Universität Leipzig
Institut für Informatik
Abteilung Automatische Sprachverarbeitung
Topicmapslab

Bachelorarbeit

CouchTM - Eine Topic-Maps-Engine mit CouchDB-Backend

Vorgelegt von
Hans-Henning Koch
Mainzer Straße 2a, 5/411
04109 Leipzig
3. November 2010

Betreut von Prof. Dr. Gerhard Heyer und Benjamin Bock

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Leipzig, 3. November 2010 _____

Unterschrift

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1. Einleitung

Topic Maps ist eine Technologie aus dem Bereich der Wissensmodellierung. Es ist ein abstraktest Modell zur Formulierung von Wissensstrukturen. Dazu existieren verschiedene Arten von Topic-Map-Konstrukten. Die zentralen Konstrukte sind Topics. Durch sie werden Aussagegegenstände in der Topic Map repräsentiert. Die Topics können durch Occurrences mit Informationsressourcen verknüpft werden. Topics können Names besitzen, die in einem bestimmten Kontext die Bezeichnungen des Topics darstellen. Weiterhin können Names Variants besitzen, die alternative Topic-Namen darstellen, welche in einem bestimmten Kontext besser geeignet sind, als der entsprechende Name. Um Beziehungen zwischen Topics darzustellen, gibt es Associations und AssociationRoles. Die Associations repräsentieren Beziehungen zwischen einem oder mehreren Topics. AssociationRoles verknüpfen Topics und Associations und modellieren so die Beteiligung von Aussagegegenständen an einer Beziehung. Bis auf Variants können Konstrukte auch einen Typ besitzen, welcher die Art des Konstruktes näher definiert. Eine Topic Map ist schließlich das Konstrukt, welches das so modellierte Wissen in Form von Topic-Map-Konstrukten beinhaltet. Die theoretische Grundlage für Topic Maps bildet das Topic-Maps-Referenzmodell[ISOb]; das Topic-Maps-Datenmodell[ISOa] beschriebt, wie die Theorie in abstrakten Strukturen dargestellt werden kann (in verkürzter Form oben geschehen). Um Topic Maps in Anwendungen einsetzen zu können, bedarf es einer Topic-Maps-Engine. Das ist eine Programmbibliothek, durch die Topic Maps erstellt, modifiziert und abgefragt werden können. Dabei stellt die Engine auch sicher, dass alle im Topic-Maps-Datenmodell angeführten Bedingungen eingehalten werden.

Es existiert bereits eine Vielzahl von unterschiedlichen Topic-Maps-Engines. Neu entwickelte Engines müssen daher Innovationen betreffend der Funktionalität oder der eingesetzten Technologien beinhalten, um sich von der breite Masse abzugrenzen. CouchTM soll keine neuen Funktionalitäten implementieren, aber im Backend-Bereich mit CouchDB, einem dokumentenorientierten Datenbankmanagementsystem, eine Technologie einsetzen, die bisher von keiner Topic-Maps-Engine unterstützt wird. Somit ist CouchTM die erste Engine, welche CouchDB und insbesondere ein dokumentenorientiertes DBMS benutzt. Topic-Map-Konstrukte können dabei als Dokumente aufgefasst werden. Da sowohl Dokumente und teilweise auch Topic-Map-Konstrukte eine variable Struktur besitzen können, ist solch eine Repräsentation auch durchaus angebracht. Sie entspricht auch viel mehr der Natur von Topic-Map-Konstrukten, als die Repräsentation in relationalen Datenbankmanagementsystemen. Die in einem Konstrukt enthaltene Informationseinheit kann als ein ganzes, in einem Dokument, in der Datenbank gespeichert werden. Sie muss nicht aufgesplittet werden, um so über viele Tabellen verteilt in der Datenbank gespeichert zu werden.

CouchTM soll in Java geschrieben werden und die bestehenden TMAPI-Interfaces im-

plementieren. Die Engine soll außerdem automatisches Mergen unterstützen. Zunächst muss ein Konzept gefunden werden, wie Topic-Maps- Konstrukte am besten auf dokumentenorientierte Datenbanken abgebildet werden können. Weiterhin müssen Views entwickelt werden, durch welche die Konstrukte wieder aus der Datenbank abgefragt werden können. Zum Testen der Implementation stehen die TMAPI- und RTM-Tests zur Verfügung. Sämtliche Tests dieser Testframeworks sollen von CouchTM erfolgreich absolviert werden. Neben der Korrektheit soll auch die Geschwindigkeit der Implementation getestet werden. Aus diesen Tests ist dann abzuleiten, ob sich dokumentenorientierte Datenbankmanagementsysteme für den Einsatz als Topic-Maps-Engine-Backend eignen und in welchen Szenarien CouchTM zum Einsatz kommen kann.

Einleitend werden vier Topic-Maps-Engines kurz vorgestellt, um einen Überblick über bestehende Engines zu erlangen. Diese werden im Folgenden auch zum Vergleich der Geschwindigkeit herangezogen. Als nächstes wird näher auf CouchDB eingegangen. Es wird erklärt, was dokumentenorientierte Datenbanksysteme sind und von den anderen Datenbanktechnologien abgegrenzt. Weiterhin werden die grundlegenden Konzepte hinter CouchDB erläutert und an Beispielen veranschaulicht. Im darauf folgenden Kapitel wird näher auf CouchTM eingegangen und das Zusammenwirken mit der Datenbank und Topic Maps erläutert. Außerdem wird noch angesprochen, welche Funktionalitäten es nicht in die aktuelle Version von CouchTM geschafft haben, aber in Folge einer Weiterentwicklung interessant sind. Die letzten beiden Kapitel beschäftigen sich mit der Evaluation von CouchTM. Es werden die Tests zur Korrektheit und Performance vorgestellt und die Ergebnisse diskutiert.

2. Topic-Maps-Engines Überblick

Es gibt die verschiedensten Topic-Maps-Engines mit einer Vielzahl an unterschiedlichen Backends. Jedoch benutzt bisher keine dieser Engines ein dokumentenorientiertes Datenbanksystem. Um einen Überblick über bestehende Topic-Maps-Engines und ihre Backends zu erlangen, werden in diesem Kapitel einige bekannte Engines kurz vorgestellt.

2.1. tinyTiM

TinyTiM[1] ist eine schlanke, Open-Source, Java TMAPI Implementation. Es besitzt kein persistentes, sondern nur ein in-memory-Backend. Dies mag ein Nachteil sein, jedoch ist es auch der Grund für eine der großen Stärken von tinyTiM: die Geschwindigkeit. In vielen Tests ist tinyTiM hierbei den anderen Engines mit persistentem Backend überlegen, da alle Operationen im schnellen Arbeitsspeicher durchgeführt werden können und der Overhead eines DBMS sowie das langsamere Schreiben auf die Festplatte entfallen.

Es ist jedoch auch möglich, Topic Maps in verschiedenen Fileformaten zu laden und zu serialisieren. Hierzu besitzt tinyTIM eigene TopicMapReader und -writer, oder man benutzt Tools wie z.B TMAPIX[2]. Dies ist eine TMAPI-Extension, welche für alle TMAPI implementierenden Engines einsetzbar ist und es erlaubt, Topic Maps in verschiedenen Formaten in die Engines zu laden und wieder zu serialisieren. Unterstützt werden unter anderem CTM, CXTM, JTM, RDF, TMXML und XTM in verschiedenen Versionen.

Trotzdem ist es jedoch kaum möglich, diese Engine in einer Multi-User-Umgebung einzusetzen oder dort, wo die Dauerhaftigkeit einer Operation von Interesse ist. Vor allem bei größeren Topic Maps ist das Serialisieren mit TMAPIX vergleichsweise teuer und kann daher nicht nach jeder Schreiboperation durchgeführt werden. Auch die anderen ACID-Kriterien, atomacy, consistency und isolated executions, sind kaum anwendbar.

2.2. Ontopia

Die Ontopia Topic Maps Engine[3] ist Teil eines Software-Development-Kit für Topic-Maps-Applikationen, früher als Ontopia Knowledge Suit (OKS) bezeichnet. Das Softwarepaket beinhaltet Tools zum Erstellen, Verwalten und Betreiben von Topic-Maps-Applikationen. Ontopia wurde in Java geschrieben und besitzt eine eigene API, aber auch die TMAPI wird

[1] tinyTiM, http://tinytim.sourceforge.net/
[2] TMAPIX, http://code.google.com/p/tmapix/
[3] Ontopia, http://www.ontopia.net

unterstützt.

Auch mit Ontopia können Topic Maps in XML ähnlichen Formaten wie XTM importiert und exportiert, aber auch in einem RDBMS verwaltet werden. Hierzu verfügt die Engine über ein RDBMS-Connector Add-on, welches den Zugriff auf die unterstützten RDBMS verwaltet. Unterstützt werden PostgreSQL, Oracle, SQL Server, MySQL und H2 Database Engine. Auf Grund der einheitlichen API macht es für den Entwickler keinen Unterschied, welches Backend er nutzt und auch der Wechsel von Backends ist sehr einfach zu realisieren. Neben den persistenten Backends besitzt Ontopia noch ein in-memory-Backend.

Die Ontopia Topic Maps Engine verfügt weiterhin über eine Reihe von Tools, welche den Umgang mit Topic Maps erleichtern. Die Full-Text Search Integration ermöglicht es, nach Text in Topic Maps zu suchen. Eine Query-Engine mit der Sprache Tolog erlaubt es, Anfragen an eine Topic Map zu stellen. Und die Ontopia Schema Tools ermöglichen es, die Struktur und Constraints auf Topic Maps zu beschreiben.

Ontopia begann als ein kommerzielles Produkt, wurde jedoch kürzlich auf Open-Source umgestellt. Es kommt bei vielen Projekten zum Einsatz, z.B. beim Bürgerportal der Stadt Bergen oder im Office of Naval Intelligence, um Daten über ausländische Flotten und Schiffe zu verwalten.

2.3. RTM

Ruby Topic Maps, kurz RTM[4], ist ein Open-Source-Projekt an der Universität Leipzig. Es begann als Diplomarbeit[Boc09] und wurde seit dem ständig gepflegt und weiterentwickelt. RTM ist eine in Ruby[5] geschriebene Topic-Maps-Engine, welche das TMDM implementiert und eine speziell dafür entwickelte domain-specific Language (DSL) benutzt. Die Engine verfügt standardmäßig über ein in-memory-Backend sowie ein persistentes Backend, welches auf ActiveRecord von Ruby beruht. Das in-memory-Backend nutzt SQLite :memory:, das persistente Backend unterstützt sämtliche Datenbanken, die über ActiveRecord angesprochen werden können.

Zusätzlich können auch noch die Backends anderer Topic-Maps-Engines benutzt werden, z.B. Ontopia, tinyTIM und auch CouchTM. RTM, in diesem Zusammenhang auch als JRTM bezeichnet, fungiert in diesem Fall als Wrapper und erlaubt es, auf in Java geschriebene und TMAPI implementierende Topic-Maps-Engines zuzugreifen [BBSM09]. Hierzu wird JRuby[6], eine in Java geschriebener Ruby-Interpreter, verwendet. Durch Ausnutzung der Namensgebungskonventionen von Java und JRuby können die Java-Funktionen der TMAPI sehr einfach gemapt werden. Die Verwendung als Wrapper ist gegenwärtig das hauptsächliche Einsatzgebiet von RTM.

RTM verfügt über umfangreiche Tests, welche auch zum Testen von CouchTM verwendet wurden.

[4]Ruby Topic Maps, http://rtm.topicmapslab.de/
[5]Ruby, http://www.ruby-lang.org/
[6]JRuby, http://jruby.org/

2.4. MajorToM

MajorToM[7] ist eine weitere Topic-Maps-Engine des Topicmapslab der Universität Leipzig und befindet sich derzeit noch in der Entwicklungsphase. Das Ziel des Projektes ist es, eine leichtgewichtige, mergende und flexible Topic-Maps-Engine zu schaffen. Es werden die TMAPI-Interfaces implementiert, und somit ist die Engine leicht in jeder auf dieses Interface ausgelegten Applikation einsetzbar. Jedoch ist MaJorToM keine reine TMAPI-Implementation, sondern erweitert sie in seinen eigenen Interfaces. So werden z.B., anders als in der TMAPI, supertype-subtype-Relationen explizit unterstützt. Die Interfaces beinhalten aber auch einige neue Ansätze: Durch die Einführung spezieller Occurrence-Typen wird die Modellierung von Zeit und Raum unterstützt. Jedes Objekt besitzt eine persistente History und die Engine stellt ein Eventmodell zur Verfügung, welches Listener einer Applikation benachrichtigt, wenn ein spezielles Event ausgelöst wird. Weiterhin besitzt die Engine ein Transaktionsmodell, das es erlaubt, eine Reihe von Änderungen in einer atomaren Transaktion zusammen zu fassen.

Um die gewünschte Flexibilität zu gewährleisten, erfolgt eine Trennung von Applikations- und Informationsdomäne. Die Kommunikation geschieht über Object-Stubs, welche kein Wissen darüber haben, was sie darstellen. Dieses Wissen besitzt die Engine und kann die Daten der Object-Stubs über abstrakte Methoden abfragen.

MajorToM verfügt über ein in-memory-Backend, sowie ein relationales Datenbankbackend, das zur Zeit MySQL und PostgreSQL unterstützt. Optimiert wurde das relationale Datenbankbackend bisher nur für PostgreSQL.

[7] MaJorToM, http://code.google.com/p/majortom/

3. CouchDB

3.1. Allgemeines

CouchDB[8] ist ein Open-Source dokumentenorientiertes DBMS, welches von der Apache Software Foundation entwickelt wird. Dokumentenorientiert bedeutet, dass ein Datensatz nicht wie bei relationalen Datenbanken über viele Tabellen verteilt gespeichert wird, sondern als ein Ganzes, also in einem Dokument, in der Datenbank abgelegt wird. Zur Zeit befindet es sich in Version 1.0.1.

CouchDB ist damit der Familie der NoSQL-Datenbanken zuzuordnen. NoSQL bedeutet zum einen, dass nicht(nur) SQL als Anfragesprache zur Verfügung steht und zum anderen, dass die Konzepte von denen der relationalen Datenbanken sehr stark abweichen. Solche Datenbanken sind meist für den Einsatz im Web gedacht und dahingehend angepasst. Es wird kein striktes Schema benötigt, da in den meisten Anwendungsfällen nur mit semistrukturierten Daten gearbeitet wird. Weiterhin wird das Konzept der Konsistenz anders interpretiert als bei relationalen Datenbanken. Das CAP-Theorem besagt, dass aus den Anforderungen Konsistenz, Verfügbarkeit und Partitionstoleranz nur zwei gleichzeitig erfüllt werden können [Bre00]. Hierbei sind im Web-Umfeld Verfügbarkeit und Partitionstoleranz wesentlich wichtiger als die Konsistenz. Daher machen diese Datenbanken leichte Abstriche hinsichtlich der Konsistenz und garantieren nur "eventual consistency". Dies bedeutet, dass nach Veränderungen letztlich alle Prozesse die geänderten Daten erhalten, es jedoch ein kleines Zeitfenster gibt, in dem noch die veralteten Daten gelesen werden.

Die Gruppe der NoSQL-Datenbanken lässt sich in 4 Klassen unterteilen: Key-Value-Stores, dokumentenorientierte Datenbanken, Column-Stores und Graphendatenbanken [NSQ10].

Key-Value-Stores sind hierbei die rudimentärste Klasse. Sie erlauben Anfragen nach Keys, welche auch sehr performant sind, jedoch keine komplexeren Anfragen. Es gibt sie als on-disk Datenbanken und in-memory-Datenbanken, deren Einsatzgebiet verteilte Cachesysteme sind. Der bekannteste Vertreter der Klasse der Key-Value-Stores ist Amazons S3[9].

Eine weitere Klasse sind die dokumentenorientierten Datenbanken. Sie bestehen aus semi-strukturierten Dokumenten, die meist in XML, JSON oder ähnlichen Auszeichnungssprachen verfasst sind. Dadurch kann jeder Eintrag in der Datenbank eine eigene Struktur haben und es wird kein Datenbankschema benötigt. Weiterhin erlauben diese Datenbanken auch komplexe Anfragen, die meist in einer Scriptsprache wie z.B. Javascript verfasst werden. Die bekanntesten Vertreter sind hier CouchDB und MongoDB[10].

[8] CouchDB, http://couchdb.apache.org
[9] Amazon Simple Storage Service (S3), http://aws.amazon.com/s3/
[10] MongoDB, www.mongodb.org/

In Column-Stores speichert man die Daten nach Spalten, wodurch ähnliche Daten zusammen gespeichert werden. Anfragen können hier nach Key und Attributwerten gestellt werden. Diese Art von Datenbanken sind daher besonders geeignet, wenn Anfragen an eine große Menge ähnlicher Daten gestellt werden, wie es z.B. bei Datawarehouses der Fall ist. Die bekanntesten Vertreter dieser Klasse sind Google BigTable[11] und HBase[12].

Die letzte Klasse sind die Graphendatenbanken, in denen die Daten in Graphen- oder Netzwerkstruktur abgelegt werden. Durch diese Struktur skalieren sie gut und es ist in ihnen besonders performant, Relationen zu durchlaufen. Bekannte Vertreter dieser Klasse sind HyperGraphDB[13] und Neo4J[14].

Bei der Entwicklung von CouchDB stand die Skalierbarkeit (scale-out) und der Einsatz im Internet im Vordergrund. Zahlreiche Websites nutzen CouchDB, z.B. pkw.de oder die Torrent Suchmaschine gpirate, doch es kommt auch in anderen Gebieten zum Einsatz. So wird CouchDB z.B. von Ubuntu benutzt, um Adress- und Bookmarkdaten zu synchronisieren.

CouchDB ist in Erlang[15] geschrieben, einer funktionalen Programmiersprache, die ursprünglich für den Einsatz von real-time Telekommunikationsanwendungen entwickelt wurde [Arm07]. Erlang bietet vor allem eine gute Unterstütung von Nebenläufigkeit. Eine Applikation besteht hauptsächlich aus einer Menge von leichtgewichtigen Prozessen, welche mittels message passing mit einander kommunizieren und keine geteilten Zustände haben. Dadurch kann mit Erlang leicht eine lock-freie Nebenläufigkeit erreicht werden. Dies ist ganz im Sinne von CouchDBs Designkonzept und mit ein Grund, weswegen es in Erlang entwickelt wurde.

3.2. Datenbankzugriff und Transaktionen

Der Zugriff auf CouchDB erfolgt mittels einer RESTful JSON-API. Jedes Dokument wird als eine Ressource behandelt und besitzt eine eindeutige URI. Die REST-Methoden GET, PUT, POST und DELETE werden genutzt, um Operationen auf diesen Ressourcen durchzuführen [FST02]. Die Datenbank- und View-API funktionieren in gleicher Weise.

```
Beispiel: Erstellen einer Datenbank
PUT http://dbadress/databasename

Beispiel: Abfragen eines Dokuments
GET http://dbadress/database/documentid

Beispiel: Löschen eines Dokuments
DELETE http://dbadress/database/documentid?rev=current_revisionnumber
```

[11] Google BigTable, http://labs.google.com/papers/bigtable-osdi06.pdf
[12] HBase, http://hbase.apache.org/
[13] HyperGraphDB, http://www.kobrix.com/hgdb.jsp
[14] Neo4J, http://neo4j.org/
[15] Erlang programming language, http://www.erlang.org/

Alle Transaktionen in CouchDB genügen weitestgehend den ACID Kriterien, wobei unter Konsistenz die "eventual consistency" verstanden wird. Sie sind "all or nothing", können also nur erfolgreich oder nicht erfolgreich sein, d.h. die Daten werden nur im Erfolgsfall geschrieben. Das Speichern erfolgt in zwei Schritten. Im 1. Schritt wird das geänderte Dokument und alle View-Updates synchron geschrieben. Im 2. Schritt wird der Datenbankheader mit neuer Sequenznummer der Datenbank, bestehend aus zwei identischen und aufeinander folgenden Teilen geschrieben. Kommt es in der ersten Phase zu einem Crash, so werden partiell geschriebene Änderungen einfach vergessen. Bei einem Crash in der zweiten Phase ist ein Header immer noch intakt. Dieser wird übernommen und Dokumente, welche nicht der Datenbank mit dieser Sequenznummer angehören, werden verworfen. Nach einem Crash entfallen also aufwendige Wiederherstellungsmaßnahmen. Die Konsistenz ist ebenfalls gewährleistet, da auf Grund des Fehlens eines Schemas keine diesbezüglichen Inkonsistenzen auftreten können. Wie oben beschrieben, verursacht auch ein Crash keine Inkonsistenzen. Das "isolated executions"-Kriterium ist teilweise erfüllt und CouchDB benötigt dazu, anders als RDBMS, keine Locks. Wird ein Dokument gelesen, während es noch einen Schreibvorgang durchläuft, wird die Version mit der Revisionsnummer vor dem Schreibvorgang zurück gegeben, bis der Schreibvorgang abgeschlossen und somit die neue Version verfügbar ist. Schreiben zwei Prozesse das selbe Dokument, so wird die zuerst verarbeitete Änderung geschrieben, die zweite Transaktion enthält dann eine veraltete Revisionsnummer. Die in ihr enthaltenen Änderungen werden nicht geschrieben, sondern es wird eine Fehlermeldung zurück gegeben, die einen Revisionskonflikt anzeigt. Es können also zwar zwei Operationen auf das selbe Dokument zugreifen, jedoch keine Inkonsistenzen im Sinne der "eventual consistency" entstehen. Eine positive Rückmeldung einer schreibenden Transaktion erfolgt erst, nachdem die Daten auch wirklich geschrieben wurden. Somit ist schließlich das letzte Kriterium, "Durability", erfüllt.

3.3. Dokumente

In CouchDB wird ein Datensatz nicht aufgesplittet und auf viele Tabellen verteilt gespeichert, sondern als ein Ganzes in einem semi-strukturierten Dokument. Die Dokumente sind hierbei JSON[16]-Objekte, also eine Menge von Key-Value-Paaren, wobei die Werte wiederum JSON-Objekte sein können. Die Struktur eines Dokumentes kann jederzeit beliebig verändert werden und unterliegt keinen Beschränkungen. Daher benötigt CouchDB, anders als RDBMS, keine Schemata. Folglich existieren auch keine Referenzen zwischen Dokumenten. Dies kann jedoch mit Views modelliert werden.

Ein Dokument muss eine ID besitzen, durch die es eindeutig in der Datenbank identifiziert wird. Die ID kann entweder beim Erstellen des Dokumentes angegeben werden, oder wird andernfalls automatisch einem neuen Dokument von der Datenbank zugewiesen. Weiterhin besitzt jedes Dokument eine Revisionsnummer. Diese wird beim Erstellen automatisch zugewiesen und bei Änderungen automatisch erhöht. In der Antwort der Datenbank

[16]Javascript Object Notation, http://www.json.org/

bei einer schreibenden Aktion findet sich dann die neue Revisionsnummer des Dokumentes. Veraltete Revisionen eines Dokumentes bleiben weiterhin in der Datenbank gespeichert, bis diese automatisch oder manuell gelöscht werden. Sie können bis dahin jedoch durch Angabe dieser Revisionsnummer immer noch abgefragt werden. Lesende Anfragen an die Datenbank ohne eine Revisionsnummer liefern stets die aktuellste Version eines Dokumentes zurück.

Auf Dokumenten gibt es keine Locks, das heißt, dass Nutzer gleichzeitig das Selbe Dokument lesen und auch schreiben können. Wird bei einer schreibenden Aktion festgestellt, dass für das betreffende Dokument die Revisionsnummer veraltet ist, so wird ein Fehler zurück gegeben. Das Dokument muss erneut geladen und die Änderungen abermals ausgeführt werden, bevor sie schließlich gespeichert werden kïnen.

Eine zweite Art von Dokumenten sind die design documents. Der ID bzw. dem Namen dieser Dokumente muss "_design/" vorangestellt werden. Sie beinhalten keine Daten, wie es normale Dokumente tun, sonder eine Menge an Javascript-Funktionen, die auf den normalen Dokumenten arbeiten und der CouchDB nutzenden Applikation mehr Funktionalität anbieten. Die am meisten genutzten Bestandteile von Designdokumenten sind zweifelsohne die Views, auf welche im folgenden Abschnitt näher eingegangen wird. Es existieren jedoch noch eine Reihe von weiteren nützlichen Bestandteilen: list-, show- und validate-Funktionen sowie Librarys und Attachments. Auch Designdokumente besitzen eine Revisionsnummer und werden wie normale Dokumente repliziert.

Jedes Designdokument kann eine Validierungsfunktion validate_doc_update besitzen. Werden mehrere Funktionen benötigt, so muss für jede ein Designdokument angelegt werden. Valitationsfunktionen werden bei jeder Änderung eines Dokumentes sowie bei der Erstellung eines Dokumentes aufgerufen, bevor es gespeichert wird. Designdokumente sind jedoch davon ausgenommen. Der Sinn der Validierungsfunktion ist es, wie der Name schon sagt, die betreffenden Dokumente zu validieren. Typischerweise wird getestet, ob das Dokument bestimmte Attribute besitzt, ob ein Attributwert eine Bestimmte Form hat (z.B. Datum) und der gleichen.

In vielen Anwendungsfällen, vor allem im Web-Umfeld, ist es wünschenswert, dass das Resultat einer Anfrage an die Datenbank nicht im JSON-Format vorliegt, sondern in einer Form, die direkt ausgegeben werden kann, wie z.B. HTML. Hierfür stellt CouchDB show-Funktionen zur Verfügung. Sie sind ebenfalls Javascript-Funktionen, welche in Designdokumenten unter dem Schlüssel "shows" gespeichert werden. Sie werden für ein bestimmtes Dokument, identifiziert durch seine ID, aufgerufen und geben eine HTTP-Antwort, bestehend aus Header und Body, zurück. Im Body Teil kann beschrieben werden, wie das Dokument aufbereitet werden soll (z.B. Einfügen von <h1> und </h> vor bzw. nach doc.title). Im Header wird der content-type des Bodys spezifiziert. Wird kein spezieller Header angegeben, so wird automatisch einer mit dem content-type text/html angefügt. Shows erlauben es auch, Binärdaten in base64 zurück zu geben.

Die letzte Art von Funktionen in Designdokumenten sind list-Funktionen. Sie sind den show-Funktionen sehr ähnlich, allerdings mit dem kleinen Unterschied, dass sie nicht auf einzelnen Dokumenten, sondern auf Viewresultaten arbeiten. Ihr Schlüssel in Designdokumenten ist "lists" und aufgerufen werden sie mit einem Viewnamen und gegebenenfalls Keys.

```
Im folgenden sei _design/example der Name eines Designdokumentes.

Beispiel: Abfragen des Views test mit dem Key foo
GET http://dbadress/databasename/_design/example/_view/test?key={foo}

Beispiel: Abfragen des Shows für das Dokument mit der ID foo
GET http://dbadress/databasename/_design/example/_show/foo

Beispiel: Abfragen der List examplelist für den View test und dem Key foo
GET http://dbadress/databasename/_design/example/_list/examplelist/
                                                        viewname?key={foo}
```

Zur Speicherung der Dokumente kommen bei CouchDB zwei B+-Bäume zum Einsatz. In ihnen werden die Dokumente nach ID und nach Sequenznummer der Datenbank gespeichert. Wird ein Dokument geändert, gelöscht oder neu erstellt, so wird der betroffene Knoten des Baumes nicht mitten im File geändert, sondern eine Kopie erstellt und am Ende des Files angefügt. Alle anderen Knoten, auf die sich diese Änderung auch ausgewirkt haben, werden ebenfalls kopiert und in der geänderten Form am Ende des Files angehängt. Dieser Mechanismus hat sowohl Vor- als auch Nachteile. Hält man an einer Wurzel des Baumes fest, so hat man zu jeder Zeit eine konsistenten und sich nicht verändernde Sicht auf die Datenbank, auch wenn weiter auf ihr gearbeitet wird. Die Datenbank Files wachsen dadurch jedoch relativ stark an. Daher muss gelegentlich ein Compaction-Prozess ausgeführt werden, welcher die Files aufräumt.

3.4. Views

Ohne Views wäre es nur möglich, Dokumente anhand ihrer ID abzufragen. Die Datenbank wäre somit quasi nur ein Key-Value-Store. Das ist natürlich nicht genug für ein vollwertiges DBMS, doch durch Views wird dieser Makel behoben. Durch sie ist es auch möglich, komplexe Anfragen an die Datenbank zu stellen. Views werden in Designdokumenten unter dem Schlüssel “views” gespeichert und auch mit repliziert.

Views bestehen standardmäßig aus ein bis zwei Javascript-Funktionen: einer Map-Funktion und optional einer Reduce-Funktion. Existiert eine Reduce-Funktion, so wird mit MapReduce gearbeitet - mehr dazu im folgenden Abschnitt.

Die Map-Funktion durchläuft nacheinander die Dokumente in der Datenbank und kann mittels emit Key-Value-Paare zurückgeben, die JSON-Objekte beinhalten. Weiterhin wird noch die ID des Dokumentes angegeben, dessen Betrachtung das emit ausgelöst hat. Das Resultat einer Viewanfrage, also alle ausgegeben Key-Value-Paare, sind nach ihrem Key geordnet. Die Values können komplette Dokumente aus der Datenbank sein, aber auch neue Objekte, die aus Teilen von Dokumenten zusammengesetzt oder berechnet wurden. Da immer nur ein Dokument betrachtet wird, kann aber jedes emit einer Map-Funktion nur

Daten, die aus einem einzelnen Dokument stammen, zurückgeben - zumindest bis Version 0.10. In Version 0.11 wurde include_docs für Views eingeführt. Durch dieses Feature ist es möglich, in einem View anstelle des betrachteten Dokumentes durch emitten des Values {_id: doc.somevalue} das Dokument zurück zu geben, welches die ID hat, die durch den Wert eines Attributes im eigentlich betrachteten Dokument bestimmt ist. Dieses Dokument wird jedoch nicht in das Erstere eingebettet, sondern nur mit in das Resultat des Views übernommen. Das Einbetten kann allerdings auf Applikationsebene sehr einfach selber gemacht werden. Die ID dieses Vieweintrages ist dabei diejenige des von der Map-Funktion betrachteten Dokumentes. Die Anzahl an emits pro betrachtetem Dokument ist nicht festgesetzt.

Views werden im Allgemeinen mit Parametern aufgerufen. Der gebräuchlichste Parameter ist hierbei "key=xxx", wobei xxx ein JSON Objekt ist. In diesem Falle werden nur diejenigen Einträge eines Views zurück gegeben, deren Key dem des Aufrufes entspricht. Weitere Parameter sind z.B: "startkey" und "endkey", durch welche eine Keyrange festgelegt werden kann, "limit" zum Festlegen der maximal gewünschten Resultate oder "descending", um die Ordnung der Resultate umzukehren.

```
Beispiel:
map: function(doc) {
      if(doc.age == 'male') {
           emit(doc.age, doc);
      }
}
```

Dieser View gibt alle Dokumente zurück, die ein "age"-Key mit dem Value "male" haben. Durch einen Aufruf mit z.B. "?key={value}" kann das Resultat des Aufrufes auf solche Dokumente beschränkt werden, bei denen "age" gleich dem übergebenen Wert ist.

Wird ein View das erste Mal aufgerufen, so wird die Map-Funktion (und falls vorhanden, auch die Reduce-Funktion) wie oben beschrieben abgearbeitet. Dabei werden die Resultate in einem B+-Tree, dem View-Index, gespeichert und auch die aktuelle Sequenznummer der Datenbank wird erfasst. Bei jedem geänderten oder neu erstellten Dokument ändert sich jedes mal die Sequenznummer der Datenbank. Wird nun ein View erneut aufgerufen, so vergleicht CouchDB die aktuelle Sequenznummer mit jener im View gespeicherten. Dadurch können alle Änderungen im Vergleich zum vorhergehenden Aufruf identifiziert werden. Daraufhin wird die Map-Funktion für alle geänderten und neuen Dokumente aufgerufen und die jeweils emitteten Rows dem View-Index angefügt sowie die im View gespeicherte Sequenznummer aktualisiert. Dies geschieht auch für alle anderen Views im selben design document. Jedes Dokument in der Datenbank muss also pro Revisionsnummer nur höchstens einmal von jeder View-Funktion betrachtet werden.

In B+-Trees sind die Anfragen an einen Key oder eine Key-Range sehr performant - im worst-case in O(log n). Durch ihre Verwendung ist es auch möglich auf großen Datenmengen noch effizient zu arbeiten, da nur beim ersten Aufruf alle Dokumente in der Datenbank

betrachtet werden müssen. Der Unterschied zu normalen B-Bäumen ist, dass die Daten nur in den Blättern des Baumes gespeichert werden; die inneren Knoten enthalten nur Keys. Sie sind etwas performanter als B-Bäume, brauchen allerdings etwas mehr Speicherplatz.

In der Entwicklungsphase kann man auch mit den weniger performanten temporary Views arbeiten, die nicht in der Datenbank gespeichert, sondern bei jedem Aufruf neu erzeugt werden. Weiterhin ist es möglich, andere Sprachen als JavaScript für Views zu verwenden, was allerdings manuell konfiguriert werden muss.

3.5. MapReduce

Ist eine Reduce-Funktion vorhanden, so wird mit angepasstem MapReduce gearbeitet [DG04]. Die Reduce-Funktion hat drei Argumente: keys - ein Array der von der Map-Funktion emitteten Keys, values - die dazugehörigen von der Map-Funktion emitteten values und rereduce. Rereduce ist ein boolean Wert, der angibt, ob die zu reduzierenden Werte aus den Blättern des B+-Trees (rereduce = false) oder inneren Knoten (rereduce = true) kommen. So kann durch eine einfache if-Anweisung in der Reduce-Funktion erkannt werden, welcher Fall gerade vorliegt.

Die Reduce-Funktion arbeitet also auf den von der Map-Funktion kommenden Daten und berechnet daraus dann, beginnend bei den Blättern des B+-Trees, die Zwischenwerte für jeden inneren Knoten des Baumes. An der Wurzel angekommen wird noch einmal reduziert und dann das Endergebnis zurückgegeben. Gibt es in einem Blatt eine Änderung, so müssen nur diejenigen Werte neu berechnet werden, die sich in den inneren Knoten des Baumes befinden, welche auf dem Pfad vom betreffenden Blatt zur Wurzel liegen.

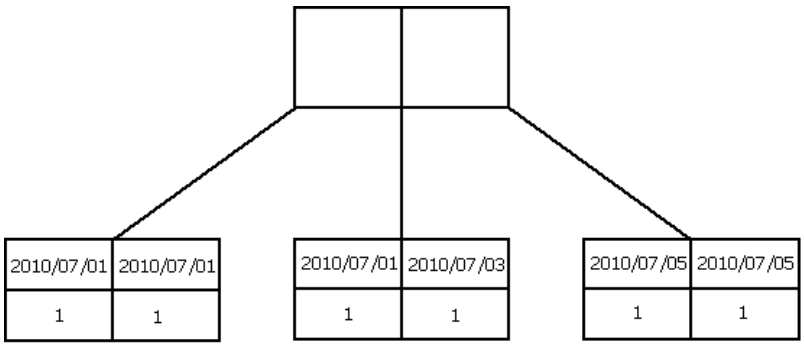

Abbildung 3.1.: View B+-Tree

```
Beispiel: MapReduce                          Map results:
map : function(doc) {                        "2010/07/1", 1
        if(doc.type == 'blogpost') {         "2010/07/1", 1
          emit(doc.date, 1);                 "2010/07/1", 1
```

```
        }                                          "2010/07/3", 1
}                                                  "2010/07/5", 1
                                                   "2010/07/5", 1
reduce: function(keys, values, rereduce) {
          return sum(values);
}
```

3.6. Replikation und Verteilung

CouchDB besitzt einen einfachen aber sehr mächtigen Replikaitonsmechanismus. Mit einer POST-Anfrage an "http://dburl/_replicate" für ein Objekt {"source" : "db1", "target" : "http://targeturl/db2" } wird die Datenbank db1 nach db2 repliziert. Eine ähnliche Anfrage an db2, nur mit vertauschten source und target, bewirkt, dass auch in die andere Richtung repliziert wird. Um eine kontinuierliche Replikation zu erwirken, durch welche Änderungen sofort repliziert werden, kann noch "countinuous":"true" eingefügt werden.

Ähnlich den Revisionsnummern bei Dokumenten, besitzen Datenbanken eine Sequenznummer, die sich immer dann erhöht, wenn ein Dokument geändert wird. Kommt es nun zur Replikation, so wird anhand der Sequenznummer festgestellt, welche Daten verändert wurden und nur diese werden übertragen.

Bei den Replikationen, vor allem wenn sie nicht continuous sind, kann es natürtlich zu Konflikten kommen. Ist dies der Fall, so werden die betreffenden Dokumente mit einem conflict-Flag versehen. Es wird zufällig eine richtige Revision ausgewählt, diese als aktuelles Dokument gespeichert und die andere Version mit einer Revisionsnummer niedriger in die Datenbank übernommen. Da kein Algorithmus feststellen kann, welche Änderung die bessere ist und CouchDB auch nicht mergt, ist dies die einzige Möglichkeit für die automatische Konfliktbehandlung. Da sowohl die nicht übernommene Änderung als auch das conflict-Flag vorliegen, kann der Konflikt jedoch noch nachträglich auf Applikationsebene mit dem dafür benötigem Wissen aufgelöst werden.

Weiterhin ist es leicht möglich, mit CouchDB Clusters einzurichten. Hierfür gibt es CouchDB Lounge, eine proxy-basierte Partitionierungs- und Clusteranwendung[17]. Sie besteht aus zwei Hauptteilen: einem dumbproxy und einem smartproxy. Der dumbproxy ist für alle Anfragen zuständing, die keinen View betreffen und leitet die Anfragen an den betreffenden Knoten weiter. Der smartproxy ist für die Views zuständig. Er leitet Viewanfragen an alle andere Knoten im Cluster weiter. Die Arbeit wird somit verteilt, was der Performance zugute kommt. Die Antworten der Knoten müssen dann nur noch zusammengefügt und an den Absender der Anfrage abgeschickt werden. Um Dokumente einem Speicherknoten zuzuordnen benutzt Lounge consistent hashing. Dazu werden die ersten Zeichen der docID eines Dokumentes genommen, daraus ein Hashwert berechnet und anhand dieses wird dem Dokument dann ein Speicherknoten zugewiesen. CouchDB Lounge unterstützt zur Zeit nur CouchDB in Version 0.10.0.

[17] CouchDB Lounge, http://code.google.com/p/couchdb-lounge/

3.7. Sicherheit

Frisch nach der Installation von CouchDB sind noch keine Sicherheitsmaßnahmen aktiv. Es ist aber so konfiguriert, dass nur Anfragen aus dem loopback network interface (127.0.0.1 oder localhost) entgegengenommen werden. Wird eingestellt, dass CouchDB auch im Netzwerk verfügbar ist, so müssen die Sicherheitsmaßnahmen manuell konfiguriert werden.

Eine erste Methode, um CouchDB vor unberechtigten Zugriffen und Manipulationen zu schützen, ist die Userauthetifikation. Hierfür gibt es zwei Arten von Usern: Administratoren und Reader. Ein Administrator darf Datenbanken erstellen und löschen, Dokumente erstellen, ändern und löschen, compaction auslösen, die task status Liste lesen, den Configfile lesen und ändern sowie den Server neu starten. Sobald ein Admin User existiert, wird bei jeder solcher Aktion eine Authentifikation notwendig. Des weiteren gibt es für Dokumente noch Readerlisten. Besitzt ein Dokument eine Readerliste, so darf es nur von Usern aus dieser Liste betrachtet werden. Es wird weiterhin dynamisch aus Viewresultaten herausgefiltert, sollte der Reader nicht in der Readerliste enthalten sein. CouchDB unterstützt bisher nur Authentifizierungen über http, nicht aber über https. Jedoch kann hierfür ein SSL Proxy benutzt werden.

Weiterhin unterstützt CouchDB auch Cookieauthentifikation. Der User logt sich einmal ein, z.B. mittels eines HTML-Formulars. Werden seine Daten akzeptiert, so wird ein Cookie gesetzt, das für seine Lebensdauer (default ist 10 Minuten) zur Authentifikation bei den folgenden Anfragen genutzt werden kann. Um die Cookieauthetifizierung zu nutzen, muss diese jedoch erst im Configfile aktiviert werden.

Ein letzter Sicherheitsmechanismus, der von CouchDB zur Verfügung gestellt wird, sind die schon bekannten Validierungsfunktionen. Durch sie ist es möglich, eigene Sicherheitsmodelle zu implementieren. Im “userCtx”-Argument von Validierungsfunktionen werden die Daten des Users, der eine Anfrage gestellt hat, gespeichert. Nun kann in den Validierungsfunktionen ein beliebiger Code implementiert werden, der feststellt, ob die Aktion für den User zulässig ist oder nicht. Falls nicht, wird ein Fehler geworfen und die Aktion erfolglos abgebrochen.

3.8. CouchDB Performance

Ungeachtet der CouchDB nutzenden Applikation, ist die Performance von CouchDB von drei Faktoren abhängig: vom der CPU, vom Arbeitsspeicher und von der Festplatte. Die Prozessorleistung wird hauptsächlich gebraucht, um View-Indizes zu berechnen. Sie wird daher um so wichtiger, je mehr Views in einer Datenbank existieren und wenn es viele Schreib- und Lesezugriffe gibt, da die Indizes aufgrund dessen oft geupdatet werden müssen. Der Arbeitsspeicher wird von CouchDB in großem Maße dazu verwendet, Knoten von B-Trees zu cachen. Je mehr Arbeitsspeicher zur Verfügung steht, desto mehr kann gecached werden. Weil dadurch Zugriffe auf die im Vergleich langsame Festplatte entfallen, kommt es zu einem Performancegewinn. Die Geschwindigkeit der Festplatte hat natürtlich auch Einfluss auf die Performance. Besonders gut eignen sich SSD Festplatten, da sie ohne

großen Overhead neue Daten an den Datenbank- oder Viewindexfile anhängen können und gleichzeitig Daten aus diesen lesen können [ALS10].

Um eine Aussage über die Geschwindigkeit von CouchDB zu machen, bedarf es eines Vergleichs mit anderen Datenbanksystemen. Solche Vergleiche sind allerdings schwierig, da sie stark von den Anwendungsfällen abhängen und unterschiedliche Datenbanktechnologien wie dokumentenorientierte Datenbanksystem und relationale Datenbanksysteme nicht unbedingt eins zu eins verglichen werden können.

Zum Vergleich von CouchDB und MySQL existieren nur wenige Benchmarks. Die meisten nutzen zudem ältere Versionen. Jedoch herrscht ein Konsens darüber, dass CouchDB zumindest in Nicht-Cluster-Umgebungen langsamer ist als MySQL. Der Grund daür ist vor allem, dass das Erstellen und updaten von View-Indizes in CouchDB wenig performant ist. Dies hat auch zur Folge, dass die Geschwindigkeit von CouchDB sinkt, je größer die Datenbank wird. Ein Test[18] mit Version 0.10 von CouchDB hat das bestätigt. Dabei existierten in der Datenbank keine Views, sondern der Anstieg ist alleine auf die Updates der ID- und Sequenznummer-Indizes zurückzuführen.

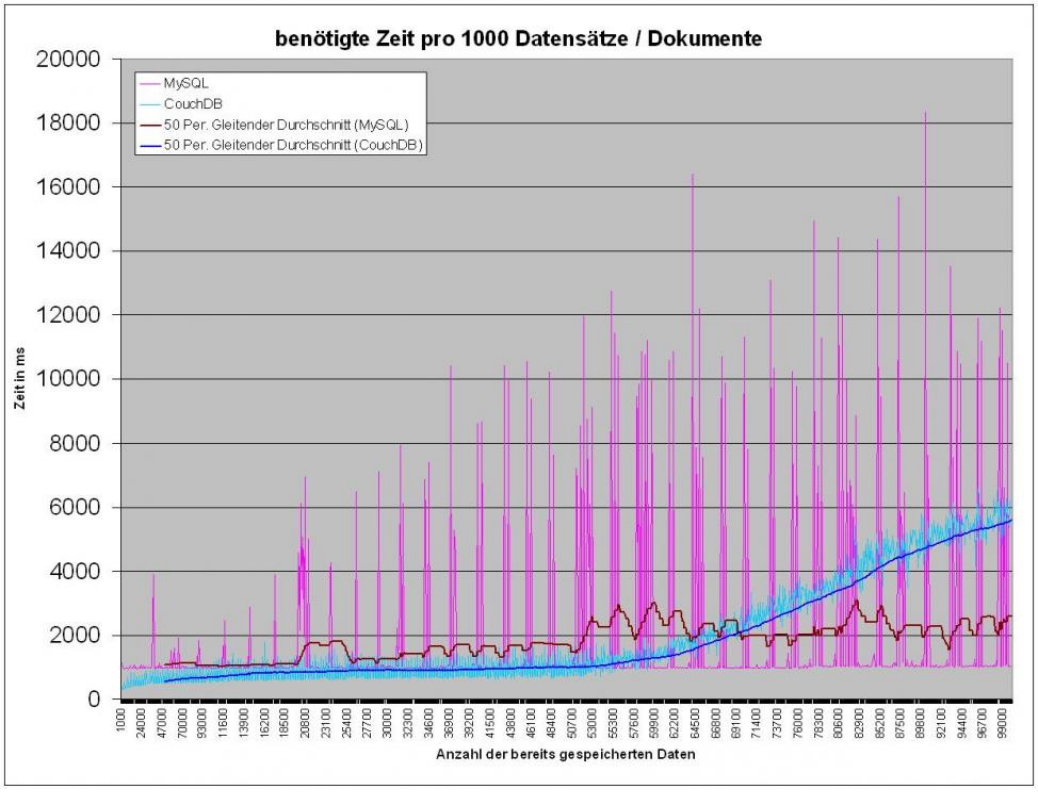

Abbildung 3.2.: CouchDB vs. MySQL, Schreiben von 100000 Datensätze schreiben

Weiterhin ist interessant, wie CouchDB im Vergleich zu anderen dokumentenorientier-

[18]Performance CouchDB vs. MySQL, http://www.java-forum.org/data-tier/105102-performance-couchdb-nosql-vs-mysql.html

ten Datenbanksystemen abschneidet. Der nach CouchDB am weitesten verbreitete Vertreter dieser Klasse ist MongoDB. Zu MongoDB existieren eine Reihe von Benchmarks[19]. Neben CouchDB wurde MongoDB auch mit einigen anderen Datenbanksystemen verglichen. In einem Test, in dem 2.8 Millionen Dokumente mittels bulk_write geschrieben und anschließend wieder geladen wurden, brauchte CouchDB etwa doppelt so lange wie MongoDB. Das Lesen dauerte gut dreimal so lange[20]. In einem weiteren Test wurde MongoDB mit MySQL verglichen[21]. In diesen Benchmarks erzielte MongoDB ebenfalls bessere Ergebnisse als MySQL.

[19] MongoDB Benchmarks, http://www.mongodb.org/display/DOCS/Benchmarks

[20] CouchDB vs. MongoDB, http://bcbio.wordpress.com/2009/05/10/evaluating-key-value-and-document-stores-for-short-read-data/

[21] Benchmarking Tornado Session, http://milancermak.posterous.com/benchmarking-tornados-sessions-0
Benchmarking MongoDB vs. MySQL, http://obvioushints.blogspot.com/2009/07/benchmarking-mongodb-vs-mysql.html

4. CouchTM

Bei der Entwicklung von CouchTM diente tinyTIM als Grundlage. Es wurde die grobe Struktur übernommen, jedoch musste vieles an CouchDB angepasst und neu implementiert werden.

Jedes Topic-Map-Konstrukt implementiert projektinterne Interfaces (IConstruct und das entsprechende Construct-Interface wie z.B. IVariant), welche die entsprechenden TMAPI-Interface (Version 2.0.2)[HS10] erweitern und projektspezifische Ergänzungen enthalten. Intern wird mit den internen Interfaces gearbeitet, die jederzeit auf die entsprechenden TMAPI-Interfaces gecastet werden können und umgekehrt. Jedes Konstrukt wird in der Datenbank durch genau ein Dokument repräsentiert.

Weiterhin wurden eine Reihe von Utilities implementiert, welche z.B. das Mergen übernehmen, die DB ansprechen, sich um die Umwandlung von JSON-Objekten in Konstrukte kümmern u.v.m.

Im Paket Tests befindet sich eine Klasse, welche die in der TMAPI mit enthaltenen Tests startet. In diesem Paket können im Zuge der Weiterentwicklung von CouchTM noch enginespezifische Tests hinzugefügt werden. Insgesamt enhält CouchTM die folgenden Pakete:

Package	Beschreibung
de.topicmapslab.couchtm.core	Implementierungen der verschiedenen Topic-Map-Konstrukte und Engine Bestandteilen
de.topicmapslab.couchtm.internal.api	Interfaces
de.topicmapslab.couchtm.internal.utils	projektinterne Utilities
de.topicmapslab.couchtm.utils	auch extern verwendbare Utilities
de.topicmapslab.couchtm.tests	Tests

Tabelle 4.1.: CouchTM Packages

Es werden zusätzlich folgende externe Pakete benötigt:

Package	Version	Beschreibung
org.tmapi	2.0.3-SNAPSHOT	TMAPI, wird kein tm.clear() benötigt, kann auch mit 2.0.2 compiliert werden
org.apache.httpcomponents.client	4.0.3	Kommunikation mit der Datenbank

org.apache.httpcomponents.core	4.0.1	Konfiguration des Httpclients
org.json	20090211	JSON Klassen
gnu.trove	2.0.4	performante Collections (optional)

Tabelle 4.2.: CouchTM externe Pakete

Um die Trove Collections zu nutzen, muss die Datei trove.jar in Version 2.0.4 einfach im Classpath vorhanden sein. Ist dies dies der Fall, so wird es von der Engine automatisch erkannt und anstelle der normalen CollectionFactory wird die TroveCollectionFactory verwendet.

CouchTM arbeitet nur auf einer CouchDB Version 1.0.1 Datenbank korrekt. Ältere Versionen werden nicht unterstützt, da es dort leichte Unterschiede in der View-API gibt. Ob CouchTM auf zukünftigen Releases von CouchDB funktioniert, kann nicht mit Sicherheit gesagt werden. Für 1.x.x Versionen ist es jedoch wahrscheinlich.

CouchTM wurde unter Eclipse als Maven Projekt entwickelt. Das Projekt wird im Maven-Repository des Topicmapslab und auf Google Code unter http://code.google.com/p/couchtm verfügbar sein.

4.1. Topic-Map-Konstrukte

Die Topic-Map-Konstrukte setzen sich aus zwei Teilen zusammen: einer Construct-Klasse mit Attribute und Funktionen, die alle Konstrukte benötigen und einer zweiten Klasse, die konstruktspezifisch ist und die Construct Klasse erweitert.

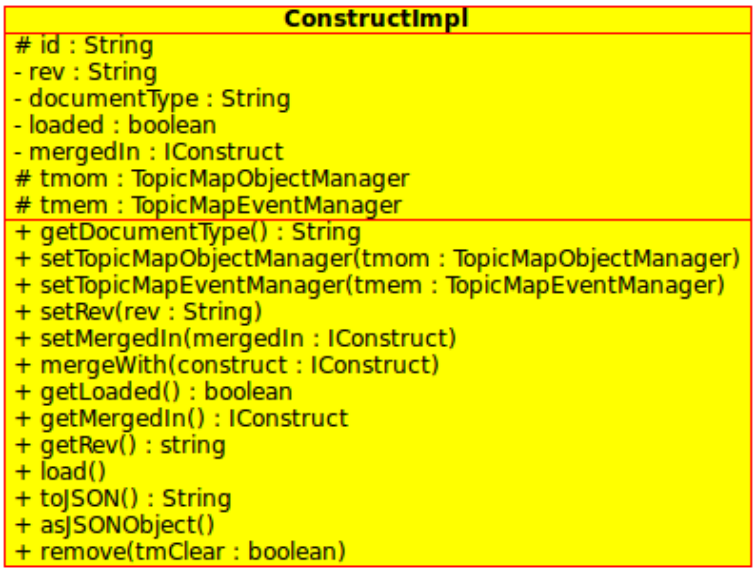

Abbildung 4.1.: Reduzierte UML-Darstellung der Construct-Klasse

In der Construct-Klasse befindet sich die ID eines Konstruktes, welche auch die ID in der Datenbank ist und vom Typ java.util.UUID ist. Die Vergabe der ID erfolgt beim Erstellen eines Konstruktes und kann im Nachhinein nicht mehr verändert werden. Der ID einer Topic Map wird "ctm-" vorangestellt. Sie ist Gleichzeitig der Name der Datenbank, in der sämtlichte Konstrukte dieser Topic Map (auch die Topic Map selber, jedoch ohne Felder für Topics und Associations) gespeichert werden. Weiterhin gibt es noch die Datenbank ctm-topicmaps, welche die Zuordnungen von Topic Map IDs zu Identifiern speichert. Da für Datenbanknamen Einschränkungen gelten, die auf Identifier nicht zutreffen, können diese nicht als Datenbankname verwendet werden.

Das rev-Attribut enthält die Revisionsnummer eines jeden Konstruktes. Die Revisionsnummer wird nach dem Speichern (bzw. Erstellen) eines Konstruktes von der Datenbank zurückgegeben und daraufhin in dem Konstrukt aktualisiert.

Jedes Topic-Map-Konstrukt besitzt ein documentType-Attribut, das angibt, um welche Art von Konstrukt es sich handelt. Dieser Wert wird im Konstruktor gesetzt und ist nicht änderbar. Der Dokumenttyp wird von Views benötigt, um festzustellen, von welchem Konstrukttyp ein jedes Dokument in der Datenbank ist.

In mergedIn wird gespeichert, in welches Konstrukt das aktuelle gemerged wurde. Dies hat keinen Einfluss auf die Repräsentation in der Datenbank. Es dient nur dazu, Funktionsaufrufe an ein Konstrukt, welches bereits gemergt wurde, an das richtige Konstrukt weiter zu leiten.

In den meisten Dokumenten werden in ihren Attributen weitere Konstrukte per ID referenziert. Würde man beim Laden eines Konstruktes aus der Datenbank sämtliche in ihm und wieder diesen referenzierten Konstrukte mit laden, so würde das bedeuten, dass die komplette Datenbank in die Engine geladen wird. Es wird stattdessen für jeden solcher Attributwerte nur ein Konstrukt mit ID und Dokumenttyp erstellt und das loaded Flag auf false gesetzt. Wird nun eine Funktion dieses nur teilweise geladenen Konstruktes aufgerufen und ist loaded == false, so wird das Konstrukt erst vollständig geladen, bevor die Funktion abgearbeitet wird. Falls ein referenziertes Konstrukt bereits geladen wurde, so muss dieser Umweg natürlich nicht gegangen werden.

Um in CouchDB gespeichert werden zu können, muss ein Konstrukt als JSONString vorliegen. Hierzu besitzt jedes Konstrukt eine tostring()-ähnliche Methode namens toJSON(), die einen JSONString aus dem von der Funktion toJSONObject() erzeugten JSONObject-Repräsentation des Konstruktes generiert. Auch die typspezifischen Klassen haben solch eine Methode, welche auf die in der Oberklasse zurückgreift und das dort erzeugte JSONObject vervollständigt.

Für den Datenbankzugriff kennt jedes Konstrukt den TopicMapObjectManager der Topic Map und besitzt für die Eventbehandlung einen TopicMapEventManager. Wird auf einem Konstrukt eine Aktion ausgeführt, die auch Änderungen in anderen Konstrukten nach sich ziehen könnte, so wird die Funktion "_fireEvent" aufgerufen. Sie übergibt das Event sowie die alten und neuen Werte des geänderten Attributes an den TopicMapEventManager, der es dann weiter verarbeitet.

Schließlich wurde noch die remove()-Methode für Konstrukte dahingehend geändert, dass

man ein tmClear-Flag (wird gesetzt bei topicmap.clear()) mit übergeben kann. Ist das Flag gesetzt, so wird das Konstrukt einfach aus der Datenbank löscht, ohne Constraints beachten zu müssen, da dies beim Löschen aller Konstrukte in der Topic Map nur unnötigen Overhead verursacht. Beim Löschen einer kompletten Topic Map wird einfach die dazugehörige Datenbank gelöscht, was am performantesten ist.

Es existieren verschiedene Formate für die Repräsentation von Topic Maps in JSON: JTM[22] und TM/JSON[23]. Allerdings können diese leider weder engineintern noch für die Repräsentation in der Datenbank genutzt werden. In ihnen wird eine Topic Map als einzelnes Dokument aufgefasst, in dem sämtliche Konstrukte an den entsprechenden Stellen eingebettet sind. In CouchTM ist jedoch jedes Konstrukt ein eigenes Dokument. Die Form der zur Speicherung in der Datenbank benutzt JSON-Repräsentationen der Konstrukte orientieren sich jedoch, ausgenommen der Einbettungen, so weit es geht an JTM, wobei allerdings einzelne Bezeichnungen von Attributen abweichen.

4.2. Datenbankzugriff

Der Datenbankzugriff geschieht mit zwei Klassen im inetnal.utils Paket. SysDB beinhaltet Funktionen für GET-, POST-, PUT- und DELETE-Methoden sowie weitere Funktionen zum Erstellen und Verwalten von Topic Maps, die vom TopicMapSystem gebraucht werden. So wird in SysDB z.B. getestet, ob die Datenbank erreichbar ist und sie besitzt ein TopicMapDB Objekt in der gespeichert ist, welche Topic Map ID welchem ItemIdentifier zuzuordnen ist. DB ist die Klasse, die von Topic Maps benutzt wird, um das Speichern und Laden von Topic-Map-Konstrukten sowie den Zugriff auf Views zu erledigen. Sie erbt von SysDB, sodass auch hier die GET-, PUT- und DELETE-Methoden zur Verfügung stehen. Für jede Topic Map bzw. Datenbank gibt es genau eine DB Instanz. Wird eine neue Topic Map erstellt, so ist DB auch dafür verantwortlich, die Views zu erzeugen.

Die Adresse und der Port des CouchDB Servers kann in den Properties der TopicMapSystemFactory unter den Schlüssel “DB” und “PORT” gesetzt werden. Geschieht dies nicht, so wird werden die Defaultwerte “localhost” und “5984” benutzt.

Alle Anfragen, in denen Topic-Map-Konstrukte aus der Datenbank geladen oder geändert werden, passieren zuerst den TopicMapObjectManager, bevor die von DB verarbeitet werden. Dieser besitzt einen Cache, in dem alle Konstrukte, welche bisher erstellt oder geladen wurden, in Maps unter den Schlüsseln ID und IID (bei Topics auch noch SID und SLO) gespeichert sind. Bei Anfragen nach ID oder einem Identifier kann zunächst in diesem Cache nachgeschaut werden, bevor die Datenbank abgefragt werden muss. Views werden nicht gecached und Anfragen an sie müssen daher direkt an die Datenbank weiter gegeben werden. Muss nun für eine Anfrage die Datenbank abgefragt werden, so ruft der TopicMapObjectManager eine entsprechende Funktion in der DB Klasse auf. Für jeden View, für Anfragen nach ID und Identifier sowie für das Speichern und Löschen von Konstrukten besitzt DB

[22] JTM 1.1, http://www.cerny-online.com/jtm/1.1/
[23] TM/JSON

eine Funktion, die daraus eine Anfrage erstellt und abschickt.

Das Resultat dieser Anfragen wird an die Klasse JSONToObject aus dem Paket internal.utils weitergegeben. Diese Klasse besitzt statische Funktionen, um aus dem JSONObject des Resultates die Topic-Map-Konstrukte heraus zu filtern und daraus CouchTM-Objekte zu erstellen. Konstrukte, die in Attributen eines Konstruktes stehen, werden wie oben beschrieben nur teilweise geladen. Allerdings wird auch hier zuerst geschaut, ob sie nicht schon im Cache vorhanden sind; falls dies so ist, werden sie gleich übernommen. Auch für Anfragen, die keine Topic-Map-Konstrukte zurück geben besitzt JSONToObject Methoden, um die Resultate aufzubereiten.

Wurden die Objekte erzeugt, so gibt DB diese an den TopicMapObjectManager zurück, der die neuen Konstrukte in den Cache einträgt bzw. bereits bestehende aktualisiert und schließlich das Resultat der Anfrage an die aufrufende Funktion zurück gibt.

4.3. Views

CouchTM benötigt insgesamt 27 Views. Views kommen genau dann zum Einsatz, wenn es nicht möglich ist, eine Anfrage nach einer ID zu stellen. Dies ist z.B. der Fall, wenn ein Index abgefragt wird, ein neues Konstrukt erstellt wird (Check, ob ein solches schon existiert) oder ein Konstrukt geändert wurde (Check, ob gemergt werden muss).

Viewname	Verwendung
getconstructbylocator	Topic createCheck
gettopicbysubjectidentifier	tm.getTopicBySubjectIdentifier, createName ohne Type, Topic merge
gettopicbysubjectlocator	tm.getTopicBySubjectLocator, Topic createCheck
gettopicsbytm	tm.getTopics
getassociationsbytm	tm.getAssociations
getassociationsbytypescope	Association createCheck
getconstructsbytm	tm.clear
getnamesbytypevaluescope	Name createCheck und mergeCheck
getoccurrencesbytypevaluedatatypescope	Occurrence createCheck und mergeCheck
getassociationsbytheme	scopedIndex.getAssociationThemes
getnamesbytheme	scopedIndex.getNameThemes
getvariantsbytheme	scopedIndex.getVariantThemes
getoccurrencesbytheme	scopedIndex.getOccurrenceThemes
getoccurrencesbytype	typeINstanceINdex.getOccurrenceTypes
getnamesbytype	typeInstanceINdex.getNameTypes

getrolesbytype	typeInstanceIndex.getRoleTypes
getassociaitonsbytype	typeInstanceIndex.getAssociationTypes
gettopicsbytype	typeInstanceIndex.getTopicTypes
getnamesbyvalue	literalIndex.getNames
getvariantsbydatatypevalue	literalIndex.getVariants
getoccurrencesbydatatypevalue	literalIndex.getOccurrences
getrolebytypeplayerparent	Role createCheck und mergeCheck
getvariantbyvaluedatatypescopeparent	Variant createCheck und mergeCheck
getassociationbytypescoperoles	Association mergeCheck
gettopicsbyreified	Topic mergeCheck
gettopicbyitemidentifier	topicMergeCheck
getrolesbyplayer	topic.mergeWith

Tabelle 4.3.: CouchTM Views

Alle Views haben eine ähnliche Struktur. Die meisten Anfragen geben nur einen Typ von Konstrukten zurück. Daher wird zunächst der documentType der Dokumente auf Gleichheit auf den gewünschten Typ getestet. Danach wird der Key bestimmt und mit dem ganzen betrachteten Dokument als Value emittet. Für Keys, die ein Array enthalten, muss dieses Array vor dem Stellen der Anfrage noch sortiert werden. Das Array im Key, den der View emittet, muss in ihm nach der selben Weise sortiert werden. Die Sortierung erfolgt in beiden Fällen nach der lexikographischen Ordnung.

Da nur komplette Dokumente als Values gebraucht werden, ist es nicht sinnvoll, MapReduce zu benutzen.

```
Beispiel: getnamesbytypevaluescope (create Check)
map : function(doc) {
        if(doc.documenttype == 'NAME') {
          if(doc.scope) {
            scope = new Array(doc.scope.length);
            for(var i = 0; i < doc.scope.length; i++) {
              scope[i] = doc.scope[i];
            }
            scope.sort();
          } else {
            scope = new Array('none');
          }
        emit([doc.topicmap, doc.type, doc.value, scope, doc.parent], doc);
}
```

4.4. Event Handling

Jedes Topic-Map-Konstrukt besitzt einen TopicMapEventManager aus dem Package internal.utils. Der TopicMapEventManager kennt die Topic Map, zu der das Konstrukt gehört sowie den TopicMapObjectManager und liest in seinem Konstruktor das automerge-Attribut des TopicMapSystem aus.

Der TopicMapEventmanager empfängt die von dem ihm zugehörigen Konstrukt gefeuerten Events mit samt dem alten sowie dem neuen Wert des geänderten Attributes und verarbeitet diese. Könnte das Event einen Mergevorgang auslösen, so wird durch eine Hilfsklasse das Konstrukt bestimmt, in welches das Event werfende Konstrukt gemergt werden soll und beide werden an die für das Mergen zuständige Klasse übergeben. Das einzige Event, welches kein Merging auslösen kann ist, SET_REIFIER.

4.5. Merging

Wird eine Änderung an einem Attribut eines Konstruktes vorgenommen, das Merging nach sich ziehen könnte, so testet der TopicMapEventManager zunächst, ob für dieses Konstrukt ein weiteres existiert, sodass das im TMDM beschriebene Mergingkriterium für diesen Konstrukttyp erfüllt ist. Hierzu steht die Hilfsklasse MergeCheck im Paket internal.utils zur Verfügung. Sie besitzt für jeden Konstrukttyp statische Methoden, die das zu testende Konstrukt als Argument übergeben bekommen und über den TopicMapObjectManager sowie DB einen MergeCheck-View aufrufen. Diese Views werden mit allen im Mergekriterium angeführten Attributen als Key abgefragt. Ist das Resultat nicht “null”, sonder ein Topic-Map-Konstrukt, so ist das Konstrukt gefunden, in das gemergt werden muss.

Das Mergen selber wird in der Hilfsklasse MergeUtils im Paket internal.utils ausgelöst. Diese Klasse besitzt auch für jeden Konstrukttyp eine merge-Methode, in der die mergeWith-Methode im Event auslösenden Konstrukt aufgerufen wird, die daraufhin das Mergen abwickelt. Danach wird das nun überflüssige Konstrukt aus der Datenbank gelöscht und in dem Konstrukt, welches noch im Speicher ist, das mergedIn Attribut gesetzt. Eventuell werden noch Änderungen an anderen Konstrukten vorgenommen, die durch den Mergevorgang nötig geworden sind.

4.6. Erweiterungsmöglichkeiten

CouchTM besitzt alle essentiellen Funktionalitäten einer Topic-Maps-Engine, doch bieten sich viele Möglichkeiten, diese noch zu erweitern oder zu verfeinern. Das liegt einerseits daran, dass die Bearbeitungszeit der Bachelorarbeit begrenzt ist und sich somit auf das Wesentliche beschränkt werden muss. Andererseits haben sich auch in der Auswertung einige Punkte ergeben, an denen noch Verbesserungen an CouchTM vorgenommen werden könnten. Auch aus der ständige Weiterentwicklung von CouchDB und der Implementation neuer Funktionalitäten ergeben sich neue Ansätze, die in CouchTM integriert werden könnten.

4.6.1. Multi-User-Unterstützung

Zur Zeit wird der Mehrbenutzerbetrieb noch nicht explizit unterstützt. CouchTM kann zwar von mehreren Benutzer gleichzeitig auf der selben Datenbank verwendet werden, doch treten Probleme auf, wenn zwei Nutzer gleichzeitig das selbe Konstrukt ändern wollen. Bei einem Nutzer kommt es dann beim Speichern zu einem Revisionskonflikt. Das Konstrukt müsste neu geladen, die Änderungen erneut ausgeführt und schließlich gespeichert werden. In der aktuellen Version von CouchTM gibt es noch keine Behandlung von Revisionskonflikten. Dies ist auch schwierig zu gestalten, da TMAPI nicht für alle Funktionen, die ein Konstrukt ändern, das Werfen von Exceptions erlaubt, durch die dem Nutzer ein Revisionskonflikt signalisiert werden könnte. Bei den Funktionen, die das Werfen von Exceptions erlauben, ist der Exceptiontyp nicht für diesen Zweck vorgesehen. Hier wäre eine IO-Exception für alle Funktionen in der TMAPI wünschenswert. Diese könnte zwar manuell ergänzt werden, jedoch wäre CouchTM dann nicht mehr vollkommen TMAPI konform. Löst eine Änderung einen Mergevorgang aus, so wird dieses Problem noch komplexer, da alle in diesem Mergevorgang geänderten Dokumente in der aktuellen Revision vorliegen und erfolgreich in die Datenbank geschrieben werden müssen. Sobald ein solches Dokument in einer veralteten Revision vorliegt, müsste der gesamte Vorgang scheitern und die bereits geschriebenen Änderungen rückgängig gemacht werde. Nun kann es aber auch bei der Rückabwicklung wieder zu einem Revisionskonflikt kommen und eine Lösung dieses Problems ist ohne Datenverlust kaum machbar. Bis Version 0.9 von CouchDB gab es noch eine Bulk-Update-Funktion, die fehlschlug, sobald auch nur ein Dokument nicht geschrieben werden konnte. Diese Funktion wurde jedoch entfernt, da dies nicht im Sinne des CouchDB Replikations- und Partitionierungsmodells war. In der aktuellen Version es möglich, mit Bulk-Update auch das Schreiben von Dokumenten mit veralteten Revisionen zu erzwingen. Hierbei werden die beiden jeweils betroffenen Dokumente mit einem conflicted-Flag versehen und dann zufällig eine der beiden Revisionen als aktuelle, die andere Version als vorhergehende gespeichert. Es tritt aber auch hierbei obiges Problem auf, wobei die Rekonstruktion der Änderungen im Nachhinein, bei einer späteren Konfliktbehandlung, noch schwieriger wird. Die einzig naheliegende Lösung für dieses Problem, auch wenn sie gegen die Designprinzipien von CouchDB verstößt, wäre die Einführung eines Lock-Proxys, der für alle Dokumente, die hinsichtlich des Mergevorgangs relevant sind, Locks vergibt und das Schreiben erst dann freigibt, wenn für all diese Dokumente Locks gesetzt werden konnten.

4.6.2. TopicMapObjectManager

Auch der Cache des TopicMapObjectManagers bietet noch einige Verbesserungsmöglichkeiten. Zum Beispiel könnte die Größe des Caches beschränkt werden um den Speicherbedarf zu begrenzen, was bei sehr großen Topic Maps von Vorteil wäre. Hierzu kann eine LinkedHashMap verwendet werden, die es erlaubt, die MAX_ENTRIES festzusetzen und beim Erreichen dieses Wertes das älteste Element entfernt. In dem Element können dann alle Werte, bis auf die ID, auf “null” und das loaded-Flag auf “false” gesetzt werden. Wird nun doch wieder auf das Konstrukt zugegriffen, so wird die load()-Funktion aufgerufen und das

Konstrukt neu aus der Datenbank geladen. Weiterhin könnten auch Viewresultate gecached werden. Zu jedem gecachten View müssen dann noch die Sequenznummer der Datenbank und die Parameter des Aufrufes gespeichert werden. Hat sich die Sequenznummer bei einer erneuten Anfrage an den View nicht geändert und sind die Parameter die selben oder ist ersichtlich, dass das Resultat der Anfrage eine Teilmenge des Resultates einer frühreren Anfrage ist[24], so kann auf den Cache zurück gegriffen werden. Die Interaktion mit der Datenbank beschränkt sich in diesem Falle nur auf das Abfragen der Sequenznummer. Dies ist jedoch nur bei Topic Maps sinnvoll, in denen es kaum oder im besten Fall gar keine Schreibzugriffe gibt.

4.6.3. MaJorToM-Interfaces

MaJorToM ist ein vergleichsweise neues Projekt, welches zu Beginn der Arbeit noch nicht zur Verfügung stand. Anstelle der TMAPI-Interfaces könnten in CouchTM die MaJorToM-Interfaces implementiert werden. Da diese eine Erweiterung der TMAPI sind, muss die bisherige Implementation nicht verändert werden. Nur die Funktionalitäten, um die MaJorToM die TMAPI-Interfaces erweitert (z.B. die supertype-subtype Beziehungen), müssen noch implementiert werden.

4.6.4. Neues Eventkonzept

Bisher veranlassen Topic-Map-Konstrukte bei Änderungen, die kein Event auslösen, selbst das Schreiben in die Datenbank. Bei vielen dieser Aktionen muss jedoch nicht nur das eine Konstrukt gespeichert werden, sondern auch andere, auf die sich diese Änderungen auch ausgewirkt haben (Z.B. muss bei topic.createOccurrence() nicht nur die Occurrence gespeichert werden, sonder auch das Topic, da sich sein Occurrences-Attribut geändert hat). Bei Aktionen wie dem Löschen eines Konstruktes, welches das Parent eines ihm untergeordneten Konstruktes ist, kann es sogar vorkommen, dass es vor dem Löschen bei bisheriger Implementation noch einmal gespeichert werden muss. Also veranlasst eine Änderungen X Schreiboperationen, die einzeln ausgeführt werden.

CouchDB unterstützt Bulk-Write-Operationen, in denen mehrere Schreiboperationen zusammengefasst werden könnten. Allerdings existiert dabei das schon erwähnte Problem mit den Revisionskonflikten, das aber mit einem Lock-Proxy gelöst werden kann und somit auch den Einsatz von Bulk-Writes ermöglichen würde. Jedoch muss hierzu von einer zentralen Stelle erfasst werden, welche Schreiboperationen zu einer Aktion gehören.

Um auch das angesprochene unnötige Schreiben von Zwischenergebnissen einer Aktion zu verhindern, muss das Eventkonzept grundlegend verändert werden. Konstrukte dürfen ihr Schreiben nicht mehr selber veranlassen und auch keine anderen Konstrukte ändern. Jede Aktion, nicht nur die, welche Merging nach sich ziehen könnten, löst ein Event aus, das

[24] Beispiel: Der Key [value, []] wurde schon gecached Das Resultat von [value, [bla, blub]] ist für gleiche Werte von Value schon enthalten, da es eine Art von Spezialisierung der ersten Anfrage ist.

vom TopicMapEventManager verarbeitet wird. Es werden alle nötigen Schreiboperationen festgestellt, in einem Job zusammengefasst und an den Lock-Proxy geschickt. Der Proxy bearbeitet den Job, speichert die Konstrukte und gibt das Resultat an den TopicMapEventManager zurück. Der TopicMapEventManager übernimmt schließlich die Änderungen in die Konstrukte und aktualisiert die Revisionsnummern.

Es werden folglich die Speicherung von Konstrukten, die Programmlogik und das Anbieten von TMAPI-Funktionalität an den Nutzer exakt getrennt. Damit wäre es auch möglich, die Eventbehandlung auf dem Proxy laufen zu lassen und so den Kern der Engine auf den Server zu verlagern. Der Vorteil wäre, dass die Kommunikation über ein Netzwerk auf ein Minimum reduziert wird, da nur ein Job übertragen wird und nicht eine Reihe von GET- und PUT-Anfragen. Dies wäre vor allem von Vorteil, wenn sich die Datenbank nicht in einem lokalen Netzwerk, sondern im Internet befindet, wo die Latenz eine zunehmende Rolle spielt. Andererseits hat es einen höheren Leistungsbedarf auf dem Server zur Folge.

5. CouchTM Tests

Um CouchTM zu testen, standen drei Testsuites zur Verfügung: die TMAPI-Tests, die RTM- bzw. JRTM-Tests und die CXTM-Tests[25]. Mit ihnen wird die Korrektheit und Vollständigkeit der Engine getestet. Sie sind aber auch auf viele andere Engines anwendbar und eignen sich daher ebenfalls, um die Performance von CouchTM mit denen anderer Engines zu vergleichen. Das Konzept hinter der Entwicklung von CouchTM war Test-Driven-Development und die TMAPI-Tests waren die dem zu Grunde liegenden Tests. Die Zielsetzung bestand also darin, dass sämtliche TMAPI-Tests erfolgreich absolviert werden mussten. Selbiges galt aber auch für die RTM-Tests, durch welche nachträglich noch viele Fehler behoben werden konnten, die nicht durch die TMAPI-Tests abgedeckt wurden. Abschließend wurden noch die CXTM-Tests durchgeführt, um einen Einblick in das Import und Export-Verhalten von CouchTM zu erhalten und den Entwicklern dieser relativ neuen Tests Feedback zu geben.

5.1. TMAPI-Tests

Die TMAPI-Tests sind JUnit Tests, welche im org.tmapi.tests Paket enthalten sind. Sie sind die rudimentärsten Tests, die eine Implementation bestehen muss. CouchTM enthält im Paket "de.topicmapslab.couchtm.tests" die Klasse AllTMAPITests, durch welche die Tests aus dem Projekt heraus gestartet werden können und dadurch bei jedem Build mit Maven durchlaufen werden.

In der aktuellen Version der TMAPI-Tests (2.0.2) schlägt der Test testRoleAssociationFilter im Testcase TestTopic fehlt. CouchTM unterstützt automerging und dieses Feature ist auch standardmäßig aktiviert. Je nach gesetztem Feature gibt es eigene Tests dafür. In diesem Test wird jedoch ein Szenario modelliert, in dem automatisch gemergt wird, obwohl angenommen wird, dass dieses Feature deaktiviert ist. Es werden zwei Roles mit gleichen Playern und Parents, aber unterschiedlichen Typen erzeugt. Danach wird der Typ der zweiten Role auf den der ersten gesetzt. Da jetzt Typ, Player und Parent gleich sind, wird wie im TMDM beschrieben gemergt. Im Folgenden erwartet ein Assert in diesem Test, dass die parent Association zwei Roles besitzt. Da jedoch gemergt wurde, besitzt sie nur noch eine Role und der Test schlägt fehl. Da Maven für einen erfolgreichen Build fordert, dass alle Tests fehlerfrei durchlaufen werden müssen, kann CouchTM bislang nur gebuilded werden, wenn die Tests auskommentiert werden. Das Problem mit diesem Test wurde an die Entwickler von TMAPI weitergeleitet. Es ist davon auszugehen, dass er in einer der folgenden

[25] CXTM Tests, http://cxtm-tests.sourceforge.net/

Versionen behoben wird.

Alle weiteren TMAPI-Tests werden von CouchTM erfolgreich durchlaufen.

5.2. RTM-Tests

Als zweites Testframework dienten die RTM-Tests. Sie wurden in Ruby beschrieben, doch kann JRTM als Wrapper eingesetzt werden, sodass sie auch auf CouchTM anwendbar sind.

Bei den RTM-Tests mussten kleine Änderungen vorgenommen werden, da RTM standadmäßig von einer nicht persistenten Engine ausgeht. In der Datei "rtm/spec/spec_helper.rb", welche den Tests die genutzte Topic Map zur Verfügung stellt, wurde daher "tm.clear()" vor der Rückgabe der Topic Map eingefügt, um eine nicht persistente Engine zu simulieren. Die Tests in "rtm/spec/rtm/navigation/topic/supertypes_spec.rb" greifen nicht auf den spec_helper zurück. Daher musste auch dort für die beiden erstellten Topic Maps je ein "tm.clear()" eingefügt werden. tm.clear() ist in RTM so implementiert, dass zuerst die Variants einer Topic Map gelöscht werden, dann die Names, Occurrences, Roles, Associations, Topics die Typen sind und schließlich die restlichen Topics. Dies hat leider zur Folge, dass der Aufruf auf CouchTM sehr imperformant ist. Nach jedem Löschen muss das Parent gespeichert werden, da sich eines seiner Attribute geändert hat und gegebenen Falls muss getestet werden, ob für das Parent dann das Mergekriterium erfüllt ist. Daher wurde die 2.0.3-SNAPSHOT Version der TMAPI verwendet, welche die Funktion tm.clear() enthält, die wie bereits beschrieben performanter implementiert wurde. JRTM benutzt nun diese Methode und nicht die aus RTM. Weiterhin wurde in "rtm/spec/rtm/engine_spec.rb" noch ein Eintrag für CouchTM hinzugefügt, weil dieser Test für dort nicht aufgeführte Engines failt.

Im Testfile "rtm/spec/rtm/io/tmapix_from_spec.rb" schlagen zwei Tests fehl, jedoch liegt dies nicht an CouchTM. In den Tests wird jeweils eine Topic Map in den Formaten XTM 1.0 bzw. LTM durch TMAPIX geladen. In beiden Fällen kommt es im Parser von TMAPIX zu einer Parserexception, noch bevor etwas an CouchTM übergeben wird.

Alle weiteren RTM-Tests werden von CouchTM erfolgreich durchlaufen. Da die nicht bestandenen Tests nicht auf einen Fehler in CouchTM zurück zu führen sind, kann man sogar sagen, dass CouchTM alle RTM-Tests erfolgreich absolviert.

5.3. CXTM-Tests

Schließlich wurde CouchTM noch mit den CXTM-Tests getestet. In dieser Testsuite wird geprüft, ob eine Engine Topic Map input Files in verschiedenen Syntaxen richtig verarbeitet. Obwohl CouchTM selber nicht das Importieren von Topic Maps unterstützt, können die Tests trotzdem mit Hilfe von TMAPIX durchgeführt werden. Um Fehler zu erkennen, die nur durch diesen Zwischenschritt entstanden sind, wurden die Ergebnisse mit denen von tinyTiM verglichen. Die CXTM-Tests gibt es leider nicht als separate JUnit Tests, doch sie

sind mit in den Tests für TMAPIX enthalten. Es musste für CouchTM nur ein neuer Task couchtmTests in der build.gradle hinzugefügt werden.

Für jeden unterstützten Topic Map Syntax beinhalten die CXTM-Tests Cannonicalization-Tests. In diesen Tests wird ein Inputfile geladen, als CXTM-File serialisiert und schließlich mit einem Baselinefile verglichen wird. Damit solch ein Test erfolgreich ist, muss der generierte File mit dem baseline File Character für Character übereinstimmen. Für manche Syntaxen gibt es auch noch Invalitity-Tests. Dies sind Inputfiles, welche gemäß des Syntaxes Fehler enthalten und somit von der Engine abgelehnt werden müssen, damit der Test erfolgreich ist. Die in den CXTM-Tests getesteten Syntaxen sind XTM 1.0, XTM 2.0, XTM 2.1, CTM, LTM 1.3, TM/XML, JTM 1.0 und verschiedene RDF Formate.

Um die CXTM-Tests auf CouchDB laufen zu lassen, bedarf es auch hier kleiner Änderungen. Es wird wieder davon ausgegangen, dass die Engines kein persistentes Backend haben. Daher müssen in der setUp-Methode der Klasse AbstractValidCXTMReaderTestCase, AbstractinvalidCXTMReaderTestCase und AbstractCXTMWriterTestCase die Topic Maps im Topic-Map-System gelöscht werden. Da die von CouchTM zur Kommunikation mit der Datenbank benötigten httpcomponents das commons-logging Paket benutzen, die CXTM Tests aber auf SLF4J ausgelegt sind, muss in der build Datei unter testRuntime noch das Paket jcl-over-slf4j geladen werden, um Kompatibilität herzustellen.

Als Referenzengine wurde tinyTiM verwendet, bei welcher von den 3151 Tests 21 fehl schlugen. Bei CouchTM schlugen 152 Tests fehl. Da sich die CXTM-Tests selbst noch in der Entwicklungsphase befinden, sind die Ergebnisse dieser nicht so aussagekräftig, wie bei den anderen Testframeworks. Durch das Fehlschlagen von zwei RTM-Tests, in denen auch Topic Maps importiert wurden, obwohl CouchTM nicht beteiligt war, ist es auch nicht auszuschließen, dass selbiges Problem erneut auftrat.

6. CouchTM Performance

Neben der Korrektheit von CouchTM ist natürlich die Geschwindigkeit von großem Interesse. Doch anders als die Korrektheit, ist die Geschwindigkeit nicht so gut zu erfassen. Sie hängt von verschiedenen Faktoren ab wie Festplatten- und Prozessorleistung, den Anfwendungsfällen und bedarf eines Vergleichs, um aussagekräftig zu sein. Daher wurden verschiedene Tests unterschiedlicher Komplexität mit unterschiedlichen Rechnerkonfigurationen durchgeführt und die Ergebnisse mit denen der in Kapitel 1 genannten Engines verglichen. Weiterhin werden die einfachen Schreib- und Leseoperationen im Detail untersucht, um Ansatzpunkte für eine weitere Optimierung aufzuzeigen. Die Resultate der Tests und eine Auswertung werden in diesem Kapitel präsentiert.

6.1. Performance Tests

Die Performance Tests wurden auf zwei Systemen mit den folgenden Konfigurationen durchgeführt:

	System 1
Prozessor	AMD Athlon 64 X2 4200+
RAM	3027MB @ 667 MHz
HDD 1	ATA Corsair CMFSSD-64GBG2D (0.2ms, SSD)
HDD 2	ATA Maxtor 6V160E0 (9.3ms, 7200 U/Min)
Betriebssystem	Ubuntu 10.04 LTS, Kernel 2.6.32-24-generic
CouchDB Version	1.0.1
	System 2
Prozessor	Intel Pentium M 2 GHz
RAM	2048 MB @ 319 MHz
HDD 1	ATA HTS721080G9AT00 (11ms, 7200 U/Min)
Betriebssystem	Ubuntu 10.04 LTS, Kernel 2.6.32-24-generic
CouchDB Version	1.0.1

Tabelle 6.1.: Testsysteme

Sofern nicht anders angegeben, befinden sich die Datenbankfiles auf HDD1. Alle Tests

wurden auf einem frisch gestartetem System ausgeführt. Zum starten der Tests der Testframeworks wurde ein Python-Script verwendet, sodass der Overhead minimal war. Alle anderen Tests wurden aus Eclipse heraus gestartet und der Overhead war somit etwas größer.

Als Vergleichsengines wurden benutzt: tinyTIM 2.0 mit in-memory-Backend, MaJorToM 1.1.1 mit PostgreSQL-Backend und Ontopia 5.1 mit PostgreSQL-Backend. PostgreSQL kam in Version 8.4 zum Einsatz.

Jeder Test wurde fünf mal durchgeführt, um zufällige Einflüsse zu minimieren. Für die Auswertung wurden nur die Mittelwerte aus den fünf Durchläufen betrachtet. Sämmtliche gemessenen Werte sowie der Mittelwert und die Standardabweichung jeder Messreihe sind im Anhang zu finden.

6.1.1. Schreibgeschwindigkeit einfacher Operationen

Im ersten Test werden 10000 Topics in eine leere Topic Map geschrieben. Er dient dazu, die Lesegeschwindigkeit separat zu messen, da dies in komplexeren Tests nicht mehr möglich ist. Die im Test zum Einsatz kommende Topic Map und die benötigten Locators werden vor jedem Test erstellt. Die dafür benötigte Zeit fließt nicht mit in das Messergebnis ein. Die gemessene Zeit bezieht sich nur auf den mit angegebenen Codeblock.

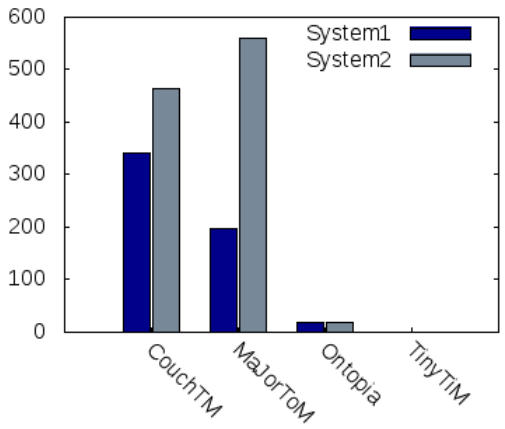

Abbildung 6.1.: Schreiben von 10000 Topics

Codeblock der Messung:

```
for(Locator indicator : indicators) {
    topic = tm.createTopicBySubjectIdentifier(indicator);
    ids.add(topic.getId());
}
```

Bei den für MaJorToM gemessenen Werten ist auffällig, dass sie auf System1 linear ansteigen. Ist k der gemessene Wert aus dem ersten Durchlauf, so ist der Wert für den n-ten Durchlauf in etwa n*k. Auf System2 ist auch ein Anstieg zu beobachten, doch ist er

dort nicht linear. Nach jedem kompletten Durchlauf, das heißt nach sowohl Lesen als auch Schreiben, wurde die Topic Map durch “tm.remove()” gelöscht. Trotzdem scheinen vorhergegangene Durchläufe die im Weiteren folgenden zu beeinflussen.

CouchTM schafft es in diesem Szenario 29,2 (System1) bzw. 21,6 (System2) Topics pro Sekunde zu erstellen und in die Datenbank zu schreiben. Die anderen Engines erzielen hierbei bessere Werte: MaJorToM schafft 50,6 bzw. 17,9 Topics pro Sekunde, Ontopia schafft 523 bzw. 508 Topics pro Sekunde und tinyTiM sogar über 100000 Topics pro Sekunde auf beiden Systemen. Die von tinyTiM erzielten Werte können jedoch nicht als Maßstab genommen werden, da es nur ein in-memory-Backend besitzt. Sie dienen nur zur Veranschaulichung der Unterschiede von Engines mit in-memory und persistentem Backend. In der Schreibleistung kann sich CouchTM am ehesten mit MaJorToM messen. Auf dem schnellen System ist zwar MaJorToM performanter, auf dem langsamen System ist es allerdings anders herum. Weiterhin sind die für CouchTM gemessenen Werte nahezu konstant, wärend die Werte bei MaJorToM ansteigen, je mehr Daten zuvor geschrieben wurden. Ontopia ist von den Engines mit persistentem Backend bei weitem die performanteste und um ein vielfaches schneller als CouchTM und MaJorToM.

Da CouchTM automatisch mergt, wird vor dem Schreiben eines neuen Topics erst einmal geschaut, ob ein Topic mit diesem Locator nicht schon in der Datenbank existiert. Um nur die Geschwindigkeit des eigentlichen Schreibens eines Konstruktes zu ermitteln, muss von den gemessenen Schreibwerten noch das Lesen abgezogen werden. Somit kommt CouchTM auf 34.3 Topics pro Sekunde auf System1 und 25 Topics pro Sekunde auf System2. Die für View-Updates benötigte Zeit ist in diesen Werten jedoch immer noch enthalten.

Weiterhin stellt sich noch die Frage, welchen Anteil die Kommunikation mit der Datenbank und der engineinterne Teil haben. Hierzu werden der Schreibtest ein weiteres Mal ohne Beteiligung von CouchTM durchgeführt. Für die Kommunikation mit der Datenbank kommen die selben Methoden wie bei CouchTM zum Einsatz. Die zu speichernden Topics liegen aber schon als JSON-Strings vor, in denen nur der Locator angepasst werden muss. Ein Check, ob ein Topic mit dem selben Locator bereit existiert, findet nicht statt. Die Datenbank ist die selbe wie bei CouchTM, beinhaltet also auch sämtliche Views. Welchen Einfluss die Anzahl der Views hat, wird im folgenden noch separat diskutiert.

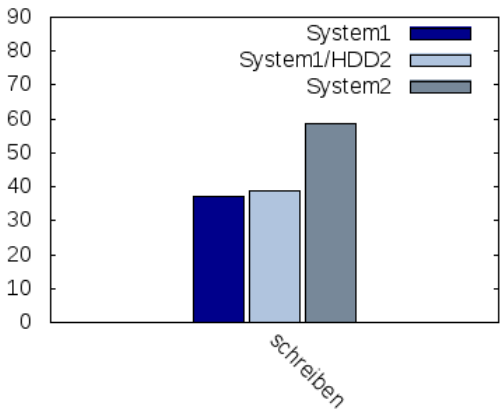

Abbildung 6.2.: Schreiben vom 10000 Topics ohne Engine

Auf System1 benötigt der Schreibtest nur etwa 10% und auf System2 etwa 13% der Zeit, die von CouchTM erzielt wird. Der Anteil der Engine und der View-Updates an einer Schreiboperation ist also enorm.

Bulk-Writes werden von CouchTM momentan noch nicht unterstützt, sie könnten jedoch in Folge einer Weiterentwicklung interessant werden. Um den Geschwindigkeitsunterschied zwischen single und Bulk-Writes zu untersuchen, werden die 10000 Topics noch mit einer modifizierten CouchTM Version in einem einzigen bulk write geschrieben. Bis auf zwei Unterschiede werden die Topics genau so erstellt, wie von createTopicBySubjectIdentifier. Das Speichern erfolgt nicht nach dem Erstellen eines jeden Topics, sondern nach dem Erstellen aller Topics. Die Revisionsnummern werden nach dem Speichern nicht aktualisiert. Außerhalb dieses Szenarios funktioniert diese Version jedoch nicht.

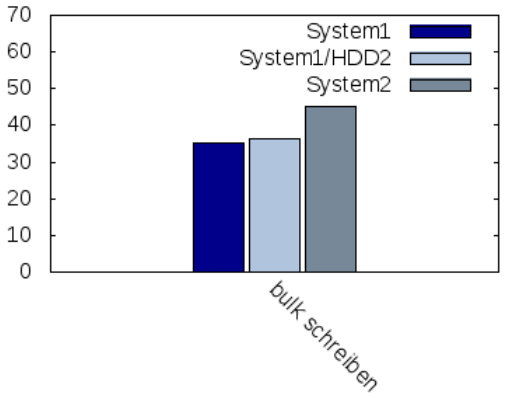

Abbildung 6.3.: Bulk-Write von 10000 Topics

Auf allen Systemen verringert sich die benötigte Zeit erheblich. Durch den Einsatz von Bulk-Writes wurde die Zeit zum Schreiben von 10000 Topics auf etwa 10% der der Single-Writes reduziert. Die Werte sind sogar noch besser, als das einzelne Schreiben ohne Engine. Auch die für View-Updates benötigte Zeit reduziert sich durch Bulk-Writes sehr stark. Auf realen Daten kann allerdings selten (außer beim Importieren von Topic Maps) solch eine große Anzahl an Konstrukten mit einem Mal geschrieben werden. Dennoch verspricht der Einsatz von Bulk-Writes einen Performancegewinn.

6.1.2. Lesegeschwindigkeit einfacher Operationen

In den Lesetests werden die durch den Schreibtest geschriebenen 10000 Topics wieder geladen. Um mögliche das Ergebnis beeinflussende Chaches zu umgehen, wird nach jedem Lesen und dem Schreiben der Daten “tm.close()” aufgerufen. Danach wird die Topic Map vom TopicMapSystem neu angefordert. In tinyTiM ist dies jedoch aufgrund der fehlenden Persistenz nicht möglich und auch nicht sinnvoll. Auch bei MaJorToM kommt “tm.close()” nicht zum Einsatz, da der Befehl dort das Löschen der Topic Map nach sich zieht.

Bei der Lesegeschwindigkeit werden 2 Fälle unterschieden: das Lesen nach ID und das

Lesen nach Locator. Diese Unterscheidung wird gemacht, um den Einfluss von Views auf die Lesegeschwindigkeit aufzuzeigen. Für die Abfrage nach ID kann die CouchDB Document-API benutzt werden, es wird kein View aufgerufen. Die Abfrage nach Locator funktioniert hingegen nur unter Benutzung eines Views. Auch bei den Engines mit RDBMS-Backend sind Unterschiede zu erwarten. Für die Abfrage nach ID wird der Primärschlüssel, also ein Index, benutzt. Bei der Abfrage nach Locator muss solch ein Index nicht zwingend existieren.

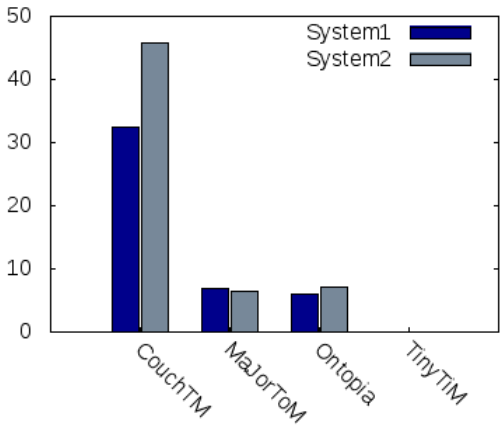

Abbildung 6.4.: Lesen nach ID von 10000 Topics

Codeblock der Messung:

```
for(String id : ids) {
    tm.getConstructById(id);
}
```

Die Leseleistung von CouchTM nach ID liegt bei 308 Topics pro Sekunde auf System 1 bzw. 219 Topics pro Sekunde auf System2. MaJorToM und Ontopia liegen bei 1500 Topics pro Sekunde auf beiden Systemen, da für diese Art von Anfrage ein Index in der Datenbank existiert. tinyTiM ist auch hier mit 2,5 Millionen Topics pro Sekunde allen Engines mit RDBMS-Backend überlegen. Die Messwerte von MaJorToM steigen bei diesem Test, anders als beim Schreiben, nicht an, je mehr Daten sich in der Datenbank befinden.

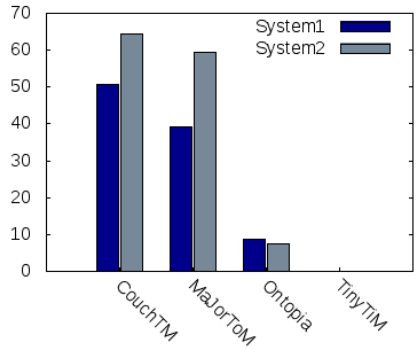

Abbildung 6.5.: Lesen nach Locator von 10000 Topics

Codeblock der Messung:

```
for(Locator indicator : indicators) {
    tm.getTopicBySubjectIdentifier(indicator);
}
```

Bei der Leseleistung nach Locator kann CouchTM fast mit MaJorToM gleichziehen. CouchTM liest hier 197 bzw. 155 und MaJorToM bei 256 bzw. 168 Topics pro Sekunde. MaJorToM besitzt für Locators keinen Index in der Datenbank, was den Unterschied zum Lesen nach ID erklärt. Ontopia besitzt auch für Locators einen Index und schafft dadurch 1366 bzw. 1142 Topics pro Sekunde; tinyTiM erzielt mit 2,5 Millionen Topics pro Sekunde wieder das beste Ergebnis. Auch hier steigen die Messwerte von MaJorToM nicht an.

Bei den Lesetests wurde ebenfalls untersucht, welcher Anteil auf die Engine und welcher auf die Kommunikation mit der Datenbank entfällt. Die in den Tests gemessene Zeit umfasst nur das Senden der Anfrage an die Datenbank und das Empfangen der Antwort, die ohne CouchTM aber nicht weiter verarbeitet wird.

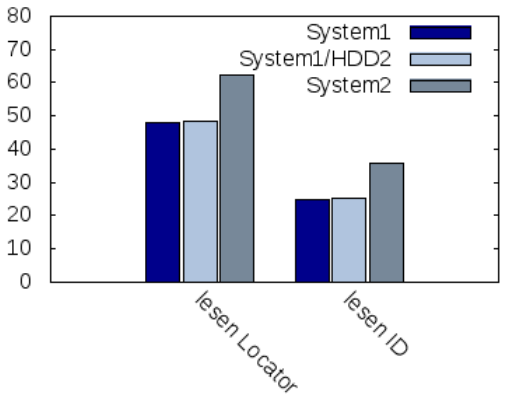

Abbildung 6.6.: Lesen von 10000 Topics ohne Engine

Der Geschwindigkeitszuwachs beim Lesen ist weitaus geringer als beim Schreiben. Auf System1 wird etwa 80% und auf System2 etwa 75% der Zeit, die mit CouchTM benötigt wird, gebraucht. Im Gegensatz zum Schreiben ist beim Lesen der Spielraum für mögliche Optimierungen weitaus geringer.

6.1.3. Performance der Testframeworks

Die Laufzeiten der Testframeworks, die zum Testen der Korrektheit benutzt wurden, bieten auch eine gute Möglichkeit, um die Geschwindigkeit von Topic-Maps-Engines zu vergleichen. Anders als bei den Schreib- und Lesetests sind die auf der Topic Map durchgeführten Aktionen komplexer und die Messung fließt das Erstellen der Infrastruktur mit ein. Die TMAPI-Tests bestehen aus vielen eher kurzen und einfachen Tests. Vor bzw. nach jedem Test werden die setUp- bzw. tearDown-Methode aufgerufen. In der setUp-Methode

werden die TopicMapSystemFactor, das TopicMapSystem und die Topic Map erstellt; in der tearDown-Methode wird die Topic Map (und bei CouchTM damit auch die Datenbank) gelöscht.

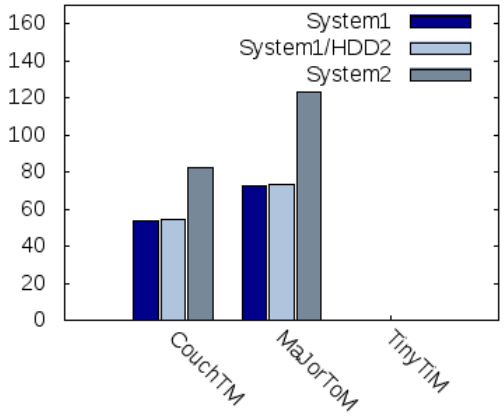

Abbildung 6.7.: Geschwindigkeit der TMAPI-Tests

Auf Ontopia DB konnten die TMAPI-Tests nicht durchgeführt werden. Aus unbekannten Gründen terminierten die Tests nach 10 Stunden immer noch nicht. Sie werden von tinyTiM fehlerfrei durchlaufen, bei CouchTM tritt der eine schon angesprochene Fehler auf und bei MaJorToM gibt es 6 Fehler in den Tests. CouchTM schneidet hier besser ab als MaJorToM und benötigt für die TMAPI-Tests auf System 1 nur 74% der Zeit, die MaJorToM braucht. Auf System2 sind es sogar nur 66% der Zeit. Bei beiden Engines hat die Festplatte in System1 nur sehr geringen Einfluss. Durch die geringere Prozessorleistung auf System2 erhöht sich die benötigte Zeit jedoch deutlich: bei CouchTM um 50% und bei MaJorToM sogar über 65%.

Ähnlich den TMAPI-Tests fließt auch bei den RTM-Tests das Erstellen der Infrastruktur mit in die Messung ein. Sie beinhalten auch viele eher kleine und einfache Testcases. Allerdings wird auch zusätzlich noch in den Tests “rtm/spec/rtm/io/tmapix_to_spec.rb”, “rtm/spec/rtm/io/ tmapix_from_spec.rb” und “rtm/spec/rtm/io/to_rdf_spec.rb” das Importieren und Exportieren von Topic Maps in verschiedenen Formaten getestet. Der Anteil dieser drei Tests an der gesamten Zeit beläuft sich auf System 1 auf ca. 51% für CouchTM und 37% für MaJorToM und auf System2 auf 55% bzw. 40%. Weiterhin wird nicht nach jedem Test tm.remove() sondern tm.close() aufgerufen, was je nach Engine unterschiedliche Folgen hat: MaJorToM löscht mit diesem Befehl die von dem Test geschriebenen Daten, CouchTM hingegen beendet nur die Verbindung mit der Datenbank. Für MaJorToM wurden, wie für CouchTM, nur die Standardtests aus dem Ordner “rtm/sepc” verwendet und die enginespezifischen Tests weggelassen.

Die RTM-Tests konnten wiederum auf Ontopia DB nicht durchgeführt werden, da es Probleme mit der Konfiguration gab. Anders als bei den TMAPI-Tests laufen die RTM-Tests auf MaJorToM weitaus schneller als auf CouchTM. Auf System 1 braucht CouchTM mehr als 400% der Zeit, die MaJorToM benötigt. Aus System2 sind es nur knapp 140%. Bei

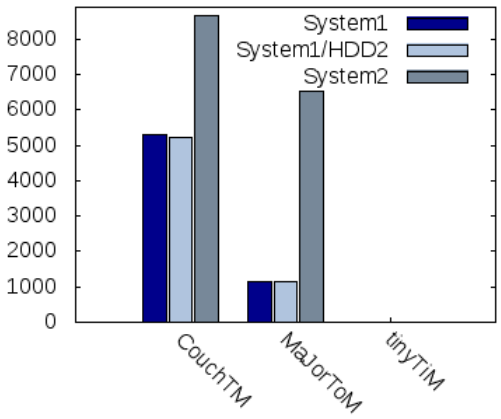

Abbildung 6.8.: Geschwindigkeit der RTM-Tests

den RTM-Tests ist auffällig, dass der Geschwindigkeitszuwachs von System2 zu System1 bei MajorToM enorm ist. Auf System2 brauchen die Tests mehr als 500% der Zeit von System1. Auch bei CouchTM ist ein Unterschied festzustellen, doch fällt dieser mit 160% weitaus geringer aus.

Schließlich wird noch die Geschwindigkeit der CXTM-Tests gemessen. Im Unterschied zu den Testframeworks, bestehen sie ausschließlich aus dem Importieren und Exportieren von Topic Maps mit TMAPIX. Daher fließt neben dem Erstellen der Infrastruktur auch noch die zur Verarbeitung mit TMAPIX benötigte Zeit in die Messwerte ein. Die Topic Map bei diesen Tests sind sehr klein und bestehen meist nur aus wenigen Konstrukten.

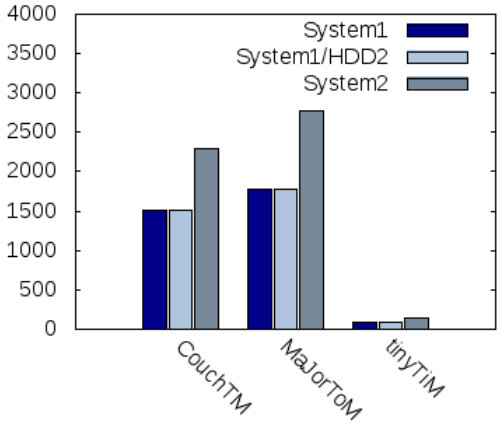

Abbildung 6.9.: Geschwindigkeit der CXTM-Tests

Die CXTM-Tests konnten mit Ontopia nicht durchgeführt werden, da sie, ähnlich den TMAPI-Tests, nicht in absehbarer Zeit terminierten. Die Anzahl der fehlgeschlagenen Tests variieren bei MajorToM zwischen 305 und 384, bei den anderen Engines ist sie hingegen konstant. Bei den CXTM-Tests ist CouchTM performanter als MaJorToM. Auf System 1 wird nur 85% der Zeit von MaJorToM benötigt, auf System2 sind es 82%. Wie schon bei den vorangegangenen Tests, hat auch hier die Prozessorleistung einen deutlichen Einfluss auf die

Geschwindigkeit. Wegen der geringen Komplexität der zu Grunde liegenden Topic Maps, ist die benötigte Zeit bei CouchTM weitaus geringer, als der Import - und Export-Anteil der RTM-Tests, obwohl dort bedeutend weniger Topic Maps importiert wurden. MaJorToM benötigt hingegen mehr Zeit. Selbiges gilt auch für tinyTiM, was auf den größeren Anteil des Overhead für das Erstellen der Infrastruktur zurüchzuführen ist. Das unterschiedliche Verhalten von CouchTM und MaJorToM bei den RTM- und CXTM-Tests lässt sich damit erklären, dass CouchDB mit zunehmender Datenbankgröße langsamer wird. In dem Test in Kapitel 3 wurde dies schon aufgezeigt und durch die große Anzahl an Views in CouchTM wird dieses Verhalten hier noch verstärkt.

6.1.4. Importieren/Exportieren von Topic Maps mit TMAPIX

Die bisherigen Tests arbeiteten alle auf Daten, die hauptsächlich zu Testzwecken erstellt wurden. Um auch die Geschwindigkeit auf realen Daten zu messen, wird noch das Importieren und Exportieren von Topic Maps getestet, die realitätsnahe Daten enthalten. Hierzu wird TMAPIX-IO, Version 1.0.0, verwendet. Als Topic Maps kommen die Leipzig Topic Map[26] und die Donald Duck Topic Map[27] von Maiana zum Einsatz. Beide Topic Maps sind unterschiedlich komplex: die Leipzig Topic Map beinhaltet 183 Topics, 101 Associations, 136 Occurrences, 72 Names und 260 Locators; die Donald Duck Topic Map beinhaltet 278 Topics, 950 Associations, 668 Occurrences, 494 Names und 555 Locators.

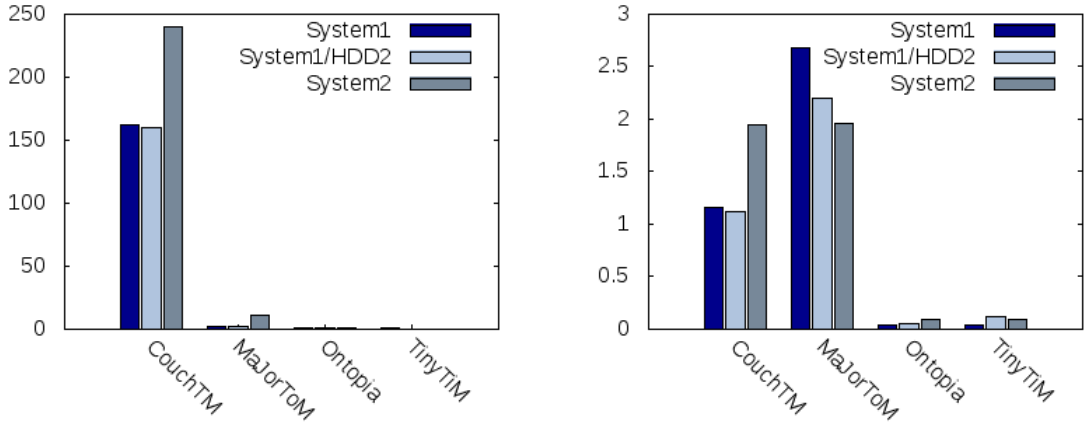

Abbildung 6.10.: Leipzig Topic Map import/export

Auch bei den Import- und Export-Tests zeigt sich, dass CouchTM auf komplexen Daten weniger performant ist als die anderen Engines. Das Importieren dauert ein Vielfaches der Zeit, die die anderen Engines brauchen. Weiterhin scheint es keinen linearen Zusammenhang zwischen der Größe der Topic Map und der benötigten Zeit zu geben. Vor allem bei der kleineren Leipzig Topic Map schneidet CouchTM besonders schlecht ab. Im Vergleich zu der Anzahl der anderen Konstrukttypen, besitzt sie verhältnismäßig viele Topics. Es scheint

[26]Leipzig Topic Map, http://maiana.topicmapslab.de/u/uta/tm/leipzig
[27]Donald Duck Topic Map, http://maiana.topicmapslab.de/u/lmaicher/tm/ducks

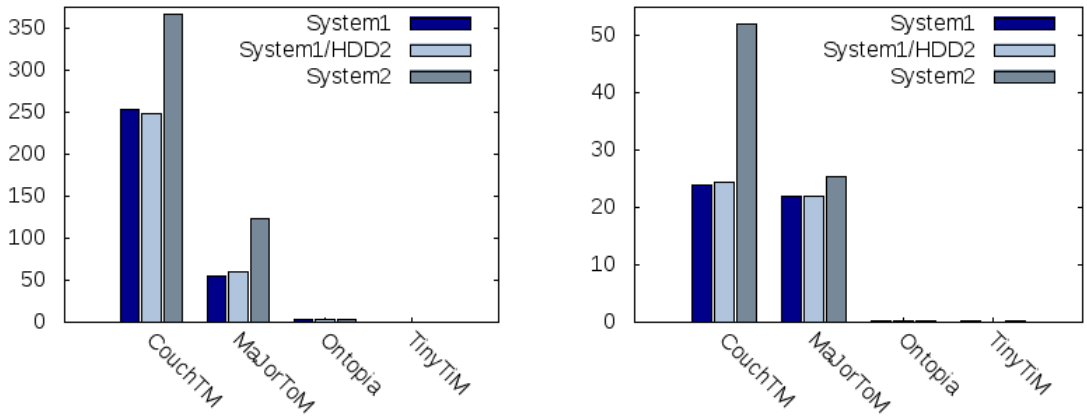

Abbildung 6.11.: Donald Duck Topic Map import/export

also, als hätte die Anzahl der Topics den größten Einfluss auf die benötigte Zeit beim Importieren von Topic Maps. Beim Exportieren der Leipzig Topic Map ist CouchTM schneller als MaJorToM und bei der Donald Duck Topic Map sind die beiden Engines, ausgenommen auf System 2, gleichauf. Der Aufbau der Topic Map scheint daher bei CouchTM keinen großen Einfluss auf die Geschwindigkeit beim Exportieren zu haben. Bei CouchTM und MaJorToM ist weiterhin zu beobachten, dass die benötigte Zeit für das Exportieren überproportional zu der Größe der Topic Map zunimmt. Ontopia ist sowohl beim Importieren als auch beim Exportieren weitaus schneller und erreicht fast die von tinyTiM erzielten Werte.

6.1.5. Einfluss von Prozessor- und Festplattenleistung

Um den reinen Einfluss der Geschwindigkeit der Festplatte zu messen, wurden alle schreib- und lese Tests auf System1 nicht nur auf der sehr schnellen SSD-Festplatte, sondern auch auf einer relativ alten und langsamen herkömmlichen Festplatte durchgeführt. Sämtliche anderen Tests wurden bereits auf beiden Festplatten absolviert, doch nur so kann festgestellt werden, dass etwaige Unterschiede nicht durch den Einsatz der Engine verfälscht werden.

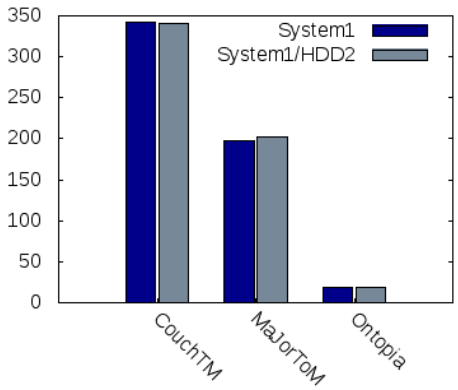

Abbildung 6.12.: Schreiben von 10000 Konstrukten, verschiedenen Festplatten

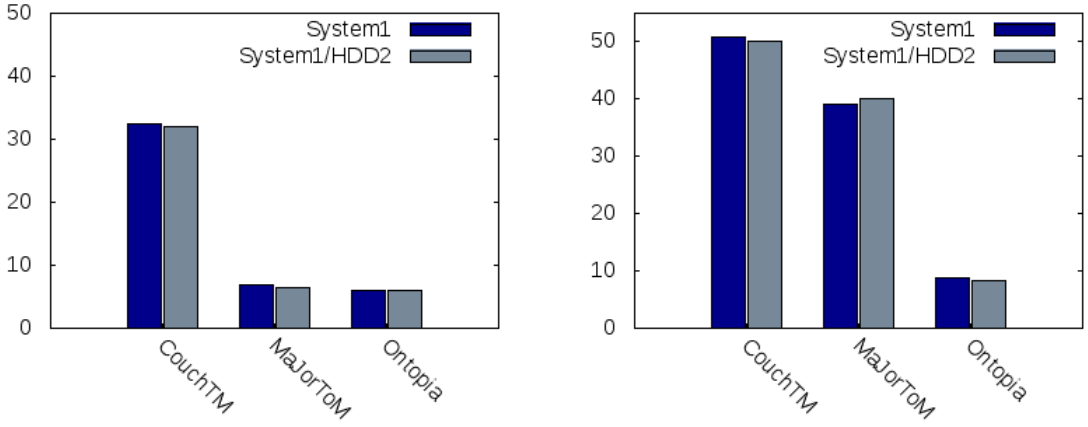

Abbildung 6.13.: Lesen von 10000 Konstrukten nach ID/Locator, verschiedenen Festplatten

Bei keinem dieser Test konnte ein signifikanter Unterschied zwischen den beiden Systemen festgestellt werden. Weder für CouchTM noch für Ontopia oder MaJorToM spielte es Rolle, auf welcher Festplatte die Daten gespeichert wurden. Bei aktuellen Topic-Maps-Engines scheint die Geschwindigkeit der Festplatte unter gegebenen Bedingungen keine bzw. nur eine untergeordnete Rolle zu spielen. Die Größe der in den Tests verwendeten Daten ist jedoch im Vergleich zum zur Verfügung stehenden Arbeitsspeicher eher gering, sodass sie von den DBMS problemlos im Arbeitsspeicher gehalten werden können und beim Lesen der Zugriff auf die Festplatte minimal ist. Auch beim Schreiben ist die Festplattenleistung wieder erwartend nicht von Bedeutung, sondern die Prozessorleistung ausschlaggebend für die Geschwindigkeit.

Alle Tests wurden auch auf zwei Systemen mit unterschiedlichen Prozessoren durchgeführt. System1 besitzt einen Dual-Core-Prozessor mit 2.2 GHz pro Kern, System 2 einen Single-Core-Prozessor mit 2 GHz. Bei Ontopia und tinyTiM lassen sich keine Geschwindigkeitsveränderung zwischen den beiden Systemen feststellen, wobei bei die für tinyTiM gemessenen Werte meist so klein waren, dass aus ihnen kaum allgemeingültige Schlüsse gezogen werden können. Bei CouchTM und MaJorToM liegt hingegen ein Geschwindigkeitsunterschied vor. Mit CouchTM sind die Schreib- und Lesetests auf System1 etwa 25% schneller als auf System2. Das Importieren und Exportieren von Topic Maps und die Tests der Testframeworks sind sogar 25%-35% schneller. Bei MaJorToM ist ähnliches zu beobachten. Der Geschwindigkeitszuwachs ist sogar noch leicht größer.

6.1.6. Einfluss der Anzahl der Views

Die wichtigste Ressource für die Geschwindigkeit von CouchTM ist also die Prozessorleistung. Doch wozu wird die Prozessorleistung benötigt? Bei jedem Zugriff auf einen View muss für jedes geänderte oder neue Dokument überprüft werden, ob sich dadurch ein View-Index ändert. Dazu werden die Map- und Reducefunktionen eines jeden Views in der Datenbank

durchlaufen und etwaige Änderungen der Indizes gespeichert. Diese Aufrufe verursachen daher auf Datenbanken mit vielen Views hohe Kosten für diese Aktionen. Da CouchTM mit 27 Views arbeitet kann angenommen werden, dass die View-Updates einen großen Einfluss auf die totale Zeit einer schreibenden Transaktion haben. Da vor jedem Schreiben eines Konstruktes ein View abgefragt wird, um zu testen, ob solch ein Konstrukt schon existiert, ist der Aufwand für das Aktualisieren der Views den schreibenden Aktionen zuzurechnen. Lesende Transaktionen sollten von der Anzahl der Views nicht beeinflusst werden. Um dies zu untersuchen, wurden alle Schreib- und Lesetests auf einer modifizierten Version von CouchTM durchgeführt. Diese beinhaltet nur die zwei für diese Tests benötigten Views, arbeitet allerdings nur in diesem Szenario korrekt.

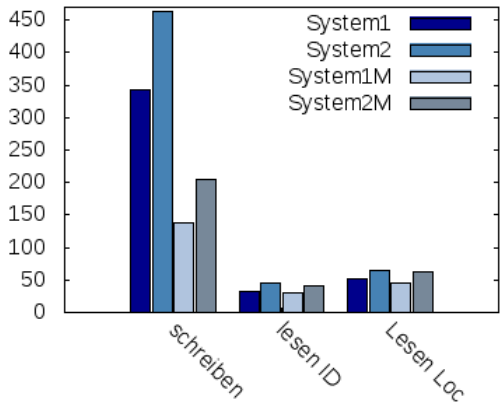

Abbildung 6.14.: Lesen und schreiben von 10000 Konstrukten in einer DB mit 2 Views

Wie erwartet führt eine Reduzierung der Anzahl der Views zu einer deutlich besseren Geschwindigkeit. Die Version mit zwei Views brauchte auf System1 nur 40% der Zeit im Vergleich zur normalen Version und auf System2 sind es nur 44% der Zeit. Auf die Lesezeit hat die Reduzierung der Viewanzahl wie erwartet keinen Einfluss.

7. Zusammenfassung

In dieser Arbeit wurde gezeigt, dass sich dokumentenorientierte Datenbanksysteme am Beispiel von CouchDB für den Einsatz als Topic-Maps-Engines-Backend eignen. Dazu wurde mit CouchTM eine Topic-Maps-Engine mit CouchDB-Backend implementiert und ausgiebig getestet. CouchTM ist damit nicht nur die erste Engine mit CouchDB-Backend, sonder auch die erste Engine mit einem dokumentenorientierten Datenbankbackend überhaupt. Das Abbilden der Topic-Map-Konstrukte auf die Datenbank ist dabei vergleichsweise einfach. Jedes Konstrukt wird durch ein Dokument repräsentiert. Zu jeder Topic Map im System existiert eine eigene Datenbank, in die die zugehörigen Konstrukte gespeichert werden. Anfragen, die in Engines mit relationalem Datenbankbackend ein SQL-Query benötigen, werden in CouchTM durch CouchDB-Views realisiert (ausgenommen Anfragen nach ID, für die die Document-API genutzt werden kann).

Unter Verwendung der TMAPI-Tests und RTM-Tests wurde die Korrektheit der Implementierung gezeigt. Zwar konnten nicht alle TMAPI- und RTM-Tests erfolgreich absolviert werden, doch die wenigen Tests, die fehl schlugen, sind nicht auf Fehler in der Implementation zurück zu führen. Weiterhin wurden noch die CXTM-Tests durchgeführt. CouchTM besteht 95% dieser. Das Bestehen aller CXTM-Tests war kein Ziel dieser Arbeit. Dies scheint zur Zeit auch nicht möglich zu sein, da keine der Referenzengines alle dieser Tests besteht. Ursächlich dafür ist, dass sich die CXTM-Tests selber noch in der Entwicklungsphase befinden.

Neben der Korrektheit wurde noch die Geschwindigkeit von CouchTM anhand von verschiedenen Tests gemessen und mit der von den in Kapitel 2 genannten Engines verglichen. Da bisher keine Topic-Maps-Engine ein dokumentenorientiertes Datenbanksystem als Backend benutzt, konnte CouchTM nicht mit einer Engine verglichen werden, welche die selbe Backendtechnologie besitzt. Weil tinyTiM, im Gegensatz zu den anderen Engines, nur ein in-memory-Backend benutzt, sind die von dieser Engine erzielten Werte nicht denen der Engines mit persistentem Backend zu vergleichen. Von den Engines mit persistentem Backend erzielte Ontopia, welches mit PostgreSQL-Backend getestet wurde, die besten Werte. Die dritte Vergleichsengine war MaJorToM, ebenfalls mit PostgreSQL-Backend. Die Geschwindigkeit von CouchTM ist am ehesten in der Region von MaJorToM anzusiedeln. Bei den TMAPI- und CXTM-Tests war CouchTM performanter als MaJorToM; bei den anderen Tests war es umgekehrt. Da CouchDB noch nicht mit der Geschwindigkeit von relationalen Datenbanksystemen mithalten kann, ist dies akzeptabel. CouchTMs Geschwindigkeit nimmt ab, je mehr Daten sich in einer Topic Map befinden. Dies haben die RTM-Tests sowie die Import- und Export-Tests gezeigt. Da CouchDB mit zunehmender Datenmenge in einer Datenbank langsamer wird, ist dies nicht verwunderlich. CouchDB befindet sich

jedoch noch in einem sehr frühen Stadium der Entwicklung (Version 1.0 wurde erst Mitte 2010 veröffentlicht), weswegen die weitere Entwicklung abzuwarten ist. Eine Verbesserung der Leistung von CouchDB in folgenden Versionen würde sich umgehen positiv auf die Geschwindigkeit von CouchTM auswirken. Mit MongoDB gibt es ein weiteres populäres dokumentenorientiertes Datenbanksystem, dessen Einsatz als Topic-Maps-Engine-Backend interessant wäre, da es aktuell schneller als CouchDB ist.

Momentan gibt es zwei Einsatzgebiete von CouchDB, die sich direkt aus den Resultaten der Performancetests ableiten lassen. Wird mit Topic Maps gearbeitet, auf denen viele Änderungen durchgeführt werden, so eignet sich CouchTM nur dann, wenn diese Topic Maps klein sind, da die Geschwindigkeit von schreibenden Aktionen abnimmt, je größer die Topic Map wird. Verursacht wird dieses Verhalten durch die vielen View-Updates. Überwiegen die lesenden Aktionen stark, so ist die Anzahl der View-Updates weitaus geringer und CouchTM kann auch auf größeren Topic Maps eingesetzt werden. Aufgrund der guten Skalierbarkeit von CouchDB könnte CouchTM jedoch auf einem CouchDB-Cluster auch auf großen Topic Maps mit einem höherem Anteil an schreibenden Aktionen performant laufen. Allerdings unterstützt das daür benötigte CouchDB Lounge nicht die aktuelle Version, sondern nur 0.10. Mangels der dazu benötigten Infrastruktur konnte der Einsatz von CouchTM auf einem CouchDB-Cluster jedoch nicht getestet werden. Außerdem bedarf es kleinen Änderungen an CouchTM, damit es auf CouchDB 0.10 läuft. In Szenarien, in denen mehrere User ein Konstrukt gleichzeitig schreiben könnten, ist CouchTM wegen der fehlenden Konfliktbehandlung oder -vermeidung noch nicht einsetzbar.

Im Zuge der Weiterentwicklung von CouchTM gibt es drei Dinge, die vorrangig behandelt werden sollten:

Zur Zeit ist CouchTM noch nicht in Umgebungen einsetzbar, in denen mehrere Nutzer gleichzeitig das selbe Konstrukt ändern könnten. Um dies zu realisieren, müssen erst einmal alle zu einer Aktion auf der Topic Map gehörenden Schreiboperationen in einer Transaktion bzw. Job zusammen gefasst werden. Weiterhin bedarf es eines Lock Proxys, an welchen diese Jobs geschickt werden. Der Lock-Proxy vergibt, wie der Name schon sagt, Locks auf alle Dokumente der Datenbank, die durch diesen Job geändert werden und führt dann diese Schreiboperationen aus. Gibt es nun solche Jobs, so können auch alle darin beinhalteten Operationen in einem Bulk-Write ausgeführt werden. Das bringt einen Performancegewinn im Gegensatz zu den jetzt eingesetzten Single-Writes.

Weiterhin sollte noch versucht werden, die Anzahl der von CouchTM benötigten Views zu reduzieren. Eine Reduzierung der Views zieht, wie in den Performancetests gezeigt wurde, eine Verbesserung der Geschwindigkeit nach sich. Eine Reduzierung kann dadurch erreicht werden, dass Views zusammengefasst werden, sodass sie von mehreren Arten von Anfragen genutzt werden können. Dabei darf jedoch die Komplexität der View-Funktionen nicht steigen. Es wäre z.B. denkbar für jeden TMAPI-Index nur einen View zu benutzen und den documentType mit den in den Key zu nehmen.

Schließlich bietet es sich noch an, anstelle der TMAPI die MaJorToM-Interfaces zu implementieren und CouchTM um die in den Interfaces enthaltenen zusätzlichen Funktionalitätet zu erweitern.

A. Anhang

A.1. Messwerte

Die Einheiten sämmtlicher gemessener Werte sind Millisekunden; bei den RTM-Tests und CXTM-Tests sind es Sekunden.

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
schreiben	337614	346733	344556	339416	342462	342156	3706
lesen ID	32383	31914	33110	32418	32488	32462	427
lesen Locator	50945	50572	50548	50751	51110	50785	241

Tabelle A.1.: CouchTM System1 schreiben/lesen

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
schreiben	341932	340726	341265	341663	340882	341293	509
lesen ID	32178	31786	32081	31848	32001	31979	162
lesen Locator	50441	50162	50298	50058	50105	50212	156

Tabelle A.2.: CouchTM System1/HDD2 schreiben/lesen

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
schreiben	138914	138956	137378	138573	137717	138307	719
lesen ID	30253	30018	29881	29848	29733	29946	199
lesen Locator	45737	45938	45844	46125	45947	45918	143

Tabelle A.3.: CouchTM System1 schreiben/lesen 2 Views

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
schreiben	413588	468612	477557	474376	482271	463280	28219
lesen ID	46077	46574	47281	47216	41370	45703	2472
lesen Locator	61130	63272	66346	63541	67972	64452	2703

Tabelle A.4.: CouchTM System2 schreiben/lesen

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
schreiben	167567	182964	261842	210264	203769	205281	35870
lesen ID	39009	37193	46273	43708	41491	41534	3619
lesen Locator	54628	73977	67304	58958	57700	62513	7942

Tabelle A.5.: CouchTM System2 schreiben/lesen 2 Views

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
schreiben	149	83	83	85	83	97	29
lesen ID	7	3	3	3	3	3.8	1.8
lesen Locator	6	4	4	4	4	4.4	0.9

Tabelle A.6.: tinyTiM System1 schreiben/lesen

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
schreiben	117	60	80	61	63	76	24
lesen ID	5	3	2	2	2	2.8	1.3
lesen Locator	4	2	3	3	3	3	0.7

Tabelle A.7.: tinyTiM System2 schreiben/lesen

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
schreiben	59489	127750	197965	268080	335494	197755	109471
lesen ID	7768	6124	6219	8265	6099	6895	1040
lesen Locator	41267	39306	38191	38347	38373	39096	1290

Tabelle A.8.: MaJorToM System1 schreiben/lesem

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
schreiben	60584	135674	202249	271078	339447	201806	109614
lesen ID	6558	6543	6537	6598	6497	6546	36
lesen Locator	39498	39581	41379	39929	39473	39972	807

Tabelle A.9.: MaJorToM System1/HDD2 schreiben/lesen

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
schreiben	358620	424967	548931	673305	790084	559181	176552
lesen ID	4577	4566	7021	11219	4546	6385	2904
lesen Locator	57902	60851	58851	55748	63699	59410	3019

Tabelle A.10.: MaJorToM System2 schreiben/lesen

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
schreiben	19369	19051	19191	18854	19119	19116	119
lesen ID	7529	5730	5569	5946	5430	6040	854
lesen Locator	7724	9962	10292	7885	7883	8749	1265

Tabelle A.11.: Ontopia System1 schreiben/lesen

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
schreiben	18887	18860	18749	18606	18730	18776	112
lesen ID	6403	7017	5383	5391	5402	5919	753
lesen Locator	7743	7882	8149	8171	9226	8234	583

Tabelle A.12.: Ontopia System1/HDD2 schreiben/lesen

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
schreiben	21565	20571	20010	16599	19743	19697	1868
lesen ID	4280	5800	7740	11537	5945	7060	2787
lesen Locator	10204	4901	10116	6446	4929	7319	2668

Tabelle A.13.: Ontopia System2 schreiben/lesen

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
lesen	37246	37349	37061	36966	37181	37161	151
schreiben ID	24832	24897	24945	24946	25024	24929	71
schreiben Locator	47879	48057	48066	48515	48318	48167	250

Tabelle A.14.: System1 lesen/schreiben ohne Engine

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
lesen	37403	36819	36743	36686	36804	36891	291
schreiben ID	25173	25193	25178	25143	25107	25159	34
schreiben locator	48440	48590	48407	48267	48361	48413	118

Tabelle A.15.: Systm1/HDD2 lesen/schreiben ohne Engine

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
lesen	62446	59076	57947	58449	54527	58489	2828
schreiben ID	36045	36183	35131	35889	35797	35809	407
schreiben Locator	59028	62111	61792	63953	64382	62253	2125

Tabelle A.16.: System2 lesen/schreiben ohne Engine

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
System1	36520	35305	35192	34800	35038	35371	670
System1/HDD2	38350	37263	35710	35283	34971	36315	1439
System2	46083	43276	45965	43525	47220	45214	1729

Tabelle A.17.: Bulk schreiben

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
System1	51739	53461	54113	54504	54543	53672	1164
System1/HDD2	54728	53663	53700	54187	54195	54094	436
System2	72153	87589	84000	82343	85721	82361	6031

Tabelle A.18.: CouchTM TMAPI-Tests

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
System1	72721	71943	72275	72413	72672	72404	316
System1/HDD2	74203	72969	73180	73379	73228	73391	476
System2	117287	121451	128630	130192	119698	123451	5664

Tabelle A.19.: MaJorToM TMAPI-Tests

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
System1	304	393	361	136	128	264	125
System2	271	139	288	85	322	221	102

Tabelle A.20.: tinyTiM TMAPI-Tests

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
System1	5272	5322	5345	5304	5342	5317	30
System1/HDD2	5201	5179	5255	5281	5242	5231	41
System2	8774	8430	8484	8656	8872	8643	187

Tabelle A.21.: CouchTM RTM-Tests

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
System1	1143	1136	1135	1132	1135	1136	4
System1/HDD2	1124	1135	1137	1129	1133	1132	5
System2	6515	6471	6519	6574	6625	6541	60

Tabelle A.22.: MaJorToM RTM-Tests

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
System1	12.8	12.9	12.8	12.8	12.9	12.84	0.05
System2	12.6	11.7	11.7	11.9	12.9	12.2	0.55

Tabelle A.23.: tinyTiM RTM-Tests

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
System1	1502	1497	1482	1562	1547	1518	35
System1/HDD2	1482	1500	1497	1532	1511	1504	19
System2	2259	2207	2315	2284	2375	2288	63

Tabelle A.24.: CouchTM CXTM-Tests

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
System1	1768	1751	1782	1755	1794	1770	18
System1/HDD2	1782	1745	1775	1784	1771	1771	16
System2	2765	2691	2814	2785	2845	2780	58

Tabelle A.25.: MaJorToM CXTM-Tests

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
System1	94.8	96.5	96.5	96.2	96.1	96	0.7
System2	111.3	156.3	148.5	161.2	141.7	143.8	19.6

Tabelle A.26.: tinyTiM CXTM-Tests

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
Leipzig import	168134	163404	158690	159282	159257	161753	4036
Leipzig export	1201	1096	1089	1312	1114	1162	95
Donald import	263324	260819	248336	248015	246072	253313	8091
Donald export	26701	23072	23314	23142	23127	23871	1584

Tabelle A.27.: CouchTM System1 TMAPIX-Tests

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
Leipzig import	162837	159262	159503	159676	159733	160202	1484
Leipzig export	1195	1092	1104	1089	1094	1115	45
Donald import	248647	248962	249040	247991	248063	248540	492
Donald export	27179	23644	24207	23592	23634	24451	1546

Tabelle A.28.: CouchTM System1/HDD2 TMAPIX-Tests

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
Leipzig import	212926	242950	247281	252481	247129	240553	15810
Leipzig export	1579	1290	3147	2212	1495	1944	755
Donald import	343571	370803	363072	404469	355105	367404	23040
Donald export	49042	52476	51750	55543	51208	52003	2357

Tabelle A.29.: CouchTM System2 TMAPIX-Tests

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
Leipzig import	2470	2326	2087	2311	2322	2303	137
Leipzig export	3078	2534	2180	2198	3380	2674	537
Donald import	59758	52795	54763	52313	52573	54440	3127
Donald export	19997	21396	24128	24092	20450	22012	1980

Tabelle A.30.: MaJorToM System1 TMAPIX-Tests

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
Leipzig impoer	2450	2143	2129	2125	2132	2196	142
Leipzig export	2421	2170	2136	2161	2119	2201	124
Donald import	89877	52917	52899	51704	51486	59777	16840
Donald export	23215	21673	23547	23203	18809	22089	1973

Tabelle A.31.: MaJorToM System1/HDD2 TMAPIX-Tests

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
Leipzig import	12341	10055	10154	11626	9904	10816	1099
Leipzig export	1979	1902	1880	2156	1886	1960	116
Donald import	125855	123956	121560	127106	118040	123303	3610
Donald export	24427	25674	26303	26293	24316	25402	976

Tabelle A.32.: MaJorToM System2 TMAPIX-Tests

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
Leipzig import	1012	761	780	770	776	820	108
Leipzig export	86	41	25	26	25	41	26
Donald import	4546	4011	3919	3925	3953	4070	268
Donald export	193	124	160	150	148	155	25

Tabelle A.33.: Ontopia System1 TMAPIX-Tests

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
Leipzig import	991	722	757	748	757	795	111
Leipzig export	84	64	50	48	46	58	16
Donald import	4537	4008	3997	3949	3965	4091	250
Donald export	198	146	141	139	136	152	26

Tabelle A.34.: Ontopia System1/HDD2 TMAPIX-Tests

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
Leipzig import	765	593	617	641	570	637	76
Leipzig export	82	276	29	28	53	94	104
Donald import	3977	3572	3721	3648	3530	3690	177
Donald export	200	171	164	396	415	269	125

Tabelle A.35.: Ontopia System2 TMAPIX-Test

s

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
Leipzig import	106	74	39	38	39	59	30
Leipzig export	63	21	42	39	20	37	18
Donald import	329	249	257	275	284	279	31
Donald export	290	124	91	88	106	140	85

Tabelle A.36.: tinyTiM System1 TMAPIX-Tests

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
Leipzig import	100	44	287	275	39	149	123
Leipzig export	61	255	21	21	273	126	127
Donald import	490 2	57	259	248	253	301	106
Donald export	113	109	130	101	128	116	12

Tabelle A.37.: tinyTiM System1/HDD2 TMAPIX-Tests

	1. Messung	2. Messung	3. Messung	4. Messung	5.Messung	Mittelwert	Standardabweichung
Leipzig import	349	44	164	147	112	163	114
Leipzig export	88	53	75	183	79	96	51
Donald import	1064	267	661	1418	714	825	436
Donald export	304	99	324	342	287	271	98

Tabelle A.38.: tinyTiM System2 TMAPIX-Tests

Literaturverzeichnis

[ALS10] J. Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide*. O'Reilly Media, Inc., 2010.

[Arm07] Joe Armstrong. History of erlang. In *3rd ACM SIGPLAN conference on History of programming languages*, 2007.

[BBSM09] Arnim Bleier, Benjamin Bock, Uta Schulze, and Lutz Maicher. JRuby Topic Maps. In Maicher and Garshol [MG09], pages 195–205.

[Boc09] Benjamin Bock. Topic-Maps-Middleware. Modellgetriebene Entwicklung kombinierbarer domänenspezifischer Topic-Maps-Komponenten. Diplomarbeit, Universität Leipzig, 2009.

[Bre00] Eric A. Brewer. Towards Robust Distributed Systems. In Neiger [Nei00], page 7.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004.

[FST02] Roy T. Fielding, Day Software, and Richard N. Taylor. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology*, 2:115–150, 2002.

[HS10] Lars Heuer and Johannes Schmidt. TMAPI 2.0. In Maicher and Garshol [MG08], pages 129–136.

[ISOa] ISO13250-2.
ISO/IEC IS 13250-2:2008: Information Technology - Document Description and Processing Languages - Topic Maps - Data Model.
International Organization for Standardization, Geneva, Switzerland.

[ISOb] ISO13250-5.
ISO/IEC IS 13250-5:2007: Information Technology - Document Description and Processing Languages -

Topic Maps - Reference Model.
International Organization for Standardization, Geneva, Switzerland.

[MG08] Lutz Maicher and Lars Marius Garshol, editors. *TMRA 2008: Subject-centric Computing*. University of Leipzig, 2008.

[MG09] Lutz Maicher and Lars Marius Garshol, editors. *TMRA 2009: Linked Topic Maps*. University of Leipzig, 2009.

[Nei00] Gil Neiger, editor. *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA*. ACM, 2000.

[NSQ10] Nosql - The Definite Guide, 2010. http://nosql-berlinbuzzwords2010.heroku.com/.