Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Fachbereich Informatik
Arbeitsbereich Datenbanksysteme

Bachelorarbeit Medieninformatik

Implementation of an XML to Relational Data Format Conversion Tool

Alexander Schiller

18. September 2014

Gutachter
Prof. Dr. rer. nat. Torsten Grust
Fachbereich Informatik
Arbeitsbereich Datenbanksysteme
Universität Tübingen

Betreuer
Tobias Müller

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderenWerken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum: _____ Unterschrift: _____

Abstract

Viele Nicht-Informatiker möchten mit Daten arbeiten, die in XML Dumps enthalten sind. Um diese Daten in eine Form zu bringen, in der diese einfacher zu verwerten sind, haben wir ein Konvertierungstool geschrieben, das XML Dokumente in CSV Dateien umwandelt. Dabei ist das Programm weitgehend unabhängig vom verfügbaren Arbeitsspeicher, was es möglich macht auch sehr große XML Dumps zu konvertieren, und von den Programmierkenntnisse des Anwenders.

Inhaltsverzeichnis

1. Einleitung

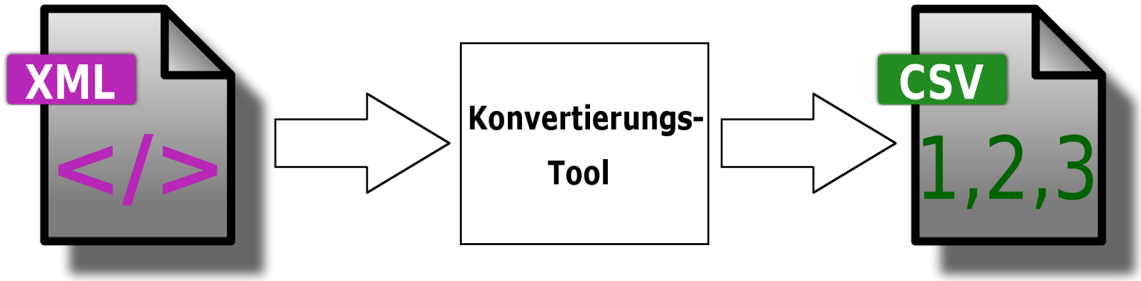

Abbildung 1.1.: Konvertierung

Bei dieser Arbeit haben wir ein Konvertierungstool, in Java, implementiert, welches mittels Query Tree Patterns, Daten von XML Dokumenten extrahieren und in eine CSV Datei schreiben kann. Die XML Dokumente werden mit Hilfe von SAX eingelesen und die Daten mittels YFilter extrahiert.
Durch die Verwendung von SAX besteht die Möglichkeit auch große XML Dokumente, die nicht komplett in den Arbeitsspeicher geladen werden können, zu konvertieren.
Mit den Eingangs implementierten Query Tree Patterns, kann der Benutzer Daten aus öffentlich zugänglichen XML Dumps von Wikipedia und StackOverflow extrahieren, ohne dass er dafür im Programmcode, die Queries anpassen muss. Somit ist das Programm auch für Laien geeignet.

1.1. Überblick

In dieser Arbeit werden zunächst die Grundsätzlichen Techniken aufgezeigt, die das Konvertierungstool ermöglicht haben. In Kapitel zwei erläutern wir SAX, was das einlesen von großen XML Dokumenten ermöglicht. YFilter, wird in Kapitel drei, als Technik um die zu extrahierenden Daten zu lokalisieren, erläutert. Dabei wurde in dem Kapitel näher darauf eingegangen wie sich die benutzten Queries zusammensetzen und wie sich der NFA darauf aufbauen lässt.
Im Kapitel vier, Implementierung wurde näher darauf eingegangen wie sich diese Techniken

in diesem Programm wiederfinden. Des weiteren wird hier auch der Aufbau des Outputs besprochen.
Messungen deuten, in Kapitel fünf, schließlich noch an, wie der Speicherverbrauch während der Konvertierung aussieht.

2. SAX

Die Simple API for XML spezifiziert eine Menge von Methoden für den Zugriff auf XML-Dokumente mittels eines SAX-Parsers. Anders als DOM modelliert SAX XML Dokumente in einem Stream von Event Callbacks, weshalb es als eine "aktive" API bezeichnet wird[1]. Es werden Events für spezielle Inhalte, wie z.B. in Tabelle 2.1 gezeigt, an den Event Callback Handler geschickt. Die Events kommen in der Reihenfolge an, in der sie in dem Dokument auftauchen, wie in Abbildung 2.1 gezeigt. Für XML Attribute gibt es keine eigenen Events. Die Attribute werden beim Start Element Event mitgeschickt.
Da es für XML Dokumente keine Größenbeschränkung gibt und sie somit sehr groß sein können, ist es nicht immer möglich diese komplett in den Arbeitsspeicher zu laden. Durch die Modellierung der Dokumente in einem Stream, ist es möglich dieses Problem zu umgehen.
Eine vorzeitige Beendigung des Streaming Vorgangs, lässt sich nur mittels Exceptions bewerkstelligen.

Tabelle 2.1.: Auswahl von SAX Events

Event	**Bedeutung**
startDocument()	Anfang der XML Datei
endDocument()	Ende der XML Datei
startElement(..., String localName, Atributes atts, ...)	Start des Elements **localName** mit den Attributen **atts**
endElement(..., String localName, ...)	Ende des Elements **localName**
characters(char[] ch, ...)	Zeichen die zwischen den tags stehen

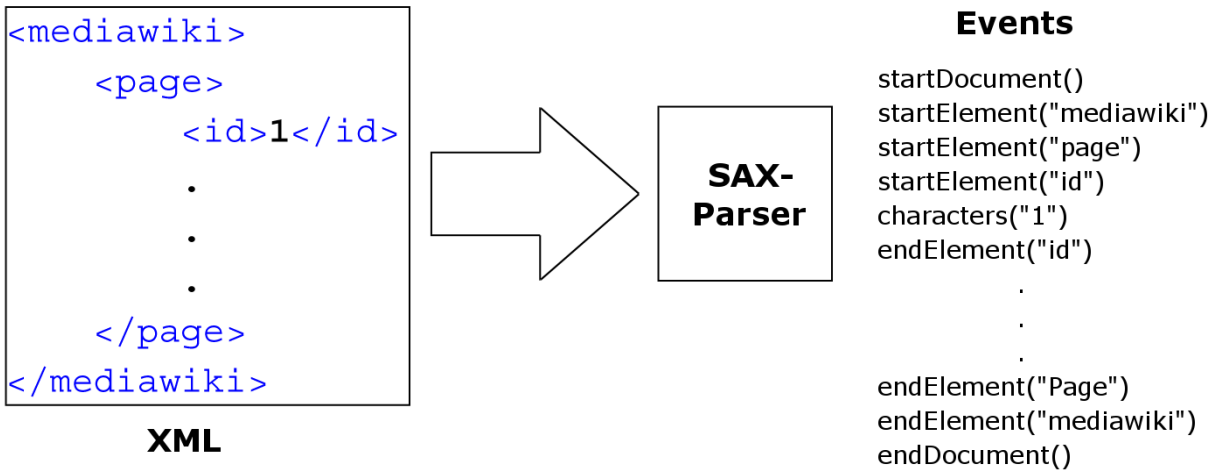

Abbildung 2.1.: SAX Eventstream

3. YFilter

3.1. Übersicht

YFilter[2] ist eine Technik für on-the-fly Filterung von XML Datenströmen. Es werden Queries verwendet, die auf XPath basieren, um die zu filternden Daten zu lokalisieren.
Die Queries werden in genau einem Nondeterministic Finite Automaton (NFA) umgewandelt, der während die Daten gelesen werden, durchlaufen wird.

3.2. Aufbau

Listing 3.1: Wikipedia pages-logging XML Ausschnitt, entnommen aus: 'dzwiki-20140416-pages-logging' [3]

```
1  <mediawiki xmlns="http://www.mediawiki.org/xml/export-0.8/" xmlns:xsi="http
       ://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.
       mediawiki.org/xml/export-0.8/ http://www.mediawiki.org/xml/export-0.8.
       xsd" version="0.8" xml:lang="dz">
2    ...
3    <logitem>
4      <id>10</id>
5      <timestamp>2005-09-15T17:26:27Z</timestamp>
6      <contributor>
7            <username>Jvano</username>
8            <id>23</id>
9      </contributor>
10     <comment>Created the user [[User:Jvano|Jvano]] ([[User talk:Jvano|Talk
           ]] | [[Special:Contributions/Jvano|contribs]])</comment>
11     <type>newusers</type>
12     <action>newusers</action>
13     <logtitle>Special:Userlogin</logtitle>
14     <params xml:space="preserve"/>
15   </logitem>
16   ...
17 </mediawiki>
```

Listing 3.2: Queries für Wikipedia pages-logging

```
Q01: /mediawiki/logitem/id/text()$log_id
Q02: /mediawiki/logitem/timestamp/text()$log_timestamp
Q03: /mediawiki/logitem/contributor/username/text()$log_user_text
Q04: /mediawiki/logitem/contributor/id/text()$log_user
Q05: /mediawiki/logitem/comment/text()$log_comment
Q06: /mediawiki/logitem/type/text()$log_type
Q07: /mediawiki/logitem/action/text()$log_action
Q08: /mediawiki/logitem/logtitle/text()$log_title
Q09: /mediawiki/logitem/params/@xml:space$log_params
Q10: /mediawiki/logitem/\n
```

Listing 3.3: Beispiel Output CSV Datei Ausschnitt vom Dump aus Listing 3.1

```
1 log_id,log_timestamp,log_user_text,log_user,log_comment,log_type,log_action,
     log_title,log_params
2 10,2005-09-15T17:26:27Z,Jvano,23,Created the user [[User:Jvano|Jvano]] ([[
     User talk:Jvano|Talk]] | [[Special:Contributions/Jvano|contribs]]),
     newusers,newusers,Special:Userlogin,preserve
3 ...
```

3.2.1. Queries

Queries, die angeben wo sich die zu filternden Daten in dem Dokument finden lassen, sind in einem XPath basierten Format aufgebaut. Somit setzen sich Queries aus einem oder mehreren sogenannten Lokalisierungsschritten zusammen. Listing 3.1 zeigt einen Ausschnitt eines Wikipedia pages-logging Dump, von "dzwiki-20140416-pages-logging"[3], für den es die Queries aus Listing 3.2 gibt. In Listing 3.3 ist dafür ein Ausschnitt einer möglichen Output CSV Datei zu sehen
Die Knoten geben XML tags an, das heißt also der Knoten "logitem" steht für den XML tag <logitem>. Auch der Wildcard Operator * wird unterstützt und steht für ein beliebiges tag. Des Weiteren gibt es Achsen (Slash "/" und Doppelslash"//") die eine hierarchische Beziehung zwischen Knoten darstellen. Ein Slash gibt dabei eine Eltern-Kind Beziehung, ein Doppelslash eine Vorfahre-Nachfahre Beziehung an.
Eine Eltern-Kind Beziehung bedeutet, dass sich das Kind tag direkt innerhalb des Eltern tag befindet, also z.B. "logitem/id" dass sich direkt innerhalb des tags <logitem> das tag <id> befindet:

```
<logitem>
   <id></id>
   ...
</logitem>
```

Bei einer Vorfahre-Nachfahre Beziehung, kann sich das Nachfahre tag irgendwo innerhalb des Vorfahre tags befinden. Dieses kann also auch beliebig tief verschachtelt sein. So z.B. "mediawiki//username":

```
<mediawiki>
   <logitem>
      <contributor>
         <username></username>
      </contributor>
   </logitem>
</mediawiki>
```

Möchte man nun den Text zwischen den tags herausfiltern, muss man nach dem entsprechenden tag "/text()" angeben. So bedeutet ".../type/text()", dass der Text zwischen den tags <type> und </type> herausgefiltert wird.
Befinden sich nun die zu filternden Daten nicht innerhalb der tags als Text, sondern sind als Attribute der tags gegeben, muss man diese, nach entsprechendem tag, mit @ markieren. Zum Beispiel sieht man im XML Ausschnitt, aus Listing 3.4, in Zeile 3, das tag "row" mit den Attributen "Id", "UserID", "Name" und "Date". Um die Informationen eines davon, z.B. Id, zu filtern muss man dies folgendermaßen Angeben: ".../row/@Id". Wenn man die Informationen mehrerer Attribute filtern möchte, kann man dies entweder mit mehreren Queries bewerkstelligen oder man kann die benötigten Attribute direkt in einer Query mit weiteren @ anhängen: ".../row/@Id@UserId@Name@Date"

Listing 3.4: Stackoverflow Badges XML Ausschnitt, entnommen aus: 'academia.stackexchange.com' [4]

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <badges>
3    <row Id="1" UserId="2" Name="Autobiographer" Date="2012-02-14T20
        :34:13.217"/>
4    ...
5 </badges>
```

3.2.2. Queries Sonderfälle

Listing 3.5: Wikipedia XML Ausschnitt, entnommen aus: 'dzwiki-20140416-pages-meta-history' [3]

```
...
  <page>
    <title>Template:Sitesupportpage</title>
    <id>19</id>
    ...
    <revision>
      <id>8</id>
      ...
    </revision>
    <revison>
      <id>3916</id>
      ...
    </revision>
    ...
  </page>
  ...
```

Verschachtelung

Es kann bei XML Dumps vorkommen, dass man innere und äußere Tabellen findet. So gibt es z.B. bei den Wikipedia Dumps Informationen zu den einzelnen Seiten (<page>) und innerhalb dieser gibt es noch Informationen zu jeder einzelnen Revision (<revision>), wie in Listing 3.5 zu sehen. Da wir bei einer Revision auch immer wissen wollen zu welcher Seite sie gehört, müssen wir die Daten der Seite immer auch dazu schreiben, auch wenn sie nur einmal außen auftauchen. Unsere Lösung für dieses Problem ist, dass man bei den Queries vor den gewünschten Daten das Sonderzeichen Raute setzt. So wird mit der Query ".../page/title/#text()"der Text, zwischen den <title></title> tags, bei jeder neuen Zeile dazugeschrieben.

Umbenennung

Wenn man sich den XML Ausschnitt aus Listing 3.5 ansieht, wird einem vielleicht auffallen, dass das tag <id> mehrmals auftaucht. Die CSV Dateien sind so aufgebaut, dass der Name der Spalte gleich dem Namen des tags ist, in dem sich der gewünschte Text befindet, bzw. gleich dem Namen des gewünschten Attributs ist. Allerdings dürfen zwei Spalten nicht gleich heißen, dies bedeutet, dass es nicht mehrere verschiedenen Spalten mit dem Namen id geben kann. Um dieses Problem zu umgehen gibt es die Möglichkeit, in den Queries, den entsprechenden Spaltennamen umzubenennen. Hierfür setzt man am Ende das Dollarzeichen

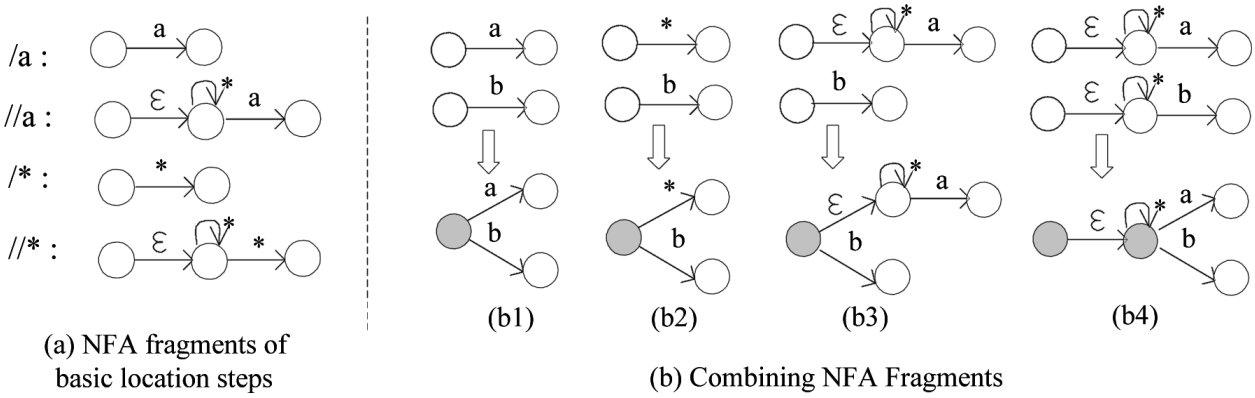

Abbildung 3.1.: Beispiele von Fragmenten und Kombinationen entnommen aus [2]

ein und gibt dann den neuen Namen an. So wird zum Beispiel mit der Query ".../page/id/text()$rev_page" die ID der Seite in der Spalte rev_page gespeichert, während die ID der Revision mit der Query ".../revision/id/text()$rev_id" in die Spalte rev_id gespeichert wird.

Terminierung von Tupeln

Als letzten Sonderfall, speziell für unsere Anwendung, muss eine Query angegeben werden, die bestimmt wann eine neue Zeile anfängt, also ein Erkennungszeichen wann ein Tupel vollständig gelesen wurde und somit auch das Nächste anfängt. Hierfür wird an dem tag, bei dem eine neue Zeile angefangen werden soll ein \n eingefügt. So soll z.B. bei jeder Revision eine neue Zeile angefangen werden: ".../revision/\n".

3.2.3. NFA

Es gibt vier (Grund-)Lokalisierungsschritte, die in den Queries benutzt werden können. Diese sind "/a", "//a", "/*" und "//*", wobei a für einen beliebigen Übergang steht und * der Wildcard Operator ist[2]. Abbildung 3.1(a) zeigt auf wie die einzelnen Schritte in NFA Fragmente übersetzt werden. Anzumerken ist das bei Schritten mit "//" ein ε-Übergang zu einem Zustand mit einer Schleife erzeugt wird. Dieser Übergang ist wichtig um bei der Kombinierung von Fragmenten mit "/" und "//" die verschiedene Semantik beider Schritte zu erhalten.
Um die einzelnen NFA Fragmente zu kombinieren geht man wie folgt vor: Als erstes gibt es einen Anfangszustand den alle Fragmente gemeinsam haben. Um ein neues Fragment dem kombinierten NFA (bzw. Anfangs einem anderen Fragment) hinzuzufügen, durchqueren wir, über die Übergänge, den NFA bis entweder der Endzustand des neuen Fragments erreicht wurde oder bis ein Zustand erreicht wurde, bei dem kein vorhandener Übergang mit dem des Fragments übereinstimmt. Im ersten Fall wird aus dem Zustand, im kombinierten NFA, ein Endzustand gemacht, falls dieser es davor noch nicht war. Im zweiten Fall wird neuer

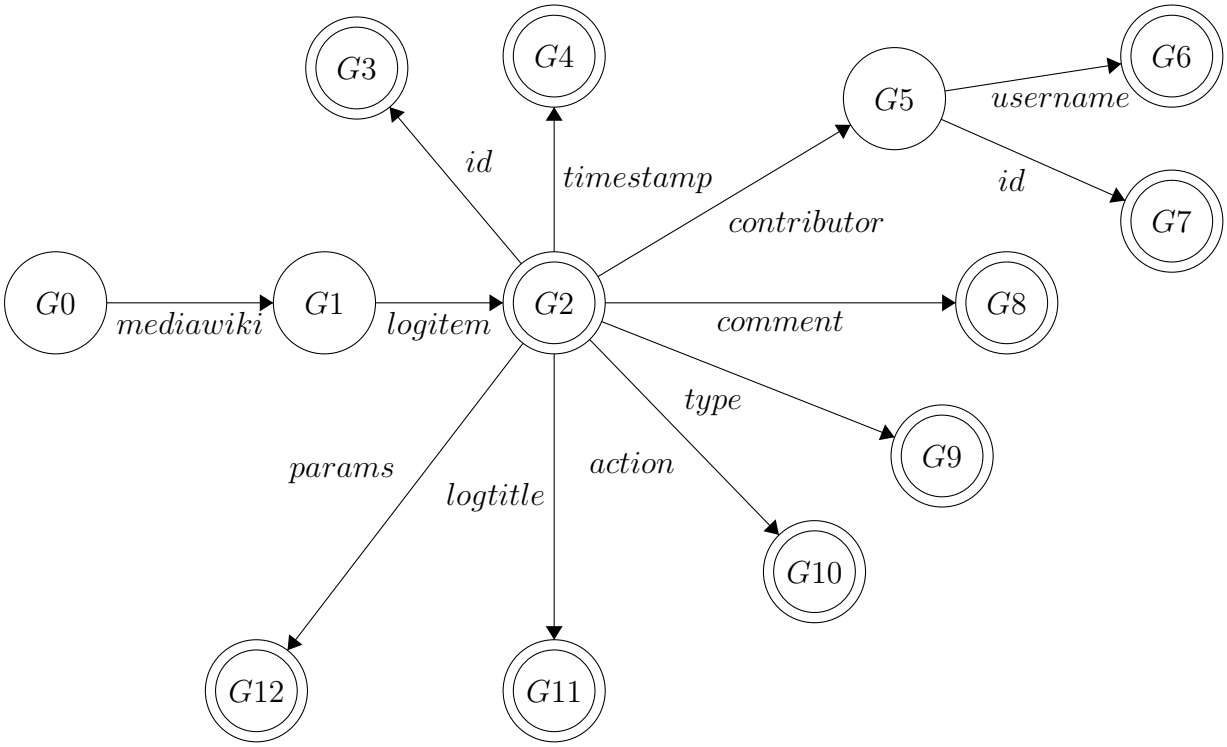

Abbildung 3.2.: Beispiel NFA resultierend von den Queries aus Listing 3.2

Zweig aus dem zuletzt erreichten Zustand des kombinierten NFA erstellt. Der Zweig besteht aus dem Übergang und dem Rest des Fragmentes. Die Abbildung 3.1(b) zeigt vier Beispiele solcher Kombinationen und die Abbildung 3.2 zeigt einen kombinierten NFA von den Queries aus Listing 3.2 an.
Die Konstruktion des NFA ist beim YFilter ein inkrementeller Prozess. Dies bedeutet dass neue Queries jederzeit einfach hinzugefügt werden können. Dadurch lässt sich der Anwendungsbereich dieses Tools leicht auf andere Dumps erweitern.

4. Implementierung

4.1. Übersicht

Die Implementierung besteht aus folgenden nennenswerten Teilen. Aus dem YFilter, den SAX Handlern und dem Output.

4.2. YFilter

4.2.1. Queries

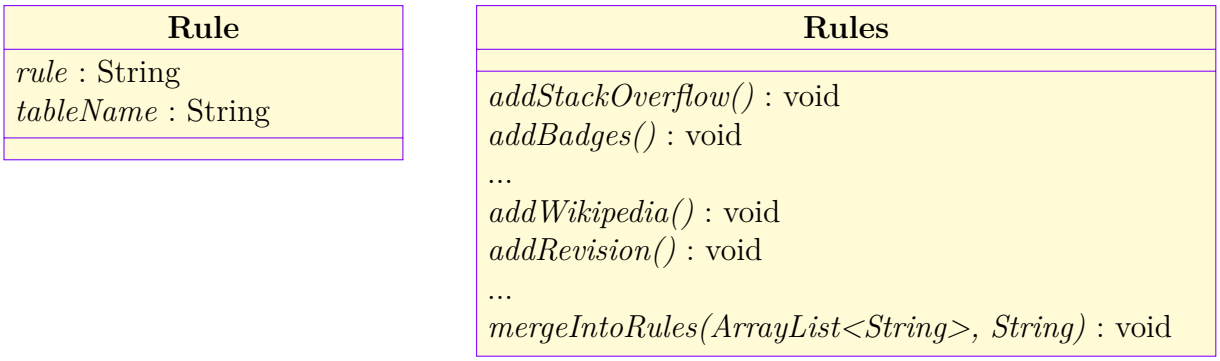

Abbildung 4.1.: Klassendiagramm für Queries

Die für den YFilter benötigten Queries werden in der Klasse **Rule** repräsentiert. Diese besitzt zwei Attribute, zum einen die Query selbst(*rule*) und zum anderen den Namen der Tabelle zu der diese Query gehört.
Gesammelt werden alle Queries, für die unterstützten Dump-Quellen, in der Klasse **Rules**, die eine Unterklasse von ArrayList<Rule> ist. Zur besseren Übersicht sind die Queries in Methoden angeordnet, die beim Konstruktor aufgerufen werden. Zuerst werden hierbei Methoden aufgerufen die angeben zu welcher Dump-Quelle die Queries gehören (StackOverflow, Wikipedia). In diesen Methoden befinden sich weitere Methoden die genauer angeben um welche Dumps es sich handelt (z.B. StackOverflow -> Badges) und in diesen stehen letztendlich die Queries.
Mit *mergeIntoRules(...)* werden die Queries schließlich als Regeln (**Rule**, mit Query und Tabellennamen) in die **Rules** Liste eingefügt.
In der Klasse **RulesParser** werden die Queries, aus **Rules**, mit der Methode *parse(...)*, siehe

Anhang A.1, in den NFA umgewandelt, wie in Kapitel 3.2.3 beschrieben. Gleichzeitig wird dabei auch die Struktur der Tabellen erstellt.

4.2.2. NFA

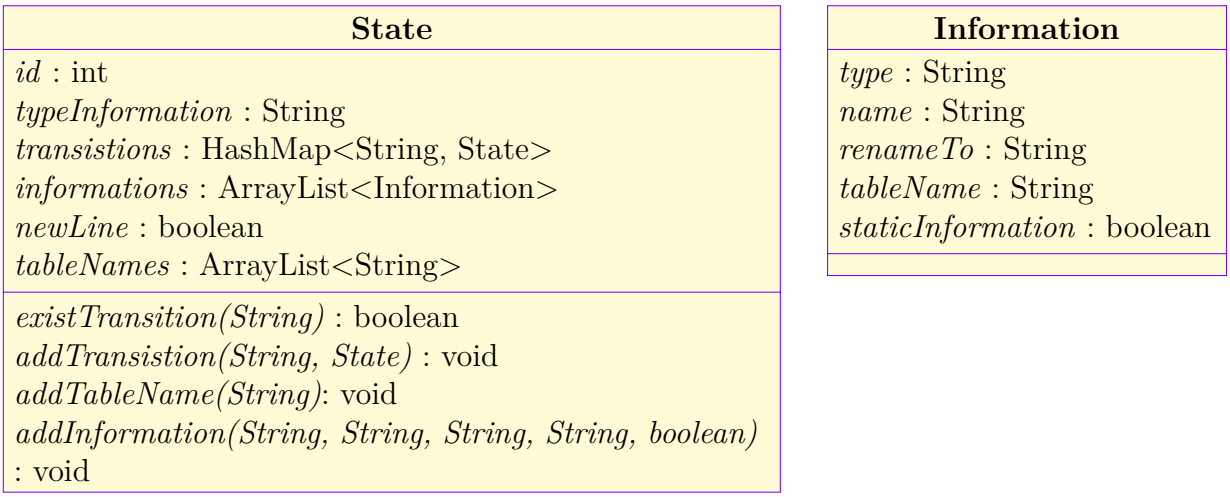

Abbildung 4.2.: Klassendiagramm für NFA

Der NFA selbst besteht aus Zuständen (**State**). Ein Zustand hat eine ID, Informationen um welchen Typ es sich handelt (z.B. Endzustand oder //-Kind) und eine HashMap die angibt mit welchem Übergang (Name der XML tags) welcher Zustand erreicht wird. Es gibt eine Methode mit der sich überprüfen lässt ob ein bestimmter Übergang bereits vorhanden ist und eine Methode mit der sich neue Übergänge anlegen lassen.
Da es möglich sein kann, dass ein Zustand zu mehreren Tabellen gehört, besitzen Zustände auch eine Liste mit Namen von Tabellen zu denen sie gehören. Des weiteren besitzt ein Zustand eine Angabe darüber, ob bei diesem Zustand eine neue Zeile beginnen muss und eine Liste aus Objekten der Klasse **Information**. Diese Listen lassen sich mit entsprechenden Methoden auch erweitern (*addTableName(...)* bzw. *addInformation(...)*).
Information stellt die Daten dar, die extrahiert werden sollen. Dabei wird mit *type* angegeben, ob es sich um Text zwischen XML tags oder um Attribute handelt. Weiterhin wird der Name des tags/Attributs und die potentielle Umbenennung angegeben (falls keine Umbenennung stattfinden soll, wird auch hier der Name des tags/Attributs genommen). Auch enthält ein Objekt dieser Klasse den Namen der Tabelle zu der diese **Information** gehört und ob es sich um statische Informationen handelt, also ob die Informationen in jeder Zeile neu vorkommen sollen.

4.3. SAX Handler

4.3.1. Übersicht

Für das Einlesen der XML Dokumente benötigen wir zwei verschiedene SAX Event Handler. Zum einen benötigen wir einen Handler (**TableChecker**), der überprüft um welchen Dump es sich wahrscheinlich handelt, also welche Tabelle wir daraus gewinnen können. Zum anderen benötigen wir einen Handler (**XMLContentHandler**), der die Daten extrahiert und an den Output schickt. Um Redundanz einzudämmen haben wir eine gemeinsame abstrakte Oberklasse eingeführt (**SAXHandler**).
Ein **SAXHandler** hat als Attribute, eine Liste aus allen möglichen momentanen Zuständen des NFA (zu Beginn nur der Anfangszustand) und einen Stack mit den vorangegangenen Listen, wie in Abbildung 4.3 zu sehen ist.
Beim Start eines Elements wird die Liste mit den momentanen Zuständen auf den Stack geschoben und eine neue Liste mit den neuen Zuständen erstellt. Für jeden Zustand aus der Liste mit momentanen Zuständen wird überprüft ob ein Übergang mit dem Namen des Elements, mit dem Wildcard Operator * oder mit Epsilon vorhanden ist. Die Zustände die mit einem oder mehreren dieser Übergänge erreicht werden, werden in die neue Liste eingetragen. Neben den Übergängen wird auch überprüft ob es sich bei dem Zustand um ein "//-Kind" handelt. Sollte dies der Fall sein wird der Zustand selbst zu der Liste mit neuen Zuständen eingefügt. Zum Schluss wird dann die alte Liste mit den momentanen Zuständen durch die neue Liste ersetzt.
Beim Ende eines Elementes wird die Liste mit den momentanen Zuständen mit der älteren Liste aus dem Stack ersetzt.

4.3.2. TableChecker

Für die Initialisierung und um alle Daten in die richtige Spalte einzutragen, da nicht unbedingt in jeder Zeile alle Informationen vorkommen, ist es vorteilhaft im Vornherein zu wissen zu welcher Tabelle das eingelesene XML Dokument gehört. Um das herauszufinden haben wir den **TableChecker** implementiert. Hier wird das XML Dokument eine Sekunde lang, bzw. bis zum Ende, falls dies weniger Zeit benötigt, eingelesen und währenddessen überprüft zu welcher Tabelle das Dokument gehört. Dies wird erreicht in dem gezählt wird, wie viele Endzustände, zu welcher Tabelle gehören. Hierfür gibt es die Variable *counter*, die für jeden Tabellenkandidaten, die Anzahl speichert. Die Tabelle mit der höchsten Anzahl wird am Ende, in der Methode *finishCheck()*, mit einer **CounterResult** Exception zurückgegeben, um den eventuell weiteren Lesevorgang des XML Dokuments zu beenden, siehe Kapitel 2.

4.3.3. XMLContentHandler

Für das Extrahieren der Daten ist ein separater SAX Handler zuständig. Für dieses Aufgabe haben wir den **XMLContentHandler** implementiert. Hier werden die zu extrahierenden

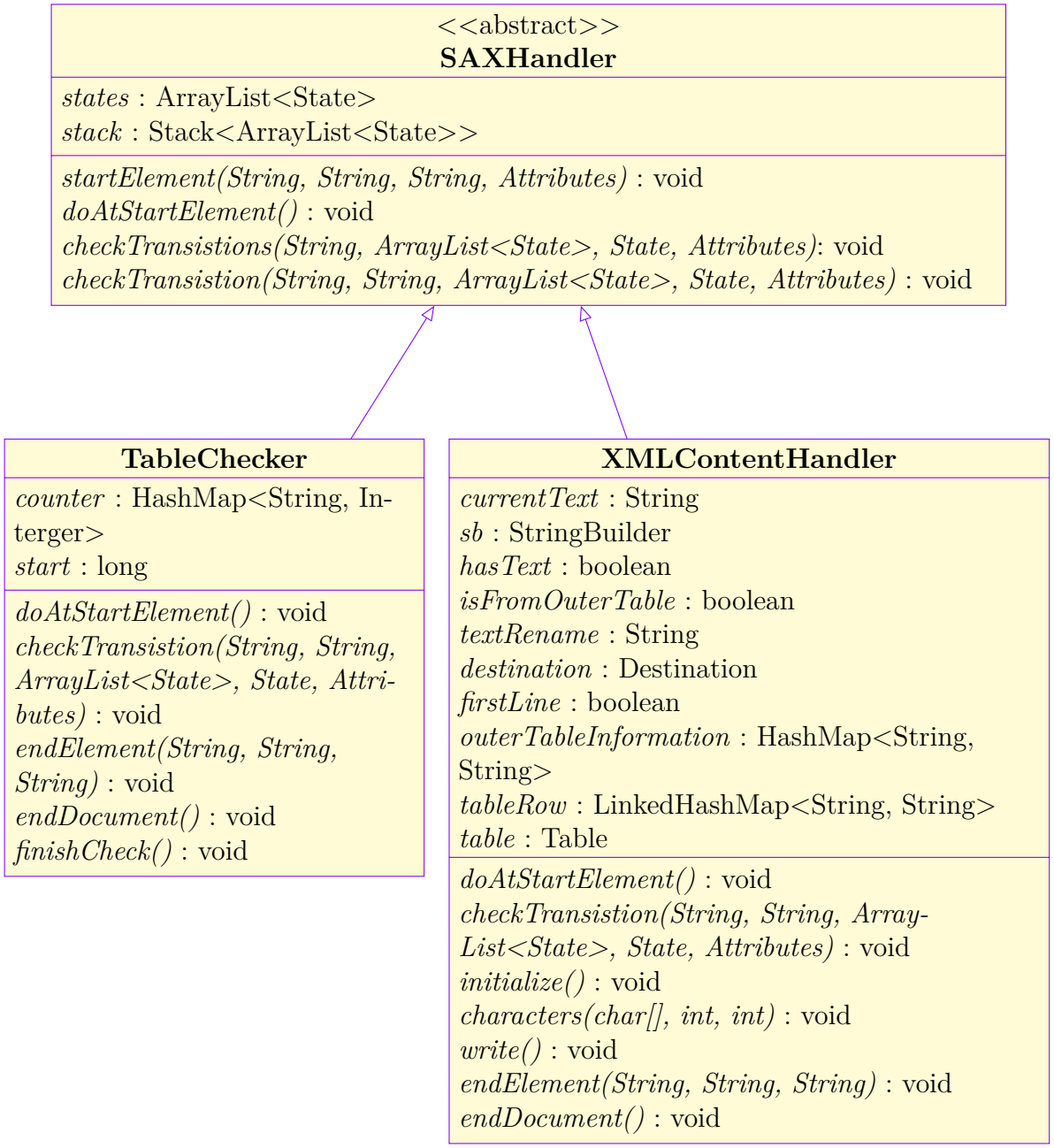

Abbildung 4.3.: Klassendiagramm für SAX

Daten einer Zeile in einer LinkedHashMap (*tableRow*) zwischengespeichert. Als Datenstruktur wurde eine LinkedHashMap benutzt, da in dieser, die Reihenfolge beibehalten wird. Als Schlüssel werden die Spaltennamen der Tabelle benutzt.
Wenn eine Zeile komplett eingelesen wurde, wird der Inhalt an den Output (*destination*) übergeben und die Werte der *tableRow* werden wieder auf null gesetzt. Die Daten einer eventuellen äußeren Tabelle, siehe Kapitel 3.2.2 Verschachtelung, die in einer HashMap (*outerTableInformation*) zwischengespeichert werden, werden im Anschluss wieder in die *tableRow* eingetragen.
Das Extrahieren der Daten findet zum Teil in der Methode *checkTransition(...)* statt. Falls es sich hier bei den neuen Zuständen um Endzustände handelt und sie zu dieser Tabelle gehören, werden zum einen die Liste aus Informationen des Zustandes geholt und zum anderen wird überprüft ob dieser Zustand eine neue Zeile angibt. Sollte der Zustand eine neue Zeile angeben und es nicht die erste Zeile sein (Attribut *firstLine* ist false), so wird, wie bereits erwähnt, der Inhalt von *tableRow* an den Output übergeben und wieder auf null gesetzt.
Bei der Liste aus Informationen, wird für jede **Information** überprüft ob es sich um ein Attribut oder um Text zwischen den XML tags handelt.
Sollte es sich um Text handeln, so kann man die Daten zu diesem Zeitpunkt noch nicht zwischenspeichern, da sie noch nicht eingelesen wurden. Deswegen werden hier nur verschiedene Attribute der Klasse (*hasText*, *textRename* für den (unbenannten) Spaltennamen und *isFromOuterTable* falls der Text zu einer äußeren Tabelle gehört) gesetzt, die am Ende des Elements ausgewertet werden.
Sollte es sich um ein Attribut handelt, so werden die Daten davon, aus dem passenden Attribut von den, aus *startElement(...)*, mitgelieferten *Attributes* gewonnen.
Um den Text zwischen den XML tags zu speichern ist man auf das *characters* Event angewiesen. Hier wird einem ein char Array gegeben, mit einem Startpunkt und der Länge, aus den man den gerade eingelesen Text extrahieren kann. Es ist aber nicht gewährleistet, dass der komplette Text zwischen zwei XML tags unbedingt in einem *characters* Event kommt. Deswegen muss man die Zeichen von allen *characters* Events zwischen *startElement* und *endElement* miteinander verbinden. Aus Performance gründen haben wir uns dabei für einen StringBuilder entschieden. Der StringBuilder wird bei jedem neuen Start eines Elementes wieder entleert (durch das setzen der Länge auf 0).

4.4. Output

Zu diesem Zeitpunkt besteht die Möglichkeit, die Daten in einer CSV Datei auszugeben. Für eine leichtere Erweiterung auf andere Formate, haben wir das Interface **Destination** erstellt, welches die Ausgabeformate implementieren müssen.
Als Attribute besitzt die **CSV** Klasse einen Writer um die Datei zu schreiben und einen Pfad wohin die Datei geschrieben werden soll. Des weiteren eine Table, die Informationen zu den Spaltennamen für den Tabellenkopf hat, einen Null Wert, der geschrieben wird wenn es zu dieser Spalte keine Informationen gibt (in der LinkedHashMap den Wert null hat), einen Delimiter, der die einzelnen Spalten von einander trennt und eine Angabe in welche

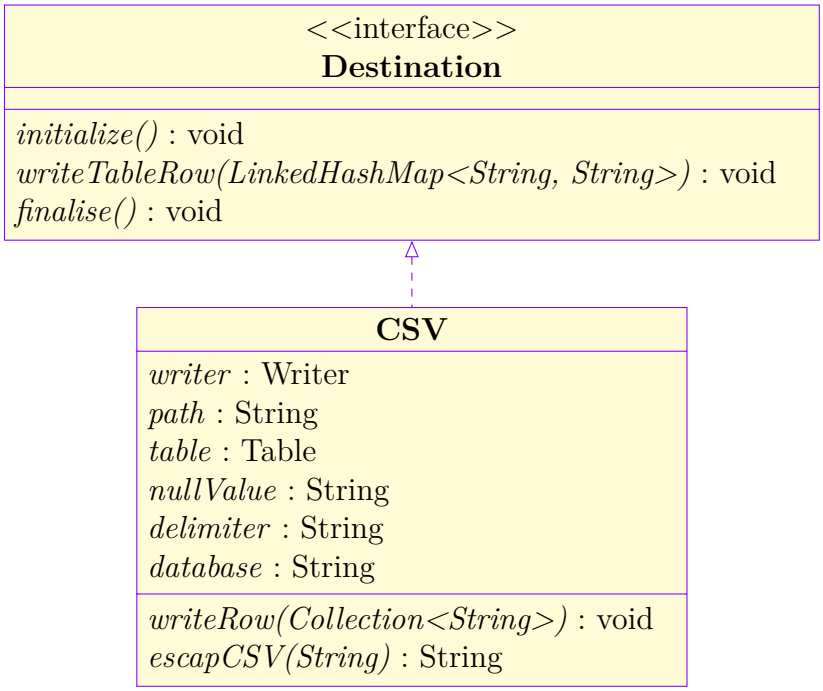

Abbildung 4.4.: Klassendiagramm des Outputs

Datenbank die CSV Datei eingelesen werden soll.
Mit der Methode *initialize()* wird der Tabellenkopf in die Datei geschrieben und mit den Methoden *writeTableRow(...)* und *writeRow(...)* werden die zu diesem Zeitpunkt zwischengespeicherten Inhalte in die Datei geschrieben und anschließend in der LinkedHashMap auf null gesetzt.
Bevor die Inhalte in die CSV Datei geschrieben werden können muss überlappendes Escaping maskiert werden. Hierfür ist die Methode *escapeCSV(...)* da. In dieser wird überprüft ob sich im gegeben String ein Delimiter, ein Zeilenumbruch oder Anführungszeichen befinden. Sollte dies der Fall sein so werden alle möglicherweise vorhandenen Anführungszeichen mit zwei Anführungszeichen ersetzt und der komplette String wird mit Anführungszeichen umschlossen. Dadurch wird beim Einlesen in eine Datenbank das ungewollte Escaping verhindert.
Ist als Zieldatenbank PostgreSQL angegeben, so muss auf einen weiteren Spezialfall geachtet werden. PostgreSQL bietet die Möglichkeit CSV Dateien mittels eines COPY Befehls zu importieren. Dafür werden aber Rechte benötigt die man unter Umständen nicht hat. Als alternative dafür gibt es den \COPY Befehl. Hierbei besteht jedoch das Problem, wenn in einer Zeile nur Backslash Punkt (\.) steht, es als End of Data erkannt wird und somit der Rest nicht mehr gelesen wird[5].

```
...
\.
...
```

Um dies zu umgehen muss man diese Zeichenabfolge in Anführungszeichen setzen.

```
...
"\."
...
```

Da dieser Spezialfall bei MySQL nicht auftritt, wird eine CSV Datei mit dieser Regelung falsch eingelesen. Aus diesem Grund muss der Benutzer vor der Konvertierung angeben, für welche Zieldatenbank die CSV Datei bestimmt ist.
Mit der *finalise()* Methode wird schließlich beim Writer ein *flush()* ausgeführt und er wird geschlossen.

5. Messungen

Die Messung aus Abbildung 5.1, wurde bei der Konvertierung des XML Dumps *jawiki-20140714-pages-meta-history3*[3], mit einer Größe von 66,79 GB, mittels JConsole (Version 1.7.0_45-b31), durchgeführt. JConsole ist eine in JDK mitgelieferte Applikation zur Überwachung von Java Prozessen mittels Java Management Extension (JMX).
Die Messung fand auf einen PC mit dem Kernel Linux 3.4.63-2.44-desktop x86_64, der Distribution openSUSE 12.2 (x86_64) und dem KDE 4.8.5 (4.8.5) "release 2" statt. Der PC ist ausgestattet mit einem Intel(R) Pentium(R) D CPU 3.40GHz Prozessor und 2GB Arbeitsspeicher.
Die, für die Messung, verwendete Java Version war "1.7.0_45" und die Version der VM war OpenJDK 64-Bit Server VM (build 24.45-b08, mixed mode).
Zu Erkennen ist bei dieser Messung, dass der Speicherverbrauch des Programms relativ konstant bleibt und sich nicht proportional zu der Menge an eingelesenen Daten verhält.

Eine andere Messung, siehe Abbildung 5.2, bei der die Datei *dewiki-20131024-pages-meta-history2*[3], mit der Größe von 803 GB, konvertiert wurde, fand auf einem Linux Server statt. Die Messung zeigt auch einen relativ konstanten Verlauf an, allerdings mit einem hohen Durchschnitts Speicherverbrauch.
Die Messung wurde mit dem Befehl

```
ps -o pid,uname,pcpu,pmem,comm,rss -p $(pidof java)
```

alle zehn Sekunden, in einer Schleife, durchgeführt. Für die Messung des Speicherverbrauch wurde hierbei die RSS (Resident Set Size) verwendet, was die Summe aus der Größe des Codes und der Größe der Daten angibt. Allerdings wird bei der Messung mit RSS auch der Speicherverbrauch der sogenannten "Shared Libraries" dazu gezählt, welche allen Programmen zur Verfügung stehen aber nur einmal im Arbeitsspeicher auftauchen. Dies könnte auch eine mögliche Erklärung für den hohen Durchschnitts Speicherverbrauch der Messung sein. Die genaue Ursache konnten wir jedoch, in der Kürze der Zeit, nicht ermitteln.

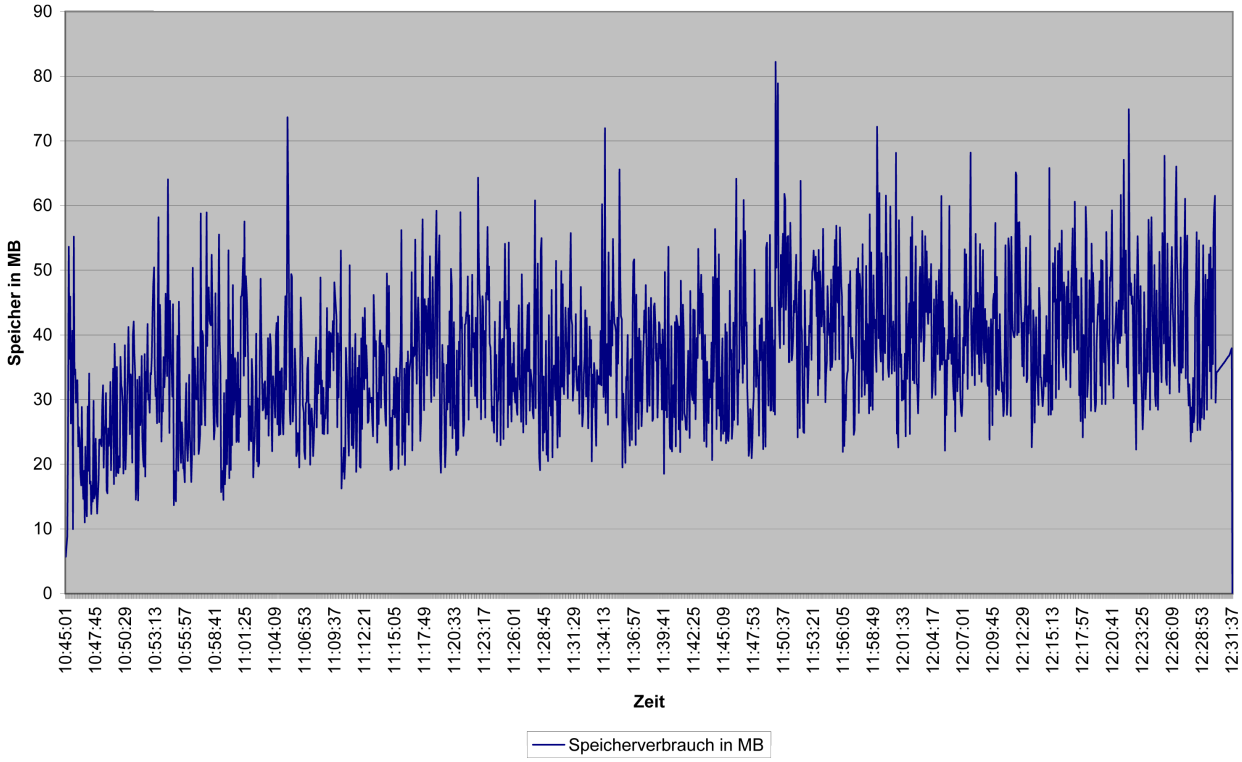

Abbildung 5.1.: Speicherverbrauch während Konvertierung, Messung mit JConsole

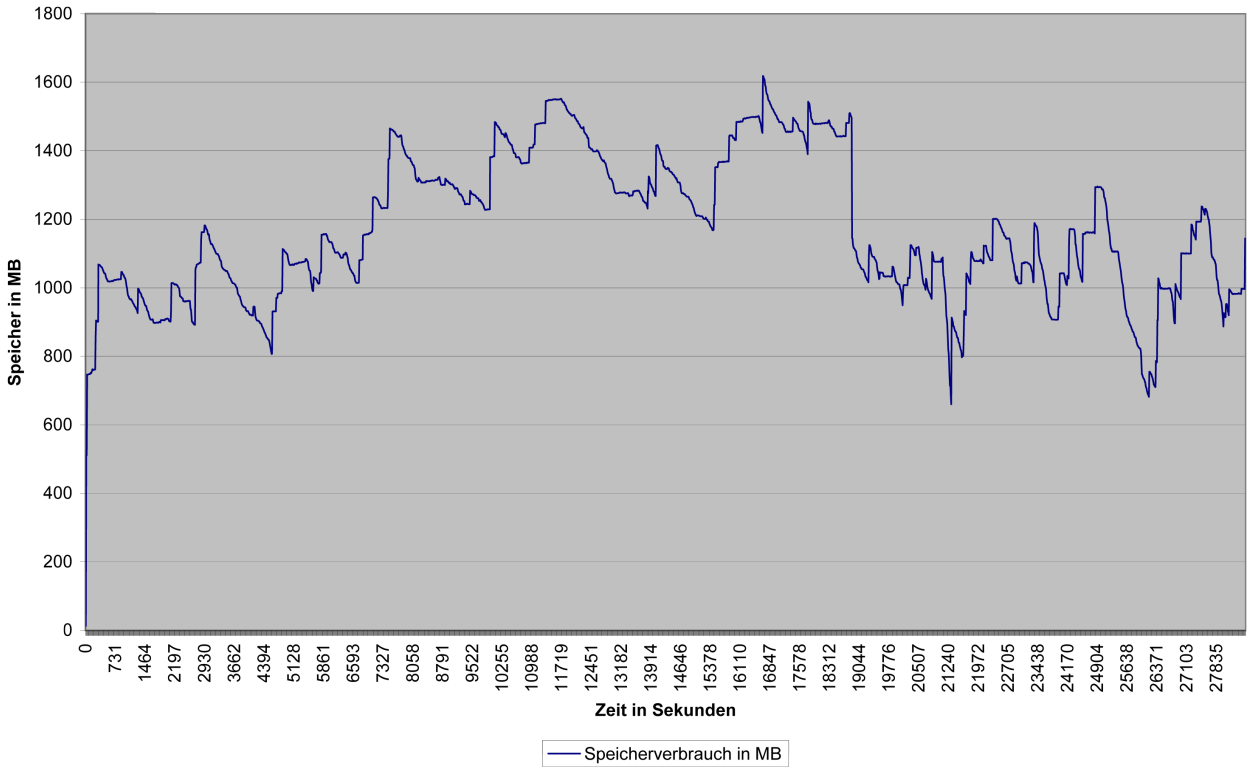

Abbildung 5.2.: Speicherverbrauch während Konvertierung, Messung mit RSS

6. Zusammenfassung

Im Zuge dieser Arbeit haben wir ein Tool entwickelt, welches die Konvertierung von XML Dumps in CSV Dateien ermöglicht. Out of the box werden XML Dumps von Wikipedia und StackOverflow unterstützt, d.h. der Benutzer kann diese Dumps direkt umwandeln ohne eigene Queries festzulegen.
Die Dateigröße der XML Dumps spielt für die Konvertierung keine große Rolle und wird erst durch äußere Faktoren, wie z.B. die Hardware oder das Dateisystem, beschränkt.
Unsere Messungen deuten an, dass der Speicherverbrauch, während des Konvertierungsvorgangs relativ Konstant bleibt und sich nicht proportional zu der Menge der eingelesenen Daten verhält.

6.1. Ausblick

Durch die leichte Erweiterbarkeit des NFA, lässt sich die Unterstützung des Programms einfach auf weitere XML Dumps erweitern. Hierzu müssen nur neue Queries hinzugefügt werden.
Auch lässt sich die Unterstützung auf weitere Datenbanksysteme einbauen. Hierfür müssen die Ausgegebenen SQL Befehle erweitert werden und die CSV Datei auf mögliche Sonderfälle angepasst werden.
Des weiteren wäre es möglich den Output, durch das Interface **Destination**, auf weitere Ausgabeformate zu erweitern oder eine Möglichkeit einzubauen, die Daten direkt in eine Datenbank einzutragen.

Abbildungsverzeichnis

Literaturverzeichnis

[1] Brownell, David: *SAX2*. Sebastopol, CA, USA : O'Reilly & Associates, Inc., 2002. – ISBN 0–596–00237–8

[2] Diao, Yanlei ; Franklin, Michael J.: High-Performance XML Filtering: An Overview of YFilter. In: *IEEE Data Engineering Bulletin* 26 (2003), S. 41–48

[3] `http://dumps.wikimedia.org/backup-index.html`

[4] `https://archive.org/details/stackexchange`

[5] `http://www.postgresql.org/docs/9.2/static/app-psql.html#APP-PSQL-META-COMMANDS-COPY`

[6] `http://www.postgresql.org/docs/9.2/static/sql-copy.html`

[7] `http://dev.mysql.com/doc/refman/5.1/de/load-data.html`

A. Anhang

A.1. Parsen der Queries

Methoden der Klasse **RulesParser** zum Umwandeln der Queries in einen NFA und zur Erstellung der Struktur der Tabellen

Listing A.1: parse

```
1  //wandelt Regeln in einen NFA um
2    public State parse(ArrayList<Table> tables){
3       //Anzahl der Zustaende
4       numberOfStates = 1;
5       //Startzustand
6       State state = new State(1);
7       //Tabelle mit allen Zustaenden
8       HashMap<Integer, State> table = new HashMap<Integer, State>();
9       //Startzustand in die Tabelle
10      table.put(1, state);
11      //geht Schleife fuer alle Regeln durch
12      for (Rule r :this.rules){
13         //hole die rule aus der momentanen Regel
14         String rule = r.getRule();
15         //letztes tag
16         String tag = "";
17         //ID des momentanen Zustandes
18         int currentStateID = 1;
19         //setzt Startzustand als momentanen Zustand
20         state = table.get(1);
21         //entfernt das erste "/" aus der Regel
22         rule = rule.substring(1);
23         //gibt an, an wievielter Stelle sich das naechste "/" befindet
24         int index = rule.indexOf("/");
25         //geht Schleife durch, solange es weitere "/" in dieser Regel gibt
               (falls keine weiteren vorhanden: index = -1)
26         while(index != -1){
27            //String der bis zum naechsten "/" = uebergangssymbol
28            String tmp = rule.substring(0, index);
29            tag = tmp;
```

```
30            //der restliche String ergibt die neue rule
31            rule = rule.substring(index+1);
32            //Falls nicht "//" ("//" entspricht tmp = "")
33            if (!tmp.equals("")) {
34               //erstellt uebergang mit tmp falls noch nicht vorhanden und
                     gibt neue currentStateID zurueck
35               currentStateID = createTransition(state, table, tmp, false, r)
                     ;
36            }
37            //Falls "//" (tmp = "")
38            else{
39               //erstellt uebergang mit "epsilon" falls noch nicht vorhanden
                     und gibt neue currentStateID zurueck
40               currentStateID = createTransition(state, table, "epsilon",
                     false, r);
41            }
42            //der neue momentane Zustand wird in state gespeichert
43            state = table.get(currentStateID);
44            //index des naechsten "/" in rule
45            index = rule.indexOf("/");
46         }
47         //Tabellenname
48         String tableName = r.getTableName();
49         //falls Regel mit "text()" aufhoert
50         if(rule.contains("text()")){
51            //falls sich eine unbenennung ($) in der Regel befindet
52            String renameTo = tag;
53            if(rule.contains("$")){
54               String[] split = rule.split("\\$");
55               renameTo = split[1];
56            }
57            //fuegt dem Zustand passende Informationen hinzu
58            state.addInformation("text", tag, renameTo, tableName, rule.
                  contains("#"));
59            //erzeugtTabelle mit Spalte bzw. fuegt Spalte bestehender Tabelle
                  ein
60            generateTables(tables, tableName, renameTo);
61            //momentanenn Zustand auf Endzustand setzen
62            state.setTypeInformation("Endzustand");
63            //fuege Tabellenname zum Zustand hinzu
64            state.addTableName(tableName);
65         }
66
```

```
67          //falls Regel mit Attributen (@...) aufhoert
68          else if(rule.contains("@")){
69             //das erste @ entfernen
70             rule = rule.substring(1);
71             //die einzelnen Attribute splitten
72             String attributes[] = rule.split("@");
73             //dem momentanen Zustand alle Attribute hinzufuegen
74             for(String attr : attributes){
75                boolean staticInformation = false;
76                if(attr.contains("#")){
77                   attr = attr.substring(1);
78                   staticInformation = true;
79                }
80                //falls sich eine unbenennung ($) in der Regel befindet
81                String renameTo = attr;
82                if(attr.contains("$")){
83                   String[] split = attr.split("\\$");
84                   attr = split[0];
85                   renameTo = split[1];
86                }
87                state.addInformation("attribute", attr, renameTo, tableName,
                     staticInformation);
88                //erzeugtTabelle mit Spalte bzw. fuegt Spalte bestehender
                     Tabelle ein
89                generateTables(tables, tableName, renameTo);
90                //state.addAttribute(attr);
91             }
92             //momentanen Zustand auf Endzustand setzen
93             state.setTypeInformation("Endzustand");
94             //fuege Tabellenname zum Zustand hinzu
95             state.addTableName(tableName);
96          }
97
98          //falls mit Zeilenumbruch aufhoert
99          else if(rule.equals("\n")){
100            //gibt an dass bei diesen Zustand eine neue Zeile kommt
101            state.setNewLine(true);
102            state.setTypeInformation("Endzustand");
103            //fuege Tabellenname zum Zustand hinzu
104            state.addTableName(tableName);
105         }
106
107         //falls Regel mit weiterem uebergang endet
```

```
108         else{
109            //erstellt uebergang mit rule falls noch nicht vorhanden und gibt
                   neue currentStateID zurueck
110            currentStateID = createTransition(state, table, rule, true, r);
111         }
112      }
113      //Startzustand zurueckgeben
114      return table.get(1);
115   }
```

Listing A.2: createTransition

```
1  //erzeugt einen uebergang falls dieser noch nicht besteht und gibt die ID
       des neuen momentanen Zustandes zurueck
2     private int createTransition(State state, HashMap<Integer, State> table,
          String transistionSymbol, boolean end, Rule r) {
3        int currentStateID;
4        //Falls dieser uebergang (vom momentanen Zustand mit transistionSymbol
             zu einem neuen Zustand) noch nicht existiert
5        if (!state.existTransition(transistionSymbol)){
6           //Anzahl der Zustaende wird erhoeht
7           numberOfStates = numberOfStates + 1;
8           //ID des momentanen Zustandes (der neue Zustand) wird angepasst
9           currentStateID = numberOfStates;
10          //neuer Zustand wird erstellt
11          State tmpState = new State(numberOfStates);
12          //falls transistionSymbol epsilon ist, dem neuen Zustand die
                typeInformation "//Kind" hinzufuegen
13          if(transistionSymbol.equals("epsilon")){
14             tmpState.setTypeInformation("//Kind");
15          }
16          //falls es sich um einen Endzustand handelt
17          if (end){
18             tmpState.setTypeInformation("Endzustand");
19             //fuege Tabellenname zum Zustand hinzu
20             tmpState.addTableName(r.getTableName());
21          }
22          //uebergang vom alten Zustand in den neuen wird erzeugt
23          state.addTransistion(transistionSymbol, tmpState);
24          //neuer Zustand wird in die Tabelle eingefuegt
25          table.put(numberOfStates, tmpState);
26       }
27       //Falls dieser uebergang schon existiert
28       else{
```

```
29          //ID des momentanen Zustandes (der ueber den uebergang erreicht
                wird) wird angepasst
30          currentStateID = state.getTransistion(transistionSymbol).getId();
31       }
32       return currentStateID;
33    }
```

Listing A.3: generateTables

```
1  //erzeugtTabelle mit Spalte bzw. fuegt Spalte bestehender Tabelle ein
2     private void generateTables(ArrayList<Table> tables, String tableName,
3           String columnName) {
4        //hole passende Tabelle
5        Table table = getTable(tables, tableName);
6        if(table == null){
7           table = new Table(tableName, columnName);
8           tables.add(table);
9        }
10       else{
11          table.addColumn(columnName);
12       }
13    }
```

Listing A.4: getTable

```
1  //hollt passende Tabelle aus Liste
2     private Table getTable(ArrayList<Table> tables, String tableName) {
3        for (Table table : tables){
4           if (table.getName().equals(tableName)){
5              return table;
6           }
7        }
8        return null;
9     }
```

A.2. SQL Befehle zur Importierung

SQL Befehle für die Importierung der CSV Datei in die jeweilige Datenbank. Dabei steht TableName für den Namen der Tabelle in der die Daten importiert werden sollen, Pfad steht für den Pfad der CSV Datei und delimiter steht für das Zeichen dass die Einträge von einander trennt, wie z.B. ein Komma

A.2.1. PostgreSQL

SQL Befehl für die Importierung der CSV Datei in eine PostgreSQL Datenbank [6]. Falls Tab (\t) der Delimiter ist muss ein E davor angegeben werden: E'\t'. Wie null Werte in der CSV Datei dargestellt werden wird in 'nullWert' angegeben.

```
1 \COPY TableName
2 FROM 'Pfad'
3 WITH DELIMITER 'delimiter'
4 CSV HEADER
5 NULL 'nullWert'
```

A.2.2. MySQL

SQL Befehl für die Importierung der CSV Datei in eine MySQL Datenbank [7].

```
1 LOAD DATA LOCAL INFILE 'Pfad'
2 INTO TABLE TableName
3 FIELDS TERMINATED BY 'delimiter'
4 ENCLOSED BY '"'
5 LINES TERMINATED BY '\n'
6 IGNORE 1 LINES
```