Bachelorarbeit:

Vom Waldbau zum Wissensgraphen: Entwicklung einer RDF-Generierungspipeline mit SPARQL-Construct-Queries

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Lehrstuhl für Algorithmen und Datenstrukturen

Felix Vierthaler

Erstprüferin: Prof. Dr. Hannah Bast

Abgabedatum: 20.09.2026

universität freiburg

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Freiburg, 20.09.2026 |

Ort, Datum | Unterschrift

KI-Nutzung

Ich habe für verschiedene Aspekte dieser Arbeit KI-Werkzeuge eingesetzt. ChatGPT und Claude Code habe ich als Diskussionspartner zu Themen dieser Arbeit genutzt. Diese habe ich als unterstützende Werkzeuge betrachtet, welche nicht die eigentliche Planung und Umsetzung dieser Arbeit übernommen haben.

Für vereinzelte Sätze habe ich Formulierungsvorschläge von ChatGPT oder Claude Code generieren lassen und teilweise übernommen. Jedoch habe ich keine KI-Werkzeuge genutzt, um längere Passagen schreiben zu lassen.

Ich habe diese Arbeit mit dem Satzsystem Typst erstellt. Dabei habe ich Claude Code genutzt, um gestalterische Elemente wie Formatierung, Layout und Komponenten zu programmieren. Der Inhalt der Arbeit wurde von Claude Code dabei nicht verändert.

Ich habe Claude Code und die VS-Code-Erweiterung LTeX+ (basierend auf Language-Tool) genutzt, um Rechtschreibfehler und Grammatikfehler zu finden und Korrekturen vorzuschlagen. Alle Änderungen am Dokument habe ich dabei von Hand durchgeführt.

Ich habe Claude Code genutzt, um die Matplotlib-Diagramme nach meinen Vorstellungen zu gestalten. Die Datengrundlage der Diagramme wurde von mir generiert.

Abstract

Besonders angesichts des Klimawandels ist es wichtig, bestehende Daten über den Wald sinnvoll zu nutzen, um waldbauliche Entscheidungen zu optimieren. Hierfür müssen Daten aus unterschiedlichen Quellen und Formaten miteinander verknüpft und in ein gemeinsames Datenmodell überführt werden. Dafür eignet sich das Resource Description Framework (RDF), ein Datenmodell, das Informationen als Graph mit Knoten und Kanten darstellt. In dieser Arbeit präsentieren wir Qonstructor, eine domänenunabhängige RDF-Generierungspipeline. Qonstructor generiert in mehreren iterativen Schritten mithilfe von SPARQL-Construct-Queries aus Rohdaten einen verknüpften RDF-Wissensgraphen. SPARQL-Construct-Queries sind Abfragen an einen RDF-Wissensgraphen, die wiederum einen RDF-Wissensgraphen als Ergebnis liefern. Wir haben mit Qonstructor aus 34 Rohdateien und 37 SPARQL-Construct-Queries einen Wissensgraphen mit circa 117 Millionen Kanten zum Thema Waldbau generiert. Der generierte Wissensgraph dient als Basis für verschiedene Demoanwendungen.

Inhaltsverzeichnis

1 Einleitung ... 1
2 Grundlagen ... 3
2.1 Geodaten ... 3
2.2 Wissensgraphen ... 4
2.3 RDF ... 5
2.4 SPARQL ... 7
2.5 Verwendete Software ... 9
3 Verwandte Arbeiten ... 12
3.1 Programmatischer Ansatz ... 12
3.2 RDF-Mapping-Standards ... 13
3.3 SPARQL-basierte RDF-Generierung ... 14
3.4 Motivation der Entwicklung von Qonstructor ... 15
4 Qonstructor ... 16
4.1 Designziele ... 16
4.2 Systemarchitektur ... 16
4.3 Konfiguration ... 20
4.4 Generic-Generation-Stage ... 20
4.5 Construct-Stages 1-N ... 25
4.6 Export-Stage ... 25
4.7 QLever-Integration ... 26
4.8 Parallelität ... 27
4.9 Containerisierung und Nutzungsworkflow ... 27
5 Forstgraph ... 29
5.1 Ontologische Grundlagen ... 29
5.2 IRIs ... 30
5.3 Daten und Preprocessing ... 30
5.4 Stage-Konfigurationsdateien ... 33
5.5 Generic-Generation-Stage ... 34
5.6 Construct-Stage 1 - Normalisierung ... 34
5.7 Construct-Stage 2 - Linking ... 43
5.8 Construct-Stage 3 - Inference ... 44
5.9 Ontologie im Überblick ... 45
6 Demoanwendungen ... 47
6.1 Ontologie-Browser ... 47

6.2 Ontologie-Graph ... 47
6.3 Demo-App ... 48
6.4 Beispiel-Query ... 51
7 Theoretische Analyse ... 54
7.1 Laufzeit von CSV2RDF ... 54
7.2 Intra-Stage-Parallelisierung ... 55
7.3 End-to-End-Laufzeitanalyse ... 56
8 Empirische Analyse ... 58
8.1 Laufzeit von CSV2RDF ... 58
8.2 Intra-Stage-Parallelisierung ... 59
8.3 End-to-End-Laufzeitanalyse ... 61
9 Fazit und Ausblick ... 64
9.1 Verbesserungen der aktuellen Architektur ... 64
9.2 Workflow-Orchestration-Framework ... 65
9.3 Datenaggregation und Preprocessing ... 65
9.4 Datenvalidierung ... 66
9.5 Erweiterung der Datengrundlage ... 66
Danksagung ... 67
Bibliografie ... 68

1 Einleitung

Wälder sind komplexe Systeme, die sich im Laufe der Zeit kontinuierlich verändern. Insbesondere der Klimawandel beeinflusst die Waldökosysteme zunehmend negativ. Hitzeperioden, der Borkenkäfer und das Eschentriebsterben sind Schadfaktoren, die durch den Klimawandel verschärft werden. Daher wird es für den Waldbau immer wichtiger, Wälder an die sich wandelnden Umweltbedingungen anzupassen. Dafür müssen zahlreiche Faktoren, wie z.B. Standortsbedingungen, Bestandsdaten, Baumarten und klimatische Einflüsse, berücksichtigt und miteinander in Beziehung gesetzt werden [1].

Es existieren bereits viele verschiedene heterogene Datensätze über den Wald in Baden-Württemberg. Jedoch ist eine reine Masse an Daten nutzlos, wenn diese in isolierten und komplexen Datenbanksystemen vorliegen. Erst durch die Verknüpfung der Daten kann ihr volles Potenzial ausgeschöpft werden.

Genau hier setzt das Konzept von Wissensgraphen an. Diese ermöglichen die semantische Verknüpfung von Wissen in Form eines Graphen. Dabei werden Entitäten als Knoten und deren Beziehungen zueinander als Kanten dargestellt. Das Resource Description Framework (RDF) ist ein W3C-Standard zur Darstellung von Informationen als gerichteter und beschrifteter Graph [2], [3].

RDF-Wissensgraphen können aus bestehenden Datensätzen generiert werden. Dies ermöglicht die Verknüpfung von vorher verteiltem Wissen. Die Transformation verschiedener heterogener Datensätze aus verschiedenen Quellformaten in einen konsistent strukturierten Wissensgraphen ist jedoch nicht trivial. Die Daten weisen unterschiedliche Strukturen und Formate auf und müssen daher auf ein gemeinsames, semantisch konsistentes Datenmodell abgebildet werden.

Diese Arbeit ist im Kontext des WWW4.0-Projekts [4] (Wissen über den Wald im Wandel 4.0) entstanden, eines Zusammenschlusses aus Soziologie, Informatik und Fachleuten im Thema Forst. Das Ziel des Projektes ist, einen Wissensgraphen zum Thema Forst in Baden-Württemberg aus bestehenden Datensätzen zu entwickeln. Auf dem Wissensgraphen sollen dann verschiedene Anwendungen wie Apps oder Websites aufgebaut werden.

Das Ziel dieser Arbeit ist es, einen ersten Prototyp des Forstgraphen zu bauen. Dafür haben wir die RDF-Generierungspipeline Qonstructor entwickelt und mithilfe dieser den Forstgraphen generiert. In dieser Arbeit geht es sowohl um Qonstructor als auch den Entwicklungsprozess des Forstgraphen.

Qonstructor besteht aus mehreren Stages. In der ersten Stage werden Source-Dateien mit sogenannten Generator-Programmen in einen generischen RDF-Graphen umgewandelt. In den weiteren Stages wird der generische RDF-Graph durch SPARQL-Construct-Queries iterativ zum finalen Wissensgraphen transformiert und angereichert. SPARQL (SPARQL Protocol and RDF Query Language) ist eine Abfragesprache für RDF-Daten. Eine SPARQL-Construct-Query ist eine Abfrage an einen RDF-Graphen, die wiederum einen RDF-Graphen als Ergebnis liefert. Qonstructor ist nicht domänenspezifisch auf die Forst-Daten angepasst, sondern kann für beliebige Source-Daten genutzt und erweitert werden.

Der Forstgraph wird aus insgesamt 34 verschiedenen Source-Dateien generiert. Dafür haben wir 37 Construct-Queries geschrieben, welche den generischen Graphen iterativ zum Forstgraphen transformieren. Das Resultat ist eine vernetzte Repräsentation der Daten mit insgesamt circa 117 Millionen Triples.

Zur Reproduzierbarkeit unserer Arbeit und aller Ergebnisse haben wir im Git-Repository https://github.com/ad-freiburg/wald-4.0 alle entwickelten Skripte und Software zusammengestellt. Zusätzlich gibt es dort zu jedem Inhalt eine passende Anleitung.

Im Folgenden erläutern wir die Struktur dieser Arbeit. Zunächst erklären wir in Kapitel 2 die für die weitere Arbeit erforderlichen Grundlagen. Der Fokus liegt hier auf Wissensgraphen und verwandten Konzepten. Danach gehen wir in Kapitel 3 auf verschiedene Ansätze der RDF-Generierung ein und zeigen entsprechende Werkzeuge auf. In Kapitel 4 zeigen wir anschließend die Architektur und Funktionsweise von Qonstructor. Darauf aufbauend erklären wir in Kapitel 5 das konkrete Design und Bauen des Forstgraphen mithilfe von Qonstructor. In Kapitel 6 zeigen wir danach verschiedene Demoanwendungen, welche den Forstgraphen nutzen. Im Anschluss daran widmen wir uns in Kapitel 7 Aspekten wie Laufzeit und Parallelisierung verschiedener Teile der Pipeline. Als Nächstes überprüfen wir diese Theorien in Kapitel 8. Zuletzt geben wir in Kapitel 9 einen kurzen Rückblick auf die Arbeit und liefern mögliche Weiterentwicklungen sowie Forschungsmöglichkeiten.

2 Grundlagen

In diesem Kapitel erklären wir grundlegende Begriffe und Konzepte, welche für diese Arbeit relevant sind. Zusätzlich zeigen wir verwendete Software.

2.1 Geodaten

Große Teile der Source-Dateien des Forstgraphen sind Geodaten, also Daten, die einen räumlichen Bezug zur Erdoberfläche herstellen. Um geographische Positionen auf der Erde darzustellen, werden Koordinaten genutzt. Ein CRS [5] (Coordinate Reference System) legt dabei fest, wie geographische Positionen und Koordinaten zusammenhängen. Für diese Arbeit haben wir das CRS „WGS84“ [6] (World Geodetic System 1984) genutzt. Es gibt zwei Kategorien von Geodaten: Vektordaten und Rasterdaten.

Vektordaten bestehen aus diskreten Objekten mit Attributen und einer Geometrie. Attribute beschreiben dabei die Eigenschaften der Objekte. Die Geometrie beschreibt das Objekt räumlich als Punkt, Linie, Fläche oder eine Kombination daraus. Für den Forstgraphen haben wir Vektordaten in den Binär-Formaten Shapefile [7] und GeoPackage [8] bekommen, jedoch gibt es viele weitere Formate, in denen Vektordaten abgespeichert werden können. Abbildung 1 zeigt beispielhafte Vektordaten.

Rasterdaten stellen die Erdoberfläche als regelmäßiges Raster dar. Dabei kann jede Zelle in diesem Raster einen Wert wie z.B. Höhe oder Temperatur angeben. Ein Raster kann mehrere Bänder beinhalten. Pro Zelle und Band kann ein Wert eines bestimmten Datentyps gespeichert werden. TIFF [9] ist ein gängiges Format zum Speichern von Raster-/Bilddateien. GeoTIFF [10] ist eine Erweiterung von TIFF, die zusätzlich Georeferenzierungsinformationen enthält, um so einen geographischen Zusammenhang des Rasters zur Erdoberfläche zu ermöglichen. COG [11] (Cloud Optimized GeoTIFF) ist eine Optimierung des GeoTIFF-Standards, die es ermöglicht, Rasterdaten effizient über das Internet zu nutzen. Abbildung 2 zeigt beispielhafte Rasterdaten.

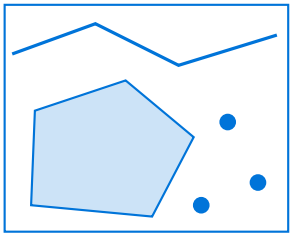

Abbildung 1: Vektordaten

10	25	40	55	60
20	45	70	60	45
35	65	85	50	30
30	50	55	35	15

Abbildung 2: Rasterdaten

2.2 Wissensgraphen

Schon in den 2000er Jahren wurden mit dem Semantic Web, Linked Data und RDF wichtige Grundlagen für Wissensgraphen geschaffen. 2007 wurde Freebase ins Leben gerufen, der erste große und öffentlich zugängliche Wissensgraph. Im Jahr 2012 hat Google seinen Wissensgraphen angekündigt, wodurch dieser Begriff populärer wurde. Danach sind auch andere große Firmen wie Airbnb, Amazon, eBay oder Facebook mit eigenen Wissensgraphen gefolgt. Das Thema bekam daraufhin aus akademischen Kreisen mehr Aufmerksamkeit und es wurden viele Publikationen dazu veröffentlicht [3], [12].

Um Wissensgraphen zu definieren, brauchen wir zuerst eine feste Definition des Begriffs „Wissen“. Hier stützen wir uns auf die Definition von „explizitem Wissen“, also Wissen, das aufgeschrieben werden kann. Ein Beispiel wäre eine einfache Aussage wie „Freiburg ist eine Stadt in Baden-Württemberg“ [3].

Es gibt keine allgemeingültige Definition von Wissensgraphen. Wir definieren einen Wissensgraphen als einen Graphen zur strukturierten Darstellung, Sammlung und Vermittlung von Wissen. Die Knoten des Graphen repräsentieren relevante Entitäten oder Werte. Die Kanten repräsentieren verschiedene Beziehungen zwischen diesen Entitäten [3].

Wissensgraphen können auf verschiedenen Graph-Modellen basieren. Wir konzentrieren uns hier explizit auf „Edge-labeled directed Graphs“, da diese die Basis von RDF-Graphen bilden. Das sind Graphen mit Knoten, welche über gerichtete und beschriftete Kanten miteinander verbunden sind. Die andere große Kategorie von Wissensgraphen sind Property-Graphs, welche für diese Arbeit aber nicht relevant sind [3].

Abbildung 3 zeigt einen beispielhaften Wissensgraphen, der ein „Edge-labeled directed Graph“ ist:

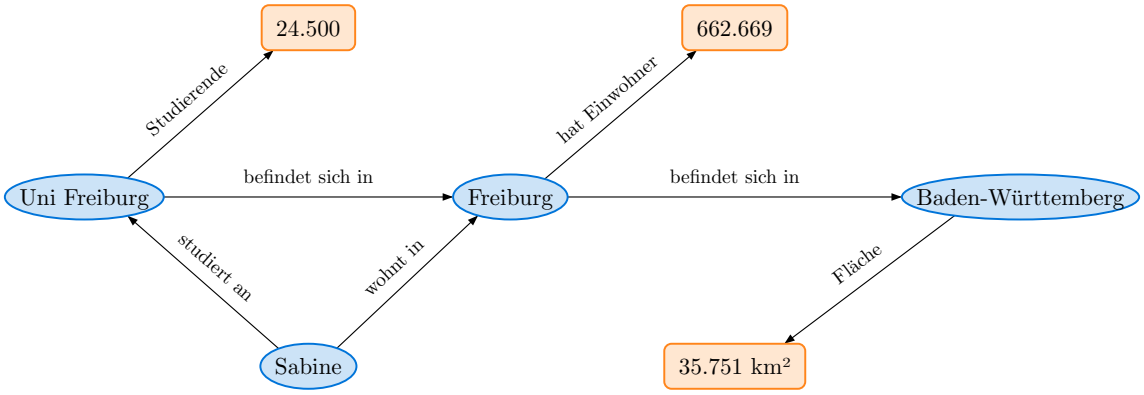

Abbildung 3: Beispielhafter Wissensgraph

2.3 RDF

In diesem Abschnitt beschäftigen wir uns mit dem Resource Description Framework (RDF) [2]. RDF ist ein W3C-Standard für Wissensgraphen, welcher das Graph-Modell „Edge-labeled directed Graph“ nutzt.

Um diese Graphen formal zu definieren, werden in RDF Triples genutzt. Ein Triple besteht dabei aus Subjekt, Prädikat und Objekt. Jedes Triple beschreibt eine Kante in dem Graphen. Das Subjekt ist der Ausgangsknoten, das Prädikat ist das Kantenlabel und das Objekt ist der Zielknoten:

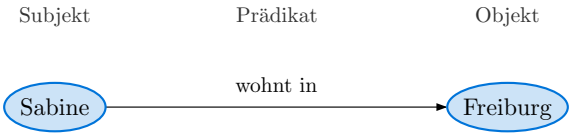

Abbildung 4: Darstellung eines Triples

Ein RDF-Graph besteht somit aus einer Menge an Triples. Dabei ist das Subjekt eine IRI oder eine Blank Node (Blank Nodes ignorieren wir hier). Das Prädikat ist immer eine IRI. Das Objekt ist eine IRI, ein Literal oder eine Blank Node. Im Folgenden führen wir die Begriffe IRI und Literal genauer ein [2].

2.3.1 IRI

Eine IRI (Internationalized Resource Identifier) ist ein Unicode-String nach [13] und wird oft umgeben von Angle-Brackets dargestellt. IRIs sind meist Web-URLs unter einer eindeutigen Domain und sind daher global einzigartig. Dann ist mit dieser IRI auf der gesamten Welt, in jedem Datensatz, dieselbe Resource gemeint. Unter einer IRI können im Browser oftmals Informationen zum durch die IRI referenzierten Objekt oder Prädikat angezeigt werden.

```
1 <https://example.org/Freiburg>
2 <https://wald.cs.uni-freiburg.de/ontology#Baumart_BAh>
3 <https://wald.cs.uni-freiburg.de/ontology#Bestand_0000937a206951c2>
```

Codeblock 1: Beispiel-IRIs

2.3.2 Literale

Literale werden für Werte wie Strings oder Zahlen genutzt. Ein Literal besteht aus einer lexikalischen Form und einer optionalen Datentyp-IRI. Die lexikalische Form ist eine Unicode-Repräsentation des Wertes. Die Datentyp-IRI definiert den Typ dieses Werts. Hier gibt es noch die Erweiterung von „Language-tagged Strings“, worauf wir aber nicht weiter eingehen, da der Forstgraph dieses Feature aktuell nicht nutzt [2].

```
"Freiburg"
"42"^^<http://www.w3.org/2001/XMLSchema#integer>
"3.14159"^^<http://www.w3.org/2001/XMLSchema#decimal>
```

Codeblock 2: Beispiel-Literale

2.3.3 Serialisierung

Ein RDF-Graph kann mit verschiedenen Standards serialisiert werden. Für diese Arbeit relevant sind die Formate N-Triples [14] und Turtle [15]. Ein N-Triples-Dokument enthält eine Aneinanderreihung von RDF-Triples. Jedes Triple wird durch `'.'` terminiert. Ansonsten können beliebige Leerzeichen und Zeilen hinzugefügt werden.

IRIs müssen in diesem Format umgeben von `'<'` und `'>'` angegeben werden. Literale werden als lexikalische Form umgeben von `'"'` dargestellt. An das Literal kann mit `'^^'` eine Datentyp-IRI angehängt werden.

```
<https://example.org/Freiburg> <https://example.org/befindetSichIn> <https://example.org/Badenwürttemberg> .
<https://example.org/Freiburg> <https://example.org/einwohner> "662669"^^<http://www.w3.org/2001/XMLSchema#integer> .
```

Codeblock 3: Beispieldokument im N-Triples-Format

Turtle [15] ist eine kompaktere, ausdrucksstärkere Serialisierung von RDF, welche N-Triples als Teilmenge enthält. Wir gehen hier nur auf zwei Features von Turtle ein, welche für die Arbeit relevant sind.

Am Anfang der Datei sind Präfixe definiert, welche dann in den Triples genutzt werden können, um IRIs abzukürzen (z.B. `ex:Freiburg` statt `<https://example.org/Freiburg>`).

Wenn mehrere aufeinanderfolgende Triples dasselbe Subjekt haben, muss das Subjekt nur im ersten Triple ausgeschrieben werden. Bei den folgenden Triples kann das Subjekt jeweils weggelassen werden. Die Triples werden dann untereinander mit einem `';'` statt einem `'.'` voneinander getrennt. Nur das letzte Triple wird durch ein `'.'` abgeschlossen.

```
@prefix ex: <https://example.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:Freiburg ex:befindetSichIn ex:Badenwürttemberg ;
  ex:einwohner "662669"^^xsd:integer .
```

Codeblock 4: Beispieldokument im Turtle-Format

2.3.4 Vokabulare

Ein Vokabular legt fest, welche Begriffe und Beziehungen in einem RDF-Graphen genutzt werden können. RDF ist hauptsächlich ein Standard zur Modellierung von Informationen als gerichteter Graph. Jedoch beinhaltet RDF auch das RDF-Vokabular. Für diese Arbeit

nutzen wir daraus nur das Prädikat `rdf:type`, das beschreibt, zu welcher Klasse ein Objekt gehört [2].

Es gibt verschiedene weitere Vokabulare, wie z.B. RDFS [16] oder GeoSPARQL [17], welche auf dem RDF-Standard aufbauen und ihn um weitere Begriffe und Funktionen zur semantischen Modellierung von Informationen erweitern.

RDFS erweitert das RDF-Vokabular um Begriffe zur Beschreibung von Klassen, Eigenschaften und deren Beziehungen untereinander. Wir nutzen RDFS für folgende Modellierungsmöglichkeiten:

```
@prefix ex: <https://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

ex:Freiburg rdf:type ex:Stadt .
ex:Stadt rdf:type rdfs:Class .
ex:Freiburg rdfs:label "Freiburg im Breisgau" .
```

Codeblock 5: Modellierungsbeispiel mit RDFS

GeoSPARQL stellt ein Konzept zum Arbeiten mit Geodaten wie z.B. WKT-Geometrien in RDF-Graphen bereit. WKT-Geometrien [18] sind eine standardisierte Textform für geometrische Objekte. Wir nutzen GeoSPARQL für folgende Modellierungsmöglichkeiten:

```
@prefix ex: <https://example.org/> .
@prefix geo:  <http://www.opengis.net/ont/geosparql#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

ex:Freiburg ex:befindetSichIn ex:Badenwürttemberg ;
  geo:hasGeometry ex:Freiburg_Geometry .
ex:Freiburg_Geometry rdf:type geo:Geometry ;
  geo:asWKT "POLYGON((0 0, 10 0, 10 10, 0 10, 0 0))"^^geo:wktLiteral .
```

Codeblock 6: Modellierungsbeispiel mit GeoSPARQL

2.4 SPARQL

SPARQL [19] ist eine Query-Sprache für RDF-Graphen. SPARQL kann RDF-Graphen nach optionalen oder erforderlichen Graph-Patterns durchsuchen. Außerdem kann nach Konjunktion oder Disjunktion mehrerer Graph-Patterns gesucht werden. Zusätzlich werden Features wie z.B. Aggregation, Subqueries, Negation und ausgeprägte Wertetests unterstützt.

Es gibt für SPARQL-Queries verschiedene Query-Formen. Die für uns relevanten Query-Formen sind `SELECT` und `CONSTRUCT`. Wir führen beide Formen ein.

2.4.1 Select-Query

Codeblock 7 zeigt eine exemplarische SPARQL-Select-Query. Im Körper des Where-Blocks befindet sich das Graph-Pattern, welches wiederum aus vier Triple-Patterns besteht. Die Elemente mit einem führenden Fragezeichen sind Variablen. In diesem Beispiel wird nach beliebigen Entitäten gesucht, die vom Typ `ex:Stadt` sind und sich in `ex:Badenwürttemberg` befinden. Zudem soll die Stadt ein WKT-Literal besitzen. Dabei bedeutet `geo:hasGeometry/geo:asWKT`, dass über die beiden Prädikate `geo:hasGeometry` und `geo:asWKT` traversiert wird. Somit ist `?stadt geo:hasGeometry/geo:asWKT ?wkt` eine Abkürzung für `?stadt geo:hasGeometry ?geo` und `?geo geo:asWKT ?wkt`.

Beim Ausführen der SPARQL-Select-Query wird das Graph-Pattern gegen den RDF-Graphen abgeglichen und nach allen passenden Matches gesucht. Es wird also nach allen Subgraphen gesucht, bei denen die Triple-Patterns übereinstimmen. Für jedes Match gibt es ein Solution-Mapping, also eine Belegung aller Variablen. Diese Solution-Mappings können in einer Tabelle, mit den Variablennamen als Spaltennamen, dargestellt werden. Tabelle 1 zeigt beispielhafte Ergebnisse der Select-Query.

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  PREFIX geo: <http://www.opengis.net/ont/geosparql#>
3  PREFIX ex: <https://example.org/>
4
5  SELECT * WHERE {
6    ?stadt rdf:type ex:Stadt .
7    ?stadt ex:befindetSichIn ex:Badenwürttemberg .
8    ?stadt ex:einwohner ?einwohner .
9    ?stadt geo:hasGeometry/geo:asWKT ?wkt .
10 }
```

Codeblock 7: Beispielhafte SPARQL-SELECT-Query

?stadt	?einwohner	?wkt
ex:Karlsruhe	307.000	POLYGON((...))
ex:Stuttgart	630.000	POLYGON((...))
ex:Mannheim	314.000	POLYGON((...))

Tabelle 1: Beispielhaftes Ergebnis der Select-Query

2.4.2 Construct-Query

Die Query in Codeblock 8 ersetzt den Select-Block durch einen Construct-Block. Der Construct-Block definiert ein Template, das aus beliebigen Triple-Patterns besteht. Diese Triple-Patterns können Variablen aus dem Where-Block beinhalten. Damit ist diese Query eine Construct-Query.

Die Ausführung dieser Construct-Query ist bis zu den Solution-Mappings gleich wie die Ausführung einer Select-Query. Danach wird für jedes einzelne Solution-Mapping das Template ausgefüllt. Die daraus entstehenden Triples bilden dann selbst wieder einen RDF-Graphen. In dieser Arbeit nutzen wir Construct-Queries, um iterativ RDF-Daten zu transformieren. Codeblock 9 zeigt ein beispielhaftes Ergebnis der Query.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX ex: <https://example.org/>

CONSTRUCT {
  ?stadt ex:einwohner ?einwohner .
  ?stadt ex:wkt ?wkt .
} WHERE {
  ?stadt rdf:type ex:Stadt .
  ?stadt ex:befindetSichIn ex:Badenwürttemberg .
  ?stadt ex:einwohner ?einwohner .
  ?stadt geo:hasGeometry/geo:asWKT ?wkt .
}
```

Codeblock 8: Beispielhafte SPARQL-Construct-Query

```
ex:Karlsruhe ex:einwohner "307000" .
ex:Karlsruhe ex:wkt "POLYGON (...)" .
ex:Stuttgart ex:einwohner "630000" .
ex:Stuttgart ex:wkt "POLYGON (...)" .
ex:Mannheim ex:einwohner "314000" .
ex:Mannheim ex:wkt "POLYGON (...)" .
...
```

Codeblock 9: Beispielhaftes Ergebnis der Construct-Query

2.4.3 GeoSPARQL

Es gibt verschiedene Erweiterungen zu SPARQL. Eine davon ist GeoSPARQL [17], das in Abschnitt 2.3.4 als Vokabular eingeführt wird. Jedoch erweitert der GeoSPARQL-Standard auch SPARQL selbst um Funktionen, die das Arbeiten mit geometrischen Objekten erleichtern. Eine Funktion ist z.B. `geof:distance(geom1, geom2, unit)`, welche die Distanz zweier geometrischer Objekte berechnet.

2.5 Verwendete Software

Wir haben für diese Arbeit verschiedene Software genutzt, um einfacher und effizienter mit RDF-Graphen sowie den Rohdaten arbeiten zu können. Diese stellen wir hier kurz vor.

2.5.1 QGIS

Wir haben das Open-Source-GIS-Programm (Geoinformationssystem) QGIS [20] genutzt, um verschiedene Geodatenformate darzustellen. Dabei können mehrere Layer geladen und visualisiert werden. Dieses Tool ist vor allem relevant, um Source-Daten besser zu verstehen, um dann RDF daraus zu modellieren.

2.5.2 QLever

Der wichtigste Ausgangspunkt, um mit RDF-Daten arbeiten zu können, ist ein RDF-Store. Für diese Arbeit haben wir QLever [21] genutzt, eine vom „Lehrstuhl Algorithmen und Datenstrukturen“ an der Universität Freiburg entwickelte Open-Source-Software. QLever ermöglicht das Laden von RDF-Daten in Form von Turtle oder N-Triples und stellt eine SPARQL-Schnittstelle zum geladenen RDF-Graphen bereit.

QLever besteht aus zwei ausführbaren Dateien. Der `qlever-index` baut die für QLever nötige Indexstruktur, welche dann von `qlever-server` für das effiziente Bearbeiten von SPARQL-Queries genutzt wird. Es gibt ein Python-CLI-Paket mit gleichem Namen „qlever“, das als benutzerfreundliches CLI-Interface dient.

Die QLever-CLI nutzt ein „Qleverfile“ als zentrale Konfigurationsdatei. Jedoch kann auch alles über Kommandozeilenargumente konfiguriert werden.

2.5.3 QLever-UI

Als Oberfläche zum Schreiben und Ausführen von SPARQL-Queries haben wir die QLever-UI [22] genutzt, eine Web-Anwendung mit Features wie Autocompletion, Linting, Tabs und Query-Analyse.

2.5.4 Petrimaps

Um Geometrien aus dem RDF-Graphen zu visualisieren, haben wir Petrimaps [23] genutzt. Petrimaps lädt Geometrien aus einem RDF-Graphen im Vorhinein in den Arbeitsspeicher und rendert auf Anfrage die Geometrien serverseitig. Das Frontend basiert auf Leaflet [24] und stellt die gerenderten Tiles vom Backend auf der Karte dar. Diese Architektur ermöglicht das schnelle Darstellen von Millionen von Geometrien.

2.5.5 GDAL

Um effizient mit Geodaten arbeiten zu können, haben wir GDAL [25] genutzt. GDAL ist eine C++-Bibliothek, die einen Abstraktionslayer für Geodaten wie z.B. Shapefiles, Geodatabases oder Rasterformate bildet. GDAL beinhaltet mehrere Kommandozeilenprogramme wie z.B. „ogr2ogr“ oder „gdalwarp“, welche in Abschnitt 5.3 benutzt werden. Außerdem gibt es Wrapper-Libraries für Python.

2.5.6 TIFF-Proxy

Es gibt keinen etablierten Standard für die Integration von Rasterdaten in RDF-Wissensgraphen. Der einfachste Weg ist, alle Rasterzellen zu materialisieren und direkt in den Wissensgraphen zu laden. Jedoch wird das ab einer bestimmten Größe von Rasterdaten

durch die Menge an generierten Triples ineffizient. Daher haben wir den TIFF-Proxy entwickelt, einen HTTP-Server, der On-Demand Pixelwerte aus TIFF-Dateien ausliest und zurückgibt. Per HTTP-Request können Geometrien und Rasterdatensatz-Bezeichner übergeben werden, worauf der Server mit allen Rasterzellen innerhalb dieser Geometrien für die jeweiligen Rasterdatensätze antwortet. QLever hat den Feature-Branch `qlproxy-service`, der den sogenannten „Magic Service“ `qlproxy` implementiert. So können innerhalb von SPARQL-Queries Anfragen an den TIFF-Proxy gestellt werden. Die Integration von Rasterdaten in QLever befindet sich derzeit noch in Entwicklung, weshalb wir in dieser Arbeit nicht genauer auf die Implementierungsdetails eingehen.

3 Verwandte Arbeiten

Der Hauptteil unserer Arbeit kann in die Teile Qonstructor und das Bauen des Forstgraphen aufgeteilt werden. Wir beschäftigen uns in diesem Kapitel mit verwandten Arbeiten zu Qonstructor. Wir sehen das Bauen des Forstgraphen als Anwendungsfall von Qonstructor und führen hier daher keine verwandten Arbeiten zum Forstgraphen auf.

Es gibt hauptsächlich drei Wege, aus strukturierten Daten RDF zu generieren. Die konzeptionell einfachste Möglichkeit ist es, ein einfaches Programm oder Skript zu schreiben, welches die Daten nach einem bestimmten Schema in RDF umwandelt. Wir nennen diesen Ansatz den programmatischen Ansatz. Des Weiteren gibt es RDF-Mapping-Languages, die mithilfe von formalen Mappings definieren, wie Rohdateien in RDF umgewandelt werden sollen. Die letzte Möglichkeit ist es, die RDF-Query-Language SPARQL zu nutzen, um die Daten zu verarbeiten.

Im Folgenden schauen wir uns verschiedene Projekte aus diesen drei Kategorien an. Wir zeigen für jedes Tool die Unterschiede zu unserer Arbeit auf.

3.1 Programmatischer Ansatz

Der programmatische Ansatz ist der allgemein flexibelste Ansatz. Eingabedaten können per Skript oder Programm beliebig zu RDF weiterverarbeitet werden. Ein einfaches Beispiel wäre hier ein Skript, das CSV-Dateien mit fest definiertem Schema in RDF umwandelt.

Jedoch gibt es bei diesem Ansatz ein paar offensichtliche Nachteile. Die Logik des Mappings ist im Code selbst vergraben. Es wird also immer ein Programmierer benötigt, der den Code versteht, um Änderungen am Mapping vorzunehmen.

Für diese Kategorie geben wir kein konkretes Tool an, da solche Tools meist auf bestimmte Use-Cases spezialisiert sind. Jedoch kann das Generieren von RDF im Code selbst leichter gestaltet werden, indem verschiedene Bibliotheken genutzt werden. Für Python ist RDFLib [26] der Standard. RDFLib hat einen weitaus größeren Funktionsumfang als das reine Schreiben von RDF-Graphen. Es können z.B. auch RDF-Graphen eingelesen und SPARQL-Queries ausgeführt werden.

In unserer Arbeit nutzt Qonstructor den programmatischen Ansatz für die Generic-Generation-Stage. Darauf gehen wir in Abschnitt 3.4 noch genauer ein.

3.2 RDF-Mapping-Standards

Es gibt verschiedene RDF-Mapping-Standards. R2RML, RML und YARRRML sind die bekanntesten und bauen alle aufeinander auf. Im Folgenden gehen wir genauer auf diese Standards ein.

3.2.1 R2RML

R2RML [27] ist eine W3C-Sprache, die definiert, wie relationale Datenbanken in RDF umgewandelt werden. Die Mappings werden im Turtle-Format definiert. Es handelt sich hier um einen reinen Standard, der selbst noch keine Software beinhaltet.

Der klare Nachteil dieses Standards ist, dass nur relationale Datenbanken unterstützt werden. Dieser Nachteil wird durch den folgenden Standard behoben.

3.2.2 RML

RML (RDF-Mapping-Language) [28] ist eine Generalisierung von R2RML. Der größte Unterschied ist, dass RML auch mit anderen Datenformaten wie z.B. CSV-Dateien oder JSON-Dateien funktioniert. Die Mappings werden auch hier im Turtle-Format definiert.

Morph-KGC [29] ist eine performante Python-Implementierung des RML-Standards. Da RML eine Generalisierung von R2RML ist, unterstützt Morph-KGC auch den R2RML-Standard.

Ein Nachteil von Mappings im Turtle-Format ist, dass diese sehr ausführlich sind und schnell unübersichtlich werden können. Hier schafft der folgende Wrapper Abhilfe.

3.2.3 YARRRML

YARRRML [30] ist ein Wrapper um den RML-Standard, der es ermöglicht, die RML-Mappings in YAML zu definieren. Dafür gibt es z.B. den YARRRML-Parser, welcher Transpiling von YARRRML zu RML und umgekehrt ermöglicht. Wichtig zu erwähnen ist, dass es noch weitere RML-Erweiterungen gibt, auf die wir hier nicht weiter eingehen.

Der Nachteil von YARRRML im Vergleich zu RML ist, dass ein zusätzlicher Transpiling-Schritt von YARRRML zu RML nötig ist, um aus den Eingabedaten RDF zu generieren. Jedoch hat die Morph-KGC-Implementierung diesen Schritt fest eingebaut.

Da R2RML, RML und YARRRML alle aufeinander aufbauen, sprechen wir in diesem Abschnitt allgemein von RML, um die Unterschiede zu unserer Arbeit aufzuzeigen.

RML nutzt formale Mappings, um aus Source-Dateien verschiedener Formate einen RDF-Graphen aufzubauen. Es ist auch möglich, zwischen verschiedenen Dateien Linking zu erstellen.

RML kann nur Source-Dateien in RDF umwandeln, aber nicht RDF zu RDF. Qonstructor ist so entworfen, dass in iterativen Schritten das RDF immer weiter angereichert werden kann. Dafür werden SPARQL-Construct-Queries statt statischer Mappings genutzt. SPARQL-Construct-Queries bieten außerdem viele Features, welche mit den statischen Mappings von RML nicht möglich sind. Im Folgenden gehen wir auf andere SPARQL-Construct-basierte Tools ein.

3.3 SPARQL-basierte RDF-Generierung

SPARQL ist eine vollwertige Abfragesprache für RDF-Graphen. Die Features von SPARQL können durch SPARQL-Construct-Queries genutzt werden, um neue RDF-Graphen zu bauen. Beispiel-Features sind Aggregation, Filter, Group By, Datentyp-Casting, Subqueries und Value Clauses [19]. Im RML-Mapping-Standard gibt es dafür zum Teil keine Alternativen.

3.3.1 TARQL

TARQL [31] ist ein Kommandozeilenprogramm, das Tabellendaten in Form von CSV oder TSV per SPARQL-Query in RDF umwandeln kann. Es können in der SPARQL-Query selbst die Rohdaten definiert und abgefragt werden.

Der Nachteil von TARQL ist, dass es klar auf das CSV- und TSV-Format begrenzt ist. Zudem ist es darauf ausgelegt, einzelne Dateien zu bearbeiten.

3.3.2 SPARQL-Anything

SPARQL-Anything [32] verallgemeinert den Ansatz von TARQL und unterstützt deutlich mehr Dateiformate (XML, JSON, CSV, HTML, Excel, Text, Binary, EXIF, File System, Zip/Tar, Markdown, YAML, Bibtex, DOCX). Alle Dateiformate werden über das einheitliche Facade-X-Modell virtuell als RDF dargestellt. Dieses virtuelle RDF kann dann per SPARQL-Query abgefragt werden. Construct-Queries generieren den RDF-Output.

SPARQL-Anything ermöglicht es im Vergleich zu TARQL, mehrere Source-Dateien zu definieren und in einer Construct-Query zu nutzen.

Da TARQL und SPARQL-Anything vom Konzept her sehr ähnlich sind, gehen wir hier gemeinsam auf die Unterschiede zu dieser Arbeit ein. Auch diese beiden Tools sind dafür ausgelegt, in einem Schritt Rohdaten zu RDF umzuwandeln. Es gibt keinen eingeplanten

Workflow, um mehrere Construct-Queries ausführen zu können und einen gemeinsamen RDF-Graphen daraus zu erstellen. Qonstructor ist im Gegensatz dazu darauf ausgelegt, mehrere Construct-Queries in mehreren iterativen Schritten parallel auszuführen.

3.4 Motivation der Entwicklung von Qonstructor

Die Tools aus den vorherigen Abschnitten sind eher Minimalansätze, welche einzelne elementare Funktionen implementieren. Sie liefern jedoch kein Gesamtkonzept für die Generierung komplexer Wissensgraphen. Diese Tools könnten aber für einzelne Schritte in einer RDF-Generierungspipeline genutzt werden. Qonstructor ist im Gegensatz dazu eine „opinionated“ Pipeline zur einfachen Erstellung von Wissensgraphen in mehreren Schritten. Der Workflow für die iterative Entwicklung von Wissensgraphen ist hier im Design mit inbegriffen. Um das zu ermöglichen, sind die SPARQL-Engine QLever sowie die QLever-UI fest in Qonstructor integriert. So können generierte RDF-Daten direkt nach der Generierung angeschaut und verifiziert werden. Trotzdem werden verschiedene Ideen und Ansätze der Tools von Qonstructor auch genutzt.

4 Qonstructor

Das finale Ziel unserer Arbeit ist das Bauen eines Wissensgraphen zum Thema Waldbau. Dafür haben wir die RDF-Generierungspipeline Qonstructor entwickelt. In Abschnitt 3.4 gehen wir konkret darauf ein, warum bestehende Tools wie RML oder SPARQL-Anything für diese Aufgabe nicht ausgereicht haben.

In diesem Kapitel zeigen wir zunächst die Designziele (Abschnitt 4.1) und Systemarchitektur (Abschnitt 4.2) von Qonstructor. Danach erklären wir die Konfiguration (Abschnitt 4.3), die verschiedenen Stages (Abschnitt 4.4, Abschnitt 4.5 und Abschnitt 4.6) und die QLever-Integration (Abschnitt 4.7) genauer. Zuletzt gehen wir auf die Containerisierung ein und zeigen zwei beispielhafte Nutzungsworkflows (Abschnitt 4.9).

4.1 Designziele

Das Ziel von Qonstructor ist es, aus Source-Dateien in diversen Formaten einen resultierenden RDF-Graphen zu generieren. Dafür haben wir verschiedene Designziele festgelegt:

1. **Domänenunabhängig:** Qonstructor soll nicht auf unseren Use-Case zugeschnitten sein, sondern für verschiedene Domänen genutzt werden können.
2. **Erweiterbarkeit ohne Codeänderungen:** Neue Daten sollen möglichst einfach über Konfigurationsdateien in den Wissensgraphen eingegliedert werden können.
3. **Trennung von Parsing und Semantik:** Das Parsen der Source-Dateien soll getrennt sein von den Schritten, die das RDF semantisch anreichern.
4. **Iterative Generierung:** Die Daten sollen in klar nachvollziehbaren, sequentiellen Schritten transformiert und angereichert werden können.
5. **Verifizierbarkeit nach jeder Stage:** Die resultierenden Daten jedes Schrittes sollen einzeln abfragbar sein, um diese verifizieren zu können.
6. **Reproduzierbarkeit:** Qonstructor soll auf verschiedenen Maschinen ohne viel Aufwand mit derselben Eingabe denselben Wissensgraphen generieren.

4.2 Systemarchitektur

Qonstructor ist ein Python-CLI-Programm, das verschiedene Prozesse orchestriert. Bei jedem Start von Qonstructor werden die Konfigurationsdateien eingelesen und ein Cleaning-Schritt durchlaufen. Danach werden die sequentiellen Stages ausgeführt, welche mit 0 startend durchnummeriert sind (Designziel 4).

Stage_0 ist die Generic-Generation-Stage. Hier werden beliebige Source-Dateien mit sogenannten Generator-Programmen in generisches RDF umgewandelt. Wir definieren RDF_0 als die Vereinigung aller generierten generischen RDF-Dateien. Jedes Generator-Programm ist für ein bestimmtes Dateiformat gedacht. Es können einfach neue Generator-Programme in Qonstructor integriert werden. Welche Source-Dateien mit welchen Generator-Programmen umgewandelt werden, wird in den Konfigurationsdateien definiert (Designziel 2). Aus dem generischen RDF_0 wird dann ein QLever-Index gebaut und ein QLever-Server gestartet. Wir nennen den Wissensgraphen, der von diesem QLever-Server bereitgestellt wird, Graph_0. Abschnitt 4.4 erklärt die Generic-Generation-Stage genauer.

$\text{Stage}_i, i \in [1, N]$ sind die Construct-Stages. Jede Construct-Stage besteht aus einer Menge von Construct-Queries, die in den Konfigurationsdateien definiert werden (Designziel 2). Jede Construct-Stage Stage_i bekommt als Eingabe einen Wissensgraphen Graph_{i-1} der vorherigen Stage und führt darauf die Construct-Queries aus. Alle Ergebnisse der Queries kombiniert ergeben RDF_i. Aus dem resultierenden RDF_i sowie dem RDF aller vorherigen Construct-Stages $\cup_{j=1}^{i-1} \text{RDF}_j$ wird ein neuer QLever-Index gebaut und ein QLever-Server gestartet. Dieser QLever-Server stellt den Wissensgraphen Graph_i bereit. Wir nennen die Aggregation vorheriger RDF-Ergebnisse RDF-Forwarding. Der beschriebene Prozess wird sequentiell für alle Construct-Stages von 1 bis N durchgeführt. Das Endergebnis aller Construct-Stages ist der Wissensgraph Graph_N. Abschnitt 4.5 erklärt die Construct-Stages genauer.

Stage_{N+1} ist die Export-Stage. Hier wird eine einzelne Construct-Query auf dem letzten Graphen Graph_N der Construct-Stages ausgeführt. Die Export-Query kann in den Konfigurationsdateien definiert werden. Das Ergebnis dieser Stage ist $\text{RDF}_{\text{Export}}$. Abschnitt 4.6 geht genauer auf die Export-Stage ein.

Stage_0 ist die einzige Stage, die Source-Dateien verarbeitet. RDF_0 wird nicht in die nächsten Wissensgraphen geforwardet. Die erste Construct-Stage muss also alle Informationen aus den generischen RDF-Daten, die in die nächsten Stages kommen sollen, per Construct-Queries an Graph_0 generieren. Dadurch ist eine klare Trennung zwischen generischem RDF und modelliertem RDF gewährleistet (Designziel 3).

Es gibt zwei verschiedene Gründe, warum Qonstructor mehr als eine Construct-Stage braucht. Der erste Grund ist, dass es Queries geben kann, die wiederum auf das Ergebnis vorheriger Queries angewiesen sind. Dann müssen die Queries entsprechend ihrer Abhängigkeit in verschiedenen Stages ausgeführt werden. Der zweite Grund ist die dadurch ermöglichte Trennung konzeptionell verschiedener Queries für mehr Struktur.

Am Ende jeder Stage $\mathbf{Stage}_i, i \in [0, N]$ wird ein QLever-Server gestartet, der jeweils $\mathbf{Graph}_i$ zur Verfügung stellt. Diese QLever-Server bleiben über die Pipeline hinweg bestehen und können mit der QLever-UI abgefragt werden. So ist eine Verifizierung der Daten nach jeder Stage möglich (Designziel 5).

In Abbildung 5 ist die Pipeline-Architektur visualisiert. Links sind die Source-Dateien und rechts das exportierte RDF.

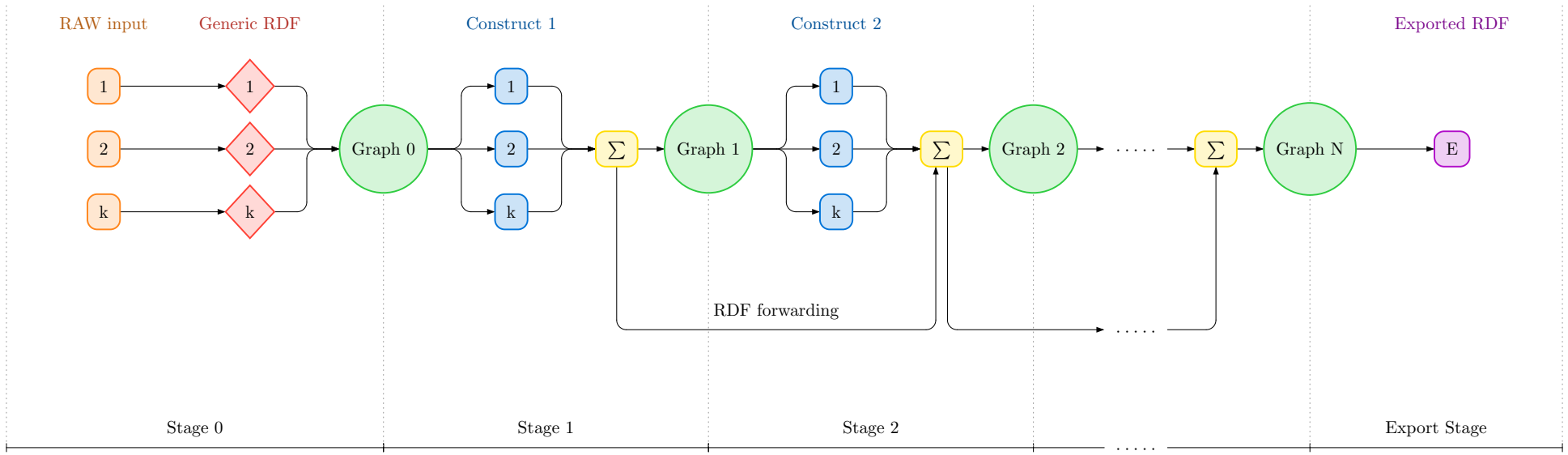

Abbildung 5: Qonstructor-Pipeline-Architektur

4.3 Konfiguration

Die gesamte Persistenz von Qonstructor befindet sich innerhalb eines Ordners oder unter einem symbolischen Link mit dem Namen `workspace`. Der Workspace beinhaltet die Source-Dateien, die Pipeline-Konfigurationsdatei und die Stage-Konfigurationsdateien. Während der Laufzeit von Qonstructor werden außerdem verschiedene Dateien dort abgelegt.

Qonstructor hat zwei verschiedene Konfigurationsebenen. Es gibt die Pipeline-Konfigurationsdatei `config.yaml`, welche im Workspace liegt. Diese ist die Haupt-Konfigurationsdatei. Zusätzlich gibt es die Stage-Konfigurationsdateien, welche in einer beliebigen Ordnerstruktur innerhalb des Unterordners `configs` liegen. Diese definieren, welche Daten wie in RDF umgewandelt werden sollen.

Bei jedem Start der Qonstructor-Pipeline werden die Pipeline-Konfigurationsdatei sowie alle Stage-Konfigurationsdateien geparst und validiert. Wenn während der Validierung der Stage-Konfigurationsdateien Fehler auftreten, werden diese über alle Konfigurationsdateien hinweg gesammelt und ausgegeben. Nur wenn alle Konfigurationsdateien erfolgreich validiert wurden, läuft die Pipeline nach diesem Schritt weiter.

4.4 Generic-Generation-Stage

In der Generic-Generation-Stage werden die Source-Dateien in generisches RDF umgewandelt. Dafür wird auf jeder Source-Datei einer der definierten Generatoren ausgeführt.

In einer Stage-Konfigurationsdatei können Sources unter dem Schlüssel `sources` definiert werden. Eine Source definiert, wie eine Source-Datei in generisches RDF umgewandelt werden soll. Jede Source hat die Schlüssel `name`, `path` und `type`. Der `name` ist der Name der Source und wird als Dateiname zum Speichern des RDF genutzt. Der `path` ist der Pfad der Source-Datei. Der `type` wählt einen Generator, welcher für die Erstellung des generischen RDF genutzt wird. Optional können unter dem Schlüssel `config` Schlüssel-Wert-Paare gesetzt werden, welche dann als Kommandozeilen-Flags an den Generator mitgegeben werden können. Codeblock 10 zeigt beispielhafte Sources einer Stage-Konfigurationsdatei.

```yaml
sources:
  - name: Baumarten
    path: lfv/Baumarten/Baumartenliste.csv
    type: csv
  - name: Standortsdatenblätter_comma_seperated
    path: fva/Standortsdatenblätter/Standortsdatenblätter.csv
    type: csv
    config:
      sub-delimiter: ','
```

Codeblock 10: Auswahl eines Generators über den Source-Type

Ein Generator ist ein Template-String, der mit den Variablen `name`, `source_file`, `target_file` und `flags` gefüllt wird. Ein ausgefüllter Template-String ist dann ein Shell-Kommando, das eine Source-Datei in eine Gzip-komprimierte Turtle-Datei umwandelt. In der Pipeline-Konfigurationsdatei können die Generatoren unter dem Schlüssel `generators` definiert werden. Codeblock 11 zeigt die Generatoren aus der Default-Pipeline-Konfigurationsdatei.

```
generators:
  csv: "csv2rdf2 {{ flags }} -n {{ name }} {{ source_file }} | gzip -1 -c > {{ target_file }}"
  json: "json2rdf {{ source_file }} | gzip -1 -c > {{ target_file }}"
  tif: "tif2rdf {{ source_file }} | gzip -1 -c > {{ target_file }}"
```

Codeblock 11: Default-Generatoren der Pipeline-Konfigurationsdatei

Ein Generator kann beliebige Programme innerhalb des Shell-Kommandos ausführen. Wir nennen diese Programme Generator-Programme. Ein Generator-Programm bekommt als Eingabe einen Dateipfad und einen Datensatznamen und streamt das resultierende RDF als Turtle über stdout (Standard Output). Die Generatoren pipen die Ausgabe der Generator-Programme dann in Gzip und speichern das Ergebnis ab.

Im Folgenden gehen wir auf die von uns entwickelten Generator-Programme CSV2RDF, JSON2RDF und TIF2RDF ein. Wir haben diese Shell-Programme in Qonstructor eingebaut, da wir entsprechende Dateiformate als Source-Dateien haben.

4.4.1 Mapping-Grundstruktur

Wir haben für die Generatoren eine Mapping-Grundstruktur entwickelt. Diese Grundstruktur kann auch von weiteren Generatoren implementiert werden. Dieser Ansatz hat den Vorteil, dass alle Daten aus unterschiedlichen Source-Formaten im generischen RDF eine einheitliche Grundstruktur haben. Somit ist das Arbeiten mit diversen Datenformaten einfacher.

Für die Grundstruktur benötigen wir die folgenden RDF-Präfixe:

1. `@prefix gen: <generic#>`: wird als Präfix für rein strukturelle Prädikate genutzt (z.B. `gen:child`)
2. `@prefix dat: <data#>`: wird als Präfix für inhaltliche Prädikate genutzt (z.B. `dat:<Spaltenname>`)

Um eine Datei in RDF darzustellen, braucht es immer ein Objekt der Klasse `gen:Root`. Dieses Objekt nutzt das Präfix `gen` (z.B. `gen:Dateiname`) und fungiert als eine Art Anker für alle anschließenden Daten.

Jedes Objekt kann rekursiv mit folgenden Regeln erweitert werden:

1. Dictionary: Jedes Objekt kann über das Prädikat `dat:<key>` entweder Literale oder weitere Objekte per Schlüssel-Wert-Zuweisung anhängen. Dann ist `<key>` der Schlüssel und das Literal oder Objekt der Wert.

2. Kollektion: Jedes Objekt kann über das Prädikat `gen:child` Elemente einer Kollektion speichern. Die Elemente können entweder Literale oder wieder Objekte sein. Wenn die Elemente Objekte sind, kann ihnen über das Prädikat `gen:id` eine Reihenfolge zugewiesen werden. Somit wird die Kollektion zu einer Liste.

Wie genau die IRIs der Objekte generiert werden, damit es keine Kollisionen gibt, bleibt dabei der Implementierung überlassen. Codeblock 12 zeigt diese Grundstruktur:

```
@prefix dat: <data#> .
@prefix gen: <generic#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

gen:Beispiel rdf:type gen:Root .

# Dictionary: Literal
gen:Beispiel dat:titel "Hallo Welt" .

# Dictionary: verschachteltes Objekt
gen:Beispiel dat:autor gen:Beispiel_autor .
gen:Beispiel_autor dat:name "Max Mustermann" .

# Kollektion: Literale (kein gen:id)
gen:Beispiel gen:child "Tag1" .
gen:Beispiel gen:child "Tag2" .

# Liste: Objekte (gen:id gibt die Reihenfolge)
gen:Beispiel gen:child gen:Beispiel_1 .
gen:Beispiel_1 gen:id "1"^^xsd:int ;
  dat:wert "Erster Eintrag" .

gen:Beispiel gen:child gen:Beispiel_2 .
gen:Beispiel_2 gen:id "2"^^xsd:int ;
  dat:wert "Zweiter Eintrag" .
```

Codeblock 12: Beispiel für die Mapping-Grundstruktur

4.4.2 CSV2RDF

CSV (Comma-Separated Values) ist ein textbasiertes Tabellenformat. Um CSV-Dateien in generisches RDF umzuwandeln, haben wir das Shell-Programm CSV2RDF entwickelt, das die vorher definierte Mapping-Grundstruktur nutzt.

Eine CSV-Datei kann einfach mit der Grundstruktur dargestellt werden. Es gibt ein Root-Objekt, das eine Liste von Row-Objekten beinhaltet. Die Row-Objekte fungieren als Dictionaries mit den Spaltennamen der Tabelle als Schlüssel und den Inhalten der Felder

als Werte. Codeblock 13 ist eine CSV-Datei, die von CSV2RDF zum in Codeblock 14 gezeigten RDF umgewandelt wird.

```
Abkürzung,Artname,Gruppe,WET-Zuordnung
Abg,Küstentanne,Tannen,Tannen-Mischwald
Abi,sonstige Tannenart,Tannen,Tannen-Mischwald
```

Codeblock 13: Beispielhafte CSV-Eingabedatei

```
@prefix gen: <generic#> .
@prefix dat: <data#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

gen:BaumartenBaumgruppen rdf:type gen:Root .
gen:BaumartenBaumgruppen gen:child gen:Row_BaumartenBaumgruppen_01 .
gen:Row_BaumartenBaumgruppen_01 gen:id "1"^^xsd:int ;
  dat:Abkürzung "Abg" ;
  dat:Artname "Küstentanne" ;
  dat:Gruppe "Tannen" ;
  dat:WET-Zuordnung "Tannen-Mischwald" .
gen:BaumartenBaumgruppen gen:child gen:Row_BaumartenBaumgruppen_02 .
gen:Row_BaumartenBaumgruppen_02 gen:id "2"^^xsd:int ;
  dat:Abkürzung "Abi" ;
  dat:Artname "sonstige Tannenart" ;
  dat:Gruppe "Tannen" ;
  dat:WET-Zuordnung "Tannen-Mischwald" .
```

Codeblock 14: Aus CSV generiertes RDF im Turtle-Format

4.4.3 JSON2RDF

JSON ist ein textbasiertes Datenformat, das auf der Verschachtelung von Listen und Dictionaries basiert. Um JSON in RDF umzuwandeln, haben wir das Shell-Programm JSON2RDF entwickelt, das auf der vorher definierten Mapping-Grundstruktur aufbaut.

Eine JSON-Datei kann einfach mit der Grundstruktur dargestellt werden, da die Struktur von Listen und Dictionaries direkt auf diese übertragbar ist. Codeblock 15 zeigt eine JSON-Datei, die von JSON2RDF in das RDF in Codeblock 16 umgewandelt wird.

```
[
  {
    "überschrift": "Jungbestandspflege (Jpfl)",
    "stichpunkte": "Förderung klimaanpassungsfähiger (Lb-)Baumarten.",
    "wet": "kx"
  },
  {
    "überschrift": "Jungbestandspflege (Jpfl)",
    "stichpunkte": "Ggf. Mischwuchsregulierung zur Sicherung der
Mischbaumarten.",
    "wet": "p"
  }
]
```

Codeblock 15: Beispielhafte JSON-Eingabedatei

```
@prefix gen: <generic#> .
@prefix dat: <data#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

gen:WaldbaulicheMaßnahmen rdf:type gen:Root .
gen:WaldbaulicheMaßnahmen gen:child gen:WaldbaulicheMaßnahmen_1 .
gen:WaldbaulicheMaßnahmen_1 gen:id "1"^^xsd:int ;
  dat:überschrift "Jungbestandspflege (Jpfl)"^^xsd:string ;
  dat:stichpunkte "Förderung klimaanpassungsfähiger
(Lb-)Baumarten."^^xsd:string ;
  dat:wet "kx"^^xsd:string .
gen:WaldbaulicheMaßnahmen gen:child gen:WaldbaulicheMaßnahmen_2 .
gen:WaldbaulicheMaßnahmen_2 gen:id "2"^^xsd:int ;
  dat:überschrift "Jungbestandspflege (Jpfl)"^^xsd:string ;
  dat:stichpunkte "Ggf. Mischwuchsregulierung zur Sicherung der
Mischbaumarten."^^xsd:string ;
  dat:wet "p"^^xsd:string .
```

Codeblock 16: Aus JSON generiertes RDF im Turtle-Format

4.4.4 TIF2RDF

Um die Metadaten einer TIFF-Datei in RDF umzuwandeln, haben wir ein einfaches Python-Skript mit dem Namen TIF2RDF entwickelt. Dieses Skript bekommt als Eingabe einen Ordnerpfad und generiert aus allen TIFF-Dateien in diesem Ordner RDF-Triples der Metadaten. Dabei hält sich TIF2RDF nicht an die Grundstruktur aus Abschnitt 4.4.1. Das liegt daran, dass wir die Metadaten von allen TIFF-Dateien im Wissensgraphen gleich modellieren wollen. Wir brauchen hier also keine Trennung von Parsing und semantischer Modellierung. Codeblock 17 zeigt das resultierende RDF einer TIFF-Datei im Turtle-Format:

```
@prefix dat: <data#> .
@prefix gen: <generic#> .
@prefix geo: <http://www.opengis.net/ont/geosparql#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

gen:Tif_Jahresniederschlag rdf:type gen:Tif ;
    rdfs:label "Jahresniederschlag" ;
    dat:bandCount "1"^^xsd:int ;
    dat:boundingBox "POLYGON((7.435 ... 47.528))"^^geo:wktLiteral ;
    dat:crs "EPSG:4326" ;
    dat:dtype "float32" ;
    dat:fieldHeight "1250.97"^^xsd:float ;
    dat:fieldWidth "828.74"^^xsd:float ;
    dat:height "202"^^xsd:int ;
    dat:width "276"^^xsd:int .
```

Codeblock 17: Aus einer TIFF-Datei generiertes RDF

4.5 Construct-Stages 1-N

In den Construct-Stages wird das generische RDF aus der Generic-Generation-Stage iterativ per Construct-Queries in ein modelliertes RDF umgewandelt. Dafür werden in einer Construct-Stage SPARQL-Construct-Queries an den QLever-Server der vorherigen Stage gestellt und die Ergebnisse, in Kombination mit den Ergebnissen der vorherigen Construct-Stages, wieder als QLever-Server zur Verfügung gestellt. In Abschnitt 4.2 ist der genaue Ablauf beschrieben.

In einer Stage-Konfigurationsdatei können Queries unter dem Schlüssel `queries` definiert werden. Queries haben die Schlüssel `name`, `stage` und `query`. Der `name` wird für den Dateinamen genutzt, unter dem das Ergebnis der Query Gzip-komprimiert gespeichert wird. Die `stage` definiert die Construct-Stage, in der die Query ausgeführt wird. Die `query` ist eine SPARQL-Construct-Query. Codeblock 18 zeigt zwei beispielhafte Queries.

```yaml
queries:
  - name: Standortsdatenblätter_Baumarten
    stage: 1
    query: |
      PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
      ...
  - name: insert_type_geometry
    stage: 2
    query: |
      PREFIX geo: <http://www.opengis.net/ont/geosparql#>
      ...
```

Codeblock 18: Beispielhafte Queries in einer Stage-Konfigurationsdatei

Qonstructor führt alle Construct-Stages bis zur höchsten Construct-Stage aus, für die noch Queries definiert sind. Es können maximal 9 Construct-Stages ausgeführt werden (Stages 1 bis 9). Da für die QLever-UI alle QLever-Endpunkte manuell konfiguriert werden müssen, braucht es ein festes Limit. Wir haben uns pragmatisch für den Constraint von 9 entschieden. Für den Forstgraphen haben wir nur drei Construct-Stages genutzt, es gibt also noch viel Spielraum.

Die Construct-Stages senden jeweils SPARQL-Queries an den Wissensgraphen der vorherigen Stage. Wir legen die Integration von QLever in Abschnitt 4.7 dar.

4.6 Export-Stage

Die Export-Stage führt eine einzelne Construct-Query auf dem Graphen Graph_N aus. Das Ergebnis ist $\text{RDF}_{\text{Export}}$, eine Gzip-komprimierte Datei im N-Triples-Format. Die Export-Query kann in der Pipeline-Konfigurationsdatei gesetzt werden. Die Default-Export-Query exportiert alle Triples aus dem Graphen. Es könnte jedoch auch eine Query

eingestellt werden, welche die Triples nach bestimmten Kriterien filtert. Codeblock 19 zeigt die Export-Query-Konfiguration der Default-Pipeline-Konfigurationsdatei.

```
export_query: |
  CONSTRUCT { ?s ?p ?o } WHERE { ?s ?p ?o }
```

Codeblock 19: Default-Export-Query

4.7 QLever-Integration

Qonstructor nutzt QLever als RDF-Query-Engine. Viele der Source-Dateien für den Forstgraphen sind Geodaten, weshalb wir für den Forstgraphen eine RDF-Query-Engine benötigen, die den GeoSPARQL-Standard unterstützt. Für die Laufzeit von Qonstructor macht der Index-Bau einen erheblichen Anteil aus. QLever liegt beim Index-Bau in Benchmarks vorne und unterstützt zusätzlich den GeoSPARQL-Standard [21], [33], [34], [35].

Qonstructor steuert QLever über die Python-basierte QLever-CLI. Dabei werden alle Konfigurationsparameter aus der Pipeline-Konfigurationsdatei unter dem Schlüssel `qlever` als Kommandozeilenargumente direkt an die QLever-CLI übergeben. So sind keine Qleverfiles für die Konfiguration der einzelnen QLever-Instanzen nötig. Codeblock 20 zeigt die QLever-Konfiguration der Default-Pipeline-Konfigurationsdatei.

```
qlever:
  stxxl_memory: 10G
  memory_for_queries: 10G
  cache_max_size: 10G
  cache_max_size_single_entry: 5G
  timeout: 1000s
  access_token: 1234abcd
  vocabulary_type: on-disk-compressed
  num_threads: 8
  num_triples_per_batch: 5000000
```

Codeblock 20: QLever-Konfiguration in der Pipeline-Konfigurationsdatei

Am Ende der Generic-Generation-Stage sowie der Construct-Stages wird jeweils ein QLever-Index gebaut und dann ein QLever-Server mit diesem Index gestartet. Die QLever-Server werden nur für die nächste Stage benötigt, werden jedoch nicht automatisch gestoppt. So ist eine dauerhafte Verifizierung der einzelnen Stages per QLever-UI möglich. In der QLever-UI sind die QLever-Server für $\text{Stage}_i, i \in [0, 9]$ vorkonfiguriert und können über ein Dropdown-Menü ausgewählt werden. Die QLever-Server werden erst beim erneuten Ausführen von Qonstructor oder beim Stoppen des Qonstructor-Containers gestoppt.

4.8 Parallelität

Viele der Pipeline-Schritte laufen parallel, entweder durch das Nutzen von Software wie QLever, welche intern schon viel parallelisiert, oder durch die Architektur der Qonstructor-Pipeline selbst.

Das Laden der Konfigurationsdateien und der Cleaning-Schritt sind im Vergleich zu den eigentlichen Stages vernachlässigbar und daher nicht parallelisiert.

Alle Stages bis auf die Export-Stage bauen am Ende einen QLever-Index. Das Bauen des QLever-Indexes läuft standardmäßig auf allen verfügbaren Kernen. Dieser Schritt ist also QLever-intern parallelisiert. Das Starten des QLever-Servers läuft nicht parallel.

Die Generic-Generation-Stage führt für jede Datei ein separates Shell-Kommando für die Umwandlung aus. Daher ist es einfach möglich, diese Tasks zu parallelisieren. In der Pipeline-Konfigurationsdatei kann mit der Option `generic_workers: k` die maximale Anzahl an parallelen Prozessen für die Generic-Generation-Stage gesetzt werden.

Jede Construct-Stage sendet eine Menge an Construct-Queries an QLever. QLever kann Construct-Queries parallel bearbeiten. Die Parallelität von QLever selbst kann in der Pipeline-Konfigurationsdatei unter `qlever` mit `num_threads: n` gesetzt werden. Wie viele Tasks Qonstructor für das Bearbeiten der Construct-Queries parallel laufen lässt, kann mit der Option `construct_workers` eingestellt werden.

4.9 Containerisierung und Nutzungsworkflow

Ein Ziel von Qonstructor ist es, möglichst reproduzierbar zu sein (Designziel 6). Daher wird Qonstructor innerhalb eines Docker-Containers [36] zur Verfügung gestellt. Es gibt ein Dockerfile, das den Qonstructor-Container definiert. Zusätzlich wird Docker Compose [37] genutzt, um den Qonstructor-Container in Kombination mit der QLever-UI einfach starten zu können. Es gibt zusätzlich ein Makefile, das verschiedene Docker-Kommandos abkürzt.

Um die Bedienung von Qonstructor zu verdeutlichen, zeigen wir im Folgenden zwei beispielhafte Nutzungsworkflows. Im ersten Nutzungsworkflow wird ein neues Qonstructor-Projekt erstellt und eine erste Source-Datei in RDF umgewandelt:

1. Git-Repository klonen
2. Workspace-Ordner einrichten
3. Qonstructor-Container via `make up` starten
4. Auf die Shell des Qonstructor-Containers via `make shell` zugreifen

5. Source-Dateien in `workspace/data` einfügen
6. Sources und Queries in den Stage-Konfigurationsdateien definieren
7. Pipeline mit dem Kommando `pipeline` starten
8. Auf die QLever-UI im Browser zugreifen

Im zweiten Nutzungsworkflow wird zum bestehenden Qonstructor-Projekt eine weitere Source-Datei sowie eine Construct-Query hinzugefügt:

1. Neue Source-Dateien in `workspace/data` einfügen
2. Neue Stage-Konfigurationsdatei `workspace/configs/neue-daten.yaml` anlegen
3. In `neue-daten.yaml` Sources definieren
4. Pipeline mit dem Kommando `pipeline -c neue-daten.yaml -s 0` nur für die neue Konfigurationsdatei und Stage 0 ausführen
5. Die QLever-UI mit dem Graphen der Generic-Generation-Stage öffnen und eine Construct-Query schreiben sowie testen
6. Construct-Query in `neue-daten.yaml` als Construct-Query der ersten Stage kopieren
7. Pipeline mit dem Kommando `pipeline -c neue-daten.yaml -s 1` für Stage 1 ausführen
8. QLever-UI mit dem Graphen der ersten Construct-Stage öffnen und die Daten verifizieren

5 Forstgraph

In Kapitel 4 führen wir die RDF-Generierungspipeline Qonstructor ein. In diesem Kapitel geht es um die Erstellung des Forstgraphen mithilfe der Qonstructor-Pipeline. Wir nennen die Qonstructor-Pipeline mit der Konfiguration zur Generierung des Forstgraphen ab sofort Forst-Pipeline.

5.1 Ontologische Grundlagen

In diesem Abschnitt erklären wir die grundlegende Ontologie des Forstgraphen und die daraus abgeleiteten Modellierungsregeln. Diese Regeln sorgen für eine einheitliche Struktur des Wissensgraphen. Der Forstgraph nutzt für die Verlinkung von Objekten und Klassen den RDFS-Standard. Außerdem wird GeoSPARQL genutzt, um Geometrien als WKT abzuspeichern.

Im Folgenden sind die Modellierungsgrundsätze festgelegt:

1. Alle Objekte im Wissensgraphen besitzen einen `rdf:type`. Jedes `rdf:type`-Triple verweist auf eine Klasse.
2. Jede Klasse ist selbst durch `rdf:type rdfs:Class` gekennzeichnet.
3. Wenn möglich nutzen Objekte ein `rdfs:label` für menschlich lesbare Bezeichnungen. Das Objekt `wp:Baumart_BAh` hat z.B. das `rdfs:label "Bergahorn"`.
4. Jedes Objekt im Graphen kann per `geo:hasGeometry` eine Geometrie besitzen. Das Objekt ist dann vom `rdf:type geo:Feature`. Die Geometrie ist vom `rdf:type geo:Geometry`.
5. Jede Geometrie kann per `geo:asWKT` einen WKT-String als Literal abgespeichert haben. Der Datentyp des WKT-Strings muss `geo:wktLiteral` sein.

An folgenden Beispiel-Triples sind diese Grundsätze erkennbar:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix wp: <https://wald.cs.uni-freiburg.de/ontology#> .
@prefix geo:  <http://www.opengis.net/ont/geosparql#> .

wp:Betrieb rdf:type rdfs:Class .
wp:Betrieb_192 rdf:type wp:Betrieb ;
  rdf:type geo:Feature ;
  rdfs:label "Stadtwald Freiburg im Breisgau" ;
  geo:hasGeometry wp:Geometry_483 .
wp:Geometry_483 rdf:type geo:Geometry ;
  geo:asWKT "POLYGON(...)"^^geo:wktLiteral .
```

Codeblock 21: Beispiel-Triples zu den Modellierungsgrundsätzen

5.2 IRIs

Qonstructor nutzt SPARQL-Construct-Queries, um die Daten von generischem RDF zu normalisiertem RDF umzuwandeln. In diesem Prozess müssen passende IRIs für alle Objekte generiert werden. Dieser Prozess wird auch IRI-Minting genannt. In diesem Abschnitt erklären wir, wie in den Construct-Queries IRIs gemintet werden.

Für den gesamten Forstgraphen nutzen wir das Präfix `wp:` (`https://wald.cs.uni-freiburg.de/ontology#`). Folgende Regeln haben wir bezüglich der IRIs definiert:

1. Die IRI einer Klasse mit dem Namen `Klassenname` ist `wp:Klassenname`. Der Name der Klasse ist großgeschrieben (z.B. `wp:Baumart`).
2. Alle Objekte einer Klasse `Klassenname` haben eine IRI, die mit `wp:Klassenname_` anfängt (z.B. `wp:Baumart_BAh`). Dabei ist der String nach dem `'_'` beliebig, muss aber eindeutig für das jeweilige Objekt sein. Wir nennen diesen String ab sofort Objekt-Identifier.

5.3 Daten und Preprocessing

Die Persistenz von Qonstructor befindet sich vollständig im Workspace. Die beiden Unterordner `data` und `configs` werden manuell mit Source-Dateien und Stage-Konfigurationen gefüllt, die restlichen Unterordner-Inhalte werden daraus im Laufe der Pipeline generiert. In diesem Abschnitt geht es um die Source-Daten im Ordner `workspace/data`, aus denen der Forstgraph gebaut wird.

Für den Forstgraphen wurden uns während der Projektlaufzeit verschiedene Daten zur Verfügung gestellt. Teile davon sind öffentliche Daten, die frei über das Internet verfügbar sind, andere Daten sind intern und dürfen nicht veröffentlicht werden. Manche Daten wurden uns per Mail zugesendet und andere über landesinterne Clouds zur Verfügung gestellt. Die Daten liegen in verschiedenen Formaten vor, unter anderem CSV-Dateien, Excel-Dateien, GeoPackages, Shapefiles, TIFF-Dateien und PDF-Dateien.

Im ersten Schritt haben wir die Daten in ihrem originalen Format im Ordner `workspace/raw` einsortiert. Dabei haben wir sie nach Datenquelle und Datensatz sortiert und Zip-Dateien entpackt. Der Ordner `workspace/raw` wird von Qonstructor selbst nicht genutzt.

Parallel haben wir im Ordner `workspace/data` dieselbe Ordnerstruktur wie in `workspace/raw` erstellt. Mithilfe von einfachen Preprocessing-Skripten haben wir die Daten aus `workspace/raw` in die einheitlichen Formate CSV, JSON und TIFF umgewandelt und in `workspace/data` abgespeichert. Hier wäre eine Automatisierung möglich gewesen, war jedoch für die bisherige Menge an Daten nicht nötig. Darauf gehen wir in Abschnitt 9.3 noch genauer ein.

Im Folgenden erklären wir, wie die verschiedenen Dateitypen transformiert wurden.

Esri-Shapefiles haben wir mit dem Kommandozeilenprogramm `ogr2ogr` (ein Teil von GDAL) in CSV-Dateien umgewandelt. Hierbei ist wichtig, das gewünschte CRS klar festzulegen. Außerdem muss angegeben werden, dass die Geometrien in einer Spalte `AS_WKT` als WKT-Strings abgespeichert werden sollen.

```
ogr2ogr -f "CSV" [output.csv] [input.shp] \
  -lco GEOMETRY=AS_WKT \
  -t_srs EPSG:4326 [layer_name]
```

Codeblock 22: Umwandlung von Shapefiles zu CSV mit ogr2ogr

GeoPackages haben wir wie Esri-Shapefiles mit `ogr2ogr` zu CSV umgewandelt. Ein GeoPackage kann aber mehrere Layer beinhalten, weshalb das Kommando für alle Layer einzeln ausgeführt werden muss.

```
ogr2ogr -f "CSV" [output.csv] [input.gpkg] \
  -lco GEOMETRY=AS_WKT \
  -t_srs EPSG:4326 \
  [layer name]
```

Codeblock 23: Umwandlung von GeoPackages zu CSV mit ogr2ogr

TIFF-Dateien haben wir mit dem Kommandozeilenprogramm `gdalwarp` (Teil von GDAL) zu COG (Cloud Optimized GeoTIFF) im globalen Koordinatensystem transformiert. Dabei haben wir die verlustfreie Kompressionsmethode `DEFLATE` gewählt.

```
gdalwarp -of COG -t_srs EPSG:4326 -co "COMPRESS=DEFLATE" \
  [input.tif] [output.tif]
```

Codeblock 24: Umwandlung von TIFF-Dateien zu COG mit gdalwarp

xlsx-Dateien (Microsoft Excel-Dateien) haben wir mit LibreOffice Calc [38] als CSV-Dateien exportiert.

PDF-Dateien haben wir über eine separate Pipeline mithilfe eines LLMs (Large Language Model) in JSON umgewandelt. Für den finalen Forstgraphen nutzen wir nur die Source-Datei `WaldbaulicheMaßnahmen.json`, welche aus einer PDF-Datei `WET2024` [39] generiert wurde. Wir gehen hier nicht weiter auf diese externe Pipeline ein.

Tabelle 2 zeigt eine Übersicht aller Source-Daten nach dem Preprocessing-Schritt. Alle diese Source-Dateien kombiniert bilden die Datenbasis der Forst-Pipeline. Die Pfade in der Tabelle gehen von `workspace/data` aus.

Ordner	Datei
ausrichtung	gelände.csv
	hangneigungsklassen.csv
waldnaturschutz	FFH.csv
	NSG.csv
	SPA.csv
tif	Jahresniederschlag.tif
	Sommertemperatur.tif
	StabilityBuche.tif
	StabilityDouglasie.tif
	StabilityFichte.tif
	StabilityKiefer.tif
	StabilityTanne.tif
	WHSK.tif
	baumartenkarte.tif
fva/Baumarteneignung	Baumarteneignung.csv
fva/RegionaleGliederung	RegionaleGliederung.csv
	Wuchsgebiete.csv
fva/Standortsdatenblätter	Standortsdatenblätter.csv
fva/Standortskartierung	Einzelsignaturflächen.csv
	Einzelsignaturlinien.csv
	Einzelsignaturpunkte.csv
	Standorte.csv
lfv/Baumarten	BaumartenBaumgruppen.csv
	Baumartenliste.csv
lfv/Bestände	Bestände.csv
	Waldeinteilung_BW.csv
	Waldeinteilung_Freiburger_Stadtwald.csv
lfv/FGeo/Abteilungen	Abteilungen.csv
lfv/FGeo/Betriebsflächen	Betriebsflächen.csv
lfv/FGeo/Betriebsnamen	Betriebsnamen.csv
lfv/WET2024	BaumgruppeWET.csv
	WETs.csv
	WaldbaulicheMaßnahmen.json
lfv/Waldbesitzarten	Waldbesitzarten.csv

Tabelle 2: Source-Dateien der Forst-Pipeline nach Preprocessing

5.4 Stage-Konfigurationsdateien

Zusätzlich zu den Source-Dateien im Ordner `workspace/data` sind die Stage-Konfigurationsdateien der zweite wichtige Teil der Forst-Pipeline. Diese können in einer beliebigen Ordnerstruktur innerhalb von `workspace/configs` abgelegt werden.

Wir haben die Stage-Konfigurationsdateien nach folgendem Schema organisiert: Alle Stage-Konfigurationsdateien für den finalen Forstgraphen sind unter `configs/final` sortiert. So können neue Stage-Konfigurationsdateien in einem anderen Unterordner getestet werden. Im Ordner `configs/final` sind die Stage-Konfigurationsdateien ähnlich organisiert wie die Source-Daten in `workspace/data`.

Die folgende Tabelle stellt alle Stage-Konfigurationsdateien sowie deren Pfade dar. Für jede Stage-Konfigurationsdatei ist angegeben, wie viele Sources und Queries jeweils definiert werden. Zusätzlich ist für die Queries eine Verteilung auf die Construct-Stages angegeben.

Ordner	**Datei**	**Sources**	**Queries**	**Stage 1|2|3**
final	`Inference.yaml`	0	2	0|0|2
	`Linking.yaml`	0	2	0|2|0
final/ausrichtung	`Ausrichtung.yaml`	2	2	2|0|0
final/fva	`Baumarteneignung.yaml`	1	1	1|0|0
	`Einzelsignaturen.yaml`	3	3	3|0|0
	`Linking.yaml`	0	1	0|1|0
	`RegionaleGliederung.yaml`	2	2	2|0|0
	`Standorte.yaml`	1	2	2|0|0
	`Standortsdatenblätter.yaml`	2	2	2|0|0
final/lfv	`Linking.yaml`	0	4	0|4|0
final/lfv/atomic	`Abteilungen.yaml`	1	1	1|0|0
	`Baumarten.yaml`	2	1	1|0|0
	`Bestände.yaml`	1	3	2|0|1
	`Betriebe.yaml`	2	2	2|0|0
	`Forstbezirke.yaml`	1	1	1|0|0
	`WET.yaml`	3	3	3|0|0
	`Waldbesitzarten.yaml`	1	1	1|0|0
final/tif	`Tifs.yaml`	1	1	1|0|0
final/waldnaturschutz	`Schutzgebiete.yaml`	3	3	3|0|0

Tabelle 3: Stage-Konfigurationsdateien der Forst-Pipeline

Insgesamt besteht die Konfiguration aus 19 einzelnen Stage-Konfigurationsdateien. Diese beinhalten insgesamt 26 Source-Definitionen und 37 Construct-Queries. Das sind weniger

Source-Definitionen als Source-Dateien aus Abschnitt 5.3. Das liegt daran, dass TIF2RDF mit einer einzelnen Source-Definition einen ganzen Ordner mit TIFF-Dateien verarbeitet.

5.5 Generic-Generation-Stage

Die Generic-Generation-Stage ist die erste Stage von Qonstructor. Hier werden Source-Dateien in ein generisches RDF umgewandelt. Die Konfiguration dafür befindet sich auch innerhalb der Stage-Konfigurationsdateien unter dem Key `sources`. Dabei folgen alle Stage-Konfigurationsdateien einem ähnlichen Muster. Codeblock 25 zeigt die Sources der Stage-Konfigurationsdatei `Schutzgebiete.yaml`.

```yaml
sources:
  - name: FFHgebiete
    path: waldnaturschutz/FFH.csv
    type: csv
  - name: Naturschutzgebiete
    path: waldnaturschutz/NSG.csv
    type: csv
  - name: Vogelschutzgebiete
    path: waldnaturschutz/SPA.csv
    type: csv
```

Codeblock 25: Source-Definitionen in Schutzgebiete.yaml

5.6 Construct-Stage 1 - Normalisierung

In diesem Abschnitt führen wir die Construct-Queries der ersten Stage ein. Wir nennen diese Stage auch Normalisierung, da hier aus generischem RDF ein normalisiertes RDF generiert wird. Die Normalisierung ist die einzige Stage, in der das Eingabe-RDF nicht mit in die nächste Stage wandert. Alle Daten, die vom generischen Graphen in den finalen Forstgraphen wandern sollen, müssen manuell per Construct-Queries in der Normalisierung generiert werden. Hier wird also die Grundstruktur des Forstgraphen aufgebaut, was diese Stage zur wichtigsten Construct-Stage macht.

In den folgenden Abschnitten gehen wir zuerst auf Grundkonzepte ein, die für alle Construct-Queries der Normalisierung gelten. Danach zeigen wir drei Beispielqueries, die diese Grundkonzepte nutzen. Insgesamt beinhaltet die Normalisierung zu viele Queries, als dass wir hier alle einzeln erklären könnten.

5.6.1 Query-Grundstruktur

Die meisten Source-Daten des Forstgraphen sind CSV-Dateien. Codeblock 26 zeigt eine Construct-Query, die generisches RDF einer CSV-Datei in normalisiertes RDF umwandelt (die Query ist dabei leicht vereinfacht).

```
PREFIX wp: <https://wald.cs.uni-freiburg.de/ontology#>
PREFIX dat: <data#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX gen: <generic#>

CONSTRUCT {
  ?iri rdf:type wp:Naturschutzgebiet ;
       rdfs:label ?OBJEKT ;
       wp:objekt_nr ?OBJEKT_NR .
} WHERE {
  gen:Naturschutzgebiete gen:child ?row .
  ?row dat:OBJEKT ?OBJEKT .
  ?row dat:OBJEKT_NR ?OBJEKT_NR .
  ?row dat:WKT ?WKT_ BIND (STRDT(?WKT_, geo:wktLiteral) AS ?WKT)
  BIND(IRI(CONCAT(STR(wp:),"Naturschutzgebiet_",(SUBSTR(SHA256(?WKT),1,16))))
AS ?iri)
}
```

Codeblock 26: Construct-Query zur Normalisierung einer CSV-Datei

Der Where-Block der Query filtert über den eindeutigen Tabellennamen nach allen Zeilen dieser Tabelle. Die Variable `?row` wird dabei für jede gefundene Zeile gebunden. Über die Spaltennamen der Tabelle können einzelne Variablen für die Felder dieser Zeile gebunden werden. Die Bind-Clause generiert eine IRI für das Objekt, worauf wir in Abschnitt 5.6.2 genauer eingehen. Im Construct-Block kann dann definiert werden, in welcher Struktur die Daten aus dem Where-Block als RDF ausgegeben werden sollen.

In dieser Query werden also alle Zeilen einer Tabelle zu Objekten einer bestimmten Klasse umgewandelt. Dabei wird jedes Feld einer Zeile zu einem Triple. Der Großteil aller Construct-Queries in Stage 1 folgt diesem einfachen Muster. Für JSON- oder TIFF-Dateien sehen die Construct-Queries ähnlich aus.

5.6.2 IRI-Minting

Für das Minting von Objekt-IRIs nutzen wir zwei verschiedene Optionen. Dafür brauchen wir verschiedene SPARQL-Funktionen. Mit der `BIND()`-Clause kann einer Variable ein Wert zugewiesen werden. `CONCAT()` kann mehrere Strings zusammenfügen. `IRI()` konvertiert einen String zu einer IRI, welche dann in dem Construct-Block einer Construct-Query genutzt werden kann. `STR()` konvertiert alle möglichen Datentypen sowie IRIs zu Strings. Im Folgenden stellen wir beide Optionene des IRI-Minting vor.

5.6.2.1 Minting via Concat

Wir bezeichnen die folgende Methode des IRI-Minting ab sofort als „Minting via Concat“. Dabei wird der Objekt-Identifier aus einem oder mehreren Werten zusammengefügt. Wichtig ist dafür, dass der entstehende Objekt-Identifier das Objekt eindeutig beschreibt.

Zusätzlich muss die entstehende IRI valide sein. Codeblock 27 zeigt verschiedene Beispiele für Minting via Concat.

```
1 BIND (IRI(CONCAT(STR(wp:Baumart_),?baumart)) AS ?iri)
2 BIND (IRI(CONCAT(STR(wp:), "Betrieb_",?fbez_,"_",?betr_)) AS ?betrieb_iri)
3 BIND (IRI(CONCAT(STR(wp:), "RegionaleEinheit_", REPLACE(?nr, "/", "-"))) AS ?
  iri)
```

Codeblock 27: Beispiele für Minting via Concat

Alle Beispiele folgen dem gleichen Konzept. Es gibt eine IRI-Basis wie z.B. `wp:Baumart_`, welche per `STR()` von einer IRI zu einem String umgewandelt wird. Dann werden verschiedene Werte von Variablen via `CONCAT()` angefügt. Zuletzt wird das Ergebnis mit `IRI()` vom String zurück zur IRI konvertiert und dann mit `BIND()` an eine Variable gebunden.

5.6.2.2 Minting via SHA

Wir bezeichnen die folgende Methode ab sofort als „Minting via SHA“. Sie kann genutzt werden, wenn „Minting via Concat“ nicht möglich ist oder z.B. zu lange IRIs generieren würde. Bei Minting via SHA wird der Objekt-Identifier als SHA-Hash aus beliebigen Werten der Prädikate des Objekts gebildet. Damit die IRI übersichtlich bleibt, verwenden wir nur die ersten 16 Stellen des Hashes als Objekt-Identifier. Auch hier müssen diese Werte das Objekt selbst eindeutig beschreiben. Jedoch können auch sehr lange oder nicht IRI-konforme Werte, wie z.B. WKT-Strings, als Eingabe für SHA genutzt werden. Diese Methode fungiert somit als Fallback von „Minting via Concat“. Codeblock 28 zeigt Beispiele von Minting via SHA.

```
1 BIND (IRI(CONCAT(STR(wp:),"Vogelschutzgebiet_",(SUBSTR(SHA256(?WKT),1,16))))
  AS ?iri)
2 BIND (IRI(CONCAT(STR(wp:),"Geom_",(SUBSTR(SHA256(?WKT),1,16)))) AS ?geom_iri)
```

Codeblock 28: Beispiele für Minting via SHA

In beiden Beispielen wird ein String via `SHA256()` gehasht. Das Ergebnis wird durch `SUBSTR()` auf 16 Zeichen gekürzt und mit dem Präfix per `CONCAT()` zusammengefügt. Zuletzt wird der String per `IRI()` zu einer IRI umgewandelt und per `BIND()` an eine Variable gebunden.

5.6.3 Linking während der Normalisierung

Wir definieren einen Link als ein Triple, das zwei Objekte verschiedener Klassen miteinander verbindet. Meistens sollen nicht nur zwei explizite Objekte miteinander verbunden werden, sondern viele Objekte einer Klasse mit vielen Objekten einer anderen Klasse. Die meisten Links können schon während der Normalisierung generiert werden.

Wir nehmen den Fall an, dass es für beide Klassen `Klasse1` und `Klasse2` eine jeweilige Query gibt, die alle Objekte innerhalb dieser Klassen generiert. `Query1` erstellt Objekte der `Klasse1` und `Query2` der `Klasse2`. Um einen Link zwischen Objekten dieser beiden Klassen herzustellen, müssen die Linking-Triples in einer der beiden Queries generiert werden. `Query1` und `Query2` generieren jeweils IRIs für ihre Objekte. Um in `Query1` Links zu `Query2` zu generieren, müssen dort zusätzlich IRIs für Objekte der `Klasse2` generiert werden. Dafür braucht es die Werte aller Prädikate, die für die Generierung der IRI nötig sind. Das ist die Grundvoraussetzung, welche das Linking ermöglicht. Mit dieser generierten IRI kann dann im Construct-Block der Query ganz einfach ein Triple generiert werden, welches die beiden Objekte verbindet. Dabei ist es nicht relevant, in welche Richtung der Link zeigt. Der Link kann sogar bidirektional sein. Codeblock 29 zeigt Linking während der Normalisierung anhand eines Beispiels.

```
PREFIX wp: <https://wald.cs.uni-freiburg.de/ontology#>
PREFIX dat: <data#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX gen: <generic#>

CONSTRUCT {
  ?iri rdf:type wp:RegionaleEinheit ;
    rdfs:label ?regionaleeinheit ;
    wp:nr ?nr .
  ?iri wp:wuchsgebiet ?wuchsgebiet_iri .  # link 1
  ?wuchsgebiet_iri wp:regionaleEinheit ?iri .  # link 2
} WHERE {
  gen:RegionaleGliederung gen:child ?row .
  ?row dat:Nr ?nr .
  ?row dat:RegionaleEinheit ?regionaleeinheit .
  ?row dat:WG_RE_F ?wg_re_f .
  BIND (IRI(CONCAT(STR(wp:), "RegionaleEinheit_", REPLACE(?nr, "/", "-")))
AS ?iri)  # primary iri
  BIND (IRI(CONCAT(STR(wp:), "Wuchsgebiet_", ?wg_re_f)) AS ?wuchsgebiet_iri)
# foreign iri
}
```

Codeblock 29: Beispiel für Linking während der Normalisierung

In dieser Query werden Objekte der Klasse `wp:RegionaleEinheit` generiert. Im Where-Block werden die IRIs für die `wp:RegionaleEinheit` sowie das zugehörige `wp:Wuchsgebiet` gemintet. Im Construct-Block wird dann ein bidirektionaler Link zwischen `wp:RegionaleEinheit` und `wp:Wuchsgebiet` erzeugt.

5.6.4 Typecast

Die Literale der generischen Triples, welche aus den Source-Dateien generiert werden, haben in den meisten Fällen keine Datentyp-IRI. Daher müssen die Literale innerhalb der Construct-Queries zum jeweiligen Datentyp umgewandelt werden. Dafür gibt es zwei Methoden. Die erste Methode ist `STRDT()`, was für „String + Datatype“ steht. Hier

wird eine Datentyp-IRI an einen String angefügt. Es wird nicht geprüft, ob der String tatsächlich der entsprechende Datentyp ist. Die zweite Methode ist ein Typecast zu einem XSD-Typ, wie z.B. `xsd:integer()`. Hier ist fest definiert, wie die entsprechenden Datentypen geparst und validiert werden. Falls das Parsing oder die Validierung fehlschlägt, wird die entsprechende Variable nicht gebunden. Codeblock 30 zeigt ein Beispiel für beide Methoden.

```
?row dat:VERTIKAL_ZONAL ?vertikal_zonal_ BIND(xsd:int(?vertikal_zonal_) AS ?
vertikal_zonal)
?row dat:Shape ?wkt_ BIND (STRDT(?wkt_, geo:wktLiteral) AS ?wkt)
```

Codeblock 30: Beispiele für Typecasts

5.6.5 Optionale Triples

Es haben nicht immer alle Objekte einer Klasse dieselben Prädikate. Beim Schreiben einer Construct-Query ist es wichtig, entsprechende Triple-Patterns in eine Optional-Clause zu setzen. Dann werden auch Objekte bei der Abfrage berücksichtigt, die diese Triple-Patterns nicht enthalten. Codeblock 31 zeigt ein optionales Triple-Pattern.

```
OPTIONAL {?row dat:NUART ?nuart}
```

Codeblock 31: Beispiel für ein optionales Triple-Pattern

In den Source-Dateien des Forstgraphen kommen außerdem oft Einträge wie „keine Angabe“ oder „keine Eingabe zum Wasserhaushalt“ vor, oftmals auch mit unterschiedlichen Schreibweisen innerhalb einer Tabellenspalte. Im Forstgraphen sollen für diese Werte keine Triples entstehen. Daher ist es wichtig, diese Ergebnisse in den Construct-Queries der ersten Stage herauszufiltern. Codeblock 32 zeigt, wie wir ungültige Werte in der Forst-Pipeline filtern:

```
OPTIONAL { ?row dat:WHH ?whh_ BIND (IF(?whh_ != "keine Eingabe Wasserhaushalt"
&& ?whh_ != "keine Angabe zum Wasserhaushalt", ?whh_, 1/0) AS ?whh ) }
```

Codeblock 32: Beispiel zum Filtern ungültiger Werte

5.6.6 Beispiel-Query 1

Als Nächstes untersuchen wir eine Construct-Query aus der ersten Construct-Stage. Diese Query kombiniert die Grundkonzepte aus den vorherigen Abschnitten, führt aber keine zusätzliche Komplexität ein. Codeblock 33 zeigt die Query `RegionaleEinheit` aus der Stage-Konfigurationsdatei `RegionaleGliederung.yaml`, welche die Daten aus der Source-Datei `RegionaleGliederung.csv` nutzt.

```
PREFIX wp: <https://wald.cs.uni-freiburg.de/ontology#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX dat: <data#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX gen: <generic#>

CONSTRUCT {
  ?iri rdf:type wp:RegionaleEinheit ;
    rdfs:label ?regionaleeinheit ;
    wp:nr ?nr ;
    wp:vertikalZonal ?vertikal_zonal ;
    wp:regionWaldgesellschaft ?region_waldgesellschaft ;
    wp:regionWaldentwicklungstyp ?region_waldentwicklungstyp ;
    wp:regionWaldbaurichtlinie ?region_waldbaurichtlinie ;
    wp:klimaHaupttyp ?klimahaupttyp ;
    wp:klimatönung ?klimatoenung ;
    geo:hasGeometry ?geom_iri .
  ?geom_iri geo:asWKT ?wkt .
  ?iri wp:wuchsgebiet ?wuchsgebiet_iri .
  ?wuchsgebiet_iri wp:regionaleEinheit ?iri .
} WHERE {
  gen:RegionaleGliederung gen:child ?row .
  ?row dat:Nr ?nr .  # Nummerierungscode
  ?row dat:RegionaleEinheit ?regionaleeinheit .  # vollständiger Name
  ?row dat:WG_RE_F ?wg_re_f .  # foreign key
  ?row dat:VERTIKAL_ZONAL ?vertikal_zonal_ BIND(xsd:int(?vertikal_zonal_) AS ?
vertikal_zonal)
  # regional
  ?row dat:Region_Waldgesellschaft ?region_waldgesellschaft .
  ?row dat:Region_WaldEntwicklungsTyp ?region_waldentwicklungstyp .
  ?row dat:Region_Waldbaurichtlinie ?region_waldbaurichtlinie .
  # Klima
  ?row dat:KlimaHauptTyp ?klimahaupttyp .
  ?row dat:KlimaToenung ?klimatoenung .
  # shape
  ?row dat:Shape ?wkt_ BIND (STRDT(?wkt_,geo:wktLiteral) AS ?wkt)
  # iris
  BIND (IRI(CONCAT(STR(wp:), "RegionaleEinheit_", REPLACE(?nr, "/", "-")))
AS ?iri)  # primary key
  BIND (IRI(CONCAT(STR(wp:), "Wuchsgebiet_", ?wg_re_f)) AS ?wuchsgebiet_iri)
# foreign key
  BIND (IRI(CONCAT(STR(wp:), "Geom_", (SUBSTR(SHA256(?wkt),1,16)))) AS ?
geom_iri)
}
```

Codeblock 33: Construct-Query „RegionaleEinheit“

Das Ziel der Query ist es, Objekte der Klasse `wp:RegionaleEinheit` zu generieren. Zuerst werden die Werte relevanter Prädikate aus der Tabelle extrahiert. Auf eines der Literale wird ein Typecast angewendet und der WKT-String wird durch die passende Datentyp-IRI ergänzt. Am Ende des Where-Blocks werden per „Minting via Concat“ IRIs für die `wp:RegionaleEinheit` und das zugehörige `wp:Wuchsgebiet` gemintet. Zusätzlich wird eine IRI für die Geometrie per „Minting via SHA“ gemintet. Im Construct-Block der Query wird die finale Struktur der Daten in RDF modelliert und unter anderem der bidirektionale Link zwischen `wp:RegionaleEinheit` und `wp:Wuchsgebiet` definiert.

Codeblock 34 zeigt einen Auszug aus dem Ergebnis der Query im Turtle-Format (der WKT-String wurde abgekürzt).

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX wp: <https://wald.cs.uni-freiburg.de/ontology#>

wp:RegionaleEinheit_7-05
  rdf:type wp:RegionaleEinheit ;
  rdfs:label "Westliches Bodenseegebiet" ;
  wp:nr "7/05" ;
  wp:vertikalZonal 0 ;
  wp:regionWaldgesellschaft "Westliches Bodenseegebiet" ;
  wp:regionWaldentwicklungstyp "Südwestdeutsches Alpenvorland Jungmoräne (mit Hegau, Klettgau und Adelegg)" ;
  wp:regionWaldbaurichtlinie "noch keine Zugeordnet" ;
  wp:klimaHaupttyp "Submontaner Buchenwald, (örtlich) mit Tanne, örtlich mit Fichte" ;
  wp:klimatönung "intermediär" ;
  geo:hasGeometry wp:Geom_2cd119c113432240 ;
  wp:wuchsgebiet wp:Wuchsgebiet_7 .

wp:Geom_2cd119c113432240
  geo:asWKT "MULTIPOLYGON (((9.386 ... 47.663)))"^^geo:wktLiteral .

wp:Wuchsgebiet_7
  wp:regionaleEinheit wp:RegionaleEinheit_7-05 .
```

Codeblock 34: RDF-Auszug der Query „RegionaleEinheit“

5.6.7 Beispiel-Query 2

Wir schauen uns nun die Construct-Query `Baumartenpfleglichkeit` aus der Stage-Konfigurationsdatei `Baumarteneignung.yaml` an. Diese Query wandelt generisches RDF der Source-Datei `Baumarteneignung.csv` in normalisiertes RDF um. Anhand dieser Query zeigen wir das Konzept von „Unpivoting“.

Zuerst erklären wir die Source-Datei. `Baumarteneignung.csv` hat 13 Metadaten-Spalten:

```
Regional_Z      Name            RST_SIGEL       Waldgesell      Kartierung
LEGQ_F          LEGF_F          LEGF_Strin      RZST_F          RZE_F
BAOES_F         Label_InFo      URL_INTERN
```

Für uns relevant ist nur `RZST_F`, ein Fremdschlüssel zur Klasse `Standort`.

Zusätzlich existieren für 24 verschiedene Baumartenabkürzungen jeweils 5 Spalten für die Attribute Eignung, Vitalität, Pfleglichkeit, Sicherheit und Leistung. Die Spaltennamen der Tabelle entstehen durch Verbinden der Baumartenabkürzung und des Attributs. Durch die Spaltennamen-Begrenzung von 10 Zeichen bei Shapefiles wurde der Name jeweils gekürzt.

Baumarten:

```
FI      TA      KIE     DGL     ELAE    JLAE    BU      TREI
STEI    HBU     REI     BAH     SAH     ES      WLI     SLI
BUL     VKIR    VOBE    MEBE    ELSBE   ASPE    FAH     SER
```

Spaltennamen für die Baumart `FI`:

```
FI_Eignung      FI_Vitalit      FI_Pflegli      FI_Sicherh      FI_Leistun
```

Insgesamt hat die Tabelle also $24 * 5 + 13 = 133$ Spalten. Jedoch sollen diese Spalten nicht eins zu eins in Triples übertragen werden. Wir wollen die Baumart sowie Werte für Eignung, Vitalität, Pfleglichkeit, Sicherheit und Leistung jeweils durch eigene Triples darstellen. Diese Triples fassen wir dann unter Objekten der Klasse `wp:Baumartenpfleglichkeit` zusammen, welche folgende Prädikate haben:

```
rdf:type            wp:baumart_f        wp:rzst_f           wp:eignung
wp:vitalität        wp:pfleglichkeit    wp:sicherheit       wp:leistung
```

Das relationale Pendant in einer SQL-Tabelle wird „Unpivoting“ genannt, also das Überführen von Spalten mit ähnlichen Informationen in Zeilen einer gemeinsamen Spalte. Daher nennen wir diese Art von Construct-Query ab sofort Unpivot-Query.

Die Unpivot-Query in Codeblock 35 modelliert die Daten nach diesem Prinzip.

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX wp: <https://wald.cs.uni-freiburg.de/ontology#>
PREFIX dat: <data#>
PREFIX gen: <generic#>

CONSTRUCT {
  ?iri rdf:type wp:Baumartenpfleglichkeit .
  ?iri wp:rzst_f ?rzst_f .
  ?iri wp:vitalität ?vitalität .
  ?iri wp:pfleglichkeit ?pfleglichkeit .
  ?iri wp:sicherheit ?sicherheit .
  ?iri wp:leistung ?leistung .
  ?iri wp:eignung ?eignung .
  ?iri wp:baumart_f ?baumart .
} WHERE {
  {
    SELECT ?rzst_f ?baumart (MAX(?vitali) AS ?vitalität) (MAX(?pfegl) AS ?
pfleglichkeit) (MAX(?sicherh) AS ?sicherheit) (MAX(?leistun) AS ?leistung)
(MAX(?eignun) AS ?eignung) WHERE {
      gen:Baumarteneignung gen:child ?row .
      ?row dat:RZST_F ?rzst_f_ BIND (xsd:int(?rzst_f_) AS ?rzst_f)
      ?row ?p ?o .

      BIND(STRAFTER(STR(?p), "_") AS ?keyword)
      BIND(STRBEFORE(STRAFTER(STR(?p), "#"), "_") AS ?baumart)
      FILTER(?keyword in ("Vitali", "Pflegl", "Sicherh", "Leistun", "Eignun"))

      BIND(IF(?keyword = "Vitali", xsd:int(?o), 1/0) AS ?vitali)
      BIND(IF(?keyword = "Pflegl", xsd:int(?o), 1/0) AS ?pflegl)
      BIND(IF(?keyword = "Sicherh", xsd:int(?o), 1/0) AS ?sicherh)
      BIND(IF(?keyword = "Leistun", xsd:int(?o), 1/0) AS ?leistun)
      BIND(IF(?keyword = "Eignun", ?o, 1/0) AS ?eignun)
    } GROUP BY ?rzst_f ?baumart
  }
  BIND(IRI(CONCAT(STR(wp:),"Baumartenpfleglichkeit_",
(SUBSTR(SHA256(CONCAT(STR(?rzst_f), ?baumart)),1,16)))) AS ?iri)
}
```

Codeblock 35: Unpivot-Query „Baumartenpfleglichkeit“

Die Query iteriert über Prädikate und Objekte aller Zeilen der Tabelle (`?row ?p ?o .`). Der Wert von `dat:RZST_F` wird für jede `?row` direkt extrahiert, `?keyword` und `?baumart` werden jeweils aus den Prädikat-IRIs (`?p`) extrahiert. Mithilfe eines Filters werden nur Zeilen berücksichtigt, bei denen `?keyword` einem der Werte `Vitali`, `Pflegl`, `Sicherh`, `Leistun` oder `Eignun` entspricht. Zu diesem Zeitpunkt sind `?keyword` und `?baumart` korrekt zugewiesen. Danach wird der Wert von `?o`, abhängig vom jeweiligen `?keyword`, an die entsprechende Variable (`?vitali`, `?pflegl`, `?sicherh`, `?leistun` oder `?eignun`) gebunden.

Dieser Teil der Query befindet sich in einer Subquery, welche nach `?rzst_f` und `?baumart` gruppiert. Es werden so für jeden Standort und jede Baumart genau die Werte `Vitalität`, `Pfleglichkeit`, `Sicherheit`, `Leistung` und `Eignung` extrahiert. Eine IRI wird aus `rzst_f` und `baumart` per „Minting via SHA“ generiert. Im Construct-Block werden dann die gewünschten Triple-Patterns definiert.

Codeblock 36 zeigt einen Auszug aus dem resultierenden RDF der Unpivot-Query im Turtle-Format.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX wp: <https://wald.cs.uni-freiburg.de/ontology#>

wp:Baumartenpfleglichkeit_bf080dceb3307ac4
  rdf:type wp:Baumartenpfleglichkeit ;
  wp:rzst_f 22388 ;
  wp:vitalität 3 ;
  wp:eignung "b" ;
  wp:baumart_f "BAH" .

wp:Baumartenpfleglichkeit_4309d1357c8e8b52
  rdf:type wp:Baumartenpfleglichkeit ;
  wp:rzst_f 22393 ;
  wp:vitalität 3 ;
  wp:eignung "m/w" ;
  wp:baumart_f "BAH" .

wp:Baumartenpfleglichkeit_850223fccae4139a
  rdf:type wp:Baumartenpfleglichkeit ;
  wp:rzst_f 22395 ;
  wp:vitalität 3 ;
  wp:eignung "m/w" ;
  wp:baumart_f "BAH" .
```

Codeblock 36: RDF-Auszug der Unpivot-Query „Baumartenpfleglichkeit“

5.7 Construct-Stage 2 - Linking

In diesem Abschnitt schauen wir uns die Construct-Queries der zweiten Stage an. Diese Stage nennen wir Linking, da sie nur Queries beinhaltet, die Objekte verschiedener Klassen linken. Viele Klassen können bereits während der Normalisierung gelinkt werden. Es gibt jedoch Fälle, in denen das nicht möglich oder nicht gewünscht ist. Wir haben diese Form des Linkings genutzt, um in den Construct-Queries der ersten Stage weniger Abhängigkeiten einzubauen. Codeblock 37 zeigt eine beispielhafte Linking-Query.

```
PREFIX wp: <https://wald.cs.uni-freiburg.de/ontology#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

CONSTRUCT {
  ?standort wp:baumartenpfleglichkeit ?baumartenpfleglichkeit .
} WHERE {
  ?baumartenpfleglichkeit rdf:type wp:Baumartenpfleglichkeit ;
    wp:rzst_f ?rzst_f .
  ?standort rdf:type wp:Standort ;
    wp:nr ?rzst_f .
}
```

Codeblock 37: Linking-Query von Standort und Baumartenpfleglichkeit

In dieser Construct-Query werden Objekte der Klassen `wp:Standort` und `wp:Baumartenpfleglichkeit` über einen gemeinsamen Wert miteinander gelinkt. Im Construct-Block wird dann das entsprechende Triple zum Linken definiert.

5.8 Construct-Stage 3 - Inference

In diesem Abschnitt schauen wir uns die Construct-Queries der dritten Stage an. Diese Stage nennen wir Inference. Hier werden aus den schon bestehenden Daten im Graphen neue Daten generiert. Dafür können beliebig komplexe Queries genutzt werden, um den Graphen mit ihren Ergebnissen anzureichern.

Insgesamt gibt es drei Inference-Queries im Forstgraphen, von denen wir im Folgenden zwei zeigen.

Die Query in Codeblock 38 ermittelt alle Geometrie-Objekte `?geometry` und fügt im Construct-Block das Triple `?geometry rdf:type geo:Geometry` hinzu.

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

CONSTRUCT {
  ?geometry rdf:type geo:Geometry .
} WHERE {
  ?s geo:hasGeometry ?geometry .
}
```

Codeblock 38: Inference-Query zur Typisierung von Geometrien

Die Query aus Codeblock 39 ist etwas komplexer. Ein Bestand hat das Prädikat `wp:bestandesalter`. Wenn der Bestand nur eine Teilfläche hat, dann ist das Bestandesalter eine einzelne Zahl. Wenn ein Bestand zwei Teilflächen hat, dann besteht das Bestandesalter aus zwei Zahlen, getrennt durch ein `'/'`. Das Ziel der Query ist, für jede Teilfläche ein Alter zu generieren.

```
PREFIX wp: <https://wald.cs.uni-freiburg.de/ontology#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

CONSTRUCT {
  ?teilfläche wp:alter ?alter .
}
WHERE {
  ?bestand rdf:type wp:Bestand .
  ?bestand wp:bestandesalter ?bestandesalter .
  ?bestand wp:teilfläche ?teilfläche .
  ?teilfläche wp:id ?teilfläche_id .
  BIND (IF(CONTAINS(?bestandesalter,"/"),xsd:int(STRBEFORE(?bestandesalter,"/")),xsd:int(?bestandesalter)) AS ?alter1)
  BIND (IF(CONTAINS(?bestandesalter,"/"),xsd:int(STRAFTER(?bestandesalter,"/")),1 / 0) AS ?alter2)
  BIND (IF(?teilfläche_id = "1",?alter1,?alter2) AS ?alter)
}
```

Codeblock 39: Inference-Query zur Aufteilung des Bestandesalters

5.9 Ontologie im Überblick

In den vorherigen Abschnitten führen wir die Grundbausteine für den Forstgraphen ein und zeigen einige Beispielqueries, welche diese Grundbausteine nutzen. Um einen besseren Gesamtüberblick über den Forstgraphen zu bekommen, gehen wir in diesem Abschnitt auf die abstrakte Struktur und Ontologie des Wissensgraphen ein.

Eingabe für die Forst-Pipeline sind 34 Source-Dateien und 19 Stage-Konfigurationsdateien mit insgesamt 37 Construct-Queries. Daraus werden 28 verschiedene Klassen und 121 verschiedene Prädikate generiert. Der resultierende Forstgraph besteht aus insgesamt 117.075.599 verschiedenen Triples.

Um einen Überblick über die Verbindungen der verschiedenen Klassen zu geben, haben wir in Abbildung 6 einen Graphen visualisiert, in dem jeder Knoten eine Klasse darstellt. Es gibt eine Verbindung von `Klasse1` zu `Klasse2`, wenn es ein Objekt in `Klasse1` gibt, das einen Link zu einem Objekt in `Klasse2` hat. Das Kantenlabel ist das Prädikat des Triples.

Die Klasse `geo:Geometry` haben wir bewusst aus dem Klassengraphen ausgeschlossen, da fast alle Klassen eine Verbindung zu `geo:Geometry` haben und der resultierende Klassengraph sonst unübersichtlich wäre.

Die folgenden Klassen haben außer zu `geo:Geometry` keine externen Verbindungen und werden daher zur Übersichtlichkeit nicht im Klassengraphen dargestellt:

```
Class                   FFH                   Naturschutzgebiet
Schutzgebiet            Vogelschutzgebiet     Waldbesitzfläche
```

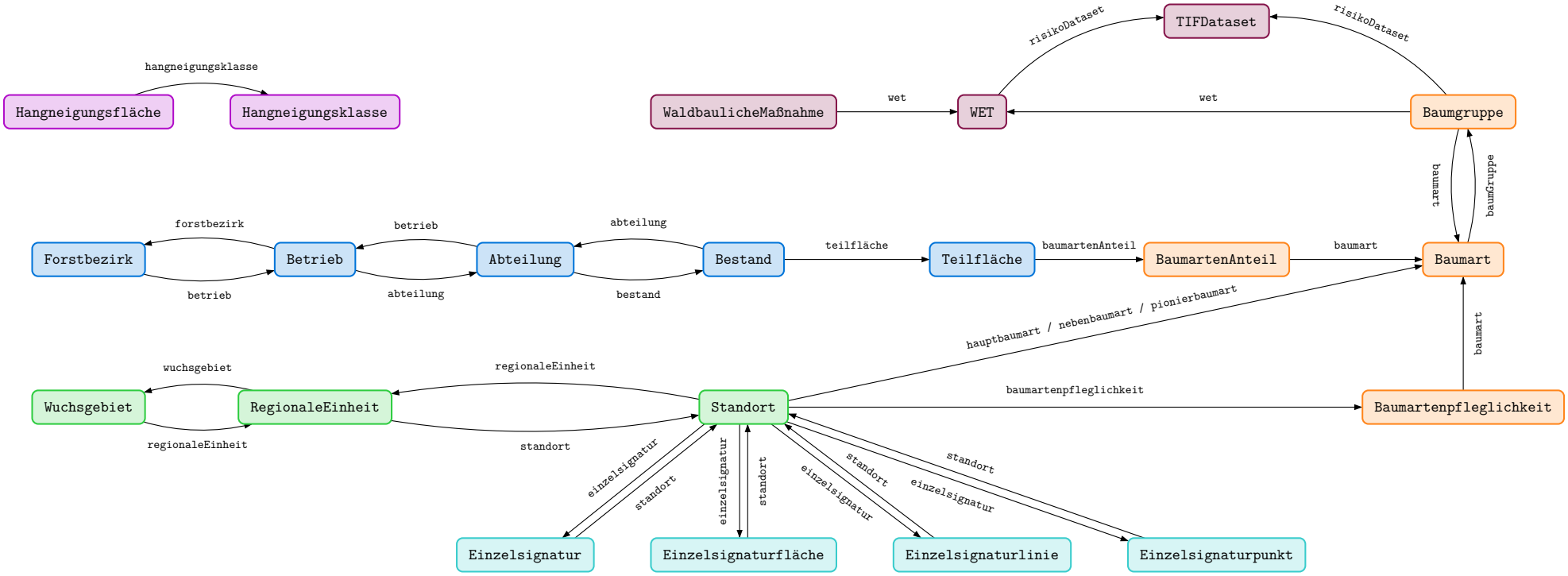

Abbildung 6: Ontologie im Überblick

6 Demoanwendungen

In diesem Kapitel zeigen wir verschiedene Demoanwendungen und eine Query, welche wir im Rahmen des WWW4.0-Projekts erstellt haben. Diese nutzen den Forstgraphen via SPARQL-Queries als Datenbasis und zeigen somit dessen Funktionalität.

6.1 Ontologie-Browser

Um die Daten im Forstgraphen einfach zugänglich und durchsuchbar zu machen, haben wir den Ontologie-Browser entwickelt, eine Single-Page-Website, welche dynamisch Informationen zu Objekten im Forstgraphen darstellt.

Die IRIs des Forstgraphen sind valide URLs. Daher bietet es sich an, Informationen für die jeweiligen IRIs über die URL im Browser anzuzeigen. Es werden eingehende und ausgehende Triples per SPARQL-Queries ermittelt und in einer Tabelle dargestellt. Weitere Objekte können wiederum angeklickt werden. So ist ein interaktives „Erforschen“ des Wissensgraphen möglich. Wir haben diese Anwendung konzipiert und mithilfe von Claude Code [40] implementiert. Abbildung 7 zeigt einen Screenshot des Ontologie-Browsers.

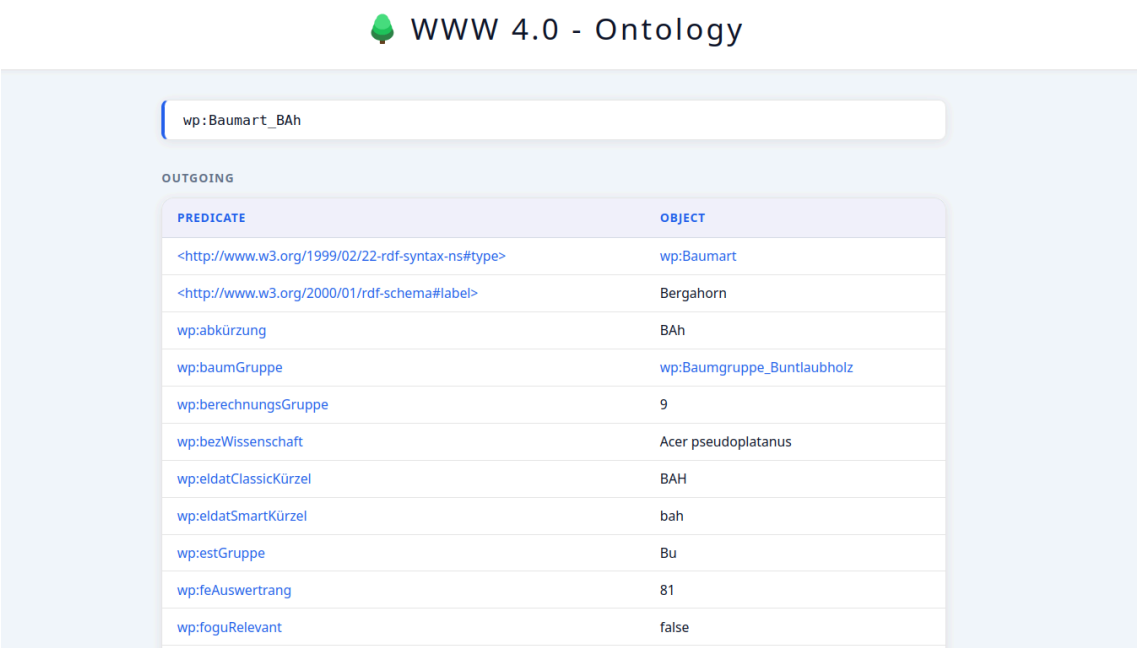

Abbildung 7: Ontologie-Browser

6.2 Ontologie-Graph

Um ein besseres Verständnis für die abstrakte Struktur des Wissensgraphen zu ermöglichen, haben wir den Ontologie-Graphen entwickelt.

Der Ontologie-Graph visualisiert den Forstgraphen ähnlich wie der Klassengraph aus Abschnitt 5.9. Knoten stellen RDF-Klassen dar und Kanten die Verbindungen von Objekten dieser Klassen. Der Unterschied ist, dass der Ontologie-Graph dynamisch über SPARQL-Queries aus dem Forstgraphen generiert wird. So können Änderungen der Ontologie in Echtzeit betrachtet werden. Wir haben diese Anwendung konzipiert und dann mithilfe von Claude Code umgesetzt. Die Anwendung nutzt Sigma.js [41] für die Visualisierung des Graphen. Abbildung 8 ist ein Screenshot des Ontologie-Graphen.

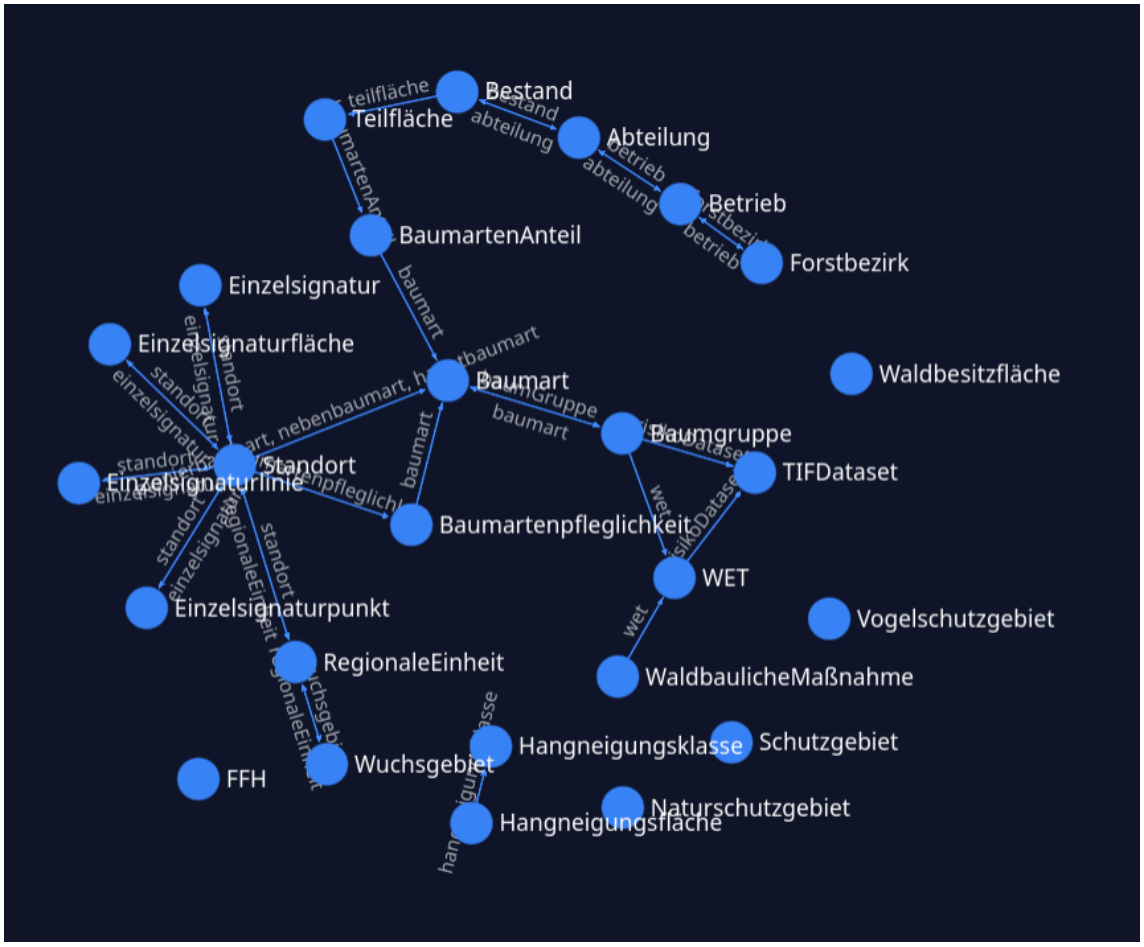

Abbildung 8: Ontologie-Graph

6.3 Demo-App

Ein Ziel des WWW4.0-Projekts ist es, eine App für den Waldbau zu entwickeln. Eine Demoversion wurde bereits von Ioannis Nezis, einem Mitglied des WWW4.0-Projekts, entworfen. Die App ist eine PWA (Progressive Web-App), welche im Browser genutzt werden kann und auf mobilen Geräten installierbar ist. Abbildung 9 zeigt einen Screenshot der Demo-App.

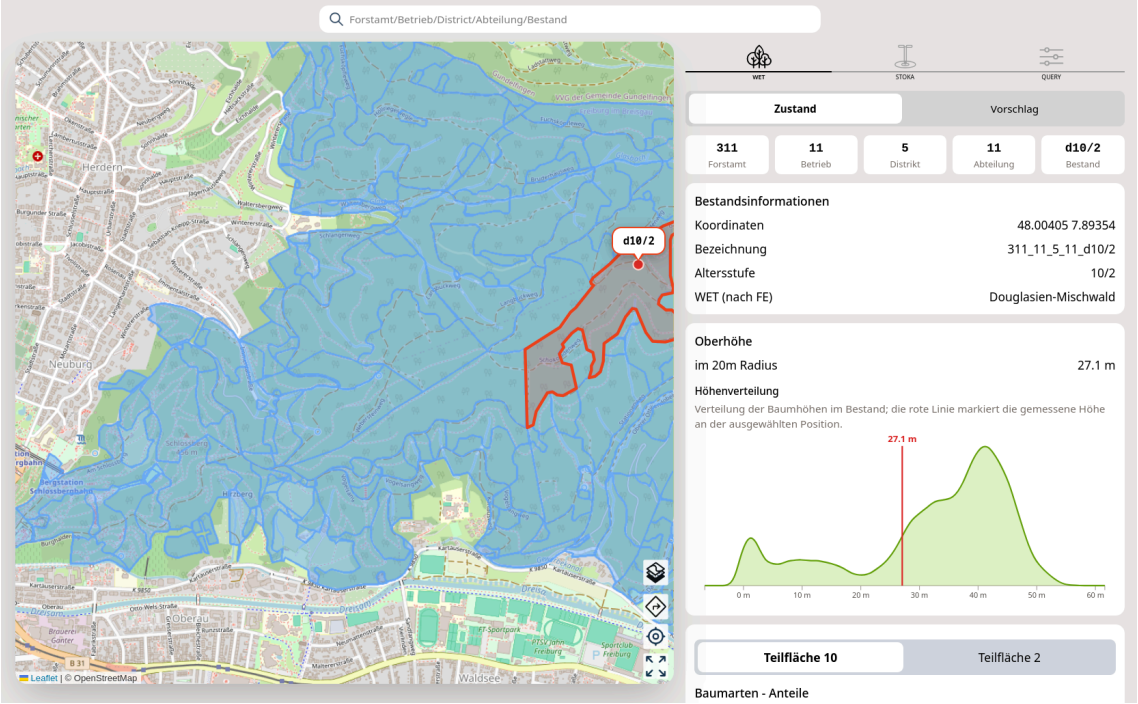

Abbildung 9: Demo-App

Die App beinhaltet drei Use-Cases: den WET-Use-Case (Waldentwicklungstyp), den STOKA-Use-Case (Standortskartierung) und den Use-Case von parametrisierten Queries. Bis jetzt ist nur der erste Use-Case fertig implementiert, weshalb wir uns auf diesen beschränken.

Wir betrachten im Folgenden den WET-Use-Case genauer, ohne dabei fachlich in die Tiefe des Waldbaus zu gehen. Der WET-Use-Case wird durch das Klicken auf die Geometrie eines Bestandes (unterste forstliche Waldeinteilung) gestartet. Eine SPARQL-Query ermittelt dann die IRI des Bestandes anhand der geklickten Koordinate. Über den Bestand werden einfache Informationen wie die Bezeichnung oder die Altersstufe ermittelt und ausgegeben (Abbildung 10).

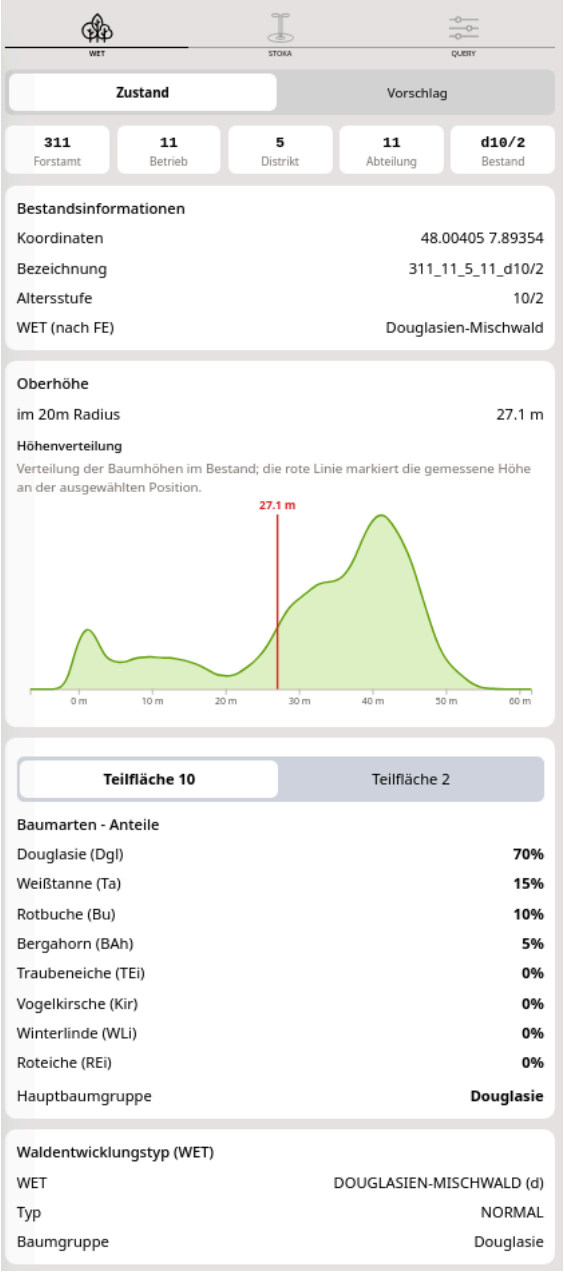

Abbildung 10: Demo-App Zustand

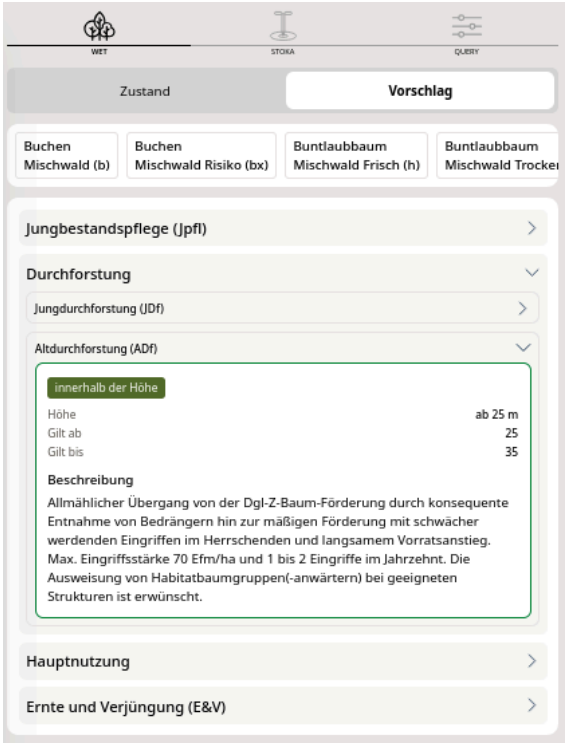

Abbildung 11: Demo-App Vorschlag

Im nächsten Schritt wird die Waldoberhöhe in einem Radius von 20 Metern um die ausgewählte Koordinate ermittelt. Dafür wird der TIFF-Proxy mit der WHSK (Waldhöhenstrukturkarte) angefragt. Außerdem wird eine Höhenverteilung der WHSK über den gesamten ausgewählten Bestand angezeigt. Darauf ist die ermittelte Waldoberhöhe markiert.

Als Nächstes werden Informationen über die Baumartenanteile innerhalb des Bestands gesammelt und dargestellt. Ein Bestand kann eine oder zwei Teilflächen haben, zwischen

denen über ein Menü gewechselt wird. Es werden separate Baumartenanteile für jede Teilfläche dargestellt. In einer weiteren Query wird die Hauptbaumgruppe ermittelt.

Danach wird aus den vorherigen Informationen sowie verschiedenen TIFF-Risikokarten der WET ermittelt. Auf der Seite `Vorschlag` (Abbildung 11) werden die waldbaulichen Maßnahmen des vorher ermittelten WETs angezeigt. Es wird dabei automatisch nach der Waldoberhöhe gefiltert, welche an der ausgewählten Koordinate ermittelt wurde.

6.4 Beispiel-Query

Codeblock 40 zeigt eine ausgewählte SPARQL-Query, die verschiedene Aspekte des Forstgraphen sowie von RDF-Wissensgraphen im Allgemeinen veranschaulicht.

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX wp: <https://wald.cs.uni-freiburg.de/ontology#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX qlproxy: <https://qlever.cs.uni-freiburg.de/qlproxy/>
PREFIX ogc: <http://www.opengis.net/rdf#>
PREFIX osmrel: <https://www.openstreetmap.org/relation/>
PREFIX osm: <https://www.openstreetmap.org/>
PREFIX osmkey: <https://www.openstreetmap.org/wiki/Key:>

SELECT DISTINCT ?dataset ?baumart ?baumhöhe ?centroid WHERE {
  # Straßen in Freiburg von OSM
  SERVICE <https://qlever.dev/api/osm-planet> {
    VALUES ?highway { "motorway" "trunk" "primary" "secondary" "tertiary"
"unclassified" "residential" "living_street" }
    osmrel:62768 ogc:sfContains ?osm_id .
    ?osm_id osmkey:highway ?highway ;
            rdf:type osm:way ;
            geo:hasGeometry/geo:asWKT ?streetwkt .
  }
  {
    {
      # Bestände in denen es einen Eschenanteil gibt
      SELECT ?geometry WHERE {
        ?betrieb rdf:type wp:Betrieb ;
                 rdfs:label "Stadtwald Freiburg im Breisgau" ;
                 wp:abteilung ?abteilung .
        ?abteilung wp:bestand ?bestand .
        ?bestand geo:hasGeometry/geo:asWKT ?geometry ;
                 wp:teilfläche ?teilfläche .
        ?teilfläche wp:baumartenAnteil ?baumartenanteil .
        ?baumartenanteil wp:baumart wp:Baumart_Es .
      }
    }
    # Qlproxy tif-punkte anfragen
    BIND ("merged" AS ?dataset)
    SERVICE qlproxy: {
      _:config qlproxy:endpoint <https://wald.cs.uni-freiburg.de/tifproxy> ;
               qlproxy:input-dataset ?dataset ;
               qlproxy:input-filter_wkt ?geometry ;
               qlproxy:output-band_1 ?baumart ;
               qlproxy:output-band_2 ?baumhöhe ;
               qlproxy:output-centroid ?centroid ;
```

```
44                  qlproxy:output-row ?row .
45      }
46    }
47    # Filter
48    FILTER (?baumhöhe > 5) # Filtern nach Baumhöhe
49    FILTER (?baumart = 8) # Filtern nach Baumart
50    FILTER (geof:metricDistance(?centroid,?streetwkt) <= 50) # Abstand zur
Straße
51  }
```

Codeblock 40: SPARQL-Query zur Ermittlung potenziell gefährdeter Eschen

Wir erklären im Folgenden die grobe Struktur und Funktionalität der Query, ohne dabei auf konkrete SPARQL-Syntax einzugehen. Zuerst werden die Geometrien aller öffentlichen Straßen innerhalb Freiburgs vom OpenStreetMap-RDF-Graphen [42] per SPARQL-Service-Request abgefragt.

Danach werden alle Bestände im Stadtwald Freiburg mit einem Eschenanteil ausgewählt. Anschließend werden alle Pixel-Koordinaten und Werte der TIFF-Datei `merged` innerhalb dieser Eschenbestände ermittelt. Die TIFF-Datei ist ein Verschnitt der WHSK (Waldhöhenstrukturkarte) und der Baumartenkarte, welche Baumarten klassifiziert.

Daraufhin werden alle Pixel-Koordinaten mit Bäumen größer als 5 Meter und einer „sonstigen Baumart“ ausgewählt. Eschen sind nicht als einzelne Klasse im Baumarten-Datensatz klassifiziert, sondern fallen unter den Punkt „sonstige Baumart“. Außerdem wird nach Entfernung zu einer öffentlichen Straße von weniger als 50 Metern gefiltert.

Das Ergebnis der Query ist eine Menge von Pixeln, bei denen alle Filter zutreffen. Es ist eine grobe Abschätzung dafür, wo Eschen stehen könnten, die eine Gefahr für den Straßenverkehr darstellen. In Abbildung 12 ist das Ergebnis mit Petrimaps als Heatmap dargestellt. Diese Query zeigt, wie einfach es möglich ist, verschiedene Daten aus dem Wissensgraphen sowie von externen Wissensgraphen zu kombinieren, um komplexe Fragestellungen zu beantworten.

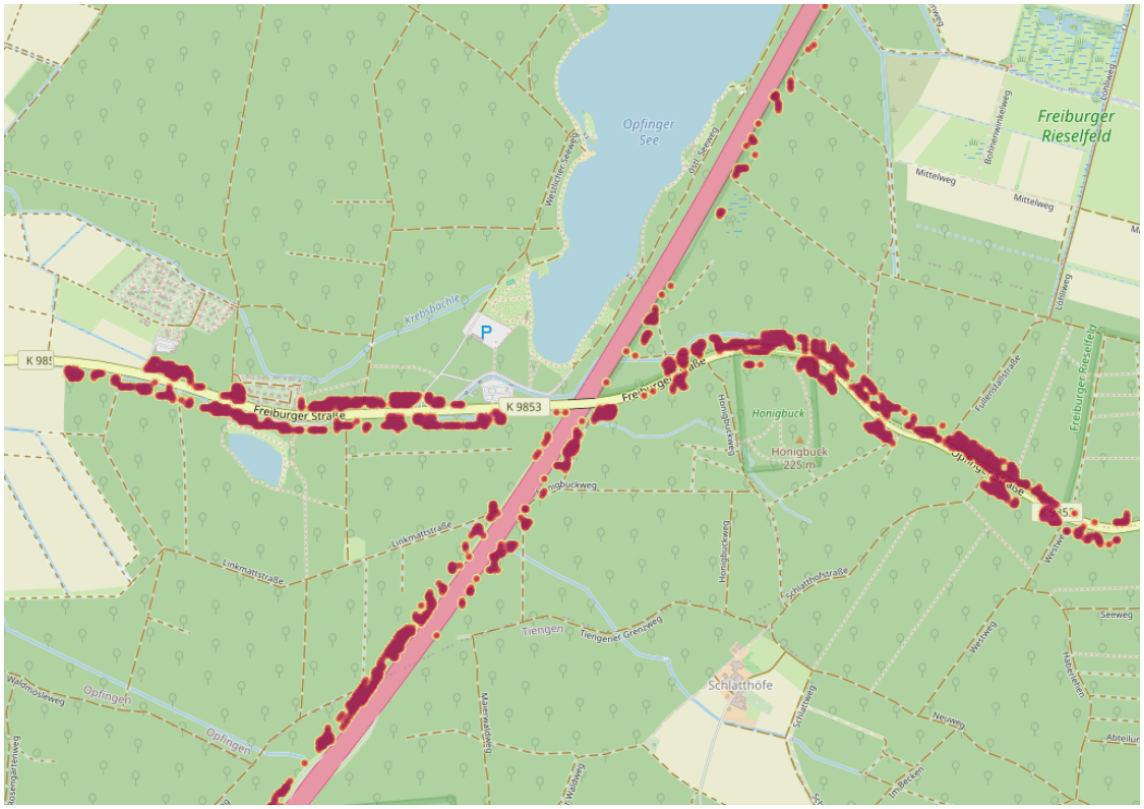

Abbildung 12: Potenziell gefährdete Eschen in der Nähe von öffentlichen Straßen

7 Theoretische Analyse

Die Qonstructor-Pipeline wandelt in mehreren Schritten Rohdaten in RDF um. Der Forstgraph wird mithilfe von Qonstructor aufgebaut. Für dieses Projekt sind die schnelle Generierung und die allgemeine Funktionalität des Forstgraphen die relevanten Ziele. Die Funktionalität wird bereits in Kapitel 6 demonstriert. In der theoretischen und empirischen Analyse widmen wir uns daher der Laufzeit der Forst-Pipeline.

Da Qonstructor kein singuläres Programm ist, sondern externe Software orchestriert, hängt die Laufzeit der einzelnen Stages von externer Software ab. Eine theoretische Laufzeitanalyse oder Speicheranalyse von Qonstructor im Ganzen ist für diese Arbeit daher zu komplex. Deshalb fokussieren wir uns in der theoretischen Analyse auf einzelne Aspekte von Qonstructor, welche für die Laufzeit der Forst-Pipeline am relevantesten sind.

7.1 Laufzeit von CSV2RDF

Wir analysieren zuerst die Laufzeit und den Speicherverbrauch von CSV2RDF. Für den Forstgraphen werden 24 von 34 Source-Dateien mit CSV2RDF in generisches RDF umgewandelt. Bei TIFF-Dateien werden nur Triples zu den Metadaten generiert. Die einzige JSON-Datei ist vergleichsweise klein. Daher sind TIF2RDF und JSON2RDF für die Gesamtlaufzeit der Forst-Pipeline vernachlässigbar.

Für die folgende theoretische Analyse definieren wir n als Dateigröße der Eingabedatei in Zeichen.

CSV2RDF ist ein C++-Programm und startet in der `main()`-Funktion. Es werden zuerst die Argumente geparst und die Input/Output-Streams erstellt. Es gilt für den Anfang des Programms also $T_{\text{init}}(n) \in O(1)$. Danach wird die eigentliche Funktion `processCSV()` aufgerufen.

Es werden verschiedene String-Operationen definiert, welche wiederum von `processCSV()` genutzt werden. Dazu gehören `parseCsvLine()`, `trim()`, `sanitizeForUri()`, `escapeString()` und `isNumeric()`. All diese Funktionen iterieren maximal konstant oft über den Eingabestring. Es gilt also $\forall f \in \{\text{parseCsvLine}, \text{trim}, \text{sanitizeForUri}, \text{escapeString}, \text{isNumeric}\}$: $T_f(i) \in O(i)$, wobei i die Länge des Eingabestrings bezeichnet.

Die Funktion `processCSV()` kann grob in zwei Schritte getrennt werden. Der erste Schritt ist das Loading. Hier wird der Header mit `std::getline()` eingelesen und per `parseCsvLine()` geparst. Danach werden alle weiteren Zeilen in einer Schleife mit `std::getline()` eingelesen

und in einem `std::vector<std::string>` gespeichert. Da hier einmal über die gesamte Eingabedatei iteriert wird, gilt also $T_{\text{load}}(n) \in O(n)$.

Der zweite Schritt ist das Processing. Hier wird über alle vorher gespeicherten Zeilen iteriert. In einer Schleife wird jede Zeile per `parseCsvLine()` geparst und über alle resultierenden Felder iteriert. In dieser Iteration wird dann noch der Spezialfall der Subdelimiter behandelt. Insgesamt wird in diesem Schritt maximal konstant oft über die gesamte Eingabe iteriert. In verschiedenen Teilen der Schleifen werden Triples generiert und über den Output-Stream ausgegeben. Es wird jedoch maximal ein Triple pro Feld generiert. Das dabei geschriebene Triple besteht aus dem Feldwert plus Präfix, Spaltenname und Zeilenidentifier. Der Feldwert ist Teil der Eingabe und zählt somit zu n dazu. Der Rest ist nach oben durch eine Konstante begrenzt. Die Generierung der Triples erhöht somit nicht die Laufzeit des Processing-Schritts. Es gilt also $T_{\text{calc}}(n) \in O(n)$.

Insgesamt gilt für die Laufzeit also $T_{\text{CSV2RDF}}(n) = T_{\text{init}}(n) + T_{\text{load}}(n) + T_{\text{calc}}(n)$. Da alle Terme in $O(n)$ liegen, gilt auch $T_{\text{CSV2RDF}}(n) \in O(n)$. Für den Speicherverbrauch ergibt sich durch den Loading-Schritt $S_{\text{CSV2RDF}}(n) \in O(n)$.

7.2 Intra-Stage-Parallelisierung

Wir schauen uns nun die Parallelisierung der Stages an. Dies ist ein relevanter Aspekt für die Gesamtlaufzeit der Forst-Pipeline, da so theoretisch eine Reduktion der Laufzeit einzelner Stages um den Faktor der Anzahl an Workern möglich ist. Wir betrachten nur den Teil der Stages, der parallelisierbar ist, also klammern wir hier erst einmal den Index-Bau und Start-Prozess von QLever aus.

In der Generic-Generation-Stage besteht ein Task aus dem Generieren von RDF für eine Rohdatei. Dafür werden externe Shell-Kommandos aufgerufen. Die eigentliche Arbeit passiert also nicht innerhalb des Python-Interpreters. Die extern aufgerufenen Programme laufen im Fall der Forst-Pipeline jeweils nur in einem Prozess.

Für die Construct-Stage ist ein Task das Ausführen einer SPARQL-Construct-Query. Hier wird die Query an den lokalen QLever-Server gesendet und von diesem bearbeitet. Auch hier passiert die eigentliche Arbeit also nicht innerhalb des Python-Interpreters. Für den QLever-Server kann die Option `num_threads` gesetzt werden, welche die maximal parallel bearbeiteten Queries angibt. Eine genaue Analyse des Multi-Threading-Verhaltens von QLever ist hier jedoch nicht Teil der Arbeit. Daher setzen wir für die weitere Analyse `num_threads` immer gleich der Anzahl an Workern und nehmen an, dass jede Query genau einen Worker beansprucht.

Die Generic-Generation-Stage sowie die Construct-Stages sind per List-Scheduling parallelisiert. Die Tasks brauchen untereinander keine weitere Kommunikation und sind somit „embarrassingly parallel“.

Python-Threads sind wegen des GIL (Global Interpreter Lock) normalerweise nicht echt parallel. Da sowohl bei der Generic-Generation-Stage als auch den Construct-Stages die eigentliche Arbeit außerhalb des Python-Interpreters passiert, ist hier der Einfluss des GIL nicht relevant. Die Tasks können somit tatsächlich parallel auf mehreren CPU-Kernen ausgeführt werden.

Während die Python-Threads auf die Ergebnisse warten, brauchen sie keine CPU-Zeit. Somit kann vereinfacht jeder Task als eigene Einheit betrachtet werden, der auf genau einem Worker läuft.

So können wir die Parallelisierung der Tasks auf den Workern innerhalb einer einzelnen Stage mithilfe von Graham (1966) [43] modellieren. Dafür definieren wir N als die Anzahl an Tasks, t_i als die Laufzeit von Task i, W als die Anzahl an Workern und $T = \sum_{i=1}^{N} t_i$ als die addierte Laufzeit aller Tasks. Dann gilt für die Gesamtbearbeitungszeit M:

$$\max\left(\frac{T}{W}, \max_i t_i\right) \leq M \leq \frac{T}{W} + \max_i t_i$$

Im besten Fall dauern alle Tasks genau gleich lange, dann gibt es einen Speedup um ungefähr W. Der schlechteste Fall wäre, wenn ein einzelner sehr langer Task dominieren würde und somit mehr Worker kein weiteres Speedup bringen würden. Ein solcher dominanter Task wird auch Straggler genannt.

7.3 End-to-End-Laufzeitanalyse

Die Qonstructor-Pipeline durchläuft mehrere Schritte und Stages. Der allgemeine Programmablauf ist sequentiell. Manche Teile der Stages sind jedoch parallelisiert.

Das Laden der Konfigurationsdateien und das Cleanen der Ordner laufen sequentiell. In jeder Stage gibt es einen parallelisierbaren Anteil, bei dem die Tasks parallel laufen. Danach wird in jeder Stage ein QLever-Index gebaut. Dieser Prozess wird von QLever selbst möglichst gut parallelisiert. Danach wird der QLever-Server gestartet. Da wir hier das Multi-Threading-Verhalten von QLever nicht genauer untersuchen können, definieren wir den QLever-Index-Bau und das Starten des QLever-Servers als sequentiell. Die Export-Stage am Ende führt nur eine Construct-Query aus und ist somit auch sequentiell.

Die sequentiellen Teile profitieren nicht von mehr Workern. Um das zu modellieren, benutzen wir das Gesetz von Amdahl [44].

Dafür definieren wir $P \in [0, 1]$ als den parallelisierbaren Anteil der Forst-Pipeline und $1 - P$ als den nicht parallelisierbaren Anteil. W ist die Anzahl an Workern für die Generic-Generation-Stage und die Construct-Stages. Wir definieren $T_{\text{parallel}} = T_0 + T_1 + T_2 + T_3$ als die Summe der Laufzeit aller Tasks in allen vier Stages und definieren T_{seriell} als Summe aller restlichen Teile der Pipeline. Dann gilt $P = \frac{T_{\text{parallel}}}{T_{\text{parallel}} + T_{\text{seriell}}}$. $S(W)$ ist dann der mögliche Speedup:

$$S(W) = \frac{1}{(1 - P) + \frac{P}{W}}$$

Obwohl die parallelisierbaren Anteile fast linear mit mehr Workern skalieren, ist der Speedup der Gesamtpipeline gedeckelt. Der maximal mögliche Speedup ist $\lim_{W \to \infty} S(W) = \frac{1}{1-P}$. Sobald dieser Deckel erreicht wird, lohnt es sich also nicht, die reine Worker-Anzahl zu erhöhen. Dann müssen andere Optimierungen durchgeführt werden.

8 Empirische Analyse

In diesem Kapitel überprüfen wir die theoretische Analyse aus Kapitel 7 empirisch. Tabelle 4 zeigt die Spezifikationen der Maschine, auf der wir alle Experimente durchgeführt haben.

Eigenschaft	**Wert**
Prozessor	AMD Ryzen 9 5900X
Physische Kerne	12
Logische Kerne	24
Maximale Taktfrequenz	4,95 GHz
Arbeitsspeicher	128 GiB
Betriebssystem	GNU/Linux
Kernel	7.1.3-070103-generic
Architektur	x86_64

Tabelle 4: Spezifikationen der Testmaschine

8.1 Laufzeit von CSV2RDF

Um die Laufzeit von CSV2RDF zu messen, haben wir aus der Datei `Bestände.csv` ein Sample von 1.000 Zeilen extrahiert. Diese Datei beinhaltet Geometrien als WKT-Strings und ist somit repräsentativ für einen Großteil der CSV-Dateien aus der Forst-Pipeline. Daraus haben wir durch Aneinanderreihen insgesamt 13 Testdatensätze in Größen von 10.000 bis 4.000.000 Zeilen generiert. Anschließend haben wir CSV2RDF auf jedem Testdatensatz fünfmal ausgeführt. Tabelle 5 zeigt den Mittelwert und die Standardabweichung der jeweiligen fünf Durchläufe. Abbildung 13 stellt die Daten mit log-Skalierung auf der x- und y-Achse sowie einer linearen Regressionsgeraden dar.

Zeilen	**Mittelwert (s)**	**Standardabweichung (s)**
10000	0.336	0.009
20000	0.668	0.004
40000	1.334	0.011
60000	2.004	0.023
80000	2.658	0.013
100000	3.402	0.116
200000	6.74	0.126
400000	15.002	3.411
600000	20.2	0.231

Zeilen	Mittelwert (s)	Standardabweichung (s)
800000	27.354	0.381
1000000	36.374	3.413
2000000	68.62	0.218
4000000	138.97	1.554

Tabelle 5: Laufzeit von CSV2RDF nach Zeilenanzahl

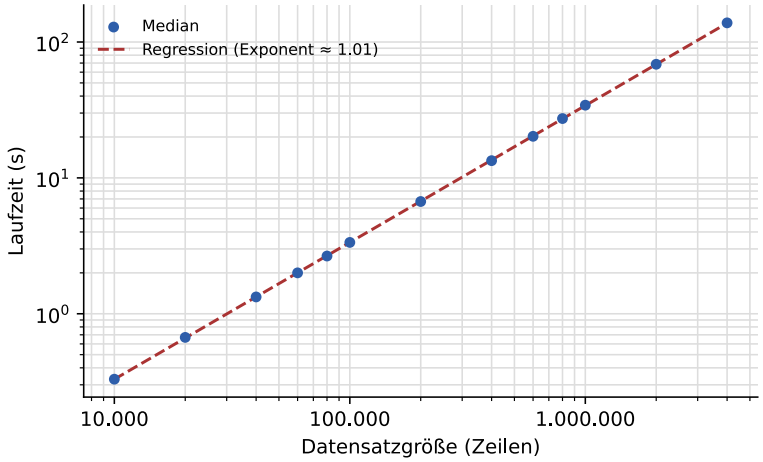

Abbildung 13: Laufzeit von CSV2RDF

Der berechnete Exponent aus der linearen Regression beträgt $\approx 1,01$. Das liegt sehr nahe am theoretischen Wert von 1 für lineare Laufzeit. Die Messung bestätigt somit eine annähernd lineare Laufzeit von CSV2RDF.

8.2 Intra-Stage-Parallelisierung

Für die empirische Evaluation der Intra-Stage-Parallelisierung betrachten wir die Generic-Generation-Stage und die erste Construct-Stage getrennt voneinander. Wir betrachten nur die erste Construct-Stage, da hier ein Großteil aller Construct-Queries ausgeführt wird.

Zuerst haben wir die beiden Stages mit jeweils nur einem Worker ausgeführt. Abbildung 14 und Abbildung 15 zeigen jeweils die Laufzeit der zehn längsten Tasks beider Stages. Auf der y-Achse befinden sich jeweils die einzelnen Tasks und auf der x-Achse die jeweiligen Ausführungszeiten in Sekunden.

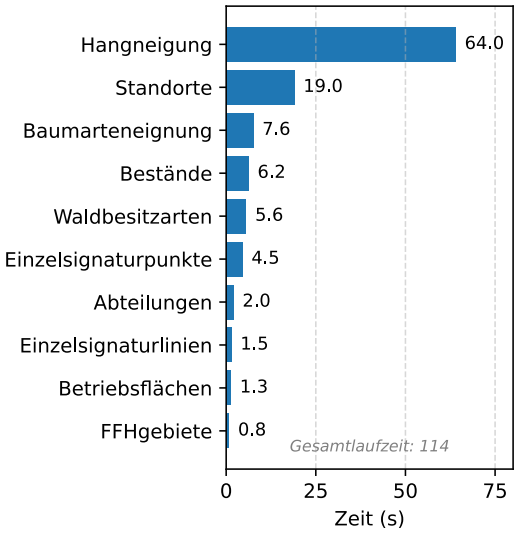

Abbildung 14: Laufzeit der Generic-Generation-Stage pro Task

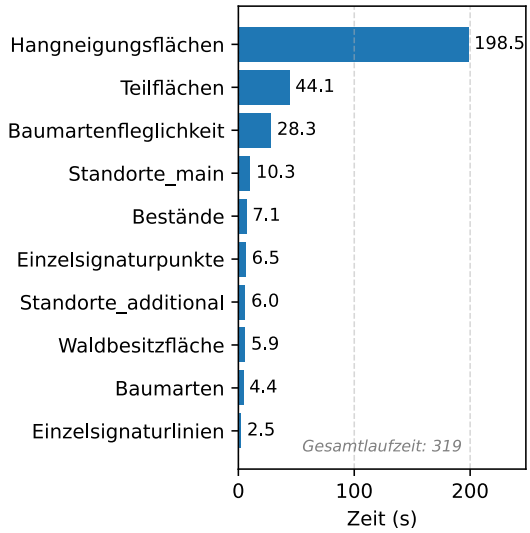

Abbildung 15: Laufzeit der Construct-Stage pro Task

Bei der Generic-Generation-Stage ist die Source-Datei mit dem Namen `Hangneigung` mit 64s am teuersten. Die nächstteuerste Source ist `Standorte`. Das liegt an der Dateigröße der Sources. Die `Hangneigung` hat ca. 15.000.000 Zeilen und die `Standorte` nur ca. 500.000.

Bei der Construct-Stage braucht die Generierung der `Hangneigungsflächen` 198,5s. Die nächste Query braucht nur 44,1s. Das liegt wiederum an der Menge an Triples, die durch die Generic-Generation-Stage aus der `Hangneigung` generiert werden. Die beiden nächsten Queries `Teilflächen` und `Baumartenpfleglichkeit` sind Unpivot-Queries, auf die wir in Abschnitt 5.6.7 genauer eingehen.

Mithilfe der Gesamtlaufzeit aller Tasks $T = \sum_{i=1}^{N} t_i$ und der Laufzeit des längsten Tasks $\max_i t_i$ lässt sich die Graham-Schranke, wie in Abschnitt 7.2 beschrieben, für eine beliebige Anzahl an Workern W berechnen. Im Folgenden berechnen wir beispielhaft die Graham-Schranke für die Generic-Generation-Stage mit $W = 16$:

$$\max\left(\frac{T}{W}, \max_i t_i\right) \leq M \leq \frac{T}{W} + \max_i t_i$$

$$\Rightarrow \max\left(\frac{114}{16}, 64\right) \leq M \leq \frac{114}{16} + 64$$

$$\Rightarrow 64 \leq M \leq 71,13$$

Als Nächstes haben wir beide Stages mit jeweils $W \in (1, 2, 4, 8, 16)$ Workern ausgeführt. Dabei bestimmt W die Variablen `construct_workers`, `generic_workers` und `num_threads` in der Pipeline-Config.

Abbildung 16 und Abbildung 17 sind Scatter-Plots für die Generic-Generation-Stage und die erste Construct-Stage. Auf der x-Achse befindet sich jeweils die Anzahl an Workern und auf der y-Achse die Ausführungszeit. Die berechnete Graham-Schranke ist grau hinterlegt.

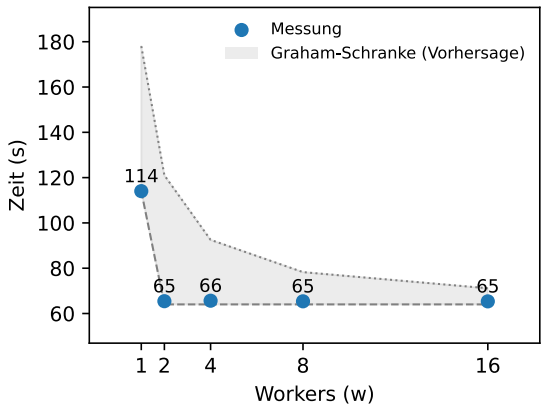

Abbildung 16: Laufzeit der Generic-Generation-Stage mit verschiedenen Workern

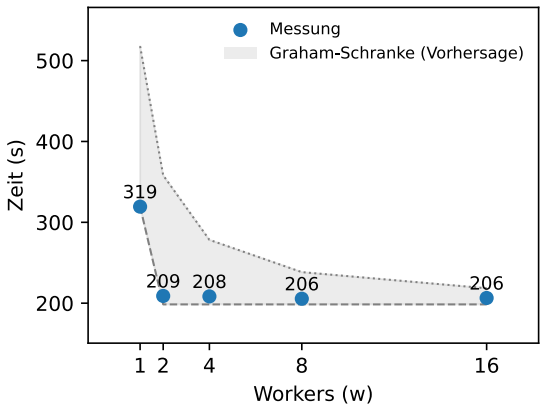

Abbildung 17: Laufzeit der Construct-Stage mit verschiedenen Workern

Bei beiden Stages ist von einem bis zwei Workern eine deutliche Verbesserung zu erkennen. Bei der Generic-Generation-Stage und der Construct-Stage ist der Speedup von einem zu zwei Workern jeweils $\frac{114}{65} \approx 1,75$ und $\frac{319}{209} \approx 1,53$. Bei mehr Workern wird der Speedup nicht erkennbar besser.

Die gemessenen Ergebnisse beider Stages liegen ungefähr auf der unteren Grenze der Graham-Schranke. Bei beiden Stages gibt es einen Straggler, welcher die Optimierung durch Parallelisierung begrenzt. In Abbildung 14 und Abbildung 15 sind die Straggler deutlich sichtbar. Sie dominieren die Laufzeit jeweils schon ab $W \geq 2$. Auch für $W \to \infty$ werden beide Stages nicht schneller laufen als ihr langsamster Task.

8.3 End-to-End-Laufzeitanalyse

Um die Laufzeit der gesamten Forst-Pipeline empirisch zu untersuchen, haben wir jeweils die gesamte Pipeline für alle $W \in (1, 2, 4, 8, 16)$ Worker ausgeführt.

Um diese Ergebnisse besser zu verstehen, haben wir die einzelnen Stages der Pipeline in Abbildung 18 dargestellt. Die x-Achse zeigt die einzelnen Stages jeweils mit einem und 16 Workern. Die y-Achse zeigt die Zeit in Sekunden.

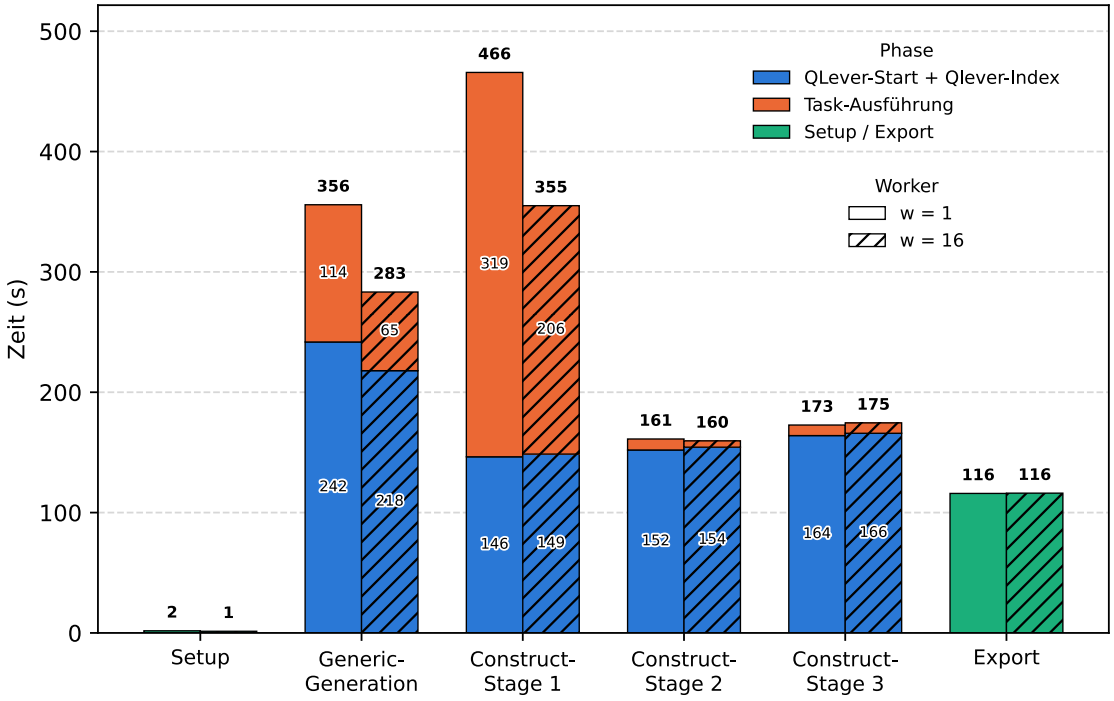

Abbildung 18: Laufzeit aller einzelnen Stages für $W = 1$ und $W = 16$

Das Cleanup und das Config-Parsing sind vernachlässigbar. Der QLever-Index und der QLever-Start brauchen bei einem oder 16 Workern ungefähr gleich lange. Das liegt daran, dass diese Teile unabhängig von der Anzahl an Workern QLever-intern parallelisiert sind.

QLever-Index und QLever-Start brauchen in der Generic-Generation-Stage länger als in der ersten Construct-Stage. Das liegt an dem deutlich größeren generischen RDF im Vergleich zu dem daraus produzierten normalisierten RDF. In der zweiten und dritten Construct-Stage steigen die QLever-Index- und QLever-Start-Zeiten wieder leicht an, da hier jeweils mehr Triples dazukommen.

Die Task-Ausführungen der Generic-Generation-Stage und der ersten Construct-Stage spiegeln wider, was in Abschnitt 8.2 behandelt wird.

Die zweite und dritte Construct-Stage haben einen sehr kleinen Task-Anteil, da hier nur wenige Queries ausgeführt werden, die insgesamt wenige Triples generieren. Trotzdem muss danach jeweils der QLever-Index neu gebaut werden.

Wir können aus den gegebenen Werten mithilfe der Amdahl-Formel den möglichen Speedup für W Worker voraussagen. Wir berechnen diesen exemplarisch für $W = 16$:

Die Amdahl-Formel lautet:

$$S(W) = \frac{1}{(1-P) + \frac{P}{W}}$$

Wir können den parallelisierbaren Teil P einfach aus der Gesamtlaufzeit der Pipeline sowie der Task-Ausführungszeit der Stages berechnen (bei $W = 1$).

$$P = \frac{114 + 319 + 9 + 9}{1273} = 0,35$$

Daraus folgt der nicht parallelisierbare Teil:

$$1 - P = 1 - 0,35 = 0,65$$

Der maximal berechnete Speedup nach Amdahl bei 16 Workern ist also:

$$S(16) = \frac{1}{0,65 + \frac{0,35}{16}} = 1,49$$

Abbildung 19 stellt die tatsächliche Laufzeit der gesamten Forst-Pipeline für alle Worker $W \in (1, 2, 4, 8, 16)$ dar. Die berechnete Amdahl-Vorhersage ist grau hinterlegt.

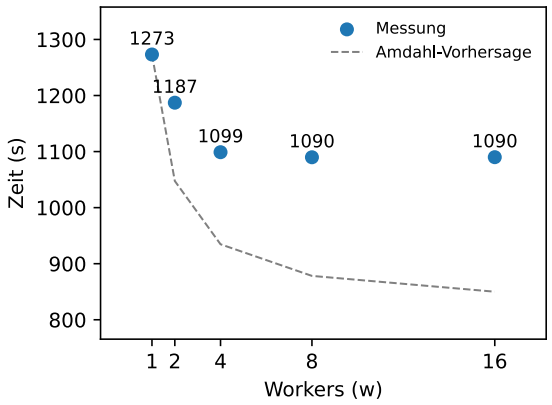

Abbildung 19: Gesamtlaufzeit der Pipeline mit verschiedenen Workern

Von einem zu vier Workern gibt es einen Speedup von $\frac{1273}{1099} \approx 1,16$. Der Speedup von einem zu 16 Workern ist $\frac{1273}{1090} \approx 1,17$. Von vier zu 16 Workern gibt es also keine relevante Verbesserung der Geschwindigkeit.

Amdahl sagt für 16 Worker den Speedup von $1,49$ voraus, was deutlich größer ist als der gemessene Wert von $1,17$. Allgemein verlaufen die gemessenen Werte deutlich über der Amdahl-Vorhersage. Das liegt daran, dass Amdahl annimmt, der parallelisierbare Anteil lasse sich mit wachsendem W beliebig verkleinern. In Abschnitt 8.2 wird aber gezeigt, dass es Straggler gibt, welche den Speedup deckeln.

Für die Forst-Pipeline ist eine Anzahl von vier Workern sinnvoll, da mehr Worker keine wesentliche Verbesserung des Speedups bewirken.

9 Fazit und Ausblick

In diesem Kapitel geben wir zuerst einen Rückblick auf die Arbeit und gehen dann auf mögliche Weiterentwicklungen ein.

Das finale Ziel dieser Arbeit ist das Erstellen eines Forstgraphen, der einen realen Nutzen für den Waldbau hat. In Kapitel 6 haben wir die Funktionalität des Forstgraphen anhand von Beispielen belegt. Zur Generierung des Forstgraphen haben wir Qonstructor entwickelt, eine domänenunabhängige RDF-Generierungspipeline. Qonstructor hat das iterative und effiziente Designen sowie Bauen des Forstgraphen ermöglicht. Wir haben das Laufzeitverhalten der Forst-Pipeline untersucht und sind zu dem Schluss gekommen, dass mehr als vier parallele Worker für die Tasks in den Stages keinen relevanten Speedup bringen.

Trotz Erreichen der ursprünglichen Ziele gibt es viel Verbesserungspotenzial für Qonstructor und den Forstgraphen. Wir gehen im ersten Schritt auf mögliche Weiterentwicklungen von Qonstructor ein. Danach zeigen wir mögliche Features auf, die außerhalb der aktuellen Architektur liegen. Dabei gehen wir gemäß einer logischen Implementierungsreihenfolge vor.

9.1 Verbesserungen der aktuellen Architektur

Wir gehen zuerst auf konkrete Verbesserungsmöglichkeiten von Qonstructor ein, welche die Geschwindigkeit der Forst-Pipeline verbessern würden. Anschließend widmen wir uns noch möglichen Verbesserungen der Benutzerfreundlichkeit.

Die zweite und dritte Construct-Stage generieren nur wenige Triples und brauchen für die Tasks daher nur sehr kurz. Das Bauen des QLever-Index dieser Stages braucht unverhältnismäßig länger. Hier könnten SPARQL-Update-Queries genutzt werden, um Triples direkt in den Graphen einzufügen. Dann muss der QLever-Index nicht neu gebaut werden. SPARQL-Update-Queries können auf dem QLever-Server jedoch nicht parallel ausgeführt werden, da sie potenziell voneinander abhängen. Hier müsste getestet werden, ab wann sich SPARQL-Update-Queries lohnen.

Der Speedup durch mehr Worker für die Tasks der ersten beiden Stages ist jeweils durch einen Straggler begrenzt. Ein Lösungsansatz wäre hier das Parallelisieren innerhalb eines Tasks. CSV2RDF kann so erweitert werden, dass eine Datei mit mehreren parallelen Workern in RDF umgewandelt werden kann. So könnte der Straggler der Generic-Generation-Stage beseitigt werden. Für die Construct-Stages müsste die interne Parallelisierung des QLever-Servers berücksichtigt werden.

Es gibt jedoch auch allgemeine Verbesserungsmöglichkeiten von Qonstructor, unabhängig von der konkreten Forst-Pipeline. Beispielsweise ist die CLI von Qonstructor bisher sehr minimal und könnte um Tooling erweitert werden, welches das Arbeiten mit der Pipeline erleichtert. Ein Beispiel wären Kommandos zum expliziten Starten und Stoppen der QLever-Server einzelner Stages.

Die angesprochenen Verbesserungen in Qonstructor zu integrieren, würde nach unserer Einschätzung ungefähr eine Woche dauern.

9.2 Workflow-Orchestration-Framework

Die Qonstructor-Pipeline ist ein All-in-one-Tool, das Daten in ihrer rohen Form zu finalem RDF umwandelt. Dabei übernimmt Qonstructor Aufgaben der Workflow-Orchestrierung, Steuerung von QLever, Handling von Dateien und so weiter. Nachteile eines solchen monolithischen Designs sind geringe Flexibilität und hohe Komplexität.

Ein alternativer Ansatz besteht darin, für einzelne Bereiche existierende spezialisierte Tools einzusetzen oder eigene Tools zu entwickeln. Durch die Kombination dieser Tools könnten dann komplexere Aufgaben realisiert werden.

Bei Qonstructor könnte z.B. die Workflow-Orchestrierung in ein externes Tool wie Dagster [45] oder Prefect [46] ausgelagert werden. Diese Tools beinhalten Features wie eine Benutzeroberfläche, Retry Handling, Scheduling, Monitoring, Observability, Caching von Ergebnissen und so weiter. Diese Features in Qonstructor selbst einzubauen, wäre mit einem erheblichen Entwicklungsaufwand verbunden und würde unnötige Komplexität einführen.

Um die aktuelle Qonstructor-Pipeline, mit allen Konfigurationen des Forstgraphen, auf ein Workflow-Orchestration-Framework umzubauen, würden wir vermutlich zwei bis drei Wochen brauchen.

9.3 Datenaggregation und Preprocessing

Wir haben die Source-Daten des Forstgraphen manuell zusammengeführt und auch manuell das Preprocessing ausgeführt. Eine Verbesserung wäre es, alle Daten automatisch aus verschiedenen Quellen zu laden und auch das Preprocessing automatisch zu starten.

Im Folgenden skizzieren wir eine mögliche Umsetzung: Es könnten alle Daten in einem Object-Store gespeichert werden. Dieser kann auch lokal, z.B. mit MinIO [47], gehostet werden. Das ermöglicht Features wie Web-Zugriff, Access-Control und Versionierung. Daten, die sich nicht ändern und einmalig geliefert werden, kommen dauerhaft in

den Object-Store. Bei diesen kann der Schreibzugriff entfernt werden. Daten, welche regelmäßig aktualisiert werden, können bei Änderungen automatisch von der Pipeline heruntergeladen werden. Wenn sich eine Datengrundlage ändert, werden alle weiteren Schritte wie das Preprocessing automatisch von der Pipeline ausgeführt.

Ein solcher Prozess ist mit einem Workflow-Orchestration-Framework umsetzbar. Nach unserer Einschätzung würde diese Implementierung ungefähr ein bis zwei Wochen in Anspruch nehmen, wenn die nötigen Daten schon online abrufbar sind.

9.4 Datenvalidierung

Aktuell validieren wir die resultierenden RDF-Daten nur „per Hand“ mit SPARQL-Queries. Eine Verbesserung wäre es, die Pipeline um einen festen Validierungsschritt zu erweitern. Dieser könnte SHACL [48] (Shapes-Constraint-Language) nutzen.

Es können dann sogenannte SHACL-Shapes definiert werden, welche auf dem Graphen getestet werden. Mithilfe eines Workflow-Orchestration-Frameworks könnte SHACL so integriert werden, dass geänderte RDF-Daten automatisch validiert werden. Wie bei Unit-Tests in Software-Projekten kann so bei Änderungen automatisch sichergestellt werden, dass das Ergebnis noch den Erwartungen entspricht.

Nach unserer Einschätzung würde die Implementierung eines solchen Validierungsschritts ungefähr eine Woche in Anspruch nehmen. Für das Schreiben der nötigen SHACL-Shapes würden je nach Komplexität der Tests ein bis zwei Wochen benötigt werden.

9.5 Erweiterung der Datengrundlage

Im Forstgraphen sind verschiedene Datensätze integriert. Jedoch gibt es bereits weitere Daten, die uns zur Verfügung stehen. Dabei handelt es sich zum Teil um Datenbankdumps mit mehreren hundert Tabellen.

Bevor wir weitere Daten in die Forst-Pipeline einfügen, werden wir die möglichen Verbesserungsvorschläge aus den vorherigen Abschnitten evaluieren. Entsprechend werden wir die gewählten Verbesserungen implementieren und erst dann weitere Daten hinzufügen. So können wir einen möglichen Migrationsaufwand gering halten.

Wir können für das Integrieren neuer Datensätze keinen zeitlichen Rahmen festlegen, da dies von der Komplexität der jeweiligen Daten abhängt.

Danksagung

Ich bedanke mich an dieser Stelle bei allen, die mich bei dieser Arbeit unterstützt haben. Mein besonderer Dank gilt meiner Betreuerin Prof. Dr. Hannah Bast, die mich mit ihrem breiten Wissen und ihren Ideen stetig unterstützt hat. Des Weiteren danke ich Thomas Breunig und PD Dr. Tobias Schlechtriemen, die den Rahmen des WWW4.0-Projekts ermöglicht haben, in dem diese Arbeit entstanden ist. Zusätzlich danke ich Ioannis Nezis, Tessa Fischer und meinen Eltern für das Korrekturlesen meiner Arbeit.

Bibliografie

[1] „Waldzustandsbericht 2025“, Forstliche Versuchs- und Forschungsanstalt Baden-Württemberg (FVA), 2025. [Online]. Verfügbar unter: https://www.fva-bw.de/fileadmin/publikationen/wzb/WZB2025.pdf

[2] R. Cyganiak, D. Wood, und M. Lanthaler, „RDF 1.1 Concepts and Abstract Syntax“, W3C, Feb. 2014. [Online]. Verfügbar unter: https://www.w3.org/TR/rdf11-concepts/

[3] A. Hogan *u. a.*, „Knowledge Graphs“, Juni 2021. doi: 10.1145/3447772.

[4] „WWW4.0 – Wissen über den Wald im Wandel 4.0“. Zugegriffen: 24. August 2026. [Online]. Verfügbar unter: https://uni-freiburg.de/www40/

[5] R. Lott, „OGC Abstract Specification Topic 2: Referencing by coordinates“, Open Geospatial Consortium, OGC 18-005r5, Juli 2021. [Online]. Verfügbar unter: https://docs.ogc.org/as/18-005r5/18-005r5.html

[6] „Department of Defense World Geodetic System 1984, Its Definition and Relationships with Local Geodetic Systems“, National Geospatial-Intelligence Agency, NGA.STND.0036_1.0.0_WGS84, Juli 2014. [Online]. Verfügbar unter: https://earth-info.nga.mil/php/download.php?file=coord-wgs84

[7] „ESRI Shapefile Technical Description“, Esri, Juli 1998. [Online]. Verfügbar unter: https://www.esri.com/content/dam/esrisites/sitecore-archive/Files/Pdfs/library/whitepapers/pdfs/shapefile.pdf

[8] J. Yutzler, „OGC GeoPackage Encoding Standard“, Open Geospatial Consortium, OGC 12-128r17, Feb. 2021. [Online]. Verfügbar unter: https://docs.ogc.org/is/12-128r17/12-128r17.html

[9] „TIFF, Revision 6.0“, Adobe Developers Association, Juni 1992. [Online]. Verfügbar unter: https://web.archive.org/web/20180810205359/https://www.adobe.io/content/udp/en/open/standards/TIFF/_jcr_content/contentbody/download/file.res/TIFF6.pdf

[10] E. Devys, T. Habermann, C. Heazel, R. Lott, und E. Rouault, „OGC GeoTIFF Standard“, Open Geospatial Consortium, OGC 19-008r4, Sep. 2019. [Online]. Verfügbar unter: https://docs.ogc.org/is/19-008r4/19-008r4.html

[11] J. Maso, „OGC Cloud Optimized GeoTIFF Standard“, Open Geospatial Consortium, OGC 21–26, Juli 2023. [Online]. Verfügbar unter: https://docs.ogc.org/is/21-026/21-026.html

[12] C. Bizer, T. Heath, und T. Berners-Lee, „Linked Data – The Story So Far“, 2009. doi: 10.4018/jswis.2009081901.

[13] M. Dürst und M. Suignard, „Internationalized Resource Identifiers (IRIs)“, IETF, RFC 3987, Jan. 2005. [Online]. Verfügbar unter: https://www.rfc-editor.org/rfc/rfc3987

[14] G. Carothers und A. Seaborne, „RDF 1.1 N-Triples: A line-based syntax for an RDF graph“, W3C, Feb. 2014. [Online]. Verfügbar unter: https://www.w3.org/TR/n-triples/

[15] E. Prud'hommeaux und G. Carothers, „RDF 1.1 Turtle: Terse RDF Triple Language“, W3C, Feb. 2014. [Online]. Verfügbar unter: https://www.w3.org/TR/turtle/

[16] D. Brickley und R. V. Guha, „RDF Schema 1.1“, W3C, Feb. 2014. [Online]. Verfügbar unter: https://www.w3.org/TR/rdf-schema/

[17] N. Car und T. Homburg, „OGC GeoSPARQL – A Geographic Query Language for RDF Data“, Open Geospatial Consortium, OGC 22-047r1, Sep. 2022. [Online]. Verfügbar unter: https://docs.ogc.org/is/22-047r1/22-047r1.html

[18] J. R. Herring, „OGC Simple Feature Access – Part 1: Common Architecture“, Open Geospatial Consortium, OGC 06-103r4, Mai 2011. [Online]. Verfügbar unter: https://www.ogc.org/standards/sfa/

[19] S. Harris und A. Seaborne, „SPARQL 1.1 Query Language“, W3C, März 2013. [Online]. Verfügbar unter: https://www.w3.org/TR/sparql11-query/

[20] „QGIS Geographic Information System“. Zugegriffen: 24. August 2026. [Online]. Verfügbar unter: https://www.qgis.org/

[21] H. Bast und B. Buchhold, „QLever: A Query Engine for Efficient SPARQL+Text Search“, in *Proceedings of the 26th ACM International Conference on Information and Knowledge Management (CIKM '17)*, ACM, 2017, S. 647–656. doi: 10.1145/3132847.3132921.

[22] „QLever UI“. Zugegriffen: 24. August 2026. [Online]. Verfügbar unter: https://github.com/ad-freiburg/qlever-ui

[23] „QLever Petrimaps“. Zugegriffen: 24. August 2026. [Online]. Verfügbar unter: https://github.com/ad-freiburg/qlever-petrimaps

[24] „Leaflet“. Zugegriffen: 24. August 2026. [Online]. Verfügbar unter: https://leafletjs.com/

[25] „GDAL/OGR Geospatial Data Abstraction Software Library“. Zugegriffen: 24. August 2026. [Online]. Verfügbar unter: https://gdal.org/

[26] „RDFLib“. Zugegriffen: 24. August 2026. [Online]. Verfügbar unter: https://github.com/RDFLib/rdflib

[27] S. Das, S. Sundara, und R. Cyganiak, „R2RML: RDB to RDF Mapping Language“, W3C, Sep. 2012. [Online]. Verfügbar unter: https://www.w3.org/TR/r2rml/

[28] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, und R. Van de Walle, „RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data“, in *Proceedings of the 7th Workshop on Linked Data on the Web (LDOW 2014), co-located with WWW 2014*, CEUR-WS.org, 2014. [Online]. Verfügbar unter: https://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf

[29] J. Arenas-Guerrero, D. Chaves-Fraga, J. Toledo, M. S. Pérez, und O. Corcho, „Morph-KGC: Scalable knowledge graph materialization with mapping partitions“, 2024. doi: 10.3233/SW-223135.

[30] P. Heyvaert, B. De Meester, A. Dimou, und R. Verborgh, „Declarative Rules for Linked Data Generation at your Fingertips!“, in *Proceedings of the 15th Extended Semantic Web Conference (ESWC 2018): Posters and Demonstrations*, Springer, 2018. doi: 10.1007/978-3-319-98192-5_40.

[31] „TARQL: SPARQL for Tables“. Zugegriffen: 24. August 2026. [Online]. Verfügbar unter: https://github.com/tarql/tarql

[32] E. Daga, L. Asprino, P. Mulholland, und A. Gangemi, „Facade-X: An Opinionated Approach to SPARQL Anything“, 2021. doi: 10.3233/ssw210035.

[33] „QLever performance evaluation and comparison to other SPARQL engines“. Zugegriffen: 4. September 2026. [Online]. Verfügbar unter: https://github.com/

ad-freiburg/qlever/wiki/QLever-performance-evaluation-and-comparison-to-other-SPARQL-engines

[34] „RDF Graph Database Performance Evaluation“. Zugegriffen: 4. September 2026. [Online]. Verfügbar unter: https://qlever.dev/evaluation/www/

[35] „Scaling Wikidata/Benchmarking: Final Report“. Zugegriffen: 4. September 2026. [Online]. Verfügbar unter: https://www.wikidata.org/wiki/Wikidata:Scaling_Wikidata/Benchmarking/Final_Report

[36] „Docker“. Zugegriffen: 24. August 2026. [Online]. Verfügbar unter: https://www.docker.com/

[37] „Docker Compose“. Zugegriffen: 24. August 2026. [Online]. Verfügbar unter: https://docs.docker.com/compose/

[38] „LibreOffice“. Zugegriffen: 24. August 2026. [Online]. Verfügbar unter: https://www.libreoffice.org/

[39] „WET 2024 – Waldentwicklungstypen im Klimawandel“, ForstBW, 2024. [Online]. Verfügbar unter: https://www.forstbw.de/fileadmin/user_upload/WET_2024_Richtlinie.pdf

[40] „Claude Code“. Zugegriffen: 24. August 2026. [Online]. Verfügbar unter: https://code.claude.com/docs/en/overview

[41] „Sigma.js“. Zugegriffen: 24. August 2026. [Online]. Verfügbar unter: https://www.sigmajs.org/

[42] „OpenStreetMap“. Zugegriffen: 24. August 2026. [Online]. Verfügbar unter: https://www.openstreetmap.org/

[43] R. L. Graham, „Bounds for Certain Multiprocessing Anomalies“, November 1966. doi: 10.1002/j.1538-7305.1966.tb01709.x.

[44] G. M. Amdahl, „Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities“, in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference (AFIPS '67)*, ACM, Apr. 1967, S. 483–485. doi: 10.1145/1465482.1465560.

[45] „Dagster“. Zugegriffen: 24. August 2026. [Online]. Verfügbar unter: https://dagster.io/

[46] „Prefect“. Zugegriffen: 24. August 2026. [Online]. Verfügbar unter: https://www.prefect.io/

[47] „MinIO“. Zugegriffen: 24. August 2026. [Online]. Verfügbar unter: https://min.io/

[48] H. Knublauch und D. Kontokostas, „Shapes Constraint Language (SHACL)“, W3C, Juli 2017. [Online]. Verfügbar unter: https://www.w3.org/TR/shacl/

Abbildungsverzeichnis

Abbildung 1 Vektordaten 3
Abbildung 2 Rasterdaten 3
Abbildung 3 Beispielhafter Wissensgraph 4
Abbildung 4 Darstellung eines Triples 5
Abbildung 5 Qonstructor-Pipeline-Architektur 19
Abbildung 6 Ontologie im Überblick 46
Abbildung 7 Ontologie-Browser 47
Abbildung 8 Ontologie-Graph 48
Abbildung 9 Demo-App 49
Abbildung 10 Demo-App Zustand 50
Abbildung 11 Demo-App Vorschlag 50
Abbildung 12 Potenziell gefährdete Eschen in der Nähe von öffentlichen Straßen 53
Abbildung 13 Laufzeit von CSV2RDF 59
Abbildung 14 Laufzeit der Generic-Generation-Stage pro Task 60
Abbildung 15 Laufzeit der Construct-Stage pro Task 60
Abbildung 16 Laufzeit der Generic-Generation-Stage mit verschiedenen Workern . . . 61
Abbildung 17 Laufzeit der Construct-Stage mit verschiedenen Workern 61
Abbildung 18 Laufzeit aller einzelnen Stages für $W = 1$ und $W = 16$ 62
Abbildung 19 Gesamtlaufzeit der Pipeline mit verschiedenen Workern 63

Tabellenverzeichnis

Tabelle 1 Beispielhaftes Ergebnis der Select-Query 8
Tabelle 2 Source-Dateien der Forst-Pipeline nach Preprocessing 32
Tabelle 3 Stage-Konfigurationsdateien der Forst-Pipeline 33
Tabelle 4 Spezifikationen der Testmaschine 58
Tabelle 5 Laufzeit von CSV2RDF nach Zeilenanzahl 58

Codeblockverzeichnis

Codeblock 1 Beispiel-IRIs ... 5
Codeblock 2 Beispiel-Literale ... 6
Codeblock 3 Beispieldokument im N-Triples-Format ... 6
Codeblock 4 Beispieldokument im Turtle-Format ... 6
Codeblock 5 Modellierungsbeispiel mit RDFS ... 7
Codeblock 6 Modellierungsbeispiel mit GeoSPARQL ... 7
Codeblock 7 Beispielhafte SPARQL-SELECT-Query ... 8
Codeblock 8 Beispielhafte SPARQL-Construct-Query ... 9
Codeblock 9 Beispielhaftes Ergebnis der Construct-Query ... 9
Codeblock 10 Auswahl eines Generators über den Source-Type ... 20
Codeblock 11 Default-Generatoren der Pipeline-Konfigurationsdatei ... 21
Codeblock 12 Beispiel für die Mapping-Grundstruktur ... 22
Codeblock 13 Beispielhafte CSV-Eingabedatei ... 23
Codeblock 14 Aus CSV generiertes RDF im Turtle-Format ... 23
Codeblock 15 Beispielhafte JSON-Eingabedatei ... 23
Codeblock 16 Aus JSON generiertes RDF im Turtle-Format ... 24
Codeblock 17 Aus einer TIFF-Datei generiertes RDF ... 24
Codeblock 18 Beispielhafte Queries in einer Stage-Konfigurationsdatei ... 25
Codeblock 19 Default-Export-Query ... 26
Codeblock 20 QLever-Konfiguration in der Pipeline-Konfigurationsdatei ... 26
Codeblock 21 Beispiel-Triples zu den Modellierungsgrundsätzen ... 29
Codeblock 22 Umwandlung von Shapefiles zu CSV mit ogr2ogr ... 31
Codeblock 23 Umwandlung von GeoPackages zu CSV mit ogr2ogr ... 31
Codeblock 24 Umwandlung von TIFF-Dateien zu COG mit gdalwarp ... 31
Codeblock 25 Source-Definitionen in Schutzgebiete.yaml ... 34
Codeblock 26 Construct-Query zur Normalisierung einer CSV-Datei ... 35
Codeblock 27 Beispiele für Minting via Concat ... 36
Codeblock 28 Beispiele für Minting via SHA ... 36
Codeblock 29 Beispiel für Linking während der Normalisierung ... 37
Codeblock 30 Beispiele für Typecasts ... 38
Codeblock 31 Beispiel für ein optionales Triple-Pattern ... 38
Codeblock 32 Beispiel zum Filtern ungültiger Werte ... 38
Codeblock 33 Construct-Query „RegionaleEinheit“ ... 39
Codeblock 34 RDF-Auszug der Query „RegionaleEinheit“ ... 40

Codeblock 35 Unpivot-Query „Baumartenpfleglichkeit“ 42
Codeblock 36 RDF-Auszug der Unpivot-Query „Baumartenpfleglichkeit“ 43
Codeblock 37 Linking-Query von Standort und Baumartenpfleglichkeit 43
Codeblock 38 Inference-Query zur Typisierung von Geometrien 44
Codeblock 39 Inference-Query zur Aufteilung des Bestandesalters 44
Codeblock 40 SPARQL-Query zur Ermittlung potenziell gefährdeter Eschen 51