Institut für Informatik der Freien Universität Berlin
Arbeitsgruppe Software Engineering

Persistierung dynamischer RDF-Graphen mit GAE

Bachelorarbeit

Benjamin Weißenfels
bw@pixeldrama.de

Betreuer: Dr. Edzard Höfig
Gutachter: Prof. Dr. Ina Schieferdecker
Zweitgutachter: Prof. Dr. Agnès Voisard

Berlin, 1. August 2013

Diese Arbeit hat zum Ziel große RDF-Graphen in der Cloud zu speichern und zu verändern. Als Cloud-Computing-Plattform kommt Google Appengine zum Einsatz. Es wird versucht ein möglichst effizientes Datenmodell zu entwickeln, das die Besonderheiten des zugrunde liegenden verteilten Datenbankbanksystems Bigtable berücksichtigt. Weiter wird eine REST-API implementiert, mittels derer RDF-Graphen manipuliert werden können, sowie ein Data-Access-Object, das von den nativen Datenbankzugriffen abstrahiert. Die Evaluierung erfolgt durch Laufzeitmessungen von verschiedenen Implementierungen des Data-Access-Objects.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, 19. September 2013

Inhaltsverzeichnis

1 Einführung **6**

1.1 Einleitung . . . 6
1.2 Problemstellung und Motivation . . . 7
1.3 Aufbau der Arbeit . . . 8
1.4 Methodik . . . 8

2 Grundlagen **10**

2.1 Mathematische Grundlagen . . . 10
2.2 Resource Description Framework (RDF) . . . 11
2.2.1 Einführung . . . 11
2.2.2 RDF Formalia . . . 13
2.2.3 RDF Definitionen . . . 16
2.2.4 SPARQL . . . 17
2.2.5 Das Problem der transitiven Hülle . . . 17
2.3 Google App Engine . . . 18
2.3.1 Einführung . . . 18
2.3.2 Sicherheitslimitierungen und abrechenbare Resourcen . . . 19
2.3.3 App Engine Datastore . . . 19
2.3.4 Instanzen . . . 25
2.3.5 GAE Laufzeitumgebungen . . . 25
2.4 Weitere Triplestores, die nicht auf relationalen Datenbanksystemen aufsetzen 26
2.5 REST . . . 27

3 Implementierung **28**

3.1 Werkzeuge . . . 30
3.2 Frameworks . . . 30
3.2.1 Apache Jena . . . 30
3.2.2 Jersey . . . 30
3.2.3 Objectify . . . 30

3.3 API . . . 33
3.3.1 Persistierung und Löschen von Tripeln . . . 33
3.3.2 Abfragen von Tripeln . . . 34
3.4 Datenbankschemata . . . 35
3.4.1 Flaches Datenbankschema (Flat-Schema) . . . 35
3.4.2 Verschachteltes Datenbankschema (Nested-Schema) . . . 37
3.4.3 Verzeichnis-Datenbankschema (Dict-schema) . . . 39
3.4.4 Verzeichnis-Datenbankschema mit Rückwärtsreferenzen (Dictbackref-Schema) . . . 40
3.5 Geschätzte Schreib- und Lesekosten der Datenbankschemata für den Import von RDF-Tripeln . . . 42
3.6 Das Blank-Node-Problem . . . 45
3.6.1 Interne Repräsentation von Blank Nodes . . . 46
3.6.2 Probleme beim Speichern von Blank Nodes . . . 46
3.7 Schnelles Zählen und gleichzeitige Schreibzugriffe auf den Triplestore . . . 49

4 Evaluierung 50
4.1 Test-Pipeline . . . 50
4.1.1 Vorgehen . . . 51
4.1.2 Erstellen und Abfragen von Logs . . . 52
4.2 Persistierung der RDF-Tripel . . . 53
4.3 Abfragen . . . 55

5 Fazit 62
5.1 Kritische Betrachtung des Prototypen . . . 62
5.1.1 Persistierung multipler Graphen . . . 62
5.1.2 Erweiterung der API für Limit- und Offset-Anfragen . . . 64
5.1.3 Speichern von großen Literalen in Blobs . . . 64
5.2 Ausblick . . . 64
5.2.1 Implementierung eines SPARQL-Endpoints . . . 64
5.2.2 Performance Tests auf Open-Source Implementierung des GAE-Interface . . . 65

1 Einführung

1.1 Einleitung

Gegenstand dieser Arbeit ist es sehr große Graphen in Form von RDF-Tripeln in der Cloud zu speichern und zu manipulieren. RDF steht für Resource Description Framework. Ein RDF-Tripel ist ein einfacher Aussagesatz, der aus Subjekt, Prädikat und Objekt besteht. Eine Menge von RDF-Tripeln ist ein RDF-Graph. Mit den Begriffen RDF und Cloud baut diese Arbeit auf zwei Themen auf, die in den letzten Jahren in der Informatik und auch allgemein in der Gesellschaft und Wirtschaft zunehmend häufiger diskutiert werden: dem Semantic Web und dem Cloud-Computing. Der Begriff RDF-Graph aus dem Titel dieser Arbeit wird dem Bereich Semantic Web zugeordnet und Cloud ist als Synonym für Cloud-Computing anzusehen.

Cloud-Computing steht im weitesten Sinne für das Auslagern von IT-Infrastruktur und von Software zu Drittanbietern über eine Netzwerkanbindung. Eine feste Definition des Begriffs Cloud-Computing war geraume Zeit nicht möglich [11] und der Begriff Cloud (Wolke) selbst steht schon für eine gewisse Unschärfe. Im Kapitel 2 wird eine genauere Definition von Cloud-Computing gegeben. Viele große Internetdienstleister wie Amazon, Google, Microsoft oder Apple bieten mittlerweile Cloud-Computing in verschiedenen Ausprägungen an. Motivation für die Nutzung von Clouddiensten sind unter anderem die große Flexibilität bei der Planung von IT-Projekten, Kostensenkung, Skalierbarkeit und Verfügbarkeit. Ein denkbares Szenario: Eine Webapplikation, die Videos streamt und in verschiedene Formate umwandelt, wird in kurzer Zeit sehr erfolgreich. Statt physisch neue Server zu kaufen und zu warten, buchen die Entwickler der imaginären Webapplikation über ein Webfrontend mehr Rechenkapazität und Bandbreite. Auf der einen Seite sind in diesem Szenario die Entwickler von einer Reihe an organisatorischen Herausforderungen befreit, auf der anderen Seite steigt die Abhängigkeit von einem zusätzlichen Dienstleister. Ein weiteres Beispiel ist der Webdienst Dropbox[1]. Dieser bietet Synchronisierung von Dateien zwischen Rechnern in der Cloud an, basierend auf

[1] `https://www.dropbox.com/` (abgerufen 31.07.2013)

Amazons Simple-Storage-Service (S3)[2] und ist damit ein Clouddienstleister, der für seine Dienstleistung einen weiteren Clouddienstleister benutzt.[3] Damit kann Dropbox sehr flexibel auf höheren Festplattenspeicherbedarf reagieren.

Semantic Web, ebenfalls ein unscharfer Begriff, bezeichnet aus der Sicht des World Wide Web Consortium (W3C) das Web der Daten.[4] Derzeit ist das Web dokumentorientiert. Es ist für Menschen lesbar, interpretierbar und navigierbar, nicht aber für Maschinen. Um die Vision von Aaron Swartz wahr zu machen, dass Computer ähnlich wie Menschen das Web benutzen [12] und Daten, die von Software verarbeitet und gespeichert werden, wie z. B. Wikis, Foren oder Suchmaschinen, auch für Maschinen navigierbar und verständlich werden (Linked Data)[5], wurden von der W3C Semantic Web Activity zahlreiche Spezifikationen entwickelt: RDF, SPARQL, GRDDL, RIF, OWL 2, POWDER usw.[6] RDF ist entwickelt worden, um Inhalte einheitlich zu strukturieren, während SPARQL Protocol And RDF Query Language (SPARQL) zum Abfragen von RDF-Daten gedacht ist.

1.2 Problemstellung und Motivation

Datensätze, die als RDF verfügbar gemacht werden, sind oftmals sehr groß oder können groß werden. Beispiele sind Fahrpläne des öffentlichen Nahverkehrs, Emissionswerte von Industriegebieten, Klimakarten und so weiter. Das Projekt DBpedia, dass die deutsche Wikipedia auf RDF abbildet, speichert rund 1,89 Milliarden RDF-Tripel[7]. Es ist davon auszugehen, dass ein System, das RDF-Tripel persistiert und abfragt (häufig Triplestore[8] genannt), auch skalieren muss. An dieser Stelle bietet sich Cloud-Computing an, bei dem man bei Bedarf Rechenzeit, Speicher oder Bandbreite dazubuchen kann. Eine weitere wichtige Anforderung an den Triplestore ist, dass er auch bei großen Graphen und einfachen Anfragen reaktiv bleibt, damit der Triplestore für Clients nicht zum Engpass wird.

Ziel dieser Arbeit ist der Versuch einen effizienten Triplestore mit Googles Webentwicklungsplattform Google-App-Engine (GAE) zu implementieren.

[2] `http://aws.amazon.com/de/s3/` (abgerufen 31.07.2013)
[3] `https://www.dropbox.com/help/7/en` (abgerufen 26.07.2013)
[4] `http://www.w3.org/2001/sw/` (abgerufen 26.07.2013)
[5] `http://linkeddata.org/` (abgerufen 26.07.2013)
[6] `http://www.w3.org/2001/sw/Specs` (abgerufen 20.5.2013)
[7] `http://wiki.dbpedia.org/Datasets` (abgerufen 4.4.2013)
[8] `http://www.w3.org/wiki/LargeTripleStores` (abgerufen 26.07.2013)

1.3 Aufbau der Arbeit

Zunächst werden mathematische und technologische Grundlagen erläutert. Anschließend wird die prototypische Implementierung vorgestellt. Dabei wird auf Entscheidungsprozesse bezüglich der Architektur des Prototypen eingegangen. Weiter wird die Testumgebung vorgestellt und anschließend die Tests ausgewertet und diskutiert. Abschließend wird ein Fazit gezogen, dass diese Arbeit kritisch betrachtet und einen Ausblick gibt, an welcher Stelle noch Forschungsbedarf besteht.

1.4 Methodik

Diese Arbeit geht in der Hauptsache empirisch vor. GAE ist eine proprietäre Plattform und damit sind nur Teile des Quellcodes der Cloud-Computing-Plattform einsehbar und die eingesetzte Hintergrundtechnologie ist nur so weit dokumentiert, wie es Google befürwortet. Für Vergleiche mit anderen Systemen muss man Kenntnisse über die verwendete Hardware, die Netzwerkanbindung und die durchschnittliche Auslastung der Hardware haben, also eine Laborsituation herstellen können. Somit bleibt nur einen Versuchsaufbau zu schaffen, welcher, nach Studium der von Google [3] veröffentlichten Dokumentation und Empfehlungen[9], möglichst optimal die zu definierenden Anforderungen versucht umzusetzen und anschließend Performanztests durchzuführen.

Die zu definierenden Anforderungen sind:

1. Eine API , die es ermöglicht Tripel zu speichern, zu löschen und zu suchen. Aus diesen drei einfachen Operationen lassen sich weitere Operationen ableiten, wie z. B. Verändern einzelner Tripel oder Tiefensuche auf Tripeln.

2. Effizienter Import für große Mengen an Tripeln.

3. Effizienter und skalierender Zugriff und Suche auf RDF-Tripeln.

Abbildung 1.1 gibt einen groben Überblick über das Szenario, dass aus den Anforderungen und dem Einsatz von GAE entsteht. Das Frontend ist in diesem Fall ein Script, das Tripel zufällig schreibt, sucht und verändert. Im Application-Server läuft der zu implementierende Prototyp, der möglichst effizient auf die Datenbank zugreift, in diesem Fall der App Engine Datastore.

[9] Ryan Barrett (Google) gibt Hinweise, wie man den App Engine Datastore effizient benutzt: `https://sites.google.com/site/io/under-the-covers-of-the-google-app-engine-datastore` (abgerufen 31.07.2013)

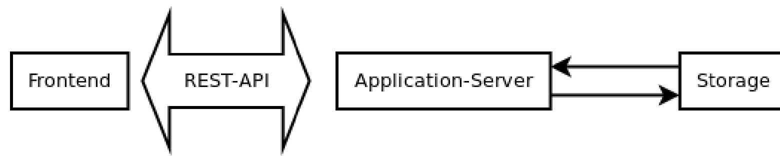


Abbildung 1.1: Szenario

Ausgewertet werden die Laufzeiten der Operationen des Data-Access-Objects, das im Application-Server angesiedelt ist. Die Netzwerklatenzen durch die REST [4] Anbindung werden nicht berücksichtigt.

2 Grundlagen

2.1 Mathematische Grundlagen

RDF-Daten sind mathematisch gesehen gerichtete Graphen. Ein Graph besteht aus einer Menge von Knoten und einer Menge von Kanten, formal ausgedrückt: $G = (V, E)$.

Definition 1. *(Transitive Hülle eines gerichteten Graphs) Ist definiert als der Graph* $G' = (V, E')$ *eines gegebenen Graphs* $G = (V, E)$ *und den Knotenpaaren* $v_i, v_j \in V$ *mit*

$$E' = \{(v_i, v_j) \mid v_i \rightarrow v_{k_1}, v_{k_1} \rightarrow v_{k_2}, \ldots, v_{k_n} \rightarrow v_j\}$$

Vereinfacht gesprochen, enthält E' *alle Knotenpaare, die direkt oder indirekt verbunden sind.*

Gerichtete Graphen sind Graphen, bei denen die Kanten nur in eine Richtung traversiert (beschritten) werden können. Dies kann die transitive Hülle eines gerichteten Graphen gegenüber eines normalen Graphen verkleinern, und tut es in der Praxis oft, es ist aber nicht zwangsläufig, da Zyklen erlaubt sind. Ein Zyklus ist ein Pfad ausgehend von einen Knoten, auf dem sich der Knoten wieder selbst erreicht:

$$(v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_1) \in E$$

Der kleinste mögliche Zyklus (Schleife) wäre beispielsweise ein Knoten mit einer Kante auf sich selbst:

$$(v \rightarrow v) \in E$$

Diese Pfade sind potentiell unendlich lang.

Definition 2. *(Transitive Hülle eines Knotens) Ist definiert als*

$$T(v) = \{v\} \cup \{u \mid v \rightarrow w_1, v \rightarrow w_2, \ldots, w_k \rightarrow u\} \text{ mit } v, u, w \in V$$

enthält also alle Knoten, die von v *erreichbar sind.*

Graphen (und auch RDF-Graphen), können Schleifen und lange Pfade aufweisen. Da in der Praxis Speicherplatz und Rechenzeit für die Traversierung eingeschränkt ist, ist es sinnvoll die Länge der Pfade zum Testen einzuschränken.

Definition 3. *(Beschränkte transitive Hülle eines Knotens) Ist für $v, u, w \in V$ und $n, k \in \mathbb{N}$ definiert als:*

$$C(v, 0) = \{v\}$$
$$C(v, n) = \{v\} \cup \{u \mid v \to w_1, v \to w_2, \ldots, w_k \to u \ \wedge k \leq n\}$$

Die Länge des Pfads ist also beschränkt und Pfadlänge 0 ist erlaubt.

2.2 Resource Description Framework (RDF)

2.2.1 Einführung

RDF wurde entwickelt, um Daten im Web automatisiert zu verarbeiten. Das folgende Zitat aus der W3C Empfehlung von 1999 verdeutlicht, worum es bei RDF geht:

> Resource Description Framework (RDF) is a foundation for processing metadata; it provides interoperability between applications that exchange machine-understandable information on the Web. RDF emphasizes facilities to enable automated processing of Web resources.[13]

Seit diesem Vorschlag hat sich RDF weiterentwickelt und beschränkt sich nicht auf Metadaten, sondern ist auch eine maschinell interpretierbare Beschreibung der Daten an sich.[2]

Das RDF-Datenmodell soll also Resourcen im Web beschreiben und mit Semantik aufladen, damit sie zumindest interpretierbar für Maschinen sind und soll damit zur Entwicklung des Semantic Webs beitragen. Dafür bedient sich RDF einfacher Aussagen, die aus Subjekt, Prädikat und Objekt bestehen. Das Subjekt ist immer eine Resource, während das Prädikat (Property genannt) eine Eigenschaft dieser Resource darstellt und das Objekt schließlich den Wert dieser Eigenschaft angibt. Ein Objekt kann wiederum auch eine Resource sein, womit sich eine Verkettung von Aussagen ergibt. Eine Resource kann mehrere Properties besitzen. Die Aussage „SV Werder Bremen steht auf dem 15. Tabellenplatz und spielt im Weserstadion“, würde in RDF den Graphen von Abbildung 2.1 ergeben.

Wenn man eine neue Aussage über das Weserstadion tätigt, z. B. „Das Weserstadion wurde 1923 gebaut“ erhält man eine Verkettung (Abb. 2.2).

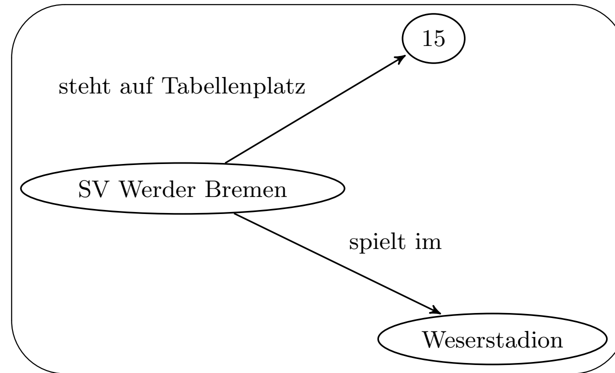


Abbildung 2.1: Einfache Aussagen als gerichteten Graphen abbilden.

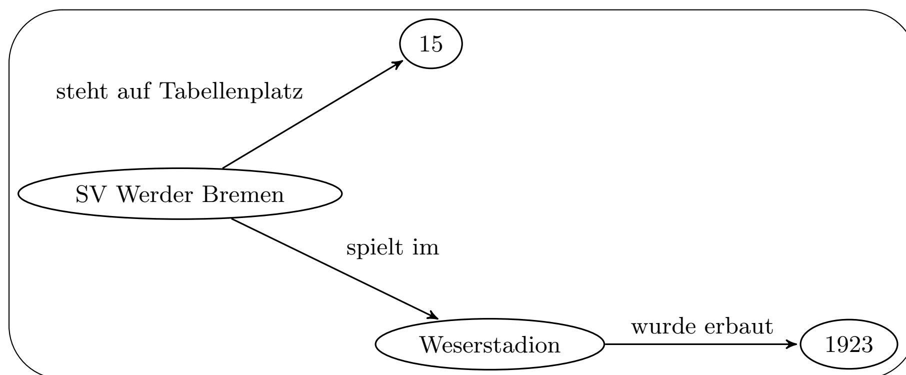


Abbildung 2.2: Verkettung

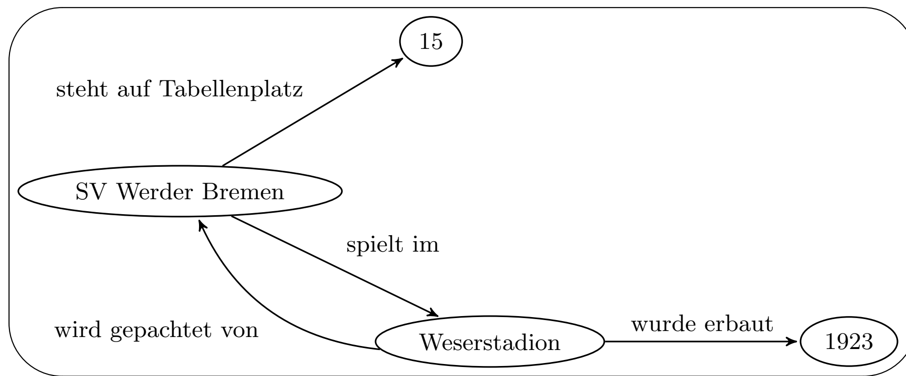


Abbildung 2.3: Zyklischer Graph

Fügt man noch die Aussage „Das Weserstadion wird gepachtet vom SV Werder Bremen“ ein, erhält man einen zyklischen Graphen (Abb. 2.3).

2.2.2 RDF Formalia

In RDF ist jede Aussage (Statement) ein Fakt über eine Resource. Damit diese Fakten von Maschinen analysiert werden können, sind für Tripel folgende formale Auflagen zu erfüllen [2]:

1. Das Subjekt ist entweder eine RDF URI Reference oder ein Blank Node (oft anonyme Resource genannt).
2. Das Prädikat ist immer eine RDF URI Reference.
3. Das Objekt ist entweder eine RDF URI Reference, ein Literal oder ein Blank Node.

RDF URI Reference

Mit RDF URI Reference wird eine Resource bestimmt, basierend auf einer URI. Manche dieser URIs oder ein führender Teil der URIs können eine Bedeutung (Semantik) besitzen. Solche Teile nennt man auch Vokabulare. Das FOAF-Vokabular[1] ist eines der bekanntesten. Für URIs werden häufig Präfixe vergeben, die kürzer sind als die eigentliche URI und damit besser lesbar (Source-Code 2.1).

Literale

Knoten, die Literale sind, stellen im Graph terminale Knoten dar und sind uninterpretierte Zeichenketten. Sie können einen Sprachtag (Source-Code 2.2, Z. 2) besitzen und getypt

[1] `http://xmlns.com/foaf/0.1/` (abgerufen 31.07.2013)

```
1       http://xmlns.com/foaf/0.1/Person -> foaf:Person
2       http://www.w3.org/1999/02/22-rdf-syntax-ns#XMLLiteral -> rdf:XMLLiteral
3       http://dbpedfdia.org/resource/ -> dbpedia
4       http://dbpedia.org/property/ -> dbpprop
5       http://www.w3.org/2001/XMLSchema\# -> xsd
```

Source-Code 2.1: Beispiele für Präfixe

```
1 "SV Werder Bremen"
2 "SV Werder Bremen"@de
3 "SV Werder Bremen"^^http://example.org/datatype:string
```

Source-Code 2.2: Beispiel für getaggte Literale

sein. Der Typ wird als URI angegeben (Source-Code 2.2, Z. 3). Es gibt keine Vorschriften wie ein Datentyp gebildet werden muss und in RDF ist nur ein Datentyp vordefiniert.[2]

Blank Nodes

Diese Knoten werden nicht mit einer URI bezeichnet und werden deswegen auch anonyme Resourcen genannt. Die Benennung ist beliebig und ist nur einem lokalen Graphen eindeutig. Mit der Einführung von Blank Nodes sind eine Reihe von Problemen bei der Verarbeitung von RDF-Graphen verbunden. Da Blank Nodes nur lokale Bezeichner sind und keine Semantik besitzen, kann ein Fakt über eine Resource auf beliebig viele Arten modelliert werden.[3] Eine Prüfung auf strukturelle Gleichheit (Isomorphie) von zwei RDF-Graphen ist ein NP-schweres Problem, sollten Blank Nodes miteinander in diesen Graphen verkettet werden.[7] Es wird noch diskutiert, wie dieses Problem umgangen werden kann. Einfach auf Blank Nodes zu verzichten, ist wohl nicht mehr möglich, da sie in zahlreichen Vokabularen wie FOAF oder VCARD bereits eingesetzt werden. Ein Vorschlag ist das Bereinigen von RDF-Graphen mittels Skolemisation. Damit ist gemeint Blank Nodes durch eindeutige Konstanten zu ersetzen.

Formal korrekter RDF-Graph

Nach diesen Ausführungen der Formalia würde der RDF-Graph, der den Sachverhalt aus Abbildung 2.3 darstellt, mit dem RDF-Standard entsprechenden Labeln aussehen wie in Abbildung 2.4.

[2]Der vordefinierte Datentyp ist rdf:XMLLiteral. Man kann in RDF aber das vordefinierte XML Schema für Datentypen benutzen: `http://www.w3.org/TR/rdf-concepts/#dfn-rdf-XMLLiteral` (abgerufen 20.05.2013).

[3]`http://www.w3.org/TR/rdf-mt/#unlabel` (abgerufen 20.05.2013)

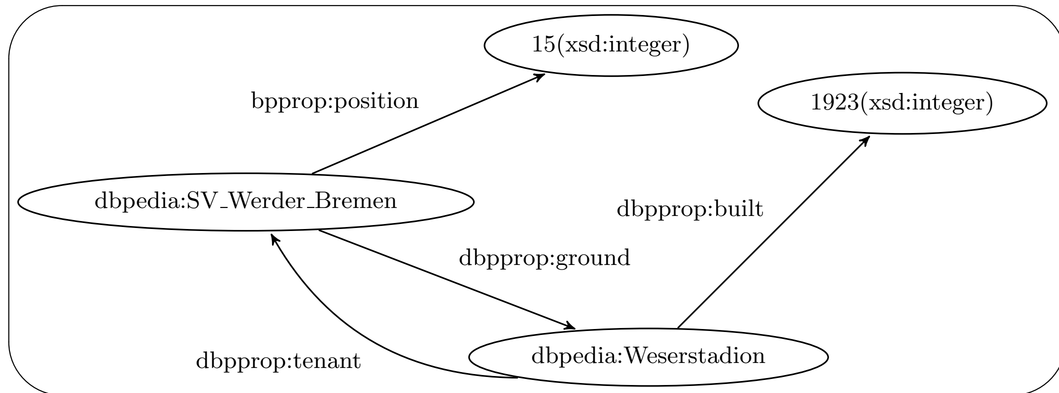


Abbildung 2.4: Ein RDF-Graph mit formal korrekten Labeln. Die Präfixe wurden zuvor im Source-Code 2.1 definiert.

Serialisierung von RDF

Interessant für diese Bachelorarbeit sind die maschinenlesbaren Darstellungsformen. Um RDF-Graphen zu serialisieren, gibt es mehrere Formate. Die wichtigsten Formate werden kurz vorgestellt:

RDF/XML Ist ein Standard, der auf XML aufgebaut wurde. Ein Vorteil ist es, dass viele Werkzeuge existieren, die XML verarbeiten können. Ein Nachteil dagegen ist, dass die Darstellung in XML sich nicht gut eignet, die Graphstruktur für Menschen erfassbar zu machen.[4]

N-Triples Ist ein zeilenbasiertes Format, welches ursprünglich zum Testen von RDF-Anwendungen entwickelt wurde.[1] Jedes Tripel wird in der Form „`s p o .`“ leerzeichensepariert notiert und durch einen Zeilenumbruch getrennt. Eine Menge dieser Tripel bildet einen RDF-Graphen.

N-Quads Baut auf dem N-Triples-Format auf und erweitert dieses um einen Kontexteintrag: „`s p o c .`“. Die Bedeutung des Kontextes ist frei definierbar und kann benutzt werden um Graphnamen zu vergeben. Somit lassen sich z. B. mehrere RDF-Graphen in einer RDF-Datei abbilden. Der Kontext darf eine Resource, ein Blank Node oder ein Literal sein.

N3 und Turtle Sind der Versuch für den Menschen besser lesbare Formate zu entwickeln. Sie ähneln dem zeilenbasierten N-Triples-Format, lassen aber z. B. Präfixdefinitionen

[4] Joshua Tauberer diskutiert dies in seinem Artikel „What Is RDF“: `http://www.xml.com/pub/a/2001/01/24/rdf.html` (abgerufen 31.07.2013)

zu (Source-Code 2.1). Die Turtle-Notation ist eine Teilmenge von N3 und somit immer valides N3.

JSON-LD Basiert auf dem verbreiteten JSON-Format, das häufig in REST-Schnittstellen zum Einsatz kommt. Es ähnelt dem N-Quad-Format, da es zu jedem Tripel den Kontext angibt.[5] Es ist in der Praxis noch nicht sehr verbreitet und wird von den bekanntesten RDF-Frameworks Jena und Sesame noch nicht unterstützt.

2.2.3 RDF Definitionen

In diesem Abschnitt werden einige Begriffe für RDF definiert, die später in der Arbeit referenziert werden.

Definition 4 (RDF-Term). *Ein RDF-Term t ist ein Element aus der Vereinigung von* $U \cup L \cup B$.

$$U = \text{Menge aller RDF-URIs}$$
$$L = \text{Menge aller Literale}$$
$$B = \text{Menge aller Blank Nodes}$$

Definition 5 (RDF-Triple). *Ein RDF-Tripel besteht aus drei RDF-Termen und wird im Folgenden als* (s, p, o) *oder* $< s, p, o >$ *notiert, wobei gilt:*

$$s \in U \cup B$$
$$p \in U$$
$$o \in U \cup L \cup B$$

Definition 6. *(Tripelmuster) Ist ein Tripel, in dem ein oder mehrere RDF-Terme durch den Platzhalter ? ersetzt werden.*

Definition 7. *(RDF-Subgraphanfrage) Ist ein Tripel aus einem Tripelmuster und einer Zahl* $n_{\in \mathbb{N}} \geq 0$, *also* $((s, p, o), n)$, *wobei n die transitive Hülle des resultierend Graphen beschränkt. Werden in dem Tripelmuster Platzhalter verwendet, wird die transitive Hülle für alle getroffenen RDF-Terme berechnet.*

[5] `http://json-ld.org/` (abgerufen 20.05.2013)

```
1      Prefix cat:<http://dbpedia.org/resource/Category>
2
3  SELECT ?title WHERE {
4    ?game <http://purl.org/dc/terms/subject> cat:First-person_shooters> .
5    ?game foaf:name ?title .
6  }
7  ORDER by ?title
```

Source-Code 2.3: Tiefe SPARQL-Abfrage, die alle Titel von Computerspielen zurückgibt, welche der Kategorie First Person Shooter zuzuordnen sind.

Definition 8. *(Gleichheit von RDF-Graphen) Zwei RDF-Graphen G und G′ sind gleich, wenn eine Funktion ϕ : RDF-Term $\mapsto$ RDF-Term existiert, für die gilt:[2]*

$$\phi(B) = B' \tag{2.1}$$

$$\phi(U) = U \tag{2.2}$$

$$\phi(L) = L \tag{2.3}$$

$$(s, p, o) \in G \iff (\phi(s), p, \phi(o)) \in G' \tag{2.4}$$

2.2.4 SPARQL

Der SPARQL-Standard lässt sich ein zwei Teile untergliedern: in die Abfragesprache und das Abfrageprotokoll. Die Abfragesprache definiert die Syntax, um RDF-Graphen zu durchsuchen oder zu beschreiben. Ein Beispiel für eine SPARQL-Abfrage ist in Source-Code 2.3 beschrieben. Das SPARQL-Abfrageprotokoll ist ein Standard, der SPARQL-Endpoints ermöglicht und damit RDF-Daten über das Internet verfügbar macht.[10]

2.2.5 Das Problem der transitiven Hülle

Eine wichtige funktionale Anforderung an den Prototypen ist die Berechnung der transitiven Hülle (Def. 3). Relationale Datenbanksysteme genause wie NoSQL-Datenbanksysteme sind üblicherweise nicht auf die Repräsentation und Durchsuchung von Graphendaten spezialisiert, prinzipiell lassen sich aber Graphen in relationalen Datenbanken und NoSQL-Datenbanken abbilden. Ein Forschungsgebiet sind Graphdatenbanken, die über effiziente Algorithmen und Datenmodelle für das Traversieren von Graphen verfügen. Potentiell muss jedoch auch der in dieser Arbeit prototypisch implementierte Triplestore einen Graph aus den RDF-Tripeln, die flach in einer NoSQL-Datenbank gespeichert werden, effizient aufbauen.

Ein Beispiel für Abfragen über zwei Knoten in einem Graphen zeigt Source-Code 2.3.

Diese Anfrage kann leicht beantwortet werden. Es müssen alle RDF-Tripel mit den passenden Prädikaten und Objekten geholt werden und die Objekte derjenigen Tripel zurückgegeben werden, die ein identisches Subjekt haben. Die Pfadlänge beträgt drei und ist schon vor der Anfrage festgelegt.

Schwieriger wird es, wenn bestimmte Relationen mit variabler Pfadlänge gesucht werden. Eine typische Anfrage wäre der Bekanntschaftsgrad einer Person, z. B. wieviele Personen kennen `Alice` direkt oder indirekt.[6] Dann ist die Länge des Pfades unbestimmt und die nächsten RDF-Tripel, die durchsucht werden, müssen ausgehend von einem Wurzelknoten erfragt werden. Enthalten diese Tripel noch keine Antwort und beinhalten diese weitere Resourcen als Objekt, muss wiederum eine neue Anfrage gestellt werden. Es entsteht eine Tiefen- oder Breitensuche auf den RDF-Daten.

Um diese Graphtraversierung zu simulieren, wurde der `limit`-Parameter in der API (Abs. 3.3) eingeführt und in der Evaluierung das Verhalten der Datenbankschemata bei solchen Abfragen untersucht.

2.3 Google App Engine

2.3.1 Einführung

Die Google App Engine (GAE) ist eine kommerzielle Entwicklerplattform und Laufzeitumgebung für Webanwendungen, die Cloud-Computing ermöglicht.

Definition 9. *The NIST Definition of Cloud Computing Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.[9]*

Nach der Definition des National Institute of Standards and Technology (NIST) gilt GAE als Platform as a Service(PaaS) und ist eine Ausprägung des Service Modells (Def. 9). Damit ist eine Plattform gemeint, die es Anwendern ermöglicht, ihre Applikation auf Infrastruktur von Drittanbietern laufen zu lassen. Mit Infrastruktur ist nicht nur Hardware gemeint, sondern es werden oftmals Schnittstellen zu verschiedenen Services, Libraries und Werkzeuge für das Monitoring und Administrieren der Applikation bereit

[6]Beispiele für solche Anfragen findet im Virtuoso Open-Source Wiki: `http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VirtsubClassOfOrientedSubsumptionTransitiveOptions` (abgerufen am 2.07.13)

gestellt.[9] So existieren bei GAE Schnittstellen zu Google Cloude Storage, XMPP (der Google-Chat) oder OAuth[7].

Kommerzielle Anbieter einer PaaS-Infrastruktur führen zahlreiche Begrenzungen für die Applikationen ein, wie z. B. Begrenzung des Speicherverbrauchs, der Laufzeit von bestimmten Operation oder des Trafficvolumens und weichen diese Limits gegen Bezahlung auf. Das ist das typische Geschäftsmodell von PaaS-Anbietern.

2.3.2 Sicherheitslimitierungen und abrechenbare Resourcen

GAE unterteilt die Limitierung von Rechenresourcen in Sicherheitslimitierungen und abrechenbare Resourcen.[8] Die Sicherheitslimitierungen sollen gewährleisten, dass jede Applikation auf GAE zuverlässig läuft und nicht eine Applikation dauerhaft zu viele Hardwareresourcen beansprucht. Eine wichtige Eigenschaft der Sicherheitslimitierungen ist, dass sie nicht aufhebar sind oder umgegangen werden können, auch nicht gegen Bezahlung. Die kritischste Sicherheitslimitierung für den prototypischen Triplestore in dieser Arbeit ist die 60-Sekunden-Frist, innerhalb derer jeder HTTP-Request abgeschlossen sein muss. Für die meisten Operationen auf dem Triplestore müssen die Antwortzeiten wesentlich kürzer sein, um praktischen Nutzen zu gewährleisten. Niemand möchte über 60 Sekunden auf eine Antwort warten. Es gibt aber Operationen wie das Zählen (Abs. 3.7) aller gespeicherten Triple, die unter Umständen länger dauern können als 60 Sekunden. Das liegt an der Datenbank Management Software (DBMS), die von GAE zum Speichern von Daten benutzt wird. Zu den kritischsten abrechenbaren Resourcen gehört die Anzahl der Schreib- und Lesezugriffe auf die Datenbank. Da diese Arbeit untersucht, ob sich GAE für einen hochskalierenden Triplestore eignet, ist das tägliche Limit für die Datenbankoperationen mit dem Free Account niedrig (Tabelle 2.1).

2.3.3 App Engine Datastore

Für die Persistierung von Daten stellt GAE den Google App Engine Datastore (AED) zur Verfügung. Dieser basiert auf Googles Bigtable, einem verteilten Datenbanksystem, dass für große Datenmengen vorgesehen ist.

Die klassischen Datenbanksysteme folgten in der Regel dem relationalen Datenbankmodell. Eine Relation ist in diesem Zusammenhang eine Menge von Tupeln, wobei

---

[7] `https://tools.ietf.org/html/rfc6749` (abgerufen 27.07.2013)
[8] `https://developers.google.com/appengine/docs/quotas#Safety_Quotas_and_Billable_Quotas` (abgerufen 27.07.2013)
[9] `https://developers.google.com/appengine/docs/quotas#Safety_Quotas_and_Billable_Resources` (abgerufen 27.07.2013)

Resource	Free Account Limit	bei aktivierter Bezahlung
Speicherplatz	1 GB	1 GB kostenlos, dann 1GB/0,24 $
Indizes	200	200
Schreiboperationen	50000	unbegrenzt
Leseoperationen	50000	unbegrenzt

Tabelle 2.1: GAE Datenbanklimitierungen, die Zähler der genutzten Resourcen werden für den Free-Account alle 24 Stunden zurückgesetzt.[9]

normalerweise ein Tupel ein Objekt oder einen Fakt repräsentiert und als eine Zeile in einer Tabelle gespeichert wird. Die Tupel haben eine Menge an Attributen, die als Spalten in der Tupeltabelle angesehen werden können. Die Mächtigkeit der Attributmenge aller Tupel einer Tabelle ist gleich und die Werte der Attribute können Verweise auf andere Tupel in der gleichen oder in anderen Tabellen sein.

AED gehört zu den NoSql-Datenbanken, die mit dem Paradigma der relationalen Datenbanken brechen. Dabei meint NoSql nicht notwendigerweise den Verzicht auf SQL, sondern NoSql-Datenbanken besitzen mehrere charakteristische Eigenschaften, die sie von relationalen Datenbanksystemen unterscheiden:

Kein Schema auf den gespeicherten Daten Die Anzahl der Attribute und die Typen der Attribute eines gespeicherten Datensatzes sind nicht festgelegt und können variieren.

Spaltenorientiert Relationale Datenbanken speichern die Daten in der Regel reihenorientiert. Das bedeutet, dass für jeden Datensatz alle Attribute hintereinander notiert werden. Bei Spaltenorientierung wird erst ein Attribut aller Datensätze aufgelistet, dann das nächste Attribut aller Datensätze usw.

Horizontal skalierend (Distributed Storage) Dies meint in Bezug auf NoSql-Datenbanken, dass eine Datenbank bei höheren Anforderungen nicht auf leistungsfähigere Hardware umzieht, sondern durch Ergänzung mit neuen Servern den höheren Anforderungen gerecht wird und so die Last auf mehrere Rechner verteilt wird (Load Balancing).

Keine Joins und Abfragenplanung Klassische relationale Datenbankmanagementsysteme (RDBMS) führen Joins von Tabellen durch, was bei großen Tabellen zu Laufzeitproblemen führen kann, wenn z. B. der Query-Planer sich verschätzt.

Geringes oder einfaches Transaktionmanagment Transaktionen sind im Datenbankkontext eine atomare Kette von Operationen. Schlägt eine Operation der Kette fehl, werden alle schon vorgenommen Änderungen wieder zurückgenommen (Rollback). Alle Operation in einer Transaktion werden also komplett oder gar nicht durchgeführt. Das gewährleistet, dass die Daten in einem konsistenten Zustand bleiben. NoSql-Datenbanken sind auch Datenbanken und verfügen deswegen über ein Transaktionsmanagement, oft aber wird Transaktionsmanagement (Atomicity, Consistency, Isolation, Durability) nur auf eine einzelne Zeile oder Spalte angewendet.

Von den oben genannten Charakteristika treffen auf AED zu, dass die Daten schemalos gespeichert werden, ein einfaches Transaktionsmanagement verwendet wird und AED horizontal skaliert sowie keine komplexe Abfragenplanung betrieben wird. Eine SQL-Schnittstelle wird nicht angeboten, dafür eine an SQL angelehnte Abfragesprache GQL[10].

Entitäten

Der App Engine Datastore speichert Objekte schemalos. Gespeicherte Objekte werden Entitäten genannt. Jede Entitität muss einen Schlüssel besitzen. Wird kein Schlüssel vergeben, wird ein zufälliger Schlüssel vom App Engine Datastore zugewiesen. Der Schlüssel kann auch Hierarchien zwischen den Entitäten abbilden, diesen werden Entitätengruppen genannt (Source-Code 2.4). Schreiboperationen auf Entitätengruppen sind atomar. Um Entitäten aus dem App Engine Datastore zu lesen, muss man entweder den Schlüssel kennen oder Indizes über die Eigenschaften (Properties) der Entitäten anlegen. Jede Entität, die in den App Engine Datastore geschrieben wird, löst zwei Schreiboperationen aus:

1. Die Entität wird in einer Tabelle mit den Spalten Schlüssel und Entität eingetragen.
2. Der Schlüssel der Entität wird in eine Tabelle mit den Spalten Applikationsbezeichner (Name der GAE-Anwendung) und Schlüssel eingetragen.

Enititäten können also direkt aus der Entitätentabelle geholt werden, wenn der Schlüssel vorher bekannt ist. Da Bigtable wie eine multidimensionale Zuordnung[11] agiert, ist dies der schnellste Zugriff auf Entitäten.

[10] `https://developers.google.com/appengine/docs/python/datastore/gqlreference` (abgerufen 31.07.2013)

[11] Schlüssel in Bigtable setzten sich aus Zeilenschlüssel, Spaltenschlüssel und einen Zeitstempel zusammen und werden auf eine uninterpretierte Folge von Zeichen geordnet [3]:
`(row:string, column:string, time:int64)` $\rightarrow$ `string`

```
1 appid/Grandparent:Harald
2 appid/Grandparent:Harald/Parent:Benjamin
3 appid/Grandparent:Harald/Parent:Benjamin/Child:Ephraim
4 appid/Grandparent:Harald/Parent:Benjamin/Child:Joshua
5 appid/GrandParent:Elisabeth/Parent:Gaby/Child:Benjamin
6 appid/Parent:Lisa
```

Source-Code 2.4: Entitäten Schlüssel

Entititätengruppen

Jede Entität gehört einer Entitätengruppe an. Hat eine Entität keine Elternentität ist sie eine Stammentität. Eine Stammentität ohne Nachfahren bildet eine einelementige Entitätengruppe (Source-Code 2.4, Z. 6). Eine Entitätengruppe wird über zusammengesetzte Entitätenschlüssel definiert, wobei gilt: Zwei Entitäten mit identischem Ancestor-Path gehören einer Entitätengruppe an.

Entitätenschlüssel

Jeder Entitätenschlüssel besteht aus drei Teilen:

1. Dem Applikationsbezeichner: der Name der GAE-Anwendung.
2. Der ID, sie besteht aus zwei Teilen:
   a) Dem Typen der Entität.
   b) Einem Namen oder einer Zahl.
3. Dem Pfad (Ancestor-Path): Der Ancestor-Path bildet sich aus einer Verkettung von IDs. Damit können Hierarchien von Entitäten gebildet werden. Zwei identische IDs können niemals den gleichen Ancestor-Path besitzen. So referenzieren in Source-Code 2.4 die Zeilen 2 und 3 unterschiedliche Entitäten. Die Schlüssel dürfen maximal 500 Character lang sein. Folglich ist die Länge des Ancestor-Paths beschränkt.

Indizes

Mit Indizes können Entitäten im AED gesucht und gefiltert werden. Dafür wird ein Präfixscan oder Intervallscan auf den Indextabellen durchgeführt. Im App Engine Datastore wird für jede Eigenschaft einer Entität, die indiziert werden soll, zwei Einträge in zwei Tabellen vorgenommen. Damit fallen pro Index zwei Schreiboperationen an.

1. Der Schlüssel der Entität, der Name der Eigenschaft und der Wert der Eigenschaft wird in eine nach dem Namen der Eigenschaft aufsteigend sortierte Tabelle eingetragen.

2. Der Schlüssel der Entität, der Name der Eigenschaft und der Wert der Eigenschaft wird in eine nach dem Namen der Eigenschaft absteigend sortierte Tabelle eingetragen.

Entitäten können nur über Eigenschaften gefunden werden, wenn eine Eigenschaften zuvor in den Index aufgenommen wurde. Generell gelten Indexscans als deutlich langsamer als Zugriffe über Entititätenschlüssel. Da Entitätentabellen niemals gescannt werden, sondern nur die Index- und Schlüsseltabellen, ist es auch nicht möglich Entitäten im Appengine-Datastore auf Substrings zu filtern.

Operatoren

Folgende Operatoren können auf Eigenschaften von Entitäten angewendet werden: $>, <, !=, =, \text{IN}$. Interessant ist der IN-Operator. Dieser bekommt eine Liste von Werten und einen Eigenschaftsnamen und prüft für jede Entitität mit diesen Eigenschaftsnamen, ob der Eigenschaftswert in der Liste ist. Das wäre eigentlich eine Oder-Operation auf Eigenschaftswerten, aber es wird sequentiell für jeden Wert in der Liste eine Abfrage mit dem Gleichheitsoperator durchgeführt. Für eine Liste mit n-Wertenwird GAE n-Abfragen und n- Indexscans durchführen. Bei großen Indextabellen und großen Wertelisten ist dieser Operator sehr langsam.

Präfixscan und Rangescan

Für die Gleichheitsoperatoren werden Präfixscans durchgeführt, für alle Ungleichheitsoperatoren hingegen wird ein Intervall berechnet, dass dann gescannt wird (Abb. 2.5). Diese Scans werden entweder auf Indextabellen oder auf der Schlüsseltabelle vorgenommen. Die Scans werden niemals auf der Entitätentabelle durchgeführt. GAE speichert die Entitäten aller GAE-Applikationen in einer einzigen Tabelle als uninterpretierte Byte-Folge, weshalb Scans auf einzelne Eigenschaften von Entitäten nicht performant wäre, angesichts der Größe der Tabelle und der notwendigen Deserialisierung jeder Entität.

[12] „Under the covers of the App Engine Datastore“ `http://snarfed.org/datastore_talk.html` (abgerufen 30.07.2013)

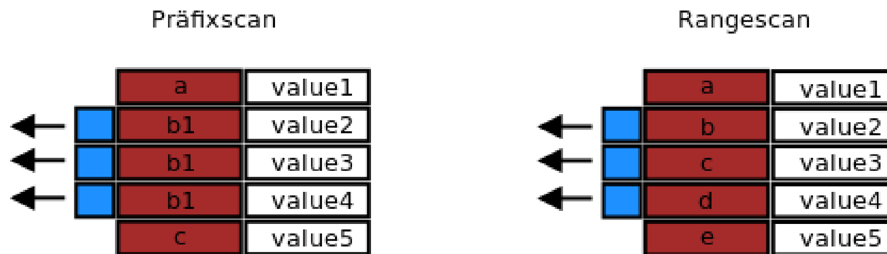


Abbildung 2.5: Präfix- und Rangescans[12]

Low-Level-API

Die Low-Level-Api bildet die Schnittstelle zu dem App Engine Datastore und umfasst vier Methoden:[13]

1. `put()`
2. `get()`
3. `delete()`
4. `query()`

Alle Objekte, die im App Engine Datastore gespeichert werden, haben den Typ `Entity`. Die Methoden `put()`, `get()`, `delete()` können als Batch-Operationen durchgeführt werden, was das Speichern oder Abrufen von mehreren Entititäten wesentlich beschleunigt. Die Methode `query()` kann nicht als Batch-Operation ausgeführt werden. Möchte man mehrere Abfragen erstellen, bietet die Low-Level-API asynchronen Zugriff auf den App Engine Datastore. Maximal 10 asynchrone Zugriffe sind gleichzeitig erlaubt.

Transaktionen im App Engine Datastore

Jeder Schreibzugriff auf einer Entität ist atomar. Deswegen können innerhalb einer Batch-Operationen Änderungen einzelner Entitäten fehlschlagen und Änderungen anderer Entitäten wiederum erfolgreich sein. Wird aber eine Batchoperation abgebrochen, beispielsweise aufgrund eines Timeouts, ist es umständlich zu verifzieren, welche Entitäten schon persistiert wurden. Um dieses Problem einzuschränken, können Transaktionen auf Entitätengruppen durchgeführt werden.

Bigtable verteilt die Entitätentabelle auf hunderte Server und die Entitäten werden nach ihrem Schlüssel lexikalisch sortiert, was ermöglicht, dass Entitäten innerhalb einer

[13] `https://developers.google.com/appengine/docs/java/javadoc/com/google/appengine/api/datastore/package-summary` (abgerufen 31.07.2013)

Gruppe in physicher Nähe gespeichert werden können (auf einem Server und in einem Tablet). Auf dieser Grundlage werden Transaktionen auf einer Entitätengruppe realisiert. Wird innerhalb einer Transaktion versucht Entitäten verschiedener Entitätengruppen zu verändern, bleiben die Daten im ursprünglichen Zustand und es wird eine Exception zurückgegeben. Bigtable muss nicht Schreibzugriffe auf andere Server und Tablets locken und bleibt somit reaktiv für Schreibzugriffe.

Zusätzlich sind Cross-Group-Transaktionen erlaubt. Cross-Group-Transaktionen sind Transaktionen auf anderen Entitätengruppen innerhalb einer Transaktion, sind aber auf fünf Transaktionen beschränkt.

Memcache

Der Memcache beschleunigt Abfragen des App Engine Datastore. Es können aber im Gegensatz zum App Engine Datastore beliebige Objekte, die serialisierbar sind, im Memcache gespeichert werden. Der Memcache agiert dabei als assoziativer Speicher. Operationen auf dem Memcache sind nicht atomar und es werden keinerlei Garantien zur Speicherdauer von Objekten gegeben. Zugriffe auf die Memcache-API gehören zu den abrechenbaren Resourcen.

2.3.4 Instanzen

GAE erzeugt dynamisch Instanzen einer Applikation. Eine Applikation kann viele (bis zu 500) Instanzen haben. Instanzen werden von GAE initial für eine Applikation erzeugt, wenn ein Request abgesetzt wird, der dann als ein Loading-Request bezeichnet wird. Weiterhin können Instanzen im Vorhinein durch Warmup-Request geladen werden oder mehrere Instanzen in den Wartemodus versetzt werden, damit keine Loading-Requests entstehen. Loading-Requests sind, je nach Anzahl der Resourcen und der Bibliotheken die geladen werden müssen, langsam.

2.3.5 GAE Laufzeitumgebungen

Die Laufzeitumgebung von GAE bedient drei Programmiersprachen: Java, Python und Go. Die Unterstützung für die von Google entwickelte Sprache Go trägt aber noch Experimentierstatus. In dieser Arbeit wurde die Java-Laufzeitumgebung benutzt. Es gibt folgende Sicherheitslimitierungen für die Java-Laufzeitumgebung (JRE):

1. Schreiben auf das Dateisystem ist nicht erlaubt.

2. Keine Sockets können geöffnet werden. Kommunikation ist nur den Anwendungsprotokollen des TCP/IP-Stacks möglich, wie z. B. HTTP oder XMPP.
3. Es können keine Threads erzeugt werden. Die Schnittstellen zum App Engine Datastore bieten allerdings asynchrone Methoden an.

Es gibt eine JRE-Whitelist für alle Java-Klassen, die verwendet werden dürfen.[14] Der Einstiegspunkt für jede GAE-Applikation ist eine Methode des abstrakten Java HTTP-Servlets. Mit dem Aufruf einer dieser Methoden beginnt das Timing, welches überwacht, dass die Methode eine Antwort innerhalb von 60 Sekunden liefert.

2.4 Weitere Triplestores, die nicht auf relationalen Datenbanksystemen aufsetzen

Aktuelle Triplestores für große RDF-Graphen speichern RDF-Tripel im Umfang von mehreren Millionen bis hin zum unteren zweistelligen Milliardenbereich[15]. Die höchste Performanz bietet nach eigenen Angaben OWLIM[16], und wird auch außerhalb des akademischen Kontextes eingesetzt.[17] OWLIM benutzt für die Persistierung kein Datenbanksystem sondern Amazon Elastic Block (EBS), dass als ein performantes, horizontal skalierendes Dateisystem agiert, nicht als Datenbanksystem.

Ein weiterer Triplestore, der einen cloudbasierten Ansatz verfolgt, ist das Open-Source-Projekt CumulusRDF[18][6]. Dieser Triplestore benutzt das verteilte Datenbanksystem Apache Cassandra, einen Open-Source-Klon von Googles Bigtable. Getestet wurde mit den Daten von Dbpedia mit circa 120 Millionen RDF-Tripeln. Die Entwickler kommen zu dem Schluß, dass ihr Triplestore mit den aktuellen Lösungen mithalten kann. CumulusRDF bietet momentan aber CRUD-Operationen und keine SPARQL-Engine um komplexe Abfragen zu ermöglichen.

Zuletzt erwähnt sei eine prototypische Implementierung eines Triplestores mit MongoDB, einer semistrukturierten, dokumentorientierten NoSql-Datenbank. Dort wird ebenfalls mit den Daten von Dbpedia gegen eine Installation von Virtuoso getestet. Die Testgraphen haben eine maximale Größe von 238965 Tripeln. Im Fazit wird MongoDB als eher ungeeignet für große RDF-Graphen erachtet.[8]

[14] `https://developers.google.com/appengine/docs/java/jrewhitelist` (abgerufen 18.05.2013)
[15] Ein Ausreißer stellt AllegroGraph da. In einer AllegroGraph-Instanz sollen über eine Billiarde Triple gespeichert worden sein. `http://www.w3.org/wiki/LargeTripleStores` (abgerufen 18.05.2013)
[16] `http://www.ontotext.com/owlim` (abgerufen 17.05.2013)
[17] Die „Word Cup 2010 Website“ der BBC setzte OWLIM ein. `http://www.bbc.co.uk/blogs/bbcinternet/2010/07/bbc_world_cup_2010_dynamic_sem.html` (abgerufen 17.05.2013)
[18] http://code.google.com/p/cumulusrdf/ (abgerufen 17.05.2013)

2.5 REST

REST wird für die Kommunikation zwischen Frontend und Applikations-Server eingesetzt (Abb. 1.1). Die REST-Systemarchitektur wurde von Roy Thomas Fielding [4] entwickelt und ist ein Programmierparadigma für Web-Services. Es ist ursprünglich bei der Spezifizierung von HTTP entstanden. REST ist nicht grundsätzlich auf HTTP beschränkt. Mittels REST sollen Resourcen im Internet verfügbar gemacht werden. REST besteht aus mehreren Prinzipien:

1. REST ist zustandslos. Das bedingt, dass für jede Operation der Client alle nötigen Parameter mitliefern muss.

2. Resourcen sind eindeutig adressierbar. In der Praxis wird dies mittels URIs realisiert.

3. Die Repräsentation von Resourcen ist variabel. In HTTP ist ein Header-Feld spezifiziert, mit dem Clients die bevorzugte Repräsentation angeben können.[19]

4. Es gibt eine definierte Menge an Operationen zur Manipulation von Resourcen. Da REST in der Praxis immer auf HTTP aufsetzt, sind diese Operationen durch die HTTP-Methoden GET, PUT, DELETE, POST, HEAD usw. festgelegt. Diese Operationensnamen werden auch Verben genannt.

Die Vorteile von REST gegenüber ähnlichen Ansätzen wie Simple Object Access Protocol (SOAP) ist der geringe Verbrauch an Bandbreite, da aufgrund der Zustandslosigkeit viel mit Cachingverfahren optimiert wird. Außerdem ist HTTP sehr verbreitet und es existieren auf allen gängigen Betriebsystemen und Plattformen Implementierungen des Protokolls.

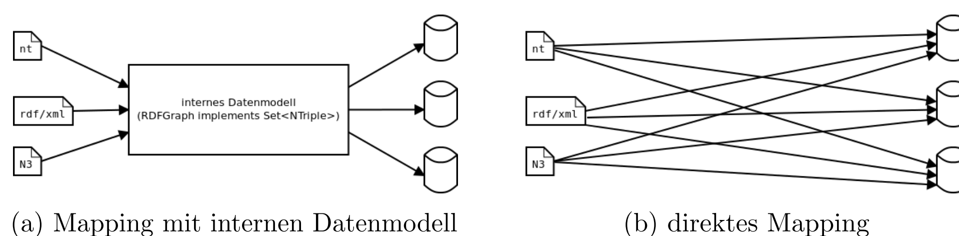
[19] Dieses Header-Feld hat den Schlüssel „accept“ und die Werte sind in der Regel MIME-Types (`http://www.iana.org/assignments/media-types` (abgerufen 18.05.2013)). Es ist auch möglich, dieses Feld mit einer Wildcard zu belegen.

3 Implementierung

Eingangs erfolgt eine kleine Diskussion wie der Prototyp aufgebaut wurde und warum bestimmte Abstraktionsschichten für die Implementierung gewählt wurden.

Zunächst müssen die RDF-Daten in ein internes Datenmodell umgewandelt werden. Dieser Schritt wurde gewählt, um unabhängig zu sein von anderen Frameworks und um das Abbilden (Mapping) auf unterschiedliche Repräsentationen von RDF-Daten flexibel zu halten. Dafür wurde eine Klasse NTriple eingeführt. Diese hat drei Felder subject, predicate und object und repräsentiert damit eine Zeile des NT-Formats. Ein RDF-Graph wird intern als ein Set<NTriple> dargestellt, was der Definition eines RDF-Graphen nach den Spezifikation des W3C sehr nahe kommt.[1] Zusätzlich wird der RDF-Graph (Set<NTriple>) im Prototypen durch ein Interface gekapselt um noch weitere Methoden (z.B. Äquivalenztests) zuzusichern. Mit diesem internen Datenmodell arbeitet der Application-Server (Abb. 1.1) und es bildet somit die Schnittstelle zwischen der Kommunikation mit dem Frontend und der jeweiligen Ausprägung eines Datenbankschemas. Ein großer Vorteil des internen Datenmodells ist das leichte Hinzufügen von neuen Eingabeformaten und Datenbankschemata. Für jedes neue Format oder Schema muss nur genau ein Mapping implementiert werden. Der Aufwand bei nachträglichen Änderungen ist gering (Abb. 3.1).

[1] http://www.w3.org/TR/rdf11-concepts/#section-rdf-graph (abgerufen 30.06.13)



(a) Mapping mit internen Datenmodell

(b) direktes Mapping

Abbildung 3.1: Mapping zwischen internem Datenmodell und Eingabeformaten sowie Datenbankschemata. Die Anzahl der Mappings mit internem Datenmodell ist geringer und lässt sich allgemein ausdrücken als: $m+n \leq m \cdot n$. Dieses Konzept findet man häufig im Compilerbau und wird Zwischen-Code genannt. Dies soll die Portabilität von Quellsprachen erhöhen.

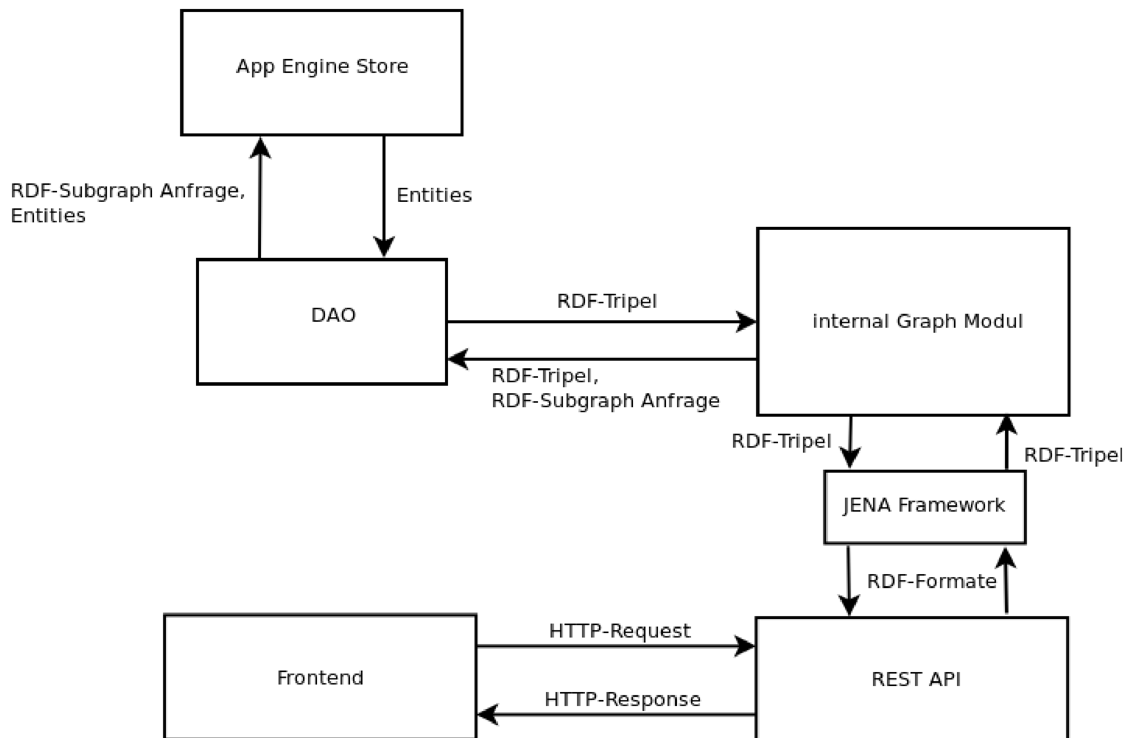


Abbildung 3.2: Detailierter Überblick über den Prototypen

Das interne Datenmodell befindet sich auf der Abbildung 3.2 in dem `Internal Graph Modul`. Das Mapping von RDF-Tripeln auf das jeweilige Datenbankschema wird von einer Ausprägung des Data-Access-Object-Interfaces (DAO)[2] übernommen. Für jedes neue Datenbankschema muss also das DAO-Interface implementiert werden. Einige Operationen sind in einer abstrakten DAO-Klasse zusammengefasst, wie z. B. die `count`-Methode (genauer behandelt in Abs. 3.7). Durch die Einführung des DAO-Interface kann jedes beliebige Datenbanksystem eingesetzt werden, eine Implementierung für das relationale Datenbanksystem Postgres[3] wäre denkbar, um Benchmark-Tests durchzuführen usw.

Die Entscheidung eine REST-API zu entwickeln wurde getroffen, um eine hohe Entkopplung vom Frontend zu erreichen, so dass beispielsweise eine spätere Anbindung an eine SPARQL-Engine sehr wenig Einschränkungen respektive Programmiersprache oder Technologie hat.

[2]DAO steht für Data-Access-Object und bildet eine Abstraktionsschicht zwischen Datenbank und Anwendungslogik

[3]`http://www.postgresql.org/` (abgerufen 1.07.2013)

3.1 Werkzeuge

Für die Entwicklung des Prototypen kommt die Source Code Management Software (SCM) Git zum Einsatz, die ursprünglich für die Entwicklung des Linux-Kernels programmiert wurde. Damit lässt sich der Source-Code versionieren und ältere Zustände der Software wiederherstellen. Git bietet zudem Funktionen für die kollaborative Entwicklung.

Zusätzlich wird das Build-Tool Maven[4] verwendet. Damit lassen sich über eine Konfigurationsdatei Bibliotheken und Frameworks einbinden. Zusätzlich werden Tests und Deploymentprozesse automatisiert. Google stellt ein eigenes Maven-Plugin[5] bereit. Durch die Verwendung von Maven ist die Entwicklung des Prototypen weitesgehend unabhängig von der Wahl der Integrated Development Environment (IDE).

3.2 Frameworks

3.2.1 Apache Jena

Das Apache Jena Framework[6] ist ein auf Java basierendes Toolkit für die Erstellung und Verarbeitung von RDF-Daten. Es kann relationale Datenbanken wie Postgres als Triplestore benutzen oder die RDF-Daten auf dem Dateisystem speichern. Apache Jena unterstützt SPARQL und kann die wichtigsten RDF-Formate parsen und schreiben.

3.2.2 Jersey

Jersey ist eine Referenzimplementierung der Java API for RESTful Web Service (JAX-RS) Spezifikation, die das Entwickeln von REST-Webservices erleichtert. Mittels Annotationen können Parameter vom HTTP-Servlet abgefragt werden und HTTP-Methoden deklarativ entwickelt werden (Source-Code 3.1). Damit Jersey auf GAE eingesetzt werden kann, muss das automatische Generieren der WADL-Datei deaktiviert werden.

3.2.3 Objectify

Das Objectify-Framework bildet eine Abstraktion der Low-Level-API und ist auf GAE Anwendungen zugeschnitten. Das Framework Apache Java Data Objects (JDO) dagegen ist die Implementierung des Java Specification Request 12 (JSR 12)[7], eine Spezifikation zur Persistierung von Java Objekten. JDO verfolgt damit einen generischen Ansatz.

[4] `http://maven.apache.org/` (abgerufen 31.07.2013)
[5] `https://code.google.com/p/maven-gae-plugin/` (abgerufen 31.07.2013)
[6] `http://jena.apache.org/` (abgerufen 31.07.2013)
[7] `http://www.jcp.org/en/jsr/detail?id=12` (abgerufen 30.07.2013)

```
@PUT
@Path("/triple")
public void triple(@QueryParam("subject") String subject,
                   @QueryParam("predicate") String predicate,
                   @QueryParam("object") String object) {
    access.persistTriple(new NTriple(subject, predicate, object));
}
```

Source-Code 3.1: Beispiel für eine Implementierung eines HTTP-PUT-Request mit dem Jersey-Framework.

Einige Eigenschaften des App Engine Datastore sind aber nicht in JDO spezifiziert und werden deswegen nicht unterstützt.[8] Dies ist der Hauptgrund Objectify zu benutzen. Weitere Vorteile gegenüber der direkten Nutzung der Low-Level-API:

- **Plain Old Java Object (POJO)[9] mit Objectify:** Die Low-Level-API ist beschränkt auf Objekte vom Typ `Entity` und `Key`. In Objectify werden POJOs durch Anreicherung mit Annotationen auf `Entity` abgebildet. Dadurch wird Typsicherheit erreicht und der Source-Code wird schlanker und besser lesbar. Der Beispiel-Code 3.2 zeigt ein einfaches Java-Datenmodell, das eine Relation zwischen Club und Fußballspieler modelliert. Das Abbilden auf ein Entity-Objekt (Z. 28-32) entfällt.
- **Caching:** Objectify benutzt zwei Arten von Caching:
  1. **Session-Cache:** Für jedes erzeugtes Objectify-Objekt werden Entititäten gecached. Dieser Cache überdauert in der Regel nur einen HTTP-Request.
  2. **Memcache:** Wird von Objectify automatisch für alle Objekte, die mit `@Cache` annotiert sind, verwendet.

Erwähnt sei noch das TWIG-Framework[10], das ähnlich wie Objectify eine spezielle Abstraktion der Low-Level-API darstellt. Der einzige Vorteil des Twig-Frameworks, asynchrone Query-Operationen, ist mit dem letzten Release von Objectify nicht mehr gegeben.[11]

[8] Unterstützt wird nicht der Memcache oder das automatische Laden von Vorgängerobjekten aus dem Ancestor-Path.

[9] POJOs sind einfache Java-Datenobjekte, die kein Gebrauch von Veerbung, Polymorphie, Annotationen usw. machen. Das Konzept von POJOs wird nicht so streng gehandhabt, so sind die POJOs in Objectify mit Annotationen angereichert.

[10] `https://code.google.com/p/twig-persist/` (abgerufen 18.05.2013)

[11] `https://code.google.com/p/objectify-appengine/wiki/IntroductionToObjectify#Asynchronous_Calls` (abgerufen 30.07.2013)

```
@Entity
class Player
{
    @Id
    String name;

    @Parent;
    Key<Club> club;
}

@Entity
class Club
{
    @Id
    String name;
}

Player player = new Player();
player.name = "Claudio Pizarro";

// objectify code
player.club = Key.create(Club.class, "Werder Bremen");

// persist with objectify
ofy.save(player);

// low level api
Entity playerEntity = new Entity("Player");
playerEntity.setProperty("name", player.name);
Key clubKey = new KeyFactory.Builder("Club", "Werder Bremen")
    .addChild("Player", player.name);
playerEntity.setProperty("club", clubKey);

// persist with low level api
datatstore.put(playerEntity);
```

Source-Code 3.2: POJOs in Objectify

3.3 API

Der Prototyp bietet eine REST-Schnittstelle an (Abb. 3.2). Diese umfasst Funktionen zum Persistieren und Abfragen von RDF-Triplen. Die Methoden werden zusammen mit einem Beispiel für HTTP-Requests vorgestellt. Die Variable `<triple>` in den HTTP-Aufrufen werden vorab in BNF erläutert:

$$
\begin{aligned}
\langle \text{triple} \rangle &\models \text{subject}{=}\langle \text{uri} \rangle \;\&\; \text{predicate}{=}\langle \text{uri} \rangle \;\&\; \text{object}{=}\langle \text{object} \rangle \\
\langle \text{object} \rangle &\models \langle \text{literal} \rangle \mid \langle \text{uri} \rangle
\end{aligned}
$$

Es sind auch unterspezifzierte Tripel möglich. Damit ergeben sich fünf weitere Ausdrücke für Triple:

$$
\begin{aligned}
\langle \text{s,?,?} \rangle &\models \text{subject}{=}\langle \text{uri} \rangle \\
\langle \text{?,p,?} \rangle &\models \text{predicate}{=}\langle \text{uri} \rangle \\
\langle \text{?,?,o} \rangle &\models \text{object}{=}\langle \text{object} \rangle \\
\langle \text{s,p,?} \rangle &\models \text{subject}{=}\langle \text{uri} \rangle \;\&\; \text{predicate}{=}\langle \text{uri} \rangle \\
\langle \text{?,p,o} \rangle &\models \text{predicate}{=}\langle \text{uri} \rangle \;\&\; \text{object}{=}\langle \text{object} \rangle
\end{aligned}
$$

3.3.1 Persistierung und Löschen von Tripeln

- **insert** (`PUT <project.url>/pdgs/triple?<tripel>`)
  Fügt ein RDF-Tripel in den Triplestore ein.

- **insertBulk** (`POST <project.url>/pdgs/triples`)
  Fügt mehrere RDF-Tripel innerhalb eines HTTP-Requests hinzu. Es werden mehrere RDF-Formate unterstützt (nt, rdf, tt, n3). Der Client muss den passenden MIME-Type setzen.

- **delete** (`DELETE <project.url>/pdgs/triple?<tripel>`)
  Löscht ein Triple.

- **deleteBulk** (`DELETE <project.url>/pdgs/triples`)
  Löscht mehrere RDF-Tripel innerhalb eines HTTP-Requests. Es werden mehrere RDF-Formate unterstützt (nt, rdf, tt, n3). Der Client muss den passenden MIME-Type setzen.

3.3.2 Abfragen von Tripeln

- **count** (GET <project.url>/rest/api/count)
  Zählt alle im Triplestore gespeicherten Tripel.

- **get** (GET <project.url>/rest/api/triple?<tripel>)
  Gibt genau ein Tripel zurück, wenn es im Triplestore gespeichert ist. Wenn nicht, ist die Antwort leer.

- **getDeep** (GET <project.url>/rest/api/triple?<tripel>&limit=undefined
  Berechnet die transitive Hülle des Objekts des abgefragten Tripels (Kapitel 2.1). Wie die vorige get-Methode liefert dieser Aufruf nur ein Tripel zurück, wenn es im Triplestore gespeichert ist. Zusätzlich wird das Objekt geprüft, ob es eine Resource ist und ob diese im Triplestore gespeichert ist. Ist dies der Fall, wird die Methode getByValueDeep mit dem Objekt als Subjekt aufgerufen. Alle gefundenen Tripel werden als Antwort zurückgegeben.

- **getLimited** (GET <project.url>/rest/api/triple?<tripel>&limit=3
  Berechnet die eingeschränkte transitive Hülle (Kapitel 2.1). Ähnlich wie getDeep werden die Objekte geprüfte, ob es auflösbare Resourcen sind. Ist dies der Fall, wird getByValueLimited aufgerufen, mit dem Objekt als Subjekt.

- **getByValue** (GET <project.url>/rest/api/triple?<s,?,?>)
  Sucht alle Tripel die s als Subjekt haben, die Antwort kann also mehr als ein Tripel enthalten. Die Methode kann auf alle Arten von unterspezifizierten Tripeln angewendet werden.

- **getByValueDeep**
  (GET <project.url>/rest/api/triple?<s,?,?>&limit=undefined)
  Berechnet die transitive Hülle des Objekt des abgefragten Tripels (Kapitel 2.1). Sucht alle Tripel die s als Subjekt haben. Alle Objekte der getroffenen Tripel werden geprüft, ob sie Resourcen sind und ob diese im Triplestore enthalten sind. Ist dies der Fall, wird getByValueDeep mit diesen Objekten als Subjekte ausgeführt. Alle gefundenen Tripel werden als Antwort zurückgegeben. Die Methode getByValueLimited kann für alle unterspezifizierten Tripel benutzt werden.

- **getByValueLimited** (GET <project.url>/rest/api/triple?<s,?,?>&limit=3)
  Berechnet die eingeschränkte transitive Hülle des Objekts des abgefragten Tripels (Kapitel 2.1). Dafür werden alle Tripel, die s als Subjekt haben, gesucht. Alle Objekte der getroffenen Tripel werden wiederum geprüft, ob sie Resourcen sind und

diese im Triplestore enthalten sind. Ist dies der Fall, wird `getByValueLimited` mit diesen Objekten als Subjekte ausgeführt. Alle gefundenen Tripel werden als Antwort zurückgegeben. Die Methode `getByValueLimited` kann für alle unterspezifizierten Tripel benutzt werden.

3.4 Datenbankschemata

In den klassischen relationalen Datenbankmodellen werden Datenmodelle mit einem festen Datenbankschema definiert. Ein Datenmodell im Kontext Datenbanksysteme ist eine beliebige Abbildung eines Sachverhaltes mit Hilfe einer Beschreibungssprache (DDL) und einer Sprache zum Manipulieren der Daten (DML)[12]. Mit der DDL wird ein Datenbankschema definiert.[5]

Im App Engine Datastore gibt es kein festes Schema. Die DDL besteht aus den verschiedenenen Ausprägungen der Entitäten (Abschnitt 2.3.3), die eine unterschiedliche Anzahl an Attributen haben können. Somit können mehrere Datenbankschemata nebeneinander existieren. Dies bedeutet mehr Flexibilität bei der Benutzung des App Engine Datastores, dafür trägt die Anwendung selber die Verantwortung für die Einhaltung des Schemas und die Konsistenz der Daten im App Engine Datastore. Es empfiehlt sich also keine komplexen Abhängigkeiten zwischen Entitäten zu modellieren. Für diesen Anwendungsfall wäre der Einsatz von relationalen Datenbanksystemen sinnvoller.

Die DML wird in der Low-Level-API definiert (Abs. 2.3.3) und wird direkt in der Anwendung eingesetzt. Aus der Sicht der Nutzer ist die DML die im Abschnitt 3.3 definierte REST-API.

Es wurden in dieser Arbeit vier Datenbankschemata entwickelt, die im Folgenden vorgestellt werden.

3.4.1 Flaches Datenbankschema (Flat-Schema)

Dies ist das einfachste Datenbankschema. Jedes Tripel wird auf genau ein Entitätobjekt vom Typ Triple abgebildet (Source-Code 3.3). Der Schlüssel jedes Triples (s, p, o) wird aus dem Hash der Konkatenierung $s \circ p \circ o$ abgeleitet und im Feld `id` gespeichert. Das Hashen des Schlüssels ist notwendig, da die Länge von Schlüsseln auf 500 Character beschränkt ist und die Konkatenierung von Literalen diese Länge überschreiten kann. Der Ansatz des Hashings funktioniert auch bei Blank Nodes, da diese intern mit einem Universally Unique Identifier (UUID) versehen werden.

[12]Bei klassischen RDBMS ist die DML fast immer SQL

```java
@Entity
public class Triple {

  @Id
  private Long id;

  @Index
  public String subject;

  @Index
  public String predicate;

  @Index
  protected String object;

  private String text;
}
```

Source-Code 3.3: POJO für das flaches Datenmodell.

Das zusätzliche Feld `text` ist eingeführt worden, um den Primärtext großer Literale speicherbar und auffindbar zu machen. Im App Engine Datastore werden nur Strings bis zur Länge von 500 Character indiziert. Überschreitet o diese Länge wird ein Hash von o erzeugt und in dem Feld `object` gespeichert. Der Primärtext wird in `text` gespeichert. Wird für o kein Hash erzeugt, wird `text` mit `null` belegt. Alle Properties, die mit null belegt sind, werden nicht im App Engine Datastore gespeichert, so dass kein zusäztlicher Speicherplatz benötigt wird.

Als Hash-Algorithmus wurde Secure Hash Algorithm 1 (SHA-1) gewählt, da er eine hohe Kollisionssicherheit bietet. Kollisionssicherheit ist eine geforderte Eigenschaft von kryptographischen Hashfunktionen, im Gegensatz zu nicht kryptographischen Hashfunktionen, bei denen mit Kollisionen gerechnet werden muss.

Vorteile

- Einfügen/Löschen eines Tripels kostet jeweils einen Aufruf von `put()` und einen Aufruf von `get()`. Der `get`-Aufruf dient der Überprüfung, ob das Tripel bereits existiert, und muss nur für den Zähler vorgenommen werden. Würde kein Zähler benutzt werden, könnte das Tripel immer mit einem `put`-Aufruf persistiert werden.

- Einfügen/Löschen von mehreren Tripeln kann als Batch-Operation effizient durchgeführt werden, so dass nur ein Aufruf von `put()` nötig ist. Durch die Nutzung eines Zählers jedoch, muss die Batch-Operation auf vier Tripel begrenzt werden.

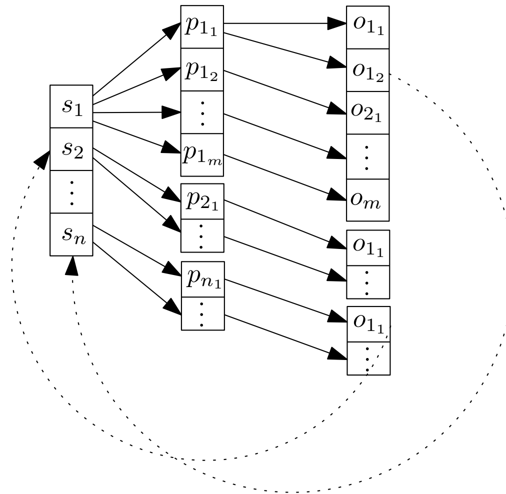


Abbildung 3.3: Verschachteltes Datenbankschema

Nachteile

- Für Anfragen von unterspezifierten Tripel muss eine Suche auf der Indextabelle ausgeführt werden.
- Für Anfragen mit transitiver Hülle größer als null, müssen für n unterschiedliche Resourcen n neue Suchen auf der Indextabelle durchgeführt werden. Je nach Graphenstruktur und Länge des Pfades, können es sehr viele Indexscans werden.

3.4.2 Verschachteltes Datenbankschema (Nested-Schema)

Das verschachtelte Datenbankschema versucht die vielen Indexscans bei Berechnung der transitiven Hülle (Abs. 2.2.5) zu vermeiden, die beim Flat-Schema anfallen, indem jedem Prädikat eine Resource und jedem Objekt ein Prädikat im Ancestor-Path zugeordnet wird (Abb. 3.3). Eine Menge von RDF-Tripeln $\{(s, p, o)\}$ wird auf eine Menge $\{(s, \{(p, \{o\})\}\}$ abgebildet. Jedes Subjekt s wird also nur einmal gespeichert und jedes Prädikat p nur einmal pro Subjekt.

Um diese Abbildung zu erreichen, werden drei POJOs eingeführt: `Subject`, `Predicate` und `RDFObject` (Source-Code 3.4). `Subject` speichert eine Menge von Schlüsseln auf Predicate und `Predicate` speichert wiederum Schlüssel auf `RDFObject`. Mit der Variable `parent` wird der Ancestor-Path aufgebaut.

Mit diesem Schema lassen sich die Schlüssel für die nächsten Resourcen im Pfad eines Suchbaums von den Objekten ableiten. Damit lässt sich eine effiziente Breitensuche (BFS) implementieren, da `get()` als Batch-Operation mit den abgeleiteten Schlüsseln aus dem

```
@Entity
class Subject {
    @Id String subject;
    Set<Key<Predicates>> predicateRefs;
}

@Entity
class Predicate {
    @Id String predicate;
    @Parent Key<Subject> parent;
    Set<Key<RDFObject>> objectRefs;
}

@Entity
class RDFObject {
    @Id String object;
    @Parent Key<Predicate> parent;
    String text;
}
```

Source-Code 3.4: Die POJOs für das Nested-Schema.

vorhergehenden Schritt ausgeführt werden kann.

Vorteile

- Alle Suchanfragen mit unterspezifizierten Tripeln, die zumindest s definiert haben, lassen sich ohne Indexscans beantworten.
- Die transitive Hülle ist vollständig ohne Indexscans berechenbar und durch Batch-Operationen lassen sich Datastore-Abrufe minimieren.
- Schreibzugriffe müssen wegen des Problems des schnellen Zählens (Abschnitt 3.7) innerhalb einer Transaktion erfolgen. Da bei vielen RDF-Graphen üblicherweise mehrere Fakten zu einem Subjekt gespeichert werden, lassen sich diese in einer Transaktion zu einer Entitätengruppe zusammenfassen. Damit ist der Import und gleichzeitiges Schreiben zwar abhängig von der RDF-Graphenstruktur, aber verglichen mit dem Flat-Schema schneller.

Nachteile

- Für unterspezifierte Tripel ohne Definition von s sind weiterhin Indexscans notwendig.
- Für den Aufbau eines vollständigen Tripels sind immer mindestens drei Datastore-Aufrufe von `get()` notwendig:

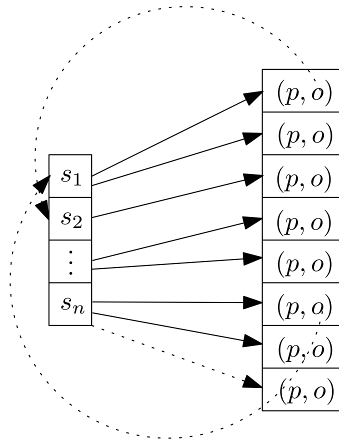


Abbildung 3.4: Dict-Schema: Aus den Objekten (o) können die Schlüssel zu anderen Subjekten abgeleitet werden (gestrichelte Pfeile), die im Triplestore gespeichert sind.

```
@Entity
class Subject {
    @Id String subject;
    Set<Key<Predicates>> predicateRefs;
}

@Entity
class Fact {
    @Id id;
    @Parent Key<Subject>;
    String predicate;
    String object;
}
```

Source-Code 3.5: POJO für die Repräsentation eines Fakts über eine Resource.

1. Zuerst wird die Resource mit der Liste aller Prädikate geholt.
2. Anschließend werden alle benötigten Prädikate geladen.
3. Zuletzt werden alle benötigten Objekte geladen.

3.4.3 Verzeichnis-Datenbankschema (Dict-schema)

Ähnlich dem Nested-Schema werden einem Subjekt ein oder mehrere Fakten zugewiesen. Ein Fakt ist ein Tupel, das aus einem Prädikat und Objekt besteht: (s, p). Jedes Tupel ist eindeutig über das zugeordnete Subjekt. Damit wird eine Menge von RDF-Tripeln $A = \{(s, p, o)\}$ auf eine Menge $B = \{(s, \{(p, o)\})\}$ abgebildet (Abb. 3.4).

Das Dict-Schema dupliziert nicht mehr Daten als das Nested-Schema und muss für die initiale Speicherung eines Tripels eine Entität weniger schreiben. Drei Fälle sind im

zu speichernde RDF-Terme	Nested	Dict	Flat	Dictbackref
(s, p, o)	3	2	1	4
(p, o)	1	1	1	3
(o)	1	1	1	3

Tabelle 3.1: Anzahl der zu speichernden Entitäten

Vergleich zum Nested-Schema zu betrachten (Tabelle 3.1). In allen Fällen schneidet das Dict-Schema mindestens gleich gut oder besser ab und hat die gleichen Vor- und Nachteile des Nested-Schemas. Aus diesem Grund wird das Nested-Schema in der Evaluation nicht betrachtet.

Vorteile

- Besitzt alle Vorteile des Nested-Schemas.
- Weniger Schreibzugriffe beim Speichern eines Tripels gegenüber dem Nested-Schema (Tab. 3.1).
- Eine `get`-Operation weniger für das Aufbauen eines Triples gegenüber dem Nested-Schema.

Nachteil

Das Dict-Schema erbt den Haupnachteil der vorigen Schemata: Indexscans sind ebenso wie beim Nested-Schema und Flat-Schema nötig.

3.4.4 Verzeichnis-Datenbankschema mit Rückwärtsreferenzen (Dictbackref-Schema)

Dieses Schema basiert auf dem Dict-Schema. Die Hauptidee dieses Schemas ist es Index-Scans zu vermeiden, da diese langsam sind und nicht als Batch-Operation ausgeführt werden können.

Um dieses Ziel zu erreichen, werden zwei neue POJOs eingeführt (Source-Code 3.6):

1. Um ein Prädikat auf eine Menge von Fakten abzubilden.
2. Um ein Objekt auf eine Menge von Fakten abzubilden.

```
1 @Entity
2 class BackrefPredicate {
3     @Id String predicate;
4     Set<Key<Fact>> factReferences;
5     Set<Key<Backrefs> backrefs;
6 }
```

Source-Code 3.6: POJO für die Modellierung von Referenzen auf Prädikate.

Dieses Schema benutzt das Fact-POJO des Dict-Schemas. Es werden aber keine Indizes für die Felder `predicate` und `object` angelegt, sondern jeweils eine Entität vom Typ `BackRef`. Diese `BackRef`-Entität speichert alle Schlüssel auf Fakten, die dieses Prädikat besitzen im Feld `factReferences`. Die `BackRef`-Entität bekommt als `Id` das Prädikat zugewiesen. Damit ist gewährleistet, das für jedes Prädikat nur eine `BackRef`-Entität angelegt wird. Für Objekte wird analog vorgegangen.

Wird ein Tripel mit dem Muster $(?, p, ?)$ gesucht, kann nun aus p der Schlüssel zur `BackRef`-Entität abgeleitet und diese geholt werden und dann die Schlüssel zu den Fakten aufgelöst werden. Damit können alle Fakten, die ein bestimmtes Prädikat besitzen über zwei `load`-Operationen berechnet werden, ohne auf Index-Scans zurückgreifen zu müssen.

Eine Schwierigkeit des Dictbackref-Schemas ist das sehr ungleich verteilte Auftreten von bestimmten Prädikaten und Objekten. So besitzen Subjekte häufig das `rdfs:type`-Prädikat. Oder bestimmte RDF-Graphen, wie z. B. die Sammlung der Abstrakta von Wikipedia-Artikeln mit $\approx$ 3,8 Millionen Tripeln, bei der jedes Subjekt das Prädikat `dbpedia-owl:abstract` trägt.[13] Für solche RDF-Graphen mit n distinkten RDF-Tripeln, müsste die `Backref`-Entität n Schlüssel speichern, was bei den erwähnten RDF-Graphen über Abstrakta an der zulässigen Anzahl von 5000 Elementen pro `Collection` und an der Größe von 1M pro Entität scheitern würde. Aus diesem Grund ist ein Limit für die Anzahl von Schlüsseln pro `BackRef`-Entität eingeführt worden. Ist dies erreicht, wird eine neue `BackRef`-Enität erzeugt und dort der Schlüssel zur `Fact`-Entität gespeichert.

Vorteile

- Besitzt alle Vorteile des Dict-Schemas.
- Keine Indexscans für unterspezifierte Abfragen notwendig.

[13] `http://downloads.dbpedia.org/preview.php?file=3.8_sl_en_sl_long_abstracts_en.nt.bz2` (abgerufen 19.06.2013)

Nachteile

- Nur ein Tripel kann pro Transaktion geschrieben werden, da mehrere Fakten über ein Subjekt wie im Dict- oder Nested-Schema zu einer Entitätengruppe zusammengefügt werden können, nicht aber die `BackRef`-Entitäten. Das macht den Import von Daten sehr langsam, da keinerlei Batch-Operationen möglich sind.

- Backrefs müssen gesplittet werden, damit sie die maximal erlaubte Größe von 1 MB für eine Entität nicht überschreiten.

- Das Dictbackref-Schema ist verglichen mit den vorherigen Schemata komplex, was die Relationen der Entitäten untereinander betrifft und damit schwerer zu warten.

3.5 Geschätzte Schreib- und Lesekosten der Datenbankschemata für den Import von RDF-Tripeln

Die Schätzung der Kosten für den Import von Tripeln sind weniger Laufzeitproblemen geschuldet als tatsächlichen finanziellen Kosten, die entstehen. Die Tabellen 3.2 listen die Kosten für das Schreiben eines einzelnen RDF-Tripels auf. Die Schätzungen in Tabelle 3.2a gehen immer vom Worst-Case (schlechtester Fall) aus. Der Worst-Case ist für die drei Datenbankschemata (Flat, Nested, Dict) definiert, als ein zu speicherndes Tripel (s, p, o), das sich in s von allen bereits im Triplestore gespeicherten Tripeln unterscheidet.

Für das Flat-Schema ist der Best-Case (bester Fall) und Average-Case (durchschnittlicher Fall) gleich dem Worst-Case. Sobald ein Tripel noch nicht im Datastore gespeichert ist, also sich in mindestens einem Element unterscheidet, muss das Tripel vollständig neu geschrieben werden. Es gilt also für die Gesamtanzahl von Lese - und Schreiboperationen eines RDF-Graphen für das Flat-Schema: $|\{(s, p, o)\}| \times 9$.

Das Nested sowie Dict-Schema hängen bei der Anzahl an Lese- und Schreibvorgängen von der Struktur des RDF-Graphen ab. Beide Schemata erreichen im Best-Case keine besseren Wert als sieben Operationen pro Tripel.

Das Dictbackref-Schema bildet eine Ausnahme, da sich hier die Anzahl an Leseoperationen nicht unabhängig von der Anzahl der schon gespeicherten Tripeln entwickelt. Durch die Splittung der BackRef-Entitäten bei häufig vorkommenen Prädikaten und Objekten, müssen vor dem Schreiben eines neuen Tripels alle `BackRef`-Entitäten geladen werden. Sei g eine Funktion, die das Vorkommen von p zählt und f eine Funktion, die das Vorkommen von o in allen gespeicherten Tripeln zählt und c die Limitierung der Größe der Schlüsselmenge für eine `BackRef`-Entität, dann ergibt sich folgende Formel für

Schema	Operation			Gesamt
	Read	Write		
		Index	Entity	
Flat	$1_{\texttt{Triple}}$	$2_s + 2_p + 2_o$	$2_{\texttt{Triple}}$	9
Nested	$1_{\texttt{subject}}$	$2_p + 2_o$	$2_{\texttt{Subject}} + 2_{\texttt{Predicate}} + 2_{\texttt{RDFObject}}$	11
Dict	$1_{\texttt{Subject}}$	$2_p + 2_o$	$2_{\texttt{Subject}} + 2_{\texttt{Fact}}$	9

(a) Worst-Case

Schema	Operation			Gesamt
	Read	Write		
		Index	Entity	
Flat	$1_{\texttt{Triple}}$	$2_s + 2_p + 2_o$	$2_{\texttt{Triple}}$	9
Nested	$1_{\texttt{Subject}} + 1_{\texttt{Predicate}}$	2_o	$1_{\texttt{Predicate}} + 2_{\texttt{RDFObject}}$	7
Dict	$1_{\texttt{Subject}}$	$2_p + 2_o$	$1_{\texttt{Subject}} + 2_{\texttt{Fact}}$	8

(b) Best-Case

Tabelle 3.2: Lese- und Schreibkosten der einzelnen Datenbankschemata für ein Tripel, das sich in allen RDF-Termen s, p und o von allen gepeicherten Tripeln unterscheidet, werden in Tabelle 3.2a gelistet.

Die Tabelle 3.2b führt die Anzahl an Datenbankoperationen auf, wenn $(s, p, *)$ eines Triples (s, p, o) schon gespeichert wurden, da dies für Nested- und Dict-Schema der Best-Case ist.

die Lesezugriffe:

$$\text{dictbackrefReads}(p, o, c) = \left(1 + \frac{\lceil f(p) \rceil + \lceil g(o) \rceil}{c}\right) \tag{3.1}$$

Im Folgenden ein Rechenbeispiel: Im Prototypen ist standardmäßig $c = 500$ gesetzt. Mit dem Tool `SimpleRDFStats`, welches noch genauer in Kapitel 4 behandelt wird, wurde in einem RDF-Graphen mit $500\,k$ Tripeln[14] das häufigeste Vorkommen eines Fakts über ein Subjekt gefiltert:

- Das Prädikate `dbpedia-owl:country` trat 15929 Mal auf.
- Das Objekt `dbpedia/United_States` trat 14042 Mal auf.

Möchte man nun ein Tripel $(s, \texttt{dbpedia} - \texttt{owl} : \texttt{country}, \texttt{dbpedia/United_States})$ speichern, sind nach der oben genannten Formel 61 Lesezugriffe nötig. Das ist ein Extrembeispiel, tritt aber in der Praxis häufig auf.

Der Worst-Case für das Dictbackref-Schema entsteht, wenn alle Tripel eines RDF-Graphens den gleichen Wert für Prädikat und Objekt haben. Sei x die Anzahl der gespeicherten Tripel und ersetze f und g aus der Formel (3.1) mit x, da die Ergebnisse von f und g in diesem Fall immer gleich der Anzahl der im Triplestore gespeicherten Tripel ist, dann gilt:

$$r(x) = 1_{\texttt{Subject}} + 2\frac{x_{\texttt{Backref}}}{c} \tag{3.2}$$

und für die Schreibkosten:

$$w(x) = \begin{cases} c \mid x & : \quad 2_{\texttt{Subject}} + 2_{\texttt{Fact}} + 6_{\texttt{BackRef}} \\ c \nmid x & : \quad 2_{\texttt{Subject}} + 2_{\texttt{Fact}} + 2_{\texttt{BackRef}} \end{cases} \tag{3.3}$$

Für die ersten drei Schemata und mit der Annahme des Worst-Case können die Kosten

[14] Es wurden die ersten $500\,k$ Tripel aus dem nt-file `http://downloads.dbpedia.org/3.8/en/mappingbased_properties_en.nt.bz2` (abgerufen 20.06.2013) Tripel als Eingabe für `SimpleRDFStats` benutzt.

Operation	Kosten
Write	$0.09\,\$$ per $100\,k$ operations
Read	$0.06\,\$$ per $100\,k$ operations
Small	$0.01\,\$$ per $100\,k$ operations

Tabelle 3.3: Preise für Datenbankoperationen[15]

mit Hilfe der Tabelle 3.3 für den Import eines RDF-Graphens geschätzt werden:

$$\text{dictCosts}(x) = \text{flatCosts}(x) = \frac{0,06\,\$ + 0,09\,\$ \cdot 8}{100\,k} \cdot x \tag{3.4}$$

$$\text{nestestCosts}(x) = \frac{0,06\,\$ + 0,09\,\$ \cdot 10}{100\,k} \cdot x \tag{3.5}$$

$$\text{dictBackRefCosts}(x) = \frac{\sum_{i=0}^{x-1} 1 + 0,06\,\$ \cdot r(i) + 0,09\,\$ \cdot w(i)}{100\,k} \tag{3.6}$$

Der Import von einer Million Tripel würde also im ungünstigsten Fall für das Dict und Flat-Schema $7,80\,\$$ kosten, bei der Wahl des Dictbackref-Schemas mit $c = 500$ sogar $607\,\$$, wenn der für dieses Schema definierte Worst-Case angenommen wird, dass alle Subjekte das gleiche Prädikat und Objekt besitzen.

3.6 Das Blank-Node-Problem

Für die Lösung des Blank-Node-Problems müssen bei der Eingabe von Blank Nodes einige Restriktionen beachtet werden. Der Triplestore fügt alle RDF-Tripel intern zu einem einzigen großen RDF-Graphen zusammen. Das funktioniert, weil jede Resource über eine RDF URI Reference eindeutig bestimmt ist und darauf bauen die Datenbankschemata auf. Blank Nodes dagegen sind nicht durch URIs eindeutig bestimmbar, sondern hängen von einer Resource ab, die durch eine RDF URI Reference eindeutig bestimmt ist.

Bei Blank Nodes hat die Bezeichnung selbst keinerlei semantische Bedeutung, sondern die Semantik leitet sich aus dem Kontext ab, in dem sie sich befinden. Blank Nodes sind dazu da, Relationen zwischen Resourcen zu modellieren, bei denen eine gemeinsame Resource noch nicht bekannt ist. Ein Beispiel wäre, das *Bob* und *Alice* einen gemeinsamen Freund besitzen, dessen Name aber unbekannt ist. Ein Weg diese Art von Relationen auch ohne Blank Nodes auszudrücken, wäre eine Ontologie zu definieren, die eine Re-

[15] `https://developers.google.com/appengine/docs/billing?hl=de` (abgerufen 05.06.2013)

source beinhaltet für *unkekannte Freunde*. Diese Lösung gilt als nachhaltiger als der Einsatz von Blank Nodes und sollte vor Veröffentlichung von RDF-Datensätzen auch angewandt werden. Allerdings ist es für eine schnelle Modellierung von RDF-Graphen oft zu umständlich eine Ontologie zu definieren oder zu erweitern und an dieser Stelle sind Blank Nodes nützlich.

3.6.1 Interne Repräsentation von Blank Nodes

Blank Nodes werden intern als normale Resourcen behandelt. Dies ist möglich, da jeder Blank Node, bevor er gespeichert wird, auf eine URI abgebildet wird und damit intern eindeutig identifzierbar ist. Jeder Blank Node erhält den Präfix:

$$\texttt{genid} = \texttt{pixeldrama.de/pdgs/blank/node} \tag{3.7}$$

Dieser Präfix wird konkateniert mit einer zufällig erzeugten UUID. UUID ist ein Standard für global eindeutige Kennungen, die mit ausreichend hoher Wahrscheinlichkeit kollisionsfrei sind.

Das Vorgehen, Blank Nodes auf eindeutige Konstanten abzubilden, wird im Kontext RDF-Graphen als Skolemisation[16] bezeichnet und wird z.B. vom Jena-Framework angewendet.

3.6.2 Probleme beim Speichern von Blank Nodes

Aus Sicht des Triplestores ist es nicht möglich zu entscheiden, ob ein Blank Node mit einem Fakt erweitert oder ein neuer Blank Node angelegt werden soll. Diese Entscheidung muss der Client treffen. Blank Nodes werden beim Speichern auf eindeutige URIs abgebildet, so dass der Client sie nicht mehr direkt abfragen kann.

Die Abbildung 3.5 verdeutlicht das Problem. Wird der Graph innerhalb eines HTTP-Request persistiert, bleibt die Struktur des Graphen erhalten. Source-Codes 3.7 gibt den Graphen im nt-Format wieder. Teilt nun der Client die Zeilen 1 und 2 und 1 und 3 auf zwei HTTP-Request auf (z. B. eine nachträgliche Änderung/Hinzufügung), dann würde ein anderer Graph entstehen als ursprünglich modelliert wurde (Abb. 3.6). Bob würde nun zwei unterschiedliche Personen gesehen haben, von denen eine Alice und eine Sally kennt. Splittet der Client einen solchen Graphen auf mehrere Request auf, muss er die interne UUID benutzen, da sonst der RDF-Graph aus Abbildung 3.6 gespeichert wird.

[16] `http://www.w3.org/TR/rdf11-concepts/#section-skolemization` (abgerufen 4.07.2013)

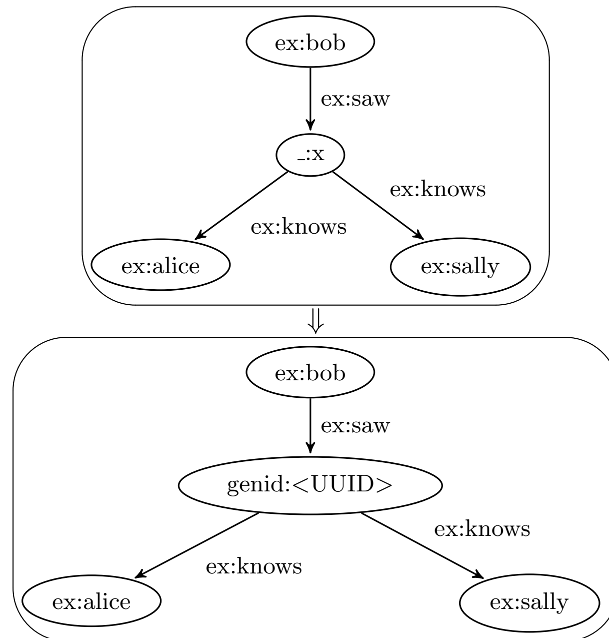


Abbildung 3.5: Entscheidung beim Einfügen von Blank Nodes. Wird der Ausgangsgraph innerhalb eines Request gespeichert, entspricht der intern gespeicherte Graph, dem vorher modellierten. Lediglich der Präfix `genid` muss durch das Zeichen _ ersetzt werden und die Gleichheit der RDF-Graphen wäre hergestellt.

```
1 ex:bob ex:saw _:x .
2 _:x ex:knows ex:alice .
3 _:x ex:knows ex:sally .
```

Source-Code 3.7: Modelliert eine Person Bob, die eine unbekannte Person gesehen hat. Die unbekannte Person kennt wiederum Alice und Sally.

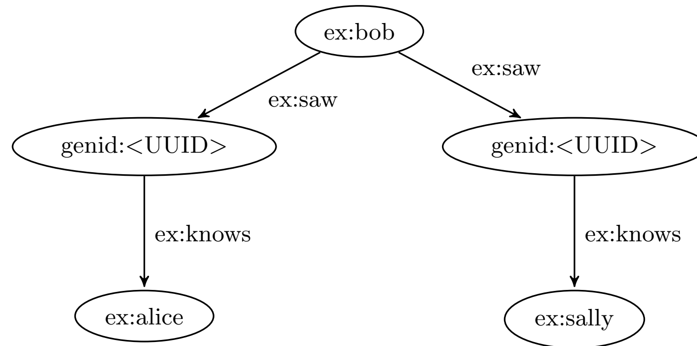


Abbildung 3.6: Falscher RDF-Graph resultierend daraus, dass die Daten aus Source-Code 3.7 in zwei HTTP-Request persistiert wurden: Zuerst Zeile eins und zwei, anschließend Zeile eins und drei.

```
1 ex:bob ex:saw _:x .
2 _:x ex:knows ex:alice .
3 ex:bob ex:saw _:y .
4 _:y ex:knows ex:alice .
```

Source-Code 3.8: Modelliert eine Person Bob, die zwei unbekannte Personen gesehen hat. Diese unbekannten Person kennen beide Alice.

Ein mit diesem Problem verwandtes Problem ist das Speichern von zwei anonymen Resourcen, die beide auf die gleiche Resource verweisen. Um in dem Szenario von Bob und Alice zu bleiben: Bob hat zwei unterschiedliche unbekannte Personen gesehen, die beide Alice kennen. Angenommen der Client würde zuerst die ersten zwei Zeilen des Source-Codes 3.8 versuchen zu persistieren und anschließend die letzten zwei Zeilen, dann würde man zwei isomorphe Subgraphen speichern, denn getrennt voneinander besitzen sie die gleiche Struktur. Der Triplestore speichert dieses Beispiel zwar korrekt ab, würde also die Isomorphie der zwei Graphen ignorieren, allerdings erkennt er damit auch keine fehlerhaften, doppelt gespeicherten Blank Nodes.

Das erstgenannte Problem lässt sich vermeiden, indem der Client immer verkettete Blank Nodes in einem HTTP-Request persistiert. Gelingt dies nicht (Fehler bei der Verbindung, interne Serverfehler usw.), kann der Client immer noch über die vom Triplestore vergebenen eindeutigen UUIDs die Tripel löschen und Isomorphietests ausführen, um die die Integrität des RDF-Graphen wieder herzustellen.

3.7 Schnelles Zählen und gleichzeitige Schreibzugriffe auf den Triplestore

Der naive Ansatz, über alle Tripel im Datastore zu iterieren, überschreitet bei einer größeren Anzahl von Tripeln die Zeitlimitierung von 60 Sekunden. Deswegen wurde ein Count-Objekt eingeführt, dass bei allen Schreibvorgängen aktualisiert wird und anschließend im Datastore gespeichert wird.

Dieser Ansatz ist vergleichbar mit dem Zähler bei gängigen Implementierungen der abstrakten Datenstruktur einer verketteten Liste (z. B. LinkedList in Java). Bei diesen wird beim Hinzufügen eines Elements der Zählen inkrementiert, so dass der Abruf der Anzahl von gespeicherten Elementen in der verketteten Liste in konstanter Zeit möglich ist und nicht bei jedem Abruf über alle Element iteriert werden muss.

Der Nachteil dieses Ansatzes ist, dass das Aktualisieren des Count-Objekts nur innerhalb einer Transaktion erledigt werden kann, um keine Inkonsistenzen bei gleichzeitigen Schreibzugriffen zu verursachen.[17] Das wirkt sich am stärksten auf das Flat-Schema aus, bei dem das Schreiben eines Tripels eine Cross-Group-Transaktion kostet und somit höchsten vier Tripel in einer Batch-Operation geschrieben werden können. Beim Nested-Schema dagegen können die Prädikate und Objekte eines Subjekts zu einer Entitäten-Gruppe zusammengefasst werden und innerhalb einer Transaktion in einer Batch-Operation geschrieben werden.

Für das Nested-, Dict- und Dictbackref-Schema ist das Ausführen der Schreibvorgänge innerhalb von Transaktionen zwingend notwendig, da bei Fehlern, wie z. B. Timeouts, Objekt oder Prädikate ohne das zugehörige, auf sie verweisende Subjekt gespeichert werden könnten und somit Artefakte im Datastore verbleiben. Das Flat-Schema jedoch benötigt für das Schreiben den Transaktionsmechanismus nur wegen des „schnellen Zählens“, da das Schreiben und Lesen genau einer Entität immer atomar ist.

Bei gleichzeitigem Schreiben von Tripeln kann ein Count-Objekt zum Engpass werden. Dem kann man mit der Technik des „sharding counters“ entgegen wirken.[18] Damit ist der Nachteil verbunden, dass einfache `count`-Aufrufe mehr als ein Lesezugriff verursachen und somit teurer sind.

[17] Cross-Group-Transaktionen sind nur sehr eingeschränkt durchführbar (Abschnitt: Transaktionen im App Engine Datastore)

[18] `https://github.com/GoogleCloudPlatform/appengine-sharded-counters-java` (abgerufen am 14.06.2013)

4 Evaluierung

4.1 Test-Pipeline

Für die Tests wurden zwei Kommandzeilenprogramme entwickelt, um die RDF-Graphen für den Import aufzubereiten und die Struktur eines RDF-Graphens zu analysieren:

1. `SimpleRDFStats`[1]: Diese Program erstellt einfache Statistiken über RDF-Graphen. Das Eingabeformat ist auf N-Triples beschränkt. Es kann sowohl eine einzelne Datei verarbeitet werden als auch Verzeichnisse mit mehreren Dateien, wobei alle Dateien zu einem großen RDF-Graphen zusammengeführt werden. Das Programm gibt Informationen über folgende Eigenschaften aus:
   a) Anzahl der gelesenen Tripel.
   b) Anzahl der gelesenen distinkten Tripel.
   c) Anzahl der gelesenen distinkten Subjekte.
   d) Frequenz der am häufigsten vorkommenden Prädikate.
   e) Frequenz der am häufigsten vorkommenden Objekte.
   f) Das Subjekt, welches den längsten Pfad in seiner transitiven Hülle hat.

   Diese Daten sind wichtig, um gezielte Suchanfragen zu entwickeln, z. B. Aufbauen eines RDF-Graphen mit größerer Tiefe oder Anzahl von Schreib- und Lesezugriffe zu erklären.

2. `pdgs-cli`[2]: Dieses in Haskell geschriebene Programm wurde entwickelte, um Daten für den Triplestore aufzubereiten, zu importieren und um Log-Einträge (Abs. 4.1.2) abzufragen und für Plots aufzubereiten. Es verarbeitet als Eingabeformat N-Triples.

[1] Ein Release des Programms ist auf github verfügbar: `https://github.com/pixeldrama/simplerdfstats` (abgerufen 31.07.2013)

[2] Source-Code des `pdgs-cli`-Programms ist unter `https://github.com/pixeldrama/pdgs` (abgerufen 31.07.2013) einsehbar und Kompilate stehen unter `https://github.com/pixeldrama/pdgs-cli/releases` (abgerufen 31.07.2013) bereit.

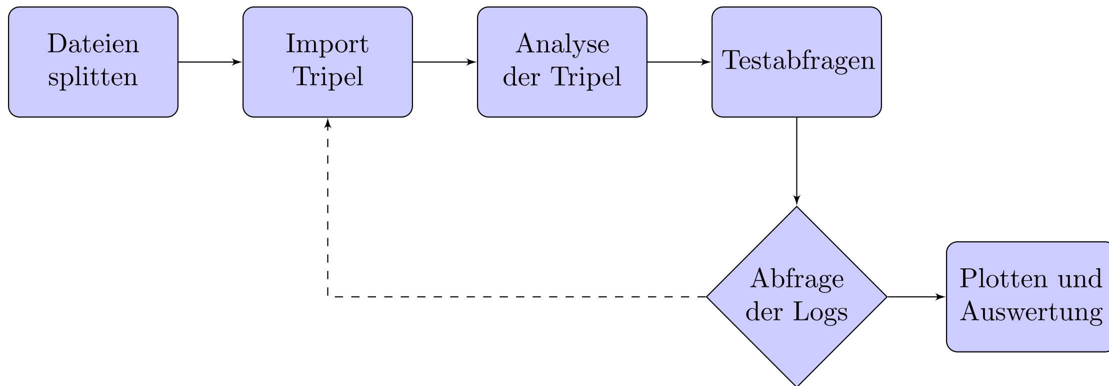


Abbildung 4.1: Test-Pipeline

Im ersten Schritt (Abb. 4.1) werden mit `pdgs-cli` große NT-Triples-Dateien für den Import gesplittet. Dies ist nötig, damit verschiedene Restriktionen von GAE nicht verletzt werden:

1. Die Größe eines einzelnen HTTP-Request ist beschränkt auf 32 MB.
2. Die Beschränkung von Lese- und Schreibzugriffe auf 50 k pro Tag.
3. Die Laufzeit eines Prozesses ist auf 60 s beschränkt.

Gerade der letzte Punkt ist wichtig. Bricht GAE einen Prozess beim Import ab, wäre das hinsichtlich der Konsistenz der Daten kein Problem, da alles innerhalb einer Transaktion geschrieben wird, hinsichtlich der Effizients jedoch auf jedenfall, weil viele Daten gesendet wurden, welche den Transformationsprozess zu den einzelnen Datenbankschemata verlangsamen und Bandbreite verbrauchen. Was den zweiten Punkts betrifft, kann `pdgs-cli` die Anzahl der NT-Triples-Dateien beschränken, damit noch Lesezugriffe für die Auswertung der Logs zur Verfügung stehen.

Der zweite Schritt ist der Import von Daten. `pdgs-cli` verschiebt alle erfolgreiche importierten NT-Dateien anschließend in einen Ordner, damit diese von `SimpleRDFStats` lokal analysiert werden können.

Je nachdem wie der Import und die Analyse der importierten Daten verliefen, wurde der Datenimport wiederholt oder weitergeführt und Abfragen für die Tests abgeleitet.

4.1.1 Vorgehen

Vor jedem Test wurden Warmup-Requests abgesetzt, damit GAE eine Instanz der Applikation lädt und benötigte Resourcen von der Festplatte liest. Sonst würde der erste

CPU	Intel(R) Core(TM) i3-2310M CPU @ 2.10GHz
Cache-Size	3072 KB
Prozessoren	4
Prozessortaktung	800 Mhz
RAM	8 GB

Tabelle 4.1: Spezifikation für lokale Tests.

Aufruf über die REST-API ein Loading-Request sein, der deutliche Latenzen haben kann. Es wurden also nur sogenannte *warme* Testläufe vorgenommen.

GAE bietet die Möglichkeit mehrere Instanzen der Applikation im Wartemodus auszuführen, so dass auf Peeks schnell reagiert werden kann. Standardmäßig, wenn nichts anderes erwähnt, ist bei allen Applikationen immer genau eine Instanz im Wartenmodus und die Latenzzeit für Request auf 500ms gesetzt, bis zusätzliche Instanzen geladen werden.

Jedes Datenbankschema erhält eine eigene Applikation. Letztendlich werden alle Tripel in der gleichen Bigtable-Tabelle gespeichert, allerdings gelten die Restriktionen für jede Applikation einzeln, so dass testweise mehr Tripel importiert werden können. Zusätzlich können Statistiken, die GAE selber über die Daten, Requests, Speicherverbrauch usw. erhebt, leichter ausgewertet werden.

Manche Tests wurden lokal wiederholt, um Effekte, die nicht in der Laufzeitumgebung von GAE gemessen und nachgewiesen werden konnten, darzustellen. GAE stellt für Entwickler einen Development-Server bereit, der einen verteilten Appengine-Datastore simuliert. Die Daten der lokalen Messungen sind nicht ohne weiteres vergleichbar mit den Messungen von GAE. Die technischen Daten für die lokalen Tests können Tabelle 4.1 entnommen werden.

4.1.2 Erstellen und Abfragen von Logs

Um den Import und Abfragen der Tripel analysieren zu können, wurde ein POJO `LogEntry` sowie eine zusätzliche Klasse eingeführt, die das `DAO`-Interface des Prototypen (Abb. 3.2) implementiert. Diese ruft mittels Delegation das eigentliche, darunter liegende `DAO`-Objekt für das jeweilige Datenbankschema auf und speichert danach das `LogEntry`-Objekt in der Datenbank.

Damit die Logs abgefragt werden können, wurde eine REST-API entwickelt. `pdgs-cli` kann diese Logs abfragen und konvertiert sie in ein tabseperatiertes Format, um von Statistik-Tools verarbeitet werden zu können.

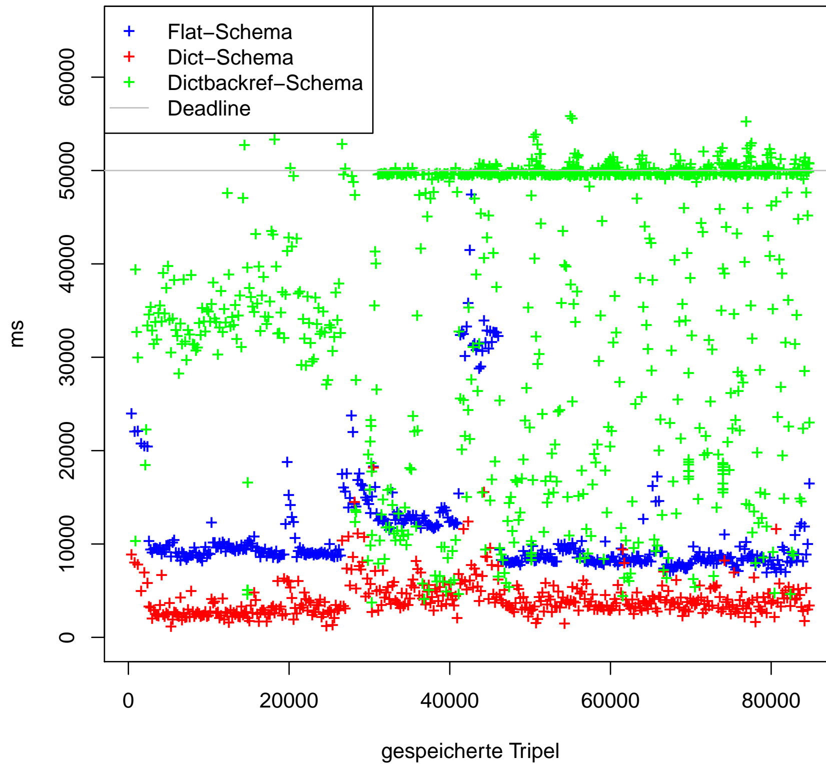


Abbildung 4.2: Persistierung von Tripel-Chunks der Größe 200.

4.2 Persistierung der RDF-Tripel

Für die Persistierung eines großen RDF-Graphen wurde ein Graph in Chunks von 200 Tripeln gesplittet. Benutzt wurden Daten von Dbpedia. Da die Beschränkung der Lese- und Schreibzugriffe mit dem Schreiben von $\approx$ 5000 Tripeln erreicht war, musste der Import auf mehrere Tage verteilt werden, um eine signifikante Menge an Tripeln zu speichern. Es erfolgten keine gleichzeitigen Schreibzugriffe. Benutzt wurde der Datensatz `Mapping-Based-Properties` von DBpedia[3], da dessen Struktur nicht flach ist und dieser im Verhältnis zur Tripelanzahl viele unterschiedliche Prädikate enthält (Tab. 4.2).

Die Abbildung 4.2 zeigt das Ergebnis für den Import von $\approx$ 80 k Tripeln. Das Dictbackref-Schema hat, wie erwartet, die schlechtesten Schreibwerte, da pro Transaktion nur ein Tripel geschrieben werden kann. Das Flat-Schema kann vier Tripel pro

[3]`http://downloads.dbpedia.org/3.8/en/mappingbased_properties_en.nt.bz2` (abgerufen 8.07.2013)

Tripel Anzahl	41121	84750
Subjekte	4958	10462
Prädikate	709	835
Objekte	25182	48307
längster Pfad	4	5
häufigstes Prädikat	3899 (foaf:name)	7993 (foaf:name)
häufigstes Objekt	804	1901

Tabelle 4.2: RDF-Graphstruktur eines Subgraphen des Datensatzes.

Transaktion schreiben und ist damit wesentlich schneller. Das Dict-Schema kann beliebig viele Fakten über ein Subjekt in einer Transaktion zusammenfassen, was sich bei RDF-Graphen mit dieser Struktur (Tab. 4.2) bemerkbar macht, da im Schnitt $\approx 500\,ms$ schneller als das Flat-Schema. Damit schneidet das Dict-Schema am besten ab. Der leichte Anstieg ab ungefähr 30 k Tripeln kann für das Flat-Schema und dem Dict-Schema nicht mit dem Datenbankschema erklärt werden, und hat wahrscheinlich seine Ursache in der Auslastung von GAE. Beim Dictbackref-Schema steigen allerdings die Lesezugriffe mit der Anzahl der Tripel im Triplestore, so dass ab 30 k Tripel fast immer die Deadline erreicht wird.

Bei der Messung kam es zu Problemen. Die Chunk-Größe der Tripel wurde mit 200 bewusst niedrig gewählt, damit die Deadline von 60 s nicht erreicht wird. Trotzdem brach das Dictbackref-Schema häufig mit einer `DeadLineExceed`-Exception ab, was zur Folge hatte, dass keine Log-Einträge mehr geschrieben wurden. Aus diesem Grund wurde in dem Prototypen ein Zeitpuffer von mindestens 10 s festgelegt und eine `PartialPersistException` geworfen, wenn dieser Puffer unterschritten wird, damit noch genug Zeit bleibt einen Log-Eintrag zu erstellen. Aus der Abbildung 4.2 ist ersichtlich, dass dieser Puffer ab $\approx$ 30 k importierten Tripel eingeführt wurde, da dort die Grenze von 50 s nicht mehr deutlich überschritten wird. Die deutlich schnelleren Transaktionen des Dictbackref-Schemas sind Wiederholungen von abgebrochenen Schreibvorgängen, die durch eine `PartialPersistException` oder `DeadLineExceedException` verursacht wurden.

Das DictBackref-Schema verzeichnet eine steigende Anzahl an Lesezugriffen in Abhängigkeit von der Gesamtanzahl der gespeicherten Tripel. Dieses Verhalten ist in Abschnitt 3.5 vorhergesagt worden und ist in einer expliziten Messung der Lese- und Schreibkosten in Tabelle 4.3 aufgeführt.

Zusätzlich wurde eine Messreihe mit gleichzeitigen Schreibzugriffen erstellt. Diese wurden lokal und auf GAE durchgeführt. Allerdings wurden auf GAE nur jeweils zwei

Schema	gespeicherte Tripel	Operationen	Anzahl der Operationen	
			% von 50k ops	absolute
Flat	0	read	9	≈ 0,00k
		write	59	≈ 0,03k
	81221	read	9	≈ 0,00k
		write	59	≈ 0,03k
Dict	0	read	2	≈ 0,00k
		write	46	≈ 0,03k
	81221	read	2	≈ 0,00k
		write	46	≈ 0,03k
Dictbackref	0	read	29	≈ 0,01k
		write	49	≈ 0,02k
	81221	read	60	≈ 0,03k
		write	52	≈ 0,03k

Tabelle 4.3: Lese- und Schreibkosten der einzelnen Datenbankschemata in Abhängigkeit von der importierten Tripel-Menge. Importiert wurden 3529 Tripel in einen leeren Triplestore und in einen Triplestore in dem zuvor 81221 importiert wurden. Das Dictbackref-Schema zeigt einen deutlichen Anstieg mit zunehmender Tripel-Menge bei den Lesezugriffen. Die anderen Schemata bleiben dagegen stabil.

Messreihen pro Schema vorgenommen, da diese durch die Limitierung des Free-Accounts auf mehrere Tage verteilt werden mussten.

Verwendet wurden die ersten 1000 Tripel des Datensatzes `Mapping-Based-Properties`. Diese wurden mit bis zu acht Threads importiert. Es wurde pro Request immer ein Tripel geschrieben, so dass der durchschnittliche Durchsatz gemessen werden konnte. Die Tabelle 4.4 gibt Auskunft über die Resultate. Mit zunehmender Thread-Anzahl nimmt die Schreibrate deutlich ab, was durch den gleichzeitigen Zugriff auf das Count-Objekt verstärkt wird. Wird die Technik der „sharding counters“ verwendet, wird das Abnehmen der Schreibrate gedämpft.

Allerdings sind diese Messungen nur ein Hinweis darauf, wie sich die Applikation bei gleichzeitigen Schreibzugriffen auf GAE verhält. Bei den lokalen Messungen läuft immer nur eine Instanz der Webapplikation, während auf GAE dynamisch bis zu 500 zusätzliche Instanzen erzeugt werden können, wenn ein Request zu lange in der Pending-Queue wartet. Grundsätzlich lässt sich aber festhalten, dass sich bei gleichzeitigen Schreibzugriffen der Einsatz von mehreren Count-Objekten empfiehlt.

4.3 Abfragen

In diesem Abschnitt wird diskutiert, wie sich die einzelnen Datenbankschemata bei einfachen Abfragen verhalten, wie beim Aufbau eines Graphens und ob die theoretischen

Instanz	C	T	Ø ms	M (ms)
lokal	1	1	5,57	5
		2	27,64	7
		4	437,36	301,5
		8	967,52	700,5
	1000	1	5,22	4
		2	8,25	5
		4	49,87	35
		8	150,69	126
GAE	1	8	647,61	126
	1000		190,51	157

(a) Flat-Schema

Instanz	C	T	Ø ms	M (ms)
lokal	1	1	8,2	7
		2	42,67	10
		4	541,68	353
		8	-	-
	1000	1	7,53	6
		2	13,05	8,5
		4	63,03	40
		8	180,85	131
GAE	1	8	632,42	120
	1000		148,64	115

(b) Dict-Schema

Instanz	Counter	Thread	Ø ms	M (ms)
lokal	1	1	11,17	10
		2	80,16	15
		4	933,86	587
	1000	8	-	-
		1	5,22	4
		2	8,25	5
		4	49,87	35
		8	150,69	126
GAE	1	8	995,99	262
	1000		229,10	181

(c) Dictbackref-Schema

Tabelle 4.4: Gleichzeitige Schreibzugriffe der Datenbankschemata auf den lokalen Entwicklungsserver und auf GAE. Das Dict- und Dictbackref-Schema konnte lokal nicht mit 8 Threads gemessen werden, da der Entwicklungsserver reproduzierbar nach ein paar hundert geschriebenen Tripeln abstürzte. Die Ursache für den Absturz konnte nicht gefunden werden. Auf GAE tracht das Problem nicht auf.
C = Anzahl Counter, T = Anzahl Thread, Ø= durchschnittliche Schreibrate eines Tripels, M = Median über die Schreibdauer aller Tripel

Überlegungen, aufgrund derer die Datenbankschemata entwickelt wurden, sich mittels Messungen nachweisen lassen und sich vielleicht bestätigen.

Einzelne Tripel (voll spezifizierte Abfragen)

Diese Art der Abfrage wird nicht benötigt, um Relationen zwischen Dingen, die in einem RDF-Graph modelliert werden, aufzubauen, sondern sind nur dazu da zu prüfen, ob ein Fakt existiert oder nicht und wenn er existiert gegebenfalls zu löschen. Die Antwort enthält also immer genau ein Tripel oder keines.

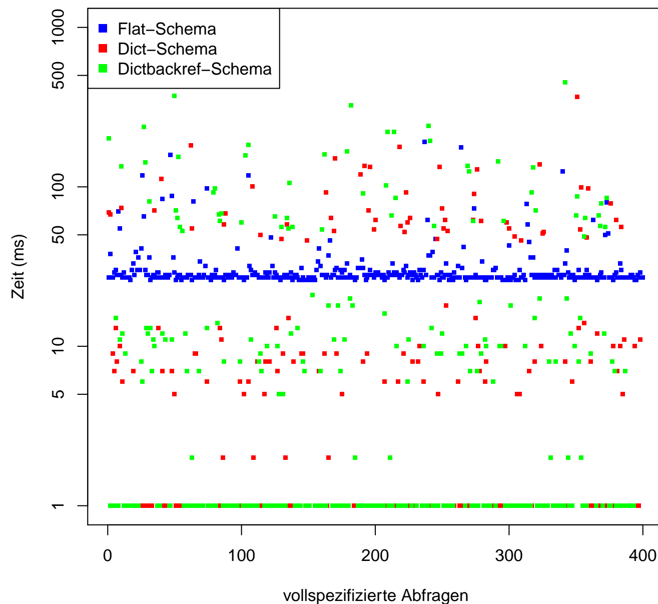
Eine voll spezifizierte Abfrage enthält ein RDF-Muster, bei dem alle RDF-Elemente definiert sind. Daraus kann der Triplestore den Schlüssel mittels Konkatenierung und MD5-Hash für das Flat-Schema bestimmen und anschließend das Tripel laden. Das Dict-BackrefSchema und Dict-Schema können die Schlüssel für die `Resource`- und die `Fact`-Entität ableiten und können innerhalb einer Batchoperation beide Entitäten laden und das Tripel zurückgegeben. Keines der Schemata sollte also bei diesen Anfragen merklich langsamer sein. Verwendet wurden zufällig gewählte 400 Tripel. Abbildung 4.3 zeigt das Resultat. Das DictBackref- und Dictschema pendeln sich bei $1\,ms$ ein, während das Flat-Schema etwas länger braucht. Die Latenz von $8\,ms$ im Durchschnitt kann nicht dem Flatschema zugeschrieben werden, sondern wird wahrscheinlich von der Auslastung des GAE-Servers verursacht.

Abfrage von Resourcen (`getBySubject:` $(s, ?, ?)$)

Unterspezifierte Anfragen können mehrere Tripel treffen. In diesem Fall werden alle Fakten über eine Resource zurückgegeben. Wie eine solche Anfrage in SPARQL formuliert wird, zeigt der Source-Code 4.1. Diese Anfrage findet in der Praxis Verwendung (im Gegensatz zu den vollspezifierten Anfragen, die eher administrativen Charakter haben).

Verglichen wurde das Flat-Schema mit dem Dict-Schema, da die Implementierungen dieser Abfrage beim Dict-Schema und DictBackref-Schema identisch sind und somit die Messungen keine großen Unterschiede aufweisen sollten.

Die Ergebnisse überraschen, da das Dict-Schema eigentlich dem Flat-Schema überlegen sein sollte. Tatsächlich ist aber das Flat-Schema im Durchschnitt schneller auf GAE, obwohl für jede Abfrage ein Index-Scan ausgeführt werden muss. Eine Erklärung dafür wäre, dass die Index-Scans auf einer anderen Maschine ausgeführt werden (die Indextabellen liegen höchstwahrscheinlich auf anderen Servern als die Entitätentabelle) und optimiert sind für Präfix-Scans, so dass für die wenigen Resourcen (Subjekte in Tab. 4.2), die zu dem Zeitpunkt des Tests gespeichert waren, kein messbarer Effekt nachzuweisen



(a) Die einzelnen Abfragen sind auf x-Achse eingetragen. Gut zu erkennen ist, wie jedes Schema sich, bis auf wenige deutlich abweichende Werte, auf einen Wert einpendelt.

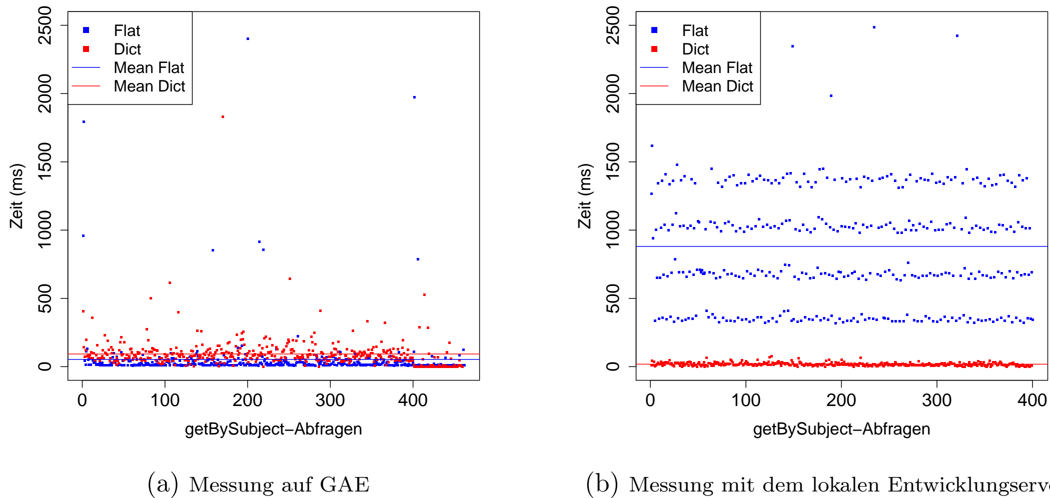
Schema	Median	Ø	Min	Max
Flat	28	31	26	192
Dict	1	13	0	368
DictBackref	1	20	0	453

(b) Die gemessene Werte bei Lesezugriffen in ms.

Abbildung 4.3: Lesezeiten von 400 Abfragen. Die Anfragen sind zufällig generiert und zeigen das erwarterte Resultat, dass die Lesezeiten für voll spezifizierte Anfragen unabhängig von der Struktur des RDF-Graphens sind. Getestet wurde gegen den Graphen aus Tabelle 4.2 mit 46027 importierten Tripeln.

```
1      PREFIX dbp:<http://dbpedia.org/resource/>
2
3  SELECT ?predicate ?object WHERE {
4    dbp:SV_Werder_Bremen ?predicate ?object .
5  }
```

Source-Code 4.1: SPARQL-Query: Gibt alle Fakten über den SV Werder Bremen zurück. Das Subjekt ist festgelegt.



(a) Messung auf GAE

(b) Messung mit dem lokalen Entwicklungserver

Abbildung 4.4: Messungen von `getBySubject`-Abfragen. Abgfragt wurden Resourcen des RDF-Graphen aus Tabelle 4.2. Jeder Punkt markiert eine gesampelte Abfrage. Bei den lokalen Messungen (Tab. 4.4b) ist es sehr deutlich, dass die Tripel innerhalb von 0 bis 1 ms geholt werden können, da diese ohne Index-Scan aufgelöst werden können.

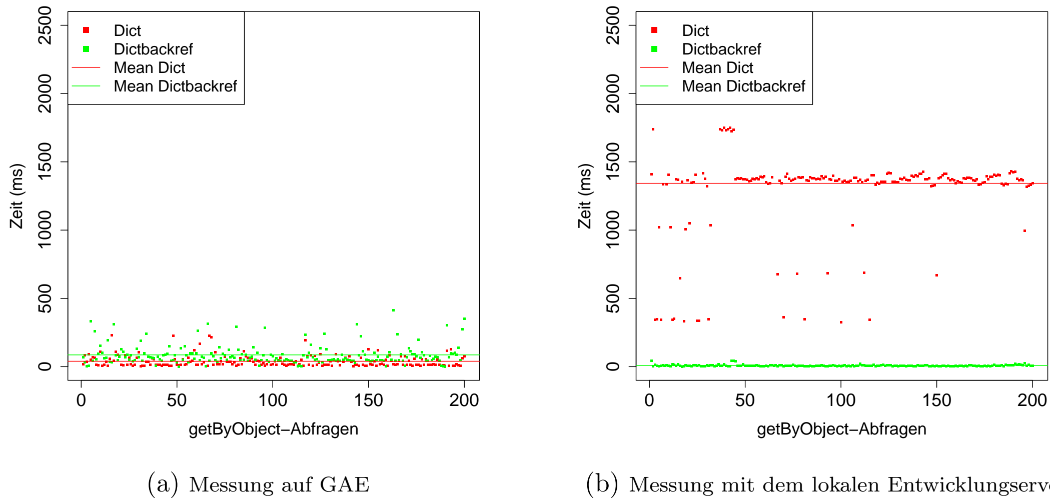
ist (Abb. 4.4a). Ein anderes Bild ergibt sich bei den lokalen Messungen, bei denen das Flat-Schema klar abfällt und das Dict-Schema deutlich schneller abschneidet (Abb. 4.4b), so wie es ursprünglich auf GAE erwartet wurde.

Abfrage von Objekten (`getByObject`:$(?, ?, 0)$)

Die Resultate dieser Messreihe sind auch repräsentativ für `getByPredicate`-Abfragen, da `getByObject`-Abfragen ebenfalls Index-Scans benötigen. Verglichen wurden das Dict-Schema und das DictBackref-Schema, da letzteres auf diese Art der Anfragen optimiert wurde und das Flat-Schema ebenso wie das Dict-Schema Index-Scans benötigt.

Überraschenderweise schneidet das Dict-Schema besser ab und das bei dieser Art der Abfrage schneller (erwartete) Dictbackref-Schema ist etwas langsamer (Abb. 4.5).

Bei den lokalen Messungen dagegen ist das Dictbackref-Schema deutlich schneller und hat keine abweichende Werte nach oben oder unten, wie es eigentlich auch auf GAE erwartet worden wäre. Es wird weiterhin vermutet, dass die importierte Tripelmenge zu klein ist, um auf GAE einen Effekt festzustellen.



(a) Messung auf GAE

(b) Messung mit dem lokalen Entwicklungserver

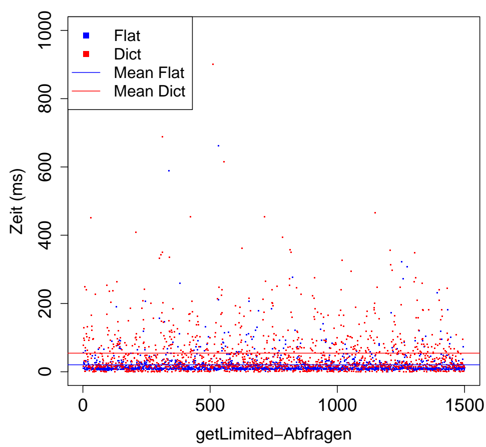
Abbildung 4.5: Messungen von `getByObject`-Abfragen. Abgfragt wurden Resourcen des RDF-Graphen aus Tabelle 4.2. Jeder Punkt markiert eine gesampelte Abfrage. Es tritt der gleiche Effekt wie bei den `getBySubject`-Abfragen auf (Abb. 4.4b). Das Auflösen über die Entitäten-Schlüssel ist auf dem lokalen Entwicklungsserver wesentlich schneller, während auf GAE kein signifikanter Geschwindigkeitsvorteil messbar ist.

Abfrage der transitiven Hülle (`getBySubjectDeep:` $((s, p, o), n)$)

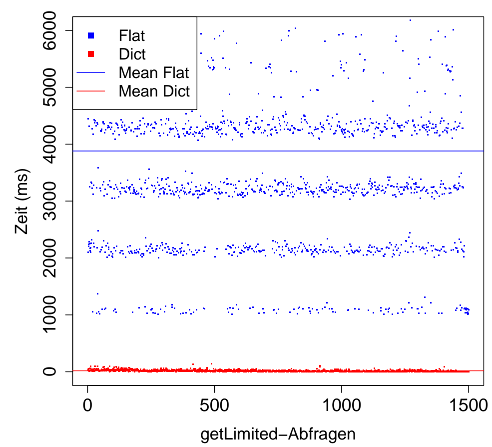
Für die Auswertung dieser Art der Abfragen wurden nur das Flat-Schema und Dict-Schema verwendet, da das Dictbackref-Schema wie das Dict-Schema ebenfalls keine Indexscans für den Aufbau der transitiven Hülle benötigt. Es wurden voll spezifizierte Tripel abgefragt und von den Treffern ausgehend eine transitive Hülle der Größe 5 aufgebaut.

Da die Index-Scans bei den `getByObject` nicht weniger performant waren, ist es nicht überraschend, dass das Flat-Schema bei diesem Test nicht schlechter abschneidet. Der zweite Vorteil, dass das Dict-Schema bei jedem Schritt im Suchbaum die nächsten Resourcen in einer Batch-Operation zusammenfassen kann, ist ebenfalls nicht auf GAE messbar. (Abb. 4.6a)

Auf Abbildung 4.6b sind die Suchtiefen des Flat-Schemas gut erkennbar. Es gibt in dem RDF-Graphen keinen längeren Suchpfad als 5 (Tab. 4.2), und die Abstände zwischen der Dauer der verschiedenen Suchtiefen ist mit ≈ 1100 ms relativ stabil. Dass die Anfragen mit Suchtiefe 1 sich beim Flat-Schema um ≈ 1100 ms einpendeln, liegt darin begründet, dass ausgehend von den gefundenen Resourcen immer noch mindestens ein Index-Scan durchgeführt werden muss, um zu prüfen, ob die Resource aufgelöst werden kann.



(a) Messung auf GAE



(b) Messung mit dem lokalen Entwicklungserver

Abbildung 4.6: Messergebnisse von getLimited-Abfragen. Abgefragt wurden 1500 zufällig ausgewählte Tripel aus dem RDF-Graphen der Tabelle 4.2 mit 84750 gespeicherten Tripeln. Jeder Punkt markiert eine gesampelte Abfrage.

5 Fazit

5.1 Kritische Betrachtung des Prototypen

Die Datenbankschemata konnten ihre Vorteile nur auf der lokalen Testinstallation ausspielen. Das Dictbackref-Schema schneidet beim Schreiben von Tripeln am schlechtestens ab und das Dict-Schema erzeugt höhere Schreibkosten pro RDF-Tripel (Tab. 3.1) als das Flat-Schema. Aus diesem Grund ist das Flat-Schema als Standard im Prototypen gesetzt. Es ist aber nicht auszuschließen, dass bei größeren RDF-Graphen das Flat-Schema bei unterspezifierten Anfragen und Aufbau der transitiven Hülle langsamer ist. Um eine kritische Anzahl an Tripeln zu speichern, damit diese Nachteile des Flat-Schemas sich nachweisbar auswirken, muss wohl auf einen Billing-Account zurückgegriffen werden und somit Teil zukünftiger Untersuchungen bleiben. So konnte der Nachweis, dass sich GAE für einen skalierenden Triplestore eignet, mit diese Arbeit nicht erbracht werden, dazu wäre der Einsatz von Geldmitteln notwendig. Die Vermutung liegt nahe, dass GAE auch mit sehr großen Datenmengen umgeben kann, da die darunter liegende Technologie Bigtable ein hochskalierendes System ist.

Der Prototyp wurde hauptsächlich entwickelt, um festzustellen wie gut sich die NoSQL-Datenbank, welche GAE mit Googles Bigtable bereitstellt, für RDF-Daten eignet. Es gibt noch keinen praktischen Anwendungsfall des Prototypens. Konkrete Anwendungen könnten weitere Engpässe aufdecken, als in dieser Arbeit behandelte.

Im Folgenden werden eine Reihe von *Features* behandelt, die sich nicht unmittelbar aus den funktionalen Anforderungen für die Testfälle ergeben haben (deswegen gar nicht oder nur teilweise umgesetzt), die aber für den praktischen Nutzen des Prototypen relevant sein könnten.

5.1.1 Persistierung multipler Graphen

Das Speichern von unterschiedlichen Graphen ermöglicht das mehrfache Speichern identischer Tripel in verschiedenen Kontexten. Möglich wäre das durch die Einführung eines Graph-Labels, das die Sichtbarkeit von Tripeln einschränkt. Mehrere Vorteile ergeben sich aus diesem Konzept:

1. Es lassen sich mehrere RDF-Graphen mit teilweise identischen Tripeln erstellen und folglich unterschiedliche Weltansichten in einer Instanz des Prototypen modellieren.
2. Tripel können in einer Batch-Operation wesentlich leichter gelöscht oder hinzugefügt werden.
3. Blank Nodes können innerhalb einer Transaktion geschrieben werden.
4. Es können beliebig Tripel innerhalb einer Transaktion geschrieben werden, was die Schreibrate pro Tripel vermutlich erhöht. Allerdings gilt dies nicht für gleichzeitige Schreibvorgänge.

Dieses Konzept nennt sich auch *Named Graphs* und ist dem `N-Quad`-Format entlehnt.

Umsetzung im Prototypen

Im Prototypen werden alle Tripel, die ohne Kontext gespeichert werden, in dem Standard-Graphen gespeichert, so wie bisher auch. Wird ein Kontext angegeben, so wird dieser beim Flat-Schema in dem Tripel-POJO (Source-Code 3.3) in einem zusätzlichen Feld gespeichert. Der Schlüssel (ID) wird nun aus dem MD5-Hash aller vier Felder bestimmt, um eindeutig zu bleiben. Ähnlich wird bei den verschachtelten Datenbankschemata vorgegangen. Dort wird der Schlüssel für die Resourcen aus der Konkatenierung von Subjekt und Kontext erzeugt. Falls der Schlüssel zu lang ist, wird er ebenfalls gehashed. Die Facts müssen nicht angepasst werden, da sich ihr Schlüssel aus dem Resourcen-Schlüssel des Subjekts ergibt und damit transitiv vom Graphnamen abhängt.

Schreiben von Blank Nodes innerhalb von Transaktionen

Durch das globale Modell eines sehr großen RDF-Graphen im Datastore, können nicht beliebig verkettete Blank Nodes innerhalb einer Transaktion geschrieben werden, was zu Problemen führt, wenn es zu Schreibabbrüchen kommt (Abs. 3.6.2). Mit der Einführung von Named-Graphs könnte eine Wurzel-Entität für jeden Graphen definiert werden, so dass das Schreiben von Minimum-Selfcontained-Graphs(MSG) innerhalb einer Transaktion durchgeführt werden kann. Diese Art der Speicherung hätte zur Folge, dass transaktionale, gleichzeitige Schreibzugriffe nur noch sequentiell auf den gesamten RDF-Graphen durchgeführt werden können.

5.1.2 Erweiterung der API für Limit- und Offset-Anfragen

Der Prototyp unterstützt derzeit keine Begrenzung von Abfragen. Diese wäre sinnvoll, wenn z. B. alle Tripel mit dem Prädikat `foaf:name` abgefragt werden. Diese Abfrage würde bei einem Graphen aus Tabelle 4.2 mehrere tausend RDF-Tripel zurückliefern und damit die Lesekosten des Free-Accounts mit einer Abfrage aufbrauchen. Günstiger wäre ein Verhalten des Prototypens, das standardmäßig ein Limit und ein Offset für Abfragen setzte und diese nur durch explizite Parametrisierung aufgehoben werden könnte, damit nicht per Zufall hohe Lesekosten entstehen. Dies ist aber noch nicht umgesetzt.

5.1.3 Speichern von großen Literalen in Blobs

Die GAE-API beschränkt die Größe von Enititäten auf ein Megabyte. Enthält der RDF-Graph große literarische Texte, muss auf die Blobstore-API ausgewichen werden, bei der große binäre Objekte gespeichert werden können. Entitäten können Schlüssel für Blob-Objekte nativ speichern.

5.2 Ausblick

Im Unterschied zu den Punkten, die in der kritischen Betrachtung aufgeführt wurden, sind die hier diskutierten Themen nicht unmittelbar aus den Anforderungen abzuleiten.

5.2.1 Implementierung eines SPARQL-Endpoints

Ein SPARQL-Endpoint ermöglicht das Abfragen von Triplestores mit der Abfragesprache SPARQL via HTTP. Dafür wurde vom W3C ein Protokoll spezifiziert.[1] Da der Prototyp das Jersey-Framework einsetzt, ist die Implementierung dieses Protokolls nicht der aufwändige Teil.

Eine größere Herausforderung stellt es dar SPARQL-Abfragen zu prozessieren. Hierzu könnte auf die SPARQL-API von Jena zurückgegriffen werden, da dieses Framework im Prototypen bereits zum Einsatz kommt. Jena führt SPARQL-Queries auf das interne Jena-Modell aus, das wiederum an eine spezifische Graphimplementierung gekoppelt ist. Eine Implementierung des Graph-Interface[2], die Methoden auf das DAO des Prototypen delegiert, könnte eine Annäherung an die Lösung dieses Problems sein. An dieser Stelle ist weitere Evaluation nötig.

[1] `http://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/` (abgerufen 15.07.2013)

[2] `http://jena.apache.org/documentation/javadoc/jena/com/hp/hpl/jena/graph/Graph.html` (abgerufen 31.07.2013)

5.2.2 Performance Tests auf Open-Source Implementierung des GAE-Interface

Durch Cloud-Computing ist ein neuer, versatiler Markt entstanden, in den viele Anbieter drängen. Daraus lässt sich die Frage ableiten, wie gut sich eine auf GAE optimierte Software portieren lässt, wenn die Preispolitik oder andere Faktoren den Betrieb einer Applikation mit GAE nicht mehr opportun erscheinen lassen oder unmöglich machen.

Grundsätzlich ist es möglich die Applikation in dem lokalen Entwicklungsserver von Google laufen zu lassen. Dieser Ansatz skaliert jedoch nicht horizontal, d. h. der Entwicklungsserver besitzt keine API, um Load-Balancing zu betreiben. Mit dieser Lösung gibt es jedoch keine Vorteile die Applikation gegen die GAE-API zu entwickeln.

Eine weitere Möglichkeit ist AppScale[3]. Diese ermöglicht GAE-Applikation auf mehrere virtuelle Maschinen zu verteilen oder in anderen PaaS-Umgebungen laufen zu lassen, z.B. Amazon EC2 oder Rackspace. Appscale benutzt zur Persistierung den lokalen Datenspeicher des Appengine-SDKs, das auch für die lokalen Tests in dieser Arbeit verwendet wurde, und nicht Cassandra[4]. Damit gelten die Erkenntnisse dieser Arbeit auch für Applikationen, die mit Appscale betreiben werden.

Die letzte Alternative, die zu dem Zeitpunkt dieser Arbeit angeboten wird ist CapeDwarf[5]. Das ist eine Laufzeitumgebung für GAE-Applikationen in einem JBOSS-Server[6] oder in der OpenShift-Cloud von RedHat. Eine Testinstallation scheiterte, weil JBoss einen Fehler bei der Erzeugung von Jersey-Servlets besaß.

Es gibt also einige Entwicklungen, die sich damit beschäftigen, GAE-Apps ohne Portierungsaufwand in einer anderen Umgebungen laufen zu lassen. Diese Projekte stehen aber noch am Anfang und sind für den produktiven Einsatz noch nicht gedacht. Sollten diese Open-Source-Implementierung stabil sein und andere Technologien zum Persistieren anbieten als den lokalen Datenspeicher des Appengine-SDKs, sollten die Performanztests auch auf diesen Plattformen durchgeführt werden.

[3] `http://www.appscale.com/` und der Quellcode: `https://github.com/AppScale` (abgerufen 15.07.2013)

[4] Cassandra ist ein Open-Source-Klon von Googles Bigtable. Appscale setzt bei der Installation Cassandra Nodes auf, benutzt sie allerdings nicht für GAE-Apps.

[5] `http://www.jboss.org/capedwarf` (abgerufen 15.7.2013)

[6] Der Server heißt mittlerweile wildfly `http://wildfly.org/` (abgerufen 15.7.2013)

Literatur

[1] Dave Beckett und Jan Grant. *RDF Test Cases*. Recommendation. http://www.w3.org/TR/2004/REC-rdf-testcases-20040210/. Latest version available at http://www.w3.org/TR/rdf-testcases. W3C, Feb. 2004.

[2] Jeremy J. Carroll und Graham Klyne. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. Recommendation. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/. Latest version available at http://www.w3.org/TR/rdf-concepts/. W3C, Feb. 2004.

[3] Fay Chang u. a. „Bigtable: a distributed storage system for structured data“. In: *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*. OSDI ’06. Seattle, WA: USENIX Association, 2006.

[4] Roy Thomas Fielding. „Architectural styles and the design of network-based software architectures“. AAI9980887. Diss. 2000. ISBN: 0-599-87118-0. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[5] A. Kemper und A. Eickler. *Datenbanksysteme: Eine Einführung*. Oldenbourg Wissenschaftsverlag, 2011. ISBN: 9783486598346. URL: http://books.google.de/books?id=xpNefMq5nYwC.

[6] Günter Ladwig und Andreas Harth. „CumulusRDF: Linked Data Management on Nested Key-Value Stores“. In: *Proceedings of the 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2011) at the 10th International Semantic Web Conference (ISWC2011)*. 2011.

[7] Christina Lantzaki, Yannis Tzitzikas und Dimitris Zeginis. „Demonstrating Blank Node Matching and RDF/S Comparison Functions“. In: *International Semantic Web Conference (Posters & Demos)*. 2012.

[8] Marcus Nitzscke. „Implementierung einer RDF-Storage Lösung für MongoDB“. Bachelor Thesis. Universität Leipzig, 2011. URL: http://www.kendix.org/media/files/thesis.pdf.

[9] Tim Grance Peter Mell. *The NIST Definition of Cloud Computing.* Techn. Ber. National Institute of Standards und Technology, Information Technology Laboratory, 2011.

[10] Eric Prud'hommeaux und Andy Seaborne. *SPARQL Query Language for RDF.* Recommendation. `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`. Latest version available at `http://www.w3.org/TR/rdf-sparql-query/`. W3C, Jan. 2008.

[11] W. Streitberger und A. Ruppel. *Cloud Computing Sicherheit - Schutzziele.Taxonomie.Marktübersicht.* Fraunhofer SIT, Darmstadt, 2009, S. 5.

[12] Aaron Swartz. *Aaron Swartz's A Programmable Web: An Unfinished Work.* Morgan & Claypool, 2013, S. 1.

[13] Ralph R. Swick und Ora Lassila. *Resource Description Framework (RDF) Model and Syntax Specification.* Recommendation. `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222`. Latest version available at `http://www.w3.org/TR/REC-rdf-syntax`. W3C, Feb. 1999.

Abbildungsverzeichnis

1.1 Szenario 9

2.1 Einfache Aussagen als gerichteten Graphen abbilden. 12
2.2 Verkettung 12
2.3 Zyklischer Graph 13
2.4 Formaler RDF-Graph 15
2.5 Präfix- und Rangescans 24

3.1 Mapping zwischen internem Datenmodell und Eingabeformat sowie Datenbankschemata 28
3.2 Detailierter Überblick über den Prototypen 29
3.3 Verschachteltes Datenbankschema 37
3.4 Dict-Schema 39
3.5 Entscheidung beim Einfügen von Blank Nodes 47
3.6 Falscher RDF-Graph 48

4.1 Test-Pipeline 51
4.2 Persistierung von Tripel-Chunks der Größe 200. 53
4.3 Plot von 400 `get`-Abfragen 58
4.4 Messungen von getBySubject-Abfragen 59
4.5 Messungen von getByObject-Abfragen 60
4.6 Messungen von getLimited-Abfragen 61

Tabellenverzeichnis

2.1 GAE Datenbanklimitierungen 20

3.1 Anzahl der zu speichernden Entitäten 40
3.2 Kosten für das Schreiben eines Tripels 43
3.3 GAE Preise für Datenbankoperationen 45

4.1 Spezifikation für lokale Tests. 52
4.2 RDF-Graphstruktur eines Subgraphen des Datensatzes. 54
4.3 Lese- und Schreibkosten in Abhängigkeit der gespeicherten Tripel 55
4.4 Gleichzeitige Schreibzugriffe 56

Source-Code-Verzeichnis

2.1 Beispiele für Präfixe 14
2.2 Beispiel für getaggte Literale 14
2.3 Tiefe SPARQL-Abfrage 17
2.4 Entitäten Schlüssel 22

3.1 Beispiel für eine Implementierung eines HTTP-PUT-Request mit dem Jersey-Framework. 31
3.2 POJOs in Objectify 32
3.3 POJO für das flaches Datenmodell. 36
3.4 Die POJOs für das Nested-Schema. 38
3.5 POJO für die Repräsentation eines Fakts über eine Resource. 39
3.6 POJO für die Modellierung von Referenzen auf Prädikate. 41
3.7 Modelliert eine Person Bob, die eine unbekannte Person gesehen hat. Die unbekannte Person kennt wiederum Alice und Sally. 47
3.8 Modelliert eine Person Bob, die zwei unbekannte Personen gesehen hat. Diese unbekannten Person kennen beide Alice. 48

4.1 SPARQL-Query 58

Symbolverzeichnis

AED App Engine Datastore, Seite 19

API Application Programming Interface, Seite 8

BFS Broad First Search, Seite 37

BNF Backus-Naur Form, Seite 33

CRUD CREATE, READ, UPDATE, DELETE, Seite 26

DAO Data Access Object, Seite 29

DBMS Datenbank Management Software, Seite 19

DDL Data Definition Language, Seite 35

DML Data Manipulation Language, Seite 35

EBS Amazon Elastic Block Store, Seite 26

FOAF Friend-Of-A-Friend, Seite 13

GAE Google App Engine, Seite 7

GRDDL Gleaning Resource Descriptions from Dialects of Languages, Seite 7

HTTP Hypertext Transfer Protocol, Seite 26

IDE Integrated Development Environment, Seite 30

IP Internet Protocol, Seite 26

JAX-RS Java API for RESTful Web Service, Seite 30

JRE Java-Runtime-Environment, Seite 25

JSON Javascript Simple Object Notation-Linked Data, Seite 16

JSON-LD Javascript Simple Object Notation, Seite 16

k Kilo, Seite 45

MB Mega Byte, Seite 42

ms Millisekunden, Seite 52

MSG Minimum-Selfcontained-Graphs, Seite 63

N3 Notation3, Seite 15

NP Nichtdeterministische Polynomialzeit, Seite 14

OAuth OAuth 2.0 authorization protocol, Seite 19

OWL 2 Web Ontology Language 2, Seite 7

PaaS Platform as a Service, Seite 18

POJO Plain Old Java Object, Seite 31

POWDER Protocol for Web Description Resources, Seite 7

RDBMS Relationales Datenbank Management System, Seite 20

RDF Resource Description Framework, Seite 6

REST Representational State Transfer, Seite 9

RIF Rule Interchange Format, Seite 7

s Sekunden, Seite 51

S3 Simple Storage Service, Seite 7

SCM Source Code Management, Seite 30

SOAP Simple Object Access Protocol, Seite 27

SPARQL SPARQL Protocol and RDF Query Language, Seite 7

SQL Structured Query Language, Seite 20

TCP Transmission Control Protocol, Seite 26

Tripel Steht für eine Aussage (Statement) in RDF, Seite 6

Turtle Terse RDF Triple Language, Seite 15

URI Uniform Resource Identifier, Seite 13

UUID Universally Unique Identifier, Seite 35

VCARD elektronische Visitenkarte, Seite 14

WADL Web Application Description Language, Seite 30

XML Extensible Markup Language, Seite 14

XMPP Extensible Messaging and Presence Protocol, Seite 19