

# Leistungsanalyse und -optimierung semantischer Abfragen

## BACHELORARBEIT

KIT – KARLSRUHER INSTITUT FÜR TECHNOLOGIE  
FRAUNHOFER IOSB – FRAUNHOFER-INSTITUT FÜR OPTRONIK,  
SYSTEMTECHNIK UND BILDAUSWERTUNG

**Erik Kristiansen**

20. Dezember 2019

Verantwortlicher Betreuer:	Prof. Dr.-Ing. Jürgen Beyerer
Betreuende Mitarbeiter:	Philipp Hertweck, M.Sc. Tobias Hellmund, M.Sc.



## Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht habe und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der gültigen Fassung beachtet habe.

Karlsruhe, den 20. Dezember 2019

---

(Erik Kristiansen)



# Inhaltsverzeichnis

<b>Einleitung</b>	<b>1</b>
<b>1 Stand der Technik</b>	<b>3</b>
1.1 Semantic Web . . . . .	3
1.1.1 Repräsentation mit RDF, OWL . . . . .	3
1.1.2 Persistenz mit Hilfe eines Triple-Stores . . . . .	6
1.1.3 Ontologie-Zugriff mittels SPARQL . . . . .	6
1.1.4 SPARQL Performance-Tipps . . . . .	6
1.2 Performance Evaluierung durch Benchmarks . . . . .	7
1.2.1 Was ist ein Benchmark . . . . .	7
1.2.2 Benchmark für Performance Evaluierung . . . . .	8
1.3 Implementierungen von Triple-Stores . . . . .	8
1.3.1 Aufbau . . . . .	8
1.3.2 Triple-Store Implementierungen . . . . .	11
1.3.3 Speicher . . . . .	13
<b>2 Problemstellung</b>	<b>15</b>
2.1 Benchmarks . . . . .	16
2.2 Queries . . . . .	16
2.3 Evaluierung . . . . .	16
<b>3 Benchmark für semantische Anfragen</b>	<b>19</b>
3.1 Aufbau eines Benchmarks . . . . .	19
3.2 Kategorisierungsschema für Benchmarks . . . . .	20
3.3 Benchmarks zur Evaluierung von Triple-Stores . . . . .	21
3.4 Berlin SPARQL Benchmark . . . . .	24
3.4.1 Verwendete Metriken . . . . .	25
3.4.2 Ontologie Schema . . . . .	25

<b>4</b>	<b>Optimierungsmöglichkeiten semantischer Anfragen</b>	<b>27</b>
4.1	Elemente einer SPARQL-Query . . . . .	27
4.2	Vergleich von Query-Designs . . . . .	34
4.3	Nach Optimiererausgabe . . . . .	36
<b>5</b>	<b>Aufbau der Evaluation</b>	<b>39</b>
5.1	Beteiligte Komponenten . . . . .	39
5.1.1	Bereits existente Komponenten . . . . .	39
5.2	Integration des Berlin Benchmarks . . . . .	41
5.3	Weiterentwicklung des Berlin Benchmarks . . . . .	41
5.4	Systemsetup . . . . .	42
5.4.1	Heap-Space . . . . .	42
5.5	Ausführungsmethodik . . . . .	42
<b>6</b>	<b>Ausführung</b>	<b>45</b>
<b>7</b>	<b>Evaluation</b>	<b>49</b>
7.1	Präsentation der Messergebnisse . . . . .	49
7.2	Diskussion der Messergebnisse . . . . .	52
7.2.1	Einfluss der Query Elemente . . . . .	52
7.2.2	Liste optimierter Queries . . . . .	53
7.2.3	Verteilung der Daten . . . . .	54
7.2.4	Cacheverhalten . . . . .	55
7.3	Empfehlungen zur Formulierung der Queries . . . . .	58
<b>8</b>	<b>Fazit</b>	<b>59</b>
8.1	Zusammenfassung . . . . .	59
8.2	Ausblick . . . . .	60
<b>Anhang</b>		<b>61</b>
1	Messergebnisse . . . . .	61
2	Queries . . . . .	65
3	Optimiererausgabe . . . . .	71
<b>Literatur</b>		<b>95</b>
<b>Abbildungsverzeichnis</b>		<b>99</b>

Listings	101
----------	-----

Glossary	105
----------	-----





# Einleitung

Der Einsatz von semantischen Technologien findet in der heutigen Zeit immer mehr Verwendung. Neben der Gesundheitsbranche oder „Smart Cities“, finden semantische Technologien verstärkt Einsatz in kritischen Infrastrukturen [Kon+][KK12][Hel+]. Durch den verstärkten Einsatz rücken die Antwortzeiten von Abfragen immer mehr in den Vordergrund. So spielen in Echtzeitsystemen, also Systeme zur unmittelbaren Abwicklung von Prozessen, oder bei anderen zeitkritischen Anwendungen die Antwortzeiten der Abfragen eine sehr wichtige Rolle. Um Abfragezeiten so gering wie möglich zu halten, werden oftmals die Systeme intern optimiert. Dies erfordert allerdings ein gutes Verständnis über einen Ausschnitt der Semantic-Web Technologien und die verwendeten Daten und ist deshalb oftmals nur schwer anwendbar. In dieser Arbeit wurde die Optimierung der Antwortzeiten über die Abfrageebene betrachtet. Die Standardsprache für das Abfragen von Daten ist SPARQL und wurde deshalb in dieser Arbeit analysiert. Der Vorteil der Optimierung der Abfrageebene ist, dass dies ohne tiefes technisches Wissen oder Wissen über die Daten möglich ist. Um dieses Problem zu betrachten wurden die Arbeit in zwei Teile geteilt:

Im ersten Teil wurde untersucht, was Performance bedeutet und wie man diese systematisch ermitteln kann. Vorhandene Benchmarks sind evaluiert und ein passender ausgewählt und angepasst worden.

Im zweiten Teil wurden SPARQL-Abfragen formuliert, ausgeführt und deren Laufzeiten gemessen. Anhand dieser Messungen wurden die Ausführungszeiten, sowie die Formulierungen verglichen.

Mithilfe des Benchmarks wurden dann die Auswirkungen mithilfe des Benchmarks überprüft und die Ergebnisse in einer Tabelle gespeichert. Mit diesen Ergebnissen wurden Graphen über die Optimierung der einzelnen Möglichkeiten gebildet. Anhand dieser konnte dann in der Evaluierung eine Liste an Tipps für das Schreiben von semantischen Abfragen erstellt werden. Die Ergebnisse und verwendeten Skripte wurden in Github hochgeladen und können unter <https://github.com/DrKingSchultz2/LAOSA> gefunden werden.



# 1 Stand der Technik

## 1.1 Semantic Web

Das „Semantic Web“, auf Deutsch „semantisches Web“, ist als eine Erweiterung des existierenden World Wide Web's gedacht. Als Ergebnis dieser Erweiterung soll das World Wide Web von einer großen Ansammlung an Hyperlinks zu einer großen Ansammlung an Wissen gemacht werden, welche von Maschinen durchlaufen und interpretiert werden können. Hierdurch sollen Suchanfragen, Terminplanungen, Maschine-Maschine-Interaktionen, etc. leichter gemacht werden. Das Paper von Tim Berners-Lee, James Hendler and Ora Lassila beschreibt einen solchen Anwendungsfall [BHL+01]. Damit so ein „Semantic Web“ funktionieren kann, muss es Computern möglich sein, Informationen als strukturierte Sets zu erhalten und über diese dann mit Interferenzregeln ein automatisches Reasoning auszuführen. Um ebendiese Struktur zu erhalten, werden verschiedene Technologien verwendet. Die Abbildung 1.1 zeigt die Struktur des Semantic Web Stacks. In der Graphik wird dargestellt, wie die verschiedenen Komponenten eines Semantic-Web's aufeinander aufbauen. Zwei dieser Technologien sind: eXtensible Markup Language (XML) und Resource Description Framework (RDF). Diese werden im Folgenden näher beschrieben.

### 1.1.1 Repräsentation mit RDF, OWL

XML ist eine standardisierte Auszeichnungssprache für Informationen, welche für Maschinen leicht lesbar ist. Hierbei besteht XML aus sogenannten Tags, welche die Informationen in Kategorien einteilt. Ein Tag ist ein Begriff, welcher die nachfolgenden Informationen einteilt und klarstellt, was diese Informationen bedeuten.

```
1 <Person>
2   <Name> Erik </Name>
3   <Age> 23 </Age>
4 </Person>
```

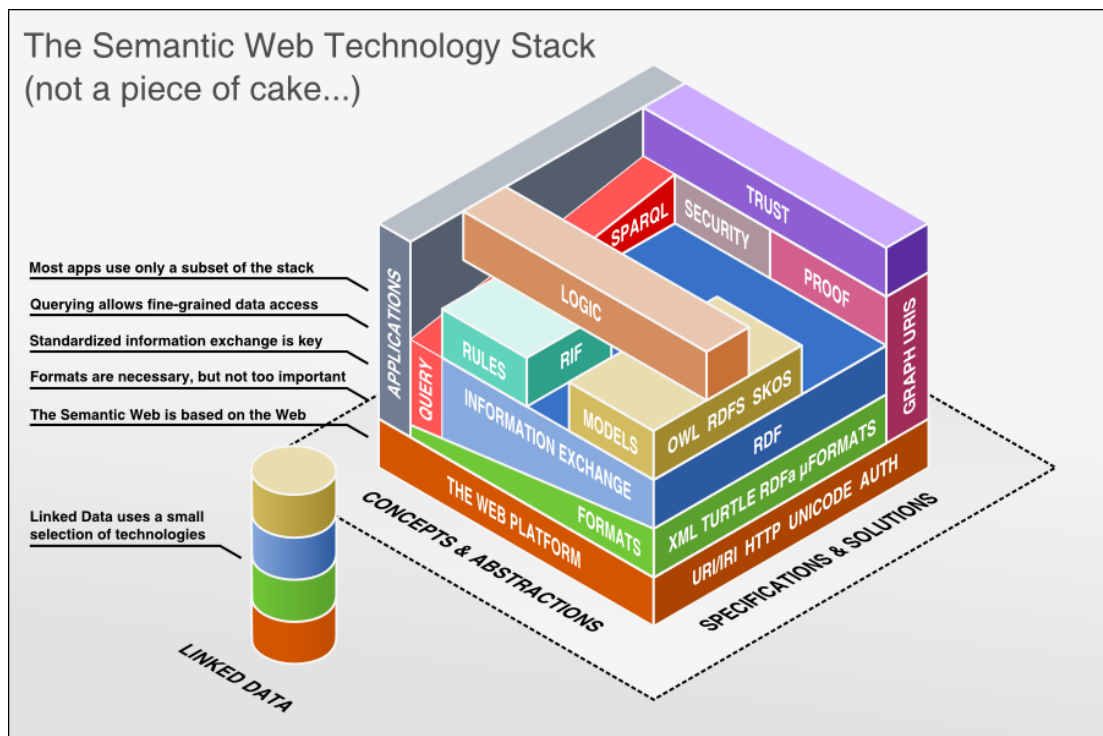


Abbildung 1.1: Semantic Web Stack [Gai12]

---

Listing 1.1: XML Beispiel

XML ermöglicht jedem Nutzer eigene Tags zu schreiben, um für Webseiten oder ähnliches eine Struktur aufzubauen. Diese Tags erlauben es dann Programmen oder Skripten, sinnvoll mit diesen Daten umzugehen [W3C19b]. Damit ein Programm oder Skript mit den Daten sinnvoll arbeiten kann, muss diesem vorher klar sein, welche Bedeutung die verschiedenen Tags haben. Diese Bedeutung ist aber sehr variabel, da jeder Nutzer eigene Tags erstellen kann. Dadurch lässt sich mithilfe von XML zwar eine gute Struktur für Dokumente und Informationen erreichen, allerdings ist diese für Maschinen nur sehr schwer auswertbar.

Dieses Problem der Bedeutung der Daten löst RDF. RDF kodiert Aussagen als Triple, welche aus einem Subjekt, einem Objekt und einem Prädikat bestehen. Durch diese Triple-Form kann man Objekten Merkmale mit bestimmten Werten geben. Um den Tripeln Bedeutung zu geben, wird jedem Teil eine URI (Universal Resource Identifier) zugeteilt. Dies ermöglicht es jedem Nutzer nur durch die Definition einer neuen URI überall im Web ein neues Konzept oder Verb zu definieren.

Da es möglich ist, dass es verschiedene URI's für dasselbe Konzept geben kann, werden Informationskollektionen verwendet, welche solche Probleme beheben können. Diese Informationskollektionen beschreiben dabei Beziehungen zwischen verschiedenen Konzepten. Diese Informationskollektionen werden „Ontologien“ genannt.

Eine Ontologie speichert Relationen und Informationen und ermöglicht mit einem Set an Inferenzregeln auch Reasoning über die Daten. Ein Reasoner ist ein Programm, welches über die Ontologie iteriert und dabei implizite Verbindungen zwischen Daten explizit darstellt. So stellt ein Reasoner beispielsweise fest, dass wenn ein Student ein Mensch ist, und Bernd ein Student ist, gleichzeitig Bernd auch ein Mensch ist.

Bernd → Student → Mensch

**Bernd → Mensch**

Eine Ontologie kann auf verschiedenen Detailleveln beschrieben werden. Das Datenset, welches in einer Ontologie gespeichert wird, kann verschiedene Formate haben, allerdings sind diese leicht ineinander zu übersetzen.

OWL (Web Ontology Language) ist beispielsweise eine Ontologie mit einem bestimmten Speicherformat. OWL wurde damals vom W3C entwickelt. Das W3C (World Wide Web Consortium) ist ein Gremium für die Standardisierung der Techniken im Internet (World Wide Web).

### 1.1.2 Persistenz mit Hilfe eines Triple-Stores

Triple-Stores ermöglichen es Daten, welche in Triple-Form dargestellt sind, zu speichern, über diese zu suchen oder auch einen Reasoner auszuführen. Ein Tripel besteht aus drei Teilen, einem Subjekt, einem Prädikat und einem Objekt. So ist beispielsweise der Satz „Paul liebt Sara“ ein Tripel, denn Paul ist das Subjekt, liebt ist das Prädikat und Sara ist das Objekt. Ein Triple-Store ermöglicht es ähnlich zu SQL-Anfragen, über die gespeicherten Daten zu suchen. Hierfür werden sogenannte Queries geschrieben, welche vom Triple-Store mithilfe von Query-Parsern bearbeitet werden. Ein Triple-Store besteht aus einem Query Parser, welcher eingehende Queries übersetzt, und aus einem Optimierer, welcher übersetzte Queries optimiert. Die optimierten Queries werden dann ausgeführt und die Daten werden aus dem Speicher entsprechend geholt. Eine Query ist hierbei eine Abfrage nach Informationen aus der Ontologie. Hier wird, ähnlich zu SQL-Abfragen, angegeben, welche Werte aus dem Triple-Store extrahiert werden sollen und welche Vorgaben diese erfüllen müssen. Der Query-Parser überprüft die Syntax der Anfrage und überführt sie in eine abstrakte Darstellung. Optimierungen können dann auf dieser abstrakten Darstellung durchgeführt werden.

### 1.1.3 Ontologie-Zugriff mittels SPARQL

Um auf die Daten in einer Ontologie zuzugreifen, benötigt man, wie bei anderen Datenbanken, eine bestimmte Abfrage-Sprache. Für Ontologien wurde SPARQL entwickelt, welche es ermöglicht, Triple Anfragen leicht verständlich darzustellen. Ein weiterer wichtiger Punkt bei SPARQL ist, dass sie mächtig genug ist, auch komplizierte Anfragen auszudrücken. SPARQL wurde, wie OWL, zusammen mit W3C (bzw. heutzutage hauptsächlich von W3C) entwickelt und folgt auch dessen Standards, nach welchen sich auch andere RDF-Query-Sprachen richten. SPARQL ist die momentan meist verwendete Sprache für Anfragen über RDF. 1.2 ist eine beispielhafte Abfrage von SPARQL an einem Triple-Store.

```
1 select * where {  
2   ?product rdf:type Product  
3 }
```

Listing 1.2: SPARQL-Abfrage

### 1.1.4 SPARQL Performance-Tipps

In dieser Bachelorarbeit wird nach Performancetipps für das Design von SPARQL-Queries gesucht. Im Internet findet man nur wenige Paper, welche bei dem Design einer Query helfen. Ein

Paper welches die Performance von SPARQL überprüft und Vorschläge für eine Effizienzsteigerung dieser macht ist [MUA10]. In diesem Paper wird vorgestellt, wieso SPARQL-Abfragen langsamer sind als SQL und es werden verschiedene Konzepte vorgestellt, wie man die Performance von einer Abfrage erhöhen könnte. Allerdings beruhen diese Tipps auf einer Änderung tieferliegender Komponenten, für welches viel technisches Verständnis benötigt wird. Dies ist für normale Nutzer, welche keinen tieferen Einblick oder Zugriff auf das System haben, nicht relevant. Andere Paper befassen sich zwar mit Design-Optimierungen, allerdings sind diese meist auf SQL-Optimierungen basiert und die Effektivität nicht evaluiert. [Ves13] (16.12.2019) ist ein Blog, welcher SPARQL-Query-Design-Optimierungen vorstellt. Im Folgenden die Liste dieser Optimierungen:

- Value Equality vs Term Equality  
Nach der Quelle ist Term Equality schneller, da dieser nur überprüft ob die Typen gleich sind, während Value Equality überprüft ob die Werte gleich sind.
- Using overly broad Filters or Filters which can be changed into triple patterns  
Die Filter sollen nicht auf einem großen Set an Daten laufen und wenn möglich ein Tripel anstelle eines Filters benutzen.
- Avoid Select \*  
Besser definieren, welche Daten zurückgegeben werden müssen, damit weniger Daten zurückgegeben werden und der Ausführer nicht benötigte Variablen löschen kann.
- Avoid Distinct  
Anstelle von Distinct Reduced verwenden, da Distinct sehr komplex ist.
- Use Limit  
Nur nach so vielen Ergebnissen fragen wie benötigt.

Diese Design-Optimierungen sind zwar für SPARQL relevant, allerdings wurden keine Tests durchgeführt, welche auch die Optimierung belegen und zeigen, wie groß der Performanceunterschied wirklich ist.

## 1.2 Performance Evaluierung durch Benchmarks

### 1.2.1 Was ist ein Benchmark

Um Anwendungen, aber auch Anfragen, hinsichtlich ihrer Geschwindigkeit bewerten zu können, bedarf es einem festgelegten Prozess, der zu reproduzierbaren Ergebnissen und vor allem

zu einer vergleichbaren Quantifizierung kommt. Um dies zu erreichen, wurden Benchmarks entwickelt. Die Definition eines Benchmarks ist der kontinuierliche Vergleich von Prozessen, um die Leistungslücke systematisch zu erschließen [Wüb19]. Benchmarking findet in vielen verschiedenen Gebieten Anwendung, wie zum Beispiel der Betriebswirtschaft, der Finanzwirtschaft oder der IT. Ziel eines Benchmarks ist es, möglichst objektiv verschiedene Systeme miteinander zu vergleichen zu können. Um dies zu erreichen, gibt es in jedem Benchmarking-Gebiet bestimmte Kriterien, über welche ein Benchmark die Systeme miteinander vergleicht.

### 1.2.2 Benchmark für Performance Evaluierung

Meist wird ein Benchmark für die objektive Analyse und den Vergleich von Leistungsmessungen verwendet. Eine Übersetzung von „Benchmark“ ist auch „Leistungsvergleichstest“. Für eine gute Messung der Leistung muss ein Test mehrmals mit verschiedenen Werten ausgeführt werden. Dies dient dem Herausrechnen von Fehlerquellen und statistischen Ausreißern. Um vergleichbare Messungen zu erhalten, ist es wichtig, eine fest definierte Testumgebung zu verwenden. Durch diese Homogenität sind die Tests leicht reproduzierbar und vergleichbar. Um die Messergebnisse interpretieren zu können, werden aus diesen verschiedene Metriken berechnet. Mithilfe dieser Metriken kann man einen Vergleich zu anderen Systemen ziehen.

## 1.3 Implementierungen von Triple-Stores

### 1.3.1 Aufbau

Wie in Kapitel Unterabschnitt 1.1.2. Persistenz mit Hilfe eines Triple-Stores eingeführt, besteht ein Triple-Store aus mehreren Elementen, wie einem Query-Parser, einem Optimierer, einem Reasoner, einem SPARQL-Endpoint und einem Query-Ausführer. Diese kann man im Schaubild 1.3 zusammen mit den verschiedenen Zusammenhängen deutlich sehen. Im Folgenden werden diese kurz erläutert.

#### **SPARQL-Endpoint:**

Der SPARQL-Endpoint ist die Schnittstelle zwischen dem Triple-Store und dem Nutzer. Der Nutzer schickt Anfragen an den SPARQL-Endpoint und bekommt die Resultate auch wieder über diesen zurück.

#### **Query-Parser:**

Der Query-Parser bekommt die Queries des Nutzers durch den Endpoint geliefert. Hier wird die Query dann aus der Hochsprache, also der Sprache die für den Nutzer am verständlichsten ist und von SPARQL definiert wird, in die SPARQL-Algebra übersetzt. Die SPARQL-Algebra ist eine Sprache, in welcher die Query so vereinfacht wie möglich steht und für Maschinen



gut lesbar ist. Die Sprache besteht aus einer fest definierten Liste an Operatoren und ist eine abstrakte Repräsentation der Abfrage.

#### **Optimierer:**

Der Optimierer arbeitet auf der SPARQL-Algebra. Ein Optimierer kann verschiedene Optimierungsstrategien verfolgen. Es gibt eine statistisch basierte Strategie, eine fixe Strategie oder die Strategie ohne Neusortierung. Sind mehrere Strategien gegeben, so werden sie wie folgt priorisiert: Statistisch >Fix >Ohne Neusortierung.

Die **statistische Strategie** beruht auf gegebenen Metadaten des Datensets. In diesen Metadaten stehen beispielsweise der Zeitpunkt, wann der Graph erstellt wurde, und wie groß der Graph ungefähr ist. Auch stehen für die jeweiligen Prädikate eine ungefähre Angabe, wie viele im Datenset vorzufinden sind. Eine ungefähre Angabe der Anzahl reicht, da diese nur den Optimierer unterstützen bei der Bevorzugung eines Ausführungsplans über einen Anderen. Bei der **fixen Strategie**, gibt es einen Satz an Regeln, nach welchen optimiert wird. Diese Regeln können durch den Triple-Store schon vorgegeben sein oder durch den Nutzer selbst erstellt werden.

Bei der **Strategie ohne Neusortierung**, werden nur die Filter optimiert, während der Rest der Query so bleibt wie er war. Die Filter werden optimiert, indem die Filter so klein wie möglich gehalten werden und an der Stelle eingefügt werden, an dem die Filtervariable das erste mal verwendet wird.

Diese vorgestellten Strategien werden kategorisiert als „statische“ Optimierungen, da diese vor der Ausführung und auf statischen Daten ausgeführt werden. Neben der statischen Optimierung gibt es auch eine sogenannte „dynamische“ Optimierung. Bei dieser werden die Anfragen während der Ausführung basierend auf vorherigen Anfragen und Ergebnissen optimiert.

#### **Reasoner:**

Der Reasoner arbeitet, wie in Abschnitt 1.1.2 bereits erklärt, auf der Ontologie. Das Programm stellt implizite Informationen, wie in 1.2 mit dem gestrichelten Pfeil dargestellt, explizit dar.

Ein Datenset kann auf verschiedene Arten in einem Triple-Store gespeichert werden. Diese sind:

- *Native*: Das System wird von Grund auf neu gebaut. Dies hat den Vorteil, dass man die Datenbank dem RDF entsprechend aufbauen kann, wodurch man schon eine Performancesteigerung bemerken kann.
- *RDBMA-backed*: Der Triple-Store wird mithilfe eines RDF-Layers auf ein RDBMS abgebildet. Ein RDBMS ist ein Relational Database Management System und ist die Basis für SQL.

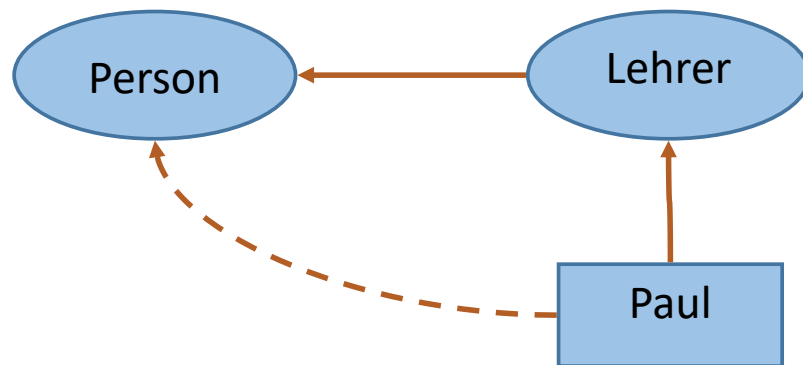


Abbildung 1.2: Funktion Reasoner

- *NoSQL*: Der Triple-Store wird mithilfe eines NoSQL Stores dargestellt. NoSQL-Stores besitzen kein festes Schema und beruhen nicht auf einem relationalen System.

### 1.3.2 Triple-Store Implementierungen

Im Folgenden werden verschiedene Triple-Store Implementierungen vorgestellt. Hierunter ist ein kommerziell erwerblicher Triple-Store, WebGenesis, und zwei „free-to-use“, Apache-Rya und Apache-Jena-Fuseki.

#### Fuseki

Fuseki ist ein Triple-Store, welcher im Hintergrund das semantische Framework Apache Jena nutzt um SPARQL-Anfragen zu bearbeiten.

Fuseki kann Daten auf verschiedene Weisen speichern. Es gibt die Option ein Datenset entweder „in-memory“, „persistent“ oder „persistent mit TDB2“ zu speichern. Dies bezieht sich auf die verschiedenen Arten von Jena Informationen zu speichern. Von Jena werden Informationen als gerichteter Graph gespeichert. Dies wird durch die Darstellung der Informationen als Tripel erreicht. Diese Art Informationen zu speichern, bedeutet, dass es für Jena möglich ist, eine Anzahl an verschiedenen Speicherstrategien äquivalent anzubieten. Zudem ist es auch möglich externe Triple-Stores mit Jena zu verbinden, solange ein Adapter existiert, welcher die Zugriffe der API ermöglicht. Eine API (Application Programming Interface) ist die Schnittstelle zwischen der „Außenwelt“ und dem Inneren. Sie gibt gewisse Funktionen frei, mit welchen man dann kontrolliert Dinge im Inneren machen kann.

Sind die Daten im Triple-Store abgelegt, so verwendet Fuseki Apache ARQ um Queries zu parsen und einen Reasoner über die Daten laufen zu lassen. Apache ARQ ist ein SPARQL Prozessor für Jena, welcher SPARQL Queries übersetzt und nach Möglichkeit optimiert.

Jena besitzt auch eine „inference API“, welche starke Ähnlichkeiten mit Reasonern aufweist, da mithilfe gegebener Regeln Folgerungen über das Datenset getroffen werden. In Jena sind schon die Regeln von OWL und RDFS (RDF Schema), auch eine Ontologie, eingebaut, allerdings ist es auch möglich einen externen Reasoner mit der API zu verbinden und somit speziellere Reasoning Algorithmen laufen zu lassen. Diese spezielleren Algorithmen können dann komplexer oder besser an das Szenario angepasst worden sein.

#### WebGenesis

WebGenesis ist eine serviceorientierte Entwicklungsplattform und ein Ablaufsystem für Web-basierte Informationssysteme. Im Hintergrund werden die Daten mithilfe eines Triple-Stores

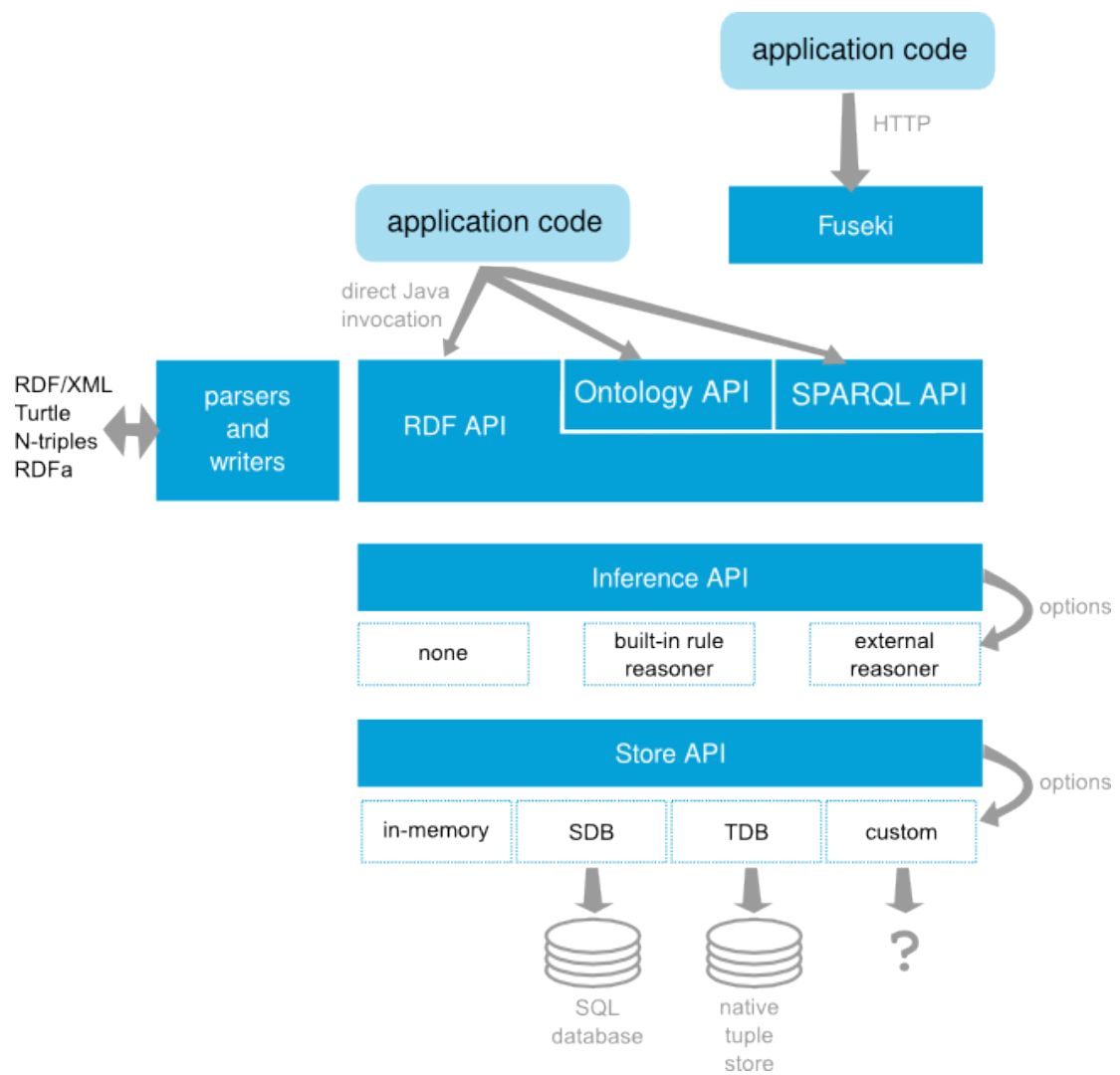


Abbildung 1.3: Jena Architektur [Apa19b]

bearbeitet. Für den Zugriff und die Bearbeitung der Daten läuft Jena als Semantic Web Framework und unterstützt somit Anfragen sowohl im SPARQL- als auch im SQL-Format.

### **Apache Rya**

Apache Rya ist eine der neuesten Triple-Stores zum Zeitpunkt der Arbeit. Rya basiert nicht auf dem Framework Jena sondern auf Accumulo und sondert sich damit von den anderen vorgestellten Triple-Stores ab.

In dieser Arbeit wurde der Fuseki Tripel-Store verwendet, da dieser am einfachsten zu verwenden war und auch Parameter, wie Heap-Space 5.4.1, einstellbar waren. Allerdings wurde bei der Konstruktion der Tests darauf geachtet, dass diese auch bei andere Triple-Stores ohne große Änderungen anwendbar sind. Dies wurde erreicht, indem alle Anfragen und Aktionen auf den Tripel-Store nur über den standardisierten SPARQL-Endpoint ausgeführt wurden.

### **1.3.3 Speicher**

Daten können im System auf verschiedene Weisen gehalten werden. Hier werden diese Arten kurz mit ihren Vor- und Nachteilen aufgezeigt.

#### **in-memory**

Die Daten werden „in-memory“, also im Arbeitsspeicher, gespeichert. Dies wird gemacht, da der Arbeitsspeicher wesentlich höhere Zugriffsgeschwindigkeiten bietet als Festplattenlaufwerke. Ein Nachteil bei der Speicherung der Daten „in-memory“ ist, dass beim Herunterfahren der Maschine die Daten verschwinden und für die nächste Verwendung wieder hineingeladen werden müssen. Da alle Daten im Arbeitsspeicher gelagert werden, steigt dessen benötigte Größe mit der Anzahl der Daten. Dies wird in heutigen Computersystemen durch Auslagerung einiger Daten aus der Memory in den Festplattenspeicher erreicht. Dies führt sehr schnell zu großen Performance-Verlusten, da die Zugriffszeiten sich in einem Faktor von 1000 unterscheiden.<sup>1</sup> Ist der Arbeitsspeicher voll, so hat das System die Möglichkeit Daten auszulagern. Hierbei werden die Daten vom Arbeitsspeicher auf den Festplattenspeicher übertragen. Durch diese Übertragung werden die Daten im Arbeitsspeicher danach für das Löschen freigegeben. Für dieses Löschen existiert ein Prozess „Garbage Collector“, welcher solche Daten endgültig löscht. Ist der Arbeitsspeicher zu klein gewählt, kann es vorkommen, dass der Garbage Collector mit

---

<sup>1</sup> <https://www.storagereview.com/node/2700> - 16.12.2019

seiner Arbeit nicht hinterherkommt. Dies kann dann entweder dazu führen, dass die Maschine nur sehr langsam arbeitet oder sogar einen Fehler wirft.

### **Persistenter Speicher**

Werden die Daten persistent gespeichert, werden die Daten auf der Festplatte oder der SSD gespeichert. Dies hat den Vorteil, dass die Daten persistent gespeichert werden, also auch nach einem Neustart noch vorhanden sind. Der Nachteil bei dieser Methodik ist, dass die Zugriffsgeschwindigkeit im Vergleich zum Arbeitsspeicher sehr viel geringer ist.

## 2 Problemstellung

In sehr vielen Bereichen des allgemeinen Lebens werden mittlerweile Datenbanken eingesetzt. Ob an der Kasse im Supermarkt, im Online-Shop, bei einem Verein oder bei Notfalleinsatzkräften, überall werden Datenbanken und mittlerweile auch Triple-Stores verwendet. Mit der Verwendung von Datenbanken in Echtzeitsystemen, wie der Koordination von Einsatzkräften in einem Krisengebiet, wird die Frage nach der Leistung der Datenbank sehr wichtig [Her+18]. So wird beispielsweise in dem Projekt „Semantic Queries Supporting Crisis Management Systems“ geschrieben, wie wichtig es ist, up-to-date Informationen im Krisenmanagement zu besitzen [Sch+19].

Um Datenbanken in kritischen Anwendungen zu verwenden, müssen sie Datenanfragen innerhalb einer bestimmten Zeit beantworten können. Diese Problemstellung wird in dieser Bachelorarbeit aufgegriffen und mit der Themenstellung „Leistungsanalyse und -optimierung von semantischen Anfragen“ untersucht.

Um die Frage nach der Dauer der Anfrage zu beantworten, wurde die Fragestellung wie in Abbildung 2.1 gezeigt zuerst in zwei Unterfragen aufgeteilt und danach wieder zusammengeführt. Die beiden Unterfragen ergeben sich hierbei aus der Frage nach der Leistungsanalyse und der Frage nach der Leistungsoptimierung. Die Frage nach der Leistungsanalyse wird mit der Frage nach einem semantischen Benchmark untersucht, da mit einem Datenbank-Benchmark Unterabschnitt 1.2.2. Benchmark für Performance Evaluierung untersucht wird, welche Performance eine Datenbank bietet. „Welcher Benchmark eignet sich für das messen von semantischen Abfragen?“

Die zweite Frage nach der Leistungsoptimierung wurde mit der Frage nach der Optimierung von semantischen Queries untersucht. Hierbei soll die Leistungsoptimierung nicht von den verwendeten Komponenten abhängen, sondern vom Design der semantischen Query. „Welche Komponenten einer Query können getauscht werden um dasselbe Ergebnis zu erzielen“.

## 2.1 Benchmarks

In dieser Unterfrage wird untersucht, welche Benchmarks es gibt, worin sie sich unterscheiden und welcher Benchmark für die Problemstellung am besten geeignet ist. Hierfür wurden zuerst einige Benchmarks ausgesucht und untersucht. In einem ersten Schritt wurden die Eigenschaften der Benchmarks verglichen. Danach wurde ein Kategorisierungsschema anhand der verschiedenen Komponenten der Benchmarks konstruiert und die verschiedenen Benchmarks darin eingeteilt.

## 2.2 Queries

In dieser Unterfrage wird untersucht, wie eine Query, was eine SPARQL Anfrage ist, geschrieben werden kann, damit diese möglichst performant, also so leistungsstark wie möglich, ist. Hier wurde angefangen mit der Untersuchung der verschiedenen Komponenten einer Query. Aufbauend auf dieser Untersuchung wurden, wo möglich, unterschiedliche Queries definiert, die das gleiche Ergebnis liefern. Ein Vergleich dieser soll helfen, den Einfluss der Query-Komponenten auf die Ausführungszeit zu beurteilen.

## 2.3 Evaluierung

Nachdem die beiden Unterfragen beantwortet wurden, werden die gefundenen Ergebnisse zusammengeführt und evaluiert. Bereits bei ersten Tests war ersichtlich, dass es bei einigen Query-Paaren, trotz unterschiedlich formulierter Anfrage, keine Unterschiede in der Abfragegeschwindigkeit gab. Daraus folgte die Vermutung, dass der Optimierer die beiden Queries wieder in dasselbe Format umändere. Um diese Vermutung zu bestätigen, wurde für jedes Query-Paar das interne Ergebnis des Optimierers untersucht. Daraus ließen sich Vermutungen ableiten, welche der beiden Queries aus welchem Grund schneller ausgeführt werden kann. Abschließend wurden diese Vermutungen durch experimentelle Überprüfung mittels des Benchmarks verifiziert, um abschließend Empfehlungen für das Query-Design aussprechen zu können.



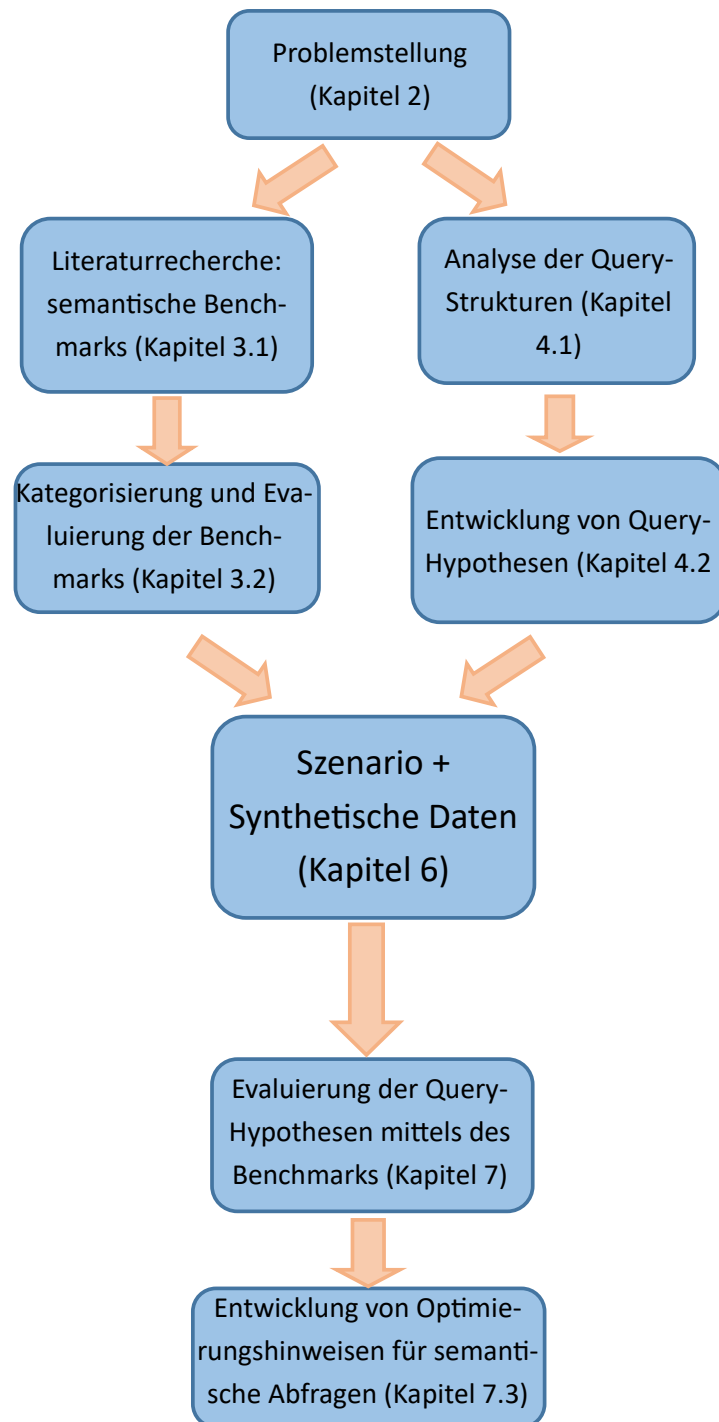


Abbildung 2.1: Vorgehensweise



## 3 Benchmark für semantische Anfragen

### 3.1 Aufbau eines Benchmarks

Wie in 1.2.1 beschrieben, besteht ein Benchmark für jedes Gebiet aus verschiedenen Metriken. Hier wurde der Benchmark auf dem Gebiet der Triple-Stores betrachtet. Um die Benchmarkergebnisse nun objektiv miteinander vergleichen zu können, müssen bestimmte Metriken erfüllt sein, damit diese miteinander verglichen werden können. Diese Metriken werden für die Vereinheitlichung der Messergebnisse und somit auch den objektiven Vergleich benötigt. Diese Metriken wurden aus dem Paper [Sal+19] genommen. Diese sind wie folgt:

- **Anfragebearbeitung:** Hier werden sowohl die Ausführungszeit als auch die CPU- und Arbeitsspeicher-Nutzung von Queries gemessen. Da allerdings, sowohl die CPU-, als auch die Arbeitsspeichernutzung stark vom System abhängig sind und heutzutage bei den wenigsten Systemen für Probleme sorgt, werden diese Metriken von den meisten Benchmarks nicht gemessen. Bei der Messung der Ausführungszeit ist es üblich, Query-Mixes pro Stunde oder Queries pro Sekunde zu messen, da oft sehr viele Queries gemessen werden und einzelne Queries sehr schnell ausgewertet sind. Hierbei ist ein Query-Mix eine bestimmte Reihenfolge von Queries welche ausgewertet werden.
- **Import:** Hier wird gemessen, wie lange es dauert, Daten in den Triple-Store zu laden und dort zu indexieren. Dies ist wichtig, da man Queries erst ausführen kann, wenn beide Schritte durchgeführt sind.
- **Ergebnismenge:** Um verschiedene Systeme vergleichen zu können, ist es wichtig, dass beide Systeme dieselben Ergebnisse liefern. Dementsprechend müssen die Resultsets vollständig und korrekt sein.
- **Parallele Ausführung mit/ohne Update:** Manche Benchmarks messen ob ein Triple-Store parallel laufen kann, wenn gleichzeitig mehrere Nutzer auf einem Triple-Store Daten lesen und schreiben.

Beinhaltet ein Benchmark diese Metriken, so kann man verschiedene Triple-Store-Implementierungen

und Programme miteinander vergleichen.

Gängige Komponenten eines Benchmarks sind:

- **Datengenerator:** Ein Generator, der ein Datenset, meistens auch beliebiger Größe, erstellen kann. Die Größe des Datensets wird vom Datengenerator durch die Anzahl der Tripel in dem Datenset pro Prädikat variiert.
- **Driver:** Der Driver ist ein ausführbares Programm, welches gegen einen gegebenen SPARQL-Endpoint eine vom Benchmark vorgegebene oder benutzerdefinierte Reihe an Queries testet und nach dem Durchlauf eine Liste mit gemessenen Metriken ausgibt.
- **Query-Template-Parser:** Ein Query-Template-Parser ist ein Teilprogramm des Drivers und ermöglicht es Query-Templates mit festen Werten zu füllen, welche dann dem SPARQL-Endpoint geschickt werden. Ein Query-Template ist hierbei eine Anfrage, welche Variablen für bestimmte feste Werte, wie Zahlen oder bestimmte Knoten, enthält. Diese Werte werden jede Anfrage variiert, sodass der Tripel-Store mit zufälligen Queries arbeiten muss.
- **Querygenerator:** Ein Generator, der für einen LSQ Dump („Linked SPARQL Query“ einem Query-Log-Dump vom Server mit einer Normalisierung über die Daten) oder ähnliches entsprechende Queries generiert. Die entstandenen Queries sollen dann für das Datenset eine realistische Anwendung sein, wodurch man auch in dem System mit „realistischen Daten“ arbeiten kann.

### 3.2 Kategorisierungsschema für Benchmarks

Um die Eignung der Benchmarks, für die in dieser Arbeit betrachteten Fragestellung zu prüfen, wurde ein Kategorisierungsschema basierend auf den im vorangegangenen Abschnitt aufgeführten Komponenten entwickelt. Die Motivation dieses Kategorisierungsschemas war das Finden eines für diese Arbeit passenden Benchmarks. Dieser Benchmark muss über frei verfügbaren Code und eine Lizenz für das Ändern des Codes verfügen. Weiterhin muss er ein Datenset generieren und mit Query-Templates arbeiten können. Viele Benchmarks erfüllen nicht alle Kriterien, sondern nur einen Teil dieser.

Im Folgenden werden die verschiedenen Kategorien, in welche die Benchmarks eingeteilt werden, aufgelistet und zu jedem Punkt wird ein Anwendungsfall angegeben:

- **Datensetgenerator:** Ein System wird auf die Ladegeschwindigkeit verschieden großer Datensets getestet.

- **Querygenerator:** Für ein System sollen, mit einem vordefinierten Datenset, Queries erstellt werden.
- **Driver:** Ein, mit Informationen gefülltes, System soll getestet werden.
- **Generator+Driver:** Ein Benchmark wird benötigt, welcher Datensets erstellen kann, und diese dann mit vorhandenen Queries testen kann.
- **Speziell:** Der Benchmark hat keinen eigenen Generator oder Driver.

### 3.3 Benchmarks zur Evaluierung von Triple-Stores

In diesem Abschnitt wurden alle betrachteten Benchmarks in einer Tabelle 3.1 mit den verschiedenen möglichen Eigenschaften verglichen. Mithilfe der Ergebnissen dieser Tabelle wurden die Benchmarks dann in die verschiedenen oben genannten Kategorien eingeteilt.

- **BO** (*Benutzerdefinierte Ontologie/Datenset*): Die zum Test verwendete Ontologie kann frei gewählt werden.
- **DaGe** (*Hat Datengenerator*): Es ist möglich, ein Datenset mit einer benutzerdefinierten Größe zu bauen.
- **QuGe** (*Hat Querygenerator*): Es ist möglich, mit einem LSQ-Dump Queries zu erstellen.
- **BQ** (*Benutzerdefinierte Queries*): Die Queries sind veränderbar.
- **Driver** (*Hat Driver*): Der Benchmark hat einen Driver, welcher gegebene Queries gegen einen SPARQLEndpoint testet, und zudem eine Evaluierung der Ergebnisse liefert.
- **MC** (*Multi-Client möglich*): Für den Benchmark ist es möglich, mehrere User darzustellen und Update- und Abfrage-queries auf die User zu verteilen.
- **Letztes Update:** Letzte (Code)-Änderung des Benchmarks.
- **Verfügbar:** Quellcode des Benchmarks ist frei verfügbar.
- **Autor:** Autor(en) des Benchmarks.
- **Lizenz:** Lizenz des Benchmarks.

Benchmarks	BO	BQ	MC	DaGe	QuGe	Driver	Letztes update	Lizenz
Berlin SPARQL Benchmark	Nein	Ja	Ja	Ja	Nein	Ja		2012 Apache License 2.0
Lehigh University Benchmark	Nein	Ja	Nein	Ja	Nein	Ja		2004 GNU General Public License (GPL)
FedBench	NB	NB	NB	Nein	Nein	Ja		2013 GNU Lesser GPL
Feasible	Nein	Ja	NB	Nein	Ja	Nein		2018 GNU Affero General Public License v3.0
LargeRDFBench	Nein	Ja	Nein	Nein	Nein	Nein		2018 GNU Affero General Public License v3.0
University Ontology Benchmark	Nein	Nein	NB	Ja	Nein	Nein		NB
SPARQL Performance Benchmark	Nein	Nein	NB	Ja	Nein	Nein		Berkeley License
Social Network Intelligence Benchmark	Nein	Nein	NB	Ja	Nein	Ja		NB
Linked Data Integration Benchmark	Nein	Ja	NB	Ja	Nein	Nein		2012 BSD License
Linked Open Data Quality Assessment	Nein	Nein	NB	Nein	Nein	Ja		2012 BSD License
LinkBench	Ja	Ja	NB	Ja	Nein	Ja		2015 Apache License 2.0
Waterloo SPARQL Diversity Test Suite	NB	Ja	NB	Ja	Ja	Nein		MIT License
Semantic Publishing Benchmark	Nein	Nein	NB	Ja	Nein	Ja		2019 NB
IGUANA	Ja	Ja	NB	Nein	Nein	Ja		2019 GNU Affero General Public License v3.0
SPARQLBench	NB	Ja	NB	Nein	Ja	Nein		2019 GNU General Public License (GPL)

Abbildung 3.1: Benchmarks bezüglich verschiedener Punkte

Anhand der Tabelle lassen sich die verschiedenen Benchmarks in die verschiedenen Kategorien einteilen.

Die folgenden Benchmarks wurden aus der Liste des W3C für RDF Benchmarks genommen. [W3C18]. In die Kategorie „Datensetgenerator“ fallen der University Ontology Benchmark,<sup>1</sup> der SPARQL Performance Benchmark<sup>2</sup> und der Linked Data Integration Benchmark.<sup>3</sup>

In die Kategorie „Querygenerator“ fallen Feasible,<sup>4</sup> Waterloo SPARQL Diversity Test Suite<sup>5</sup> und SPARQLBench.<sup>6</sup>

In die Kategorie „Nur Driver“ fallen IGUANA,<sup>7</sup> Linked Open Data Quality Assessment,<sup>8</sup> Fed-Bench.<sup>9</sup>

In die Kategorie „Generator + Driver“ fallen Berlin SPARQL Benchmark,<sup>10</sup> Lehigh University Benchmark,<sup>11</sup> Social Network Intelligence Benchmark,<sup>12</sup> LinkBench,<sup>13</sup> Semantic Publishing Benchmark.<sup>14</sup>

Der letzte Benchmark ist LargeRDFBench.<sup>15</sup> Dieser hat eine große Anzahl an Ontologien zur Verfügung und besitzt auch Queries, hat allerdings keinen eigenen Generator oder Driver. Dementsprechend fällt dieser in die Kategorie der „speziellen“ Benchmarks.

Mithilfe von Google Scholar wurden die in der Fachliteratur meist rezipierten Benchmarks herausgefunden, indem verglichen wurde, wie oft diese von anderen Papern zitiert wurden. Google Scholar ist die Standardsuchmaschine für Paper und zeigt bei den Ergebnissen auch die Anzahl der Referenzen mit an. Die meist verwendeten Benchmarks sind der Berlin SPARQL Benchmark (BSBM), LUBM und SP<sup>2</sup>Bench.

Der BSBM wurde 2009 veröffentlicht und besteht aus einem skalierbaren Datenset, drei verschiedenen Query-Mixes, Performance-Metriken, welche die Ergebnisse auswerten, einem Daten-Generator und einem Test-Driver, welcher die Queries ausführt.

<sup>1</sup> <https://www.cs.ox.ac.uk/isg/tools/UOBMGenerator/>

<sup>2</sup> <http://dbis.informatik.uni-freiburg.de/forschung/projekte/SP2B/>

<sup>3</sup> <http://wifo5-03.informatik.uni-mannheim.de/bizer/lodib/lodib.htmltools>

<sup>4</sup> <https://github.com/dice-group/feasible>

<sup>5</sup> <https://github.com/comunica/watdiv-docker>

<sup>6</sup> <https://github.com/dice-group/SPARQL-Bench>

<sup>7</sup> <https://github.com/dice-group/IGUANA>

<sup>8</sup> <http://lodqa.wb3g.de/lodqa.htmldownload>

<sup>9</sup> <https://code.google.com/archive/p/fbench/downloads>

<sup>10</sup> <http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/BenchmarkRules/index.html#datagenerator>

<sup>11</sup> <http://projects.semwebcentral.org/projects/lubm/>

<sup>12</sup> <http://ldbcouncil.org/developer/snb>

<sup>13</sup> <https://github.com/facebookarchive/linkbench>

<sup>14</sup> [https://github.com/ldbc/ldbc\\_spm2.0](https://github.com/ldbc/ldbc_spm2.0)

<sup>15</sup> <https://github.com/dice-group/LargeRDFBench>

SP<sup>2</sup>Bench wurde ebenfalls 2009 veröffentlicht, hat aber im Gegensatz zu BSBM keinen Test-Driver.

Der LUBM-Benchmark wurde 2005 veröffentlicht und ähnelt dem BSBM-Benchmark sehr, allerdings hat der LUBM-Benchmark keine Query, welche auch komplexere SPARQL-Operatoren abfragt. Ein SPARQL-Operator ist ein Operator in einer SPARQL Anfrage, wie zum Beispiel ein Filter-Operator oder ein LIMIT-Operator.

Der IGUANA-Benchmark wurde noch als Auswahl genannt, da der Benchmark sehr neu ist und daher noch nicht so bekannt ist. IGUANA ist der aktuellste Benchmark mit einem Veröffentlichungsdatum von 2017. Allerdings hat IGUANA keinen Daten-Generator.

Die restlichen Benchmarks wurden hier nicht näher betrachtet, da sie entweder keinen Test-Driver hatten (FedBench, Feasible, LargeRDFBench, UOBM, WatDiv), keinen Daten-Generator hatten (LODQA) oder nicht mehr verfügbar waren (SIB, THALIA, JustBench).

### 3.4 Berlin SPARQL Benchmark

Der Berlin SPARQL Benchmark (BSBM) wurde in dieser Arbeit ausgewählt, da er alle geforderten Kriterien erfüllt. Der Berlin SPARQL Benchmark beinhaltet einen Datengenerator, welcher frei skalierbare Datensets ermöglicht, und 12 Query-Templates, welche zur Ausführungszeit mit zufallsgenerierten Werten gefüllt werden. BSBM hat einen Driver, welcher verschiedene Querymixes ausführt, und beinhaltet verschiedene Query-Mix Templates, welche ein realistisches Szenario nachahmen sollen. Ein solches Szenario soll beispielsweise das Such- und Navigationsverhalten eines Kunden auf der Suche nach einem Produkt darstellen. [Biz12] Die Queries können sehr einfach verändert werden, und können sich dementsprechend auch an verschiedene Datensets anpassen. Der Driver berechnet verschiedene Metriken wie zum Beispiel AQET (Average Query Execution Time), wobei er einmal den geometrischen und einmal den arithmetischen Mittelwert berechnet, minQET / maxQET (min/max Query Execution Time), QPS (Queries per Second), Average result, min/max result.

- AQET: Durchschnittliche Ausführzeit einer Query.  

$$AQET = \frac{\sum executionTime}{nrRuns}$$
 über mehrere Ausführungen
- min-/maxQET: Die minimale und maximale Ausführzeit einer Query.
- QPS: Durchschnittliche Ausführmenge einer Query in einer Sekunde.  

$$QPS = \frac{queryMixtotalRuntime * aqet}{queryMixMultiThreadRuntime}$$
 oder ohne Multithreading  $\frac{1}{aqet}$
- AVGResults: Durchschnittliche Anzahl an zurückgegebenen Tripeln  

$$AVGResult = \frac{\sum numberResults}{nrRuns}$$



### 3.4.1 Verwendete Metriken

In dieser Arbeit wurden die Metriken QPS und AVGResults verwendet.

QPS wurde verwendet, da man mit diesen einen Vergleich zwischen den einzelnen Query-Designs ziehen kann. AVGResults wurde betrachtet, da es an manchen Stellen der Designs unterschiedlich große Rückgaben gab und diese unterschiedlichen Größen auch die Ausführungszeit beeinflussen.

### 3.4.2 Ontologie Schema

Das Schema der Ontologie von BSBM soll einen E-Commerce Datenset darstellen. Dieses Datenset besteht aus Produkten, welche jeweils einen Hersteller, einem Typ, einigen Features, Bewertungen und einem Angebot besitzen. Ein Produkt besitzt außerdem noch einen Titel, eine Beschreibung und besitzt mehrere eigene Zufallswerte. Ein Produkttyp verweist auf sich selbst, da ein Typ ein Untertyp eines anderen Typs sein kann. Eine Bewertung besitzt einige spezielle Werte und jede Bewertung wurde von einer Person geschrieben. Ein Angebot für ein Produkt wird von einem Verkäufer angeboten. Einen zusammenhängenden Graphen kann man in 3.2 sehen. In diesem Graphen sind nur die, in dieser Arbeit verwendeten Werte, inkludiert.

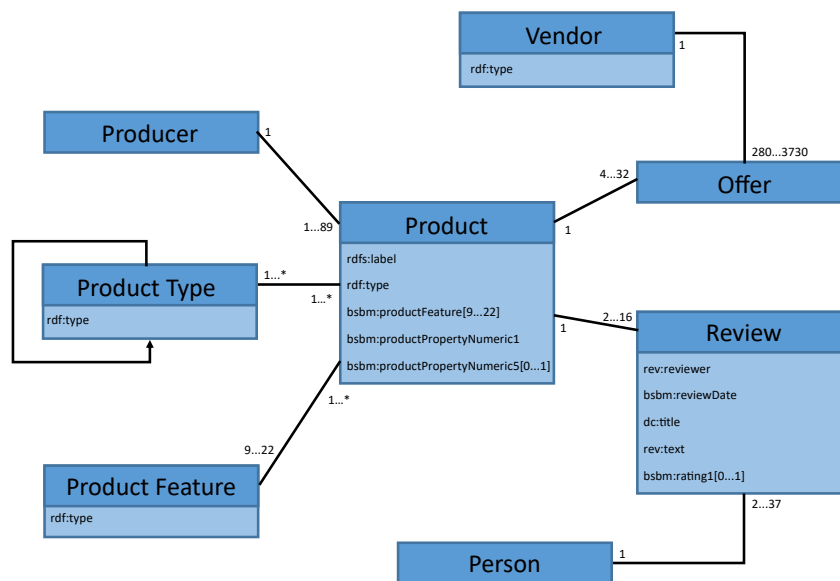


Abbildung 3.2: Ontology Schema

## 4 Optimierungsmöglichkeiten semantischer Anfragen

### 4.1 Elemente einer SPARQL-Query

Um mögliche Einflussfaktoren innerhalb der Queries zu ermitteln, wurden die Bestandteile einer Query genauer betrachtet. Eine Query besteht aus einer Angabe, welche Werte zurückgegeben werden sollen (Zeile 2), einem Teil, bei welchem die Daten eingeschränkt werden (Zeilen 4-5) und einem Teil, bei welchem die Ergebnisse nochmal bearbeitet werden können (Zeile 7).

```
1
2 select ?product where {
3
4     ?product bsbm:ProductPropertyNumeric1 ?value.
5     Filter (?value < 2)
6
7 } LIMIT 5
```

Listing 4.1: Query

Als Weiteres wurde das mögliche Vorwissen für eine Query betrachtet, welches vorhanden sein könnte. Vorwissen sind hierbei zusätzliche Informationen, welche ein Benutzer hat, eine Maschine jedoch nur durch Reasoning in Erfahrung bringen kann. Es wird untersucht, ob durch das zusätzliche Einbringen von diesen Informationen ein Leistungszuwachs er erreichen ist. Mithilfe der Dokumentation des W3C [W3C19a] konnten verschiedene Komponenten, wie die Struktur oder die Operatoren einer Query, gefunden werden, welche einen Einfluss auf die Leistung einer Query haben können. Anhand dieser Komponenten und deren Wirkung auf die Query wurden 6 Oberpunkte für das Design von semantisch gleichen Queries identifiziert. Diese 6 Oberpunkte sind:

**Zusätzliche Information:** Man kann einer Query zusätzliche Informationen geben, wodurch

der Triple-Store diese nicht erst bei der Ausführung erkennen muss. So könnte man beispielsweise den Typ eines Prädikates mit angeben.

**Reasoning:** Man kann Informationen, welche durch Reasoning entstehen würden, explizit in der Query angeben, damit diese nicht erst während der Laufzeit erkannt werden müssen.

**Datengröße verändern:** Man kann die zu bearbeitende Datengröße verändern. Dies kann entweder die Größe des Datensets während der Ausführung sein oder auch die Größe der Ausgabe. Es wird angenommen, dass die Laufzeit mit einem größeren Datenset und Ausgabeset wächst.

**Tauschen von Operatoren:** Man kann Operatoren mithilfe anderer darstellen, aber dennoch dieselbe Abfrage haben. Ein Operator ist hierbei eine Operation innerhalb der Query. Beispiele hierfür sind das Filtern bestimmter Daten oder das Hinzufügen gewisser Constraints Operatoren in einer Query. Dies wird gemacht, da angenommen wird, dass verschiedene Operatoren verschiedene Komplexitäten haben und durch die unterschiedliche Darstellung, der Operatoren in den Queries das gleiche Ergebnis erreicht werden kann.

**Graph-Struktur vs Daten:** Hier soll nach einer gewissen Struktur gesucht werden anstelle von Daten. Hierbei soll untersucht werden, ob es einen Unterschied macht, ob nach einer gewissen Datenstruktur oder nach Daten gefiltert wird. Beispielsweise: Suche nach einer Klasse A, die eine Referenz zu Klasse B und C hat. Im Gegensatz zu: Suche nach einer Klasse A, die die Werte b, c und d hat.

**Query-Struktur ändern:** Man kann die Query-Struktur ändern. Hier wird untersucht, ob es einen Unterschied macht die Reihenfolge der Tripel zu verändern.

Bis hier hin wurden einige potenzielle Einflussfaktoren einer Query aufgezeigt. Im Folgenden soll untersucht werden, welche dieser Faktoren einen Einfluss auf die Geschwindigkeit der Ausführung der Query hat. Um dies zu prüfen, wurden jeweils zwei Queries gegenübergestellt: i)(bzw. A) ohne diesen Faktor ii)(bzw. B) mit diesem Faktor. Die Unterschiede wurden rot markiert, um die Unterschiede klar zu machen. Soweit möglich wurde darauf geachtet, dass ansonsten keine weitere Eigenschaft (bspw. Größe der Antwort) geändert wurde. Die verschiedenen Design-Unterschiede sind wie folgt:

#### 1. **Zusätzliche Info**

Zusätzliche Information, welche implizit im Graphen vorhanden ist, aber explizit angegeben wird.

##### a) **Typ**

i. **Typ nicht angegeben**

In der Query werden keine Typen für die Instanzen angegeben, außer es ist notwendig

*Suche alle Instanzen die von Typ7 sind.* 8.14

ii. **Typ angegeben**

In der Query wird der Typ der Instanzen mitangegeben.

*Suche alle Instanzen mit Typ Product. **Suche alle Produkte die von Typ7 sind.*** 8.14

2. **Info durch Reasoning explizit**

Man gibt explizit eine Verbindung an, welche man sonst nur durch Reasoning bekommen würde

a) **Angabe des Obertypes**

i. **Obertyp/Untertyp nicht angegeben**

Man gibt die Oberklasse von einer Klasse nicht an.

*Suche alle Produkte mit ProductType8* 8.8

ii. **Obertyp/Untertyp angegeben**

Man gibt die Oberklasse von einer Klasse explizit an.

*Such alle Produkte mit ProductType1 **ProductType8 ist eine Subklasse von ProductType1*** 8.8

b) **Abfrage einer Relation einer Oberklasse**

i. **Relation abfragen ohne Oberklasse**

Man fragt nach einer Relation der Oberklasse.

*Suche alle Freunde von einem Lehrer*

ii. **Relation abfragen mit Oberklasse**

Man fragt nach einer Relation der Oberklasse und gibt diese spezifisch an.

***Lehrer ist eine Person**, gib alle Freunde von einem Lehrer aus*

c) **Simple Anfrage**

Hier wird angenommen, dass in dem Datenset alle Personen Lehrer sind.

i. **Simple Abfrage ohne Oberklasse**

Gib die oberste Klasse an, so dass erst durch alle Klassen gegangen werden muss.

*Suche alle Personen.*

ii. **Simple Abfrage mit Oberklasse**

Gib die Klasse direkt über den Daten an, sodass nicht erst durch alle Klassen

gegangen werden muss.

*Suche alle Lehrer.*

#### d) **Prepare**

##### i. **Prepare() nicht aufrufen**

Mit einem vorgelegten Prepare() die durch den Reasoner bereitgestellten Regeln vorbereiten.

##### ii. **Prepare() aufrufen**

Den Reasoner nicht vorbereiten.

### 3. **Datengröße**

Hier wird die Komponente der Datengröße variiert. Entweder bei der Größe der Ergebnismenge oder beim Speichern der Zwischenergebnisse zwischen Tripeln. Die Zwischenergebnisgröße wurde gemessen, indem die Query an dieser Stelle beendet und die Ergebnisse ausgegeben wurden.

#### a) **Filter**

##### i. **FilterPos**

###### A. **Position am Ende**

Filter am Ende der Query platzieren.

*Suche alle Personen und Namen und filter nach Personen mit Alter über 50*  
8.27

###### B. **Position oben**

Filter einsetzen sobald möglich.

*Such alle Personen, filter diese nach Alter über 50, und hole dann die Namen der gefilterten.* 8.27

##### ii. **FilterGr**

###### A. **Größe groß**

Ein Filter, welcher mehrere Eigenschaften filtert.

*Filter(age >50 und height <164)* 8.23

###### B. **Größe klein**

Mehrere Filter, welche jeweils nur nach einer Eigenschaft filtern.

*Filter (age >50) Filter (height <164).* 8.23

##### iii. **FilterTyp**

**A. String filtern**

Nach String filtern.

*Filter (name == „John“) 8.25*

**B. Numerisch filtern**

Nach numerischen Daten filtern.

*Filter (age >50) 8.25*

**b) Limit****i. Kein Limit/Offset**

Die Ausgabemenge nicht beschränken.

*Gib 1000 Resultate aus. 8.31*

**ii. Limit/Offset**

Die Ausgabemenge beschränken.

*Gib von 1000 Resultaten nur 5 aus. 8.31*

**c) Select****i. Select \***

Die Ausgabeprojektion nicht beschränken.

*Gib alles für eine Person aus 8.44*

**ii. Select ?a ?b**

Die Ausgabeprojektion auf bestimmte Werte beschränken.

*Gib für eine Person das Alter aus 8.44*

**iii. Select count(\*)**

Die Ausgabemenge zählen.

*Gib die Anzahl aller Personen aus 8.44*

**d) Anzahl****i. Große Ergebnismenge**

Große Ergebnismenge ausgeben.

*Nach allen Menschen suchen 8.2*

**ii. Kleine Ergebnismenge**

Kleine Ergebnismenge ausgeben.

*Nach allen Lehrern suchen 8.2*

**4. Operatoren**

Operatoren anders darstellen, um die Komplexität niedrig zu halten

a) **Textsuche**

Nutzung eingebauter, spezieller Funktionen im Vergleich zu generischen Funktionen

i. **Regex**

Regex-Filter.

*Filter(regex(„^herbert“), name) 8.40*

ii. **Str-Func**

Eingebaute STR-Funktionen.

*Filter(StrStarts(„herbert“, name)) 8.40*

b) **Union**i. **Optional**

OPTIONAL verwenden

*Suche alle Personen die OPTIONAL einen grünen Pulli besitzen 8.53*

ii. **UNION**

UNION verwenden

*Suche alle Personen und vereinige (UNION) diese mit allen Personen mit einem grünen Pulli 8.53*

c) **Distinct**i. **Distinct**

Distinct versichert, dass die Kardinalität von Tripeln gleich 1 ist.

*Suche alle DISTINCT Personennamen 8.20*

ii. **Reduced**

Reduced versichert das die Kardinalität zwischen 1 und der maximalen Kardinalität ist.

*Suche alle REDUCED Personennamen 8.20*

iii. **Reduced + LIMIT (Auf alle Tripel)**

Reduced + LIMIT MAX verändert Reduced, da die Werte nochmals durchlaufen werden und dadurch zusätzliche doppelte Daten entfernt werden.

*Suche alle REDUCED Personennamen und LIMIT (Anzahl Personennamen) 8.20*

d) **Minus**

Hier wird angenommen, dass Lehrer keinen Spaß an ihrem Job haben. Menge A - Menge B ergibt dieselben Ergebnisse wie Menge A filter (Daten c)



i. **Minus**

Menge A - Menge B schneller

*Alle Personen MINUS alle Lehrer* 8.33

ii. **Filter**

Menge A filter (Daten c)

*Alle Personen FILTER hat Spaß am Job* 8.33

5. **Graph-Struktur vs Daten**

Suche nach einer Struktur von Klassen, im Gegensatz zur Suche nach verschiedenen Daten. Die Ergebnismenge beider Suchen sind ungefähr gleich groß. 8.47

a) **Graph-Struktur**

Suche nach einer Graph-Struktur

*Suche alle Klassen A, die eine Referenz zu den Klassen B und C haben*

b) **Daten**

Suche nach Daten

*Suche alle Klassen A, die die Werte b, c und d haben*

6. **Query-Struktur**

Veränderung des Aufbaus der Query.

a) **Reihenfolge**i. **Triple Reihenfolge ungeordnet**

Die Reihenfolge der Triple ist zufällig

*Suche alle Namen, von allen Personen über 50* 8.42

ii. **Triple Reihenfolge geordnet**

Die Reihenfolge der Triple ist sortiert von der kleinsten Ergebnismenge zur größten Ergebnismenge.

*Suche alle Personen über 50 und gib die Namen aus* 8.42

b) **Invers**i. **Vorwärts Query**

Abfrage gemäß der Struktur der Daten

*Suche von allen Personen, die „Sara“ heißen die Reviews* 8.29

ii. **Rückwärts Query**

Inverse Abfrage entgegen der Datenstruktur.

*Suche alle Reviews von Personen, die „Sara“ heißen.* 8.29

### c) Subselect

#### i. Kein Subselect

Nach allen Daten auf einmal fragen

*Gib von allen Personen, die „Sara“ heißen das Alter an* 8.49

#### ii. Subselect

Subqueries verwenden, um das Datenset klein zu halten.

*Gib von allen Personen die „Sara“ heißen **5 zurück** und gib von denen das Alter an* 8.49

### Paths

#### 1. Property Paths nicht angeben

Alle Triple angeben, welche in der Reihenfolge auftreten

*Suche nach allen Personen, die „Kevin“ heißen, die Reviews haben und nach der Beschreibung der Reviews.*

#### 2. Property Paths angeben

Property Paths verwenden

*Gib von allen Personen, die „Kevin“ heißen **die Beschreibung der Reviews** zurück*

## 4.2 Vergleich von Query-Designs

Für die nachfolgenden Queries wird der Header jeweils vor die Query gesetzt, außer ein Header ist explizit angegeben. Variable Produktnamen, welche während der Ausführung zufällig ausgefüllt werden, werden durch zwei Prozentzeichen und einen Variablennamen in der Mitte dargestellt, wie in 8.23 zu sehen. Hier wurden beispielsweise zwei Query-Schemas gezeigt 8.2. Die restlichen Query-Schemen befinden sich im Anhang 2. Listing 8.1 ist ein Header, welcher vor jedes der anderen Listings geschrieben werden kann, sodass diese Werte zurückgeben. Wie in Listing 8.2 beispielhaft zu sehen, besteht ein Listing aus zwei (in den Fällen Select, Minus und Distinct aus 3) Designs, welche jeweils voneinander mit einem „vs“ abgetrennt sind.

```

1
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3 PREFIX bsbm: <http://www4.wiwiiss.fu-berlin.de/bizer/bsbm/v01/
  vocabulary/>
4 PREFIX dc: <http://purl.org/dc/elements/1.1/>
5 PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

```

```
6 PREFIX rev: <http://purl.org/stuff/rev#>
7
8 select ?product ?review ?date ?feature ?vendor ?number ?
   textual5 ?person where {
```

Listing 4.2: Header

```
1   ?feature rdf:type bsbm:ProductFeature.
2
3 VS
4
5   ?vendor rdf:type bsbm:Vendor.
```

Listing 4.3: Anzahl Ergebnisse

```
1   ?review bsbm:rating1 ?rating.
2   ?review dc:date ?date.
3   filter (?rating >= %x% && ?date < %currentDate%)
4
5 VS
6
7   ?review bsbm:rating1 ?rating.
8   filter (?rating >= %rating%)
9   ?review dc:date ?date.
10  filter (?date < %currentDate%)
```

Listing 4.4: Filter Größe

```
1   ?review bsbm:rating1 ?rating.
2   filter(?rating < %rating%)
3
4
5 VS
6
7   ?review dc:title ?title.
8   filter(regex(?title, "%word%"))
```

Listing 4.5: Filter Nummer vs String

### 4.3 Nach Optimiererausgabe

Wie in Unterabschnitt 1.3.1. Aufbau beschrieben, wird üblicherweise die Query vor der eigentlichen Ausführung optimiert. Dies liefert einige Erkenntnisse über das mögliche Performance-Verhalten der Queries. So kann man bei machen Query-Designs direkt sehen, wie diese einen Effekt auf die Optimierung haben, und wie man die Query noch weiter designen könnte, um die Query vermutlich schneller zu machen. Um die Query lokal zu optimieren und die optimierte Query ausgeben zu lassen, wurde hier der Optimierer von Apache Jena ARQ genommen. ARQ ist ein SPARQL Prozessor für Jena, welcher Queries auf zwei Arten optimieren kann: Statisch und Dynamisch. Bei der statischen Optimierung wird die Query unabhängig von irgendwelchen Daten optimiert, indem die Query in SPARQL-Algebra übersetzt wird und die statischen Regeln (wie in Unterabschnitt 1.3.1. Aufbau beschrieben) darauf angewandt werden. Bei der dynamischen Optimierung wird die Query während der Ausführung optimiert. Hier werden Optimierungen ausgeführt, welche die Ergebnismenge so schnell wie möglich einschränken, wie zum Beispiel eine Änderung der Reihenfolge der Triple.

Mit dem Optimierer wurden dann die verschiedenen Query-Designs optimiert und anhand der optimierten Queries wurden dann die optisch schnellsten Queries ermittelt. Eine Beispiel-ausgabe ist in 4.7 dargestellt. Diese Ausgabe zeigt beispielhaft, wie trotz der verschiedenen Designs von FilterGr, der Optimierer die Queries gleich optimiert.

Der Optimierer übersetzt die Queries zuerst in SPARQL-Algebra und optimiert diese dann dort statisch mithilfe der gegebenen Regeln. Mithilfe dieser optimierten Ausgaben wurde überprüft, inwiefern die statische Optimierung mit den verschiedenen Designs arbeitet, und ob es grobe Unterschiede gibt, welche auf einen Performanceunterschied deuten könnten.

```
1 (prefix ((xsd: <http://www.w3.org/2001/XMLSchema#>)
2         (bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
3           v01/vocabulary/>)
4         (dc: <http://purl.org/dc/elements/1.1/>))
5 (project (?review ?rating2)
6   (filter (< ?rating2 8)
7     (sequence
8       (filter (>= ?rating1 4)
9         (bgp (triple ?review bsbm:rating1 ?rating1)))
10        (bgp (triple ?review bsbm:rating2 ?rating2))))))
```

Listing 4.6: ARQ-FilterGr

```
1 (prefix ((xsd: <http://www.w3.org/2001/XMLSchema#>)
2         (bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
3           v01/vocabulary/>)
4         (dc: <http://purl.org/dc/elements/1.1/>))
5 (project (?review ?rating2)
6   (filter (< ?rating2 7)
7     (sequence
8       (filter (>= ?rating1 3)
9         (bgp (triple ?review bsbm:rating1 ?rating1)))
          (bgp (triple ?review bsbm:rating2 ?rating2))))))
```

Listing 4.7: ARQ-FilterGr2

Im Folgenden wurde aufgelistet, bei welchen Designs, mithilfe der Optimiererausgabe ein Performanceunterschied erwartet wurde und welches der Designs als performanter vermutet wurde. Diese Erwartungen wurde mithilfe von den statisch optimierten Queries erstellt. Ein Performanceunterschied wurde erwartet, sobald sich die optimierten Queries unterschieden:

**Erwartungen:**

**Filternum:** Hier wird erwartet, dass das Design mit dem Zahlenfilter schneller ist. Diese Vermutung entsteht, da das Filtern nach einer Zahl als schneller vermutet wird, als das Filtern nach einer Volltextsuche.

**Ober:** Hier wird erwartet, dass das Design ohne die Angabe der Oberklasse schneller ist, da weniger Tripel gesucht werden müssen. 8.36

**Type:** Hier wird erwartet, dass das Design ohne die Angabe der Typen schneller ist, da weniger Tripel gesucht werden müssen. 8.51

**Union:** Es wird erwartet, dass das Design mit dem Operator OPTIONAL schneller ist als das, mit dem Operator UNION, da UNION ein Projekt mehr startet als OPTIONAL. 8.53

**Anzahl:** Der optimierte Code unterscheidet sich zwar nicht in den Operatoren, allerdings wird erwartet, dass das Design mit geringerer Rückgabe schneller ist, da weniger Daten übertragen werden müssen. 8.2

**Limit:** Hier wird erwartet, dass das Design mit dem LIMIT-Operator schneller ist, da weniger Daten zurückgegeben werden müssen. 8.31

**Regex:** Der Code unterscheidet sich zwar nicht stark, allerdings wird erwartet, dass der eingebaute STR-Operator eine bessere Performance hat. 8.40

**Reihenfolge:** Es wird erwartet, dass das Design, welcher die Daten schneller einschränkt schneller ist, da dadurch mit weniger Daten gearbeitet werden muss. 8.42

**Select:** Es wird erwartet, dass das Design, welches die Ausgabe der Daten einschränkt schneller ist, da wieder weniger Daten zurückgegeben werden müssen. 8.44

**Struktur:** Es wird erwartet, dass das Design, welches nach den Daten sucht schneller ist, da diese die Ergebnisse schneller einschränken. 8.47

**Subselect:** Hier wird erwartet, dass das Design mit dem Subselect schneller ist, da die Ergebnismenge früh stark eingeschränkt wird. 8.49

**Distinct:** Hier wird erwartet, dass das Design mit dem REDUCED-Operator schneller ist, da nicht garantiert wird, dass alle Duplikate gelöscht werden und deshalb vermutet wird, dass Distinct die Ergebnisse überprüft, während Reduced diese einfach zurück gibt. 8.20

**Minus:** Es wird erwartet, dass das Design mit der Wahl nach einem Tripel zu suchen schneller ist, da keine Mengenoperationen ausgeführt werden müssen. 8.33

**FiterPos:** Da die optimierten Ausgaben gleich sind, wird kein Unterschied in der Performance erwartet. 8.27

**FilterGr:** Da die optimierten Ausgaben gleich sind, wird kein Unterschied in der Performance erwartet. 8.23

**Invers:** Da die optimierten Ausgaben gleich sind, wird kein Unterschied in der Performance erwartet. 8.29

**Path:** Da die optimierten Ausgaben bis auf die Namensgebung der Variablen gleich ist, wird kein Unterschied in der Performance erwartet. 8.38

In der Liste wurden die Queries in zwei Gruppen aufgeteilt: in eine Gruppe, in welcher vermutet wird, dass es einen Performanceunterschied geben wird, und in eine Gruppe, in welcher vermutet wird, dass es keinen Performanceunterschied zwischen den Query-Designs geben wird.

## 5 Aufbau der Evaluation

Dieses Kapitel beschreibt den Zwischenschritt zwischen der Beantwortung der einzelnen Unterfragen und der Zusammenführung dieser Abbildung 2.1. Vorgehensweise. Es werden die möglichen Probleme und deren Lösung zusammen mit den, für die Verschmelzung der beiden Teilfragen benötigten, Komponenten erklärt.

### 5.1 Beteiligte Komponenten

In Kapitel 3 wurden vorhandene Benchmarks evaluiert. Dabei hat sich gezeigt, dass sich der BSBM für die Anwendung in dieser Arbeit am besten eignet. In Kapitel 4 wurden die Query Komponenten herausgefunden und anhand dieser einige Oberpunkte für verschiedene Designs überlegt. Anhand dieser gefundenen Oberpunkte ergaben sich dann 17 verschiedene Query-Designs, welche unterschiedlich aufgebaut waren, jedoch ein gleiches Ergebnis zurückgaben. Die bis hierhin gewonnenen Ergebnisse werden nun in diesem Kapitel zusammengeführt, um mithilfe von Benchmarks, die in Abschnitt 4.3 aufgestellten Hypothesen zur Ausführungsgeschwindigkeit zu prüfen.

Für die Ausführung dieses Tests werden die folgenden Komponenten benötigt:

1. Benchmark mit:
  - a) Driver
  - b) Query-Templates
  - c) Datensetgenerator
2. Triple-Store mit:
  - a) SPARQL-Endpoint
  - b) Optimierer

#### 5.1.1 Bereits existente Komponenten

Der BSBM ist eine der Hauptkomponenten für den Testdurchlauf. Er enthält einige der benötigten Komponenten:

Einen **Driver**, welcher für die Ausführung und Auswertung der Queries gegen einen Endpoint benötigt wird.

Einen **Datensetgenerator**, welcher für die Erstellung eines Datensets variabler Größe für den Triple-Store benötigt wird.

**Querytemplates**, welche für die Ausführung sinnvoll sind, da man so schnell verschiedene „Eingaben“ testen kann.

Die Evaluierung wurde in dieser Arbeit gegen den Fuseki Triple-Store ausgeführt. Der Aufbau ermöglicht jedoch problemlos, ohne größere Anpassung, eine Ausführung gegen weitere Triple-Stores.

Einige Komponenten mussten noch hinzugefügt werden, damit ein reibungsloses Ausführen möglich war. So wurde beispielsweise der BSBM erweitert und neue Programme geschrieben, damit der Prozess automatisiert ablaufen konnte.

Für die Evaluation wurden der BSBM-Benchmark verwendet, zusammen mit den in Kapitel 5 vorgestellten Queries. Um die Queries variabel zu halten, wurden sogenannte Query-Templates geschrieben, welche durch den Benchmark mit zufällig gewählten Werten gefüllt werden. Ein Query-Template ist eine normale Query, jedoch wird eine Variable eingeführt, welche während der Ausführungsphase mit Zufallswerten gefüllt wird. Eine Variable wird definiert mithilfe eines Prozentzeichens jeweils vor und nach dem Variablennamen.

```
1 select ?product where {  
2   ?product bsbm:type \%ProductTypeVariable\%.  
3 }  
4
```

Listing 5.1: Header

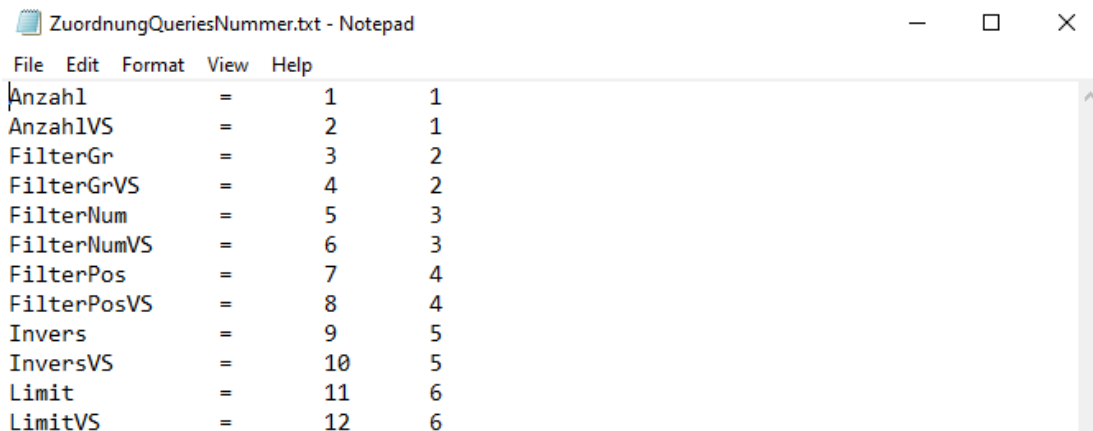
Um die Testausführung zu automatisieren, wurde eine .bat Datei entwickelt, die alle benötigten Komponenten für die Testausführung vorbereitet und den eigentlichen Benchmark startet. Das Programm akzeptiert mehrere Kommandozeilenparameter. Diese sind:

-endpoint <endpoint:url>: erwartet eine URL mit einem SPARQL-Endpoint.

-name <Queries.txt>: Erwartet einen Pfad zu einer .txt-Datei, welche die zu prüfenden Queries nummeriert und in nummerierte Gruppen einteilt 5.1.

-dir <QueryTestRun>: Erwartet einen Pfad zu einem Ordner, in welchen dann die Messergebnisse geschrieben werden.





Anzahl	=	1	1
AnzahlVS	=	2	1
FilterGr	=	3	2
FilterGrVS	=	4	2
FilterNum	=	5	3
FilterNumVS	=	6	3
FilterPos	=	7	4
FilterPosVS	=	8	4
Invers	=	9	5
InversVS	=	10	5
Limit	=	11	6
LimitVS	=	12	6

Abbildung 5.1: Gruppeneinteilung

-query <queries>: Erwartet einen Pfad zu einem Ordner, in welchem die zu prüfenden Queries mitsamt der sparql.txt-Datei liegen.

-help: Beschreibt die verschiedenen Kommandozeilenparameter und deren Verwendung.

## 5.2 Integration des Berlin Benchmarks

Für die Arbeit wurden der Datensetgenerator und der Testdriver des BSBM verwendet. Mit dem Datensetgenerator wurden viele verschiedene Datensets erzeugt, welche dann in WebGenesis und Fuseki hochgeladen wurden. Es hat sich schnell gezeigt, dass, abhängig von den gewählten Startparametern des Triple-Stores (bspw. der zur Verfügung stehende HeapSpace), nur eine begrenzte Anzahl an Tripeln geladen werden konnten. Die Datensetgröße wurde so gewählt, dass die Daten noch ohne Fehler in den TripleStore geladen werden konnten. Ohne Fehler bedeutet hierbei, dass das Datenset für eine gegebene Heap-Größe ohne „OutOfMemory Exception“ oder „GC Overhead Limit Exception“ hochlädt, bzw. nicht abbricht oder abstürzt.

## 5.3 Weiterentwicklung des Berlin Benchmarks

Für die Bachelorarbeit musste der Berlin Benchmark an einigen Punkten erweitert werden. Als erstes wurde der TestDriver des Berlin SPARQL Benchmark leicht erweitert. Dies wird benötigt, um eine weitere Variable bei den Query-Templates verwenden zu können. Eine weitere Änderung war die zusätzliche Ausgabe der Messwerte nach jedem Durchlauf einer Query und das Schreiben der gesammelten Messdaten für alle Queries in eine Datei. Der Benchmark akzeptiert jetzt zwei neue Kommandozeilenparameter: -dof und -sepm. -dof benötigt einen

Pfad zu einer Excel-Datei, in welche für jede Iteration eines Querymixes die benötigte Zeit geschrieben wird. -sepm benötigt ebenfalls einen Pfad zu einer Excel-Datei, in welche dann nach der gesamten Ausführung alle berechneten Werte geschrieben werden.

Eine weitere Änderung des Benchmarks war, dass dieser nun auch eine zufällige ReviewURI in einer Query-Template verwenden kann.

Die letzte Änderung betraf die Queries. Für diese Arbeit mussten die Queries angepasst und neue hinzugefügt werden. Die neuen Queries wurden dann wie in Kapitel 5.2 beschrieben gebaut.

```
java -cp Benc/V8/* benchmark.testdriver.TestDriver -w 30 -  
    runs 150 -ucf querie/sparql.txt -sepm RunResult/  
    GeneralRunMix.xlsx -dof RunResult/Query1/Anzahl.xlsx -o  
    RunResult/Query1/Anzahl.xml <SPARQL-Endpoint-URL>
```

Listing 5.2: Beispiel Kommando

## 5.4 Systemsetup

Für die Ausführung wird ein Fuseki-Server über Docker verwendet. Docker läuft auf einem Laptop mit Windows 10, einer Intel i7 CPU und mit 16GB RAM.

### 5.4.1 Heap-Space

Der Heap-Space ist eine Art Arbeitsspeicher, welchen jeder Prozess „privat“ für sich hat. Der Heap-Space kann für jeden Prozess beliebig groß gesetzt werden. Der Heap-Space gibt dann für das Programm an, wie viel Arbeitsspeicher dieser verwenden darf. Ebenso wie der Arbeitsspeicher kann der Heap-Space auch Daten auslagern. Auch hier muss der Garbage Collector die Daten erst löschen, bevor wieder an dieser Stelle etwas gespeichert werden kann. Dementsprechend kann wie im Arbeitsspeicher auch an dieser Stelle ein „OutOfMemoryException“ auftreten.

## 5.5 Ausführungsmethodik

Während der Ausführung wurde festgestellt, dass es einen Zusammenhang zwischen der Performance der Queries und des Java Heap Spaces gab. Hieraus ergab sich dann der Java Heap Space als weiterer Parameter, welchen man pro Ausführung anpassen und somit die Ausführungszeit verändern kann. In diesem Setup wurden verschiedene Datensetgrößen mit

jeweils einer minimalen HeapSpace-Größe und einer sehr großen verglichen. Dabei ist die minimale Heap-Space-Größe die kleinste Größe, bei denen noch alle Triple ohne Fehler in den Triple-Store geladen werden können. Bei der großen Heap-Space-Größe wurde die Heap-Space-Größe so gewählt, dass der Arbeitsspeicher, welcher auf dem Computer verfügbar war, ausgereizt wurde.

Die Performancemessung der Queries wurde in verschiedenen Phasen durchgeführt. Zuerst begann die Aufwärmphase, darauf folgend wurden dann gewertete Queries gegen den Triple-Store Endpoint laufen gelassen und gemessen. Die Messung wurde für jede Query wiederholt und separat gespeichert. Um die Unterschiede herauszufinden, wurden für die Werte QPS und Average Result die Werte für die verschiedenen Designs verglichen. Für die Designs, welche stark unterschiedliche Average Result's hatten, wurde die Query insofern verändert, dass der Unterschied minimal wurde und die Query einen möglichst geringen Komplexitätsanstieg hatte.



## 6 Ausführung

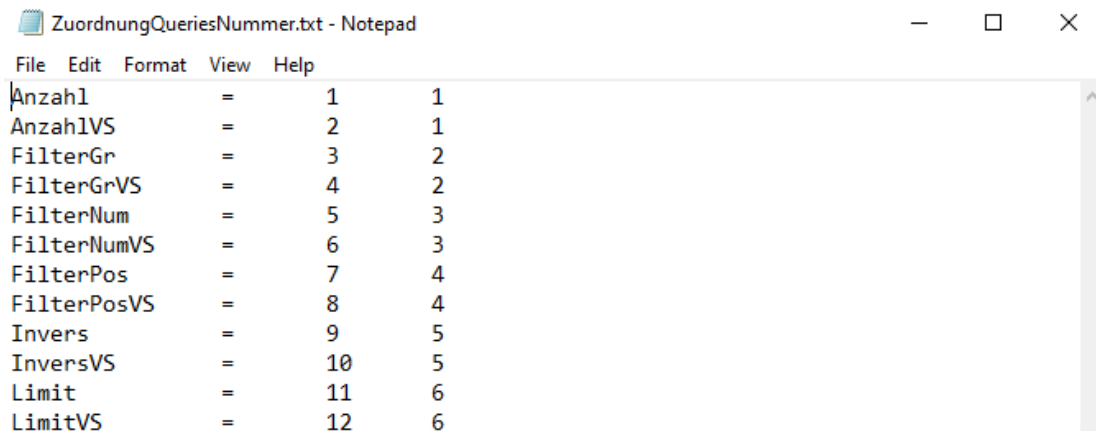
Die Ausführung des Benchmarks gegen den Tripel-Store wurde in mehreren Schritten durchgeführt. Eine Ausführungsgraph wird in Abbildung 6.5 dargestellt.

Vor der Ausführung des Benchmarks müssen in den Tripel-Store Daten geladen werden und verschiedene Query-Designs erstellt werden 6.1. Die Namensgebung muss aus dem Wort „query“ und einer Zahl bestehen. In der Datei „query1“ wird die Query mit eventuellen Variablen gespeichert. In der Datei „query1desc“ werden dann die verwendeten Variablen erklärt, damit der Benchmark die Variablen durch passende Werte ersetzen kann. Die Datei „query1valid“ ist leer und wird nur benötigt, falls bereits bekannt ist, welche Daten zurückgegeben werden sollten und diese auf Korrektheit überprüft werden sollen. Die Datei „query1Valued“ wird benötigt, falls Variablen in der Query verwendet wurden. In dieser Datei werden die Variablen durch Werte ersetzt, da die Datei nicht für die Ausführung gegen den Tripel-Store verwendet wird, sondern für die statische Optimierung verwendet wird.

Eine weitere Textdatei wird benötigt, um dem verwendeten Skript zu sagen, welche Queries getestet werden sollen. Die Datei besteht aus einem Namen für die Query, einer Zahl, welche sich auf die Zahl im Namen der Query-Datei bezieht und einer Zahl, welche die Query in Gruppen teilt, damit zwei Queries leichter miteinander verglichen werden können.

query1.txt	15.10.2019 12:20	Text Document	1 KB
query1desc.txt	16.10.2019 10:35	Text Document	1 KB
query1valid.txt	02.10.2019 12:45	Text Document	0 KB
query1Valued.txt	15.10.2019 12:20	Text Document	1 KB
query2.txt	29.10.2019 12:38	Text Document	1 KB
query2desc.txt	16.10.2019 10:35	Text Document	1 KB
query2valid.txt	02.10.2019 12:45	Text Document	0 KB
query2Valued.txt	29.10.2019 12:38	Text Document	1 KB
query3.txt	25.10.2019 11:52	Text Document	1 KB
query3desc.txt	25.10.2019 11:52	Text Document	1 KB
query3valid.txt	02.10.2019 12:45	Text Document	0 KB
query3Valued.txt	06.11.2019 09:35	Text Document	1 KB

Abbildung 6.1: Design des Query Speicherns



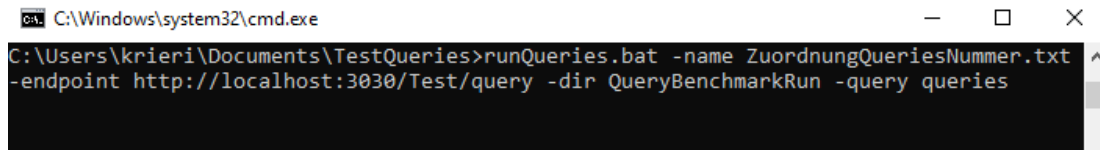
File	Edit	Format	View	Help
Anzahl	=	1	1	
AnzahlVS	=	2	1	
FilterGr	=	3	2	
FilterGrVS	=	4	2	
FilterNum	=	5	3	
FilterNumVS	=	6	3	
FilterPos	=	7	4	
FilterPosVS	=	8	4	
Invers	=	9	5	
InversVS	=	10	5	
Limit	=	11	6	
LimitVS	=	12	6	

Abbildung 6.2: Design der Datei für das Ausführen der Queries

Sind alle Daten in den Tripel-Store geladen und die Dateien sind alle erstellt kann man mit der Ausführung beginnen. Der Benutzer übergibt dem Skript `runQueries.bat` Kommandozeilenparameter, welche auf die Dateien und den Endpoint verweisen. Werden diese nicht gegeben, so werden vordefinierte Standardwerte verwendet. Die verschiedenen Befehle und die Standardwerte können mit `runQueries.bat -help` angezeigt werden. 6.3

Wird das Skript ausgeführt, so liest es aus der Datei 6.2 eine Query. Diese Query wird an den Benchmark geschickt, welcher die Query 150 mal gegen den Tripel-Store schickt und die Laufzeit misst. Die gemessenen Laufzeiten und die berechneten Auswertungen werden dann vom Benchmark an dem vom Skript übergebenen Ort gespeichert. Ist der Benchmark fertig, so holt sich das Skript die nächste Query und wiederholt diese Ausführung bis keine weiteren Queries mehr in der Datei zu finden sind. 6.4

Diese Ausführungsmethodik wurde mit verschiedenen Heap-Space Größen und Tripel Mengen wiederholt. Die Heap-Space Größe wurde für jede Tripel-Menge mehrmals unterschiedlich gewählt. So wurde die Heap-Space Größe einmal minimal gewählt, sodass die Tripel gerade so ohne Fehler in den Tripel-Store geladen werden konnten. Dies wurde durch einfaches Ausprobieren der Heap-Space Größe erreicht. Hierbei wurde mit einem sehr kleinem Heap-Space angefangen und solange Fehler geworfen wurden, wurde dieser schrittweise leicht erhöht. Nach dem Ausführen des Skriptes mit dem minimalen Heap-Space, wurde der Heap-Space stark vergrößert um eine Art des maximalen Heap-Spaces zu erreichen. Hierbei wurde der Heap-Space in dieser Durchführung um den verbleibenden Arbeitsspeicher im System erhöht.



```
C:\Windows\system32\cmd.exe
C:\Users\krieri\Documents\TestQueries>runQueries.bat -name ZuordnungQueriesNummer.txt
-endpoint http://localhost:3030/Test/query -dir QueryBenchmarkRun -query queries
```

Abbildung 6.3: Kommandozeilen Argument

```
143: 13.74ms, total: 15ms
144: 4.58ms, total: 5ms
145: 3.97ms, total: 5ms
146: 4.52ms, total: 5ms
147: 3.68ms, total: 5ms
148: 4187.75ms, total: 4189ms
149: 4.55ms, total: 5ms

Scale factor:          5000
Number of warmup runs: 30
Seed:                  808080
Number of query mix runs (without warmups): 150 times
min/max Querymix runtime: 0.0031s / 6.2230s
Total runtime:         107.296 seconds
QMpH:                  5032.81 query mixes per hour
CQET:                  0.71531 seconds average runtime of query mix
CQET (geom.):          0.01352 seconds geometric mean runtime of query mix

Metrics for Query:     2
Count:                 150 times executed in whole run
AQET:                  0.715306 seconds (arithmetic mean)
AQET(geom.):           0.013524 seconds (geometric mean)
QPS:                   1.40 Queries per second
minQET/maxQET:         0.00306300s / 6.22298900s
Average result count:  48.00
min/max result count:  48 / 48
Number of timeouts:    0
```

Abbildung 6.4: Während der Ausführung

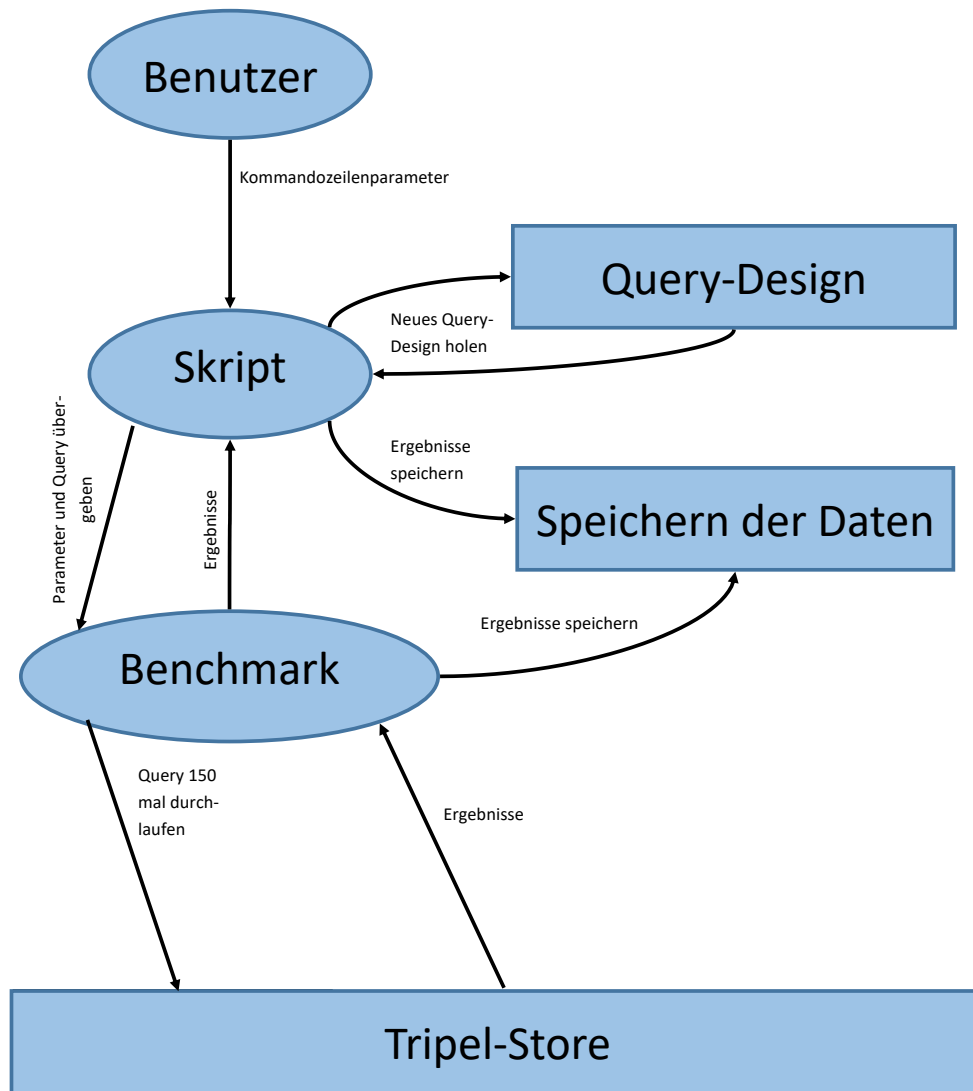


Abbildung 6.5: Ausführung



## 7 Evaluation

### 7.1 Präsentation der Messergebnisse

Wie in Kapitel Abbildung 6.5. Ausführung beschrieben, wurde jedes Query-Design mehrmals mit unterschiedlichen Parametern ausgeführt. So gab es 5 verschiedene Ausführungen. Drei Ausführungen mit jeweils 1,8 Mio. Tripeln und den Heap-Space-Größen 1,1 GB, 4 GB und 7 GB und zwei Ausführungen mit jeweils 4 Mio. Tripeln und den Heap-Space-Größen 2,25 GB und 7 GB.

An dieser Stelle werden nur die zwei errechneten Graphen mit jeweils 4 Mio. Tripeln dargestellt. Die weiteren Graphen können im Anhang gefunden werden. Alle Graphiken wurden in zwei Teile aufgeteilt, damit die Graphen größer dargestellt und Unterschiede leichter zu erkennen sind.

7.1 und 7.2 zeigen den Durchlauf von 4 Mio. Tripeln mit einer Heap-Space-Größe von 2,25 GB.

7.3 und 7.4 zeigen den Durchlauf von 4 Mio. Tripeln mit einer Heap-Space-Größe von 7GB.

Die Graphiken zeigen jeweils gruppierte Balken, welche die unterschiedlichen Queries per Second der Designs der Query zeigen. Dies stellt die unterschiedlichen Queries gegenüber und erlaubt so den Einfluss der Query-Elemente abzuleiten. Die Balken zeigen an, wie oft die spezielle Anfrage pro Sekunde ausgeführt werden kann. Hierbei wird mit der durchschnittlichen Laufzeit gerechnet. Das bedeutet also, dass mit steigendem Balken die benötigte Laufzeit pro Query-Anfrage sinkt.

Ein wichtiger Faktor, der beachtet werden muss, ist die Skalierung der unterschiedlichen Graphiken. So steigt die Skalierung von 7.1 in 0,2er Schritten, während die Skalierung von 7.3 in 20er Schritten steigt.

Die verwendeten Skripte und die Messergebnisse sind in Github mit der CC-BY-SA-4.0 Lizenz hochgeladen. <https://github.com/DrKingSchultz2/LAOSA>

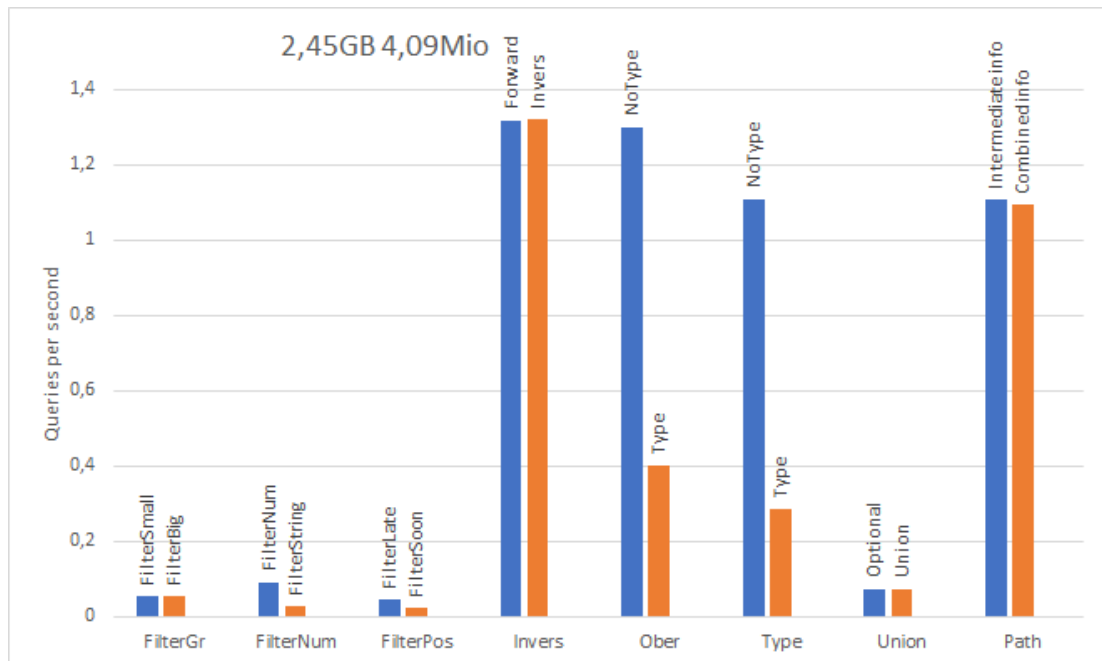


Abbildung 7.1: 2,25GB und 4,09Mio Tripel

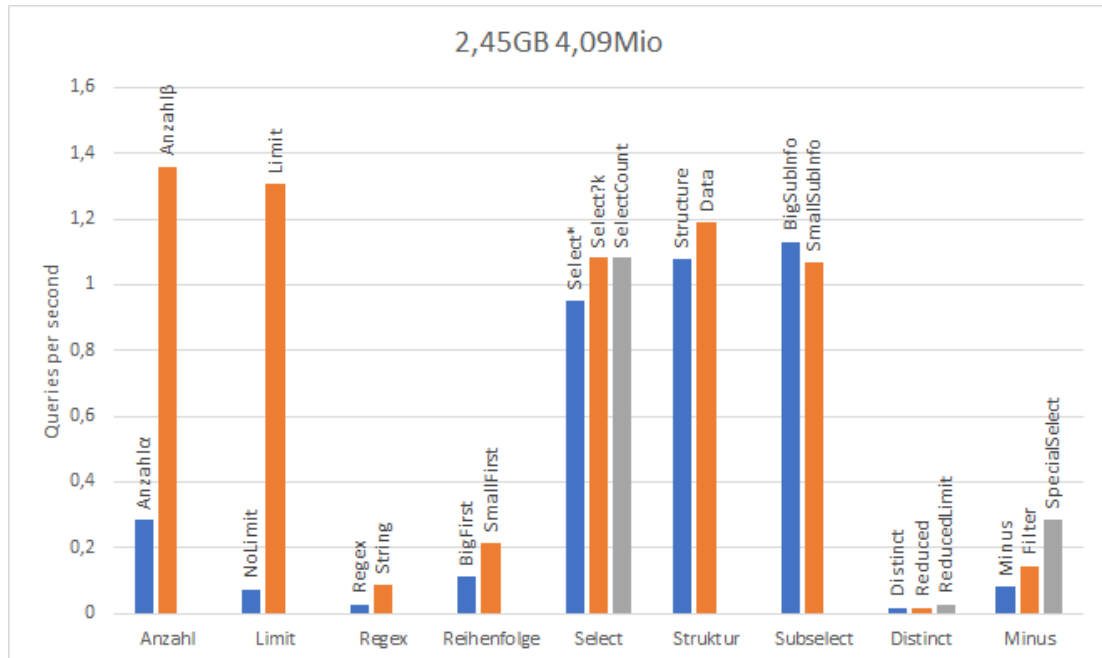


Abbildung 7.2: 2,25GB und 4,09Mio Tripel Design 2

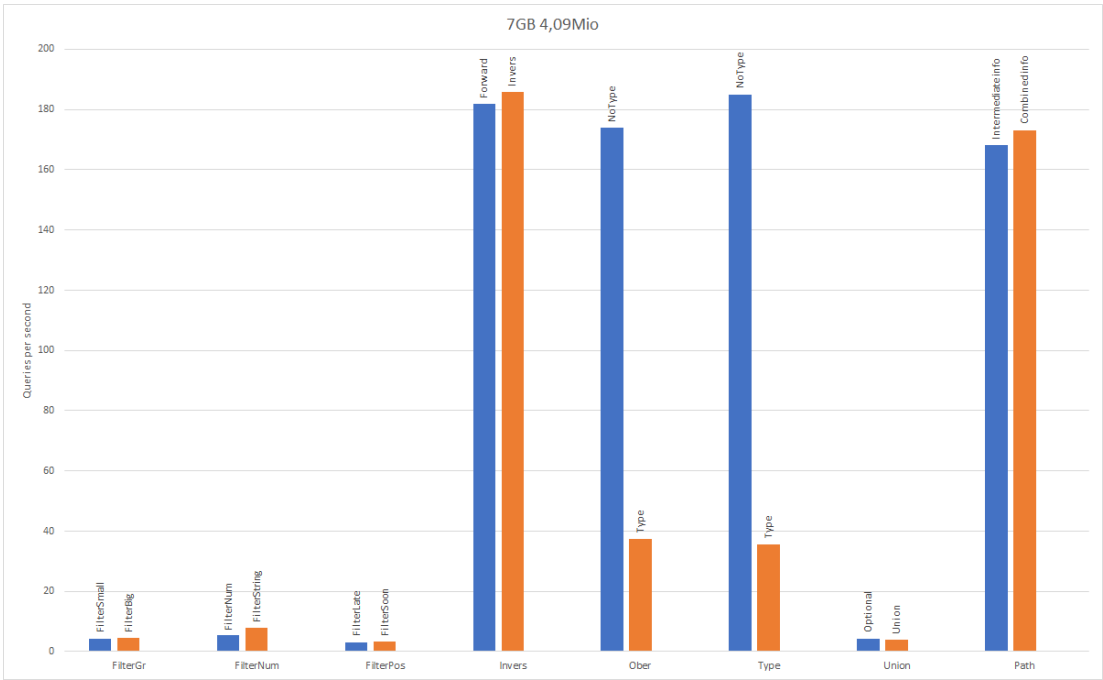


Abbildung 7.3: 7GB und 4,09Mio Tripel

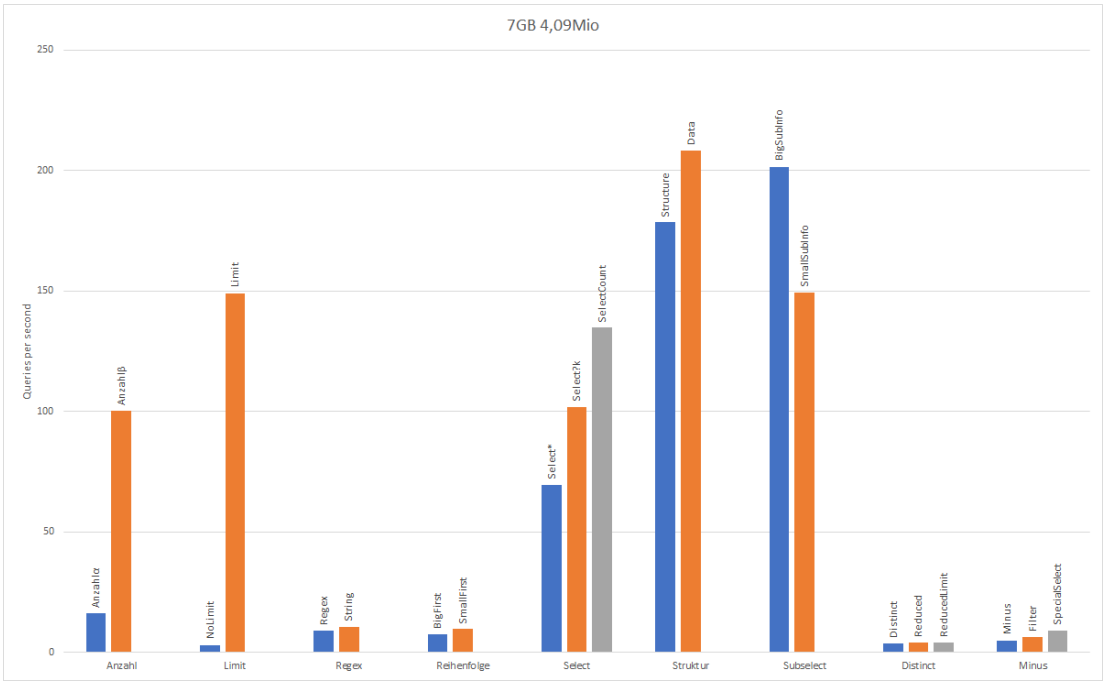


Abbildung 7.4: 7GB und 4,09Mio Tripel Design 2

## 7.2 Diskussion der Messergebnisse

### 7.2.1 Einfluss der Query Elemente

Im Folgenden wurden die Messergebnisse auf den Einfluss der Query-Elemente analysiert. Wie man in 7.1 sieht, zeigen einige Balken sehr deutlich, welches Design eine bessere Performance hatte. Allerdings kann es durch die Skalierung kommen, dass trotz einer Optimierung die Balken nah aufeinander liegen. 7.1 zeigt den Graphen des Benchmarks mit 4 Mio. Tripeln und einer Heap-Space-Größe von 2,45GB. Die Schrittgröße des Graphen ist mit 0,2 Queries per Second (QPS) sehr klein. Dadurch sind die Optimierungen bei FilterNum und FilterPos, bei welchen jeweils das 1. Design das schnellere ist, nicht so deutlich, wie bei Ober und Type, bei welchen ebenfalls das erste Design das schnellere ist.

Im Gegensatz dazu kann man sehen, dass sich die Designs der Komponenten FilterGr, FilterPos, Invers, Union und Path nur sehr gering unterscheiden.

Im fortgesetzten Graphen 7.2 ist die Schrittgröße ebenfalls 0,2 QPS. In diesem Graphen haben alle Designs ein Teil-Design, welches mindestens 10% schneller ist als das andere. Hier sind bei Anzahl, Limit, Regex, Reihenfolge und Struktur jeweils das zweite Design das schnellere. Nur bei Subselect ist das erste Design das schnellere. Bei Select, Distinct und Minus gibt es drei Ansätze für das Design. Hier sind bei Select das zweite und dritte Design am schnellsten, bei Distinct ist nur das dritte Design am schnellsten und bei Minus ist das zweite Design schneller als das erste und das dritte ist am schnellsten.

Gesammelt lässt sich für einen Triple-Store mit einem kleinen Heap-Space und vielen Tripeln sagen, dass die Performance mit geringerer Rückgabe, wie bei Anzahl und Limit, und einer schnellen Einschränkung der Suchergebnisse, wie bei Reihenfolge, steigt. Ebenfalls ist es schneller, nach einer Zahl zu filtern, als nach einem String.

Der Graph 7.3 stellt den Benchmark über den Triple-Store mit 4 Mio. Tripeln und 7GB Heap-Space dar. Hier ist die Schrittgröße der Skalierung bei 20 QPS. Hier haben sich einige Änderungen im Vergleich zu 7.1 ergeben. So ist nun das Filtern nach einem String schneller als nach einer Zahl und FilterPos zeigt keine Leistungsunterschiede. Ansonsten zeigen sich keine wesentlichen Unterschiede, abgesehen von der Skalierung, zwischen den beiden Graphen.

Zwischen den Graphen 7.2 und 7.4 kann man ebenfalls Unterschiede sehen. So sieht man bei allen Queries einen starken Performanceunterschied zwischen den verschiedenen Designs. So hat sich der Unterschied zwischen den Designs von Regex, Reihenfolge, Minus und Distinct stark verkleinert, während sich bei den restlichen Designs die Performanceunterschiede vergrößert haben. Bei Select und Distinct gab es einen Unterschied zwischen den performanteren TeilDesigns im Vergleich zu dem Tripel-Store mit dem kleinen Heap-Space. Bei Distinct haben

sich alle Designs so angenähert, dass kein Leistungsunterschied von 10% mehr erreicht wurde und es somit nach der Definition auch keine Optimierung gab. Bei Select gab es einen starken Unterschied zwischen den einzelnen Teildesigns, wodurch das performanteste Design das dritte war.

Zusammengefasst lässt sich sagen, dass mit einem kleineren Rückgabewert, bei allen Heap-Space-Größen die Performance trotzdem wächst, während sich kleinere Unterschiede wie Filter oder die Reihenfolge der Tripel mit größerem Heap-Space angleichen.

### 7.2.2 Liste optimierter Queries

Da es wegen der unterschiedlichen Skalierung bei manchen Designs äußerst schwierig ist zu sehen, ob eines der Designs eine Optimierung gegenüber der anderen aufweist, wurde eine Tabelle 7.5 eingefügt, welche für jedes Query-Design angibt, ob es eine Optimierung für diese Heap-Space und Triple Konfiguration gab. In dieser Tabelle wurde ein Haken (✓) gesetzt, falls es eine Optimierung gab und ein Kreuz (X) gesetzt, falls nicht. Gab es einen Performanceunterschied, so wurden außerdem an den Haken kleine Indexe geschrieben, welche sich auf Kapitel 4 beziehen und die schnellere Query angeben.

In Tabelle 7.5 kann man sehen, dass es für einige Schemen ein Design gibt, welches das Performanteste ist. So kann man sehen, dass bei den Schemen Anzahl und Limit jeweils das Design mit der geringeren Rückgabe das performanteste war. Bei den Schema Regex sieht man, dass in allen Fällen das Design mit den eingebauten String-Funktionen am schnellsten war. Eine Optimierung konnte auch mit allen Parametern, also der Heapspace-Größe und der Anzahl der Tripel im Tripel-Store erreicht werden, indem die Reihenfolge der Tripel mit dem einschränkendsten Tripel anfängt. Bei der Wahl der Projektion des Selects, war das Performanteste das Select, welches nur die Anzahl der Tripel zurückgegeben hat. Aber auch bei der Einschränkung der Projektion des Selects konnte immer eine Optimierung gegenüber dem Select erreicht werden, welches alle Variablen projizierte. Bei dem Schema Minus konnte durchgehend eine Optimierung beobachtet werden. Die letzten Schemen, bei denen es durch alle Parameter ein optimiertes Design gab, waren die Schemen Ober und Typ. Hier war es am performantesten die Typen und Obertypen nicht mit anzugeben, was man im Schaubild 7.4 erkennen kann. Bei den Schemen Struktur und Subselect konnte nur bei einer großen Anzahl Tripel ein Design über das andere gewählt werden. Hier war es bei der „Struktur“ am schnellsten nach einer Struktur zu suchen und bei dem Schema Subselect, war es am schnellsten keine Subselects zu verwenden und somit die Anfragemenge nicht bereits in der Query einzuschränken. Bei

Query	7GB 4Mio	2,45GB 4Mio	7GB 1,8Mio	4GB 1,8Mio	1,11GB 1,8Mio
Anzahl	✓ <sub>ii</sub>	✓ <sub>ii</sub>	✓ <sub>ii</sub>	✓ <sub>ii</sub>	✓ <sub>ii</sub>
Limit	✓ <sub>ii</sub>	✓ <sub>ii</sub>	✓ <sub>ii</sub>	✓ <sub>ii</sub>	✓ <sub>ii</sub>
Regex	✓ <sub>ii</sub>	✓ <sub>ii</sub>	✓ <sub>ii</sub>	✓ <sub>ii</sub>	✓ <sub>ii</sub>
Reihenfolge	✓ <sub>ii</sub>	✓ <sub>ii</sub>	✓ <sub>ii</sub>	✓ <sub>ii</sub>	✓ <sub>ii</sub>
Select	✓ <sub>iii</sub>	✓ <sub>ii</sub>	✓ <sub>iii</sub>	✓ <sub>iii</sub>	✓ <sub>iii</sub>
Struktur	✓ <sub>ii</sub>	✓ <sub>ii</sub>	X	X	X
Subselect	✓ <sub>i</sub>	✓ <sub>i</sub>	X	X	X
Distinct	X	✓ <sub>iii</sub>	✓ <sub>ii</sub>	✓ <sub>ii</sub>	✓ <sub>ii</sub>
Minus	✓ <sub>ii</sub>	✓ <sub>ii</sub>	✓ <sub>ii</sub>	✓ <sub>ii</sub>	✓ <sub>ii</sub>
FilterNum	✓ <sub>i</sub>	✓ <sub>ii</sub>	✓ <sub>i</sub>	✓ <sub>i</sub>	✓ <sub>ii</sub>
Ober	✓ <sub>i</sub>	✓ <sub>i</sub>	✓ <sub>i</sub>	✓ <sub>i</sub>	✓ <sub>i</sub>
Typ	✓ <sub>i</sub>	✓ <sub>i</sub>	✓ <sub>i</sub>	✓ <sub>i</sub>	✓ <sub>i</sub>
Union	X	X	✓ <sub>i</sub>	✓ <sub>i</sub>	X
FilterGr	X	X	X	X	X
FilterPos	X	✓ <sub>ii</sub>	X	X	X
Invers	X	X	X	✓ <sub>i</sub>	X
Paths	X	X	X	X	✓ <sub>ii</sub>

Abbildung 7.5: Checkbox

dem Schema FilterNum gab es ebenfalls durchgängig ein optimiertes Design. Dies wechselte allerdings mit größerem Heap-Space. So war bei geringem Heap-Space das Filtern nach Zahlen schneller, während bei großem Heap-Space das Filtern nach einem String performanter war. Bei dem Schema Union gab es nur eine Optimierung bei großem Heap-Space und wenig Tripeln. Hier war das Design mit dem Optional-Befehl schneller. Bei den restlichen Schemen FilterGr, FilterPos, Invers, Paths waren keine oder nur vereinzelt geringe Optimierungen zu sehen. Deswegen kann man bei diesen Komponenten nicht von einer generellen Optimierung gesprochen.

Als Schlussfolgerung kann gesagt werden, dass die Designs, welche die Ergebnismenge zu Beginn der Query einschränken, durchgehend am performantesten sind.

### 7.2.3 Verteilung der Daten

Während der Messung wurden auch die Rohdaten vor der Auswertung gespeichert. Anhand dieser kann man einen Boxplot erstellen, in welchem man die Verteilung der Daten auslesen kann. Dies wurde gemacht, da es oftmals „Ausreißer“ gab, welche sich stark von den generellen

Werten differenzierten. Dies kann durch einen Boxplot gut dargestellt werden.

Bei einem Boxplot werden aus allen Werte einer Wertereihe folgende Werte berechnet und angezeigt.

**Median:** 50% der Werte sind kleiner als dieser Wert. (Strich in der Box)

**Mittelwert:** Der Mittelwert aller Werte.  $\text{mittel} = \frac{\sum x}{n}$  (Kreuz)

**Unteres Quartil:** 25% der Werte sind kleiner als dieser Wert (Beginn der Box)

**Oberes Quartil:** 75% der Werte sind kleiner als dieser Wert. (Ende der Box)

**Interquartilsabstand (IQA):** Wertebereich in dem sich die mittleren 50% der Daten befinden (Ausdehnung der Box)

**Unterer Whisker:** Das Minimum ohne „Ausreißer“. Berechnet aus  $\text{unteresQuartil} - 1,5 * IQA$ .

**Oberer Whisker:** Das Maximum ohne „Ausreißer“. Berechnet aus  $\text{oberesQuartil} + 1,5 * IQA$ .

Anhand der Größe der Box kann man erkennen, wie fokussiert die Daten liegen. Sind die Daten näher aneinander, ist die Box kleiner als bei stark unterschiedlichen Daten.

Wie man in 7.6 sieht, liegen die Daten im Vergleich zu 7.8 sehr viel fokussierter. Hierbei ist der Unterschied zwischen den Größen des Heap-Spaces zu betrachten. Durch einen größeren Heap-Space liegen die Daten fokussierter, da das System die Daten in-memory speichern kann und die Daten nicht erst von dem Festplattenspeicher holen muss.

#### 7.2.4 Cacheverhalten

Während der Messungen und auch im Boxplot 7.6 konnte man sehen, wie oft auf eine langsame Query mehrere schnelle folgten. Im Boxplot kann man diese langsamen Queries als Maximalwert sehen. Um dieses Verhalten erklären zu können, wurden die Daten im Folgenden noch einmal näher betrachtet. Dies wurde durch einen Zeitgraph 7.9 über die Messdaten visualisiert. An diesem kann man deutlich ein Muster im Cacheverhalten erkennen. Nach ein paar Testläufen wurde klar, dass sich der Cache auch nach einigen Sekunden nicht invalidiert. Dies lässt vermuten, dass der Cache sich mit jeder Anfrage mehr füllt und sich komplett invalidiert, sobald er überläuft.

Dieses Phänomen wurde in dieser Bachelorarbeit nicht näher betrachtet, kann aber für ein zukünftiges Thema interessant sein.

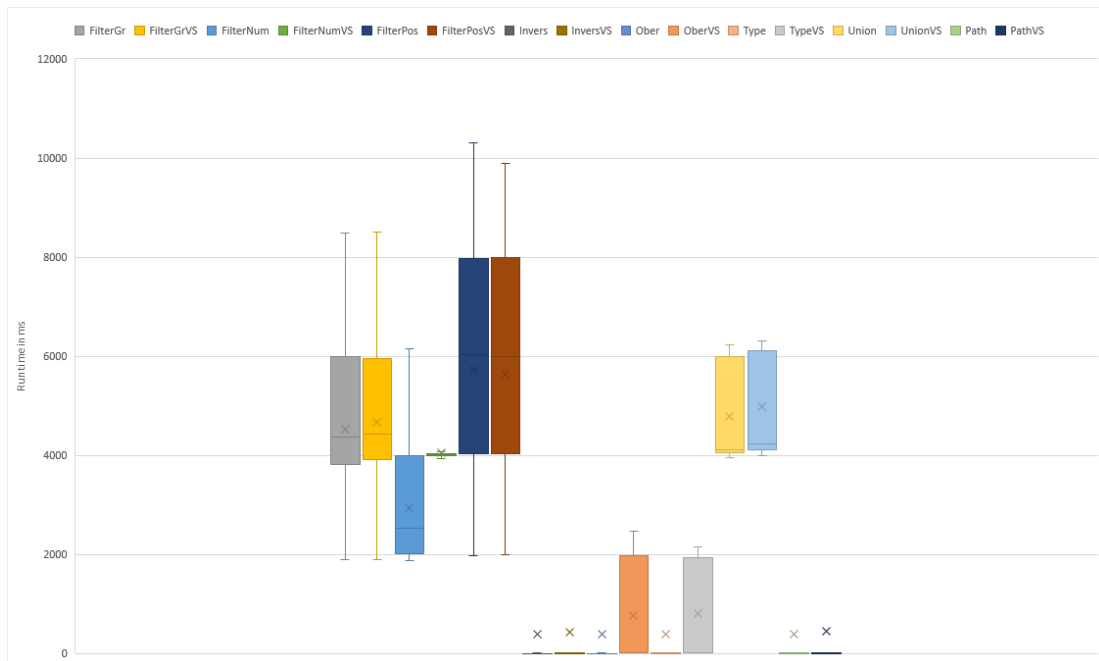


Abbildung 7.6: Boxplot 1,1GB mit 1,08Mio Tripeln

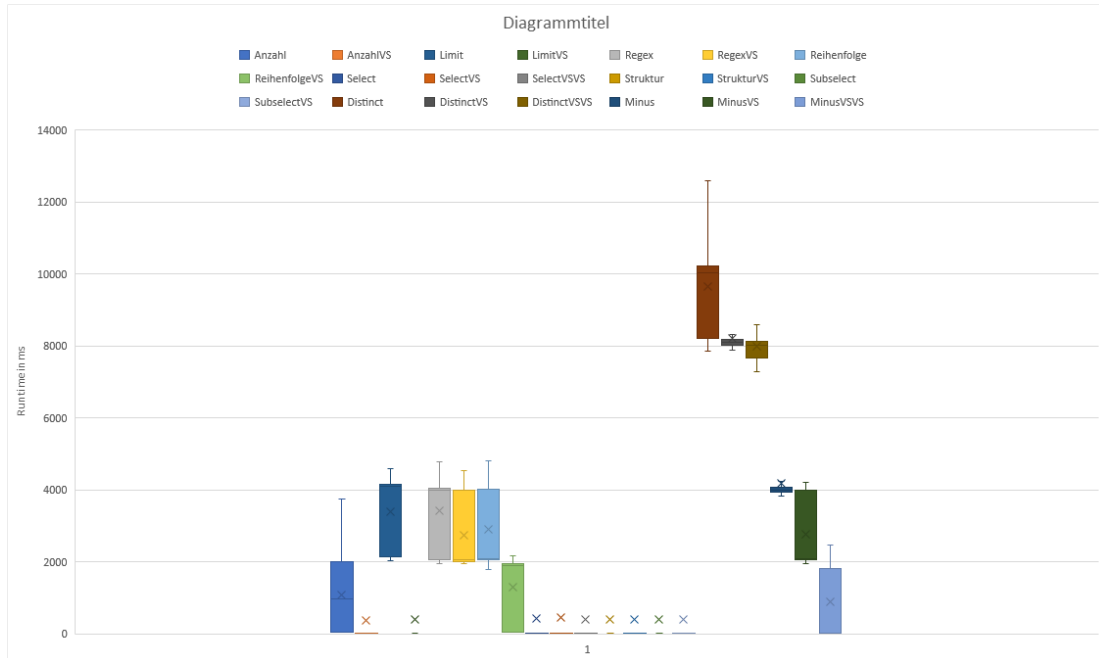


Abbildung 7.7: Boxplot 1,1GB mit 1,08Mio Tripeln Design 2



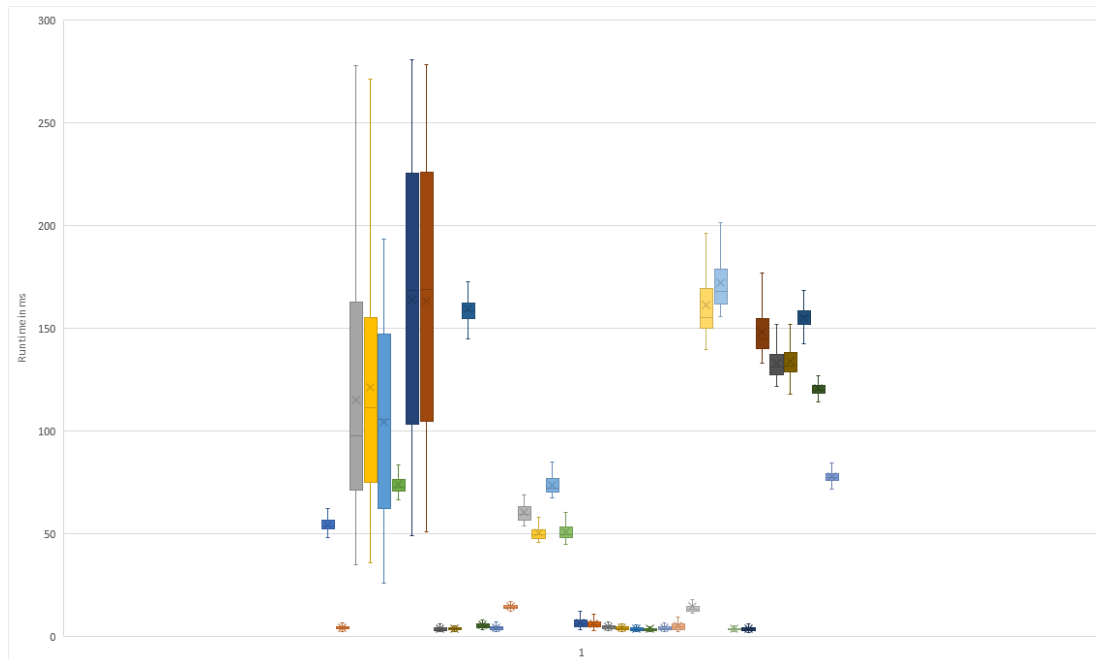


Abbildung 7.8: Boxplot 7GB mit 1,08Mio Tripeln

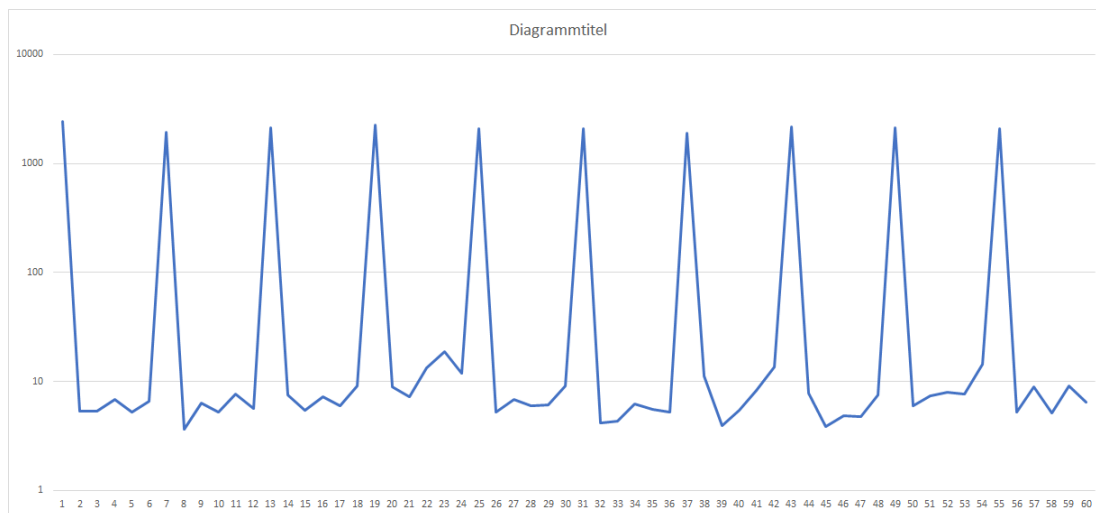


Abbildung 7.9: Cacheverhalten

### 7.3 Empfehlungen zur Formulierung der Queries

Basierend auf den Ergebnissen der Messung wurde der Einfluss der Query Elemente untersucht. Soweit es der Anwendungsfall zulässt sollte bei der Formulierung der SPARQL-Queries folgende Punkte beachtet werden.

1. **Zwischenwertmenge so klein wie möglich halten.**

Die Zwischenwertmenge sollte schon zu Beginn der Query so stark wie möglich eingeschränkt werden. Dies ist durch frühes Filtern oder einen stark einschränkenden Tripel erreichbar.

2. **Die Ausgabemenge limitieren**

Je weniger Daten zurückgegeben werden müssen, desto schneller ist die Anfrage.

3. **Unnötige Typisierung weglassen**

Wie bei dem Schema Typ schon ersichtlich, ist es am performantesten eine Typisierung weg zu lassen.

4. **Wenn möglich nach Zahlen filtern**

Das Filtern nach Zahlen ist generell performanter als nach Zeichenketten.

5. **Wenn möglich mit den STR-Funktionen arbeiten anstelle von Regex**

Die eingebauten STR-Funktionen sind schneller als Regex und haben auch ein breites Spektrum an Möglichkeiten.

6. **Nicht alles ausgeben lassen, sondern nur die benötigten Informationen**

Das Select sollte eingeschränkt werden.

7. **Nach einer Struktur und nicht nach Daten suchen**

Bei größeren Datensätzen eher nach einer Struktur filtern als nach Daten.

8. **Falls möglich Reduced anstelle von Distinct verwenden**

9. **Einen Filter anstelle des Minus-Operator verwenden**

## 8 Fazit

### 8.1 Zusammenfassung

Um diese Überlegungen zu validieren, wurden diese mithilfe eines Benchmarks überprüft. Für die Wahl des Benchmarks wurde ein Kategorisierungsschema für die unterschiedlichen Benchmarks für SPARQL-Queries entworfen. Mithilfe dieses Kategorisierungsschemas wurde der Berlin SPARQL Benchmark als geeigneter Benchmark ausgewählt. Der Benchmark wurde aufgrund des frei verfügbaren Codes, dem vorhandenen Datensetgenerator und der bereits existierenden Query-Templates. Damit der Benchmark weitere Query-Template-Variablen erkennen konnte und verschiedene Zwischenergebnisse, wie die einzelnen Query-Laufzeiten, ausgeben konnte, wurde der Benchmark für diese Arbeit leicht erweitert. Danach wurden die verschiedenen Komponenten einer SPARQL-Query untersucht und überlegt, ob ein Leistungszuwachs der Query durch eine Änderung der Komponenten erreicht werden könnte. Mithilfe dieser Überlegungen wurden 37 verschiedene Query-Designs gefunden.

Für die Automatisierung der Tests der verschiedenen Query-Designs wurde ein Skript entwickelt, welches den Benchmark für jede Query testet und die Ergebnisse zusammengefasst in eine Datei schreibt.

Nach der Zusammenführung und Ausführung dieser Komponenten konnten Graphen für die einzelnen Query-Laufzeiten erstellt werden. Mithilfe dieser Graphen war es an einigen Stellen sehr leicht eine Optimierung zu sehen. An den Stellen, an denen keine klare Optimierung zu sehen war, wurden die Rohdaten betrachtet, ob es einen Unterschied von mindestens 10% gegeben hat. Die Ergebnisse wurden ebenfalls in einer Tabelle dargestellt. Diese Tabelle besteht aus den Query-Schemas und den verschiedenen Parametern, welche man für den Triple-Store wählen konnte, nämlich der Heap-Space-Größe und der Anzahl Tripel in diesem Triple-Store. In dieser Tabelle wurde eingetragen, falls für ein Query-Schema eine Optimierung bezüglich der verschiedenen Triple-Store Parametern zu sehen war und welches der Designs für das Schema das optimalere war.

Mithilfe dieser Tabelle und den Graphen konnten einige Tipps für das Design von Queries erstellt werden.

## 8.2 Ausblick

Wie in Kapitel Abbildung 7.9. Cacheverhalten schon beschrieben, wurde bei einigen Queries und speziell bei der Wahl eines geringen Heap-Spaces ein Cacheverhalten bemerkt. Dies wurde in dieser Arbeit nur oberflächlich untersucht. Es wurde festgestellt, dass der Cache sich immer wieder invalidiert hat. Durch einen kurzen Test konnte festgehalten werden, dass dies nicht zeitlich bedingt war, sondern an der Anzahl und Größe der Anfragen liegen muss.

In Zukunft kann untersucht werden, ob durch eine Veränderung des Caches bessere Laufzeiten erreicht werden können oder wodurch diese Invalidierung entsteht.

# Anhang

## 1 Messergebnisse

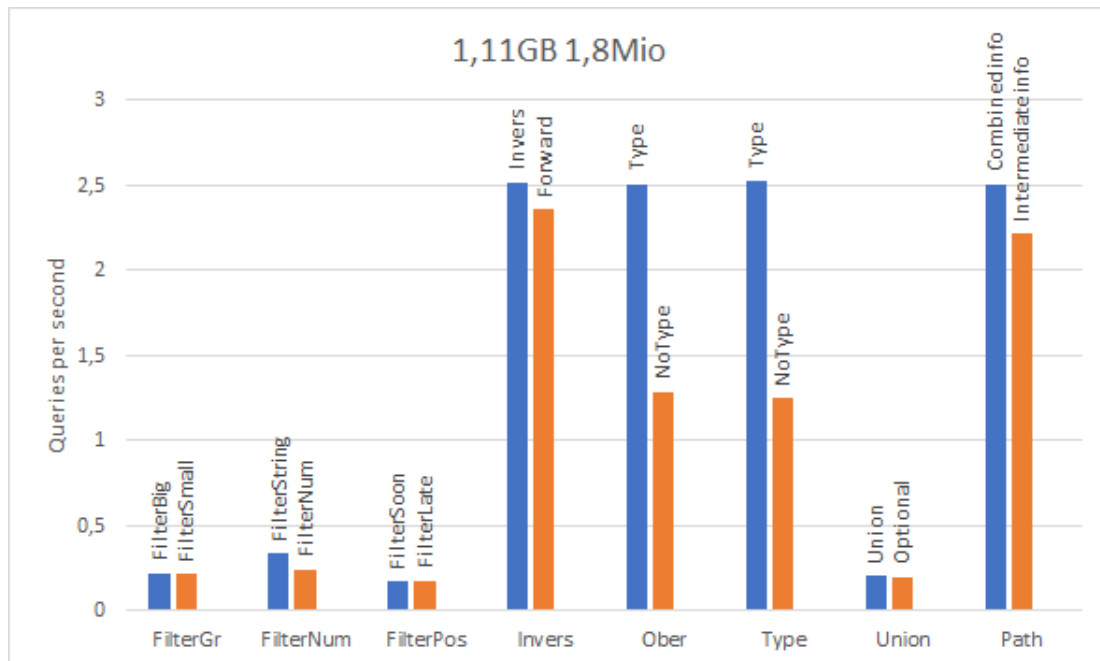


Abbildung 1: 1,11GB und 1,08Mio Tripel

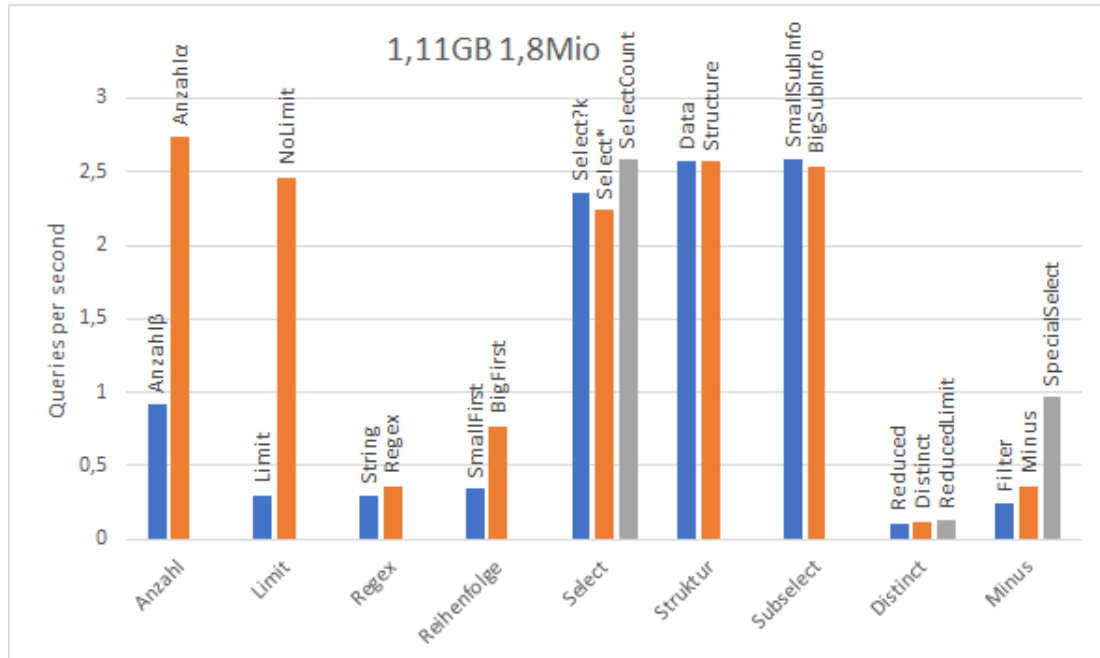


Abbildung 2: 1,11GB und 1,08Mio Tripel Design 2

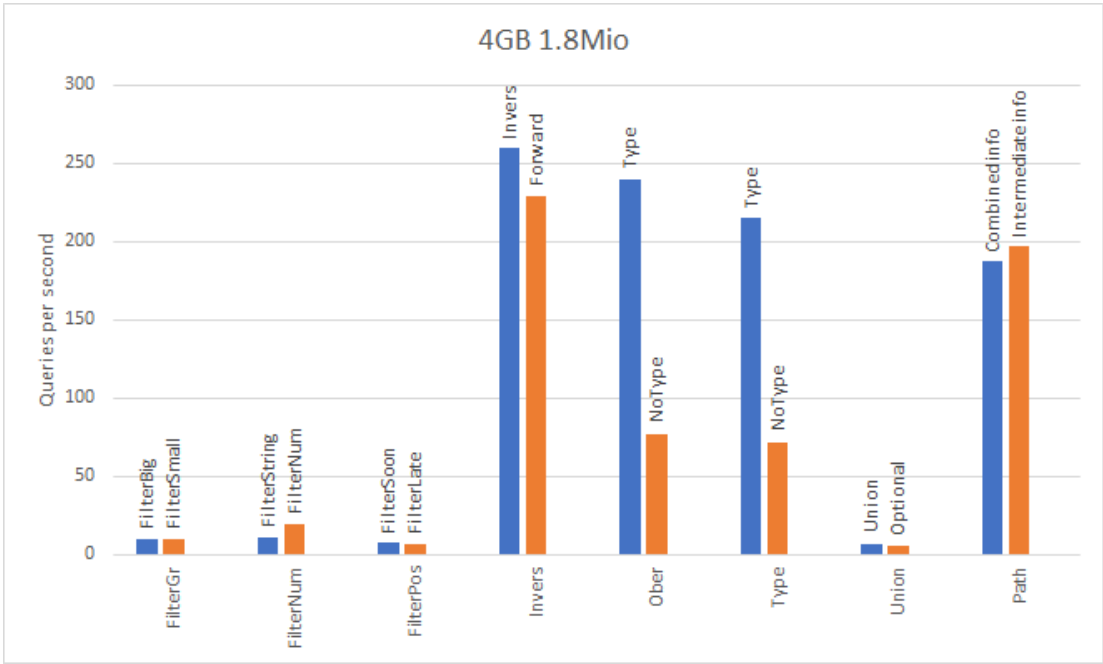


Abbildung 3: 4GB und 1,08Mio Tripel

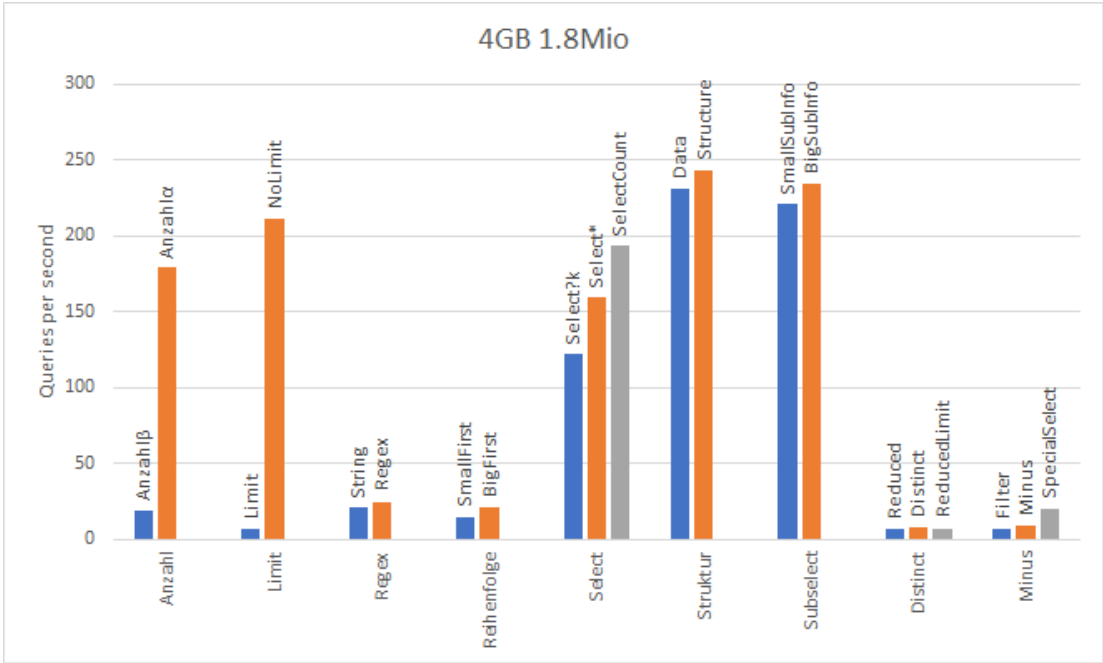


Abbildung 4: 4GB und 1,08Mio Tripel Design 2

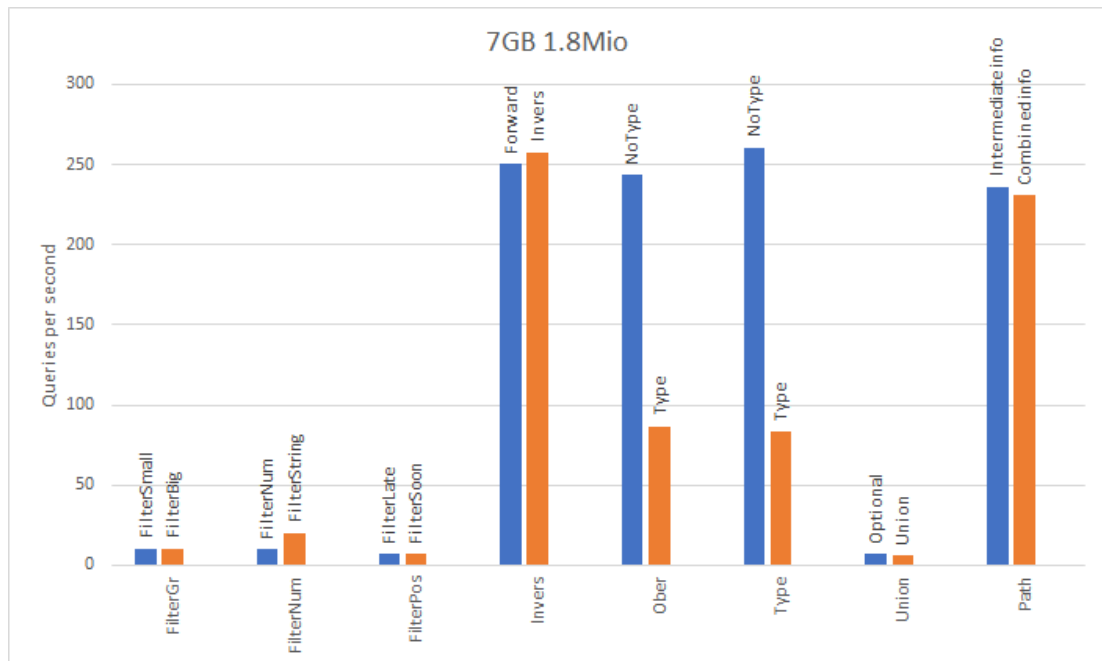


Abbildung 5: 7GB und 1,08Mio Tripel

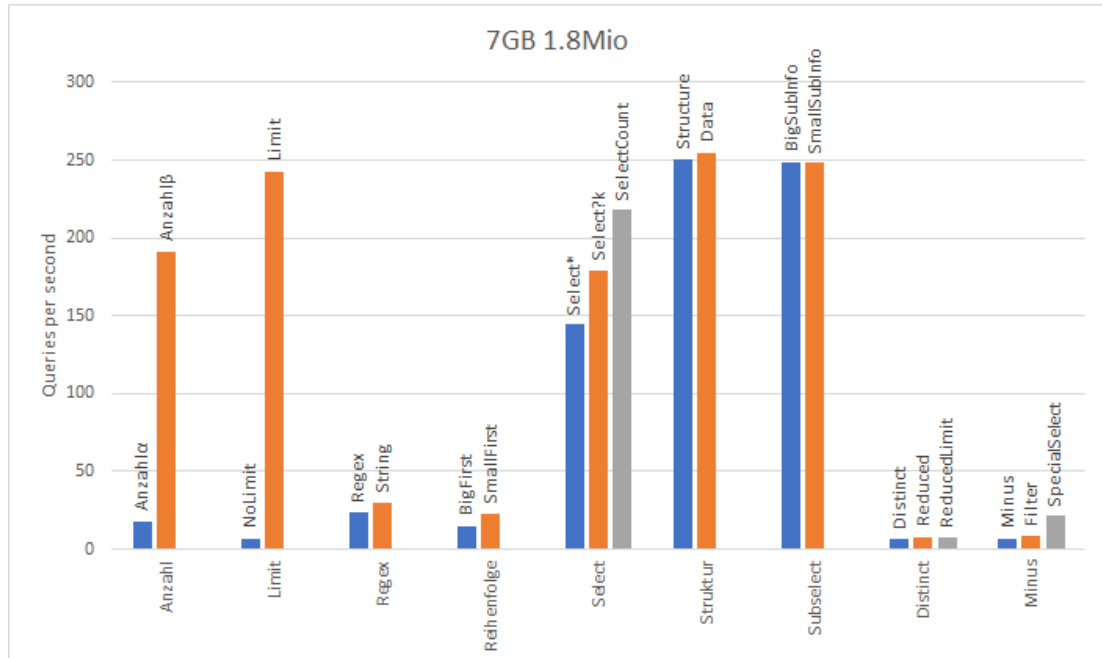


Abbildung 6: 7GB und 1,08Mio Tripel Design 2



## 2 Queries

```
1
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3 PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
  vocabulary/>
4 PREFIX dc: <http://purl.org/dc/elements/1.1/>
5 PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
6 PREFIX rev: <http://purl.org/stuff/rev#>
7
8 select ?product ?review ?date ?feature ?vendor ?number ?
  textual5 ?person where {
```

Listing 8.1: Header

```
1   ?feature rdf:type bsbm:ProductFeature .
2
3 VS
4
5   ?vendor rdf:type bsbm:Vendor .
```

Listing 8.2: Anzahl Ergebnisse

```
1   ?review bsbm:rating1 ?rating.
2   ?review dc:date ?date.
3   filter (?rating >= %x% && ?date < %currentDate%)
4
5 VS
6
7   ?review bsbm:rating1 ?rating.
8   filter (?rating >= %rating%)
9   ?review dc:date ?date.
10  filter (?date < %currentDate%)
```

Listing 8.3: Filter Größe

```
1   ?review bsbm:rating1 ?rating.
2   filter(?rating < %rating%)
```

```
3
4
5  VS
6
7  ?review dc:title ?title.
8  filter(regex(?title, "%word%"))
```

Listing 8.4: Filter Nummer vs String

```
1  ?review bsbm:rating1 ?rating.
2  ?review dc:date ?date.
3  filter (?rating > = %rating%)
4
5  VS
6
7  ?review bsbm:rating1 ?rating.
8  filter (?rating > = %rating%)
9  ?review dc:date ?date.
```

Listing 8.5: Filter Position

```
1  %review% rev:reviewer ?person.
2
3  VS
4
5  ?person \^rev:reviewer %review%.
```

Listing 8.6: Vorwärts vs Rückwärts

```
1  ?review bsbm:rating1 ?rating.
2
3  VS
4
5  ?review bsbm:rating1 ?rating.
6  Limit 5
```

Listing 8.7: Limit/Offset

```
1  ?product rdf:type %ProductType% .
2
3  VS
4
5  ?product rdf:type bsbm:Product .
6  %ProductType% rdf:type bsbm:ProductType .
7  ?product rdf:type %ProductType% .
```

Listing 8.8: Obertyp/Untertyp

```
1  ?review rev:text ?text
2  filter regex(?text, "%word%")
3
4  VS
5
6  ?review rev:text ?text
7  filter STRSTARTS(?text, "%word%")
```

Listing 8.9: Regex vs STR-Function

```
1  ?product rdfs:label ?label.
2  ?product bsbm:productPropertyTextual5 ?textual5.
3
4  VS
5
6  ?product bsbm:productPropertyTextual5 ?textual5.
7  ?product rdfs:label ?label.
```

Listing 8.10: Reihenfolge Triple

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  PREFIX bsbm: <http://www4.wiwiiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
3  select * where {
4    ?product rdf:type %ProductType%.
5    ?product bsbm:productFeature ?feature.
6  }
```

```

7
8 VS
9
10 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
11 PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
    vocabulary/>
12 select ?feature where {
13     ?product rdf:type %ProductType%.
14     ?product bsbm:productFeature ?feature.
15 }
16
17 VS
18
19 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
20 PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
    vocabulary/>
21 select (count(?feature) as \count)where?productrdf:type\%ProductType\%

```

Listing 8.11: Select vs Select specific vs select count()

```

1     ?product rdf:type %ProductType1%.
2     ?product bsbm:productFeature %ProductFeature1%.
3
4 VS
5
6     ?product bsbm:productPropertyNumeric1 %X%.

```

Listing 8.12: Graph-Struktur vs Daten

```

1     ?product rdf:type %ProductType%.
2     ?product rdfs:label ?label.
3 \textbf{\textcolor{rgb}{0.55,0,0}{AFTER}} LIMIT 5
4
5 VS
6
7     {select ?product where {
8         ?product rdf:type %ProductType%.

```

```
9      } LIMIT 5}
10    ?product rdfs:label ?label.
11 \textbf{\textcolor{rgb}{0.55,0,0}{AFTER}} LIMIT 5
```

Listing 8.13: Subselect

```
1    ?product  rdf:type  %ProductType%.
2
3  VS
4
5    ?product rdf:type bsbm:Product .
6    %ProductType% rdf:type bsbm:ProductType .
7    ?product  rdf:type %ProductType% .
```

Listing 8.14: Typ angeben

```
1    ?product rdfs:label ?label.
2    OPTIONAL {
3      ?product bsbm:productPropertyNumeric5 ?number
4    }
5
6  VS
7
8    {select ?product where {
9      ?product rdfs:label ?label.
10     }
11   }
12   UNION {select ?product ?number where {
13     ?product rdfs:label ?label.
14     ?product bsbm:productPropertyNumeric5 ?number.
15   }
16 }
```

Listing 8.15: optional vs UNION

```
1
2 PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
```

```

3 select distinct ?feature where {
4   ?product bsbm:productFeature ?feature
5 }
6
7 VS
8
9 PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
10 select reduced ?feature where {
11   ?product bsbm:productFeature ?feature
12 }
13
14 VS
15
16 PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
17 select reduced ?feature where {
18   ?product bsbm:productFeature ?feature
19 } LIMIT 15000

```

Listing 8.16: Distinct vs Reduced vs Reduced + LIMIT

```

1
2   ?product rdfs:label ?label
3   filter(not exists {?product bsbm:productPropertyNumeric5 ?
   numeric})
4
5
6 VS
7
8   ?product rdfs:label ?label .
9   MINUS {?product bsbm:productPropertyNumeric5 ?numeric}
10
11
12 VS
13

```

```
14 ?product bsbm:productPropertyNumeric3 ?numeric
```

Listing 8.17: Minus vs Filter

### 3 Optimiererausgabe

```
1 PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
3
4 SELECT  ?feature
5 WHERE
6   { ?feature  rdf:type  bsbm:ProductFeature }
7   - - - - -
8   -
9   (prefix ((rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
10          (bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
11            v01/vocabulary/>))
12   (project (?feature)
13    (bgp (triple ?feature rdf:type bsbm:ProductFeature))))
```

Listing 8.18: AnzahlOpt

```
1 PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
3
4 SELECT  ?vendor
5 WHERE
6   { ?vendor  rdf:type  bsbm:Vendor }
7   - - - - -
8   -
9   (prefix ((rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
10          (bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
11            v01/vocabulary/>))
12   (project (?vendor)
```

```
11 (bgp (triple ?vendor rdf:type bsbm:Vendor))))
```

Listing 8.19: AnzahlOpt Design 2

```
1 PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
  vocabulary/>
2
3 SELECT DISTINCT ?feature
4 WHERE
5   { ?product bsbm:productFeature ?feature }
6   -
7   -
8   (prefix ((bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
9     v01/vocabulary/>))
10    (distinct
      (project (?feature)
        (bgp (triple ?product bsbm:productFeature ?feature))))))
```

Listing 8.20: Distinct

```
1 PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
  vocabulary/>
2
3 SELECT REDUCED ?feature
4 WHERE
5   { ?product bsbm:productFeature ?feature }
6   -
7   -
8   (prefix ((bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
9     v01/vocabulary/>))
10    (reduced
      (project (?feature)
        (bgp (triple ?product bsbm:productFeature ?feature))))))
```

Listing 8.21: Distinct Design 2

```
1 PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
  vocabulary/>
```



```

2
3 SELECT REDUCED  ?feature
4 WHERE
5   { ?product  bsbm:productFeature  ?feature  }
6 LIMIT    15000
7 -----
8   -
9 (prefix ((bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
10    v01/vocabulary/>))
11    (slice _ 15000
12      (reduced
13        (project (?feature)
14          (bgp (triple ?product bsbm:productFeature ?feature))))
15      ))

```

Listing 8.22: Distinct Design 3

```

1 PREFIX  xsd:  <http://www.w3.org/2001/XMLSchema#>
2 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
3    vocabulary/>
4
5 PREFIX  dc:   <http://purl.org/dc/elements/1.1/>
6
7 SELECT  ?review ?rating2
8 WHERE
9   { ?review  bsbm:rating1  ?rating1 ;
10     bsbm:rating2  ?rating2
11     FILTER ( ( ?rating1 >= 3 ) && ( ?rating2 < 7 ) )
12   }
13 -----
14   -
15 (prefix ((xsd: <http://www.w3.org/2001/XMLSchema#>)
16    (bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
17      v01/vocabulary/>)
18    (dc: <http://purl.org/dc/elements/1.1/>))
19 (project (?review ?rating2)
20   (filter (< ?rating2 7)

```

```

17      (sequence
18        (filter (>= ?rating1 3)
19          (bgp (triple ?review bsbm:rating1 ?rating1)))
20        (bgp (triple ?review bsbm:rating2 ?rating2))))))

```

Listing 8.23: FilterGr

```

1 PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
2 PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
  vocabulary/>
3 PREFIX dc: <http://purl.org/dc/elements/1.1/>
4
5 SELECT ?review ?rating2
6 WHERE
7   { ?review bsbm:rating1 ?rating1
8     FILTER ( ?rating1 >= 4 )
9     ?review bsbm:rating2 ?rating2
10    FILTER ( ?rating2 < 8 )
11  }
12  - - - - -
13  -
14  (prefix ((xsd: <http://www.w3.org/2001/XMLSchema#>)
15          (bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
16            v01/vocabulary/>)
17          (dc: <http://purl.org/dc/elements/1.1/>)))
18  (project (?review ?rating2)
19    (filter (< ?rating2 8)
20      (sequence
21        (filter (>= ?rating1 4)
22          (bgp (triple ?review bsbm:rating1 ?rating1)))
23        (bgp (triple ?review bsbm:rating2 ?rating2))))))

```

Listing 8.24: FilterGr Design 2

```

1 PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
  vocabulary/>
2

```

```

3 SELECT  ?review
4 WHERE
5   { ?review  bsbm:rating1  ?rating
6     FILTER ( ?rating < 5 )
7   }
8   -
9   (prefix ((bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
10     v01/vocabulary/>))
11     (project (?review)
12       (filter (< ?rating 5)
13         (bgp (triple ?review bsbm:rating1 ?rating))))))

```

Listing 8.25: FilterNum

```

1 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
2 PREFIX  dc:  <http://purl.org/dc/elements/1.1/>
3
4 SELECT  ?review
5 WHERE
6   { ?review  dc:title  ?title
7     FILTER regex(?title, "test")
8   }
9   -
10  (prefix ((bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
11    v01/vocabulary/>)
12    (dc: <http://purl.org/dc/elements/1.1/>))
13    (project (?review)
14      (filter (regex ?title "test")
15        (bgp (triple ?review dc:title ?title))))))

```

Listing 8.26: FilterNum Design 2

```

1 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>

```

```

2 PREFIX  dc:    <http://purl.org/dc/elements/1.1/>
3
4 SELECT  ?review
5 WHERE
6   { ?review  bsbm:rating1  ?rating ;
7         dc:date           ?date
8     FILTER ( ?rating >= 8 )
9   }
10  - - - - -
11  -
12  (prefix ((bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
13           v01/vocabulary/>)
14           (dc: <http://purl.org/dc/elements/1.1/>))
15  (project (?review)
16  (sequence
17  (filter (>= ?rating 8)
18  (bgp (triple ?review bsbm:rating1 ?rating)))
19  (bgp (triple ?review dc:date ?date))))))

```

Listing 8.27: FilterPos

```

1 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
2 PREFIX  dc:    <http://purl.org/dc/elements/1.1/>
3
4 SELECT  ?review
5 WHERE
6   { ?review  bsbm:rating1  ?rating
7     FILTER ( ?rating >= 3 )
8     ?review  dc:date       ?date
9   }
10  - - - - -
11  -
12  (prefix ((bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
13           v01/vocabulary/>)
14           (dc: <http://purl.org/dc/elements/1.1/>))

```

```

13 (project (?review)
14   (sequence
15     (filter (>= ?rating 3)
16       (bgp (triple ?review bsbm:rating1 ?rating)))
17     (bgp (triple ?review dc:date ?date))))))

```

Listing 8.28: FilterPos Design 2

```

1 PREFIX  rev:  <http://purl.org/stuff/rev#>
2
3 SELECT  ?person
4 WHERE
5   { <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
6     dataFromRatingSite1/Review5>
7     rev:reviewer  ?person
8   }
9   - - - - -
10  (prefix ((rev: <http://purl.org/stuff/rev#>))
11    (project (?person)
12      (bgp (triple <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
13        v01/instances/dataFromRatingSite1/Review5> rev:reviewer
14          ?person))))))

```

Listing 8.29: Invers

```

1 PREFIX  rev:  <http://purl.org/stuff/rev#>
2
3 SELECT  ?person
4 WHERE
5   { ?person ^rev:reviewer <http://www4.wiwiss.fu-berlin.de/
6     bizer/bsbm/v01/instances/dataFromRatingSite1/Review5> }
7   - - - - -
8   (prefix ((rev: <http://purl.org/stuff/rev#>))
9     (project (?person)
10       (bgp (triple <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/

```

```
v01/instances/dataFromRatingSite1/Review5> rev:reviewer
?person))))
```

Listing 8.30: Invers Design 2

```
1 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
2
3 SELECT  ?review
4 WHERE
5   { ?review  bsbm:rating1  ?rating }
6   - - - - -
7   -
8 (prefix ((bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
   v01/vocabulary/>))
9 (project (?review)
   (bgp (triple ?review bsbm:rating1 ?rating)))))
```

Listing 8.31: Limit

```
1 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
2
3 SELECT  ?review
4 WHERE
5   { ?review  bsbm:rating1  ?rating }
6 LIMIT   5
7   - - - - -
8   -
9 (prefix ((bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
   v01/vocabulary/>))
10 (slice _ 5
11 (project (?review)
   (bgp (triple ?review bsbm:rating1 ?rating)))))
```

Listing 8.32: Limit Design 2

```

1 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
    vocabulary/>
2 PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3
4 SELECT  ?product ?label
5 WHERE
6   { ?product rdfs:label ?label
7     FILTER NOT EXISTS { ?product bsbm:productPropertyNumeric5
8       ?numeric }
9   }
10  - - - - -
11  -
12 (prefix ((bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
13   v01/vocabulary/>)
14   (rdfs: <http://www.w3.org/2000/01/rdf-schema#>))
15   (project (?product ?label)
16     (filter (notexists (bgp (triple ?product bsbm:
17       productPropertyNumeric5 ?numeric)))
18       (bgp (triple ?product rdfs:label ?label)))))

```

Listing 8.33: Minus

```

1 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
    vocabulary/>
2 PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3
4 SELECT  ?product ?label ?numeric
5 WHERE
6   { ?product rdfs:label ?label
7     MINUS
8       { ?product bsbm:productPropertyNumeric5 ?numeric }
9   }
10  - - - - -
11  -
12 (prefix ((bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
13   v01/vocabulary/>)

```

```

12      (rdfs: <http://www.w3.org/2000/01/rdf-schema#>))
13 (project (?product ?label ?numeric)
14   (minus
15    (bgp (triple ?product rdfs:label ?label))
16    (bgp (triple ?product bsbm:productPropertyNumeric5 ?
      numeric))))))

```

Listing 8.34: Minus Design 2

```

1 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
2
3 SELECT  ?product ?label ?numeric
4 WHERE
5   { ?product  bsbm:productPropertyNumeric3  ?numeric }
6   - - - - -
7   -
8 (prefix ((bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
   v01/vocabulary/>))
9 (project (?product ?label ?numeric)
   (bgp (triple ?product bsbm:productPropertyNumeric3 ?
      numeric))))

```

Listing 8.35: Minus Design 3

```

1 PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
3
4 SELECT  ?product
5 WHERE
6   { ?product  rdf:type  <http://www4.wiwiss.fu-berlin.de/bizer
   /bsbm/v01/instances/ProductType1> }
7   - - - - -
8   -
9 (prefix ((rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
   (bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/

```



```

        v01/vocabulary/>))
10 (project (?product)
11   (bgp (triple ?product rdf:type <http://www4.wiwiss.fu-
        berlin.de/bizer/bsbm/v01/instances/ProductType1>))))

```

Listing 8.36: Ober

```

1 PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
3
4 SELECT  ?product
5 WHERE
6   { ?product  rdf:type  bsbm:Product .
7     <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
       ProductType1>
8       rdf:type  bsbm:ProductType .
9     ?product  rdf:type  bsbm:ProductType
10  }
11  - - - - -
12  -
13  (prefix ((rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
14          (bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
15              v01/vocabulary/>))
16  (project (?product)
17    (bgp
18      (triple ?product rdf:type bsbm:Product)
19      (triple <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
20          instances/ProductType1> rdf:type bsbm:ProductType)
21      (triple ?product rdf:type bsbm:ProductType)
22    )))

```

Listing 8.37: Ober Design 2

```

1 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
2 PREFIX  rev:  <http://purl.org/stuff/rev#>

```

```

3 PREFIX  bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm
    /v01/instances/>
4
5 SELECT  ?person
6 WHERE
7   { <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
    dataFromProducer1/Product3> ^bsbm:reviewFor ?review .
8     ?review rev:reviewer ?person
9   }
10  - - - - -
11  -
12  (prefix ((bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
    v01/vocabulary/>)
13          (rev: <http://purl.org/stuff/rev#>)
14          (bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/
    bsbm/v01/instances/>))
15  (project (?person)
16    (bgp
17      (triple ?review bsbm:reviewFor <http://www4.wiwiss.fu-
    berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/
    Product3>)
18      (triple ?review rev:reviewer ?person)
19    )))

```

Listing 8.38: Path

```

1 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
    vocabulary/>
2 PREFIX  rev: <http://purl.org/stuff/rev#>
3 PREFIX  bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm
    /v01/instances/>
4
5 SELECT  ?person
6 WHERE
7   { <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
    dataFromProducer1/Product3> ^bsbm:reviewFor/rev:reviewer

```

```

      ?person }
8  - - - - -
9  (prefix ((bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
    v01/vocabulary/>)
10      (rev: <http://purl.org/stuff/rev#>)
11      (bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/
    bsbm/v01/instances/>))
12  (project (?person)
13      (bgp
14      (triple ??P1 bsbm:reviewFor <http://www4.wiwiss.fu-
    berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/
    Product3>)
15      (triple ??P1 rev:reviewer ?person)
16      )))

```

Listing 8.39: Path Design 2

```

1  PREFIX  rev:  <http://purl.org/stuff/rev#>
2
3  SELECT  ?review
4  WHERE
5      { ?review  rev:text  ?text
6        FILTER regex(?text, "^test")
7      }
8  - - - - -
9  (prefix ((rev: <http://purl.org/stuff/rev#>))
10      (project (?review)
11          (filter (regex ?text "^test")
12              (bgp (triple ?review rev:text ?text))))))

```

Listing 8.40: Regex

```

1  PREFIX  rev:  <http://purl.org/stuff/rev#>
2
3  SELECT  ?review

```

```

4 WHERE
5   { ?review rev:text ?text
6     FILTER strstarts(?text, "test")
7   }
8   - - - - -
9   -
10  (prefix ((rev: <http://purl.org/stuff/rev#>))
11    (project (?review)
12      (filter (strstarts ?text "test")
13        (bgp (triple ?review rev:text ?text))))))

```

Listing 8.41: Regex Design 2

```

1 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
2 PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3
4 SELECT  ?product ?label ?textual5
5 WHERE
6   { ?product rdfs:label ?label ;
7     bsbm:productPropertyTextual5 ?textual5
8   }
9   - - - - -
10  -
11  (prefix ((bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
12    v01/vocabulary/>)
13    (rdfs: <http://www.w3.org/2000/01/rdf-schema#>))
14    (project (?product ?label ?textual5)
15      (bgp
16        (triple ?product rdfs:label ?label)
17        (triple ?product bsbm:productPropertyTextual5 ?textual5)
18      )))

```

Listing 8.42: Reihenfolge

```

1 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>

```

```

2 PREFIX   rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3
4 SELECT   ?product ?label ?textual5
5 WHERE
6   { ?product   bsbm:productPropertyTextual5   ?textual5 ;
7           rdfs:label                           ?label
8   }
9 - - - - -
10
11 (prefix ((bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
12         v01/vocabulary/>)
13         (rdfs: <http://www.w3.org/2000/01/rdf-schema#>))
14 (project (?product ?label ?textual5)
15         (bgp
16           (triple ?product bsbm:productPropertyTextual5 ?textual5)
17           (triple ?product rdfs:label ?label)
18         )))

```

Listing 8.43: Reihenfolge Design 2

```

1 PREFIX   rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX   bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
3           vocabulary/>
4
5 SELECT   *
6 WHERE
7   { ?product   rdf:type                <http://www4.wiwiss.fu-
8           berlin.de/bizer/bsbm/v01/instances/ProductType5> ;
9           bsbm:productFeature   ?feature
10   }
11 - - - - -
12
13 (prefix ((rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
14         (bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
15         v01/vocabulary/>))
16 (bgp

```

```

13      (triple ?product rdf:type <http://www4.wiwiss.fu-berlin.de
        /bizer/bsbm/v01/instances/ProductType5>)
14      (triple ?product bsbm:productFeature ?feature)
15  ))

```

Listing 8.44: Select

```

1  PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
        vocabulary/>
3
4  SELECT  ?feature
5  WHERE
6  { ?product  rdf:type          <http://www4.wiwiss.fu-
        berlin.de/bizer/bsbm/v01/instances/ProductType5> ;
7          bsbm:productFeature  ?feature
8  }
9  - - - - -
10 (prefix ((rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
11         (bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
12         v01/vocabulary/>))
13 (project (?feature)
14 (bgp
15 (triple ?product rdf:type <http://www4.wiwiss.fu-berlin.
16 de/bizer/bsbm/v01/instances/ProductType5>)
    (triple ?product bsbm:productFeature ?feature)
    )))

```

Listing 8.45: Select Design 2

```

1  PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
        vocabulary/>
3
4  SELECT  (COUNT(?feature) AS ?count)
5  WHERE

```

```

6 { ?product rdf:type <http://www4.wiwiss.fu-
   berlin.de/bizer/bsbm/v01/instances/ProductType5> ;
7     bsbm:productFeature ?feature
8 }
9 - - - - -
10 (prefix ((rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
11     (bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
12     v01/vocabulary/>))
13 (project (?count)
14     (extend ((?count ?.0))
15     (group () ((?.0 (count ?feature)))
16     (bgp
17     (triple ?product rdf:type <http://www4.wiwiss.fu-
18     berlin.de/bizer/bsbm/v01/instances/ProductType5>)
19     (triple ?product bsbm:productFeature ?feature)
20     )))))

```

Listing 8.46: Select Design 3

```

1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
3
4 SELECT ?product
5 WHERE
6 { ?product rdf:type <http://www4.wiwiss.fu-
   berlin.de/bizer/bsbm/v01/instances/ProductType5> ;
7     bsbm:productFeature <http://www4.wiwiss.fu-
8     berlin.de/bizer/bsbm/v01/instances/
9     ProductFeature142>
10 }
11 - - - - -
12 (prefix ((rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
13     (bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/

```

```

        v01/vocabulary/>))
12 (project (?product)
13     (bgp
14         (triple ?product rdf:type <http://www4.wiwiss.fu-berlin.
            de/bizer/bsbm/v01/instances/ProductType5>)
15         (triple ?product bsbm:productFeature <http://www4.wiwiss
            .fu-berlin.de/bizer/bsbm/v01/instances/
            ProductFeature142>)
16     )))

```

Listing 8.47: Struktur

```

1 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
    vocabulary/>
2
3 SELECT  ?product
4 WHERE
5     { ?product  bsbm:productPropertyNumeric1  65  }
6  - - - - -
7  -
8  (prefix ((bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
    v01/vocabulary/>))
9  (project (?product)
    (bgp (triple ?product bsbm:productPropertyNumeric1 65))))

```

Listing 8.48: Struktur Design 2

```

1 PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3
4 SELECT  ?product
5 WHERE
6     { ?product  rdf:type      <http://www4.wiwiss.fu-berlin.de/
            bizer/bsbm/v01/instances/ProductType5> ;
7         rdfs:label  ?label
8     }
9 LIMIT   5

```



```

10  - - - - -
11  (prefix ((rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
12          (rdfs: <http://www.w3.org/2000/01/rdf-schema#>))
13    (slice _ 5
14      (project (?product)
15        (bgp
16          (triple ?product rdf:type <http://www4.wiwiss.fu-
17              berlin.de/bizer/bsbm/v01/instances/ProductType5>)
18          (triple ?product rdfs:label ?label)
19        )))

```

Listing 8.49: Subselect

```

1  PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3
4  SELECT  ?product
5  WHERE
6    { { SELECT  ?product
7        WHERE
8          { ?product  rdf:type  <http://www4.wiwiss.fu-berlin.de
9              /bizer/bsbm/v01/instances/ProductType5> }
10         LIMIT    5
11       }
12       ?product  rdfs:label  ?label
13     }
14  LIMIT    5
15  - - - - -
16  -
17  (prefix ((rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
18          (rdfs: <http://www.w3.org/2000/01/rdf-schema#>))
19    (slice _ 5
20      (project (?product)
21        (sequence
22          (slice _ 5

```

```

21      (project (?product)
22        (bgp (triple ?product rdf:type <http://www4.wiwiss
                .fu-berlin.de/bizer/bsbm/v01/instances/
                ProductType5>))))
23      (bgp (triple ?product rdfs:label ?label))))))

```

Listing 8.50: Subselect Design 2

```

1 PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
    vocabulary/>
3
4 SELECT  ?product
5 WHERE
6   { ?product  rdf:type  <http://www4.wiwiss.fu-berlin.de/bizer
    /bsbm/v01/instances/ProductType5> }
7   - - - - -
8   -
9   (prefix ((rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
10      (bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
11      v01/vocabulary/>))
12      (project (?product)
13        (bgp (triple ?product rdf:type <http://www4.wiwiss.fu-
14        berlin.de/bizer/bsbm/v01/instances/ProductType5>))))))

```

Listing 8.51: Type

```

1 PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
    vocabulary/>
3
4 SELECT  ?product
5 WHERE
6   { ?product  rdf:type  bsbm:Product .
7     <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
8     ProductType5>
9     rdf:type  bsbm:ProductType .

```

```

9      ?product  rdf:type  <http://www4.wiwiss.fu-berlin.de/bizer
      /bsbm/v01/instances/ProductType5>
10  }
11  - - - - -
12  -
13  (prefix ((rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
14          (bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
15              v01/vocabulary/>))
16  (project (?product)
17  (bgp
18  (triple ?product rdf:type bsbm:Product)
19  (triple <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
      instances/ProductType5> rdf:type bsbm:ProductType)
20  (triple ?product rdf:type <http://www4.wiwiss.fu-berlin.
      de/bizer/bsbm/v01/instances/ProductType5>)
21  )))

```

Listing 8.52: Type Design 2

```

1  PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
      vocabulary/>
2  PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3
4  SELECT  ?product ?number
5  WHERE
6  { ?product  rdfs:label  ?label
7    OPTIONAL
8    { ?product  bsbm:productPropertyNumeric5  ?number  }
9  }
10 - - - - -
11 -
12 (prefix ((bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
13     v01/vocabulary/>)
14     (rdfs: <http://www.w3.org/2000/01/rdf-schema#>))
15 (project (?product ?number)
16 (conditional

```

```

15      (bgp (triple ?product rdfs:label ?label))
16      (bgp (triple ?product bsbm:productPropertyNumeric5 ?
      number))))))

```

Listing 8.53: Union

```

1 PREFIX  bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
  vocabulary/>
2 PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3
4 SELECT  ?product ?number
5 WHERE
6   { { SELECT  ?product
7       WHERE
8         { ?product  rdfs:label  ?label }
9     }
10  UNION
11    { SELECT  ?product ?number
12        WHERE
13          { ?product  rdfs:label          ?label ;
14            bsbm:productPropertyNumeric5  ?number
15          }
16    }
17  }
18  - - - - -
19  -
20  (prefix ((bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
21    v01/vocabulary/>)
22    (rdfs: <http://www.w3.org/2000/01/rdf-schema#>))
23  (project (?product ?number)
24    (union
25      (project (?product)
26        (bgp (triple ?product rdfs:label ?/label)))
27      (project (?product ?number)
28        (bgp
29          (triple ?product rdfs:label ?/label)

```

```
28         (triple ?product bsbm:productPropertyNumeric5 ?  
29             number)  
        ))))
```

Listing 8.54: Union Design 2



## Literatur

- [o4a] *International Semantic Web Conference*. 2004.
- [o4b] *International Semantic Web Conference*. 2004.
- [o4c] *KR*. 2004.
- [10] *Extended Semantic Web Conference*. 2010.
- [15] *International Semantic Web Conference*. 2015.
- [19a] 2019.
- [19b] *The World Wide Web Conference*. 2019.
- [Apa19a] Apache Jena. *Apache Jena - ARQ - A SPARQL Processor for Jena*. 6.12.2019. URL: <https://jena.apache.org/documentation/query/index.html>.
- [Apa19b] Apache Jena. *Apache Jena - Jena architecture overview*. 6.12.2019. URL: [https://jena.apache.org/about\\_jena/architecture.html](https://jena.apache.org/about_jena/architecture.html).
- [Apa19c] Apache Jena. *Apache Jena - Jena architecture overview*. 6.12.2019. URL: [https://jena.apache.org/about\\_jena/architecture.html](https://jena.apache.org/about_jena/architecture.html).
- [Apa19d] Apache Jena. *Apache Jena - Jena Full Text Search*. 6.12.2019. URL: <https://jena.apache.org/documentation/query/text-query.html>.
- [Apa19e] Apache Jena. *Apache Jena - TDB Optimizer*. 6.12.2019. URL: <https://jena.apache.org/documentation/tdb/optimizer.html>.
- [BHL+01] Tim Berners-Lee, James Hendler, Ora Lassila u. a. „The semantic web“. In: *Scientific american* 284.5 (2001), S. 28–37.
- [Biz12] Schultz Andreas Bizer Chris. *Berlin SPARQL Benchmark*. 23.03.2012. URL: <http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/index.html#usecases>.
- [CRL10] Roger Castillo, Christian Rothe und Ulf Leser. *RDFMatView: Indexing RDF Data for SPARQL Queries*. Professoren des Inst. für Informatik, 2010.

- [Gai12] Gaidot Regis. *semantic\_web\_technology\_stack.png* (PNG-Grafik, 900 × 600 Pixel). 20.01.2012. URL: [http://bnode.org/media/2009/07/08/semantic\\_web\\_technology\\_stack.png](http://bnode.org/media/2009/07/08/semantic_web_technology_stack.png).
- [Haa+04a] Peter Haase u. a. „A comparison of RDF query languages“. In: *International Semantic Web Conference*. 2004, S. 502–517. URL: [https://www.researchgate.net/profile/Peter\\_Haase/publication/221466538\\_A\\_Comparison\\_of\\_RDF\\_Query\\_Languages/links/0fcfd5110fe676bc2d000000/A-Comparison-of-RDF-Query-Languages.pdf](https://www.researchgate.net/profile/Peter_Haase/publication/221466538_A_Comparison_of_RDF_Query_Languages/links/0fcfd5110fe676bc2d000000/A-Comparison-of-RDF-Query-Languages.pdf).
- [Haa+04b] Peter Haase u. a. „A comparison of RDF query languages“. In: *International Semantic Web Conference*. 2004, S. 502–517.
- [Hab15] Jean Habimana. „Query Optimization Techniques-Tips For Writing Efficient And Faster SQL Queries“. In: *International Journal Of Scientific & Technology Research* 4.10 (2015), S. 22–26.
- [Har18] Seaborne Andy Harris Steve. *SPARQL 1.1 Query Language*. 9.10.2018. URL: <https://www.w3.org/TR/sparql11-query/>.
- [Hel+] Tobias Hellmund u. a. „Employing Geospatial Semantics and Semantic Web Technologies in Natural Disaster Management“. In: ().
- [Her+18] Philipp Hertweck u. a. „The Backbone of Decision Support Systems: The Sensor to Decision Chain“. In: *International Journal of Information Systems for Crisis Response and Management (IJISCRAM)* 10.4 (2018), S. 65–87. URL: <https://www.igi-global.com/ViewTitle.aspx?TitleId=235420&isxn=9781522543855>.
- [Hes] Hesse Ralf. *DBIS: Query Optimization in RDF Databases*. URL: <http://www.dbis.informatik.hu-berlin.de/forschung/projekte/query-optimization-in-rdf-databases.html>.
- [Jor13] Jordan. *The Importance of W3C Standards | Cross-Browser Compatible Websites*. 2013. URL: <https://www.bopdesign.com/bop-blog/2013/06/the-importance-of-w3c-standards/>.
- [KK12] Haydar Kurban und Mika Kato. „A Web-based Application for Estimation of Personal Vulnerability in Disasters“. In: (2012).
- [Kon+] Efstratios Kontopoulos u. a. „Applying Semantic Web Technologies for Decision Support in Climate-Related Crisis Management“. In: ().



- [MR12] Sanjay Kumar Malik und S. Rizvi. „A framework for SPARQL query processing, optimization and execution with illustrations“. In: *International Journal of Computer Information Systems and Industrial Management Applications* 4 (2012), S. 208–218.
- [MUA10] Michael Martin, Jörg Unbehauen und Sören Auer. „Improving the performance of semantic web applications with SPARQL query caching“. In: *Extended Semantic Web Conference*. 2010, S. 304–318.
- [New19] New Vantage. *Big Data Statistics 2019*. 2019. URL: <https://techjury.net/stats-about/big-data-statistics/#gref>.
- [Ont] Ontotext. *What is the semantic web*. URL: <https://www.ontotext.com/knowledgehub/fundamentals/what-is-the-semantic-web/>.
- [Pato4] Peter F. Patel-Schneider. „What is OWL (and why should I care)?“ In: *KR*. 2004, S. 735–737.
- [Sal+19] Muhammad Saleem u. a. „How Representative Is a SPARQL Benchmark? An Analysis of RDF Triplestore Benchmarks?“ In: *The World Wide Web Conference*. 2019, S. 1623–1633.
- [SBHo6] Nigel Shadbolt, Tim Berners-Lee und Wendy Hall. „The semantic web revisited“. In: *IEEE intelligent systems* 21.3 (2006), S. 96–101.
- [Sch+19] Manfred Schenk u. a. „Semantic Queries Supporting Crisis Management Systems“. In: 2019, S. 38–43. URL: [http://www.thinkmind.org/download.php?articleid=semapro\\_2019\\_3\\_20\\_30011](http://www.thinkmind.org/download.php?articleid=semapro_2019_3_20_30011).
- [SMN15] Muhammad Saleem, Qaiser Mehmood und Axel-Cyrille Ngonga Ngomo. „Feasible: A feature-based sparql benchmark generation framework“. In: *International Semantic Web Conference*. 2015, S. 52–69.
- [Ves] R. Vesse. „SPARQL Optimization 101, Tutorial at ApacheCon North America 2014 (2014)“. In: URL <https://events.static.linuxfound.org/sites/events/files/slides/SPARQL%20Optimisation%20101%20Tutorial.pdf> 20101 ().
- [Ves13] Vesse Rob. *SPARQL Query: Tuning SPARQL for Better Performance - Cray*. 2013. URL: <https://www.cray.com/blog/tuning-sparql-queries-performance/>.
- [W3C18] W3C. *RdfStoreBenchmarking - W3C Wiki*. 18.10.2018. URL: <https://www.w3.org/wiki/RdfStoreBenchmarking>.

- 
- [W3C19a] W3C. *SPARQL Query Language for RDF - Formal Defintions*. 16.12.2019. URL: [https://www.w3.org/2001/sw/DataAccess/rq23/sparql-defns.html#defn\\_TriplePattern](https://www.w3.org/2001/sw/DataAccess/rq23/sparql-defns.html#defn_TriplePattern).
- [W3C19b] W3C. *XML Essentials - W3C*. 19.06.2019. URL: <https://www.w3.org/standards/xml/core>.
- [Wüb19] Klaus Wübbenhorst. *Definition: Benchmarking*. 17.12.2019. URL: <https://wirtschaftslexikon.gabler.de/definition/benchmarking-29988>.
- [Zai18] Jennifer Zaino. *Semantic Web and Semantic Technology Trends in 2019 - DATAVERSITY*. 2018. URL: <https://www.dataversity.net/semantic-web-semantic-technology-trends-2019/>.

# Abbildungsverzeichnis

1.1	Semantic Web Stack [Gai12]	4
1.2	Funktion Reasoner	10
1.3	Jena Architektur [Apa19b]	12
2.1	Vorgehensweise	17
3.1	Benchmarks bezüglich verschiedener Punkte	22
3.2	Ontology Schema	26
5.1	Gruppeneinteilung	41
6.1	Design des Query Speicherns	45
6.2	Design der Datei für das Ausführen der Queries	46
6.3	Kommandozeilen Argument	47
6.4	Während der Ausführung	47
6.5	Ausführung	48
7.1	2,25GB und 4,09Mio Tripel	50
7.2	2,25GB und 4,09Mio Tripel Design 2	50
7.3	7GB und 4,09Mio Tripel	51
7.4	7GB und 4,09Mio Tripel Design 2	51
7.5	Checkbox	54
7.6	Boxplot 1,11GB mit 1,08Mio Tripeln	56
7.7	Boxplot 1,11GB mit 1,08Mio Tripeln Design 2	56
7.8	Boxplot 7GB mit 1,08Mio Tripeln	57
7.9	Cacheverhalten	57
1	1,11GB und 1,08Mio Tripel	62
2	1,11GB und 1,08Mio Tripel Design 2	62
3	4GB und 1,08Mio Tripel	63
4	4GB und 1,08Mio Tripel Design 2	63

5	7GB und 1,08Mio Tripel . . . . .	64
6	7GB und 1,08Mio Tripel Design 2 . . . . .	64

## Listings

1.1	XML Beispiel . . . . .	3
1.2	SPARQL-Abfrage . . . . .	6
4.1	Query . . . . .	27
4.2	Header . . . . .	34
4.3	Anzahl Ergebnisse . . . . .	35
4.4	Filter Größe . . . . .	35
4.5	Filter Nummer vs String . . . . .	35
4.6	ARQ-FilterGr . . . . .	36
4.7	ARQ-FilterGr2 . . . . .	36
5.1	Header . . . . .	40
5.2	Beispiel Kommando . . . . .	42
8.1	Header . . . . .	65
8.2	Anzahl Ergebnisse . . . . .	65
8.3	Filter Größe . . . . .	65
8.4	Filter Nummer vs String . . . . .	65
8.5	Filter Position . . . . .	66
8.6	Vorwärts vs Rückwärts . . . . .	66
8.7	Limit/Offset . . . . .	66
8.8	Obertyp/Untertyp . . . . .	67
8.9	Regex vs STR-Function . . . . .	67
8.10	Reihenfolge Triple . . . . .	67
8.11	Select vs Select specific vs select count() . . . . .	67
8.12	Graph-Struktur vs Daten . . . . .	68
8.13	Subselect . . . . .	68
8.14	Typ angeben . . . . .	69
8.15	optional vs UNION . . . . .	69
8.16	Distinct vs Reduced vs Reduced + LIMIT . . . . .	69

8.17	Minus vs Filter . . . . .	70
8.18	AnzahlOpt . . . . .	71
8.19	AnzahlOpt Design 2 . . . . .	71
8.20	Distinct . . . . .	72
8.21	Distinct Design 2 . . . . .	72
8.22	Distinct Design 3 . . . . .	72
8.23	FilterGr . . . . .	73
8.24	FilterGr Design 2 . . . . .	74
8.25	FilterNum . . . . .	74
8.26	FilterNum Design 2 . . . . .	75
8.27	FilterPos . . . . .	75
8.28	FilterPos Design 2 . . . . .	76
8.29	Invers . . . . .	77
8.30	Invers Design 2 . . . . .	77
8.31	Limit . . . . .	78
8.32	Limit Design 2 . . . . .	78
8.33	Minus . . . . .	78
8.34	Minus Design 2 . . . . .	79
8.35	Minus Design 3 . . . . .	80
8.36	Ober . . . . .	80
8.37	Ober Design 2 . . . . .	81
8.38	Path . . . . .	81
8.39	Path Design 2 . . . . .	82
8.40	Regex . . . . .	83
8.41	Regex Design 2 . . . . .	83
8.42	Reihenfolge . . . . .	84
8.43	Reihenfolge Design 2 . . . . .	84
8.44	Select . . . . .	85
8.45	Select Design 2 . . . . .	86
8.46	Select Design 3 . . . . .	86
8.47	Struktur . . . . .	87
8.48	Struktur Design 2 . . . . .	88
8.49	Subselect . . . . .	88
8.50	Subselect Design 2 . . . . .	89
8.51	Type . . . . .	90
8.52	Type Design 2 . . . . .	90

---

8.53	Union . . . . .	91
8.54	Union Design 2 . . . . .	92





# Glossar

*Berlin SPARQL Benchmark* Ein Performance-Benchmark für die Analyse von Triple-Stores.

*BSBM* Berlin Sparql BenchMark.

*OWL* Web Ontology Language.

*PGFplots* Eine Sammlung von TikZ-Paketen, die ein direktes Erzeugen von Diagrammen aller Art (inkl. 3D-Diagramme) direkt aus LaTeX heraus ermöglicht.

*Query* Eine semantische Abfrage gegen eine Datenbank.

*Uniform Resource Identifier* Eine Sammlung von LaTeX-Paketen, die ein direktes Erzeugen von (technischen) Zeichnungen, Diagrammen, etc. in LaTeX erlaubt.

*URI* Uniform Resource Identifier.

*W3C* World Wide Web Consortium.

*Web Ontology Language* Eine Ontologie-Sprache für Triple.

*World Wide Web Consortium* Eine Organisation, für die Standardisierung der World Wide Webs

*XML* eXtended Markup Language.