Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

DANIEL KNOGL
MASTER THESIS

DESIGN AND IMPLEMENTATION OF GRAPH-DB BASED STORAGE FOR WIKIPEDIA ARTICLES

Eingereicht am 30. September 2016

Betreuer: Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 30. September 2016

License

Erlangen, 30. September 2016

Abstract

Despite the lack of crucial features, Wiki Markup is still the primary data format in Wikipedia. The Wiki Object Model (WOM) features a modern alternative based on a tree structure. The use of a grapbased Storage for integrating WOM as the primary data format in Wikipedia seems likely. Managing the immense revision history of Wikipedia articles is one of many problems when facing this approach. In most cases, the difference between a revision and its succesor is small. Hence, there are many redundancies inside the database. To solve this problem we have to reduce the amount of redundancies. For this purpose an algorithm was designed connecting nearby revision graphs and reusing parts of the predecessor graph. Moreover, strategies for traversing WOM resources are introduced and user-defined edges between two arbitrary nodes are established. Multiple tests with real Wikipedia articles are performed for evaluating performance and storage savings. Thereby different configurations are tested. Redundancies between nearby revisions are stripped down to a minimum when using the graphbased storage for Wikipedia articles. In addition, all the advantages provided by WOM are given.

Zusammenfassung

Selbst auf aktuellem Stand der Technik werden Seiten der Wikipedia stets im textuellen Wikitext gespeichert. Das auf einer Baumstruktur basierende Wiki Object Model (WOM) bietet eine moderne Alternative zum aktuellen Format. Zur Integration von WOM bietet sich die Verwendung einer Graphdatenbank an. Ein Problem dieses Ansatzes liegt im Verwalten der umfangreichen Revisionshistory der Wikipedia. Oft liegen nur kleine Änderungen zwischen zwei Revisionen, die auf Datenbankebene zu erheblichen Redundanzen führen. Ein Ziel dieser Arbeit ist die Reduzierung redundanter Graphen mit Hinblick auf Speicherplatzeinsparungen. Dafür wurde ein Algorithmus entwickelt, der die Graphen benachbarter Revisionen miteinander verknüpft um Teilgraphen wiederzuverwenden. Zusätzlich werden Strategien zur Traversierung von WOM Ressourcen innerhalb der Graphdatenbank vorgestellt mit der Möglichkeit nutzerspezifische Kanten zwischen zwei beliebigen WOM Knoten zu setzen. Zur Evaluation wurden mehrere Tests mit realen Wikipedia Artikeln durchgeführt, um Leistungsfähigkeit und Menge der Einsparungen zu analysieren. Dabei sind unterschiedliche Konfigurationen getestet worden. Durch Verwendung des graphbasierten Speichers ist sichergestellt, dass Redundanzen benachbarter Revisionen auf ein geeignetes Minimum reduziert sind und sämtliche Vorteile des WOM Formats gegeben werden.

Inhaltsverzeichnis

1 Graphbasierter Speicher für Wikipedia Artikel **1**
- 1.1 Zielsetzung und Absicht . . . 2
- 1.2 Technische Anforderungen . . . 3
  - 1.2.1 Datenrepräsentation . . . 3
  - 1.2.2 Datenvolumen . . . 5
  - 1.2.3 Anfrageverhalten . . . 6
- 1.3 Funktionale Anforderungen . . . 7

2 Architektur und Design **11**
- 2.1 Datenbanksystem . . . 11
  - 2.1.1 Titan . . . 12
  - 2.1.2 Apache Cassandra . . . 13
- 2.2 Verwaltung von Wikipedia-Seiten . . . 14
  - 2.2.1 Knotentypen . . . 14
  - 2.2.2 Kantentypen . . . 15
  - 2.2.3 Abbildung von WOM-Ressourcen . . . 16
  - 2.2.4 Traversierung von WOM-Ressourcen . . . 17
  - 2.2.5 Graphoptimierung benachbarter Revisionen . . . 18
  - 2.2.6 Begrenzung der Baumtiefe . . . 20
- 2.3 Nutzerspezifische Kanten . . . 21
  - 2.3.1 Traversierung von nutzerspezifischen Kanten . . . 22

3 Implementierung **28**
- 3.1 Datenbanksystem . . . 29
  - 3.1.1 TinkerPop . . . 29
  - 3.1.2 Titan mit Cassandra Backend . . . 30
- 3.2 Seitenverwaltung . . . 33
  - 3.2.1 Traversieren von WOM-Ressourcen . . . 36
  - 3.2.2 Algorithmus zur Graphoptimierung . . . 39
  - 3.2.3 Begrenzung der Baumtiefe . . . 51
- 3.3 Nutzerspezifische Kanten . . . 55

4 Evaluation **59**
4.1 Bewertung hinsichtlich der Anforderungen 59
4.2 Leistungsfähigkeit und Performanz 60
4.3 Speicherplatzeinsparungen 65

5 Zusammenfassung und Ausblick **67**

Literaturverzeichnis **70**

1 Graphbasierter Speicher für Wikipedia Artikel

Der Zuwachs an Daten und das daraus resultierende Wissen ist steigt stetig an. In vielen Fällen werden diese Daten in sogenannten Wikis verwaltet und bereitgestellt. Dabei können Benutzer nicht nur die Inhalte abrufen sondern auch eigene Änderungen und Daten in die Wiki laden. Das Ziel ist eine gemeinschaftliche Dokumentation einer kollektiven Wissensbasis. Das populärste Beispiel ist die Wikipedia mit Millionen von Wissensartikeln. Seit dem 15. Januar 2001 können Artikel in der Wikipedia erstellt und abgefragt werden[1]. Dabei werden sämtliche Artikel in einer entsprechenden Wiki Markup Language (WML) verfasst. Weil kein allgemeiner Standard unter den vielen Wiki Projekten herrscht, existieren unterschiedliche Arten von WMLs. Das gemeinsame Ziel ist die einfache textuelle Repräsentation eines Artikels um anschließend daraus Hypertext Markup Language (HTML) für das Anzeigen im Browser zu erzeugen. Dohrn und Riehle (2013) haben analysiert, dass die Verwendung von WML mit Nachteilen verbunden ist. So werden in vielen Wiki-Engines keine automatisierten Transformationen für Autoren angeboten, mit der Möglichkeit mehrere Artikel gleichzeitig zu modifizieren. Auch die fehlende Abstraktion der Artikelinhalte zur Förderung von Datenaustausch, Inhaltstransformationen, visueller Bearbeitung von Artikeln und Abfragen sind auf die Verwendung von WML zurückzuführen. Eine Alternative bietet das von Dohrn und Riehle (2013) entwickelte Wiki Object Model (WOM). Durch die Verwendung von WOM werden Artikel als Baumstruktur mit verschiedenen Knoten als Syntaxelemente des ursprünglichen Wiki Markups repräsentiert. Dadurch eröffnen sich neue Möglichkeiten zum Abspeichern und Verwalten solcher baumartigen Artikel. Naheliegend wäre demnach die Verwendung eines graphbasierten Speichers für Wikipedia Artikel.

[1] https://en.wikipedia.org/wiki/Wikipedia

1.1 Zielsetzung und Absicht

Seit Januar 2015 hat die englische Wikipedia ihren Artikelbestand von 4,8 Millionen auf 5,1 Millionen im Januar 2016 vergrößert [2]. Dabei wurden bei einem jährlichen Revisionszuwachs von über 9 Prozent insgesamt über 38.340.000 neue Revisionen in der Wikipedia erzeugt. Der Grund für diese extreme Zunahme liegt im Content Management System (CMS) der Wikipedia und dessen Historie für Artikelrevisionen. Wird ein neuer Artikel angelegt oder ein bereits existierender Artikel bearbeitet, erzeugt das CMS eine eigenständige Revision. Um eine vollständige Historie zu gewährleisten, werden alte Revisionen nicht gelöscht sondern bleiben im CMS bestehen. Die durchschnittliche Anzahl der Revisionen pro Artikel liegt aktuell bei 92,6 und steigt seit mehreren Jahren stetig an weil immer öfter Artikel geändert oder aktualisiert werden. Oft liegen nur kleine Änderungen zwischen zwei benachbarten Artikelrevisionen. Trotzdem wird jede Revisionen vollständig neu abgespeichert, was zu einer hohen Datenredundanz führt.

Ziel dieser Arbeit ist das Design und die prototypische Implementierung eines graphbasierten Speichers für Wikipedia Artikel. Dabei wird eine eine Lösung vorgestellt um Wikipedia Artikel zu verwalten. Ein Teil dieser Lösung ist die umfangreiche Revisionshistorie der Wikipedia, die in jedem Fall gegeben sein muss. Zusätzlich werden Strategien zum Traversieren und Verarbeiten von Wikipedia Artikeln in der Datenbank präsentiert. Ein weiteres Ziel ist die Reduzierung der bereits erwähnten Redundanzen zwischen benachbarten Revisionen. Hier wird ein Algorithmus gezeigt, der Teile des WOM-Baums der Vorgängerrevision wiederverwendet um den Baum der Nachfolgerrevision zu erzeugen. Weitere Unterziele sind die Etablierung nutzerspezifischer Kanten im Graphspeicher, sowie das Traversieren der WOM-Ressource innerhalb der Datenbank.

Nicht Teil dieser Arbeit ist die Verwaltung der Meta-Informationen einer Seite oder Revision. Es macht keinen Unterschied, ob die Beziehung zwischen Vorgänger- und Nachfolger-Revision in der Datenbank existiert oder nicht. Der graphbasierte Speicher verwaltet ausschließlich Artikel beziehungsweise WOM-Ressourcen. Dadurch ist der Graphspeicher kein vollständiges CMS für Wikipedia Artikel sondern vielmehr Teil eines Gesamtsystems. Dieses System nutzt den graphbasierten Speicher um WOM-Ressourcen zu verwalten. Die umfangreichen Funktionalitäten einer Wiki-Engine werden demnach nicht durch den graphbasierten Speicher für Wikipedia Artikel bereitgestellt.

[2] https://stats.wikimedia.org/EN/TablesWikipediaEN.htm

1.2 Technische Anforderungen

Im folgenden Abschnitt werden sämtliche technischen Anforderungen des graphbasierten Datenspeichers behandelt. Dazu gehört das primäre Datenformat und die genaue Repräsentation der Artikel auf Datenebene. Zusätzlich werden weitere Anforderungen an das Datenvolumen und die Kapazität der Datenbank gestellt. Diesbezüglich müssen Menge und Größe der Wikipedia Daten untersucht und spezifiziert werden. Des Weiteren wird das Anfrageverhalten der Nutzer analysiert und bewertet. Häufige Operationen des graphbasierten Speichers müssen untersucht und als Anforderung gestellt werden.

1.2.1 Datenrepräsentation

Artikel in der Wikipedia werden in Wikitext verfasst. Diese leichtgewichtige Sprache ermöglicht es dem Nutzer, einen strukturierten Text zu erstellen und diesen über eine Wiki-Engine als Datei abzuspeichern. In Wikipedia wird dafür die Software MediaWiki eingesetzt um Artikel oder Seiten zu verwalten. Die WML stellt das zentrale Datenformat der Artikel dar und wird bei Anfragen über WikiMedia nach HTML konvertiert. Obwohl WMLs mittlerweile ein breites Spektrum an Features anbieten, entstehen durch die Verwendung klare Nachteile. Ein großes Problem ist die fehlende Abstraktion des Datenformats. Dadurch entstehen Schwierigkeiten beim visuellen Bearbeiten von Artikeln, Inhaltstransformationen, Abfragen, Datenaustausch oder Abspeichern von Daten. Abhilfe schafft hier ein neues Format, welches von Dohrn und Riehle (2011) entwickelt wurde. WOM versucht da anzuknüpfen, wo herkömmliche WMLs versagen. Mit WOM ist ein Format entstanden, das deutlich besser verarbeitet werden kann als herkömmliche WMLs. Unter anderem können Refactorings oder Artikeltransformationen an mehreren Artikeln deutlich effektiver gelöst werden. Im Folgenden werden weitere Details von WOM erläutert, um ein besseres Verständnis über die genauen Anforderungen zu bekommen. Das WOM-Format bildet die Grundlagen für Design- und Architekturentscheidungen. Um diese Entscheidungen nachvollziehen zu können, ist ein tieferer Einblick in das Format erforderlich.

Wie bereits erwähnt wurde, das WOM-Format von Dohrn und Riehle (2011) an der Friedrich-Alexander-Universität Erlangen entwickelt. Seit 2011 existieren mehrere Versionen von WOM, weshalb im weiteren Verlauf ausschließlich auf das neuste WOM 3.0 Format Bezug genommen wird. WOM lehnt sich dabei stark an HTML oder XHTML an und besteht aus verschiedenen Knoten. Diese Knoten können unterschiedliche Eigenschaften besitzen und legen den Grundstein für das Format. Die Menge aller Knoten bildet ähnlich wie im Domain Object Model (DOM) bei HTML einen Baum, der sich problemlos traversieren lässt. Ein

Großteil der aufkommenden Knotenypen wurde direkt aus einer Untermenge von XHTML übernommen. Alle restlichen Knoten sind WOM spezifisch und können der Arbeit von Dohrn und Riehle (2011) entnommen werden. Die Eltern-Kind-Beziehungen der Knotentypen untereinander werden über Regeln festgelegt. Diese Regeln definieren, welche Knotentypen als potentielle Kind-Knoten in Frage kommen und welche Knotentypen als Kind-Knoten ausgeschlossen werden müssen. In Abbildung 1.1 wird ein Beispiel gezeigt, wie ein herkömmlicher, in Wikitext verfasster Artikel als WOM-XML dargestellt wird. Die WOM-Knoten entsprechen den einzelnen Syntax Elementen der WML und bilden eine Baumstruktur.

```
== Simple Page ==

This is a '''simple page'''. It contains a link and a list:
* [[SomewhereIItem 1]]
* Item 2
```

```
<article xmlns:mww="http://sweble.org/schema/mww30" version="3.0"
       title="Simple_Page" xmlns="http://sweble.org/schema/wom30">
    <body>
        <section level="2">
            <heading>
                <rtd>==</rtd>
                <text> Simple Page </text>
                <rtd>==</rtd>
            </heading>
            <body>
                <text></text>
                <p topgap="0" bottomgap="0">
                    <text>This is a </text>
                    <b>
                        <rtd>'''</rtd>
                        <text>simple page</text>
                        <rtd>'''</rtd>
                    </b>
                    <text>. It contains a link and a list:</text>
                </p>
                <ul>
                    <li>
                        <rtd>*</rtd>
                        <text> </text>
                        <intlink target="Somewhere">
                            <rtd>[[Somewhere</rtd>
                            <title>
                                <rtd>|</rtd>
                                <text>Item 1</text>
                            </title>
                            <rtd>]]</rtd>
                        </intlink>
                    </li>
                    <li>
                        <rtd>*</rtd>
                       <text> Item 2</text>
                    </li>
                </ul>
            </body>
        </section>
    </body>
</article>
```

Abbildung 1.1: Wikitext (links) und WOM-XML im Vergleich.

Sämtliche Seiten aus der Wikipedia werden im Weiteren als WOM betrachtet. Die Konvertierung von Wikitext in das WOM-Format wird als gegeben vorausgesetzt und als Anforderung definiert. Grund für diese Annahme sind die oben genannten Vorteile sowie die Nähe zum graphbasierten Speicher. Ziel ist die Entwicklung einer Graphdatenbank, die Wikipedia-Seiten als WOM verwaltet.

1.2.2 Datenvolumen

Als Datenquelle wird die englische Wikipedia mit mehr als 5.214.073 Artikeln und 39.934.093 Seiten insgesamt festgelegt (Stand: August 2016). Die Gesamtanzahl der Seiten setzt sich demnach aus aus allen Artikeln, internen Seiten, Weiterleitungen und sonstigen Seiten zusammen. Die englische Version der Wikipedia ist die bisher größte Distribution der Wikipedia. Um die vollständige Menge an Daten zu betrachten, dürfen nicht nur alle Seiten betrachtet werden, sondern sämtliche Revisionen, die in der Wikipedia existieren. Die gesamte Anzahl der Revisionen beträgt über alle Seiten der Wikipedia mehr als 843.105.041 und setzt sich aus sämtlichen Editierungen zusammen. Das Gesamtvolumen aller Revisionen ergibt somit eine durchschnittliche Anzahl von 21,1 Revisionen pro Seite. Editierungen und Revisionen können hier gleichgesetzt werden, weil jede Editierung automatisch zu einer neuen Revision führt. Da die eigentlichen Artikel mit Inhalt ungefähr ein Achtel aller Seiten ausmachen, werden diese nochmals separat betrachtet. Es ergibt sich eine durchschnittliche Anzahl von 92,1 Revisionen pro Artikel, was deutlich über dem Durchschnitt der Revisionen pro Seite liegt. Als Mindestvolumen werden somit alle Artikelrevisionen festgelegt, wodurch sich eine Gesamtanzahl von über 480.216.123 Revisionen ergibt.

Die genaue Größe der Daten in Byte kann auf Grund der enormen Menge nur grob bestimmt werden. Die öffentlich verfügbaren Dumps der Wikipedia-Datenbank enthalten sämtliche Seiten mit Edithistorie und damit über 100 GB an komprimierten XML Daten. Dabei handelt es sich ausschließlich um Text- und Metadaten ohne Bilder, Videos oder sonstige Media-Daten. Im unkomprimierten Zustand steigt die Größe der XML Daten auf mehrere Terrabyte an und übersteigt bereits die Verarbeitungskapazität kleiner Datenbanksysteme. Zwar handelt es sich bei den Daten um reine XML-Repräsentationen der Seiten, dennoch kann davon ausgegangen werden, dass die Datenmenge im WOM-Format nicht erheblich kleiner ist. Im Grunde müssen sämtliche Text- und Metadaten abgebildet werden, um eine Rekonstruktion des originalen Artikels zu ermöglichen. Diese Daten beinhalten das meiste Volumen und benötigen somit die größte Speicherkapazität.

Werden diese Daten auf das WOM-Format bezogen, müssen weitere Größen betrachtet werden. Jede WOM-Ressource besteht aus Knoten. Um sämtliche Knoten aller WOM-Ressourcen abzubilden, muss eine geeignete Graphdatenbank verwendet werden. Hinzu kommt, dass implizite Kanten zwischen den einzelnen Knoten existieren, die ebenfalls von der unterliegenden Graphdatenbank verwaltet werden müssen. Um eine bessere Vorstellung von der Menge an Knoten und Kanten zu bekommen, werden im Folgenden die genauen Mengen analysiert. Die analysierten Daten stammen direkt aus $\approx$ 48.000 Artikeln der englischen Wikipedia und wurden dem offiziellen Wikipedia Dump vom 13.01.2016 entnommen. Alle Seiten wurden im WOM-Format vorgelegt und hinsichtlich der Knotenan-

zahl analysiert. Als Ergebnis lässt sich eine durchschnittliche Anzahl von über 2.841 Knoten pro Revision errechnen. Wird diese Anzahl auf mehr als 800 Millionen Seitenrevisionen hochgerechnet, ergibt sich einen Gesamtmenge von über 2,2 Billionen Knoten, die von der Graphdatenbank unterstützt werden müssen. Allerdings werden keine Speicheroptimierungen und Maßnahmen zur Reduzierung von redundanten Knoten einbezogen, weshalb die eigentliche Gesamtanzahl der Knoten deutlich geringer ist. Dennoch muss im schlimmsten Fall von einer derartig großen Knotenmenge ausgegangen werden, weshalb die genannte Zahl als Mindestanforderung an die Graphdatenbank gilt.

Zusätzlich zum aktuellen Datenvolumen kommt der stetige Wachstum der Wikipedia hinzu. Aktuell sieht die englische Wikipedia einen Wachstum von 795 Artikel pro Tag mit über 92,6 Editierungen pro Monat. Diese Umstände erfordern nicht nur ein System, das mit riesigen Mengen von Daten fertig wird, sondern zudem noch hoch skalierbar ist. Als weitere Anforderung wird demnach die Skalierbarkeit und der ständige Wachstum an Daten definiert.

1.2.3 Anfrageverhalten

Der größte Unterschied zu herkömmlichen Datenbanken ist die Abfragecharakteristik in der Wikipedia. Die auftretenden schreibenden Operationen in der Wikipedia sind das Anlegen und Erstellen neuer und das Bearbeiten bereits vorhandener Artikel. Durch die umfangreiche Historie werden bereits vorhandene Seiten weder gelöscht noch geändert. Selbst das Bearbeiten von Artikeln führt zu einer eigenständigen und neuen Version dieses Artikels ohne Änderungen in der Vorgängerversion. Somit stehen lediglich Einfügeoperationen als schreibende Tätigkeit im Vordergrund.

Hauptaugenmerk der Wikipedia liegt dennoch auf den lesenden Operationen. Aus diesem Grund müssen Anfragen schnell verarbeitet und in einem angemessen Zeitfenster beantwortet werden. Im Gegensatz zu schreibenden Operationen zählt hier die Antwortzeit, die dementsprechend optimiert werden muss. Zusätzlich muss beachtet werden, dass hauptsächlich die aktuelle Version der Seite angefordert wird und vorherige Versionen nur im Einzelfall angefragt werden.

Als Anforderung wird demnach ein System definiert, welches lesende Anfragen in angemessener Zeit beantworten kann. Schreibende Anfragen hingegen können deutlich zeitintensiver und aufwändiger sein. Hier wird kein genaues Zeitfenster festgelegt, sondern vielmehr die Anforderung, dass schreiben Operationen in absehbarer Zeit terminieren.

1.3 Funktionale Anforderungen

Der nachfolgende Abschnitt enthält alle funktionalen Anforderungen, die beim Entwickeln des graphbasierten Speicher von Bedeutung sind. Dabei werden sämtliche funktionalen Aspekte mit mehreren Anwendungsfällen abgedeckt und modelliert. Hierbei handelt es sich um unterschiedliche Funktionen und Aufgaben, die der gaphbasierte Speicher erfüllen muss, um den Anforderungen gerecht zu werden.

Als primärer Anwendungsfall zählt das Verwalten von Wikipedia-Seiten. Dazu gehört, dass Nutzer in der Lage sind WOM-Ressourcen abzuspeichern. Des Weiteren müssen diese Ressourcen von der Datenbank bereitgestellt werden, das heißt Nutzer können per Anfrage auf die Seite zugreifen und erhalten ein WOM-Ressource zurück. Wichtig hierbei ist die Unterscheidung zwischen Revision und Seite. Der Anwender muss immer eine Revision zusätzlich zur eigentlichen Seite angeben um eine eindeutige Revisionshistorie zu ermöglichen. Die Revision wird bei späteren Anfragen benötigt, um die Seite in der gewünschten Version zu erhalten. Abbildung 1.2 zeigt den Anwendungsfall, um eine WOM-Ressource aus der Datenbank anzufordern. Die darauffolgende Abbildung 1.3 beschreibt den umgekehrten Fall, wenn eine Ressource in die Datenbank geladen wird. Weitere Anwendungsfälle auf diesem Gebiet müssen nicht berücksichtigt werden, weil ausschließlich Einfüge- und Leseoperationen im graphbasierten Speicher auftreten.

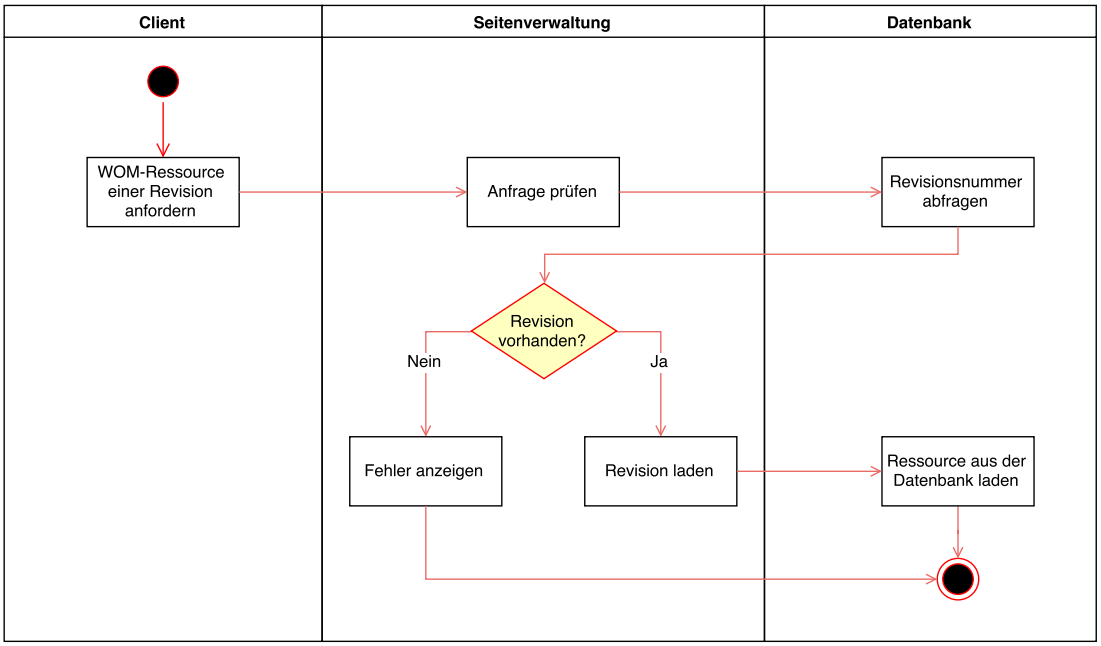

Abbildung 1.2: Anfordern einer WOM-Ressource aus der Datenbank.

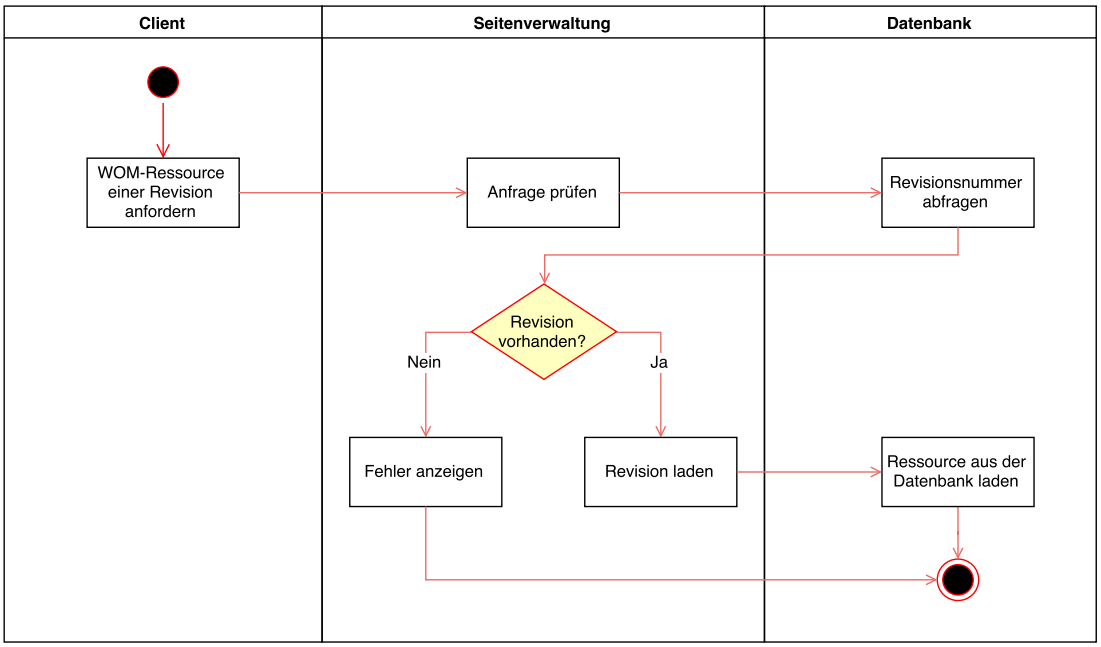

Abbildung 1.3: Laden einer WOM-Ressource in die Datenbank.

Ein weiterer Anwendungsfall betrifft das Traversieren oder Durchlaufen einer WOM-Ressource, ohne dabei den vollständigen Graphen aus der Datenbank zu laden. Über einen sogenannten Walker kann der Nutzer die Ressource traversieren und jeden einzelnen WOM-Knoten abrufen. Dabei wird das Traversieren auf eine Ressource einer bestimmten Revision beschränkt, mit ausschließlich lesenden Operationen. Eine erweiterte Version des Walkers erlaubt zusätzlich den Wechsel von Revisionsbereichen und deckt einen weiteren Anwendungsfall ab. Bisher wurden Revisionen lediglich als baumartige Strukturen betrachtet, die untereinander keiner Verbindungen besitzen. Die Vorstellung trifft nur bedingt zu, weil Artikel in der Wikipedia untereinander sehr wohl Verbindung besitzen; beispielsweise über Weiterleitungen oder Links. Um diese Verbindungen auf Graphenebene zu modellieren, können Nutzer eigenständige Kanten registrieren und verwenden. Solche nutzerspezifischen Kanten können zwischen zwei Knoten unterschiedlicher Artikel oder Revisionen liegen und dadurch einen Wechsel des Revisionsbereichs beim Traversieren erzeugen. Die einfache Version des Walkers verbietet das Durchlaufen von nutzerspezifischen Kanten und verbleibt deshalb stets in einem vordefinierten Revisionsbereich. Um dennoch das Traversieren solcher Kanten zu ermöglichen, wird eine erweiterte Version des Walkers angeboten, mit der Funktion solche Kanten zu durchlaufen. Der erweiterte Walker enthält zusätzlich alle funktionalen Aspekte des Vorgängers und schränkt dessen Nutzbarkeit in keinster Weise ein.

Der letzte Anwendungsfall betrifft das Registrieren und Verwenden von nutzerspezifischen Kanten. Wie bereits erwähnt, können Nutzer eigene Kantentypen registrieren und verwenden, um beispielsweise Weiterleitungen, Links oder Ähnliches zu modellieren. Für die Verwendung solcher Kanten müssen spezielle Maßnahmen vorgenommen werden, weil aus den einzelnen WOM-Ressourcen zusammenhängende Graphen entstehen können. Prinzipiell kann jeder Knoten als potenzieller Verbindungspartner in Frage kommen, wodurch Strukturen aller Art entstehen können. Um beim Traversieren Schleifen oder Ähnliches zu vermeiden, müssen solche nutzerspezifischen Kanten beim Anfordern der Ressource ignoriert werden. Ausschließlich unter Verwendung des erweiterten Walkers können solche Kanten angelaufen und genutzt werden, weshalb diese Verbindungen oder Kanten vollständig transparent zur restlichen Anwendung sind. Somit hat deren Verwendung keine Auswirkungen auf die restlichen Funktionen des Graphspeichers. Als Vorbedingung müssen jegliche nutzerspezifischen Kanten über ein Register angemeldet werden. Der Registrierungsvorgang soll sicherstellen, dass nur bekannte Kantentypen verwendet werden und nicht wahrlos irgendwelche unbekannten Typen im Graphspeicher liegen. Der in Abbildung 1.4 gezeigte Vorgang beschreibt das Registrieren von nutzerspezifischen Kantentypen. Die entsprechende Kante kann anschließend mit dem registrierten Kantentyp erzeugt werden.

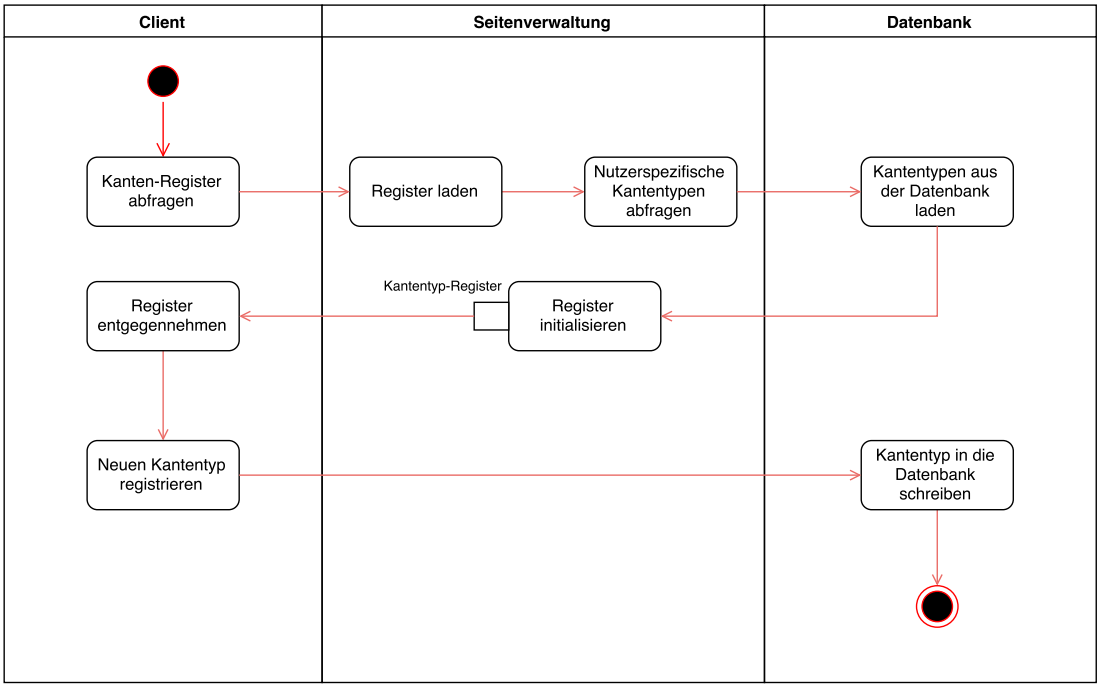

Abbildung 1.4: Registrieren einer nutzerspezifischen Kante.

2 Architektur und Design

Das folgende Kapitel beschreibt Architektur und Design des graphbasierten Speichers für Wikipedia Artikel. Sämtliche Design- und Architekturentscheidungen werden im Verlauf dieses Kapitels erläutert, weshalb Bezug auf die bereits genannten Anforderungen genommen wird. Zu Beginn wird das darunterliegende Datenbanksystem analysiert und ausgewählt, wobei die beiden Hauptkomponenten Titan und Apache Cassandra vorgestellt werden. Anschließend wird Design und Prinzip der Seitenverwaltung ausführlich erläutert, welche als Hauptanforderung an das System betrachtet wird. Der darauffolgende Abschnitt behandelt die Speicheroptimierungen im Bezug auf Revisionen und Redundanzen innerhalb der WOM-Ressourcen. Hier wird ein Algorithmus vorgestellt, der es ermöglicht, die offensichtlichen Redundanzen zweier benachbarter Revisionen zu dezimieren, um im Gegenzug Speicherplatz einzusparen. Des Weiteren wird das Traversieren und Durchlaufen der Ressourcen behandelt, wozu unter anderem der Walker zählt. Zuletzt werden nutzerspezifische Kanten betrachtet, sowie die Registrierung und Verwendung solcher Verbindungen.

2.1 Datenbanksystem

Der Graphspeicher für Wikipedia Artikel basiert auf einem herkömmlichen Datenbanksystem, welches sämtliche Aufgaben im Bereich Datenerhaltung und Persistenz übernimmt. Letztendlich handelt es sich um einen vollständigen Graphen, der als Endresultat in Datenbanksystem verwaltet werden muss. Aus diesem Grund liegt es nahe, eine Datenbank zu verwenden, welche bereits einen Graphen als Grundlage verwendet und sämtliche in Kapitel 2 genannten Anforderungen erfüllen kann. Der aktuelle Stand im Bereich Graphdatenbanken und NoSQL-Datenbanken zeigt einige Möglichkeiten auf, mit denen eine Realisierung des Graphspeichers möglich wäre. Als potentielle Kandidaten kommen Titan[1],

[1] http://titan.thinkaurelius.com

OrientDB[2] oder ArangoDB[3] in Frage. Die Entscheidung basiert auf Lizenzen und datenbanktechnischen Limitierungen, welche den Anforderungen widersprechen. Die drei genannten Datenbanksysteme sind somit die einzigen Graphdatenbanken ohne sichtbare Einschränkungen im Bezug auf alle Anforderungen. Zu ArangoDB konnten zum Zeitpunkt der Auswahl keine Informationen über Limitierungen oder Einschränken gefunden werden. Zudem lagen keine weiteren Tests oder Benchmarks vor, weshalb der Fokus auf Titan und OrientDB gesetzt wurde. Werden die beiden restlichen Kandidaten über Benchmarks verglichen, zeigt sich eine starke Tendenz. Die Ergebnisse von Beis, Papadopoulos, und Kompatsiaris (2015) sprechen deutlich für die Verwendung von Titan als Datenbanksystem, weil Einfügeoperationen deutlich schneller durchlaufen als bei OrientDB. Vor allem bei einer großen Anzahl von Knoten hat Titan deutlich besser abgeschnitten. Übertragen auf eine WOM-Ressoure ist mit einer Anzahl von mehreren tausend Knoten zu rechnen. Aus diesem Grund bietet Titan die bessere Alternative. Allerdings deckt kein einziger Test die Menge an Knoten und Kanten ab, welche die Graphdatenbank bei einer vollständigen Persistierung aller Wikipedia-Seiten hätte. Als Anhaltspunkt werden die offiziellen Limitierungen der Datenbank betrachtet, welche im Fall von Titan bei 2^{59} möglichen Knoten mit 2^{60} Kanten liegen. Bei einer Gesamtmenge von mehreren Billionen WOM-Knoten, verteilt auf alle Wikipedia-Seiten, ist eine Realisierung problemlos möglich. Die wohl populärste und laut vielen Testergebnissen schnellste Graphdatenbank Neo4J[4] musste auf Grund der starken Limitierungen ausgeschlossen werden. Neo4J unterstützt zum Zeitpunkt der Datenbankauswahl lediglich eine maximale Anzahl von 2^35 Knoten. Die theoretische Abbildung aller Wikipedia-Seiten mit mehren Billionen Knoten wäre demnach nicht möglich. Abhilfe könnte ein Partitionieren der Knoten nach Chairunnanda, Forsyth, und Daudjee (2012) schaffen, allerdings übersteigt diese Lösung den Kontext der Arbeit. Aus diesem Grund wird im weiteren Verlauf Titan als primäre Graphdatenbank verwendet.

2.1.1 Titan

Titan ist eine Graphdatenbank von DataStax und wurde ursprünglich von Aurelius LLC entwickelt. Titan wurde unter der Apache 2.0 Lizenz veröffentlicht und zählt zu den Open Source Softwareprodukten. Eine der Primärfuntionen von Titan ist die Verwaltung riesiger Graphen mit mehreren Milliarden Knoten und Kanten. Dabei wurde Titan im Hinblick auf Skalierbarkeit optimiert und kann über ein Cluster auf mehrere Maschinen verteilt werden. Zusätzlich ermöglicht Titan die Ausführung komplexer Traversierungen im Stil einer transaktionalen Datenbank. Die Anzahl der Rechner im Cluster bestimmt die Skalierung des

[2]http://orientdb.com
[3]https://www.arangodb.com
[4]https://neo4j.com

Graphen und der transaktionalen Fähigkeiten. Im Gegensatz zu Konkurrenzprodukten ermöglicht Titan die Verwendung verschiedener Backends als eigentlichen Datenspeicher. Je nach Anwendung können unterschiedliche Datenbanken wie beispielsweise Apache Cassandra[5], HBase[6] oder BerkleyDB[7] verwendet werden. Allerdings gewährleisten die genannten Backends unterschiedliche Ziele des von Brewer (2000) vorgestellten CAP-Theorems, weshalb deren Auswahl essenziell für eine erfolgreiche Umsetzung ist. Das von Apache entwickelte Cassandra hat sich hier als beste Wahl herausgestellt und bietet sowohl Partitionierung als auch Verfügbarkeit. Auch hinsichtlich der Verwendung wird Cassandra unter der liberalen Apache 2.0 Lizenz angeboten, was im Gegensatz zu BerkleyDB kein Ausschlusskriterium darstellt. Zwar werden Verfügbarkeit und Konsistenz in BerkleyDB gewährleistet und laut (Jouili & Vansteenberghe, 2013) werden bessere Ergebnisse beim Laden großer Datenmengen erzielt, allerdings unterliegt der Nutzung eine proprietäre Lizenz. HBase wurde als nicht geeignet eingestuft, weil die Verfügbarkeit nur bedingt gewährleistet werden kann. Angesichts dieser Tatsachen eignet sich Titan als potentielles Datenbanksystem zur Speicherung von Wikipedia-Seiten im WOM-Format.

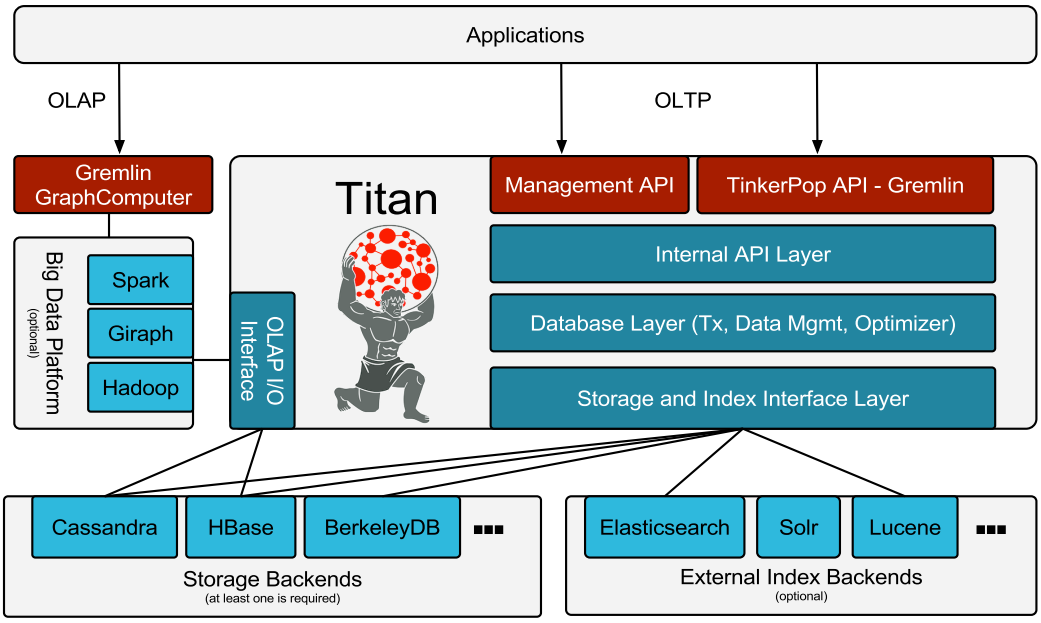

Abbildung 2.1: Abstrakte Architektur von Titan.

[5]http://cassandra.apache.org
[6]https://hbase.apache.org
[7]http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html

2.1.2 Apache Cassandra

Im Jahre 2009 wurde Cassandra von der Apache Software Foundation aufgenommen und zählt zu den sogenannten NoSQL-Datenbanken. Die Verwendung unterliegt der Apache 2.0 Lizenz und stellt somit kein Problem dar. Cassandra bietet hohe Verfügbarkeit und lineare Skalierbarkeit mit gleichzeitiger Fehlertoleranz und Beständigkeit. Um zu verstehen, warum Cassandra die genannten Eigenschaften besitzt und gewährleistet, müssen Architektur und Funktionen der Datenbank näher betrachtet werden. Das grundlegende Konzept bei Cassandra ist das Erzeugen von dezentralisierten Knoten, welche in einem Cluster zusammenarbeiten. Jeder Knoten dieses Clusters ist identisch, weshalb das Problem des Single Point of Failure weitestgehend behoben wird. Es entsteht demnach kein Flaschenhals, wenn bestimmte Konten ausfallen, wodurch eine hohe Verfügbarkeit gegeben ist. Ein Cluster kann theoretisch aus beliebig vielen Knoten bestehen und bietet zudem eine lineare Skalierung. Bekannte Vertreter sind unter anderem Apple[8] mit 27.000 Knoten oder Netflix[9] mit über 2.500 Knoten. Durch automatische Replikation können Fehlertoleranz und Beständigkeit gewährleistet werden, um Datenverlust bestmöglich zu vermeiden. Eine solche Replikation kann auf mehreren Knoten über mehrere Datenzentren geschehen und bietet im Bezug auf die Anforderungen einen ausreichenden Schutz.

2.2 Verwaltung von Wikipedia-Seiten

Die Verwaltung der Wikipedia-Seiten zählt zur Primärfunktion des Graphspeichers und legt die Grundlage für alle weiteren Funktionalitäten. Hauptaufgabe der Seitenverwaltung besteht im Speichern und Bereitstellen von Wikipedia-Seiten. Dazu gehört das Verwalten und effiziente Abspeichern von Seiten-Revisionen sowie das Abbilden von WOM-Ressourcen auf die interne Graphdatenbank. Zusätzlich müssen Maßnahmen zur Speicheroptimierung getroffen werden, damit eine effektive Integration aller Wikipedia-Seiten in den Graphspeicher möglich wird. Im Folgenden werden die Grundkonzepte zu Architektur und Design der Seitenverwaltung näher erläutert. Besonders relevant sind dabei die unterschiedlichen Knoten- und Kantentypen sowie der Algorithmus zur Dezimierung von Redundanzen zwischen Ressourcen benachbarter Revisionen.

[8] https://www.apple.com
[9] https://www.netflix.com

2.2.1 Knotentypen

Jede Wikipedia-Seite wird als WOM-Ressource im Graphspeicher verwaltet und besteht aus einer Menge von Knoten. Diese WOM-Knoten werden letztendlich als sogenannte Vertices in der Graphdatenbank abgebildet und können einer eindeutigen WOM-Ressource zugeordnet werden. Im weiteren Verlauf wird eine strikte Unterscheidung zwischen WOM-Knoten und Vertex gemacht. Dabei richtet sich der Begriff Vertex an den persistenten Knoten in der Graphdatenbank. Zusätzlich zur eigentlichen Ressource werden weitere Daten in der Graphdatenbank verwaltet. Dazu gehören beispielsweise alle Revisions- und Seiteninformationen. Ohne Vorhandensein dieser Daten wäre eine eindeutige Zuordnung der WOM-Ressource zur betreffenden Seite und Revision nicht möglich. Des Weiteren werden interne Daten über nutzerspezifische Kanten oder ähnliches in der Graphdatenbank abgelegt, wodurch eine Vielzahl unterschiedlicher Arten von Vertices entsteht. Aus diesem Grund werden sogenannte Knotentypen innerhalb der Graphdatenbank eingeführt, welche sich in Anzahl und Art der vorhandenen Vertex-Properties unterscheiden. Im Laufe der Arbeit werden immer wieder verschiedene Knotentypen mit unterschiedlichen Aufgaben vorgestellt. Um einen groben Überblick zu bekommen werden im Anschluss alle vorhandenen Knotentypen der Graphdatenbank aufgelistet.

Resource Nodes repräsentieren einen beliebigen WOM-Knoten innerhalb einer WOM-Ressource. Die Menge aller verbundenen *Resource Node* bilden den WOM-Baum der ursprünglichen Ressource. Später können durch nutzerspezifische Kanten die WOM-Bäume gegenseitig verbunden werden, weshalb die Aussage nur bedingt zutrifft. Je nach Abbildungsverfahren werden unterschiedliche Properties im Vertex verwendet.

Page Nodes repräsentieren eine Wikipedia-Seite und kann als Wurzelknoten aller Revisionen einer Seite betrachtet werden. Mit Hilfe des *Page Node* werden sämtliche Revisionen und alle korrespondierenden Ressourcen nach der zugehörigen Seite gegliedert. Dieser Knotentyp besitzt die Seitenidentifikationsnummer als Vertex-Property.

Revision Nodes repräsentieren eine Seiten-Revision und ist in jedem Fall mit einem *Page Node* verbunden. Zusätzlich verweist der *Revision Node* auf ein *Resource Node*, der die Wurzel einer WOM-Ressource darstellt. *Revision Nodes* bildet somit das Bindeglied zwischen der eigentlichen Ressource und der dazugehörigen Seite.

2.2.2 Kantentypen

Die Vertices in der Graphdatenbank werden über Kanten miteinander verbunden und bilden dadurch den endgültigen Graphen. Um eine klare Struktur zu gewährleisten, werden verschiedene Kantentypen mit unterschiedliche Rollen eingeführt. Als Paradebeispiel eignet sich die baumartige Struktur der WOM-Ressource mit Eltern-Kind-Beziehungen zwischen den einzelnen Knoten. Diese Art von Beziehung wird über einen entsprechenden Kantentyp realisiert und in der Graphdatenbank umgesetzt. Weitere Kantentypen finden sich beispielsweise zwischen Seiten- und Revisionsknoten oder in nutzerspezifischen Kanten. Ein weiterer Grund, warum Kantentypen notwendig sind, ist die eindeutige Identifizierung von Revisionsbereichen, welche in Abschnitt 2.2.4 näher behandelt wird. Im weiteren Verlauf werden immer wieder neue Kantentypen vorgestellt. Ungeachtet dessen werden im Anschluss sämtliche festen Kantentypen des Graphspeichers aufgelistet.

Child Edges nehmen die Rolle einer Eltern-Kind-Beziehung ein. Der ausgehende Knoten wird als Eltern-Knoten angesehen, welcher mittels der *Child Edge* auf einen Kind-Knoten zeigt. Dieser Kantentyp wird ausschließlich innerhalb einer WOM-Ressource verwendet und grenzt dadurch den Bereich einer Ressource ab.

History Edges nehmen die Rolle einer Historien-Beziehung ein und verbindet den ausgehenden Vertex mit dem Teil-Graphen einer älteren Revision. Mit Hilfe der *History Edge* werden Speicheroptimierungen vorgenommen, weil Redundanzen benachbarte Revisionen reduziert werden können.

Revision Edges sind das Bindeglied zwischen Seiten- und Revisionsknoten. Dieser Kantentyp besitzt keine weiteren Funktionen oder Rolle und dient ausschließlich zum Verbinden eines *Revision Node* mit der dazugehörigen Seite.

2.2.3 Abbildung von WOM-Ressourcen

Wie bereits in 1.2.1 erläutert werden sämtliche Wikipedia-Seiten im WOM-Format repräsentiert. WOM selbst kann als herkömmliche Baumstruktur mit Knoten und Eltern-Kind-Beziehungen betrachtet werden. Um diese Struktur in einer Graphdatenbank abzubilden, müssen Knoten des Baumes als Vertices und Eltern-Kind-Beziehungen als Kanten in der Graphdatenbank betrachtet werden. Die Herausforderung liegt im Abbilden der Knoten auf einzelne Vertices und kann durch unterschiedliche Ansätze gelöst werden. Das Ziel ist, alle Attribute und Eigenschaften eines WOM-Knotens in die Properties eines Vertex zu legen, ohne dabei Informationen zu verlieren. Der ursprüngliche WOM-Knoten muss anhand die-

ser Informationen vollständig rekonstruierbar sein. Nur unter Einhaltung dieser Vorgaben ist eine vollständige Abbildung der WOM-Ressource möglich.

Als einfachste Variante kann der vollständiger WOM-Knoten in ein einziges Vertex-Property abgelegt werden. Dabei wird der WOM-Knoten in ein geeignetes Format serialisiert und anschließend als Property-Value abgespeichert. Die Effektivität und Wirtschaftlichkeit diese Methode hängt stark vom Serialisierungsvorgang ab und kann durch geeignete Auswahl stark beeinflusst werden. Beispielsweise erreichen Binärformate bessere Kompression und nehmen dadurch deutlich weniger Speicherplatz ein als Semi-Formale Notationen wie JavaScript Object Notation (JSON) oder Extensible Markup Language (XML). Allerdings können JSON- oder XML-Dokumente deutlich schneller geparst werden, was im Hinblick auf Performanz ein Vorteil ist. Nichtsdestotrotz muss bei jeder Anfrage der vollständige WOM-Knoten serialisiert oder deserialisiert werden. Dieses Problem zeigt sich deutlich beim Filtern von Knoten. Da jedes Knoten-Attribut ausschließlich im serialisierten Zustand vorliegt müssen sämtliche Knoten vorab deserialisert werden, um eine anschließende Filterung durchführen zu können. Dieser aufwändige Vorgang bringt nicht nur zeitliche Einbußen, sondern benötigt zudem mehr Hauptspeicher als theoretisch erforderlich.

Eine intuitive Variante ist das Abbilden der einzelnen Knoten-Attribute und auf mehrere Vertex-Properties. Dabei werden alle Eigenschaften und Attribute eines WOM-Knotens als Property-Value in jeweils ein Vertex-Property geschrieben. Die eindeutige Wiederherstellung des Knotens muss gewährleistet werden, ohne dabei Informationen der ursprünglichen WOM-Ressource zu verlieren. Aus diesem Grund muss bei der Realisierung eine Auswahl getroffen werden, welche Attribute zur Wiederherstellung nötig sind. Ein Vorteil dieser Strategie ist, dass der Serialisierungsprozess gespart wird und Attribute direkt über Vertices abgefragt werden können.

Innerhalb des WOM-Formats werden Kanten als implizit vorausgesetzt, weil jeder WOM-Knoten sogenannte Kind-Knoten besitzen kann. Diese Kind-Knoten sind indirekt mit den Eltern-Knoten verbunden und besitzen eine gegenseitige Referenzierung. In der Graphdatenbank müssen solche Referenzen über Kanten gelöst werden. Als neuer Kantentyp wird die sogenannte *Child Edge* etabliert, welche die Rolle einer Eltern-Kind-Beziehung einnimmt. Dadurch können sämtliche WOM-Knoten verbunden werden, um die ursprüngliche Baumstruktur innerhalb der Graphdatenbank abzubilden. Sämtliche *Child Edges* sind gerichtete Kanten, weshalb ein eingehender und ausgehender Vertex existiert. Der ausgehende Vertex nimmt die Rolle des Eltern-Knoten an, sein Partner die Rolle des Kind-Knotens. Dadurch ist eine klare Rollenverteilung zwischen den beiden Knoten einer *Child Edge* festgelegt. Beim späteren Traversieren des Graphen sind diese Rollen wichtig, weil zwischen Eltern- und Kind-Knoten unterschieden werden muss.

Die genannten Abbildungsvarianten in Kombination mit *Child Edges* legen die Grundlage für ein erfolgreiches Abspeichern der WOM-Ressource im Graphspeicher. Alle nötigen Informationen der WOM-Knoten und deren Anordnung im WOM-Baum können durch das genannte Verfahren in der Graphdatenbank persistiert und anschließend abgefragt werden. Die Abbildung der WOM-Ressourcen auf die interne Graphdatenbank ist somit gewährleistet. Der Beispielgraph aus Abbildung 2.2 zeigt eine Seite mit zwei Revisionen. Unterhalb der *Revision Nodes* beginnen die eigentlichen WOM-Ressourcen, die ausschließlich aus *Resource Nodes* bestehen.

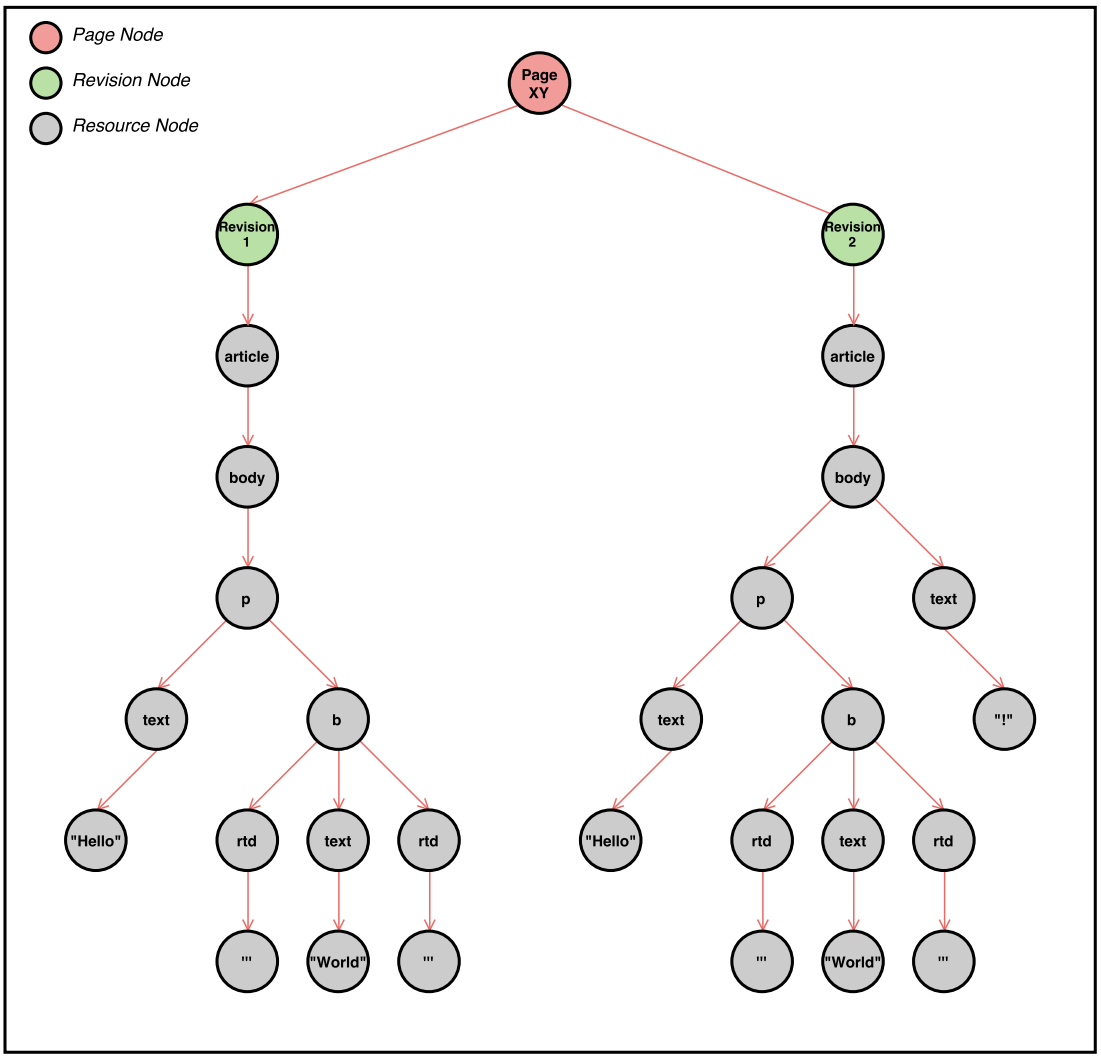

Abbildung 2.2: Beispielgraph einer Seite mit zwei Revisionen.

2.2.4 Traversierung von WOM-Ressourcen

Die bisherigen Funktionalitäten der Seitenverwaltung decken lediglich das vollständige Abspeichern und Abfragen von WOM-Ressourcen ab. Zusätzlich dazu sollen WOM-Ressourcen traversiert werden können, ohne dabei die komplette Ressource aus der Graphdatenbank zu laden. Diese Traversierung wird mit Hilfe eines sogenannten Tree-Walkers umgesetzt, der jeden Knoten einer bestimmten WOM-Ressource anlaufen kann. Der Tree-Walker startet bei einem beliebigen Knoten der WOM-Ressource und kann von dieser Position aus alle nächstgelegenen Knoten abfragen. Da der WOM-Ressource eine Baumstruktur zugrunde liegt, zählen dazu sämtliche Geschwister-, Kind- und Eltern-Knoten. Ein wichtiger Punkt, der im weiteren Verlauf an Bedeutung gewinnt, ist der Bereich einer WOM-Ressource innerhalb des Graphen der Seitenverwaltung. Dieser Bereich wird fortan als Revisionsbereich bezeichnet, weil dadurch die WOM-Ressource einer Revision innerhalb der Datenbank abgegrenzt wird. Wird ein Tree-Walker erzeugt, ist dieser ausschließlich für den Bereich einer bestimmten WOM-Ressource mit fester Seite und Revision bestimmt. Dieser hypothetische Bereich kann nicht durch den Tree-Walker verlassen werden und beschränkt das Traversieren auf eine einzige WOM-Ressource. Bisher wurden ausschließlich *Child Edges* verwendet, wodurch ein Bereichswechsel ohnehin unmöglich ist, weil keinerlei Verbindungen zwischen unterschiedlichen WOM-Ressourcen bestehen, mit Ausnahme der Verbindung über den Revisions- oder Seitenknoten. In den nachfolgenden Abschnitten werden weitere Kantentypen vorgestellt, mit denen eine solche Verbindung möglich wäre. Aus diesem Grund muss die Vorstellung eines Revisionsbereichs eingeführt werden, weil sonst Probleme beim Traversieren einer WOM-Ressource entstehen können.

Ein wichtiger Punkt des Tree-Walkers sind die eigentlichen Knoten, die als Ergebnis an den Nutzer zurückgegeben werden. Laut der W3C DOM Spezifikation[10] enthält jeder Knoten grundlegende Funktionalitäten wie das Abrufen von Geschwister-, Kind- oder Elternknoten. Der Tree-Walker hingegen holt lediglich eine flache Kopie des Knotens aus der Graphdatenbank ohne Kind- oder Eltern-Knoten. Beim Aufruf der Knotenfunktionen würden keine weiteren Knoten gefunden werden, weil diese noch in der Graphdatenbank als Vertices liegen. Deshalb werden Proxy-Knoten als Ergebnis zurückgeben, mit der Aufgabe, entsprechende Funktionen auf die Graphdatenbank umzulenken. Dadurch kann der Nutzer problemlos mit dem Knoten arbeiten, ohne dabei weitere Kenntnisse über den internen Graphspeicher oder ähnliches besitzen zu müssen.

[10] http://www.w3.org/DOM

2.2.5 Graphoptimierung benachbarter Revisionen

Ein großes Problem der Seitenverwaltung stellt die umfangreiche Seitenhistorie dar. Jede Wikipedia-Seite besteht in der Regel aus mehreren Revisionen, wobei die Menge aller Revisionen als Seitenhistorie bezeichnet werden kann. Jede Revision basiert auf einer Vorgänger-Revision und kann selbst als Nachfolger bezeichnet werden. Ausnahme stellt die initiale Revision bei Erzeugung einer neuen Seite dar, weil diese keinen Vorgänger besitzt. Wird eine bereits vorhandene Seite editiert und anschließend abgespeichert, entsteht eine neuen Revision dieser Seite. Die Problematik liegt hier in Art und Umfang der Änderung, da oft nur kleine Editierungen von Revision zu Revision vorgenommen werden. Dieser Umstand sorgt dafür, dass der Unterschied zwischen Vorgänger- und Nachfolger-Revision sich lediglich auf kleine Teile des Baumes beschränken kann. Würden demnach beide WOM-Ressourcen in der Graphdatenbank liegen, wäre ein Großteil der beiden Graphen redundant. Aus diesem Grund wird eine Strategie vorgestellt, mit deren Hilfe solche Redundanzen dezimiert werden können. Ziel des Algorithmus ist die Reduzierung von Redundanzen benachbarter Revisionen um Speicherplatz und Vertices einzusparen. Als benachbarte Revisionen werden aufeinanderfolgende Vorgänger- und Nachfolger-Revisionen definiert, wodurch der Algorithmus in seiner Funktion eingegrenzt wird. Es werden ausschließlich redundante Teil-Graphen zwischen dem direkten Vorgänger und Nachfolger beseitigt ohne dabei auf ältere Revisionen Bezug zu nehmen.

Die Grundidee der Strategie liegt im Verbinden des Nachfolger-Graphen mit Vertices aus der Vorgänger-Revision. Dabei wird der redundante Teil des Nachfolger-Graphen ignoriert und nicht in die Graphdatenbank geschrieben. Stattdessen werden neue Verbindungen zu bereits existierenden Teil-Graphen gelegt, die in der Vorgänger-Revision liegen. Abbildung 2.3 zeigt die Graphen einer Vorgänger- und Nachfolger-Revision. Der mit rot markierte und eingekreiste Bereich stellt den redundanten Teil-Graphen dar, der in beiden Revisionen vorhanden. Lediglich der blau markierte Bereich der zweiten Revision wurde neu eingefügt. Demnach besitzen beide Graphen in diesem Zustand redundante Daten. Um diese Redundanzen zu entfernen, müssten sämtliche Knoten der Vorgänger-Revision 1 von dem Graphen der Revision 2 referenziert werden. In diesem Fall würde Revision 2 ausschließlich die beiden neuen Knoten erzeugen und alle weiteren Knoten einsparen. In der Praxis ist diese Umsetzung nicht möglich, weil dadurch die Baumstruktur des Graphen der Nachfolger-Revision zerstört wird und eine eindeutige Traversierung der Ressource nicht mehr möglich ist. Als Lösung werden deshalb nur Teile des Graphen aus der Vorgänger-Revision übernommen. Dabei wird darauf geachtet, dass die vollständige Baumstruktur des Nachfolgers erhalten bleibt und keine Probleme beim Traversieren entstehen.

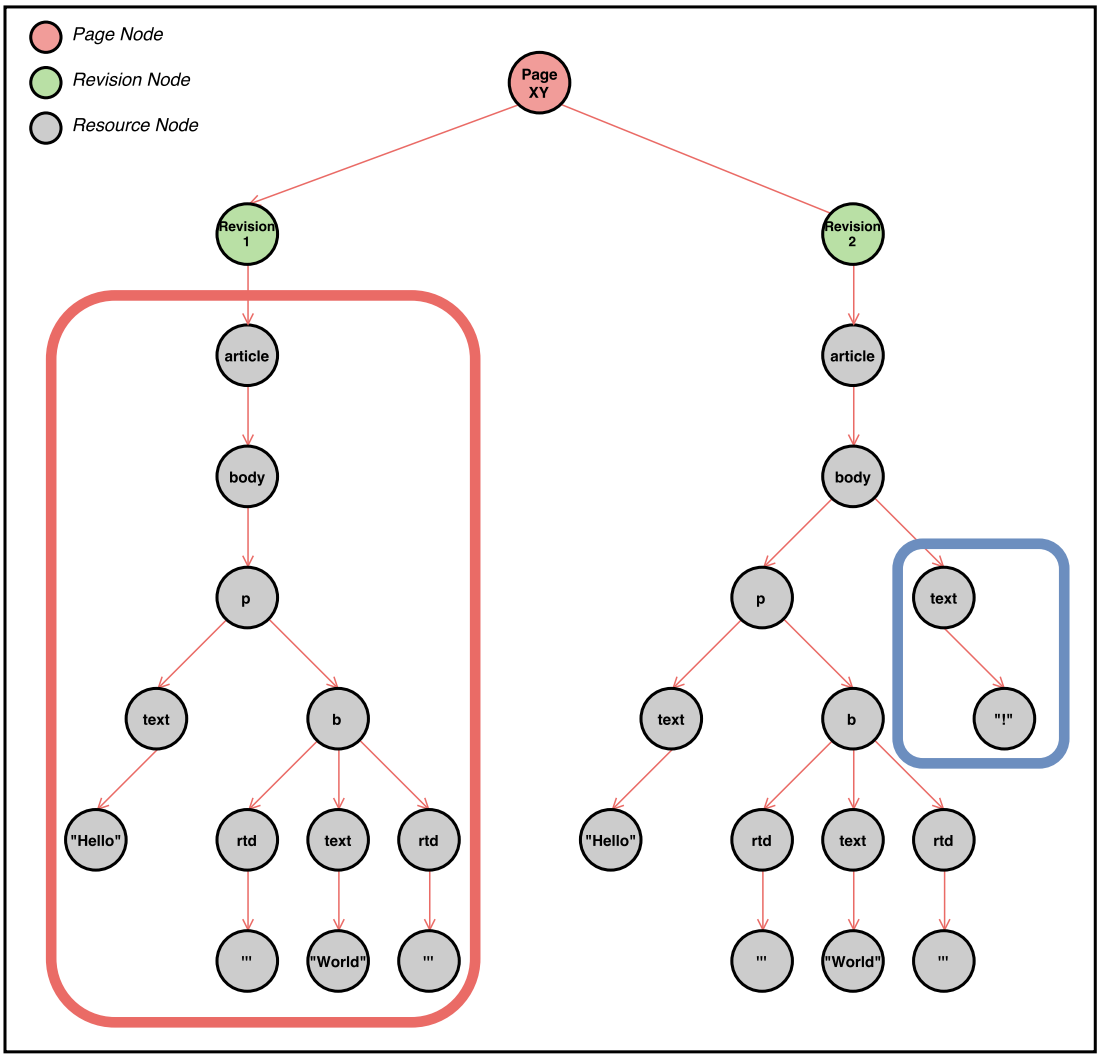

Abbildung 2.3: Redundanter Graph zweier Revisionen.

Bei dem Graphen aus Abbildung 2.4 zeigt sich eine deutliche Veränderung der Nachfolger-Revision. Der redundante Teil-Graph wurde fast vollständig entfernt und mit Hilfe der blauen *History Edge* aus dem Vorgänger-Graphen referenziert. Die verbleibenden Redundanzen beziehen sich auf den rot markierten Bereich. Diese Knoten können nicht aus der Vorgänger-Revision referenziert werden, weil sonst keine eindeutige Baumstruktur gegeben ist. Allgemein können keine Knoten, deren Kind-Knoten von Änderungen betroffen sind, durch *History Edges* referenziert werden. Dadurch können redundante Knoten innerhalb des Baums bestehen bleiben. Die Auswirkungen dieser Redundanzen sind jedoch nur geringfügig, weil alle textuellen Inhalte an den Blatt-Knoten einer WOM-Ressource hängen.

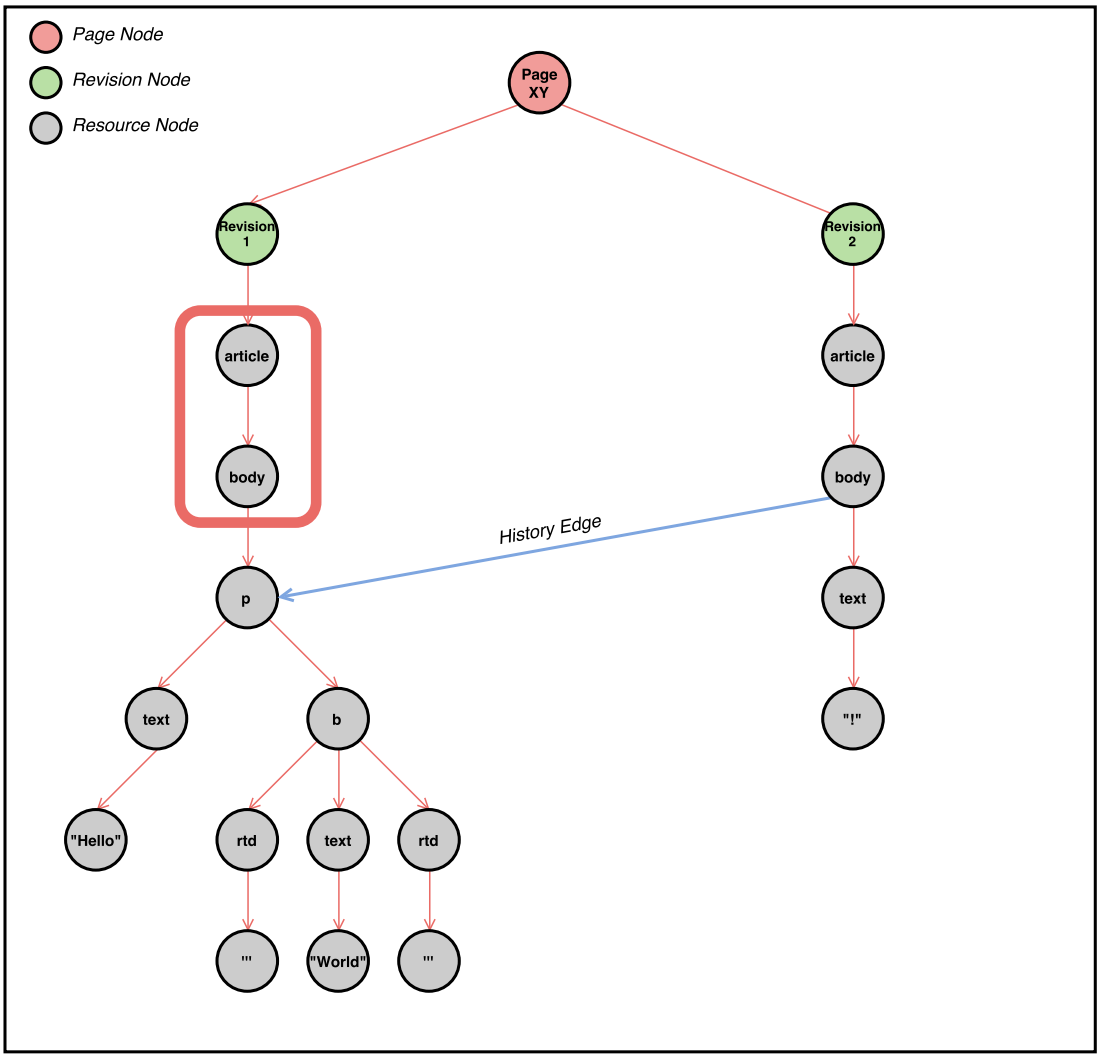

Abbildung 2.4: Optimierter Graph minimaler Anzahl an Redundanzen.

Als Voraussetzung zur Anwendung des Algorithmus wird ein Verfahren benötigt, das die Unterschiede zweier WOM-Ressourcen analysiert und redundante Teil-Graphen aufdeckt. Für diese Aufgabe wurde der HDDiff-Algorithmus von Dohrn und Riehle (2014) entwickelt und publiziert. Mit Hilfe des HDDiff können zwei unterschiedliche WOM-Ressourcen miteinander verglichen werden. Als Ergebnis werden alle Operationen aufgelistet, die zur Transformation der ersten in die zweite WOM-Ressource nötig sind. Diese Änderungsoperationen liefern eine Aussage darüber, welche Teilbäume der ursprünglichen Revision von Änderungen unberührt bleiben. Diese Abschnitte können als redundant betrachtet und ohne Weiteres von der Nachfolger-Revision übernommen werden. An dieser Stelle kommt der vorgestellte Algorithmus zur Anwendung und Redundanzen werden über das Setzen von *History Edges* weitestgehend entfernt.

2.2.6 Begrenzung der Baumtiefe

Selbst durch die Reduzierung von Redundanzen und das Optimieren des Graphen entstehen Unmengen von Vertices. Beim Abfragen einer Ressource könnten hier potentielle Probleme entstehen weil jeder Vertex aus der Datenbank geladen werden muss. Bei einer Baumstruktur bedeutet dass erheblichen Mehraufwand, weil Traversierungen teuer sind. Um dieser Problematik entgegenzuwirken wird ein weiteres Feature im graphbasierten Speicher etabliert. Der Nutzer hat die Möglichkeit bestimmte Knoten zu limitieren. Dafür werden die entsprechenden Typen oder Knotennamen angegeben an denen limitiert werden soll. Wird eine neue Ressource in den Graphspeicher geladen müssen diese Knoten einer gesonderten Behandlung unterzogen werden. Dafür wird die Ressource traversiert und wie gewohnt in die Graphdatenbank geschrieben. Stößt der Prozess beim Traversieren auf einen limitierten Knoten wird diese vollständig in eine binäre Repräsentation serialisiert und als Vertexpropertie abgespeichert. Anschließend endet die Traversierung an dieser Stelle ohne dabei die Kind-Knoten der limitierten Eltern-Knotens zu berücksichtigen. Diese sind bereits zusammen mit dem Elternknoten in serialisierter Form abgespeichert worden. Dadurch müssen deutlich weniger Vertices erzeugt werden, weil sämtliche Kind-Knoten ausschließlich im serialisierten Zustand vorliegen. Die Reduktion der Vertexanzahl hängt dabei stark von der Knotenauswahl ab.

Um limitierte Knoten wiederherzustellen müssen beim Leseprozess die entsprechenden Kind-Knoten deserialisiert werden. Dafür muss stets überprüft werden ob es sich bei dem aktuellen Knoten um einen limitierten Knoten handelt. Trifft dieser Fall zu, kann die Traversierung auf Datenbankebene gestoppt und der Knoten mit seinen Kinder deserialisiert werden. Diese Prüfung muss ebenfalls beim Optimieren der Graphen benachbarter Ressourcen berücksichtigt werden. Serialisierte Knoten können nicht durch *History Edges* referenziert werden. Dadurch müssen im schlimmsten Fall Redundanzen geduldet werden, wenn im Gegenzug die Anzahl der Vertices verringert wird.

2.3 Nutzerspezifische Kanten

Eine weitere große Funktionalität des Graphspeichers sind nutzerspezifische Links oder Kanten innerhalb der Datenbank. In der Wikipedia werden Seiten oft durch Weiterleitungen oder Links referenziert und können dadurch an beliebigen Stellen erreicht werden. Beispielsweise enthält der Artikel *Animal* eine Weiterleitung auf den Artikel *Fish*. Um grundsätzlich die Referenzierung von Seiten, Revisionen oder einzelnen Knoten zu ermöglichen, werden sogenannte nutzerspezifische Kanten etabliert. Diese Verbindungen erlauben dem Nutzer, zwischen zwei belie-

bigen Knoten des Graphspeichers eine Kante zu setzten. Die Rolle und Art der Kante wird durch den Nutzer selbst festgelegt und kann frei definiert werden. Die Nutzung einer solchen Kante erfordert die Registrierung des Kantentyps in einem Register und unterliegt keinerlei Einschränkungen. Nur mit Hilfe des Registers kann sichergestellt werden, dass jeder Kantentyp in der Graphdatenbank bekannt ist und keine Kanten wahllos im Graphspeicher liegen.

Werden nutzerspezifsiche Kanten gesetzt, kann die bisherige abgeschlossene Sicht einer WOM-Ressource nicht mehr gewährleistet werden. Bisher wurde die WOM-Ressource als abgeschlossener Graph innerhalb der Graphdatenbank betrachtet, der durch einen *Revision Node* mit dem restlichen Graphen verbunden ist. Diese Sichtweise wird durch die Einführung von nutzerspezifischen Kanten nichtig, weil unterschiedliche WOM-Ressourcen über diese Art von Kanten untereinander verbunden werden können. Somit wird der Bereich einer WOM-Ressource nicht mehr durch die Revision abgegrenzt, sondern kann unter Umständen in andere Ressourcen übergehen. Um dennoch die Bereiche einer WOM-Ressource abzustecken, müssen klare Grenzen definiert werden. Als Basis können die bereits bekannten Kantenypen *Child Edge* und *History Edge* verwendet werden. Durch das Folgen dieser Kantentypen ist sichergestellt, dass der aktuelle Bereich einer beliebigen WOM-Ressource nicht verlassen wird und kein Bereichswechsel vorgenommen werden muss. Die Ausnahme sind nutzerspezifische Kanten, wobei zwischen zwei Fällen unterschieden werden muss.

Eine Variante ist das Verbinden zweier Knoten innerhalb derselben Ressource. Die nutzerspezifische Kante liegt demnach zwischen zwei Vertices, welche einen identischen Revisionsbereich besitzen. In diesem Fall muss beim Folgen der Kante kein Bereichswechsel stattfinden, weil die WOM-Ressource nicht verlassen wird. Der kritische Fall ist das Verbinden zweier Knoten unterschiedlicher WOM-Ressourcen. In diesem Szenario liegt die nutzerspezifische Kante zwischen zwei Vertices mit heterogenen Revisionsbereichen. Das Folgen der Kante würde einen Bereichswechsel bedeuten, weil eine neue WOM-Ressource betreten wird. Um diesen Bereichswechsel zu ermöglichen, muss die Seiten- und Revisionsnummer der Ziel-Ressource durch die nutzerspezifischen Kanten mitgeteilt werden. Demnach müssen alle nutzerspezifischen Kanten mit Bereichswechsel die nötigen Informationen der Ziel-Ressource selbst mitführen.

Unter Beachtung dieser beiden Fälle können nutzerspezifische Links problemlos in den Graphspeicher integriert werden. Da beim bisherigen Traversieren oder Abfragen von WOM-Ressourcen ausschließlich die bereits bekannten Kantentypen berücksichtigt werden, hat die Verwendung von nutzerspezifischen Kanten keine Auswirkungen auf die bisherigen Funktionalitäten. Allerdings müssen Methoden zur Traversierung dieser Kanten geschaffen werden um einen Nutzen zu erzielen.

2.3.1 Traversierung von nutzerspezifischen Kanten

Die bereits etablierten Funktionen aus 2.2.4 müssen durch die Verwendung von nutzerspezifischen Kanten erweitert werden. Der ursprüngliche Tree-Walker dient ausschließlich zur Traversierung einer WOM-Ressource mit fester Seite und Revision. Sämtliche nutzerspezifischen Kanten werden weder berücksichtigt noch wahrgenommen und müssen über einen erweiterten Tree-Walker berücksichtigt werden. Mit Hilfe dieses Graph-Walkers sollen die ursprünglichen Funktionen des Tree-Walkers um das Folgen nutzerspezifischer Kanten erweitert werden. Der Graph-Walker hat die Aufgabe beim folgen einer nutzerspezifischen Kante gegebenenfalls einen Bereichswechsel durchzuführen. Nach erfolgreichem Bereichswechsel befindet sich der Graph-Walker in einer anderen WOM-Ressource und passt dementsprechend alle Funktionen auf die neue Umgebung an. Abbildung 2.5 zeigt einen beispielhaften Bereichswechsel von *Revision 3* nach *Revision 2* über eine nutzerspezifische Kante von *Knoten F* nach *Knoten B*. Die Besonderheit an diesem Fall ist, dass *Knoten B* ebenfalls von *Revision 1* genutzt wird. Ohne weitere Informationen kann nach Traversieren der nutzerspezifischen Kante kein eindeutiger Revisionsbereich festgelegt werden. Ausschließlich durch die Nummer der Revision an der nutzerspezifischen Kante kann eine klare Aussage drüber getroffen werden, zu welchem Graphen verlinkt wurde. Im Beispiel wird *Revision 2* als Ziel angegeben und verhindert dadurch eine Verwechslung der Graphen.

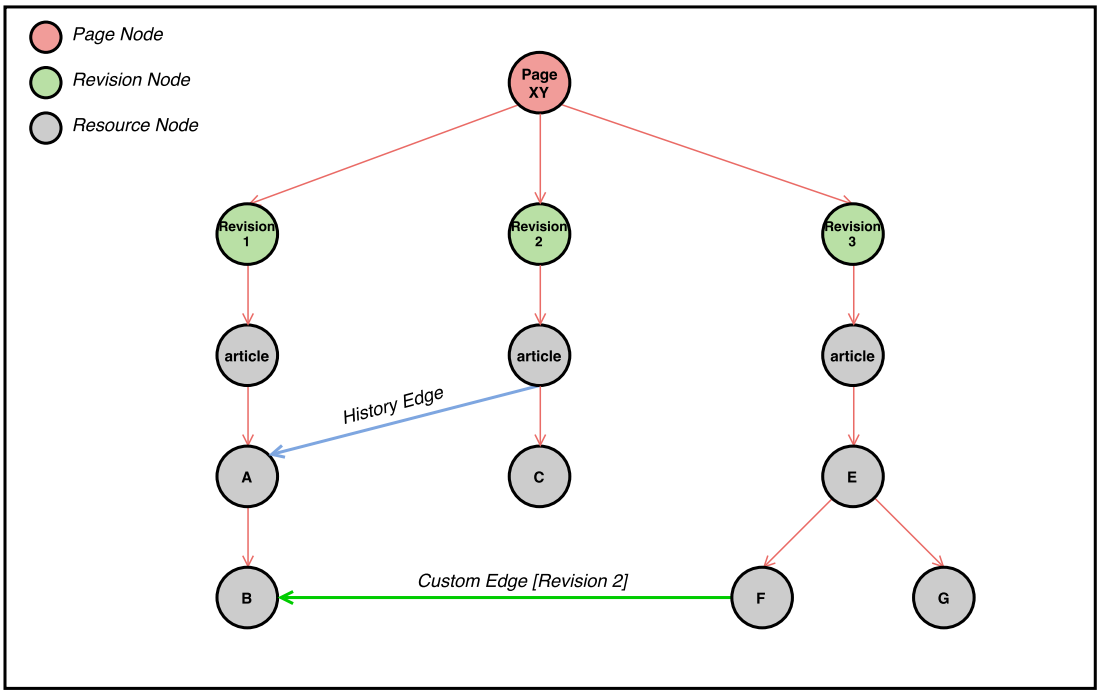

Abbildung 2.5: Anwendungsfall einer nutzerspezifischen Kante.

3 Implementierung

Im folgenden Kapitel werden Implementierung und Umsetzung des Graphspeichers auf Grundlage der Designentscheidungen aus Kapitel 2 behandelt. Es werden verschiedene Technologien vorgestellt, die zur Realisierung benötigt wurden. Dabei sollen konkret die beiden Hauptgebiete Datenbanksystem und Seitenverwaltung mit technischen Aspekten erläutert werden. Hier sind vor allem die Revisionshistorie und der Algorithmus zur Graphoptimierung von Bedeutung. Deren Implementierung kann als Hauptproblem des Graphspeichers betrachtet werden und stellt die größte Herausforderung dar. Allerdings müssen auch Umsetzungsdetails bezüglich der Abbildung von WOM-Ressourcen auf die Graphdatenbank geklärt werden. Zu den weiteren Themen zählt Titan mit Cassandra Backend und TinkerPop[1] als universelle Zugriffsschnittstelle. Zuletzt wird die Realisierung von nutzerspezifischen Kanten behandelt, wobei deren Integration eine wichtige Rolle spielt.

Um einen Eindruck von der Implementierung zu bekommen, werden zunächst Aspekte der Programmierung geklärt. Der Quellcode wurde vollständig in der Programmiersprache Java geschrieben und mit herkömmlichen Entwicklungstools wie Maven[2], Git[3] und IntelliJ IDEA[4] unterstützt. Qualität und Style des Quellcodes halten sich an die Lehren von Martin (2013) und entsprechen den grundlegenden Clean Code Regeln. Angewandte Architekturmuster wurden dem Buch von Gamma, Helm, Johnson, und Vlissides (1994) entnommen. Die nachfolgenden Abschnitte ersetzen keine vollständige Dokumentation und enthalten lediglich essentielle Bestandteile des graphbasierten Speichers. Nebensächliche oder unbedeutende Teile der Implementierung werden teilweise übersprungen und sind nicht Bestandteil dieses Kapitels. Eine syntaktische Korrektheit muss nicht immer zwangsläufig gegeben sein, vielmehr zählen die strukturellen und semantischen Aspekte.

[1] http://tinkerpop.apache.org
[2] https://maven.apache.org
[3] https://git-scm.com
[4] https://www.jetbrains.com/idea/

3.1 Datenbanksystem

Die Design- und Architekturentscheidungen aus 2.1 legen den Grundstein für die Implementierung der Datenbankschicht. Als maßgebende Technologie wurde Titan mit Cassandra als zugehöriges Backend festgelegt. Hintergrund war die Verwendung einer Graphdatenbank, die potentiell alle gestellten Anforderungen erfüllen kann. Zusätzlich zu Titan und Cassandra werden weitere Komponenten vorgestellt, die zu einer erfolgreichen Implementierung beigetragen haben. Hierzu gehört beispielsweise das Apache TinkerPop Framework, welches eine einfache und wiederverwendbare Schnittstelle zu Titan und Graphdatenbanken allgemein anbietet. In den folgenden beiden Abschnitten werden alle wichtigen Details zur Implementierung der Datenbankschicht erläutert. Allerdings handelt es sich dabei ausschließlich um die Graphdatenbank, auf der im graphbasierten Speicher für Wikipedia-Artikel aufsetzt wird. Sämtliche domänenspezifische Einzelheiten werden in den darauffolgenden Kapiteln behandelt.

3.1.1 TinkerPop

TinkerPop wird unter der Apache 2.0 Lizenz von der Apache Software Foundation angeboten. Bei TinkerPop handelt es sich um ein Framework zur Verarbeitung von Graphen, das mittlerweile in seiner dritten Version erhältlich ist. Hersteller von Graphdatenbanken können dieses Framework implementieren um ihr Produkt TinkerPop-fähig zu machen. Dadurch können Nutzer über eine einheitliche Schnittstelle auf verschiedenste Graphdatenbanken zugreifen ohne herstellerspezifische oder proprietäre APIs zu verwenden. Bekannte Vertreter mit TinkerPop-fähigen Graphen sind beispielsweise Neo4J, OrientDB, IBM Graph[5] und viele weitere. Im Fall von Titan werden neben der eigenen Java API ebenfalls alle wichtigen Funktionen über TinkerPop 3 angeboten.

Durch TinkerPop werden datenbankspezifische Implementierungen hinfällig, weil alle Funktionen über ein einziges Framework gelöst werden können. Ein TinkerPop-fähiger Graph kann mit Hilfe der funktionalen Gremlin Sprache durch komplexe und verschachtelte Abfragen traversiert werden. (Rodriguez, 2015) Dabei kann der Nutzer über mehrere sequentielle Schritte atomare Operationen auf dem Datenstrom modellieren. Die Java API bietet einfache Methodenaufrufe und unterstützt verschiedenste Graphtechnologien. Dabei spielt es beispielsweise keine Rolle, ob ein transaktionaler in-memory Graph benötigt wird oder eine verteilte Graphdatenbank auf mehreren Rechnern läuft. TinkerPop bietet die optimale Lösung zur Verarbeitung und Traversierung des Graphen und wird als Grundlage für alle Graphdatenbankanfragen verwendet.

[5] http://www.ibm.com/analytics/us/en/technology/cloud-data-services/graph/

3.1.2 Titan mit Cassandra Backend

Die Verwendung und Integration von Titan besteht zum Großteil aus Konfigurationen. Titan wird als Maven Dependency in das Projekt eingebunden und verwendet eine zentrale Datei zur Konfiguration. Dessen Inhalte definieren beispielsweise das verwendete Backend oder weitere Titan-spezifische Eigenschaften. In Listing 3.1 ist ein Auszug aus der Konfigurationsdatei von Titan zu sehen mit Cassandra als Backend, zugehöriger Host-Adresse und aktivem Datenbankcache. Titan bietet eine Vielzahl von Parametern für unterschiedliche Bereiche der Datenbank an und kann beliebig angepasst werden.

Listing 3.1: Ausschnitt aus Titan-Konfigurationsfile.

```
storage.backend=cassandra
storage.hostname=127.0.0.1:9160
cache.db−cache = true
```

Cassandra wird unabhängig davon auf mehreren Rechnern als Cluster installiert. Dabei müssen lediglich die von Cassandra mitgelieferten Konfigurationsfiles angepasst und konfiguriert werden. Anschließend können die verschiedenen Knoten beginnend mit den Seed-Knoten nacheinander gestartet und im Cluster integriert werden. Die Anzahl und Menge der Knoten hängt von der letztendlichen Last ab und kann beliebig erweitert werden. Die Instanzen von Titan und Cassandra laufen auf unterschiedlichen Rechnern und werden im sogenannten Remote Server Mode betrieben. Dadurch werden Lese- und Schreibzugriffe auf das Cassandra Cluster durch die Titan-Instanz in der Java Virtual Machine (JVM) ermöglicht. Cassandra hingegen verwaltet den eigentlichen Graph mit allen Vertices und Kanten. Diese logische Trennung bietet eine bessere Skalierung als der Local Server Mode mit einer lokalen Cassandra Datenbank im selben Host.

Für sämtliche Funktionen zum Erzeugen, Bearbeiten oder Abfragen des Graphen bietet Titan eine umfangreiche Java API an. Dafür wird ein `TitanGraph` Objekt benötigt, welches über `TitanFactory` geöffnet werden kann. Beim Initialisieren wird das Konfigurationsfile an die Factory übergeben und das Graphobjekt erzeugt. Dem internen Graph liegt ein Schema zugrunde, welches sämtliche Properties und Indices der Graphdatenbank definiert. Dieses Schema muss zusätzlich über das Graphobject erstellt und verwaltet werden. Ist das Graphobjekt geöffnet sind beliebige Operationen in der Graphdatenbank möglich. Jede Operation ist mit einer Transaktion verbunden und muss nach erfolgreicher Durchführung bestätigt werden. Im Fehlerfall kann ein Rollback durchgeführt werden, um den vorherigen Zustand wiederherzustellen. In Listing 3.2 wird eine beispielhafte Transaktion durchgeführt, die zwei Vertices anlegt und im Fehlerfall einen Rollback durchführt. Das Erzeugen von Vertices startet automatisch eine neue Transaktion, die bei Erfolg bestätigt werden muss. `TitanGraph` implementiert

das TinkerPop Interface `Graph` und unterstützt demnach TinkerPop Funktionalitäten. Dies ermöglicht die Verwendung der funktionalen Gremlin Sprache zur Traversierung und Verarbeitung des Graphen. Aus diesem Grund werden bis auf die Titan-spezifische Schema-Erzeugung ausschließlich TinkerPop 3 Methodenaufrufe verwendet.

Listing 3.2: Transaktion zur Erzeugung eines Vertex.

```
try
{
    Vertex vertexA = graph.addVertex("name", "Paul");
    Vertex vertexB = graph.addVertex("name", "Franz");
    graph.tx().commit();
} catch (Exception e)
{
    graph.tx().rollback();
{
```

Für die Initialisierung des Graphobjekts wurde eine Factory nach dem Architekturmuster von Gamma u. a. (1994) implementiert. Zusätzlich existiert eine übergeordnete Factory zur Erzeugung dieser datenbankspezifischen Factories. Dadurch besteht die Möglichkeit, ohne größeren technischen Aufwand mehrere TinkerPop-fähige Datenbanken als potentielles Datenbanksystem zu verwenden. Es genügt, eine Implementierung der entsprechenden Datenbank-Factory und das Einbinden derselben in die übergeordnete `GraphFactory`. Listing 3.3 zeigt die Implementierung dieser Factory mit einer statischen Methode für alle weiteren datenbankspezifischen Factories, die ihrerseits wiederum das eigentliche Graph Object bereitstellen.

Listing 3.3: Übergeordnete Factory für Graph Factories.

```
abstract class GraphFactory
{
    static GraphFactory getGraphFactory(GraphType graphType)
    {
        switch(graphType)
        {
            case Titan: return new TitanGraphFactory();

            case Other: return new OtherGraphFactory();
        }
    }

    abstract Graph getGraph();
}
```

Im Fall von Titan wurde die Klasse `TitanGraphFactory` von `GraphFactory` abgeleitet und implementiert. Über die Funktion `getGraph()` wird das zentrale Graphobjekt nach dem Singleton Architekturmuster von Gamma u. a. (1994) bereitgestellt. Zusätzlich wird ein Titan-spezifisches Graphdatenbankschema erstellt und initialisiert. Das Schema definiert alle Datentypen und Indices von verwendeten Vertex- oder Kanten-Properties. Die Erstellung eines Schemas wird zwingend für Sortierungen auf Datenbankebene benötigt und findet in späteren Kapiteln Verwendung. Der Beispielcode aus Listing 3.6 zeigt die Initialisierung des Graphobjekts und den Aufruf der Schema-Erzeugung. Aus mangelnder Relevanz wird die eigentliche Implementierung des Schemas vernachlässigt und kann in der offiziellen Titan Dokumentation nachgelesen werden[6].

Listing 3.4: TitanGraph Factory als Singleton.

```
class TitanGraphFactory extends GraphFactory
{
    private static Graph titanGraph;

    Graph getGraph()
    {
        if (titanGraph == null)
        {
            titanGraph = TitanFactory.open(properties);
            makeGraphSchema();
        }
        return titanGraph;
    }
}
```

Die gesamte Persistenzschicht besteht demnach aus einer `GraphFactory` mit abgeleiteten Factories für entsprechende Datenbanken. Als primäre Factory wird eine Implementierung der `GraphFactory` für Titan als Graphdatenbank verwendet. Diese `TitanGraphFactory` stellt das benötigte Graphobjekt bereit und kann per TinkerPop 3 für Schreib- und Lesezugriffe verwendet werden. Die eigentliche Implementierung von Zugriffsobjekten wird durch die umfangreiche und intuitive TinkerPop 3 API hinfällig. Das eigentliche Graphobjekt wird als Singleton durch die korrespondierende Factory bereitgestellt um möglichen Overhead durch die Öffnung des Graphen zu vermeiden. Letztendlich bietet die Kombination von Titan/Cassandra und TinkerPop eine übersichtliche und wiederverwendbare Implementierung.

[6] http://s3.thinkaurelius.com/docs/titan/1.0.0/configuration.html

3.2 Seitenverwaltung

Auf Basis des Datenbanksystems kann im weiteren Verlauf die Seiten- und Revisionsverwaltung aufgesetzt werden. Die grundlegende Strategie zur Verwaltung aller Wikipedia-Seiten wurde bereits in Kapitel 3.2 festgelegt und wird im weiteren Verlauf realisiert. Hauptproblem stellt das Abbilden einer Wikipedia-Seite auf die interne Graphdatenbank dar. Beim Abspeichern einer Seite werden sämtliche Knoten als Vertices in die Graphdatenbank geladen. Umgekehrt wird beim Auslesen die ursprüngliche Wikipedia-Seite erzeugt und als Ressource zurückgegeben. Dieser Vorgang erfordert die Implementierung einer zentralen Schnittstelle, die alle Aufgaben zur Persistierung einer Seite übernimmt. Dabei muss eine geeignete Umsetzung der Designvorlagen gewährleistet werden. Als Grundstruktur wird der sogenannte `GraphStorageManager` definiert. Dieser bietet sämtliche Funktionen zur Verwaltung oder Verarbeitung von Wikipedia-Seiten an und dient als zentrale Anlaufstelle. Wikipedia-Seiten werden im WOM-Format an den `GraphStorageManager` übergeben und als solche abgespeichert. Im Gegenzug erhält der Nutzer eine WOM-Ressource als Antwort beim Abfragen einer Wikipedia-Seite. Zusätzlich werden Artikel- und Revisionsnummer der Wikipedia Seite übergeben, um die Ressource im Graph einzuordnen.

Listing 3.5: Interface mit Signaturen der Hauptfunktionen.

```
public interface GraphStorageManager
{
    void wipeGraph();
    void createGraph(long articleId, long revisionId, Node resource);
    Node retrieveGraph(long articleId, long revisionId);
}
```

Das Abspeichern oder Laden von Wikipedia-Seiten ist eine der beiden Hauptaufgaben des `GraphStorageManager`. Hierfür wird eine vollständige Seite als WOM-Ressource mit gültiger Artikel- und Revisionsnummer an den Manager übergeben. Dieser prüft, ob bereits ein Vertex für den ausgewählten Artikel vorliegt und erzeugt diesen gegebenenfalls. Zusätzlich wird ein neuer Vertex für die übergebene Revision erzeugt und mit dem Artikelvertex verbunden. Für diese Art von Verbindung wird ein spezieller Kantentyp verwendet. Solche *Revision Edges* sind ausschließlich zwischen Artikel- und Revisionsvertices zu finden. Anschließend kann die eigentliche WOM-Ressource in die Datenbank geladen werden. Diese entspricht einer Baumstruktur und wird im ersten Schritt vollständig traversiert. Dabei wird jeder einzelne WOM-Knoten abgefragt und als Vertex in die Graphdatenbank geschrieben. In Kapitel 2.2.3 wurden bereits verschiedene Strategien zur Abbildung von WOM-Knoten auf Vertices gegenübergestellt. Als Endresultat wird der vollständige WOM-Knoten serialisiert und als Vertex-Property abgelegt.

Grund für diese Entscheidung ist die deutlich bessere Performanz bei geringerer Anzahl an Properties. Des Weiteren werden sämtliche Eltern-Kind-Beziehungen zweier WOM-Knoten über Kanten gelöst. Dadurch bleibt die interne Baumstruktur der Ressource in der Datenbank erhalten und kann durch Traversieren der Kanten rekonstruiert werden.

Listing 3.6: Schreiben einer Ressource in die Datenbank.

```
void traverseAndCreateGraph(Vertex parentNode, Node resource, int position)
{
    Vertex nextParentNode = VertexMapper.fromNode(resource);
    parentNode.addEdge(CHILD_EDGE.getLabel(), nextParentNode,
        NodePosition.get(), position);

    int i = 0;
    for (Node childNode = graph.getFirstChild();
         childNode != null; childNode = childNode.getNextSibling())
    {
        traverseAndCreateGraph(nextParentNode, childNode, i++);
    }
}
```

Beim Lesen einer Seite muss der `GraphStorageManager` die entsprechende WOM-Ressource aus der Graphdatenbank laden. Dafür werden die Seiten- und Revisionsnummer beim Methodenaufruf übergeben. Anschließend wird der zugehörige Seitenvertex aus der Graphdatenbank geladen und als Einstiegspunkt ausgewählt. Der entsprechende Revisionsvertex kann im nächsten Schritt über das Traversieren der *Revision Edge* erreicht werden. Die gewünschte Ressource ist direkt mit dem Vertex der Revision verbunden und kann von diesem Punkt aus vollständig traversiert werden. Dafür startet der `GraphStorageManager` am Wurzelknoten und läd rekursiv alle weiteren Vertices der Ressource. An jedem Vertex wird der ursprüngliche WOM-Knoten durch Auslesen des serialisierten Knotens im Vertex-Property und anschließendem Deserialisieren erzeugt. Damit der ursprüngliche Baum der WOM-Ressource entsteht, müssen zusätzlich alle Kind-Knoten an den entsprechenden Eltern-Knoten gesetzt werden. Das Quellcodebeispiel aus Listing 3.7 zeigt die triviale Umsetzung um eine Ressource aus der Graphdatenbank zu laden. Beim ersten Aufruf der Funktion wird der Wurzelvertex der Ressource übergeben. Anschließend werden alle Vertices der Kinder geladen und iteriert um bei jedem Vertex die Funktion erneut rekursiv aufzurufen. Zusätzlich werden zurückgegebene Kind-Knoten am deserialisierten Eltern-Knoten gesetzt, wodurch letztendlich die vollständige Ressource entsteht. Dieser triviale Ansatz löst zwar das Problem eine Ressource aus dem Speicher zu laden, könnte aber durch Multi-Threading und Parallelisierung deutlich verbessert werden.

Listing 3.7: Laden einer Ressource aus der Datenbank.

```
Node traverseAndRetrieveGraphSerial(Vertex parentVertex)
{
    Node parentNode = VertexMapper.toNode(parentVertex);
    Iterator<Vertex> childVertices = graphDAO.retrieveChildVertices(
        parentVertex, Order.incr);

    while (childVertices.hasNext())
    {
        Node childNode = traverseAndRetrieveGraphSerial(childVertices.next()
            );
        parentNode.appendChild(childNode);
    }
    return parentNode;
}
```

TinkerPop bietet für das Traversieren und Laden vollständiger Bäume eine eigene Funktion an. Dadruch ist es möglich, über ein Statement die gesamte Ressource aus dem Speicher zu laden. Als Datenstruktur wird ein `Tree` Objekt zurückgegeben, das ähnlich einer Map genutzt werden kann. Sämtliche Vertices oder Kanten liegen in den Keys; die zugehörigen Values enthalten weitere `Tree` Objekte. Im Gegensatz zum trivialen Ansatz müssen sämtliche Vertices vom Client aus sortiert werden, weil das Statement keine Möglichkeiten zur Sortierung bietet. Der kompakte Funktionsaufruf ist in Listing 3.8 sichtbar und kann in diesem Fall nicht parallelisiert werden.

Listing 3.8: Statement zum Laden des vollständigen Baums.

```
Tree tree = graph.traversal().V(parentVertex).repeat(outE().otherV())
    .emit().tree().next();
```

Das Löschen und Ändern bereits vorhandener Seiten muss nicht behandelt werden, weil durch die umfangreiche Revisionshistorie ausschließlich neue Ressourcen entstehen. Jede schreibende Operation führt demnach zu einer Vergrößerung des gesamten Graphen in der Datenbank und kann auf eine Einfügeoperation zurückgeführt werden. Die bisherige Umsetzung behandelt lediglich das vollständige Einfügen und Auslesen einer WOM-Ressource. Diese triviale Strategie erfüllt bereits die grundlegenden Anforderungen des Graphspeichers und ermöglicht das theoretische Abspeichern von Wikipedia-Artikeln. In der Realität treten dennoch gravierende Probleme auf, weil dadurch unglaubliche Mengen von Vertices und Kanten entstehen. Um dieser Problematik entgegenzuwirken, werden im weiteren Verlauf Ansätze zur Reduzierung der Vertexanzahl vorgestellt und implementiert.

3.2.1 Traversieren von WOM-Ressourcen

Das Auslesen von Wikipedia-Seiten wird durch den `GraphStorageTreeWalker` erweitert. Dieser ermöglicht das Traversieren einer WOM-Ressource ohne dabei den vollständigen Baum aus der Graphdatenbank zu laden. Der Walker startet an einem Einstiegsknoten und kann von dieser Position aus alle nächsten Knoten abfragen. Sämtliche Funktionen werden durch das `TreeWalker` Interface der W3C API festgelegt und durch den `GraphStorageTreeWalker` implementiert[7]. Der in Listing 3.9 abgebildete Quellcode entspricht dem `TreeWalker` Interface des W3C und enthält die Signaturen aller Funktionen, die durch den `GraphStorageTreeWalker` implementiert wurden.

Listing 3.9: TreeWalker Interface des W3C.

```
public interface TreeWalker
{
    public Node getRoot();
    public int getWhatToShow();
    public NodeFilter getFilter();
    public boolean getExpandEntityReferences();
    public Node getCurrentNode();
    public void setCurrentNode(Node currentNode) throws DOMException;
    public Node parentNode();
    public Node firstChild();
    public Node lastChild();
    public Node previousSibling();
    public Node nextSibling();
    public Node previousNode();
    public Node nextNode();
}
```

Um eine Instanz des `GraphStorageTreeWalker` zu erzeugen, muss der Revisionsbereich über die Artikel- und Revisionsnummer festgelegt werden. Dieser Bereich kann weder verlassen noch geändert werden und beschränkt sich auf die gewünschte WOM-Ressource. Artikel- oder Revisionsvertices können durch den TreeWalker nicht abgerufen werden, weil diese keine WOM-Knoten repräsentieren. In Listing 3.10 ist der Konstruktor des `GraphStorageTreeWalker` zu sehen. Dem Walker wird zusätzlich ein `NodeFilter` Objekt übergeben, wodurch sämtliche Knoten gefiltert werden können. `NodeFilter` wird ebenfalls durch die W3C API bereitgestellt und bietet eine Möglichkeit, Knoten nach selbst definierten Kriterien zu filtern. Dadurch können beim Traversieren einzelne Knoten übersprungen oder ignoriert werden.

[7] http://www.w3.org/2003/01/dom2-javadoc/org/w3c/dom/traversal/TreeWalker.html

Listing 3.10: Konstruktor des GraphStorageTreeWalker.

```
public class GraphStorageTreeWalker extends TreeWalker
{
    public GraphStorageTreeWalkerImpl(long articleId, long revisionId,
        NodeFilter nodeFilter, Graph graph)
    {
        this.nodeFilter = nodeFilter;
        this.graphDAO = new GraphDAOImpl(graph);
        this.scope = new GraphStorageScope(articleId, revisionId);
        this.rootNode = retrieveRootNode();
    }
    ...
}
```

Beim Initialisieren des `GraphStorageTreeWalker` wird zuerst der Wurzelknoten der gewünschten Ressource ausfindig gemacht. Dafür wird wie bereits in Abschnitt 3.2 der Artikelvertex als Einstiegspunkt verwendet um anschließend über den Revisionsvertex an die Wurzel zu kommen. Daraufhin wird der Wurzelknoten mit entsprechendem Vertex gesetzt und der Walker ist zur Verwendung bereit. Zu den gängigsten Operationen zählt das Abfragen der Geschwister-, Kind- oder Elternknoten. Zusätzlich kann die Ressource über die Funktionen `nextNode()` und `previousNode()` traversiert werden, wodurch jeder Knoten der Ressource nach einer vordefinierten Ordnung traversiert wird. Im Fall des `GraphStorageTreeWalker` wird stets der erste Kindknoten aufgerufen oder, falls dieser nicht existieren sollte, der nächste Geschwisterknoten. Gibt keiner der beiden Fälle einen Treffer, werden so lange Elternknoten abgerufen, bis ein nächster Geschwisterknoten exisitiert. Sollte dies nicht der Fall sein, befindet sich der Walker am Ende des Dokuments.

Listing 3.11: Überschriebende Funktionen des Node Interface.

```
public interface Node
{
    ...
    public Node getParentNode();
    public NodeList getChildNodes();
    public Node getFirstChild();
    public Node getLastChild();
    public Node getPreviousSibling();
    public Node getNextSibling();
    public boolean hasChildNodes();
    ...
}
```

Eine weitere Besonderheit des `GraphStorageTreeWalker` sind die Knotenobjekte, die der Nutzer bei einer erfolgreichen Traversierung als Ergebnis erhält. Keines dieser Objekte steht in Zusammenhang mit weiteren Knoten und existiert vollkommen eigenständig. Allerdings ist in der Realität jeder Knoten Teil einer WOM-Ressource und dadurch mit weiteren Knoten über Eltern-Kind-Beziehungen verbunden. Um trotzdem die Funktionen und Eigenschaften des Knoten zu gewährleisten, werden spezielle Proxyknoten an den Nutzer zurückpropagiert. Das zugehörige Architekturmuster kann von Gamma u. a. (1994) entnommen werden. Durch Proxies können alle Funktionen eines Knotens transparent verwendet werden ohne dabei dem Nutzer die interne Implementierung zu offenbaren. `GraphStorageProxyNode` Objekte arbeiten direkt auf der Graphdatenbank und verwalten das eigentliche Knotenobjekt. Ein Großteil der Funktionen wird direkt an das reale Knotenobjekt weitergegeben, die restlichen Funktionen leitet der Proxy zur Graphdatenbank und erzeugt dementsprechend weitere Proxyobjekte als Ergebnis. Listing 3.12 zeigt eine überschriebenen Funktion mit Datenbankweiterleitung. Wird beispielsweise der erste Kind-Knoten über einen Proxy-Knoten abgefragt, entsteht ein Datenbankaufruf mit anschließender Proxyerzeugung. Im Grunde verhalten sich die ausgewählten Funktionen der Proxy-Knoten ähnlich wie beim `GraphStorageTreeWalker` und bieten teilweise identische Funktionalitäten an. Für die Erzeugung eines Proxy wird der Vertex und Revisionsbereich des korrespondierenden Knotens benötigt. Mit diesen Informationen kann der ursprüngliche WOM-Knoten wiederhergestellt und durch den Proxy verwaltet werden.

Listing 3.12: Überschriebene Funktion mit Datenbankweiterleitung.

```
public Node getFirstChild()
{
    Optional<Vertex> firstChildVertex = graph.traversal().V(currentVertex).
        outE()
        .has(NodePosition.get(), 0);

    Node firstChild;
    if (firstChildVertex.isPresent())
    {
        firstChild = new GraphStorageProxyNode(scope, currentVertex);
    } else
    {
        firstChild = null;
    }
    return firstChild;
}
```

3.2.2 Algorithmus zur Graphoptimierung

Der Algorithmus zur Graphoptimierung benachbarter Revisionen stellt die größte technische Herausforderung des graphbasierten Speichers dar. Hintergrund war die Reduzierung von Redundanzen in der Graphdatenbank zwischen zwei aufeinanderfolgenden Revisionen. Als Grundlage dient das Laden von Ressourcen aus Abschnitt 3.2. Damit eine Umsetzung des vorgestellten Ansatzes aus Kapitel 2 möglich ist, muss ein neuer Kantentyp im System etabliert werden. Die sogenannten *History Edges* verbinden Teile des Graphen der Vorgängerrevision mit dem Graphen der aktuellen Revision. Dadurch können redundante Teilgraphen benachbarter Revisionen dezimiert werden. Bisher wurden ausschließlich *Child Edges* zur Modellierung von Eltern-Kind-Beziehungen verwendet. Im weitern Verlauf müssen ebenfalls *History Edges* beim Traversieren oder Auslesen von WOM-Ressouren berücksichtigt werden. Ein deutlicher Mehraufwand entsteht, wenn neue Ressourcen in die Graphdatenbank geladen werden, weil mehrere Schritte notwendig sind um Redundanzen zu erkennen und diese zu beheben. Im folgenden Abschnitt werden die entsprechenden Arbeitsabläufe und deren Realisierung genauer erklärt.

Damit der Algorithmus angewendet werden kann, muss ein Vorgänger der gewünschten Revision existieren. Aus diesem Grund kann die initiale Revision einer Seite weder optimiert noch reduziert werden. Ist eine Vorgängerrevision verfügbar, werden zunächst beide Ressourcen miteinander verglichen. Dabei müssen alle Änderungen zwischen Vorgänger und Nachfolger extrahiert und analysiert werden. Hierfür wird der von Dohrn und Riehle (2014) entwickelte HDDiff-Algorithmus als Werkzeug eingesetzt. Mit Hilfe des HDDiff können alle Unterschiede zwischen zwei WOM-Ressourcen analysiert werden. Als Ergebnis werden sämtliche Operationen aufgelistet, die zur Transformation der Vorgänger- in die Nachfolger-Revision nötig sind. Grundsätzlich existieren fünf verschiedene Operationen um eine Ressource in die gewünschte Form zu transformieren.

INSERT Operationen bedeuten das Einfügen eines neuen Knotens oder Teilbaums an die entsprechende Stelle in der Ressource.

UPDATE Operationen werden auf einen bereits existierenden Knoten angewendet um dessen Attribute oder Inhalte zu verändern.

DELETE Operationen löschen den angegebenen Knoten oder Teilbaum vollständig von der Ressource.

MOVE Operationen verschieben einen vollständigen Knoten oder Teilbaum an eine neue Stelle in der Ressource.

SPLIT Operationen teilen einen Knoten mit Textinhalt an der angegebenen Textstelle und erzeugen daraus zwei seperate Knoten.

Der HDDiff-Algorithmus erzeugt als Endresultat ein *Edit Script*, das einer Liste von Änderungsoperationen entspricht. Damit die Ergebnisse zur Identifizierung von Redundanzen verwendet werden können, muss das *Edit Script* in eine andere Form gebracht werden. Mit Hilfe des `EditScriptToChangeTreeConverter` wird das *Edit Script* in einen sogenannten Änderungsbaum konvertiert. Dieser besteht aus `ChangeTreeNode` Objekten, die wiederum sämtliche Änderungsoperationen als `ChangeTreeOperation` enthalten. Je nach Art der Änderung können diese Objekte unterschiedliche Attribute enthalten. Beispielsweise enthält eine *INSERT* Operation zusätzlich zur Einfügeposition den entsprechenden Knoten oder Teilbaum, der eingefügt werden soll. Des Weiteren werden die urspünglichen fünf Operationen um eine zusätzliche Änderungsoperation erweitert. Diese *NODE* Operation zeigt auf einen Knoten, dessen Kindknoten Änderungen beinhalten. Darüber hinaus beinhaltet die *NODE* Operation eine weitere Liste an Änderungsoperationen. Diese Änderungen verweisen auf die Kinder des referenzierten Knotens. Durch die Verschachtelung dieser *NODE* Operationen entstehen Baumstrukturen, die letztendlich den Änderungsbaum bilden. Dabei werden *NODE* Operationen als Innerknoten, die restlichen Operationen als Blattknoten betrachetet. Eine weitere Besonderheit ist das Aufspalten der *MOVE* Operation in zwei seperate *MOVE_ FROM* und *MOVE_ TO* Operationen. *MOVE_ FROM* Operationen zeigen an, welcher Knoten verschoben wird, *MOVE_ TO* Operation geben die Einfügeposition des Knotens an. Demnach existieren insgesamt sieben verschiedene Änderungsoperationen innerhalb des Änderungsbaums. Ein großer Vorteil, der durch die Konvertierung entsteht, ist das einfache Traversieren des Änderungsbaums. Wird zur gleichen Zeit die Ressource der Vorgängerrevision traversiert, können die Änderungsoperationen direkt auf die Knoten angewendet werden. Durch die Baumstruktur der Änderungen wird die Tiefenposition implizit gegeben. Werden die beiden Bäume aufeinandergelegt können die Änderungsoperationen direkt an den entsprechenden Stellen ausgeführt werden ohne dabei die passende Ebene im Baum suchen zu müssen. Im weiteren Verlauf legt der Änderungsbaum die Grundlage für eine bestmögliche Optimierung des Graphen.

Wird eine neue Revision erstellt, gibt der Nutzer die Seiten- und Revisionsnummer mit der korrespondierenden Ressource an den `GraphStorageManager`. Zusätzlich muss die Nummer der zugehörigen Vorgänger-Revision angegeben werden, ansonsten würde es sich bei der neuen Ressource um die initiale Revision einer Seite handeln. In diesem Fall kann der Algorithmus nicht angewendet werden, weil keine Vorgänger-Revision existiert. Sind beide Revisionen verfügbar, können die entsprechenden Ressourcen mit Hilfe des HDDiff auf Änderungen hin analysiert und der Änderungsbaum erstellt werden. Dazu holt der `GraphStorageManger` die WOM-Ressource der Vorgängerrevision aus der Graphdatenbank und steckt diese zusammen mit der bereits übergebenen WOM-Ressource in den HDDiff-Algorithmus. Das erhaltene *Edit Script* wird anschließend in einen Änderungsbaum konvertiert.

Der eigentliche Algorithmus besteht aus zwei essentiellen Arbeitsschritten. Im ersten Schritt wird der Änderungsbaum traversiert. Wehrendessen müssen die betroffenen Vertices der Vorgängerrevison aus der Graphdatenbank geladen und in `VertexWrapper` Objekte verpackt werden, damit die Änderungsoperationen an den entsprechenden Stellen druchgeführt werden können. Listing 3.13 zeigt die Implementierung der `VertexWrapper` Klasse. Zusätzlich zum eigentlichen WOM-Knoten beinhaltet ein `VertexWrapper` Objekt seine Kindknoten, die wiederum als `VertexWrapper` verpackt sind. Die restlichen Attribute der Klasse werden im zweiten Arbeitsschritt beim Erzeugen der Vertices benötigt. Dafür wird die Baumstruktur aus `VertexWrapper` Objekten traversiert, um Schritt für Schritt die Ressource der Nachfolgerrevision in die Graphdatenbank zu laden. Die beiden Prozesse werden im weiteren Verlauf näher erläutert.

Listing 3.13: Implementierung der VertexWrapper Klasse.

```
public class VertexWrapper
{
    Vertex vertex;
    Vertex previousVertex;
    List<VertexWrapper> childNodes;
    Node node;

    //flags
    boolean isNew;
    boolean isUpdated;
}
```

Wie bereits erwähnt, werden im ersten Arbeitsschritt alle Operationen des Änderungsbaums auf die WOM-Ressource der Vorgänger-Revision angewendet. Dafür ist die zentrale Funktion aus Listing 3.14 zuständig. Beim initialen Aufruf der Funktion werden der Wurzelknoten und eine Liste mit Änderungsoperationen als Parameter übergeben. Beim Wurzelknoten handelt es sich um die Wurzel der Vorgänger-Revision verpackt als `VertexWrapper` Objekt. Die Liste an Operationen beinhaltet alle Änderungen der direkten Kinder des übergebenen Knotens. Aus diesem Grund müssen müssen alle direkten Kinder aus der Graphdatenbank geladen werden. Anschließend werden in einer Schleife alle Änderungsoperationen nacheinander abgefragt und auf die entsprechenden `VertexWrapper` Objekte der Kind-Knoten angewendet. Zu diesem Zeitpunkt werden noch keine neuen Vertices in der Graphdatenbank erzeugt. Es handelt sich hierbei um das Anwenden der Änderungsoperationen auf die temporären `VertexWrapper` Objekte im Hauptspeicher. Daher entstehen keine Änderungen oder Schreibzugriffe in der Graphdatenbank. Im Folgenden werden die unterschiedlichen Arten von Operationen vorgestellt. Hierbei spielt die individuelle Behandlung jedes Änderungstyps eine wichtige Rolle.

Listing 3.14: Funktion zum Durchführen aller Änderungen

```
VertexWrapper getRevisionGraph(VertexWrapper parentNode, Collection<
    ChangeTreeNode> changes)
{
    List<VertexWrapper> childNodes = getChildVertices(parentNode);
    parentNode.childNodes = childNodes;

    for (ChangeTreeNode change : changes)
    {
        switch (change.getOp())
        {
            case NODE:
                handleNode((CTNNode) change, childNodes);
                break;

            case INSERT:
                handleInsert((CTNInsert) change, childNodes);
                break;

            case UPDATE:
                handleUpdate((CTNUpdate) change, childNodes);
                break;

            case DELETE:
                handleDelete((CTNDelete) change, childNodes);
                break;

            case MOVE_FROM:
                handleMoveFrom((CTNMove) change, childNodes);
                break;

            case MOVE_TO:
                handleMoveTo((CTNMove) change, childNodes);
                break;

            case SPLIT:
                handleSplit((CTNSplit) change, childNodes);
                break;
        }
    }
    return parentNode;
}
```

Damit alle Änderungsoperationen abgearbeitet werden können, müssen sämtliche Kind-Knoten aus der Graphdatenbank geholt werden. Die Funktion aus Listing 3.15 übernimmt diese Aufgabe. Über ein Statement werden die entsprechenden Vertices sortiert aus der Graphdatenbank geladen und iteriert. Anschließend wird aus jedem Vertex ein `VertexWrapper` Objekt erzeugt und in die Knotenliste geschrieben. Der WOM-Knoten wird dabei deserialisiert und ebenfalls im `VertexWrapper` abgelegt. Am Ende wird die Liste aller Kind-Knoten zurückgegeben und die Behandlung der Änderungsoperationen beginnt.

Listing 3.15: Laden der Kind-Knoten aus der Datenbank.

```
List<VertexWrapper> getChildVertices(VertexWrapper vertexWrapper)
{
    List<VertexWrapper> childVertices;

    if (vertexWrapper.childNodes.isEmpty())
    {
        childVertices = new LinkedList<>();
        Iterator<Vertex> vertices = graph.traversal().V(parentVertex)
            .outE(CHILD_EDGE.getLabel(), HISTORY_EDGE.getLabel())
            .order().by(NodePosition.get(), order).otherV();

        while (vertices.hasNext())
        {
            Vertex childVertex = vertices.next();
            Node childNode = VertexMapper.toNode(childVertex);

            VertexWrapper childVertexWrapper = VertexWrapper.get(
                childVertex);
            childVertexWrapper.node = childNode;
            childVertices.add(childVertexWrapper);
        }
    } else
    {
        childVertices = vertexWrapper.childNodes;
    }

    return childVertices;
}
```

NODE Operationen dienen ausschließlich zur Traversierung der Ressource und wurden künstlich durch die Konvertierung des *Edit Scripts* erzeugt. Wird eine *NODE* Operation angewendet, ist der referenzierte Knotens nur indirekt von Änderungen betroffen. Ein `NODE` bedeutet immer, dass die Kinder des referenzierten Knoten Änderungen beinhalten. Das `CTNNode` Objekt enthält die Liste der Änderungen zusammen mit der Position des betroffenen Knotens in der Knotenliste. Der Quellcode in Listing 3.16 beinhaltet die Vorgehensweise beim Auftreten einer *NODE* Operation. Im `VertexWrapper` wird ein Flag gesetzt, dass ein neuer Vertex für den WOM-Knoten erzeugt werden muss. Anschließend wird die Funktion aus Listing 3.14 erneut aufgerufen.

Listing 3.16: Handler für NODE Operationen.

```
void handleNode(CTNNode node, List<VertexWrapper> childNodes)
{
    VertexWrapper nextParentNode = vertices.get(node.getPos());
    vertexWrapper.isNew = true;
    getRevisionGraph(nextParentNode, node.getChanges());
}
```

INSERT Operationen betreffen das Einfügen neuer Knoten. Beim Anwenden eines *INSERT* wird der mitgelieferte Knoten oder Baum aus dem `CTNInsert` Objekt geladen und als `VertexWrapper` verpackt. Dabei entsteht im Fall eines einzufügenden Baums ebenfalls ein Baum aus `VertexWrapper` Objekten. Anschließend wird der konvertierte Baum in die Liste der Knoten eingefügt. Listing 3.17 zeigt die Funktion zur Behandlung von *INSERT* Operationen.

Listing 3.17: Handler für *INSERT* Operationen.

```
void handleInsert(CTNInsert insert, List<VertexWrapper> childNodes)
{
    VertexWrapper wrappedTree = wrapTree(insert.getTree());
    childNodes.add(insert.getPos(), insertedTreeRootNode);
}
```

Um den WOM-Knoten in ein `VertexWrapper` Objekt zu konvertieren, wird die Funktion aus Listing 3.18 verwendet. Dafür wird eine flache Kopie des WOM-Knotens erzeugt, die keine weiteren Eltern- oder Kind-Knoten enthält. Anschließend wird das entsprechende `VertexWrapper` Objekt mit der Knotenkopie initialisiert. Zusätzlich wird ein entsprechendes Flag gesetzt, dass ein neuer Vertex für diesen Knoten erzeugt werden muss. Für jeden weiteren Kind-Knoten wird diese Funktion erneut aufgerufen.

Listing 3.18: Konvertieren eines WOM-Knotens in VertexWrapper.

```
VertexWrapper wrapTree(Node tree)
{
    Node flatNode = treeNode.cloneNode(false);
    VertexWrapper node = new VertexWrapper.get(flatNode)
    node.isNew = true;

    for (Node childNode = treeNode.getFirstChild();
        childNode != null; childNode = childNode.getNextSibling())
    {
        node.childNodes.add(wrapTree(childNode));
    }

    return node;
}
```

UPDATE Operationen aktualisieren die Inhalte des betroffenen Knotens. Dabei können sich Attribute oder Daten im Knoten ändern. Identität und Typ des Knotens bleiben dabei stets erhalten und können durch diese Art von Operation nicht verändert werden. Die in Listing 3.19 abgebildete Funktion behandelt die Durchführung einer *UPDATE* Operation und aktualisiert den referenzierten Knoten. Dabei wird ein weiteres Flag im `VertexWrapper` Objekt gesetzt, das anzeigt, ob ein Update durchgeführt wurde. Bei Änderungen im Bereich der Attribute werden lediglich die Attribute des im `VertexWrapper` liegenden Knotenobjekts aktualisiert. Sind Inhaltsänderungen vorhanden, müssen weitere Maßnahmen getroffen werden. Grund dafür ist das Verhalten der *UPDATE* Operation. Werden beispielsweise Textinhalte an Blatt-Knoten verändert zeigt die *UPDATE* Operation auf den Eltern-Knoten des eigentlichen Text-Knotens. Im WOM-Format würde beim Setzten des Textinhalts am Eltern-Knoten automatisch ein neuer Blatt-Knoten erzeugt werden. Dieses Verhalten sorgt dafür, dass bei inhaltlichen Updates ein völlig neuer WOM-Knoten entsteht, der per `VertexWrapper` als Kind-Knoten eingefügt werden muss. Zusätzlich muss beim Eltern-Knoten das Flag gesetzt werden, dass ein neuer Vertex für diesen WOM-Knoten erzeugt werden muss.

Listing 3.19: Handler für *UPDATE* Operationen.

```
void handleUpdate(CTNUpdate update, List<VertexWrapper>
    childNodes)
{
    VertexWrapper childNode = childNodes.get(update.getPos());
    childNode.isUpdated = true;

    if (update.getNewValue() != null)
    {
        childNode.setTextContent(update.getNewValue());
        childNode.childNodes = wrapTree(childNode.node).childNodes;
        childNode.node = childNode.node.cloneNode(false);
        childNode.isNew = true;
    }

    if (update.getAttributeChanges() != null)
        updateAttrs(childNode.node, update.getAttributeChanges());
}
```

DELETE Operationen führen zum direkten Löschen eines Knotens oder Teilbaums. Der betroffene Knoten wird direkt aus der Knotenliste entfernt und ist damit nicht mehr verfügbar. Listing 3.20 zeigt den Quellcode der Funktion zum Behandeln von *DELETE* Operationen.

Listing 3.20: Handler für *DELETE* Operationen.

```
void handleDelete(CTNDelete delete, List<VertexWrapper> children)
{
    VertexWrapper refChild = children.get(delete.getPos());
    children.remove(refChild);
}
```

MOVE_FROM Operationen referenzieren einen zu verschiebenden Knoten oder Teilbaum. Der betroffene Knoten wird an der entsprechenden Position aus der Knotenliste entfernt und zwischengespeichert. Dafür wird die Referenz des Knotens zusammen mit der zugehörigen `MOVE` Nummer als Key in einer Map abgelegt. Die korrespondierende *MOVE_ TO* Operation verwendet diesen Eintrag um den entsprechenden Knoten an seine neue Position zu verschieben. Die Reihenfolge, in der die beiden `MOVE` Operationen auftreten, ist nicht festgelegt. Aus diesem Grund müssen besondere Maßnahmen getroffen werden, wenn zugehörige *MOVE_ TO* Operation bereits vor der *MOVE_ FROM* Operation ausgeführt wurden. Ein Zwischenspeichern des Knotens würde demnach zu keiner Verschiebung führen, weil die korrespondierende *MOVE_ TO* Operation bereits durchgelaufen ist. In diesem Fall

übernimmt die *MOVE_FROM* Operation den kompletten Verschiebungsprozess. Die Information, an welcher Stelle eingefügt werden muss, wird durch die vorangegangene *MOVE_TO* Operation durch einen Platzhalter in der Map bereitgestellt. Um zu prüfen, ob ein Platzhalter vorhanden ist, muss die Funktion aus Listing 3.21 lediglich den Map-Eintrag an der entsprechenden `MOVE` Nummer erstellen. Wird ein Objekt zurückgegeben, muss es sich hierbei um einen Platzhalter handeln, der durch eine vorangegangene *MOVE_TO* Operation entstanden ist. Falls kein Objekt in der Map liegt, müssen keine weiteren Maßnahmen getroffen werden und die zugehörige *MOVE_TO* wird zu einem späteren Zeitpunkt auftreten.

Listing 3.21: Handler für *MOVE_FROM* Operationen.

```
void handleMoveFrom(CTNMove move, List<VertexWrapper>
    childNodes)
{
    VertexWrapper movedChild = childNodes.get(move.getPos());
    vertices.remove(movedChild);

    VertexWrapper placeholder = moveMap.put(move.getId(),
        movedChild);
    if (placeholder != null)
    {
        placeholder.set(movedChild);
    }
}
```

MOVE_TO Operationen sind das Gegenstück zur *MOVE_FROM* Operation und fügen den verschobenen Knoten an der entsprechenden Stelle ein. Dafür muss der Eintrag des Knotens in der Map abgerufen und in die Knotenliste an der gegebenen Position eingefügt werden. Der Quellcode in Listing 3.22 behandelt die Durchführung einer *MOVE_TO* Operation. Der Fall, in dem die zugehörige *MOVE_FROM* Operation noch nicht ausgeführt wurde und kein Eintrag in der Map vorhanden ist, erfordert die Erzeugung eines Platzhalters. Dafür wird ein leeres `VertexWrapper` Objekt verwendet und an der gegebenen Position in die Knotenliste eingefügt. Zusätzlich wird ein Map-Eintrag erstellt, der die `MOVE` Nummer und das Platzhalterobjekt beinhaltet. Die kommende *MOVE_FROM* Operation kann dieses Platzhalterobjekt aus der Map laden und den korrekten Knoten setzten.

Listing 3.22: Handler für *MOVE_ TO* Operationen.

```
private void handleMoveTo(CTNMove move, List<VertexWrapper>
    childNodes)
{
    int position = move.getPos();
    VertexWrapper movedChildNode = moveMap.get(move.getId());

    if (movedChildNode == null)
    {
        VertexWrapper vertexPlaceholder = VertexWrapper.
            getPlaceholder();
        vertices.add(position, vertexPlaceholder);
        moveMap.put(move.getId(), vertexPlaceholder);
    } else
    {
        vertices.add(position, movedChildNode);
    }
}
```

SPLIT Operationen spalten einen Blattknoten mit textuellem Inhalt an der angegebenen Textstelle in zwei separate Knoten mit den jeweiligen Textteilen. Ähnlich wie bei *UPDATE* Operationen werden die Elternknoten der zu teilenden Textknoten referenziert. Aus diesem Grund müssen ebenfalls spezielle Schritte unternommen werden, um die korrekten Knoten zu teilen. Zuerst muss die in Listing 3.23 abgebildete Funktion die entsprechenden Knoten aus der Datenbank holen. Dafür wird der Elternknoten an gegebener Position aus der Knotenliste geholt und über die Funktion `retrieveNode(VertexWrapper node)` vollständig geladen. Grund für diese Vorgehensweise ist die Tatsache, dass sämtliche WOM-Knoten ausschließlich flache Kopien der originalen Knoten sind und keine weiteren Eltern- oder Kind-Knoten enthalten. Anschließend kann der Textinhalt an der entsprechenden Stelle gespalten werden. Aus den beiden Teilen werden im Anschluss zwei separate WOM-Knoten erzeugt. Dafür wird der alte WOM-Knoten mit dem ersten Textsegment aktualisiert und ein weiterer neuer WOM-Knoten mit dem übrigen Textsegment erzeugt. Danach werden beide Knoten in richtiger Reihenfolge an der ursprünglichen Position in die Knotenliste eingefügt.

Listing 3.23: Handler für *SPLIT* Operationen.

```
void handleSplit(CTNSplit split, List<VertexWrapper> childNodes)
{
    VertexWrapper childNode = vertices.get(split.getPos());
    Node splitNode = retrieveGraph(childNode);
    String text = splitNode.getTextContent();
    String textA = text.substring(0, split.getSplitPos());
    String textB = text.substring(split.getSplitPos());

    Node nodeB = previousNode.getOwnerDocument().createElementNS
        (Wom3Node.WOM_NS_URI, "text");
    previousNode.setTextContent(textA);
    nodeB.setTextContent(textB);

    VertexWrapper splitNodeA = wrapTree(previousNode);
    VertexWrapper splitNodeB = wrapTree(nodeB);

    vertices.remove(childNode);
    vertices.add(split.getPos(), splitNodeB);
    vertices.add(split.getPos(), splitNodeA);
}
```

Sind alle Rekursionsschritte der Funktion aus Listing 3.14 mit sämtlichen Änderungsoperationen durchgelaufen, wird ein `VertexWrapper` Objekt an den Aufrufer zurückgegeben. Dieses Objekt beinhaltet eine Baumstruktur mit allen Informationen, die nötig sind, um den Graphen der Nachfolger-Revision zu erzeugen. Erst an dieser Stelle setzt der zweite Arbeitsschritt des Algorithmus an und erzeugt neue Vertices in der Graphdatenbank. Die entsprechende Funktion ist in Listing 3.24 abgebildet und erzeugt die Ressource der Nachfolger-Revision auf Datenbankebene. Dabei wird der `VertexWrapper` Baum traversiert um sukzessiv die Vertices und Kanten in der Graphdatenbank zu erstellen.

Listing 3.24: Funktion zum Erzeugen des Graphen der Revision.

```
void doCreateRevisionGraph(VertexWrapper parentNode)
{
    for (int pos = 0; pos < parentNode.childNodes.size(); pos++)
    {
        VertexWrapper childNode = parentNode.childNodes.get(pos);
        connectNodes(rootNode, childNode, pos);
        doCreateRevisionGraph(childNode);
    }
}
```

Die Hauptfunktionalität liegt beim Verbinden der Knoten und wird durch die Funktion in Listing 3.32 realisiert. Als Parameter wird der Eltern- und Kind-Knoten sowie die zugehörige Position übergeben. Ein gültiges `Vertex` Objekt im Elternknoten ist vorauszusetzen und wird beim ersten Aufruf durch den Wurzelknoten der Ressource gegeben. Anschließend können unterschiedliche Fälle auftreten, die je nach Situation zu einer anderen Behandlung führen. Als erstes wird überprüft, ob das Flag zur Erzeugung eines neuen Vertex gesetzt ist. Sollte dieser Fall zutreffen, wird ein neuer Vertex für den entsprechenden Knoten erzeugt und mit dem Vertex des Eltern-Knoten über eine *Child Edge* verbunden. Ansonsten wird im nächsten Schritt das Flag für Änderungen der Knotenattribute betrachtet. Bei gesetztem Flag wird ebenfalls ein neuer Vertex erzeugt und mit dem Vertex des Eltern-Knoten über eine *Child Edge* verbunden. Zusätzlich müssen hier alle Kinder des ursprünglichen Vertex mit dem neuen Vertex über *History Edges* verbunden werden. Falls keiner der beiden genannten Fälle zutreffen sollte, enthält der Kind-Knoten keine Änderungen. Daher kann eine *History Edge* zwischen dem ursprünglichen Vertex des Kind-Knotens aus der Vorgängerrevision und dem Vertex des übergebenen Elternknotens gesetzt werden.

Listing 3.25: Funktion zum Erzeugen und Verbinden von Vertices.

```
void connectNodes(VertexWrapper parentNode, VertexWrapper childNode,
        int position)
{
    if (childNode.isNew)
    {
        childNode.vertex = VertexMapper.fromNode(childNode.node);
        rooNode.vertex.addEdge(CHILD_EDGE.getLabel(),
            childNode.vertex, NodePosition.get(), position);
    }
    else if (childNode.isUpdated)
    {
        childNode.vertex = VertexMapper.fromNode(childNode.node);
        rooNode.vertex.addEdge(CHILD_EDGE.getLabel(),
            childNode.vertex, NodePosition.get(), position);
        moveChildVertices(childNode.previousVertex, childVertex);
    }
    else
    {
        rooNode.vertex.addEdge(HISTORY_EDGE.getLabel(),
            childNode.vertex, NodePosition.get(), position,
            RevisionId.get(), scope.getRevisionId());
    }
}
```

Nachdem die Funktion aus Listing 3.24 vollständig durchgelaufen ist, endet der Algorithmus. Damit die erzeugten Vertices in die Graphdatenbank geschrieben werden, muss eine Bestätigung der Transaktion folgen. Anschließend können sämtliche Map-Einträge der `MOVE` Operationen gelöscht werden. Der entstandene Graph in der Datenbank entspricht der gewünschten WOM-Ressource und setzt sich aus Vertices der Vorgänger- und Nachfolger-Revision zusammen. Werden Vertices der Vorgängerrevision verwendet, sind diese ausschließlich über *History Edges* mit den Vertices der neuen Revision verbunden. Die Tatsache, dass Vertices der Vorgängerrevision ihrerseits wiederum durch *History Edges* aus einer Vor-Vorgängerrevision referenziert sein könnten, bedeutet, dass sich eine Revision aus Vertices unterschiedlicher vorangegangener Revisionen zusammensetzen kann.

Hintergrund des Algorithmus war die Eliminierung redundanter Knoten oder Teil-Graphen. In der Realität können nicht alle Redundanzen vermieden werden, weshalb selbst nach Anwendung des Algorithmus identische Knoten mehrfach abgespeichert werden. Das Problem liegt an der Struktur des Baums. Wird beispielsweise ein Kind-Knoten verändert, muss der zugehörige Vertex des Eltern-Knotens automatisch neu erzeugt werden, selbst wenn dieser keine eigenen Änderungen enthält. Ohne Erzeugung eines neues Vertex müsste der ursprüngliche Vertex des Eltern-Knotens mit dem neuen Vertex des Kind-Knotens verbunden werden. In diesem Fall wäre ohne weitere Informationen keine eindeutige Traversierung der Ressource mehr möglich, weil der Vertex des Eltern-Knotens beide Versionen des Kind-Knotens referenziert.

Durch die Verwendung des Algorithmus müssen beim Laden oder Traversieren einer Ressource minimale Anpassungen vorgenommen werden. Bisher wurden sämtliche Kind-Knoten durch Traversieren der *Child Edges* des Eltern-Knoten abgefragt. Diese Funktion wird durch das zusätzliche Filtern nach *History Edges* erweitert. Beim Abfragen von Kind-Knoten müssen demnach beide Kantentypen berücksichtigt werden. Anschließend können die bisherigen Funktionalitäten problemlos weiterverwendet werden.

3.2.3 Begrenzung der Baumtiefe

Für die variable Tiefenbegrenzung von Vertices anhand einer vordefinierten Liste von WOM-Knotentypen müssen weitere Anpassungen vorgenommen werden. Die Namen der betroffenen Knoten werden in einer Liste verwaltet. Dafür müssen lediglich zu Beginn der Initialisierung des `GraphStorageManager` die Knotennamen angegeben werden. Anhand dieser Liste können die restlichen Funktionen überprüfen ob es sich bei einem Knoten um einen sogenannten limitierten Knoten handelt. Listing 3.26 zeigt die erweiterte Funktion des `GraphStorageManager` zum Erstellen einer neuen Revision. Ist der Knotenname in der Liste vorhanden

wird die Traversierung an dieser Stelle beendet. Über den `LimitedVertexMapper` wird der Knoten mit allen Kinder in ein Vertex umgewandelt, der als Property den Serialisierten Knoten enthält.

Listing 3.26: Funktion zum Laden einer Ressource aus der Datenbank.

```
private void traverseAndCreateGraph(Vertex parentNode, Node graph,
        int position)
{
    boolean isLimited = limitedNodes.contains(graph.getNodeName());
    Vertex nextParentNode = LimitedVertexMapper.fromNode(graph,
        isLimited);
    parentNode.addEdge(CHILD_EDGE.getLabe(), nextParentNode,
            NodePosition.get(), position);

    if (!isLimited)
    {
        int i = 0;
        for (Node childNode = graph.getFirstChild();
              childNode != null; childNode = childNode.getNextSibling())
        {
            traverseAndCreateGraph(nextParentNode, childNode, i++);
        }
    }
}
```

Beim Auslesen der Ressource müssen keine weiteren Anpassungen vorgenommen werden. Durch Verwendung des `LimitedVertexMapper` wird im Falle eines limitierten Knotens der vollständige WOM-Knoten deserialisiert und zurückgegeben. Ist das entsprechende Vertex-Property nicht gesetzt wird der umgekehrte Abbildungsprozess wie gewohnt fortgesetzt. Die Traversierung muss beim Auslesen nicht gestoppt werden weil die Vertices limitierter Knoten keine ausgehenden *Child Edges* mehr besitzen.

Weitere Anpassungen müssen im Algorithmus aus Abschnitt 3.2.2 vorgenommen werden. Dabei sind genau zwei Stellen von Änderungen betroffen. Zum Einen wird die in Listing 3.27 abgebildete Funktion durch die Überprüfung des Knotens erweitert. Falls es sich dabei um einen limitierten Knoten handelt muss der komplette WOM-Knoten in einen Baum aus `VertexWrapper` Objekten umgewandelt werden.

Listing 3.27: Funktion zum Laden aller Kind-Knoten aus der Datenbank.

```
List<VertexWrapper> getChildVertices(VertexWrapper vertexWrapper)
{
    List<VertexWrapper> childVertices;
    if (vertexWrapper.childNodes.isEmpty())
    {
        childVertices = new LinkedList<>();
        Iterator<Vertex> vertices = graph.traversal().V(parentVertex)
            .outE(CHILD_EDGE.getLabel(), HISTORY_EDGE.getLabel())
            .order().by(NodePosition.get(), order).otherV();

        while (vertices.hasNext())
        {
            Vertex childVertex = vertices.next();
            Node childNode = LimitedVertexMapper.toNode(childVertex);
            VertexWrapper childVertexWrapper;

            if (limitedNodes.contains(childNode.getNodeName()))
            {
                childVertexWrapper = wrapTree(childNode);
            } else
            {
                childVertexWrapper = VertexWrapper.get(childVertex);
                childVertexWrapper.node = childNode;
            }
            childVertices.add(childVertexWrapper);
        }
    } else
    {
        childVertices = vertexWrapper.childNodes;
    }
    return childVertices;
}
```

Die zweite Stelle betrifft das Erzeugen der Vertices im letzten Schritt des Algorithmus. Hier muss bei der Funktion aus Listing 3.28 eine Prüfung eingefügt werden. Falls der Knoten limitiert werden muss, wird der `VertexWrapper` Baum in einen WOM-Baum transformiert. Anschließend kann dieser Baum als Vertex-Property abgelegt werden. Zuletzt wird eine *Child Edge* zwischen dem Vertex des Elternknoten und des neu entstandenen Vertex gesetzt. Damit keine weiteren Kind-Knoten des limitierten Knoten traversiert werden müssen zusätzlich alle Kinder aus der entsprechen Liste des `VertexWrapper` gelöscht werden.

Listing 3.28: Erzeugen und Verbinden von Vertices.

```
void connectNodes(VertexWrapper parentNode, VertexWrapper childNode,
        int position)
{
    if (limitedNodes.contains(childNode.node.getNodeName()))
    {
        Node limitedTree = getNodeTree(childNode);
        childNode.vertex = LimitedVertexMapper.fromNode(childNode.node,
            true);
        rooNode.vertex.addEdge(CHILD_EDGE.getLabel(),
            childNode.vertex, NodePosition.get(), position);
        childNode.childNodes.clear();
    }
    else if (childNode.isNew)
    {
        childNode.vertex = LimitedVertexMapper.fromNode(childNode.node);
        rooNode.vertex.addEdge(CHILD_EDGE.getLabel(),
            childNode.vertex, NodePosition.get(), position);
    }
    else if (childNode.isUpdated)
    {
        childNode.vertex = LimitedVertexMapper.fromNode(childNode.node);
        rooNode.vertex.addEdge(CHILD_EDGE.getLabel(),
            childNode.vertex, NodePosition.get(), position);
        moveChildVertices(childNode.previousVertex, childVertex);
    }
    else
    {
        rooNode.vertex.addEdge(HISTORY_EDGE.getLabel(),
            childNode.vertex, NodePosition.get(), position,
            RevisionId.get(), scope.getRevisionId());
    }
}
```

Durch die Implementierung der genannten Änderungen kann eine Tiefenbegrenzung auf Vertexebene stattfinden. Der entstandene Mehraufwand ist minimal und birgt keine gravierenden Leistungseinbrüche. Dafür müssen deutlich weniger Vertices erzeugt und aus der Datenbank geladen werden. Demnach kann eine Leistungssteigerung erwartet werden weil die Datenbank ein Flaschenhals der Anwendung ist. Ein Nachteil dieser Tiefenbegrenzung entsteht beim Dezimieren von Redundanzen. Durch die verkapselten und serialisierten WOM-Bäume können unter Umständen deutlich weniger *History Edges* gesetzt werden. Das führt

wiederum zu mehr Redundanzen, die durch den Algorithmus zur Graphoptimierung reduziert werden sollten. Die optimale Lösung liegt im Mittelmaß aus der Tiefenbegrenzung und Minimierung von Redundanzen.

3.3 Nutzerspezifische Kanten

Die Realisierung nutzerspezifischer Verbindungen greift in die bisherige Implementierung insofern ein, dass weitere Kantentypen zwischen beliebigen Vertices entstehen. Bisher wurden ausschließlich feste Kantentypen verwendet, zu denen *Revision Edges*, *Child Edges* und *History Edges* zählen. Durch die Umsetzung von nutzerspezifischen Kanten wird eine Vielzahl neuer Kantenypen vorgestellt, die durch den Nutzer definiert und verwaltet werden. Der Graphspeicher muss diese Kanten separieren und als eigenständige Untermenge aller Kanten betrachten. Beim Traversieren von Ressourcen werden zunächst sämtliche nutzerspezifischen Kanten als transparent betrachtet und ignoriert. Nur durch explizite Anweisungen und eine erweiterte Form des `GraphStorageTreeWalker` können diese Kanten genutzt werden.

Um einen neuen Kantentyp zu definieren, muss über den `GraphStorageManager` eine sogenannte `GraphStorageEdgeRegistry` angefordert werden. Dieses Register bietet die Möglichkeit, über das in Listing 3.29 abgebildete Interface eigene Kantentypen zu registrieren. Wird ein `GraphStorageEdgeType)` erstellt, kann dieser über das Register abgefragt und verwendet werden. Die Erzeugung und Verwendung von nicht registrierten Kanten ist nicht möglich.

Listing 3.29: Interface zum Erzeugen von nutzerspezifischen Kantentypen.

```
interface GraphStorageEdgeRegistry
{
    Optional<GraphStorageEdgeType> registerEdgeType(String label);
    void deregisterEdgeType(GraphStorageEdgeType edgeType);
    boolean isRegistered(String label);
    Optional<GraphStorageEdgeType> retrieveEdgeType(String label);
    Collection<GraphStorageEdgeType> retrieveEdgeTypes();
}
```

Um nutzerspezifische Kanten zwischen zwei Vertices zu erzeugen, muss das entsprechende Kantenobjekt beim Erstellen einer neuen Revision übergeben werden. Dafür holt sich der Nutzer den gewünschten Kantentyp aus dem Register und initialisiert ein neues `GraphStorageEdge` Objekt. Der entsprechende `GraphStorageEdge.Builder` benötigt hierfür einen gültigen Kantentyp und die Referenzen der beiden Knoten, zwischen denen die Kante liegen soll. Solche

`GraphStorageEdgeReference` Objekte setzen sich aus dem zugehörigen Revisionsbereich `GraphStorageScope` und dem Pfad zum eigentlichen Knoten zusammen. Dabei entspricht der Pfad einem Array aus `int`-Werten, welche die Positionen der Knoten im Baum an der entsprechenden Tiefe angeben. Optional kann eine Kantenidentitätsnummer durch den Builder gesetzt werden, die eine bereits existierende Kante in der Graphdatenbank referenziert. Diese Nummer ist für das spätere Traversieren der Kanten zwingend notwendig und wird durch den Graphspeicher gesetzt. Abschließend kann der Builder das `GraphStorageEdge` Objekt erzeugen und dem Nutzer zur Verfügung stellen. Beim erstellen einer neuen Revision kann eine beliebige Anzahl solcher Objekte an den `GraphStorageManager` übergeben werden. Dieser Erstellt die Revision und setzt die übergebenen Kanten an den referenzierten Knoten. Dazu werden die referenzierten Vertices aus der Graphdatenbank geladen und eine neue Kante mit entsprechendem Label und Revisionsbereich des eingehenden Vertex gesetzt.

Listing 3.30: Funktion zum Erzeugen von nutzerspezifischen Kanten.

```
void doCreateLinks(Collection<GraphStorageEdge> edges)
{
    for (GraphStorageEdge edge : edges)
    {
        GraphStorageNodeRef outNodeRef = edge.getOutNodeRef();
        GraphStorageNodeRef inNodeRef = edge.getInNodeRef();

        Vertex outVertex = findVertexByReference(outNodeRef);
        Vertex inVertex = findVertexByReference(inNodeRef);

        if (fromVertex != null && toVertex != null)
        {
            GraphStorageScope inNodeScope = inNodeRef.getScope();
            long destinationArticleId = inNodeScope.getArticleId();
            long destinationRevisoinId = inNodeScope.getRevisionId();

            outVertex.addEdge(
                edge.getEdgeType().getLabel(),
                inVertex,
                ArticleId.get(), destinationArticleId,
                RevisionId.get(), destinationRevisionId;

            graph.tx().commit();
        }
    }
}
```

Standardmäßig wird das Traversieren nutzerspezifischer Kanten nicht unterstützt und erfordert eine Anpassung des bestehenden `GraphStorageTreeWalker`. Hierfür stellt der Graphspeicher den sogenannten `GraphStorageGraphWalker` bereit, der sämtliche Funktionen des Tree-Walker übernimmt und zusätzlich das Traversieren nutzerspezifischer Kanten anbietet. Im Detail wird der Walker um die beiden Funktionen `getCurrentEdges()` und `walkTo(GraphStorageEdge edge)` erweitert. Durch `getCurrentEdges()` können alle existenten Kanten des aktuellen Knotens abgerufen werden, wobei der Graphspeicher die `GraphStorageEdge` Objekte mit zugehöriger Kantennummer erzeugt und diese an den Aufrufer zurückgibt. Anschließend können über `walkTo(GraphStorageEdge edge)` die übergebenen Kanten verwendet und durchlaufen werden.

Listing 3.31: Abfragen aller vorhandenen nutzerspezifischen Kanten.

```
Set<GraphStorageEdge> getCurrentEdges()
{
    Set<GraphStorageEdge> edges = new HashSet<>();
    Iterator<Edge> edgeIterator = retrieveEdges(currentVertex);

    while (edgeIterator.hasNext())
    {
        Edge edge = edgeIterator.next();

        if (!edge.label().equals(CHILD_EDGE.getLabel())
                && !edge.label().equals(HISTORY_EDGE.getLabel()))
        {
            long destinationArticleId = edge.value(ArticleId.get());
            long destinationRevisionId = edge.value(RevisionId.get()):
            GraphStorageScope inScope = new GraphStorageScope(
                destinationArticleId, destinationRevisionId);

            edges.add(new GraphStorageEdge.Builder()
                .setEdgeType(edge.label())
                .setOutNodeReference(currentScope)
                .setInNodeReference(inScope)
                .setEdgeId(edge.id())
                .build());
    }
    return edges;
}
```

Damit eine nutzerspezifische Kante traversiert werden kann, muss das entsprechende Kantenobjekt über `getCurrentEdges()` angefordert werden. Nur dadurch ist sichergestellt, dass eine valide Kantennummer gesetzt ist. Ohne diese Nummer kann die gewünschte Kante nicht identifiziert und verwendet werden. Anschließend wird der eingehende Vertex über eine Query abgefragt und als aktueller Vertex im Walker gesetzt. Zusätzlich muss der Revisionsbereich zusammen mit dem aktuellen WOM-Knoten aktualisiert werden. Diese Informationen werden durch das Kantenobjekt und den aktuellen Vertex bereitgestellt. Abschließend wird die neue Position des Knotens im Bezug auf den Elternknoten ermittelt und ebenfalls aktualisiert. Nach erfolgreicher Traversierung befindet sich der `GraphStorageGraphWalker` auf dem neuen Knoten und kann wie gewohnt verwendet werden.

Listing 3.32: Folgen einer nutzerspezifischen Kante.

```
Node walkTo(GraphStorageEdge edge)
{
    if (edge.getEdgeId().isPresent())
    {
        currentVertex = graph.traversal().V(currentVertex).bothE()
                .hasId(edge.getEdgeId()).otherV().tryNext();
        currentScope = edge.getInNodeReference().getScope();
        currentNode = new ProxyNode(currentScope, currentVertex);
        currentPosition = getCurrentPosition();
    }
    return currentNode;
}
```

4 Evaluation

Ziel dieses Kapitels ist eine Evaluation der Umsetzung des graphbasierten Speichers hinsichtlich der Anforderungen aus Kapitel 1. Zusätzlich werden weitere wichtige Eigenschaften des Graphspeichers analysiert und bewertet. Hintergrund ist die Beantwortung der Frage ob die Verwaltung von Wikipedia-Seiten in einem graphbasierten Speicher möglich ist und welche Auswirkungen und Schwierigkeiten dabei auftreten. Des Weiteren werden verschiedene Tests und Analysen vorgestellt, die auf Leistungsfähigkeit und Schnelligkeit der Implementierung abzielen. Hier wird die Wirtschaftlichkeit in Frage gestellt und geprüft. Hinzu kommt ein Test der sämtliche Einsparungen im Bereich Vertex und Datenvolumen analysiert. Dabei wird untersucht ob durch die Verwendung des Algorithmus zur Graphoptimierung ein erkennbarer Mehrwert entsteht. Zuletzt behandelt das Kapitel Optimierungsvorschläge und Verbesserungen sowie Nachteile und Risiken des graphbasierten Speichers.

4.1 Bewertung hinsichtlich der Anforderungen

Im folgenden Abschnitt werden sämtliche technischen und funktionalen Anforderungen untersucht und mit der prototypischen Implementierung des graphbasierten Speichers verglichen. Als Einstieg sind alle technischen Anforderungen und Zielsetzungen zu betrachten. Dabei steht das verwendete WOM Format und dessen Datenvolumen im Vordergrund. Im Anschluss werden funktionale Aspekte und Anforderungen in den Vordergrund gerückt.

Das WOM-Format konnte problemlos in den Graphspeicher integriert werden. Die darunterliegende Datenbank bietet durch die interne Graphstruktur eine triviale Abbildung des WOM-Baums auf Datenbankebene. Die Struktur der WOM-Ressource bleibt im Speicher erhalten ohne dabei relationale Abbildungsstrategien zu verwenden. Im Vergleich zu herkömmlichen SQL Datenbanken wurde kein Mehraufwand betrieben um eine rekursive Baumstruktur in den Speicher zu schreiben. Darüber hinaus waren keine komplexen Frameworks oder APIs nötig, um eine gekapselte Persistenzschicht zu implementieren. Die Funktionen des Tin-

kerPop Frameworks bieten durch fluent Interfaces und intuitive Methodenaufrufe eine vollständige Schnittstelle zur Graphdatenbank und machen die Verwendung zusätzlicher Frameworks auf dem Gebiet überflüssig.

Hinsichtlich des enormen Datenvolumens aller Wikipedia-Seiten kann keine eindeutige Aussage darüber getroffen werden, ob Titan als Graphdatenbank diese gewaltige Menge stemmen kann. Die Ladezeit der gesamten Wikipedia würde das verfügbare Zeitkontingent übertreffen und konnte aus zeittechnischen Gründen nicht vollzogen werden. Bei einer realistischen Ladezeit von durchschnittlich 50 ms pro Seite würde das vollständige Laden der Wikipedia in den Graphspeicher mit über 800 Millionen Revisionen durch eine Instanz mehr als 462 Tage dauern. Aus diesem Grund kann nur ein kleiner Teil dieser Gesamtmenge in den Graphspeicher geladen und getestet werden. Einen weiteren limitierenden Faktor stellt das Auslesen der entsprechenden Seiten dar, was zu einer weitaus höheren Gesamtladezeit einzelner Revisionen führt. Um dennoch eine Aussage darüber zu treffen, ob der graphbasierter Speicher das Potential hat, die Menge der Wikipedia-Seiten zu stemmen, wird eine kleinere Teilmenge der Seiten in den Graphspeicher geladen. Werden die Limitierungen der Graphdatenbank betrachtet, sind potentiell 2^{59} Vertices mit 2^{60} Kanten möglich. Diese Menge würde bei 800 Millionen Revisionen mit durchschnittlich 2.841 Knoten und der daraus resultierenden Gesamtzahl von jeweils $\approx 2^{41}$ Knoten und Kanten völlig ausreichen. Selbst angesichts des enormen Zuwachses würde das Knotenlimit nicht in absehbarer Zeit überschritten werden.

Das verwendete Backend kann zum potentiellen Flaschenhals der Applikation werden. Mit Cassandra ist eine lineare Skalierbarkeit gegeben. Dadurch wären die Lastprobleme mit Erweiterung des Clusters gelöst. Auch die vielen populären Anwendungsfälle sprechen für Cassandra als geeignetes Backend für Titan. Die Frage wie sich Cassandra bei einer Knotenanzahl im Billionenbereich verhält wurde bisher nicht beantwortet. Dafür werden weiteren Tests benötigt, die nicht Teil dieser Arbeit sind.

4.2 Leistungsfähigkeit und Performanz

Der nachfolgende Abschnitt zielt auf eine Bewertung der Leistungsfähigkeit des graphbasierten Speichers anhand verschiedener Tests ab. Zu Bewerten sind Lese- und Schreibgeschwindigkeit in unterschiedlichen Konfigurationen des Graphspeichers. Die Performanz spielt eine wichtige Rolle im Bezug auf Wirtschaftlichkeit und Benutzbarkeit im Live-Betrieb. Die Wikipedia stellt Unmengen an Informationen in kürzester Zeit bereit ohne dabei auf lange Wartezeiten zu setzten. Diese Eigenschaften müssen durch den Graphspeicher gewährleistet und in folgenden Tests bewertet werden.

Im ersten Versuch wurden 45.305 Revisionen in den graphbasierten Speicher geladen. Dafür wurde ein Cassandra Cluster mit drei Knoten aufgesetzt und als Titan-Backend verwendet. Sämtliche Daten stammen aus den ersten zwei Dateien von insgesamt 27 des offiziellen Wikipedia-Dumps. Der Graphspeicher wurde ohne Nebenläufigkeit und Begrenzung der Baumtiefe konfiguriert, dass bedeutet schreibende Anfragen arbeiten auf genau einem Thread ohne weitere Begrenzung der Baumtiefe. In Tabelle 4.1 werden die wichtigsten Kenngrößen des Versuchs aufgelistet.

Tabelle 4.1: Kenngrößen aus Versuch 1.

	Avg	**Median**	**Max**	**Min**
Knoten	2858	2083	92.817	2
Vertices	69	13	92.314	2
Schreibgeschwindigkeit in ms	137	99	21.209	7
Lesegeschwindigkeit in ms	19.472	14.414	567.884	19

Besonders Auffällig ist die extrem hohe Zeit beim Lesen von Ressourcen. In Abbildung 4.1 werden Knotenanzahl und Lesegeschwindigkeit gegenübergestellt. Dabei ist ein linearer Anstieg der Lesezeit bei steigender Anzahl an Knoten zu erkennen. Bereits bei einer Knotenanzahl von über 5.000 beträgt die Zeit zum Lesen einer einzigen Ressource mehr als 50 Sekunden. Diese Zeit ist in keinem Fall mit der Anwendung vereinbar und muss im weiteren Verlauf optimiert.

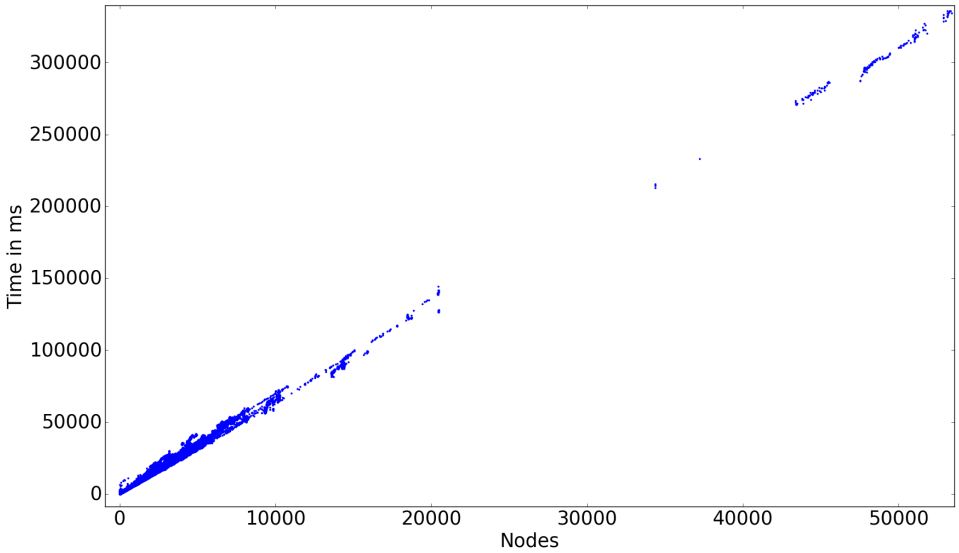

Abbildung 4.1: Lesegeschwindigkeit im Verhältnis zur Knotenanzahl.

Werden die Zeiten zum Einfügen einer Ressource mit Anzahl der Knoten verglichen, zeigt sich in Abbildung 4.2 ein ganz anderes Bild. Im Gegensatz zur Lesegeschwindigkeit ist kein linearer Anstieg zu beobachten. Die Knotenanzahl spielt demnach keine essentielle Rolle beim Einfügen von Ressourcen. Wird die Anzahl der Vertices verglichen, wird deutlich, dass selbst bei großen Ressourcen nur eine geringe Anzahl an Vertices erzeugt wird. Diese Beobachten bestätigen die These, dass oft nur kleine Änderungen zwischen einzelnen Ressourcen liegen. Dadurch müssen durch die Graphoptimierung deutlich weniger Vertices eingefügt werden. Das erklärt den großen Unterschied zwischen Schreib- und Lesezugriffe.

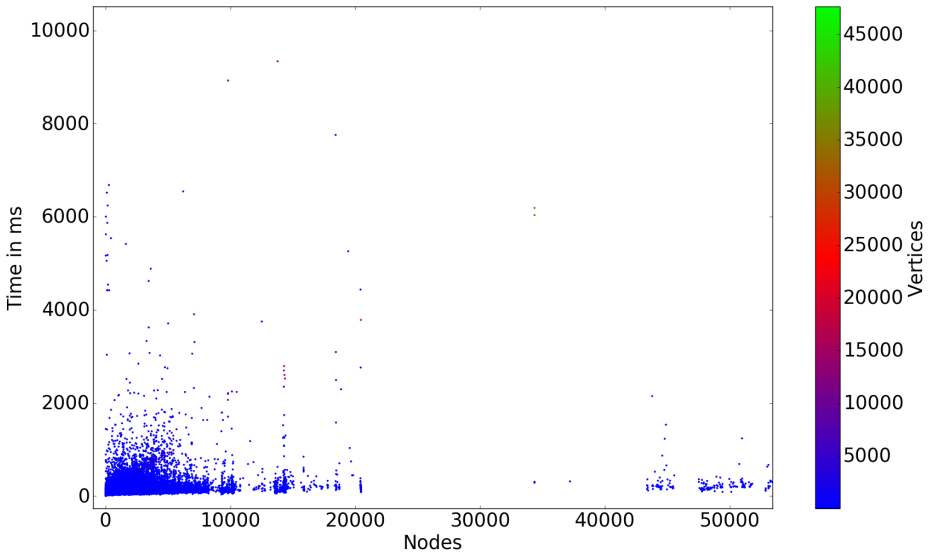

Abbildung 4.2: Schreibgeschwindigkeit im Verhältnis zur Knotenanzahl.

Tabelle 4.2: Kenngrößen aus Versuch 2.

	Avg	**Median**	**Max**	**Min**
Knoten	3902	2733	92.817	2
Vertices	41	12	21.340	2
Schreibgeschwindigkeit in ms	39	28	27.304	3
Lesegeschwindigkeit in ms	4.245	3.047	86.541	9

Die Diagramme verdeutlichen die Problematik der Lesegeschwindigkeit. Aus diesem Grund werden im nächsten Experiment Optimierungsmaßnahmen durchgeführt. Dabei sind 86755 Revisionen in den graphbasierten Speicher eingefügt worden. Zusätzlich wurden alle p-Knoten limitiert. Dadurch entstehen keine neuen Vertices für darunterliegende Knoten. Es wird eine Steigerung der Lese- und Schreibgeschwindigkeit erwartet. Sämtliche Kenngrößen des Versuchs können aus Tabelle 4.2 entnommen werden.

Als Ergebnis ist eine deutliche Verbesserung der durchschnittlichen Schreib- und Lesegeschwindigkeit zu beobachten. Im Vergleich zum vorherigen Versuch werden Anfragen fast um ein vierfaches schneller beantwortet. Das Diagramm aus Abbildung 4.3 lässt ebenfalls auf eine linearen Anstieg der Lesezeit schließen und zeigt eine deutliche Steigerung der Lesegeschwindigkeit.

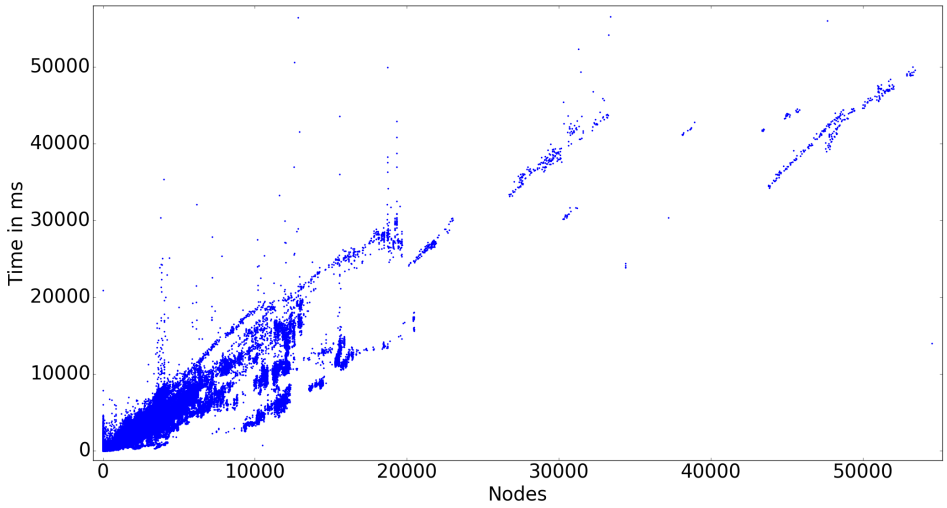

Abbildung 4.3: Lesegeschwindigkeit im Verhältnis zur Knotenanzahl.

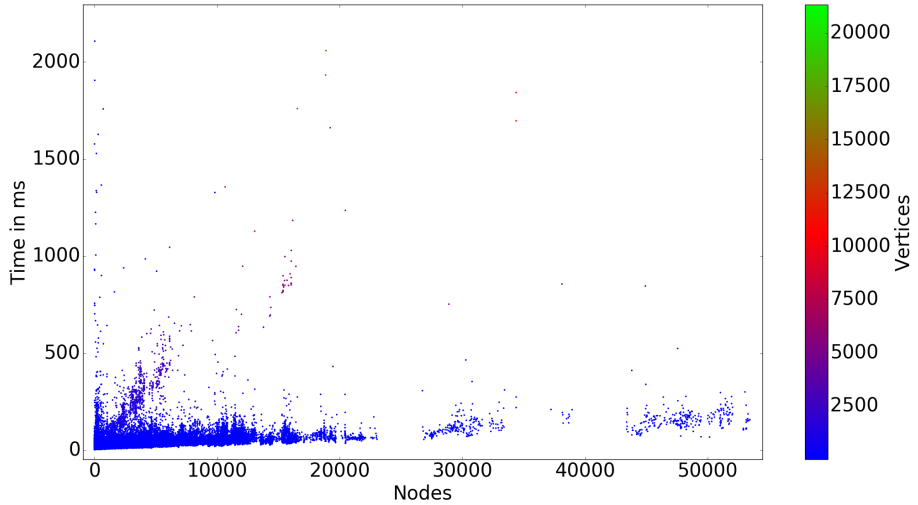

Abbildung 4.4: Schreibgeschwindigkeit im Verhältnis zur Knotenanzahl.

Auch bei Betrachtung der Schreibgeschwindigkeit in Abbildung 4.4 ist eine deutliche Verbesserung zu erkennen. Der Großteil aller Einfügeoperationen liegt im Bereich von unter 500 Millisekunden, was im Bezug auf die gestellten Anfor-

derungen ausreichend ist. Selbst Ressourcen mit über 50.000 Knoten können in weniger als 1 Sekunde in die Datenbank eingefügt werden. Dafür müssen weniger als 5.000 Vertices in der Graphdatenbank erstellt werden.

Hauptproblem des graphbasierten Speichers sind die extremen Lesezeiten. Eine weitere Möglichkeit zur Optimierung bietet Nebenläufigkeit. Dafür wird die Funktion aus Abschnitt 3.2 parallelisiert. Über einen Threadpool werden mehrere Threads gestartet, die jeweils Kind-Knoten aus der Graphdatenbank holen. Zusätzlich wird die Tiefenbegrenzung auf p-Knoten beibehalten. Bei diesem Versuch wurden 106.478 Revisionen verarbeitet und in den Graphspeicher eingefügt. Tabelle 4.3 beinhaltet sämtliche Ergebnisse dieses Vorgangs. Über Parallelisierung konnten wiederum bessere Ergebnisse beim Lesen von Ressourcen erzielt werden. Das Diagramm in Abbildung 4.5 zeigt einen leichten linearen Anstieg der Lesezeit. Nichtsdestotrotz liegt die Lesezeit in einem Bereich, der für die gedachte Anwendungen völlig unbrauchbar ist.

Tabelle 4.3: Kenngrößen aus Versuch 3.

	Avg	**Median**	**Max**	**Min**
Knoten	3275	2134	92.817	2
Vertices	55	13	42.006	2
Schreibgeschwindigkeit in ms	135	100	55.827	4
Lesegeschwindigkeit in ms	1.193	719	24.730	9

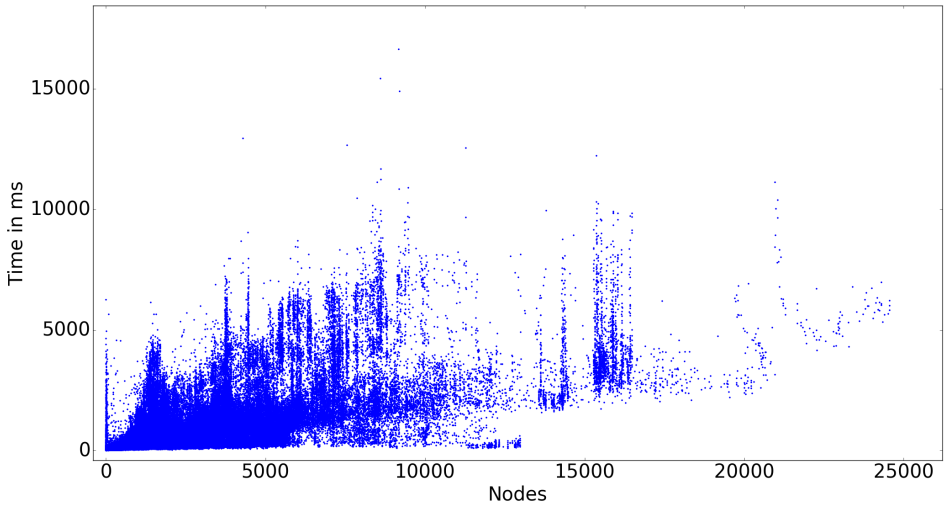

Abbildung 4.5: Lesegeschwindigkeit im Verhältnis zur Knotenanzahl.

4.3 Speicherplatzeinsparungen

Bisher wurden ausschließlich Kennzahlen und Diagramme bezüglich der Leistungsfähigkeit des graphbasierten Speichers betrachtet. Im weiteren Verlauf wird durch einen weiteren Test die Speicherplatzeinsparungen untersucht, die durch die Graphoptimierung entstanden sind. Ohne den Algorithmus zur Graphoptimierung würde die Anzahl der Vertices der Knotenanzahl entsprechen. Das Diagramm in Abbildung 4.6 zeigt einen Versuch mit 7.724 Revisionen. Dabei wurde die Gesamtgröße der textuellen Inhalte jeder Ressource untersucht, die verarbeitet wurde. Die Ergebnisse aus Tabelle 4.4 zeigen einen potentiellen Speicherplatzbedarf von durchschnittlich 63,3 KB. Durch die Graphoptimierung wurde dieser Bedarf auf 6,3 KB reduziert. Das entspricht einer Speicherplatzreduktion von über 90 Prozent.

Tabelle 4.4: Kenngrößen aus Versuch 4.

	Avg	**Median**	**Max**	**Min**
Knoten	5.902	6.366	13.244	2
Vertices	65	15	4.604	2
Artikelinhalt in Bytes	63.349	67.997	151.756	0
Graphinhalt in Bytes	6.343	3.783	88.833	0
Schreibgeschwindigkeit in ms	45	41	6.294	5
Lesegeschwindigkeit in ms	1.908	1.973	5.493	11

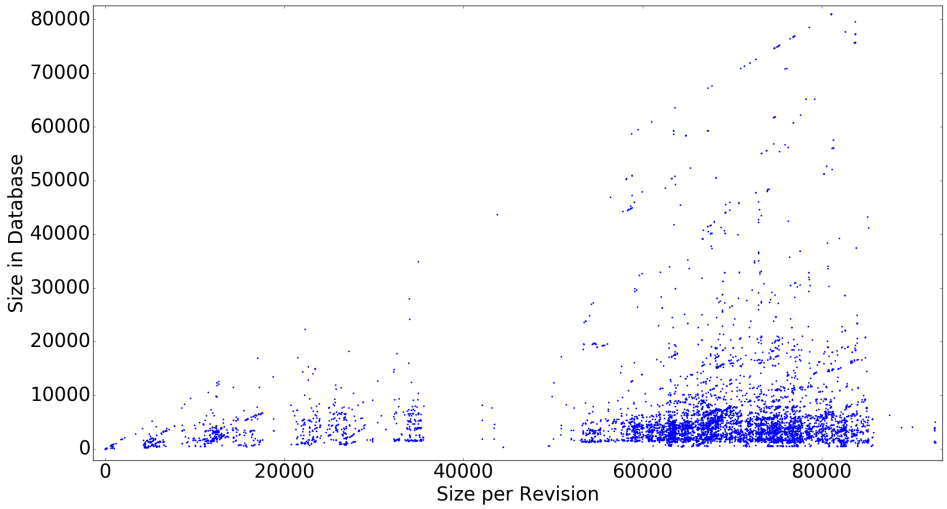

Abbildung 4.6: Originale Artikelgröße im Verhältnis zum echten Graphinhalt.

Die Evaluation hat ein großes Problem des graphbasierten Speichers aufgezeigt. Die Problematik liegt beim Laden von Ressourcen aus der Graphdatenbank. Selbst mit allen Optimierungen konnte keine angemessene Lesegeschwindigkeit erzeugt werden. Grund für diesen Leistungseinbruch ist das Abfragen der Vertices. Es konnte eindeutig beobachtet werden, dass mit sinkender Vertexanzahl die Lesegeschwindigkeit schneller wird. Zusätzlich konnte diese Geschwindigkeit durch Parallelisierung der Abfragen gesteigert werden. Nichtsdestotrotz sind weitere Maßnahmen nötig, um brauchbare Ergebnisse zu erzielen. Vorteile des graphbasierten Speichers sind die Speicherplatz- und Vertexeinsparungen. Die Evaluation hat gezeigt, dass durch die Graphoptimierung der Speicherplatzbedarf deutlich sinkt. Des Weiteren liegt die Schreibgeschwindigkeit in einem Bereich, der toleriert werden kann. Allerdings stößt der graphbasierte Speicher an die Limitierungen der Graphdatenbank, weil eine deutliche Verbesserung der Lesegeschwindigkeit unter Titan bisher nicht möglich ist. Unabhängig davon können die Konzepte dieser Arbeit auf sämtliche graphbasierten Systeme angewendet werden. Aus diesem Grund müssen klare Grenzen zwischen der internen Graphdatenbank und dem graphbasierten Speicher gesetzt werden. Zwar können eine Vielzahl von Optimierungen vorgenommen werden, dennoch ist die Anwendung von der Leistungsfähigkeit der Graphdatenbank abhängig. Als Fazit der Evaluation hat der graphbasierte Speicher das Potential Wikipedia-Artikel zu verwalten, jedoch hängt die Leistungsfähigkeit stark von der internen Datenbank ab.

5 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde die Konzeption und prototypische Umsetzung eines graphbasierten Speichers für Wikipedia-Artikel vorgestellt. Es sind Design- und Architekturentscheidungen mit Hilfe gestellter Anforderungen getroffen und im weiteren Verlauf realisiert worden. Die Hauptprobleme und größten technischen Herausforderungen der Arbeit betreffen das Verwalten und Laden von Wikipedia-Artikeln in den graphbasierten Speicher. Dabei spielt das Abbilden der Wikipedia-Artikel auf die interne Graphdatenbank eine wichtige Rolle. Jede Wikipedia Seite liegt im WOM-Format vor und beschreibt eine Baumstruktur aus WOM-Knoten mit Eltern-Kind-Beziehungen zwischen den einzelnen Knotenelementen. Dieser WOM-Baum entspricht einem Graphen, der durch die interne Graphdatenbank Titan verwaltet wird. Die Abbildung solcher Baumstrukturen auf einen Graphen bestehend aus Vertices und Kanten ist eine der Aufgaben, die der graphbasierte Speicher übernehmen muss. Als effizienteste und schnellste Strategie hat sich das binäre Serialisieren der WOM-Knoten in ein einziges Vertex-Property durchgesetzt. Dadurch konnten bessere Ergebnis erzielt werden als beispielsweise beim Abbilden einzelner Knotenfelder auf mehrere Properties. Um die ursprüngliche Baumstruktur zu sichern, wurden spezielle Kantentypen zur Modellierung von Eltern-Kind-Beziehungen verwendet. Weitere Maßnahmen zu Abbildung von WOM-Ressourcen waren durch die Verwendung einer Graphdatenbank hinfällig.

Ein weiterer Inhalt dieser Arbeit war das sprunghafte Traversieren von WOM-Ressourcen in der Graphdatenbank. Damit der Graph Schritt für Schritt durchlaufen werden kann, wurde der sogenannte Tree-Walker implementiert. Der Walker startet am Wurzelknoten des Graphen und bietet Möglichkeiten zur Traversierung nächstgelegener Knoten an. Die entsprechenden Vertices werden nur bei expliziten Anfragen aus der Graphdatenbank geladen. Als Resultat erhält der Nutzer Proxyobjekte, die wiederum verwendet werden können um weitere Knoten der Ressource abzufragen. Dafür leiten Proxy-Knoten entsprechende Anfragen direkt an die Graphdatenbank weiter. Zusätzlich bietet der Tree-Walker das Filtern von Knoten nach selbst definierten Kriterien an. Dadurch eröffnen sich

Möglichkeiten zur Analyse und Verarbeitung von WOM-Ressourcen innerhalb der Datenbank.

Im weiteren Teil der Arbeit stand die umfangreiche Revisionshistorie der Wikipedia im Vordergrund. Für jede Seite der Wikipedia können mehrere Versionen in der Datenbank existieren. Diese Revisionen spiegeln die Historie einer Seite wider und sind zeitlich voneinander abhängig. In der Realität können für eine einzige Wikipedia-Seite beliebig viele Revisionen entstehen, die in vielen Fällen das Limit von tausend überschreiten. Oft liegen nur kleine Änderungen zwischen einzelnen Revisionen, was zu einer hohen Redundanz an Daten führt. Ein Ziel dieser Arbeit war die Reduzierung solcher Redundanzen und das effiziente Abspeichern der Revisionen. Hierfür wurde ein Algorithmus vorgestellt, welcher mit Hilfe von speziellen Kanten Teile des Graphen der Vorgänger-Revision wiederverwendet um den Graphen der Nachfolger-Revision zu erzeugen. Diese Strategie war durch die Verwendung des HDDiff-Algorithmus möglich. Durch den daraus resultierenden Änderungsbaum konnten Redundanzen erkannt und reduziert werden. Allerdings war die vollständige Eliminierung von Redundanzen benachbarter Revisionen nicht möglich, weil dadurch im Vergleich zum eigentlichen Nutzen ein deutlicher Mehraufwand entstehen würde.

Im Anschluss darauf wurden nutzerspezifische Kanten im graphbasierten Speicher etabliert. Dadurch ist es möglich, eigene Kantentypen zu definieren und diese im Graphspeicher zu verwenden. Die Traversierung dieser Kanten wurde über spezielle Funktionen des erweiterten Walkers angeboten und konnte ausschließlich durch diesen vollzogen werden. Nutzerspezifische Kantentypen können zwischen zwei beliebige Knoten gesetzt werden. Dabei spielt die Seite oder Revision der jeweiligen Knoten keine Rolle.

Im Mittelpunkt der darauffolgenden Evaluation war die mangelhafte Lesegeschwindigkeit. Dabei wurden Optimierungen auf Graph und Datenbankebene vorgenommen, um eine brauchbare Lesezeit zu bekommen. Leider konnte selbst durch die Reduzierung der Vertices und neben-läufigen Abfragen keine optimalen Ergebnisse erzielt werden. Im Vergleich zur ursprünglichen Zeit konnten erhebliche Verbesserungen beobachtet werden. Jedoch befindet sich die endgültige Zeit in einem Bereich der für Verwaltungsaufgaben zu hoch ist. Allerdings konnten erhebliche Speicherplatz-Einsparungen als Nebeneffekt der Graphoptimierung erzielt werden. Selbst die Schreibgeschwindigkeit befindet sich in einem Bereich des machbaren. Letztendlich müssen weitere Wege oder Datenbanken gefunden werden, um eine entsprechende Lösung zu finden.

Mit Hinblick auf zukünftige Arbeiten und Verbesserungen des graphbasierten Speichers für Wikipedia-Artikel können einige Punkte aufgelistet werden. Ein großes Thema ist die Verbesserung der Lesezeit von WOM-Ressourcen. Der aktuelle Vorgang kann durch Parallelisierung weiter verbessert werden und beinhaltet großes Verbesserungspotential. Auch die fehlenden Funktionalitäten der internen

Graphdatenbank bei großen Datenbankanfragen lassen Platz nach oben. Hier wäre das Testen weiterer Datenbanken der nächste Schritt. Ein weiterer Punkt sind die mangelnden Informationen über Leistungsfähigkeit und Lastverteilung bei Datenmengen im Größenbereich der englischen Wikipedia. Hier muss eine Möglichkeit gefunden werden, um die enorme Menge an Daten in absehbarer Zeit in den graphbasierten Speicher zu laden. Auch wenn die aktuelle Zeit zum Schreiben von Revisionen für den Live-Betrieb völlig ausreichend ist, würde das Einfügen der vollständigen Wikipedia mehrere Monate dauern. Als letzten Punkt kann der graphbasierte Speicher als Service bereitgestellt werden, der beispielsweise über eine REST Schnittstelle nach Fielding (2000) sämtliche Ressourcen bereitstellt. Für diese Erweiterungen wurden bereits Maßnahmen und Designentscheidungen getroffen, die nicht Teil dieser Arbeit sind. Im Grunde wäre die Umsetzung einer Mikroservice Architektur nach Newman (2015) möglich, mit verteilten Instanzen des Graphspeichers. Der Blick in die Zukunft lässt viele Möglichkeiten offen um das aktuelle Design zu erweitern. Mit der gegenwärtigen Architektur des graphbasierten Speichers für Wikipedia-Artikel sind fast keine Grenzen gesetzt.

Literaturverzeichnis

Beis, S., Papadopoulos, S., & Kompatsiaris, Y. (2015). Benchmarking Graph Databases on the Problem of Community Detection. In N. Bassiliades, M. Ivanovic, M. Kon-Popovska, Y. Manolopoulos, T. Palpanas, G. Trajcevski, & A. Vakali (Hrsg.), *New Trends in Database and Information Systems II: Selected papers of the 18th East European Conference on Advances in Databases and Information Systems and Associated Satellite Events, ADBIS 2014 Ohrid, Macedonia, September 7-10, 2014 Proceedings II* (S. 3–14). Cham: Springer International Publishing. doi:10.1007/978-3-319-10518-5_1

Brewer, E. A. (2000). Towards Robust Distributed Systems (Abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing* (S. 7–). PODC '00. Portland, Oregon, USA: ACM. doi:10.1145/343477.343502

Chairunnanda, P., Forsyth, S., & Daudjee, K. (2012). Graph Data Partition Models for Online Social Networks. In *Proceedings of the 23rd ACM Conference on Hypertext and Social Media* (S. 175–180). HT '12. Milwaukee, Wisconsin, USA: ACM. doi:10.1145/2309996.2310026

Dohrn, H. & Riehle, D. (2011). Design and Implementation of the Sweble Wikitext Parser: Unlocking the Structured Data of Wikipedia. In *Proceedings of the 7th International Symposium on Wikis and Open Collaboration* (S. 72–81). WikiSym '11. Mountain View, California: ACM. doi:10.1145/2038558.2038571

Dohrn, H. & Riehle, D. (2013). Design and Implementation of Wiki Content Transformations and Refactorings. In *Proceedings of the 9th International Symposium on Open Collaboration* (2:1–2:10). WikiSym '13. Hong Kong, China: ACM. doi:10.1145/2491055.2491057

Dohrn, H. & Riehle, D. (2014). Fine-grained Change Detection in Structured Text Documents. In *Proceedings of the 2014 ACM Symposium on Document Engineering* (S. 87–96). DocEng '14. Fort Collins, Colorado, USA: ACM. doi:10.1145/2644866.2644880

Fielding, R. (2000). *REST: Architectural Styles and the Design of Network-based Software Architectures* (Doctoral dissertation, University of California, Irvine).

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software (Adobe Reader)*. Pearson Education. Zugriff unter https://books.google.de/books?id=6oHuKQe3TjQC

Jouili, S. & Vansteenberghe, V. (2013 September). An Empirical Comparison of Graph Databases. In *Social Computing (SocialCom), 2013 International Conference on* (S. 708–715). doi:10.1109/SocialCom.2013.106

Martin, R. (2013). *Clean Code - Refactoring, Patterns, Testen und Techniken für sauberen Code: Deutsche Ausgabe*. mitp Professional. mitp/bhv. Zugriff unter https://books.google.de/books?id=zJZbAgAAQBAJ

Newman, S. (2015). *Building Microservices*. O'Reilly Media. Zugriff unter https://books.google.de/books?id=RDl4BgAAQBAJ

Rodriguez, M. A. (2015). The Gremlin Graph Traversal Machine and Language (Invited Talk). In *Proceedings of the 15th Symposium on Database Programming Languages* (S. 1–10). DBPL 2015. Pittsburgh, PA, USA: ACM. doi:10.1145/2815072.2815073