Bachelorarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Prüfsummenberechnung bei RDF-Daten

Thomas Pilger

Matrikelnummer: 3694335

pilger@inf.fu-berlin.de

Betreuer: Dr. Edzard Höfig

Eingereicht bei: Prof. Dr. Ina Schieferdecker

Zweitgutachten: Prof. Dr. Elfriede Fehr

Berlin, 11. Januar 2013

Zusammenfassung

Das *Resource Description Framework* (RDF) ist eine wichtige Komponente des *Semantischen Webs*, um von Menschen zusammengetragene Informationen in für Maschinen interpretierbare Daten umzuwandeln. Durch das heute allgegenwärtige Internet können sehr viele Personen Zugang zu Daten bekommen. Es besteht allerdings die Gefahr, dass die Daten manipuliert und dadurch verfälscht werden. Eine Möglichkeit, diesem Problem zu begegnen, besteht in der Erstellung von Prüfsummen.

Diese Bachelorarbeit beschäftigt sich mit Möglichkeiten, Prüfsummen direkt aus RDF-Daten zu berechnen. Die Möglichkeiten sollen nicht auf einem speziellen Repräsentationsformat basieren. Aufbauend darauf wird ein Algorithmus vorgestellt, der dazu geeignet ist, einen RDF-Graph im Arbeitsspeicher eines Computers zu hashen.

Außerdem erfolgt eine prototypische Implementierung des herausgearbeiteten Algorithmus in Java. Diese Implementierung wird durch Messung von Laufzeit und Speicherplatzverbrauch auf ihre Praxistauglichkeit getestet.

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, 11. Januar 2013

Thomas Pilger

Inhaltsverzeichnis

Abkürzungsverzeichnis

GUI	Graphical User Interface
ID	Identifikationsnummer
I/O	Input/Output
KiB	Kibibyte
MD5	Message-Digest Algorithm 5
MiB	Mebibyte
MSG	Minimum Self-contained Graph
NIST	National Institute of Standards and Technology
NSA	National Security Agency
RDF	Resource Description Framework
SHA	Secure Hash Algorithm
SWP	Semantic Web Publishing
UML	Unified Modeling Language
URI	Uniform Resource Identifier
VM	Virtual Machine
W3C	World Wide Web Consortium
XOR	eXclusive OR
XML	eXtensible Markup Language

1 Einführung

1.1 Motivation

In der heutigen Zeit ist es üblich, fast alle verfügbaren Daten online in das *World Wide Web* zu stellen. Zu diesem Zweck gibt es diverse Möglichkeiten ihrer Präsentation, wie Webseiten, Blogs oder soziale Netzwerke. Unter Berücksichtigung der Entwicklungen der letzten Jahre, wie Cloud-Computing oder mobiles Breitband-Internet, wird sich dieser Trend aller Wahrscheinlichkeit nach fortsetzen oder vielleicht sogar noch weiter verstärken.

Dadurch, dass unterschiedliche Arten von Daten online verfügbar gemacht werden, sind jetzt schon unvorstellbar große Datenmengen entstanden. Diese können oft nur durch Menschen interpretiert werden. Damit auch ein Computer diese Daten verarbeiten kann, werden Metadaten benötigt. Genau auf diesen Umstand zielt das Konzept des *Semantischen Webs* ab. In diesem Rahmen ist eine mögliche Beschreibungssprache für Daten das *Resource Description Framework* (RDF).

Sehr viele Menschen haben Internetzugriff und somit auch Zugang zu RDF-Daten. Deshalb ist es notwendig, soweit möglich, eine Manipulation dieser Daten zu verhindern. Wenn das nicht möglich ist, ist eine Manipulation der Daten aufzudecken. Dies wird üblicherweise durch Prüfsummen erreicht. Außerdem ist sicherzustellen, dass die RDF-Daten tatsächlich von einer bestimmten Person stammen und nicht von einer anderen, die sich nur als diese ausgibt. Zu ihrer Identifizierung werden derzeit Signaturen benutzt.

1.2 Ziele

Es ist das Hauptziel dieser Arbeit, einen Algorithmus zur Erstellung einer Prüfsumme aus RDF-Daten zu finden und zu beschreiben. Darauf aufbauend könnte eine spätere Arbeit einen Algorithmus zur elektronischen Signatur von RDF-Daten zur Verfügung stellen, da für die Signierung eine Prüfsumme benötigt wird. Die Möglichkeiten zur Signierung oder Verschlüsselung von RDF-Daten werden in meiner Arbeit nicht behandelt.

Des Weiteren ist das Ziel, den Algorithmus zur Erstellung der Prüfsumme so zu gestalten, dass er auch in der Praxis eingesetzt werden kann. Daher sollte er ermöglichen, dass auch Teildaten eine Prüfsumme erhalten, damit eine Person, die lediglich einen kleinen Teil der RDF-Daten benötigt, nicht den kompletten

Datensatz herunterladen und prüfen muss. Das kann bei großen Datenmengen sehr lange dauern.

Für RDF-Daten gibt es verschiedene Möglichkeiten der Repräsentation in Dateien, z. B. durch N-Triples oder RDF/XML. Die Prüfsumme ist für jedes einzelne Format unterschiedlich. Daher wird angestrebt, die Prüfsummen von RDF-Daten im Speicher eines Computers zu berechnen, so dass eine Unabhängigkeit dieses Repräsentationsformats gegeben ist. Es ist außerdem zu erwarten, dass die Geschwindigkeit, mit der die Prüfsummen berechnet werden können, höher ist, wenn mit Daten im Speicher gearbeitet wird.

Um die Praxistauglichkeit des Algorithmus zu testen, wird im Rahmen dieser Arbeit ein Prototyp implementiert und die Laufzeit sowie der Speicherplatzbedarf verschiedener Hashfunktionen und Testdatensätze getestet. Die Testdatensätze entnehme ich von DBpedia in der Version 3.8[1]. DBpedia extrahiert Informationen von Wikipedia[2] und stellt diese als RDF-Datensätze in unterschiedlichen Dateiformaten zentral auf einer Webseite bereit. Ich habe auch RDF-Datensätze auf anderen Webseiten gefunden, jedoch beinhalten diese nur eine geringe Anzahl von Tripeln. DBpedia bietet eine sehr große Anzahl verschiedener RDF-Datensätze in unterschiedlichen Größen an. Daher habe ich mich für DBpedia entschieden.

1.3 Aufbau der Arbeit

In Kapitel 2 werden Grundlagen wie der RDF-Standard sowie kryptographische Hashfunktionen angesprochen und über verwandte Arbeiten berichtet, während sich Kapitel 3 mit dem Algorithmus zur Prüfsummenerstellung von RDF-Daten beschäftigt. Die Implementierung und das Testen dieses Algorithmus wird in Kapitel 4 durchgeführt. Im abschließenden Kapitel 5 werden die Erkenntnisse diskutiert und zusammengefasst. Außerdem gibt es einen Ausblick auf mögliche zukünftige Arbeiten.

[1] http://wiki.dbpedia.org/Downloads38
[2] http://www.wikipedia.org/

2 Grundlagen

2.1 Das Resource Description Framework (RDF)

Der RDF-Standard[1–6] wurde vom *World Wide Web Consortium* (W3C)[3] entwickelt und ist eine Beschreibungssprache für Ressourcen im Internet. Hauptsächlich besteht die RDF-Sprache aus Aussagen, die jeweils aus Subjekt, Prädikat und Objekt bestehen. Hierbei bezeichnet das Subjekt die Ressource, über die eine Information gegeben wird, das Prädikat die Eigenschaft des Subjekts und das Objekt den Wert dieser Eigenschaft. Prinzipiell sind diese RDF-Aussagen wie Schlüssel-Wert-Paare in einer Hashtabelle zu betrachten. Subjekt, Prädikat und Objekt sollten jeweils aus einem *Uniform Resource Identifier* (URI) bestehen. Jedoch können Subjekt und Objekt auch eine anonyme Ressource sein (nur mit einer ID versehen) und das Objekt zusätzlich auch ein Literal.

Trotz der Einfachheit des RDF gibt es eine Reihe von Darstellungsmöglichkeiten und Repräsentationsformaten. Eine Menge von RDF-Aussagen kann als gerichteter Graph (siehe Abbildung 1) oder als Menge von Tripeln in der Tripelnotation *N-Triples*[4] (siehe Abbildung 2) dargestellt werden.

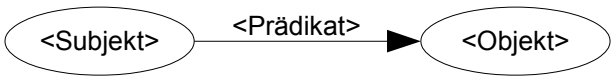

Abbildung 1: Konzept eines RDF-Tripels als gerichteter Graph

In der Darstellung als Graph sind die Subjekte und Objekte die Knoten und die Prädikate die Kanten. Wenn ein Subjekt oder Objekt eine anonyme Ressource sein sollte, so wird diese als leerer Knoten dargestellt.

```
<Subjekt> <Prädikat> <Objekt> .
```

Abbildung 2: Konzept eines RDF-Tripels in Tripelnotation (N-Triples)

In der Tripelnotation wird jede RDF-Aussage als ein Tripel aus Subjekt, Prädikat und Objekt dargestellt. Hierbei ist die Reihenfolge zu beachten. Am Anfang steht immer das Subjekt, dann folgt das Prädikat und am Ende das Objekt. Anonyme Ressourcen werden in der Tripelnotation durch eine lokal eindeutige

[3] http://www.w3.org/
[4] http://www.w3.org/TR/rdf-testcases/#ntriples

ID beschrieben. Die Eindeutigkeit muss nur innerhalb eines RDF-Datensatzes bestehen.

RDF-Daten können in verschiedenen Repräsentationsformaten gespeichert werden. Zum Beispiel können die RDF-Daten in Tripelnotation oder im RDF/XML-Format in eine Datei gespeichert werden. Für die vorliegende Arbeit ist es nicht relevant, welches Dateiformat gewählt wird, da es für den Algorithmus zur Prüfsummenerstellung nur auf die Repräsentation im Speicher ankommt.

Es ist wichtig, dass zur Erstellung einer Prüfsumme alle RDF-Aussagen komplett berücksichtigt werden. Es darf kein Teil einer Aussage weggelassen werden, da sonst die Manipulation der Daten erheblich einfacher wird.

2.2 Kryptographische Hashfunktionen

Es wurden in den vergangenen Jahren viele kryptographische Hashfunktionen entwickelt. Klaus Schmeh nennt in seinem Buch über Kryptographie[7] ca. 20 verschiedene kryptographische Hashfunktionen und deren Varianten. Populäre Vertreter von kryptographischen Hashfunktionen sind z. B. der *Message-Digest Algorithm 5* (MD5), entwickelt von Ronald Rivest oder der *Secure Hash Algorithm-1* (SHA-1), der vom *National Institute of Standards and Technology* (NIST) und der *National Security Agency* (NSA) entwickelt wurde.

Es ist optimal, wenn kryptographische Hashfunktionen für zwei verschiedene beliebige Eingaben stets auch unterschiedliche Hashwerte ausgeben. Falls für zwei verschiedene Eingaben gleiche Hashwerte auftreten, nennt man dies eine *Kollision*. Eine Kollision ist ein mögliches Einfallstor für einen Angreifer.

Kryptographische Hashfunktionen bilden unendlich viele Eingaben auf endlich viele Hashwerte ab. Daher ist es nicht möglich, Kollisionen komplett zu vermeiden. Zum Beispiel bildet der MD5 eine beliebige Eingabe auf einen Hashwert von 128 Bit Länge ab. Beim SHA-1 sind es 160 Bit.

In der Regel steigt die Sicherheit der Hashwerte mit ihrer Länge. Allerdings gilt das nur, wenn der Algorithmus keinen effizienteren Angriff zulässt als einen *Geburtstagsangriff*. Der Geburtstagsangriff dient hier als Vergleichsmaßstab.

Basiert ein Angriff auf dem Geburtstagsproblem, nennt man ihn Geburtstagsangriff. Beim Geburtstagsproblem lautet die Fragestellung: „Wie viele Menschen müssen in einem Raum versammelt sein, damit mit einer Wahrscheinlichkeit von 1/2 mindestens zwei davon am gleichen Tag Geburtstag haben?“[7, S. 212] Dies ist schon bei ungefähr 22 Menschen der Fall. Man kann das Geburts-

tagsproblem auch verallgemeinern. Bei t Tagen im Jahr „beträgt die Anzahl der Menschen, die zusammenkommen müssen, damit die Wahrscheinlichkeit eines gleichen Geburtstags 1/2 ist, stets etwas mehr als die Wurzel von t.“[7, S. 213]

Überträgt man das Geburtstagsproblem auf kryptographische Hashfunktionen, geht es um die Frage, wie viele Eingaben durchprobiert werden müssen, um mit einer Wahrscheinlichkeit von 1/2 eine Kollision zu erhalten. Dies kann analog wie folgt berechnet werden. Die Anzahl der durchschnittlich zu probierenden Eingaben, um eine Kollision zu finden, ist ungefähr die Wurzel aus der Anzahl der möglichen Hashwerte. Bei einem 128 Bit langen Hashwert wie dem MD5 sind das im Schnitt ca. $\sqrt{2^{128}} = 2^{64}$ Versuche. Beim SHA-1 mit 160 Bit sind das ca. 2^{80}.

Bei den beiden vorgenannten Hashfunktionen wurden jedoch effizientere Angriffe als der Geburtstagsangriff entdeckt. Laut Vlastimil Klíma[8] kann für einen MD5-Hash auf einem PC schneller als in einer Minute eine Kollision gefunden werden. Schmeh schreibt treffend zu Klímas Arbeit: „MD5 ist damit endgültig tot.“[7, S. 224]

Auch für den SHA-1 ist eine Methode entwickelt worden, wie Kollisionen effizienter generiert werden können als durch einen Geburtstagsangriff[9]. Diese Methode, eine Kollision zu berechnen, erfordert momentan noch eine zu lange Rechenzeit, um sie in der Praxis effektiv einzusetzen. Bevor eine Kollision berechnet wäre, wäre die zu manipulierende Information bereits überholt. Es ist aber durchaus realistisch, dass in den nächsten Jahren durch weitere Verbesserungen der Methode und der Rechenleistung eine Kollision innerhalb weniger Tage oder gar Stunden berechnet werden kann. Daher sollten diese beiden Hashfunktionen nicht mehr benutzt werden.

Das NIST veröffentlichte, noch bevor diese Angriffe bekannt wurden, weitere SHA-Versionen: SHA-224, SHA-256, SHA-384 und SHA-512. Diese vier Algorithmen gehören zur so genannten SHA-2 Familie von Hashfunktionen. Die Zahl in ihrem Namen steht jeweils für die Länge des Hashwerts in Bit. Obwohl sie viele Gemeinsamkeiten mit dem SHA-1 besitzen, gelten Sie derzeit als sicher.

Nichtsdestotrotz waren diese Algorithmen für das NIST anscheinend immer noch nicht ausreichend. Am 2. November 2007 wurde ein Wettbewerb für eine neue kryptographische Hashfunktion gestartet. Dieser Wettbewerb ist nun beendet und am 2. Oktober 2012 wurde der SHA-3 Gewinner veröffentlicht[10]. Der SHA-3 kann, ähnlich wie der SHA-2, für unterschiedlich lange Hashwerte implementiert werden.

2.3 Verwandte Arbeiten

In der Literatur sind bereits einige Arbeiten zu dem Thema zu finden, wie RDF-Graphen gehasht bzw. signiert werden können. Aber im Vergleich zu anderen Themen in der Informatik gibt es zu diesem Thema eher wenig Literatur.

Es gibt zum Beispiel eine Arbeit von Jeremy J. Carroll[11], die sich mit der Signierung von RDF-Graphen beschäftigt. In seiner Arbeit wird ein Algorithmus vorgestellt, der RDF-Graphen in eine Normalform bringt, die dann als Grundlage für das Signieren benutzt werden kann. Dieser Algorithmus arbeitet auf RDF-Daten in Form von N-Triples. Da dieser Ansatz auf einem speziellen Dateiformat basiert, kann er für die vorliegende Arbeit nicht gewählt werden. Denn diese Arbeit beschäftigt sich mit der repräsentationsunabhängigen Prüfsummenerstellung aus RDF-Daten.

Daneben gibt es eine Arbeit von Giovanni Tummarello u.a.[12], in der die Unterteilung eines RDF-Graphen in *Minimum Self-contained Graphs* (MSGs) beschrieben wird. MSGs sind zusammenhängende Teilgraphen eines RDF-Graphen. Diese können dann gehasht und signiert werden. Das ist eine Möglichkeit, die für die vorliegende Arbeit gewählt werden kann. Diese Idee ist zwar nicht ideal für die Speicherung der Prüfsummen, dennoch werde ich sie in meiner Arbeit verwenden. Auf die Art und Weise der Verwendung werde ich im weiteren Verlauf meiner Arbeit noch eingehen.

Mark Giereth[13] beschreibt eine Methode, wie sensible Daten innerhalb eines RDF-Graphen verschlüsselt werden können. Dabei bleiben die unkritischen Daten für jeden lesbar. In seiner Arbeit wird der zu verschlüsselnde RDF-Graph zuerst in ein frei wählbares Repräsentationsformat für RDF-Daten serialisiert. Nach Verschlüsselung und Transformation der serialisierten sensiblen Daten, werden diese wieder in den Graphen eingearbeitet. Das Ergebnis ist ein RDF-konformer Graph. Der Schwerpunkt von Giereths Arbeit ist die Verschlüsselung von Teilen eines RDF-Graphen. In meiner Arbeit ist das aber kein Thema. Daher werden die Ergebnisse von Giereths Arbeit nicht verwendet.

Ein weiterer Ansatz wurde von Craig Sayers und Alan H. Karp[14, 15] entwickelt. In ihren Arbeiten beschreiben sie, wie ein RDF-Graph im Speicher gehasht werden kann. Dabei werden für die einzelnen RDF-Tripel Teilhashes errechnet, die durch ihre inkrementelle Verknüpfung den gesamten Hash ergeben. Dieser Ansatz wird für die vorliegende Arbeit gewählt, da er sich durch seine Flexibilität auszeichnet.

Carroll u.a. beschreiben in [16] die Idee, den RDF-Standard um so genannte *Named Graphs* zu erweitern. Dabei werden die normalen RDF-Graphen mit einem Namen versehen. Auch diese Idee wird in der vorliegenden Arbeit benutzt, da sie eine Möglichkeit bietet, Teile von RDF-Graphen konkret zu adressieren.

3 Ein Algorithmus zur Prüfsummenerstellung

Mit den in Kapitel 2 dargestellten Grundlagen, wird in diesem Kapitel ein Algorithmus entwickelt, der es ermöglicht, Prüfsummen aus RDF-Daten im Arbeitsspeicher eines Computers zu errechnen und im RDF-Graphen zu speichern. Zunächst gehe ich darauf ein, wie die Hashwerte berechnet werden können, dann folgen Möglichkeiten zu deren Speicherung und schließlich wird der ausgewählte Algorithmus als Ganzes beschrieben und eine theoretische Analyse seiner Laufzeit vorgenommen.

3.1 Inkrementelles Hashing von RDF-Tripeln

Es ist ein Ziel dieser Arbeit, Prüfsummen aus Teilen von RDF-Daten zu bilden und diese zu speichern. Die Voraussetzung dafür ist ein ausreichend flexibler Algorithmus für die Prüfsummenerstellung. Der von Sayers und Karp in [14] vorgeschlagene Algorithmus erfüllt diese Voraussetzung, da er einzelne RDF-Tripel hasht und verknüpft.

Die Idee hinter diesem Algorithmus ist, dass zunächst jeder einzelne Tripel eines RDF-Datensatzes durch eine übliche Hashfunktion, wie dem SHA-1, gehasht wird. Danach werden die einzelnen Hashwerte durch eine Verknüpfungsfunktion miteinander verbunden. Sayers und Karp schlagen dafür die XOR-Funktion, die Multiplikationsfunktion modulo n und die Additionsfunktion modulo n vor. Jedoch empfehlen sie nicht, die XOR-Funktion zu benutzen, da für sie bereits effiziente Algorithmen zur Berechnung von Kollisionen existieren.

Sayers und Karp beschreiben weiterhin, dass sie bei der multiplikativen Verknüpfungsfunktion eine gute Sicherheit erwarten, aber dafür die Rechenoperationen relativ ressourcenintensiv sind. Die additive Verknüpfungsfunktion bietet dagegen einen guten Kompromiss zwischen Sicherheit und Geschwindigkeit. Ob sich die Wahl der Verknüpfungsfunktion in der Praxis tatsächlich messbar auf die Rechenzeit auswirkt, wird in meiner Arbeit mit Hilfe verschiedener Testdatensätze getestet.

3.2 Das Problem der anonymen Ressourcen

Jede anonyme Ressource (leerer Knoten) bekommt eine frei wählbare ID. Dadurch ist es möglich, dass ein Programm eine bestimmte ID vergibt und ein anderes für dieselbe Ressource eine andere. Anonyme Ressourcen eines RDF-

Graphen in die Prüfsumme zu integrieren ist eine Herausforderung.

Zwei Graphen, die dieselbe anonyme Ressource mit einer anderen ID beinhalten, bleiben in ihrer Struktur und semantischen Bedeutung gleich (siehe Abbildung 3). Die aus diesem Graph erstellte Prüfsumme sollte daher ebenfalls gleich bleiben. Die ID eines leeren Knotens muss jedoch in die Prüfsumme einfließen, um der unbemerkten Manipulierbarkeit des Graphen entgegenzuwirken.

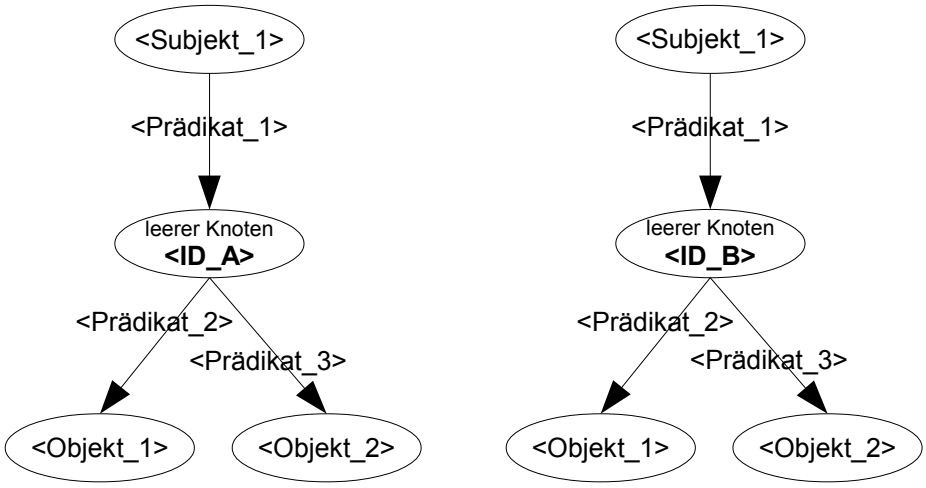

Abbildung 3: Zwei semantisch gleiche RDF-Graphen mit demselben leeren Knoten, aber mit verschiedenen IDs des leeren Knoten.

Um dieses Problem zu lösen, schlagen Sayers und Karp in [14] vor, zu allen leeren Knoten eines RDF-Graphen ein zusätzliches Tripel hinzuzufügen. Dieses Tripel beinhaltet dann den leeren Knoten als Subjekt und die gewählte feste ID als Literal im Objekt. Als feste ID nehmen Sayers und Karp die ID, die während der erstmaligen Erstellung der Prüfsumme des RDF-Graphen vorliegt.

Bei dieser Idee ist es vorgesehen, einzelne Tripel zu einem RDF-Graph hinzuzufügen. Es ist wichtig, dass diese zusätzlichen Tripel ebenfalls gehasht werden, da sonst eine unbemerkte Manipulation dieser Tripel möglich wäre. Das zusätzliche Hashing stellt kein Problem dar, da beim inkrementellen Hashing (siehe Kapitel 3.1) einzelne Tripel gehasht und dann miteinander verknüpft werden. Daher verwende ich auch diese Idee für den Algorithmus zur Prüfsummenerstellung.

3.3 Die Speicherung von Prüfsummen in RDF-Graphen

Zu jedem Tripel ein weiteres Tripel mit der dazugehörigen Prüfsumme in einen bestehenden RDF-Graphen einzufügen, ist die naive Methode zur Speicherung der Prüfsummen. Dieses Vorgehen führt jedoch zu einem hohen Speicherplatzverbrauch. Außerdem ist es nicht notwendig, Prüfsummen einzelner Tripel zu speichern, da es für einen Signaturersteller, der den Graphen signieren möchte, nicht sinnvoll ist, jedes Tripel einzeln zu signieren.

Daher sollten die Tripel zunächst zu logischen Einheiten, also Teildatensätzen von Tripeln, zusammengefasst werden. Dann wird zu jedem Teildatensatz die zugehörige Teilprüfsumme und ihre Berechnungsmethode gespeichert. Um die Gesamtprüfsumme eines RDF-Graph zu erhalten, müssen danach alle Teilprüfsummen miteinander verknüpft werden. Dies erfolgt ebenfalls durch die vorgestellte Methode in 3.1.

Für Begriffe, die für das Signieren von RDF-Daten benötigt werden, benutzen Carroll u.a. in [16] ein spezielles RDF-Vokabular, das *Semantic Web Publishing Vocabulary* (SWP Vocabulary)[5]. Das Vokabular sieht beispielsweise für eine Prüfsumme den URI

`http://www.w3.org/2004/03/trix/swp-2/digest`

vor. Zur besseren Übersicht werde ich in meiner Arbeit, ähnlich wie in [16], den ersten Teil des URI abkürzen. Dann erhält man den Begriff

`swp:digest`

Um die benutzte Methode zur Erstellung der Prüfsumme zu beschreiben, gibt es im SWP-Vokabular den Begriff

`swp:digestMethod`

Nachfolgend werden weitere Ideen zur Bildung der Teildatensätze erläutert. Die Idee unter 3.3.1 ist von mir entwickelt worden, die beiden darauf folgenden sind aus der Literatur entnommen.

[5] http://www.w3.org/2004/03/trix/swp-2/

3.3.1 Zum Subjekt zugehörige Prüfsummen

Eine Idee besteht darin, dass alle Tripel eines RDF-Datensatzes, die dasselbe Subjekt enthalten, in Teildatensätzen zusammengefasst werden. Diese Idee ist einfach umzusetzen und hat eine große Performance. Aus den erstellten Teildatensätzen können dann die passenden Prüfsummen errechnet und als zusätzliche Tripel abgespeichert werden. Die zusätzlichen Tripel enthalten jeweils das betreffende Subjekt, das Prädikat `swp:digest` bzw. `swp:digestMethod` und im Objekt die errechnete Prüfsumme bzw. die Methode der Prüfsummenberechnung.

Bei dieser Idee kann es jedoch auftreten, dass jedes Subjekt nur in einem Tripel vorkommt. Dadurch hat man im Vergleich zum naiven Ansatz keinen Vorteil. Das andere Extrem ist, wenn alle Tripel dasselbe Subjekt haben. Dadurch gibt es keine Unterteilung der RDF-Daten. Eine sinnvolle Zusammenfassung in Teildatensätzen kann also durch diese Idee nicht immer gewährleistet werden. Daher wird diese Idee in der vorliegenden Arbeit nicht weiter untersucht.

3.3.2 Minimum Self-contained Graphs (MSGs)

Tummarello u.a. schlagen in [12] die Unterteilung eines RDF-Graphen in Teilgraphen, den *Minimum Self-contained Graphs* (MSGs), vor. Ein MSG ist folgendermaßen definiert:

> „Given an RDF statement s, the Minimum Self-contained Graph (MSG) containing that statement, written MSG(s), is the set of RDF statements comprised of the following:
>
> 1. The statement in question;
> 2. Recursively, for all the blank nodes involved by statements included in the description so far, the MSG of all the statements *involving* such blank nodes;“[12, S. 1020]

Also enthält ein MSG entweder nur einen Tripel ohne leere Knoten (siehe Abbildung 4, MSG 1) oder mehrere Tripel, falls mindestens ein leerer Knoten in der anfangs ausgewählten Aussage vorkommt (siehe Abbildung 4, MSG 2 und MSG 3). Ein MSG umschließt praktisch seine leeren Knoten. Laut Tummarello u.a.[12] gehört ein RDF-Tripel immer zu genau einem MSG. Außerdem gibt es gemäß [12] immer nur genau eine Möglichkeit der Zerlegung eines RDF-Graphen in MSGs.

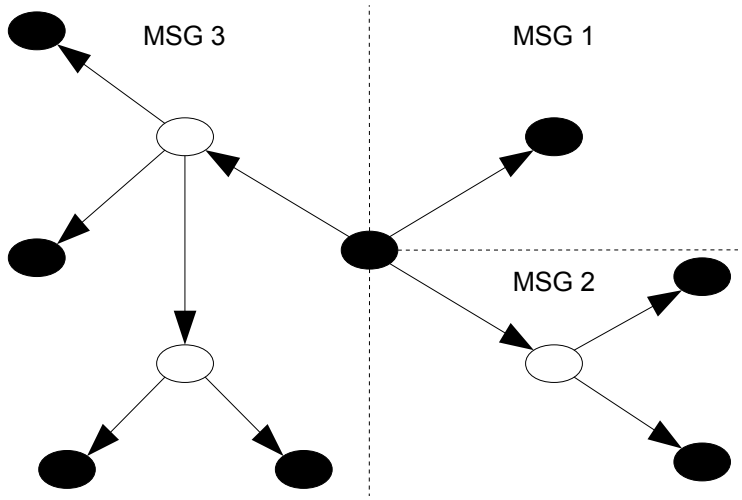

Abbildung 4: Aufteilung eines RDF-Graphen in seine MSGs. Die schwarzen Knoten sind mit einem URI oder einem Literal gefüllt, die weißen sind leere Knoten

Wie bei der Idee in Kapitel 3.3.1, wo zu jedem Subjekt weitere Tripel hinzugefügt werden, kann man auch zu jedem MSG zwei weitere Tripel mit der Prüfsumme und der Berechnungsmethode speichern. Falls ein MSG nur ein Tripel enthält, kann man Subjekt, Prädikat und Objekt analog wählen. Bei einem MSG mit mehreren Tripeln gibt es jedoch das Problem, dass die Tripel verschiedene Subjekte besitzen. Da ein MSG seine leeren Knoten komplett umschließt, kann man einen davon als Subjekt für die beiden zusätzlichen Tripel benutzen.

Wenn bei der Idee von Tummarello u.a. die Prüfsumme eines Teildatensatzes, also hier eines MSG, berechnet werden soll, muss der MSG zuerst neu gefunden werden. Wünschenswert ist allerdings, dass man das nicht jedes Mal tun muss. Außerdem kann es bei dieser Unterteilung der RDF-Daten, genauso wie bei der Idee aus 3.3.1, geschehen, dass für jedes einzelne Tripel die Prüfsumme und die Berechnungsmethode gespeichert werden müssen. Das ist dann der Fall, wenn der RDF-Graph keine leeren Knoten enthält. Auch das andere Extrem ist hier wiederum denkbar, wenn jedes Tripel mindestens einen leeren Knoten beinhaltet. Dann kann der RDF-Graph aus einem großen MSG bestehen. Daher kann auch mit der Idee von Tummarello u.a. eine sinnvolle Unterteilung eines RDF-Graphen nicht immer gewährleistet werden. Folglich kann diese Idee zur Speicherung von Prüfsummen in einem RDF-Graph in dieser Form nicht

benutzt werden. Ich werde diese Idee im Verlauf meiner Arbeit trotzdem verwenden, jedoch den Graphen anders unterteilen.

3.3.3 Named Graphs

Eine weitere Idee wurde von Carrol u.a. in [16] vorgestellt. In der Arbeit wird der RDF-Standard um so genannte *Named Graphs* erweitert. Ein Named Graph ist ein RDF-Graph, dem zusätzlich ein Name zugeordnet wird. Dieser Name muss ein URI sein.

Carrol u.a. benutzen in ihrer Arbeit eine spezielle Notation, um Named Graphs darzustellen. Diese Notation wird in ihrer Arbeit *TriG* genannt. Ein konzeptionelles Beispiel dafür ist in Abbildung 5 dargestellt. In dieser Abbildung sieht man, dass TriG die Tripelnotation um den Namen des Graphen und geschweifte Klammern erweitert. Außerdem kann man in der TriG-Syntax auch den Namen des Graphen als Subjekt benutzen.

```
<Graph_1> { <Subjekt_1> <Prädikat_1> <Objekt_1> .
            <Subjekt_2> <Prädikat_2> <Objekt_2> .
            <Graph_1> <Prädikat_3> <Objekt_3> }

<Graph_2> { <Subjekt_4> <Prädikat_4> <Objekt_4> .
            <Subjekt_5> <Prädikat_5> <Objekt_5> }
```

Abbildung 5: Konzept von Named Graphs notiert in TriG-Syntax

Wenn Named Graphs zur Speicherung von Prüfsummen benutzt werden, kann man in jeden Named Graph zwei weitere Tripel einfügen. Beide Tripel haben dann als Subjekt den Namen des Graphen, als Prädikat `swp:digest` bzw. `swp:digestMethod` und als Objekt die errechnete Prüfsumme bzw. ihre Berechnungsmethode.

Der Vorteil bei dieser Idee ist, dass man selbst entscheiden kann, wie viele Tripel zu einem Named Graph, also einem Teildatensatz zusammengefügt werden. Dadurch wird vermieden, dass ein Signaturersteller jedes Tripel einzeln oder nur einen großen RDF-Graphen signieren muss. Durch diesen Ansatz kann also eine sinnvolle Unterteilung eines RDF-Graphen in Teildatensätze gewährleistet werden. Jedoch muss dafür der RDF-Standard erweitert werden.

Das World Wide Web Consortium überarbeitet einen Teil des RDF-Standards. Dabei wird darauf geachtet, dass Mengen von RDF-Graphen, so genannte *RDF Datasets*, unterstützt werden. Der Entwurf ist in [17] zu finden. Daraus schließe

ich, dass auch das W3C die Ideen, die hinter Named Graphs stecken, befürwortet.

Ich halte die Idee von Named Graphs für die beste Alternative. Dadurch ist man in der Lage, die Größe der Teildatensätze selbst festzulegen und vermeidet damit einen unnötig hohen Speicherplatzverbrauch. Ebenso wird, durch eine geeignete Wahl der Teildatensatzgröße, die schnelle Berechnung einer Signatur ermöglicht. Es ist noch zu bestimmen, wie groß die Teildatensätze gewählt und welche Teildaten in ihnen enthalten sein müssen. Dies wird im nachfolgenden Abschnitt diskutiert, und ich werde dazu einen Vorschlag unterbreiten.

3.4 Die Unterteilung eines RDF-Datensatzes in Teildatensätze

Eine Person, die bestimmte RDF-Daten benötigt, sollte nicht den kompletten Datensatz herunterladen müssen, sondern nur einen kleinen Teil davon. Daher ist es ungünstig, willkürlich ausgewählte Tripel zu Teildatensätzen zusammenzufassen. Für eine sinnvolle Unterteilung von RDF-Daten in Teildatensätze muss der geeignete Inhalt und die passende Größe bestimmt werden.

3.4.1 Inhalt der Teildatensätze

Wenn man den Inhalt der Teildatensätze richtig wählt, wird eine geringe Anzahl der Teildatensätze ausreichen, um die gewünschten Daten bereitzustellen. Daher muss der Inhalt davon abhängen, was eine Person aus den RDF-Daten benötigt. Dies ist jedoch nicht vorhersehbar.

Ich gehe davon aus, dass sich eine Person, die gewisse RDF-Daten benötigt, für ein bestimmtes Subjekt interessiert. Also braucht diese Person alle Tripel, die dieses Subjekt enthalten. Einige von diesen Tripeln können eine anonyme Ressource als Objekt beinhalten. In diesem Fall benötigt die Person wahrscheinlich auch die Tripel, die diese anonyme Ressource als Subjekt enthalten. Dies muss dann rekursiv weitergeführt werden, bis man zu Tripeln ohne eine anonyme Ressource im Objekt kommt. Jedoch gibt es bei diesem Algorithmus keine eindeutige Aufteilung eines RDF-Graphen, da die anonyme Ressource im Objekt des zuerst betrachteten Tripels ebenfalls Objekt eines Tripels mit einem anderem Subjekt sein kann. Dadurch würden anonyme Ressourcen mehrfach in unterschiedlichen Teildatensätzen vorkommen. Das ist unerwünscht. Falls nämlich die Daten der anonymen Ressource geändert werden, muss man diese Änderung in jedem Teildatensatz durchführen. Das ist ineffizient und fehleranfällig.

Wie bereits in 3.3.2 beschrieben gibt es nur eine mögliche Zerlegung eines RDF-Graphen in MSGs. Tummarello u.a. definieren in [12] weiterhin eine *RDF Neighborhood*. Dies ist der RDF-Graph, der aus allen MSGs entsteht, die eine bestimmte Ressource beinhalten. Ich werde diesen RDF-Graphen im Folgenden eine *RDF-Nachbarschaft* nennen. Dieser Graph schließt die Informationen ein, die eine Person vermutlich benötigt, wenn sie nach einem bestimmten Subjekt sucht. Die RDF-Nachbarschaft beinhaltet jedoch mehr als die benötigten Informationen. Der entstandene Overhead muss aber leider in Kauf genommen werden, um eine Redundanz von Daten zu vermeiden.

Es ist optimal, wenn in einem Teildatensatz alle Tripel enthalten sind, die in der RDF-Nachbarschaft einer angefragten Ressource liegen. Das führt aber in der Regel zu großer Redundanz. Um eine RDF-Nachbarschaft ohne große Redundanz zu bilden, werden immer nur vollständige MSGs in einem Teildatensatz gespeichert. Darüber hinaus muss mindestens ein vollständiger MSG in einem Teildatensatz gespeichert werden. Bei Bedarf wird dann die RDF-Nachbarschaft zu einer bestimmten Ressource aus den Teildatensätzen gebildet.

Hierbei besteht noch immer das Problem, dass man nicht genau weiß, wie viele Tripel in einem MSG enthalten sind. Die Anzahl kann zwischen einem und allen Tripeln eines RDF-Datensatzes variieren. Daher kann eine allgemeingültige Begrenzung der Größe für einen Teildatensatz nicht so festgelegt werden, dass immer mindestens ein MSG hineinpasst. Um trotzdem eine Begrenzung zu erhalten, benutze ich für meine Arbeit die RDF-Datensätze von DBpedia als Basis. Diese Datensätze sind nicht für alle Einsatzgebiete von RDF repräsentativ, jedoch bieten sie einen ausreichenden und plausiblen Anhaltspunkt für meine Abschätzung.

3.4.2 Schätzung der Größe eines MSG

Durch eine grobe manuelle Sichtung der von DBpedia bereitgestellten RDF-Datensätze konnte ich feststellen, dass sie keine anonymen Ressourcen beinhalten. Dadurch lässt sich die Anzahl der Tripel, die ein MSG in der Praxis enthält, also nicht abschätzen.

Eine typische Verwendung für eine anonyme Ressource ist die Abbildung der Adresse einer Person. Dieses Beispiel wird auch in [2] verwendet. Die Adresse einer Person besteht in Deutschland üblicherweise aus Straße, Hausnummer, Postleitzahl und Stadt. Das sind dann insgesamt 5 Tripel, die dieselbe anonyme Ressource beinhalten: Ein Tripel mit der Person als Subjekt und der anonymen

Ressource als Objekt und jeweils ein Tripel mit der anonymen Ressource als Subjekt und einem Teil der Adresse. Natürlich sind viele andere Möglichkeiten der Verwendung anonymer Ressourcen denkbar, daher muss diese Abschätzung bei individueller Verwendung angepasst werden. Ich wähle in meiner Arbeit jedoch die Anzahl von 5 Tripeln pro MSG als festen Wert. Dieser ist meiner Einschätzung nach für die meisten Anwendungsfälle ausreichend. Meine Recherchen haben ergeben, dass eine Überschreitung von 5 Tripeln pro MSG so gut wie gar nicht vorkommt.

Für weitere Abschätzungen werden von DBpedia die RDF-Datensätze „Titles“, „Short Abstracts“, „Extended Abstracts“ und „Images“ der deutschsprachigen (de) Wikipedia im N-Triple-Format verwendet. Ich halte sie, durch ihre unterschiedlich großen Tripelanzahlen und Datensatzgrößen, für eine repräsentative Auswahl aus den Datensätzen von DBpedia. Eine Übersicht der verwendeten Datensätze ist in Tabelle 1 dargestellt.

Datensatz	Tripelanzahl	Datensatzgröße
Titles(de)	705 883	79,7 MiB
Short Abstracts(de)	650 082	253,8 MiB
Extended Abstracts(de)	650 082	406,5 MiB
Images(de)	2 468 204	499,2 MiB

Tabelle 1: Übersicht der verwendeten Datensätze für die Schätzung der Größe eines MSG.

Jedes Tripel besteht aus drei Ressourcen und jede Ressource beinhaltet eine gewisse Anzahl an Zeichen. Dabei können URIs, leere Knoten oder Literale theoretisch eine unendlich große Länge besitzen. Jedoch ergibt sich durch die Auswertung der vier betrachteten Datensätze (siehe Tabelle 2) eine maximale Anzahl der Zeichen eines Subjekts von 548, eines Prädikats von 44 und eines Objekts von 31 045. In dieser Tabelle fällt bei den Datensätzen „Short Abstracts(de)“ und „Extended Abstracts(de)“ die besonders große Zeichenanzahl bei den Objekten auf. Das liegt daran, dass diese Objekte eine Zusammenfassung eines Artikels aus Wikipedia enthalten. Bei den beiden Datensätzen „Titles(de)“ und „Images(de)“ ist diese Zeichenanzahl wesentlich geringer.

In den Datensätzen von DBpedia kommen keine anonymen Ressourcen vor, daher ist die Anzahl der Zeichen ihrer IDs in Tabelle 2 nicht berücksichtigt. Die Zeichenlänge der IDs von leeren Knoten hängt von ihrer Speicherrepräsentation ab. Das Framework *Apache Jena*[6], welches ich für die Implementierung des

[6] http://jena.apache.org/

	Maximale Zeichenanzahl			
Datensatz	Subjekt	Prädikat	Objekt	Summe
Titles(de)	236	42	148	426
Short Abstracts(de)	236	44	15 380	15 660
Extended Abstracts(de)	236	36	31 045	31 317
Images(de)	548	38	548	1 134
Spaltenmaximum	548	44	31 045	

Tabelle 2: Maximale Anzahl der Zeichen der Ressourcen ausgewählter Datensätze von DBpedia.

Prototyps benutze, verwendet 27 Zeichen für eine ID. Dieser Wert übersteigt keinen Wert, der in Tabelle 2 vorkommt. Daher müssen die Maximalwerte auch nicht nach oben korrigiert werden.

Daraus ergibt sich also für einen MSG eine maximale Größe von

$$5 \text{ Tripel} * 31\,317 \ \tfrac{Zeichen}{Tripel} = 156\,585 \text{ Zeichen}.$$

Ein Zeichen wird in Java durch Unicode mit 16 Bit pro Zeichen codiert. Daraus ergibt sich also die maximale Größe für einen MSG mit $156\,585 * 16 = 2\,505\,360$ Bit. Das entspricht ca. 305,83 *Kibibyte* (KiB).

Üblicherweise ist ein MSG jedoch nicht so groß. Daher möchte ich zusätzlich die durchschnittliche Anzahl der Zeichen in einem MSG abschätzen. In Tabelle 3 ist die durchschnittliche Anzahl der Zeichen einer Ressource in den ausgewählten Datensätzen aufgelistet.

	Durchschn. Zeichenanzahl		
Datensatz	Subjekt	Prädikat	Objekt
Titles(de)	45,15	42,00	19,25
Short Abstracts(de)	45,66	44,00	286,07
Long Abstracts(de)	45,66	36,00	522,77
Images(de)	77,43	36,70	85,22
Gewichtetes Mittel	63,10	38,50	167,57

Tabelle 3: Durchschnittliche Anzahl der Zeichen der Ressourcen ausgewählter Datensätze von DBpedia.

Damit beträgt die mittlere durchschnittliche Anzahl der Zeichen eines Subjekts 63,1, eines Prädikats 38,5 und eines Objekts 167,57 Zeichen. In den betrachteten Datensätzen kommen keine leeren Knoten vor. Würde man leere Knoten

berücksichtigen, würde sich der errechnete Durchschnittswert etwas verringern. Um auf der sicheren Seite zu sein, werden an dieser Stelle jedoch leere Knoten nicht berücksichtigt.

Aus diesen Daten ergibt sich die mittlere durchschnittliche Größe eines MSG von

$$5 \text{ Tripel} * (63{,}1 + 38{,}5 + 167{,}57) \ \tfrac{Zeichen}{Tripel} = 1\,346 \text{ Zeichen (gerundet).}$$

In der 16-Bit-Codierung in Java ergibt das einen Wert von $1\,346 * 16 = 21\,536$ Bit, also ca. 2,63 KiB. Es fällt auf, dass dieser Durchschnittswert im Verhältnis zum abgeschätzten Maximalwert von 305,83 KiB klein ist. Ich gehe davon aus, dass nur wenige Ausreißer existieren, die eine so große Anzahl an Zeichen besitzen.

Damit habe ich also eine maximale und eine durchschnittliche Größe abgeschätzt, die sich für einen MSG ergeben können. Diese Schätzung wird nachfolgend bei der Festlegung der Größe eines Teildatensatzes verwendet.

3.4.3 Größe eines Teildatensatzes

Ein Teildatensatz darf nicht zu klein sein. Ansonsten muss ein Signaturersteller sehr viele Signaturen ausstellen. Hinzu kommt, wenn der Teildatensatz zu klein gewählt wird, müssen beim Auslesen viele Prüfsummen miteinander verknüpft werden. Ein Teildatensatz darf aber auch nicht zu groß sein. Das ist nicht effizient, da man bei zu wenigen Teildatensätzen einen großen Overhead von nicht benötigten Daten hat. Also ist hier ein Kompromiss zu finden. Zur Festlegung der Größe der Teildatensätze kann man sich statische und dynamische Methoden vorstellen.

Bei einer *statischen Methode* ist die Größe der Teildatensätze immer gleich, jedoch variiert ihre Anzahl. Dadurch kann es zu einer sehr hohen Anzahl der Teildatensätze kommen, wenn der gesamte Datensatz entsprechend groß ist. Dies ist für einen Signaturersteller jedoch ungünstig.

Eine flexible Datensatzgröße wird bei einer *dynamischen Methode* relativ zur Größe des kompletten Datensatzes gewählt. Das hat den Vorteil einer begrenzten Anzahl von Teildatensätzen, aber den Nachteil, dass der einzelne Teildatensatz sehr groß werden kann und dann viel Overhead entsteht. Das ist ungünstig für eine Person, die nur einzelne Daten braucht.

Natürlich ist es auch denkbar, diese Methoden zu kombinieren. Man kann z. B. eine obere Grenze der Größe statisch festlegen. Bei kleineren Datensätzen hingegen wird die zulässige Größe des Teildatensatzes relativ zur Größe des gesamten Datensatzes gewählt. Bei dieser Kombination der Methoden wird der Overhead bei großen Datensätzen zwar begrenzt, jedoch hätte ein Signaturersteller viele Datensätze zu signieren. Es besteht auch die Möglichkeit, dass man eine untere Grenze der Größe eines Teildatensatzes festlegt und die Größe bei großen Datensätzen relativ auswählt. Dies führt zu großem Overhead, begünstigt jedoch den Signaturersteller.

Ich messe dem Sachverhalt, dass beim Herunterladen eines Teildatensatzes wenig Overhead für den Datennutzer enthalten ist, ein größeres Gewicht zu, als dem Sachverhalt, dass der Signaturersteller begünstigt wird. Denn eine Signatur wird in der Regel nur einmal vorgenommen, während die Daten oft gelesen werden. Die RDF-Daten können sich allerdings auch ändern. Dann müssen Prüfsummen und Signaturen der betroffenen Teildatensätze neu berechnet werden. Selbst in diesem Fall ist es von Vorteil, wenn ein Teildatensatz nicht allzu groß ist, da auch die nicht von der Änderung betroffenen Teile dieses Teildatensatzes bei der Neuberechnung berücksichtigt werden müssen. Daher schlage ich keine dynamische Methode vor, die von der Größe des Datensatzes abhängt.

Ich erachte es für sinnvoll, dass ein MSG m komplett in einem Teildatensatz enthalten ist. Daher schlage ich vor, dass die Größe von m die untere Begrenzung für die Größe eines Teildatensatzes bildet. Da es aber auch geschehen kann, dass m klein ist, schlage ich zusätzlich eine statische Mindestgröße vor, so dass mehrere MSGs in einem Teildatensatz enthalten sein können. Dadurch gibt es etwas Overhead. Diesen halte ich aber für vertretbar, weil damit auf einen Signaturersteller Rücksicht genommen wird.

Damit ein Teildatensatz nicht zu groß wird, muss auch eine statische Maximalgröße bestimmt werden. Falls ein einzelner MSG größer ist als die festgelegte statische Höchstgrenze, wird dieser aber trotzdem komplett in den Teildatensatz übernommen, da es wichtiger ist, dass der komplette MSG in einem Teildatensatz enthalten ist, als dass die Maximalgröße eingehalten wird.

Der in 3.4.2 abgeschätzte Durchschnittswert beträgt 2,63 KiB. Ich wähle eine Mindestgröße von 5 KiB. Damit wird besser sichergestellt, dass ein Signaturersteller weniger Teildatensätze signieren muss. Um wenigstens zwei MSGs der Mindestgröße speichern zu können, lege ich eine Maximalgröße von 10 KiB fest. Das entspricht in der Unicode-Codierung von Java 2 560 bis 5 120 Zeichen. Damit wird dem Kompromiss zwischen wenigen Signaturen und kleinem Overhead Rechnung getragen.

3.5 Beschreibung und Laufzeitanalyse des gewählten Algorithmus

In Abbildung 6 wird die Reihenfolge zur Erstellung und Speicherung der Prüfsummen dargestellt. Bevor dieser Algorithmus ausgeführt werden kann, müssen die RDF-Daten in den Speicher geladen werden. Für diese Arbeit wird allerdings vorausgesetzt, dass diese Daten bereits in den Speicher geladen worden sind. Denn das Ziel dieser Arbeit ist, Prüfsummen aus bereits im Speicher liegenden RDF-Daten zu berechnen. Der Algorithmus ist in drei Teile aufgeteilt, die im Folgenden näher beschrieben werden.

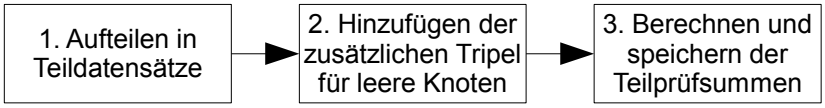

Abbildung 6: Übersicht der Schritte zur Erstellung und Speicherung der Prüfsummen

3.5.1 Aufteilen in Teildatensätze

Im ersten Teil des Algorithmus wird die Aufteilung des gesamten RDF-Datensatzes in Teildatensätze vorgenommen. Abbildung 7 zeigt den Ablaufplan dazu.

Der Ausgangspunkt ist ein im Speicher liegender RDF-Datensatz d. Jetzt wird ein leerer Datensatz d' für Named Graphs angelegt. Jeder erstellte Teildatensatz wird in Datensatz d' als Named Graph eingefügt. Jetzt wird d sukzessive MSG für MSG abgearbeitet. Dabei werden alle Tripel eines gefundenen MSG temporär gespeichert und aus dem Datensatz d entfernt. Wenn ein einzelner MSG größer oder gleich der in 3.4.3 festgelegte Mindestgröße ist, wird er in einen eigenen Named Graph eingefügt. MSGs, die kleiner als die Mindestgröße sind, werden so lange zu einem temporären Teildatensatz hinzugefügt, bis dieser größer oder gleich der Mindestgröße ist. Da die hinzugefügten MSGs immer kleiner als die Mindestgröße sind, bleibt die Größe der Teildatensätze immer unter der in 3.4.3 festgelegten Maximalgröße. Wenn kein MSG mehr in d enthalten ist, werden noch die im temporären Teildatensatz gespeicherten MSGs in d' eingefügt. Dann enthält der Datensatz d' alle erstellten Teildatensätze als Named Graphs und wird schließlich ausgegeben.

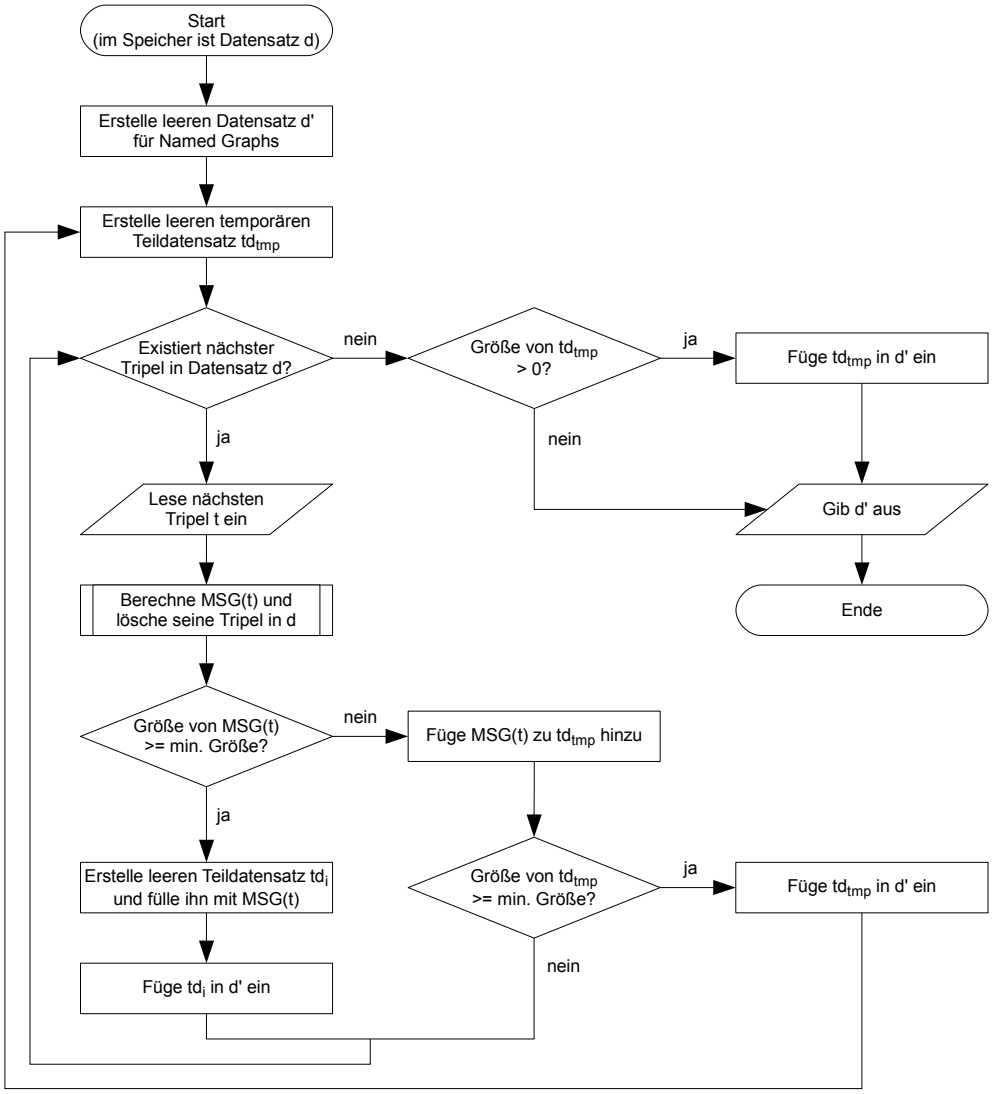

Abbildung 7: Ablaufplan zur Erstellung der Teildatensätze

Das Unterprogramm „Berechne MSG(t) und lösche seine Tripel in d“ funktioniert ähnlich wie eine Breitensuche. Zunächst wird eine leere Menge m_{MSG} und eine leere Warteschlange erstellt. Dann wird das Tripel t_0, von dem der MSG errechnet werden soll, an die Warteschlange angefügt. Jetzt wird t_0 aus der Warteschlange als zu untersuchendes Tripel x entnommen. Danach wird es zu m_{MSG} hinzugefügt und aus d gelöscht.

Nun wird Tripel x auf leere Knoten untersucht. Wenn x keinen leeren Knoten enthält, ist es abgearbeitet. Falls leere Knoten enthalten sind, wird für jeden von ihnen ein anderes Tripel in d gesucht, das ebenfalls diesen leeren Knoten enthält. Wenn dafür ein passendes Tripel gefunden wird, wird es an die Warteschlange angefügt, zu m_{MSG} hinzugefügt und aus d gelöscht. Wenn kein weiteres Tripel gefunden wird, dass den zu suchenden leeren Knoten beinhaltet, wird nichts weiter getan. Danach ist x abgearbeitet und das nächste Tripel wird aus der Warteschlange als neues Tripel x entnommen. Diese Schritte werden nun so lange wiederholt, bis die Warteschlange leer ist. Am Ende enthält m_{MSG} alle Tripel, die zum MSG von t gehören und kann als Ergebnis ausgegeben werden.

Die einfachen Schritte im ersten Teil des Algorithmus haben höchstens eine lineare asymptotische Laufzeit. Lediglich das Unterprogramm zur Berechnung des MSG benötigt eine größere Laufzeit. Wenn bei dieser Berechnung ein Tripel aus der Warteschlange entnommen wird, müssen im schlechtesten Fall alle Tripel, die noch in d enthalten sind, betrachtet werden, um ein passendes Tripel zu finden. Das ergibt dann also für n Tripel

$$(\text{n - 1}) + (\text{n - 2}) + \ldots + 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)*n}{2} = \frac{n^2-n}{2}$$

Tripel, die auf einen passenden leeren Knoten untersucht werden müssen. Dies bedeutet also, dass die asymptotische Laufzeit des ersten Teils für den schlechtesten Fall in $O(n^2)$ liegt. Der beste Fall tritt auf, wenn kein leerer Knoten existiert und dadurch keine weiteren Tripel gesucht werden müssen. Dann liegt die asymptotische Laufzeit des ersten Teils des Algorithmus in $O(n)$.

3.5.2 Hinzufügen der zusätzlichen Tripel für leere Knoten

Nachdem die Teildatensätze gebildet worden sind, folgt im zweiten Teil das Hinzufügen von zusätzlichen Tripeln für die leeren Knoten. Hierbei geht der Algorithmus alle Tripel der Teildatensätze durch und fügt, wie in 3.2 beschrieben, für jeden leeren Knoten ein zusätzliches Tripel mit der ID des leeren Knotens

ein. Der Algorithmus liest also insgesamt n Tripel und überprüft, ob sie leere Knoten enthalten. Für jeden gefundenen leeren Knoten muss geprüft werden, ob das passende zusätzliche Tripel schon im Teildatensatz enthalten ist, oder nicht. Wenn das Tripel noch nicht vorhanden ist, dann wird es eingefügt.

Bei n Tripeln kann es maximal 2n leere Knoten geben, da nur Subjekt und Objekt eines Tripels leere Knoten sein dürfen. Für jeden der maximal 2n leeren Knoten muss geprüft werden, ob das zugehörige Tripel bereits enthalten ist. Diese Prüfung, so schreiben Sayers und Karp in [15], kann für einen leeren Knoten mit frei wählbarer ID in annähernd konstanter Zeit durchgeführt werden. Das geht unter Verwendung einer Hashtabelle für die IDs der gefundenen leeren Knoten. Denn die Suche nach einem Schlüssel in einer Hashtabelle benötigt annähernd konstante Zeit.

Wenn die ID des leeren Knotens bereits in der Hashtabelle enthalten ist, wird der nächste geprüft. Wenn die ID des leeren Knotens noch nicht in der Hashtabelle steht, wird sie dort eingefügt und die RDF-Daten werden um den zusätzlichen Tripel erweitert. Diese Operationen benötigen annähernd konstante bzw. konstante Zeit. Also liegt die asymptotische Laufzeit des zweiten Teils für n Tripel im schlechtesten Fall in $O(n)$. Auch im besten Fall kann die asymptotische Laufzeit nicht besser sein, da immer alle n Tripel eingelesen werden müssen.

3.5.3 Berechnen und speichern der Teilprüfsummen

Im dritten Teil des Algorithmus werden die Prüfsummen der Teildatensätze berechnet und gespeichert. Außerdem wird noch ein Tripel eingefügt, der die Methode der Prüfsummenberechnung enthält.

Sayers und Karp beschreiben in [14], dass für die Berechnung der Prüfsumme eines Tripels konstante Zeit benötigt wird, wenn man davon ausgeht, dass es eine Begrenzung der Anzahl der im Tripel enthaltenen Zeichen gibt. Durch das Hinzufügen der Tripel für die leeren Knoten gibt es nun mehr als n Tripel, die gehasht werden müssen. Maximal können dies 3n Tripel sein. Die asymptotische Laufzeit für die Berechnung für 3n Tripel liegt in $O(n)$.

Die Berechnung der Prüfsummen der Teildatensätze ist auch in linearer Zeit möglich. Für die Verknüpfung zweier Prüfsummen mit der additiven bzw. multiplikativen Verknüpfungsfunktion wird konstante Zeit benötigt, da die Prüfsummenlänge begrenzt ist. Daraus folgt für maximal 3n Tripel eine lineare asymptotische Laufzeit.

Auch das Einfügen der Tripel mit der Teilprüfsumme und der verwendeten Methode benötigt eine konstante asymptotische Laufzeit. Wenn jedes Tripel ein MSG ist und dieses wenigstens die Mindestgröße besitzt, muss für n Teildatensätze jeweils die Teilprüfsumme eingefügt werden. Aus dem vorhergesagten folgt dann, dass für die maximal n Teildatensätze ebenfalls eine lineare asymptotische Laufzeit erforderlich ist.

Also liegt die asymptotische Laufzeit des letzten Teils des Algorithmus im schlechtesten Fall in $O(n)$. Da hier mindestens alle n Tripel einmal gelesen werden müssen, kann die asymptotische Laufzeit auch im besten Fall nicht besser sein.

4 Implementierung

In diesem Kapitel wird zunächst die Konzeption des Prototyps beschrieben und dann auf die implementierte Software eingegangen. Danach findet die Auswertung der Testdatensätze statt.

4.1 Konzeption

4.1.1 Generelle Festlegungen

Die Implementierung des Prototyps erfolgt in der Programmiersprache Java, da dies Vorgabe für meine Arbeit ist. Es sind selbstverständlich auch andere Programmiersprachen dafür möglich. Ich benutze für die Implementierung die derzeit aktuelle Java-Version (Java SE 7 Update 10) in der 32-Bit-Fassung. Meine Wahl fällt auf die 32-Bit-Version und nicht auf die 64-Bit-Version, da meiner Meinung nach die 32-Bit-Version derzeit auf den meisten Systemen benutzt wird.

Alle benötigten Hashfunktionen bis auf den SHA-3 sind in dieser Java-Version bereits enthalten. Ich benutze für den SHA-3 die Java-Implementierung aus der Bibliothek *sphlib 3.0*[7] von *Projet RNRT SAPHIR*[8].

Zur Verarbeitung der RDF-Daten benutze ich das Framework *Apache Jena* in der derzeit aktuellen Version 2.7.4[9].

4.1.2 Übersicht der Programmmodule

In Abbildung 8 ist ein UML-Paketdiagramm dargestellt, welches die folgenden Module des Prototyps beinhaltet:

Das Paket *Graphical User Interface* (GUI) beinhaltet eine einfache Oberfläche für den Benutzer. Es dient zur Auswahl der Ein- und Ausgabedatei, sowie zur Einstellung der Art, wie die Prüfsummen berechnet werden sollen. Die Einstellungen beinhalten, welche Hashfunktion und welche Verknüpfungsfunktion benutzt werden sollen.

Das GUI greift auf das Paket *Logik* zu, das den Ablauf des gesamten Programms steuert. Hier werden alle Einstellungen aus dem GUI ausgelesen und

[7] http://www.saphir2.com/sphlib/
[8] http://www.crypto-hash.fr/
[9] Download unter http://jena.apache.org/download/index.html

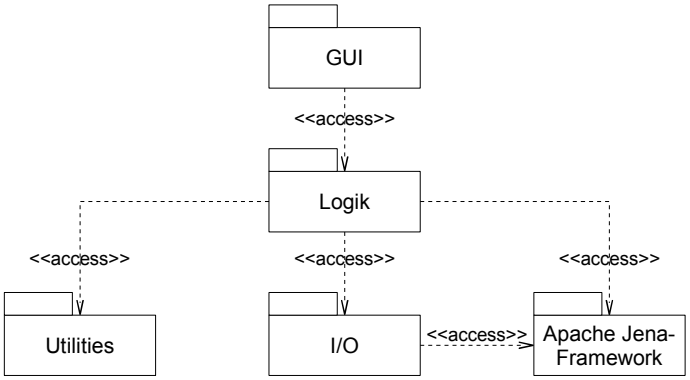

Abbildung 8: UML-Paketdiagramm der Programmmodule des Prototyps

ausgewertet. Nach dem Einlesen einer RDF-Datei werden alle Schritte des entwickelten Algorithmus, wie in Kapitel 3.5 beschrieben, mit den vorgenommenen Einstellungen durchgeführt und die veränderten RDF-Daten in die angegebene Ausgabedatei geschrieben.

Das Paket *Utilities* enthält diverse Komponenten des Programms. Dies schließt die additive und multiplikative Verknüpfungsfunktion sowie den Algorithmus zur Berechnung eines MSG ein. Außerdem ist hier die Implementierung des SHA-3 enthalten.

Des Weiteren gibt es das Paket *Input/Output* (I/O), welches für das Einlesen der RDF-Datei und die Ausgabe des Ergebnisses zuständig ist. Um die RDF-Datei korrekt einzulesen und in den Arbeitsspeicher zu schreiben, muss das I/O-Paket auf das *Apache Jena*-Framework zugreifen. Das Framework wird auch zur Erstellung der Ausgabedatei benötigt.

4.2 Beschreibung der Software

Abbildung 9 zeigt einen Screenshot des GUI des programmierten Prototyps.

Auf der linken Seite unter *Eingabe* kann die Eingabedatei und das Dateiformat ausgewählt werden. Der Prototyp unterstützt 4 verschiedene Dateiformate. Unter *Ausgabe* kann die Ausgabedatei angegeben werden. Es wird nur das TriG-Format, wie von Carrol u.a. in [16] vorgestellt, unterstützt. Da die Ausgabedatei lediglich zu Testzwecken dient, werden keine weiteren Formate benötigt.

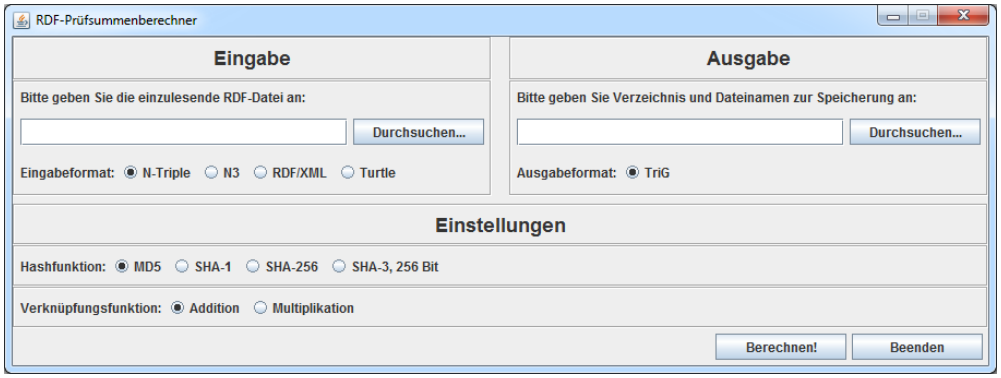

Abbildung 9: Screenshot des GUI des programmierten Prototyps

In den *Einstellungen* können derzeit 4 verschiedene Hashfunktionen und die additive bzw. multiplikative Verknüpfungsfunktion gewählt werden.

4.3 Durchführung und Auswertung der Tests

Für die Tests beschränke ich mich auf den SHA-256 und den SHA-3 mit 256 Bit Hashwertlänge. Ich wähle hier die Länge von 256 Bit, da dies für die relativ geringe Anzahl der Zeichen in einem Tripel ausreicht. In [18] wird beschrieben, dass die Hashfunktionen SHA-224 und SHA-256 für eine Eingabe von weniger als 2^{64} Bit Länge ausreichen, um einen sicheren Hash zu erstellen. Für größere Eingaben sind die Hashfunktionen SHA-384 und SHA-512 zu benutzen. Diese beiden Hashfunktionen reichen für Eingaben von weniger als 2^{128} Bit Länge aus. Ich wähle nicht den SHA-224, da ich auf der sicheren Seite sein möchte und einen Sicherheitsbonus veranschlage.

Die beiden anderen vorgestellten Hashfunktionen MD5 und SHA-1 wähle ich für die Tests ebenfalls nicht, da sie, wie in 2.2 beschrieben, nicht mehr sicher sind. Ich habe die Tests mit diesen Hashfunktionen trotzdem durchgeführt. Die Messergebnisse sind für den interessierten Leser in den Tabellen 6 und 7 im Anhang enthalten.

Das Testsystem ist ein PC mit einem Intel Core i5-2500K Prozessor, 8 Gibibyte Arbeitsspeicher und dem Betriebssystem Windows 7 in der 64-Bit-Version. Außerdem ist ein 32 Bit Java Runtime Environment in der Version 7 Update 10 installiert.

4.3.1 Auswahl von Testmethode und Testdaten

Da Java mehrere Threads benutzt, ist es sinnvoll, für das Testen der Laufzeit nur den Thread zu betrachten, in dem der Algorithmus ausgeführt wird. Daher benutze ich für die Messung der Laufzeit die Methode `getCurrentThreadCpuTime` aus dem Interface `java.lang.management.ThreadMXBean`. Die Laufzeit wird von dem Augenblick an gemessen, ab dem der eigentliche Algorithmus anfängt, bis zu seinem Endzeitpunkt. Das heißt, dass das Einlesen in den Arbeitsspeicher und die Dateiausgabe nicht enthalten sind.

Zur Ermittlung des Speicherplatzbedarfs benutze ich die Methoden `getHeapMemoryUsage` und `getNonHeapMemoryUsage` aus dem Interface `java.lang.management.MemoryMXBean`. Der Speicherplatzbedarf wird zum Zeitpunkt direkt nach der Ausführung des Algorithmus gemessen. Hier erwarte ich die größte Speicherauslastung, da erst zu diesem Zeitpunkt alle zusätzlichen Tripel erstellt und eingefügt wurden.

Bei der Durchführung von einigen Tests vorab ist mir aufgefallen, dass der Platzbedarf im Arbeitsspeicher schnell sehr groß wird. Dieser Speicherplatzbedarf gerät schnell an die Grenzen des bereitgestellten Speicherplatzes für die virtuelle Maschine von Java. Auf dem Testrechner ist die Höchstgrenze für den Speicherplatz der Java VM ca. 1 600 MiB. Daher kann ich die in 3.4.2 vorgestellten Datensätze nicht komplett verwenden, sondern muss kleinere Teile von ihnen benutzen. Also wurden die Dateien gekürzt und jeweils die ersten 1 000, 10 000, 100 000, 200 000 und 300 000 Tripel in separate Dateien geschrieben. Diese Normierung hat den Vorteil, dass es keine unterschiedlichen Tripelanzahlen pro Datensatz gibt und man dadurch den Vergleich der Laufzeiten und des benötigten Speichers direkt vornehmen kann.

Des Weiteren mussten wenige Tripel aus den Datensätzen „Short Abstracts(de)“ und „Extended Abstracts(de)“ korrigiert werden, damit sie das Apache Jena Framework einwandfrei lesen konnte. Sie entsprachen an wenigen Stellen nicht der Standardcodierung.

Bei meinen Tests vorab sind mir außerdem große Schwankungen der Laufzeit und des Speicherplatzbedarfs aufgefallen. Wenn das Programm zum ersten Mal aufgerufen wird, ist bei der Ausführung eines Tests grundsätzlich mit einer längeren Laufzeit zu rechnen, als wenn bereits ein Test durchgeführt wurde. Da die Laufzeit besonders bei wenigen Tripeln klein ist, ist auch die zusätzlich benötigte Laufzeit zur Initialisierung im Verhältnis größer als bei vielen Tripeln. Daher messe ich die Laufzeit bei 1 000 Tripeln bei der dritten Ausführung, bei 10 000 Tripeln bei der zweiten Ausführung und bei den RDF-Datensätzen mit

mehr Tripeln bei der ersten Ausführung.

Im Gegensatz zur Laufzeit, die am Anfang größer ist und dann etwas abnimmt, steigt der Speicherplatzbedarf mit jeder Ausführung leicht an. Daher nehme ich hier den Speicherplatzbedarf bei der ersten Durchführung des Tests. Hier ist es auch wichtig, dass das Programm vor jeder Änderung der Einstellungen neu gestartet wird, damit der Speicherplatzbedarf zurückgesetzt wird.

Durch die beschriebenen Verbesserungen aufgrund der Auswertung meiner Tests vorab sind die Schwankungen der Laufzeitmessungen kleiner geworden. Die Schwankungen bei der Messung des Speicherplatzbedarfs sind hingegen immer noch groß. Dies habe ich wiederum durch weitere Tests festgestellt. Wegen der noch verbleibenden Ungenauigkeiten können nur grobe Aussagen über Tendenzen gemacht werden.

4.3.2 Auswertung der Messung der Laufzeit

In Tabelle 4 sind die Ergebnisse der Messung der Laufzeiten unter Verwendung von SHA-256 und SHA-3 in der 256-Bit-Version abgebildet.

Daten-		Laufzeit bei einer Tripelanzahl von				
satz	Einstellungen	1 000	10 000	100 000	200 000	300 000
Titles (de)	SHA-256, Add	0,015 s	0,514 s	28,688 s	110,932 s	166,187 s
	SHA-256, Mul	0,062 s	0,858 s	32,541 s	118,997 s	178,558 s
	SHA-3, Add	0,015 s	0,483 s	28,844 s	111,540 s	168,512 s
	SHA-3, Mul	0,046 s	0,889 s	33,540 s	119,793 s	181,725 s
Short Abstr. (de)	SHA-256, Add	0,031 s	0,577 s	30,310 s	113,989 s	171,179 s
	SHA-256, Mul	0,062 s	0,967 s	33,899 s	121,103 s	181,975 s
	SHA-3, Add	0,062 s	0,717 s	31,621 s	117,156 s	174,736 s
	SHA-3, Mul	0,062 s	1,123 s	35,115 s	124,098 s	186,530 s
Ext. Abstr. (de)	SHA-256, Add	0,046 s	0,733 s	31,527 s	116,236 s	178,137 s
	SHA-256, Mul	0,062 s	1,014 s	34,788 s	123,381 s	187,294 s
	SHA-3, Add	0,078 s	0,951 s	33,477 s	121,384 s	184,642 s
	SHA-3, Mul	0,093 s	1,248 s	37,128 s	127,686 s	193,347 s
Images (de)	SHA-256, Add	0,015 s	0,312 s	14,835 s	56,363 s	161,663 s
	SHA-256, Mul	0,046 s	0,670 s	18,564 s	62,946 s	173,145 s
	SHA-3, Add	0,015 s	0,405 s	15,397 s	58,531 s	163,598 s
	SHA-3, Mul	0,062 s	0,748 s	19,437 s	65,177 s	174,440 s

Tabelle 4: Ergebnisse der Messung der Laufzeiten unter Verwendung von SHA-256 und SHA-3 (256 Bit)

In der Tabelle fällt auf, dass die Werte in der ersten Spalte bei 0,015 Sekunden anfangen und auch teilweise gleich sind, während dies bei den anderen Spalten nicht der Fall ist. Das liegt an der Auflösung des benutzten Timers, die 15,6001 ms beträgt. Das bedeutet, dass die Ergebnisse nur ein Vielfaches dieses Wertes annehmen können. Bei der einen Messung kann es geschehen, dass der Timer gerade eine Einheit der Auflösung weiter gesprungen ist, und bei der anderen Messung nicht. Das führt bei kleinen Werten zu einer großen relativen Ungenauigkeit.

Des Weiteren fällt auf, dass die Laufzeit bei allen Messungen mit der multiplikativen Verknüpfungsfunktion länger ist, als bei der additiven Verknüpfungsfunktion. Das ist nicht verwunderlich, da die implementierte Byte-Multiplikation auf der schriftlichen Methode zur Multiplikation zweier Zahlen basiert und somit mehrere Byte-Additionen hintereinander ausführt.

Unter Verwendung des SHA-3 in der 256-Bit-Version ist die Laufzeit in der Regel etwas länger als beim SHA-256. Ausnahmen dieser Regel gibt es in der ersten Spalte bei einer Tripelanzahl von 1 000. Hier sind manche Werte gleich. Das ist durch die Auflösung des Timers zu erklären. Die längere Laufzeit beim SHA-3 liegt wahrscheinlich am Algorithmus selbst.

Ferner steigt die Laufzeit, wie zu erwarten war, mit der Anzahl der Tripel an. In Abbildung 10 sind die Werte von Zeile „Short Abstr.(de)“ aus Tabelle 4 als Kurven dargestellt. Die Kurven der Zeilen „Titles(de)“ und „Ext. Abstr.(de)“ haben einen ähnlichen Verlauf, daher dient dieses Diagramm exemplarisch für diese drei Zeilen. Wie das Diagramm veranschaulicht, steigt die Laufzeit zunächst mehr als linear an. Dann flachen die Kurven zum Ende hin ab.

Die Verläufe der Kurven sind unerwartet, da sie keinen einfachen funktionalen Zusammenhang zur Tripelanzahl erkennen lassen. Wenn man jedoch zusätzlich die durchschnittlich benötigte Laufzeit pro Tripel (siehe Abbildung 11) betrachtet, kann man ein Muster erkennen. Zunächst steigt hier die benötigte Laufzeit pro Tripel an und bleibt dann konstant auf einem Niveau.

Daraus folgere ich, dass sich die Laufzeit pro Tripel bis zu einem bestimmten Schwellenwert steigert und ab diesem Schwellenwert auf ein gleichbleibendes Niveau einpendelt. Wenn das auch bei größeren Tripelanzahlen anhält, entspricht das, wie erwartet, einem linearen asymptotischen Laufzeitverhalten. Aber es ist auch möglich, dass dieser Schwellenwert nur eine Stufe ist und dass bei steigenden Tripelanzahlen immer höhere Stufen erreicht werden.

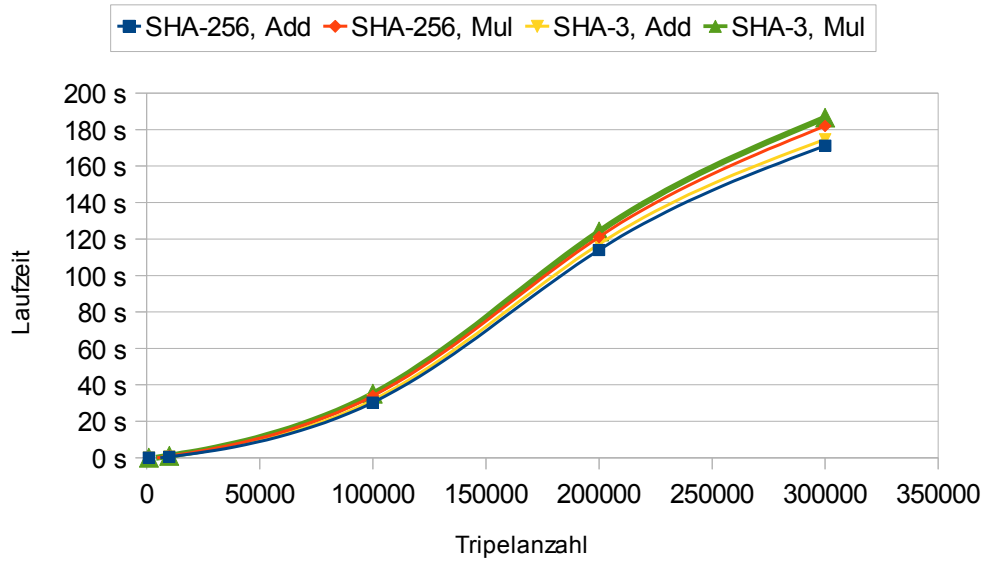

Abbildung 10: Gesamtlaufzeit des Algorithmus unter Verwendung von SHA-256 und SHA-3, verwendeter Datensatz: „Short Abstracts(de)“

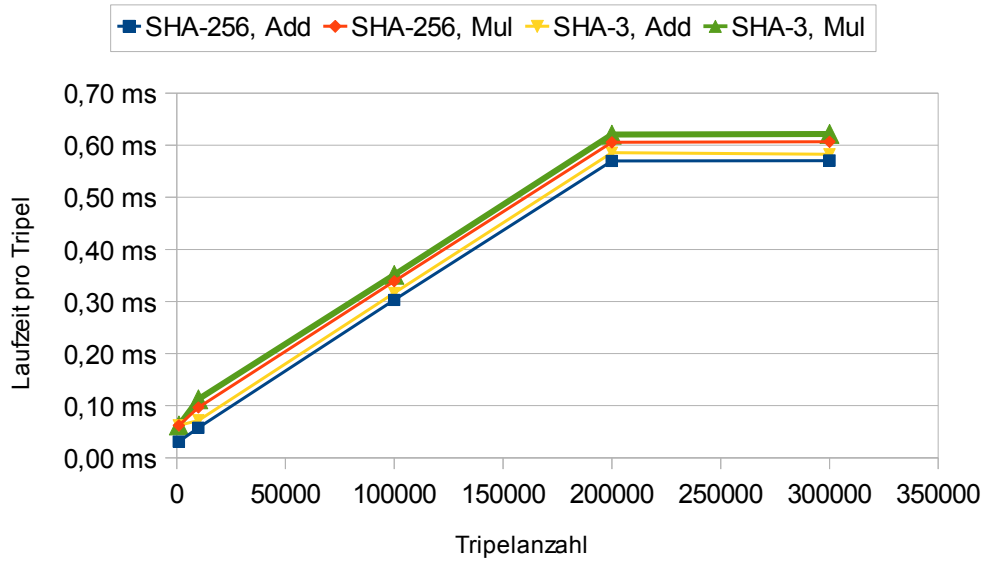

Abbildung 11: Durchschnittlich benötigte Laufzeit pro Tripel unter Verwendung von SHA-256 und SHA-3, verwendeter Datensatz: „Short Abstracts(de)“

Die Kurven der Werte von Zeile „Images(de)“ sehen anders aus als die Kurven der drei anderen Datensätze und sind in Abbildung 12 dargestellt. Hier ist die Steigung bis zum Ende der Messreihe überproportional. Das kann daran liegen, dass das Programm den im letzten Absatz beschriebenen Schwellenwert für diesen Datensatz noch nicht erreicht hat.

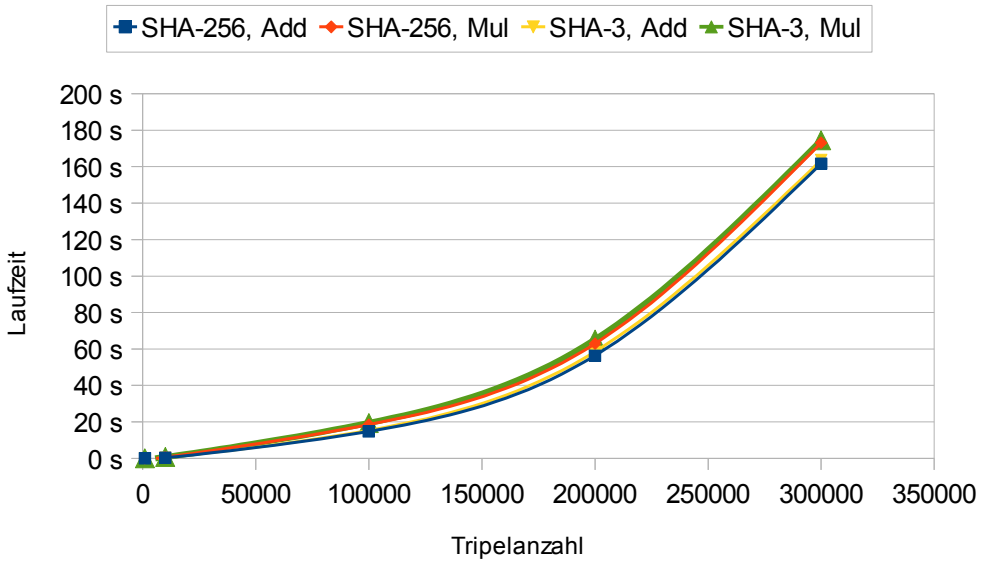

Abbildung 12: Gesamtlaufzeit des Algorithmus unter Verwendung von SHA-256 und SHA-3, verwendeter Datensatz: „Images(de)“

Die Auswertung der Laufzeitmessungen lässt keinen eindeutigen linearen Zusammenhang erkennen. Außerdem ist der aufgrund der Messreihen erkennbare funktionale Zusammenhang der vier Datensätze nicht gleich. Meiner Meinung nach sind hier zu wenige Messdaten vorhanden, um eindeutige Aussagen darüber zu treffen, wie sich die Laufzeit im Allgemeinen verhält. Dennoch halte ich den Prototypen bis zu einer gewissen Tripelanzahl für praxistauglich. Bei einer erheblich größeren Anzahl der Tripel, als in den Messreihen vorhanden, kann das Programm durchaus mehrere Tage benötigen, um die erforderlichen Teildatensätze und Prüfsummen zu berechnen. Das ist in der Praxis nicht akzeptabel.

4.3.3 Auswertung der Messung des Speicherplatzbedarfs

Die Ergebnisse der Messung des Speicherplatzbedarfs des Prototyps unter Verwendung von SHA-256 und SHA-3 in der 256-Bit-Version sind in Tabelle 5 abgebildet.

Datensatz	Einstellungen	Speicherplatzbedarf bei einer Tripelanzahl von 1 000	10 000	100 000	200 000	300 000
Titles (de)	SHA-256, Add	18 MiB	28 MiB	140 MiB	227 MiB	395 MiB
	SHA-256, Mul	18 MiB	27 MiB	118 MiB	254 MiB	369 MiB
	SHA-3, Add	18 MiB	29 MiB	139 MiB	211 MiB	396 MiB
	SHA-3, Mul	19 MiB	28 MiB	117 MiB	256 MiB	369 MiB
Short Abstr. (de)	SHA-256, Add	23 MiB	56 MiB	321 MiB	782 MiB	1 303 MiB
	SHA-256, Mul	23 MiB	61 MiB	375 MiB	637 MiB	1 171 MiB
	SHA-3, Add	17 MiB	59 MiB	320 MiB	770 MiB	1 304 MiB
	SHA-3, Mul	21 MiB	61 MiB	371 MiB	624 MiB	1 179 MiB
Ext. Abstr. (de)	SHA-256, Add	20 MiB	77 MiB	707 MiB	1 294 MiB	1 528 MiB
	SHA-256, Mul	23 MiB	82 MiB	550 MiB	1 136 MiB	1 417 MiB
	SHA-3, Add	20 MiB	77 MiB	707 MiB	1 302 MiB	1 533 MiB
	SHA-3, Mul	23 MiB	82 MiB	550 MiB	1 154 MiB	1 395 MiB
Images (de)	SHA-256, Add	18 MiB	24 MiB	142 MiB	284 MiB	368 MiB
	SHA-256, Mul	18 MiB	25 MiB	126 MiB	242 MiB	347 MiB
	SHA-3, Add	18 MiB	26 MiB	142 MiB	284 MiB	273 MiB
	SHA-3, Mul	18 MiB	25 MiB	127 MiB	244 MiB	348 MiB

Tabelle 5: Ergebnisse der Messung des Speicherplatzbedarfs des Prototyps unter Verwendung von SHA-256 und SHA-3 (256 Bit)

Grundsätzlich steigt der benötigte Speicherplatz mit der Anzahl der Tripel. Es gibt in dieser Tabelle nur wenige Ausnahmen, die ich mit Messungenauigkeiten erkläre. In Abbildung 13 sind die Werte der Zeile „Short Abstracts(de)“ in Diagrammform dargestellt. Dieses Diagramm dient exemplarisch für alle Diagramme, die man aus den Werten der Zeilen in der Tabelle erhalten würde, da hier wiederum alle Kurven einen ähnlichen Verlauf haben. Man kann am Diagramm erkennen, dass sich der Speicherplatzbedarf bei steigender Anzahl an Tripeln ebenfalls vergrößert.

Betrachtet man das Diagramm zu dem durchschnittlich benötigten Speicherplatz pro Tripel (siehe Abbildung 14), kann man erkennen, dass die Kurven zunächst sehr stark abfallen, um dann fast konstant zu verlaufen. Ich schließe daraus, dass das Programm einen konstanten Grundbedarf an Arbeitsspeicher hat und zusätzlich einen variablen Speicherplatzbedarf pro Tripel. Bei wenigen Tripeln ist der Grundbedarf anteilig am Gesamtbedarf groß. Der Anteil

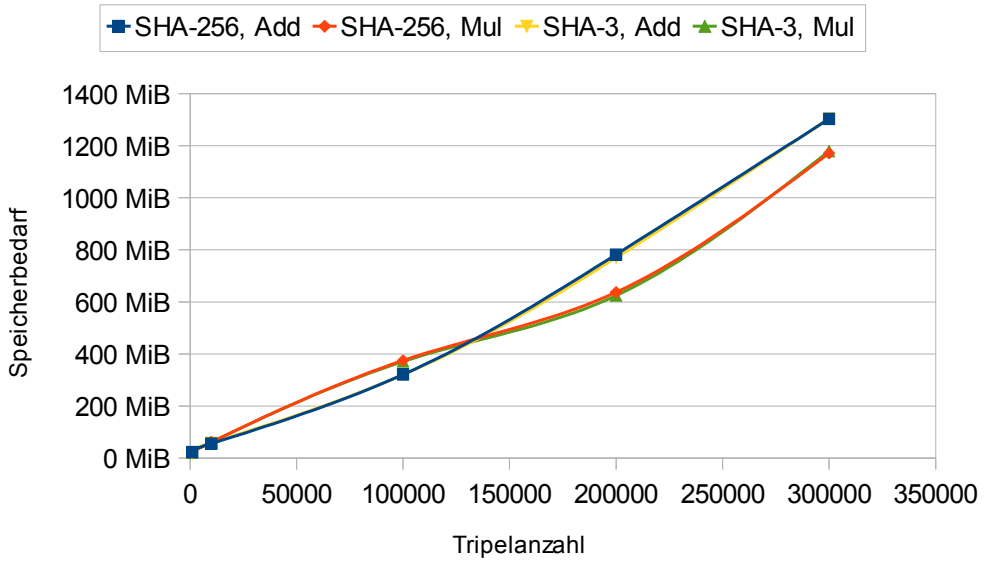

Abbildung 13: Speicherplatzbedarf des Prototyps unter Verwendung von SHA-256 und SHA-3, verwendeter Datensatz: „Short Abstracts(de)“

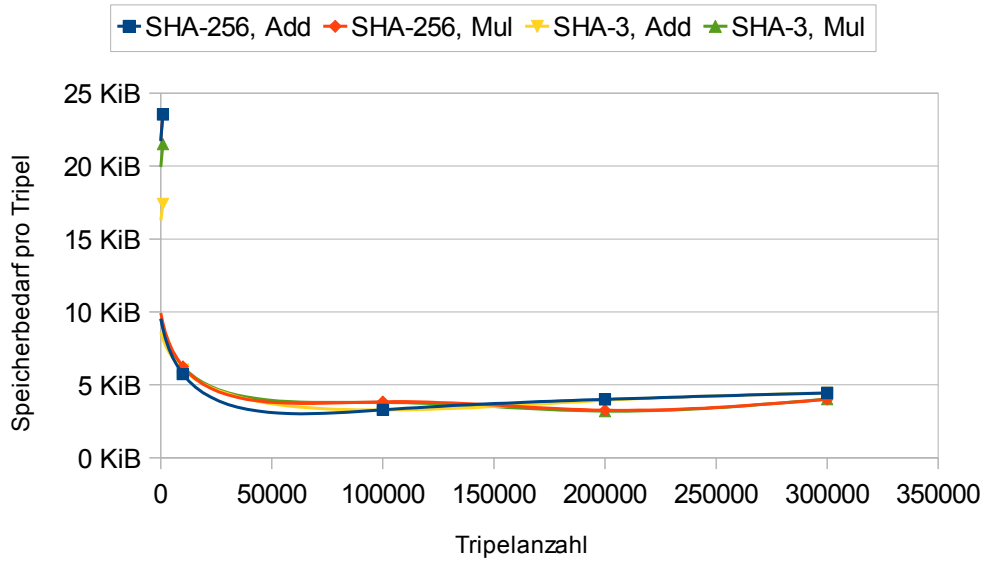

Abbildung 14: Durchschnittlich benötigter Speicherplatzbedarf pro Tripel unter Verwendung von SHA-256 und SHA-3, verwendeter Datensatz: „Short Abstracts(de)“

des Grundbedarfs nimmt dann mit steigender Tripelanzahl ab. Da die Kurven nach ihrem starken Abfall annähernd konstant verlaufen, schließe ich auf einen linearen Zusammenhang zwischen der Anzahl der Tripel und dem benötigten Arbeitsspeicher.

Beim Speicherplatzbedarf gibt es keine Überraschungen. Wie erwartet steigt der Speicherplatzbedarf proportional zur Anzahl der Tripel. Der Prototyp benötigt insgesamt recht viel Arbeitsspeicher, so dass damit nicht die kompletten von DBpedia bereitgestellten Datensätze verarbeitet werden können. Jedoch sind diese Datensätze auch sehr umfangreich und es besteht die Frage, ob in der Praxis tatsächlich so große Datensätze benutzt werden. Falls man doch solch große Datensätze verarbeiten möchte, muss man auf andere Speichermodelle, wie Datenbanken oder Dateien, zurückgreifen.

5 Zusammenfassung

5.1 Fazit

In meiner Arbeit habe ich einen Algorithmus vorgestellt, der Prüfsummen aus RDF-Daten im Speicher eines Computers berechnen kann. Dieser Algorithmus unterstützt die Aufteilung eines großen RDF-Datensatzes in mehrere kleinere Datensätze durch das Feature, Teilprüfsummen zu errechnen. Dies wirkt sich positiv für potentielle Nutzer aus, die nur einen kleinen Teil der Daten benötigen. Dadurch müssen sie nicht den gesamten RDF-Datensatz, sondern nur einen kleinen Teil überprüfen, um sicherzustellen, dass niemand die Daten manipuliert hat. Jedoch muss jemand, der die Daten signieren möchte, mehrere Teildatensätze signieren.

Des Weiteren habe ich einen Prototyp zur Umsetzung dieses Algorithmus konzeptioniert und implementiert. Durch Testläufe habe ich Messungen zu Laufzeit und Speicherplatzbedarf durchgeführt, um die Praxistauglichkeit des Algorithmus zu testen. Dabei habe ich festgestellt, dass der Prototyp, bis zu einer gewissen Anzahl an zu verarbeitenden Tripeln, brauchbar ist.

Bei der Auswertung der Testdatensätze konnte ich feststellen, dass die Laufzeit mit steigender Anzahl der Tripel zunimmt. Es ist allerdings kein einfacher funktionaler Zusammenhang zur Anzahl der Tripel abzuleiten. Meines Erachtens ist eine Laufzeitlänge von mehr als einer Stunde für einen potentiellen Nutzer nicht akzeptabel. Ab wann die Länge der Laufzeit des Prototyps diesen Grenzwert überschreitet, ist jedoch aus den Messergebnissen nicht abzuleiten. Dies muss an dieser Stelle offen bleiben und obliegt weiteren Untersuchungen. Bei diesen Untersuchungen muss die Messpunktdichte erhöht werden, um mehr Stützpunkte für eine abzuleitende Kurve zu erhalten. Außerdem muss der Wertebereich weit über die Tripelanzahl von 300 000 hinaus gehen. Da aber ein potentieller Nutzer mit Hilfe des erstellten Prototyps in ca. 3 Minuten bereits 300 000 Tripel bearbeitet hat, wird der Prototyp für die meisten Fälle in der Praxis im Hinblick auf die Laufzeit tauglich sein.

Bei den Testdatensätzen „Extended Abstracts(de)“ und „Short Abstracts(de)“ konnte ich feststellen, dass bei einer Tripelanzahl von 300 000 die Höchstgrenze für den Speicherplatz der Java VM von ca. 1 600 MiB fast erreicht wurde. Also ist zu erwarten, dass diese Grenze bei einer Tripelanzahl, die nur wenig darüber liegt, überschritten wird. Bei den Datensätzen „Titles(de)“ und „Images(de)“ wird diese Grenze erst bei ca. der vierfachen Anzahl der Tripel erreicht. Das liegt daran, dass die Anzahl der Zeichen innerhalb der letztgenannten beiden

Datensätze erheblich geringer ist.

Die Ergebnisse machen deutlich, dass der Speicherplatzbedarf von der Art der Daten abhängig ist. Da die Anzahl der Zeichen in den Tripeln von „Long Abstracts(de)“ und „Short Abstracts(de)“ außergewöhnlich groß ist, wird der Prototyp für die meisten Anwendungen in der Praxis im Hinblick auf den Speicherplatzbedarf tauglich sein.

Die Wahl der Verknüpfungsfunktion hat sich auf die Rechenzeit messbar ausgewirkt. Die durchgeführten Tests lassen erkennen, dass die Rechenzeit mit der multiplikativen Verknüpfungsfunktion länger ist, als mit der additiven Verknüpfungsfunktion.

Alles in allem ist es möglich, durch den vorgestellten Algorithmus RDF-Daten im Arbeitsspeicher zu hashen. Der Arbeitsspeicher begrenzt jedoch die Größe der RDF-Datensätze, die verarbeitet werden können. Außerdem sollte ein Nutzer nicht länger als eine Stunde auf ein Ergebnis warten müssen. Daher ist der Algorithmus für die Praxis tauglich, aber bei sehr großen Datenmengen nicht einsetzbar.

5.2 Ausblick

Der Prototyp sollte entsprechend der Aufgabenstellung zeigen, dass es möglich ist, den entwickelten und vorgestellten Algorithmus umzusetzen. Dieses Ziel wurde erreicht. Der Prototyp wurde in relativ kurzer Zeit erstellt und ist daher nicht gänzlich ausgereift. Hier besteht noch Potenzial für eine Optimierung.

Für eine Weiterentwicklung des Prototypen sind verschiedene Verbesserungsmöglichkeiten denkbar, die in zukünftigen Arbeiten berücksichtigt werden können:

Es ist möglich, bestimmte Teile des Algorithmus, z. B. die Berechnung der Teilprüfsummen, zu parallelisieren und auf mehrere Rechenkerne aufzuteilen, um die Gesamtlaufzeit zu verringern.

Um das Problem mit dem unzureichenden Arbeitsspeicher zu lösen, wird man eine andere Möglichkeit zur Zwischenspeicherung, z. B. eine Datenbank, wählen müssen. Eine weniger aufwendige Lösung wäre, den Prototypen in 64 Bit zu kompilieren und auf einem 64 Bit Java Runtime Environment auszuführen. Dadurch kann dann wenigstens der vorhandene Arbeitsspeicher komplett genutzt werden.

Anhang

Laufzeit und Speicherplatzverbrauch mit MD5 und SHA-1

Daten-satz	Einstellungen	Laufzeit bei einer Tripelanzahl von 1 000	10 000	100 000	200 000	300 000
Titles (de)	MD5, Add	0,015 s	0,436 s	28,719 s	111,057 s	166,484 s
	MD5, Mul	0,015 s	0,530 s	29,562 s	111,868 s	169,791 s
	SHA-1, Add	0,015 s	0,421 s	28,470 s	111,150 s	167,264 s
	SHA-1, Mul	0,046 s	0,608 s	29,749 s	113,755 s	171,242 s
Short Abstr. (de)	MD5, Add	0,015 s	0,530 s	29,343 s	113,053 s	169,760 s
	MD5, Mul	0,046 s	0,639 s	30,170 s	114,645 s	171,975 s
	SHA-1, Add	0,015 s	0,561 s	29,827 s	113,069 s	169,744 s
	SHA-1, Mul	0,046 s	0,670 s	31,137 s	116,080 s	173,426 s
Ext. Abstr. (de)	MD5, Add	0,015 s	0,608 s	30,045 s	114,676 s	174,299 s
	MD5, Mul	0,031 s	0,670 s	31,153 s	117,733 s	176,015 s
	SHA-1, Add	0,031 s	0,670 s	30,841 s	115,003 s	175,126 s
	SHA-1, Mul	0,046 s	0,795 s	31,590 s	118,201 s	179,884 s
Images (de)	MD5, Add	0,015 s	0,280 s	14,570 s	54,600 s	160,041 s
	MD5, Mul	0,015 s	0,280 s	15,194 s	56,597 s	164,159 s
	SHA-1, Add	0,015 s	0,280 s	14,414 s	55,910 s	161,585 s
	SHA-1, Mul	0,031 s	0,405 s	16,005 s	58,157 s	165,439 s

Tabelle 6: Ergebnisse der Messung der Laufzeiten mit den Hashfunktionen MD5 und SHA-1

Daten-		Speicherplatzbedarf bei einer Tripelanzahl von				
satz	Einstellungen	1 000	10 000	100 000	200 000	300 000
Titles (de)	MD5, Add	18 MiB	25 MiB	151 MiB	222 MiB	372 MiB
	MD5, Mul	15 MiB	26 MiB	124 MiB	263 MiB	324 MiB
	SHA-1, Add	18 MiB	25 MiB	110 MiB	234 MiB	415 MiB
	SHA-1, Mul	16 MiB	26 MiB	116 MiB	248 MiB	311 MiB
Short Abstr. (de)	MD5, Add	20 MiB	48 MiB	434 MiB	666 MiB	1 114 MiB
	MD5, Mul	21 MiB	50 MiB	379 MiB	622 MiB	1 088 MiB
	SHA-1, Add	20 MiB	58 MiB	329 MiB	786 MiB	1 265 MiB
	SHA-1, Mul	21 MiB	59 MiB	330 MiB	787 MiB	904 MiB
Ext. Abstr. (de)	MD5, Add	21 MiB	69 MiB	648 MiB	1 106 MiB	1 433 MiB
	MD5, Mul	22 MiB	81 MiB	512 MiB	1 333 MiB	1 461 MiB
	SHA-1, Add	22 MiB	77 MiB	475 MiB	1 264 MiB	1 505 MiB
	SHA-1, Mul	23 MiB	81 MiB	622 MiB	1 124 MiB	1 411 MiB
Images (de)	MD5, Add	20 MiB	29 MiB	144 MiB	282 MiB	280 MiB
	MD5, Mul	18 MiB	23 MiB	118 MiB	242 MiB	254 MiB
	SHA-1, Add	17 MiB	26 MiB	110 MiB	232 MiB	357 MiB
	SHA-1, Mul	19 MiB	25 MiB	125 MiB	253 MiB	251 MiB

Tabelle 7: Ergebnisse der Messung des Speicherplatzbedarfs mit den Hashfunktionen MD5 und SHA-1

Literatur

Alle URLs wurden am 11.01.2013 abgerufen.

[1] Graham Klyne und Jeremy J. Carroll. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C. 2004-02-10. URL: `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`.

[2] Frank Manola und Eric Miller. *RDF Primer*. W3C. 2004-02-10. URL: `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`.

[3] Patrick Hayes. *RDF Semantics*. W3C. 2004-02-10. URL: `http://www.w3.org/TR/2004/REC-rdf-mt-20040210/`.

[4] Jan Grant und Dave Beckett. *RDF Test Cases*. W3C. 2004-02-10. URL: `http://www.w3.org/TR/2004/REC-rdf-testcases-20040210/`.

[5] Dan Brickley und R.V. Guha. *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C. 2004-02-10. URL: `http://www.w3.org/TR/2004/REC-rdf-schema-20040210/`.

[6] Dave Beckett. *RDF/XML Syntax Specification (Revised)*. W3C. 2004-02-10. URL: `http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/`.

[7] Klaus Schmeh. *Kryptografie: Verfahren, Protokolle, Infrastrukturen*. 4. Aufl. iX-Edition. Heidelberg: dpunkt.Verlag, 2009. ISBN: 978-3-89864-602-4.

[8] Vlastimil Klíma. „Tunnels in Hash Functions: MD5 Collisions Within a Minute“. In: *IACR Cryptology ePrint Archive* 2006 (2006), S. 105. URL: `http://eprint.iacr.org/2006/105`.

[9] Xiaoyun Wang, Yiqun Lisa Yin und Hongbo Yu. „Finding Collisions in the Full SHA-1“. In: *Advances in Cryptology - CRYPTO 2005*. Hrsg. von Victor Shoup. Bd. 3621. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, S. 17–36. ISBN: 978-3-540-28114-6. DOI: `10.1007/11535218_2`. URL: `http://dx.doi.org/10.1007/11535218_2`.

[10] Chad Boutin. *NIST Selects Winner of Secure Hash Algorithm (SHA-3) Competition*. NIST. 2012-10-02. URL: `http://www.nist.gov/itl/csd/sha-100212.cfm`.

[11] Jeremy J. Carroll. *Signing RDF Graphs*. Techn. Ber. HPL-2003-142. HP Labs, 2003-07-23. URL: `http://www.hpl.hp.com/techreports/2003/HPL-2003-142.html`.

[12] Giovanni Tummarello u. a. „Signing individual fragments of an RDF graph“. In: *Special interest tracks and posters of the 14th international conference on World Wide Web*. WWW '05. ACM, 2005, S. 1020–1021. ISBN: 1-59593-051-5. DOI: 10.1145/1062745.1062848. URL: http://doi.acm.org/10.1145/1062745.1062848.

[13] Mark Giereth. „On partial encryption of RDF-Graphs“. In: *Proceedings of the 4th international conference on The Semantic Web*. ISWC '05. Springer-Verlag, 2005, S. 308–322. ISBN: 3–540-29754-5, 978-3-540-29754-3. DOI: 10.1007/11574620_24. URL: http://dx.doi.org/10.1007/11574620_24.

[14] Craig Sayers und Alan H. Karp. *Computing the digest of an RDF graph*. Techn. Ber. HPL-2003-235(R.1). HP Labs, 2004-03-23. URL: http://www.hpl.hp.com/techreports/2003/HPL-2003-235R1.html.

[15] Craig Sayers und Alan H. Karp. *RDF Graph Digest Techniques and Potential Applications*. Techn. Ber. HPL-2004-95. HP Labs, 2004-05-28. URL: http://www.hpl.hp.com/techreports/2004/HPL-2004-95.html.

[16] Jeremy J. Carroll u. a. „Named Graphs, Provenance and Trust“. In: *Proceedings of the 14th international conference on World Wide Web*. WWW '05. ACM, 2005, S. 613–622. ISBN: 1-59593-046-9. DOI: 10.1145/1060745.1060835. URL: http://doi.acm.org/10.1145/1060745.1060835.

[17] Richard Cyganiak und David Wood. *RDF 1.1 Concepts and Abstract Syntax*. W3C. 2012-06-05. URL: http://www.w3.org/TR/2012/WD-rdf11-concepts-20120605/.

[18] U.S. Department of Commerce / National Institute of Standards and Technology. *FIPS Publication 180-4: Secure Hash Standard (SHS)*. Techn. Ber. FIPS PUB 180-4. U.S. Department of Commerce / National Institute of Standards and Technology, 2012-03. URL: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf.