Generalisierung der Persistenzschicht im semantischen Daten-Wiki OntoWiki

Diplomarbeit

Matthias Molitor

Ettighofferstr. 15, 53123 Bonn
molitor@cs.uni-bonn.de
Matrikelnummer 1856389

Bonn, den 7. Juli 2014

Rheinische Friedrich-Wilhelms-Universität Bonn
Informatik – Enterprise Information Systems
Professor Dr. Sören Auer

universitätbonn

Erklärung

Hiermit erkläre ich, diese Diplomarbeit selbständig durchgeführt zu haben. Alle Quellen und Hilfsmittel, die ich verwendet habe, sind angegeben. Zitate habe ich als solche kenntlich gemacht.

Bonn, den 7. Juli 2014

Matthias Molitor

Danksagung

Mein Dank gilt zuerst meinem Betreuer Prof. Dr. Sören Auer, der mir den nötigen Freiraum zur Umsetzung der Arbeit gegeben hat, aber bei auftretenden Problemen immer Rat wusste und motivierende Worte fand.

Meinem Kommilitonen Bernd Schönbach danke ich dafür, dass er mit mir während der Entstehung der Arbeit nahezu jedes Wochenende in den Räumlichkeiten der Fachschaft eine Bürogemeinschaft gebildet hat.

Des Weiteren danke ich meinem Korrekturleser Malte Wunsch, der diese Arbeit mit seinen zahllosen Verbesserungsvorschlägen und Anmerkungen bereichert hat.

Besonderen Dank verdient meine Frau Sara, die mich in den vergangenen Monaten noch mehr als sonst entlastet und unterstützt hat. Ohne sie wäre diese Arbeit nicht möglich gewesen.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Zielsetzung . . . 2
- 1.3 Aufbau der Arbeit . . . 2

2 Grundlagen **5**
- 2.1 Semantic Web . . . 5
  - 2.1.1 Resource Description Framework . . . 6
  - 2.1.2 Abfrage von Daten mit SPARQL . . . 9
- 2.2 Softwareentwicklung . . . 12
  - 2.2.1 Design Patterns . . . 12
  - 2.2.2 Test Driven Development . . . 13
  - 2.2.3 PHP . . . 13
- 2.3 OntoWiki . . . 14
- 2.4 Erfurt Framework . . . 15
- 2.5 Verwandte Arbeiten . . . 15

3 Anforderungserhebung **17**
- 3.1 Szenarien . . . 17
  - 3.1.1 Erstellen einer Website . . . 17
  - 3.1.2 Datenpflege in Großunternehmen . . . 18
  - 3.1.3 Frontend für RDF-Datenspeicher . . . 18
  - 3.1.4 Wechsel des Datenspeichers nach Testphase . . . 18
  - 3.1.5 Anbindung eines neuen Datenspeichers . . . 18
- 3.2 Probleme . . . 19
  - 3.2.1 Vermeidung von Verzögerungen . . . 19
  - 3.2.2 Verknüpfung von Daten . . . 19
  - 3.2.3 Zugriffsschutz . . . 20
  - 3.2.4 Integration in bestehende Systemlandschaften . . . 20
  - 3.2.5 Initiale Inbetriebnahme . . . 20
  - 3.2.6 Migration . . . 20
  - 3.2.7 Anpassungsfähigkeit . . . 21
- 3.3 Anforderungen . . . 21

4 Analyse des existierenden Systems und Erweiterung der Architektur 25
4.1 Analyse des existierenden Systems . . . 25
4.1.1 Komponenten . . . 25
4.1.2 Datenspeicher-Abstraktion . . . 27
4.1.3 Existierende Adapter . . . 29
4.1.4 Automatisierte Tests . . . 31
4.1.5 Identifizierte Probleme . . . 32
4.2 Erweiterung der Architektur . . . 34
4.2.1 Grundkonzept für die Erweiterung . . . 34
4.2.2 Infrastruktur für die Anbindung von RDF-Datenspeichern . . . 39

5 Anbindung neuer Datenspeicher 45
5.1 Oracle . . . 45
5.1.1 Einordnung . . . 45
5.1.2 Funktionsweise . . . 46
5.1.3 Anbindung . . . 48
5.1.4 Probleme bei der Umsetzung . . . 51
5.1.5 Optimierung der Performanz . . . 57
5.2 Stardog . . . 65
5.2.1 Einordnung . . . 65
5.2.2 Funktionsweise . . . 66
5.2.3 Anbindung . . . 66
5.2.4 Probleme bei der Umsetzung . . . 67
5.2.5 Optimierung der Performanz . . . 68
5.3 Neo4j . . . 71
5.3.1 Einordnung . . . 71
5.3.2 Funktionsweise . . . 71
5.3.3 Anbindung . . . 72
5.3.4 Probleme bei der Umsetzung . . . 73

6 Vergleich der unterstützten Datenspeicher 75
6.1 Vergleich ausgewählter Eigenschaften . . . 75
6.1.1 Open Source . . . 76
6.1.2 Native RDF-Unterstützung . . . 76
6.1.3 Installationsaufwand . . . 76
6.1.4 SPARQL Update . . . 77
6.1.5 SPARQL ASK . . . 77
6.1.6 Reasoning . . . 77
6.2 Vergleich der Performanz . . . 77
6.2.1 Zielsetzung . . . 78
6.2.2 Datenbasis . . . 79
6.2.3 Auswahl der SPARQL-Abfragen . . . 79

6.2.4 Setup . . . 80
6.2.5 Ablauf des Benchmarks . . . 81
6.2.6 Ergebnisse . . . 82
6.2.7 Fazit . . . 87

7 Evaluation der Anforderungsumsetzung 89
7.1 Funktionale Anforderungen . . . 89
7.2 Nicht-Funktionale Anforderungen . . . 89
7.2.1 Korrektheit . . . 90
7.2.2 Abwärtskompatibilität . . . 90
7.2.3 Plattformunabhängigkeit . . . 90
7.2.4 Wartbarkeit . . . 90
7.2.5 Erweiterbarkeit . . . 91
7.2.6 Performanz . . . 92
7.2.7 Transparente Installation . . . 93
7.3 Übersicht . . . 93

8 Zusammenfassung und Ausblick 95
8.1 Zusammenfassung . . . 95
8.2 Ausblick . . . 96
8.2.1 Abhängigkeit zu relationaler Datenbank auflösen . . . 96
8.2.2 Verbesserung des SPARQL-Parsers . . . 96
8.2.3 Extraktion weiterer Bibliotheken aus Erfurt . . . 97
8.2.4 Unterstützung von Datei-Importen . . . 98

Quellenverzeichnis 101

Abbildungsverzeichnis 107

Tabellenverzeichnis 109

Listingverzeichnis 111

A Daten der DBpedia-Benchmarks 113

B Inhalt der DVDs 129

Kapitel 1

Einleitung

1.1 Motivation

Semantische Daten haben in den vergangenen Jahren verstärkt Einzug in unseren Alltag gehalten und an Bedeutung gewonnen. Im Gegensatz zu den im Internet noch vorherrschenden Textdaten zielen diese verknüpften Daten (englisch: Linked Data) darauf ab, Maschinen in gewissem Maße einen Zugang zur Semantik der angebotenen Informationen zu gewähren. Heute liefern Webseiten mit Microdata annotierten HTML-Code aus [Hic13], um Suchmaschinen einen einfacheren Zugang zu den behandelten Themen und Konzepten zu bieten. Der Supercomputer Watson [FBC+10] verwendet große Mengen semantischer Daten als Teil seiner Wissensbasis. Er trat in der Quizshow Jeopardy gegen menschliche Gegner an und konnte den Sieg für sich verbuchen. Ein weiteres, bekannteres Beispiel für die Nutzung semantischer Daten ist die auf Apple iPhones installierte Sprachassistenz-Software Siri [QS13].

Angesichts der Vorteile, welche die Nutzung maschinenlesbarer Daten bietet, scheint es verwunderlich, dass ein Großteil der Informationen im Netz, beispielsweise die Datenmassen von Wikipedia, noch immer primär in reiner Textform vorliegt. Um die Arbeit mit semantischen Daten zu erleichtern, hat die Forschungsgruppe Agile Knowledge Engineering and Semantic Web (AKSW) der Universität Leipzig OntoWiki entwickelt.

Wie traditionelle Wikis ermöglicht OntoWiki das kollaborative Bearbeiten von Informationen, verwaltet diese aber in Form von maschinenlesbaren, semantischen Daten. Die webbasierte Oberfläche bietet sinnvolle Eingabehilfen und vermeidet, dass der Nutzer sich mit den Details des Datenmodells auseinandersetzen muss. Eine Weiterentwicklung dieses Wiki-Systems bietet die Chance, zusätzliche Nutzer zu gewinnen und die Verwendung strukturierter Daten zu fördern.

1.2 Zielsetzung

Das erste Ziel der vorliegenden Diplomarbeit ist die Anbindung weiterer Datenbanksysteme zum Persistieren von OntoWiki-Daten. Diese Systeme sollen verschiedene Einsatzgebiete abdecken. Ist dieses Ziel erreicht, sollen OntoWiki-Administratoren bei ihrer Entscheidung über das in ihrer OntoWiki-Instanz zu verwendende Datenbanksystem unterstützt werden.

Zur Erfüllung dieser Ziele sind drei Aufgaben zu bearbeiten:

- Die Generalisierung der Persistenzschicht als Voraussetzung zur Anbindung weiterer Datenbanksysteme
- Die konkrete Anbindung weiterer Datenbanksysteme
- Der Vergleich der angebundenen Systeme zur Beschreibung ihrer Vor- und Nachteile

1.3 Aufbau der Arbeit

Die vorliegende Arbeit ist wie folgt strukturiert: In Kapitel 2 werden Grundlagen des Semantic Web und der Softwareentwicklung behandelt. Außerdem wird OntoWiki, das im Zuge dieser dieser Arbeit erweitert werden soll, kurz vorgestellt.

Kapitel 3 zeigt mögliche Einsatzszenarien für das Wiki und ermittelt daraus resultierende Problemstellungen. Aus diesen werden schließlich Anforderungen entwickelt, welche die Grundlage für Design und Implementierung bilden.

In Kapitel 4 wird zunächst die Software-Architektur von OntoWiki analysiert. Anschließend wird ein Entwurf zur Generalisierung der Persistenzschicht vorgestellt, der sich in das bestehende System einfügt.

Kapitel 5 behandelt die konkrete Erweiterung von OntoWiki um Schnittstellen zu Oracle[1], Stardog[2] und Neo4j[3]. Es wird auf die internen Funktionsweisen dieser Systeme, deren Anbindung an das Wiki und festgestellte Probleme sowie Maßnahmen zur Verbesserung der Leistung eingegangen.

Das Kapitel 6 beschäftigt sich mit dem Vergleich der bereits vorhandenen sowie der neu angebundenen Systeme. Dieser Vergleich umfasst sowohl deren Eigenschaften als auch deren Performanz. Die Performanz wird per Benchmark ermittelt und anschließend diskutiert.

In Kapitel 7 wird geklärt, welche der in Kapitel 3 definierten Anforderungen umgesetzt werden konnten.

Die Arbeit wird mit Kapitel 8 geschlossen, indem nach einer Zusammenfassung ein Ausblick auf zukünftig mögliche Vertiefungen und Erweiterungen gegeben wird.

[1] http://www.oracle.com/us/products/database/overview/index.html
[2] http://stardog.com/
[3] http://www.neo4j.org/

Der Anhang enthält die in den Benchmarks ermittelten Daten sowie Informationen zu den auf DVD mitgelieferten Komponenten, die während der Entwicklung verwendet wurden.

Kapitel 2

Grundlagen

Im Folgenden werden die zum Verständnis dieser Arbeit notwendigen Grundlagen erläutert. Diese umfassen das Semantic Web mit seinen angrenzenden Technologien, ausgewählte Themen der Softwareentwicklung und einen Überblick über OntoWiki. Abschließend werden verwandte Arbeiten und Projekte vorgestellt.

2.1 Semantic Web

Das Semantic Web wird als eine Erweiterung des World Wide Web gesehen, das sich zum Ziel setzt Informationen maschinenlesbar zur Verfügung zu stellen [BHL01]. Im Gegensatz zum Forschungsfeld der künstlichen Intelligenz ist es nicht Ziel des Semantic Web, Maschinen ein echtes Verständnis der Inhalte zu ermöglichen. Stattdessen sollen Maschinen in die Lage versetzt werden Informationen zu verwenden, d. h. sie müssen diese auslesen und Schlüsse daraus ziehen können [HKR+06].

Damit Maschinen die Informationen verwenden können, müssen diese manuell aufbereitet und strukturiert zur Verfügung gestellt werden. Um Aussagen (englisch: statements) maschinenlesbar formulieren zu können, wurden vom World Wide Web Consortium[1] (W3C) einheitliche, offene Standards geschaffen. Mit Hilfe des Resource Description Frameworks (RDF) können Aussagen ausgedrückt werden. Regeln innerhalb von Wissensdomänen, welche die Grundlage zum Ableiten von Informationen darstellen, lassen sich mit RDF-Schema oder OWL (Web Ontology Language) beschreiben [BGM14; HKR+06; Owl12; SRM+14].

Durch die Aufbereitung der Daten wird den Computern deren Semantik zugänglich. Dies verhindert unter anderem Probleme mit Mehrdeutigkeiten, die bei der Verwendung der natürlichen Sprache entstehen [HKR+06]. So muss aus einem Text in natürlicher Sprache beispielsweise nicht unmittelbar klar werden, ob mit dem Homonym „Bank“ ein Platz zum Sitzen oder ein

[1] http://www.w3.org

Geldinstitut gemeint ist.

Wie das World Wide Web setzt das Semantic Web auf Dezentralisierung: Wissensbasen können aufeinander aufbauen und zueinander in Beziehung gesetzt werden, Maschinen können ihre extrahierten Informationen teilen. Auf Kosten der Konsistenz wird dadurch uneingeschränktes Wachstum ermöglicht [BHL01].

Ein Beispiel für das Bereitstellen von Information in maschinenlesbarer Form ist DBpedia[2]. Ziel dieses Projekts ist es, das in Wikipedia enthaltene Wissen strukturiert zur Verfügung zu stellen. Die strukturierten Daten stehen untereinander in Beziehung und lassen sich mit Hilfe der Abfragesprache SPARQL abrufen. Damit lassen sich komplexe Fragen wie z. B. „Welche italienischen Künstler waren im 15. Jahrhundert tätig?“ beantworten [ABK+07]. DBpedia war eine der Wissensquellen für den Supercomputer Watson von IBM [FBC+10].

2.1.1 Resource Description Framework

Das Resource Description Framework (RDF) ist ein Modell für semantische Daten, das die Formulierung von Aussagen über beliebige Ressourcen (Dokumente, Personen etc.) erlaubt. Jede Aussage wird als Tripel angegeben und besteht aus Subjekt, Prädikat und Objekt [SRM+14]. Tabelle 2.1 zeigt Beispiele für Aussagen in Tripel-Form.

Tabelle 2.1: Beispiele für Aussagen in Tripel-Form in natürlicher Sprache.

Subjekt	*Prädikat*	*Objekt*
Leonardo da Vinci	ist eine	Person
Leonardo da Vinci	ist ein	Künstler
Leonardo da Vinci	wurde geboren am	15.04.1452
Leonardo da Vinci	malte	Mona Lisa

Serialisierung von RDF-Daten

Zur global eindeutigen Identifizierung von Ressourcen und Prädikaten wird diesen in RDF ein International Resource Identifier (IRI) zugewiesen. Eine IRI ist die Verallgemeinerung eines Uniform Resource Identifiers (URI), die eine größere Menge von Zeichen zulässt. Jede gültige URI ist automatisch auch eine IRI. Da die IRIs lediglich als Identifikator dienen, müssen sie nicht per Netzwerk erreichbar sein. Werte wie Zahlen oder Zeichenketten werden als Literale kodiert. Ein Beispiel ist das Geburtsdatum in Tabelle 2.1.

[2] http://dbpedia.org

Listing 2.1: Tripel aus Tabelle 2.1 kodiert mit Hilfe von URIs und Literalen.

```
1 <http://dbpedia.org/resource/Leonardo_da_Vinci>
      ↪ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
      ↪ <http://dbpedia.org/ontology/Person> .
2 <http://dbpedia.org/resource/Leonardo_da_Vinci>
      ↪ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
      ↪ <http://dbpedia.org/ontology/Artist> .
3 <http://dbpedia.org/resource/Leonardo_da_Vinci>
      ↪ <http://dbpedia.org/ontology/birthDate>
      ↪ "1452-04-15"^^<http://www.w3.org/2001/XMLSchema#date> .
4 <http://dbpedia.org/resource/Leonardo_da_Vinci>
      ↪ <http://dbpedia.org/property/artist>
      ↪ <http://dbpedia.org/resource/Mona_Lisa> .
```

Literale sind in Tripeln nur als Objekt erlaubt [SRM+14]. Listing 2.1 zeigt die Beispieldaten in Form von URIs und Literalen. Zur Darstellung wurde die N-Triples-Syntax verwendet. Diese sieht die Notation von einem Tripel pro Zeile vor. Jede Zeile muss mit einem Punkt beendet werden. IRIs werden in spitzen Klammern, Literale in Anführungszeichen eingeschlossen [BCS14].

Neben N-Triples gibt es weitere Möglichkeiten, um RDF-Daten zu serialisieren. Beispiele sind Turtle [BBP+14] oder RDF/XML [GSB14]. Im weiteren Verlauf wird das N-Triples-Format wegen seiner Einfachheit verwendet.

Datentypen und Sprachinformationen

Dem Geburtsdatum im Beispiel 2.1 wurde zusätzlich ein Datentyp zugewiesen. Dieser wird ebenfalls durch eine IRI repräsentiert und durch die Zeichenkette „^^“ vom Wert des Literals getrennt. Datentypen weisen einem String eine spezielle Bedeutung zu [SRM+14]. Es lassen sich beliebige IRIs als Datentyp nutzen. Die Verwendung weit verbreiteter Typen (z. B. denen aus XML-Schema [BMP+12]) hat den Vorteil, dass viele RDF verarbeitende Systeme diese validieren oder konvertieren können.

Anstelle eines Datentyps kann einer Zeichenkette eine Sprache zugewiesen werden. Dazu wird der Sprachcode durch ein „@“-Symbol von dieser getrennt [SRM+14], also z. B. `"Italy"@en`.

Beschreibung von Anwendungsdomänen

Um Anwendungsdomänen modellieren zu können, wurden Ontologiesprachen wie RDFS und OWL entwickelt [HKR+06]. Diese Sprachen ermöglichen die Definition von Regeln, die ebenfalls als Tripel ausgedrückt werden [BGM14; Owl12]. Das Beispiel in Listing 2.2 definiert, dass Gemälde Kunst sind. Da die Mona Lisa ein Gemälde ist, handelt es sich nach dieser Regel

Listing 2.2: Beispiel für eine mit RDFS ausgedrückte Regel.

```
1 <http://dbpedia.org/ontology/Painting>
      ↪ <http://www.w3.org/2000/01/rdf-schema#subClassOf>
      ↪ <http://dbpedia.org/ontology/Artwork> .
2 <http://dbpedia.org/resource/Mona_Lisa>
      ↪ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
      ↪ <http://dbpedia.org/ontology/Painting> .
```

Listing 2.3: Darstellung eines Quads im N-Quads-Format.

```
1 <http://dbpedia.org/resource/Leonardo_da_Vinci>
      ↪ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
      ↪ <http://dbpedia.org/ontology/Person> <http://dbpedia.org> .
```

automatisch um Kunst. Neben solchen Hierarchien lassen mit Ontologiesprachen zulässige Subjekte und Objekte für Relationen, Einschränkungen von Werten etc. beschreiben [BGM14; Owl12].

Mit Hilfe der Ontologiesprachen wurden Vokabulare zur strukturierten Beschreibung spezifischen Domänenwissens entwickelt. Ein Beispiel ist das „Friend of a Friend“-Vokabular (FOAF), das zur Beschreibung von Personen und deren Beziehungen verwendet werden kann [BL14].

In den Beispielen aus Listing 2.1 und 2.2 wurde das von DBpedia definierte Vokabular verwendet.

Tripelmengen als Graph

Eine Menge von Tripeln lässt sich als gerichteter Graph auffassen. Dazu werden Subjekte und Objekte als Knoten und Prädikate als gerichtete Kanten interpretiert. Abbildung 2.1 visualisiert die Daten aus Listing 2.1 als Graph.

Aufteilung von Tripelmengen in Graphen

Um Tripel in verschiedene Mengen aufzuteilen, kann jedem Tripel eine zusätzliche Eigenschaft, der Graph oder auch Kontext, zugewiesen werden. Man spricht in diesem Fall von Quads. Diese lassen sich analog zu N-Triples im N-Quads-Format serialisieren [Car14]. Listing 2.3 zeigt ein serialisiertes Quad. Zusätzlich zu den Tripel-Bestandteilen wurde der Graph als vierte Eigenschaft angegeben. Dieser wird ebenfalls durch eine IRI repräsentiert.

Das Konzept der Graphen ist bei der Abfrage von Daten mit SPARQL relevant, da SPARQL die Angabe von Graphen als Datenquellen erlaubt.

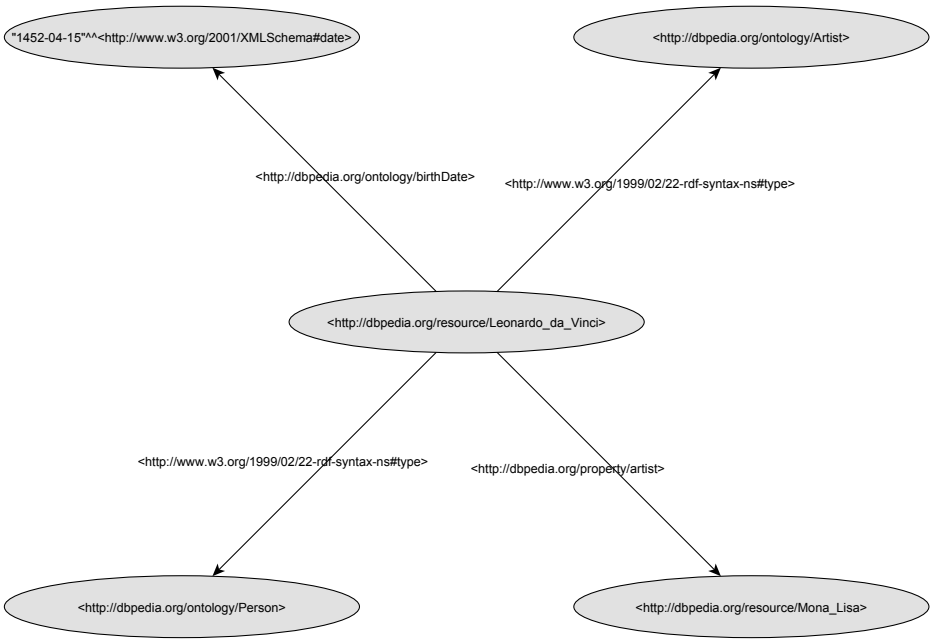


Abbildung 2.1: Tripel aus Listing 2.1, visualisiert als Graph.

2.1.2 Abfrage von Daten mit SPARQL

SPARQL ist eine Abfragesprache für RDF-Daten. Der Name ist ein rekursives Akronym, das für **S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage steht [DuC11].

Zum Spezifizieren der gewünschten Daten werden in SPARQL sogenannte Triple Patterns verwendet. Diese ähneln der in Abschnitt 2.1.1 beschriebenen Serialisierung im N-Triples-Format. Zusätzlich kann jeder Bestandteil des Tripels durch eine Variable ersetzt werden [HS13]. Variablen beginnen mit einem Fragezeichen.

Die gesuchten Daten müssen alle angegebenen Triple Patterns erfüllen. Die Abfrage in Listing 2.4 verwendet zwei Tripel Patterns, um mit Hilfe von DBpedia italienische Künstler zu ermitteln. Das Ergebnis der Abfrage sind keine Tripel, sondern eine tabellarische Darstellung der Belegung der angefragten Variablen. Tabelle 2.2 zeigt einen Ausschnitt dieser tabellarischen Darstellung.

Hinzufügen von Bedingungen

Mit Hilfe des FILTER-Konstrukts lassen sich innerhalb einer SPARQL-Abfrage Bedingungen ausdrücken, die sich schwer oder gar nicht als Triple Pattern formulieren lassen [DuC11; HS13].

Listing 2.4: Abfrage zum Ermitteln italienischer Künstler.

```
1 SELECT ?italianArtist
2 WHERE {
3   ?italianArtist <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
      ↪ <http://dbpedia.org/ontology/Artist> .
4   ?italianArtist <http://dbpedia.org/ontology/nationality>
      ↪ <http://dbpedia.org/resource/Italians> .
5 }
```

Tabelle 2.2: Gekürzte Liste der Abfrageergebnisse aus Listing 2.4.

?italianArtist
<http://dbpedia.org/resource/Bramantino>
<http://dbpedia.org/resource/Cesare_Maccari>
<http://dbpedia.org/resource/Girolamo_de_Rada>
<http://dbpedia.org/resource/Palma_il_Giovane>
<http://dbpedia.org/resource/Caravaggio>
<http://dbpedia.org/resource/Leonardo_da_Vinci>
...

Die Abfrage in Listing 2.5 erweitert jene aus 2.4 um eine Filter-Bedingung, die nur Künstler akzeptiert, die im 15. Jahrhundert geboren wurden oder gestorben sind. Um Daten in einer Filter-Bedingung prüfen zu können, müssen diese zunächst via Triple Pattern ermittelt werden. Die Abfrage zeigt außerdem, wie die Schreibweise von IRIs durch die Definition eines Präfixes verkürzt werden kann [HS13].

Kombination von Ergebnismengen

Mittels `UNION`-Schlüsselwort lassen sich Teilergebnisse kombinieren [DuC11; HS13]. Die Abfrage in Listing 2.6 kombiniert zwei Mengen, um sowohl italienische als auch deutsche Künstler zu selektieren. Die Bedingungen für die Mengen der italienischen und deutschen Künstler sind in zusätzliche Klammern eingefasst und formen jeweils ein sogenanntes Group Graph Pattern [HS13]. Damit die Teilergebnisse sinnvoll vereinigt werden können ist es wichtig, dass sie dieselbe Struktur aufweisen. Im vorliegenden Beispiel liefern beide Teilergebnisse eine Belegung für die Variable `?artist`.

Listing 2.5: Abfrage zum Ermitteln italienischer Künstler, die im 15. Jahrhundert geboren wurden oder gestorben sind.

```
1 PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
2 SELECT ?italianArtist
3 WHERE {
4   ?italianArtist <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
      ↪ <http://dbpedia.org/ontology/Artist> .
5   ?italianArtist <http://dbpedia.org/ontology/nationality>
      ↪ <http://dbpedia.org/resource/Italians> .
6   ?italianArtist <http://dbpedia.org/ontology/birthDate> ?birthDate .
7   ?italianArtist <http://dbpedia.org/ontology/deathDate> ?deathDate .
8   FILTER(
9     (?birthDate >= "1400-01-01"^^xsd:date && ?birthDate <
      ↪ "1500-01-01"^^xsd:date)
10    ||
11    (?deathDate >= "1400-01-01"^^xsd:date && ?deathDate <
      ↪ "1500-01-01"^^xsd:date)
12  )
13 }
```

Listing 2.6: Abfrage zum Ermitteln italienischer und deutscher Künstler.

```
1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX dbpediaOnt: <http://dbpedia.org/ontology/>
3 SELECT ?artist
4 WHERE {
5   {
6     ?artist rdf:type dbpediaOnt:Artist .
7     ?artist dbpediaOnt:nationality
      ↪ <http://dbpedia.org/resource/Italians> .
8   } UNION {
9     ?artist rdf:type dbpediaOnt:Artist .
10    ?artist dbpediaOnt:nationality
      ↪ <http://dbpedia.org/resource/Germany> .
11  }
12 }
```

Angabe der Graphen für die Abfrage

In den bisherigen Beispielen für SPARQL-Abfragen wurde nicht angegeben, aus welchem Graph die Tripel stammen, die von der Abfrage berücksichtigt werden. Da der SPARQL-Standard [HS13] nicht definiert, wie in diesem Fall zu verfahren ist, hängt das Verhalten von dem verwendeten RDF-Datenspeicher (englisch: Triple Store) ab.

Mit Hilfe der Schlüsselworte `FROM` und `FROM NAMED` erlaubt SPARQL es, eine beliebige Anzahl von Graphen bei der Abfrage zu berücksichtigen. Die

Listing 2.7: Bilden des abgefragten Default Graph mit `FROM`.

```
1 SELECT *
2 FROM <http://dbpedia.org>
3 FROM <http://dbpedia.org/resource/classes#>
4 WHERE {
5   ?subject ?predicate ?object .
6 }
```

Listing 2.8: Abfrage eines Graphen mit `FROM NAMED` und `GRAPH`-Schlüsselwort.

```
1 SELECT *
2 FROM NAMED <http://dbpedia.org>
3 WHERE {
4   GRAPH ?g {
5     ?subject ?predicate ?object .
6   }
7 }
```

nach `FROM` notierten Graphen werden zu einem sogenannten Default Graph zusammengeführt (vgl. Beispiel in Listing 2.7). Dieser enthält alle Tripel aus den angegebenen Graphen, allerdings keine Informationen über deren Ursprungs-Graph [HS13]. Bei Verwendung von `FROM NAMED` bleibt die Zugehörigkeit zum Graph erhalten. Die Tripel werden aber nicht in den Default Graph importiert und müssen unter Angabe eines Graphen adressiert werden (vgl. Beispiel in Listing 2.8) [HS13].

2.2 Softwareentwicklung

Die folgenden Abschnitte verschaffen einen kurzen Überblick über die für das Verständnis der Arbeit notwendigen Konzepte und Technologien der Softwareentwicklung. Diese umfassen Design Patterns (Abschnitt 2.2.1), Test Driven Development (Abschnitt 2.2.2) und die zum Einsatz kommende Programmiersprache PHP (Abschnitt 2.2.3).

2.2.1 Design Patterns

Design Patterns (deutsch: Entwurfsmuster) sind Lösungsvorlagen zu wiederkehrenden Entwurfsproblemen. Der Begriff stammt ursprünglich aus der Architektur und wird heute auch in der Softwareentwicklung verwendet. Design Patterns werden nicht erfunden, sondern aus in der Praxis erprobten Lösungen extrahiert [GHJ+94].

Listing 2.9: Änderung eines Variablentyps und Prüfung von Objekteigenschaften zur Laufzeit in PHP.

```
1  $date = '1452-04-15';
2  $date = new DateTime($date);
3  if (method_exists($date, 'format')) {
4    echo $date->format('Y');
5  }
```

Die bekanntesten Muster stammen aus dem Buch „Design Patterns“ [GHJ+94]. Daneben existieren domänenspezifische Patterns, z. B. für Webapplikationen oder Messaging-Lösungen [Fow02; HW03]. Um die Kommunikation über Lösungen zu erleichtern, besitzt jedes Muster einen eigenen Namen.

Das Gegenstück zu Design Patterns sind Anti-Patterns: Lösungen die sich in der Praxis nicht bewährt haben und dementsprechend vermieden werden sollten [BMM98; FBB+99].

2.2.2 Test Driven Development

Test Driven Development (TDD) ist eine Entwicklungsmethode, bei der das Verhalten einer Softwarekomponente vor deren Entwicklung mit Hilfe automatisierter Tests spezifiziert wird. Die Entwicklung einer Komponente ist dann abgeschlossen, wenn diese von allen Tests akzeptiert wird [Bec02].

TDD führt in der Regel zu robusteren Entwürfen und garantiert, dass das definierte Verhalten bei späterer Überarbeitung des Codes (sogenanntes Refactoring) erhalten bleibt [Bec02; FBB+99].

2.2.3 PHP

PHP ist eine weit verbreitete[3] Skriptsprache, die primär in der Webentwicklung zum Einsatz kommt. Die Sprache verwendet eine C-ähnliche Syntax und lässt sich sowohl prozedural als auch objektorientiert verwenden.

Im Gegensatz zu Sprachen wie Java verwendet PHP keine statische, starke Typisierung. Typinformationen werden erst zur Laufzeit geprüft. Der Typ einer Variable kann sich daher während der Programmausführung ändern. Eine manuelle Prüfung von Objekteigenschaften ist nicht unüblich (vgl. Beispiel in Listing 2.9). Bei unachtsamer Programmierung kann es zu Laufzeitfehlern, wie z. B. dem Zugriff auf nicht existierende Methoden, kommen.

PHP unterstützt gängige Features der objektorientierten Programmierung, wie beispielsweise Interfaces, Klassen und einfache Vererbung. Bei

[3] Laut TIOBE-Index im Juni 2014, abrufbar unter http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

konsequenter Verwendung dieser Sprachbestandteile lassen sich die Unsicherheiten in der Typisierung deutlich reduzieren.

2.3 OntoWiki

Die Webanwendung OntoWiki ist ein Wiki-System zum kollaborativen Bearbeiten von semantischen Daten [ADL+07; TFH10]. Im Gegensatz zu anderen Wikis mit semantischen Auszeichnungen (z. B. Semantic MediaWiki) setzt OntoWiki nicht auf die Verwendung von Annotationen oder das Hinzufügen von Metadaten zu Textdaten, sondern verwendet das RDF-Modell zum Speichern der Daten [ADL+07; KVD+07].

Die Daten lassen sich mit dem Editor RDFauthor im Webbrowser bearbeiten. Dieser berücksichtigt die mittels Ontologiesprachen definierten Strukturen [THA+10]. Das in PHP geschriebene Wiki bietet gezielte Navigation durch RDF-Graphen, Suchen innerhalb der Daten und Änderungshistorien von Datensätzen. Eine Plugin-Schnittstelle ermöglicht die Anbindung weiterer Funktionalitäten [FMT+14; HDM+09]. Abbildung 2.2 zeigt die Nutzeroberfläche des Wikis.

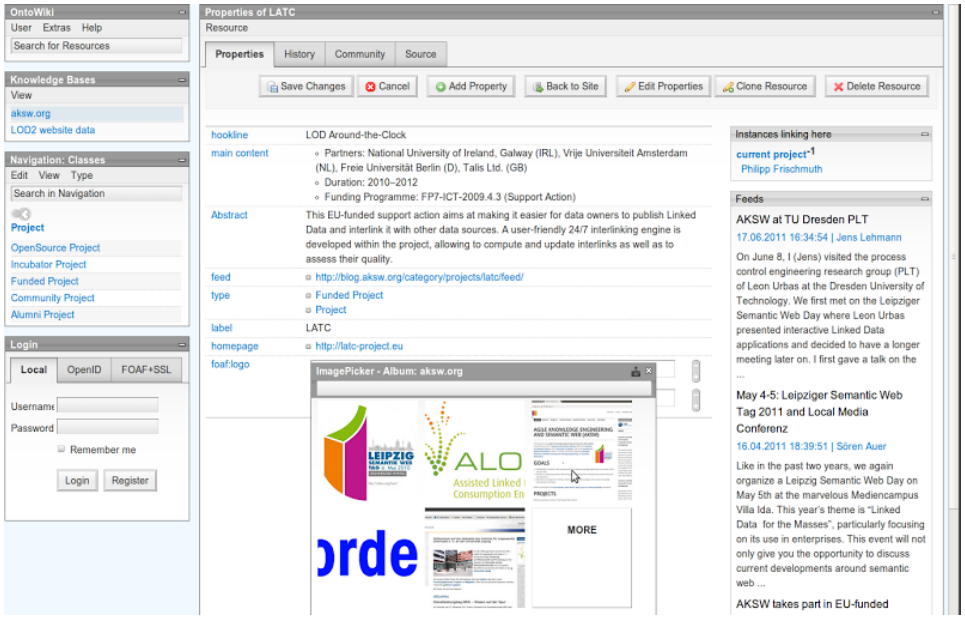


Abbildung 2.2: Nutzeroberfläche von OntoWiki[a].

[a] Screenshot von https://github.com/AKSW/OntoWiki

Das Wiki baut auf der Zend Framework 1[4] Applikationsinfrastruktur auf. Grundlage für die Verwaltung der semantischen Daten ist das Frame-

[4] https://github.com/zendframework/zf1

work Erfurt[5]. Dieses übernimmt die Verarbeitung der RDF-Daten und die Kommunikation mit den für die Persistenz verantwortlichen RDF-Datenspeichern [FMT+14; HDM+09].

2.4 Erfurt Framework

Erfurt ist ein Framework zum Erstellen von Web-Anwendungen, die mit semantischen Daten arbeiten. Die Aufgabe von Erfurt ist die Kommunikation mit RDF-Datenspeichern über eine einheitliche Schnittstelle und das Bereitstellen häufig benötigter Funktionalität wie Caching, Logging und Rechte-Management [MUA10]. Zusätzlich bietet es Basis-Komponenten, die beispielsweise zur Analyse und zum programmatischen Erstellen von SPARQL-Abfragen oder zum Parsen von RDF-Daten verwendet werden können. Wie OntoWiki verwendet Erfurt Teile des Zend Framework 1 zur Ausführung von Grundfunktionen, wie z. B. dem Senden von HTTP-Anfragen.

2.5 Verwandte Arbeiten

Bezüglich Aufbau und Funktionsumfang von OntoWiki existieren zahlreiche Arbeiten der AKSW-Forschungsgruppe. Einen aktuellen Überblick bietet [FMT+14]. Arbeiten wie [HDM+09] und [MUA10] beschäftigen sich mit Details des zugrunde liegenden Frameworks Erfurt. Die Generalisierung des Datenzugriffs steht in keiner der Arbeiten im Fokus.

Im Bereich der relationalen Datenbanken existieren zahlreiche Projekte, die eine Vereinheitlichung des Datenzugriffs umsetzen. Dazu gehören beispielsweise JDBC[6] für Java sowie Doctrine DBAL[7] für PHP.

Für semantische Datenbanken ist diese Entwicklung noch nicht so weit fortgeschritten. Ein einheitlicher Datenzugriff lässt sich für Java-Applikationen mit Apache Jena[8] oder Sesame[9] erreichen, deren APIs von vielen RDF-Datenspeichern unterstützt werden. Für PHP-Applikationen steht keine Bibliothek mit ähnlich starker Verbreitung zur Verfügung.

[5] https://github.com/AKSW/Erfurt
[6] http://www.oracle.com/technetwork/java/javase/jdbc/index.html
[7] http://www.doctrine-project.org/projects/dbal.html
[8] https://jena.apache.org/
[9] http://www.openrdf.org/

Kapitel 3

Anforderungserhebung

Bei Generalisierungen besteht die Gefahr, Konzepte unangemessen stark zu verallgemeinern. Dies kann unnötigen Aufwand für die Entwicklung und das Verständnis bedeuten. Auf Code-Ebene zeigt sich zu starke Generalisierung durch das Anti-Pattern „Speculative Generality“ [FBB+99]. Um diese Gefahr bereits auf der Entwurfsebene zu reduzieren, werden in diesem Kapitel Nutzungs-Szenarien vorgestellt. Aus diesen Szenarien werden Probleme abgeleitet, anhand derer sich Anforderungen an die Erweiterung stellen lassen [RR06]. Mit Hilfe dieser Anforderungen lässt sich die Generalisierung der Persistenzschicht von OntoWiki zielgerichtet durchführen [RR06].

3.1 Szenarien

Im Folgenden werden unterschiedliche Szenarien für den Einsatz von OntoWiki vorgestellt. Je nach Szenario entstehen unterschiedliche Erwartungen an das Wiki und das darunter liegende Framework.

3.1.1 Erstellen einer Website

Eine Forschungsgruppe möchte ihre Mitglieder und durchgeführte Projekte auf einer Website vorstellen.

Die Informationen müssen sinnvoll verknüpft werden können. Es soll beispielsweise möglich sein, über ein Projekt zu den beteiligten Personen zu navigieren. Auf den persönlichen Seiten der Mitglieder soll sichtbar sein, an welchen Projekten sie beteiligt waren.

Bei Datenänderungen sollen die Anpassungen umgehend auf der Website zur Verfügung stehen. Die Pflege darf nur wenig Aufwand kosten, da es sich dabei nicht um die Haupttätigkeit der Gruppenmitglieder handelt.

3.1.2 Datenpflege in Großunternehmen

Eine Abteilung in einem Großunternehmen möchte Teile ihrer Arbeit in Form von semantischen Daten verwalten, um ihre Wissensdomäne besser beschreiben und Zusammenhänge bei Bedarf schneller modellieren zu können.

Da es nicht jedem Mitarbeiter gestattet ist alle Daten einzusehen, ist ein entsprechendes Rechte-Management nötig. Weil die Auslastung der Mitarbeiter hoch ist, sollen Verzögerungen durch die Einführung der neuen Software möglichst vermieden werden.

Die Firma verfügt über eine bestehende Infrastruktur, welche die Speicherung von Daten in relationalen Datenbanken vorsieht. Die Unternehmensleitung hat die Richtlinie herausgegeben, dass neue Komponenten Schnittstellen besitzen müssen, mit denen sie in diese Infrastruktur integriert werden können.

3.1.3 Frontend für RDF-Datenspeicher

Ein Unternehmen verwendet einen RDF-Datenspeicher, der durch Inferenz aus Basis-Daten neue Fakten generiert. Die ermittelten Informationen werden von nachfolgenden Systemen verwendet, die programmatisch direkt auf die Datenbank zugreifen.

Um die Pflege der Basis-Daten zu erleichtern, soll OntoWiki als User-Interface eingesetzt werden.

3.1.4 Wechsel des Datenspeichers nach Testphase

Ein Nutzer verwendet OntoWiki seit geraumer Zeit. Bei der Einrichtung hat er zur Datenhaltung MySQL gewählt, da er im Umgang mit diesem Datenbanksystem bereits Erfahrung hatte.

Durch das Wachstum der Datenmenge hat sich die Performanz deutlich verschlechtert. Daher sollen die bestehenden Daten jetzt in einen anderen Datenspeicher migriert werden, der RDF nativ unterstützt. Dadurch erhofft er sich eine Leistungssteigerung des Systems.

3.1.5 Anbindung eines neuen Datenspeichers

Ein Unternehmen verwendet OntoWiki zur Datenpflege. Zur besseren Integration in die restliche Infrastruktur soll ein RDF-Datenspeicher eingesetzt werden, der vom Wiki nicht unterstützt wird.

Um eine erneute Einarbeitung der mit Datenpflege betrauten Mitarbeiter zu vermeiden, soll das Wiki an diese neue Datenquelle angebunden werden. Der Umfang dieses Projekts darf nur wenige Wochen umfassen, da der Entwicklungsaufwand sonst die Vorteile der vermiedenen Einarbeitung überwiegt.

3.2 Probleme

In den vorgestellten Szenarien werden zahlreiche Probleme aufgezeigt, die das Wiki und die darunter liegende Infrastruktur betreffen. Nachfolgend werden diese Probleme identifiziert, analysiert und einem Teilsystem zugeordnet.

3.2.1 Vermeidung von Verzögerungen

Nutzer des Wikis verwenden dieses, um ihre fachlichen Probleme zu lösen. Es ist davon auszugehen, dass es sich nicht um professionelle Systemadministratoren oder Experten im Umgang mit semantischen Daten handelt (Szenario 3.1.1 und 3.1.2). Dennoch soll die Verwendung des Wikis reibungslos ablaufen und Verzögerungen sollen vermieden werden. Dazu sind unterschiedliche Aspekte zu betrachten:

Installation
: Die Installation soll ohne Expertenkenntnisse durchführbar sein. Dies betrifft sowohl das Wiki-System als auch die anzubindende Datenbank.

Bedienbarkeit
: Um eine aufwändige Einarbeitung zu vermeiden, soll das Wiki möglichst intuitiv benutzbar sein. Dies betrifft weniger die Persistenzschicht als viel mehr die Benutzeroberfläche des Wikis, die bereits umfangreiche Hilfen bereitstellt [THA+10].

Antwortzeiten
: Um flüssig mit dem System arbeiten zu können, sollen die Antwortzeiten möglichst kurz sein. Dieser Aspekt betrifft die Persistenzschicht, da diese die Grundfunktionen zur Datenverwaltung bereitstellt.

Ausfälle
: Um flüssiges Arbeiten zu ermöglichen soll das System eine hohe Ausfallsicherheit aufweisen. Dies betrifft sowohl die angebotenen Funktionen des Wikis als auch das darunter liegende Persistenzsystem, auf dem diese aufbauen.

3.2.2 Verknüpfung von Daten

Um verschiedene Sichten auf Daten erhalten und zwischen diesen navigieren zu können (Szenario 3.1.1), müssen diese flexibel verknüpft werden können.

Da OntoWiki auf RDF-Daten setzt, ist dies bereits nativ möglich. Zur Abfrage wird SPARQL verwendet. Diese Sprache muss von der Persistenzschicht unterstützt werden. Zum Erstellen von Webseiten ermöglicht das Wiki das Einbetten solcher Abfragen in Templates.

3.2.3 Zugriffsschutz

In Situationen mit vielen Benutzern (Szenario 3.1.2) kann eine Rechteverwaltung nötig sein. Das Wiki bietet dafür eine Lösung auf SPARQL-Graph-Ebene. Für die darunter liegende Persistenzschicht ist diese völlig transparent.

3.2.4 Integration in bestehende Systemlandschaften

Wie in Szenario 3.1.2 ist es häufig notwendig, dass sich eine Software-Lösung in eine bestehende Systemlandschaft integrieren lässt.

Dafür sind Schnittstellen sinnvoll, über die externe Programme auf die Daten zugreifen können. Eine Schnittstelle zur Abfrage der verwalteten Daten via SPARQL wird vom Wiki zur Verfügung gestellt [TFH10].

Manchmal ist die Gegenrichtung gewünscht: Die Software-Lösung soll bestehende Infrastruktur nutzen. Denn viele Unternehmen haben sich vor längerer Zeit für eine bestimmte Infrastruktur entschieden, diese lizenziert und in ihre Prozess- und Systemlandschaft eingebunden. Häufig sind relationale Datenbanken Bestandteil der bestehenden Infrastruktur. Ein kompletter Technologiewechsel ist in solchen Situationen kaum praktikabel.

In Szenario 3.1.3 wird das Wiki als Benutzeroberfläche für eine Datenbank verwendet. Die Datenbank wird von weiteren Programmen direkt abgefragt. Um diese Programme nicht auszubremsen, bietet sich die Unterstützung spezialisierter und schneller RDF-Datenbanksysteme an.

3.2.5 Initiale Inbetriebnahme

Das Szenario 3.1.4 ist ein Beispiel dafür, dass zum Ausprobieren und zur initialen Inbetriebnahme einer Software-Lösung häufig die am schnellsten verfügbare Technologie verwendet wird.

Der Einstieg kann unterstützt werden, indem die Persistenzschicht mit Systemen zusammenarbeitet, die weit verbreitet sowie einfach zu installieren und zu konfigurieren sind.

3.2.6 Migration

In den Szenarien 3.1.4 und 3.1.5 wird ersichtlich, dass es bei Änderung der Einsatzbedingungen sinnvoll sein kann den verwendeten Datenspeicher zu wechseln.

Dazu ist es zum einen wichtig, dass bestehende Daten sich migrieren lassen. Hierfür bietet OntoWiki umfangreiche Import- und Export-Möglichkeiten an. Zum anderen ist es entscheidend, dass die verschiedenen Datenspeicher eine einheitliche Menge an Features bereitstellen. Die Verwendung proprietärer, nicht flächendeckend verfügbarer Funktionalitäten bindet eine

Anwendung an ein bestimmtes System. Dieser Vendor-Lock-In verhindert die Migration zu alternativen Lösungen [BMM98].

Im Bereich der RDF-Datenspeicher besteht das Problem, dass die Standards noch jung (z. B. SPARQL 1.1 aus 2013 [HS13]) und nicht komplett und einheitlich umgesetzt sind. Zudem bieten einige Hersteller proprietäre Zusatzfunktionen.

3.2.7 Anpassungsfähigkeit

Szenario 3.1.5 zeigt, dass es mitunter nötig ist, Software an eigene Bedürfnisse anpassen zu können. Eine Anpassung der Datenhaltung durch Dritte ist nur möglich, wenn sich Fremdkomponenten in das System integrieren lassen. Außerdem darf eine Erweiterung nicht zu aufwändig sein, da sie sich sonst nicht rentiert.

3.3 Anforderungen

Aus den identifizierten Problemen sollen konkrete Anforderungen abgeleitet werden. Es wird nach [RR06] zwischen funktionalen und nicht-funktionalen Anforderungen entschieden. Eine funktionale Anforderung definiert, was eine Software-Lösung tun soll. Eine nicht-funktionale Anforderung legt fest, wie etwas getan werden soll.

Im Kontext einer Website könnte folgendes eine funktionale Anforderung sein: *„Nach dem Login zeigt die Seite den Namen des Benutzers.“* Die folgende Aussage wäre ein Beispiel für eine nicht-funktionale Anforderung: *„Die Website muss innerhalb von 500 Millisekunden ausgeliefert werden.“*

Aus den Szenarios und Problemen lassen sich die folgenden Anforderungen an die Persistenzschicht ableiten. Es handelt sich primär um nicht-funktionale Anforderungen, da die Persistenzschicht im Infrastruktur-Bereich angesiedelt ist. Jede Anforderung besteht aus einer Beschreibung, einer Begründung der Notwendigkeit und der Angabe der Quellen aus denen sie hervorgegangen ist.

1. **Funktionale Anforderungen**

   (a) **Anbindung einer Datenbank für gängige Infrastrukturen**

   Beschreibung
   OntoWiki soll an ein Datenbanksystem angeschlossen werden, das in bestehenden Infrastrukturen weit verbreitet ist.

   Begründung
   In bestehenden IT-Landschaften ist die Einführung neuer Systeme häufig problematisch. OntoWiki soll die Möglichkeit bieten mit gängigen Systemen zusammenzuarbeiten.

Quelle
Problem 3.2.4

(b) **Anbindung eines nativen RDF-Datenspeichers**

Beschreibung
OntoWiki soll in der Lage sein mit einem Datenbank-Management-System zu arbeiten, das RDF-Daten nativ unterstützt.

Begründung
Wenn die Entscheidung zur Nutzung semantischer Daten unabhängig von bestehender Infrastruktur getroffen werden kann, dann liegt es nahe ein spezialisiertes System zu wählen, das RDF-Daten möglichst effizient verwaltet.

Quelle
Problem 3.2.1

(c) **Anbindung einer Datenbank für Einstieg und Test**

Beschreibung
Es soll eine Datenbank angebunden werden, die weit verbreitet, einfach zu beschaffen und leicht zu bedienen ist.

Begründung
Die Einstiegshürde zum Testen des Wikis soll gering sein. Daher soll ein Datenspeicher unterstützt werden, der leicht zu beschaffen und ohne großen Administrationsaufwand in Betrieb zu nehmen ist.

Quelle
Problem 3.2.5

2. **Nicht-Funktionale Anforderungen**

(a) **Korrektheit**

Beschreibung
Die Persistenzschicht soll Daten korrekt speichern, löschen und auslesen.

Begründung
Systeme, die die gewünschten Operationen nicht oder nur teilweise korrekt ausführen sind unbenutzbar oder behindern den Endnutzer bei der Arbeit.

Quelle
Problem 3.2.1

(b) **Abwärtskompatibilität**

Beschreibung
Bestehende Schnittstellen zur Persistenzschicht sollen erhal-

ten bleiben. Ein Update des Frameworks soll nach der Generalisierung der Persistenzschicht möglich sein.

Begründung

Die Persistenzschicht ist als Framework frei verfügbar. Es ist davon auszugehen, dass sie als Komponente in Software-Lösungen Dritter verwendet wird. Eine Änderung der Schnittstellen erschwert für diese Software-Lösungen das Update dieser Komponente, da der Zugriff auf die Schnittstellen überprüft und ggf. angepasst werden muss. Das Problem wird dadurch verschärft, dass die Nutzer des Frameworks nicht bekannt und nicht bestimmbar sind.

Quelle

Problem 3.2.6

(c) **Plattformunabhängigkeit**

Beschreibung

Die Persistenzschicht soll sich nicht auf proprietäre Funktionalitäten verlassen. Verwendete Features müssen von allen Systemen nativ unterstützt oder ggf. simuliert werden.

Begründung

Die Nutzung proprietärer Funktionalitäten resultiert in einem Vendor-Lock-In [BMM98].

Quelle

Problem 3.2.6

(d) **Wartbarkeit**

Beschreibung

Der Code soll wartbar sein, d. h. es ist auf Verständlichkeit und ausreichende Dokumentation zu achten.

Begründung

Quellcode wird häufiger gelesen als geschrieben [Mar08]. Unverständliche Konstrukte oder fehlende Dokumentation erschweren die Erweiterung oder die Korrektur von Fehlern.

Quelle

Problem 3.2.7

(e) **Erweiterbarkeit**

Beschreibung

Das zu entwickelnde System soll sich in Zukunft an weitere Datenbanken anbinden lassen. Dazu muss das System so entworfen werden, dass es Erweiterungspunkte und Schnittstellen bietet, die sich von externen Entwicklern nutzen lassen, deren Code in einem separaten Projekt gepflegt wird.

Begründung
Mit den zur Verfügung stehenden Mitteln ist es nicht möglich, das Wiki an alle existierenden Datenspeicher anzubinden. Um das Wiki mit nicht offiziell unterstützen Datenbanken nutzen zu können, muss für jeden (notwendiges Fachwissen vorausgesetzt) eine Erweiterung möglich sein.

Quelle
Problem 3.2.7

(f) **Performanz**

Beschreibung
Bei der Verwendung von Standard-Funktionen (Daten anzeigen, Datensatz anlegen etc.) soll das Wiki innerhalb von 1 Sekunde antworten.

Begründung
Bei zu langer Verarbeitungsdauer ist das flüssige Arbeiten mit dem System nicht möglich.

Quelle
Problem 3.2.1

(g) **Transparente Installation**

Beschreibung
Wenn ein Datenspeicher eine weitergehende Installation oder Konfiguration benötigt, soll diese so weit wie möglich transparent und automatisch erfolgen.

Begründung
Die Inbetriebnahme wird durch manuelle Schritte komplizierter und fehleranfälliger. Eine weitgehende Automatisierung vereinfacht diese und spart dem Nutzer Zeit.

Quelle
Probleme 3.2.1 und 3.2.5

Da die Anforderungen in manchen Situationen miteinander in Konflikt stehen können, sind diese nach Priorität sortiert. Dies erleichtert die Entscheidung, wenn Abwägungen getroffen werden müssen (z. B. Plattformunabhängigkeit zu Lasten der Performanz).

Kapitel 4

Analyse des existierenden Systems und Erweiterung der Architektur

Dieses Kapitel behandelt den Entwurf der Architekturerweiterung zur Generalisierung der Persistenzschicht.

Dazu werden zuerst die Bestandteile von OntoWiki untersucht, um wiederverwendbare Komponenten zu identifizieren. Ein besonderer Fokus liegt auf dem zugrunde liegenden Framework Erfurt. Eine nähere Betrachtung des Zusammenspiels der Komponenten zeigt Stärken und Schwächen des aktuellen Systems auf. Auf Grundlage dieser Erkenntnisse werden schließlich Entscheidungen zum Design der neuen, für die Erweiterung notwendigen Komponenten getroffen.

4.1 Analyse des existierenden Systems

Zunächst werden die für die Erweiterung zentralen Komponenten identifiziert und einer näheren Betrachtung unterzogen. Anschließend wird geprüft, wie die bislang unterstützten RDF-Datenspeicher angebunden sind. Zusätzlich wird untersucht, welche Maßnahmen zur Qualitätssicherung getroffen wurden. Abschließend werden die erkannten Probleme benannt.

4.1.1 Komponenten

Erfurt ist als Framework für Social Semantic Software konzipiert. Abbildung 4.1 zeigt eine Übersicht seiner Komponenten.

Obwohl die meisten Komponenten einzeln genutzt werden können, ist die `Applikation` der zentrale Einstiegspunkt in Erfurt. Diese Komponente kümmert sich um das Einlesen der Konfiguration, verwaltet Plugins, initialisiert Subkomponenten und bietet Zugriff auf diese.

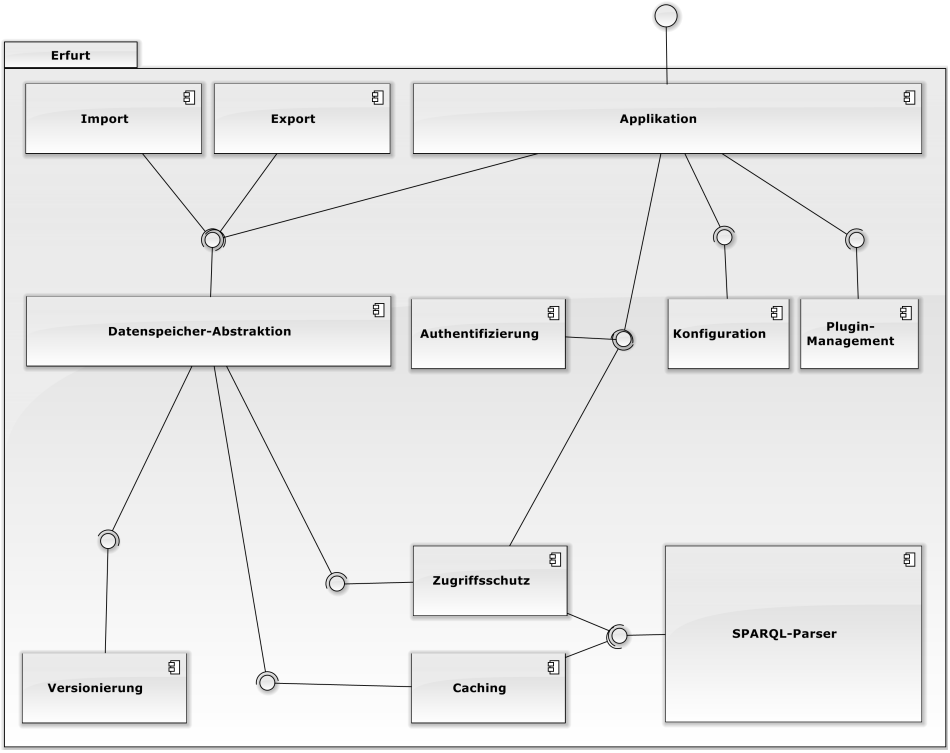


Abbildung 4.1: Komponenten im Erfurt-Framework.

Die `Datenspeicher-Abstraktion` bietet eine einheitliche API zum Zugriff auf verschiedene RDF-Datenspeicher, die mit sogenannten Adaptern angebunden werden. Die mitgelieferten Adapter erlauben die Verwendung der multifunktionalen Datenbank Virtuoso[1] sowie einer MySQL-Datenbank[2] als Datenspeicher. Da MySQL nativ keine RDF-Daten unterstützt, erfolgt hier ein Mapping auf das relationale Datenmodell. Als zentraler Zugriffspunkt kümmert sich die `Datenspeicher-Abstraktion` außerdem um Belange wie Versionierung, Caching und Zugriffsschutz auf Graph-Ebene. Die dazu verwendeten `Caching`- und `Zugriffsschutz`-Komponenten verlassen sich auf eine umfangreiche Komponente zum Analysieren und Umschreiben von SPARQL-Abfragen, den `SPARQL-Parser` [MUA10].

Auf der `Datenspeicher-Abstraktion` bauen weitere Komponenten, beispielsweise zum Daten-Import und -Export, auf. Damit verbunden ist das Parsen und Erstellen der verschiedenen Ein- und Ausgabeformate wie RDF/XML oder Turtle.

Für die Generalisierung der Persistenzschicht ist primär die

[1] http://virtuoso.openlinksw.com/
[2] http://www.mysql.com/

`Datenspeicher-Abstraktion` und deren Adapter-System relevant. Hier gilt es, die bestehenden Schnittstellen zu identifizieren und Abhängigkeiten zu anderen Komponenten herauszuarbeiten.

4.1.2 Datenspeicher-Abstraktion

Der zentrale Einstiegspunkt zum Zugriff auf die RDF-Datenspeicher-Funktionalität ist die `Erfurt_Store`-Klasse (im Folgenden kurz `Store` genannt), die in Abbildung 4.2 dargestellt ist.

Auf eine Instanz dieser Klasse lässt sich mittels `Erfurt_App` zugreifen. Die Klasse `Erfurt_App` ist als Singleton implementiert und fungiert als Registry für die Applikation, d. h. sie erzeugt und verwaltet alle wichtigen Komponenten in einer auf Erfurt aufbauenden Applikation [Fow02; GHJ+94]. Die vorhandene `reset`-Methode sowie die Parametrisierung der `getInstance`-Methode weisen darauf hin, dass die durch Anwendung des Singleton-Musters implizierte Einzigartigkeit des `Erfurt_App`-Objekts nicht gegeben ist.

Bei dem `Store` handelt es sich mit 42 öffentlichen Methoden und einer Länge von mehr als 2000 Zeilen um eine umfangreiche Klasse [BMM98]. Dies weist darauf hin, dass die Klasse mehrere Aufgaben erfüllt [FBB+99]. Tatsächlich kümmern sich 22 der Methoden um die Verarbeitung von SPARQL-Abfragen und RDF-Daten, 4 stehen mit dem Im- und Export in Verbindung und 9 kümmern sich um die Verwaltung von SQL-Belangen. Die restlichen Methoden dienen der allgemeinen Verwaltung (Setzen von Abhängigkeiten und Optionen etc.) oder der Installation. In den Methoden-Signaturen werden die Begriffe „Graph“ und „Model“ synonym verwendet.

Zum Durchführen seiner Operationen bedient sich der `Store` zahlreicher Subkomponenten, wie z. B. Cache, Versionierung und Logging. Diese bezieht er direkt über die `Erfurt_App`, wodurch eine enge Kopplung zu dieser Klasse und globale Abhängigkeiten entstehen. Die Subkomponenten verwendet der `Store` primär, um die Verwaltung von RDF-Daten um weitere Funktionalität, wie beispielsweise Caching, Logging und Zugriffsschutz, anzureichern.

Für die eigentliche Interaktion mit dem zugrunde liegenden RDF-Datenspeicher sind Objekte vom Typ `Erfurt_Store_Adapter_Interface` verantwortlich. Diese werden vom `Store` konfigurationsabhängig erzeugt, wobei davon ausgegangen wird, dass die das Interface implementierenden Klassen im Erfurt-Namespace existieren. Anschließend delegiert der `Store` Anfragen zum Einfügen und Auslesen von RDF-Daten an diesen Adapter. Nach Existenz-Prüfung werden teilweise Methoden aufgerufen, die nicht im Interface definiert sind. Zudem ist auffällig, dass SQL-Abfragen an diesen Adapter weitergeleitet werden, sofern er das Interface `Erfurt_Store_Sql_Interface` erfüllt. Die Implementierung dieses Interfaces wird von Erfurt nicht erzwungen. OntoWiki kann allerdings Grundfunktionen wie Versionierung und Caching nicht anbieten, wenn der verwendete Adapter das Interface nicht er-

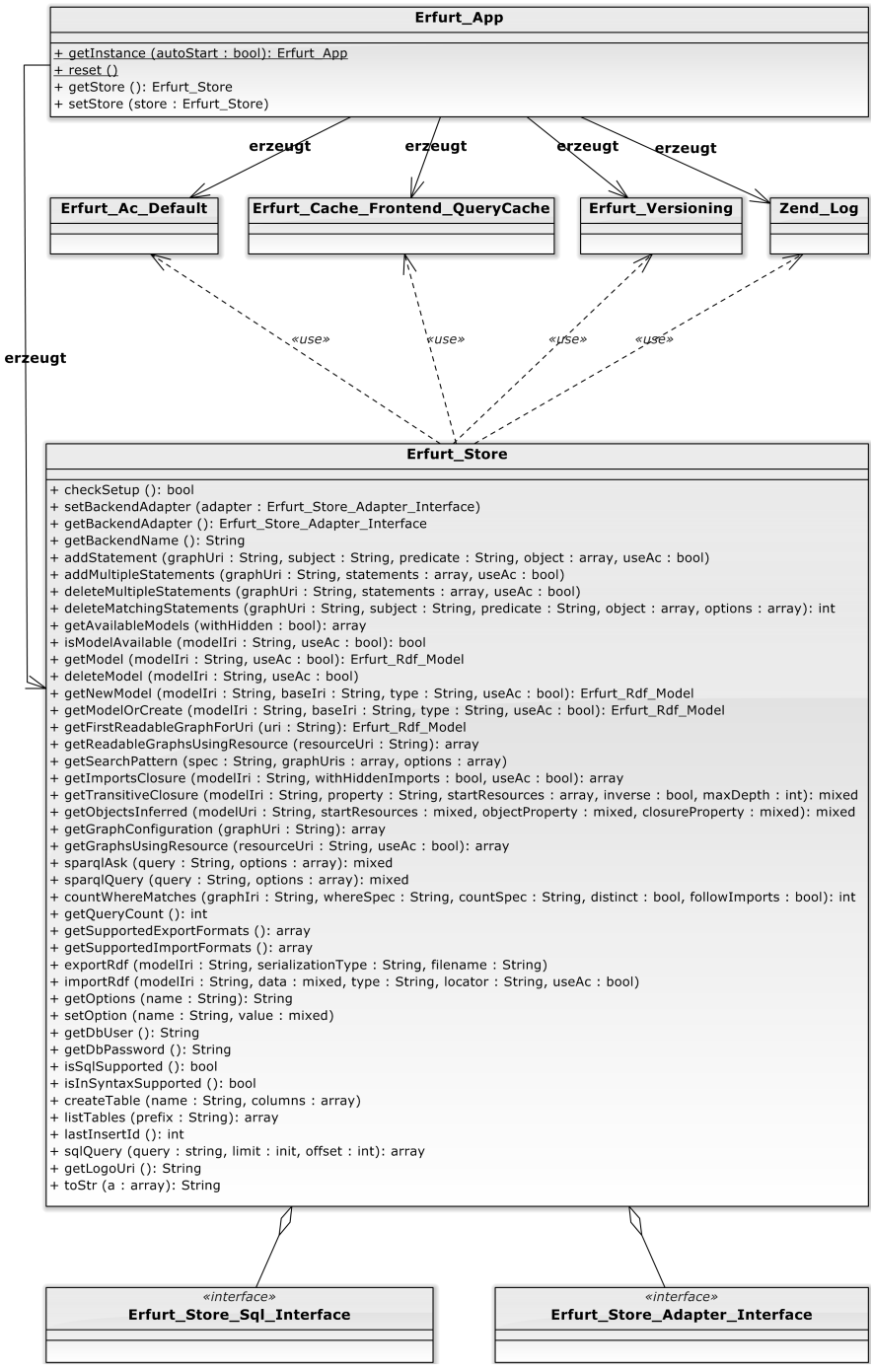


Abbildung 4.2: `Erfurt_Store` und relevante angrenzende Elemente.

füllt. Das bedeutet, dass die Adapter-Instanzen eine Doppelrolle bei der Verwaltung semantischer und relationaler Daten übernehmen müssen.

4.1.3 Existierende Adapter

Eine nähere Betrachtung der existierenden Adapter soll klären, welche Möglichkeiten zur Erweiterung der Persistenzschicht bestehen und wie diese genutzt werden.

Obwohl weitere Adapter existieren, werden lediglich die Virtuoso- und die MySQL-Implementierungen (`Erfurt_Store_Adapter_Virtuoso` und `Erfurt_Store_Adapter_EfZendDb`) zur näheren Analyse herangezogen, da die Inbetriebnahme nur für diese Systeme dokumentiert und der Status der anderen Implementierungen unklar ist[3]. Die Untersuchung dieser Adapter ist interessant, weil Virtuoso SPARQL-Abfragen nativ unterstützt, wohingegen der MySQL-Adapter auf dem relationalen Modell aufbaut.

Abbildung 4.3 zeigt die beiden Adapter in einem Klassendiagramm.

Virtuoso

Virtuoso ist eine multifunktionale Datenbank, die neben der Unterstützung von RDF-Daten Fähigkeiten zur Verarbeitung von relationalen Daten, XML und Dokumenten mitbringt. Für OntoWiki ist Virtuoso der empfohlene RDF-Datenspeicher.

Der Virtuoso-Adapter (`Erfurt_Store_Adapter_Virtuoso`) kommuniziert über einen ODBC-Treiber mit dem Server. Zur Kommunikation werden die Abfrage-Sprachen SPARQL und SQL verwendet. Virtuoso unterstützt SPARQL nativ. Daher ist der Aufwand zum Senden der entsprechenden Abfragen überschaubar. Da die `sparqlQuery`-Methode frei wählbare Ausgabeformate unterstützt, müssen die vom Datenbanksystem erhaltenen Ergebnisse mittels verschiedener `ResultConverter`-Klassen passend umgewandelt werden.

Neben den durch Interfaces vorgegebenen Methoden stellt der Virtuoso-Adapter weitere öffentliche Methoden bereit, die zum Teil interne Informationen preisgeben (z. B. `connection`) und außerhalb der Adapter-Klasse niemals aufgerufen werden. Andere Methoden wie `getImportsClosure` sind in keinem Interface definiert, werden aber vom `Store` aus Diagramm 4.2 benutzt.

MySQL

MySQL ist ein relationales Datenbank-Management-System, dessen Community Edition unter einer Open-Source-Lizenz zur Verfügung steht. Für

[3] https://github.com/AKSW/OntoWiki/wiki/Setup

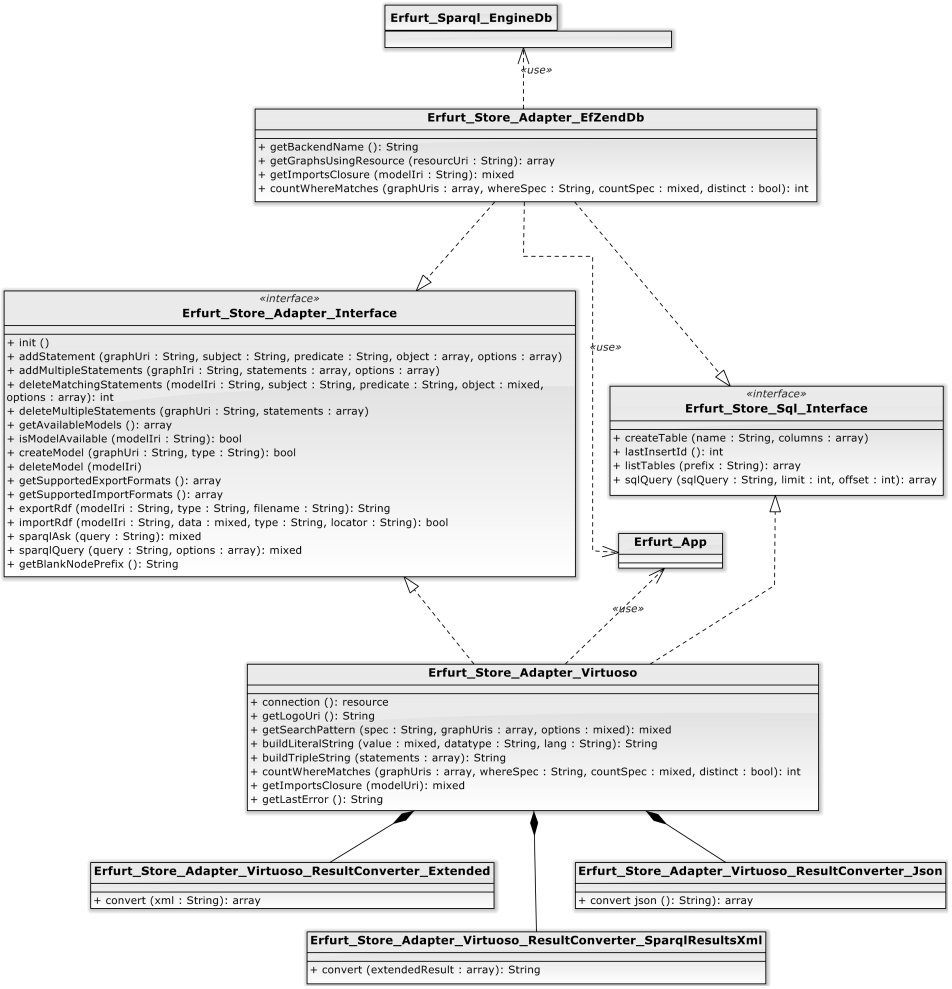


Abbildung 4.3: Klassendiagramm der im Erfurt-Framework bereits existierenden Adapter für Virtuoso und MySQL.

OntoWiki ist MySQL als RDF-Datenspeicher interessant, da es weit verbreitet[4] und daher auf vielen Systemen bereits vorhanden ist. Dadurch ist es mit diesem Adapter schnell möglich, eine Test-Installation von OntoWiki aufzusetzen.

Da MySQL nur SQL-Befehle verarbeiten kann, müssen SPARQL-Abfragen zunächst in SQL übersetzt werden. Dafür verwendet der Adapter `Erfurt_Store_Adapter_EfZendDb` Klassen aus dem

[4] Laut http://db-engines.com im Juni 2014. Die Website bezieht Suchmaschinentreffer, Suchmaschineneingaben, Anforderungen in Job-Angeboten, in professionellen Netzwerken angegebene Kenntnisse, Anzahl technischer Diskussionen sowie die Relevanz in sozialen Netzwerken bei der Berechnung der Rangfolge mit ein.

`Erfurt_Sparql_EngineDb`-Paket. Komponenten aus diesem Paket übernehmen zudem das Umformen der Abfrage-Ergebnisse in die unterstützten Formate.

Aufgrund des erhöhten Entwicklungsaufwands durch das Mapping von SPARQL auf SQL unterstützt der MySQL-Adapter noch nicht alle SPARQL-Features. Es fehlt beispielsweise die Möglichkeit zur Nutzung von Aggregat-Funktionen, die im SPARQL 1.1-Standard [HS13] definiert wurden.

Das Anlegen der benötigten Tabellen zum Speichern von RDF-Daten erfolgt automatisch, wenn während der Ausführung eines Befehls ein beliebiger Fehler auftritt. Dadurch ist keine manuelle Einrichtung der Datenbank-Strukturen notwendig.

Generelle Unzulänglichkeiten der Adapter

Der Virtuoso- und der MySQL-Adapter implementieren sowohl das Interface für die SPARQL-Verarbeitung (`Erfurt_Store_Adapter_Interface`) als auch das für die Verarbeitung von SQL-Befehlen (`Erfurt_Store_Sql_Interface`). Sie erfüllen die im Abschnitt 4.1.2 identifizierte Doppelrolle. Dies gelingt ihnen, da beide Datenspeicher eine native SQL-Schnittstelle anbieten.

Beide Adapter greifen exzessiv auf die durch die `Erfurt_App`-Klasse angebotenen Komponenten zu, um Querschnittsbelange wie Logging zu erfüllen. Dadurch entstehen globale Abhängigkeiten, welche die separate Verwendung oder das Erstellen von Tests für die Adapter erschweren.

Zum Import von Daten verlassen sich beide Implementierungen darauf, dass die Datenspeicher Zugriff auf das Dateisystem des Webservers haben. Dadurch entsteht zusätzlicher Aufwand zur Rechteverwaltung des Dateisystems. Außerdem erfordert es, dass Webserver und Datenspeicher auf dem gleichen System installiert sein müssen.

Zusätzlich zu den Interface-Methoden werden weitere öffentliche Methoden bereitgestellt. Diese enthalten teilweise optionale Funktionalität. Einige werden nicht vom übrigen Framework verwendet. Unter den Interface-Methoden finden sich ebenfalls einige, die niemals aufgerufen werden (z. B. `getBlankNodePrefix`). Die Funktionalität anderer ließe sich auf Methoden einer tiefer liegenden Abstraktionsschicht zurückführen (z. B. `isModelAvailable` auf `getAvailableModels` oder `addMultipleStatements` auf `addStatement`).

4.1.4 Automatisierte Tests

Um die korrekte Funktionalität der Adapter sicherzustellen, bringt Erfurt eine Reihe von Integrations-Tests mit. Die Tests lassen sich mit jedem korrekt konfigurierten Adapter ausführen und prüfen das Zusammenspiel des

Gesamtsystems.

Da die Tests das Gesamtsystem abdecken, sind Teilerfolge schwer prüfbar. Die Tests eignen sich gut, um in der Endphase der Entwicklung die Funktionalität eines Adapters zu prüfen, sind aber während der Entwicklung nur begrenzt hilfreich.

4.1.5 Identifizierte Probleme

Ausgehend von der vorangegangenen Analyse lassen sich folgende Probleme identifizieren:

1. **Globale Abhängigkeiten**
   Durch den Zugriff auf globale Komponenten via `Erfurt_App`, der vom `Store` und von den Adaptern ausgeht, entsteht ein unübersichtliches Geflecht von Abhängigkeiten. Die globalen Komponenten lassen sich schwer kontrollieren und austauschen, wodurch die Testbarkeit und damit langfristig die Verlässlichkeit leidet.
2. **Adapter müssen mit Erfurt ausgeliefert werden**
   Da der `Store` sich um die Erzeugung der Adapter kümmert und diese nur im Namespace des Erfurt-Frameworks sucht, ist es unmöglich Adapter hinzuzufügen ohne Erfurt zu modifizieren. Dies steht der Erweiterbarkeit entgegen und macht es externen Entwicklern unmöglich, eigene Adapter nicht invasiv in das System einzubinden.
3. **Vermischung von Produktiv- und Setup-Code**
   Die Vermischung der Funktionen für den laufenden Betrieb und die Installation macht es auf den ersten Blick einfach einen Adapter zu aktivieren, führt aber zu wenig intuitiven Ausführungspfaden und verletzt das „Principle of least Astonishment“ [SK09]. Dieses besagt, dass unerwartete Nebeneffekte vermieden werden sollten.

   Die fehlende Trennung der Betriebs- und Installations-Belange sorgt außerdem dafür, dass das Verhalten des Codes bei temporären Problemen (z. B. wenn der Datenspeicher nicht erreichbar ist) unklar ist [Mar08].
4. **Mehrere erforderliche Interfaces**
   Die Notwendigkeit, sowohl ein SPARQL- als auch ein SQL-Interface zu implementieren, erschwert die Erweiterung. Es ist unrealistisch anzunehmen, dass jeder RDF-Datenspeicher gleichzeitig die Fähigkeit zur Ausführung von SPARQL und SQL besitzt. Dadurch ist es deutlich schwerer, solche Datenspeicher als eigenständige Lösung anzubinden. Des Weiteren gibt es viele Hersteller-spezifische SQL-Features und subtile Unterschiede in den SQL-Dialekten. Diese werden vom aufrufenden Code nicht ausreichend berücksichtigt.
5. **Mangelnde Typsicherheit**
   Die bereits unter Punkt 4 thematisierte Trennung der SPARQL- und

SQL-Interfaces verschleiert zudem die Anforderungen an die Adapter-Objekte. Es existiert beispielsweise Code, der einen SPARQL-Adapter erwartet, aber Methoden aus dem SQL-Interface auf diesem aufruft. Dadurch kann es trotz Einhaltung der Methodensignaturen zu Fehlern kommen. Da PHP dynamisch typisiert ist, werden diese erst zur Laufzeit erkannt.

6. **Große Anzahl zu implementierender Methoden**
   Um funktionsfähig zu sein, muss ein Adapter 20 Methoden implementieren: 16 aus dem SPARQL-Interface und 4 aus dem SQL-Interface. Viele dieser Methoden sind mit Hilfe anderer abbildbar und stellen damit keine Voraussetzung für die Grundfunktionalität dar. Diese Vereinigung mehrerer Abstraktionsschichten und die Verletzung des „Single Responsibility"-Prinzips durch die Implementierung von SPARQL- und SQL-Belangen führt zu geringer Kohäsion in den Adapter-Klassen [Mar08]. Diese werden dadurch größer als nötig [Mar08], wodurch sich der Aufwand für die Entwicklung von Adaptern erhöht.
7. **Optionale Methoden**
   Die Verwendung optionaler Methoden, deren Existenz zur Laufzeit geprüft wird, führt zum einen zu einem Mangel an Dokumentation, da diese nicht in einem Interface aufgeführt werden und nur durch Studium des aufrufenden Codes ermittelt werden können. Zum anderen verhindert dieses Konstrukt den Einsatz von Entwurfsmustern wie Decorator (vgl. [GHJ+94]), da die optionalen Methoden von umhüllten Objekten für den aufrufenden Code nicht ermittelbar sind. Durch diese Einschränkung wird beispielsweise die transparente Behandlung von Querschnittsbelangen erschwert (vgl. Punkt 8).

   Ein weiteres Problem ist, dass die Einstufung einer Methode als „optional" schwer durchsetzbar ist. In Erfurt finden sich beispielsweise Stellen im Code, die keine Prüfung auf Existenz dieser Methoden durchführen und ein Vorhandensein stillschweigend annehmen.
8. **Wiederholte Implementierung von Querschnittsbelangen**
   Querschnittsbelange wie Logging oder die Messung der Abfragedauer werden in jedem Adapter separat implementiert. Dies führt dazu, dass die entsprechende Funktionalität nicht wiederverwendet oder zentral modifiziert werden kann.
9. **Verteilte Umwandlung von Abfrageergebnissen**
   Die Ergebnisse einer SPARQL-Abfrage können in drei Formaten abgerufen werden: Tabellarisch, als XML und in einer erweiterten Form, die Typ-Informationen enthält und an das SPARQL-JSON-Format [SCG+13] angelehnt ist. Die Umformung erfolgt in jedem Adapter separat. Wenn ein neues Format hinzugefügt werden soll, dann muss dieses von allen Adaptern unterstützt werden. Das stellt eine Verletzung des „Open/Closed"-Prinzips dar [Mar08]. Dieses besagt, dass

Software-Komponenten offen für Erweiterungen, aber geschlossen für Modifikationen sein sollen.

Ein brauchbarer Erweiterungsentwurf muss sich mit den vorgestellten Problemen auseinandersetzen und sie berücksichtigen. Es ist denkbar, dass nicht für jeden Punkt eine angemessene Lösung gefunden werden kann.

4.2 Erweiterung der Architektur

Die bestehende Infrastruktur sieht die Erweiterung des Frameworks via neuer Adapter vor. Es ist grundsätzlich denkbar, zusätzliche RDF-Datenspeicher mit Hilfe von neuen Adapter-Klassen anzubinden. Angesichts der ermittelten Probleme ist allerdings absehbar, dass dieses Vorgehen langfristig der Erweiterbarkeit und Wartbarkeit im Wege stehen würde.

Im Folgenden wird ein Grundkonzept zur Anbindung neuer RDF-Datenspeicher vorgestellt, das zur Wahrung der Abwärtskompatibilität eine Abstraktionsschicht unterhalb der bestehenden Adapter einführt. Anschließend werden optional verwendbare Komponenten präsentiert, die das Anbinden neuer Datenspeicher zusätzlich erleichtern sollen.

4.2.1 Grundkonzept für die Erweiterung

Um die Abwärtskompatibilität zu gewährleisten, soll das bestehende Adapter-Konzept grundsätzlich weiter verwendet werden. Allerdings sind Vereinfachungen notwendig, um die Erweiterbarkeit des Systems zu unterstützen.

Dazu wird ein generischer Adapter erstellt, der die Verbindung zur bisherigen Erfurt-Infrastruktur darstellt. Dieser ist in der Lage einen Großteil der geforderten Funktionalität durch Komposition von Grundfunktionen bereitzustellen. Die Ausführung der Grundfunktionen wird an eine tiefere Abstraktionsschicht delegiert. Da die Bezeichnung „Adapter“ belegt ist, soll für diese Schicht der Begriff des „Connectors“ geprägt werden. Zur Anbindung weiterer RDF-Datenspeicher genügt demzufolge das Hinzufügen einer Connector-Implementierung, welche die Grundfunktionalität für die Verwaltung von RDF-Daten (Einfügen und Löschen, Abfrage via SPARQL) bereitstellt. Abbildung 4.4 zeigt die notwendigen Komponenten in einem Klassendiagramm.

Trennung von SQL- und RDF-Funktionalität

Zur Auflösung des Zwangs, mehrere Interfaces implementieren zu müssen (Problem 4), wurde die `SparqlSqlCombination`-Klasse erstellt. Diese Klasse wurde nach dem Adapter-Muster entworfen [GHJ+94]. Sie implementiert beide Interfaces und delegiert Methodenaufrufe an zwei separate Objekte, die jeweils eine der Schnittstellen umsetzen müssen.

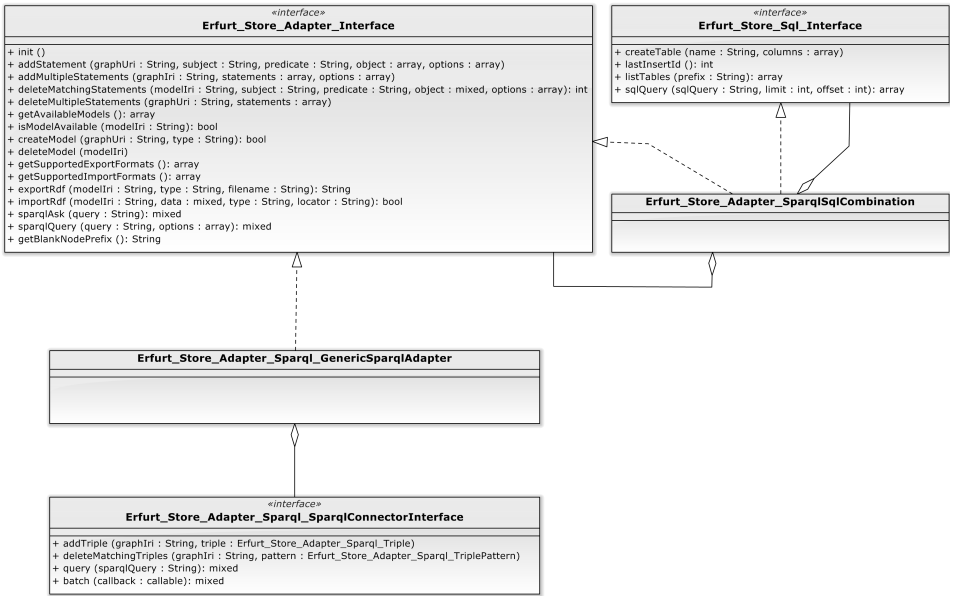


Abbildung 4.4: Konzept zur Anbindung weiterer Datenspeicher via Connector.

Diese Zwischenschicht erlaubt die beliebige Kombination von RDF- und relationalen Datenspeichern. Es lässt sich beispielsweise ein reiner RDF-Datenspeicher mit einer MySQL-Datenbank kombinieren, um alle Anforderungen von Erfurt abzudecken.

Die strikte Trennung der Verantwortlichkeiten erlaubt auf der neuen Abstraktionsschicht zudem die typsichere Programmierung (Problem 5).

Reduktion der Anzahl der zu implementierenden Methoden

Da der `GenericSparqlAdapter` viele der von Erfurt erwarteten Operationen auf eine kleine Menge von Grundfunktionen reduziert, muss ein Connector anstelle von 16 nur noch 4 Methoden implementieren (Problem 6).

Die Minimalfunktionalität wird von den drei Methoden `addTriple`, `deleteMatchingTriples` und `query` abgedeckt. Diese erlauben es, RDF-Daten einzufügen, zu löschen und via SPARQL auszulesen. Mit der Methode `batch` können Operationen zwecks Integrität und Performanz zusammengefasst werden. Dazu wird eine Referenz auf eine Funktion übergeben, die der Connector in einer Transaktion ausführen oder deren Basisoperationen er zusammenfassen kann.

Abschaffung optionaler Methoden

Alle nicht in Interfaces erfassten, optionalen Methoden konnten auf Grundfunktionen zurückgeführt werden. In Anlehnung an das Template-Method-Muster [GHJ+94] wurden sie als Ausweichlösung direkt im `Store` implementiert. Diese wird dann verwendet, wenn der Adapter keine eigene Implementierung bereitstellt. Das Problem 7 wird dadurch gelöst, da die optionalen Methoden bei der Umsetzung nicht mehr berücksichtigt werden müssen. Weil alle relevanten Methoden in Interfaces deklariert sind, wird eine Reihe weiterer Design Patterns anwendbar [GHJ+94]. Dies wurde z. B. bei der Trennung der RDF- und SQL-Funktionalität ausgenutzt.

Nur ein Ergebnisformat

Um zu vermeiden, dass jeder Connector sich um die Umwandlung von Abfrageergebnissen in eines der gewünschten Zielformate kümmern muss (Problem 9), wird ein einziges Ergebnisformat gewählt. Dieses muss von allen Connector-Implementierungen bei der Rückgabe von SPARQL-Abfrageergebnissen verwendet werden. Es wurde die erweiterte Form gewählt, da diese bereits in Teilen von Erfurt verwendet wird und im Gegensatz zur tabellarischen Form kein Informationsverlust vorliegt.

Die Umformung in das gewählte Zielformat wird schließlich vom `GenericSparqlAdapter` durchgeführt. Dieser kann das vom Connector gelieferte, erweiterte Format in alle Formate überführen, die in Erfurt verwendet werden. Folglich muss jeder Connector lediglich noch ein Ergebnisformat kennen und neue Formate müssen nur in den `GenericSparqlAdapter` integriert werden, wodurch eine kombinatorische Explosion bei der Handhabung der Formate verhindert wird.

Getrennte Implementierung von Querschnittsbelangen

Durch die Vereinfachung der Komponenten lassen sich Querschnittsbelange besser handhaben (Problem 8). Abbildung 4.5 zeigt beispielhaft eine Komponente zum Loggen langsamer SPARQL-Abfragen, die als Decorator implementiert ist [GHJ+94]. Mit dieser ist es möglich, einen beliebigen Connector zu umhüllen und die Ausführungszeit der Abfragen zu messen. Der Code zum Loggen bleibt wiederverwendbar und das Vorhandensein dieser Zwischenschicht ist für den inneren Connector völlig transparent. Durch das Entfernen des Decorators lässt sich das Feature außerdem vollständig Deaktivieren, was der Performanz zugute kommt.

Erzeugung von Adaptern

Um Adapter außerhalb des Erfurt-Namespaces erzeugen zu können (Problem 2) und damit die Erweiterbarkeit durch externe Entwickler zu gewähr-

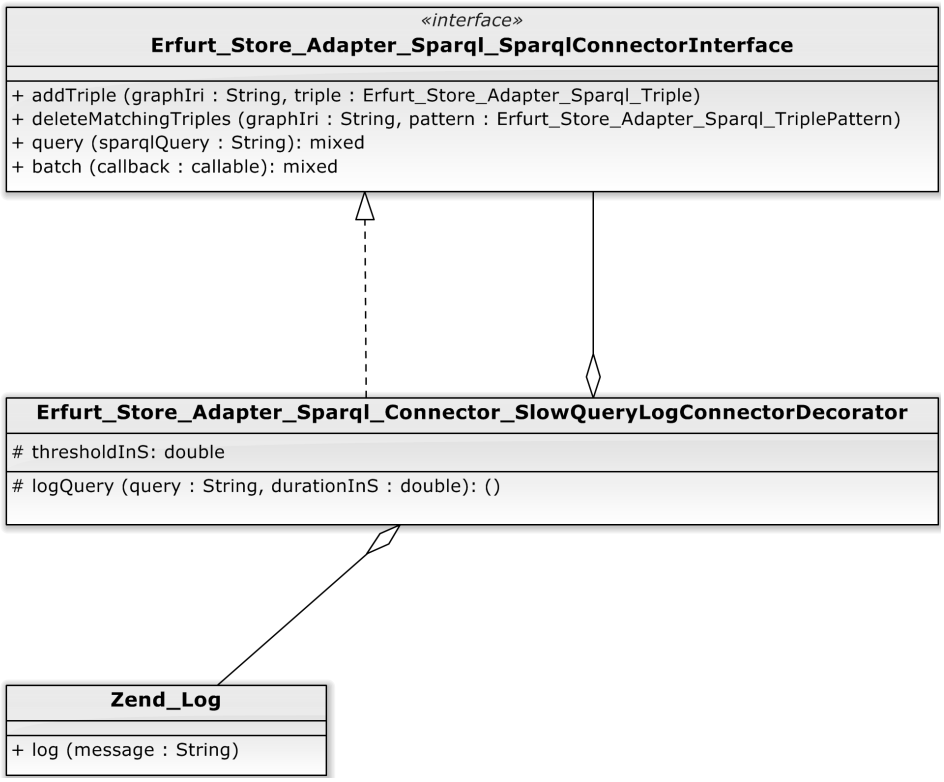


Abbildung 4.5: Implementierung von Querschnittsbelangen mittels Decorator am Beispiel des Loggings von langsamen Abfragen.

leisten, wurde der `Store` zunächst so angepasst, dass er Factory-Implementierungen zur Erstellung von Adaptern zulässt [GHJ+94].

Darauf aufbauend wurde eine Factory konzipiert (Abbildung 4.6), die einen Dependency Injection Container[5] verwendet, um beliebige Adapter zu erzeugen. Das Dependency Injection Pattern (auch unter Inversion of Control bekannt) erlaubt die Konfiguration von Abhängigkeiten in einem Objekt-Graph [Fow04]. Die Konfiguration der erzeugten Objekte bietet Kontrolle über die eingesetzten Klassen, wodurch diese nicht mehr zwangsläufig aus Erfurt stammen müssen. Außerdem konnte das Problem der globalen Abhängigkeiten (Problem 1) gelöst werden, da diese jetzt konfiguriert und über den Konstruktor oder Setter-Methoden explizit zugewiesen werden.

Mit den Features des verwendeten Dependency Injection Containers lassen sich Setup (z. B. das Anlegen von Datenbank-Tabellen) und ausführender Code in separate Klassen trennen (Problem 3): Für die Installation verantwortliche Objekte müssen das `SetupInterface` implementieren. Sie wer-

[5] http://symfony.com/doc/current/components/dependency_injection/introduction.html

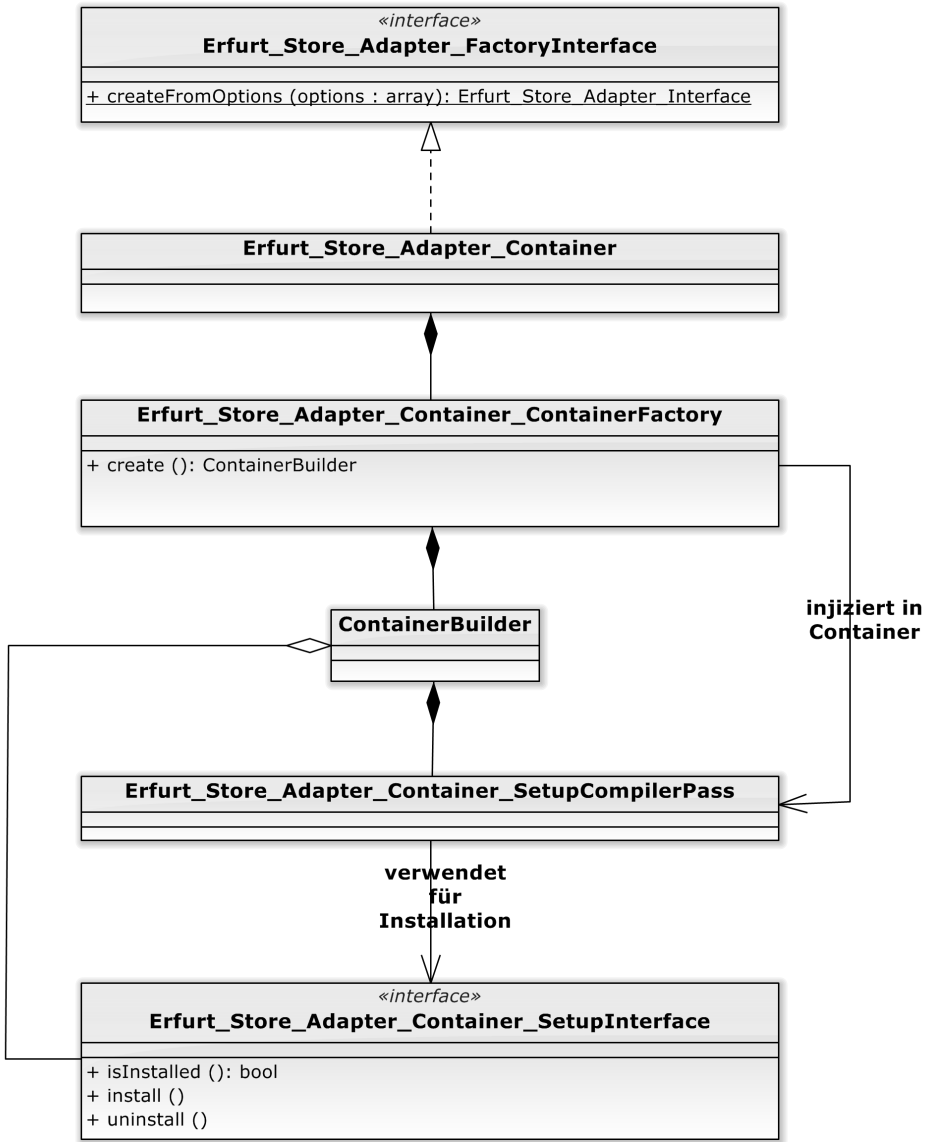


Abbildung 4.6: Aufbau der Factory, die zum Erzeugen eines Adapters einen Dependency Injection Container verwendet.

den im Container konfiguriert und mit einem speziellen Tag versehen. Beim Kompilieren des Containers, d. h. dem Übersetzen der Konfigurationsdatei in PHP-Code, ermittelt ein sogenannter CompilerPass diese Objekte und verwendet sie, um bei Bedarf die Installation durchzuführen. Da der Container nur dann kompiliert wird, wenn sich die zugrunde liegende Konfiguration ändert, findet die Installation nicht mehr im Fehlerfall statt, sondern wird lediglich bei einer Neukonfiguration gestartet.

4.2.2 Infrastruktur für die Anbindung von RDF-Datenspeichern

Das vorgestellte Konzept stellt das Grundgerüst für die Anbindung weiterer RDF-Datenspeicher dar. Darauf aufbauend sollen Maßnahmen ergriffen und zusätzliche Komponenten bereitgestellt werden, welche die Entwicklung neuer Connector-Implementierungen erleichtern.

Die Nutzung dieser Komponenten soll optional bleiben, damit die Einfachheit des Grundkonzepts erhalten bleibt. Es soll möglich werden, neue RDF-Datenspeicher initial schnell und minimalistisch anzubinden und die Implementierung anschließend mit Hilfe der Komponenten zu verbessern (z. B. in Hinblick auf die Performanz oder Wiederverwendbarkeit).

Im Folgenden werden die Maßnahmen und Komponenten vorgestellt, welche die Anbindung neuer Datenspeicher erleichtern sollen.

Verwaltung von Abhängigkeiten

Ursprünglich wurden benötigte Bibliotheken (im Wesentlichen das Zend Framework 1[6]) bei der Installation von Erfurt heruntergeladen, entpackt und dem System zur Verfügung gestellt. Zum Verwenden neuer Bibliotheken mussten die zugrunde liegenden Build-Skripte angepasst und der dahinter liegende Prozess verstanden werden. Es war notwendig eine passende Quelle (z. B. ein herunterladbares Archiv oder das Versionskontrollsystem) zu finden, den Download-Prozess zu automatisieren und die Bibliothek in Erfurt zu integrieren. Dadurch wurde die Verwendung fremder Bibliotheken erschwert.

Die heutzutage frei verfügbaren Bibliotheken haben aufgrund des jahrelangen Einsatzes in der Praxis und der Vielzahl an Entwicklern, die diese weiterentwickeln und überprüfen, häufig eine höhere Qualität als sie von Einzelpersonen mit limitierter Zeit erreicht werden könnte. Die Verwendung erprobter Software ist ein zentraler Bestandteil, wenn es darum geht die Verlässlichkeit der gesamten Implementierung zu verbessern. Bei der Erweiterung von Erfurt wurden zahlreiche Fremdbibliotheken zum Lösen von Teilaufgaben (z. B. zur Kommunikation mit Datenbanken oder über HTTP) verwendet.

Um die Verwaltung dieser Abhängigkeiten zu erleichtern wurde Composer[7] verwendet, der sich in der PHP-Welt zum Quasi-Standard zur Installation von Bibliotheken entwickelt hat. Composer ist ein Tool zur Verwaltung von Abhängigkeiten und mit Bundler[8] (Ruby), npm[9] (Node.js) oder

[6] https://github.com/zendframework/zf1
[7] https://getcomposer.org/
[8] http://bundler.io/
[9] https://www.npmjs.org/

Maven[10] (Java) vergleichbar. Es greift auf einen zentrales Repository[11] zu, um automatisch zu ermitteln, woher eine Bibliothek bezogen werden kann. Anschließend installiert er diese und kümmert sich um deren Abhängigkeiten. Die heruntergeladenen Klassen werden der Applikation per Autoloading[12] zur Verfügung gestellt, d. h. PHP lädt diese bei der ersten Verwendung automatisch nach.

Um eine Fremdbibliothek zu verwenden, muss diese unter Angabe der gewünschten Version in die Konfigurationsdatei `composer.json` eingetragen werden. Anschließend werden bei der Installation alle noch fehlenden Komponenten automatisch geladen.

System zur Ergebniskonvertierung

Obwohl die Anzahl der notwendigen Konvertierungen von Abfrageergebnissen durch Einführung eines zentralen Formats signifikant reduziert werden konnte (vgl. Abschnitt 4.2.1), ist zumindest die Umformung der Ergebnisse in dieses Übergangsformat für jeden Connector relevant.

Um die Transformationen zwischen den Formaten wiederverwenden zu können, wurde die vom Virtuoso-Adapter bereitgestellte Infrastruktur zur Umformung ausgebaut. Abbildung 4.7 zeigt die zur Verfügung stehenden Konverter. Durch das Einführen eines gemeinsamen Interfaces

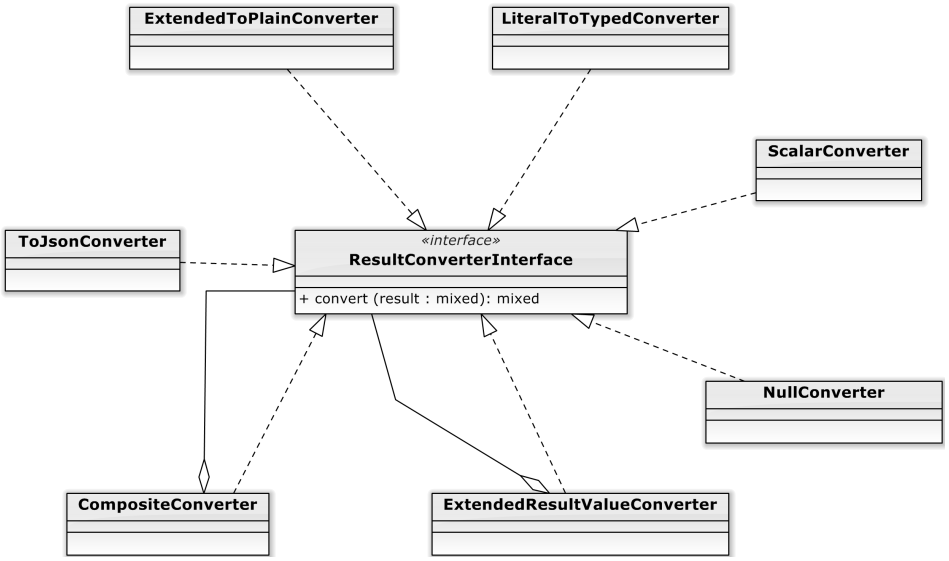


Abbildung 4.7: Infrastruktur zum Umformen von Abfrageergebnissen.

[10] http://maven.apache.org/
[11] https://packagist.org/
[12] http://www.php.net/manual/en/language.oop5.autoload.php

(`ResultConverterInterface`) werden Entwurfsmuster wie Decorator oder Composite möglich [GHJ+94]. Dem „Pipes and Filters“-Konzept folgend lassen sich Konverter in Reihe schalten, um über Zwischenformate ein gewünschtes Zielformat zu erzeugen [HW03].

Die zur Verfügung gestellten Implementierungen bieten zahlreiche Möglichkeiten, beispielsweise die Umformung in gängige Ergebnisformate wie JSON oder die tabellarische Form sowie die Konvertierung von Ergebniswerten in ihre entsprechenden PHP-Typen. Spezielle Transformationen lassen sich häufig durch die Komposition bestehender Umformungen oder minimale Erweiterungen erreichen.

Gruppierung von Speicheroperationen

Zur Verbesserung der Performanz beim Einfügen von Daten ist das Zusammenfassen von Operationen von zentraler Bedeutung. Um diese Aufgabe zu erleichtern wird ein Puffer-System bereitgestellt (vgl. Abbildung 4.8), welches das Verzögern und gruppierte Ausführen von Speicheroperationen erlaubt.

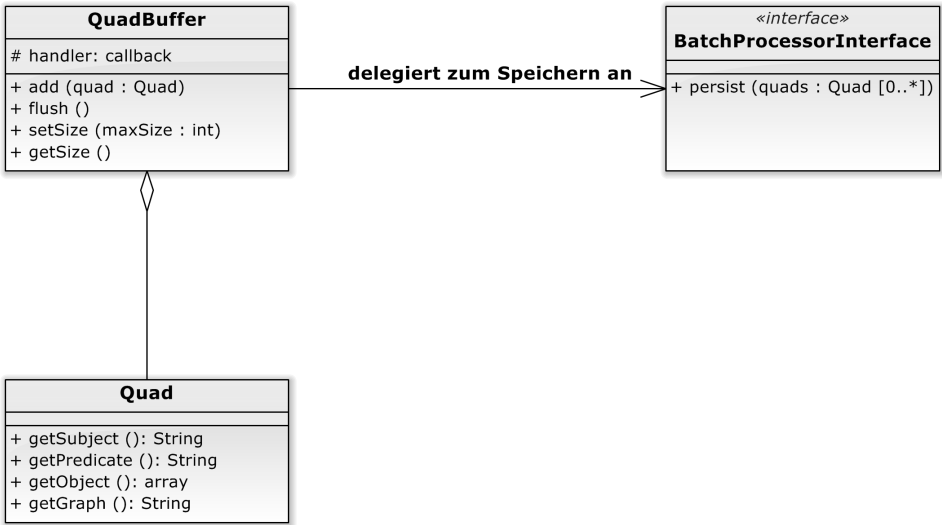


Abbildung 4.8: Puffer-System zum Verzögern und Zusammenfassen von Speicheroperationen.

Da auf dieser Abstraktionsebene jedem Tripel ein Graph zugeordnet wird, spricht man beim Speichern von `Quads`. Anstatt die Quads direkt zu speichern, werden diese zunächst in einen `QuadBuffer` eingefügt [HW03]. Dieser entscheidet anhand seiner eingestellten Maximalgröße, wie mit den derzeit vorhandenen Quads zu verfahren ist. Sobald die Maximalgröße erreicht ist, werden alle gepufferten Quads an eine Objekt-Instanz vom

Typ `BatchProcessorInterface` (Strategy-Muster nach [GHJ+94]) weitergereicht und der Puffer wird geleert. BatchProcessor-Implementierungen müssen für jeden Connector erstellt werden und sind für das möglichst effiziente Einfügen einer variablen Anzahl von Quads verantwortlich.

Im Normalbetrieb wird die Maximalgröße des Puffers auf 1 gesetzt, wodurch Speicheroperationen sofort ausgeführt werden. Im Batch-Modus kann die Maximalgröße erhöht werden, um die Einfügeoperationen automatisch zu gruppieren. Die optimale Größe hängt davon ab, welche Anzahl von Quads der jeweilige RDF-Datenspeicher am besten verarbeiten kann. Die Größe des Puffers ist durch den verfügbaren Arbeitsspeicher limitiert. Hier muss unter Umständen eine Abwägung zwischen Geschwindigkeit und Speicherverbrauch getroffen werden.

Bereitgestellte Testfälle

Zur Qualitätssicherung wurden bei der Anbindung der verschiedenen RDF-Datenspeicher Tests auf Basis von PHPUnit[13] erstellt. Durch das Entfernen der globalen Abhängigkeiten wurden isolierte Connector-Tests möglich, die die vorhandenen Integrations-Tests auf Applikations-Ebene ergänzen. Der Test-First-Ansatz sorgt für ein sauberes Klassen-Design. Die Tests verhindern außerdem das Wiederauftreten von Fehlern und ermöglichen die sichere Durchführung von Refactorings [Bec02].

Um bei der Anbindung neuer Datenspeicher unmittelbar von diesen Vorteilen profitieren zu können, wurde eine Sammlung grundlegender Integrations-Tests extrahiert, die von jedem Connector erfüllt werden sollten. Diese vorgefertigten Tests können durch Erweiterung des Basis-Testfalls bei jeder Neuentwicklung verwendet und zusätzlich um Connector-spezifische Tests erweitert werden. So kann ein einheitliches Basisverhalten aller Connector-Implementierungen sichergestellt werden. Durch die Verwendung des Continuous Integration-Dienstes Travis[14] kann die Ausführung der meisten Tests in verschiedenen Laufzeit-Umgebungen sichergestellt werden. Im Fehlerfall erhält der Entwickler über den Dienst schnelles Feedback [DMG07].

Benchmark-Infrastruktur

Analog zu den Testfällen wird eine Reihe von wiederverwendbaren Benchmarks bereitgestellt, die auf Basis des Frameworks Athletic[15] erstellt wurden.

Die Benchmark-Situationen sind auf spezielle Aspekte der Connector-Funktionalität zugeschnitten (z. B. das Einfügen oder das Auslesen großer Abfrageergebnisse) und nicht direkt mit realen Last-Situationen vergleich-

[13] http://phpunit.de/
[14] https://travis-ci.org/
[15] https://github.com/polyfractal/athletic

bar. Der gezielte Fokus auf einzelne Bereiche sorgt jedoch dafür, dass die Benchmarks sich gut zur Optimierung der jeweiligen Teilbereiche eignen.

Kapitel 5

Anbindung neuer Datenspeicher

Dieses Kapitel gibt einen Überblick über die neu hinzugefügten RDF-Datenspeicher zum Persistieren und Abfragen semantischer Daten. Kapitel 5.1 behandelt den Oracle- und Kapitel 5.2 den Stardog-Connector. Die Anbindung der Graph-Datenbank Neo4j wird in Kapitel 5.3 beschrieben.

Jedes dieser Kapitel beschreibt, wie der jeweilige RDF-Datenspeicher intern funktioniert und wie dieser an Erfurt angebunden wurde. Zudem werden während der Entwicklung aufgetretene Probleme dargestellt. Abschließend wird dargelegt, welche Performanz-Optimierungen während der Implementierung des jeweiligen Connectors durchgeführt wurden und wie sich diese ausgewirkt haben.

5.1 Oracle

5.1.1 Einordnung

Seit Version 10g Release 2 bietet die relationale Datenbank von Oracle, einem der führenden kommerziellen Datenbank-Hersteller[1], Unterstützung für die Speicherung von RDF-Daten. Zur Nutzung dieser Fähigkeiten wird die Enterprise Edition der Datenbank zusammen mit der Partitioning- und Spatial-Option benötigt.

Die vorliegende Version 12c (Release 12.1.0.1.0) enthält weitere Verbesserungen an der RDF-Komponente und dient als Grundlage, um die Oracle Datenbank als RDF-Datenspeicher an Erfurt anzubinden. Die große Präsenz von Oracle-Systemen in traditionellen IT-Landschaften macht diese relationale Datenbank zu einem Kandidaten für die Erfüllung der Anforderung 1a.

[1] Laut http://db-engines.com im Juni 2014.

5.1.2 Funktionsweise

In einer Oracle Datenbank mit *Spatial and Graph*-Option werden RDF-Daten auf das relationale Modell abgebildet. Sogenannte Models enthalten Tripel, deren URI- und Literal-Werte von Oracle in einem zentralen Daten-Pool verwaltet werden. Mit Hilfe bereitgestellter Stored Procedures können Models angelegt, verwaltet und abgefragt werden. Die den Models hinzugefügten Tripel können einem spezifischen Graphen zugeordnet werden. Inhaltlich sind Models voneinander unabhängig: Tripel können hinzugefügt und gelöscht werden, ohne andere Models zu beeinflussen.

Interne Repräsentation

Abbildung 5.1 zeigt die Struktur aus Tabellen und Views, die verwendet wird, um semantische Daten in der Datenbank zu speichern. Das dargestell-

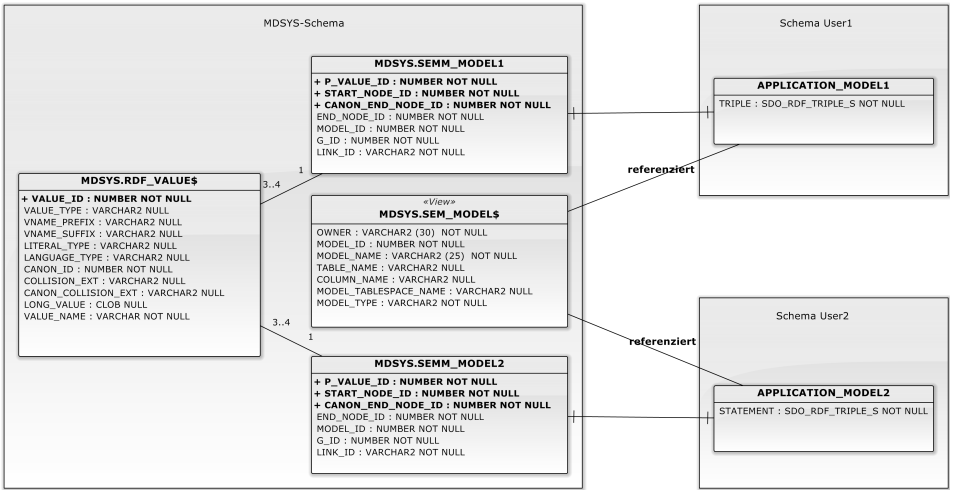


Abbildung 5.1: Speicherung von RDF-Daten in einer Oracle Datenbank nach [Mur14].

te Beispiel zeigt zwei Models, `MODEL1` und `MODEL1`. Zu jedem Model existiert eine Tripel-Tabelle, über die semantische Daten hinzugefügt und entfernt werden können. Diese Tabellen werden beim Erstellen eines Models angegeben. Anschließend ist die Datenbank dafür verantwortlich, deren Inhalt mit der internen Repräsentation der semantischen Daten synchron zu halten.

In der Darstellung werden die Tabellen `APPLICATION_MODEL1` und `APPLICATION_MODEL2` verwendet, die jeweils in einem separaten Schema angelegt wurden. Voraussetzung für die Verwendung als Tripel-Tabelle ist das Vorhandensein einer Spalte vom Typ `SDO_RDF_TRIPLE_S`. Der Name dieser Spalte kann beliebig gewählt werden. Hier wurden die Bezeichnungen `TRIPLE` und `STATEMENT` gewählt.

Die interne Speicherung von semantischen Daten erfolgt im Schema `MDSYS`. Die View `MDSYS.SEM_MODEL$` enthält Informationen über alle angelegten Models und hält mit Hilfe der Spalten `TABLE_NAME` und `COLUMN_NAME` eine Referenz auf die zugehörigen externen Tripel-Daten. Im `MDSYS`-Schema wird für jedes Model eine weitere Tabelle angelegt. `MDSYS.SEMM_MODEL1` und `MDSYS.SEMM_MODEL2` enthalten die Tripel aus den externen Tabellen in Form von Referenzen auf die Tabelle `MDSYS.RDF_VALUE$`. Diese enthält jede URI und jedes Literal, das in den Tripeln vorkommt, genau einmal.

Hinzufügen von Daten

Das Hinzufügen von Tripeln erfolgt mittels SQL unter Zuhilfenahme des bereitgestellten Typs `SDO_RDF_TRIPLE_S` über die externe Datentabelle. Die folgende Abfrage fügt dem Model `MODEL1` ein Tripel hinzu, das dem Graph *http://www.example.org/graph* zugewiesen ist:

```
INSERT INTO APPLICATION_MODEL1 (TRIPLE)
VALUES (
    SDO_RDF_TRIPLE_S(
        'MODEL1:<http://www.example.org/graph>',
        '<http://www.example.org/subject>',
        '<http://www.example.org/predicate>',
        '"Object literal."'
    )
);
```

Das hinzugefügte Tripel hat folgende Form:

```
<http://www.example.org/subject> <http://www.example.org/predicate>
    ↪ "Object literal." .
```

Das Löschen erfolgt analog mittels DELETE-Abfrage. Durch die Verwendung der vom `SDO_RDF_TRIPLE_S`-Typ bereitgestellten Eigenschaften und Methoden kann gezielt gefiltert werden.

Abfrage von Daten

Der Abruf der semantischen Daten erfolgt mit Hilfe von SPARQL-Abfragen. Da Oracle keine separate Schnittstelle für diese bereitstellt, wird die Stored Procedure `SEM_MATCH` verwendet, über welche die SPARQL-Abfrage als String übermittelt wird. Die Rückgabe lässt sich mit Hilfe einer äußeren SELECT-Abfrage wie eine Tabelle abfragen. Listing 5.1 zeigt die Verwendung der Prozedur zur Abfrage von Daten aus dem Model `MODEL1`.

Neben den angefragten Variablen `subject` und `object` werden Metadaten zu diesen geliefert. Darunter fallen z. B. die Typen der RDF-Literale oder die internen IDs der gelieferten Werte.

Eine weitere Besonderheit ist, dass die SPARQL-Abfrage nicht als Parameter eines Prepared Statements übergeben werden kann, da die Datenbank die Abfrage in SQL übersetzt und beim Kompilieren zur Erstellung

Listing 5.1: Abfragen von semantischen Daten aus einer Oracle Datenbank mittels SPARQL.

```
SELECT *
FROM TABLE(
    SEM_MATCH(
        'SELECT ?subject ?object
         WHERE {
             ?subject <http://example.org/knows> ?object .
         }',
        SEM_MODELS('MODEL1'),
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL
    )
);
```

eines Query Plans einbezieht. Das bedeutet, dass bei der Übermittlung von SPARQL-Abfragen besondere Vorsicht geboten ist, da die zur Verhinderung von SQL-Injection-Angriffen notwendige Maskierung von Eingabeparametern manuell vorgenommen werden muss.

5.1.3 Anbindung

Die Kommunikation mit dem RDF-Datenspeicher von Oracle erfolgt über SQL. Zur Anbindung von Erfurt war daher der Aufbau einer Datenbankverbindung erforderlich. Zum Herstellen dieser Verbindung aus PHP kamen drei Komponenten in Frage:

- ODBC[2]
- PHP Data Objects (PDO)[3]
- OCI8-Extension[4]

Die *ODBC*-Komponente ist seit vielen Jahren verfügbar und wird direkt mit PHP ausgeliefert. Aufgrund ihres Alters ist von einer gewissen Stabilität auszugehen. Sie bietet lediglich einen prozeduralen Zugang. Zur Verwendung sind jeweils datenbankspezifische Treiber notwendig. Da allerdings keine offiziellen Treiber für Oracle Datenbanken verfügbar sind, bietet sich die Verwendung von ODBC nicht an[5].

[2] http://php.net/manual/en/book.uodbc.php
[3] http://php.net/manual/en/book.pdo.php
[4] http://php.net/manual/en/book.oci8.php
[5] http://php.net/manual/en/intro.uodbc.php

Die *PHP Data Objects*-Extension steht seit Ende 2005 mit PHP 5.1 zu Verfügung und ist damit vergleichsweise neu. Die Komponente bietet einen einheitlichen, objektorientierten Zugriff auf die Datenbanken verschiedener Hersteller. Treiber für zahlreiche Datenbanken werden direkt mit PHP ausgeliefert. Allerdings ist der Treiber für Oracle als experimentell gekennzeichnet und in der Praxis sind diverse Probleme bekannt[6]. Da Stabilität und Robustheit wesentliche Anforderungen an die Connector-Komponenten sind, kommt PDO hier nicht zum Einsatz.

Mit *OCI8* steht eine PHP-Extension zur Verfügung, die speziell für den Einsatz mit Oracle Datenbanken konstruiert wurde. OCI8 bietet ein größtenteils prozedurales Interface zum Datenzugriff. Neben der eigentlichen Extension ist die Installation des *Oracle Instant Client*[7] auf dem Client-Rechner erforderlich. Der Zugriff über OCI8 wird von Oracle empfohlen, weswegen diese Methode trotz zusätzlicher Abhängigkeiten und des prozeduralen Interfaces gewählt wurde [JH06].

Um trotz OCI8-Extension einen objektorientierten Zugang zur Datenbank zu erhalten, wird die DBAL-Komponente (Database Abstraction Layer) aus der Bibliothek Doctrine 2[8] verwendet. Doctrine 2 DBAL bietet unter Verwendung verschiedenster APIs (unter anderem OCI8) eine einheitliche Schnittstelle für zahlreiche SQL-Datenbanken. Durch die Kapselung von Implementierungsdetails entfallen manuelle Tätigkeiten, beispielsweise die Verwaltung von Verbindungs-Ressourcen und die Fehlerbehandlung auf der systemnahen Abstraktionsebene.

Abbildung 5.2 zeigt das grobe Konzept für die Anbindung des RDF-Datenspeichers von Oracle an die von Erfurt bereitgestellte Infrastruktur. Der Oracle-spezifische Connector `OracleSparqlConnector` implementiert das `SparqlConnectorInterface`, das die Schnittstelle zu den restlichen Komponenten von Erfurt darstellt. Sämtliche Details der Implementierung werden hinter dieser Schnittstelle verborgen und sind für die anderen Subsysteme der Bibliothek nicht von Bedeutung.

Über die Datenbankverbindung `DatabaseConnection` erfolgt die eigentliche Kommunikation mit der Oracle Datenbank. Um die benötigten Ressourcen zur Laufzeit gering zu halten, wird diese Verbindung höchstens einmal pro Request aufgebaut. Da diverse Hilfskomponenten diese Verbindung ebenfalls benötigen, steht sie dem SPARQL-Connector nicht exklusiv zur Verfügung. Dies stellt prinzipiell kein Problem dar, verringert aber den Grad der Isolation und sollte bei der Implementierung berücksichtigt werden. Es muss zum Beispiel auf eine genaue Verwaltung der Transaktionen geachtet werden: Im Fehlerfall dürfen diese nicht offen zurückbleiben.

[6] Siehe http://php.net/manual/en/ref.pdo-oci.php sowie http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/configuration.html#driver.

[7] http://www.oracle.com/technetwork/database/features/instant-client/index-097480.html

[8] http://www.doctrine-project.org/projects/dbal.html

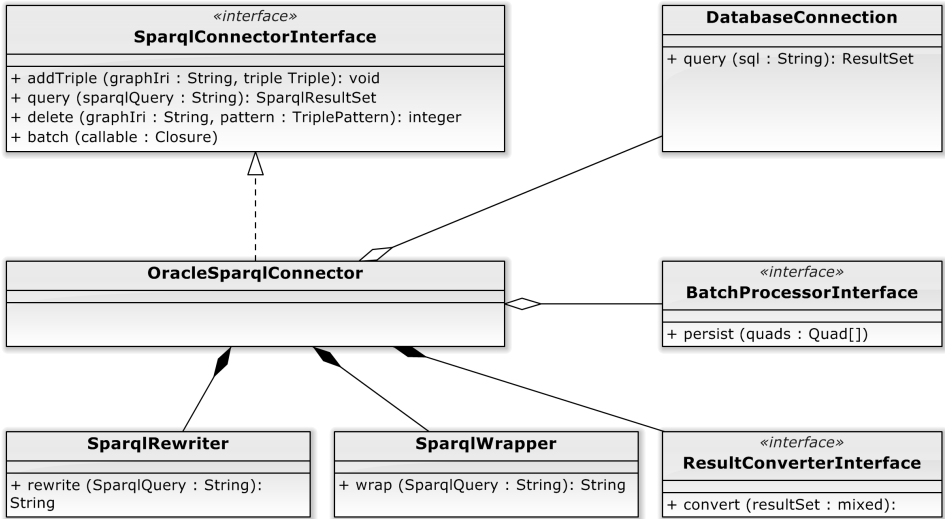


Abbildung 5.2: SPARQL-Connector für Oracle mit angrenzenden Komponenten.

Konkrete Implementierungen des `BatchProcessorInterface` werden vom Connector verwendet, um Tripel im RDF-Datenspeicher abzulegen. Es kann eine an die Betriebsumgebung angepasste Implementierung verwendet werden. Diese kann sich zum Beispiel an der Topologie der Daten oder an den zur Verfügung stehenden Berechtigungen orientieren. Der Abschnitt 5.1.5 geht im Zuge der Performanz-Optimierung näher die zur Verfügung stehenden Implementierungen ein.

Die Verarbeitung von SPARQL-Abfragen erfolgt in mehreren Schritten. Das Aktivitätsdiagramm in Abbildung 5.3 verdeutlicht diese. Da die Oracle-

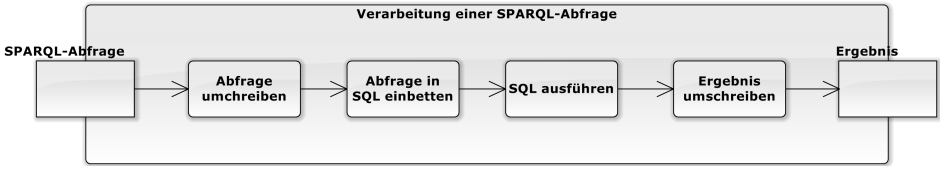


Abbildung 5.3: Verarbeitung einer SPARQL-Abfrage durch den Oracle-Connector.

Datenbank nicht alle Formen von SPARQL-Abfragen verarbeiten kann, werden diese nach Eingang zunächst durch den `SparqlRewriter` umgeschrieben. Diese Transformation umfasst z. B. die Umbenennung von Variablen

und die Normalisierung der in der Abfrage enthaltenen Literale. Anschließend wird die transformierte Abfrage vom `SparqlWrapper` in einen SQL-Befehl eingebettet. In dieser Form ist es schließlich möglich, die Abfrage zur Ausführung an die Datenbank weiterzureichen. Das erhaltene Ergebnis hat die bei SQL-Abfragen übliche Tabellenform und enthält neben den per SPARQL angeforderten Daten auch Meta-Informationen über diese. Darunter fallen z. B. Typangaben für Literale und IDs, die auf die interne RDF-Datentabelle verweisen. Um dieses Ergebnis in die von Erfurt erwartete Form zu bringen, wird eine anhand des SPARQL-Abfragetyps gewählte `ResultConverterInterface`-Implementierung verwendet. Nach der Transformation des Ergebnisses in das vom `SparqlConnectorInterface` geforderte Format kann eine Weiterverarbeitung durch die Erfurt-Infrastruktur erfolgen.

5.1.4 Probleme bei der Umsetzung

Bei der Anbindung der Oracle-Datenbank traten im Vergleich zu der Integration der anderen RDF-Datenspeicher relativ viele Probleme auf. Auf die wichtigsten Schwierigkeiten sowie die erforderlichen Anpassungen wird im Folgenden eingegangen.

Unterschiedliche Behandlung kurzer und langer Literale

Durch die zugrunde liegende Struktur der Datenbank ist es erforderlich, kurze und lange (mehr als 4000 Byte) Literale unterschiedlich zu behandeln. Erstere werden in Spalten vom Typ `Varchar2` gespeichert, wohingegen letztere als `CLOB` (Textfeld nahezu beliebiger Größe) abgelegt werden.

Teilweise ist bemerkbar, dass sich deren Behandlung intern unterscheidet. Der Datentyp des folgenden (kurzen) Literals wird korrekt als String erkannt:

```
1 "Die Antwort ist \"42\"^^<http://www.w3.org/2001/XMLSchema#integer>."^^
      ↪ <http://www.w3.org/2001/XMLSchema#string>
```

Die Zeichenkette enthält unter anderem ein maskiertes Literal. Derartige Konstrukte treten beispielsweise auf, wenn SPARQL-Abfragen als String im RDF-Datenspeicher abgelegt werden. Sobald ein solches Literal die Länge von 4000 Byte überschreitet, wird ein anderer Parser verwendet, der versucht alle Zeichen nach dem ersten Vorkommen des Substrings „^^“ als Datentyp zu interpretieren.

Um derartigen Fehlinterpretationen vorzubeugen, maskiert der Oracle-Connector zusätzlich das „^“-Zeichen. Beim Auslesen werden die Literale entsprechend demaskiert.

Listing 5.2: Schreibweisen für Literal-Strings in SPARQL-Abfragen.

```
1 "Semantic Web"
2 'Semantic Web'
3 """Semantic Web"""
4 '''Semantic Web'''
```

Lockere Behandlung von Literal-Typen

Beim Hinzufügen von Literalen führt Oracle eine Normalisierung dieser über ihre Datentyp-Grenzen hinweg durch, um festzustellen, ob es sich um ein Duplikat eines vorhandenen Wertes handelt [Mur14]. Diese Normalisierung führt beispielsweise dazu, dass die Literale `"42"^^<http://www.w3.org/2001/XMLSchema#int>` und `"42"^^<http://www.w3.org/2001/XMLSchema#unsignedByte>` als gleich angesehen werden.

In der Praxis können Erwartungen verletzt werden, wenn beispielsweise ein als `<http://www.w3.org/2001/XMLSchema#integer>` hinzugefügtes Literal beim Auslesen als `<http://www.w3.org/2001/XMLSchema#decimal>` deklariert wird.

Eingeschränkte Unterstützung von Literalrepräsentationen in SPARQL-Abfragen

Sowohl der Standard für SPARQL 1.1 als auch dessen Vorgänger sehen vier Arten vor, mit denen Literal-Strings in einer SPARQL-Abfrage repräsentiert werden können [HS13; PS08]. Die Strings können von einfachen oder doppelten Anführungszeichen umschlossen werden. Neben der Kurzform existiert jeweils eine Langform, bei der der String von drei Anführungszeichen eingeschlossen wird. Listing 5.2 zeigt den gleichen String in allen vier zur Verfügung stehenden Schreibweisen. Im Gegensatz zur Kurzform sind in der Langform zusätzliche Zeichen (z. B. Zeilenumbrüche) erlaubt, die in der Kurzform maskiert werden müssten.

Von diesen vier Schreibweisen unterstützt Oracle nur die Kurzschreibweise mit doppelten Anführungszeichen. Um trotzdem beliebige SPARQL-Abfragen zu unterstützen, analysiert der Connector eingehende Abfragen. Die enthaltenen Literale werden identifiziert und in die unterstützte Form transformiert.

Keine Berücksichtigung der Groß-/Kleinschreibung bei Variablen

Die Oracle Datenbank verwaltet ihre Tabellen und Spalten unabhängig von der Groß-/Kleinschreibung. In zurückgelieferten Ergebnissen für SQL-Abfragen werden die selektierten Spalten daher immer komplett groß geschrie-

Listing 5.3: In SQL eingebettete SPARQL-Abfrage, die eine Variable mit unterschiedlicher Groß-/Kleinschreibung verwendet.

```
SELECT subjectUri
FROM TABLE(
  SEM_MATCH(
      'SELECT ?subjectUri
       WHERE {
           ?subjectUri ?predicate ?object .
       }',
      SEM_MODELS('erfurt'),
      NULL,
      NULL,
      NULL,
      NULL,
      'STRICT_DEFAULT=T',
      NULL,
      NULL
  )
);
```

ben. Da SPARQL-Abfragen zur Übertragung in SQL eingebettet und zur Ausführung in SQL transformiert werden, unterliegen diese den gleichen Einschränkungen. Das bedeutet, dass in SPARQL definierte Variablen, unabhängig von ihrer Schreibweise, im Ergebnis komplett groß geschrieben erscheinen. Zur Verdeutlichung zeigt Listing 5.3 eine in SQL eingebettete SPARQL-Abfrage und Tabelle 5.1 ein mögliches Ergebnis dieser.

Tabelle 5.1: Mögliches Ergebnis der Abfrage aus Listing 5.3.

SUBJECTURI
http://example.org/person
http://example.org/animal

Da der SPARQL-Standard Variablen mit Groß- und Kleinbuchstaben zulässt, ist die Beibehaltung der ursprünglichen Schreibweise gewünscht [HS13]. Um das Standardverhalten der Datenbank zu umgehen, wird beim Einbetten der SPARQL-Abfragen in SQL zunächst ermittelt, welche Variablen von der SPARQL-Abfrage angefordert werden. Für jede dieser selektierten Variablen wird anschließend ein Alias definiert, das mit unveränderter Schreibweise in das Ergebnis übernommen wird. Listing 5.4 zeigt eine gekürzte Version des Beispiels aus Listing 5.3. Diese verwendet eine Alias-Definition für die selektierte Variable. Das Alias muss in Anführungszeichen gesetzt werden, da es sonst in Großbuchstaben konvertiert wird.

Listing 5.4: Abfrage aus Listing 5.3 mit Korrektur der Schreibweise der Variable via Alias.

```
1 SELECT SUBJECTURI "subjectUri"
2 FROM TABLE(
3   SEM_MATCH(
4       [...]
5   )
6 );
```

Listing 5.5: SPARQL-Abfrage mit SQL-Schlüsselwort als Variablenbezeichnung.

```
1 SELECT ?group
2 WHERE {
3      ?person <http://xmlns.com/foaf/0.1/member> ?group
4 }
```

Konflikte mit SQL-Schlüsselwörtern in SPARQL-Variablen

Während der Entwicklung fiel auf, dass syntaktisch korrekte SPARQL-Abfragen zu einem Fehler führten, wenn diese von Oracle reservierte Schlüsselwörter (z. B. *select* oder *group*) als Variablen verwendeten. Listing 5.5 zeigt eine Abfrage mit dem Schlüsselwort *group* als Variable, die in dieser Form nicht von der Oracle Datenbank ausgeführt wird.

Da der SPARQL-Standard keine Schlüsselwörter vorsieht [HS13], musste eine Transformation vorgenommen werden, um die Kompatibilität zu diesem zu gewährleisten. Es wurde eine Technik ähnlich der zur Behandlung von Variablen mit Groß-/Kleinschreibung gewählt: Im ersten Schritt werden zunächst alle SPARQL-Variablen mit einem unbedenklichen Präfix versehen, um Kollisionen mit Schlüsselwörtern auszuschließen. Beim Einbetten in SQL wird dann für jede Variable ein Alias ohne dieses Präfix definiert, um den ursprünglichen Namen wiederherzustellen. Listing 5.6 zeigt die nach diesem Schema umgeschriebene und in SQL eingebettete SPARQL-Abfrage aus Listing 5.5.

Generierter SQL-Code nicht einsehbar

An Oracle übergebene SPARQL-Abfragen werden zur Ausführung in SQL-Code übersetzt. Der erzeugte Code ist jedoch nicht direkt einsehbar. Bei der Suche nach Fehlern, deren Beschreibungen Teile des generierten SQL-Codes referenzieren, oder bei der Optimierung einzelner Abfragen ist dies ein Nachteil, da sich weder Aufbau der Abfrage und betroffene Tabellen, noch der Ausführungsplan ohne Weiteres ermitteln lassen.

Listing 5.6: In SQL eingebettete SPARQL-Abfrage aus Listing 5.5. Umgeschrieben, um Konflikte mit Schlüsselwörtern auszuschließen.

```
1 SELECT VAR_GROUP "group"
2 FROM TABLE(
3   SEM_MATCH(
4       'SELECT ?var_group
5        WHERE {
6          ?var_person <http://xmlns.com/foaf/0.1/member> ?var_group
7        }',
8       SEM_MODELS('erfurt'),
9       NULL,
10      NULL,
11      NULL,
12      NULL,
13      'STRICT_DEFAULT=T',
14      NULL,
15      NULL
16  )
17 );
```

Abhilfe schafft hier das Aktivieren des sogenannten Tracings, bei dem alle von der Datenbank ausgeführten Abfragen geloggt werden. Anschließend kann aus dieser Menge anhand des Ausführungszeitpunktes oder spezifischer Merkmale der SPARQL-Abfrage die zugehörige SQL-Abfrage ermittelt werden[9].

Globaler Namensraum für Models

Beim Anlegen eines semantischen Models in der Datenbank muss diesem ein Name zugewiesen werden. Eine Liste existierender Models kann aus der View `MDSYS.SEM_MODEL$` ausgelesen werden. Die Ausgabe dieser View ist abhängig von den Rechten des ausführenden Benutzers. Ein Administrator kann alle existierenden, ein normaler Benutzer nur seine eigenen Models abrufen. Im Gegensatz zu dieser lokalen Ansicht ist der Namensraum der angelegten Models global. Das bedeutet, dass verschiedene Benutzer kein Model mit dem gleichen Namen anlegen können. Es kann zu Namenskonflikten mit Models kommen, die für den aktuellen Benutzer nicht sichtbar sind.

Verschiedene Betriebsumgebungen wie Entwicklung, Test und Produktion sollten voneinander isolierte Models verwenden. Wenn alle Umgebungen vom gleichen Datenbank-Server bedient werden, muss der Connector den Fall unvorhersehbarer Namenskollisionen behandeln. Um diese zu vermei-

[9] Details zum Aktivieren des Tracings und zum Auswerten der Log-Dateien finden sich unter http://www.orafaq.com/wiki/SQL_Trace und http://www.orafaq.com/wiki/TKProf.

den, werden die erzeugten Models mit dem Namen des Datenbank-Benutzers als Präfix versehen. Unterschiedliche Umgebungen müssen daher mit verschiedenen Benutzern betrieben werden. Da jeder Nutzer in der Oracle Datenbank standardmäßig sein eigenes Schema erhält, hat dieses Vorgehen zudem den Vorteil, dass die Daten vollständig voneinander getrennt sind.

Eingeschränkte Möglichkeiten für Massenimporte

Für das Importieren großer Tripel-Mengen, sogenanntes Bulk Loading, bietet die Oracle Datenbank keine Lösung, die jeden Anwendungsfall optimal abdeckt. Die Dokumentation beschreibt drei verschiedene Wege für Massenimporte, die jeweils eigene Einschränkungen mit sich bringen [Mur14]:

- Import über eine Hilfstabelle
- Import mit Hilfe der Java API
- Import über die Application-Tabelle

Der Import über eine Staging-Hilfstabelle wird als schnellste Lösung beschrieben. Allerdings werden Literale mit einer Größe von mehr als 4000 Byte nicht unterstützt [Mur14], wodurch dieser Ansatz nicht als generelle Lösung verwendbar ist.

Die Java API kann RDF-Daten im N-Triple-Format importieren und auch lange Literale können korrekt verarbeitet werden. Allerdings lassen sich die Tripel keinem spezifischen Graph zuordnen [Mur14], was eine generelle Verwendung ebenfalls unmöglich macht.

Das Importieren von Tripeln über eine Application-Tabelle erfolgt mit Hilfe der von Oracle bereitgestellten und in Abschnitt 5.1.2 vorgestellten benutzerdefinierten Typen. Dieser Ansatz hat nicht die Probleme der zuvor beschriebenen Methoden [Mur14]. Er wird als der langsamste eingestuft [Mur14], kommt allerdings aufgrund der generellen Nutzbarkeit beim Oracle-Connector zum Einsatz. Auf die Performanz dieser Lösung wird im Abschnitt 5.1.5 näher eingegangen.

Grenzen bei der Verwendung von Doctrine DBAL

Trotz der objektorientierten API, die Doctrine DBAL für den Datenbankzugriff bietet, konnte die Bibliothek die an sie gerichteten Erwartungen nicht in Gänze erfüllen. In einigen Fällen, z. B. beim Setzen typisierter Sitzungsparameter, der expliziten Unterscheidung zwischen den Datentypen `BLOB` und `CLOB` oder dem Einbetten von SPARQL-Abfragen ohne Verwendung von Parametern, stieß Doctrine an seine Grenzen und es waren Erweiterungen notwendig.

Durch die einfache Konfigurierbarkeit sowie die normalisierte Fehlerbehandlung stellt Doctrine dennoch ein brauchbares Fundament dar, auf dem der Oracle-Connector aufbaut.

5.1.5 Optimierung der Performanz

Der erste Prototyp des Oracle-Connectors zeigte teils erhebliche Mängel in der Ausführungsgeschwindigkeit. Dadurch wurde die sinnvolle Verwendung mit OntoWiki unmöglich. Durch Messung der Performanz ließen sich die Ursachen ausfindig machen und größtenteils beheben. Die folgenden Abschnitte stellen die Ergebnisse der Analyse einzelner Teilprobleme sowie die ergriffenen Gegenmaßnahmen dar.

Auslesen von Daten

Bei der Leseperformanz erforderten zwei Aspekte eine nähere Analyse. Zum einen wurde ein hoher Anteil der Laufzeit darauf verwendet, die erhaltenen Ergebnisse innerhalb des PHP-Codes nachzubearbeiten. Zum anderen war ein genereller Abfall der Geschwindigkeit bemerkbar, wenn das Ergebnis Literale mit einer Größe von mehr als 4000 Byte enthielt.

Um die Auswirkungen von Code-Änderungen auf die Performanz messen zu können, wurden zwei Benchmarkszenarien erstellt. Im ersten Szenario werden Datenbestände verschiedener Größen ohne große Literale ausgelesen. Das zweite Szenario simuliert den entgegengesetzten Extremfall und lädt Datenbestände unterschiedlicher Größen, die ausschließlich Literale mit einer Größe von mehr als 4000 Byte enthalten.

Die Abbildungen 5.4 und 5.5 zeigen die Performanz der folgenden drei Implementierungen im ersten bzw. zweiten Szenario:

1. **Nachbearbeitung in PHP**
   Der initiale Code-Stand, in dem die Nachbearbeitung des Ergebnisses in PHP durchgeführt wird. Die Nachbearbeitung umfasst z. B. das Entfernen der Präfixe von Variablen-Namen.
2. **Nachbearbeitung in der Datenbank**
   In dieser Version wurde die Nachbearbeitung größtenteils aus PHP entfernt. Diese wurde durch Umschreiben der auslesenden SQL-Abfrage ersetzt.
3. **Verzögertes Laden von CLOBs**
   Dieser Stand basiert auf dem vorherigen und führt die Nachbearbeitung ebenfalls in der Datenbank aus. Zusätzlich werden Literale mit einer Größe von mehr als 4000 Byte, die intern als `CLOB` gespeichert werden, separat geladen, sobald das Ergebnis einer SPARQL-Abfrage vorliegt.

Aus den Darstellungen wird ersichtlich, dass die Anzahl der pro Sekunde ausführbaren Abfragen sinkt, je größer das ausgelesene Ergebnis ist. Dieser Sachverhalt ist unabhängig von der Größe der Literale oder der konkreten Implementierung.

Durch das Verlagern der Nachbearbeitung des Ergebnisses von PHP in die Datenbank (Version 2) konnte die Verarbeitungsrate der Abfragen in

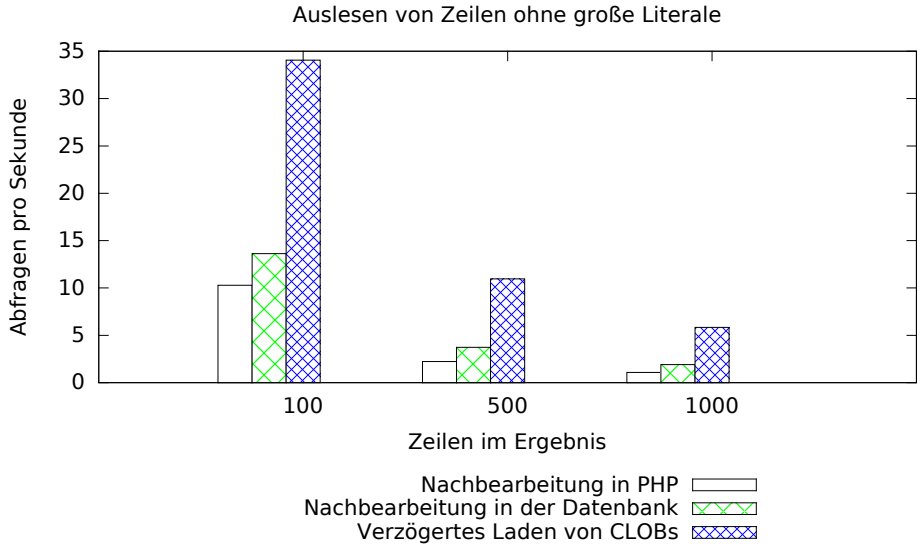


Abbildung 5.4: Auslesen von Daten mit Literalen kleiner als 4000 Byte.

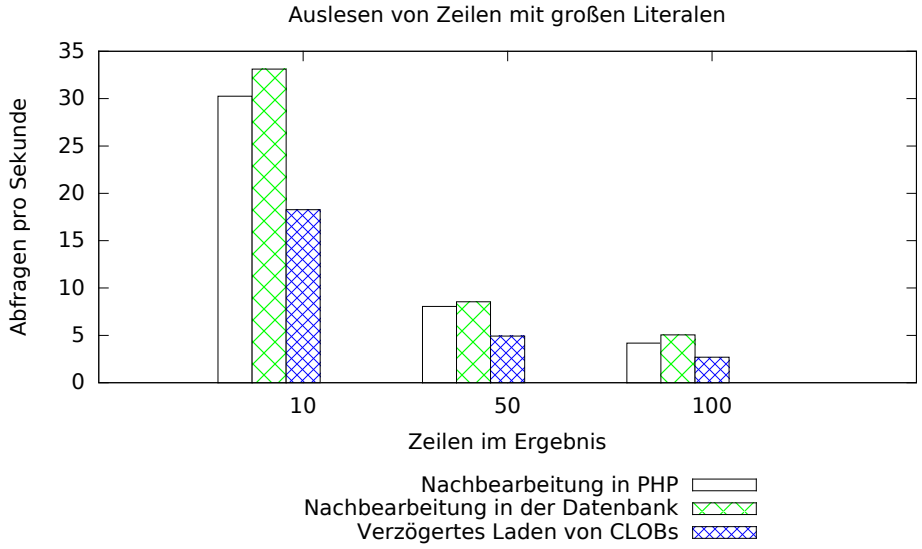


Abbildung 5.5: Auslesen von Daten mit Literalen größer als 4000 Byte.

allen Fällen erhöht werden, teilweise um mehr als 30%.

Recherchen zeigten, dass die im Ergebnis vorgesehenen `CLOB`-Spalten für große Literale dazu führten, dass die Ergebnisse zeilenweise aus der Daten-

bank geladen wurden[10]. Dies ist auch dann der Fall, wenn diese Spalten leer sind. Die auf Version 2 aufbauende Version 3 wird dieser Erkenntnis gerecht, indem sie die `CLOB`-Spalten aus dem Ergebnis entfernt und Literale mit einer Größe von mehr als 4000 Byte im Nachhinein mit separaten Abfragen anfordert. Dadurch konnte die Performanz beim Auslesen von Daten ohne große Literale nahezu um den Faktor 3 gesteigert werden. Die Geschwindigkeit beim Laden von Ergebnissen, die ausschließlich derartige Literale enthalten, halbierte sich hingegen.

Da davon auszugehen ist, dass die Anzahl großer Literale in der Überzahl der mit OntoWiki verwalteten Datenbestände eher gering ist, kommt Lösung 3 beim Oracle-Connector zum Einsatz. Der generelle Geschwindigkeitsgewinn wiegt hier schwerer als die entstehenden Nachteile beim Laden großer Daten.

Speichern von Daten

Die Performanz beim Einfügen von Datensätzen macht sich insbesondere beim Import größerer Datenmengen bemerkbar. Zum Verbessern der Geschwindigkeit wurden die folgenden Strategien erprobt:

1. **Einzelne Einfügeoperationen**
   Die einfachste Strategie: Für jedes zu speichernde Tripel wird eine INSERT-Anfrage an die Datenbank gesendet.
2. **Zusammengefasste INSERTs**
   Sofern möglich werden Einfügeoperationen gesammelt und wie in Listing 5.7 dargestellt in einen `BEGIN [...] END`-Block eingebettet und als einzelne Abfrage an die Datenbank gesendet, um die Anzahl der Roundtrips zu reduzieren.
3. **INSERT/SELECT-Kombination**
   Diese Strategie ähnelt der vorherigen, macht sich allerdings zunutze, dass eine INSERT-Anweisung mit einem SELECT kombiniert werden kann. Mehrere Speicheranfragen werden, wie in Listing 5.8 gezeigt, zu einem einzigen Befehl kombiniert.
4. **Stored Procedure**
   Hier wird eine Stored Procedure verwendet, welche die INSERTs ausführt. Dadurch müssen nur die einzufügenden Daten übertragen werden.
5. **Stored Procedure, Bulk Loading**
   Eine Abwandlung der vorherigen Prozedur, welche die Bulk Loading-Fähigkeiten der Oracle-Datenbank nutzt[11]. Bulk Loading reduziert die Anzahl der Kontextwechsel zwischen PL/SQL und SQL und verspricht eine signifikante Steigerung der Verarbeitungsgeschwindigkeit.

[10] http://php.net/manual/en/function.oci-set-prefetch.php
[11] http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/tuning.htm#LNPLS01205

Listing 5.7: Einfügeoperation für zwei Tripel, zusammengefasst zu einer SQL-Abfrage.

```
1 BEGIN
2   INSERT INTO erfurt_semantic_data (triple)
3     VALUES (SDO_RDF_TRIPLE_S(:modelAndGraph_1, :subject_1,
      ↪ :predicate_1, :object_1));
4   INSERT INTO erfurt_semantic_data (triple)
5     VALUES (SDO_RDF_TRIPLE_S(:modelAndGraph_2, :subject_2,
      ↪ :predicate_2, :object_2));
6 END;
```

Listing 5.8: Einfügeoperation für zwei Tripel, zu einem INSERT-Befehl zusammengefasst.

```
1 INSERT INTO erfurt_semantic_data (triple)
2   SELECT SDO_RDF_TRIPLE_S(:modelAndGraph_1, :subject_1, :predicate_1,
      ↪ :object_1) FROM DUAL
3   UNION ALL
4   SELECT SDO_RDF_TRIPLE_S(:modelAndGraph_2, :subject_2, :predicate_2,
      ↪ :object_2) FROM DUAL
```

Listing 5.9: Stored Procedure zum Einfügen von Tripeln unter Verwendung des Bulk Loading Features.

```
1 PROCEDURE ADD_TRIPLES(graphs IN uri_list, subjects IN uri_list,
      ↪ predicates IN uri_list, objects IN object_list) AS
2 BEGIN
3   FORALL i IN 1 .. graphs.count
4     INSERT INTO erfurt_semantic_data (triple)
5       VALUES (SDO_RDF_TRIPLE_S(
6         graphs(i),
7         subjects(i),
8         predicates(i),
9         objects(i)
10      ));
11 END;
```

Die Prozedur ist in Listing 5.9 zu sehen.

Zur Leistungsmessung der Strategien wurden die folgenden drei Szenarien modelliert:

i) **Einzelnes Einfügen von 1000 Tripeln**
Der Connector erhält 1000 Anfragen zum Einfügen je eines Tripels. Jedes Tripel muss einzeln gespeichert werden, d. h. der Connector hat

nicht die Möglichkeit diese Anfragen zusammenzufassen. Zu ähnlichen Situationen kann es kommen, wenn der Connector von mehreren, nicht zusammenhängenden Komponenten verwendet wird: Diese haben nicht das nötige (globale) Wissen, um die Grenzen für gruppierte Einfügeoperationen korrekt zu definieren.

ii) **Einfügen von 1000 Tripeln in 100er-Gruppen**
Je 100 Tripel werden im Batch-Processing-Modus eingefügt, d. h. der Connector hat die Möglichkeit diese Einfügeoperationen zusammenzufassen. In der Praxis entspricht dieses Szenario einer kleinen Datenänderung wie z. B. dem Umbenennen einer Ressource.

iii) **Gruppiertes Einfügen von 1000 Tripeln**
Der Connector erhält im Batch-Processing-Modus eine Menge von 1000 zu speichernden Tripeln. Dieses Szenario entspricht beispielsweise dem Import von Tripel-Daten aus einer Datei.

Um die Datenkonsistenz bei der Durchführung logisch zusammengehöriger Operationen zu gewährleisten, führt der Oracle-Connector Änderungen im Batch-Processing-Modus in einer Transaktion aus. Es ist der Einfügestrategie überlassen, wie viele der vorliegenden Tripel diese beim Speichern zusammenfasst. Die vorgestellten, auf INSERT-Abfragen basierenden Strategien 2 und 3 speichern je 50 Tripel auf einmal, wohingegen die auf Stored Procedures basierenden Strategien 4 und 5 je 100 Tripel zusammenfassen. Diese Größen wurden für die jeweilige Strategie optimiert. Eine Anpassung im Bereich von 50 bis 1000 verändert deren Performanz nur noch unwesentlich.

Abbildung 5.6 zeigt die Leistung der Strategien in den drei verschiedenen Szenarien. Aus der Grafik wird ersichtlich, dass die Leistung des naiven INSERT-Ansatzes (1) im Batch-Processing-Modus (Szenarien ii) und ii)) einbricht. Dies ist auf die Transaktion zurückzuführen.

Das Zusammenfassen der Inserts (2) steigert zwar die Geschwindigkeit beim Einfügen vieler Tripel deutlich, bietet aber dennoch weniger Performanz als die INSERT/SELECT-Kombination (3). Die Idee mit Hilfe einer Stored Procedure (4) Arbeit in die Datenbank zu verlagern funktioniert erst dann annehmbar, wenn die Procedure durch die Bulk Loading-Funktionalität (5) ergänzt wird. In dem Fall liefert der Ansatz eine leicht bessere Performanz als die INSERT/SELECT-Kombination.

Die Stored Procedure-Lösungen (4, 5) haben den Nachteil, dass sie, wegen der Notwendigkeit sich in der Deklaration für einen Datentyp zu entscheiden, nicht verwendet werden können, um lange Literale (vom Typ `CLOB`) zu speichern. Für diesen Sonderfall wird weiterhin die INSERT/SELECT-Kombination (3) verwendet.

Die beiden performantesten Lösungen (3 und 5) stehen dem Oracle-Connector als konkrete BatchProcessor-Implementierungen zur Verfügung. Standardmäßig wird die INSERT/SELECT-Kombination verwendet, da diese der Stored Procedure mit Bulk Loading in Bezug auf die Geschwindigkeit

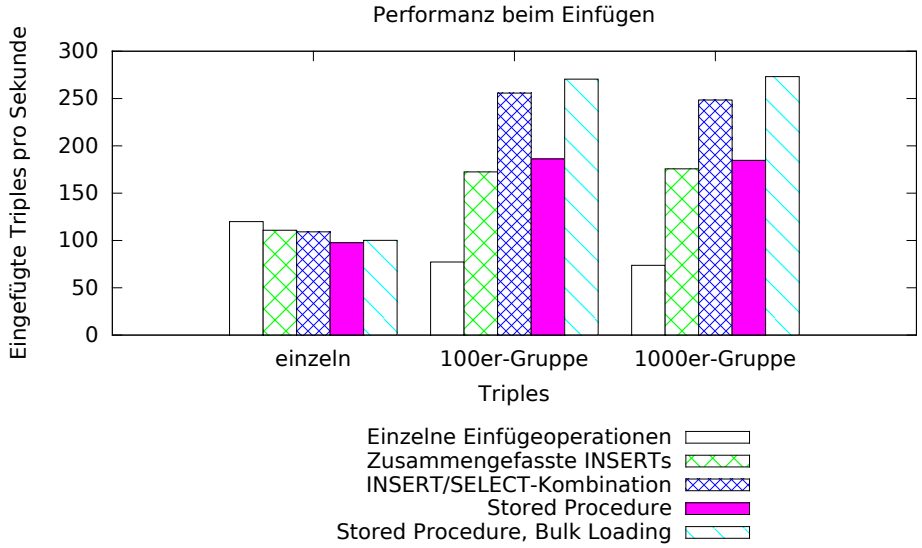


Abbildung 5.6: Leistungsvergleich der vier Speicherstrategien in drei verschiedenen Szenarios.

kaum nachsteht und mit weniger Berechtigungen auskommt, da keine Procedures angelegt werden müssen.

Komplexe Abfragen mit Filtern

Ein spezifisches Performanzproblem machte sich bei der in Listing 5.10 dargestellten SPARQL-Abfrage bemerkbar. Die Abfrage wird von OntoWiki zum Aufbau der Navigation verwendet und enthält zahlreiche Group Graph Patterns sowie FILTER-Ausdrücke. Die Laufzeit dieser Abfrage betrug ohne weitere Optimierung bis zu 80 Sekunden.

Zunächst wurde geprüft, ob es möglich ist die Abfrage in äquivalente Abfragen mit besserer Performanz umzuformulieren. Als mögliche Lösung erwies sich das Zusammenführen der regulären Ausdrücke in den Filtern. Die regulieren Ausdrücke prüfen, ob `?resourceUri` mit einem Substring aus einer fest definierten Liste beginnt:

```
13  FILTER (!isBLANK(?resourceUri))
14  FILTER (!REGEX(STR(?resourceUri),
      ↪ "(^http://www.w3.org/1999/02/22-rdf-syntax-ns#|
      ↪ ^http://www.w3.org/2000/01/rdf-schema#|
      ↪ ^http://www.w3.org/2002/07/owl#)"))
15  FILTER (!BOUND(?reg))
16  FILTER (REGEX(STR(?super), "^http://www.w3.org/2002/07/owl#") ||
      ↪ !BOUND(?super))
```

Listing 5.10: Komplexe SPARQL-Abfrage mit vielen FILTER-Ausdrücken.

```
1  SELECT DISTINCT ?resourceUri
2  FROM <http://localhost/OntoWiki/aksw>
3  WHERE {
4    {
5      ?sub <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?resourceUri
6    } UNION {
7      ?sub <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?resourceUri
8    } OPTIONAL {
9      ?resourceUri <http://ns.ontowiki.net/SysOnt/hidden> ?reg
10   } OPTIONAL {
11     ?resourceUri <http://www.w3.org/2000/01/rdf-schema#subClassOf>
       ↪ ?super
12   }
13   FILTER (!isBLANK(?resourceUri))
14   FILTER (!REGEX(STR(?resourceUri),
       ↪ "^http://www.w3.org/1999/02/22-rdf-syntax-ns#"))
15   FILTER (!REGEX(STR(?resourceUri),
       ↪ "^http://www.w3.org/2000/01/rdf-schema#"))
16   FILTER (!REGEX(STR(?resourceUri), "^http://www.w3.org/2002/07/owl#"))
17   FILTER (!BOUND(?reg))
18   FILTER (REGEX(STR(?super), "^http://www.w3.org/2002/07/owl#") ||
       ↪ !BOUND(?super))
19 }
20 LIMIT 11
```

Außerdem war es möglich die Filter zu duplizieren und direkt in die betroffenen Group Graph Pattern zu integrieren:

```
4    {
5      ?sub <http://www.w3.org/2000/01/rdf-schema#subClassOf> ?resourceUri
6      FILTER (!isBLANK(?resourceUri))
7      FILTER (!REGEX(STR(?resourceUri),
       ↪ "^http://www.w3.org/1999/02/22-rdf-syntax-ns#"))
8      FILTER (!REGEX(STR(?resourceUri),
       ↪ "^http://www.w3.org/2000/01/rdf-schema#"))
9      FILTER (!REGEX(STR(?resourceUri),
       ↪ "^http://www.w3.org/2002/07/owl#"))
10   } UNION {
11     ?sub <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?resourceUri
12     FILTER (!isBLANK(?resourceUri))
13     FILTER (!REGEX(STR(?resourceUri),
       ↪ "^http://www.w3.org/1999/02/22-rdf-syntax-ns#"))
14     FILTER (!REGEX(STR(?resourceUri),
       ↪ "^http://www.w3.org/2000/01/rdf-schema#"))
15     FILTER (!REGEX(STR(?resourceUri),
       ↪ "^http://www.w3.org/2002/07/owl#"))
16   }
```

Wie aus der ersten Messreihe in Abbildung 5.7 ersichtlich wird, bieten beide

alternative Lösungen eine deutliche höhere Performanz als die ursprüngliche Abfrage.

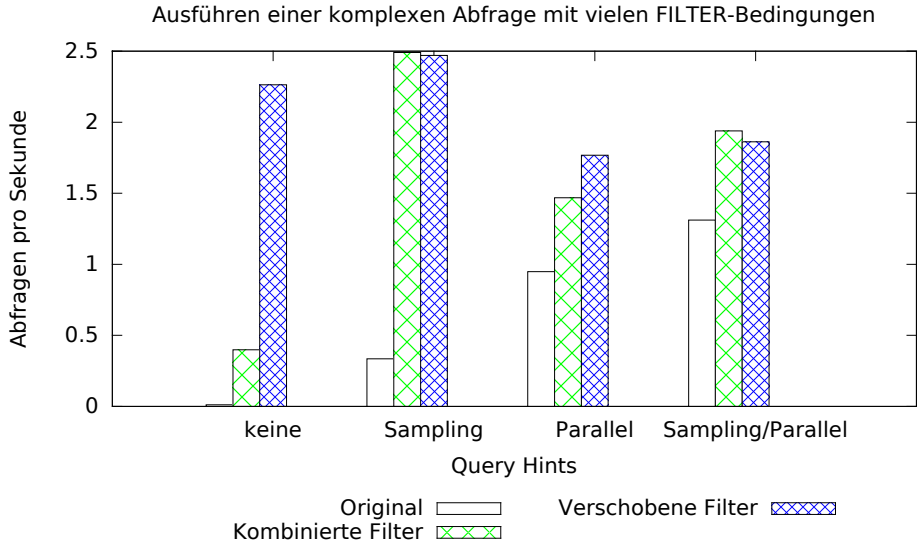


Abbildung 5.7: Performanz der Abfrage aus Listing 5.10 sowie äquivalenter Abfragen unter Verwendung verschiedener Query Hints.

Bei der Entwicklung einer Applikation ist das Umschreiben der Abfragen ein probates Mittel zur Vermeidung von Geschwindigkeitsproblemen, allerdings muss Erfurt als Basisbibliothek mit generischen Abfragen umgehen können. Bei der Optimierung können die Alternativen dennoch nützlich sein. Durch die Einbeziehung in die Messungen wird sichtbar, welche (ggf. auch negativen) Auswirkungen die Optimierungsmaßnahmen auf andere Abfragen haben.

Die automatische Umformung eingehender SPARQL-Abfragen schien ein aussichtsreicher Weg zu sein, stellte sich aber angesichts der wenigen Informationen über das generelle Verhalten der Abfragen und der fehlenden Werkzeuge für derartige Aufgaben als zu zeitintensiv heraus. Mit Query Hints[12] steht allerdings eine Methode zur Verfügung, die es erlaubt die Verarbeitung von Abfragen über Kommentare in den SQL-Befehlen zu beeinflussen.

Als hilfreich erwies sich zum einen das Dynamic Sampling[13], das die Datenbank anweist, zunächst die Beschaffenheit der Daten zu analysieren, um dann einen möglichst effizienten Ausführungsplan zu erzeugen. Zum anderen konnte die Geschwindigkeit mit dem PARALLEL-Hint gesteigert werden,

[12] http://docs.oracle.com/cd/E11882_01/server.112/e41573/hintsref.htm#PFGRF005

[13] https://blogs.oracle.com/optimizer/entry/dynamic_sampling_and_its_impact_on_the_optimizer

der dazu führt, dass rechenintensive Aufgaben bei der Abfrageverarbeitung parallelisiert werden. Abbildung 5.7 zeigt die Performanz der verschiedenen Abfragen unter Verwendung von Dynamic Sampling und Parallelisierung sowie einer Kombination dieser Modi.

Es ist ersichtlich, dass sowohl Dynamic Sampling als auch Parallelisierung die Geschwindigkeit der ursprünglichen Anfrage steigern. Allerdings kann Parallelisierung bei anderen Anfragen zu einer Verschlechterung des Durchsatzes beitragen. Diesen Erkenntnissen folgend verwendet der Oracle-Connector generell Dynamic Sampling. Die Parallelisierung wird nur dann aktiviert, wenn eine SPARQL-Abfrage viele FILTER-Ausdrücke enthält, da in einem solchen Fall eine höhere Chance besteht, dass rechenintensive Operationen von einer Parallelisierung profitieren.

Ungelöste Probleme mit ASK-Abfragen

`ASK`-Abfragen werden verwendet, um zu prüfen, ob für angegebene Bedingungen ein nicht-leeres Ergebnis existiert. Während der Entwicklung kam es bei dieser Art der Abfrage teilweise zu Auffälligkeiten. Beispielsweise war die Ausführungsdauer der in Listing 5.10 dargestellten Abfrage, umformuliert zur `ASK`-Form, bis zu 25 größer als die der ursprünglichen `SELECT`-Version.

Da `ASK`-Abfragen die Existenz eines Abfrage-Treffers bestimmen und nicht das Ergebnis ermitteln müssen, wäre im Vergleich zu einer `SELECT`-Abfrage intuitiv eine schnellere Verarbeitung zu erwarten. Eine generell höhere Laufzeit von `ASK`-Abfragen konnte nicht beobachtet werden.

Zur Lösung derartiger Laufzeitprobleme ist es denkbar, betroffene `ASK`-in entsprechende `SELECT`-Abfragen mit limitierter Ergebnisgröße zu transformieren und die ermittelten Ergebniszeilen zu zählen. Da sich im Voraus allerdings nicht bestimmen lässt, ob eine Abfrage betroffen ist, konnte eine derartige Optimierung nicht in den Oracle-Connector integriert werden.

5.2 Stardog

5.2.1 Einordnung

Stardog[14] ist ein speziell für die Verwaltung von RDF-Daten entwickeltes Datenbank-Management-System, das von Clark & Parsia vertrieben wird. Während der Implementierung wurde die Version 2.1.0 verwendet.

Die native RDF-Unterstützung von Stardog entspricht den Kriterien aus Anforderung 1b und verspricht gute Performanz.

[14] http://stardog.com/

5.2.2 Funktionsweise

Da der Quellcode von Stardog nicht einsehbar ist, ist seine Funktionsweise nicht im Detail bekannt. Dokumentiert ist, dass das in Java implementierte System primär auf einen großen Hauptspeicher setzt und die Fähigkeiten moderner Multi-Core-CPUs nutzt[15]. Zur Verwaltung großer Datenmengen wird eine Scale-Up-Strategie verfolgt, d. h. das Host-System benötigt bei wachsender Datenmenge mehr Ressourcen. Cluster-Systeme mit mehreren Maschinen sind nicht vorgesehen.

5.2.3 Anbindung

Zur Kommunikation mit dem Stardog-Server können die Sesame- bzw. Jena-API in Java, die Kommandozeile sowie Webservices verwendet werden[16]. Da die direkte Verbindung von PHP zu Java unüblich ist und es dementsprechend wenig stabile Bibliotheken für diesen Anwendungsfall gibt, liegt es nahe, die von Stardog angebotenen Webservices zum Herstellen der Verbindung zu benutzen. Lösungen mit Webservices haben sich in der PHP-Welt bewährt. Mit Guzzle[17] existiert eine Client-Bibliothek, die zahlreiche HTTP-Features unterstützt und sowohl die deklarative als auch die programmatische Definition von Webservice-Schnittstellen erlaubt.

Abbildung 5.8 zeigt, wie der Stardog-Connector in Erfurt integriert wurde. Der `StardogSparqlConnector` ist für Erfurt der definierte Zugriffspunkt auf diesen Datenspeicher. Der auf Guzzle aufbauende `ApiClient` ist für die Kommunikation mit dem Stardog-Server verantwortlich. Die öffentliche Schnittstelle des Clients orientiert sich stark an den von Stardog angebotenen Services und erreicht keine hohe Abstraktion. Stattdessen kann das Verhalten im Detail kontrolliert werden und der Client lässt sich sowohl für die Datenabfrage als auch das Datenbank-Management verwenden.

Der `DataAccessClient` stellt das Bindeglied zwischen dem Connector und der Low-Level-API dar. Er ist auf das Daten-Management spezialisiert und bietet eine für diese Aufgabe optimierte Schnittstelle an, die sich bei Bedarf z. B. automatisch um das Transaktions-Management kümmert.

Verschiedene `BatchProcessor`-Implementierungen sind für das Speichern von Tripeln verantwortlich. Es handelt sich teilweise um abstrakte Konstrukte, die beispielsweise je nach Beschaffenheit der Daten an verschiedene Speichermechanismen delegieren. Für Stardog ist das interessant, da verschiedene Importmethoden eine unterschiedliche Anzahl an Webservice-Aufrufen benötigen. Es kann z. B. sinnvoll sein, ein einzelnes Tripel mittels SPARQL Update zu sichern, wodurch nur ein Service-Aufruf entsteht, wohingegen ein Aufruf der Import-Services zwei weitere Aufrufe für das Trans-

[15] http://docs.stardog.com/admin/#sd-Capacity-Planning
[16] http://docs.stardog.com/
[17] http://guzzlephp.org/

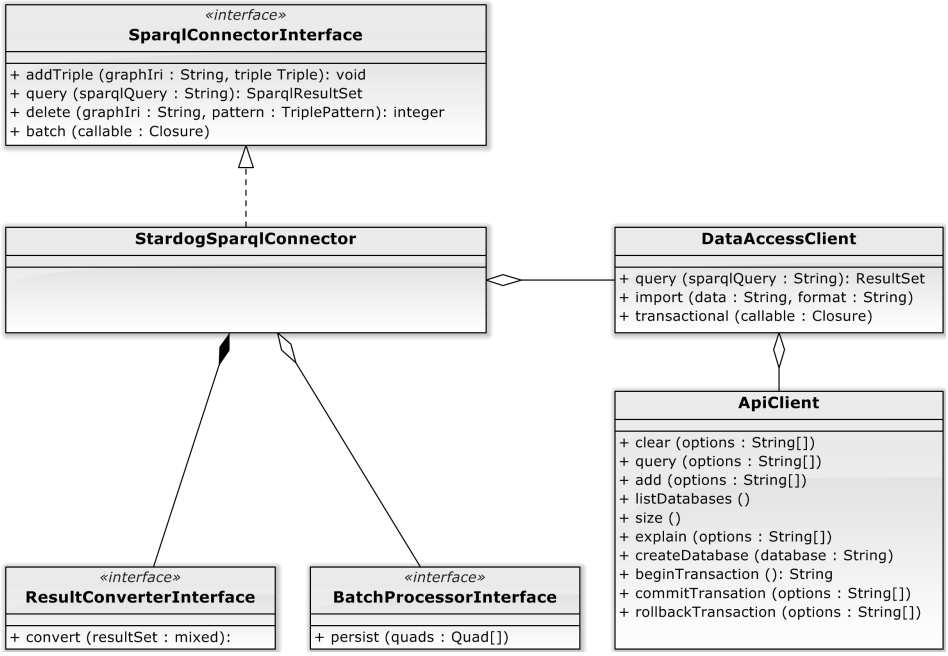


Abbildung 5.8: SPARQL-Connector für Stardog inklusive angrenzender Komponenten.

aktions-Management erfordert.

Mittels `ResultConverter` werden Ergebnisse von SPARQL-Abfragen in das interne Format von Erfurt gebracht. Ausgehend vom SPARQL-JSON-Format [SCG+13] sind dafür nur minimale Anpassungen notwendig.

5.2.4 Probleme bei der Umsetzung

Insgesamt erwies sich Stardog als stabil und wenig fehlerhaft. Dennoch verzögerten einige Probleme die Entwicklung.

Unzureichende Fehlerinformationen bei der Verwendung der Webservices

Bei der Verwendung der HTTP-Schnittstelle des Stardog-Servers stellten sich die im Fehlerfall bereitgestellten Informationen häufig als unzulänglich heraus. Der Server lieferte bei falsch aufgebauten oder nicht richtig parametrisierten HTTP-Anfragen zum Teil unbrauchbare interne Informationen, wie beispielsweise Java-Stacktraces der von Stardog verwendeten Bibliotheken, oder gar keine konkreten Anhaltspunkte bzgl. der Ursache.

Hilfreich war in solchen Fällen die Logging-Funktionalität von Guzzle. Diese erlaubt es, die Rohdaten der HTTP-Kommunikation zwischen PHP

Listing 5.11: SPARQL Update-Abfrage, die von Stardog nicht korrekt ausgeführt wird.

```
1 INSERT DATA {
2   GRAPH <http://www.uni-bonn.de/websites> {
3     <http://www.uni-bonn.de/> rdfs:label "http://www.uni-bonn.de/" .
4   }
5 }
```

und Stardog aufzuzeichnen. Anschließend können diese eingesehen und mit der von der API geforderten Struktur verglichen werden.

Fehlerhafte Bearbeitung von INSERT-Abfragen via SPARQL

Bei der Entwicklung fiel auf, dass Literale in einigen der SPARQL-Abfragen zum Einfügen von Daten durch Stardog falsch interpretiert wurden. Listing 5.11 zeigt eine von diesem Fehler betroffene SPARQL-Abfrage. Die Abfrage ordnet einer Website-Adresse ein Label zu. Obwohl das Label als nicht typisiertes Literal kodiert ist, wird es von Stardog als URI interpretiert. Es stellte sich heraus, dass dieser Sonderfall dann auftritt, wenn das Literal der URI des Subjekts oder Prädikats entspricht.

Obwohl das Einfügen von Tripeln via SPARQL Update aus Performanzsicht für kleine Tripel-Mengen durchaus sinnvoll ist, wird es vom Stardog-Connector als Konsequenz aus diesem Fehler nicht verwendet. Um Verlässlichkeit gegenüber Performanz den Vorzug zu geben, werden stattdessen alternative Import-Formate benutzt.

5.2.5 Optimierung der Performanz

Beim Ausführen von lesenden SPARQL-Abfragen erwies sich Stardog als schnell und zuverlässig. Allerdings kam es bei fehlerhaftem Transaktions-Management zu generellen Performanzproblemen und die verschiedenen Speichermöglichkeiten erforderten eine nähere Analyse.

Abfall der Performanz bei offenen Transaktionen

In einem frühen Stadium der Entwicklung war beim Durchführen von Messungen ein genereller Abfall der Geschwindigkeit bemerkbar, der auf eine große Anzahl offener Transaktionen zurückzuführen war.

Stardog bietet zum Beginnen sowie zum Abschließen und Abbrechen von Transaktionen separate Webservices an. Wird eine Transaktion begonnen, aber bis zum Ende des Programmablaufs weder abgebrochen noch bestätigt, bleibt diese offen zurück. Ab einer gewissen Menge offener Transaktionen ist eine generelle Abnahme der Performanz spürbar. Abbildung 5.9 zeigt die

Durchsatzrate der Abfragen in Abhängigkeit von den offenen Transaktionen.

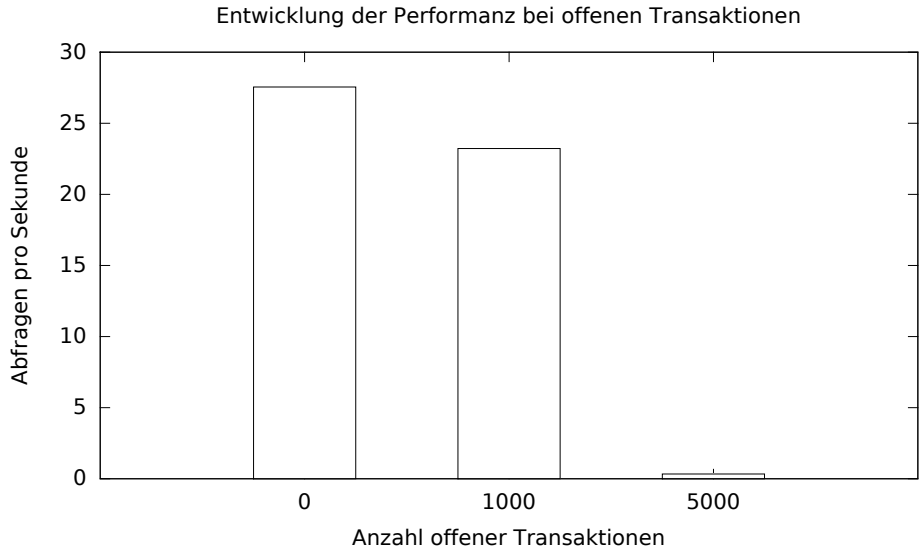


Abbildung 5.9: Abnahme der Performanz in Abhängigkeit von den offenen Transaktionen.

Es zeigt sich, dass die Geschwindigkeit in Abhängigkeit der verbleibenden Transaktionen nicht-linear abnimmt und dass der Performanz-Verlust teilweise massiv ausfällt. Zum Beheben des Problems war ein Neustart des Servers notwendig.

Da dieses Verhalten die Stabilität eines lange laufenden Server-Systems gefährdet, war es notwendig eine Strategie zur Vermeidung der beschriebenen Situation zu entwickeln. Der API-Client führt daher eine Liste aller offenen Transaktionen und nicht explizit behandelte Transaktionen werden bei Programmende automatisch abgebrochen.

Speichern von Daten

Stardog unterstützt für das Einfügen von Daten sowohl zahlreiche Import-Formate, wie beispielsweise RDF/XML, Turtle oder N-Quads[18], als auch die SPARQL Update-Syntax.

Zum Vergleich wurde mit N-Quads ein Vertreter der Import-Formate gewählt. Diesem wurde die SPARQL Update-Funktionalität gegenübergestellt. N-Quads wurde gewählt, weil das Format einfach zu erzeugen ist. Zudem ist es aufgrund der zusätzlichen Angabe eines Kontextes möglich, Tripel in mehrere Graphen einzufügen [Car14]. SPARQL Update ist interessant, weil

[18] http://docs.stardog.com/http/#sd-Stardog-HTTP-Protocol

die Abfragen von Stardog außerhalb einer Transaktion ausgeführt werden können, wodurch die Anzahl der HTTP-Anfragen reduziert wird.

Zum Überprüfen der Geschwindigkeit wurden wie bei dem Oracle-Connector drei Szenarien herangezogen (vgl. 5.1.5):

i) **Einzelnes Einfügen von 1000 Tripeln**
ii) **Einfügen von 1000 Tripeln in 100er-Gruppen**
iii) **Gruppiertes Einfügen von 1000 Tripeln**

Beim gruppierten Einfügen wird immer eine Transaktion verwendet. Um die Anzahl der HTTP-Anfragen zu reduzieren ist der Tripel-Puffer für beide Speicher-Verfahren so groß gewählt, dass er alle anfallenden Tripel fassen kann.

Abbildung 5.10 zeigt SPARQL Update und N-Quads-Import im direkten Vergleich. Beide Verfahren profitieren davon, wenn Einfügeoperationen

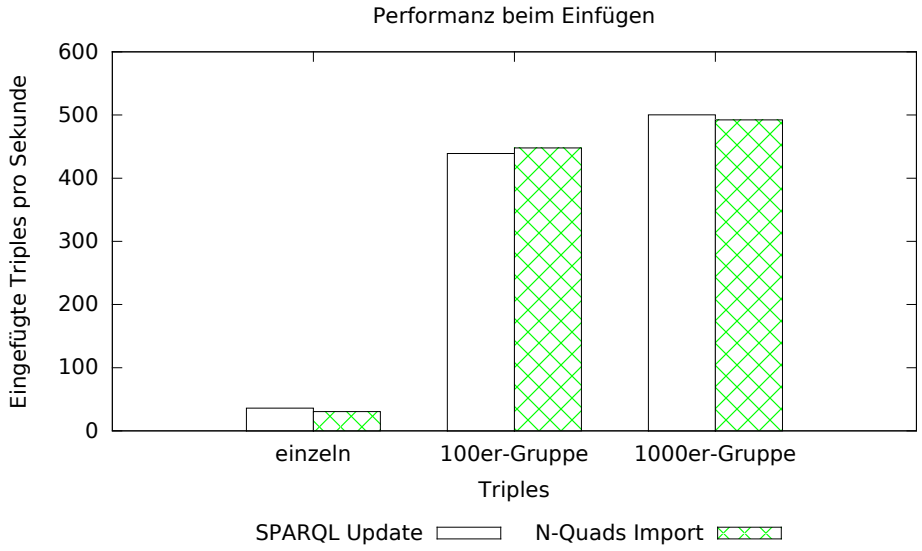


Abbildung 5.10: Leistungsvergleich von zwei Speicherstrategien in drei verschiedenen Szenarios.

zusammengefasst werden können. Beim einzelnen Einfügen ist SPARQL Update etwas schneller, da hier im Vergleich zum Import nur $\frac{1}{3}$ der HTTP-Anfragen ausgeführt werden müssen. Beim Erstellen des N-Quads-Dokuments entsteht zusätzlicher Aufwand, da Stardog dieses nicht komplett in UTF-8 akzeptiert, sondern eine Zeichen-Konvertierung gemäß alten N-Triples-Standards erwartet [BG04; Car14]. Dennoch unterscheidet sich die Leistung beider Methoden nicht wesentlich.

Weil es in Sonderfällen bei der Verwendung von SPARQL Update zu den im Abschnitt 5.2.4 beschriebenen Fehlinterpretationen kommen kann und

der Performanz-Gewinn nicht signifikant ist, kommt beim Stardog Connector der N-Quads-Import zum Einsatz.

5.3 Neo4j

5.3.1 Einordnung

Neo4j[19] ist eine in Java implementierte und weit verbreitete[20] Graph-Datenbank, die sowohl unter freier als auch kommerzieller Lizenz verfügbar ist. Die Datenbank wird als zukunftsfähige Technologie eingestuft [TW14], wodurch sich deren Verbreitung noch vergrößern dürfte. Durch die freie Verfügbarkeit und steigende Popularität erfüllt Neo4j das Profil von Anforderung 1c.

Obwohl Neo4j standardmäßig Cypher[21] und nicht SPARQL als Abfragesprache verwendet, ist eine Verwendung als RDF-Datenspeicher möglich, da sich RDF-Daten problemlos als Graph darstellen lassen. Bei der Implementierung wurde Neo4j in der Version 1.9.6 verwendet.

5.3.2 Funktionsweise

Mit einem Plugin[22] kann auch via SPARQL auf Daten in Neo4j zugegriffen werden. Es bietet weitere Webservice-Schnittstellen an und dient als Mediator zwischen dem Client und der Kernfunktionalität von Neo4j.

Neo4j speichert Daten als Knoten und gerichtete Relationen (Kanten). Da die Datenbank das Property-Graph-Modell unterstützt, lassen sich zudem beliebige Schlüssel-Wert-Paare an Knoten und Relationen anhängen [RWE13]. Abbildung 5.11 zeigt beispielhaft einen solchen Graphen. Die

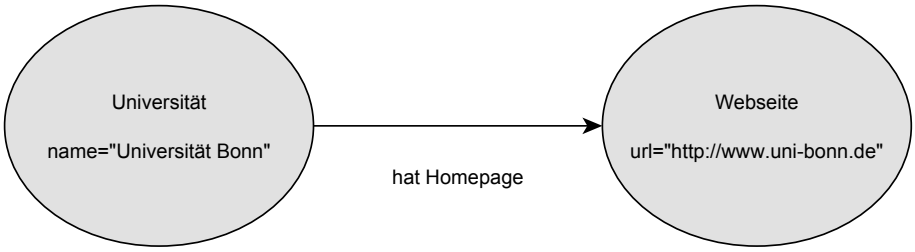


Abbildung 5.11: Beispiel für einen Property-Graph.

im Graphen enthaltenen Informationen lassen sich, unter Verwendung des FOAF-Vokabulars, wie in Listing 5.12 dargestellt als Tripel kodieren [BL14].

[19] http://www.neo4j.org/
[20] Laut http://db-engines.com im Juni 2014.
[21] http://docs.neo4j.org/chunked/stable/cypher-introduction.html
[22] https://github.com/neo4j-contrib/sparql-plugin

Listing 5.12: Informationen aus Graph 5.11 in Form von Tripeln.

```
1 <http://uni-bonn.de>     rdf:type     <http://uni-bonn.de/university> .
2 <http://uni-bonn.de>     foaf:name    "Universität Bonn"@de .
3 <http://www.uni-bonn.de> rdf:type     foaf:Document .
4 <http://uni-bonn.de>     foaf:hompage <http://www.uni-bonn.de> .
```

Um die semantischen Daten verarbeiten zu können, müssen diese in das Property-Graph-Modell überführt werden. Das verwendete SPARQL-Plugin baut auf der Blueprints-Bibliothek[23] auf, die ein solches Mapping[24] vorgibt. Subjekt und Objekt eines Tripels werden zu Knoten und Prädikate zu Kanten. Zusatzinformationen wie Sprache oder Datentyp werden als Schlüssel-Wert-Paare an den zugehörigen Knoten gespeichert. Abbildung 5.12 zeigt einen nach diesen Regeln aufgebauten Property-Graph, der die aus dem Beispielgraph 5.11 extrahierten Tripel enthält. Die Eigenschaft `c` (für „Context“) an den Relationen steht für den RDF-Graph, in dem die jeweilige Relation enthalten ist. Das Beispiel verwendet hier `<http://uni-bonn.de/graph/default>` um zu signalisieren, dass die Relationen zum Standard-Graph gehören. Die weiteren Eigenschaften (`p` und `cp`) haben technische Gründe: Sie werden indexiert und ermöglichen bei der Bearbeitung von SPARQL-Abfragen einen gezielteren Einstieg in den Graph.

5.3.3 Anbindung

Die Anbindung der Neo4j-Datenbank an Erfurt erfolgt mit Hilfe der vom SPARQL-Plugin zur Verfügung gestellten Webservice-Schnittstelle. Zusätzlich werden die ebenfalls via Webservice verfügbaren Kernkomponenten von Neo4j verwendet, um die Datenbank zu verwalten.

Abbildung 5.13 zeigt die Integration des Connectors in Erfurt. Wie üblich ist der `Neo4JSparqlConnector` die Schnittstelle zu den anderen Komponenten der Erfurt-Bibliothek. Der Connector kommuniziert auf zwei Arten mit Neo4j: SPARQL-Abfragen werden über die `SparqlApiClient`-Klasse gesendet, wohingegen die Datenmanipulation mittels `StoreManagementClient` erfolgt, der die native Abfragesprache Cypher verwendet.

Die Ergebnisse einer SPARQL-Abfrage werden als mehrdimensionales Array geliefert und enthalten Literale als RDF-Term, also z. B. `"Universität Bonn"@de`. Die Resultate müssen daher noch mittels `ResultConverter` umgeformt werden, um die vom Interface geforderte Struktur zu erhalten.

[23] https://github.com/tinkerpop/blueprints
[24] https://github.com/tinkerpop/blueprints/wiki/Sail-Implementation

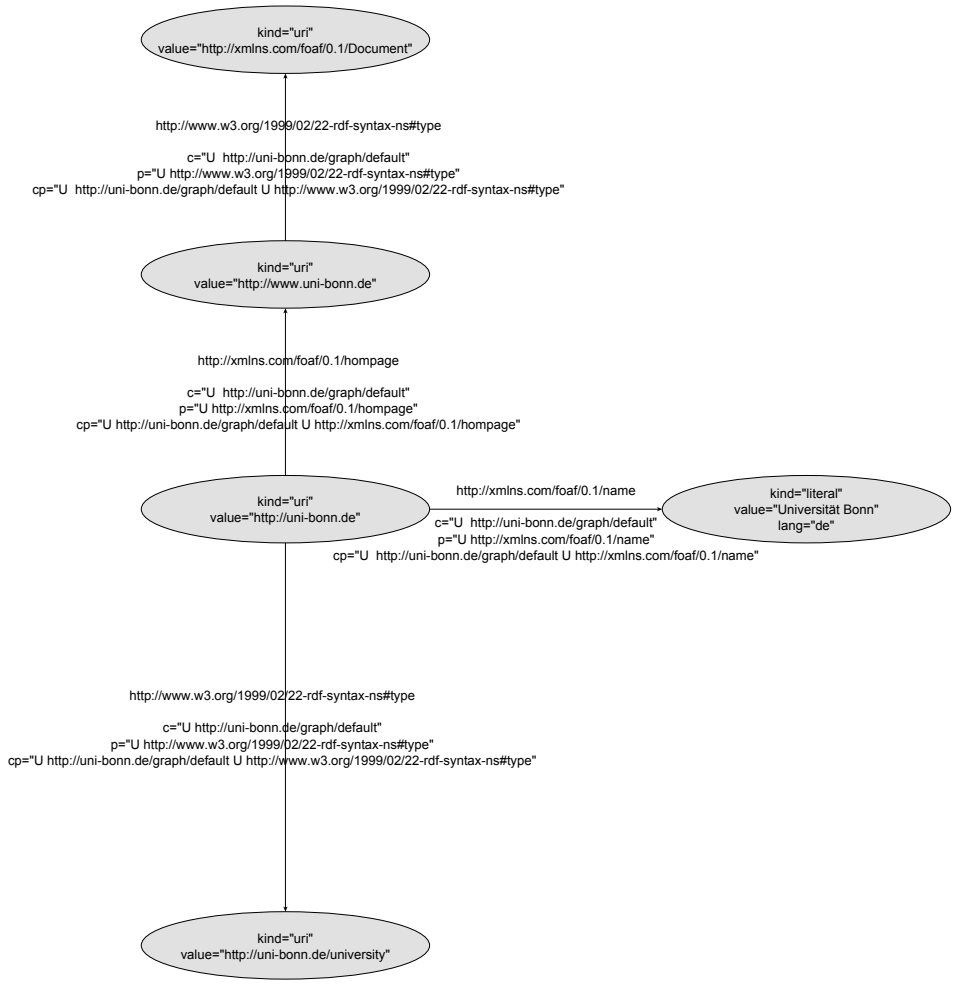


Abbildung 5.12: In einen Property-Graph überführte Tripel aus 5.12.

5.3.4 Probleme bei der Umsetzung

Neo4j erwies sich als zuverlässiger Datenspeicher. Probleme traten lediglich beim Einfügen von Tripeln auf.

Einfügen von Daten

Ursprünglich war das Einfügen von Tripeln über eine vom SPARQL-Plugin angebotene Webservice-Schnittstelle vorgesehen. Diese akzeptiert Subjekt, Prädikat sowie Objekt und fügt das resultierende Tripel in den Neo4j-Graph ein.

Während der Entwicklung offenbarten sich Schwächen in der Implementierung dieser Schnittstelle. Diese versucht zunächst ein Tripel als Kombi-

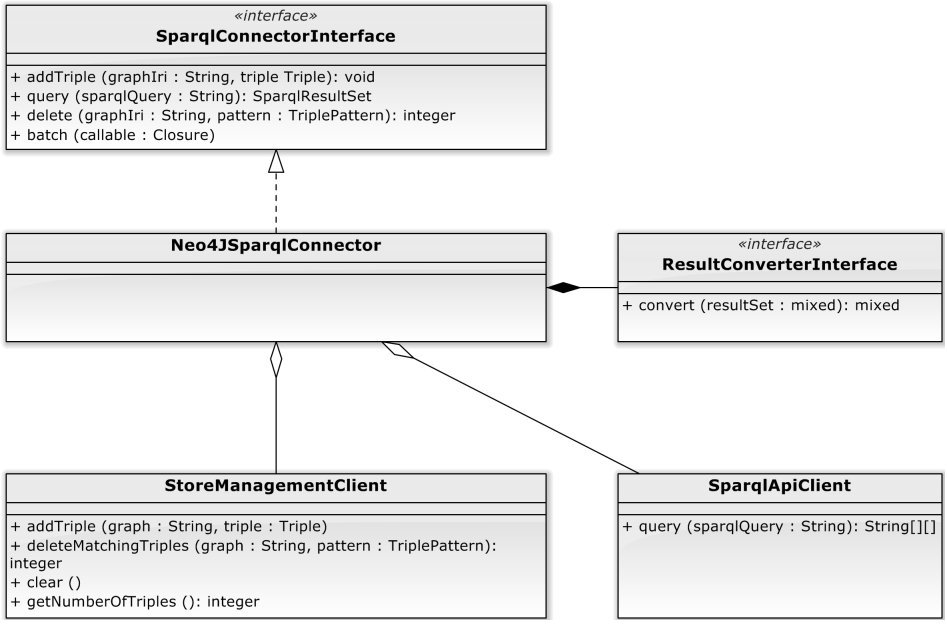


Abbildung 5.13: SPARQL-Connector für Neo4j inklusive angrenzender Komponenten.

nation von drei URIs aufzufassen und in die Datenbank einzufügen. Wenn dieser Schritt fehlschlägt, versucht das Plugin die Interpretation als zwei URIs mit einem Objekt-Literal.

In der Praxis führt dieses Vorgehen zu zahlreichen Fehlerkennungen. URI-ähnlichen Literalen wird beispielsweise der falsche Typ zugeordnet. Knoten ohne eigene IRI, sogenannte Blank Nodes, werden nicht unterstützt, obwohl diese sich in das Property-Graph-Modell überführen lassen (vgl. Abschnitt 5.3.2).

Um diese Unzulänglichkeiten zu überwinden, wird das Mapping einzufügender Tripel auf das Graph-Modell manuell in der Neo4j-Connector-Komponente vorgenommen. Die notwendige Synchronisierung mit der Datenbank erfolgt mittels Cypher.

Kapitel 6

Vergleich der unterstützten Datenspeicher

Bislang wurden die Connector-Implementierungen isoliert betrachtet. Sowohl die bereits vorhandenen als auch die im Zuge dieser Arbeit hinzugefügten Persistenzlösungen sollen nun zueinander in Bezug gesetzt werden. Dazu wird in Kapitel 6.1 ein kurzer Vergleich ausgewählter Eigenschaften der angebundenen Systeme durchgeführt. In Kapitel 6.2 wird die Performanz der Lösungen mit Hilfe eines etablierten Benchmarks ermittelt und gegenübergestellt.

6.1 Vergleich ausgewählter Eigenschaften

Die eingesetzten RDF-Datenspeicher bieten teils unterschiedliche Features. Das Fehlen bestimmter Features kann die Einsatzmöglichkeiten eines RDF-Datenspeichers einschränken oder den Aufwand für die Anbindung an Applikationen erhöhen, da die fehlenden Eigenschaften simuliert werden müssen.

Tabelle 6.1 gibt einen kurzen Überblick über wichtige Eigenschaften der von OntoWiki unterstützten Systeme.

Tabelle 6.1: Eigenschaften der verschiedenen RDF-Datenspeicher.

Eigenschaft	*Virtuoso*	*MySQL*	*Oracle*	*Stardog*	*Neo4j*
Open Source	✓	✓	✗	✗	✓
Native RDF-Unterstützung	✓	✗	✗	✓	✗
Installationsaufwand	gering	gering	hoch	gering	mittel
SPARQL-Update	✓	✗	✗	✓	✗
SPARQL ASK	✓	✓	✓	✓	✗
Reasoning	✓	✗	✓	✓	✗

6.1.1 Open Source

Der Einsatz von Open Source Software ist in der Regel kostenfrei möglich. Virtuoso, MySQL und Neo4j bieten sowohl freie als auch kommerzielle Lizenzen, wobei letztere erweiterte Funktionalität beinhalten. Um die notwendige Funktionalität für das OntoWiki bereitzustellen sind die freien Versionen ausreichend.

Oracle und Stardog sind kommerzielle Systeme, lassen sich für die Entwicklung aber kostenfrei nutzen. Die freie Version von Stardog ist auf eine Datenbankgröße von 25 MB beschränkt, wodurch die Einsatzmöglichkeiten eingeschränkt werden.

6.1.2 Native RDF-Unterstützung

Virtuoso und Stardog bieten native Unterstützung von RDF-Daten.

Die Oracle-Datenbank ist relational. Mit der Spatial und Graph-Option bietet Oracle eine Erweiterung an, welche die Fähigkeit zur Verarbeitung von RDF-Daten nachrüstet. Die RDF-Daten werden dafür auf das bestehende relationale Modell abgebildet.

Sowohl MySQL als auch Neo4j bieten keine native RDF-Unterstützung. Für MySQL wird diese Fähigkeit komplett im Erfurt-Framework simuliert. Neo4j lässt die Verwendung eines Plugins zu, das RDF-Daten auf den internen Property-Graph überträgt und die Abfrage per SPARQL ermöglicht.

6.1.3 Installationsaufwand

Anforderung 2g fordert eine möglichst transparente Installation von OntoWiki, um den Konfigurationsaufwand zu minimieren. Dementsprechend ist auch der für die Installation des eigentlichen Datenbanksystems erforderliche Aufwand relevant.

Für die meisten Systeme ist der Aufwand für eine Standardinstallation gering. Für Virtuoso und MySQL stehen Installationspakete für alle gängigen Plattformen bereit.

Stardog und Neo4j erfordern eine funktionierende Java-Version und können nach dem Herunterladen direkt gestartet werden. Problematisch gestaltet sich für Neo4j das Ermitteln (und ggf. Kompilieren) einer passenden Version des SPARQL-Plugins. Zudem funktioniert das Aktivieren des Plugins nicht mit Neo4j-Versionen, die mit dem Installationsassistenten installiert wurden.

Weit höher ist der Aufwand für die Einrichtung der Oracle-Datenbank: Zum einen hat diese spezielle Anforderungen an das verwendete System und die zur Verfügung gestellten Ressourcen, zum anderen erfordert die Installation die Einarbeitung in zahlreiche weitere Komponenten, die zur Inbetriebnahme der RDF-Funktionalität erforderlich sind. Nach der Grundinstallation müssen zudem noch die RDF-Prozeduren manuell aktiviert und

die Strukturen für die Speicherung der semantischen Daten initialisiert werden.

6.1.4 SPARQL Update

Während der SPARQL 1.0-Standard [PS08] nur den lesenden Zugriff auf Daten beschreibt, definiert SPARQL 1.1 [GPP13] auch eine Syntax zum Einfügen und Aktualisieren von Daten. Diese wird allerdings noch nicht flächendeckend unterstützt.

Für OntoWiki ist fehlende Unterstützung für die Update-Syntax kein Problem: In den wenigen Fällen, wo diese erforderlich ist, wird sie simuliert.

Bei der Anbindung von RDF-Datenspeichern bedeutet das Fehlen, dass auf andere, gegebenenfalls proprietäre Mittel zum Ändern der Daten zurückgegriffen werden muss.

Von den angebundenen RDF-Datenspeichern unterstützen Virtuoso und Stardog die Update-Syntax. Die restlichen Systeme müssen auf proprietäre Schnittstellen zurückgreifen.

6.1.5 SPARQL ASK

Mit Hilfe von ASK-Abfragen lässt sich ermitteln, ob mindestens ein Tripel existiert, das den in der Abfrage definierten Kriterien genügt.

Abgesehen von Neo4j unterstützen alle Systeme diese Form der Abfrage. Da Teile von OntoWiki ASK-Abfragen verwenden, wurde diese Funktionalität durch den Neo4j-Connector simuliert, sodass das Fehlen an der Schnittstelle zur Applikation nicht erkennbar ist.

6.1.6 Reasoning

Reasoning bezeichnet das Ermitteln weiterer Fakten (Tripel), die nicht explizit im Grunddatenbestand enthalten sind, via Inferenz.

Dieses Feature ist zum Betrieb von OntoWiki nicht notwendig, kann jedoch die Datenverwaltung erleichtern.

Reasoning wird von Virtuoso, Oracle und Stardog unterstützt. Virtuoso und Oracle speichern ermittelte Fakten als Tripel ab, wohingegen Stardog die eingehenden SPARQL-Abfragen so umschreibt, dass auch abgeleitete Fakten zurückgegeben werden.

6.2 Vergleich der Performanz

Um die neuen sowie die bereits vorhandenen Lösungen in Hinblick auf ihre Performanz in Bezug zu setzen, werden diese im Folgenden einem Benchmark unterzogen. Um sicherzustellen, dass die Ergebnisse verlässlich sind, soll ein etablierter Benchmark verwendet werden.

Die während der Entwicklung zur Performanz-Optimierung verwendeten Leistungstests sind für einen einheitlichen Leistungsvergleich nur bedingt geeignet: Zum einen verwenden diese automatisiert erzeugte Daten und zum anderen werden nur kleine Datenmengen zur Prüfung herangezogen, um während der Implementierung in akzeptabler Zeit Feedback liefern zu können. Hinzu kommt, dass nur einen Teil der SPARQL-Features durch diese Tests abgedeckt wird.

Es existieren allerdings diverse etablierte Benchmarks, die sich mit der Abfrage von RDF-Daten via SPARQL beschäftigen. Dazu gehören der Lehigh University Benchmark [GPH05], SP^2 Bench [SHL+09] und der Berlin SPARQL Benchmark [BS09]. Diese Benchmarks generieren Testdaten für ein künstliches Szenario (z. B. aus dem Bereich E-Commerce beim Berlin SPARQL Benchmark) und analysieren anschließend die Ausführungsdauer ausgewählter SPARQL-Abfragen. Der DBpedia SPARQL Benchmark [MLA+11] verfolgt einen anderen Ansatz: Als Basis verwendet dieser Datenmengen in verschiedenen Größen aus dem realen Bestand von DBpedia. Die zur Bestimmung der Performanz herangezogenen SPARQL-Abfragen werden aus den tatsächlich an DBpedia übermittelten Abfragen ausgewählt. Dazu werden diese nach verwendeten Features gruppiert. Die am häufigsten vorkommenden Abfrage-Typen werden im Benchmark verwendet [MLA+11].

Synthetisch erzeugte Daten können eine homogene Struktur aufweisen, die einzelne Abfragen stark bevor- oder benachteiligt. Speziell für Benchmarks entwickelte Abfragen können sich auf Features fokussieren, die in der Praxis eine geringe Relevanz haben. Unter solchen Voraussetzungen ist nicht sichergestellt, dass Messungen ein realitätsnahes Bild ergeben. Daher orientiert sich die Performanz-Messung in dieser Arbeit am DBpedia-Benchmark, der als einziger reale Daten und Abfragen verwendet.

6.2.1 Zielsetzung

Im Gegensatz zum eigentlichen DBpedia Benchmark ist es nicht das Ziel dieser Arbeit, nur die Performanz der zugrunde liegenden RDF-Datenspeicher zu prüfen. Vielmehr geht es primär um die Leistungsfähigkeit der Implementierungen der verschiedenen Persistenzlösungen im Zusammenspiel mit den jeweiligen Datenbanksystemen.

Um dieser Absicht gerecht zu werden, wird der Performanz-Test komplett mit den Connector-Implementierungen durchgeführt. D. h. jede Interaktion mit den Datenbanken findet über die vorhandenen Implementierungen statt.

Die Funktionalität der angebundenen RDF-Datenspeicher ist in Bezug auf die unterstützten Features heterogen. Die ausschließliche Verwendung der Connector-Implementierungen hat daher außerdem den Vorteil, dass zum Durchführen des Benchmarks keine weitere Beschäftigung mit den individuellen Systemen (z. B. zum Import der Daten) erforderlich ist. Jede

Implementierung kann direkt geprüft werden. Nachteilig ist, dass spezielle Features der Datenspeicher (z. B. schnellere Import-Mechanismen) nicht genutzt werden können, wenn diese nicht transparent vom Connector unterstützt werden.

6.2.2 Datenbasis

Zum Testen werden die vom DBpedia Benchmark bereitgestellten Daten[1] aus dem Jahr 2010 verwendet.

Die Ergebnisse der während der Implementierung verwendeten synthetischen Benchmarks zeigen, dass ein Import von 100% oder sogar 200% der DBpedia-Daten über die PHP-Schnittstellen nicht in annehmbarer Zeit möglich ist. Daher wird für die Benchmarks eine Datenmenge verwendet, die 10% der damaligen Größe von DBpedia entspricht. Aus dieser Menge werden per Zufallsgenerator kleinere Datensets erzeugt, mit denen die Entwicklung der Performanz bei wachsender Datenmenge nachvollzogen werden soll. Um reproduzierbare Ergebnisse zu erhalten, wird der Zufallsgenerator so konfiguriert, dass er immer die gleiche Folge von Zahlen liefert. Tabelle 6.2 zeigt die verwendeten Datensets und ihre jeweilige Größe in Tripeln.

Tabelle 6.2: Größe der für den Benchmark verwendeten Datenmengen. Die Ausgangsgröße (100%) entspricht 10% des DBpedia-Datenbestands im Jahr 2010.

Größe in Prozent	*Ungefähre Anzahl Tripel*
100	15.000.000
25	3.750.000
5	750.000
1	150.000

Da Virtuoso und Stardog beim Import eine Validierung durchführen, werden außerdem kleinere Korrekturen an den Daten vorgenommen. Diese umfassen z. B. die Anpassung ungültiger Datumswerte.

6.2.3 Auswahl der SPARQL-Abfragen

Die zum Testen der Performanz verwendeten SPARQL-Abfragen werden aus dem DBpedia Benchmark übernommen. Dies hat den Vorteil, dass die Gruppierung und Bewertung der Abfragen nicht erneut vorgenommen werden muss. Zudem sind die Abfragen schon auf die verwendete Datenbasis abgestimmt.

[1] Unter http://aksw.org/Projects/DBPSB.html herunterladbar.

Insgesamt existieren 25 Vorlagen für SPARQL-Abfragen. Tabelle 6.3 zeigt eine Übersicht der Abfragen mit der Anzahl beteiligter Triple Patterns und den jeweils verwendeten Sprachfeatures. Die Vorlagen werden in

Tabelle 6.3: Abfrage-Typen und von diesen verwendete Sprachfeatures.

Nr.	*#TP*[a]	*DISTINCT*	*FILTER*	*OPT*[b]	*UNION*	*LANG*	*REGEX*	*STR*
1	1	✓						
2	1		✓					
3	5			✓				
4	4				✓			
5	9	✓			✓			
6	3	✓	✓					
7	12	✓		✓				
8	4		✓		✓			
9	9			✓	✓			
10	8	✓	✓	✓				
11	9	✓		✓	✓			
12	4	✓	✓			✓		
13	3		✓		✓	✓		
14	4		✓	✓		✓		
15	2	✓	✓				✓	✓
16	3	✓	✓	✓		✓		
17	4	✓	✓		✓	✓		
18	2		✓		✓	✓		✓
19	2		✓		✓		✓	✓
20	12		✓	✓	✓	✓		
21	1							
22	2							
23	3							
24	4							
25	5							

[a] Anzahl Triple Patterns
[b] `OPTIONAL`

Listing A.1 dargestellt. Jede Vorlage enthält mindestens einen mit Prozentzeichen markierten Platzhalter (z. B. %%var%%), der vor Ausführung der Abfrage durch eine konkrete Belegung ersetzt wird. So können aus einer Vorlage zahlreiche unterschiedliche SPARQL-Abfragen erzeugt werden. Dadurch werden die RDF-Datenspeicher daran gehindert die Ergebnisse für folgende Abfragen aus dem Cache auszuliefern. Mögliche Belegungen für die Platzhalter werden zu Beginn eines Testlaufs aus den verwendeten Daten ermittelt.

6.2.4 Setup

Der Benchmark wird auf einem handelsüblichen Laptop mit 2 GHz i7 Quadcore-Prozessor und 8 GB RAM ausgeführt. Als Betriebssystem kommt Windows 7 64 Bit zum Einsatz.

PHP wird in Version 5.5.6, Java in Version 1.7.0_51 verwendet. Zur Speicherung und Abfrage der RDF-Daten kommen die folgenden Datenbanksysteme zum Einsatz:

1. **Virtuoso** Open-Source Edition in Version 6.1.7, Anbindung via ODBC[2]
2. **MySQL** Community Edition in Version 5.6.14, Anbindung via PDO[3]
3. **Oracle 12c** mit Spatial and Graph-Option, Enterprise Edition Release 12.1.0.1.0, Anbindung via OCI8[4]
4. **Stardog** in Version 2.1.0, Anbindung via HTTP
5. **Neo4j** in Version 1.9.6, Anbindung via HTTP

Mit Ausnahme von Oracle wurden alle Datenbanksysteme direkt auf dem Testsystem installiert. Die Oracle Datenbank wurde wegen ihrer speziellen Systemanforderungen auf einer mit Virtual Box[5] verwalteten virtuellen Maschine installiert. Diese Maschine verwendet Oracle Linux als Betriebssystem und hat Zugriff auf 2 Prozessoren und 2 GB RAM.

Da eine umfassende Einarbeitung in die Details der Konfiguration aller Datenbanksysteme den Zeitrahmen der Arbeit gesprengt hätte, werden alle Systeme in ihrer Standardkonfiguration verwendet. Es wird davon ausgegangen, dass die Standardkonfigurationen in sich konsistent und für kleine bis mittlere Datenmengen hinreichend sind.

6.2.5 Ablauf des Benchmarks

Zu Beginn eines Testlaufs wird der untersuchte RDF-Datenspeicher zunächst neu gestartet, um sicherzustellen, dass dessen Zwischenspeicher leer ist.

Der eigentliche Testablauf wurde mit Hilfe des Benchmark-Frameworks Athletic[6] definiert und läuft völlig automatisiert ab. Es sind folgende Phasen zu unterscheiden:

1. **Import**: Aus der bereitgestellten Datenbasis im N-Triples-Format wird die gewünschte Menge in die Datenbank geladen.
2. **Vorbereitung**: Hilfsabfragen werden verwendet, um gültige Belegungen für die Platzhalter in den Abfrage-Vorlagen zu ermitteln. Für die Datenbank dient dies zudem als kurze Aufwärmphase. Die ausgeführten Abfragen stimmen nicht mit den später gemessenen überein.
3. **Test einzelner Abfrage-Typen**: Unter Verwendung der Abfrage-Vorlagen werden die 25 verschiedenen Abfrage-Typen getestet. Jeder Typ wird bis zu 1000-mal oder bis zum Erreichen der Zeitgrenze von 5 Minuten geprüft.
4. **Test von Abfrage-Kombinationen**: In dieser Phase werden Kombinationen der unterschiedlichen Abfragen an das Datenbanksystem gesendet. Diese Phase dauert bis zu 10 Minuten.

[2] http://www.php.net/manual/en/book.uodbc.php
[3] http://www.php.net/manual/en/book.pdo.php
[4] http://www.php.net/manual/en/book.oci8.php
[5] https://www.virtualbox.org/
[6] https://github.com/polyfractal/athletic

6.2.6 Ergebnisse

Die ermittelten Daten lassen sowohl Rückschlüsse auf die Gesamtperformanz als auch auf die Leistungsfähigkeit bzgl. bestimmter Abfrage-Typen und Features zu. Ein Vergleich der Leistung über verschiedene Datengrößen hinweg kann außerdem zeigen, wie gut eine Lösung skaliert. Zunächst soll die Importdauer betrachtet werden.

Dauer des Imports

Tabelle 6.4 zeigt die Dauer der Datenimporte für die verschiedenen Implementierungen.

Tabelle 6.4: Importdauer in Stunden.

	Datenmenge		
Connector	*1%*	*5%*	*25%*
Virtuoso	0,32	0,82	3,40
MySQL	0,93	3,54	21,83
Oracle	0,50	2,48	-
Stardog	0,37	0,88	4,60
Neo4j	0,91	3,59	68,25

Virtuoso und Stardog liefern hier die beste Leistung unter allen Lösungen. Im Vergleich zu den anderen Implementierungen sind diese Systeme ca. um den Faktor 3 schneller. Bei wachsender Datenmenge skaliert Virtuoso etwas besser als Stardog.

Für die Datenmengen 1% und 5% liefert Oracle, verglichen mit MySQL und Neo4j, relativ gute Resultate. Beim Importieren einer Datenmenge von 25% scheitert das System allerdings: Nach dem Erreichen von ca. 1,2 Millionen importierten Tripeln sank die Schreibgeschwindigkeit derart drastisch ab, dass der Import nach 3 Tagen unvollendet abgebrochen wurde. Für alle Datenmengen war außerdem festzustellen, dass Oracle Literale mit beispielsweise japanischen Schriftzeichen immer dann nicht importieren konnte, wenn diesen ein Sprachcode zugeordnet war.

Für die kleineren Datenmengen skaliert die Importgeschwindigkeit von Neo4j zunächst gut. Beim Vergrößern der Datenmenge um den Faktor 5 (von 5% auf 25%) bricht die Performanz ein und die Importdauer erhöht sich ca. um den Faktor 19.

Gesamtperformanz

Die Performanz des Gesamtsystems wird in durchgeführten Abfragekombinationen (englisch: Query Mixes) pro Stunde (QMpH) gemessen. Eine Abfragekombination ist eine Menge mit jeweils einem Vertreter aller 25 Abfrage-Typen.

Abbildung 6.1 visualisiert die Ergebnisse aus Tabelle A.2.

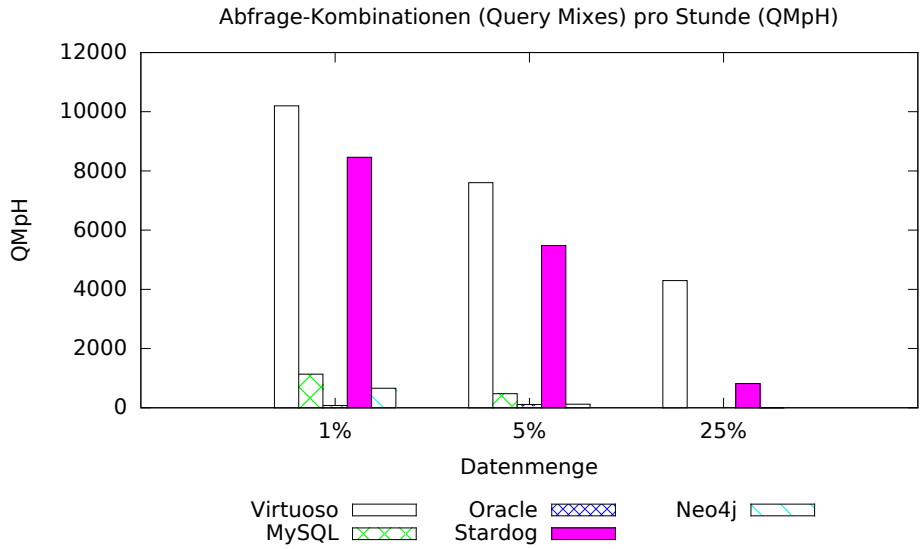


Abbildung 6.1: Abfrage-Kombinationen (Query Mixes) pro Stunde (QMpH).

In der Gesamtbetrachtung ist Virtuoso eindeutig das schnellste System, gefolgt von Stardog. Bei wachsender Datenmenge skaliert Virtuoso zudem besser als Stardog.

Angesichts der Tatsache, dass es sich nicht um ein auf RDF-Daten spezialisiertes Datenbank-Management-System handelt, liefert MySQL gute Ergebnisse für die Datenmengen der Größe 1% und 5%. Die relationale Datenbank ist hier ungefähr um den Faktor 9 bis 16 langsamer als Virtuoso. Die Datenmenge von 25% kann MySQL zwar importieren, allerdings die Abfragen in der Vorbereitungsphase nicht mehr in angemessener Zeit auf dieser ausführen. Der entsprechende Benchmark wurde nach ca. 24 Stunden in der Vorbereitungsphase abgebrochen, sodass hier keine weiteren Daten vorliegen.

Neo4j liefert keine überragenden Ergebnisse, kann die getesteten Abfragen aber auf allen Datenmengen ausführen. Beim Wechsel von 5% auf 25% Datenmenge ist ein massiver Performanzverlust (ca. Faktor 55) zu verzeichnen.

Die Oracle-Lösung liefert im gesamten Benchmark die schlechteste Leistung: Auf den testbaren Datenmengen von 1% und 5% ist sie allen anderen Implementierungen unterlegen. Interessanterweise zeigt der Benchmark einen Leistungsgewinn bei der Erhöhung der Datenmenge von 1% auf 5%. Die in Tabelle A.1 aufgeführten Daten zu den einzelnen Abfragen lassen vermuten, dass dies im Wesentlichen auf die in beiden Messungen langsam ausgeführte Abfrage 14 zurückzuführen ist. Diese erreicht für 1% Datenmenge 0,02 QpS sowie 0,04 QpS für 5% und fällt wegen der hohen Ausführungszeit insgesamt stark ins Gewicht. Durch die Verwendung des geometrischen Mittels wird der Einfluss solcher Extremwerte reduziert [Sch07]. Abbildung 6.2 zeigt das geometrische Mittel der QpS-Werte (Queries per Second) über alle Abfrage-Typen im Vergleich. Hier wird deutlich, dass Oracles Leistung beim

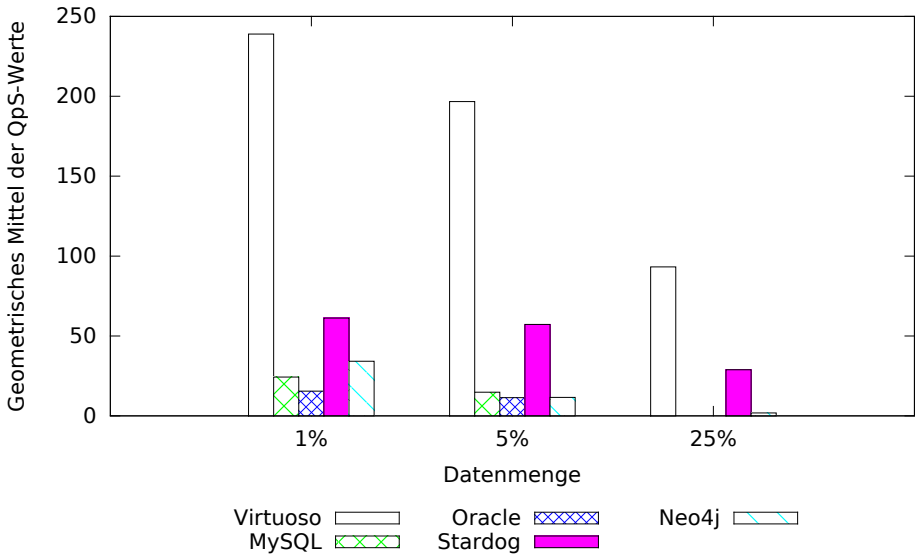


Abbildung 6.2: Geometrisches Mittel der QpS-Werte aller Abfrage-Typen (Tabelle A.3).

Wechsel von 1% auf 5% Datenmenge insgesamt abfällt.

Performanz der verschiedenen Abfrage-Typen

Tabelle A.1 zeigt die Performanz aller RDF-Datenspeicher bei der Ausführung der verschiedenen Abfrage-Typen. Die Leistung wird in Abfragen pro Sekunde (QpS) gemessen. Für die verschiedenen Datenmengen werden die Leistungsdaten in den Abbildungen 6.3, 6.4 und 6.5 visualisiert.

Virtuoso ist den anderen Datenbanksystemen in nahezu allen Abfrage-Typen überlegen. Lediglich für die Abfragen 9 und 20 liefert Stardog eine konstant bessere Leistung. Diese Abfragen verwenden die Sprachfeatures

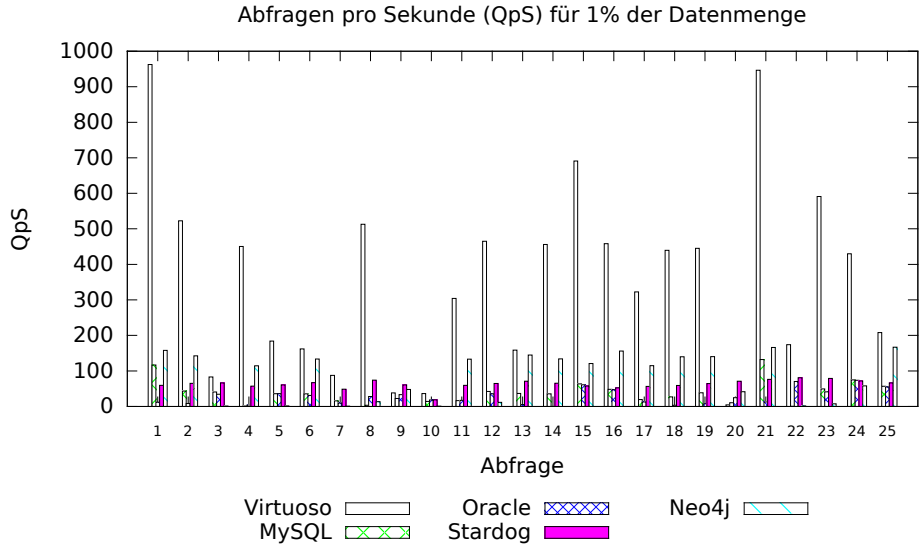


Abbildung 6.3: Abfragen pro Sekunde (QpS) für 1% der Datenmenge.

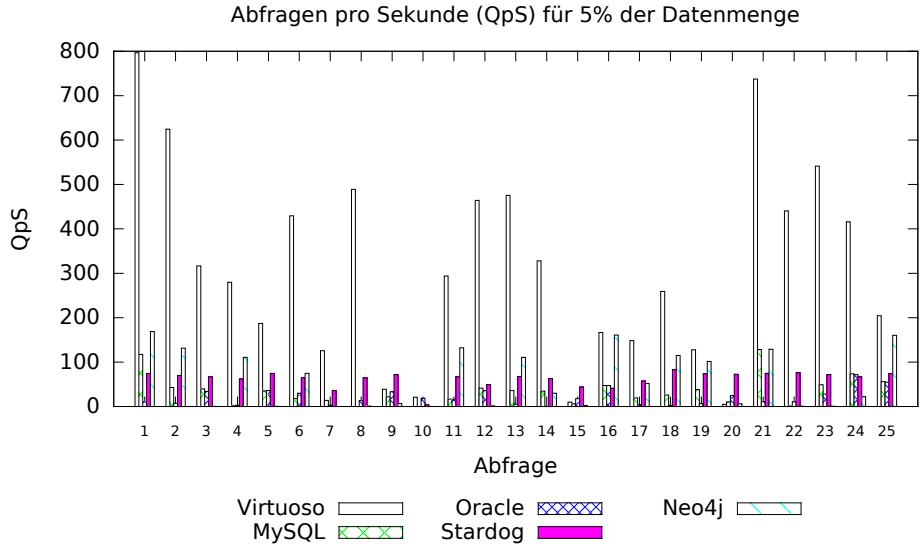


Abbildung 6.4: Abfragen pro Sekunde (QpS) für 5% der Datenmenge.

UNION und OPTIONAL bzw. UNION, OPTIONAL, FILTER und LANG. Dies lässt vermuten, dass Virtuoso die Kombination von UNION und OPTIONAL nicht sonderlich gut verarbeiten kann. Im Widerspruch dazu steht zunächst, dass Abfrage 11 ebenfalls die Features UNION und OPTIONAL verwendet, von Virtuoso aber deutlich schneller ausgeführt wird. Aller-

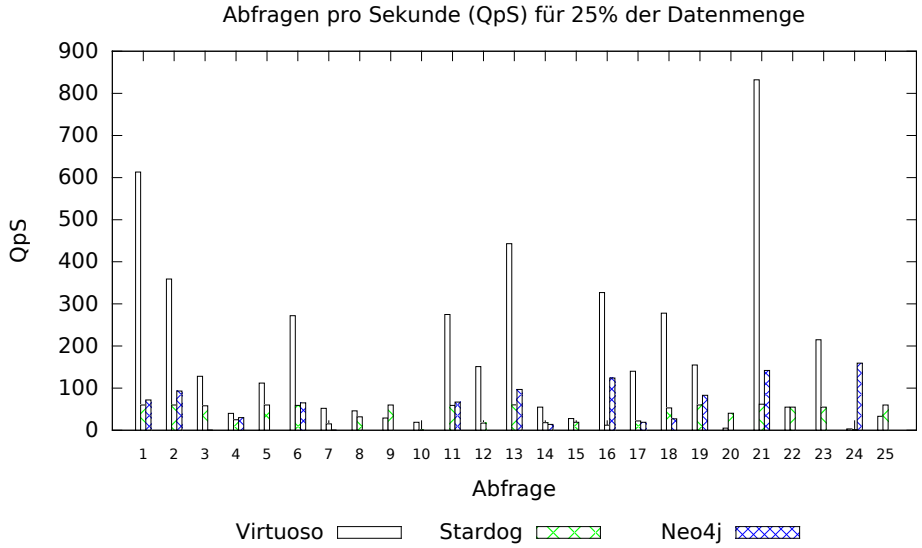


Abbildung 6.5: Abfragen pro Sekunde (QpS) für 25% der Datenmenge.

dings verwendet Abfrage 11 das OPTIONAL-Schlüsselwort nur innerhalb der UNION-Patterns, wohingegen die Abfragen 9 und 20 dieses im obersten Pattern (erst nach dem Zusammenführen der Mengen) nutzen.

Wie in [MLA+11] dokumentiert, liefert Virtuoso für Abfrage 10 eine, im Vergleich zu den meisten anderen Abfragen, relativ schlechte Performanz. Der Benchmark zeigt aber auch, dass die getesteten Alternativen keine bessere Leistung bieten.

Für Abfrage 15, welche die Features FILTER, REGEX, DISTINCT und STR nutzt, zeigt Virtuoso bei wachsender Datenmenge einen starken Leistungsverlust: Hier bricht die Performanz um über 90% ein.

MySQL liefert für die Abfragen 4 (UNION), 8 (UNION, FILTER) und 22 (2 Triple Patterns) schlechte Ergebnisse. Alle Abfragen verwenden Triple Patterns mit einem gemeinsamen Subjekt. Vergleiche mit der schnelleren Abfrage 5, die ebenfalls UNION und Triple Patterns mit gleichem Subjekt enthält, zeigen allerdings, dass dies nicht als alleiniges Kriterium für schlechte Performanz unter MySQL gelten kann. Da keine weiteren Gemeinsamkeiten erkennbar sind ist davon auszugehen, dass die Topologie der selektierten Daten hier maßgeblichen Einfluss auf die Geschwindigkeit hat.

Es stellte sich außerdem heraus, dass Abfrage 10 (OPTIONAL, FILTER, DISTINCT) unter MySQL extrem schlecht skaliert: Beim Wechsel von 1% auf 5% Datenmenge fielen die Abfragen pro Sekunde von 14,16 auf nahezu 0 (0,02).

Abfragen auf dem Datenset mit 25% Größe konnten von MySQL auch nach langer Laufzeit nicht mehr zuverlässig ausgeführt werden.

Der Oracle-Connector liefert insgesamt keine überragende Leistung. Die langsamste Abfrage ist Nummer 14 (OPTIONAL, FILTER, LANG), die bereits bei der Analyse der Gesamtperformanz in Abschnitt 6.2.6 aufgefallen ist.

Ebenfalls langsam sind die Abfragen 4 (UNION), 13 (UNION, FILTER, LANG) und 18 (UNION, FILTER, LANG, STR). Bei Berücksichtigung der Erfahrungen aus der anfänglichen Performanz-Optimierung der Oracle-Implementierung (vgl. Abschnitt 5.1.5) liegt der Schluss nahe, dass Abfragen mit Mengenoperationen wie UNION oder OPTIONAL, insbesondere bei zusätzlicher Verwendung von Filtern, zu extrem ungünstigen Ausführungsplänen führen können. Diese Pläne erfordern die Ermittlung des kartesischen Produkts der Daten und sind wenig performant.

Die Stardog-Anbindung zeichnet sich dadurch aus, dass sie keine großen Schwächen hat. Nahezu alle Abfrage-Typen werden mit akzeptabler Geschwindigkeit ausgeführt. Wie bereits zuvor festgestellt, wird bei einigen Abfragen (9 und 20) die Leistung von Virtuoso übertroffen. Lediglich Abfrage 10, welche die Features OPTIONAL, FILTER und DISTINCT verwendet, ist verhältnismäßig langsam und verzeichnet einen Geschwindigkeitsverlust von ca. 98% beim Wechsel von 1% auf 25% Datenmenge. Abfrage 24 skaliert gut beim Wechsel von 1% auf 5% Daten (ca. 6,5% Leistungsverlust), der Schritt zu einer Datenmenge von 25% führt allerdings zu einem drastischen Einbruch der QpS auf 1,64, was einem Leistungsverlust von ca. 97,5% entspricht.

Die für Neo4j gemessene Leistung ist inhomogen: Bei vielen Abfragen liefert die Lösung eine bessere Performanz als Stardog (z. B. in Abfrage 16 und 21) und reicht vereinzelt an Virtuoso heran (z. B. in Abfrage 16 bei Datenmenge 5%). Andererseits existieren zahlreiche Abfragen, die bereits auf kleinen Datenmengen schlechte Laufzeiten aufweisen. Dazu gehören beispielsweise die Abfragen 3, 7, 10, 22 und 23. Weitere Abfragen wie 8, 9, 12, 15, 20 und 25 skalieren bei wachsender Datenmenge extrem schlecht und erreichen QpS-Werte nahe 0. Auffällig ist, dass viele dieser Abfragen die Features FILTER und OPTIONAL verwenden. Zudem nutzen die Triple Patterns in den langsamen Abfragen meist die gleiche Variable im Subjekt, was den Schluss nahe legt, dass Neo4j diese Muster nicht sonderlich gut umsetzen kann.

Über alle getesteten Systeme hinweg lässt sich feststellen, dass die Anzahl der Triple Patterns in einer Abfrage keine eindeutigen Rückschlüsse auf die Performanz zulässt. Vielmehr scheint diese von der Beschaffenheit der Daten und der Art des beschriebenen Musters abzuhängen.

6.2.7 Fazit

In dem auf realen Daten basierenden DBpedia-Benchmark war Virtuoso sowohl in Hinblick auf reine Leistung als auch auf Skalierbarkeit der Gewinner.

Hilfreich dürfte unter anderem die schnelle Anbindung per ODBC gewesen sein, die es dem Virtuoso-Adapter erlaubt eine Verbindung zum Datenbank-Server offen zu halten, wohingegen Konkurrenten wie Stardog und Neo4j das zustandslose HTTP-Protokoll für die Kommunikation verwenden.

Zweitschnellstes System war Stardog, der über alle Abfrage-Typen hinweg solide Leistungen lieferte, aber nicht so gut skalierte wie Virtuoso.

Mit MySQL steht ein Datenbanksystem zur Verfügung, das kleinere Mengen von RDF-Daten gut verwalten kann. Am ehesten vergleichbar ist in diesem Bereich Neo4j. Die Graph-Datenbank leidet allerdings unter starken Leistungsschwankungen bei der Ausführung verschiedener Abfrage-Typen.

Verlierer des Benchmarks ist die Spatial and Graph-Lösung von Oracle: Diese unterliegt in der Gesamtübersicht allen anderen Connector-Implementierungen und bleibt so weit hinter den Versprechungen von Oracle (laut [Ora14] skalierbar bis zu mehreren Petabyte) zurück, dass ein Konfigurationsfehler des Systems nicht auszuschließen ist.

Kapitel 7

Evaluation der Anforderungsumsetzung

Im Folgenden wird geprüft, inwieweit die in Kapitel 3.3 formulierten Anforderungen erfüllt werden konnten.

7.1 Funktionale Anforderungen

Die funktionalen Anforderungen 1a, 1b und 1c sahen die Anbindung von OntoWiki an verschiedene Typen von RDF-Datenspeichern vor.

Mit Oracle (vgl. Kapitel 5.1) wurde ein System angebunden, das die Einbindung des Wikis in bestehende, traditionelle IT-Landschaften ermöglicht (Anforderung 1a).

Die in Anforderung 1b geforderte Anbindung eines Datenbank-Management-Systems mit nativer RDF-Unterstützung wurde durch die Integration von Stardog (Kapitel 5.2) erfüllt.

Mit Neo4j (vgl. Kapitel 5.3) konnte eine frei verfügbare Datenbank angebunden werden, die im Bereich der Graph-Datenbanken weit verbreitet ist und die Verarbeitung kleiner Mengen von RDF-Daten ermöglicht. Diese Eigenschaften prädestinieren Neo4j für den Einsatz als Einstiegs- und Test-Datenbank (Anforderung 1c). Aufgrund der wechselhaften Leistung (vgl. Abschnitt 6.2.6) muss einschränkend festgestellt werden, dass das Persistenz-System auf Basis von MySQL für diesen Zweck möglicherweise die bessere Wahl ist. Dafür spricht zudem, dass MySQL sich leichter installieren lässt als Neo4j (vgl. Abschnitt 6.1 und Tabelle 6.1).

7.2 Nicht-Funktionale Anforderungen

Da für nicht-funktionale Anforderungen häufig harte Kriterien fehlen, lässt sich deren Erfüllung schwerer prüfen.

7.2.1 Korrektheit (Anforderung 2a)

Die Korrektheit einer Implementierung lässt sich im Allgemeinen nicht nachweisen [Dij72]. Dennoch wurden während der Entwicklung Maßnahmen ergriffen, um die Korrektheit wesentlicher Abläufe sicherzustellen.

Die existierenden Integrations-Tests (vgl. Kapitel 4.1.4) wurden um weitere, detailliertere Tests ergänzt (vgl. Abschnitt 4.2.2). Diese automatisierten Tests definieren das von den Persistenz-Systemen erwartete Verhalten und stellen langfristig sicher, dass diese Erwartungen nicht verletzt werden [Bec02].

Um zu garantieren, dass die Tests bei Änderungen des Codes ausgeführt werden, wurde das Projekt an einen Continuous Integration-Server angebunden. Dieser führt die Tests bei Änderung des Quellcodes mit verschiedenen PHP-Versionen aus und garantiert schnelles Feedback, wenn ein Fehler in das Framework eingeführt worden sein sollte [DMG07].

7.2.2 Abwärtskompatibilität (Anforderung 2b)

Durch das Einführen einer Abstraktionsschicht unterhalb des existierenden Adapter-Konzepts konnte die Abwärtskompatibilität der Schnittstelle zwischen OntoWiki und Persistenzschicht erhalten werden (vgl. Kapitel 4.2.1).

Alle im Zuge der Arbeit angebundenen RDF-Datenspeicher wurden über diese neu geschaffene Connector-Schicht in die Infrastruktur des Erfurt-Frameworks integriert.

7.2.3 Plattformunabhängigkeit (Anforderung 2c)

Während der Entwicklung wurde darauf geachtet, dass keine Features verwendet werden, die nur von einzelnen Datenbanksystemen angeboten werden. Fehlende Basis-Funktionen, wie beispielsweise ASK-Abfragen im Fall von Neo4j (vgl. Kapitel 6.1.5), wurden in der Framework-Schicht simuliert. Vom Standard abweichendes Verhalten, z. B. die Verarbeitung von SPARQL-Abfragen durch Oracle, wurde durch Analyse und Umschreiben der Abfragen ebenfalls korrigiert (vgl. Kapitel 5.1.4).

Insgesamt zeigt die Anbindung von 5 verschiedenen Persistenz-Systemen mit unterschiedlichen Charakteristika, dass die Anforderung der Plattformunabhängigkeit erfüllt wurde.

7.2.4 Wartbarkeit (Anforderung 2d)

Die langfristige Wartbarkeit der neuen Komponenten wird durch die bereits erwähnten automatisierten Tests (vgl. Kapitel 4.1.4) gefördert. Diese definieren und prüfen das erwartete Verhalten, wodurch das Einführen von Fehlern bei Änderungen erschwert wird. Neu entdeckte Fehler sollten sich

schnell mit Hilfe der aufgebauten Test-Infrastruktur nachstellen lassen, wodurch ein zielgerichtetes Beheben der Defekte möglich wird.

Um die Verständlichkeit des Quellcodes zu verbessern, wurden die Bezeichnungen für diverse Konzepte vereinheitlicht. Die neu hinzugekommenen Implementierungen sprechen beispielsweise durchgängig von Graph und nicht von Model (vgl. Analyse in Kapitel 4.1.2). Ebenso wird einheitlich die Bezeichnung „Triple“ anstatt „Statement“ verwendet [Eva03].

Konzepte wie „Tripel“ oder „Quad“ wurden explizit herausgearbeitet, d. h. diese existieren als Klassen und ermöglichen ein intuitiveres Verständnis vom Zusammenspiel der Systembestandteile [Eva03]. Methoden können beispielsweise spezifizieren, dass sie ein entsprechendes Konzept als Parameter erwarten. Da die Informationen nicht mehr in Einzelteilen transportiert werden müssen, lassen sich lange, unübersichtliche Parameterlisten ebenfalls vermeiden [FBB+99].

Um die Arbeit mit dem Code zu erleichtern, wurde auf eine ausreichende Dokumentation geachtet. Beispielsweise wurden Ein- und Rückgabewerte mit ihren Wertebereichen sowie Hinweise zu Konzepten und Rollen von Klassen angegeben.

Ob die Maßnahmen die Wartbarkeit nachhaltig verbessert haben, wird sich in allen Details erst bei der zukünftigen Arbeit mit dem Quellcode zeigen. Die Anforderung 2d kann zum jetzigen Zeitpunkt zumindest als teilweise erfüllt betrachtet werden.

7.2.5 Erweiterbarkeit (Anforderung 2e)

Da das Thema dieser Arbeit die Generalisierung der Persistenzschicht ist, spielt die Erweiterbarkeit des Systems eine erhebliche Rolle. In Bezug auf diesen Punkt konnten große Fortschritte erzielt werden.

Die Trennung der Schnittstelle zu RDF- und relationalem Datenspeicher (vgl. Kapitel 4.2.1) ermöglicht es dem Entwickler, sich bei Erweiterungen des Systems auf einen konkreten Aspekt des Frameworks zu fokussieren. Zusätzlich erlaubt diese Strategie eine beliebige Kombination der bestehenden RDF- und relationalen Zugriffskomponenten.

Die Verkleinerung der Schnittstelle zu den RDF-Datenspeichern (vgl. Kapitel 4.2.1) von 16 auf 4 Methoden verringert den Entwicklungsaufwand für die initiale Anbindung neuer RDF-Datenspeicher. Dieser Effekt ließ sich bereits während der Anbindung von Oracle, Stardog und Neo4j feststellen.

Eine weitere Vereinfachung bei der Anbindung neuer Datenspeicher konnte durch das Einführen eines zentralen Ergebnisformats (vgl. Kapitel 4.2.1) erreicht werden: Diese Änderung separiert den Aspekt der Datenspeicher-Anbindung von dem der Ergebniskonvertierung und erlaubt das Hinzufügen neuer Formate für alle angebundenen Systeme an einer zentralen Stelle.

Das Konzept einer Installationskomponente (vgl. Kapitel 4.2.1) entkop-

pelt Betriebs- und Setup-Code. Der geschaffene, optionale Erweiterungspunkt vereinfacht die Weiterentwicklung, da das Installationspaket eingebunden werden kann, ohne den entwickelten Betriebscode zu modifizieren.

Das Einführen des Dependency Injection-Konzepts [Fow04] mitsamt einer etablierten Komponente zur Konfiguration der Objekt-Abhängigkeiten (vgl. Kapitel 4.2.1) fördert die Trennung der unterschiedlichen Belange des Systems und ermöglicht das Einhängen von Klassen, die nicht mit dem Projektcode ausgeliefert werden. Dies macht eine Erweiterung des Systems um externe Komponenten möglich.

Neben allen genannten strukturellen Maßnahmen, welche die Erweiterbarkeit fördern, wird Infrastruktur bereitgestellt, welche die Anbindung neuer RDF-Datenspeicher erleichtert. Dazu gehören generische Tests (vgl. Kapitel 4.2.2), die das von allen RDF-Datenspeichern erwartete Verhalten beschreiben und die genutzt werden können, um eine qualifizierte Aussage darüber zu treffen, wann die Anbindung eines Systems abgeschlossen ist.

Zusätzlich stehen Benchmarks zur Messung der Performanz zur Verfügung. Diese umfassen sowohl die im Implementierungsteil dieser Arbeit verwendeten Benchmarks mit synthetischen Daten (vgl. Kapitel 4.2.2) als auch den in Kapitel 6.2 vorgestellten DBpedia-Benchmark, der auf realen Daten beruht.

Auf Projektebene wurden ebenfalls Schritte zur Verbesserung der Erweiterbarkeit vorgenommen: Das Einführen eines Werkzeugs zur Verwaltung der Projektabhängigkeiten (vgl. Kapitel 4.2.2) erleichtert die Verwendung bestehender Bibliotheken. Der schnelle Zugriff auf frei verfügbare und etablierte Komponenten zur Lösung von Teilproblemen soll die notwendige Entwicklungszeit reduzieren.

Obwohl sich der Erfolg der vorgestellten Maßnahmen (analog zur Anforderung der Wartbarkeit) erst in der Zukunft in allen Details bewerten lässt, kann festgestellt werden, dass das Thema Erweiterbarkeit hinreichend berücksichtigt wurde, um Anforderung 2e als erfüllt ansehen zu können.

7.2.6 Performanz (Anforderung 2f)

Bereits während der Entwicklung der einzelnen Connector-Komponenten (vgl. Kapitel 5) wurde deren Performanz in verschiedenen Teilbereichen (Einfügen von Daten, Lesen von Datenmengen verschiedener Größe etc.) mit Hilfe von Benchmarks geprüft und gezielt verbessert.

Die in Anforderung 2f formulierte Erwartung einer maximalen Seitenladezeit von 1 Sekunde konnte dadurch für OntoWiki in den meisten Fällen gewährleistet werden. Es lässt sich allerdings feststellen, dass diese Ladezeit im Normalbetrieb wegen der eingesetzten Caching-Strategien [MUA10] kaum von der Leistung des RDF-Datenspeichers abhängt.

Die in Kapitel 6.2 durchgeführte Performanz-Analyse zeigt zudem die großen Leistungsunterschiede zwischen und innerhalb der verschiedenen

Connector-Implementierungen. Hier kann es bei Ausführung bestimmter Abfragen zu rapiden Leistungseinbrüchen kommen. Dies konnte bei der Verwendung von Neo4j als RDF-Datenspeicher nachgewiesen werden.

Da sich auf dieser Grundlage keine generelle Garantie für die Ausführungszeit geben lässt, kann die Anforderung 2f insgesamt nur als eingeschränkt erfüllt angesehen werden.

7.2.7 Transparente Installation (Anforderung 2g)

Das bislang mit dem Betriebscode verwobene Konzept der automatisierten Installation konnte aufgebrochen und in eine separate Komponente verschoben werden (vgl. Kapitel 4.2.1). Die neu angebundenen RDF-Datenspeicher verwenden diese, um Installationsaufgaben, z. B. das Anlegen von Datenbanken oder Tabellen, durchzuführen. Wie in Kapitel 6.1.3 dargestellt, ist die Installation einiger Systeme allerdings so komplex, dass diese nicht ohne manuelle Schritte auskommt. Die Anforderung der transparenten Installation kann daher nur teilweise als erfüllt angesehen werden.

7.3 Übersicht

Die meisten der gestellten Anforderungen konnten erfüllt werden. Tabelle 7.1 gibt einen abschließenden Überblick über die Anforderungen und den Status der Erfüllung.

Tabelle 7.1: Anforderungen und Umsetzungsstatus.

Anforderung	*Umsetzung*
1a Datenbank für gängige Infrastrukturen	✓
1b Nativer RDF-Datenspeicher	✓
1c Datenbank für Einstieg und Test	eingeschränkt
2a Korrektheit	ja, für alle überprüften Fälle
2b Abwärtskompatibilität	✓
2c Plattformunabhängigkeit	✓
2d Wartbarkeit	ja, soweit jetzt beurteilbar
2e Erweiterbarkeit	✓
2f Performanz	eingeschränkt
2g Transparente Installation	eingeschränkt

Die zur Erfüllung der nicht-funktionalen Anforderungen ergriffenen Maßnahmen unterstützen häufig mehrere Anforderungen. Tabelle 7.2 zeigt eine Übersicht der durchgeführten Maßnahmen und welche nicht-funktionalen Anforderungen damit adressiert werden.

Tabelle 7.2: Durchgeführte Maßnahmen und davon adressierte Anforderungen.

	Nicht-Funktionale Anforderungen						
Maßnahme	2a	2b	2c	2d	2e	2f	2g
Automatisierte Tests	✓			✓	✓		
Continuous Integration	✓			✓			
Verwaltung von Projektabhängigkeiten				✓	✓		
Trennung von RDF- und relationaler Schnittstelle				✓	✓		
Connector-Abstraktionsschicht		✓		✓	✓		
Dependency Injection				✓	✓		✓
Zentrales Ergebnisformat				✓	✓		
Simulation fehlenden Verhaltens			✓				
Korrektur abweichenden Verhaltens			✓				
Benchmark-Infrastruktur					✓	✓	
Installationskomponente				✓	✓		✓
Vereinheitlichung von Bezeichnungen				✓			
Explizite Modellierung von Konzepten				✓			
Dokumentation				✓	✓		

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

Ziel der Diplomarbeit war die Erweiterung der Persistenzschicht des semantischen Daten-Wikis OntoWiki um Schnittstellen zu weiteren RDF-Datenspeichern. Diese Aufgabe umfasste die Erstellung eines Konzeptes für die Generalisierung der Datenhaltungsschicht, die Implementierung der Schnittstellen zu weiteren Datenhaltungssystemen sowie den Vergleich dieser Systeme mit den bereits zuvor vorhandenen Lösungen.

Motiviert durch unterschiedliche Nutzungsszenarien für das Wiki wurden zunächst Anforderungen an den Entwurf sowie die neu anzubindenden Systeme entwickelt. Da die Persistenzschicht einen Teil der Infrastruktur ausmacht, waren diese Anforderungen primär nicht-funktional und behandelten Aspekte wie Wartbarkeit, Erweiterbarkeit und Performanz.

In der Entwurfsphase wurden zunächst die bestehende Struktur des Wikis und die vorhandenen Persistenzlösungen, Virtuoso und MySQL, analysiert. Daraufhin wurde ein Konzept entwickelt, das die Abwärtskompatibilität zum existierenden System sicherstellt, aber gleichzeitig den Anforderungen an erweiterbare und langfristig nutzbare Software gerecht wird.

In der Implementierungsphase wurden drei in ihren Charakteristika unterschiedliche Datenspeicher an das Wiki angebunden. Die Unterstützung der relationalen Oracle Datenbank ermöglicht den Betrieb des Wikis mit traditioneller IT-Infrastruktur. Mit Stardog wurde ein auf die Haltung semantischer Daten spezialisiertes System angebunden. Die Graph-Datenbank Neo4j ergänzt die Liste der nutzbaren Persistenzlösungen um ein frei verfügbares System. Während der Entwicklung wurde zudem Infrastruktur für zukünftige Erweiterungen bereitgestellt. Diese umfasst unter anderem generische Testfälle und wiederverwendbare Benchmarks.

Die Evaluation beschäftigte sich sowohl mit dem Vergleich der vorhande-

nen und neu hinzugekommenen Systeme in Bezug auf deren Eigenschaften und Performanz, als auch mit der Frage, welche der gestellten Anforderungen erfüllt werden konnten. Mit Hilfe des DBpedia-Benchmarks konnte festgestellt werden, dass keines der neu angebundenen Systeme das zuvor unterstützte Virtuoso im Bereich der Performanz übertreffen konnte. Lediglich Stardog überzeugte mit konstanter, solider Leistung, obwohl die Geschwindigkeit von Virtuoso meistens nicht erreicht wurde. Neo4j bot eine wechselhafte, stark von den ausgeführten Abfragen abhängige Leistung und die Oracle-Datenbank lieferte insgesamt schwache Ergebnisse. Von den initial definierten Anforderungen konnte der überwiegende Teil durch Entwurf und Implementierung erfüllt werden. Insbesondere in Bezug auf die Erweiterbarkeit waren Fortschritte zu verzeichnen.

8.2 Ausblick

Während der Arbeit mit dem Wiki ergaben sich zahlreiche neue Problemstellungen und Arbeitsansätze, die in der zur Verfügung stehenden Zeit nicht hinreichend bearbeitet werden konnten. Dieses Kapitel enthält Vorschläge, die bei der zukünftigen Arbeit mit OntoWiki aufgegriffen werden können.

8.2.1 Abhängigkeit zu relationaler Datenbank auflösen

Derzeit verwendet OntoWiki zur Versionierung und zum Speichern von Cache-Daten eine relationale Datenbank. Im Zuge dieser Arbeit konnte die zugehörige Schnittstelle von der RDF-Funktionalität getrennt werden (vgl. Kapitel 4.2.1), wodurch eine separate Betrachtung der Abhängigkeit vom relationalen Modell möglich wurde. Wegen der Komplexität der Interaktion mit dem bestehenden System war ein kompletter Austausch der Abhängigkeit in der zur Verfügung stehenden Zeit allerdings nicht möglich.

Damit beim Einsatz von rein semantischen Datenbanksystemen, wie beispielsweise Stardog, die Notwendigkeit des Parallelbetriebs einer relationalen Datenbank entfallen kann, wäre es denkbar, die bislang anfallenden relationalen Daten ebenfalls als RDF-Daten zu speichern. Hinweise zum Mapping von relationalen Daten auf das RDF-Modell liefern [DSC12], [ABP+12] und [ADL+09].

8.2.2 Verbesserung des SPARQL-Parsers

Bei der Entwicklung des Oracle-Connectors waren Analyse und Umschreiben der eingehenden SPARQL-Abfragen notwendig, um in Erfurt eine einheitliche Verarbeitung dieser sicherzustellen.

Dabei traten Unzulänglichkeiten der mitgelieferten Komponenten zur SPARQL-Verarbeitung zu Tage. Beispielsweise wurden neuere Konstrukte wie Aliase nicht unterstützt und gültige FILTER-Ausdrücke wurden bei

ungünstiger Formatierung nicht korrekt erkannt. Problematisch war außerdem, dass kein einheitliches System zum Analysieren und Konstruieren von SPARQL-Abfragen bereitgestellt wurde. Dadurch wurde eine aufwändige Übertragung der Daten zwischen Analyse- und Parser-Komponente notwendig.

Da die Analyse und Modifikation von SPARQL-Abfragen, sowohl in OntoWiki als auch in Erfurt, ein zentrales Thema ist, wäre es wünschenswert, zukünftig eine stabile Komponente zum Durchführen dieser Operationen zu haben. Diese sollte sowohl neue SPARQL-Konstrukte (Update-Syntax etc.) handhaben können als auch gängige Modifikationen, wie das Umbenennen von Variablen oder das Ersetzen von Teilkomponenten (z. B. Literalen), ermöglichen. Eine stabile Infrastruktur in diesem Bereich dürfte die weitere Entwicklung beschleunigen. Die Tatsache, dass die SPARQL-Analyse auch außerhalb von Erfurt interessant ist und das Fehlen entsprechender Lösungen im PHP-Ökosystem, spricht zudem dafür, eine solche Komponente als separates Paket zu veröffentlichen.

8.2.3 Extraktion weiterer Bibliotheken aus Erfurt

Durch die Einführung eines Dependency Management Tools wurden die Abhängigkeiten des Erfurt-Frameworks explizit gemacht. Während der Entwicklung zeigte sich, dass die deklarierten Abhängigkeiten häufig nur von einem Teil des Systems genutzt wurden und nicht immer für den Betrieb erforderlich waren. Beispiele für solche Abhängigkeiten sind der HTTP-Client, der nur für die Arbeit mit Webservices erforderlich ist, oder die Datenbank-Abstraktion, die nur vom Oracle-Connector genutzt wird.

Dies führt zu mehreren Problemen: Zum einen werden bei der Installation Bibliotheken mitgeladen, die von der Anwendung später nicht verwendet werden. Zum anderen lassen sich Abhängigkeiten zu kompilierten PHP-Extensions (z. B. ODBC für Virtuoso oder OCI8 für Oracle) nicht explizit angeben, da diese nicht automatisch nachinstalliert werden können und den Installationsprozess für viele PHP-Konfigurationen fehlschlagen lassen würden. Zur Verdeutlichung zeigt Abbildung 8.1 die derzeit wichtigsten Abhängigkeiten von Erfurt.

Eine mögliche Lösung dieses Problems wäre es, den Code zur Anbindung der verschiedenen RDF-Datenspeicher jeweils als eigene Bibliothek bereitzustellen. Abbildung 8.2 visualisiert die dadurch entstehende Struktur. Erfurt schlägt nur noch die Installation der unterstützten Connector-Bibliotheken vor. Was auf den ersten Blick komplizierter aussieht, hat in Bezug auf die Wart- und Erweiterbarkeit zahlreiche Vorteile: Da die Connector-Pakete nicht mehr zwangsweise mitinstalliert werden, können diese ihre Abhängigkeiten (auch kompilierte Extensions) explizit definieren. Dies stellt sicher, dass eine Connector-Installation nur dann möglich ist, wenn alle Voraussetzungen für den Einsatz erfüllt sind. Spezifische Abhängigkeiten (z. B. der

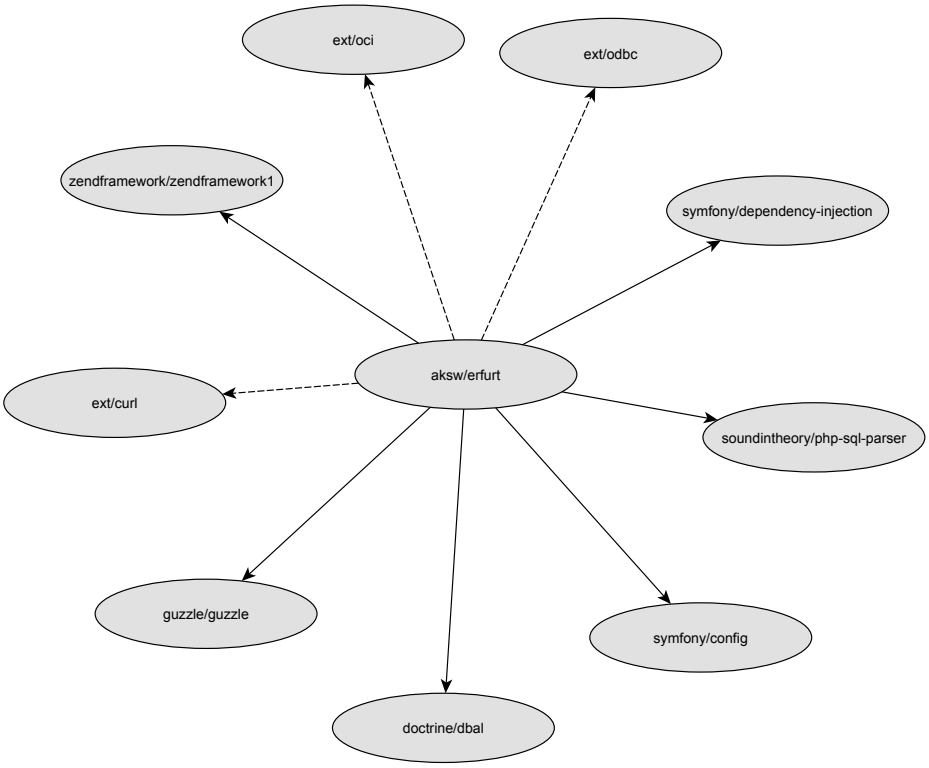


Abbildung 8.1: Derzeitige Struktur der Abhängigkeiten des Erfurt-Frameworks. Vorgeschlagene Komponenten sind mit gestrichelten Linien verbunden.

HTTP-Client Guzzle) werden nur installiert, wenn diese vom eingesetzten Connector benötigt werden. Außerdem ermöglicht eine Aufteilung die separate Weiterentwicklung der Connector-Implementierungen.

8.2.4 Unterstützung von Datei-Importen

Da unter den angebundenen RDF-Datenspeichern nur Stardog den Import von RDF-Daten aus Dateien in gängigen Formaten wie N-Triples oder N-Quads unterstützte und dieser einer veralteten Version des Standards folgte, wurde der Datei-Import bei der Implementierung bislang vernachlässigt.

Bei großen Datenmengen verspricht dieser Ansatz allerdings eine Verbesserung der Import-Dauer, da das Parsen nicht mehr innerhalb der PHP-Applikation erfolgen muss. Das entworfene Connector-Interface sieht keine Methoden zum Import von Dateien vor und die Tatsache, dass diese von einem großen Teil der RDF-Datenspeicher nicht bedient werden könnten, spricht dagegen das Interface zum Hinzufügen dieses Features zu vergrößern.

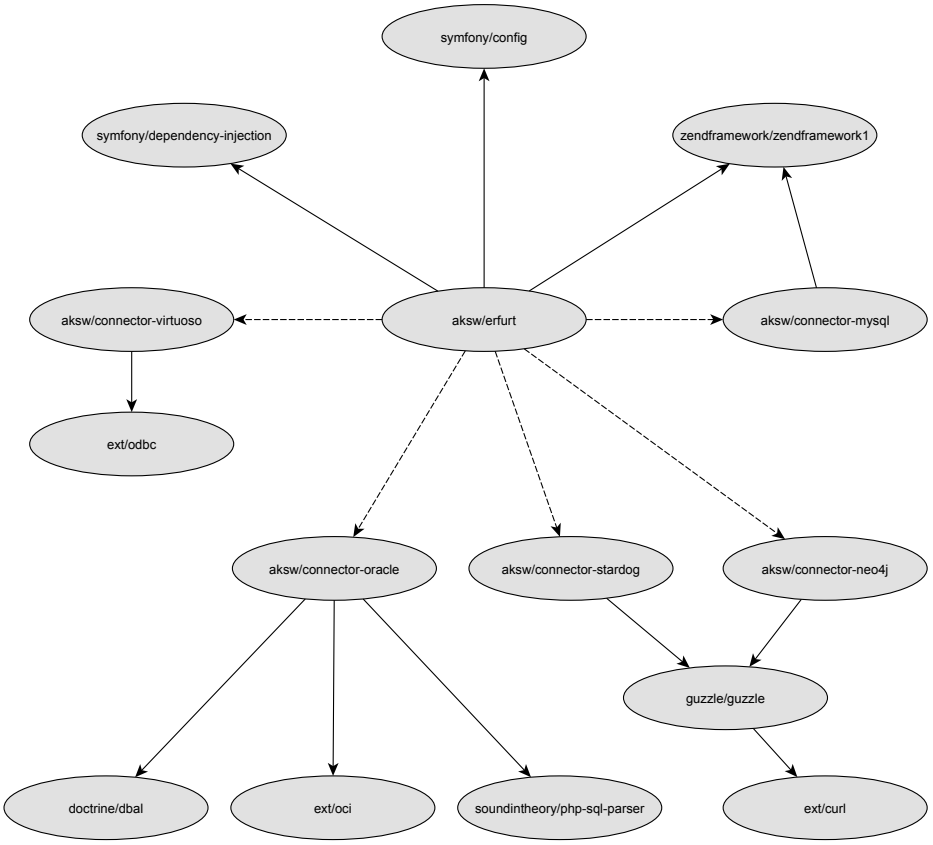


Abbildung 8.2: Erfurt-Abhängigkeiten bei Einführung eigener Bibliotheken pro Connector. Vorgeschlagene Komponenten sind mit gestrichelten Linien verbunden.

Gegen das Einführen eines zusätzlichen Interfaces sprechen die schlechten Erfahrungen, die mit den in Erfurt verwendeten, separaten Interfaces für den Zugriff auf relationale bzw. semantische Daten gemacht wurden (vgl. Kapitel 4.1.5).

Eine mögliche Lösung für das Problem, die eine lose Kopplung fördert, könnte in den Konzepten der aspektorientierten Programmierung zu finden sein [Boe06]. Diese ermöglicht das Registrieren und Ausführen von zusätzlichem Code bei ausgewählten Methodenaufrufen. Die Unterstützung für aspektorientierte Programmierung ist im PHP-Bereich unzulänglich, die Funktionalität lässt sich aber mit einem Event-System simulieren. Implementierungen werden von einigen Frameworks zur Verfügung gestellt[1]. Das Konzept hinter diesen wird in [GHJ+94] als Mediator-Muster beschrieben.

[1]Beispielsweise vom Zend Framework 2 (http://framework.zend.com/) oder von Symfony 2 (http://symfony.com/).

Quellenverzeichnis

[ABK+07] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak und Zachary Ives. „DBpedia: A Nucleus for a Web of Open Data“. In: *The Semantic Web*. Bd. 4825. Springer, 2007, S. 722–735.

[ABP+12] Marcelo Arenas, Alexandre Bertails, Eric Prud'hommeaux und Juan Sequeda. *A Direct Mapping of Relational Data to RDF*. W3C. Sep. 2012. URL: http://www.w3.org/TR/rdb-direct-mapping/ (besucht am 09. 04. 2014).

[ADL+07] Sören Auer, Sebastian Dietzold, Jens Lehmann und Thomas Riechert. „OntoWiki: A Tool for Social, Semantic Collaboration“. In: *Proc. WWW 2007 Workshop on Social and Collaborative*. Banff, Canada, Mai 2007.

[ADL+09] Sören Auer, Sebastian Dietzold, Jens Lehmann, Sebastian Hellmann und David Aumueller. „Triplify - Light-Weight Linked Data Publication from Relational Databases“. In: *18th International World Wide Web Conference*. Apr. 2009, S. 621–621.

[BBP+14] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux und Gavin Carothers. *RDF 1.1 Turtle. Terse RDF Triple Language*. W3C. Feb. 2014. URL: http://www.w3.org/TR/turtle/ (besucht am 17. 04. 2014).

[BCS14] David Beckett, Gavin Carothers und Andy Seaborne. *RDF 1.1 N-Triples. A line-based syntax for an RDF graph*. W3C. Feb. 2014. URL: http://www.w3.org/TR/n-triples/ (besucht am 05. 05. 2014).

[Bec02] Kent Beck. *Test Driven Development. By Example*. Addison Wesle, Nov. 2002. ISBN: 978-0321146533.

[BG04] Jan Grant und Dave Beckett. *RDF Test Cases*. W3C. Feb. 2004. URL: http://www.w3.org/TR/2004/REC-rdf-testcases-20040210/ (besucht am 14. 04. 2014).

[BGM14] Dan Brickley, R.V. Guha und Brian McBride. *RDF Schema 1.1*. W3C. Feb. 2014. URL: http://www.w3.org/TR/rdf-schema/ (besucht am 01. 05. 2014).

[BHL01] Tim Berners-Lee, James Hendler und Ora Lassila. „The Semantic Web“. In: *Scientific American* 284.5 (Mai 2001), S. 34–43.

[BL14] Dan Brickley und Libby Miller. *FOAF Vocabulary Specification 0.99*. Jan. 2014. URL: http://xmlns.com/foaf/spec/ (besucht am 06. 04. 2014).

[BMM98] William H. Brown, Raphael C. Malveau, Hays W. McCormick und Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architecture and Projects in Crisis*. John Wiley & Sons, März 1998.

[BMP+12] Paul V. Biron, Ashok Malhotra, David Peterson, Shudi Gao, C. M. Sperberg-McQueen und Henry S. Thompson. *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*. W3C. Apr. 2012. URL: http://www.w3.org/TR/xmlschema11-2/ (besucht am 05. 05. 2014).

[Boe06] Oliver Böhm. *Aspektorientierte Programmierung mit AspectJ 5: Einsteigen in AspectJ und AOP*. Dpunkt.Verlag GmbH, 2006. ISBN: 978-3898643306.

[BS09] Christian Bizer und Andreas Schultz. „The Berlin SPARQL benchmark“. In: *International Journal on Semantic Web and Information Systems (IJSWIS)* 5.2 (2009), S. 1–24.

[Car14] Gavin Carothers. *RDF 1.1 N-Quads. A line-based syntax for RDF datasets*. W3C. Feb. 2014. URL: http://www.w3.org/TR/n-quads/ (besucht am 14. 04. 2014).

[Dij72] Edsger W. Dijkstra. „The Humble Programmer“. In: *Commun. ACM* 15.10 (1972), S. 859–866.

[DMG07] Paul M. Duvall, Steve Matyas und Andrew Glover. *Continuous Integration. Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007. ISBN: 978-0321336385.

[DSC12] Souripriya Das, Seema Sundara und Richard Cyganiak. *R2RML: RDB to RDF Mapping Language*. W3C. Sep. 2012. URL: http://www.w3.org/TR/r2rml/ (besucht am 09. 04. 2014).

[DuC11] Bob DuCharme. *Learning SPARQL. Querying and Updating with SPARQL 1.1*. O'Reilly, Juli 2011. ISBN: 978-1-449-30659-5.

[Eva03] Eric Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Boston, MA, USA: Addison-Wesley, 2003. ISBN: 978-0321125217.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke und Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. ISBN: 978-0201485677.

[FBC+10] David A. Ferrucci, Eric W. Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya Kalyanpur, Adam Lally, J. William Murdock, Eric Nyberg, John M. Prager, Nico Schlaefer und Christopher A. Welty. „Building Watson: An Overview of the DeepQA Project." In: *AI Magazine* 31.3 (2010), S. 59–79.

[FMT+14] Philipp Frischmuth, Michael Martin, Sebastian Tramp, Thomas Riechert und Sören Auer. „OntoWiki - An Authoring, Publication and Visualization Interface for the Data Web". Noch nicht veröffentlicht, aber nach Review akzeptiert. 2014. URL: http://www.semantic-web-journal.net/system/files/swj490.pdf (besucht am 13. 04. 2014).

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002. ISBN: 978-0321127426.

[Fow04] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. Jan. 2004. URL: http://martinfowler.com/articles/injection.html (besucht am 19. 04. 2014).

[GHJ+94] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN: 0-201-63361-2.

[GPH05] Yuanbo Guo, Zhengxiang Pan und Jeff Heflin. „LUBM: A benchmark for OWL knowledge base systems". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 3.2 (2005), S. 158–182.

[GPP13] Paul Gearon, Alexandre Passant und Axel Polleres. *SPARQL 1.1 Update*. W3C. März 2013. URL: http://www.w3.org/TR/sparql11-update/ (besucht am 05. 06. 2014).

[GSB14] Fabien Gandon, Guus Schreiber und Dave Beckett. *RDF 1.1 XML Syntax*. W3C. Feb. 2014. URL: http://www.w3.org/TR/rdf-syntax-grammar/ (besucht am 11. 05. 2014).

[HDM+09] Norman Heino, Sebastian Dietzold, Michael Martin und Sören Auer. „Developing Semantic Web Applications with the OntoWiki Framework". In: *Networked Knowledge - Networked Media*. Hrsg. von Tassilo Pellegrini, Sören Auer, Klaus Tochtermann und Sebastian Schaffert. Bd. 221. Studies in Computational Intelligence. Berlin / Heidelberg: Springer, 2009, S. 61–77.

[Hic13] Ian Hickson. *HTML Microdata*. W3C. Okt. 2013. URL: http://www.w3.org/TR/microdata/ (besucht am 12. 06. 2014).

[HKR+06] Pascal Hitzler, Markus Krötzsch, Sebastian Rudolph und York Sure. *Semantic Web: Grundlagen*. Heidelberg: eXamen.press, Springer, 2006. ISBN: 978-3-540-33993-9.

[HS13] Steve Harris und Andy Seaborne. *SPARQL 1.1 Query Language*. W3C. März 2013. URL: http://www.w3.org/TR/sparql11-query/ (besucht am 31. 03. 2014).

[HW03] Gregor Hohpe und Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison Wesley, Okt. 2003. ISBN: 978-0321200686.

[JH06] Christopher Jones und Alison Holloway. *The Underground PHP and Oracle® Manual*. Release 1.2. Oracle. 2006. URL: http://www.oracle.com/technetwork/topics/php/underground-php-oracle-manual-098250.html (besucht am 24. 03. 2014).

[KVD+07] Markus Krötzsch, Denny Vrandecic, Max Völkel, Heiko Haller und Rudi Studer. „Semantic Wikipedia“. In: *Journal of Web Semantics* 5 (Dez. 2007), S. 251–261.

[Mar08] Robert C. Martin. *Clean Code. A Handbook of Agile Software Craftsmanship*. Prentice Hall, Aug. 2008. ISBN: 978-0132350884.

[MLA+11] Mohamed Morsey, Jens Lehmann, Sören Auer und Axel-Cyrille Ngonga Ngomo. „DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data“. In: *The Semantic Web – ISWC 2011*. Hrsg. von Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham Bernstein, Lalana Kagal, Natasha Noy und Eva Blomqvist. Bd. 7031. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, S. 454–469. ISBN: 978-3-642-25072-9.

[MUA10] Michael Martin, Jörg Unbehauen und Sören Auer. „Improving the Performance of Semantic Web Applications with SPARQL Query Caching“. In: *Proceedings of 7th Extended Semantic Web Conference (ESWC 2010), 30 May – 3 June 2010, Heraklion, Crete, Greece*. Hrsg. von Lora Aroyo, Grigoris Antoniou, Eero Hyvönen, Annette ten Teije, Heiner Stuckenschmidt, Liliana Cabral und Tania Tudorache. Bd. 6089. Lecture Notes in Computer Science. Berlin / Heidelberg: Springer, 2010, S. 304–318.

[Mur14] Chuck Murray. *Oracle® Spatial and Graph. RDF Semantic Graph Developer's Guide*. 12c Release 1 (12.1). Oracle. Jan. 2014. URL: http://docs.oracle.com/cd/E16655_01/appdev.121/e17895.pdf (besucht am 23. 03. 2014).

[Ora14] Oracle. *Oracle Spatial and Graph*. 2014. URL: http://download.oracle.com/otndocs/products/spatial/pdf/12c/oraspatitalandgraph_12_fo.pdf (besucht am 04. 06. 2014).

[Owl12] W3C OWL Working Group. *OWL 2 Web Ontology Language*. W3C. Dez. 2012. URL: http://www.w3.org/TR/owl-overview/ (besucht am 01. 05. 2014).

[PS08] Eric Prud'hommeaux und Andy Seaborne. *SPARQL Query Language for RDF*. W3C. Jan. 2008. URL: http://www.w3.org/TR/rdf-sparql-query/ (besucht am 31. 03. 2014).

[QS13] Qudamah Quboa und Mohamad Saraee. „A State-of-the-Art Survey on Semantic Web Mining". In: *Intelligent Information Management* 5 (2013), S. 10–17.

[RR06] Suzanne Robertson und James Robertson. *Mastering the Requirements Process*. 2. Aufl. Addison Wesley, März 2006. ISBN: 978-0321419491.

[RWE13] Ian Robinson, Jim Webber und Emil Eifrem. *Graph Databases*. O'Reilly, 2013. ISBN: 978-1-4493-5626-2.

[SCG+13] Andy Seaborne, Kendall Grant Clark, Lee Feigenbaum und Elias Torres. *SPARQL 1.1 Query Results JSON Format*. W3C. März 2013. URL: http://www.w3.org/TR/sparql11-results-json/ (besucht am 19. 04. 2014).

[Sch07] Peter M. Schulze. *Beschreibende Statistik*. 6. Aufl. Oldenbourg, 2007. ISBN: 978-3-486-58220-8.

[SHL+09] Michael Schmidt, Thomas Hornung, Georg Lausen und Christoph Pinkel. „SP2Bench: A SPARQL Performance Benchmark." In: *Semantic Web Information Management*. Hrsg. von Roberto De Virgilio, Fausto Giunchiglia und Letizia Tanca. Springer, 2009, S. 371–393. ISBN: 978-3-642-04328-4.

[SK09] Jerome H. Saltzer und Frans M. Kaashoek. *Principles of Computer System Design. An Introduction*. Morgan Kaufmann, 2009. ISBN: 978-0-12-374957-4.

[SRM+14] Guus Schreiber, Yves Raimond, Frank Manola, Eric Miller und Brian McBride. *RDF 1.1 Primer*. W3C. Feb. 2014. URL: http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140225/ (besucht am 01. 05. 2014).

[TFH10] Sebastian Tramp, Philipp Frischmuth und Norman Heino. „OntoWiki – a Semantic Data Wiki Enabling the Collaborative Creation and (Linked Data) Publication of RDF Knowledge Bases". In: *Demo Proceedings of the EKAW 2010*. Hrsg. von Oscar Corcho und Johanna Voelker. Okt. 2010.

[THA+10] Sebastian Tramp, Norman Heino, Sören Auer und Philipp Frischmuth. „Making the Semantic Data Web easily writeable with RDFauthor“. In: *Proceedings of 7th Extended Semantic Web Conference (ESWC 2010)*. Bd. 6089. Lecture Notes in Computer Science. Springer. Berlin / Heidelberg, 2010, S. 436–440.

[TW14] ThoughtWorks Technology Advisory Board. *TECHNOLOGY RADAR*. Jan. 2014. URL: http://thoughtworks.fileburst.com/assets/technology-radar-jan-2014-en.pdf (besucht am 29. 03. 2014).

Abbildungsverzeichnis

2.1 Tripel aus Listing 2.1, visualisiert als Graph. 9
2.2 Nutzeroberfläche von OntoWiki. 14

4.1 Komponenten im Erfurt-Framework. 26
4.2 `Erfurt_Store` und relevante angrenzende Elemente. 28
4.3 Klassendiagramm der im Erfurt-Framework bereits existierenden Adapter für Virtuoso und MySQL. 30
4.4 Konzept zur Anbindung weiterer Datenspeicher via Connector. 35
4.5 Implementierung von Querschnittsbelangen mittels Decorator am Beispiel des Loggings von langsamen Abfragen. 37
4.6 Aufbau der Factory, die zum Erzeugen eines Adapters einen Dependency Injection Container verwendet. 38
4.7 Infrastruktur zum Umformen von Abfrageergebnissen. 40
4.8 Puffer-System zum Verzögern und Zusammenfassen von Speicheroperationen. 41

5.1 Speicherung von RDF-Daten in einer Oracle Datenbank nach [Mur14]. 46
5.2 SPARQL-Connector für Oracle mit angrenzenden Komponenten. 50
5.3 Verarbeitung einer SPARQL-Abfrage durch den Oracle-Connector. 50
5.4 Auslesen von Daten mit Literalen kleiner als 4000 Byte. . . . 58
5.5 Auslesen von Daten mit Literalen größer als 4000 Byte. . . . 58
5.6 Leistungsvergleich der vier Speicherstrategien in drei verschiedenen Szenarios. 62
5.7 Performanz der Abfrage aus Listing 5.10 sowie äquivalenter Abfragen unter Verwendung verschiedener Query Hints. . . . 64
5.8 SPARQL-Connector für Stardog inklusive angrenzender Komponenten. 67
5.9 Abnahme der Performanz in Abhängigkeit von den offenen Transaktionen. 69

5.10 Leistungsvergleich von zwei Speicherstrategien in drei verschiedenen Szenarios. 70
5.11 Beispiel für einen Property-Graph. 71
5.12 In einen Property-Graph überführte Tripel aus 5.12. 73
5.13 SPARQL-Connector für Neo4j inklusive angrenzender Komponenten. 74

6.1 Abfrage-Kombinationen (Query Mixes) pro Stunde (QMpH). 83
6.2 Geometrisches Mittel der QpS-Werte aller Abfrage-Typen (Tabelle A.3). 84
6.3 Abfragen pro Sekunde (QpS) für 1% der Datenmenge. 85
6.4 Abfragen pro Sekunde (QpS) für 5% der Datenmenge. 85
6.5 Abfragen pro Sekunde (QpS) für 25% der Datenmenge. . . . 86

8.1 Derzeitige Struktur der Abhängigkeiten des Erfurt-Frameworks. Vorgeschlagene Komponenten sind mit gestrichelten Linien verbunden. 98
8.2 Erfurt-Abhängigkeiten bei Einführung eigener Bibliotheken pro Connector. Vorgeschlagene Komponenten sind mit gestrichelten Linien verbunden. 99

Tabellenverzeichnis

2.1 Beispiele für Aussagen in Tripel-Form in natürlicher Sprache. 6
2.2 Gekürzte Liste der Abfrageergebnisse aus Listing 2.4. 10

5.1 Mögliches Ergebnis der Abfrage aus Listing 5.3. 53

6.1 Eigenschaften der verschiedenen RDF-Datenspeicher. 75
6.2 Größe der für den Benchmark verwendeten Datenmengen. . . 79
6.3 Abfrage-Typen und von diesen verwendete Sprachfeatures. . . 80
6.4 Importdauer in Stunden. 82

7.1 Anforderungen und Umsetzungsstatus. 93
7.2 Durchgeführte Maßnahmen und davon adressierte Anforderungen. 94

A.1 Performanz einzelner Abfrage-Typen in Queries per Second (QpS). 122
A.1 Performanz einzelner Abfrage-Typen in Queries per Second (QpS) (Fortsetzung). 123
A.1 Performanz einzelner Abfrage-Typen in Queries per Second (QpS) (Fortsetzung). 124
A.1 Performanz einzelner Abfrage-Typen in Queries per Second (QpS) (Fortsetzung). 125
A.1 Performanz einzelner Abfrage-Typen in Queries per Second (QpS) (Fortsetzung). 126
A.2 Anzahl von Abfrage-Kombinationen (Query Mixes) pro Stunde (QMpH). 127
A.3 Geometrisches Mittel der QpS aller Abfragen. 127

Listingverzeichnis

2.1 Tripel aus Tabelle 2.1 kodiert mit Hilfe von URIs und Literalen. 7
2.2 Beispiel für eine mit RDFS ausgedrückte Regel. 8
2.3 Darstellung eines Quads im N-Quads-Format. 8
2.4 Abfrage zum Ermitteln italienischer Künstler. 10
2.5 Abfrage zum Ermitteln italienischer Künstler, die im 15. Jahrhundert geboren wurden oder gestorben sind. 11
2.6 Abfrage zum Ermitteln italienischer und deutscher Künstler. 11
2.7 Bilden des abgefragten Default Graph mit `FROM`. 12
2.8 Abfrage eines Graphen mit `FROM NAMED` und `GRAPH`-Schlüsselwort. 12
2.9 Änderung eines Variablentyps und Prüfung von Objekteigenschaften zur Laufzeit in PHP. 13
5.1 Abfragen von semantischen Daten aus einer Oracle Datenbank mittels SPARQL. 48
5.2 Schreibweisen für Literal-Strings in SPARQL-Abfragen. . . . 52
5.3 In SQL eingebettete SPARQL-Abfrage, die eine Variable mit unterschiedlicher Groß-/Kleinschreibung verwendet. 53
5.4 Abfrage aus Listing 5.3 mit Korrektur der Schreibweise der Variable via Alias. 54
5.5 SPARQL-Abfrage mit SQL-Schlüsselwort als Variablenbezeichnung. 54
5.6 In SQL eingebettete SPARQL-Abfrage aus Listing 5.5. Umgeschrieben, um Konflikte mit Schlüsselwörtern auszuschließen. 55
5.7 Einfügeoperation für zwei Tripel, zusammengefasst zu einer SQL-Abfrage. 60
5.8 Einfügeoperation für zwei Tripel, zu einem INSERT-Befehl zusammengefasst. 60
5.9 Stored Procedure zum Einfügen von Tripeln unter Verwendung des Bulk Loading Features. 60
5.10 Komplexe SPARQL-Abfrage mit vielen FILTER-Ausdrücken. 63
5.11 SPARQL Update-Abfrage, die von Stardog nicht korrekt ausgeführt wird. 68
5.12 Informationen aus Graph 5.11 in Form von Tripeln. 72

Anhang A

Daten der DBpedia-Benchmarks

Als Basis der Benchmarks dient eine Datenmenge von ca. 15 Millionen Tripeln aus dem DBpedia-Bestand von 2010, von der jeweils ein Teil geladen wird. Die gesamte Datenmenge entspricht etwa 10% der Größe von DBpedia in 2010.

Listing A.1: Vorlagen für die in den Benchmarks verwendeten Abfragen.

```
# 1
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?var1
FROM <http://dbpedia.org>
WHERE { %%var%% rdf:type ?var1 . }

# 2
PREFIX dbpprop: <http://dbpedia.org/property/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT *
FROM <http://dbpedia.org>
WHERE { %%var%% ?var2 ?var1. filter(?var2 = dbpedia2:caption || ?var2 =
    ↪ dbpprop:caption) }

# 3
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX space: <http://purl.org/net/schemas/space/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX dbpedia-prop: <http://dbpedia.org/property/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

```
SELECT ?var4 ?var8 ?var10
FROM <http://dbpedia.org>
WHERE {
  ?var5 dbpedia-owl:thumbnail ?var4 .
  ?var5 rdf:type dbpedia-owl:Person .
  ?var5 rdfs:label %%var%% .
  ?var5 foaf:page ?var8 .
  OPTIONAL { ?var5 foaf:homepage ?var10 .} .
}

# 4
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT ?var5 ?var6 ?var9 ?var8 ?var4
FROM <http://dbpedia.org>
WHERE {
  { %%var%% ?var5 ?var6 . ?var6 foaf:name ?var8 . }
  UNION
  { ?var9 ?var5 %%var%% ; foaf:name ?var4 . }
}

# 5
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbpp: <http://dbpedia.org/property/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?var3 ?var4 ?var5
FROM <http://dbpedia.org>
WHERE {
  { ?var3 dbpp:series %%var1%% ; foaf:name ?var4 ; rdfs:comment ?var5 ; rdf:type
    ↪ %%var0%% . }
  UNION
  { ?var3 dbpp:series ?var8 . ?var8 dbpp:redirect %%var1%% . ?var3 foaf:name ?var4 ;
    ↪ rdfs:comment ?var5 ; rdf:type %%var0%% . }
}

# 6
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT DISTINCT ?var3 ?var5 ?var7
FROM <http://dbpedia.org>
WHERE {
  ?var3 rdf:type <http://dbpedia.org/class/yago/Company108058098> .
  ?var3 dbpedia2:numEmployees ?var5
  FILTER ( xsd:integer(?var5) >= %%var%% ) .
```

```
  ?var3 foaf:homepage ?var7 .
}

# 7
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX umbelBus: <http://umbel.org/umbel/sc/Business>
PREFIX umbelCountry: <http://umbel.org/umbel/sc/IndependentCountry>
SELECT distinct ?var0 ?var1 ?var2 ?var3 ?var5 ?var6 ?var7 ?var10
FROM <http://dbpedia.org>
WHERE {
  ?var0 rdfs:comment ?var1.
  ?var0 foaf:page %%var%%
  OPTIONAL{
    ?var0 skos:subject ?var6
  }
  OPTIONAL{
    ?var0 dbpedia2:industry ?var5
  }
  OPTIONAL{
    ?var0 dbpedia2:location ?var2
  }
  OPTIONAL{
    ?var0 dbpedia2:locationCountry ?var3
  }
  OPTIONAL{
    ?var0 dbpedia2:locationCity ?var9; dbpedia2:manufacturer ?var0
  }
  OPTIONAL{
    ?var0 dbpedia2:products ?var11; dbpedia2:model ?var0
  }
  OPTIONAL{
    ?var0 <http://www.georss.org/georss/point> ?var10
  }
  OPTIONAL{
    ?var0 rdf:type ?var7
  }
}

# 8
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT ?var2 ?var4
FROM <http://dbpedia.org>
WHERE {
  {
```

```
    ?var2 rdf:type %%var1%%.
    ?var2 dbpedia2:population ?var4.
    FILTER (xsd:integer(?var4) > %%var0%%)
  } UNION {
    ?var2 rdf:type %%var1%%.
    ?var2 dbpedia2:populationUrban ?var4.
    FILTER (xsd:integer(?var4) > %%var0%%)
  }
}

# 9
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT *
FROM <http://dbpedia.org>
WHERE {
  ?var2 a <http://dbpedia.org/ontology/Settlement>; rdfs:label %%var%% .
  ?var6 a <http://dbpedia.org/ontology/Airport>.
  {
    ?var6 <http://dbpedia.org/ontology/city> ?var2
  } UNION {
    ?var6 <http://dbpedia.org/ontology/location> ?var2
  }
  {
    ?var6 <http://dbpedia.org/property/iata> ?var5.
  } UNION {
    ?var6 <http://dbpedia.org/ontology/iataLocationIdentifier> ?var5.
  } OPTIONAL {
    ?var6 foaf:homepage ?var6_home.
  } OPTIONAL {
    ?var6 <http://dbpedia.org/property/nativename> ?var6_name.
  }
}

# 10
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT DISTINCT ?var7
FROM <http://dbpedia.org>
WHERE {
  ?var3 foaf:page ?var7.
  ?var3 rdf:type <http://dbpedia.org/ontology/SoccerPlayer> .
  ?var3 dbpedia2:position ?var16 .
  ?var3 <http://dbpedia.org/property/clubs> ?var8 .
  ?var8 <http://dbpedia.org/ontology/capacity> ?var9 .
  ?var3 <http://dbpedia.org/ontology/birthPlace> ?var31 .
  ?var31 ?var33 ?var34.
  OPTIONAL {
    ?var3 <http://dbpedia.org/ontology/number> ?var35.
  }
  Filter (
    ?var33 = <http://dbpedia.org/property/populationEstimate> ||
    ?var33 = <http://dbpedia.org/property/populationCensus> ||
    ?var33 = <http://dbpedia.org/property/statPop>
```

```
    )
    Filter (xsd:integer(?var34) > %%var1%% ) .
    Filter (xsd:integer(?var9) < %%var0%% ) .
    Filter (
      ?var16 = 'Goalkeeper'@en ||
      ?var16 = <http://dbpedia.org/resource/Goalkeeper_%28association_football%29> ||
      ?var16 = <http://dbpedia.org/resource/Goalkeeper_%28football%29>
    )
}

# 11
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX umbelBus: <http://umbel.org/umbel/sc/Business>
PREFIX umbelCountry: <http://umbel.org/umbel/sc/IndependentCountry>
SELECT distinct ?var3 ?var4 ?var2
FROM <http://dbpedia.org>
 WHERE {
  {
    %%var%% dbpedia2:subsid ?var3
    OPTIONAL{?var2 %%var%% dbpedia2:parent}
    OPTIONAL{%%var%% dbpedia2:divisions ?var4}
  } UNION {
    ?var2 %%var%% dbpedia2:parent
    OPTIONAL{%%var%% dbpedia2:subsid ?var3}
    OPTIONAL{%%var%% dbpedia2:divisions ?var4}
  } UNION {
    %%var%% dbpedia2:divisions ?var4
    OPTIONAL{%%var%% dbpedia2:subsid ?var3}
    OPTIONAL{?var2 %%var%% dbpedia2:parent}
  }
}

# 12
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX space: <http://purl.org/net/schemas/space/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX dbpedia-prop: <http://dbpedia.org/property/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?var5
FROM <http://dbpedia.org>
WHERE {
  ?var2 rdf:type dbpedia-owl:Person .
  ?var2 dbpedia-owl:nationality ?var4 .
  ?var4 rdfs:label ?var5 .
  ?var2 rdfs:label %%var%% .
  FILTER (lang(?var5) = 'en')
}

# 13
PREFIX owl: <http://www.w3.org/2002/07/owl#>
```

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT *
FROM <http://dbpedia.org>
WHERE {
  {
    %%var%% rdfs:comment ?var0.
    FILTER (lang(?var0) = 'en')
  } UNION {
    %%var%% foaf:depiction ?var1
  } UNION {
    %%var%% foaf:homepage ?var2
  }
}

# 14
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT ?var6 ?var8 ?var10 ?var4
FROM <http://dbpedia.org>
WHERE {
  ?var4 skos:subject %%var%% .
  ?var4 foaf:name ?var6 .
  OPTIONAL {
    ?var4 rdfs:comment ?var8 .
    FILTER (LANG(?var8) = 'en') .
  } OPTIONAL {
    ?var4 rdfs:comment ?var10 .
    FILTER (LANG(?var10) = 'de') .
  }
}

# 15
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?var2 ?var3
FROM <http://dbpedia.org>
WHERE {
  ?var2 rdf:type %%var%% ; rdfs:label ?var3 .
  FILTER(regex(str(?var3), 'pes', 'i'))
}

# 16
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?var7 ?var4 ?var6 ?var5
FROM <http://dbpedia.org>
```

```
WHERE {
  %%var%% ?var4 ?var5 .
  OPTIONAL {?var5 rdfs:label ?var6} .
  FILTER(
    langMatches(lang(?var6),'EN') ||
    (! langMatches(lang(?var6),'*'))
  ) .
  FILTER(
    langMatches(lang(?var5),'EN') ||
    (! langMatches(lang(?var5),'*'))
  ) .
  OPTIONAL {?var4 rdfs:label ?var7}
}

# 17
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX event: <http://purl.org/NET/c4dm/event.owl#>
PREFIX rel: <http://purl.org/vocab/relationship/>
PREFIX lingvoj: <http://www.lingvoj.org/ontology#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbpprop: <http://dbpedia.org/property/>
PREFIX dbowl: <http://dbpedia.org/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX tags: <http://www.holygoat.co.uk/owl/redwood/0.1/tags/>
PREFIX db: <http://dbtune.org/musicbrainz/resource/>
PREFIX geo: <http://www.geonames.org/ontology#>
PREFIX bio: <http://purl.org/vocab/bio/0.1/>
PREFIX mo: <http://purl.org/ontology/mo/>
PREFIX vocab: <http://dbtune.org/musicbrainz/resource/vocab/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX mbz: <http://purl.org/ontology/mbz#>
SELECT DISTINCT ?var2 ?var3
FROM <http://dbpedia.org>
WHERE {
  {
    ?var2 <http://www.w3.org/2004/02/skos/core#subject> %%var%%.
  } UNION {
    ?var2 <http://www.w3.org/2004/02/skos/core#subject>
      ↪ <http://dbpedia.org/resource/Category:Prefectures_in_France>.
  } UNION {
    ?var2 <http://www.w3.org/2004/02/skos/core#subject>
      ↪ <http://dbpedia.org/resource/Category:German_state_capitals>.
  }
  ?var2 <http://www.w3.org/2000/01/rdf-schema#label> ?var3.
  FILTER (lang(?var3)='fr')
}

# 18
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT ?var3 ?var4 ?var5
```

```
FROM <http://dbpedia.org>
WHERE {
  {
    %%var%% ?var3 ?var4.
    FILTER (
      (STR(?var3) = 'http://www.w3.org/2000/01/rdf-schema#label' && lang(?var4) =
      ↪ 'en') ||
      (STR(?var3) = 'http://dbpedia.org/ontology/abstract' && lang(?var4) = 'en') ||
      (STR(?var3) = 'http://www.w3.org/2000/01/rdf-schema#comment' &&
      ↪ lang(?var4) = 'en') ||
      (
        STR(?var3) != 'http://dbpedia.org/ontology/abstract' &&
        STR(?var3) != 'http://www.w3.org/2000/01/rdf-schema#comment' &&
        STR(?var3) != 'http://www.w3.org/2000/01/rdf-schema#label'
      )
    )
  } UNION {
    ?var5 ?var3 %%var%%
    FILTER ( STR(?var3) = 'http://dbpedia.org/ontology/owner' || STR(?var3) =
     ↪ 'http://dbpedia.org/property/redirect' )
  }
}

# 19
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?var1
FROM <http://dbpedia.org>
WHERE {
  {
    ?var1 <http://www.w3.org/2000/01/rdf-schema#label> %%var%%
  } UNION {
    ?var1 <http://www.w3.org/2000/01/rdf-schema#label> %%var%%
  }.
  FILTER(
    regex(str(?var1),'http://dbpedia.org/resource/') ||
    regex(str(?var1),'http://dbpedia.org/ontology/') ||
    regex(str(?var1),'http://www.w3.org/2002/07/owl') ||
    regex(str(?var1),'http://www.w3.org/2001/XMLSchema') ||
    regex(str(?var1),'http://www.w3.org/2000/01/rdf-schema') ||
    regex(str(?var1),'http://www.w3.org/1999/02/22-rdf-syntax-ns')
  )
}

# 20
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT *
FROM <http://dbpedia.org>
WHERE {
  ?var6 a <http://dbpedia.org/ontology/PopulatedPlace>;
    ↪ <http://dbpedia.org/ontology/abstract> ?var1; rdfs:label ?var2; geo:lat ?var3;
    ↪ geo:long ?var4.
  {
    ?var6 rdfs:label %%var%%.
  } UNION {
    ?var5 <http://dbpedia.org/property/redirect> ?var6; rdfs:label %%var%%.
  } OPTIONAL {
    ?var6 foaf:depiction ?var8
  } OPTIONAL {
    ?var6 foaf:homepage ?var10
  } OPTIONAL {
```

```
    ?var6 <http://dbpedia.org/ontology/populationTotal> ?var12
  } OPTIONAL {
    ?var6 <http://dbpedia.org/ontology/thumbnail> ?var14
  } FILTER (langMatches( lang(?var1), 'de') && langMatches( lang(?var2), 'de') )
}

# 21
PREFIX dbpprop: <http://dbpedia.org/property/>
SELECT *
FROM <http://dbpedia.org>
WHERE { %%var%%  dbpprop:caption ?var0 . }

# 22
SELECT ?var2
FROM <http://dbpedia.org>
WHERE {
  ?var3 <http://xmlns.com/foaf/0.1/homepage> ?var2 .
  ?var3 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> %%var%% .
}

# 23
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX space: <http://purl.org/net/schemas/space/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX dbpedia-prop: <http://dbpedia.org/property/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?var4
FROM <http://dbpedia.org>
WHERE {
  ?var2 rdf:type dbpedia-owl:Person .
  ?var2 rdfs:label  %%var%% .
  ?var2  foaf:page ?var4 .
}

# 24
SELECT *
FROM <http://dbpedia.org>
WHERE {
  ?var1 a <http://dbpedia.org/ontology/Organisation> .
  ?var2 <http://dbpedia.org/ontology/foundationPlace> %%var0%% .
  ?var4 <http://dbpedia.org/ontology/developer> ?var2 .
  ?var4 a %%var1%% .
} LIMIT 10000

# 25
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpprop:<http://dbpedia.org/property/>
SELECT ?var0 ?var1 ?var2 ?var3
FROM <http://dbpedia.org>
WHERE {
  ?var6 rdf:type %%var%%.
  ?var6 dbpprop:name ?var0.
  ?var6 dbpprop:pages ?var1.
  ?var6 dbpprop:isbn ?var2.
  ?var6 dbpprop:author ?var3.
}
```

Tabelle A.1: Performanz einzelner Abfrage-Typen in Queries per Second (QpS).

		Datenmenge		
Nummer	*Features*	*1%*	*5%*	*25%*
Virtuoso				
1	DISTINCT	962,47	796,77	613,09
2	FILTER	522,46	624,57	359,30
3	OPTIONAL	83,23	316,74	128,63
4	UNION	450,29	279,94	40,21
5	UNION, DISTINCT	184,19	186,94	112,44
6	FILTER, DISTINCT	162,03	429,34	272,69
7	OPTIONAL, DISTINCT	87,49	125,68	52,02
8	UNION, FILTER	512,99	488,97	46,86
9	UNION, OPTIONAL	38,27	38,91	29,15
10	OPTIONAL, FILTER, DISTINCT	36,33	21,00	18,99
11	UNION, OPTIONAL, DISTINCT	304,16	293,76	275,01
12	FILTER, LANG, DISTINCT	465,11	464,23	151,35
13	UNION, FILTER, LANG	158,48	475,71	443,24
14	OPTIONAL, FILTER, LANG	455,93	328,17	55,23
15	FILTER, REGEX, DISTINCT, STR	691,15	9,75	27,90
16	OPTIONAL, FILTER, LANG, DISTINCT	458,38	166,71	327,74
17	UNION, FILTER, LANG, DISTINCT	322,36	148,36	140,66
18	UNION, FILTER, LANG, STR	439,74	259,25	278,92
19	UNION, FILTER, REGEX, STR	445,27	127,67	155,44
20	UNION, OPTIONAL, FILTER, LANG	4,59	4,92	5,08
21	1 Triple Pattern	946,66	737,42	832,59
22	2 Triple Patterns	173,92	440,50	55,98
23	3 Triple Patterns	591,01	541,39	215,83
24	4 Triple Patterns	429,91	416,12	3,87
25	5 Triple Patterns	207,94	204,28	33,05
MySQL				

Tabelle A.1: Performanz einzelner Abfrage-Typen in Queries per Second (QpS) (Fortsetzung).

1	DISTINCT	116,76	117,21	-
2	FILTER	43,54	42,83	-
3	OPTIONAL	40,78	39,75	-
4	UNION	1,06	1,40	-
5	UNION, DISTINCT	35,95	35,00	-
6	FILTER, DISTINCT	35,37	17,87	-
7	OPTIONAL, DISTINCT	15,62	13,76	-
8	UNION, FILTER	3,50	0,13	-
9	UNION, OPTIONAL	22,48	21,77	-
10	OPTIONAL, FILTER, DISTINCT	14,16	0,02	-
11	UNION, OPTIONAL, DISTINCT	16,54	16,71	-
12	FILTER, LANG, DISTINCT	42,40	41,38	-
13	UNION, FILTER, LANG	36,48	35,98	-
14	OPTIONAL, FILTER, LANG	35,39	34,86	-
15	FILTER, REGEX, DISTINCT, STR	63,68	5,90	-
16	OPTIONAL, FILTER, LANG, DISTINCT	48,35	47,23	-
17	UNION, FILTER, LANG, DISTINCT	19,59	19,32	-
18	UNION, FILTER, LANG, STR	26,78	26,09	-
19	UNION, FILTER, REGEX, STR	38,66	38,03	-
20	UNION, OPTIONAL, FILTER, LANG	10,39	10,32	-
21	1 Triple Pattern	132,16	128,28	-
22	2 Triple Patterns	0,53	1,03	-
23	3 Triple Patterns	49,52	49,15	-
24	4 Triple Patterns	74,67	73,76	-
25	5 Triple Patterns	56,98	55,90	-
Oracle				
1	DISTINCT	12,02	10,21	-
2	FILTER	8,14	7,25	-
3	OPTIONAL	34,40	33,78	-

Tabelle A.1: Performanz einzelner Abfrage-Typen in Queries per Second (QpS) (Fortsetzung).

4	UNION	3,95	2,58	-
5	UNION, DISTINCT	36,09	35,95	-
6	FILTER, DISTINCT	31,06	30,06	-
7	OPTIONAL, DISTINCT	9,63	2,74	-
8	UNION, FILTER	28,04	13,16	-
9	UNION, OPTIONAL	33,28	33,33	-
10	OPTIONAL, FILTER, DISTINCT	18,36	18,50	-
11	UNION, OPTIONAL, DISTINCT	17,04	15,09	-
12	FILTER, LANG, DISTINCT	35,44	35,53	-
13	UNION, FILTER, LANG	5,15	5,06	-
14	OPTIONAL, FILTER, LANG	0,02	0,04	-
15	FILTER, REGEX, DISTINCT, STR	60,95	17,65	-
16	OPTIONAL, FILTER, LANG, DISTINCT	47,15	47,16	-
17	UNION, FILTER, LANG, DISTINCT	12,82	3,85	-
18	UNION, FILTER, LANG, STR	3,33	2,29	-
19	UNION, FILTER, REGEX, STR	7,50	6,70	-
20	UNION, OPTIONAL, FILTER, LANG	24,80	24,49	-
21	1 Triple Pattern	12,55	10,86	-
22	2 Triple Patterns	70,28	10,44	-
23	3 Triple Patterns	41,29	27,82	-
24	4 Triple Patterns	73,66	71,78	-
25	5 Triple Patterns	54,82	54,72	-
Stardog				
1	DISTINCT	59,51	74,59	60,54
2	FILTER	65,28	69,68	60,43
3	OPTIONAL	66,22	66,86	58,85
4	UNION	57,30	61,78	25,95
5	UNION, DISTINCT	60,51	74,43	60,18

Tabelle A.1: Performanz einzelner Abfrage-Typen in Queries per Second (QpS) (Fortsetzung).

6	FILTER, DISTINCT	67,08	64,60	59,99
7	OPTIONAL, DISTINCT	48,60	35,95	15,64
8	UNION, FILTER	74,11	64,64	32,61
9	UNION, OPTIONAL	60,68	71,69	60,07
10	OPTIONAL, FILTER, DISTINCT	18,87	3,82	0,33
11	UNION, OPTIONAL, DISTINCT	59,32	67,03	59,09
12	FILTER, LANG, DISTINCT	64,57	49,31	17,05
13	UNION, FILTER, LANG	71,13	67,76	60,17
14	OPTIONAL, FILTER, LANG	65,45	62,89	18,91
15	FILTER, REGEX, DISTINCT, STR	57,75	43,84	19,37
16	OPTIONAL, FILTER, LANG, DISTINCT	52,38	41,26	12,15
17	UNION, FILTER, LANG, DISTINCT	56,33	57,29	22,10
18	UNION, FILTER, LANG, STR	59,02	83,13	53,66
19	UNION, FILTER, REGEX, STR	64,25	73,76	60,25
20	UNION, OPTIONAL, FILTER, LANG	71,05	72,75	40,45
21	1 Triple Pattern	76,23	74,79	62,39
22	2 Triple Patterns	80,87	75,92	55,05
23	3 Triple Patterns	79,03	71,72	59,93
24	4 Triple Patterns	71,72	67,10	1,64
25	5 Triple Patterns	66,09	74,53	60,31
Neo4j				
1	DISTINCT	157,69	168,68	72,20
2	FILTER	142,71	131,33	93,61
3	OPTIONAL	1,28	0,36	0,0028
4	UNION	114,63	110,78	30,94
5	UNION, DISTINCT	1,63	0,69	0,0034
6	FILTER, DISTINCT	133,40	74,68	65,94
7	OPTIONAL, DISTINCT	0,61	0,11	0,0031

Tabelle A.1: Performanz einzelner Abfrage-Typen in Queries per Second (QpS) (Fortsetzung).

8	UNION, FILTER	13,52	0,92	0,71
9	UNION, OPTIONAL	47,82	6,57	0,34
10	OPTIONAL, FILTER, DISTINCT	0,80	0,17	0,0026
11	UNION, OPTIONAL, DISTINCT	133,17	132,41	67,37
12	FILTER, LANG, DISTINCT	11,60	1,43	0,13
13	UNION, FILTER, LANG	144,79	110,88	97,37
14	OPTIONAL, FILTER, LANG	133,91	29,94	14,71
15	FILTER, REGEX, DISTINCT, STR	120,74	2,06	0,48
16	OPTIONAL, FILTER, LANG, DISTINCT	156,09	160,92	124,32
17	UNION, FILTER, LANG, DISTINCT	114,90	51,75	19,42
18	UNION, FILTER, LANG, STR	140,09	114,91	27,96
19	UNION, FILTER, REGEX, STR	140,17	101,41	83,46
20	UNION, OPTIONAL, FILTER, LANG	41,38	6,03	0,29
21	1 Triple Pattern	166,21	129,09	142,34
22	2 Triple Patterns	1,54	0,45	0,0041
23	3 Triple Patterns	7,46	0,46	0,062
24	4 Triple Patterns	57,93	22,14	159,43
25	5 Triple Patterns	166,96	160,45	0,22

Tabelle A.2: Anzahl von Abfrage-Kombinationen (Query Mixes) pro Stunde (QMpH).

	Datenmenge		
Connector	*1%*	*5%*	*25%*
Virtuoso	10201,93	7606,80	4297,39
MySQL	1137,31	477,12	-
Oracle	76,75	111,46	-
Stardog	8461,08	5482,76	816,52
Neo4j	658,51	120,92	2,20

Tabelle A.3: Geometrisches Mittel der QpS aller Abfragen.

	Datenmenge		
Connector	*1%*	*5%*	*25%*
Virtuoso	238,94	196,73	93,27
MySQL	24,27	14,78	-
Oracle	15,43	11,38	-
Stardog	61,20	57,23	28,87
Neo4j	34,18	11,55	1,82

Anhang B

Inhalt der DVDs

Die DVDs enthalten den während der Arbeit erstellten oder modifizierten Quellcode, die verwendeten Benchmark-Daten sowie ausgewählte Werkzeuge.

Quellcode

DVD 1: /Code

Erfurt/	Erfurt-Framework mit generalisierter Persistenzschicht.
OntoWiki/	Zum modifizierten Erfurt-Framework kompatibles OntoWiki.
Athletic/	Modifiziertes Athletic-Framework, das zur Durchführung der Benchmarks verwendet wurde.

Testdaten für den DBpedia-Benchmark

Die für den DBpedia-Benchmark verwendeten Testdaten wurden mit 7-Zip[1] komprimiert und liegen im N-Triples-Format vor.

DVD 1: /Benchmark

benchmark_data.zip . .	Die verwendeten Testdaten.

Virtuelle Maschine mit Oracle Datenbank

Die virtuelle Maschine mit der verwendeten Oracle 12c-Installation wurde mittels 7-Zip in 4 Dateien aufgeteilt und archiviert.

[1] http://www.7-zip.org/

Nach der Extraktion kann die Maschine als „Appliance“ in VirtualBox[2] importiert werden. Die verwendeten Nutzernamen und Passwörter finden sich in der Beschreibung der Maschine.

DVD 1: /Oracle_VM

Oracle_12c.zip.004 . . Teil 4

DVD 2: /Oracle_VM

Oracle_12c.zip.001 . . Teil 1

DVD 3: /Oracle_VM

Oracle_12c.zip.002 . . Teil 2

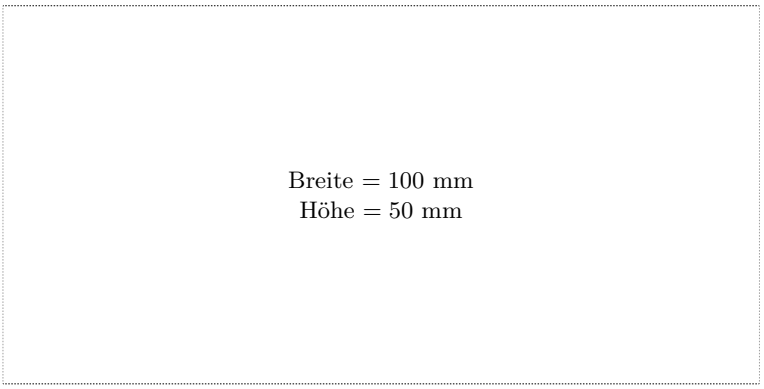
DVD 4: /Oracle_VM

Oracle_12c.zip.003 . . Teil 3

[2] https://www.virtualbox.org/

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —