27	7.52
20k	14.15	165.51	7.51
40k	14.17	187.88	7.51

Tabelle 7.15: findByPredicateValue() RSS (in MiB)

	nested	dict	virtHttp
2k	59.26	66.22	54.70
5k	59.26	94.26	54.70
10k	59.16	97.66	54.70
20k	59.13	198.38	54.70
40k	59.26	224.63	54.70

Tabelle 7.16: findByPredicateValue() VSize (in MiB)

größer werdenden Datensätzen bisher konstant blieb, steigt sie nun durchschnittlich linear an. Die beiden Virtuoso-Implementierungen verbessern hier deutlicher als bei den bisherigen Vergleichen ihre Zeit mit größer werdenden Datensätzen. Bei dem größten Datensatz benötigt wieder dict mit 6.463 s die meiste Zeit. Das macht eine Differenz von 6.457 s im Vergleich zu virtNative. Bei dem Vergleich nested – virtNative besteht eine Differenz von 0.471 s. Bei dem kleinsten Datensatz dagegen entsteht bei dem Vergleich dict – virtNative ein Unterschied von 0.32 s und bei nested – virtNative ein Unterschied von 0.035 s. Des Weiteren ist `findByPredicateValue()` die einzige Operation bei der virtHttp weniger Zeit benötigt als nested. Die Werte des benötigten Arbeitsspeichers sind bei nested und virtHttp wieder nahezu unverändert gegenüber den bisher vorgestellten Operationen. Auch das Verhalten von dict ist unverändert, jedoch benötigt es bei dem größten Datensatz sowohl bei RSS, als auch bei VSize rund 130 MiB weniger Arbeitsspeicher.

Der Grund für das Größerwerden der benötigten Zeit bei nested liegt an der Eigenschaft `value`. Auf diesem Bezeichner wurde kein Index definiert, da die Operation `findByPredicateValue()` im Vergleich zu zum Beispiel `findBySubject()` in der Praxis vermutlich weniger Verwendung finden wird. Darüber hinaus wird mit jedem zusätzlichen Index die Einfügeoperation langsamer. Hier musste also ein gutes Mittelmaß gefunden werden. Deshalb wurde nur auf der URI des Subjekts und des Prädikats ein solcher Index definiert.

Bei den obigen Ergebnissen fällt auch der enorme Zeitbedarf von dict auf. Die Ursache hierfür ist, dass die nötige Anfrage nicht mehr mit der Standardsyntax von MongoDB ausgedrückt werden konnte. Auf Grund der variierenden Bezeichner in den Doku-

menten kann man nicht über die normale Punkt-Notation in Objekten navigieren und muss so manuell über jeden Bezeichner iterieren. MongoDB ermöglicht es aber selbst geschriebene Funktionen[32] auf jedes gespeicherte Dokument anzuwenden. So kann man sehr komplexe Anfragen stellen, worunter die Performanz aber teilweise extrem leidet.

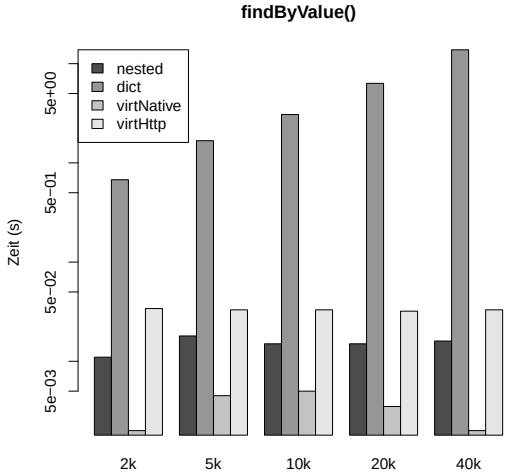

Abbildung 7.6: findByValue() – Zeit

	nested	dict	virtNative	virtHttp
2k	0.0110	0.6760	0.0020	0.0340
5k	0.0180	1.6730	0.0045	0.0330
10k	0.0150	3.0715	0.0050	0.0330
20k	0.0150	6.3355	0.0035	0.0320
40k	0.0160	13.8020	0.0020	0.0330

Tabelle 7.17: findByValue() – Zeit (in s)

Bei den in Abbildung 7.6 dargestellten Ergebnissen wird der Unterschied zwischen bestem und schlechtestem Ergebnis noch größer. Die Differenz der von dict und virtNative benötigten Zeit beträgt 13.8 s. Der Unterschied zwischen nested und virtNative wird im Gegensatz zu den Ergebnissen von `findByPredicateValue()` wieder geringer. Im schlechtesten Fall beträgt dieser Unterschied 0.014 s, im besten Fall 0.009 s. Des

[32] `http://www.mongodb.org/display/DOCS/Server-side+Code+Execution` (28.07.2011)

	nested	dict	virtHttp
2k	14.52	30.16	7.51
5k	14.05	62.19	7.51
10k	14.15	65.24	7.51
20k	14.13	102.99	7.50
40k	14.00	187.84	7.52

Tabelle 7.18: findByValue() RSS (in MiB)

	nested	dict	virtHttp
2k	59.30	66.22	54.70
5k	59.26	94.38	54.70
10k	59.26	97.59	54.70
20k	59.26	136.38	54.70
40k	59.26	224.45	54.70

Tabelle 7.19: findByValue() VSize (in MiB)

Weiteren benötigt nested bei dieser Auswertung im Schnitt nur die Hälfte der von virtHttp benötigten Zeit. Der benötigte Arbeitsspeicher ist bei `findByValue()` nahezu identisch zu dem von `findByPredicateValue()`.
Die Ursachen für die enorme Inperformanz von dict sind die selben, wie schon bei `findByPredicateValue()`. Jedoch muss bei dieser Operation wirklich jedes Dokument von der Queryfunktion geprüft werden. Bei `findByPredicateValue()` wurde dagegen auf Grund der URI des Prädikats schon eine Vorauswahl getroffen. Dieser Umstand führt durchschnittlich zu einer Verdoppelung der benötigten Zeit.

7.2.3 Datenbankgrößen

Abbildung 7.7 zeigt den benötigten logischen Festplattenspeicher bei unterschiedlich großen Datensätzen. Die y-Achse repräsentiert dabei den Speicher in MebiByte.

	nested	dict	virtNative
2k	2.07	2.14	1.72
5k	7.52	4.57	2.79
10k	14.06	8.44	4.16
20k	28.73	17.12	7.17
40k	59.88	35.74	13.87

Tabelle 7.20: logische Datenbankgrößen (in MiB)

Der logische Speicher beschreibt die Größe des rein von den hier betrachteten Daten genutzten Festplatten-Speichers. Datenbanken benötigen jedoch auf Grund von Zwis-

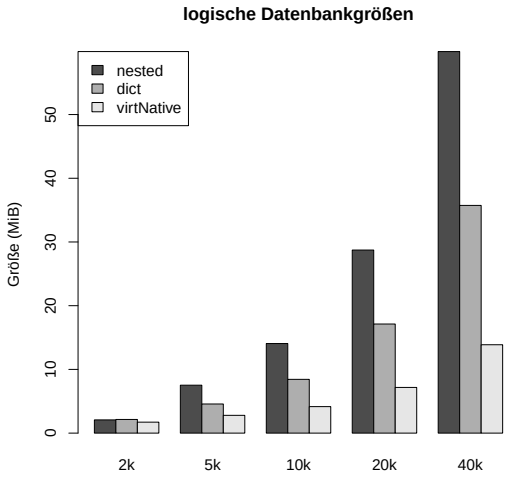

Abbildung 7.7: logische Datenbankgrößen

chenspeichern oder Verwaltungstabellen in der Realität mehr Speicher. Dieser tatsächlich auf der Festplatte eingenommene Speicherbereich wird durch den physischen Speicher betrachtet.
Anhand Abbildung 7.7 kann man erkennen, dass nested bis auf den kleinsten Datensatz durchgängig den meisten Speicher benötigt. Danach folgt dict und schließlich virtNative. Alle drei Vergleichsobjekte wachsen mit größer werdenden Datensätzen annähernd linear. Bei dem größten Datensatz nimmt nested 59.88 MiB ein, Virtuoso 13.87 MiB. Dies ergibt somit eine Differenz von 46.01 MiB. Bei dem kleinsten Datensatz besteht dagegen nur ein Unterschied von 0.35 MiB. Im Verhältnis zu dict benötigt nested, außer bei dem kleinsten Datensatz, im Schnitt um den Faktor 1.6 mehr Speicherplatz.
Dieser konstante Unterschied ist auf die Indizes zurückzuführen. Wie schon in den Analysen zu den Abfrageoperationen erläutert, verwendet nested umfangreichere Indizes, als dict. Der Unterschied des von den Indizes benötigten Speichers ist in Abbildung 7.8 dargestellt.

Aus den hier abgebildeten Werten ergibt sich ein durchschnittlicher Faktor von 3.2, um welchen die Indizes von nested mehr Speicher benötigen, als die von dict. Dieser höhere Speicherbedarf resultiert jedoch, wie in Abschnitt 7.2.2 gesehen, in signifikant besseren Abfrageergebnissen. Des Weiteren wirkt sich dieser Mehrbedarf nicht auf den

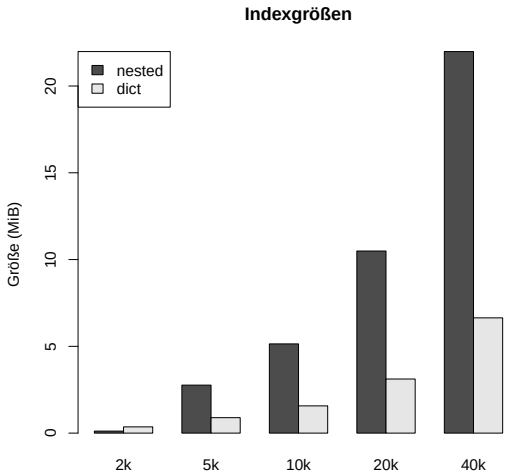

Abbildung 7.8: Indexgrößen

	nested	dict
2k	0.12	0.36
5k	2.77	0.89
10k	5.14	1.57
20k	10.49	3.12
40k	21.98	6.64

Tabelle 7.21: Indexgrößen (in MiB)

physisch benötigten Speicherplatz aus. Dieser ist in der Abbildung 7.9 dargestellt.

Der physische Speicher beschreibt die Größe des tatsächlich auf der Festplatte eingenommenen Speicherbereichs. Abbildung 7.9 zeigt, dass nested und dict jeweils gleich viel Speicher benötigen. Beide wachsen jedoch mit größer werdenden Datensätzen exponentiell an. So benötigen nested und dict bei den Datensätzen 2k und 5k jeweils 64 MiB Speicher, bei 10k und 20k benötigen sie 128 MiB und bei dem größten Datensatz 256 MiB Speicher. Bei dem kleinsten Datensatz besteht zwischen Virtuoso und den MongoDB-Implementierungen ein Unterschied von 28 MiB. Bei dem größten Datensatz dagegen ein Unterschied von 178 MiB.

Die ungewöhnlichen Größen des von MongoDB genutzten Speichers sind darauf zurückzuführen, dass MongoDB die Speicherblöcke exponentiell vergrößert. Das be-

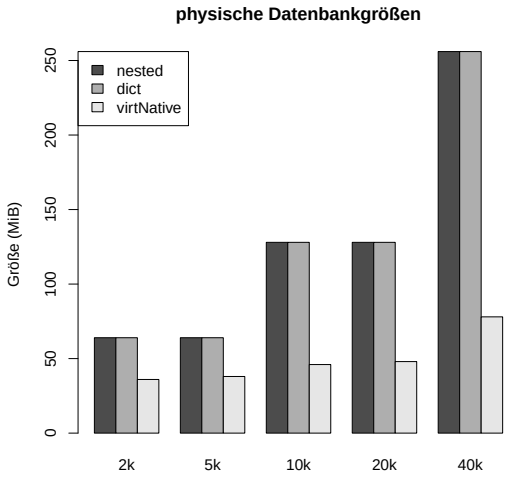

Abbildung 7.9: physische Datenbankgrößen

	nested	dict	virtNative
2k	64.00	64.00	36.00
5k	64.00	64.00	38.00
10k	128.00	128.00	46.00
20k	128.00	128.00	48.00
40k	256.00	256.00	78.00

Tabelle 7.22: physische Datenbankgrößen (in MiB)

deutet, begonnen mit einem Block von 16 MiB wird bei zu groß werdenden Daten ein Block der Größe 32 MiB hinzugefügt. Danach ein 64 MiB großer Block usw. Dies wird bis 2048 MiB so gehandhabt, danach werden nur noch 2048 MiB große Blöcke hinzugefügt.

7.3 Diskussion

Ein Ziel dieser Arbeit war der Vergleich zwischen dem implementierten System und Virtuoso. Die Auswertungen in Kapitel 7.2 bilden dabei die Grundlage für die Klärung der Frage, ob die aus dieser Arbeit entstandene Lösung eine Alternative zu der bisherigen Standard-Lösung Virtuoso darstellen kann.
Nach den durchgeführten Auswertungen muss zunächst festgehalten werden, dass die Unterschiede bei dem Einfügen von Datensätzen so enorm sind, dass man an diesem Punkt schon sagen kann, dass eine Nutzung des entwickelten Systems nur bei einer kleinen Datenbasis oder einer Datenbasis mit wenig Schreibvorgängen sinnvoll ist. Im Gegenzug muss man aber auch feststellen, dass bei den Abfrageoperationen das entwickelte System durchaus eine Alternative zu Virtuoso darstellen kann. Auch wenn die Abfragen über die ODBC-Schnittstelle von Virtuoso immer die besten Ergebnisse liefern, kann die Implementierung nested vorallem bei kleineren Datensätzen nur gering langsamere Ergebnisse vorweisen. Virtuoso wird vorallem bei großen Datensätzen schnell und kann dort größere Differenzen in den Ergebnissen herstellen. Im Vergleich zu der HTTP-Abfrage von Virtuoso kann nested sogar bei allen Operationen außer `findByPredicateValue()` bessere Ergebnisse erzielen. Diese Tatsache ist besonders im Umfeld von Linked Data interessant, wo viele Daten über die HTTP-Schnittstellen der jeweiligen Endpunkte integriert werden.
Des Weiteren kann festgehalten werden, dass die Verwendung der Implementierung dict in keinerlei Weisen sinnvoll ist. Sowohl bei der Einfügeoperation, als auch bei den Abfragen fallen die Ergebnisse durchgängig schlechter aus als bei nested.
Aus Sicht des benötigten Festplattenspeichers der Datenbanken schneiden die Implementierungen auf Basis von MongoDB schlechter ab als Virtuoso. Gerade bei großen Datenbasen benötigt Virtuoso in der Standardinstallation hier weniger Speicher. Jedoch ist dieser Unterschied auf Grund der heutigen Dimensionen im Speichersegment vermutlich vernachlässigbar.
Die Verwendung von MongoDB als Triplestore kann aber auch aus anderen Sichtweisen durchaus eine Alternative zu Virtuoso darstellen. MongoDB bietet z.B. bei den

Punkten Replikation[33] und Skalierbarkeit[34] ein sehr einfach umzusetzendes Konzept. Man kann zusammenfassend also den Schluss ziehen, dass ein Triplestore auf Basis von MongoDB keine vollständige Alternative zu bestehenden Lösungen, wie Virtuoso darstellen kann. Dennoch können vorallem kleinere Datenbasen unproblematisch mit MongoDB umgesetzt werden. MongoDB stellt dabei eine schlanke Umgebung dar, bei der keine größeren administrativen Eingriffe vor dem produktiven Einsatz benötigt werden. Zu einem vergleichbaren Fazit kommt auch der in Kapitel 4 vorgestellte Autor: "Certainly it [MongoDB, M.Nitzschke] is not going to replace dedicated RDF stores but it certainly has potential as a small-scale easy to deploy store."

[33] `http://www.mongodb.org/display/DOCS/Replication` (29.07.2011)
[34] `http://www.mongodb.org/display/DOCS/Sharding` (29.07.2011)

8 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein System entwickelt, das es ermöglicht RDF-Daten unter Verwendung von Node.js in MongoDB zu speichern. Dieses System erforderte es zunächst in Kapitel 5 eine spezielle Repräsentation des RDF-Formates zu entwickeln. Hier entstanden zwei voneinander unabhängige Ansätze. Durch diese Repräsentationen konnte dann eine API entworfen und implementiert werden, die es erlaubt RDF-Tripel in MongoDB zu speichern. Abschließend wurden die beiden erarbeiteten Implementierungen in Kapitel 7 mit dem existierenden Tripelstore Virtuoso Open-Source verglichen. Es konnten demnach alle in Kapitel 3 gestellten Anforderungen umgesetzt werden und somit auch für alle in Kapitel 1.2 dargelegten Probleme eine Lösung gefunden werden.
Der Vergleich zwischen den Implementierungen und Virtuoso zeigte, dass MongoDB als Triplestore keine vollkommene Alternative zu Virtuoso darstellen kann. Dennoch konnte bei den Abfrageoperationen gezeigt werden, dass die Unterschiede der Implementierung nested zu Virtuoso teilweise sehr gering sind. Bei einer Verwendung von Virtuoso über die HTTP-Schnittstelle war die entwickelte Variante sogar bei vier von fünf Operationen performanter. Für spezielle Anwendungsfälle, in denen vorwiegend kleinere bis mittlere Datenbasen verwendet werden und vorwiegend Abfrageoperation ausgeführt werden, kann MongoDB also durchaus eine Alternative zu bestehenden Lösungen darstellen.
Die Ergebnisse der Performanz-Vergleiche in Kapitel 7 legen jedoch auch nahe, perspektivisch an einer ODBC-Anbindung für Node.js zu arbeiten. Dies würde die Verwendung von Virtuoso ermöglichen und somit die performanteste Lösung nach den vorgestellten Ergebnissen darstellen. Die Anbindung könnte dabei entweder direkt über ODBC oder über JDBC mittels einer JDBC-ODBC-Bridge erfolgen. Es gibt derzeit

mit dem Projekt `node-odbc`[35] einen ersten Ansatz für die ODBC-Anbindung.
Inwieweit die entwickelte Lösung in der Praxis Anwendung findet, wird sich also zeigen müssen. Es gibt jedoch zum Beispiel bei dem Projekt LOD2[36] Bestrebungen die Datenbasis unter Zuhilfenahme des in dieser Arbeit entwickelten Systems auf MongoDB umzustellen.

[35] `https://github.com/w1nk/node-odbc` (09.08.2011)
[36] `http://lod2.eu` (08.08.2011)

Literaturverzeichnis

[Ale08] Keith Alexander. RDF/JSON: A Specification for serialising RDF in JSON. In *Scripting for the Semantic Web (SFSW)*, 2008.

[Bal05] Heide Balzert. *Lehrbuch der Objektmodellierung - Analyse und Entwurf mit der UML 2 (2. Aufl.).* Spektrum Akadem. Verl., 2005.

[BL98] Tim Berners-Lee. Cool URIs don't change, 1998.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[MM04] Frank Manola and Eric Miller, editors. *RDF Primer*. W3C Recommendation. World Wide Web Consortium, February 2004.

[tur08] Turtle - Terse RDF Triple Language, W3C Team Submission, 2008. See: http://www.w3.org/TeamSubmission/turtle/.

Abbildungsverzeichnis

Tabellenverzeichnis

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.

Leipzig, 28. August 2011 Unterschrift