Freie Universität Berlin

Master-Arbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Verwendung von URIs zur effizienten Einzel- und Massenabfrage von Werten aus offenen Datensätzen

Christoph Schröder

Matrikelnummer: 4594383

christoph.schroeder@fu-berlin.de

Betreuer: Prof. Dr.-Ing. Edzard Höfig

Erstgutachter: Prof. Dr.-Ing. Ina Schieferdecker (Freie Universität Berlin)

Zweitgutachter: Prof. Dr.-Ing. Edzard Höfig (Beuth Hochschule für Technik Berlin)

Berlin, 18.09.2014

Kurzfassung

Ziel dieser Masterarbeit ist es unter Verwendung von URIs effiziente Einzel- und Massenabfragen von Werten aus offenen Datensätzen zu realisieren. Dabei geht es in erster Linie um die Erstellung sogenannter „Deep-Links“ mit denen gezielt Informationen aus einem Datensatz gefiltert werden können. Zu diesem Zweck wird die Struktur verschiedener Datenformate analysiert, mit dem Ziel geeignete Informationen für die Adressierung der Daten zu finden. Diese werden für die Erstellung verschiedener URI-Muster für die Deep-Links genutzt. Die Adressierung durch Deep-Links und daraus resultierende Verfahren für Zugriffe auf die Daten werden bezüglich ihrer Performanz untersucht. Die Ergebnisse der Analyse werden in der Implementierung eines Prototypen umgesetzt. Dabei werden spezifische Anforderungen an Aspekte der Flexibilität bezüglich Schnittstelle und Erweiterbarkeit berücksichtigt.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, 17. September 2014

. . . . . .
Christoph Schröder

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich bei der Anfertigung dieser Masterarbeit unterstützt haben.

Mein Dank gilt der Arbeitsgruppe Software Engineering der Freien Universität Berlin für die konstruktiven Gespräche im Rahmen der Treffen der Arbeitsgruppe und Prof. Dr.-Ing. Ina Schieferdecker für die Bereitschaft als erste Gutachterin dieser Arbeit zu fungieren.

Besonders möchte ich mich bei Prof. Dr.-Ing. Edzard Höfig bedanken für die Bereitstellung des Themas und insbesondere für die Betreuung der Arbeit auch nach der Beendigung seiner Tätigkeit an der FU-Berlin.

Weiterer Dank gilt meinen Arbeitskollegen des Helmholtz Zentrum Berlin und meiner Familie für die Rücksichtnahme und Unterstützung insbesondere in den letzten Monaten der Bearbeitungszeit.

Inhaltsverzeichnis

1 Einleitung 1
1.1 Motivation . . . 1
1.2 Problemstellung . . . 2
1.3 Verwandte Themen . . . 2
1.4 Herangehensweise . . . 3

2 Begriffsklärung 5
2.1 Representational state transfer (REST) . . . 5
2.2 Uniform Resource Identifier (URI) . . . 7
2.2.1 Begriff . . . 7
2.2.2 Syntaktischer Aufbau . . . 7
2.3 Deep-Linking . . . 8

3 Analyse geläufiger Datenformate offener Datensätze 9
3.1 Auswahl repräsentativer Datenformate . . . 9
3.2 Extensible Markup Language (XML) . . . 10
3.3 Hypertext Markup Language (HTML) . . . 11
3.4 Comma Separated Values (CSV) . . . 11
3.5 Microsoft Office . . . 12
3.5.1 Paketstruktur . . . 13
3.5.2 WordprocessingML . . . 15
3.5.3 SpreadsheetML . . . 17
3.6 Portable Document Format . . . 20
3.6.1 Basis-Objekte . . . 21
3.6.2 Dateistruktur . . . 22
3.6.3 Dokumentstruktur . . . 24
3.6.4 Text . . . 25
3.6.5 Lesezeichen . . . 27

4 Verlinken der Datenformate durch URIs 30
4.1 Allgemeine Anforderungen an das Design . . . 30
4.2 Allgemeines Muster . . . 32
4.3 Extensible Markup Language (XML/HTML) . . . 34
4.4 Comma Separated Values (CSV) . . . 36
4.5 Microsoft Office . . . 38

4.5.1 WordprocessingML . . . 39
4.5.2 SpreadsheetML . . . 41
4.6 Portable Document Format (PDF) . . . 42

5 Verarbeitung XML-basierter Formate 46
5.1 Simple API for XML (SAX) . . . 47
5.2 Streaming API for XML (StAX) . . . 49
5.3 Document Object Model (DOM) . . . 50
5.4 Nicht-extraktives Parsing mit VTD-XML . . . 51
5.4.1 Virtual Token Descriptor (VTD) . . . 52
5.4.2 Navigation . . . 53
5.4.3 Eigenschaften . . . 54
5.5 Evaluation der XML-Parser . . . 56
5.5.1 Eigenschaften . . . 56
5.5.2 XPath/Parser-Benchmark . . . 60
5.5.2.1 Testkandidaten . . . 60
5.5.2.2 Testvorbereitung . . . 61
5.5.2.3 Durchführung . . . 63
5.5.2.4 Ergebnisse . . . 63
5.5.3 Schlussfolgerung . . . 65

6 Design und Implementierung eines Prototypen 67
6.1 Verhaltensmodellierung . . . 67
6.2 Strukturmodellierung . . . 68
6.3 Eingesetzte Frameworks und APIs . . . 71
6.4 Caching . . . 73
6.4.1 Client-Cache . . . 73
6.4.2 Server-Cache . . . 74
6.5 Vergleich der Implementierungen durch Benchmarks . . . 76
6.5.1 Testvorbereitung . . . 76
6.5.2 Durchführung . . . 77
6.5.3 Ergebnisse . . . 78

7 Schlussfolgerung 79

8 Ausblick 81

A Anhang Bilder und Tabellen 83

B Anhang UML-Diagramme 92

Literaturverzeichnis 96

1. Einleitung

1.1. Motivation

In den letzten Jahren konnte eine rasante Entwicklung der Anzahl vernetzter Geräte beobachtet werden. Das Internet wurde auf breiter Ebene verfügbar und ist heute fester Bestandteil des Alltags. Die steigende Zahl der Nutzer löste eine Entwicklung vom reinen Medium für Informationskonsum hin zu einem von Nutzer mitgestalteten interaktiven Medium aus. Ende 2004 wurde dafür der Begriff „Web 2.0“von Tim O'Reilly geprägt. Die gestiegene Zahl der vernetzten Geräte erschafft nun völlig neue Möglichkeiten zur Nutzung von Daten. Alltagsgegenstände werden durch eingebettete Systeme selbst zu intelligenten Geräten. Ausgestattet mit Sensoren und Aktoren sollen sie durch Vernetzung mit anderen Geräten den Menschen aktiv unterstützen. Die angestrebte Zukunftsvision dieser Entwicklung ist das sogenannte Internet der Dinge. Schon heute können zahlreiche Entwicklungen von Applikationen insbesondere für mobile Geräte wie Smartphones beobachtet werden, welche es sich zur Aufgabe gemacht haben online verfügbare Daten aufzubereiten und dem Nutzer in einer brauchbaren Form anzubieten. Eine grundlegende Voraussetzung dafür ist die Verfügbarkeit offener Daten. Diese werden in wachsender Zahl von der „Open Data“-Bewegung in verschiedenen Formaten zur Verfügung gestellt. Häufig fehlt es jedoch an einer adäquaten Form des Zugriffs und der Möglichkeit relevante Daten gezielt herauszufiltern.
Diese Arbeit befasst sich mit dem Verlinken von Daten basierend auf verschiedenen Formaten und Systemen mittels URI[14] und ist der für Datenzugriffe und Identifikation von Daten zuständige Teil, der im Rahmen der Forschung an „Open Data“-Portalen entwickelten Datenplattform „PANDA“werden. Der Begriff PANDA wird in dieser Arbeit an verschiedenen Stellen synonym für das entwickelte System verwendet.
Nach dem Prinzip des „Deep-Linking“ sollen Abfragen von einzelnen Werten und auch Massenabfragen auf spezifische Teile der Daten eines Datensatzes ermöglicht werden. Im Fokus stehen dabei die Schonung der Ressourcen, die Performanz der Zugriffe und eine hohe Flexibilität. In Richtung der Datenschicht soll es möglich sein weitere Datenformate mit der gleichen Methodik auf Basis der durchgeführten Analyse hinzuzufügen und das System intern zu erweitern. In Richtung der Nutzer soll eine Erweiterung der bereits genutzten Repräsentationsformate möglich sein, um die Daten in passender Form einer möglichst großen Vielfalt an Nutzern und Systemen zur Verfügung zu stellen.
Zielsetzung ist es durch den Einsatz entsprechender Technologien eine Abstraktionsschicht zwischen den Datensätzen und den Nutzern zu schaffen, welche in geeigneter Form performante Einzel- und Massenabfragen ermöglicht und aus Nutzersicht möglichst ohne die Notwendigkeit zusätzlicher Software verfügbar ist. Die adressierten Daten ließen sich so universell und gezielt nutzen. Die Anwendungsgebiete erstrecken sich dabei

von gezielten Abfragen von Daten z.B. durch Smartphone-Apps bis hin zur Möglichkeit Daten gezielt in eine andere Form zu konvertieren, ohne dabei Strukturinformationen zu verlieren.
Diese Arbeit soll exemplarisch das Vorgehen von der Analyse der Datenstruktur bis hin zum Verlinken und der Implementierung eines entsprechenden Mechanismus zur Datenextraktion zeigen und dabei auf Probleme und Aspekte der Performanz bezüglich Extraktion der Daten eingehen. Dazu sollen verschiedene repräsentative Datenformaten, wie sie typischerweise bei offenen Daten zum Einsatz kommen, genutzt werden. Verschiedene Methoden für die Verarbeitung sollen mittels Benchmarks miteinander verglichen werden um aussagekräftige Ergebnisse zu erhalten.

1.2. Problemstellung

Der Kern des Problems ist die Frage, in welcher Weise sich ein einzelnes Datum oder bestimmte Teile eines Datensatzes durch URIs adressieren lassen und wie über diese URIs performante Zugriffe auf diese Daten ermöglicht werden können. Zur Identifikation von Einzelwerten oder spezifischen Teilen aus einem Datensatz werden eindeutige Informationen benötigt, welche ggf. Teil der URI verwendet werden können. Es muss abhängig vom Datenformat oder Speichersystem festgestellt werden, ob solche Informationen existieren und unter welchen Voraussetzungen sie für die Adressierung verwendet werden können. In die URI kodierte Informationen müssen in eine der URI Syntax[14] entsprechenden Form gebracht werden. Unter Beachtung weiterer Aspekte wie der Sicherheit des Systems müssen ggf. Teile dieser Informationen ausgelagert werden. Dies muss in einer Form geschehen, die möglichst keine negativen Auswirkungen auf die Gesamtperformanz hat. Die Adressierung der Daten sollte zudem eine sinnvolle Granularität aufweisen, um Zugriffe nicht unnötig kompliziert zu gestalten.
Der Performanz Aspekt muss sowohl bei der Adressierung als auch bei der Umsetzung der Zugriffe für Einzel- und Massenabfragen berücksichtigt werden. Letzteres bietet das meiste Potential die Performanz zu steigern. Es werden Mechanismen benötigt die einen möglichst schnellen Zugriff gewähren. Diese müssen gefunden und ausreichend untersucht werden.
Desweiteren wird die Frage gestellt, wie dieses System möglichst flexibel gestaltet werden kann. Da sich Datenformate und Systeme oft gravierend unterscheiden, müssen Pattern und Methoden gefunden werden die eine Erweiterung der Datenschicht um weitere Datenformate ermöglichen. Das System soll außerdem auch nach außen in Bezug auf die Repräsentation der Daten möglichst flexibel und erweiterbar sein.

1.3. Verwandte Themen

Als Teil des semantischen Webs gibt es zahlreiche Arbeiten zum Thema „Linked Open Data“(LOD), welche sich mit dem Verlinken von Daten aus verschiedenen Quellen wie SQL Datenbanken oder Excel/CSV Dateien befassen. Einen Überblick über diese Arbeiten bietet [31]. Aufgrund vieler individueller Ansätze zu diesem Thema wurde mit

R2RML[11] sogar eine einheitliche Sprache für das Mapping vom „Resource Description Framework“(RDF)[12] zu relationalen Datenbanken entwickelt, wobei speziell auf die Charakteristiken von RDF eingegangen wurde. Ziel dieser Arbeiten ist es jedoch nicht nur die Daten online verfügbar zu machen, sondern sie in einem semantischen Kontext mit anderen Daten einzubetten. Dies kann zwar neue Aspekte bieten, vernachlässigt jedoch den Aspekt der Performanz und ist zudem unflexibel. Der von diesen Arbeiten gezeigte Vorgang zur Umsetzung kann zudem sehr komplex werden und ist häufig nur semi-automatisch. Dies stellt insbesondere in Bezug auf die Erweiterbarkeit um neue Datenformate ein Problem dar. Unser Ansatz entspricht in diesem Punkt eher der in [28] beschriebenen „Raw Conversion“, bei der auf die Anreicherung der Daten mit Kontext-Informationen für die Einbettung in einen semantischen Kontext verzichtet wurde. Im Unterschied dazu legen wir uns jedoch nicht auf ein spezielles Repräsentations-Format (RDF) fest.
Das in diesen Arbeiten beschriebene Mapping von URIs auf die Daten kann auf zwei verschiedene Arten umgesetzt sein: als Echtzeit-Mapping oder als einmalige Konvertierung und Speicherung in einem RDF-Tripplestore. Ein Echtzeit-Mapping kann je nach Umsetzung Einbußen in der Performanz der Zugriffe nach sich ziehen. Die Konvertierung dagegen ermöglicht zwar direktere Zugriffe, bedeutet jedoch automatisch eine Redundanz der Daten und somit zusätzliche Kosten in Bezug auf Speicherplatz. In unserem Fall streben wir einen möglichst direkten Zugriff in Echtzeit an. Da wir auf die Einbettung in einen semantischen Kontext verzichten und somit auch die Anreicherung der Daten mit zusätzlichen Informationen entfällt, ist eine solche Vorgehensweise prinzipiell performanter als bei den erwähnten Ansätzen. Der Bezug zu vorhandenen Metadaten soll in unserem Ansatz an anderer Stelle durch die eindeutige Identifikation mittels URI hergestellt werden. Eine Konvertierung wird in unserem Ansatz jedoch nicht ausgeschlossen. Die Adressierbarkeit der Daten mittels Deep-Links erlaubt es sogar Daten ohne Verlust von Strukturinformationen in eine andere Form zu bringen.

1.4. Herangehensweise

Da wir mit einer Vielzahl Datenformaten mit Option zur Erweiterung arbeiten wollen und sich diese oft grundlegend unterscheiden, lässt sich keine allgemeingültige Form für die Adressierung finden. Daher ist es notwendig zunächst eine repräsentative Auswahl an Datenformaten zu treffen. Diese werden dann auf ihre Struktur hin analysiert. Es werden Informationen gesucht, die für die Adressierung von Daten verwendet werden können. Dabei wird auf Aspekte des Aufwands für Extraktion anhang dieser Informationen und eventuell auftretende Probleme eingegangen.
Im nächsten Schritt werden exemplarisch konkrete URI-Muster erstellt. In diesem Zusammenhang werden die Informationen für die Adressierung auf ihre Tauglichkeit für die Verwendung in URIs hin geprüft. Dies betrifft insbesondere die Einhaltung der von RFC 3986 vorgegebenen Syntax. Ggf. ist die Verwendung bestimmter Informationen in URIs nicht möglich oder erwünscht, weshalb diese auf andere Weise abgerufen werden müssen. Dazu bedarf es einer konkreten Umsetzung, welche möglichst für alle Datenfor-

mate einheitlich sein sollte. Die genutzten Werkzeuge und deren Möglichkeiten müssen bei der Erstellung der URI-Pattern berücksichtigt werden.
Zur Steigerung der Performanz müssen insbesondere in Bezug auf Massenabfragen Mechanismen gefunden werden welche den Zugriff beschleunigen. Ein möglicher Ansatz ist das „Caching“von Ergebnissen bereits erfolgter Abfragen. In Folge dessen werden mögliche Implementierungen und Umsetzungen von Caching-Mechanismen gesucht.
Den Abschluss bildet die exemplarische Implementation eines Prototypen. Für eine größtmögliche Flexibilität sollte diese eine „Representational State Transfer“(REST) Schnittstelle verfügen, welche eine flexible Erweiterung des Systems erlaubt.

2. Begriffsklärung

In diesem Kapitel werden Begriffe erläutert, die eine wichtige Rolle in der Arbeit spielen und mehrfach verwendet wurden.

2.1. Representational state transfer (REST)

Der „Representational state transfer“ (REST)[25] ist eine im Jahr 2000 von Roy Fielding eingeführte und beschriebene Sammlung von Anforderungen an Hypermedia-Systeme, im speziellen an das „World Wide Web (WWW)“. Der Fokus liegt auf der Kommunikation von Maschine zu Maschine mit dem Ziel Eigenschaften wie Performance, Skallierbarkeit und Komplexität zu verbessern.
Die Anforderungen werden auf die Elemente aus denen die Basis der Architektur des WWW besteht angewandt. Bei diesen Elementen handelt es sich um Komponenten, Konnektoren und Daten. Dabei wird nicht auf deren Implementation eingegangen, sondern ausschließlich auf ihre Rolle innerhalb des gesamten Systems. Die von REST vorgegebenen Anforderungen sind:
(1) Client/Server-Architektur. (2) Zustandslose Kommunikation. (3) Einschränkungenen für das Caching. (4) Ebenen-System (5) Unterstützung von Code-on-demand (optional) (6) Einheitliche Interfaces für Kommunikation.
Die zustandslose Kommunikation ist einer der Kernpunkte, der REST von anderen Architekturen unterscheidet. REST setzt bei Anfragen voraus, dass alle für die Verarbeitung benötigten Informationen in der Nachricht für die Anfrage enthalten sind. Dies erlaubt eine einfachere Verarbeitung der Anfragen, da der Server sich nicht für Informationen außerhalb der Anfrage interessieren muss. In den meisten Fällen werden dadurch Hardware-Ressourcen geschont, da das Speichern der Zustände entfällt. In bestimmten Situationen kann dies jedoch auch zur Übertragung redundanter Daten führen. Ein besonderer Vorteil dieser Art der Kommunikation ist die einfache Möglichkeit der Parallelisierung.
Für das Caching wurde festgelegt, dass auf implizite oder explizite Weise festgelegt werden muss, welche Daten gecached werden können und welche nicht. Für die Kontrolle des Cachings auf Client-Seite werden Metadaten, welche zusammen mit der Antwort gesendet werden, genutzt. Diese Möglichkeit verhindert Probleme, welche auftreten können wenn das Ergebnis aus dem Client-Cache nicht mit dem Ergebnis einer erneuten Anfrage übereinstimmt.
REST erlaubt es Systeme in mehrere Ebenen aufzuteilen. Dabei kann eine Ebene nicht über die Grenzen der nächsten hinwegschauen. Ein Beispiel dafür wäre ein zwischengeschalteter Server der z.B. zur Authentifizierung dienen könnte. Der Client weiß in diesem

Fall nicht, dass er mit einem zwischengeschalteten Server verbunden ist. Eine andere Anwendungsmöglichkeit wäre die Lastverteilung zwischen mehreren End-Servern.
Die Unterstützung von Code-on-demand ist optional um Probleme mit Systemen die solche Arbeitsweise z.B. wegen Firewalls nicht zulassen, zu vermeiden. Es beschreibt die Nutzung von nachgeladenen Skripten und Applets auf dem Client, welche dessen Funktionalität erweitern sollen.
Der wichtigste Punkt der REST von anderen Architekturen unterscheiden soll, ist die Nutzung einheitlicher Interfaces. REST verwendet dazu ausschließlich bereits vorhandene Techniken des „Hypertext Transfer Protocol (HTTP)“[8], da die Implementation des Protokolls in allen Web-Clients und Servern vorhanden ist. Clients benötigen daher keine speziellen Technologien oder zusätzliches Wissen über Schnittstellen, um mit einem RESTful-Server zu kommunizieren. Um die Komponenten besser voneinander zu trennen müssen jedoch noch weitere Bedingungen erfüllt sein:

(1) Identifikation von Ressourcen.

(2) Repräsentationen enthalten alle notwendigen Informationen um eine Ressource zu manipulieren.

(3) Nachrichten sind selbstbeschreibend und enthalten alle Informationen, die zur Verarbeitung einer Ressource notwendig sind.

Die Erfüllung dieser Komponenten trennt die Komponenten weitestgehend voneinander und erlaubt eine unabhängige Entwicklung. In diesem Zusammenhang wurden zwei für REST wichtige Begriffe genannt: „Ressource“ und „Repräsentation“. Eine Ressource wird als Information jeglicher Art beschrieben, welche mit einem Namen identifiziert werden kann. Dies könnten beispielsweise ein Dokument, ein Service, eine Sammlung anderer Ressourcen aber auch reale Objekte wie eine Person sein. Ihre Identifikation soll zudem abhängig von der Zeit t sein, sodass eine Ressource R als Mapping Funktion $M_R(t)$ zu verstehen ist. Diese mapped auf eine Repräsentation und/oder auf den Identifikator der Ressource. Folglich gibt es statische und dynamische Ressourcen, welche ihre Repräsentation über die Zeit ändern oder eben nicht. Fielding legt daher besonderen Wert auf die eindeutige Unterscheidung der Ressourcen.
Eine Repräsentation ist dagegen die Darstellung einer Ressource die zum anfragenden Client geschickt wird. Sie besitzt dabei immer einen bestimmten Media Type (MIME) und besteht aus Daten + Metadaten. Clients können folglich auch verschiedene Repräsentationen der selben Ressource anfragen, wobei deren Anzahl theoretisch unbegrenzt ist.
Diese abstrakte Definition von Ressourcen ermöglicht zwei Schlüssel-Funktionsmerkmale: Zum einen ist die Definition sehr allgemein gehalten und lässt sich daher auf beliebige Quellen von Informationen unabhängig von ihrem Typ anwenden. Zum anderen erlaubt sie die späte Bindung von Referenzen auf eine bestimmte Repräsentation. Dadurch ist eine generische Schnittstelle zwischen Client und Server gegeben, bei der sich beide durch „Content Negotiation“ auf eine bestimmte Repräsentation einigen.
Zusammenfassend kann gesagt werden, dass es sich bei REST nicht um eine neue Technik handelt, sondern um Regeln die eine konsequente und sinnvolle Nutzung bereits

vorhandener Techniken fordert mit dem Ziel Webanwendungen schneller, stabiler und besser skallierbarer zu machen.

2.2. Uniform Resource Identifier (URI)

2.2.1. Begriff

Die „Uniform Resource Identifier“ (URI) sind wie der Name impliziert eine uniforme Methode um Ressourcen zu identifizieren. Der Begriff „URI“ setzt sich aus drei Eigenschaften zusamnen die ihn beschreiben. „Uniformität“ erlaubt den Einsatz verschiedener Ressourcen im selben Kontext dank einheitlicher semantischer Interpretation. Erreicht wird dies durch Schema-Definition (siehe 2.2.2), durch die URIs für verschiedene Typen von Ressourcen angepasst werden können, ohne bestehende Konventionen zu beeinflussen.
Der Begriff „Ressource“ basiert in erster Linie auf der Definition 2.1 von Fielding. Eine Ressource ist nicht zwangsläufig ein Dokument oder Bild das über das Internet abgefragt werden kann, sondern alles was sich durch einen Namen identifizieren lässt. Dies umfasst auch Dinge aus der realen Welt oder abstrakte Konzepte.
Der letzte Begriff „Identifikator“ bedeutet nicht zwangsläufig, dass die URI selbst die Identität der Ressource definiert oder den Zugriff auf sie ermöglicht. Stattdessen wird lediglich gefordert, dass eine einer Ressource zugordnete URI diese eindeutig von allen anderen Ressourcen unterscheiden muss.

2.2.2. Syntaktischer Aufbau

Die Syntax von URIs wurde in RFC3986[14] definiert. Sie entspricht der allgemeinen Form:
URI = scheme ":"hier-part ["?"query] ["#"fragment]
Die Syntax einer URI ist generisch gehalten, weshalb am Anfang jeder URI ein Schema definiert wird. Das Schema schränkt die Syntax und Semantik der URI ein und passt sie für bestimmte Anwendungsbereiche an. Dies sorgt für die Uniformität der URIs, welche durch neue Schema-Definitionen beliebig erweitert werden können. Ein Vorteil der sich daraus ergibt ist die Möglichkeit einheitliche URI-Parser zu entwickeln, welche URIs unabhägig vom Schema und damit vom Typ der Ressource genutzt werden können.
Dem Schema folgt eine Hierarchie, welche sich auch als Baumstruktur darstellen lässt. Die einzelnen Abschnitte dieser Hierarchie werden durch von der Syntax vorgegebene Zeichen getrennt, wobei wir von links nach rechts im Baum hinabsteigen und die Anfrage spezialisieren. Eingeleitet wird sie in vielen Schemata durch die Autorität, dem Teil der bestimmt wer die Autorität über die URI besitzt. Er enthält einen Host-Namen und optional Benutzerinformationen sowie den genutzten Port. Ein bekanntes Anwendungsbeispiel dafür ist das „Domain Name System (DNS)“. Es folgt der eigentliche Pfad, welcher zusammen mit einer optionalen Query die Ressource im Geltungsbereich der Autorität abhängig vom Schema identifiziert. Die Query enthält dabei nicht hierarchisch angeordnete Daten wie Parameter.

Der Fragment-Abschnitt ist ebenfalls optional und erlaubt die Identifizierung einer sekundären Ressource durch Referenz zu einer primären Ressource. In der Regel handelt es sich dabei um einen Teil oder eine bestimmte Repräsentation der primären Ressource. Angewendet wird dieser Abschnitt z.B. im Schema „http“ für die Realisierung von Sprungmarken innerhalb eines HTTP-Dokumentes.

2.3. Deep-Linking

Der Begriff „Deep-Linking“ beschreibt im Kontext des „Word Wide Web“ die direkte Verlinkung von Teilen des Inhalts einer Webseite. Webseiten bilden gemäß den URIs eine hierarchische Struktur. Ein Deep-Link nutzt daher den Hierarchie-Abschnitt und den Query-Abschnitt, um einen Nutzer gezielt auf eine in der Hierarchie tiefer liegende Ressource zu leiten. Der Vorteil dabei ist, dass diese Ressourcen einfacher und schneller zu erreichen sind. Der Nutzer muss die Hierarchie der Webseite nicht mehr selbstständig traversieren um zum gesuchten Inhalt zu gelangen und die Last des Netzwerkes wird auf das notwendige Minimum reduziert.
Die Nutzung von Deep-Links sorgte in der Vergangenheit für viele Rechtsstreitigkeiten insbesondere zwischen Inhabern von Urheberrechten und Suchmaschinen. Durch Deep-Links werden Start- bzw. Portalseiten umgangen, wodurch bei Seiteninhabern in der Regel Werbeeinahmen wegfallen. Der Rechtsstreit führte bis zum Bundesgerichtshof, der ein gefordertes Verbot jedoch ablehnte[1]. Das Thema Deep-Linking wurde ebenfalls vom World Wide Web Consortium mit Blick auf die Definition von URI und dem technischen Aspekten des WWW diskutiert[7]. Sie betonen nochmal ausdrücklich, dass URIs lediglich der Identifizierung von Ressourcen dienen, nicht aber den Zugriff darauf regeln. Aus technischer Sicht besteht zudem kein Unterschied zwischen der URI einer Startseite oder einer Unterseite, sodass beide als Deep-Link bezeichnet werden können. Es ist daher wenig sinnvoll ein Verbot zu fordern, stattdessen wird der Einsatz von vorhandenen Technologien zur Zugriffskontrolle vorgeschlagen.
Im Kontext dieser Arbeit sind Deep-Links URIs, die es ermöglichen Teile eines Dokumentes in verschiedensten Dateiformaten zu identifizieren. Auf diese Teile soll dann mittels einer im Hintergrund liegenden Datenschicht von PANDA zugegriffen werden, um sie für die Rückgabe zu extrahieren. Die Deep-Links entsprechen dabei den in dieser Arbeit definierten Schemata.

3. Analyse geläufiger Datenformate offener Datensätze

In diesem Kapitel werden häufig für offene Daten genutzte Dateiformate ausgewählt und auf ihre Struktur hin analysiert. Ziel ist es, Strukturinformationen zu finden, die für die gezielte Extraktion von Daten dienen können. Dabei werden insbesondere die Teile der Struktur hervorgehoben, welche für die Extraktion von Text von Relevanz sind. Des Weiteren wird geprüft, inwiefern Daten typisiert werden können wenn dies sinnvoll erscheint. In diesem Kapitel wird nicht darauf eingegangen, inwiefern sich die Strukturinformationen für die Verwendung in URIs eignen und wie diese auszusehen haben. Dies wird Gegenstand des Kapitels 4 sein, dessen Grundlage dieses Kapitel schaffen soll.

3.1. Auswahl repräsentativer Datenformate

Als ersten Schritt müssen zu analysierenden Datenformate ausgewählt werden. Dazu ist es notwendig eine repräsentative Menge für alle Formate zu finden, die für offenen Daten eingesetzt werden und bei denen eine Datenextraktion sinnvoll sein könnte. Zu diesem Zweck werden 15 verschiedene Datenplattformen für offene Daten untersucht. Diese geben oftmals die Anzahl verfügbarer Datensätze in einem bestimmten Format an. Das Ergebnis ist sehr breit gefächert und bedarf weiterer Filterung.
Der erste Schritt ist das Entfernen aller Datenformate aus der Liste, welche nur auf einer der Plattformen eingesetzt wurden. In diesem Fall kann davon ausgegangen werden, dass das Format sehr spezifisch ist und keine breite Anwendung auf anderen nicht untersuchten Plattformen findet..
Der zweite Schritt ist die Bereinigung der Liste von Formaten, bei denen das Deep-Linking von Daten nicht möglich oder nicht sinnvoll ist. Dazu gehören alle Bild-Dateien, welche bei Bedarf problemlos als eigenständige Ressource verlinkt werden können. Des Weiteren wurden auch Archiv-Formate aus der Liste entfernt, da diese eine Vielzahl anderer Formate enthalten die bereits in der Liste aufgenommen wurden. Worum es sich dabei in der Masse handelt ist an dieser Stelle nicht nachvollziehbar.
Die absoluten Zahlen der verbleibenden Datenformate befinden sich im Anhang auf Seite 91 in Tabelle A.1. Die baumstrukturierte **HTML** ist das am häufigsten unterstützte Format. Oft handelt es sich dabei jedoch nur um eine Vorschau der eigentlichen Daten oder um eine Erklärungen für das Verständnis von Daten und Formatierung. Dennoch wurde HTML in die Liste aufgenommen unter der Annahme, dass durch Extraktion und Deep-Linking auf HTML bereits im Web verfügbare Daten aufgearbeitet werden können. Neben HTML ist auch **XML** stark vertreten. XML wird vor allem in der Kommunikation

von Maschine zu Maschine eingesetzt und spielt daher bezüglich Datenextraktion und Deep-Linking eine wichtige Rolle.
Neben dem reinen XML enthält die Liste XML-basierte Formate wie „RSS“, „GML“ und „KML“. Das ebenfalls häufig in XML gespeichert „RDF“ besitzt oft bereits gültige Links aufgrund seines Einsatzes im Bereich „Open Linked Data“. Diese Formate werden daher außer acht gelassen.
Häufigster Vertreter der tabellenartigen Formate ist das Format **CSV**. Daneben ist auch das **XLS**-Format sehr häufig vertreten. Da XLS als überholt gilt und durch neue Office-Versionen in Zukunft durch **XLSX** verdrängt wird, wird dieses Format zusammen mit CSV ausgewählt. Die Konvertierung von einem in das andere Format stellt zumindest auf die Daten selbst betrachtet kein Problem dar.
Als Repräsentanten für Formate aus der Textverarbeitung wurden sowohl **PDF** als auch **DOCX** ausgewählt. Das DOCX-Format wird aus dem gleichen Grund wie XLSX dem DOC-Format gegenüber bevorzugt. Es dient in diesem Fall lediglich als Vergleich zu PDF, welches eine deutlich höhere Anwendung bei in offenen Datensätzen vorweisen kann.
Die Formate „ODS“, „TXT“ und „JSON“ werden ebenfalls nicht behandelt, da bereits ähnliche Formate für die Analyse ausgewählt wurden. Für ihre Verarbeitung können dennoch Rückschlüsse auf Basis der Ergebnisse der ausgewählten Formate gezogen werden.

3.2. Extensible Markup Language (XML)

Die „Extensible Markup Language“ (XML)[6] ist eine Erweiterung der nach ISO 8879 standardisierten allgemeinen Auszeichnungssprache „Standard Generalized Markup Language“ (SGML), welche als Grundlage für verschiedene Auszeichnungssprachen dienen. XML dient zur hierarchischen Darstellung strukturierter Daten. Häufige Einsatzmöglichkeiten sind der Austausch von Daten zwischen Systemen oder die Verwendung als Konfigurationsdateien.
XML-Dokumente können in verschiedenen Kodierungen vorliegen. Sie enthalten Text, der vorgegeben durch die Syntax des Standards eine hierarchische Struktur bildet. Grundlegende Struktureinheit ist der Element-Knoten zu denen auch der Wurzel-Knoten gehört. Jeder Element-Knoten kann Text-Knoten und beliebig viele andere Text-Knoten als Kinder haben. Die Knoten werden mit Tags in der Form <Knoten>Text</Knoten>definiert. Abkürzungen wie in sich geschlossene Knoten ohne Kinder der Form <Knoten/>sind ebenfalls möglich. Die Verschachtelung geschieht in Bezug auf die Ebenen in symmetrischer Reihenfolge. Das bedeutet, dass ein Knoten nicht durch ein End-Tag geschlossen werden darf, bevor nicht alle seine Kinder und deren Kinder durch ihre End-Tags geschlossen wurden. Die in den Tags kodierten Namen der Elemente sind nicht eindeutig. Besitzen mehrere Kind-Knoten des selben Eltern-Knoten den gleichen Elementnamen, so kann ein Knoten nur durch die Reihenfolge des Auftretens dieser Knoten genau referenziert werden. Den Namen kann außerdem ein Präfix welcher einen Namensraum repräsentiert zugewiesen werden. Dieser muss zuvor in einem Vorfahren des Knoten definiert worden sein. Namensräume die für das gesam-

te Dokument gelten werden im Wurzel-Knoten definiert. Dieses Konzept erlaubt die Entwicklung und das beliebige Mischen verschiedener von XML abgeleiteten Sprachen. Des Weiteren ist es möglich Attribute für Knoten zu definieren. Diese werden direkt im Start-Tag in der Form <Element AttributName="Value">kodiert. Der Name eines Attributes ist innerhalb des jeweiligen Knoten im Gegensatz zu den Elementnamen eindeutig. Werte lassen sich folglich sowohl als Text-Knoten, als auch als Attribut definieren.
XML ist selbst nicht typisiert. Bei den Werten handelt es sich um gewöhnlichen Text. XML-Dokumente können jedoch nach einem Schema[17] aufgebaut sein und nach diesem validiert werden. Eine Typisierung ist demnach nur möglich, wenn beides vorhanden ist. Der Zugriff auf Elemente, Attribute, Text oder ganze Teilbäume kann entsprechend der Baumstruktur durch Pfade innerhalb dieses Baumes realisiert werden. Speziell für XML wurde dazu die Hilfssprache XPath[16] entwickelt. Neben der Vorgabe einer Syntax für die Pfadangaben definiert der XPath-Standard eine Reihe von Funktionen bis hin zur Nutzung von regulären Ausdrücken.

3.3. Hypertext Markup Language (HTML)

Die „Hypertext Markup Language “ (HTML) ist wie XML eine Erweiterung der nach ISO 8879 standardisierten allgemeinen Auszeichnungssprache „Standard Generalized Markup Language“ (SGML). Sie wurde für den Einsatz in Word Wide Web entwickelt und gilt dort als allgemeiner Standard für die Strukturierung von Inhalten. HTML definiert dabei den dargestellten Inhalt und enthält Informationen, wie dieser dargestellt werden soll. Darüber hinaus werden auch nicht direkt sichtbare Metadaten kodiert.
Der HTML ist selbst nicht zu XML kompatibel, da er Konstrukte erlaubt die gegen die Syntax von XML verstoßen. Folglich funktionieren auch die für XML entwickelten Technologien nicht, weshalb die strengere auf XML basierende Form „Extensible Hypertext Markup Language“ (XHTML) entwickelt wurde. Sie behebt die Mängel[15], die einen Einsatz von XML-Technologien verhinderte und diente als Grundlage für die Weiterentwicklung von HTML. Es wurden diverse Werkzeuge für die Konvertierung von HTML zu XHTML entwickelt. Durch Einsatz dieser Werkzeuge wäre auch bei HTML der Einsatz von XPath für die Referenzierung von Inhalten in HTML denkbar.

3.4. Comma Separated Values (CSV)

Das textbasierte Dateiformat „Comma Separated Values“(CSV) wird für den Austausch strukturierter Daten genutzt. Es gibt keinen konkreten Standard für das Format, weshalb in erster Linie auf Absprachen zwischen Erzeuger und Konsumenten der Daten zurückgegriffen werden muss. Mit RFC4180[2] wurde erstmals eine grundlegende Beschreibung des Formates veröffentlicht, wobei man an diese nicht gebunden ist.
Vom Prinzip her handelt es sich bei dem Format um eine einfache Textdatei, welche durch Trennzeichen in Spalten und Zeilenumbrüche in Zeilen getrennt wird. Die Trennzeichen sind dabei aufgrund des fehlenden Standards nicht genau festgelegt. In RFC4180

wird das Komma-Zeichen *U+002C* genutzt, jedoch ist auch die Nutzung von Semikolon *U+003B* und Tabulator *U+0009* weit verbreitet. Des Weiteren sind die Nutzung von einem Tabellenkopf, die zusätzliche Markierung von Werten z.B. mit den Anführungszeichen *U+0022* und die Nutzung von Escape-Sequenzen z.B. für die genutzten Trennzeichen möglich.
Das einfache Format bietet eine schnelle und leicht verfügbare Alternative zu gängigen Spreadsheet-Formaten. Die Referenzierung der Daten ist durch Indizes, jeweils für Spalten und Zeilen möglich. Im einfachsten Fall können diese direkt durch die Trennung mittels Trennzeichen ermittelt werden. Für eine genauere Verarbeitung nach RFC4180, ist jedoch der Einsatz eines Parsers notwendig.
Eine Typisierung der Daten ist bei PANDA nicht möglich, da diese von Nutzerabsprachen abhängig sind. Daten aus CSV-Dateien müssen daher als Text behandelt und entsprechend beim Konsumenten der Daten konvertiert werden.

3.5. Microsoft Office

„Microsoft Office“ ist ein von Software-Hersteller Microsoft angebotenes Software-Paket bestehend aus verschiedenen Office-Programmen für unterschiedliche Anwendungszwecke. Bezüglich Datenextraktion und Verlinkung offener Daten im Rahmen von PANDA interessieren wir uns in erster Linie für die Formate des Textverarbeitungsprogramms MS-Word und des Tabellenkalkulationsprogramms MS-Excel. Wir unterscheiden hierbei die älteren Formate mit den Datei-Endungen „.doc“ bzw. „.xls“ die 1997 erschienen sind und die auf „Office Open XML“[24] basierenden Formate mit den Datei-Endungen „.docx“ bzw. „.xlsx‘, welche 2007 erschienen sind.
Die Office97-Formate wurden lange unter Verschluss gehalten und konnten nur mühsam mittels Reverse-Engeneering und mit Einschränkungen in anderen Applikationen verwendet werden. Eine ausführliche Spezifikation wurde erst 2008[10] und damit sogar nach Erscheinen des neuen Standards veröffentlicht. Aufgrund der Komplexität der alten Formate und der Tatsache, dass diese immer mehr verdrängt werden und eine Konvertierung ohne weiteres möglich ist, werden diese hier nicht weiter behandelt. Da die grundlegenden Konventionen für die Identifikation von Daten weitestgehend beibehalten wurde, ist eine Implementation dennoch möglich, sofern ein Framework mit entsprechender Unterstützung für die Verarbeitung gewählt wird.
Das OOXML-Format wurde als Folge auf die Kritik am Office97 Format entwickelt, da dieses durch den geschlossenen Standard nur begrenzte Interoperabilität erlaubte. Schon 2006 wurde mit „OpenDocument“ ein Format für Office-Dokumente in ISO/IEC 26300 standardisiert. Dennoch entwickelte Microsoft einen eigenen Standard, welcher 2008 unter ISO/IEC 29500 spezifiziert wurde. Durch die hohe Verbreitung von MS-Office gilt dieser mittlerweile als De-Facto-Standard für den Austausch von Office-Dokumenten.
In den folgenden Unterabschnitten wird das OOXML-Format in Bezug auf MS-Word und MS-Excel grob beschrieben. Dabei werden die Eigenschaften welche für die Identifikation und Extraktion von Daten von Relevanz sind hervorgehoben. Die Beschreibungen beschränken sich auf Informationen für ein grobes Verständnis der Struktur von

OOXML-basierten Dateiformaten.

3.5.1. Paketstruktur

Im Rahmen von OOXML wurden die „Open Packaging Conventions“ spezifiziert. Es handelt sich dabei um ein abstraktes Paket-Model. Dieses definiert Container, genannt Pakete, welche zur Bündelung von XML-Dateien und anderen Entitäten, wie z.B. Bilddateien, dienen sollen. Durch die Gliederung in verschiedene dedizierte Teile lassen sich durch mehrfache Verwendung eine kompaktere Dateigröße erreichen, Zugriffe vereinfachen und Effekte von Dateifehlern minimieren.
Neben den einzelnen Elementen beinhalten Pakete Informationen über den Typ der Elemente und ihren Beziehungen untereinander. Jedes Element besitzt einen Namen und einen Typ für seinen Inhalt.
Der Name dient der Referenzierung des Elementes innerhalb vom Paket. Der Elementname „*/hello/world/doc.xml*“ besteht z.B. aus 3 Teilen, wobei die ersten beiden Teile Informationen über die Hierarchie beinhalten und der letzte Teil den eigentlichen Inhalt. Bei den Namen handelt es sich um IRI[1] welche als IRI oder URI mit ein paar zusätzlichen Einschränkungen kodiert werden. Der OOXML Standard empfiehlt das Verwenden der gängigeren URIs um zu älteren ZIP-Versionen, welche keine Unicode-Zeichen unterstützen, kompatibel zu bleiben.
Der Typ eines Elementes ordnet diesem einen Mediatyp gemäß MIME[2] zu. Die Typen werden direkt im Paket in der „*[Content_Types].xml*“ als Stream für alle Elemente gespeichert. Die Wahl der Zeichen „[“ und „]“ wurden dabei explizit gewählt, da sie die Konventionen für Namen von Elementen verletzen um den Content-Type-Stream von diesen zu unterscheiden. Die Content-Type-Items entsprechen den in RFC 2616 festgelegten Konventionen. Das Listing 3.1 zeigt ein Beispiel für das Content-Type-Item des Hauptteils eines Word-Documentes.

```
<Override PartName="/word/document.xml" ContentType="
   application/vnd.openxmlformats-officedocument.
   wordprocessingml.document.main+xml"/>
```

Listing 3.1: Beispiel eines Items aus dem Content-Type-Stream

Elemente die selbst XML-Dokumente sind unterliegen bestimmten Einschränkungen. Die Kodierung muss UTF-8 oder UTF-16 sein. Außerdem dürfen keine DTDs verwendet werden. Für OOXML wird die Struktur der XML-Dokumente durch XML-Schemata, welche für jeden Typ im Standard definiert wurden, vorgegeben.
Für jedes Element können Beziehungen zu anderen Elementen gespeichert werden. Diese befinden sich in einem Unterordner mit dem Namen „*_rel*“, welcher sich auf der gleichen Ebene der Hierarchie wie das Element selbst befindet. Jedes Element der selben Ebene der Hierarchie legt, wenn es Beziehungen zu anderen Elementen besitzt, in diesem

[1] *http://tools.ietf.org/html/rfc3987*
[2] spezifiziert in RFC 2045-2049

Ordner eine gleichnamige Datei mit der Endung „*.rels*“ an, in welchem es diese Beziehungen speichert. Jede Beziehung wird durch ein Relationship-Item repräsentiert. Dieses besitzt eine ID, einen Typ der die Art der Beziehung wiederspiegelt und ein Ziel. Dabei werden die von XML-Schema definierten Typen „*xsd:ID*“ und „*xsd:anyURI*“ verwendet. Optional kann außerdem ein Zielmodus festgelegt werden, welcher entweder „*Internal*“ oder „*External*“ ist. Externe Ziele können z.B. URLs sein, während interne Ziele die Elemente des Paketes selbst sind. Diese werden nicht durch vollständige Namen wie durch IRIs, sondern durch relative Pfade, ausgehend vom Quell-Element, referenziert. Ist dieses Attribut nicht vorhanden, wird immer von einer internen Beziehung ausgegangen.
Ein Beispiel für ein internes Relationship-Item, welches das Styles-Element referenziert und ein externes Relationship-Item, welches eine URL referenziert ist in Listing 3.2 abgebildet.

```
<Relationship Id="rId1" Type="http://schemas.
   openxmlformats.org/officeDocument/2006/relationships/
   styles" Target="styles.xml"/>
<Relationship Id="rId2" Type="http://schemas.
   openxmlformats.org/officeDocument/2006/relationships/
   hyperlink" Target="https://www.google.de/" TargetMode="
   External"/>
```

Listing 3.2: Beispiel für interne und externe Relationship-Items

Die Beziehungen ermöglichen es Zusammenhänge zwischen den Elementen des Paketes zu erkennen, ohne deren Inhalt betrachten zu müssen. Ihre Nutzung innerhalb von Elementen kann auf explizite oder implizite Weise geschehen, wobei die Art nicht frei wählbar ist, sondern vom Standard festgelegt wird.
Eine explizite Referenz nutzt direkt das im Relationship-Item ID-Attribut. Die implizite Referenz dagegen nutzt ein vordefiniertes Tag und eine ID, welche im Ziel-Element definiert wurde. Zur Referenzierung des Elementes wird von beiden jedoch die im Relationship-Item definierte Target-URI genutzt. Das Relationship-Item wird bei der impliziten Referenz durch die OOXML-Konventionen mittels Tag identifiziert. Die Listings 3.3 und 3.4 zeigen Beispiele für eine explizite bzw. implizite Referenz welche die Beziehungen welche in Listing 3.2 definiert wurden nutzten.

```
<w:hyperlink r:id="rId2" w:history="1">
...
</w:hyperlink>
```

Listing 3.3: Beispiel für eine explizite Referenz

```
<!--Referenz:-->
<w:pPr>
  <w:pStyle w:val="Titel"/>
</w:pPr>
<!--Referenziert einen Style im Styles-Element:-->
<w:style w:type="paragraph" w:styleId="Titel">
...
</w:style>
```

Listing 3.4: Beispiel für eine implizite Referenz

Die nach dem OPC-Modell definierten Pakete werden physisch nach OOXML in Form von ZIP-Archiven repräsentiert. Die Elemente wiederum sind die ZIP-Items dieses Paketes. Ein OOXML-Dokument lässt sich folglich mit den üblichen Programmen für die Handhabung von ZIP entpacken und seine Elemente betrachten.

3.5.2. WordprocessingML

WordprocessingML ist die XML-basierte Auszeichnungssprache für Word-Dokumente nach dem OOXML-Standard. Wie alle OOXML-Dokumente besteht es aus mehreren Elementen verpackt in einen ZIP-Archiv, welches als Container für diese Teile dient. Ein minimales Word-Dokument besteht aus folgenden Elementen:

- „*/word/document.xml*“ - Das Hauptdokument, in dem Text und Aufbau des Dokumentes gespeichert werden.
- „*/word/styles.xml*“ - Das Styles-Element, in welchem verschiedene Styles für die Verwendung im Hauptdokument definiert werden.
- „*/word/settings.xm*“ - Allgemeine Einstellungen für das Dokument.
- „*/word/webSettings.xml*“ - Web-bezogene Einstellungen.
- „*/word/theme/theme1.xm*“ - Thema des Dokumentes welches Farben, Fonts und Effekte bestimmt.
- „*/word/fontTable.xml*“ - In das Dokument eingebettete Fonts.

Dazu kommen XML-Dateien mit Metadaten über das Dokument und die im vorherigen Abschnitt 3.5.1 beschriebenen Beziehungen und der Content-Type-Stream. Die Paketstruktur mit dedizierten Elementen erlaubt die Wiederverwendung mehrfach genutzter Definitionen wie z.B. Styles und sorgt dadurch für eine bessere Kompaktheit des gesamten Dokumentes.
Für die Extraktion und Verlinkung von Text, ist das Hauptdokument das wichtigste Element. Wo sich dieses im Paket befindet, lässt sich einfach aus den Beziehungen des Paketes ableiten. Die Hierarchie des Hauptdokumentes besteht der Reihenfolge nach aus folgenden Elementen:

- „*w:document*“ - Wurzel-Knoten.
- „*w:body*“ - Knoten mit dem Inhalt des Word-Dokumentes.
- „*w:p*“ - Paragraph.
- „*w:r*“ - Text-Run.
- „*w:t*“ - Text.

Der Wurzel-Knoten *w:document* enthält einen Knoten *w:body*, welcher wiederum den Inhalt des Word-Dokumentes und Strukturinformationen enthält. In diesem Knoten befinden sich ein oder mehrere *w:p* Knoten, welche als Paragraphen bezeichnet werden. Paragraphen sind Textabschnitte die in Word explizit mit einem „Return“ bzw. „Enter“ getrennt wurden. In der Ausgabe von MS-Word sind diese von den sogenannten „Softbreaks“ nicht immer zu unterscheiden. Bei Textumbrüchen drückt sich dies jedoch in der Regel durch voreingestellte unterschiedliche Zeilenabstände aus.
Ein Paragraph kann wiederum einen oder mehrere *w:r* Knoten umfassen. Diese als „Text-Runs“ bezeichneten Knoten umfassen Text mit einheitlicher Formatierung. Auf diese Weise lässt sich eine kompaktere Form der Formatierung umsetzen. Die Style-Informationen befinden sich dabei in einem Kind-Knoten des Text-Runs.
Das kleinste Element in der Hierarchie ist der *w:t* Knoten, welcher als Container für den eigentlichen Text dient. Ein Text-Knoten kann dabei beliebig viel Text enthalten. Dennoch ist es möglich, dass ein Text-Run mehrere dieser Knoten beinhaltet, z.B. wenn Softbreaks eingefügt werden. Das Listing 3.5 bildet ein einfaches Hauptdokument in WordML ab. Es besteht aus zwei Paragraphen, wobei im Text des ersten Paragraphen ein Softbreak für Textumbruch eingefügt wurde. Die Definition der Namensräume wurde im Beispiel ausgelassen.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<w:document ...>
  <w:body>
    <w:p>
      <w:r>
        <w:t>Hallo</w:t>
        <w:br w:type="text-wrapping"/>
        <w:t>Welt!</w:t>
      </w:r>
    </w:p>
    <w:p>
      <w:r>
        <w:t>Hallo Welt!</w:t>
      </w:r>
    </w:p>
  </w:body>
</w:document>
```

Listing 3.5: Beispiel für Hauptdokument in WordML

Eine feste Seitenstruktur gibt es in WordprocessingML nicht. Vielmehr werden Seiten durch den Knoten *<w:lastRenderedPageBreak/>* innerhalb des ersten Paragraphen einer neuen Seite markiert.
Neben den einfachen Paragraphen können auch Knoten mit Tabellen im Body des Dokumentes auftauchen. Diese bestehen der Reihenfolge nach aus folgenden Elementen:

- „*w:tbl*“ - Tabellen-Knoten.
- „*w:tr*“ - Tabellen-Zeile.
- „*w:tc*“ - Tabellen-Spalte/Zelle.

Die Tabellen sind wie in OOXML übliche zeilenorientiert. Die Knoten *w:tc* enthalten ebenfalls Paragraphen die auf die selbe Weise strukturiert sind wie jene die für den normalen Text verwendet wird.
Neben dieser grundlegenden Struktur gibt es noch zusäzliche Möglichkeiten WordML-Dokumente zu gliedern. Sie können beispielsweise in Sektionen unterteilt werden, durch das Einfügen entsprechender Knoten am Ende jeder Sektion. Diese sind jedoch nur optional und sollten daher für den allgemeinen Zugriff auf Word-Dokumente nicht verwendet werden.

3.5.3. SpreadsheetML

SpreadsheetML ist die XML-basierte Auszeichnungssprache für Excel-Dokumente nach dem OOXML-Standard. Wie alle OOXML-Dokumente besteht es aus mehreren Elementen verpackt in einen ZIP-Archiv, welches als Container für diese Teile dient. Ein minimales Excel-Dokument besteht aus folgenden Elementen:

- „*/xl/workbook.xmll*“ - Das Hauptdokument in welchem die einzelnen Tabellen verwaltet werden.
- „*/xl/worksheets/sheet1.xml*“ - Ein Tabellen-Element wobei jede Tabelle ihr eigenes Element besitzt.
- „*/xl/theme/theme1.xmll*“ - Thema des Dokumentes welches Farben, Fonts und Effekte bestimmt.
- „*/xl/styles.xmll*“ - Das Styles-Element, in welchem verschiedene Styles für die Verwendung in den Tabellen definiert werden.

Dazu kommen XML-Dateien mit Metadaten über das Dokument und die im vorherigen Abschnitt 3.5.1 beschriebenen Beziehungen und der Content-Type-Stream. Die Paketstruktur mit dedizierten Elementen erlaubt die Wiederverwendung mehrfach genutzter Definitionen wie z.B. Styles und sorgt dadurch für eine bessere Kompaktheit des gesamten Dokumentes.
Für die Daten-Extraktion und Verlinkung der Zellen sind das Hauptdokument und die darin referenzierten Tabellen-Elemente entscheidend. Die Wurzel des Hauptdokumentes enthält neben Metadaten über das Dokument eine Liste mit einer oder mehr enthaltenen Tabellen. Im *sheets* Knoten werden diese in der Form *<sheet name=“Tabelle1“ sheetId=“1“ r:id=“rId1“/>* aufgelistet. Wo sich die Tabelle im Paket befindet, kann über die ID der Beziehung im Attribut *r:id* ermittelt werden. Darüber hinaus besitzt jede Tabelle einen Namen und eine ID. Die ID ist vom Typ Integer und wird in der Regel sequentiell mit jeder hinzugefügten Tabelle inkrementiert. Der Name besteht aus einem 1 bis 31 Zeichen langen String der Unicode-Zeichen bis auf einige Einschränkungen erlaubt. Eine pseudoartige Grammatik ist in Tabelle 3.1 abgebildet. Zeichen die im zugrundeliegenden XML nicht erlaubt sind werden wie in HTML üblich kodiert.

sheet-name	=	*start-end-character* , {*character*} , *start-end-character*
start-end-character	=	%x0001-%xFFFF
character	=	%x0001-%xFFFF
invalid-start-end-character	=	%x0003 | “*“ | “?“ | “’“ | “[“ | “]“ | “\“ | “:“ | “/“
invalid-character	=	%x0003 | “*“ | “?“ | “[“ | “]“ | “\“ | “:“ | “/“

Tabelle 3.1.: Pseudo-Grammatik für Tabellennamen in SpreadsheetML

Die Daten der Tabelle werden im Tabellen-Element in Zeilen und Spalten gegliedert. Der Wurzel-Knoten enthält neben zusätzlichen Informationen über die Tabelle einen *sheetData* Knoten. SpreadsheetML ist Zeilen-orientiert, so sind die Kinder dieses Knotens Zeilen mit einem Attribut für ihren Zeilenindex. Diese enthalten wiederum Knoten, die jeweiles eine Zelle repräsentieren. Jede Zelle besitzt ein Attribut, dass die vollständige Referenz der Zelle wie z.B. „A1“ beinhaltet. Es handelt sich dabei um eine Kombination aus Spalten-Referenz und Zeilen-Referenz nach dem in Excel üblichen Schema. Die

Buchstaben gehen dabei sequentiell das Alphabet von A bis Z durch und werden dann um eine Stelle auf AA erweitert und so fortgeführt.
Der von der Zelle dargestellte Wert wird in einem Knoten v gespeichert. Es handelt sich dabei um den Wert in der Ausgabe. Formeln werden in Extra-Knoten gelistet und anhand einer Formelkette bei Updates neu berechnet. Die Formelketten werden dabei in einem separaten Element des Paketes gespeichert.
Bei einer einfachen Extraktion der Daten kann davon ausgegangen werden, dass der im Value-Knoten gespeicherte Wert bereits korrekt berechnet wurde. Wichtig wird dies hingegen bei Schreibzugriffen, welche Eingabewerte einer solchen Formelkette überschreiben und eine neue Berechnung notwendig machen.
Für die Extraktion der Daten ist des Weiteren der optional in einer Zelle als Attribut angegebene Typ von Bedeutung. Eine Zelle kann dabei folgenden Typen zugeordnet werden:

- „*bl*“ - Boolean, TRUE oder FALSE.
- „*d*“ - Date, Double-Wert.
- „*e*“ - Error, Fehlerausgabe z.B. #DIV/0!
- „*inlineStr*“ - Inline String.
- „*n*“ - Number, Double und Standard-Typ einer Zelle.
- „*s*“ - SharedString, Integer-Wert.
- „*str*“ - Formelstring ohne Berechnung.

Besonders beachtet werden müssen Zell-Knoten von den Typen *Date* und *SharedString*. In Zellen vom Typ *Date* werden einfache Double-Werte gespeichert. Diese müssen nach ISO 8601 in einen Datums-String umgewandelt werden.
Zellen ohne explizit angegeben Typ enthalten einfache Double-Werte. In der Regel handelt es sich dabei um Zellen vom Typ *Number*, jedoch ist der Typ *Date* nicht auszuschließen, da dieser ebenfalls als Double dargestellt wird. Dies muss über den angegebenen Style, welcher einen Style im Style-Element des OOXML-Paketes referenziert, geprüft werden.
Eine weitere Besonderheit ist der Typ *SharedString*. Der Wert dieser Zellen ist ein Integer, welcher einen String im SharedString-Element des OOXML-Paketes referenziert. Auf diese Weise können im Gegensatz zu InlineStrings, die Strings wiederverwendet werden um so Platz zu sparen.
Ein einfaches Beispiel für eine Tabelle ist in Listing 3.6 abgebildet. Beachte: A1 und B1 nutzen den gleichen String aus dem SharedString-Element als Text.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<worksheet ...>
  ...
  <sheetData>
    <row r="1" spans="1:2">
      <c r="A1" t="s"><v>0</v></c>
      <c r="B1" t="s"><v>0</v></c>
    </row>
    <row r="2" spans="1:2">
      <c r="A2" t="s"><v>1</v></c>
      <c r="B2"><v>5</v></c>
    </row>
  </sheetData>
  ...
</worksheet>
```

Listing 3.6: Beispiel für eine Tabelle in SpreadsheetML

3.6. Portable Document Format

Das Portable Document Format (PDF) ist ein von Adobe Systems erstmals 1993 veröffentlichtes Dateiformat für Dokumente. Es basiert teilweise auf dem ebenfalls von Adobe entwickeltem Format PostScript und hat dieses mittlerweile größtenteils abgelöst. Ziel der Entwicklung von PDF war es, ein Format zu erschaffen, welches vom Nutzer unabhängig von seiner Hardware und Software so betrachtet werden kann wie es der Autor vorgesehen hat. Dieser Punkt bezieht sich insbesondere auf die üblichen Formatierungsprobleme bei Konvertierung und Portierung auf andere Systeme wie sie bei anderen Formaten auftreten.
Die Beschreibung des PDF-Formates lässt sich in folgende Punkte gliedern:

(1) Objekte - elementare Objekte aus denen PDF aufgebaut ist.

(2) Dateistruktur - Beschreibung wie Objekte gespeichert, modifiziert oder auf ihnen zugegriffen werden kann.

(3) Dokumentstruktur - Beschreibung wie mit elementaren Objekten Komponenten (z.B. Seiten und Fonts) dargestellt werden.

(4) Content-Stream - Erläuterung wie Sequenzen von Instruktionen die Darstellung auf einer Seite oder graphischen Entität beschreiben.

Da uns lediglich die Extraktion von Text und mögliche Strukturmerkmale für die Verwendung in URIs interessieren, werden wir alle diese Punkte nur anreißen um ein Verständnis für die Struktur und Probleme damit zu schaffen. Für die hier genutzten Beschreibungen wird die PDF-Referenz Version 1.7[20] genutzt.

Ein unverschlüsseltes PDF-Dokument kann als Folge von Byte-Werten, welche Zeichen des Zeichensatzes ANSI X3.4-1986 und zusätzliche Whitespaces repräsentieren, dargestellt werden. Mit ihnen werden Token gebildet, welche Grundlage für die Objekte darstellen durch die die Struktur von PDF-Dokumenten beschrieben wird. Darüber hinaus werden sie für reservierte Schlüsselwörter des PDF-Standards und für Schlüssel in Schlüssel-Werte Paaren von Objekten genannt „Dictionaries“ verwendet. PDF ist jedoch nicht grundsätzlich auf die ASCII-Zeichen beschränkt und kann auch beliebige binäre Werte enthalten. So können String- und Stream-Objekte ausschließlich ASCII-Zeichen oder auch binäre Daten enthalten. In der Praxis werden binäre Daten wie Bilder auch in PDF binär dargestellt, um das Format kompakt zu halten.

3.6.1. Basis-Objekte

Ein PDF Dokument ist aus den Basis-Objekten „Boolean“, „Integer“, „Real“, „Strings“, „Namen“, „Arrays“, „Dictionaries“, „Streams“ und einem Null-Objekt aufgebaut.
Boolean Objekte werden mit den Schlüsselworten „true“ und „false“ notiert.
Die nummerischen Objekte **Integer** und **Real** werden durch Zahlen mit oder ohne Vorzeichen und dem „.“ zur Abgrenzung von Nachkommastellen bei Real-Werten notiert.
Bei **Strings** gibt es 2 Formen: „Literal Strings“ und „Hexadezimal Strings“ .
Literal Strings werden mit runden Klammern markiert. Innerhalb eines Strings können alle ASCII-Zeichen verwendet werden, mit Ausnahme der runden Klammer. Diese müssen dort in ausbalancierter Form oder durch Escape-Sequenzen erscheinen. Darüber hinaus werden Escape-Sequenzen für die Darstellung verschiedener Kontrollsequenzen und Zeichen außerhalb des ASCII-Zeichensatzes verwendet.
Hexadezimal-Strings sind eine einfache Möglichkeit binäre Daten in einem PDF-Dokument unterzubringen. Sie werden durch die ASCII-Zeichen 0-9 und A-F bzw. a-f eingefasst in ein Kleiner- bzw. Größerzeichen dargestellt.
Namen werden als Schlüssel zur Identifikation verwendet. obwohl sie aus ASCII-Zeichen bestehen, sind sie als atomate Einheit zu verstehen. Sie sind zudem einzigartig in ihrem Geltungsbereich. Ihre Notation besteht aus „/“ gefolgt von den ASCII-Zeichen des Namen.
Arrays in PDF sind eine eindimensionale, sequentielle und heterogene Ansammlung von Objekten. Heterogen sind sie in dem Sinne, dass die Elemente verschiedene Objekt Typen sein können, wodurch sich das Konzept von homogenen Arrays in Programmiersprachen unterscheidet. Arrays werden durch die eckigen Klammern „[“ und „]“ notiert.
Das **Dictionary** ist ein für die Definition von Dateistruktur und Dokumentenstruktur häufig genutztes Objekt. Es handelt sich um eine Sammlung von Schlüssel-Werte-Paaren eingeschlossen in jeweils 2 Kleiner- bzw. Größerzeichen. Die Schlüssel werden als Namen notiert, die Werte können beliebige Objekte sein. In 3.7 ist ein Beispiel für ein Dictionary abgebildet.

```
15 0 obj
<< /Type /Example
   /Subtype /DictionaryExample
   /Version 0.01
   /IntegerItem 12
   /StringItem (a string)
   /Subdictionary <<Item1 0.4
                    /Item2 true
                    /LastItem (not!)
                    /VeryLastItem (OK)
                  >>
>>
endobj
```

Listing 3.7: Beispiel für ein Dictionary in PDF

Ein **Stream**-Objekt ist eine sequentielle Ansammlung von Bytes. Sie werden für große Objekte wie Bilder oder Beschreibung von Seiten genutzt. Ein Stream wird durch ein Stream-Dictionary eingeleitet. Es folgt ein Zeilenumbruch und das Schlüsselwort „stream“. Nach dem Byte-Stream aus 0 oder mehr Bytes folgt das Schlüsselwort „endstream“, welches den Stream schließt. Im einleitenden Stream-Dictionary werden Informationen wie z.B. Länge des Streams und genutzte Filter gespeichert.
Objekte können mit einem Label ausgestattet werden, man spricht dann von „indirekten Objekten“ . Das in 3.7 abgebildete Dictionary ist als ein solches indirektes Objekt notiert. Als Label dient eine Kombination aus Objektnummer und Generationsnummer die dem Objekt vorangestellt wird. Die Objektnummer, im Beispiel 15, muss nicht zwangsläufig fortlaufend sein, auch wenn dies in der Praxis meist der Fall ist. Die Generationsnummer, welche der Objektnummer folgt, startet immer bei 0. Sie wird bei Updates des Objektes inkrementiert. Anschließend folgt das Schlüsselwort „obj“ gefolgt vom Objekt und dem Schlüsselwort „endobj“ . Eine Referenz auf dieses Objekt kann jetzt durch die Notation „15 0 R“ erstellt werden. Alle anderen Objekte die über ein solches Label nicht verfügen werden als „direkte Objekte“ bezeichnet.

3.6.2. Dateistruktur

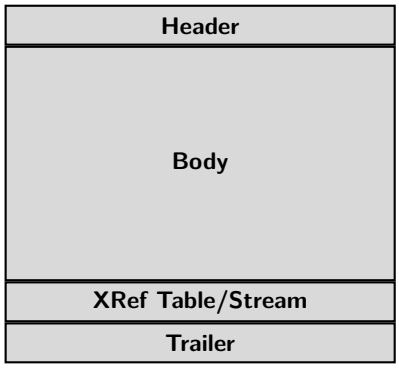


Abbildung 3.1.: PDF Dateistruktur

Eine PDF-Datei ist wie in Abbildung 3.1 zu sehen aufgebaut. Die Elemente sind:

(1) Ein einzeiliger „Header“ der zur Identifikation der Version der PDF-Spezifikation dient.

(2) Der „Body“ in welchem sich das PDF-Dokument befindet.

(3) Der „Cross-Reference Table/Stream“, welcher Informationen über indirekte Objekte

im PDF-Dokument enthält.

(4) Der „Trailer“, welcher die Position vom Cross-Reference-Table/Stream und spezieller Objekte im Body enthält.

Der **Header** besteht aus einer Zeile in der Form „%PDF–1.7“. Seit Version 1.4 wird die im Catalog-Dictionary 3.6.3 statt des Headers verwendet.
Der **Body** enthält das eigentliche PDF-Dokument und besteht aus einer Sequenz indirekter Objekte. Diese stellen die übergeordneten Komponenten, wie z.B. Seiten und Fonts, des Dokumentes dar. Es können auch Objekt-Streams, eine besondere Form von Stream-Objekten, enthalten sein.
Der **Cross-Reference-Table** listet alle indirekten Objekte des Bodies und deren Offset auf. Dadurch wird ein wahlfreier Zugriff auf diese Objekte möglich, womit der Lesezugriff beschleunigt wird. Die Einleitung erfolgt durch das Schlüsselwort „xref“, gefolgt von einer oder mehreren Untersektionen. Jede dieser Sektionen enthält Einträge für aufeinanderfolgende Objekte, „10 3“ würde z.B. Objekte mit den Objektnummern 10 bis 12 referenzieren. Es folgen die Einträge der Form „$Offset\ Generation\ Use\ eol$“. Der Offset ist eine 10stellige Zahl die einen Byte-Offset repräsentiert. Ihm folgt eine Generations-Nummer welche bei Updates inkrementiert wird. Anschließend folgt ein Schlüsselwort, welches entweder „n“für „in use“ oder „f“ für „free“ sein kann. Freie Objekte sind solche die gelöscht wurden. Damit kann die Wiederverwendung der Objektnummern verhindert werden. In 3.8 ist ein Beispiel eines Cross-Reference-Table mit 5 Einträgen in 4 Sektionen zu sehen.

```
xref
0 1
0000000000 65535 f
3 1
0000025325 00000 n
23 2
0000025518 00002 n
0000025635 00000 n
30 1
0000025777 00000 n
```

Listing 3.8: Beispiel für Cross-Reference-Table in PDF

Ab Version 1.5 kann statt dem Cross-Reference-Table auch ein Cross-Reference-Stream verwendet werden. Da dies prinzipiell keinen Unterschied macht, wird an dieser Stelle nicht weiter darauf eingegangen.
Der **Trailer** befindet sich am Ende des Dokumentes. Seine Struktur ist in 3.9 abgebildet. Neben dem „End of File“-Marker und dem Byte-Offset zum Cross-Reference-Table enthält er ein Dictionary mit allgemeinen Informationen z.B. über die Verschlüsellung

des PDF-Dokumentes und Referenzen zu wichtigen Objekten im Body wie dem Catalog-Dictionary3.6.3.

```
trailer
<< key_1 value_1
   key_2 value_2
   ...
   key_n value_n
>>
startxref
Byte offset to cross reference
%%EOF
```

Listing 3.9: Struktur von Trailer in PDF

Eine Möglichkeit Objekte kompakter zu speichern sind die Objekt-Streams, eine spezielle Form des Stream-Objektes. Ein Objekt-Stream enthält einen Stream von beliebigen indirekten Objekten mit Ausnahme anderer Stream-Objekte. Eingeleitet wird der Objekt-Stream mit N Paaren von Integer-Werten, wobei der erste Wert die Objekt-Nummer und der zweite den Offset des Objektes im Stream darstellt. Im Stream werden dann die üblichen Notationen mit den Schlüsselwörtern „obj“ und „endobj“ weggelassen. Zusätzlich kann wie bei anderen Streams ein Filter für Kompression auf den ganzen Stream angewendet werden um weiteren Platz zu sparen. Objekt-Streams sind selbst immer indirekte Objekte und besitzen daher einen Eintrag im Cross-Reference-Table/Stream.

3.6.3. Dokumentstruktur

Der Body einer PDF-Datei enthält das eigentliche PDF-Dokument. Dieses kann als Hierarchie von Objekten gesehen werden. Die Komponenten dieser Hierarchie sind Dictionaries mit Einträgen die eine Eltern-, Kind- und Geschwisterknoten Relation zwischen ihnen definieren.
An der Spitze dieser Struktur steht das **Catalog-Dictionary**. Seine Position ist für schnellen Zugriff im Trailer der PDF-Datei vermerkt. Der für uns am meisten relevante Eintrag im Catalog-Dictionary ist der Page-Tree. Er besteht auf **Page-Tree-Nodes** und **Page-Objekten**. Die Page-Objekte bilden die Blätter des Baumes und repräsentieren jeweils eine Seite. Der Baum kann prinzipiell aus einem Page-Tree-Node als Wurzel mit allen Page-Objekten als Kinder bestehen, in der Praxis erstellen Programme beim Schreiben von PDF-Dokumenten jedoch einem Suchbäumen entsprechenden Page-Tree für schnellere Zugriffe. Eine besondere Eigenschaft des Page-Trees ist die Möglichkeit zur Vererbung von Attributen an Kinder. Wenn die Attribute nicht explizit neu definiert wurden, werden immer die des Eltern-Knoten genutzt.
Das Page-Objekt ist selbst ebenfalls ein Dictionary. Es enthält weitere Attribute der Seite die es repräsentiert und eine Referenz zum eigentlichen **Content-Stream**, vorausgesetzt die Seite ist nicht leer.
Der Content-Stream ist ein Stream-Objekt, welches eine Sequenz von Instruktionen für die Beschreibung der graphischen Elemente einer Seite enthält. Diese Instruktionen sind

selbst direkte Objekte, also solche ohne Label, welche sequentiell verarbeitet werden. Indirekte Objekte sowie Referenzen zu solchen sind innerhalb eines Streams nicht erlaubt. Eine Instruktion besteht immer aus einem Operator und einem Operanden. Operatoren sind in der PDF-Spezifikation festgelegte Schlüsselwörter, welche von Namen durch das weglassen des „/“ unterschieden werden. Diese Schlüsselwörter haben ausschließlich innerhalb von Content-Streams eine Bedeutung. Die Operanden sind direkte Objekte oder Referenzen durch Namen die dem Operator zugetragen werden.
Da innerhalb von Content-Streams nur direkte Objekte erlaubt sind, jedoch in manchen Fällen trotzdem der Zugriff auf externe Ressourcen wie z.B. Fonts notwendig ist, werden durch Namen referenzierte Ressourcen genutzt. Umgesetzt wird dies durch ein **Resource-Dictionary**, welches in Page-Objekten als direktes Objekt definiert wird. Die dort genutzten Namen sind nur im Geltungsraum des jeweiligen Content-Streams von Bedeutung. Außerhalb werden ausschließlich die bereits bekannten Objektreferenzen für indirekte Objekte genutzt. Das Resource-Dictionary ist wie andere Attribute innerhalb des Page-Trees vererbbar.

3.6.4. Text

Die aus Sicht eines Betrachters vorhandene Struktur im Text eines PDF-Dokumentes ergibt sich aus Berechnungen für dessen Darstellung. Diese wird in einem Content-Stream durch Text-Objekte definiert. Die von den Text-Objekten genutzten String-Objekte unterscheiden sich von der in Abschnitt 3.6.1 beschriebenen Form durch ihre Interpretation. Der Inhalt eines String-Objektes wird als Sequenz von Zeichen-Codes interpretiert, wobei ein Zeichen als abstraktes Symbol verstanden wird. Was bei der Ausgabe von PDF-Dokumenten auf dem Bildschirm zu sehen ist sind jedoch keine Zeichen, sondern deren graphische Darstellung, genannt Glyphen.
Die Glypgen sind Gegenstand verschiedener graphischer Manipulationen. Durch eine Vielzahl von Operatoren innerhalb und außerhalb von Text-Objekten wird der Text-Status manipuliert. Dieser ist eine Teilmenge der Grafik-Status-Parameter, welche speziell die Darstellung von Text betrifft. Dazu gehören die Auswahl der Font, die Skalierung der Glyphe und andere grafische Effekte mit denen die grafische Darstellung eines Zeichens verändert wird.
Glyphen sind in sogenannten „Fonts“ organisiert. Sie definieren Glyphen für einen bestimmten Zeichensatz. In PDF werden Fonts durch Font-Dictionaries repräsentiert. Diese referenzieren die Font und enthalten zusätzliche Attribute für das Leseprogramm. Die Font selbst kann dabei in das PDF-Dokument oder das Leseprogramm eingebettet sein oder auch von externen Font-Dateien geladen werden.

```
BT
/F1 12 Tf
0 50 Td
(TEST) Tj
ET
```

Listing 3.10: Beispiel für ein Textobjekt in PDF

Das Beispiel in 3.10 zeigt ein einfaches Text-Objekt. Beginn und Ende des Objektes werden mit den Schlüsselwörtern „BT“ und „ET“ markiert. In der zweiten Zeile wird mit dem Operator „Tf“ die Font mit dem Namen „F1“ und der Schriftgrößte 12 selektiert. Der Name wurde so im Font-Dictionary festgelegt und hat nur interne Bedeutung im PDF-Dokument. Der Operator „Td“ aktualisiert die Text-Matrix, welche Teil des Text-Status ist. Dadurch wird die Startposition des Textes entsprechend des X- und Y-Wertes verschoben. Der „Tj“ Operator sorgt nun dafür, dass mit dem aktuellen Grafik-Status das String-Objekt mit dem Wert „TEST“ ausgegeben bzw. gezeichnet wird.
PDF nutzt für die Darstellung verschiedene 2-dimensionale Koordinatensysteme. Wir unterscheiden Text-Space, User-Space und Device-Space. Der Device-Space ist abhängig vom Ausgabegerät, z.B. ein Monitor oder ein Drucker. Um bei der Spezifizierung von Objekten unabhängig vom Ausgabegerät zu sein, wurde der User-Space eingeführt. Dieser stellt den nutzbaren Platz auf einer Page dar und wird im Page-Objekt initialisiert. Koordinaten in diesem System können jedoch nicht als absolut verstanden werden, da es erlaubt ist den User-Space selbst während der sequentiellen Verarbeitung zu manipulieren.
Die Transformation zwischen 2-dimensionalen Koordinatensysteme können als 3x3-Matrix dargestellt werden. Da nur 6 Elemente der Matrix veränderbar sind, werden diese als Arrays in der Form [a b c d e f] dargestellt wobei die Matrix folgender Form entspricht:

$$M = \begin{Bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{Bmatrix}$$

So lassen sich verschiedene geometrische Manipulationen darstellen, z.B. Translation durch die Matrix [1 0 0 1 t_x t_y], Skalierung durch die Matrix [s_x 0 0 s_y 0 0] und Rotation durch die Matrix [$cos\phi$ $sin\phi$ $-sin\phi$ $cos\phi$ 0 0]. Die transformierten Koordinaten werden dann in folgender Weise berechnet:

$$[x' \ y' \ 1] = [x \ y \ 1] \times \begin{Bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{Bmatrix}$$

Die Matrix T_m für die Transformation des Text-Space wird durch die Instruktion „a b c d e f Tm“ neu initialisiert. Der Initialwert von T_m ist für jede Seite gleich der Einheitsmatrix, also äquivalent zur Instruktion „1 0 0 1 0 0 Tm“. Die Ursprungskoordinate von User-Space und Text-Space sind folglich am Anfang einer Seite identisch. Die

Glyphen selbst werden zunächst in einem Glyph-Space definiert mit einer Text-Font-Matrix zu Text-Space transformiert, sodass sie gezeichnet werden können. Nach dem Zeichnen einer Glyphe erfolgt ein Update der Text-Matrix T_m nach der Formel:

$$T_m = T_m \times \begin{Bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{Bmatrix}$$

wobei gilt:

$$t_x = \left(\left(w0 - \frac{T_j}{1000}\right) \times T_{fs} + T_c + T_w\right) \times T_h$$
$$t_y = \left(w1 - \frac{T_j}{1000}\right) \times T_{fs} + T_c + T_w$$

Die Werte $w0$ und $w1$ entsprechen der horizontalen und vertikalen Verschiebung durch die Glyphe welche in Verbindung mit der Schriftgröße T_{fs} den Raum den die Glyphe im Text-Space einnimmt repräsentiert. Der Parameter T_j dient kleineren Anpassungen der Position. Diese werden innerhalb eines Arrays in Verbindung mit den String-Objekten und dem Operator TJ definiert. So kann z.B. statt „$(TEST)\ Tj$“ mit „$[(TE)-3(S)5(T)]\ TJ$“ eine Justierung der Position der Glyphen erreicht werden. Die Parameter T_c und T_w repräsentieren den Abstand zwischen Zeichen und Wörtern und T_h einen horizontalen Skalierungsfaktor.
Die vollständige Transformation vom Text-Space zum Device-Space für die Ausgabe auf dem Zielgerät wird durch folgende Formel definiert:

$$T_{rm} = \begin{Bmatrix} T_{fs} \times T_h & 0 & 0 \\ 0 & T_{fs} & 0 \\ 0 & T_{rise} & 1 \end{Bmatrix} \times T_m \times CTM$$

T_{rm} ist die temporäre Rendering-Matrix und wird vom Konzept her nach dem Zeichnen jeder Glyphe neu berechnet. CTM ist die „Current Transformation Matrix“ die geräteabhängig den User-Space in den Device-Space transformiert. In der ersten Matrix wurden verschiedene Parameter die den Text-Status manipulieren genutzt, wobei T_{fs} für die Schriftgröße, T_h für die horizontale Skalierung und T_{rise} für vertikale Verschiebung des Textes auf der Zeile. Auf die am Anfang dieser Sektion bereits beschrieben Weise, lassen sich nun Textkoordinaten in Gerätekoordinaten transformieren. Zusammengefasst lässt sich die Transformation der Koordinaten durch die folgende Formel ausdrücken:

$$\begin{aligned} Koordinaten_{Devicespace} &= Koordinaten_{Userspace} \times CTM \\ &= Koordinaten_{Textspace} \times T_{rm} \end{aligned}$$

3.6.5. Lesezeichen

Die Speicherung und Handhabung von Text im PDF-Format bietet zwar keine direkten Möglichkeiten zur internen Strukturierung, jedoch bietet PDF die Möglichkeit an

sogenannte „Outline-Objekte“ zu erstellen und als Sprungmarken zu nutzen. Diese häufig auch Lesezeichen genannten Objekte werden wie Page-Objekte in einer Hierarchie außerhalb der Page-Objekte verwaltet. Die Wurzel des Outline-Tree ist wie auch beim Page-Tree das Catalog-Dictionary, in welchem die grundlegenden Strukturen für den Aufbau eines PDF-Dokumentes verwaltet werden.
Ein Lesezeichen kann entweder direkt ein Ziel oder eine Aktion kodieren. Für Ziele gibt es wiederum verschiedene Varianten, welche die Darstellung der Seite auf die verwiesen wird nach dem Sprung dorthin bestimmten. Das Listing 3.11 zeigt wie ein Outline-Objekt mit direkt kodiertem Ziel aussehen kann. Das Ziel verweist in diesem Fall auf eine Seite, deren Objekt-Referenz hier angegeben wurde und Koordinaten auf dieser Seite. Des Weiteren wird die Position im Outline-Tree durch Angabe von Eltern und Geschwisterknoten bestimmt. Ebenso wurde ein Titel angegeben, der für die Darstellung des Lesezeichens in PDF-Leseprogrammen dient.

```
<<
/Dest [7 0 R /XYZ 0 161.936528 null]
/Next 19 0 R
/Parent 5 0 R
/Prev 86 0 R
/Title (Lesezeichen1) >>
```

Listing 3.11: Outline-Objekt mit direktem Verweis

Eine andere Variante ist die Kodierung von Aktion. Aktionen gibt es wiederum in verschiedenen Varianten. Mit ihnen können beispielsweise Hyperlinks geöffnet oder das Abspielen von Mediadateien gestartet werden. Für Verweise innerhalb von PDF-Dokumenten wird die Aktion „GoTo“ genutzt. Im Listing 3.12 ist ein Beispiel, für ein Outline-Objekt, welches auf eine solche Aktion verweist. In dem Beispiel fällt auf, dass im Outline-Objekt die Werte nicht direkt kodiert wurden, sondern ausschließlich durch Objekt-Referenzen darauf verwiesen wurde. Auch in der darunter abgebildeten Aktion wurde das Ziel */D (figure.A.11)* nicht direkt sondern in diesem Fall als Name kodiert. Solche Namen werden an vom Standard vorgegebenen Stellen anstatt von Objekt-Referenzen verwendet. Folglich muss der Name im Name-Tree gesucht und die darin befindliche Objekt-Referenz auf das Ziel-Objekt aufgelöst werden. Erst dadurch gelangt man an die Referenz auf das Page-Objekt und je nach Art des Ziels auch an referenzierte Koordinaten.

```
126 0 obj <<
/Title 127 0 R
/A 124 0 R
/Parent 733 0 R
/Prev 90 0 R
/Next 130 0 R >> endobj
...
124 0 obj << /S /GoTo /D (figure.A.11) >>
```

Listing 3.12: Outline-Objekt mit Aktion

4. Verlinken der Datenformate durch URIs

Dieses Kapitel befasst sich mit der Erstellung exemplarischer Muster für URIs, mit denen PANDA Daten der im vorangegangenen Kapitel 3 vorgestellten repräsentativen Dateiformate verlinken kann. Dazu werden zunächst Design-Ziele festgelegt und ein allgemeines Muster für die URIs erstellt. Im Anschluss werden die individuellen Pfade für die Dateiformate erstellt, wobei insbesondere das Zusammenspiel zwischen URIs, der Datenstruktur und der Verarbeitung der Dateiformate beleuchtet wird. Die hier erstellten URIs werden im Rahmen des in RFC2616[8] spezifizierten „Hypertext Transfer Protocol (HTTP)“ eingesetzt und entsprechend definiert. Des Weiteren wird die Stabilität der URIs bezüglich Veränderungen der zugrundeliegenden Dokumente durch Schreibzugriffe analysiert.
Die hier erstellten URI-Muster erheben keinen Anspruch auf Vollständigkeit. Sie wurden bezogen auf beliebige Dateien beliebigen Inhaltes des jeweiligen Datenformates erstellt ohne spezifische Voraussetzungen an den Inhalt der verlinkten Daten zu stellen die nicht durch das Datenformat vorgegeben werden. Weitere Optionen für das Verlinken der Daten und die dabei auftretenden Probleme werden diskutiert.

4.1. Allgemeine Anforderungen an das Design

Bevor nach geeigneten URI-Mustern gesucht werden kann, müssen zunächst allgemeine Kriterien für das Design der URIs und der Schnittstelle von PANDA festgelegt werden. Ein häufiger Kritikpunkt[4] ist der Fakt, dass sich entgegen ihrer Definition URIs oft ändern und Hyperlinks in denen sie genutzt wurden brechen. Dafür gibt es verschiedene Gründe für deren Beseitigung das W3C eine Reihe von Richtlinien[3] zusammen gestellt hat. Die für PANDA wichtigen Punkte sind folgende:

URI-Syntax:
URIs sollten so einfach und kurz wie möglich gehalten werden. Groß- und Kleinschreibung sollte nicht gemischt werden.
Unabhängigkeit/Stabilität:
Der Definition nach gewährleisten URIs die Identifikation von Ressourcen, nicht aber den Zugriff darauf. URIs dürfen daher nicht mit Pfaden in einem Dateisystem verwechselt werden. Werden sie trotzdem in dieser Weise genutzt, so muss die Möglichkeit eines geeigneten Mappings gegeben sein. Dieses sollte so designed sein, dass die physische Datei einer Ressource ihre Adresse im Dateisystem des Webservers verändern kann, ohne dass sich die dazugehörige URI ändert. URIs sollten zudem unabhängig vom Kontext immer die selbe Ressource referenzieren. Die URI sollte keine Hinweise auf von der Serverseite genutzten Technologien oder Dateinamenerweiterungen enthalten, da

sich diese durch Einsatz neuer Technologien oder Formate ändern können. Werden diese Regeln eingehalten, bleiben URIs stabil und Hyperlinks von denen sie benutzt werden brechen nicht. Stabil bedeutet jedoch keineswegs, dass sich der Inhalt nicht ändern darf. Es widerspricht nicht dem Konzept die Ressource selbst zu modifizieren.
Cache-Kontollinformationen:
Der URI sollten wenn möglich immer für das Caching auf Client-Seite notwendige Informationen im HTTP-Header mitgeliefert werden. Diese beinhalten die Caching-Direktive, das aktuelle Datum und das Datum der letzten Modifizierung der Ressource.
Flexible Rückgabetypen (Content Negotiation):
Die Nutzung der HTTP „Content Negotiation“ erlaubt es Nutzerpräferenzen und Fähigkeiten zu berücksichtigen und die gleiche Ressource in verschiedenen Formaten zu repräsentieren. Dadurch können die bereits angesprochenen Dateinamenerweiterungen in den URIs weggelassen werden. Außerdem erlaubt dies dem Server die Erweiterung um weitere Repräsentationsformate, ohne dabei die URIs der Ressource zu verändern.

Die meisten dieser Design-Ziele werden durch die Nutzung von REST (siehe refsec:rest) bereits konsequent umgesetzt. REST bietet eine gewisse Unabhängigkeit von erweiterten Technologien, da konsequent auf HTTP als universelle Schnittstelle gesetzt wird. Die Nutzung des „Content Negotiation“ Mechanismus von HTTP gehört zu den Anforderungen die REST an ein System stellt um sich RESTful nennen zu dürfen. Auch das Senden von Cache-Kontrollinformationen gehört zu diesen Anforderungen. Dadurch wird Caching auf Seite des Clients möglich, welche den Server unter Umständen stark entlasten kann. Die nicht von REST abgedeckten Punkte wie URI-Syntax und URI-Management, werden durch Berücksichtigung bei der Erstellung der URI-Muster umgesetzt.
Für die Kodierung von Informationen in die URI stehen uns zwei Methoden zur Verfügung: die direkte und die indirekte Variante. Bei der direkten Variante können für den Zugriff benötigte Informationen direkt aus der URI entnommen werden. Dies hat den Vorteil, dass keine zusätzlichen Mechanismen für das Mapping benötigt werden und die Verarbeitung ressourcenschonend erfolgen kann.
Bei der indirekten Variante werden die Informationen dagegen verschleiert indem sie als ID in die URI kodiert werden. Die ID und die Zuordnung von Informationen muss dazu explizit festgelegt werden. Die Datenschicht mapped dann bei einem Zugriff die ID durch einen geeigneten Mechanismus auf die zugehörigen Informationen. Das Mapping benötigt Zeit und bremst Zugriffe folglich aus. Zudem benötigen die einer ID zugewiesenen Zugriffsinformationen zusätzlichen Speicherplatz auf dem Festspeicher. Um die Zugriffe nicht zu stark auszubremsen, besteht die Möglichkeit die Informationen nach einmaligem Laden vom Festspeicher bei Systemstart im Hauptspeicher zu halten und ggf. nur bei Änderung an der Konfiguration zu aktualisieren. Da der Hauptspeicher jedoch begrenzt ist, eignet sich diese Variante eher für eine bestimmte Art von Informationen die sich bei Zugriffen selten oder gar nicht ändert. Dies trifft vor allem für solche Informationen zu, welche nicht direkt die Datenstruktur betreffen und als Metainformationen für Zugriff und Verarbeitung genutzt werden. Der Vorteil dieser Variante ist, dass sich beliebig viele Informationen hinter einer ID verstecken lassen. Dies vermeidet zum einen eine ungewollte Transparenz von sensiblen Informationen und zum anderen hat es eine

positive Wirkung auf das Design-Ziel „Stabilität“. Informationen die sich hinter der ID verbergen könnten ungehindert geändert werden, ohne dass die URI davon beeinträchtigt wird. Eine Mischung beider Methoden ist möglich und sollte sinnvoll anhand der kodierten Informationen ausgewählt werden.

4.2. Allgemeines Muster

Bevor die URIs für die spezifischen Datenformate ermittelt werden, legen wir eine allgemeine Form fest, welche für alle Formate gelten soll. Bei den in PANDA genutzten URIs handelt es sich um relative Referenzen mit absoluten Pfaden gemäß der Definitionen in RFC3986[14]. Diese können durch das ergänzen der Base-URI bestehend aus dem „http“-Schema, der Autorität des Webservers und Pfadsegmenten für die Identifikation des Service in absolute URIs umgewandelt und für Zugriffe genutzt werden. In unserem Prototypen würde die Base-URI für den lokalen Host z.B. folgendermaßen aussehen:
http://localhost:8080/PANDA/rest/Data
Die Pfadsegmente */PANDA/rest/Data* sind Teil der Base-URI und dienen zur Auswahl des Services auf dem Webserver. Dieser Teil ist in PANDA frei konfigurierbar und darf nicht mit den Pfadsegmenten der relativen URIs, welche die Daten referenzieren, verwechselt werden.
Bevor ein Server Daten aus einer Datenstruktur extrahieren kann, muss dieser wissen wo sich diese befinden und wie er darauf zugreifen kann. Im Fall von einfachen Dateien ist das minimal der Dateipfad. Dateipfade wären mit entsprechendem Encoding direkt in URIs verwendbar, jedoch aus drei Gründungen ungeeignet:

(1) URI-Syntax: Die URIs werden unnötig lang und dadurch komplizierter.

(2) Unabhängigkeit/Stabilität: Die Nutzung von Pfaden in Dateisystemen in URIs sollte vermieden werden, wenn kein entsprechendes Mapping für Änderungen der Organisation im Dateisystem besteht, da die URIs sonst ungültig werden.

(3) Sicherheitsaspekt: Die Verwendung von echten Dateipfaden eines Servers macht das System unnötig transparent und kann gar ein Sicherheitsrisiko darstellen.

Darüber hinaus kann es vom Dateiformat abhängige Parameter geben, welche den Zugriff beeinflussen. Beispielsweise benötigt das CSV-Format einen Hinweis darauf, welche Trennzeichen verwendet werden sollen. Die Trennzeichen lassen sich nicht aus einem CSV-Dokument ermitteln oder allenfalls anhand ihrer Verwendung erahnen. Folglich sollte es eine Instanz geben, welche das Dokument in das System aufnimmt und dabei entsprechende Konfiguration vorab vornimmt.
Im Gegensatz zu anderen die Datenstruktur betreffenden Informationen, werden die hier genannten Informationen vom Server vorab festgelegt und lassen sich vom Client nicht beeinflussen. Aus diesem Grund eignet sich die im Abschnitt 4.1 beschriebene Variante Informationen indirekt in eine URI zu kodieren. Dazu wird auf dem Server einmalig das Mapping einer ID auf den Datensatz mit den Zugriffsinformationen festgelegt. Die ID

kann nun als eigenständiges Segment in den Pfad der URI kodiert werden und repräsentiert dort alle Zugriffsinformationen, die der Server für den Zugriff auf die Datei oder auch auf ein Datenbanksystem benötigt.
Die Syntax der ID ist nicht vorgegeben und abhängig von der Implementierung und Anforderungen. Handelt es sich um ein einfaches System, so reicht eine einfache inkrementelle Seriennummer aus. Auch der Einsatz von lokal gültigen „Unique identifier“ (UID)[1] ist denkbar. Gehen wir davon aus, dass Dateien in einem verteilten System von verschiedenen Hosts hinzugefügt werden können, so ist auch der Einsatz von „Universally Unique identifier“ (UUID)[2] möglich. Voraussetzung ist jedoch die Kompatibilität mit der URI-Syntax. Nicht erlaubte Zeichen sollten folglich vor der Verwendung für die URI durch Prozent-Encoding umgewandelt werden. Die sich daraus ergebene allgemeine Form der von PANDA genutzten URIs ist in Tabelle 4.1 als EBNF notiert. Das Nichtterminal *data-path* beschreibt den für jede Art von Datenstruktur individuell gestalteten Teil des Pfades der URI. Die daraus abgeleiteten Nichtterminale werden in den folgenden Abschnitten dieses Kapitels spezifiziert. Die Nichtterminale *unreserved*, *pct-encoded* und *sub-delims* wurden so aus der RFC3986 übernommen und sind dort detailliert beschrieben.

panda-uri	=	“/“ , *resource-id*, “/“ , *data-path*
resource-id	=	*pchar* , *{pchar}*
pchar	=	*unreserved* | *pct-encoded* | *sub-delims* | “:“ | “@“
data-path	=	*csv-path* | *xml-path* | *html-path* | *excel-path* | *word-path* | *pdf-path*

Tabelle 4.1.: EBNF für allgemeines PANDA-URI Schema

Um die URIs so einfach wie möglich zu gestalten, verzichten wir zudem auf eine Query-Komponente. Dies schränkt uns bei der Gestaltung der URIs jedoch nicht ein, da es möglich ist alle Werte nach den hier erstellten Vorgaben direkt im Pfad zu kodieren. Der Verzicht der Query-Komponente erleichtert zudem die Verarbeitung z.B. bei der Implementation eines Server-Cache. Für das Nichtterminal *data-path* und die daraus abgeleiteten Nichtterminale legen wir ferner noch eine weitere Bedingung fest:

Satz: Die Menge S sei eine geordnete Menge von Pfadsegmenten, sodass für jedes Segment s gilt $s_n \prec s_{n+1}$. Für jede ebenso geordnete Teilmenge $T \subset S$ gilt: T ist in seinem Kontext ein eindeutiger Pfad.

Diese Regel besagt, dass nicht nur der gesamte Pfad, sondern auch jeder daraus entnommene Teilpfad in seinem Kontext eindeutig sein muss. Der Hintergrund dabei: die Eindeutigkeit einer URI kann auch durch ihre Länge gegeben sein während ihre Teilpfade nicht eindeutig sind. In diesem Punkt wird nochmal deutlich, dass die Pfade in URIs nicht mit denen in Dateisystemen verwechselt werden dürfen. Nehmen wir z.B. an

[1] *http://docs.oracle.com/javase/7/docs/api/java/rmi/server/UID.html*
[2] *http://docs.oracle.com/javase/7/docs/api/java/util/UUID.html*

es existieren zwei Pfade $A = /a/b/c/d$ und $B = /a/b/c$ ausgehend vom selben Kontext. Beide sind nach dem URI-Standard gültige Pfade einer absoluten Pfad-Referenz. Entnehmen wir jedoch den Teilpfad $/a/b/c$ aus A, so kann dieser nur eindeutig sein, wenn er nicht nur syntaktisch sondern auch semantisch zu B äquivalent ist und folglich die selbe Ressource referenziert. Dieses Problem tritt insbesondere dann auf, wenn wir verschiedene Typen von URIs für das gleiche Datenformat definieren.
Durch die Einhaltung dieser Regel nähern wir uns der semantischen Bedeutung von Pfaden in Dateisystemen an, bei denen solche Mehrdeutigkeiten generell nicht möglich sind. Dies bietet zwei entscheidende Vorteile. Zum einen besteht die Möglichkeit Teilpfade in weniger spezifischen Bulk-URIs für Massenabfragen zu verwenden und zum anderen wird die Verarbeitung durch die Eindeutigkeit der Pfadsegmente in Unabhängigkeit von den folgenden Segmenten erleichtert. Dies erlaubt z.B. die Verwendung der Pfadsegmente als Schlüssel in Hash-Tabellen, ohne dabei Mehrdeutigkeiten berücksichtigen zu müssen.
Um die Redundant in den Grammatiken für die Pfade der verschiedenen Datenformate zu minimieren, werden in Tabelle 4.2 häufig verwendete Bausteine aufgeführt. Ihre Bedeutung wird in den jeweiligen Abschnitten für die Datenformate erläutert.

spreadsheet-reference	=	“*“ | *cell-reference* | *column-reference* | *line-reference* | *table-reference*
cell-reference	=	*column-index* , *line-index*
column-reference	=	*column-index* , “*“
line-reference	=	“*“ , *line-index*
table-reference	=	*cell-reference* , *range-delimiter* , *cell-reference*
column-index	=	*upalpha* , {*upalpha*}
line-index	=	*number-nz*
index-range	=	*number-nz* , *range-delimiter* , *number-nz*
number-nz	=	*digit* , {*digit* | “0“}
range-delimiter	=	“:“ | “-“
upalpha	=	“A“ | “B“ | ... | “Z“
digit	=	“1“ | “2“ | “3“ | “4“ | “5“ | “6“ | “7“ | “8“ | “9“

Tabelle 4.2.: EBNF für häufig genutzte Bausteine in den Pfad-Schemata von in PANDA implementierten Datenformaten

4.3. Extensible Markup Language (XML/HTML)

Für die Verarbeitung und das Verlinken von Daten aus XML/HTML-Dateien die in den Abschnitten 3.2 bzw. 3.3 beschrieben wurden, benötigen PANDA zunächst die Zugriffsinformationen. Dabei kann es sich beispielsweise um einen Dateipfad in einem Dateisystem oder auch eine URL für online gelagerte Ressourcen handeln. Des Weiteren wird der Typ der Datei benötigt, um ein geeignetes Verfahren für die Verarbeitung auszuwählen. Diese Informationen lassen sich auf indirekte Weise wie im allgemeinen Muster 4.2 beschrieben in das Segment *resource-id* kodieren.

Der Zugriff auf die Daten gestaltet sich bei XML/HTML anders als bei den anderen untersuchten Formaten. Im Gegensatz zu diesen ist die Baumstruktur von XML/HTML generisch und kann somit beliebig viele Informationen zur Referenzierung enthalten. Eine CSV-Datei ließe sich beispielsweise immer als XML-Datei mit 3 Ebenen inklusive Wurzel darstellen. Diese repräsentieren die 2 Dimensionen des Formates. Die Ebenen unter dem Wurzel-Knoten wachsen jeweils ausschließlich in die Breite wenn Zeilen oder Spalten hinzugefügt werden. Die XML-Datei dagegen kann beliebig in die Tiefe wachsen. Folglich wird auch eine dynamische Pfadstruktur für die Referenzierung benötigt.
Wie in der Analyse der Datenstruktur beschrieben, wurde die Sprache XPath[16] zum Zweck der Referenzierung von Knoten, Attributen oder Teilbäumen im XML-Format entwickelt. Diese Syntax der Sprache ist der von Pfaden in URIs recht ähnlich, erlaubt jedoch Zeichen die dort nicht verwendet werden dürfen. Aus diesem Grund sollten XPaht-Ausdrücke generell mit dem in RFC3986 beschriebenen Prozent-Encoding versehen werden. Es gibt nun zwei Möglichkeiten einen XPath-Ausdruck in den Pfad der URI zu kodieren:

(1) Kodiere den prozent-encodeten XPath-Ausdruck als einzelnes Pfadsegment in die URI.

(2) Bette den XPath-Ausdruck als Teil des Pfades in die URI ein.

Die erste Variante sieht den XPath-Ausdruck als kompakten String der als einzelnes Element im Pfad gesehen werden kann. Diese Variante ist nicht mit der Standardkonfiguration von Web-Servern wie Apache nicht möglich, da das als „%2F“ encodete Zeichen „/“ aus Sicherheitsgründen nicht erlaubt ist. Um eine zusätzliche Konfiguration des Web-Servers zu vermeiden, kann der XPath-Ausdruck jedoch auch als Teil des Pfades in die URI eingebettet werden. Dabei werden die Slashes nicht encodet. Der XPath-Ausdruck muss dann auf dem Server aus mehreren Segmenten der URI zusammengesetzt werden. Für absolute Pfad-Referenzen erlaubt der URI-Standard auch leere Pfadsegmente unter der Voraussetzung, dass es sich nicht um das erste Segment handelt. In diesem ist bereits die ID für die Zugriffsinformationen kodiert, so stellt auch die Notation von relativen XPath-Ausdrücken kein Problem dar.
Um Dokumente im HTML-Format auf die gleiche Weise verarbeiten zu können, muss dieses zunächst von nicht XML-konformen Konstrukten bereinigt werden. Für diesen Zweck gibt es Programme wie „JTidy“[3], welche HTML-Dokumente an den XHTML-Standard anpassen. XHTML–Dokumente sind selbst gültige XML-Dokumente und können auf die gleiche Weise verarbeitet werden.
Das sich aus den hier aufgeführten Spezifikationen abgeleitete Schema ist als EBNF in Tabelle 4.3 abgebildet.

[3] *http://jtidy.sourceforge.net/*

xml-path	=	*XPath*
html-path	=	*XPath*

Tabelle 4.3.: EBNF für PANDA Pfad-Schema für das Verlinken von Daten aus XML/HTML-Dateien, die Grammatik der XPaht-Ausdrücke wird entsprechend dem Standard[16] definiert. Beachte: zusätzlich Prozent-Encoding notwendig.

Das XML/HTML-Format erlaubt keine garantierte Stabilität der URIs. Problematisch ist dabei die Möglichkeit Namen von Knoten mehrfach zwischen Geschwisterknoten zu vergeben. Die genaue Referenz eines Knoten ergibt sich in diesem Fall durch seine Position bezüglich der gleichnamigen Geschwisterknoten. Dies erlaubt die Erstellung von Deep-Links, welche bis auf die Granulatität von einzelnen Textknoten Daten aus XML-Dokumenten referenzieren können. Wird jedoch einer der gleichnamigen Knoten überschrieben, entfernt oder ein neuer Knoten mit diesem Namen hinzugefügt, so muss der Index für alle folgenden Knoten inkrementiert bzw. dekrementiert werden. Eine Stabilität der URIs für XML-Dokumente ist daher nur gewährleistet wenn eine der folgenden Bedingungen erfüllt ist:

(1) Das System setzt unveränderbare Daten voraus.

(2) Das System erlaubt keine Geschwisterknoten mit gleichem Namen.

(3) Das System erlaubt ausschließlich das Überschreiben oder entfernen des letzten gleichnamigen Knoten unter Geschwisterknoten. Neue Knoten müssen hinter dem letzten Knoten des gleichen Namens hinzugefügt werden.

Eine grobe Analyse der für dieses Projekt exemplarisch verwendeten offenen Daten hat jedoch gezeigt, dass die mehrfache Verwendung von Knotennamen unter Geschwisterknoten nicht der Praxis entspricht.
Die Effizienz und Geschwindigkeit der Verarbeitung von XML hängt maßgeblich vom eingesetzten XML-Parser und der Implementation von XPath ab. Da das XML-Format als Grundlage für viele weitere Formate dient, wurde die Verarbeitung von XML-Dokumenten im Kapitel 5 ausführlich inklusive Benchmarks analysiert.

4.4. Comma Separated Values (CSV)

Für die Verarbeitung und das Verlinken von Daten einer CSV-Datei in dem im Abschnitt 3.4 beschriebene CSV-Format werden zunächst verschiedene Informationen für den Zugriff und die Verarbeitung benötigt. PANDA benötigt zunächst eine Möglichkeit auf die Datei zuzugreifen z.B. über einen Dateipfad im Dateisystem. Darüber hinaus muss der genaue Typ der Datei bekannt sein, um die Art der Verarbeitung der Ressource zu bestimmen. Bei CSV-Dateien sollte außerdem ein Trennzeichen vorab festgelegt werden,

da dieses nicht vom Standard vorgegeben wird. Diese Informationen lassen sich auf indirekte Weise wie im allgemeinen Muster 4.2 beschrieben in das Segment *resource-id* kodieren.
Die Referenzierung eines einzelnen Datums in einer CSV-Datei benötigt zwei Informationen: den Zeilenindex und den Spaltenindex. Beide ließen sich als Integer in separate Pfadsegmente kodieren. Um die URIs etwas einheitlicher zu gestalten, benutzen wir jedoch die in Tabellenkalkulations-Programmen wie Microsoft Excel üblichen Zell-Referenzen. Diese bestehen aus einem oder mehreren Großbuchstaben, welche die Spalte referenziert, gefolgt von einem Integer der meist 1-basiert ist und den Index der referenzierten Zeile angibt. Die Syntax URI wird dadurch einfacher, jedoch sollte bei der Verarbeitung beachtet werden, dass ein entsprechendes Pfadsegment nun zwei Informationen für die Referenzierung enthält die ausgewertet werden müssen.
Für Massenabfragen bietet es sich an, die von Tabellenkalkulation bekannten Notationen beizubehalten und ggf. zu erweitern. Diese Notation enthält zwei Zell-Referenzen getrennt mit einem Trennzeichen. In der Regel wird dazu das Zeichen „:“ verwendet. Die Indizes für Spalten und Zeilen bilden jeweils Start- und Endwerte für eine Tabellen-Struktur die durch sie referenziert wird. Das Trennzeichen ist variabel. Es muss die Zell-Referenzen eindeutig trennen und darf dabei nicht gegen die Syntax von Pfadsegmenten in URIs verstoßen. Das Zeichen „:“ gehört prinzipiell zu den reservierten Zeichen, ist aber im Kontext von Pfadsegmenten nutzbar wenn es sich nicht um das erste Segment einer relativen Pfad-Referenz handelt. Alternativ bietet sich die Nutzung des nicht reservierten Zeichens „-“ an, da dieses häufig für die Angabe von Bereichen verwendet wird.
Wir haben nun die Möglichkeit einzelne Zellen und auch gezielt Bereiche eines CSV-Dokumentes zu referenzieren. Es fehlt die Möglichkeit einer teilweise oder gar nicht eingeschränkten Referenzierung von Zellen. Für diese Aufgabe wurde das Zeichen „*“ verwendet, welches hier als Platzhalter für jeden gültigen Index steht. Ein einzelnes „*“ soll für alle gültigen Indizes stehen und folglich alle Zellen des Dokumentes referenzieren. Spalten und Zeilen sind durch das Ersetzen des Spalten-Index bzw. Zeilen-Index durch „*“ zu referenzieren. Einen Einsatz in den Tabellen-Abfragen erlauben wir zunächst nicht, da ein Anwendungszweck dafür fraglich scheint.
Das sich aus den hier aufgeführten Spezifikationen abgeleitete Schema ist als EBNF in Tabelle 4.4 abgebildet.

csv-path	=	*spreadsheet-reference*

Tabelle 4.4.: EBNF für PANDA Pfad-Schema für das Verlinken von Daten aus CSV-Dateien, nicht aufgeführte Nichtterimale wurden in Grammatik 4.2 definiert

Die Stabilität der URIs ist nicht zwangsläufig gewährleistet, da das Löschen oder Hinzufügen Records die Indizes aller folgenden Records verändert. Innerhalb eines Records kann durch das Hinzufügen oder Entfernen eines einzelnen Spaltenwertes der gleiche Effekt für den folgenden Werte dieses Records auftreten. Eine Stabilität der URIs lässt

sich folglich nur garantieren wenn eine der folgenden Bedingungen erfüllt ist:

(1) Das System setzt unveränderbare Daten voraus.

(2) Das System erlaubt ausschließlich das Hinzufügen neuer Records hinter dem letzten Record des Dokumentes bzw. neuer Spaltenwerte hinter dem letzten Spaltenwert eines Records. Das Entfernen ganzer Records oder Spaltenwerte ist verboten. Ggf. kann ein Spaltenwert oder alle Spaltenwerte eines Records durch das leere Zeichen ersetzt werden.

CSV-Dateien lassen sich im einfachsten Fall direkt durch Trennung anhand der Trennzeichen verarbeiten. Folgt man der Empfehlung von RFC4180 wird das Format etwas komplexer, weshalb sich der Einsatz eines Parsers für die Verarbeitung anbietet. Durch den einfachen Aufbau des CSV-Formates, ist das Parsing und Zugriffe auf die Daten vergleichsweise schnell. Eine Möglichkeit den Zugriff noch schneller zu gestalten ist, ausschließlich die angefragten Records bzw. Zeilen zu parsen. Davon profitieren Zugriffe umso mehr, je spezieller sie sind und desto größer die CSV-Datei und damit die Anzahl nicht geparster Records ist.

4.5. Microsoft Office

In diesem Abschnitt werden Pfade für die Extraktion von Daten aus Dateien basierend auf WordprocessingML und SpreadsheetML definiert. Beide Sprachen sind Teil des OOXML-Standard und basieren auf der dort definierten und in Abschnitt 3.5.1 beschriebenen Paketstruktur.
Für die Verarbeitung und das Verlinken von Daten aus einer solchen Datei wird für beide Formate zunächst eine Zugriffsinformation wie z.B. der Dateipfad im Dateisystem benötigt. Darüber wird der genaue Typ der Datei benötigt, um die Art der Verarbeitung genauer zu bestimmen. Diese Informationen werden auf indirekte Weise wie im allgemeinen Muster 4.2 beschrieben in das Segment *resource-id* kodiert.
Die Ausführungen in dieser Arbeit basieren auf den neueren OOXML-Formaten, die grundlegende Gliederung der Daten wurde weitestgehend aus dem Office 97 Formaten übernommen. Die hier erstellten Pfadmuster sollten folglich hinreichend sein, um auch für die Extraktion aus diesen Formaten genutzt werden zu können.
Die grundlegenden Bausteine aus denen Dokumente nach OOXML-Standard aufgebaut sind ist das XML-Format. Die Effizienz und Geschwindigkeit der Verarbeitung hängt daher maßgeblich vom eingesetzten XML-Parser ab. Da XML weitläufig bekannt und verbreitet ist, sind individuelle Lösungen für die Verarbeitung vergleichsweise einfach zu implementieren. Die gezielte Extraktion von Knoten aus den XML-Dokumenten kann auf manuelle Weise einfach umgesetzt werden. Auch eine Automatisierung ist möglich. Dazu werden die in den folgenden Abschnitten beschriebenen und in die URI kodierten Informationen auf einen XPath-Ausdruck gemapped. Da die Effizienz der Verarbeitung von XML sowohl für das XML-Format selbst, als auch für viele moderne Office-Formate von entscheidender Wichtigkeit ist, wurde die Verarbeitung von XML-Dokumenten im Kapitel 5 ausführlich inklusive Benchmarks analysiert.

4.5.1. WordprocessingML

Für die Verlinkung von Word-Dateien in dem im Abschnitt 3.5.2 beschriebenen WordprocessingML-Format werden verschiedene Typen von Pfaden definiert. Gemäß der Bestimmungen aus Abschnitt 4.2 wird deshalb ein zusätzliches Segment zur Unterscheidung dieser Typen eingeführt. Für die Extraktion von einfachem Text benutzen wir den Bezeichner “text“ und für die Extraktion von Tabellen “tables“ und für Bilder den Bezeichner “pictures“.
Bevor die Daten extrahiert werden können, muss die Paketstruktur gemäß den Beschreibungen in Abschnitt 3.5.1 aufgelöst werden. Dazu sind keine weiteren Informationen aus der URI notwendig. Die zentrale “document.xml“ enthält sämtliche Paragraphen des Textes und die Tabellen. Ziel der Text-Extraktion sind die *<w:p>*-Knoten. Eine Extraktion auf Ebene der als *<w:t>*-Knoten repräsentierten Text-Runs ist für unseren Zweck nicht notwendig. Daher werden diese Knoten bzw. die darin enthaltenen Text-Knoten von PANDA zu einem String zusammengefasst. Die Paragraphen besitzen keine spezifischen Identifikationsmerkmale abgesehen von ihrem Auftreten im Dokument. Sie sind der Reihe nach sortiert und können deshalb mittels Index referenziert werden.
Für die Abfrage größerer Textabschnitte können zwei Indizes getrennt durch ein Trennzeichen genutzt werden. Als Trennzeichen werden die Zeichen „:“ und „-“ verwendet. Dabei ist zu beachten, dass das Zeichen „:“ zwar prinzipiell zu den in URIs reservierten Zeichen gehört, eine Verwendung im Pfad jedoch gemäß Spezifikation erlaubt ist, sofern es sich nicht um das erste Segment einer relativen Pfad-Referenz handelt. Für die Abfrage des gesamten Textes wird statt dem Index das Zeichen “*“ verwendet.
Eine Referenzierung nach Seiten ist nicht direkt möglich. Wie in der Beschreibung der Datenstrukturen erwähnt, müssten dazu sämtliche Text-Runs nach Knoten die eine neue Seite signalisieren gesucht werden. Zugunsten der Effizienz wird darauf verzichtet.
Anders als im SpreadsheetML-Format besitzen Tabellen in WordprocessingML keine extra Bezeichner die für eine Referenzierung genutzt werden können. Sie werden jeweils in einem *w:tbl*-Knoten gespeichert und können ähnlich wie Paragraphen der Reihenfolge nach durch einen Index referenziert werden. Die Notation ist dabei ebenfalls die selbe. Ebenso werden die Zeilen-Knoten *w:tr* und die Zell-Knoten *w:tc* durch Indizes ihrer Reihenfolge nach referenziert. Als Notation für die Referenzierung wird die von Excel bekannte Form für die Referenzierung Zellen und Tabellen genutzt. Für die Referenzierung von Spalten und Zeilen wird die Notation der Zell-Referenz erweitert und jeweils Spalten-Index oder Zeilen-Index durch das Zeichen „*“ ersetzt. Eine genauere Beschreibung der Notation befindet sich im Abschnitt 4.4. Die Zell-Knoten selbst enthalten wiederum die bereits bekannten Paragraphen. Folglich werden sie genauso behandelt und auf gleiche Weise in die URI kodiert.
Neben der Extraktion von Text wurde auch die Extraktion von Bildern umgesetzt. Im Gegensatz zu Excel, werden diese nicht in einem Unterpaket verwaltet sondern direkt über ihre Relationship-ID eingebettet. Die ID ist nach außen jedoch nicht transparent und daher nur bedingt für die Referenzierung in einer URI verwendbar. Stattdessen werden die Bilder ihrer Reihenfolge nach durch einen Index referenziert. Da pro URI nur ein Bild abgefragt werden soll, kann dieser nur spezifische Werte annehmen.

Das sich aus den hier aufgeführten Spezifikationen abgeleitete Schema ist als EBNF in Tabelle 4.5 abgebildet.

word-path	=	*word-text* | *word-tables* | *word-image*
word-text	=	“text“ , “/“ , *word-paragraph*
word-tables	=	“tables“ , “/“ , *word-table* , “/“ , *spreadsheet-reference* , “/“ , *word-paragraph*
word-image	=	“pictures“ , “/“ , *image-id*
word-paragraph	=	*number-nz* | *index-range* | “*“
word-table	=	*number-nz* | “*“
word-image-id	=	*number-nz*

Tabelle 4.5.: EBNF für PANDA Pfad-Schema für das Verlinken von Daten aus Word-Dateien, nicht aufgeführte Nichtterimale wurden in Grammatik 4.2 definiert

Bei der Referenzierung von Dokumenten die WordprocessingML nutzen treten die gleichen Probleme wie auch der zugrundeliegenden XML auf. Durch die Verwendung von Indizes können Schreiboperationen die Referenzierung der Paragraphen verändern. Alternativ könnte deshlab die angesprochene jedoch weniger effiziente Lösung der Referenzierung nach Seiten durch gezielte Suche aller Seitenumbrücke genutzt werden. In diesem Fall brechen Links erst dann, wenn der letzte Paragraph einer Seite entfernt wird. Dennoch kann eine Änderung der Referenzierung nach dem Einfügen von Text notwendig sein. Wollen wir beispielsweise den Inhalt von zwei bestimmten Seiten verlinken, so kann es vorkommen, dass beim Einfügen neuen Textes Inhalt der zweiten Seite auf eine dritte Seite verschoben wird. In diesem Fall müssten die Links ohnehin angepasst werden. Das Problem kann jedoch bezogen auf die Extraktion von Text als weniger kritisch eingestuft werden, da diese einen anderen Charakter besitzt als beispielsweise Extraktion aus Formaten der Tabellenkalkulationen. Diese werden häufig für Berechnungen und Statistiken eingesetzt und erfordern daher eine maximale Zuverlässigkeit. Bei der Extraktion von Text werden dagegen eher ganze Textabschnitte angesprochen als einzelne Zeilen oder Paragraphen. Es ist daher weniger wahrscheinlich, dass ein Link durch die Veränderung des Dokumentes durch Schreiboperationen bricht.
Bei der Verarbeitung des dem Office97-Standard entsprechenden Word-Formates gibt es einen entscheidenden Unterschied zum neueren OOXML-basierten WordprocessingML-Format. Es werden dort ausschließlich Paragraphen für die Gliederung genutzt. So werden selbst Leerzeilen als Paragraphen geführt. Auch Tabellen werden nicht als gesonderte Struktur gespeichert. Stattdessen werden Paragraphen die einer Tabelle angehören mit entsprechenden Attributen ausgestattet, die ihre Zugehörigkeit und Position in der Tabelle signalisieren. Filtert man die Leerzeilen und Tabellen heraus, so entstehen bei der Referenzierung der eigentlichen Text-Paragraphen im Index Lücken. Diese ließen sich nur herausfiltern, wenn bei der Verarbeitung zuvor umständlich eine Liste aller Text-Paragraphen erstellt wird. Darauf wurde jedoch verzichtet, da das Office97-Format in PANDA nur zu vergleichszwecken implementiert wurde.

4.5.2. SpreadsheetML

Für die Verlinkung von Excel-Dateien in dem im Abschnitt 3.5.3 beschriebene SpreadsheetML-Format werden verschiedene Pfad-Typen definiert. Gemäß der Bestimmungen aus Abschnitt 4.2 wird deshalb ein zusätzliches Segment zur Unterscheidung dieser Typen eingeführt. Für die Extraktion von Tabellen nutzen wir den Bezeichner "tables" und für die Extraktion von Bildern den Bezeichner "pictures".
Bevor Daten aus einer Excel-Datei extrahiert werden können, muss die Paketstruktur gemäß den Beschreibungen in Abschnitt 3.5.1 aufgelöst werden. Dazu sind keine weiteren Informationen aus der URI notwendig. Die zentrale "workbook.xml" listet nun alle in der Datei enthaltenen Tabellen. Diese lassen sich mittels der intern zugeordneten Relationship-ID oder über den eindeutigen Tabellennamen referenzieren. Die Syntax des Tabellennamen (siehe Abschnitt 3.1) entspricht den in Windows-Dateisystemen üblichen Anforderungen, kann jedoch Unicode-Zeichen enthalten. Aus diesem Grund sollte er ggf. mit von RFC3986 definierten Prozent-Encoding kodiert bzw. bei Verarbeitung dekodiert werden. Alternativ kann mit dem Zeichen "*" aus allen vorhandenen Tabellen extrahiert werden. Diese Notation ist möglich, da sie sich nicht mit der Syntax von Tabellennamen überschneidet.
Aus den referenzierten Tabellen können jetzt gezielt Daten extrahiert werden. SpreadsheetML ist wie in OOXML üblich zeilenorientiert. So wird zunächst ein Zeilen-Index für die Auswertung benötigt. Die *<row>*-Knoten besitzen jeweils ein Attribut *r*, in dem ihr Zeilen-Index vermerkt wird. Die Auswertung des Attributs ist wichtig, da der Index des Knotens nicht dem der Zeile die er repräsentiert entsprechen muss, beispielsweise beim Auftreten leerer Zeilen.
Eine Ebene tiefer befinden sich die *<c>*-Knoten, die jeweils eine Zelle im Dokument repräsentieren. Diese besitzen ebenfalls ein Attribut *r*, in welchem die vollständige Zell-Referenz notiert ist. Die Zell-Referenz besteht aus dem als Großbuchstaben angegebenen Spalten-Index und dem Zeilen-Index. Für eine Auswertung des Spalten-Index muss diese Zell-Referenz aufgespalten werden.
Nach der Referenzierung der Zelle, kann der Wert aus dem *<v>*-Knoten entnommen werden. Dabei sind die verschiedenen Zell-Typen insbesondere Shared-Strings und Datumsangaben zu beachten (beschrieben in Abschnitt 3.5.3).
Für die Notation zur Abfrage von Teilen einer Tabelle wird die von Excel bekannte Form für die Referenzierung Zellen und Tabellen genutzt. Für die Referenzierung von Spalten und Zeilen wird die Notation der Zell-Referenz erweitert und jeweils Spalten-Index oder Zeilen-Index durch das Zeichen „*“ ersetzt. Eine genauere Beschreibung der Notation befindet sich im Abschnitt 4.4.
Neben der Extraktion von Text wurde auch die Extraktion von Bildern umgesetzt, auch wenn diese in Excel eine eher untergeordnete Rolle spielen. Sie werden wie auch Diagramme im Unterpaket "drawings" verwaltet und mittels nach interner Relationship-ID referenziert. Eine Zuordnung zu einer bestimmten Tabelle wäre grundsätzlich möglich, um diesen Teil jedoch einfach zu halten, wurde darauf verzichtet. Stattdessen wird ein Bild aus der Liste aller Bilder anhand eines Index extrahiert. Da pro URI nur ein Bild abgefragt werden soll, kann dieser nur spezifische Werte annehmen.

Auf Extraktion weiterer Paket-Elemente wie Diagramme wurde verzichtet, da diese in der Regel die bereits extrahierbaren Tabellen-Daten lediglich visuell aufbereiten. Grundsätzlich wäre aber auch die Extraktion dieser Teile eines SpreadsheetML-Dokumentes möglich.
Das sich aus den hier aufgeführten Spezifikationen abgeleitete Schema ist als EBNF in Tabelle 4.6 abgebildet.

excel-path	=	*excel-table* | *excel-image*
excel-table	=	"tables" , "/" , *sheet-name* | "*" , "/" , *spreadsheet-reference*
excel-image	=	"pictures" , "/" , *excel-image-id*
excel-image-id	=	*number-nz*

Tabelle 4.6.: EBNF für PANDA Pfad-Schema für das Verlinken von Daten aus Excel-Dateien, *sheet-name* wurde in der Grammatik 3.1 definiert, sonstige nicht aufgeführte Nichtterimale wurden in Grammatik 4.2 definiert

Die URIs für SpreadsheetML sind bezüglich Schreiboperationen vergleichsweise stabil, da jede Zeile und jede Spalte ein bezeichnendes Attribut im zugrundeliegenden XML-Dokument besitzt. URIs werden dennoch ungültig, wenn die Namen der Tabellen geändert werden oder ganze Zeilen bzw. Spalten in das Dokument eingefügt werden. Anstatt der Tabellennamen könnten auch Indizes verwendet werden. Diese sind wiederum unabhängig von den Tabellennamen, ändern sich jedoch bei Einfügen neuer Tabellen bzw. bei Änderung der Reihenfolge der Tabellen. Ein Brechen der Links bei Einfügen/Entfernen von Zeilen/Spalten kann in keinem Fall ausgeschlossen werden, weil dadurch die Identifikationsmerkmale direkt überschrieben werden.
Bezüglich Referenzierung sind die dem Office97-Standard entsprechenden Excel-Dokumente äquivalent zu verarbeiten wie das SpreadsheetML-Format. Die URIs ändern sich folglich bei einer Konvertierung in das eine oder andere Format nicht. Das ältere Excel-Format wurde zu Vergleichszwecken in PANDA implementiert.

4.6. Portable Document Format (PDF)

Für die Verarbeitung und das Verlinken von Daten aus einer PDF-Datei in dem im Abschnitt 3.6 beschriebene PDF-Format, benötigt PANDA zunächst Informationen bezüglich Zugriff und Verarbeitung. Als Zugriffsinformation kann beispielsweise ein Pfad im Dateisystem dienen. Zusätzlich wird der genaue Typ der Datei benötigt um die Art der Verarbeitung zu bestimmen. Diese Informationen werden auf indirekte Weise wie im allgemeinen Muster 4.2 beschrieben in das Segment *resource-id* kodiert.
Da wir für PDF mehrere Typen von Pfaden definieren, fügen wir gemäß der Bestimmungen aus Abschnitt 4.2 ein zusätzliches Segment zur Unterscheidung dieser Typen ein. Für die Textextraktion nutzen wir den Bezeichner "text" und für die Extraktion von Bildern den Bezeichner "pictures".
Die Extraktion von Text ist in PDF nicht so trivial wie beispielsweise in Word. Wie in

Abschnitt 3.6.4 beschrieben, folgt der Text innerhalb eines Page-Objektes keiner festgelegten Struktur. Stattdessen ist alles auf dessen Darstellung ausgerichtet. Es gibt keine Paragraphen und nicht einmal eine direkt erkennbare Trennung in Zeilen. Eine Extraktion nach Koordinaten wäre die für PDF natürlichste Variante. Für die Extraktion von Text ausbeliebigen PDF-Dokumenten ist dieser Variante jedoch nicht zu gebrauchen.
Dennoch gibt es eine Möglichkeit den Text trotz der fehlenden Struktur in Zeilen zu gliedern. Dazu betrachtet man die y-Koordinaten der Glyphen und ihre Größe in Abhängigkeit von der Font der sie angehören. Durch Berechnung von Überschneidungen werden die Glyphen dann einer Zeile zugeordnet. Der Nachteil dieser Methode ist, dass der Text nicht einfach direkt extrahiert werden kann. Bei der Extraktion müssen für die Ermittlung der Koordinaten praktisch die selben Berechnungen durchgeführt werden, die auch ein Leseprogramm zur Darstellung eines PDF-Dokumentes durchführen müsste. Folglich ist die Extraktion im Vergleich zu anderen Formaten wenig effizient. Dazu kommt eine nicht genau vorhersagbare Ungenauigkeit bei der Zuordnung.
Für die Extraktion von Text kodieren wir nun den Index der Seite und der Zeile. Da die Extraktion einzelner Zeilen wenig Sinn ergibt, fügen wir die Möglichkeit von Massenabfragen durch Angabe eines Index-Bereiches hinzu. Diese besteht aus zwei Indizes getrennt durch ein Trennzeichen. Als Trennzeichen verwenden wir die Zeichen „:“ und „-“. Das Zeichen „:“ gehört prinzipiell zu den reservierten Zeichen, ist aber im Kontext von Pfadsegmenten nutzbar wenn es sich nicht um das erste Segment einer relativen Pfad-Referenz handelt. Für die Abfrage des gesamten Textes einer Seite kann statt des Zeilen-Index das Zeichen “*“ verwendet werden. Der Text ließe sich in diesem Fall auch durch Berechnung der Koordinaten extrahieren, sofern vom genutzten Framework oder der Implementierung unterstützt.
Eine Extraktion von Tabellen in einer ihrer Struktur entsprechenden Form ist durch die fehlende Struktur besonders schwierig. Tabellen werden nicht als solche in PDF definiert. Es handelt sich um gewöhnlichen Text der durch bestimmte Positionen der Glyphen für den Betrachter die Darstellung eines Objektes, wie wir es als Tabelle interpretieren, entstehen lässt. Soll eine Tabelle extrahiert werden, so muss diese zunächst als eine solche erkannt werden. Die Unterscheidung von normalem Text in beliebigen PDF-Dokumenten ist auch aufgrund verschiedener Arten wie Tabellen dargestellt werden können praktisch nicht möglich. Tabellen werden aus diesem Grund in PANDA nicht von anderem Text unterschieden.
Neben der Extraktion von Text wurde auch die Extraktion von Bildern umgesetzt. Diese entweder als eigenständige Objekte oder seltener auch “Inline“ gespeichert. Die Inline Variante wurde hier außen vor gelassen. Die Objekte mit den Bildern besitzen wie gehabt eine Objekt-Referenz werden aber indirekt über einen internen Namen referenziert und so in den Content-Stream eines Page-Objektes eingebunden. Da Name und Objekt-Referenz von außen nicht erkenntlich sind, würde es schwer fallen ein Bild dadurch zu referenzieren. Sinnvoller erscheint es also einfache Indizes zu nutzen und sowohl die Seite auf der das Bild erscheint, als auch die Nummer des Bildes bezüglich den auf der Seite auftauchenden Bildern in die URI zu kodieren. Auf diese Weise können URIs einfach anhand dessen was ein PDF-Leseprogramm anzeigt erstellt und die Bilder verlinkt werden. PDF ist in diesem Fall ausnahmsweise dem sonst besser strukturierten WordprocessingML überlegen, da

die Möglichkeit besteht Bilder gezielt von bestimmten Seiten zu extrahieren.
Das sich aus den hier aufgeführten Spezifikationen abgeleitete Schema ist als EBNF in Tabelle 4.7 abgebildet.

pdf-path	=	*pdf-text* | *pdf-image*
pdf-text	=	"text" , "/" , *pdf-pages* , "/" , *pdf-line*
pdf-image	=	"pictures" , "/" , *pdf-page*, "/" , *pdf-image-id*
pdf-pages	=	*number-nz* | *index-range* | "*"
pdf-page	=	*number-nz*
pdf-line	=	*number-nz* | *index-range* | "*"
pdf-image-id	=	*number-nz*

Tabelle 4.7.: EBNF für PANDA Pfad-Schema für das Verlinken von Daten aus PDF-Dateien, sonstige nicht aufgeführte Nichtterimale wurden in Grammatik 4.2 definiert

Die im Abschnitt 3.6.5 beschriebenen Lesezeichen wurden bei der Implementierung wegen diverser Schwierigkeiten nicht berücksichtigt. Die als Lesezeichen bezeichneten Outline-Objekte sind mehr als das. Die Referenzierung innerhalb eines PDF-Dokumentes ist nur ein kleiner Teil ihres eigentlichen Aufgabenbereiches. Folglich ist nur ein bestimmter Teil dieser Objekte für eine Extraktion von Text überhaupt von Relevanz. Das Hauptproblem besteht jedoch darin, ein Outline-Objekt gezielt zu referenzieren. Die Outline-Objekte werden im Outline-Tree organisiert. Die Struktur ist dabei jedoch vollständig dem Schreibprogramm überlassen. Im einfachsten Fall könnten alle Outline-Objekte Kinder der Wurzel des Outline-Trees sein. Da praktisch jeder Verweis in einem PDF-Dokument ein Outline-Objekt ist, wird es ohne Vorwissen über die Struktur des Outline-Trees schwierig bestimmte Outline-Objekte zu referenzieren. Eine Implementierung müsste folglich Anforderungen an die Strukturierung des Outline-Trees durch das Schreibprogramm stellen, die das PDF-Format selbst nicht vorgibt.
Alternativ besteht die Möglichkeit die Struktur des Outline-Trees vollkommen zu ignorieren durch folgende Vorgehensweise: (1) Alle Outline-Objekte z.B. mittels Tiefensuche ermitteln, (2) Outline-Objekte nach internen Verweisen filtern und (3) Zugriff über einen Index realisieren.
Auf diese Weise könnten die Outline-Objekte auch in einer URI referenziert werden. Durch dieses Verfahren werden jedoch alle Outline-Objekte vermischt. Es ist folglich nicht mehr erkennbar, ob es sich um die Referenz auf die Überschrift eines Kapitels, eines Abschnittes oder sonstiges handelt. Eine sinnvolle Anwendung in der Praxis ist daher fraglich. Dennoch wäre es mit dem Outline-Objekt nun möglich, Text ausgehend von der durch ihm referenzierten Seite und Koordinaten zu extrahieren. Da die Outline-Objekte jeweils ihre Position im Outline-Tree speichern, ist es zumindest möglich das nächste Outline-Objekt auf der selben Ebene der Hierarchie zu ermitteln und den Text wie gewollt z.B. von einem Kapitel zu extrahieren. Aufgrund der hier genannten Schwierigkeiten und des hohen Aufwands, wurde im Rahmen dieser Arbeit auf eine Implementierung basierend auf Lesezeichen verzichtet. Für spezifische PDF-Dokumente sollten die

hier aufgeführten Beschreibungen dennoch einen Ansatzpunkt für die Extraktion mittels solcher Lesezeichen bieten.
Die Stabilität der URIs verhält sich ähnlich wie bei Word. Textabschnitte können auch hier nur durch ihre Position im Gesamttext bestimmt werden. Das Einfügen und Löschen von Text verändert diese Positionen und folglich den durch einen Link referenzierten Inhalt bis hin zum ungültig Werden der URI. Auch die Referenzierung der Seiten kann dadurch betroffen sein, wenn Page-Objekte eingefügt oder gelöscht werden. Für die Praxis dürfte dies auch hier nicht von Relevanz sein, da auch hier die Verlinkung einzelner Zeilen wenig sinnvoll erscheint und eher ganze Textabschnitt bzw. Seiten verlinkt werden.

5. Verarbeitung XML-basierter Formate

XML erfreut sich als menschenlesbares Format großer Beliebtheit. Wie in Kapitel 3 gezeigt wurde, dient es als Grundlage für eine Reihe von anderen Formaten, weshalb die Performanz der Verarbeitung von XML bei dem Verlinken offener Daten eine entscheidende Rolle spielt.
Bei der Verarbeitung von XML werden wie bei vielen anderen Text-basierten Formaten Parser eingesetzt. Das Parsing ermöglicht den Zugriff auf XML Daten, die Prüfung ihrer Wohlgeformtheit und je nach Verarbeitungs-Modell ihre Validierung gegenüber XML-Schemata.
In Lam et al.[27] sehen die Autoren das Parsing als kritischen Schritt für die Gesamt-Performance an. Die Verarbeitung von XML-Dokumenten wird gemäß Abbildung 5.1 als Teil einer Kette von Schritten beschrieben. Dabei wird betont, dass das Parsing als wichtigster Schritt alle darauf folgenden Schritte maßgebend bestimmt. Die Art des Zugriffes und die Möglichkeit zur Modifikation hängt signifikant vom Model des Parsers ab. Folglich wird auch die Performance dieser Schritte vom Parsing bestimmt.

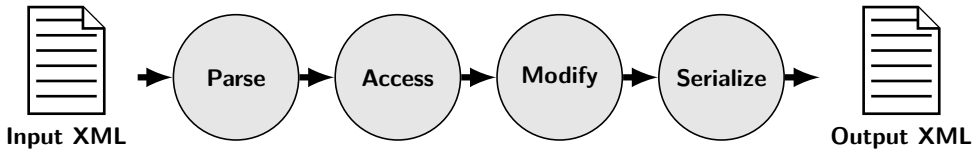


Abbildung 5.1.: Schritte bei der Verarbeitung von XML-Dokumenten nach Lam et al.[27]

Für das Parsing von XML gibt es verschiedene Modelle, welche sich zum Teil grundlegend in Herangehensweise und Charakteristika unterscheiden. Zum besseren Verständnis ihrer Funktionsweise ist es notwendig die vorhandenen Modelle in Kategorien zu unterteilen und deren Eigenschaften gegenüber zu stellen. Die Autoren von Rami et al.[21] folgen dabei einer weitläufigen Einteilung in „streamingbasierte“ und „baumbasierte“ Parsing-Modelle. In der Arbeit von Oliveira et al.[29] wird diese Unterteilung unter Rücksichtnahme auf neue Entwicklungen verfeinert und die Begriffe „streamingbasierte“ und „speicherbasierte“ Modelle verwendet. Diese Begriffe werden auch in dieser Arbeit aufgegriffen. Die verschiedenen Parsing-Modelle werden in den folgenden Abschnitten beschrieben.
Ein Ansatz die Performance zu verbessern sind binäre XML-Formate[1]. Diese Formate sind kompakter, ermöglichen eine schnellere Verarbeitung und wahlfreien Zugriff. Im

[1] *http://www.w3.org/XML/Binary/*

Gegenzug büßt man jedoch den Vorteil XML-Dateien direkt im Editor auf textueller Ebene bearbeiten zu können ein. Die Konvertierung von XML zu binärem XML und umgekehrt bleibt jedoch möglich und ist ein wichtiger Grund solche Formate überhaupt zu verwenden. Bisher hat sich jedoch kein Standard etablieren können, obwohl bereits 2011 eine Empfehlung für das Efficient XML Interchange (EXI)[2], vom W3C herausgegeben wurde. Dieser Ansatz beschränkt sich daher auf einige spezielle Anwendungsgebiete. Da sowohl Eingabe als auch Ausgabeformate keinem binären XML-Format entsprechen und folglich eine Konvertierung notwendig ist, werden diese Formate in dieser Arbeit nicht weiter betrachtet.
Ziel dieses Kapitels ist die Analyse verschiedener Parsermodelle wobei auf Aspekte der Performanz und Eignung für den Einsatz in PANDA zum Zweck der Extraktion und dem Verlinken von Daten eingegangen wird. Die verschiedenen Modelle werden zunächst vorgestellt, dann bezüglich ihrer Eigenschaften analysiert und zum Abschluss durch umfangreiche Benchmarks auf die tatsächliche Performanz hin geprüft. Die Benchmarks entsprechen dabei dem gleichen Ablauf, den auch PANDA bei der Extraktion von Daten XML-basierter Formate zum Zweck des Verlinkens dieser Daten befolgen muss.

5.1. Simple API for XML (SAX)

Die „Simple API for XML“ gehört zu den streamingbasierten Modellen für das XML-Parsing und wurde für das effiziente Lesen und Verarbeiten von XML konzipiert. Sie war die erste API für XML die sich in Java als „De-Facto-Standard“[13] durchsetzen konnte und ist heute auch in vielen anderen Sprachen verbreiten.
Der Ablauf des Parsings erfolgt gemäß Abbildung 5.2. Der SAX-Parser arbeitet sequentiell auf einem Datenstrom. Dabei werden immer nur Teile des gesamten XML-Dokumentes in den Speicher geladen. Der daraus resultierende vergleichsweise niedrige Speicherverbauch ermöglicht zumindest theoretisch die Verarbeitung unendlich großer XML-Dokumente. Die verarbeiteten XML-Elemente lösen von der API vordefinierte zustandslose Events wie z.B. Start-Element und End-Element aus. SAX wird daher häufig auch als eventbasierte API bezeichnet. Mittels Rückruffunktion (Callback) werden vom Entwickler geschriebene Handler aufgerufen, welche die entsprechenden Events verarbeiten. Die Ablaufkontrolle obliegt dabei allein dem Parser. Dieser schiebt die Daten zur Applikation ohne das diese darauf Einfluss nehmen kann. Aus diesem Grund werden SAX-Parser auch häufig als „Push-Parser“ bezeichnet.
Diese ausschließlich vorwärtsgerichtete Arbeitsweise von SAX bringt verschiedene Vor- und Nachteile mit sich. Zum einen ist wie bereits erwähnt die Verarbeitung sehr großer XML-Dokumente möglich, bei denen andere Verfahren oft an von der Hardware gesetzten Grenzen scheitern würden. Da die Zwischenergebnisse direkt verfügbar werden, muss eine Applikation nicht das Ende des Parsings abwarten um mit der Auswertung der Daten zu beginnen. Diese Eigenschaften bringen prinzipiell große Vorteile in Bezug auf Rechenzeit und Speicherverbrauch.

[2] *http://www.w3.org/TR/2011/REC-exi-20110310/*

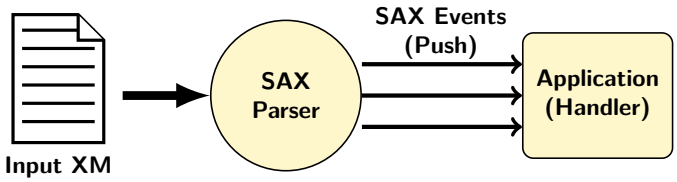


Abbildung 5.2.: SAX-Parsing Modell

Das Streaming bringt jedoch auch Nachteile mit sich. Es sind zunächst keine ausreichenden strukturellen Informationen über das gesamte XML-Dokument verfügbar. Ist das XML-Dokument nicht dem Standard entsprechend wohlgeformt bricht das Parsing im schlimmsten Fall erst nach einiger Bearbeitungszeit ab. Auch die Handhabbarkeit für den Entwickler ist im Vergleich zu anderen Verfahren schwieriger. Die Logik für das Parsing und für die Verarbeitung sind eng miteinander verzahnt. Die zustandslosen Events und die vorwärtsgerichtete Arbeitsweise machen es oft notwendig manuell zusätzliche Statusinformationen zu halten. Im schlimmsten Fall lassen Entwickler vollständige Objektbäume im Hintergrund aufbauen, was die Nutzung von SAX sinnlos werden lässt. Am besten eignet es sich daher für Anwendungsfälle, in welchen bestimmte Informationstypen unabhängig vom Ort ihres Auftretens im XML-Dokument immer gleich behandelt werden. Das Model erlaubt keine mehrfache Nutzung nach dem Parsing, sprich bei jeder Anfrage muss das Dokument erneut geparsed werden. Trotz des Geschwindigkeitsvorteils können sich bei bestimmten Anwendungsfällen große Nachteile bezüglich der Verarbeitungszeit ergeben. Die Erstellung von XML-Dokumenten sowie Modifikationen von vorhandenen XML-Dokumenten werden von der SAX von Haus aus nicht unterstützt.
Auch die Zusammenarbeit mit etablierten Technologien wie die Abfragesprache XPath und „Extensible Stylesheet Language Transformations“ (XSLT) zur Transformation von XML-Dokumenten sind von Haus aus nicht gegeben. Für das Matching von Templates auf XML-Streams wurden Lösungen auf Basis von Automaten wie z.B. TMPL[26] realisiert. Für XPath besteht jedoch zusätzlich das Problem keine rückwärtsgerichteten Achsen auswerten zu können. Der Autor von Barton et al.[22] behauptet mit seinem Algorithmus sowohl vorwärts- als auch rückwärtsgerichtete Achsen auswerten zu können. In der Arbeit von Olteanu et al.[30] werden Rewriting Techniken eingesetzt, um rückwärtsgerichtete Achsen vorab aus einem XPath-Ausdruck zu entfernen. Die Ergebnisse dieser Arbeit wurden im Projekt „SPEX“[3] umgesetzt.
SAX ist Teil von „Java API for XML Processing“ (JAXP)[4], einer ins das JDK integrierten Sammlung von Referenz-Implementationen und somit als Standard-Komponente der Java Plattform direkt verfügbar.

[3] *http://spex.sourceforge.net/#spex*
[4] *https://jaxp.java.net/*

5.2. Streaming API for XML (StAX)

Die „Streaming API for XML“ (StAX) ist ein weiteres Modell für streamingbasiertes XML-Parsing und wird in JSR 173[9] spezifiziert. Die Entwicklung von StAX orientierte sich an den bekannten Nachteilen der Modelle SAX und DOM.
StAX gehört wie SAX zu den streamingbasierten Verfahren und setzt ebenso auf vordefinierte Events. Sie unterteilt sich in zwei Sub-APIs in Form der low-level Cursor-basierten API und der Iterator-basierten API. Die Cursor-API ist auf Effizienz fokussiert während die Iterator-API ein höheres Level an Abstraktion und zusätzliche Methoden für einfachere Benutzung bietet. In ihr werden Events als eigenständige Objekte verwaltet, die beim weiteren Iterieren zwischengespeichert werden können. So ist im Gegensatz zur Cursor-API auch der Zugriff auf frühere Events möglich.
Der Unterschied zu SAX liegt im Ablauf des Parsings wie er in Abbildung 5.3 dargestellt wird. Statt Events mittels Callbacks an einen Handler zu „pushen“ obliegt die Ablaufkontrolle des Parsings allein der Applikation. Diese kann vom Parser schrittweise das nächste Event bei Bedarf abfragen, weshalb das StAX-Model auch oft als „Pull-Parsing“ bezeichnet wird. Das Pull-Parsing ermöglicht einen prozeduralen Ablauf, so kann auch Sub-Parsing realisiert werden, bei dem verschiedene Methoden das weitere Parsing übernehmen. Aufgrund des direkten Handler-Aufrufes durch den Parser, ist eine solche Ablaufkontrolle dort nicht möglich. Da die Events zustandslos sind und dem Handler nicht bekannt ist in welchem Element er sich gerade befindet müssen weitere Statusinformationen z.B. in Form von Flags gehalten werden um ähnliche Abläufe umzusetzen. Der Code von einer StAX nutzenden Applikation kann entsprechend modularer gestaltet werden und weißt eine einfachere Handhabung auf. Des Weiteren ermöglicht der prozedurale Ablauf die Verarbeitung mehrerer XML-Dokumente parallel im selben Thread. Dies ist z.B. bei Vergleichsoperationen zwischen XML-Dokumenten nützlich.

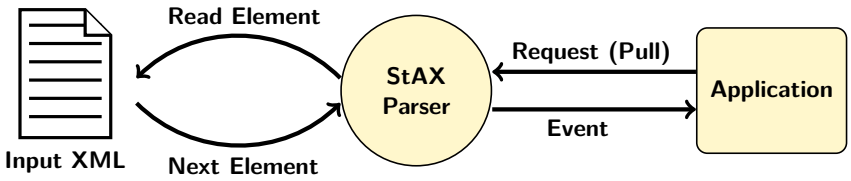


Abbildung 5.3.: StAX Parsing Modell

Die StAX bietet die Möglichkeit XML-Dokumente zu erstellen und eröffnet so Möglichkeiten für weitere Anwendungsfelder wie z.B. Databinding. Eine Modifikation[5] von vorhandenen XML-Dokumenten ist jedoch weiterhin nicht möglich.
Ein weiterer Vorteil von StAX ist, dass nicht relevante Teile aus XML-Dokumenten hergeausfiltert werden können. Zwar werden weiterhin alle Events wie bei SAX sequentiell vom Parser erzeugt, jedoch ist es möglich Events bei der Verarbeitung zu ignorieren.

[5]CRUD - Create, Read, Udate, Delete

Beispielsweise ließen sich mittels Zähler für die Verschachtelung Kinder nicht relevanter Knoten überspringen, während diese bei SAX vom Handler vollständig verarbeitet werden müssten. Die Performance kann dadurch in Abhängigkeit vom Anwendungsfall verbessert werden.
Die Vorteile gegenüber SAX bei sonst ähnlichen Eigenschaften sind der Grund dafür, dass StAX heute oft als bessere Alternative zu SAX gesehen und bevorzugt verwendet wird. StAX ist ebenfalls Teil der „Java API for XML Processing“ (JAXP)[6], einer in das JDK integrierten Sammlung von Referenz-Implementationen und somit als Standard-Komponente der Java Plattform direkt verfügbar.

5.3. Document Object Model (DOM)

Das „Document Object Model“ (DOM)[5] ist ein Standard Modell für das Parsing von XML- oder auch HTML-Dokumenten. In ihm werden Objekte und Attribute und Methoden für deren Zugriff und Manipulation für alle XML-Elemente definiert. Das DOM gehört zu den speicherbasierten Modellen für das XML-Parsing. Die Verfahren laden auf die eine oder andere Weise das vollständige XML-Dokument in den Speicher. Im Vergleich zu den streamingbasierten Verfahren bedeutet dies einen deutlich höheren Speicherverbrauch. Abhängig von der Größe des zu verarbeitenden XML-Dokumentes kann hier schnell an die Hardware-Grenzen gestoßen werden und ein Parsing mit DOM somit unmöglich machen.
Beim DOM läuft die Verarbeitung gemäß Abbildung 5.4 ab. Der DOM-Parser erzeugt auf Grundlage des Eingabe XML-Dokumentes einen Objektbaum dessen Objekte die XML-Elemente darstellen im Speicher. Anschließend kann die Applikation den Objektbaum Traversieren und Daten extrahieren. Diese Form der Darstellung entspricht der natürlichen Struktur von XML und sorgt damit für eine einfache Handhabung seitens des Entwicklers. Das DOM bietet eine hohe Flexibilität durch freie Navigation innerhalb des Baumes (wahlfreier Zugriff). Darüber hinaus können auch Modifikationen (Create, Update, Delete) am Baum einfach durchgeführt und anschließend zu einem Ausgabe XML-Dokument serialisiert werden.

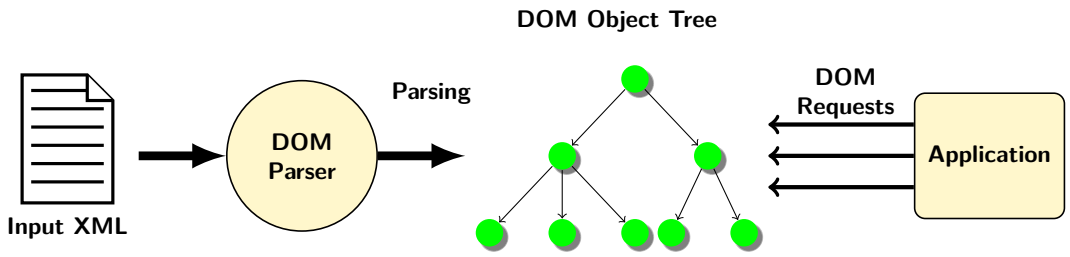


Abbildung 5.4.: DOM-Parsing Modell

[6] *https://jaxp.java.net/*

Die Objekte verbrauchen leider nicht nur viel Speicher, ihre Instanziierung verursacht auch hohe Kosten bezüglich Rechenzeit. Ein Ansatz DOM zu optimieren ist daher die Option „defer-node-expansion“ die im Vergleich zur „full-node-expansion“ nur jene Teile des Objektbaumes erstellt und in den Speicher läd die bei der Traversierung genutzt werden.
Der DOM Objektbaum erlaubt beliebig viele Zugriffe solange dieser sich im Speicher befindet. Dies erlaubt eine Stapelverarbeitung mehrerer Anfragen ohne das Dokument erneut parsen zu müssen. Wann DOM besser geeignet ist hängt also stark vom Anwendungsfall ab. In der Regel bietet sich die Nutzung als einfachste Variante für die Arbeit mit kleineren lokalen XML-Dokumenten an.
DOM ist Teil von der „Java API for XML Processing“ (JAXP)[7], einer in das JDK integrierten Sammlung von Referenz-Implementationen und somit als Standard-Komponente der Java Plattform direkt verfügbar. Darüber hinaus gibt es mehrere unabhängige Implementationen wie „dom4j“[8] und „JDOM“[9].

5.4. Nicht-extraktives Parsing mit VTD-XML

Im Jahr 2004 wurde von Jimmy Zhang[33] ein alternatives Parsing-Modell für XML-Dokumente vorgeschlagen. Dieses als nicht-extraktives Parsing bezeichnete Modell soll Defizite von DOM und SAX überwinden können und zu diesen eine neue Alternative bieten. Die Funktionsweise unterscheidet sich vollständig von den bekannten Modellen. Die Grundidee ist es, Token nicht mehr wie bei DOM in Form von Objekten-Knoten im Objektbaum oder wie bei SAX in Form von Events zu extrahieren, sondern ihre Offsets und Längenangaben für Zugriffe zu nutzen. Das Dokument muss dabei für Zugriffe unverändert im Speicher gehalten werden. Grob vereinfacht ließen sich die Token in Form von 2 Arrays beschreiben, wobei das erste die Offsets und das zweite die Länge des Tokens enthällt. Da die Indexstrukturen extern gehalten werden, kann das nicht-extraktive Parsing auch als hybride Methode zu den binären XML-Formaten gesehen werden. Solche Formate wurden jedoch mit der Absicht entwickelt, sie jederzeit durch Konvertierung in normalen XML überführen zu können. Da das XML-Dokument beim nicht-extraktiven Parsing nicht verändert wird, ist eine solche Konvertierung unnötig.
Das nicht-extraktive Parsing ist bisher wenig verbreitet. Die derzeit wichtigste Implementation des nicht-extraktiven Parsings ist das auf dem „Virtual Token Descriptor“ basierenden Projekt „Virtual Token Descriptor for XML“ (VTD-XML) von XimpleWare. Es wurde mit der Absicht entwickelt zu beweisen, dass das Modell des nicht-extraktiven Parsings funktioniert und Vorteile in der Praxis bietet. Die ursprünglich Idee war es XML mit diesem Verarbeitungsmodell auch für dedizierte Hardware wie „Field Programmable Gate Array“ (FPGA) und „Application Specific Integrated Circuit“ (ASIC) attraktiver zu machen. Konkrete Hardware wurde bis heute nicht entwickelt, dennoch ist VTD-XML mittlerweile für normale Rechensysteme in den Sprachen C, C# und Java

[7] *https://jaxp.java.net/*
[8] *http://dom4j.sourceforge.net/*
[9] *http://www.jdom.org/*

verfügbar. Das Projekt ist seit der Version 0.5 im Mai 2004“ Open Source“ . Im Jahr 2005 wurde mit der Version 1.0 die Unterstützung für XPath hinzugefügt. Die Version 1.5 fügte das „Buffer Reuse“ Feature hinzu, welches bei aufeinanderfolgender Verarbeitung mehrerer XML-Dokumente den im ersten Durchlauf allozierten Puffer-Speicher wieder verwendet statt ihn bei jedem Durchlauf neu zu allozieren. Seit Version 2.0 ist es möglich den Index permanent zusammen oder getrennt vom XML-Dokument auf der Festplatte zu speichern um das erneute Parsen später durch Laden des Indexes zu vermeiden. Aktuell steht VTD-XML bei Version 2.11.

5.4.1. Virtual Token Descriptor (VTD)

Der VTD-XML zugrundeliegende Virtual Token Descriptor ist ein binäres Encoding Format mit einer konstanten Länge von 64bit. In ihm sind gemäß Abbildung 5.5 Informationen über Offset, Länge, Tiefe der Verschachtelung und Token-Typ enthalten. Jeder VTD-Record repräsentiert dabei einen Token im Ausgangsdokument. Bei der Indizierung eines XML-Dokumentes wird ein linearer Puffer von VTD-Records für alle Token mit Ausnahme der Ending-Tags erstellt. Die Token selbst werden dabei nicht wie bei anderen Verfahren kopiert. Das XML-Dokument wird unverändert im Speicher gehalten. VTD kann daher als Hybrid zu binären XML-Formaten gesehen werden, der ein universelles Indexing und rückwärtige Kompatibilität zu XML bietet.

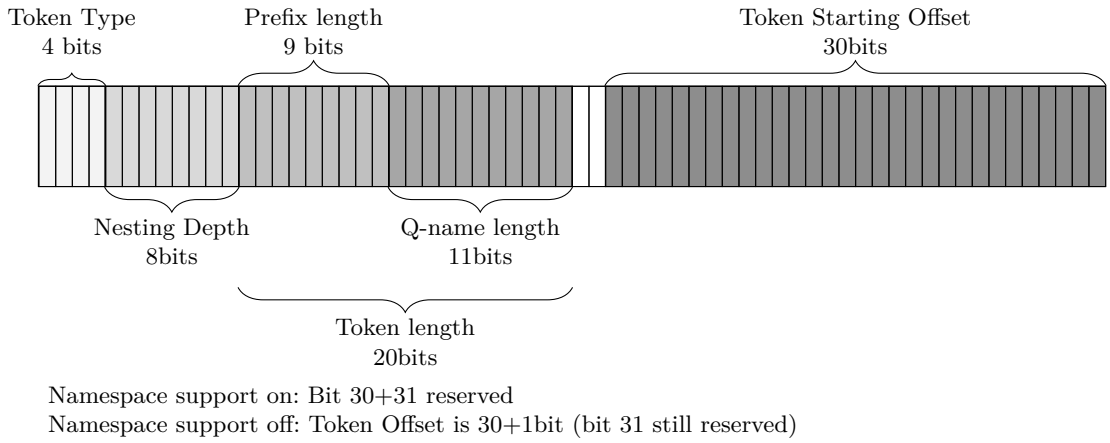


Abbildung 5.5.: VTD Record

Um das Prinzip besser anschaulich zu machen, wird für das Beispieldokument in Listing 5.1 der Inhalt des VTD-Puffer in Tabelle 5.1 dargestellt. Die Token selbst sind dort nur zur einfacheren Zuordnung aufgeführt und befinden sich nicht im VTD-Record.

```
<?xml version="1.0"?>
<Example>
  <E1>
    <E1C1>
      <E1C1C1>foo1</E1C1C1>
      <E1C1C2>foo2</E1C1C2>
    </E1C1>
    <E1C2>bar1</E1C2>
  </E1>
  <E2/>
  <E3>
    <E3C1>foo3</E3C1>
  </E3>
</Example>
```

Listing 5.1: Beispiel XML-Dokument

Index	Token	Token Type	Nesting Depth	Offset: Prefix: Q-name
3	Example	0	0	23:0:7
4	E1	0	1	35:0:2
5	E1C1	0	2	43:0:4
6	E1C1C1	0	3	54:0:6
7	foo1	5	3	54:0:4
8	E1C1C2	0	3	80:0:6
9	foo2	5	3	80:0:4
10	E1C2	0	2	116:0:4
11	bar1	5	2	116:0:4
12	E2	0	1	144:0:2
13	E3	0	1	152:0:2
14	E3C1	0	2	160:0:4
15	foo3	5	2	160:0:4

Tabelle 5.1.: Tabellarische Darstellung des VTD-Puffer + Token

5.4.2. Navigation

Die im VTD-Puffer gespeicherten Token reichen bereits für SAX ähnliche Suchläufe aus, mit dem Vorteil auch rückwärts suchen zu können. Für die Navigation mittels Cursor bedarf es jedoch zusätzlicher Informationen, die im sogenannten „Location Cache“ gespeichert werden. Der Location Cache setzt sich aus mehreren Levels zusammen, wobei gilt $\#LC <= MaximumNestingDepth$. Jeder Location Cache enthält jeweils einen Eintrag für jedes Element seines Levels. Die Einträge sind wie auch die VTD-Records 64bit lang und lassen sich daher effizient am Stück in Array-ähnlichen Strukturen speichern. Die oberen 32bit enthalten den globalen Index des Start-Tags des gespeicherten Elementes im VTD-Puffer. Die unteren 32bit enthalten den Index des ersten Kindes im nächst tieferen Location Cache Level. Ist der Knoten kinderlos wird statt dem Index der Wert „-1“ gespeichert. Für unser Beispiel aus Listing 5.1 sind die 3 Level des Location Cache in Tabelle 5.2 bis 5.4 abgebildet.
Zusätzlich zum Location Cache gibt es für die Navigation das sogenannte „Context Object“ , in dem Informationen zum Cursor gespeichert werden. Während der Location Cache die Baumstruktur des XML-Dokumentes darstellt sorgt das Context Object für Informationen, wo sich der Cursor gerade im Baum befindet. Es handelt sich dabei um ein einfaches Integer-Array der größe $CO[MaximumDepth]$. Das erste Element $CO[0]$ enthält die aktuelle Tiefe der Verschachtellung. In den restlichen Elemente des Arrays enthalten mit Ausnahme des Wurzelknotens den Index des jeweils auf einem Level besuchten Elementes im VTD-Puffer, wobei die $CO[n]$ mit $n > CurrentDepthLevel$ immer -1 ist. Auf diese Weise lässt sich z.B. der Namensraum eines gegebenen Prefix einfach

durch eine Bottom-Up-Suche ermitteln.
Damit ist jetzt alles vorhanden was für die Navigation in der Baumstruktur benötigt wird. Der Sprung zum ersten Kindknoten wird direkt über den Eintrag im Location Cache realisiert. Dabei wird zunächst der letzte Kindknoten berechnet und zwischengespeichert. Dieser befindet sich im nächst tieferen Level des Location Cache direkt vor dem ersten Kindknoten des nächsten Elementes mit Kindern im aktuellen LC-Level. Auf diese Weise lässt sich nun auch zwischen den Schwesterknoten navigieren. Die rückwärtige Navigation zu den Elternknoten wird wieder über das Context Object geregelt.
Neben dieser einfachen Navigation ist natürlich auch das wiederherstellen vom Cursor mittels Eingabe eines Index aus dem Token-Puffer möglich. Auf diese Weise lässt sich auch direkt an bereits bekannte Orte zurücknavigieren.

Index	Global VTD Index	LC Index First Child
1	4	1
2	12	/
3	13	3

Tabelle 5.2.: Location Cache Level 1

Index	Global VTD Index	LC Index First Child
1	5	1
2	10	/
3	14	/

Tabelle 5.3.: Location Cache Level 2

Index	Global VTD Index	LC Index First Child
1	6	/
2	8	/

Tabelle 5.4.: Location Cache Level 3

5.4.3. Eigenschaften

Im folgenden werden die Eigenschaften beschrieben die sich aus dem Verarbeitungs-Modell von VTD-XML ergeben.

(1) **Speicherverbrauch**
VTD-XML behauptet der speichereffizienteste Parser mit wahlfreiem Zugriff zu sein. Der Speicheraufwand von VTD-XML beträgt nach eigenen Angaben für textlastige XML-Dokumente bei 1,3-facher und für datenlastige XML-Dokumente bei 1,5 to 2-facher Größe des Ausgangsdokumentes. Der geringe Speicheraufwand im Vergleich zu DOM erklärt sich mit der Vermeidung von Mehraufwand durch die linearen Strukturen. Der VTD-puffer sowie Location Cache lassen sich in einer Array-ähnlichen Struktur halten mit einmaligem Mehraufwand für diese Datenstruktur. Bei DOM dagegen bringt jedes einzelne Objekt des DOM-Objektbaumes einen Mehraufwand mit sich. So hat ein einfaches POJO einen Mehraufwand von 16Bytes[10]. Der genaue Speicheraufwand von VTD-XML bei voller Indizierung (ein Location Cache pro Level) lässt sich folgendermaßen ermitteln:

$$VTDPuffer = (\#Token - \#StartTags) * 8 \text{ Bytes}$$

[10]Gemessen mit java.lang.instrument.instrumentation auf einer 64bit VM

$LC = \#StartTags * 8$ Bytes
$==> GesamtSpeicherverbrauch = sizeOf(Dokument) + (\#Token - \#StartTags) * 8 + (\#StartTags * 8) = sizeOf(Dokument) + 8 * \#Token$ Bytes

(2) **Rechenzeit**
Das Vermeiden der Instanziierung vieler Objekte bringt auch Vorteile bezüglich Rechenaufwand. Sowohl die Instanziierung von Objekten als auch die Allokation vieler kleiner Speicherstücke bringen jeweils Kosten mit sich. VTD-XML umgeht dies durch die Allokation großer Speicherstücke z.B. für den VTD-Puffer und direkten Operationen auf binärer Ebene statt mit Strings. Dies wird auch als Grund für den Geschwindigkeitsvorteil gegenüber SAX genannt, der mit 1,3 bis 3-facher Geschwindigkeit angegebenen wird. Auch SAX alloziert viele wenn auch kurzlebige Objekte und führt Vergleichsoperationen mit String-Objekten durch.

(3) **Extraktion**
Im Punkt Extraktion bietet VTD-XML bedeutende Vorteile. Durch das Design basierend auf Offsets und Längen lassen sich beliebige Teile wie z.B. der Teilbaum eines Elementes aus einem XML-Dokument extrahieren. VTD-XML ermöglicht dadurch XML-Dokumente nach Belieben zu zerschneiden und wieder zusammenzufügen. Zudem kann direkt auf den Inhalt des Dokumentes zugegriffen werden ohne diesen zuvor serialisieren zu müssen.

(4) **Modifikation (CRUD)**
Modifikationen laufen in VTD-XML ähnlich wie in einem Text-Editor inkrementell ab. Es ist jeweils nur der zu verändernde Teil des Dokumentes direkt betroffen. Das Update eines Tokens würde folgendermaßen ablaufen:

(a) Kopiere den Teil des Dokumentes vor dem Token
(b) Überschreibe das Token
(c) Füge den Rest des Dokumentes hinzu (Offset = Offset des Tokens + Länge)
(d) Wenn $Länge(Token_{alt}) > Länge(Token_{neu})$ fülle die fehlenden Zeichen auf, sonst Update der Indizierung vom Rest des Dokumentes neu.

Bezogen auf einzelne Änderungen ist dies trotz der Gefahr neu indizieren zu müssen sehr schnell. Zudem können multiple Modifikations-Operationen registriert werden bevor die Ausgabe generiert wird, um eine mehrfache neue Indizierung zu vermeiden. Dennoch könnte dieses Vorgehen Nachteile bei repetitiven Modifikations-Operationen geben die jeweils eine neue Indizierung nach sich ziehen können.

(5) **Limitierungen**
VTD-XML unterliegt aufgrund des Konzeptes und Entwicklungsstadiums ein paar Einschränkungen. Es wird bisher keine Schemavalidierung unterstützt, auch wenn diese als geplant angegeben wurde. Externe Entitäten wie sie in der „Document Type Definition“ (DTD) angegeben werden, werden bis auf die in XML 1.0 definierten Standards &amps; > < ' und " nicht unterstützt. Darüber

hinaus gibt es natürliche Grenzen durch die Größe der jeweiligen Felder im VTD-Record. Ein XML-Dokument darf bei eingeschalteter Unterstützung von Namensräumen 1GB groß sein. Ist diese ausgeschaltet so können Dokumente bis 2GB Größe verarbeitet werden. Die Länge des Prefix ist auf 511 Zeichen begrenzt. Der qualifizierte Name darf 2047 Zeichen lang sein. Die maximale Tiefe der Verschachtelung beträgt 254 Level.

Das nicht-extraktive Parsing bietet für viele Anwendungsfälle eine echte Alternative, auch wenn es wegen des völlig unterschiedlichen Lösungsansatzes nicht mehr mit der Definition von W3C-DOM kompatibel ist. Anwendungsszenarien sind z.B. große XML-Dokumente die sich mit DOM nicht mehr verarbeiten lassen, Performance kritische Anwendungen, XML-Datenbanken oder im Fall von VTD-XML auch eingebettete Systeme die aufgrund der linearen Strukturen von VTD-Puffer und Location Cache von darauf ausgerichteter Hardwarebeschleunigung profitieren könnten. Neben dem bekanntesten Projekt VTD-XML werden Formen des nicht-extraktiven Parsings auch im C++ Projekt „RapidXMLL“[11] und in der in D geschriebenen Bibliothek „Tango“[12] verwendet.

5.5. Evaluation der XML-Parser

5.5.1. Eigenschaften

An dieser Stelle sollen die Vor- und Nachteile der Parsing-Modelle im direkten Vergleich gegenübergestellt und zusammengefasst werden. Dabei wird auch auf die Relevanz der jeweiligen Eigenschaft für dieses Projekt eingegangen. Die Tabelle 5.5.1[27] enthält eine Übersicht von verschiedenen Eigenschaften der Parsing-Modelle die wir im folgenden einzeln betrachten wollen.

[11] *http://rapidxml.sourceforge.net/*
[12] *http://dsource.org/projects/tango*

Eigenschaft	**DOM**	**SAX**	**StAX**	**VTD**
Parsing Modell	Objectbaum im Speicher	Streaming, Events (alle)	Streaming, Events (gefilter)	VTD-Records (Indexing)
Aufwand Parsing CPU	Hoch	Medium	Medium	Niedrig
Aufwand Parsing Speicher	Hoch	Niedrig	Niedrig	Medium
Wahlfreier Zugriff	Ja	Nein	Nein	Ja
XPath (standardmäßig)	Ja	Nein	Nein	Ja
Lesen	Ja	Ja	Ja	Ja
Schreiben	Ja	Nein	Ja	Ja
Create, Read, Update, Delete (CRUD)	Ja	Nein	Nein	Ja

Tabelle 5.5.: Eigenschaften der verschiedenen Modelle für das XML-Parsing

(1) **Parsing Modell**
Die Parsingmodelle unterscheiden sich zum Teil grundlegend und bedingen die weiteren Eigenschaften der Verarbeitung. Ein direkter Vorteil des Streamings bei SAX und StAX, ist die Möglichkeit Daten zu verarbeiten bevor das Parsing abgeschlossen ist. Sowohl DOM als auch VTD-XML müssen das Parsing zunächst einmal vollständig durchlaufen, erlauben dafür jedoch einen multiplen Zugriff solange sich die vom Parser analysierte Form im Speicher befindet. Streaming-Verfahren müssen dagegen bei weiteren Zugriffen erneut parsen. Für den normalen Einsatz von PANDA sind diese Merkmale jedoch kaum relevant, da der Fokus auf der Extraktion von Daten liegt und diese in keiner besonderen Form verarbeitet werden die eine Verfügbarkeit vor Abschluss des Parsings notwendig macht. Die Ergebnisse werden am Stück über eine REST-Schnittstelle zurückgegeben.

Für Caching-Zwecke scheinen sich speicherbasierte Lösungen jedoch besser zu eigenen, da die analysierte Form des XML-Dokumentes im Speicher gehalten werden könnte und somit ein erneutes Parsing unnötig macht.

(2) **Aufwand Parsing: CPU**
Die Anforderungen an die Rechenzeit hat sich in den letzten Jahren durch immer stärker werdende Systeme relativiert. Dennoch sollten übermäßig lange Wartezeiten vermieden werden. Zum einen ist bei Plattformen wie PANDA mit einer größerer Anzahl paralleler Anfragen die verarbeitet werden müssen zu rechnen, zum anderen bedeutet eine schnellere Rückgabe auch eine schnellere Freigabe des Hauptspeichers, der zumindest für die speicherbasierten Parsing-Modelle DOM und VTD-XML von großer Wichtigkeit ist.

DOM scheint hier die größten Nachteile mit sich zu bringen, während VTD-XML vorgibt am schnellsten zu sein. Die Gründe dafür ist in der Vielzahl von Operationen wie Speicher-Allokation und Objekt-Instanziierung zu suchen. In diesem Punkt

tut sich das nicht-extraktive Parsing hervor und verspricht eine deutlich gesteigerte Performance. SAX und StAX versprechen hier ebenfalls eine im Vergleich zu DOM deutlich gesteigerte Performance.

(3) **Aufwand Parsing: Speicher**
Der Aufwand für den Speicher entscheidet in erster Linie darüber was und wie viel verarbeitet werden kann. „Was“ bezieht sich dabei auf die Größe von XML-Dokumenten die verarbeitet werden können ohne an von Hardware bedingte Grenzen zu stoßen. „Wie viele“ bezieht sich auf die parallele Anfrage mehrerer Dokumente , wobei jede Einzelanfrage ihren individuellen Speicherbedarf hat und somit in ihrer Gesamtheit ebenfalls an vom Speicher gesetzte Grenzen stoßen können.

DOM schneidet in diesem Punkt sehr schlecht ab, da der Objekt-Baum die 5- bis 10-fache Größe des Ausgangsdokumentes im Speicher belegen soll.

SAX und StAX bieten in diesem Punkt dagegen ihren größten Vorteil, da das Streaming immer gleich große Teile des XML-Dokumentes verarbeitet. Theoretisch ist so die Verarbeitung von unendlich großen XML-Dokumenten möglich und folglich auch die parallele Verarbeitung von einer großen Anzahl an XML-Dokumenten.

VTD-XML soll dagegen mittelmäßig abschneiden. Es ist deutlich effizienter als DOM, benötigt jedoch trotzdem die 1,3 bis 2-fache[34] Größe des Ausgangsdokumentes im Speicher.

Um diesbezüglich einen realistischen Bezug zu realen Daten zu bekommen, werden 266 XML-Dokumente zu Testzwecken von der Plattform für offene Daten der US-Regierung[13] heruntergeladen. Die Durchschnittsgröße dieser Dokumente beträgt 5,5MB. Von diesen 266 Dokumenten kommen nur 7 über 20MB. Die beiden größten XML-Dokumente sind 343MB bzw. 306MB groß. Wie diese Testreihe zeigt, handelt es sich bei der Mehrzahl der Dokumente um kleinere Dateien von wenigen Mega-Byte. Für PANDA bedeutet dies jedoch, dass DOM bereits mit einer kleinen aber vorhandenen Zahl von Dokumenten Probleme bereiten wird. Eine Verarbeitung von nur einem Dokument dieser Größenordnung kann je nach Ausstattung des Systems unmöglich werden. SAX und StAX dagegen sollten erwartungsgemäß keine Probleme damit haben. Für VTD-XML sind die beiden größten Dokumente mit einem heute handelsüblichen PC ebenfalls kein Problem mehr. Eine parallele Verarbeitung mehrerer dieser Dokumente könnte allerdings schwierig werden. In Anbetracht der geringen Anzahl von Dokumenten dieser Größenordnung sollte dies jedoch kein Ausschlusskriterium für die Auswahl des Parsers sein.

(4) **Lesezugriff/Wahlfreier Zugriff**
Der lesende Zugriff ist bei allen Parsern gegeben. Der wahlfreie Zugriff kann dagegen nur bei bestimmten Verarbeitungsmodellen angewendet werden. Für die reinen XML-Dokumente benötigen wir beim Lesen den wahlfreie Zugriff nicht direkt, jedoch kann er bei der Serialisierung von Teilen des XML-Dokumentes als XML-String

[13] *http://www.data.gov/*

notwendig sein. Auch bei der manuellen Verarbeitung anderer XML-basierter Formate könnte der wahlfreie Zugriff wichtig werden, insbesondere dann, wenn die von XPath gegebene Funktionalität nicht mehr für die Verarbeitung ausreicht.

VTD-XML lässt dem Nutzer hier die größte Freiheit. Mittels Index und Offset ist der Zugriff auf beliebige Teile des XML-Dokumentes möglich. DOM dagegen erlaubt den Zugriff auf einzelne Knoten. Benötigt man einen vollständigen Teilbaum dessen Wurzel ein bestimmter Knoten ist, so muss dieser traversiert und dieser Teil des XML-Dokumentes wieder serialisiert werden. SAX und StAX dagegen erlauben solche Zugriffe nicht, was deren Möglichkeiten stark einschränkt. Alle Daten die nach dem Parsing benötigt werden, müssen während des Parsings gepuffert werden. In Verbindung mit XPath könnte es hier schwierig werden Teilbäume als XML zu extrahieren.

(5) **XPath**
Da es das Ziel ist Daten mittels XPath als Teil einer URI aus einem XML-Dokument zu extrahieren, ist eine vorhandene XPath-Implementation grundlegende Voraussetzung. Typischerweise werden diese für DOM angeboten. VTD-XML bietet ebenso eine Implementation von XPath, beschränkt sich dabei jedoch auf XPath 1.0 mit Erweiterung einiger weniger XPath 2.0 Funktionen. Für den Zweck der Verlinkung von Teilen eines XML-Dokumentes reicht dies jedoch aus. SAX und StAX dagegen besitzen eine solche Implementation von Haus aus nicht. Dies liegt an der vorwärtsgerichteten Natur der Verarbeitung. Dennoch gibt es Lösungen die auf SAX aufbauen (siehe 5.1), jedoch häufig Einschränkungen unterliegen. Die schlechte Verfügbarkeit von XPath-Lösungen für die streamingbasierten Verfahren muss daher als negativ gewertet werden.

(6) **Schreiben**
Dieser Punkt bezieht sich auf die Erstellung neuer XML-Dokumente. Lediglich SAX bietet eine solche Option von Haus aus nicht. Für PANDA gibt es jedoch bisher keine Planung XML-Dokumente zu erstellen, daher kann dieser Punkt außer Acht gelassen werden.

(7) **Create, Read, Update, Delete (CRUD)**
Das Projekt PANDA konzentriert sich in erster Linie auf die Verlinkung und Extraktion von Offenen Daten für Einzel- und Massenabfragen, dennoch könnte in Zukunft die Umsetzung von Modifikations-Operationen gewünscht sein. Da Lese- und Schreiboperationen getrennt werden können, ist auch die Umsetzung mit einem anderen Verfahren denkbar. SAX und StAX sind mangels Funktionalität für diese Aufgabe ungeeignet. DOM eignet sich am Besten für eine Vielzahl von aufeinanderfolgenden Modifikationen wobei anschließend ein neues XML-Dokument reserialisiert wird. VTD-XML bietet dagegen inkrementelle Updates die ähnlich wie Texteditoren nur einen Teil des XML-Dokumentes betreffen. An der entsprechenden Stelle wird der neue Wert eingefügt bzw. der alte Wert modifiziert oder gelöscht. Der Rest des Dokumentes wird als ein Stück behandelt, anschließend angehangen und ggf. mit neu indiziert. Eine Reserialisierung ist jedoch nicht notwendig. VTD-XML eignet sich

also hervorragend für einzelne Modifikationen von XML-Dokumenten. Welches der beiden Modelle für diese Aufgabe geeigneter ist, hängt daher von den spezifischen Anforderungen ab. Da sich diese Arbeit auf die Extraktion und das Verlinken von Daten konzentriert, werden Schreiboperationen an dieser Stelle nicht weiter betrachtet.

5.5.2. XPath/Parser-Benchmark

In der Vergangenheit wurden bereits zahlreiche Studien zum Thema Performance von XML-Parsern gemacht. Bei der Evaluation für PANDA wollen wir jedoch näher an den realen Arbeitsabläufen der Plattform bleiben, um ein genaues Bild für die richtige Auswahl des Parsers zu bekommen. Wichtigste Voraussetzung ist die Vollständigkeit der Tests bezüglich der verschiedenen Parsing-Modelle. Ein vergleichsweise aktueller Test findet sich in der Arbeit von Deshmukh et al.[23]. Die Ergebnisse dieser Arbeit scheinen jedoch zu ausgewogen, was der geringen Dateigröße anzulasten ist. Bessere Ergebnisse liefert die Arbeit von Oliveira et al.[29], in der sich VTD-XML durch guten Speicherverbrauch und bezüglich Rechenzeit meist deutlich absetzen konnte. Auch in Tej et al.[32] wird dieses Bild bestätigt. Wie wir jedoch bereits wissen, beeinflusst das Parsing-Modell auch die anderen Schritte im Verarbeitungsprozess. Wir interessieren uns vor allem für den Zugriff und die Serialisierung der Ergebnisse. Da die genannten Artikel sich auf Tests des reinen Parsing Vorgangs beschränken und sich XPath-Benchmarks von VTD-XML nur von dessen Autoren selbst finden lassen, liegt die Entscheidung eigene Benchmarks für genauere Analysen durchzuführen nahe.
Der Zugriff soll mittels XPath geschehen wobei möglichst die eigens von den Testkandidaten mitgebrachte XPath-Implementierung verwendet werden soll. Ziel ist es mittels verschiedener XPath-Ausdrücke, aus verschiedenen Dokumenten unterschiedlicher Größe, Knoten zu selektieren und ihre Teilbäume als String zu serialisieren. Da die streamingbasierten Verfahren wie SAX von Haus aus keine XPath-Implementierung besitzen, suchen wir APIs die dennoch wie in Abschnitt 5.1 beschrieben eine individuelle XPath-Implementierung anbieten. Dabei sind wir durchaus bereit Einschränkungen hinzunehmen, mit der Voraussetzung, dass weiterhin die dem Deep-Linking entsprechende Verlinkung aller Teilbäume möglich bleibt.

5.5.2.1. Testkandidaten

Die Suche nach geeigneten Testkandidaten gestaltet sich insbesondere bei streamingbasierten Parsern als schwer, da diese von Haus aus kein XPath unterstützten. Die Implementierung von „SPEX“[30][14] scheint nicht über den experimentellen Status hinaus gekommen zu sein. XPath Ausdrücke werden nur in ihrer ungekürzten Form unterstützt. Funktionen wie „position()“ oder „text()“ scheinen ebenfalls nicht unterstützt zu werden. Es fehlt zudem an jeglicher Dokumentation oder Anleitungen zur Nutzung des Projektes. Dennoch konnte ein kurzer Test durchgeführt werden. Dieser wies jedoch trotz des SAX-Parsers einen sehr hohen Speicherverbrauch und eine hohe Rechenzeit auf. Der Grund

[14] *http://spex.sourceforge.net/#spex*

dafür könnte eine Fehlkonfiguration sein, dennoch scheint SPEX nicht für die praktische Anwendung geeignet.
Einen brauchbaren Eindruck macht dagegen die auf SAX basierende XPath-Implementation XMLDog[15]. Sie unterstützt eine Untermenge von XPath 1.0, verzichtet dabei jedoch auf die rückwärtigen Achsen XPath-Achsen „parent“ , „ancestor“ , „ancestor-or-self“ , „preceding-sibling“ und „preceding“ . Die Event-gesteuerte Verarbeitung ermöglicht eine sofortige Verarbeitung der Knoten nach dem selektieren. Leider ist es nicht möglich die Teilbäume der selektierten Knoten als XML-String zu extrahieren. Zu diesem Zweck muss mit DOM gepuffert werden. Es ist also abhängig von der Anzahl der selektierten Knoten zu erwarten, dass XML-Dog ähnlich schlechte Eigenschaften wie DOM aufzeigen wird.
Des Weiteren wird für die Repräsentation von DOM die Java Referenz-Implementation von DOM und darüber hinaus die unabhängigen Projekte „JDOM2“[16] und „dom4j“[17] ausgewählt. Als nicht-extraktiver Parser steht aufgrund schlechter Verbreitung des Modells lediglich VTD-XML zur Verfügung.

5.5.2.2. Testvorbereitung

Um die Tests in Bezug auf den Verwendungszweck von PANDA möglichst realistisch zu halten werden verschiedene Testdaten von der Plattform für offene Daten der US-Regierung[18] verwendet.
Es werden 4 XML-Dokumente mit einer Größe von 188 Kilobyte bis über 300 Megabyte ausgewählt. Für die einfachere Zuordnung nennen wir sie der Größe nach „Small“ , „Medium“ , „Big“ und „VeryBig“ . Die ausgewählten Testdokumente teilen sich wie die meisten XML-Dokumente offener Datenplattformen die Eigenschaft eine sehr breite und relativ flache Baumstruktur zu besitzen. Die Ausdrücke von XPath sind so ausgewählt, dass sie 1. alle Knoten, 2. den ersten Knoten des Levels der Baumstruktur mit der maximalen Breite, 3. den Knoten in der Mitte dieses Levels, 4. den letzten Knoten dieses Levels und 5. eine größere Anzahl von Knoten mittels Vergleich von Attributen und Textknoten selektieren. Eine Übersicht über die Dokumente und die benutzten XPath-Ausdrücken ist in Tabelle 5.6 abgebildet.

[15] *https://code.google.com/p/jlibs/wiki/XMLDog*
[16] *http://www.jdom.org/*
[17] *http://dom4j.sourceforge.net/*
[18] *http://www.data.gov/*

XML-Dokument	Größe	XPath-Ausdruck	#Knoten
Traffic_Flow_Counts.xml *http://catalog.data.gov/dataset/traffic-flow-counts*	Small 188KB	//*	4059
		/response/row/row[1]	1
		/response/row/row[400]	1
		/response/row/row[last()]	1
		//downtown[text()='true']	105
Baltimore_City_Employee_Salaries_FY2013.xml *http://catalog.data.gov/dataset/baltimore-city-employee-salaries-fy2013*	Medium 6881KB	//*	149827
		/response/row/row[1]	1
		/response/row/row[10000]	1
		/response/row/row[last()]	1
		//jobtitle[text()='FIRE LIEUTENANT']	136
HPI_master.xml *http://catalog.data.gov/dataset/fhfa-house-price-indexes-hpis*	Big 22049KB	//*	91721
		/rows/row[1]	1
		/rows/row[45500]	1
		/rows/row[last()]	1
		//row[@hpi_flavor='purchase-only']	18976
Genealogical_Death_Index_Beginning_1957.xml *http://catalog.data.gov/dataset/genealogical-research-death-index-beginning-1957*	Very Big 306889KB	//*	7156415
		/response/row/row[1]	1
		/response/row/row[335000]	1
		/response/row/row[last()]	1
		//gender[text()='F']	271317

Tabelle 5.6.: Testdaten und verwendete XPath-Ausdrücke

5.5.2.3. Durchführung

Mit jeder der Testdateien werden verschiedene Tests auf einem Testsystem[19] durchgeführt. Im ersten Test wird der Speicherverbrauch ermittelt indem der belegte Speicher vor und nach dem Parsing mit anschließender Selektion der Knoten mittels XPath gemessen wird. Im Fall des SAX-basierten XML-Dog ist zu beachten, dass der Aufbau eines DOM-Objektbaumes für die spätere Serialisierung aktiviert sein muss. Hier sind also stark vom XPath-Ausdruck abhängige Ergebnisse zu erwarten.
Der zweite Test misst die CPU-Zeit die Parsing und anschließende Selektion der Knoten durch XPath benötigt. Dabei werden die von „java.lang.management“ zur Verfügung gestellten Mittel zur direkten Ermittllung der CPU-Zeit des aktuellen Thread genutzt um ein möglichst unabhängiges Ergebnis zu erhalten. Auffällig ist jedoch, dass die erstmalige Initialisierung der jeweils getesteten API ein großes Maß zusätzlicher Zeit kostet. Da dies nicht den realen Anwendungsbedingungen von PANDA entspricht, wird jeweils vor der echten Messung ein Trockentest des Parsers ohne Messung durchgeführt.
Der dritte Test beschäftigt sich mit dem Mehraufwand für die Serialisierung der Ergebnisse. Die Serialisierung spielt in der Praxis meist bei Modifikationen von XML-Dokumenten eine Rolle. Im Fall von PANDA wollen wir jedoch mehr als nur einzelne Knoten extrahieren. Ziel ist die Extraktion der Teilbäume dessen Wurzelknoten die von XPath selektierten Knoten sind als XML-String, sodass ein XPath-Ausdrück als Bulk-URI für diese Teilbäume gesehen werden kann. In diesem Test wird der jeweils erste XPath-Ausdruck für das Selektieren aller Knoten verwendet um ein möglichst genaues Ergebnis zu bekommen. In der Praxis wird der Mehraufwand daher kleiner ausfallen. Ziel des Tests ist jedoch zu zeigen, wie effizient die Serialisierung insgesamt abläuft. Ausgehend vom Verfahrens-Modell ist hier ein Vorteil des nicht-extraktiven Parsings von VTD-XML zu erwarten, da das Dokument unverändert im Speicher verbleibt und in beliebige Teile geschnitten werden kann. Bei den anderen Testkandidaten werden jeweils mitgebrachte Methoden und Klassen für die Serialisierung als XML verwendet. Auch in diesem Test wird wieder die CPU-Zeit als Maß verwendet.
Im letzten Test soll die Speicher- und Ladefunktion für den Index von VTD-XML untersucht werden. Diese Funktionen ermöglichen es alle Daten des Indexing entweder separat oder als zusammengefügte Datei mit dem Eingabedokument auf der Festplatte zu speichern und zu einem späteren Zeitpunkt ohne erneutes Parsing wiederherzustellen. Da das Speichern eine einmalige Operation ist, interessiert vor allem die Frage, ob das Laden eines Indexes schneller vonstatten geht als das erneute Parsing.

5.5.2.4. Ergebnisse

Die Abbildungen der Ergebnisse werden aufgrund ihres Umfangs im Anhang platziert.
Test Speicheraufwand
Der erste Test (siehe Anhang Seite 83 bis 85, Abbildung A.1 bis A.4) bescheinigt VTD-XML einen konsequent niedrigen Speicherbrauch. Auffällig ist hierbei vor allem die Gleichmäßigkeit der Ergebnisse, denn im Unterschied zu DOM wird kein Ergebnis-

[19] Intel Core i5-4670k, 8GB DDR3, Crucial M4 SSD

Set mit selektierten Knoten aufgebaut. Besonders bei der Referenzimplementation und folglich auch XmlDog ist die Abhängigkeit zur Anzahl der selektierten Knoten zu erkennen. Dom4j und JDOM2 zeigen bei der Selektion einer Gruppe von Knoten einen vergleichsweise hohen Speicheraufwand. Da die Anzahl der selektierten Knoten dabei im Vergleich zur Gesamtzahl der Knoten relativ gering ist, ist anzunehmen, dass die XPath-Implementation jeweils einen gewisses Maß an Speicher für zusätzliche Pufferung verwendet. XmlDog zeigt hier erwartungsgemäß ein sehr gemischtes Ergebnis. Trotz interner Verwendung von DOM, ist der Speicheraufwand fast immer geringer als bei der Referenzimplementation. Auffällig ist jdoch, dass die Suche nach dem durch die XPath-Funktion „*last*()“ definierten Knoten ein besonderes Maß an Speicher benötigt. Es ist anzunehmen, dass das gesamte Dokument auf eine unvorteilhafte Weise gepuffert wird, obwohl nur ein einzelner Knoten selektiert werden soll. Unter dem Strich lässt sich bezüglich Speicherauslastung dennoch eine Rangfolge festlegen. DOM verbraucht den meisten Speicherplatz gefolgt von JDOM. Dom4j verbraucht durchschnittlich etwas weniger als JDOM. Der Speicheraufwand von XmlDog ist stark abhängig vom gewählten XPath-Ausdruck, dennoch bietet sich für die meisten XPath-Ausdrücke die eine geringe Anzahl Knoten selektieren ein Vorteil gegenüber den DOM-Parsern. Bei der Suche nach einzelnen Knoten wird sogar VTD-XML geschlagen. Der hohe Speicherverbrauch bei der Suche nach dem letzten Knoten in der maximalen Breite des Baumes könnte durch bessere Implementierung behoben werden. VTD-XML zeigt unter dem Strich das beste Ergebnis und bietet den Vorteil berechenbar zu sein, da das Ergebnis jeweils beinahe Unabhängig vom gewählten XPath-Ausdruck ist. Der Speicheraufwand für das Selektieren eines einzelnen Knoten lag bei 1,65 bis 1,9facher Größe des Eingabe-Dokumentes.

Test Rechenaufwand

Im zweiten Test (siehe Anhang Seite 85 bis 87, Abbildung A.5 bis A.8) schneiden die DOM-Parser wie erwartet schlecht ab. Dom4j scheint gegenüber JDOM2 und DOM leichte Vorteile beim Selektieren einzelner Knoten mittels XPath zu haben. Wird jedoch eine hohe Anzahl Knoten selektiert, so steigt die gebrauchte CPU-Zeit drastisch an. Die Referenzimplementation von DOM scheint in vielen Fällen sogar Vorteile gegenüber den anderen DOM-Parsern zu besitzen. Die XPath-Funktion „*last*()“ scheint hier jedoch Probleme zu bereiten. Die benötigte Zeit ist durchweg drastisch höher, selbst im Vergleich zum Selektieren aller Knoten. Der Test wurde entsprechend beim größten XML-Dokument ausgelassen. Das Problem kann jedoch auf die Implementation von XPath zurückgeführt werden, da die anderen DOM-Parser dieses Problem nicht zu kennen scheinen.

XmlDog liefert durchschnittlich das zweitbeste Ergebnis. Das Selektieren der Knoten direkt auf dem SAX-Stream geht deutlich schneller von statten als im DOM, selbst wenn ein vollständiger Objektbaum durch Selektion aller Knoten aufgebaut wird. Dennoch muss sich XmlDog bei den meisten Tests VTD-XML geschlagen geben. VTD-XML zeigt auch in diesem Test ein sehr konstantes Verhalten, dass sich kaum vom gewählten XPath-Ausdruck beeinflussen lässt.

Test Serialisierung

Der dritte Test (siehe Anhang Seite 87 bis 88, Abbildung A.9 bis A.11) verzichtet auf die Verwendung der größten XML-Datei, da bereits die Tests ohne Serialisierung

mit den DOM-Parsern nicht vollständig durchgeführt werden konnten. Der Test zeigt erwartungsgemäß einen geringen Aufwand für VTD-XML. Der zusätzliche Aufwand für JDOM2 ist ebenfalls sehr gering, jedoch in Bezug auf die zuvor benötigte Zeit für Parsing und XPath zu betrachten. Dom4j zeigte einen mittelmäßig hohen Mehraufwand für die Serialisierung. XmlDog und DOM liegen fast gleich auf. Dies erklärt sich durch die Verwendung der Referenzimplementation von DOM in XmlDog für das Puffern von Knoten.

Test Lade- und Speicherfunktion für Index von VTD-XML

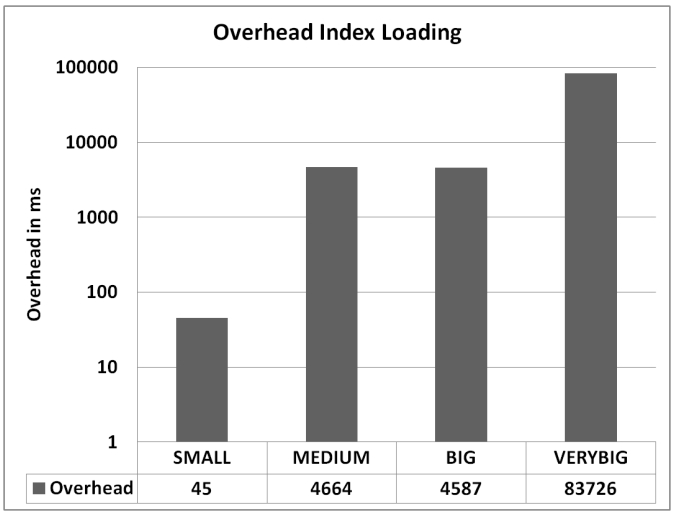


Abbildung 5.6.: Overhead VTD-Index Loading

Der Test der Speicher- und Ladefunktion für den Index von VTD-XML fällt negativ aus. Die zusätzlich benötigte Zeit wenn der Index geladen statt neu berechnet wird ist in Abbildung 5.6 zu finden. Statt des erwarteten Zeitgewinns, erleiden wir trotz des schnellen Festspeichers unseres Testsystems Verluste, welche ein persistentes Speichern des Indexes zumindest für normale Rechensysteme sinnlos erscheinen lassen. Auffällig ist hierbei außerdem, dass das Dokument mit der Größe „BIG“ weniger zusätzliche Zeit für das Laden in Anspruch nimmt. Dies erklärt sich durch eine geringere Anzahl von Knoten und folglich kleinerem VTD-Puffer und Location Cache. Die Größe des Dokumentes ergibt sich aus zahlreichen Attributen.

5.5.3. Schlussfolgerung

Die DOM-Parser zeigten wie erwartet einen sehr hohen Speicheraufwand und eine schlechte Performance bezüglich Rechenzeit. Bei größeren Dokumenten ist die Antwortzeit bei weitem zu hoch, als das diese für die Verlinkung von offenen Datensätzen eingesetzt werden könnten. Zudem zeigen sich Unregelmäßigkeiten in Abhängigkeit vom gewählten XPath-Ausdruck. So versagt die XPath-Implementation von Dom4j bei dem Selektieren einer großen Anzahl von Knoten. Die Reference-Implementation dagegen hat Probleme mit bestimmten XPath-Funktionen.

VTD-XML zeigte durchweg gute und konstante Ergebnisse. Gerade die Extraktion von Teilbäumen als XML-String erwies sich als einfach und Vorteilhaft für die Anwendung in PANDA. Dennoch setzt der Speicherverbrauch erkennbare Obergrenzen für die Verarbeitung von XML-Dokumenten.

Als interessanter Kandidat stellte sich die SAX-basierte XPath-Implementation Xml-

Dog heraus. Der Speicherverbrauch in unseren Tests ist zwar meist höher als der von VTD-XML, jedoch liegt dies in erster Linie an dem Objektbaum der trotz der SAX-basierten Verarbeitung für die Serialisierung aufgebaut werden muss. Eine Veränderung der Implementation die durch String-Extraktion direkt einen XML-String statt einen Objektbaum aufbaut, könnte den Speichervorteil auf alle XPath-Ausdrücke ausweiten. Zwar wäre die Verarbeitung weiterhin langsamer als mit VTD-XML, jedoch wäre der Einsatz für besonders große Dateien bei denen der Speicher Grenzen für die Verarbeitung setzt denkbar.
Die Entscheidung welcher Parser in PANDA für die Verarbeitung XML-basierter Formate verwendet werden soll fällt eindeutig auf VTD-XML. Das Verarbeitungsmodell von VTD-XML liefert uns konstante und berechenbare Anforderungen bezüglich Speicherverbrauch und Rechenaufwand. Besonders vorteilhaft zeigten sich die Möglichkeiten zur Extraktion welche sich durch die Natur des Modells ergeben. Dies entspricht genau unseren Absichten Massenabfragen auf XML mit Hilfe von XPath umsetzen. Dennoch bieten auch SAX-basierte XPath-Implementaitonen gewisse Vorteile und könnten sich bei besonders großen XML-Dokumenten als überlegen erweisen. Eine Erweiterung von PANDA die auf verschiedene Parser in Abhängigkeit von der Größe des Eingabe-Dokumentes setzt ist durchaus denkbar.

6. Design und Implementierung eines Prototypen

Dieses Kapitel befasst sich mit der Modellierung und Realisierung des exemplarischen Prototypen von PANDA. Der erste Schritt ist die Modellierung wichtiger Schritte vom Verhaltens des Systems mit Hilfe der UML[19]. Es folgt eine Modellierung des strukturellen Aufbaus von PANDA wobei ebenfalls die UML als Hilfsmittel eingesetzt wird.
Anschließend an die Modellierung werden die genutzten Frameworks und APIs kurz vorgestellt. Außerdem werden die in PANDA eingesetzt Verfahren für das Caching in einem eigenen Abschnitt beschrieben.
Zum Abschluss des Kapitels wird ein Vergleichstest in Form eines Benchmarks durchgeführt um Rückschlüsse auf die Effizienz verschiedener Implementierungen für den Zugriff aus gleiche oder ähnliche Dateiformate ziehen zu können.

6.1. Verhaltensmodellierung

Ein Anwendungsfalldiagramms schafft ein grobes Verständnis für die Aufgaben die das System erfüllen soll. Das Anwendungsfalldiagramm von PANDA in Abbildung 6.1 beinhalten 4 verschiedene Anwendungsfälle und einen der zusätzlich erweitert wurde
Der Benutzer steht stellvertretend für einen Client, wobei es sich um einen Webbrowser oder ein anderes HTTP-fähiges System handeln kann. Der Client sendet HTTP-Anfragen an das System von PANDA, welches dann eine entsprechende Antwort zurückliefert. Ist der Client REST-konform, so kann dieser Anwendungsfall erweitert werden. Wenn das Client-Caching vom System erlaubt wird, so kann mit den Cache-Kontrollinformationen des Clients sein Cache vom System validiert werden. Je nach Ergebnis kann der Client-Cache oder der Server die Antwort auf die Anfrage liefern.
Der Administrator kann entweder ein Mensch oder ein anderes System sein. Er kann Systemeinstellungen ändern, neue Dateien für das Deep-Linking hinzufügen oder bereits vorhandene Dateien aus dem System entfernen.

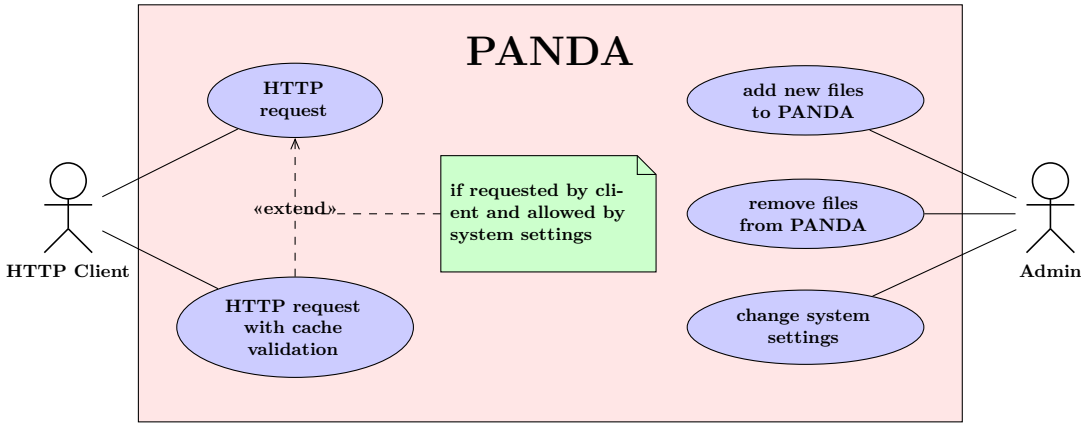


Abbildung 6.1.: Anwendungsfalldiagramm

Die wichtigsten Aufgaben des Systems von PANDA werden durch den erweiterten Anwendungsfall einer HTTP-Anfrage inklusive Validierung des Client-Cache abgedeckt. Dieser Anwendungsfall wird daher in einem Aktivitätsdiagramm (siehe Angang Seite 92, Abbildung B.1) genauer spezifiziert.
Ausgangspunkt des Anwendungsfall ist eine Anfrage des Clients an die REST-Schnittstelle von PANDA mittels HTTP-GET. Das System muss anhand der gesendeten Informationen entscheiden, ob der Client-Cache validiert werden muss. Wird die Validität bestätigt, so sendet der Server den Statuscode *HTTP 304* an den Client zurück, welcher daraufhin die Antwort auf die Anfrage aus seinem eigenen Cache entnimmt. Falls der Client keine Informationen für die Validierung des Client-Cache gesendet hat oder der Cache-Inhalt als nicht valid bewertet wurde, so wird im nächsten Schritt geprüft ob der Server-Cache aktiviert ist. Ist dies der Fall, so wird die Anwort auf die Anfrage aus dem Server-Cache generiert. Um das Modell nicht zu überladen gehen wir an dieser Stelle davon aus, dass die Daten bereits in den Cache geschrieben wurden sofern sich diese auch im Ausgangsdokument befinden.
, bereits in valider Form im Cache befinden. Falls das Server-Caching deaktiviert ist, wird die Standardvariante genutzt. Dabei werden die Daten direkt aus dem Dokument auf das sich die Anfrage bezieht extrahiert. Für den Fall, dss sich keine der Anfrage entsprechenden Daten im Dokument und folglich auch im Server-Cache befinden, wird der Statuscode *HTTP 404* zurückgegeben. Ist das Ergebnis positiv, so werden die Daten im gewünschten MediaType (wenn vom Server bereitgestellt) und der Statuscode *HTTP 200* an den Client zurück gesendet.

6.2. Strukturmodellierung

Strukturdiagramme wie das Paketdiagramm oder das Klassendigramm geben einen Überblick über die Umsetzung eines Systems. Das allgemeine Paketdiagramm in Abbildung 6.2 enthält drei Unterpakete. Das Paket ***panda.enums*** soll als Sammlung für die

gleichnamigen Aufzählungstypen dienen. Derzeit beinhaltet es lediglich eine Aufzählung unterstützer Datenformate.
Wichtiger sind die Pakete ***panda.api*** und ***panda.data***. Diese trennen die Schnittstelle des Systems, welche nach den Paradigmen von REST designed wurde, von Datenschicht. Das Paket *panda.api* definiert im Unterpaket ***panda.api.jersey*** die REST-Schnittstelle des Systems. Das Unterpaket ***panda.api.data*** beinhaltet dagegen Schnittstellen für die Kommunikation mit den Klassen der Datenschicht.
Das Paket *panda.data* ist für die Konfiguration des Systems sowie die Extraktion von Daten aus den Dateiformaten oder dem Server-Cache verwantwortlich. Im Paket ***panda.data.caching*** befindet sich die Implementierung des Server-Cache. Seine Funktionsweise wird im Unterabschnitt 6.4.2 beschrieben. Das Paket ***panda.data.configuration*** beinhaltet neben allgemeinen Einstellungen in erster Linie Klassen für die Verwaltung von physikalischen Ressourcen. In unserer Implementierung handelt es sich dabei um die Dateien in den jeweiligen Dateiformaten. Das Paket beinhaltet dazu Klassen, welche die Struktur der Konfigurations-Datei gemäß JAXB-Notationen bestimmen. Des Weiteren stellt es eine eigenständige GUI-Anwendung für die einfache Erstellung der Konfigurations-Datei zur Verfügung. Das dritte Unterpaket ***panda.data.resources*** beinhalten die Klassen, welche den Zugriff und die Extraktion auf die physikalischen Dateien umsetzen.

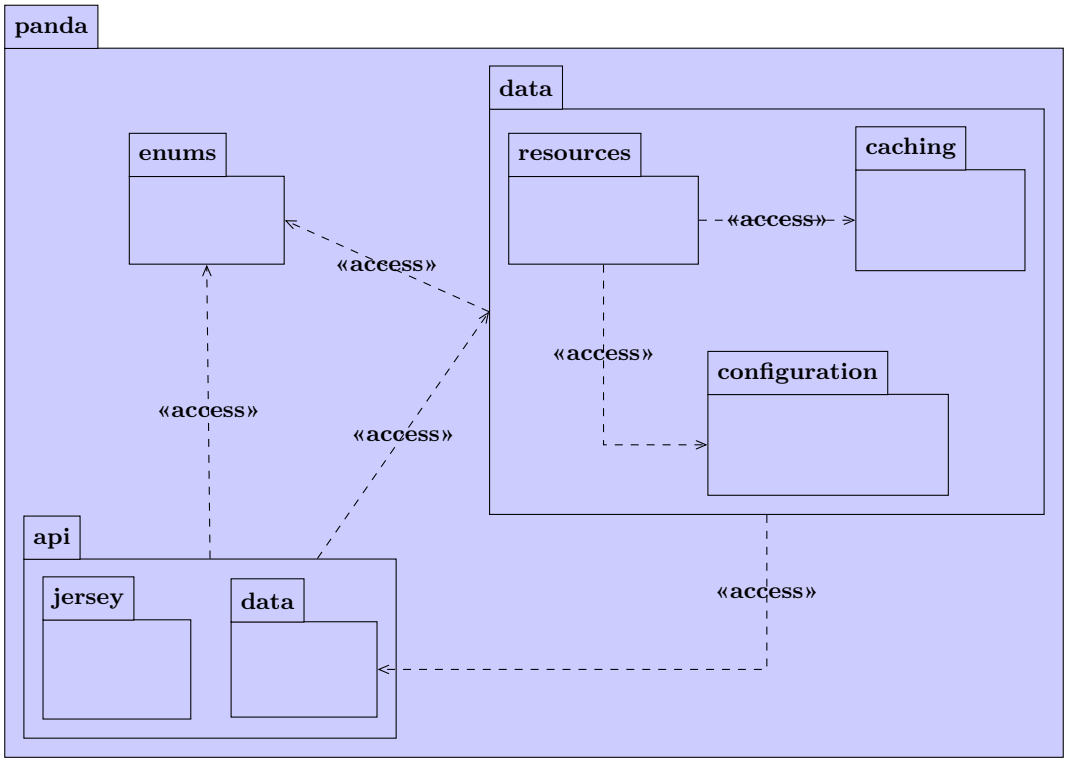


Abbildung 6.2.: Paketdiagramm von PANDA

Für ein besseres Verständnis des Systems, werden die drei wichtigten Unterpakete im folgenden noch weiter verfeinert und als Klassendiagramm abgebildet. Dabei werden Details wie Attribute und Methoden wegen des hohen Umfangs weggelassen.

panda.data.configuration (Klassendiagramm siehe Anhang Seite 93, Abbildung B.2):
Dieses Paket beinhaltet wie beschrieben alle notwendigen Klassen für die Verwaltung der verlinkten Datenformate. Das Unterpaket *panda.data.configuration.creator* enthält die separate GUI-Anwendung für das Erstellen der Konfigurationsdatei, in welcher die für Zugriffe notwendigen Metadaten einer Entität eines Datenformates abgespeichert werden. Hauptklasse der Anwendung ist **ResouceMapCreator**. Sie übernimmt Operationen wie Laden und Speichern der Konfigurationsdatei. Außerdem ist sie der zentrale Punkt von dem aus die GUI-Klassen aufgerufen werden. Die Klasse **MainWindow** definiert das Aussehen des Hauptfensters der Anwendung. Für die Darstellung der als Hashtabelle repräsentierten Map von Ressourcen wird die Klasse **ResourceTabelModel** benötigt. Diese wird auch in anderen Klassen für die tabellarische Darstellung der Ressourcen genutzt. Die Klasse **AddLocalFileDialog** wird als eigenständiges Fenster geöffnet. Mit ihm können Dateien, welche sich lokal auf der Festplatte befinden, der Konfiguration hinzugefügt werden. Eine spezieller Variante bietet das Fenster der Klasse **AddResourceDiaglog**, mit dem auch sonstige Resourcen wie z.B. solche die über eine URL abgerufen werden, zur Konfiguration hinzugefügt werden können. Für die Einschränkung seiner Texteingabefelder wird die Klasse *TextFieldLimiter* verwendet.
Das Unterpaket *panda.data.configuration.resources* definiert die JAXB-Klassen für die Datenbindung zwischen Java-Objekten und XML. Diese Werden für das Laden und Speichern genutzt und dienen Zeitgleich als Vorgabe für das Schema der Konfigurationsdatei. Die Gesamtstruktur der Konfigurationsdatei wird durch die Klasse **ResourceMapType** festgelegt. Diese benötigt für die Präzisierung die Klasse **ResourceMapEntryType**, welche die Struktur der einzelnen XML-Knoten repräsentiert. JAXB erlaubt leider nicht das direkte Abbilden von Hashtabellen auf XML, weshalb die Klasse **ResourceMapXmlAdapter** zu präzisierung der XML-Struktur genutzt werden muss. Die Klasse **ResourceMap** fasst die Definitionen zusammen und dient der Verwaltung der Hashtabelle mit den Einträgen.
Außerhalb des Paketes befindet sich die Klasse **PandaSettings**. Diese enthält neben allgemeinen Einstellungen die Instanz von *ResourceMap*, welche von PANDA mittels Datenbindung geladen wird.

panda.data.resources (Klassendiagramm siehe Anhang Seite 95, Abbildung B.4):
Dieses Paket beinhaltet voneinander weitestgehend unabhängige Klasse, die jeweils eine Form des Zuriffs auf ein bestimmtes Datenformat umsetzen. Dabei müssen sie sich streng an die Definition der URIs halten. Diese liefern zum einen die notwendige ID für die Abfrage der Metainformationen für den Zugriff auf eine Ressource und zum anderen enthalten sie Strukturinformationn die für die Einschränkung bzw. Filterung der Daten genutzt wird.
Die Klassen **DataExcelVtdResource** und **DataExcelResource** dienen für den

Zugriff auf Excel-Dokumente. Erstere funkioniert dabei ausschließlich für das neuere OOXML-Format. Sie arbeitet mit VTD-XML direkt auf den zugrundeliegenden XML-Dateien während die zweite Variante das Framework „Apache POI“ für den Zugriff nutzt. Diese Vairante erlaubt außerdem den Zugriff auf das alte Office97-Format, welcher sich dank einheitlicher Interfaces von POI kaum von der anderen Vairante unterscheidet.
Die beiden Klssen **DataWordVtdResource** und **DataWordResource** setzten in ähnliche Weise auf VTD-XML und Apache POI. Im Unterschied zu Excel sind die Interfaces von Apache POI für das Office97 und OOXML-Format nicht einheitich. Für Vergleichszwecke wurden dennoch alle Varianten implementiert.
Die Klassen **DataCsvResource**, **DataPdfResource**, **DataXmlResource** und **DataHtmlResource** setzen jeweils wie gehabt die Extraktion von Daten gemäß Definition der URIs um. Die HTML-Klasse arbeitet nicht vollständig unabhängig. Sie ruft nach Bereinigung des HTML-Dokumentes die XML-Klasse. Diese Umsetzung soll redundanten Code vermeiden.
Die Klasse **ResourceHelper** enthält statische für Berechnungen die von mehreren Klassen des Paketes benötigt werden. Diese wurden ebenfalls zur Vermeidung von Redundanz ausgelagert.

panda.data.resources (Klassendiagramm siehe Anhang Seite 94, Abbildung B.3):
Das letzte an dieser Stelle genauer beschriebene Paket enthält verschiedene Klassen, welche die REST-Schnittstelle definieren. Zentralle Klasse ist **JerseyRootResource**. Diese ist bei HTTP-Anfragen die erste Stelle die passiert werden muss. Sie entscheidet, welche der anderem Klassen für die Verarbeitung der Anfrage in Frage kommt. Für verschiedene Datenformate wurde bis jetzt die Klassen **JerseyExcelResource**, **JerseyWordResource**, **JerseyHtmlResource**, **JerseyXmlResource**, **JerseyPdfResource** und **JerseyCsvResource** implementiert. Das System lässt sich dadurch einfach um andere Datenformate erweitern. Alle Ressourcen-Klassen erweitern die Klasse **AbstractJerseyResource**, in der einheitliche Methoden wie die Validierung vom Client-Cache implementiert werden sollen. Die Klasse **PandaJerseyConfig** ist eine obligtorische Klasse die für eine JerseyApplikation implementiert sein muss. Sie konfiguriert das Servlet in dem Jersey läuft. Darüber hinaus können hier Einstellungen für „Dependency Injection“[18] zwischen den Objekten des Servlets vorgenommen werden. In PANDA wird dies zur Verteilung der Konfiguration genutzt. Diese wird zur Laufzeit in die Jersey-Klassen injiziert.

6.3. Eingesetzte Frameworks und APIs

Die exemplarische Implementierung der verschiedenen Dateiformate erforderte den Einsatz eines breiten Feldes an verschiedenen Frameworks und APIs. Die Grundlage des gesamten Systems bildet das Framework **Jersey RESTful Web Services**[1], welches als Referenzimplentation von REST für Java entwickelt wurde. Jersey setzt das REST-Paradigma sowohl für Clients als auch für Server um. REST entspricht in vielen Punkten

[1] *https://jersey.java.net/*

den Anforderungen die an das System von PANDA gestellt werden und sollte daher konsequent genutzt werden. In PANDA wurde in erster Linie auf die Implementierung von Jersey für Server gesetzt. Diese wird als Servlet in einen Webserver eingebettet wird. Jersey erlaubt die Definition von Klassen und Methoden, welche bei Anfragen anhand von URI-Muster und weiteren Parametern, z.B. für die Content-Negotiation, ausgewählt werden um eine Antwort auf die Anfrage zu generieren. Diese Klassen werden selbst als Ressourcen bezeichnet, wobei eine Ressource entsprechend der dynamischen URI-Muster eine Vielzahl von Sub-Ressourcen beinhalten kann. In PANDA wurde Jersey in den Webserver des **Apache HTTP Server Project**[2] eingebettet. Zusätzlich bietet Jersey ein eigenenes Testframework, mit dem „In-Memory-Tests“ erstellt werden können. Dabei wird ein Webserver simuliert, sodass dieser für den Test des Servers nicht notwendig ist.
An verschiedenen Stellen des Projektes wurde es notwendig Java-Klassen auf XML abzubilden. Für die XML-Datenbindung wurde zu diesem Zweck die Schnittstelle **Java Architecture for XML Binding (JAXB)**[3] eingesetzt. JAXB ist bereits vorab in Jersey integriert und kann dort vergleichsweise einfach für die Generierung von Antworten mit XML-MIME-Typ genutzt werden. Darüber hinaus wurde es für die Umsetzung des „Ressource-ID“-Systems eingesetzt. Die einer Ressource-ID zugewiesenen Zugriffsdaten werden in einer Liste als XML-Datei gespeichert. JAXB wird sowohl zum Erstellen dieser Datei als auch zum Laden dieser Datei als Objekt einer Java-Klasse eingesetzt.
Als Erweiterung zu JAXB wurde der **Jackson JSON Processor**[4] für JSON-Datenbindung eingesetzt. Dieser ist zu den JAXB Notationen kompatibel und erlaubt dadurch eine schnelle Erweiterung der optionalen MIME-Typen für die Antworten um JSON.
Für die Verarbeitung von CSV-Dateien wurde das Framework **Apache Commons CSV**[5] genutzt. Dieses beinhaltet unter anderem einen RFC4180 entsprechenden Parser.
Für die Verarbeitung von XML-Dokumenten bzw. XML-basierten Formaten wurde das Projekt **Virtual Token Descriptor for XML (VTD-XML)** eingesetzt. Eine Ausführliche Beschreibung ist im Abschnitt 5.4 zu finden. Um HTML für die XML-Verarbeitung kompatibel zu machen, wurden sowohl **JTidy**[6] als auch **HtmlCleaner**[7] genutzt. Beide dienen der Bereinigung vom HTML-Format um es zum XML-konformen XHTML-Format zu überführen.
Für die MS-Office Formate wurde verschiedene Lösungen implementiert, die für Vergleichszwecke herangezogen werden sollen. Neben einer Low-Level-Lösung welche auf VTD-XML setzt, wurde jeweils eine Lösung basierend auf der API **Apache POI** [8] umgesetzt. Das Projekt gliedert sich in verschiedene Frameworks für jeweils ein spezifische Format. Neben einer alternativen Lösung für die OOXML-Formate konnte daher

[2] *http://httpd.apache.org/*
[3] *https://jaxb.java.net/*
[4] *https://github.com/FasterXML/jackson*
[5] *http://commons.apache.org/proper/commons-csv/*
[6] *http://jtidy.sourceforge.net/*
[7] *http://htmlcleaner.sourceforge.net/*
[8] *http://poi.apache.org/*

auch eine Lösung für beiden Office97-basierten Formate von Word und Excel umgesetzt werden.
Für die Verarbeitung von PDF-Dokumenten wird die Bibliothek **Apache PDFBox**[9] eingesetzt. Dieses ist in zwei Modell-Ebenen gegliedert. Das Low-Level COS-Model bietet die Möglichkeit direkt auf Teile des PDF-Dokumentes zuzugreifen. Darauf aufbauend wurde das High-Level PD-Model implementiert, welches die Komplexität von PDF nach außen hin kapseln soll.

6.4. Caching

Eine Möglichkeit die Performance signifikant zu steigern ist das Caching. Wir unterscheiden in PANDA zwischen dem Caching auf Client-Seite und auf Server-Seite die in den folgenden Abschnitten erläutert werden.

6.4.1. Client-Cache

Die aktive Nutzung von Cache-Kontrollinformationen und Bereitstellung der Grundlagen für die Nutzung von Caching auf Seite des Clients gehört zu den Paradigmen von REST. Die Anforderungen von REST beinhalten das Senden von notwendigen Informationen zur Cache-Kontrolle wenn immer dies möglich ist. Dadurch wird eine konsequente Nutzung des in RFC2616[8] beschriebenen Cachings gewährleistet.
Von einem Client-Cache wird verlangt, dass dieser immer mit der aktuellsten der Anfrage entsprechenden Version antwortet. Um dies zu gewährleisten gibt es zwei Methoden: (1) Die Antwort auf eine Anfrage bekommt in einem Header-Feld ein Verfallsdatum für dessen Gültigkeit mitgeliefert. (2) Die Äquivalenz von Client-Cache und einer erneuten Antwort wird validiert.
Die erste Möglichkeit ist für PANDA wenig sinnvoll, da wir nicht wissen welche Auswirkungen fehlende Konsistenz der Daten beim Client hat. Daher ist es sinnvoller auf eine explizite Validierung zu setzen.
Der Code für die Validierung vom Client-Cache auf Server-Seite ist in Listing 6.1 abgebildet. Dieser setzt die in RFC2616 empfohlene Methode der Validierung durch das Header-Feld „Last-Modified“ und durch die Nutzung von „Entity Tags (ETags)“ um. Das Feld Last-Modified enthält ein HTTP-Datum das z.B. so aussehen könnte: *Last-Modified: Tue, 15 Nov 1994 12:45:26 GMT*. Der Server vergleicht den vom Client gesendeten Wert mit seinem eigenen und sendet die Ressource erneut, falls diese nicht übereinstimmen. Diese Art der Validierung ist jedoch schwach und daher ggf. unzuverlässig. Es besteht z.B. die Möglichkeit, dass eine Ressource mehrfach in der selben Sekunde geändert wird. Deshalb wird zusätzlich ETags zur Validierung genutzt. Gemäß Definition repräsentieren diese jeweils genau eine bestimmte Revision einer Ressource. Dabei hat man die Wahl schwache oder starke ETags einzusetzen. Starke ETags müssen sich bei jeder Änderung der Ressource ändern, während schwache ETags dies nur bei signifikanten semantischen Änderungen tun sollen. In PANDA werden starke ETags genutzt, die sich aus der ID der

[9] *https://pdfbox.apache.org/*

Ressource und dem genauen Datum der letzten Änderung in Millisekunden zusammensetzt. Es ist nicht festgelegt woher dieses Datum stammt. Da wir bei direktem Verlinken auf Daten von Dateien davon ausgehen müssen, dass diese extern z.B. durch einen Texteditor modifiziert werden könnten, greifen wir auf das vom Betriebssystem gespeicherte Datum der letzten Modifizierung zurück.

```
// Ermittle Datum der letzten Modifikation
Date lastModified = new Date(resource.lastModified());
// erstelle ein ETag zur Validierung
EntityTag entityTag = new EntityTag(resourse.getID() + "_"
    + lastModified.getTime(), false);
// validiere Last-Modified und ETag
ResponseBuilder response = request.evaluatePreconditions(
   lastModified,entityTag);
if (response != null) {
    // Sende HTTP 304 wenn Ressource nicht modifiziert
    return response.cacheControl(pandaSettings.
       getCacheControl()).lastModified(lastModified).tag(
       entityTag).build();
} else {
    // Generiere neue Antwort wenn Ressource modifiziert
    ...
}
```

Listing 6.1: Validierung des Client-Cache auf dem Server

Nach der Validierung des Client-Cache kann der Server die Anfrage mit dem Status-Code „HTTP 304“ beantworten, um den Client zu signalisieren, dass die Ressource nicht verändert wurde und die Anfrage aus seinem Cache beantwortet werden kann. Andernfalls wird der Server wie gehabt fortfahren und die Anfrage erneut beantworten, wobei die Antwort wie von REST vorgesehen nach Möglichkeit alle nötigen Felder zur Cache-Kontrolle enthält.

6.4.2. Server-Cache

Im Gegensatz zum Client-Cache setzt der Server-Cache keinen REST-konformen Client voraus und ist folglich immer anwendbar. Der Client-Cache funktioniert recht simpel indem er Antworten zu HTTP-Anfragen vollständig cached und mittels Informationen aus dem HTTP-Header zusammen mit dem Server eine Validierung des Caches vornehmen kann. Ein solches Design ist für den Server-Cache nicht denkbar, da dieser nur auf seine eigenen begrenzten Ressourcen setzen kann. Jede Kombination von Zugriffsinformationen bildet eine eigenständige URI und eine entsprechende Antwort die gecached werden muss. Dadurch entsteht eine unerwünschte Redundanz gespeicherter Werte. Der Server-Cache muss daher einen anderen Ansatz verfolgen.
Das Hauptproblem für niedrige Geschwindigkeit ist die sich wiederholende Verarbeitung der Dateien. Diese müssen zunächst von der Festplatte oder anderen Orten geladen

werden. Anschließend erfolgt die Extraktion der Daten gemäß den in der URI kodierten Informationen. Die Geschwindigkeit variiert dabei von Datenformat zu Datenformat stark. Der Server-Cache versucht sowohl das Laden der Dateien zu umgehen, als auch deren langsame Verarbeitung vom Format zu entkoppeln um die Geschwindigkeit zu steigern. Zu diesem Zweck kann die Einfachheit der in dieser Arbeit erstellen URIs ausgenutzt werden. Die URIs haben unserer Definition nach keine Query-Komponente. Die verbleibende Pfad-Komponente bildet eine Hierarchie. Aufgabe des Server-Caches ist es, diese Hierarchie als eine geeignete Datenstruktur im Speicher abzubilden. Dazu werden zunächst die einzelnen Werte und die dazugehörigen URIs benötigt. Das System wurde bereits so ausgelegt, dass es neben den Werten auch die dazu jeweils passende URI ermittelt. Im Bezug auf Deep-Linking ist diese als Einzelabfrage mit maximaler Granularität zu sehen. Mit diesen Werten und URIs erstellen wir eine der Hierarchie entsprechenden Datenstruktur. Die in Abschnitt 4.2 definierten Regeln sorgen außerdem dafür, dass die einzelnen Segmente der URIs selbst eindeutig sind. Durch diesen Fakt ist die Verwendung von schnellen Hashtabellen möglich, wobei die Segmente als Schlüssel dieser Hashtabelle dienen.
Ist eine Datei erst einmal in den Cache-Baum aufgenommen worden, so können Anfragen direkt auf diese Struktur statt auf die ursprüngliche Datei angewendet werden. Die URIs müssen dabei ähnlich aufgelöst werden wie es bei der Verarbeitung einer entsprechenden Datei getan worden wäre. Einzelne Segmente können durch die Hashtabellen besonders schnell aufgelöst werden. Die aufwändigen Vergleiche von Strings entfallen dabei. Bei der Anwendung von Massenabfragen aus den Cache-Baum ist eine Sortierung der Knoten hilfreich. Die Sortierung erfordert zwar zusätzlichen Aufwand beim Hinzufügen der Daten, jedoch ist die Geschwindigkeit beim Lesen als weitaus kritischer zu betrachten. Sortierte Werte erlauben es die Suche eines festgelegten Bereiches wie z.B. einer Tabelle in aus einem Excel-Dokument zu beschränken und sie nach dem Auftreten des letzten Elementes des gesuchten Bereiches abzubrechen. Dadurch müssen nicht alle Kinder eines Knotens für eine Selektion durchsucht werden.
Der Cache-Baum bietet maximale Effizienz auch bezüglich Speicherverbrauch. Er ist deutlich kleiner als die Ausgangsdatei, weil alle nicht durch die URIs referenzierten Informationen fehlen. Die Struktur wird optimal für das Referenzieren ausgenutzt, da die Hierarchie exakt abgebildet wird und es unserer Definition nach keine Redundanz dabei geben darf. Die Werte sind jeweils nur einmal in jeweils einem Blatt des Baumes gespeichert.
Diese Form des Cachings hat jedoch auch Nachteile. Die Auswertung der URIs muss auf den Cache-Baum abgebildet werden. Für XML bedeutet dies z.B. dass der gesamte XPath-Standard auf den Cache-Baum angewendet werden müsste um äquivalente Antworten aus dem Cache zu generieren. Diese Aufgabe ist im Rahmen dieser Arbeit nicht umsetzbar weshalb für solche Fälle auf eine deutlich einfachere Variante ausgewichen wird. XML-Dateien werden dabei als in ihrer ursprünglichen Form im Cache gehalten. Dadurch entfällt der Teil der Verarbeitung, der oft die meiste Zeit in Anspruch nimmt: das Laden der Datei von der Festplatte. Mit VTD-XML (siehe Seite 51 Abschnitt 5.4) steht uns zudem eine effiziente Möglichkeit zur Verarbeitung zur Verfügung, die unter geringem Mehraufwand an Speicher das Cachen von bereits durch den Parser verarbei-

teten Objekten ermöglicht. Auf diese Weise muss bei einer Anfrage auf eine im Cache liegende XML-Datei lediglich der XPath-Ausdruck selbst ausgewertet werden.
Ein anderer Nachteil ist die Notwendigkeit einer Strategie für das Caching. Für die Invalidierung des Caches wurden in PANDA lediglich die Metainformationen der verlinkten Dateien genutzt. Es ist jedoch nicht nur erforderlich veraltete Informationen aus dem Cache zu entfernen, sondern auch die Informationen die sich im Cache befinden gezielt auszusuchen. Bei einer großen Anzahl von Dateien im System muss die Anzahl der Dateien im Cache begrenzt werden. Eine entsprechende Strategie könnte beispielsweise auf Informationen wie Zugriffshäufigkeit und Datum des letzten Zugriffes setzen. Wegen des Fehlens einer solchen Strategie ist der Server-Cache von PANDA noch nicht praxistauglich.

6.5. Vergleich der Implementierungen durch Benchmarks

In diesem Abschnitt werden die verschiedenen Implementierungen miteinander durch Benchmarks verglichen. Dabei soll vor allem die Frage geklärt werden, wie gut sich die Formate und die implementierten Varianten der Verarbeitung für den Einsatz in der Realität eignen. Bei allen Tests wurde je nach Dauer des Tests über 100 oder 1000 sequentiellen Anfragen iteriert und ein Durchschnitt berechnet um Schwankungen durch Seiteneffekte zu minimieren.

6.5.1. Testvorbereitung

Der erste Schritt ist die Auswahl geeigneter Testkandidaten. Zu diesem Zweck nutzen wir wieder reale Daten von Plattformen für offene Daten. Dabei wird jeweils ein Kandidat für die Tabellenkalkulation und für Dateiformate aus der Textverarbeitung ausgewählt. Diese werden in die jeweils fehlenden Formate konvertiert. Dadurch lassen sich die verschiedenen Implementierungen zuverlässig bezüglich ihrer Verarbeitungsgeschwindigkeit miteinander vergleichen. Die ausgewählten Dokumenten in ihrem Ursprungsformat sind in Tabelle 6.1 aufgelistet.

Testdatei	Umfang
July_Spend_Publication_v1.0.doc *http://publicdata.eu/dataset/public-sector-procurement-spend/resource/f24f631b-82d3-4d2f-b8f0-e34926087f9a*	3 Seiten Text, 25 Tabellen mit insgesamt 19515 Zellenwerten
Driving-Licence-Tables-Nov2012.xls *http://data.gov.uk/dataset/driving-licence-data/resource/51efb877-14b3-4e5f-80cf-b2a8c5532cc8*	115054 Zellenwerte
Digital Elevation Models (DEM) - New York State.htm *http://cugir.mannlib.cornell.edu/transform?xml=36dea.xml*	36KB

Tabelle 6.1.: Testkandidaten für den Vergleich der Geschwindigkeit verschiedener Varianten der Verarbeitung von Dokumenten in PANDA

6.5.2. Durchführung

Für die Durchführung wird das gleiche Testsystem[10] wie schon bei der Evaluierung der XML-Parser genutzt. Auf diesem soll das PANDA-System und der Apache Webserver laufen. Die Zeit wird auf einem externen Client getestet, welcher mittels Gigabit-Ethernet direkt verbunden wird. Die Zeitangaben der Tests beinhalten folglich alle Schritte, von der HTTP-Anfrage an die REST-Schnittstelle von PANDA über die Verarbeitung der Datei aus der Daten angefragt wurden, bis hin zur vollständigen Antwort an den Client.
Der erste Test soll die Effizienz der Verarbeitung von Dateiformaten aus der Textverarbeitung ermitteln. Dazu werden alle 3 Implementierungen für die Formate von MS-Word untereinander und mit der Implementierung für PDF verglichen. Zu den 3 Implementierung für MS-Word gehören (1) die auf Apache POI basierende Variante für das alte Office97-Format, (2) die auf Apache POI basierende Variante für das neue OOXML-Format und (3) die auf VTD-XML basierende Low-Level Variante. Für das PDF-Format wird wie gehabt das Framework PDFBox eingesetzt. Zusätzlich wird die Geschwindigkeit der Implementierung des Server-Cache von PANDA für die gleichen Anfragen ermittelt. Der Test beinhaltet zum einen die Anfrage einer Seite reinen Textes und zum anderen die Anfrage von mehreren Tabellen. Der Tabellen-Test kann aus den in Abschnitt 4.6 genannten Gründen nicht durchgeführt werden.
Der zweite Test ist ähnlich aufgebaut wie der erste Test. Auch hier werden die 3 Implementierungen für die Verarbeitung von zu MS-Excel gehörenden Dateien miteinander verglichen. Zusätzlich wird auch in diesem Test die Implementierung des Server-Cache von PANDA zum Vergleich herangezogen.
Der letzte Test soll die Frage klären, welches der beiden in der Implementierung genutzten Werkzeuge am besten für die Bereinigung von HTML-Dokumenten geeignet ist, sodass sich diese die Verarbeitung mit XML-Parsern erlauben. Der Test beinhaltet je-

[10] Intel Core i5-4670k, 8GB DDR3, Crucial M4 SSD

weils die Nutzung von JTidy und HtmlCLeaner für die Bereinigung der Testdatei. Zum Vergleich wurde die gleiche HTTP-Anfrage an die gleiche bereits gereinigte Datei gestellt, um den Mehraufwand für die Bereinigung gegenüber XML-konformen Dateien zu ermitteln.

6.5.3. Ergebnisse

Die Ergebnis der Tests (siehe Angang Seite A.12, Abbildung A.12) für den Vergleich von Dateiformaten aus der Textverarbeitung zeigt, dass sich das alte Office97-Format von beiden Varianten für das neuere OOXML-Format in Sachen Geschwindigkeit der Verarbeitung absetzen kann. Da dieses Format auf lange Sicht aussterben wird, ist die Verwendung trotzdem nicht zu empfehlen. Beim OOXML-Format konnte eine signifikante Steigerung gegenüber der Apache POI-Variante erreicht werden. Durch den Einsatz des schnellen Parsers von VTD-XML und gezielter Ausrichtung auf die Extraktion der Daten wurde die Dauer für die Anfrage einer Seite Text aus dem Testdokument auf beinahe 25% gesenkt. Die Dauer der Anfrage der Tabellen konnte immerhin auf etwa 60% der ursprünglichen Dauer gesenkt werden. Die Verarbeitung des PDF-Formates ist sogar ein wenig schneller als die langsamere Verarbeitung des OOXML-Format durch POI. Die fehlende Möglichkeit eine zuverlässige Extraktion von Tabellen in PDF umzusetzen ist jedoch negativ zu werten. Der Server-Cache von PANDA liefert wie erwartet das schnellste Ergebnis. Er benötigt im Vergleich zu VTD-XML etwa 27% für eine Seite Text und gerademal 15% der Zeit für die Abfrage der Tabellen. Ein Einsatz ist demnach durchaus lohnenswert.
Die Ergebnisse vom zweiten Test (siehe Anhang Seite 89, Abbildung A.13) für Dateiformate für Tabellenkalkulation zeigt erwartungsgemäß auch hier einen Geschwindigkeitsvorteil für die VTD-XML nutzende Variante gegenüber Apache POI. Insbesondere die Anfrage von wenigen Werten konnte mit einer Dauer von ca. 5% der ursprünglichen Dauer stark beschleunigt werden. Die Anfrage aller Werte des Dokumentes dauert immerhin noch 61% des ursprünglichen Wertes. Das Office97-Format kann hier im Vergleich nur bei Abfrage aller Werte leichte Geschwindigkeitsvorteile vorweisen. Die Abfrage einer einzelnen Zelle dauert dagegen mehr als fünfmal so lang wie die selbe Abfrage auf das OOXML-Dokument durch VTD-XML. Der Server-Cache von PANDA ist auch hier am schnellsten, kann sich jedoch nicht mehr so stark absetzen wie im ersten Test. Dies mag vor allem daran liegen, dass die Struktur von SpreadsheetML weniger tief verschachtelt ist und eine deutlich schnellere Auswertung der XML-Dateien durch die Parser zulässt.
Das Ergebis des letzten Test (siehe Anhang Seite 90, Abbildung A.14) für die Verarbeitung von HTML bescheinigt JTidy einen leichten Geschwindigkeitsvorteil gegenüber HtmlCleaner. Der Vergleich zur bereits vorab gereinigten XHTML-Datei zeigt, dass der zusätzliche Aufwand nur etwas mehr als 2ms beträgt. Folglich kann für HTML bedenkenlos die gleiche Verarbeitung wie schon für XML eingesetzt werden.

7. Schlussfolgerung

In dieser Arbeit wurde exemplarisch an einer Reihe ausgewählter repräsentativer Datenformate gezeigt, wie durch Analyse der Datenstruktur und mit technischen Hilfsmitteln für die Datenverarbeitung Daten aus Dokumenten extrahiert und durch URIs adressierbar gemacht werden können.
Die erstellten URI-Muster können als Grundlage für die Implementierung weiterer Datenformate gesehen werden. So würde sich die Notation eines URI-Musters für die „JavaScript Object Notation (JSON)“[1] durch den Einsatz von JsonPath[2] sehr mit der des XML-Formates ähneln. Andere Strukturen in denen häufig Tabellen verwendet werden könnten wiederum dem URI-Muster, das für OOXML-Dokumente verwendet wird folgen.
Die Arbeit hat gezeigt, dass es verschiedene Wege gibt Informationen in eine URI zu kodieren. In der Arbeit wurden beide Varianten an sinnvollen Stellen eingesetzt. Die direkte Variante wurde eingesetzt, um variable Informationen die zu einer gezielten Extraktion von Daten aus einem Datenformat benötigt werden, in die URI zu kodieren. Die indirekte Variante wurde dagegen für Informationen verwendet, die nicht direkt die Datenstruktur betreffen sondern als Metainformationen für Zugriff und Verarbeitung dienen. Zu diesem Zweck wurde mit Hilfe von XML-Datenbindung durch JAXB ein entsprechender Mechanismus entwickelt, der für das Laden und Speichern dieser Informationen genutzt wird. Die exemplarische Umsetzung dieses Mechanismus zeigt, wie eine beliebige Menge von Zugriffsinformationen hinter einer generierten ID versteckt und in eine URI kodiert werden können. Dies erlaubt die Implementierung weiterer Datenformate mit spezifischen Zugriffsinformationen ohne von den bereits bekannten URI-Mustern stark abweichen zu müssen.
Wie beschrieben wurde, kann für keines der in dieser Arbeit umgesetzten Formate eine Stabilität der URIs bei Schreiboperationen auf den Inhalt garantiert werden. Die sich aus der Struktur ergebenden Merkmale zur Identifikation für einen Teil des Inhaltes, werden in der Regel durch den restlichen Inhalt bestimmt. Insbesondere das Einfügen und Löschen von Daten stellt hierbei ein Problem dar, während das Überschreiben oft problemlos möglich ist.
Bezüglich der Datenformate wurde festgestellt, dass PDF sich äußerst schlecht für die Extraktion von Daten eignet. Der Extraktionsprozess muss wegen fehlender Struktur dem gleichen Ablauf wie Programme für die Darstellung von PDF-Dokumenten folgen. Dieser Prozess ist rechenaufwändig und ungnau. Bestimmte Daten wie Tabellen lassen sich nicht in der Form extrahieren und verlinken wie es gewünscht wäre. Die Vielfalt

[1] *http://tools.ietf.org/html/rfc7159*
[2] *https://code.google.com/p/json-path/*

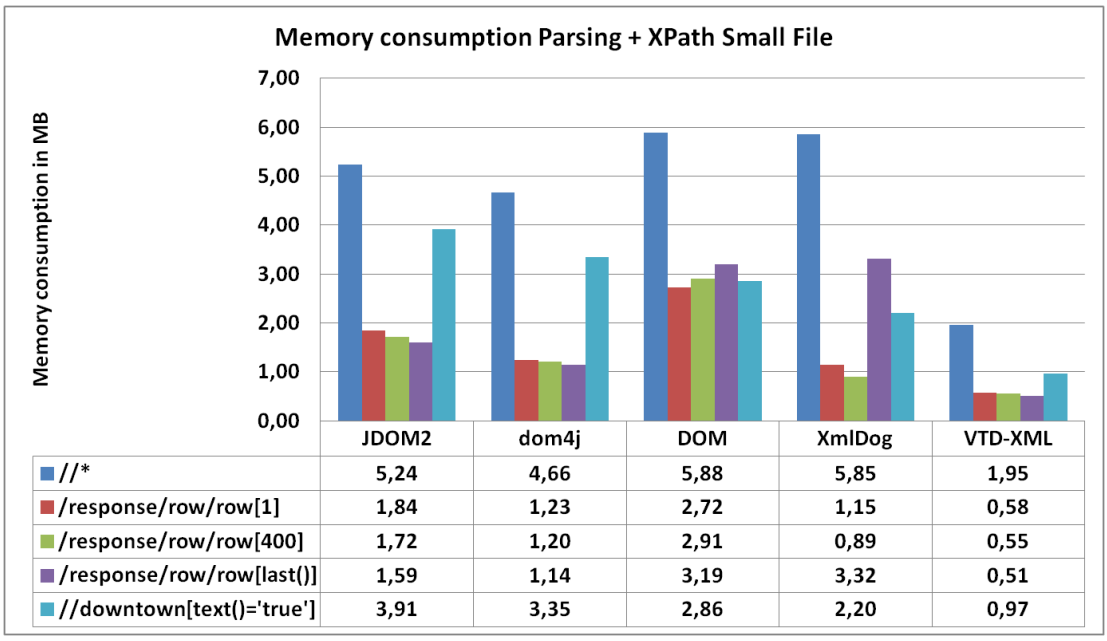
verschiedener Konstrukte die teilweise auch aus Gründen der Abwärtskompatibilität erhalten wurde und verschiedene Filter zur Datenkompression verhindern einen direkten Zugriff auf textueller Ebene und machen den Einsatz eines Frameworks zwingend notwendig.
Positiv lassen sich dagegen die XML-basierten Formate bewerten. Der OOXML-Standard gehört durch Microsoft Office und Apples iWork zu den am weitesten verbreitetsten. Daneben wird von freien Office Varianten wie Libre Office der OpenDocument-Standard eingesetzt. Für alle diese Formate steht die breite Palette von Werkzeugen für XML zur Verfügung. Durch effiziente Parser kann die Verarbeitung signifikant beschleunigt werden. Dies erlaubt uns bei diesen Formaten eine besonders effiziente Extraktion von Daten für das Verlinken. Die Office97-Formate lassen sich unseren Benchmarks nach zu urteilen sogar noch etwas schneller verarbeiten. Dennoch sind diese Formate als zweitrangig zu betrachten, da sie nach und nach vom neuen flexibleren OOXML-Standard verdrängt werden und deren Rolle bei der Verwendung für offene Daten übernehmen werden.
In der Arbeit wurde gezeigt, wie unter bestimmten Voraussetzungen an die URIs die Ergebnisse der Datenverarbeitung effizient gecached werden können. Der Cache bietet eine einheitliche Laufzeit für Formate mit einer statischen Anzahl Strukturinformationen. Der Vorteil gegenüber direkter Datenextraktion schwank in Abhängigkeit ihrer Effizienz.

8. Ausblick

Die Umsetzung und Integration von anderen Datenformaten in das Projekt kann nun nach Belieben fortgesetzt werden. Zudem könnten die URIs der vorhandenen Formate erweitert oder auf spezifische Anwendungsfälle zurecht geschnitten werden. Auch andere Anwendungsbereiche abseits vom Bereich „Open Data“, in denen das Deep-Linking von Daten sinnvoll erscheint sind denkbar. Dabei muss sich nicht auf Dokumente beschränkt werden, auch die Anwendung auf Datenbanken kann mit den in dieser Arbeit gezeigten Methoden umgesetzt werden.
Die Extraktion der Daten durch URIs kann neben dem Verlinken mittels REST-Schnittstelle auch für die Konvertierung der Daten eingesetzt werden. Mit PANDA wäre die Umsetzung automatisierter Datenextraktion und Überführung in SQL oder ähnlichem möglich.
Das Server-Caching muss für einen Einsatz in der Praxis um eine Strategie für Cache-Invalidation erweitert werden. Dazu müssten verschiedene Strategien und Metriken untersucht und ihre Auswirkung auf verschiedene Anwendungsszenarien analysiert werden. Das Verwaltungssystem für verlinkte Dateien ist derzeit noch sehr primitiv und beschränkt sich dabei auf das Dateisystem des Betriebssystems. Das Caching-System ist zudem auf Metainformationen des Dateisystems angewiesen die nicht zwangsläufig korrekt sind. Die Bearbeitung der Dateien durch externe Programme kann zudem zu unerwünschten Nebeneffekten wie inkonsistente Daten oder zeitweilige nicht Verfügbarkeit von Ressourcen führen. Eine Möglichkeit diese Probleme zu beheben und das System aufzuwerten ist der Einsatz eines Systems für Dokumentenmanagement mit integrierter Versionskontrolle.
PANDA wurde für das Verlinken von Daten mittels Deep-Links konzipiert. Der Fokus lag auf der Extraktion der Daten und der Bereitstellung der Daten mittels URI über eine REST-Schnittstelle. Schreiboperationen waren daher nicht vorgesehen. Diese führen wie im Kapitel 4 beschrieben zu Instabilität der URIs. Dennoch könnte es Anwendungsfälle geben, in denen eine mögliche Instabilität der URIs im Austausch für die Bereitstellung von Schreiboperationen hingenommen wird oder diese durch vorgegebene Bedingungen beschränkt werden. In diesem Fall ist ein Versionskontrollsystem zusätzlich für die Vermeidung von Datenverlust hilfreich.
Die Effizienz des Server-Cache kann noch weiter gesteigert werden. Die aktuelle Implementierung trägt alle Werte eines Dokumentes in den Cache-Baum ein. Auf diese Weise kann sichergestellt werden, dass bei einer Massenabfrage alle Werte die im Dokument gefunden worden wären, auch im Cache gefunden werden. Ein selektives Caching könnte dafür sorgen, dass nur häufig abgefragte Teile eines Dokumentes tatsächlich in den Cache ausgelagert werden. Problematisch daran ist jedoch bei Abfragen sicherzustellen, dass die im Cache befindlichen Informationen bezüglich der Abfrage vollständig sind. Es

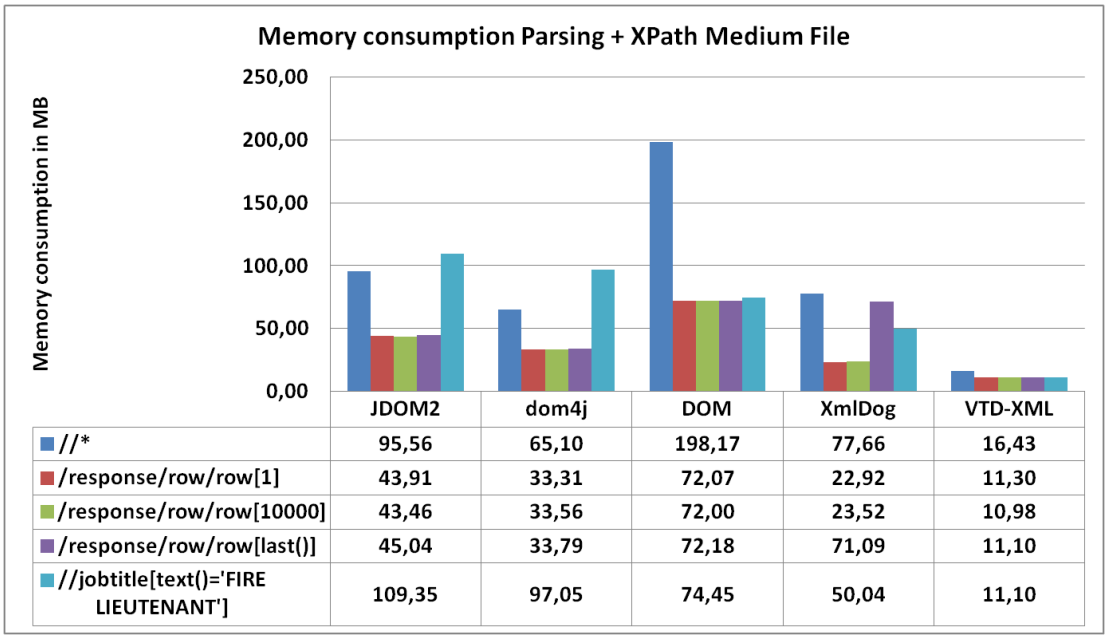
müssen entsprechende Strategien entwickelt werden, die für dieses Problem als Lösung dienen können.

A. Anhang Bilder und Tabellen



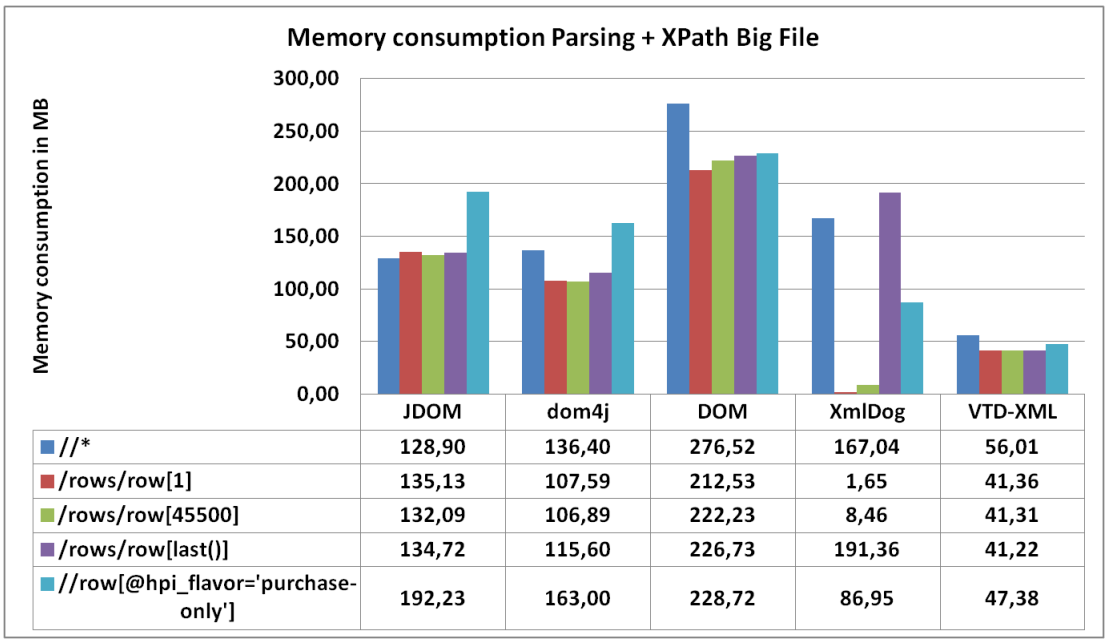
	JDOM2	dom4j	DOM	XmlDog	VTD-XML
//*	5,24	4,66	5,88	5,85	1,95
/response/row/row[1]	1,84	1,23	2,72	1,15	0,58
/response/row/row[400]	1,72	1,20	2,91	0,89	0,55
/response/row/row[last()]	1,59	1,14	3,19	3,32	0,51
//downtown[text()='true']	3,91	3,35	2,86	2,20	0,97

Abbildung A.1.: Small XML-Dokument: Speicherverbrauch Parsing + XPath



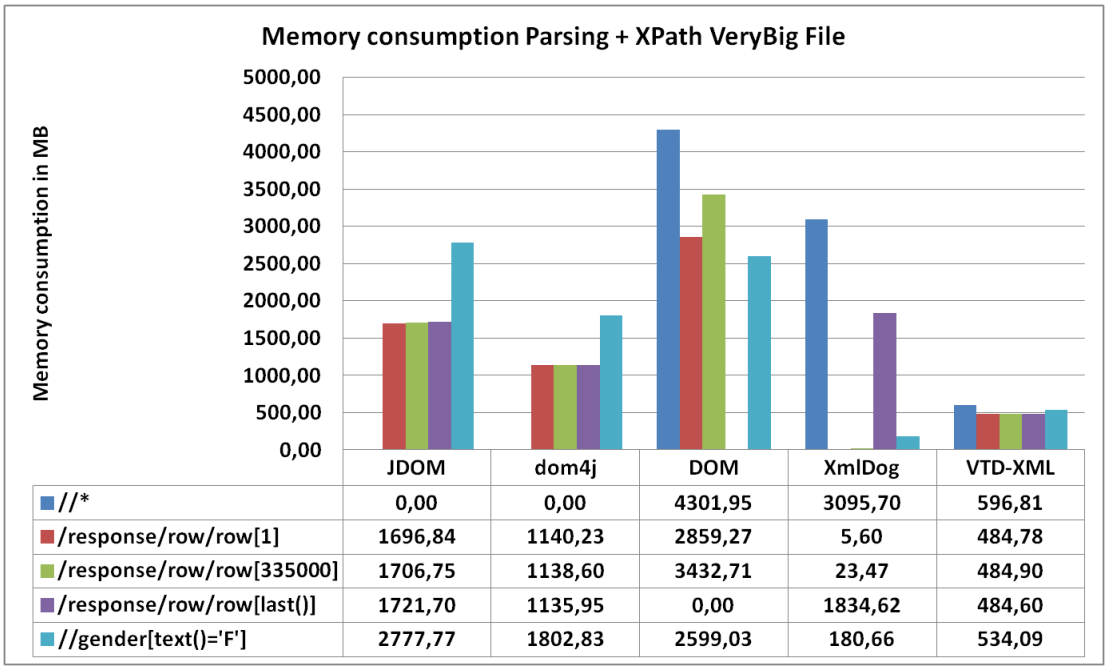
	JDOM2	dom4j	DOM	XmlDog	VTD-XML
//*	95,56	65,10	198,17	77,66	16,43
/response/row/row[1]	43,91	33,31	72,07	22,92	11,30
/response/row/row[10000]	43,46	33,56	72,00	23,52	10,98
/response/row/row[last()]	45,04	33,79	72,18	71,09	11,10
//jobtitle[text()='FIRE LIEUTENANT']	109,35	97,05	74,45	50,04	11,10

Abbildung A.2.: Medium XML-Dokument: Speicherverbrauch Parsing + XPath



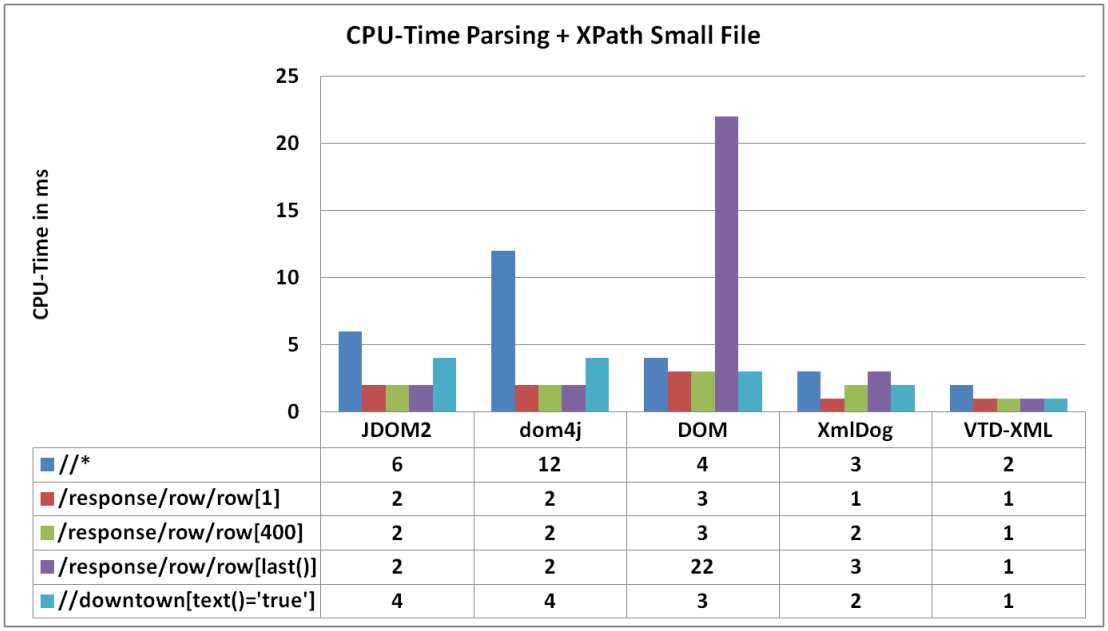
	JDOM	dom4j	DOM	XmlDog	VTD-XML
//*	128,90	136,40	276,52	167,04	56,01
/rows/row[1]	135,13	107,59	212,53	1,65	41,36
/rows/row[45500]	132,09	106,89	222,23	8,46	41,31
/rows/row[last()]	134,72	115,60	226,73	191,36	41,22
//row[@hpi_flavor='purchase-only']	192,23	163,00	228,72	86,95	47,38

Abbildung A.3.: Big XML-Dokument: Speicherverbrauch Parsing + XPath



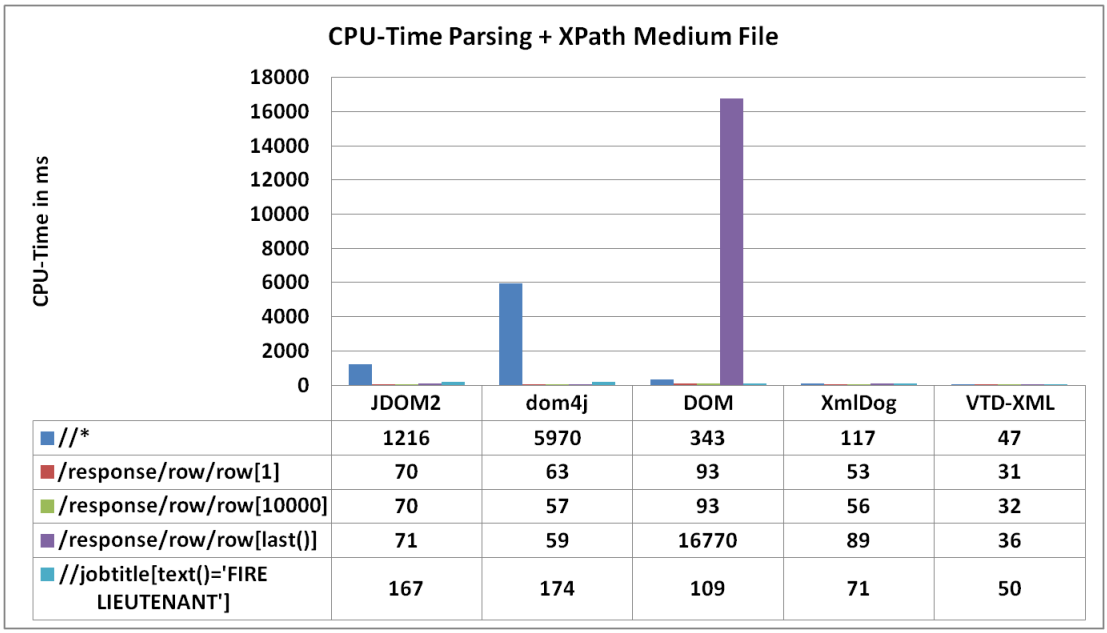
	JDOM	dom4j	DOM	XmlDog	VTD-XML
//*	0,00	0,00	4301,95	3095,70	596,81
/response/row/row[1]	1696,84	1140,23	2859,27	5,60	484,78
/response/row/row[335000]	1706,75	1138,60	3432,71	23,47	484,90
/response/row/row[last()]	1721,70	1135,95	0,00	1834,62	484,60
//gender[text()='F']	2777,77	1802,83	2599,03	180,66	534,09

Abbildung A.4.: VeryBig XML-Dokument: Speicherverbrauch Parsing + XPath, Hinweis: bei Wert gleich 0 wurde der Test nach 30+ Minuten abgebrochen



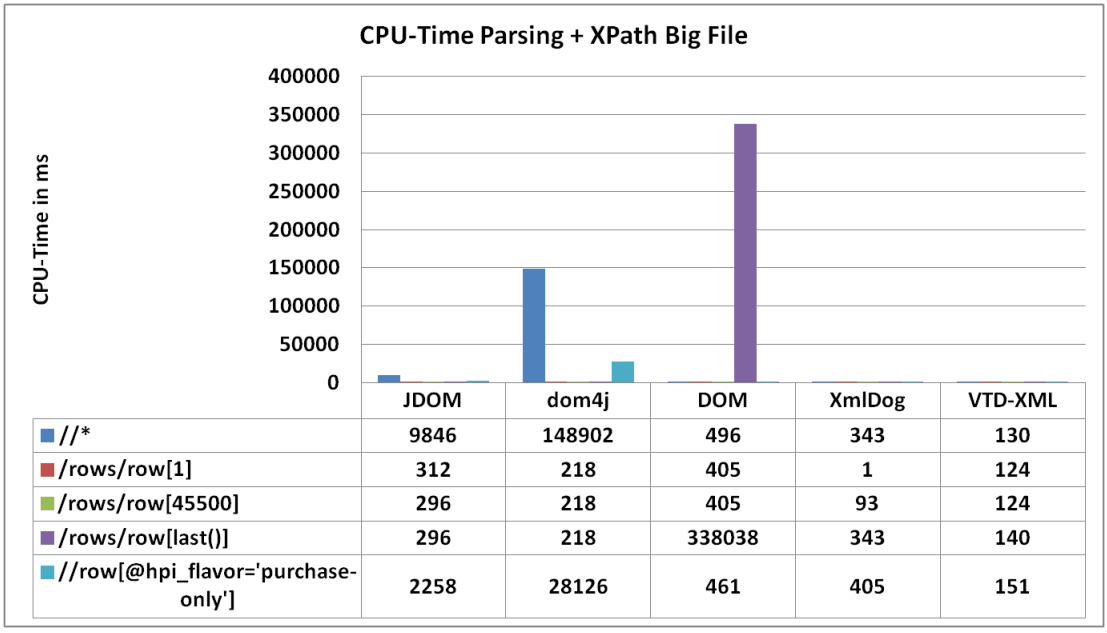
	JDOM2	dom4j	DOM	XmlDog	VTD-XML
//*	6	12	4	3	2
/response/row/row[1]	2	2	3	1	1
/response/row/row[400]	2	2	3	2	1
/response/row/row[last()]	2	2	22	3	1
//downtown[text()='true']	4	4	3	2	1

Abbildung A.5.: Small XML-Dokument: CPU-Zeit Parsing + XPath



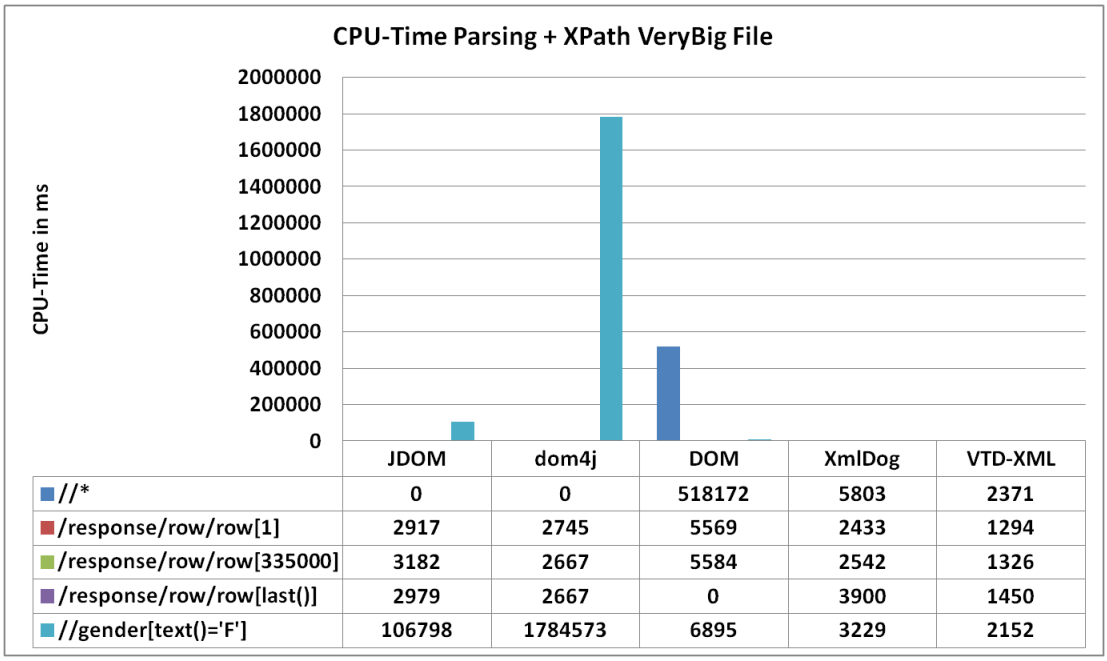
	JDOM2	dom4j	DOM	XmlDog	VTD-XML
//*	1216	5970	343	117	47
/response/row/row[1]	70	63	93	53	31
/response/row/row[10000]	70	57	93	56	32
/response/row/row[last()]	71	59	16770	89	36
//jobtitle[text()='FIRE LIEUTENANT']	167	174	109	71	50

Abbildung A.6.: Medium XML-Dokument: CPU-Zeit Parsing + XPath



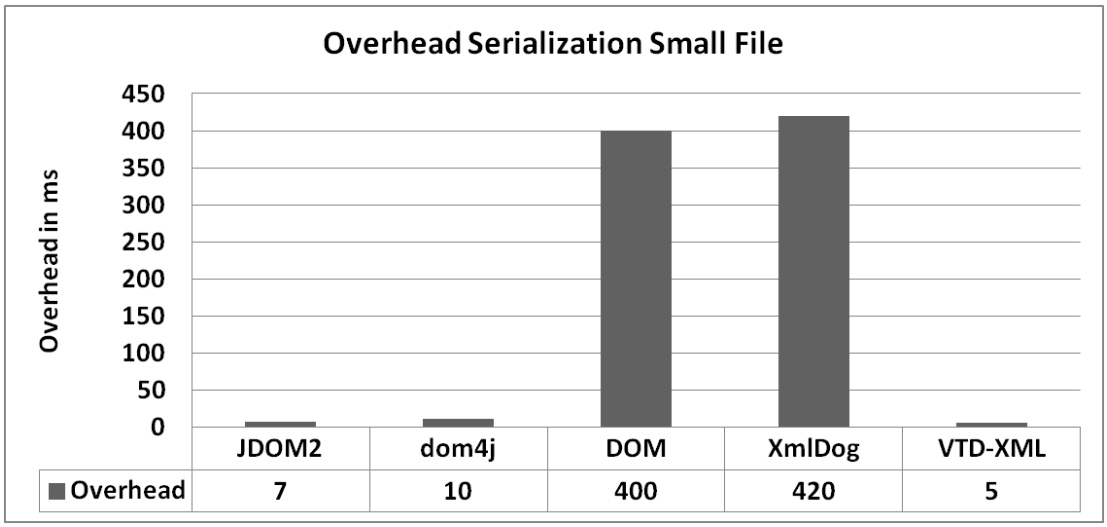
	JDOM	dom4j	DOM	XmlDog	VTD-XML
//*	9846	148902	496	343	130
/rows/row[1]	312	218	405	1	124
/rows/row[45500]	296	218	405	93	124
/rows/row[last()]	296	218	338038	343	140
//row[@hpi_flavor='purchase-only']	2258	28126	461	405	151

Abbildung A.7.: Big XML-Dokument: CPU-Zeit Parsing + XPath



	JDOM	dom4j	DOM	XmlDog	VTD-XML
//*	0	0	518172	5803	2371
/response/row/row[1]	2917	2745	5569	2433	1294
/response/row/row[335000]	3182	2667	5584	2542	1326
/response/row/row[last()]	2979	2667	0	3900	1450
//gender[text()='F']	106798	1784573	6895	3229	2152

Abbildung A.8.: VeryBig XML-Dokument: CPU-Zeit Parsing + XPath, Hinweis: bei Wert gleich 0 wurde der Test nach 30+ Minuten abgebrochen



	JDOM2	dom4j	DOM	XmlDog	VTD-XML
Overhead	7	10	400	420	5

Abbildung A.9.: Small XML-Dokument: Overhead Serialisierung

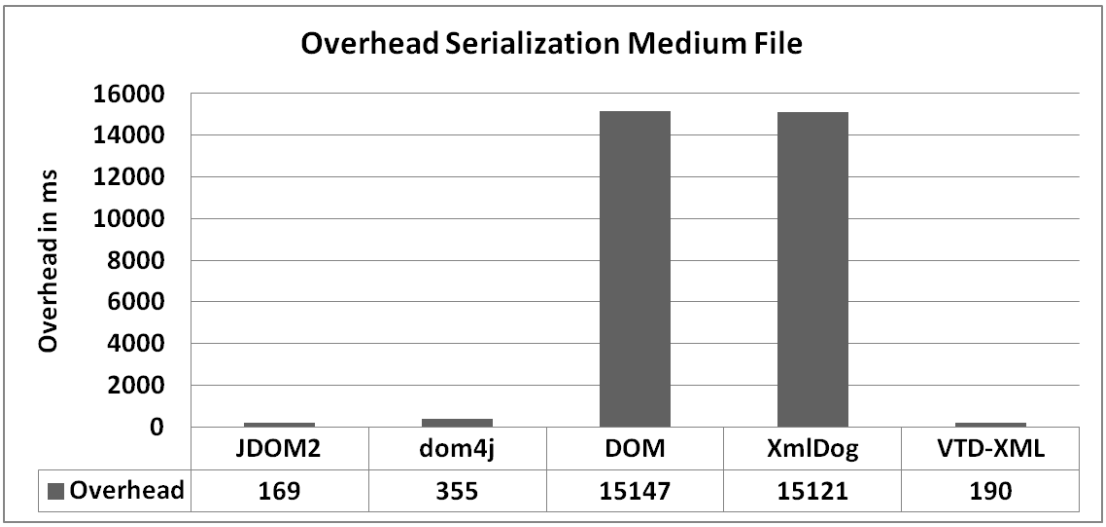


Abbildung A.10.: Medium XML-Dokument: Overhead Serialisierung

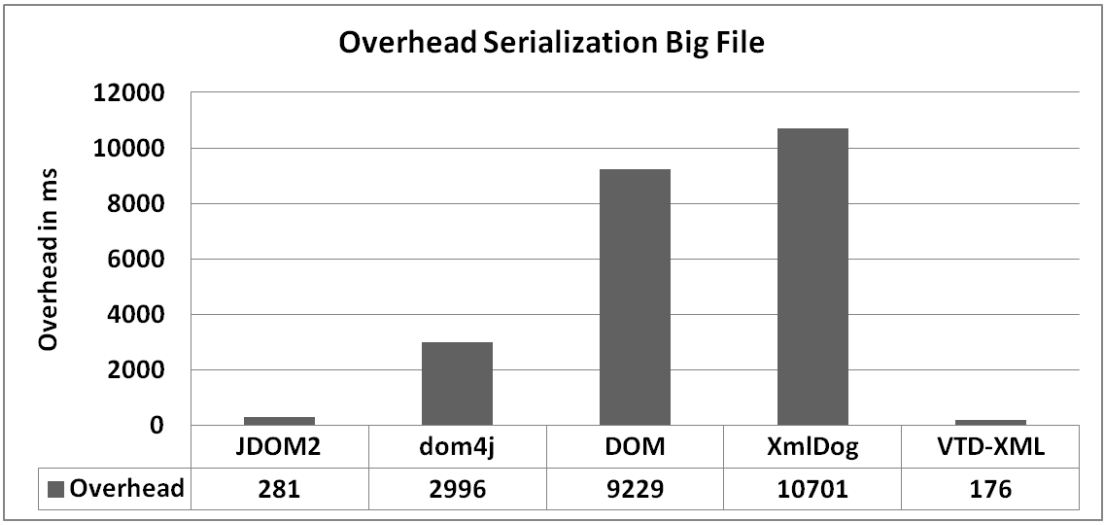


Abbildung A.11.: Big XML-Dokument: Overhead Serialisierung

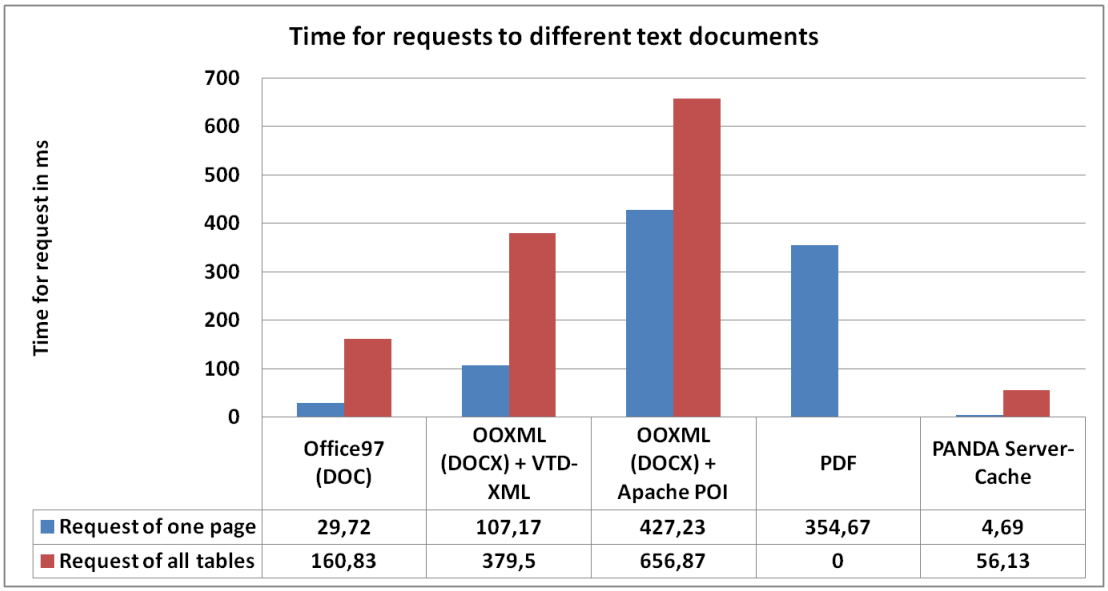


Abbildung A.12.: Antwortzeit von Anfragen auf verschiedene Datenformate der Textverarbeitung

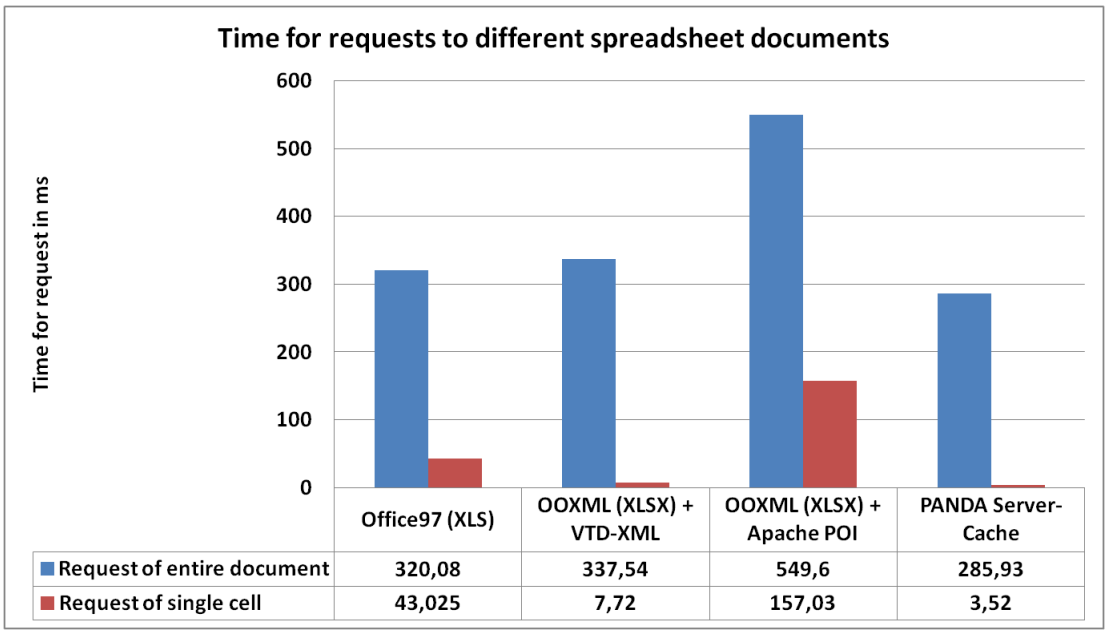


Abbildung A.13.: Antwortzeit von Anfragen auf verschiedene Datenformate der Tabellenkalkulation

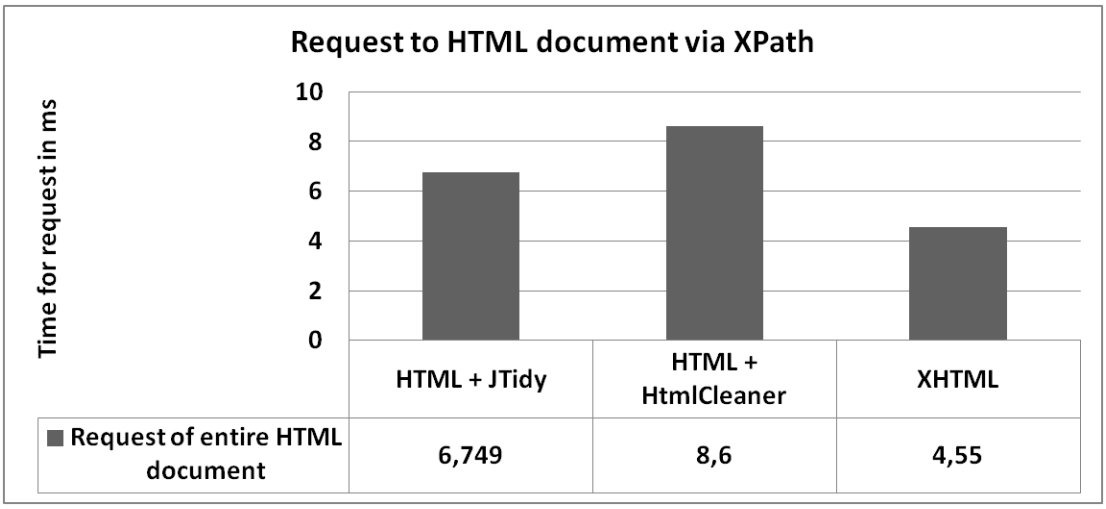


Abbildung A.14.: Antwortzeit von für Anfragen auf HTML/XHTML via XPath

	HTML	XML	PDF	CSV	XLS	GML	JSON	RDF	KML	TXT	ODS	XLSX	DOC	RSS
data.gov.uk	969	241	0	2983	1880	0	0	170	0	0	123	0	76	0
catalog.data.gov	24202	18540	1910	5994	3753	2373	5477	3004	3061	1760	0	153	18	235
data.gv.at	119	0	115	762	0	108	107	0	233	0	0	0	0	0
data.ottawa.ca	0	19	0	26	22	0	0	0	33	0	0	16	0	0
data.gc.ca	59094	59032	49403	6412	116	7520	16	1	16	312	0	0	7	1
govdata.de	3989	179	948	2450	2289	142	191	3	117	40	0	0	0	22
publicdata.eu	3731	3182	1486	12270	7538	0	2330	1790	0	0	0	0	0	0
opendata.admin.ch	11	5	13	15	2	0	0	0	0	12	1678	4	0	0
data.openpolice.ru	0	65	0	9	66	0	0	0	0	0	0	4	0	0
catalogodatos.gub.uy	7	3	4	29	6	0	0	0	0	19	0	0	0	0
dados.gov.br	151	54	14	46	25	0	6	0	0	0	0	0	0	0
datos.codeandomexico.org	0	0	1	58	46	0	2	0	0	0	0	22	0	0
data.qld.gov.au	5	14	34	396	143	0	3	0	10	51	0	49	6	0
africaopendata.org	1	1	10	756	162	0	0	0	3	0	0	21	2	0
data.sa.gov.au	30	4	2	51	66	0	2	0	8	5	0	3	156	4
Gesammtsumme	92309	81339	53940	32257	16114	10143	8134	4968	3481	2199	1801	272	265	262

Tabelle A.1.: Vorkommen spezifischer Datenformate auf ausgewählten Plattformen für Open Data

B. Anhang UML-Diagramme

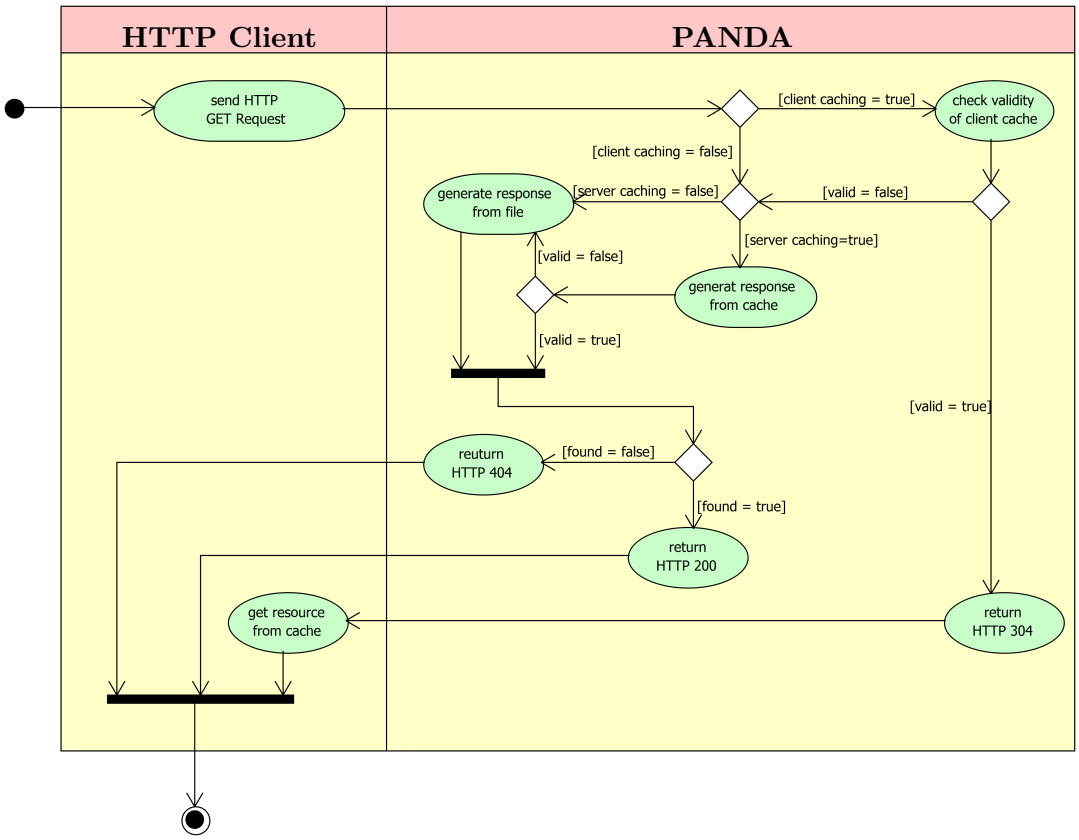


Abbildung B.1.: Aktivitätsdiagramm des erweiterten Anwendungsfalls „HTTP request with cache validation“

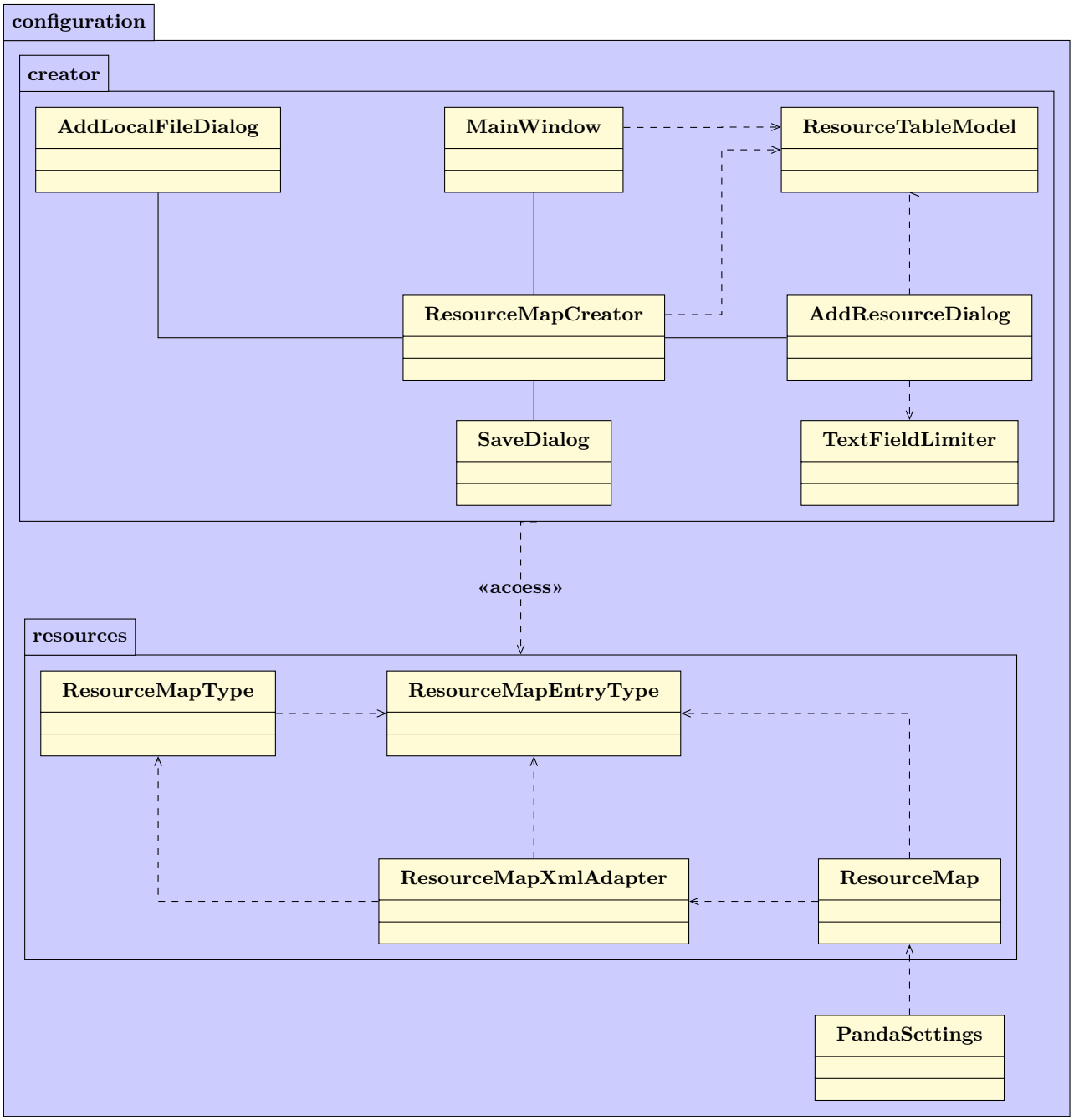


Abbildung B.2.: Paket *de.fuberlin.panda.data.configuration* mit enthaltenen Klassen für die Erstellung der Konfiguration von PANDA

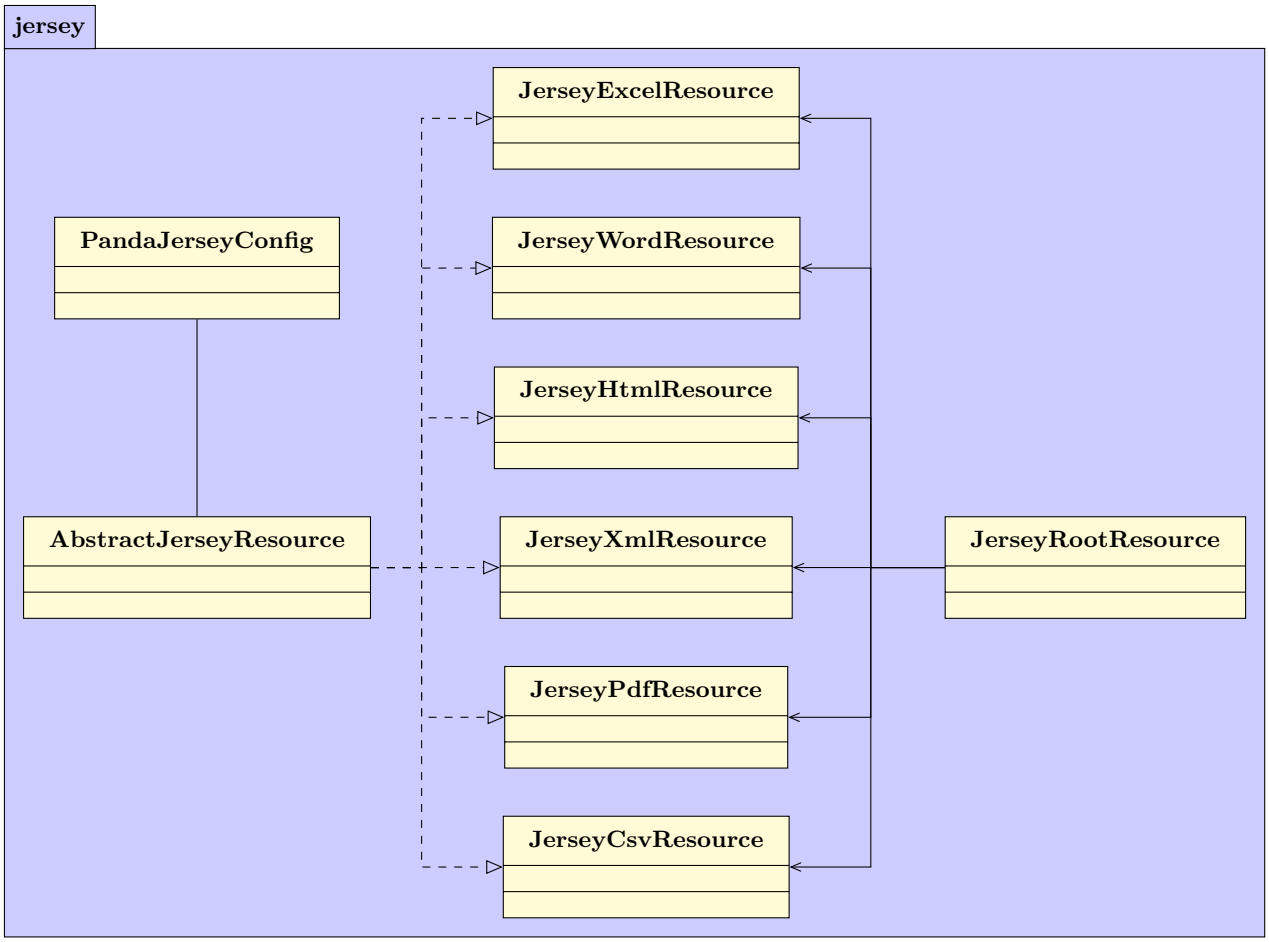


Abbildung B.3.: Paket *de.fuberlin.panda.api.jersey* mit enthaltenen Klassen für die Verarbeitung von HTTP-Anfragen

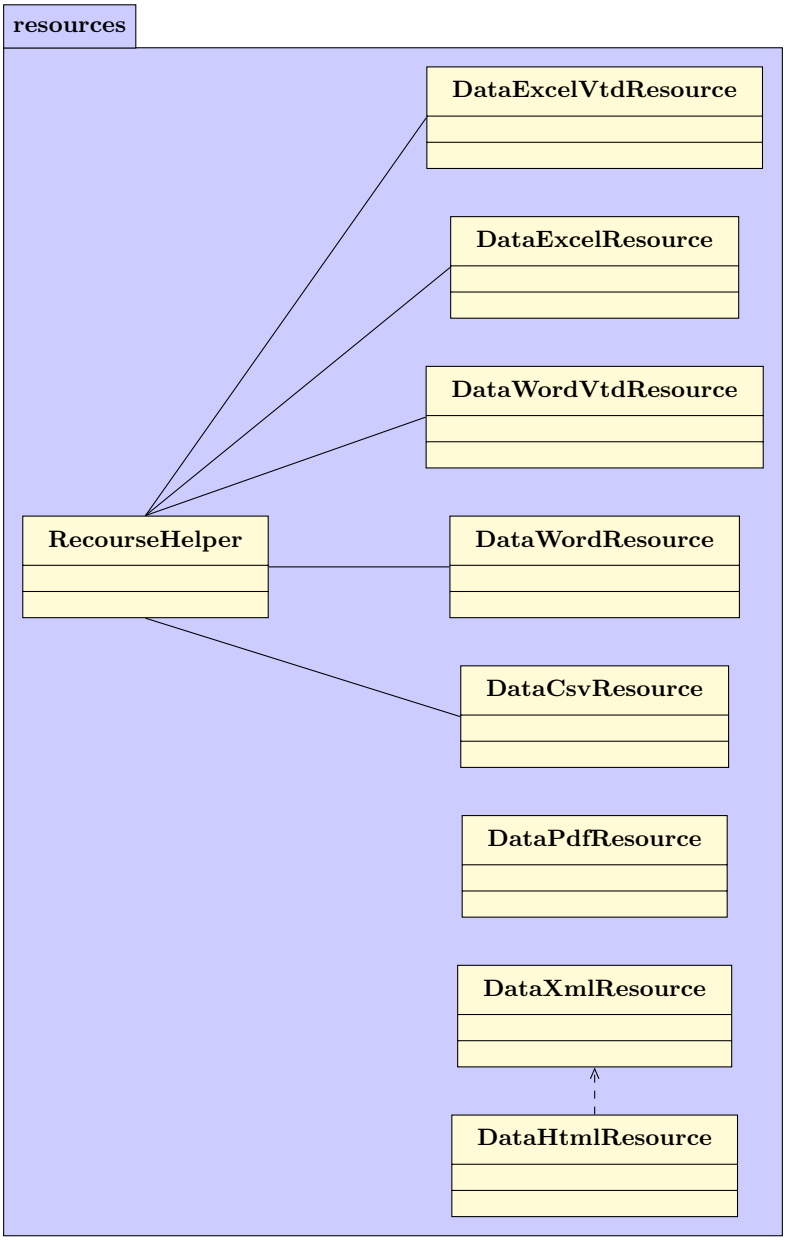


Abbildung B.4.: Paket *de.fuberlin.panda.data.resources* mit enthaltenen Klassen für die Verarbeitung der Datenformate

Literaturverzeichnis

[1] *BGH, 17.07.2003 - I ZR 259/00.*
http://dejure.org/dienste/vernetzung/rechtsprechung?Text=I%20ZR%20259/00,

[2] *Common Format and MIME Type for Comma-Separated Values (CSV) Files.*
http://tools.ietf.org/html/rfc4180,

[3] *Common HTTP Implementation Problems.*
http://www.w3.org/TR/2003/NOTE-chips-20030128/,

[4] *Cool URIs don't change.*
http://www.w3.org/Provider/Style/URI.html,

[5] *Document Object Model (DOM).*
http://www.w3.org/DOM/,

[6] *Extensible Markup Language (XML).*
http://www.w3.org/XML/,

[7] *„Deep Linking" in the World Wide Web.*
http://www.w3.org/2001/tag/doc/deeplinking.html,

[8] *Hypertext Transfer Protocol – HTTP/1.1.*
https://www.ietf.org/rfc/rfc2616.txt,

[9] *JSR 173: Streaming API for XML.*
https://jcp.org/en/jsr/detail?id=173,

[10] *Office File Formats Technical Documents.*
http://msdn.microsoft.com/en-us/library/cc313105%28v=office.12%29.aspx,

[11] *R2RML: RDB to RDF Mapping Language.*
http://www.w3.org/TR/r2rml/,

[12] *Resource Description Framework (RDF): Model and Syntax.*
http://www.w3.org/TR/WD-rdf-syntax-971002/,

[13] *SAX Webseite.*
http://www.saxproject.org/,

[14] *Uniform Resource Identifier (URI): Generic Syntax.*
http://tools.ietf.org/html/rfc3986,

[15] *Unterschiede zwischen XHTML und HTML.* *http://de.selfhtml.org/html/xhtml/unterschiede.htm*,

[16] *XML Path Language (XPath) 2.0 (Second Edition).* *http://www.w3.org/TR/xpath20/*,

[17] *XML Schema.* *http://www.w3.org/XML/Schema*,

[18] *Inversion of Control Containers and the Dependency Injection pattern.* *http://martinfowler.com/articles/injection.html*, January 2004

[19] *OMG Unified Modeling Language(OMG UML), Infrastructure.* : *OMG Unified Modeling Language(OMG UML), Infrastructure.* 2.4.1, August 2011

[20] Adobe Systems Incorporated (Hrsg.): *PDF Reference.* sixth edition. Adobe Systems Incorporated, April 2006

[21] Alnaqeib, Rami ; Alshammari, Fahad H. ; Zaidan, M. A. ; Zaidan, A. A. ; Zaidan, B. B. ; Hazza, Zubaidah M.: An Overview: Extensible Markup Language Technology. In: *CoRR* abs/1006.4565 (2010)

[22] Barton, C. ; Charles, P. ; Goyal, Deepak ; Raghavachari, M. ; Fontoura, M. ; Josifovski, V.: Streaming XPath processing with forward and backward axes. In: *Data Engineering, 2003. Proceedings. 19th International Conference on*, 2003, S. 455–466

[23] Deshmukh, Ms V. ; Bamnote, GR ; Kale, Ms P.: A Comparative Study of XML Parsers across Application. In: *International Journal of Computing and Technology(IJCAT)* 1 (2014), April, Nr. 3. – ISSN 2348–6090

[24] European Computer Manufacturers Association (Hrsg.): *ECMA-376: Office Open XML File Formats.* 4th edition. European Computer Manufacturers Association, December 2012

[25] Fielding, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*, University of California, PhD thesis, 2000

[26] Hinnerichs, Andreas: *TMPL Effiziente Identifikation von Mustern in XML Datenströmen*, Technische Universität Berlin, Institut für Telekommunikationssysteme, Diplomarbeit, 2006

[27] Lam, Tak C. ; Ding, Jianxun J. ; Liu, Jyh-Charn: XML Document Parsing: Operational and Performance Characteristics. In: *Computer* 41 (2008), Nr. 9, S. 30–37. – ISSN 0018–9162

[28] Lebo, Timothy ; Williams, Gregory T.: Converting Governmental Datasets into Linked Data. In: *Proceedings of the 6th International Conference on Semantic*

Systems. New York, NY, USA : ACM, 2010 (I-SEMANTICS '10). – ISBN 978–1–4503–0014–8, 38:1–38:3

[29] Oliveira1, Bruno ; Santos1, Vasco ; Belo2, Orlando: Processing XML with Java - A Performance Benchmark. In: *International Journal of New Computer Architectures and their Applications (IJNCAA)* 3 (2013), Nr. 1, S. 72–85. – ISSN 2220–9085

[30] Olteanu, D. ; Kiesling, T. ; Bry, F.: An evaluation of regular path expressions with qualifiers against XML streams. In: *Data Engineering, 2003. Proceedings. 19th International Conference on*, 2003, S. 702–704

[31] Spanos, Dimitrios-Emmanuel ; Stavrou, Periklis ; Mitrou, Nikolas: Bringing Relational Databases into the Semantic Web: A Survey. In: *Semant. web* 3 (2012), April, Nr. 2, 169–209. *http://dx.doi.org/10.3233/SW-2011-0055*. – DOI 10.3233/SW–2011–0055. – ISSN 1570–0844

[32] Tej, M. V. U. ; Cheelu, Dhanaraj ; Babu, M. R. ; Krishna, P. V.: Analyzing XML Parsers Performance for Android Platform. In: *International Journal of Computer Science and Information Technologies(IJCSIT)* 2 (2011), Nr. 3, S. 973–976. – ISSN 0975–9646

[33] Zhang, Jimmy: *Non-Extractive Parsing for XML.* *http://www.xml.com/pub/a/2004/05/19/parsing.html*, May 2004

[34] Zhang, Jimmy: *VTD-XML: XML Processing for the Future.* *http://www.codeproject.com/Articles/23516/VTD-XML-XML-Processing-for-the-Future-Part-I*, April 2008