INSTITUT FÜR INFORMATIK
Datenbanken und Informationssysteme

Universitätsstr. 1 D–40225 Düsseldorf

HEINRICH HEINE
UNIVERSITÄT
DÜSSELDORF

Extraktion relationaler Daten für das Semantic Web in Oracle 10g

Michael Matuschek

Bachelorarbeit

Beginn der Arbeit:	09. Mai 2006
Abgabe der Arbeit:	30. Juni 2006
Gutachter:	Prof. Dr. Stefan Conrad
	Prof. Dr. Michael Leuschel

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 30. Juni 2006

Michael Matuschek

Zusammenfassung

Im Zuge der Entwicklung des Semantic Web, aber auch in anderen Szenarien (z.B. Peer-to-Peer-Datenbanken), ist die Extraktion der Daten und Metadaten eines relationalen Datenbanksystems in geeigneter, einheitlicher Form von zunehmender Bedeutung. Einen geeigneten Rahmen für eine solche Extraktion stellt die Relational.OWL-Ontologie dar, die auf RDF und OWL basiert und für die Repräsentation von relationalen Datenbanken konzipiert wurde. Sie ermöglicht es, sowohl die Schemainformationen als auch die tatsächliche Datenbankausprägung in XML-Form darzustellen, wodurch eine relativ einfache Weiterverarbeitung an anderer Stelle ermöglicht wird.

Im Rahmen dieser Arbeit wurde beispielhaft untersucht, ob und wie eine solche Repräsentation mit Hilfe von Stored Procedures bzw. Stored Functions möglich ist, d.h. ohne den Rückgriff auf externe Programme, deren Einsatz nicht in allen Fällen möglich oder wünschenswert ist. Verwendet wurde hierzu das Oracle 10g-Datenbanksystem. Dabei wurden zunächst mit geeigneten Mechanismen des Systems bzw. der verwendeten Programmiersprache PL/SQL sowohl Daten auch als Metadaten in XML extrahiert und anschließend in die gewünschte Form gebracht. Zum Einsatz kamen dabei zwei unterschiedliche Mechanismen, die von diesem Datenbanksystem unterstützt werden: Zum einen die XML-Anfragesprache XQuery, zum anderen die Transformationssprache XSLT.

Mit beiden Ansätzen ließ sich die gewünschte Funktionalität realisieren, wobei jedoch u.a signifikante Unterschiede in der Geschwindigkeit festgestellt werden konnten. Die mit XQuery umgesetzte Lösung war sowohl bei der Daten- als auch der Metadatenextraktion dem XSLT-Pendant deutlich unterlegen, obwohl die Lösungen von der Programmlogik her recht ähnlich sind. Damit dürfte sich XQuery für den praktischen Einsatz disqualifizieren. Die XSLT-Lösung hingegen ließ nicht nur den XQuery-Ansatz hinter sich, sie lag darüber hinaus auf einem Niveau mit dem Programm Relational.OWL, welches vergleichbare Funktionalität durch Zugriffe von außerhalb des Datenbanksystems zur Verfügung stellt. Die Hoffnung, durch die Integration ins Datenbanksystem einen relevanten Geschwindigkeitsvorteil bei der Extraktion zu erhalten, erfüllte sich damit zwar nicht. Es wurde jedoch gezeigt, dass ein solcher Ansatz konkurrenzfähig ist und eine Alternative zur Abhängigkeit von externen Programmen darstellen kann. Darüber hinaus lassen die Ergebnisse bereits erahnen, dass naheliegende Erweiterungen der Funktionalität (z.B. Import von Daten in das Datenbanksystem) ebenfalls auf diesem Wege umsetzbar sind, und zwar mit praxistauglicher Leistung.

Inhaltsverzeichnis

1 Einleitung

Relationale Datenbanksysteme in ihrer bekannten Form leisten hervorragende Dienste, wenn es darum geht, Informationen konsistent und effizient zu speichern. Allerdings wird es in der heutigen Zeit immer wichtiger, dass Datenbanken in der Lage sind, die in ihnen gespeicherten Daten, und insbesondere die Struktur dieser Daten, an andere Stellen weiterzugeben.

Dazu wird in der vorliegenden Arbeit ein beispielhafter Ansatz vorgestellt, mit dem die in einem Oracle 10g-Datenbanksystem gespeicherten Daten und Metadaten so repräsentiert werden, dass sie an anderer Stelle leicht zu verarbeiten sind. Der Schlüssel dazu sind Techniken, die im Semantic Web Anwendung finden, und die es erlauben, in strukturierter Form Daten und die Beziehungen zwischen ihnen zu beschreiben. Dabei handelt es sich um das Resource Description Framework (RDF) [MSB04] sowie die Web Ontology Language (OWL) [MH04]. Die in [PC05a] vorgestellte Relational.OWL-Ontologie basiert auf diesen Frameworks und ist speziell für die Beschreibung von Datenbanken und deren Inhalten entwickelt worden. Besonders interessant ist dabei, dass die zugrundeliegenden Beschreibungstechniken eine Repräsentation in XML besitzen. Dadurch sind wir in der Lage, in unserer Lösung die datenbankeigenen Mechanismen zur XML-Generierung auszunutzen und auf dieser Basis die gewünschte Darstellung durch Transformation von XML-Dokumenten zu erreichen. Dazu werden von uns zwei verschiedene Ansätze vorgestellt (XQuery und XSLT), welche sowohl untereinander als auch mit bereits existierenden Lösungen verglichen werden.

2 Motivation

Zentrale Motivation für die vorliegende Arbeit ist die Anbindung relationaler Datenbanken an das Semantic Web. Damit die Informationen einer Datenbank für die entsprechenden Anfragesprachen „verständlich“ werden, müssen sie in geeigneter Form aufbereitet werden, d.h. in RDF bzw. OWL. Dies ist ein wichtiges Ziel, da der größte Teil der derzeit gespeicherten Daten in „klassischen“ relationalen Datenbanken vorliegt, und somit für das Semantic Web nicht oder nur über Umwege nutzbar ist.

Ein weiteres wichtiges Beispiel sind Peer-to-Peer-Datenbanken, in denen Informationen aus verschiedenartigen Quellen ausgetauscht und zusammengeführt werden müssen. Auch hier ist es erforderlich, dass für die Inhalte einer Datenbank eine für den Kommunikationspartner leicht verwertbare Darstellung gefunden wird.

Darüberhinaus ist die strukturierte Wiedergabe von Datenbankinhalten auch in anderen Szenarien wie z.B. der Datensicherung oder der Datenmigration von großem Interesse.

In allen beschriebenen Fällen ist es wünschenswert, die Repräsentation der Datenbank für den Endanwender einfach und effizient zugänglich zu machen. Eine Lösung dafür ist das in JAVA geschriebene Programm Relational.OWL [Pér06], welches das Extrahieren von Daten und Metadaten in geeigneter Form über die Schnittstelle JDBC ermöglicht. Dieser Ansatz hat jedoch Nachteile:

- Es ist nicht sichergestellt, dass das Programm überall dort ausführbar ist, wo es erforderlich ist. Das kann z.B. am Fehlen der erforderlichen JAVA-Laufzeitumgebung oder an Rechtebeschränkungen liegen.
- Das „Zwischenschalten“ eines Programms zwischen Datenbank und tatsächlicher Anwendung kann die Geschwindigkeit und Fehleranfälligkeit der Datenextraktion beeinträchtigen.
- Relational.OWL unterstützt derzeit nur DB2 und MySQL. Für Systeme wie die Oracle 10g gibt es daher momentan keinen Weg, dieses Programm einzusetzen.

Um diese Probleme zu vermeiden, wird in dieser Arbeit eine Implementierung der gleichen Funktionalität mit Hilfe von Stored Procedures bzw. Stored Functions vorgestellt. Das zugrundeliegende Datenbanksystem (in diesem Fall Oracle 10g mit der Programmiersprache PL/SQL) wird damit in die Lage versetzt, seine Daten direkt und ohne weitere Abhängigkeiten in Relational.OWL zur Verfügung zu stellen.

Die restliche Arbeit ist nun wie folgt gegliedert: Zunächst werden in Abschnitt 3 die grundlegenden Technologien bzw. Standards vorgestellt, auf denen die vorgestellte Lösung basiert. Dabei handelt es sich um die Grundzüge von RDF bzw. OWL, die Relational.OWL-Ontologie, die wichtigsten Besonderheiten der verwendeten Programmiersprache PL/SQL sowie eine Diskussion der grundlegenden Mechanismen der beiden Transformationsansätze mit XQuery und XSLT.

Anschließend werden die implementierten Prozeduren und Funktionen im Detail vorgestellt, um ihre Funktionsweise deutlich zu machen. Dabei betrachten wir gesondert die Extraktion der Rohdaten (Abschnitt 4.1), die Transformation mit XQuery (Abschnitt 4.2) bzw. XSLT (Abschnitt 4.3) sowie die abschließende Ausgabe der erstellten Dokumente (Abschnitt 4.4). In Abschnitt 5 vergleichen wir die unterschiedlichen Lösungen, bevor wir in Abschnitt 6 mit der Betrachtung einiger weitergehender Aspekte schließen.

3 Grundlagen

In diesem Abschnitt wollen wir zunächst die Grundlagen bzw. Hilfsmittel erläutern, die für die vorgestellte Lösung relevant sind.

3.1 RDF und OWL

Die Intention des Resource Description Frameworks liegt darin, Informationen über Objekte des World Wide Web bzw. über Objekte, die im WWW identifiziert werden können, maschinenlesbar zu speichern. Dazu kommen sogenannte Tripel zum Einsatz, die aus Subjekt, Prädikat und Objekt bestehen. Dabei ist das Subjekt die Ressource, über die eine Aussage getroffen wird, das Prädikat steht für die beschriebene Eigenschaft der Ressource und das Objekt für den Wert dieser Eigenschaft. Dieses Prinzip ist vergleichbar mit Aussagen, wie sie auch in natürlichen Sprachen getätigt werden können, z.B. „Diese Bachelorarbeit (Subjekt) wurde geschrieben von (Prädikat) Michael Matuschek (Objekt)“.

Diese Aussagen in Form von Tripeln lassen sich leicht als Bäume darstellen, wobei Subjekt und Objekt durch Knoten und das Prädikat durch eine Kante repräsentiert werden. Ein Beispiel dafür findet sich in Abbildung 1, dargestellt sind die Eigenschaften einer Website.

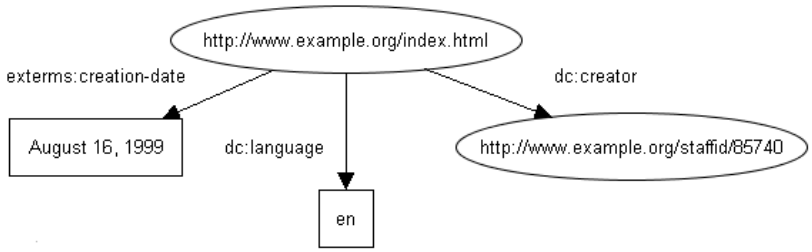

Abbildung 1: Ein RDF-Graph, in Anlehnung an [MSB04].

Aus einer Menge solcher Bäume lassen sich dann Graphen konstruieren, die vielfältige Aussagen über Objekte bzw. deren Beziehungen abbilden. Diese Form der Repräsentation ist für Menschen sehr leicht verständlich, für die maschinelle Verarbeitung interessanter ist jedoch die Repräsentation in XML. Hier dazu nochmal obiges Beispiel, diesmal als XML:

```
<rdf:Description rdf:about="http://www.example.org/index.html">
  <exterms:creation-date>August 16, 1999</exterms:creation-date>
  <dc:language>en</dc:language>
  <dc:creator rdf:resource="http://www.example.org/staffid/85740"/>
</rdf:Description>
```

Die Web Ontology Language stellt eine Erweiterung von RDF dar. Sie basiert auf dem Konzept von Klassen, die bestimmte Eigenschaften haben. Mit einer Sammlung solcher Klassen lassen sich Ontologien beschreiben, wobei sich eine Ontologie nach [Gru93] als „specification of a conceptualization“ definieren lässt. Die konkreten Objekte, über die man mit diesen Ontologien Aussagen treffen kann, bezeichnet man als Instanzen. Das Konzept von OWL bietet eine Vielzahl von Möglichkeiten, die weit über das hinausgehen, was mit RDF-Tripeln möglich ist, z.B. die Beschreibung von Klassen mit Hilfe von Mengenoperationen oder die Festlegung von Kardinalitäten. Details dazu findet man unter [DS04], ein Beispiel für die XML-Darstellung einer solchen Ontologie findet man im folgenden Abschnitt.

3.2 Relational.OWL

Die Repräsentation einer relationalen Datenbank in Relational.OWL [PC05a] gliedert sich grundsätzlich in zwei Teile: Zum einen die Beschreibung des Schemas, d.h. der Struktur der Datenbank, und die Repräsentation der tatsächlichen Datenbankausprägung. Beide Teile werden im folgenden kurz vorgestellt und anhand eines kleinen Beispiels erläutert.

3.2.1 Schemarepräsentation

Die Relational.OWL-Ontologie realisiert die Wiedergabe relationaler Datenbankschemata mit Hilfe geeigneter Klassen von Objekten, die die einzelnen Bestandteile der Datenbank wiederspiegeln, sowie mit diesen Klassen zugeordneten Eigenschaften. Die in der Ontologie definierten Klassen sind:

- `Database`
- `Table`
- `Column`
- `PrimaryKey`

Diese Klassen entsprechen in natürlicher Weise den bekannten Teilen einer relationalen Datenbank. Um den Aufbau der Datenbank beschreiben zu können, müssen diese Einzelteile charakterisiert und zueinander in Beziehung gesetzt werden. Dies geschieht mit den folgenden Eigenschaften:

- `has`
- `hasTable`
- `hasColumn`
- `isIdentifiedBy`
- `length`
- `scale`
- `references`

Die Eigenschaft `has` beschreibt dabei die allgemeine Beziehung, dass ein Objekt andere Objekte enthalten kann. Davon abgeleitet sind die Eigenschaften `hasTable` und `hasColumn`. Eine `Database`-Objekt kann `Table`-Objekte enthalten, ebenso kann ein `Table`-Objekt `Column`-Objekte enthalten. Darüberhinaus wird die `hasColumn`-Eigenschaft auch verwendet, um auszudrücken, dass ein Primärschlüssel (`PrimaryKey`) aus einer oder mehreren Spalten (`Column`) besteht. Ein `Table`-Objekt wird durch ein `PrimaryKey`-Objekt eindeutig identifiziert (`isIdentifiedBy`). Die Eigenschaften `length` und `scale` legen Eigenschaften für Datentypen fest, also z.B. die Länge eines Strings oder die Skalierung einer Dezimalzahl. Die Eigenschaft `references` drückt eine Fremdschlüsselbeziehung zwischen zwei Spalten bzw. Attributen aus. Um das Konzept zu verdeutlichen betrachten wir ein Beispiel.

Gegeben sei eine relationale Datenbank mit folgenden beiden Tabellen:

- Person (Name, Vorname, LandID)
- Land (ID, Name)

Dabei bilden `Name` und `Vorname` einer Person gemeinsam den Primärschlüssel der Tabelle `Person`, `LandID` ist ein Fremdschlüssel, der auf den Primärschlüssel `ID` in der Tabelle `Land` verweist.

Für den im Folgenden betrachteten Ausschnitt aus der Relational.OWL-Repräsentation dieser Datenbank muss Folgendes beachtet werden: Die Namensraumpräfixe `owl`, `rdf` und `rdfs` beziehen sich auf die bekannten Namensräume von OWL [MH04], RDF [MSB04] und RDF Schema [GB04]. Der Namensraum `dbs` bezieht sich auf die Definiton der Relational.OWL-Ontologie [PC05b].

Das Ergebnis sieht wie folgt aus:

```
<...>
<owl:Class rdf:ID="PERSON">
 <rdf:type rdf:resource=
  "http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#Table" />
 <dbs:hasColumn rdf:resource="#PERSON.NAME" />
 <dbs:hasColumn rdf:resource="#PERSON.VORNAME" />
 <dbs:hasColumn rdf:resource="#PERSON.LANDID" />
 <dbs:isIdentifiedBy>
  <dbs:PrimaryKey>
    <dbs:hasColumn rdf:resource="#PERSON.NAME" />
    <dbs:hasColumn rdf:resource="#PERSON.VORNAME" />
  </dbs:PrimaryKey>
 </dbs:isIdentifiedBy>
</owl:Class>
<owl:DatatypeProperty rdf:ID="PERSON.LANDID">
 <rdf:type rdf:resource=
  "http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#Column" />
 <rdfs:domain rdf:resource="#PERSON" />
 <rdfs:range rdf:resource=
  "http://www.w3.org/2001/XMLSchema#integer" />
 <dbs:references rdf:resource="#LAND.ID" />
</owl:DatatypeProperty>
</...>
```

3.2.2 Datenrepräsentation

Nachdem die Schemarepräsentation festgelegt ist, können die in der Datenbank gespeicherten Daten auf einfache Weise dargestellt werden, und zwar als Instanz der Schemaontologie. Das hat zur Folge, dass die Datenrepräsentation tatsächlich nur die Daten enthalten muss. Spezielle Eigenschaften des Schemas (z.B. Fremdschlüsselbeziehungen) müssen nicht weiter berücksichtigt werden, da diese Informationen in der zugrundeliegenden Ontologie bereits enthalten sind. Der Bezug zu dieser Ontologie wird in diesem Fall über den Namensraum `dbinst` hergestellt.

Das Ergebnis der Datenrepräsentation:

```
<...>
<dbinst:PERSON>
 <dbinst:PERSON.NAME>Wurst</dbinst:PERSON.NAME>
 <dbinst:PERSON.VORNAME>Hans</dbinst:PERSON.VORNAME>
 <dbinst:PERSON.LANDID>1</dbinst:PERSON.LANDID>
</dbinst:PERSON>
</...>
```

3.3 PL/SQL

Nach der Vorstellung des theoretischen Hintergrundes der Problemstellung werden wir jetzt die technischen „Hilfsmittel“ betrachten, die bei der Umsetzung der Lösung verwendet werden. Den Anfang macht dabei die Oracle-eigene Programmiersprache PL/SQL (Procedural Language/SQL), in der alle vorgestellten Prozeduren und Funktionen geschrieben sind.

Wie der Name schon sagt, ist PL/SQL eine prozedurale Programmiersprache, die in ihrem Aufbau (Kontrollstrukturen, Ausdrücke usw.) weitgehend mit bekannten Sprachen wie z.B. C vergleichbar ist. Daher sollen diese Aspekte hier auch nicht weiter betrachtet werden. Für eine vollständige Beschreibung der Sprache sei an dieser Stelle auf [FP03] und [Urm02] verwiesen. Im Folgenden wollen wir uns den Aspekten widmen, die PL/SQL von anderen Sprachen abgrenzen, und die in der vorliegenden Lösung zur Anwendung kommen.

3.3.1 Ausnahmebehandlung

Die Definition einer Prozedur bzw. Funktion in PL/SQL hat grundsätzlich folgenden Aufbau:

```
CREATE FUNCTION Funktionsname
(
-- Hier werden die Argumente deklariert
)
RETURN Typangabe -- Fehlt bei Prozeduren
AS
-- Hier werden Variablen und Exceptions deklariert
BEGIN
-- Hier steht das eigentliche Programm
EXCEPTION
-- Hier werden Ausnahmen behandelt
END;
```

Die Besonderheit ist der `EXCEPTION`-Block (siehe dazu auch [FP03], Kap.6). Hier werden zentral für die gesamte Prozedur bzw. Funktion die Ausnahmen behandelt. Sobald eine Anweisung (egal welche) eine Ausnahme auslöst, wird die Ausführung des Programms abgebrochen und die entsprechende Ausnahmebehandlung ausgelöst. Das sieht z.B. wie folgt aus:

```
WHEN UTL_FILE.ACCESS_DENIED THEN
DBMS_OUTPUT.PUT_LINE('Kein Zugriff auf Datei möglich');
```

Der `EXCEPTION`-Block kann auch fehlen, dann wird die Ausnahme an den Aufrufer weitergegeben. Dies ermöglicht bei einem Programm, welches selbst Prozeduren und Funktionen aufruft, die Fehlerbehandlung auf oberster Ebene zu bündeln. Ausnahmen können des Weiteren auch selbst definiert werden. Sie werden einfach wie Variablen mit dem

Datentyp EXCEPTION deklariert und können zur Laufzeit mit dem Befehl raise manuell ausgelöst werden. Dies wird z.B. benutzt, um „inhaltliche“ Fehler abzufangen, die vom System naturgemäß nicht erkannt werden.

3.3.2 Einbettung von SQL

Das bemerkenswerteste Merkmal von PL/SQL ist sicherlich die nahtlose Einbettung von SQL-Anweisungen in das Programm. Als Beispiel wird hier die SELECT-Anweisung in Anlehnung an [FP03], Kap.14 beschrieben, die für die vorliegende Lösung von großer Bedeutung ist.

Eine SELECT-Anweisung kann einfach in den PL/SQL-Code hineingeschrieben werden. Dabei ist jedoch zu beachten, dass das Ergebnis der Anfrage mit einer INTO-Klausel an Variablen gebunden werden muss. Der in der Variablen gespeicherte Wert kann dann vom Programm weiterverarbeitet werden. Das kann wie folgt aussehen:

```
CREATE PROCEDURE beispiel1
AS
id_variable NUMBER;
BEGIN
SELECT ID INTO id_variable FROM LAND WHERE NAME='Deutschland';
END;
```

Komplizierter ist der Fall, wenn eine SQL-Anfrage mehrere Zeilen zurückliefert. Die Bindung an eine normale Variable würde dann einen Laufzeitfehler auslösen. Hier kommen sogenannte CURSOR-Variablen zum Einsatz, die eine Iteration über die zurückgelieferten Tupel ermöglichen. Die Schleife bricht ab, sobald kein Tupel mehr gefunden wird. Ein Beispiel:

```
CREATE PROCEDURE beispiel2
AS
id_variable NUMBER;
CURSOR id_cursor IS SELECT ID FROM LAND;
BEGIN
OPEN id_cursor;
LOOP
FETCH id_cursor into id_variable;
EXIT WHEN id_cursor%NOTFOUND
CLOSE id_cursor;
END;
```

3.3.3 XMLType

Die letzte Besonderheit, die an dieser Stelle Erwähnung finden soll, ist weniger eine Eigenschaft von PL/SQL als vielmehr ein Merkmal des Datenbanksystems Oracle 10g. Dieses hat einen speziellen Datentyp namens XMLType, welcher explizit zur Speicherung

von XML-Dokumenten in relationalen Datenbanken gedacht ist, d.h. ein gewöhnliche Tabelle kann eine Spalte vom Typ `XMLType` aufweisen. Eine sehr ausführliche Beschreibung dieses Datentyps findet man im entsprechenden Abschnitt in [Hig03].

Der Typ `XMLType` besitzt einige Vorteile gegenüber einer „naiven“ Speicherung von XML-Dateien, z.B. als CLOB. Zum einen verwaltet dieser Typ intern die „Baumstruktur“ eines XML-Dokuments, was (wie wir später sehen werden) von essenzieller Bedeutung für die Navigation in XML-Dokumenten und deren Transformation ist. Zum anderen bietet dieser Typ Methoden an, die die Arbeit mit XML-Dokumenten wesentlich erleichtern. So können die Dokumente mit einem einfachen Methodenaufruf gegen ein Schema validiert, oder durch ein XSLT-Stylesheet transformiert werden. Des Weiteren besitzt der Typ einen Konstruktor, der die Erstellung eines `XMLType`-Objekts aus einer Zeichenkette ermöglicht, welche ein XML-Dokument darstellt. In PL/SQL kann dieser Datentyp wie jeder andere auch verwendet werden. Beispielsweise liefern die später vorgestellten Funktionen einen Wert vom Typ `XMLType` zurück.

An dieser Stelle wollen wir die allgemeine Betrachtung von PL/SQL beenden. Die Besonderheiten einiger spezieller PL/SQL-Funktionen werden später bei der Besprechung der Lösung detaillierter vorgestellt, da deren Funktionsweise im konkreten Anwendungskontext leichter nachzuvollziehen ist.

3.4 XQuery

In diesem Abschnitt wird die XML-Anfragesprache XQuery betrachtet, die derzeit beim W3C als Candidate Recommendation vorliegt [BCF+05]. XQuery ist dazu gedacht, XML-Dokumente anzufragen und die erhaltenen Informationen in nahezu beliebiger Formatierung auszugeben. Wir wollen uns hier allerdings nur den wichtigsten Asptekten widmen, für eine ausführliche Betrachtung der Sprache sei an dieser Stelle auf [LS04] verwiesen.

3.4.1 Datenmodell und Auswertungskontext

Das Datenmodell, welches XQuery zugrundeliegt, ist die Sequenz. Das bedeutet, dass jedes Ergebnis eines gültigen XQuery-Ausdrucks eine Sequenz aus atomaren Werten (z.B. Zeichenketten) oder Knoten im Sinne von XML darstellt. So wäre Folgendes ein gültiger XQuery-Ausdruck:

```
42, "Hallo" ,<a />,<b>Nochmal Hallo</b>
```

Die Arten von Knoten, die XQuery unterscheidet, sind dabei Dokumentknoten, Elementknoten, Attributknoten, Namensraumknoten, Knoten für Verarbeitungsanweisungen, Kommentarknoten sowie Textknoten (vgl. [LS04], Abschnitt 3.5). Dabei können Knoten selbst wieder Sequenzen enthalten, wodurch sich auf natürlche Weise die Baumstruktur eines XML-Dokuments ergibt. So enthält ein Elementknoten z.B. eine Sequenz von Attributknoten (also die Attribute des Elements), eine Sequenz von Textknoten, die den

Inhalt des Elements darstellen sowie weitere Elementknoten, die die Kindknoten des Elements sind. Wie auch aus obigem Beispiel ersichtlich wird, liefert XQuery aber nicht immer ein wohlgeformtes XML-Dokument. Das Ergebnis eines Ausdrucks kann auch nur eine Sequenz von atomaren Werten sein, oder es kann mehrere Dokumentknoten enthalten. Das Hauptaugenmerk in dieser Arbeit liegt aber in der Erstellung von wohlgeformten Dokumenten, weswegen wir auf diese Eigenheit von XQuery hier nicht weiter eingehen wollen.

Wie bereits im Beispiel zu erkennen war, kann das Ergebnis einer Anfrage bereits die Anfrage selbst sein. Interessanter für unsere Zwecke ist aber, wie der Inhalt eines Dokuments zur Laufzeit dynamisch bestimmt wird. Das zentrale Werkzeug, das XQuery hierfür zur Verfügung stellt, ist der Auswertungskontext, der in der XQuery-Spezifikation [BCF+05] als „enclosed expression“ bezeichnet wird. Er wird durch geschweifte Klammern begrenzt, und er dient als Hinweis für den XQuery-Prozessor, dass der eingeschlossene Ausdruck ausgewertet werden muss. Der einfachste Fall ist, dass der Wert einer Variable ausgewertet wird. Falls also z.B. die Variable `$x` den Wert 42 hat, liefert die Anfrage

```
<Ergebnis>{$x}</Ergebnis>
```

das Ergebnis

```
<Ergebnis>42</Ergebnis>
```

Weiterhin kann z.B. mit einem Auswertungskontext auch der Name eines Elementes festgelegt werden, wobei das Schlüsselwort `element` zum Einsatz kommt. So liefert folgende Anfrage das gleiche Ergebnis wie oben:

```
element{"Ergebnis"} {$x}
```

Der Inhalt eines Auswertungskontextes kann aber auch wesentlich komplexer sein. In den folgenden Abschnitten wollen wir auf Pfadausdrücke, FLWOR-Ausdrücke sowie einige weitere Konstrukte eingehen, die in einem Auswertungskontext verarbeitet werden können.

3.4.2 Pfadausdrücke

Um die Inhalte eines XML-Dokumentes auszuwerten, ist es notwendig, in der Baumstruktur des Dokuments navigieren zu können. Dabei verwendet XQuery eine Syntax, die eng an die von XPath [CD99] angelehnt ist und recht intuitiv verständlich ist. Ausgehend von ein einem so genannten Kontextknoten (der auch ein Dokumentknoten sein kann) kann man auf andere Knoten ähnlich wie in einem Dateisystem zugreifen (vgl. [LS04], Abschnitt 4). Ist die Variable `$x` z.B. statt an einen atomaren Wert an einen Knoten gebunden, würde das Beispiel aus dem vorhergehenden Abschnitt den kompletten Teilbaum ausgeben, mit `$x` als Wurzel. Der folgende Ausdruck würde eine Sequenz aller Kindelemente zurückliefern, die Bezeichnung `MYELEMENT` tragen:

```
<Ergebnis>{$x/MYELEMENT}</Ergebnis>
```

Alternativ dazu kann man sich auch alle Kindknoten ausgeben lassen, unabhängig von ihrer Bezeichnung:

```
<Ergebnis>{$x/*}</Ergebnis>
```

Auf ähnliche Art kann man auch auf alle Attributknoten zugreifen. Attributknoten werden genau wie Elementknoten angesteuert, einziger Unterschied ist ein vorangestelltes `@`:

```
<Ergebnis>{$x/@*}</Ergebnis>
```

Die Einschränkungen an die gewünschten Knoten lassen sich mit sogenannten Prädikaten (im Original „predicates“) noch weiter verfeinern. So liefert die folgende Anfrage alle Knoten mit der Bezeichnung `MYELEMENT`, die selbst widerum einen Kindknoten mit der Bezeichnung `ID` haben, dessen Wert 1 ist:

```
<Ergebnis>{$x/MYELEMENT[ID="1"]}</Ergebnis>
```

Analog lässt sich auch der Wert eines Attributs abfragen. Interessant ist in diesem Zusammenhang, dass auf die Anfrage nach dem Wert auch verzichtet werden kann, um eine existenzielle Prüfung durchzuführen. Folgende Anfrage liefert also nur die `MYELEMENT`-Knoten, die überhaupt einen Kindknoten mit der Bezeichnung `ID` haben:

```
<Ergebnis>{$x/MYELEMENT[ID]}</Ergebnis>
```

Zum Abschluss dieser kurzen Beschreibung der Pfadausdrücke sei noch erwähnt, dass man nicht nur von Kindknoten zu Kindknoten navigieren kann. XQuery bietet eine Vielzahl von Möglichkeiten, um z.B. auf Nachfolger, auf die Geschwisterknoten und auch auf Vorgänger zuzugreifen (vgl. [BCF+05], Abschnitt 3.2). Exemplarisch erwähnt sei hier der Zugriff auf den direkten Vorgänger, der auch ähnlich wie in einem Dateisystem realisiert wird. Folgender Ausdruck liefert also das Element `$x` selbst:

```
<Ergebnis>{$x/..}</Ergebnis>
```

3.4.3 FLWOR-Ausdrücke

In diesem Abschnitt wollen wir uns dem vielleicht wichtigsten Instrument von XQuery widmen, dem FLWOR-Ausdruck. FLWOR wird ausgesprochen wie das englische Wort „flower“ und steht für `for-let-where-order by-return`. Mit diesem Konstrukt sind sehr komplexe Anfragen möglich, Details dazu findet man in [LS04], Abschnitt 5.

Die `return`-Klausel entspricht dabei im Wesentlichen den Ausdrücken, die in den vorherigen Abschnitten vorgestellt wurden. Sie beschreibt die „Schablone“, nach der das

Dokument erstellt wird, und kann ihrerseits selbst wieder XQuery-Ausdrücke enthalten, auch FLWOR-Ausdrücke. Mit dieser Schachtelung ist es z.B. möglich aufwändige Operationen wie Verbund- oder Aggregatbildung zu realisieren. Der Schlüssel dazu ist die Bindung von Sequenzen an Variablen mit den Klauseln `for` und `let`.

Die `for`-Klausel ist vergleichbar mit einer `for`-Schleife in einer Programmiersprache. Sie bindet jeweils einen Wert bzw. Knoten einer Sequenz an eine Variable und führt die nachfolgenden Anweisungen für diese „einelementige Sequenz“ aus. Ein Beispiel dafür ist der folgende Ausdruck:

```
for $x in (1,2,3)
return <x>{$x}</x>
```

Das Ergebnis lautet:

```
<x>1</x><x>2</x><x>3</x>
```

Im Gegensatz dazu bindet die `let`-Klausel immer eine ganze Sequenz (die aber auch einelementig sein darf). Dann sieht obiges Beispiel so aus:

```
let $x:=(1,2,3)
return <x>{$x}</x>
```

Ergebnis:

```
<x>1 2 3</x>
```

Es ist naheliegend, dass diese Klauseln oftmals zusammen auftreten, um z.B. über alle Knoten einer Sequenz zu iterieren und sich für jeden dieser Knoten seine Kindknoten ausgeben zu lassen. Für dieses Beispiel sei `$x` wieder an einen Knoten gebunden:

```
for $y in $x/MYELEMENT
let $z:=$y/*
return <y>{$z}</y>
```

Darüber hinaus lassen sich beliebig viele `for`- und `let`-Klauseln kombinieren, um z.B. geschachtelte Schleifen zu realisieren.

Die `where`-Klausel entspricht in ihrer Semantik der `where`-Klausel aus SQL und ermöglicht die Filterung der Daten nach bestimmten Kriterien:

> „The optional where clause serves as a filter for the tuples of variable bindings generated by the for and let clauses. The expression in the where clause, called the where-expression, is evaluated once for each of these tuples. If the effective boolean value of the where-expression is true, the tuple is retained and its variable bindings are used in an execution of the return clause“([BCF+05], Abschnitt 3.8.2).

Dazu auch ein Beispiel:

```
for $x in (1,2,3)
where $x>1
return <x>{$x}</x>
```

Ergebnis:

```
<x>2</x><x>3</x>
```

Dabei ist insbesondere interessant, wie nicht nur auf atomare Werte, sondern auch auf Knoten und deren Inhalte zurückgegriffen werden kann. Dies wird im nächsten Abschnitt erläutert.

Der Vollständigkeit halber erwähnt sei hier auch die `order by`-Klausel, die es erlaubt, die Reihenfolge der Auswertung einer Sequenz zu beeinflussen. Da diese Klausel in der vorliegenden Lösung keine Anwendung findet, sei an dieser Stelle aber auf [BCF+05], Abschnitt 3.8.3 verwiesen.

3.4.4 Weitere Konstrukte

Neben den bis hierhin vorgestellten Aspekten gibt es noch weitere Ausdrücke, die der Erwähnung wert sind. Während die arithmetischen und logischen Ausdrücke dabei den gängigen Standards entsprechen, gilt es bei den Vergleichsausdrücken eine Besonderheit zu beachten (siehe dazu auch [LS04], Abschnitt 6.2).

Diese werden entsprechend des vorgestellten Datenmodells unterteilt in Vergleiche von einzelnen Werten und Sequenzen. Während die Operatoren (`eq`, `ne`, `lt`, `le`, `gt`, `ge`) für einzelne Werte vorgesehen sind und sich wie erwartet verhalten, sind die Regeln für die zugehörigen Sequenzoperatoren (=, !=, <, <=, >, >=) komplexer, z.B. bei der Definition, wann eine Sequenz von Werten größer als eine andere ist. Da der Vergleich von Sequenzen in der vorgestellten Lösung aber keine Anwendung findet, und sich die Operatoren bei einzelnen Werten identisch verhalten, soll auf diese Besonderheit hier nicht weiter eingegangen werden. Es sollte aber stets bedacht werden, dass sich die beiden Gruppen semantisch unterscheiden, auch wenn sie in der vorgestellten Lösung gegeneinander austauschbar sind.

Beachtung finden soll an dieser Stelle der konditionale Ausdruck (im Original „conditional expression“, vgl. [BCF+05], Abschnitt 3.10), der die aus universalen Programmiersprachen gewohnte Auswertung von Ausdrücken in Abhängigkeit von einer Bedingung ermöglicht. Dazu auch ein Beispiel:

```
for $x in (1,2,3)
return if ($x>1)
          then <x>{$x}</x>
          else ""
```

Dabei ist anzumerken, dass der `else`-Teil in jedem Fall vorhanden sein muss. Wenn wie in diesem Fall keine Alternative vorgesehen ist, kann man das durch Angabe einer leeren Zeichenkette anzeigen.

Zum Schluss dieser kurzen Betrachtung von XQuery sollen hier die Funktionen vorgestellt werden, die im Rahmen der Lösung verwendet werden. Zum einen ist das die Funktion `concat()`, die zwei übergebene Zeichenketten zu einer verknüpft und sich wie ähnliche Funktionen aus anderen Sprachen verhält. Die Funktionen `name()` und `data()` hingegen sind Funktionen, die auf Knoten operieren. Während erstere den Bezeichner eines Knotens liefert (und bei Anwendung auf atomare Werte einen Laufzeitfehler produziert), liefert letztere den Inhalt eines Knotens, z.B. den enthaltenen Text. Damit lassen sich diese Informationen in die vorher beschriebenen Ausdrücke integrieren. Die Anwendung von `data()` auf atomare Werte ist zwar erlaubt, aber sinnlos, da der übergebene Wert unverändert zurückgeliefert wird.

Neben den hier vorgestellten verfügt XQuery noch über eine Fülle an weiteren Funktionen. Genaueres darüber kann man in der entsprechenden Empfehlung des W3C [BCF+05] sowie in [LS04], Abschnitt 7 nachlesen. Des Weiteren verfügt XQuery auch über erweiterte Ausdrücke wie z.B. Quantoren, die hier jedoch keine Anwendung finden, weshalb dazu nochmals auf die entsprechende Literatur verwiesen sei.

3.5 XSLT

XSLT (eXtensible Stylesheet Language Transformations) ist eine Transformationssprache, die dazu dient, XML-Dokumente nach festgelegten Regeln in andere Dokumente umzuwandeln. Dazu gibt es vom W3C eine Recommendation [Cla99], d.h. es handelt sich bei XSLT um einen de-facto-Standard. Gegenüber XQuery ist XSLT weniger ausdrucksstark und flexibel, aber für unsere Zwecke mehr als ausreichend. Augenfälligster Unterschied zu XQuery ist, dass XSLT-Stylesheets, in welchen die Transformationsanweisungen gespeichert sind, selbst wohlgeformte XML-Dokumente sind. Zu XQuery gibt es zwar mit XQueryX [MM05] einen Ansatz, die Anweisungen in XML zu repräsentieren, dieser ist aber in der derzeitigen Form kaum brauchbar, da selbst einfache Anfragen im Vergleich zu XSLT sehr lang werden und für Menschen nur noch schwer verständlich sind.

3.5.1 Datenmodell und Pfadausdrücke

Das Datenmodell, auf dessen Grundlage XSLT arbeitet, ist die Baumstruktur eines XML-Dokuments (vgl. [Cla99], Abschnitt 3) . Dieses Modell ist bereits bekannt aus der Vorstellung des XQuery-Datenmodells, in dem es aber nur einen Teilaspekt darstellte. Wie zu erwarten ist, funktioniert die Navigation innerhalb der Dokumentstruktur auch wie in XQuery, und zwar mit Hilfe von Pfadausdrücken. Deswegen sollen diese hier nicht nochmals besprochen werden, es sei an dieser Stelle auf den entsprechenden Abschnitt zu XQuery verwiesen (Abschnitt 3.4.2).

Interessant ist in diesem Zusammenhang aber, dass in XQuery Knoten an Variablen gebunden werden müssen, um als Ausgangspunkt für die Navigation zu dienen. Dies ist auch in XSLT möglich, darüber hinaus ist aber implizit immer eine Liste von Knoten

vorhanden, die den aktuellen „Kontext“ für Pfadausdrücke bilden. Es gibt nur wenige Anweisungen in XSLT, die diesen Kontext verändern. Dazu zählen auch die `templates`, auf die wir zunächst eingehen wollen.

3.5.2 Templates

Um ein Dokument zu transformieren, werden von XSLT bestimmte Muster im Ausgangsdokument erkannt und entsprechend definierter Regeln umgewandelt. Dies geschieht in der `template`-Anweisung, die wie folgt aufgebaut ist:

```
<xsl:template match = "Hier wird das Muster beschrieben">
  <!-- Hier stehen die Transformationen-->
</xsl:template>
```

Das Muster ist dabei ein Pfadausdruck, der wie schon bekannt Knoten zurückliefert, die bestimmten Kriterien genügen. Wenn Knoten, welche dem geforderten Muster entsprechen, gefunden wurden, bildet diese Knotenmenge den oben beschriebenen Kontext für die Anweisungen. Diese werden nacheinander für jeden dieser Knoten ausgeführt. Dabei kann es sich einfach um die Ausgabe von XML handeln (die im folgenden Abschnitt noch näher behandelt wird), es können aber z.B. auch weitere `templates` sein, die für die jeweiligen Knoten ausgewertet werden.

Eine übliche Vorgehensweise ist es jedoch, nur ein `template` für das gesamte Dokument zu verwenden, welches als Muster das Wurzelelement des Dokuments übergeben bekommt. Dieses dient dann sozusagen als Einstiegspunkt, und die weitergehende Verarbeitung wird mit anderen Mitteln bewerkstelligt. Da dieses Verfahren auch in der vorgestellen Lösung Anwendung findet, wollen wir an dieser Stelle nicht weiter auf `templates` eingehen. Sie werden jedoch zunächst noch als Rahmen für einige Beispiele dienen. Weiterführende Details zum Thema sind unter [Cla99], Abschnitt 5 zu finden.

3.5.3 Ausgabe von XML

Wie auch XQuery bietet XSLT die Möglichkeit, im Vorfeld festgelegte XML-Fragmente auszugeben. Dazu ein Beispiel:

```
<xsl:template match = "/*">
  <Element attr="Attributinhalt">Elementinhalt</Element>
</xsl:template>
```

In diesem Beispiel wird für jedes Kindelement der Wurzel das entsprechende neue Element eingefügt. Interessanter ist jedoch, die Ausgabe zur Laufzeit zu bestimmen. Dazu dient u.a wieder der aus XQuery bekannte Auswertungskontext, der durch geschweifte Klammern markiert wird. Damit kann der Wert von Attributen (und nur von Attributen) durch einen Ausdruck bestimmt werden. Im folgenden Beispiel ist auch ersichtlich, dass man anders als in XQuery keine gesonderte Funktion braucht, um den Inhalt eines

Knotens zu erhalten. Der Pfadausdruck genügt, wobei in diesem Fall der Punkt für den aktuell betrachteten Knoten (sprich den Kontextknoten) steht:

```
<xsl:template match = "Person">
  <Person nachname="{./Nachname}">Hier stand eine Person</Person>
</xsl:template>
```

Um statt des Inhalts den Bezeichner eines Knotens abzufragen dient aber auch hier, wie in XQuery, die Funktion `name()`

Um den Namen eines Elements durch einen Ausdruck zu bestimmen, benötigt man das `<xsl:element>`-Konstrukt:

```
<xsl:template match = "Person">
  <xsl:element name = "{./Nachname}">
    Hier steht der Inhalt
  </xsl:element>
</xsl:template>
```

Das Attribut `name` ist in diesem Zusammenhang kein „herkömmliches“ Elementattribut, sondern reserviert für die Festlegung des Namens des konstruierten Elements. Wenn man trotzdem ein Attribut `name` im konstruierten Element haben möchte, kann man das `<xsl:attribute>`-Konstrukt verwenden, welches nach dem gleichen Prinzip funktioniert (vgl. [Cla99], Abschnitt 7.1).

Um nun den Inhalt eines Elementes dynamisch festzulegen, ist, anders als in XQuery, der Auswertungskontext nicht von Nutzen, da die Anweisung in geschweiften Klammeren einfach literal ausgegeben würde. An dieser Stelle kommt das `<xsl:value-of>`-Konstrukt ([Cla99], Abschnitt 7.6) zum Einsatz, welches in seinem `select`-Attribut einen Ausdruck zur Auswertung übergeben bekommt:

```
<xsl:template match = "Person">
  <Nachname>
    <xsl:value-of select = "./Nachname"/>
  </Nachname>
</xsl:template>
```

3.5.4 Iteration, konditionale Verarbeitung, Variablen

Nachdem wir die grundlegenden Mechanismen zur Ausgabe von XML kennengelernt haben, wollen wir einige weitere Aspekte einführen, die in der vorgestellten Lösung zum Einsatz kommen.

Eines der wichtigsten Hilfsmittel zur Transformation von Dokumenten ist das Erkennen und Verarbeiten von sich wiederholenden Strukturen. Neben den bereits vorgestellten `templates` dient dazu in XSLT das `<xsl:for-each>`-Konstrukt, welches zugleich auch das zweite „kontextverändernde“ Konstrukt ist (vgl. dazu [Cla99], Abschnitt 8). Es

bekommt in seinem `select`-Attribut einen Pfadausdruck übergeben, und für jeden der zurückgelieferten Knoten werden nacheinander die entsprechenden Anweisungen ausgeführt, wobei der jeweils betrachtete Knoten zum Kontextknoten wird:

```
<xsl:for-each  select = "Hier steht der Pfadausdruck">
  <!-- Hier stehen die Transformationen-->
</xsl:for-each>
```

Die Funktionsweise ist die gleiche wie bei einem `template`, aber `templates` bieten darüberhinaus noch weitere Funktionalität, wie z.B. das Unterscheiden mehrerer Verarbeitungsmodi (vgl. [Cla99], Abschnitt 5.7). Da diese Möglichkeiten in der vorgestellten Lösung jedoch nicht benötigt werden, kommt das `<xsl:for-each>`-Konstrukt zum Einsatz.

Von großer Bedeutung ist auch das konditionale Verarbeiten von Anweisungen. Dazu gibt es in XSLT zwei verschiedene Mechanismen (vgl. [Cla99], Abschnitt 9). Für einfache Abfragen gibt es das `<xsl:if>`-Konstrukt, welches in seinem `test`-Attribut einen Ausdruck übergeben bekommt, welcher zu true oder false evaluiert wird:

```
<xsl:if  test = "Hier steht die Bedingung">
  <!-- Hier stehen die Anweisungen-->
</xsl:if>
```

Eine Alternative, falls die Bedingung nicht erfüllt ist, ist nicht vorgesehen. Für komplexere Abfragen, die mehrere Bedingungen bzw. alternative Anweisungen erhalten, steht das `<xsl:choose>`-Konstrukt zur Verfügung, welches wie folgt aufgebaut ist:

```
<xsl:choose>
  <xsl:when test="Erste Bedingung">
    <!--Anweisungen-->
  </xsl:when>
  <xsl:when test="Zweite Bedingung">
    <!--Anweisungen-->
  </xsl:when>
  <xsl:otherwise>
    <!--Anweisungen-->
  </xsl:otherwise>
</xsl:choose>
```

Die Bedingungen werden nacheinander geprüft. Sobald eine erfüllte Bedingung gefunden wurde, werden die entsprechenden Anweisungen ausgeführt, und die anderen Bedingungen werden dann nicht mehr überprüft. Die Anweisungen, die im `<xsl:otherwise>`-Block stehen werden ausgeführt, wenn keine Bedingung erfüllt wurde. Dieser Block ist optional.

Der letzte Punkt, auf den wir eingehen wollen, ist die Verwendung von Variablen bzw. Parametern, die in [Cla99], Abschnitt 11 behandelt wird. Mit dem `<xsl:variable>`-Konstrukt lassen sich mit entsprechenden Ausdrücken Knoten oder atomare Werte an

Variablen binden, deren Inhalt mit den bereits vorgestellten Mechanismen ausgewertet werden kann. Dazu ist allerdings ein vorangestelltes `$` nötig:

```
<xsl:variable name = "x" select = "{3+4}"/>
<xsl:for-each select = "*">
  <Sieben>
    <xsl:value-of select="{$x}"/>
  </Sieben>
</xsl:for-each>
```

Das `<xsl:param>`-Konstrukt funktioniert auf die gleiche Weise, aber anders als bei normalen Variablen ist es möglich, von außerhalb des Stylesheets Werte an einen Parameter zu übergeben. Wie das funktioniert, werden wir später bei der Vorstellung der XSLT-basierten Lösung sehen.

4 Umsetzung der Lösung

Nachdem wir jetzt gesehen haben, wie das Ergebnis der Datenbankrepräsentation aussehen soll, und mit welchen Hilsmitteln diese angestrebt wird, wollen wir uns im nun folgenden Hauptteil der Arbeit konkret damit befassen, wie wir die Datenbank in das durch die Relational.OWL-Ontologie vorgegebene Format bringen können. Dazu betrachten wir acht von uns erstellte Funktionen bzw. Prozeduren, die in folgender Weise zusammenwirken:

- Die Funktionen `METADATA_AS_RAW_XML()` und `DATA_AS_RAW_XML()` dienen dazu, die in der Oracle 10g gespeicherten Daten bzw. Metadaten mit Hilfe datenbankeigener Funktionalität in eine „rohe“ XML-Repräsentation zu überführen, an der zunächst nur geringfügige Modifikationen vorgenommen werden. Die anderen Funktionen verwenden diese Repräsentation als Grundlage für die Transformation in die gewünschte Form mit Hilfe von XQuery bzw. XSLT.
- Die Funktionen `METADATA_AS_OWL()` und `DATA_AS_OWL()` wandeln die von den entsprechenden Funktionen gelieferten „Rohdokumente“ unter Verwendung von XQuery in das gewünschte, der Relationl.OWL-Ontologie entsprechende Format um.
- Die Funktionen `METADATA_AS_OWL_WITH_XSLT()` und `DATA_AS_OWL_WITH_XSLT()` wandeln die Rohdaten ebenfalls in das gewünschte Format um, zum Einsatz kommt hier statt XQuery jedoch XSLT.
- Die Prozeduren `EXTRACT_METADATA_INTO_FILE()` und `EXTRACT_DATA_INTO_FILE()` haben die Aufgabe, die Ergebnisse der Daten- bzw. Schemarepräsentation in entsprechende Dateien zu schreiben und bilden damit sozusagen die „Schnittstelle“ zum Benutzer des Datenbanksystems. Die Extraktion erfolgt wahlweise mit XQuery oder XSLT, wobei dann in den Prozeduren jeweils die entsprechenden Funktionen aufgerufen werden.

Die zugehörigen Quellcodes bzw. SQL-DDL-Befehle finden sich in den Listings in Anhang A. Unter Verwendung dieser Befehle lassen sich die Funktionen bzw. Prozeduren einfach in das Datenbanksystem integrieren, z.B. über die SQL-Kommandozeile. Zu beachten ist dabei, dass hierzu `CREATE ANY PROCEDURE`-Rechte erforderlich sind.

Details und Beispiele zum Aufruf der einzelnen Funktionen und Prozeduren finden sich in den entsprechenden Abschnitten. Erforderlich zum Aufruf sind jedoch in jedem Fall `EXECUTE ANY PROCEDURE`-Rechte.

4.1 Erstellung der „Rohdokumente“

4.1.1 Die Funktion METADATA_AS_RAW_XML()

Die Funktion `METADATA_AS_RAW_XML()` (Listing 1) dient dazu, die Metadaten der Datenbank, sprich die Datenbankbeschreibung, in ein XML-Dokument zu extrahieren. Dazu sind im Wesentlichen zwei Fälle zu unterscheiden: Die Ausgabe einer einzelnen Tabelle und die Ausgabe der gesamten Datenbank. Gesteuert wird dies durch das übergebene Argument. Bei übergebenem Tabellennamen wird dieser verwendet, falls kein Name angegeben wird, wird der Defaultwert „RETURN_ALL_TABLES“ benutzt. Dieser kann auch manuell übergeben werden.

Ausgangspunkt der Extraktion ist das Package `DBMS_METADATA`, welches in PL/SQL standardmäßig zur Verfügung steht. Zunächst wird die Methode `DBMS_METADATA.OPEN('TABLE')` aufgerufen. Damit wird signalisiert, dass wir Informationen über die in der Datenbank gespeicherten Tabellen abrufen möchten. Neben 'TABLE' gibt es noch eine große Zahl weiterer möglicher Parameter für verschiedenste Arten von Informationen, Details dazu findet man u.a. in der offiziellen Dokumentation von Oracle [Rap03]. Der Rückgabewert der Methode ist ein so genannter `handle`, der vergleichbar ist mit dem in Abschnitt 3.3.2 vorgestellten Cursor-Konzept von PL/SQL. Bevor wir die Informationen abrufen können, müssen wir jedoch die Methode `DBMS_METADATA.ADD_TRANSFORM()` mit dem zuvor generierten `handle` und dem Argument 'MODIFY' aufrufen. Damit wird bewirkt, dass die Metadaten in Form eines XML-Dokumentes ausgegeben werden. Die Alternative dazu wäre das Argument 'DDL', mit dem die SQL-DDL-Befehle ausgegeben werden, mit denen die Datenbank erstellt wurde.

Die Entscheidung, ob wir alle oder nur eine Tabelle exportieren, wird in der folgenden `if`-Abfrage (Listing 1, Zeile 24) getroffen. Wenn alle Tabellen ausgegeben werden sollen, brauchen wir nichts weiter zu unternehmen, da dies die Standardeinstellung ist. Falls nur eine Tabelle gewünscht ist, wird die Methode `DBMS_METADATA.SET_FILTER()` aufgerufen, mit dem betreffenden `handle`, dem Argument 'NAME' als zweitem und dem Namen der entsprechenden Tabelle als drittem Argument. Wie der Methodenname schon impliziert, werden damit die Tabellen ausgefiltert, die nicht den entsprechenden Namen tragen, womit nur die gewünschte Tabelle übrig bleibt.

Als nächstes wird in einer Schleife wiederholt die Methode `DBMS_METADATA.FETCH_CLOB()` aufgerufen, die nach und nach die Metadaten der einzelnen Tabellen liefert. Die CLOBs werden in einer Hilfsvariablen gespeichert

und jeweils in einer weiteren CLOB-Variablen mit Hilfe des entsprechenden Operators „||“ konkatenatiert.

Als letzter Arbeitsschritt wird, ebenfalls durch einfache Konkatenation, auf oberster Ebene ein `DATABASE`-Tag eingeführt, um ein wohlgeformtes XML-Dokument zu erhalten. Danach wird der Konstruktor `XMLType()` mit dem „fertigen“ CLOB aufgerufen, um ein entsprechendes Dokument zu generieren, welches von der Funktion zurückgegeben wird.

Da (anders als bei der Extraktion des Datenbankausprägung, die im folgenden Abschnitt behandelt wird) das entstandene XML-Dokument eine komplexe Struktur hat, wollen wir nun noch kurz auf die für uns wichtigen Teile des Dokuments eingehen. Dies ist für das Verständnis der Lösung von großer Bedeutung. Dazu betrachten wir zunächst eine schematische Darstellung des Dokuments (Abbildung 2), wobei zu beachten ist, dass der Baum in Wahrheit noch wesentlich mehr Informationen enthält, die für das Problem jedoch nicht von Interesse sind. Daher wurden sie der Übersichtlichkeit halber weggelassen.

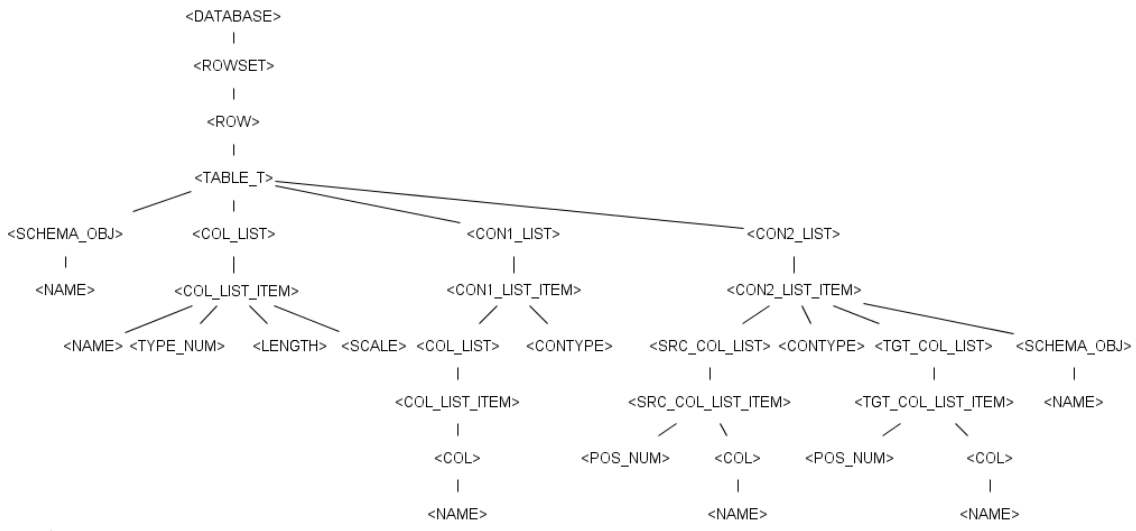

Abbildung 2: Der Aufbau des Metadaten-Dokuments.

Unterhalb des von uns eingeführten `DATABASE`-Elements befindet sich für jede extrahierte Tabelle ein `ROWSET`-Element, welches genau ein `ROW`-Element enthält. Dieses enthält wiederum genau ein `TABLE_T`-Element, und ausgehend davon erreicht man alle notwendigen Informationen über die jeweilige Tabelle.

Unter `SCHEMA_OBJ/NAME` findet man den Namen der Tabelle. Unter `COL_LIST` findet man die Spalten, für die jeweils ein `COL_LIST_ITEM`-Element existiert. Dort sind neben dem Namen der Spalte weitere Informationen gespeichert, auf die wir später noch genauer eingehen werden.

Unter CON1_LIST findet sich eine Liste bestimmter Integritätsbedingungen, wobei für uns nur der Primärschlüssel interessant ist. Dieser befindet sich in einem CON1_LIST_ITEM, dessen Wert für das Element CONTYPE 2 ist. Unter COL_LIST findet man dort die entsprechenden Spalten, die zum Primärschlüssel gehören, wobei auch hier jede in einem seperaten COL_LIST_ITEM-Element gespeichert ist, in dem man über COL/NAME auf den Namen zugreifen kann.

Unter CON2_LIST befindet sich eine Liste weiterer Integritätsbedingungen, unter anderem auch Fremdschlüssel. Für jeden in der Tabelle definierten Fremdschlüssel existiert ein eigenes CON2_LIST_ITEM, bei welchem das Element CONTYPE den Wert 4 hat. Unter SCHEMA_OBJ/NAME findet man hier jeweils die Tabelle, auf welche der Fremdschlüssel verweist, also welche bei der Erstellung in SQL-DDL nach dem Schlüsselwort `references` angegeben wurde. Das Element SRC_COL_LIST enthält die Spalten, aus denen der Fremdschlüssel besteht, während TGT_COL_LIST die entsprechenden Spalten der referenzierten Tabelle enthält. Die Zuordnung, welche „Quellspalte“ zu welcher „Zielspalte“ gehört (bei Fremdschlüsseln mit mehreren Spalten) erfolgt über das Element POS_NUM. Spalten mit der gleichen Nummer gehören zusammen.

4.1.2 Die Funktion DATA_AS_RAW_XML()

Die Funktion DATA_AS_RAW_XML() (Listing 2) hat die Aufgabe, die tatsächliche Ausprägung einer Datenbank in Form eines Dokuments vom Typ XMLType zu liefern. Genau wie bei der im vorherigen Abschnitt beschriebenen Funktion wird auch hier nach dem übergebenen Parameter entschieden, ob die gesamte Datenbank oder nur eine einzelne Tabelle verarbeitet wird. Der zugrundeliegende Mechanismus ist jedoch ein anderer.

In beiden Fällen wird dabei auf die Funktion DBMS_XMLQuery.getXML() zurückgegriffen, die das Ergebnis einer SQL-Anfrage als CLOB in XML-Form zurückliefert und standardmäßig in PL/SQL aufgerufen werden kann. Das sieht dann konkret für die in Abschnitt 3.2.1 beschriebene Relation LAND so aus:

```
<?xml version="1.0" ?>
 <ROWSET>
 <ROW num="1">
  <ID>1</ID>
  <NAME>Deutschland</NAME>
  </ROW>
 <ROW num="2">
  <ID>2</ID>
  <NAME>Spanien</NAME>
 </ROW>
</ROWSET>
```

Betrachten wir zunächst den einfacheren Fall, der im `else`-Zweig der `if`-Anweisung (Listing 2, Zeile 22) behandelt wird, nämlich dass nur eine einzelne Tabelle bzw. Relation ausgegeben wird. Dann wird der Name dieser Tabelle, der in der Variablen `table_name` übergeben wurde, per Konkatenation an den String „`select * from`“ angehängt, und

dieser String wird der Funktion `DBMS_XMLQuery.getXML()` zur Auswertung übergeben. Da der Name der Tabelle aus dem automatisch erstellten Dokument nicht ersichtlich ist, wird in einem zusätzlichen Arbeitsschritt (ebenfalls mit einfacher Konkatenation) ein Element `TABLE` auf oberster Dokumentebene eingeführt, welches als einziges Attribut den Namen der Tabelle erhält. Damit enthält das Dokument alle zur weiteren Verarbeitung notwendigen Informationen.

Etwas komplexer ist hingegen die Ausgabe der gesamten Datenbank, die im `then`-Zweig behandelt wird. Dazu wird (wie im Kapitel über PL/SQL beschrieben) ein vorher definierter Cursor auf die Anfrage „`select table_name from tabs`“ geöffnet. Die Tabelle `tabs` ist eine systemeigene Tabelle, die Informationen über alle vorhanden Tabellen eines Benutzers enthält. In der Schleife werden dann die Namen dieser Tabellen genau wie oben beschrieben an die Funktion `DBMS_XMLQuery.getXML()` übergeben, welche jeweils ein entsprechendes Dokument generiert. Diese Dokumente werden dann ebenfalls mit einem `TABLE`-Tag samt Namensattribut versehen und per Konkatenation zusammengefügt, so dass man am Ende der Schleife die gesamte Datenbank in einem einzigen CLOB vorliegen hat.

Im abschließenden Arbeitsschritt, der beiden Fällen gemein ist, werden zunächst die Verarbeitungsanweisungen (sprich die Versionsangaben) gelöscht, die bei der Extraktion mehrerer Tabellen redundant vorhanden sind. Dazu wird die Funktion `replace()` benutzt, die in der im ersten Argument übergebenen Zeichenkette das im zweiten Argument übergebene Muster sucht und durch die Zeichenkette im dritten Argument ersetzt. Wenn, wie in diesem Fall, kein drittes Argument übergeben wird, wird das entsprechende Muster einfach gelöscht. Anschließend wird (wiederum durch Konkatenation) ein umgebendes `DATABASE`-Tag eingeführt, welches dazu führt, dass die fertigen Dokumente eine einheitliche und wohlgeformte Struktur haben, egal ob sie eine oder mehrere Tabellen bzw. Relationen enthalten. Zum Schluss wird der Konstruktor `XMLType()` mit dem fertigen CLOB aufgerufen, und das Dokument wird zurückgegeben.

4.2 Umwandlung der Dokumente in Relational.OWL mit XQuery

Nachdem wir sowohl die Inhalte als auch die Metadaten der Datenbank in unformatiertem XML vorliegen haben, können die Dokumente in die gewünschte Form gebracht werden. Dazu werden zunächst die Besonderheiten der PL/SQL-Funktion `XMLQuery()` betrachtet, die uns bei der im Folgenden betrachteten Transformation mit XQuery als Werkzeug dienen wird. Details dazu findet man u.a. in [Dra05].

Der grundsätzliche Aufbau des Funktionsaufrufs sieht wie folgt aus:

```
SELECT
XMLQuery(
'
-- Hier steht der XQuery-Ausdruck
'
PASSING Eingabedokument AS "Variablenbezeichner"
RETURNING CONTENT) INTO Ausgabedokument FROM DUAL;
```

Auffällig ist zunächst, dass diese Funktion nicht wie bisher bekannte Funktionen aufgerufen werden kann. Man muss das bereits vorgestellte `SELECT ... INTO`-Konstrukt anwenden, wobei die Variable für das Ausgabedokument in jedem Fall vom Typ `XMLType` sein muss. Die Tabelle `DUAL`, an die die Anfrage gerichtet wird, ist keine Datenbanktabelle im eigentlichen Sinn, sondern ein „Dummy-Table“, welcher von PL/SQL für diese Art von Funktionsaufrufen bereitgestellt wird. Dies trägt der Tatsache Rechnung, dass ein Weglassen der `from`-Klausel in SQL nicht zulässig ist (vgl. dazu [FP03], Kap. 20).

Das Schlüsselwort `PASSING` erlaubt es, PL/SQL-Variablen, welche ebenfalls vom Typ `XMLType` sein müssen, unter dem nach `AS` angegebenen Bezeichner in der Anfrage zu verwenden, womit diese Dokumente wie im Abschnitt über XQuery beschrieben angefragt werden können.

Die Klausel `RETURNING CONTENT` ist zwingend erforderlich, eine Alternative hierzu ist derzeit nicht vorgesehen.

Erwähnt werden sollte an dieser Stelle, dass diese Funktion in Version 10.2.0.1.0 des verwendeten Datenbanksystems nur Anfragen akzeptiert, die höchstens 4000 Zeichen lang sind. Erst ein Update auf Version 10.2.0.2.0 ermöglicht die Verwendung von Anfragen, die bis zu 16000 Zeichen lang sind. Die beschränkte Anfragelänge führte zu einigen Problemen bei der Umsetzung der Lösung und sollte bei der Verwendung der Funktion stets beachtet werden.

Nachdem wir nun die Charakteristika der Funktion `XMLQuery()` kennen, können wir uns den Funktionen widmen, die davon Gebrauch machen.

4.2.1 Die Funktion METADATA_AS_OWL()

Die Funktion `METADATA_AS_OWL()` (Listing 3) erwartet zwei übergebene Argumente: Den Namen der Datenbank und den Namen der zu verarbeitenden Tabelle, wobei dieser direkt an die Funktion `METADATA_AS_RAW_XML()` übergeben und wie in Abschnitt 4.1.1 beschrieben verarbeitet wird. Die Angabe des Datenbanknamens ist notwendig, da das Abfragen des Datenbanknamens ohne Administratorrechte nicht möglich ist. Dieser Name wird in ein rudimentäres XML-Dokument eingebettet, welches z.B. wie folgt aussehen kann:

```
<dbname value="MEINEDB" />
```

Der Hintergrund ist, dass nur `XMLType`-Dokumente an die XQuery-Anfrage übergeben werden können. Daher muss der String für den Datenbanknamen auf diesem Wege „verpackt“ werden, um ihn übergeben zu können.

Das „Namensdokument“ und das von der Funktion `METADATA_AS_RAW_XML()` zurückgegebene „Rohdokument“ werden an eine XQuery-Abfrage übergeben, die wir nun näher betrachten wollen.

Zunächst wird das öffnende <rdf:RDF>-Tag mit den erforderlichen Namensräumen ausgegeben. Danach folgt die Definition der ersten Klasse der Schemaontologie, nämlich der für die gesamte Datenbank. Als ID wird hier der entspechende Wert des von uns konstruierten Namensdokuments eingetragen.

Nach der Typangabe (Database) folgt eine for-Schleife über alle Tabellennamen der Datenbank. Für jeden Namen wird die entsprechenden hasTable-Eigenschaft zur Beschreibung der Datenbank hinzugefügt. Danach wird das entsprechende Element geschlossen, und dieser Teil des Dokuments ist fertig. Für die in Abschnitt 3.2.1 beschriebene Beispieldatenbank sieht das folgendermaßen aus:

```
<...>
<owl:Class rdf:ID="MEINEDB">
 <rdf:type rdf:resource=
 "http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#Database" />
 <dbs:hasTable rdf:resource="#PERSON" />
 <dbs:hasTable rdf:resource="#LAND" />
</owl:Class>
</...>
```

Dann folgt die Beschreibung der einzelnen Tabellen. Dazu wird in einer for-Schleife über alle Tabellen jeweils ein entsprechendes <owl:Class>-Element angelegt, mit dem Namen der jeweiligen Tabelle als ID. Nach der Typangabe Table folgt in diesem Element eine weitere for-Schleife über alle Spaltennamen dieser Tabelle. Für jede Spalte wird ein hasColumn-Element angelegt, dessen Name sich aus dem Namen der Tabelle und dem der Spalte zusammensetzt.

Der nächste Schritt ist die Verarbeitung des Primärschlüssels. Dazu wird zunächst geprüft, ob in der entsprechenden Liste der Integritätsbedingungen überhaupt ein Primärschlüssel vorhanden ist, also ein Element dessen CONTYPE den Wert 2 hat. Ist dies der Fall, werden die entsprechenden <dbs:isIdentifiedBy>- und <dbs:PrimaryKey>-Tags ausgegeben. In einer weiteren for-Schleife über die Namen der Primärschlüsselspalten werden dann (wie oben) die hasColumn-Elemente erstellt, die zum Primärschlüssel gehören. Nach dem Schließen der offenen Tags ist auch dieser Teil des Dokuments fertig. Dazu wieder ein Ausschnitt aus der Beispieldatenbank:

```
<...>
<owl:Class rdf:ID="LAND">
 <rdf:type rdf:resource=
  "http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#Table" />
 <dbs:hasColumn rdf:resource="#LAND.ID" />
 <dbs:hasColumn rdf:resource="#LAND.NAME" />
 <dbs:isIdentifiedBy>
  <dbs:PrimaryKey>
   <dbs:hasColumn rdf:resource="#LAND.ID" />
  </dbs:PrimaryKey>
 </dbs:isIdentifiedBy>
</owl:Class>
</...>
```

Der letzte Teil des Dokuments beinhaltet die Beschreibung der Spalten, jeweils in einem <owl:DatatypeProperty>-Element, dessen ID sich aus Tabellennamen und Spaltennamen zusammensetzt. Dazu werden in zwei verschachtelten for-Schleifen alle Tabellen und deren Spalten durchlaufen. Der Typangabe Column folgt die Festlegung der domain, bei der es sich in diesem Fall jeweils um die zur Spalte gehörige Tabelle handelt.

Die folgende, mehrfach verzweigende if-Abfrage dient zur Festlegung des Wertebereichs (range) der Spalte. Welchen Wertebereich eine Spalte hat, lässt sich in folgender Weise aus dem vorliegenden Rohdokument ersehen:

- Bei TYPE_NUM 1 und TYPE_NUM 96 handelt es sich um die SQL-Datentypen VARCHAR bzw. CHAR. Diesen wird im fertigen Dokument der XML Schema-Datentyp string zugeordnet.
- TYPE_NUM 12 steht für den Datentyp date, der Datumsangaben speichert.
- TYPE_NUM 180 steht für den Datentyp timestamp, der gemischte Zeit/Datumsangaben speichert. Diesem wird der entsprechende Datentyp dateTime zugeordnet.
- Numerische Datentypen haben prinzipiell den TYPE_NUM-Wert 2.
- Die SQL-Datentypen numeric und decimal werden nicht weiter unterschieden. Sie haben im vorliegenden Dokument sowohl ein SCALE- als auch ein PRECISION_NUM-Element, und ihnen wird der XML Schema-Datentyp decimal zugeordnet.
- Der Datentyp integer hat ein SCALE-, aber kein PRECISION_NUM-Element. Der zugeordnete Datentyp ist auch integer.
- Der Datentyp float hat ein PRECISION_NUM-, aber kein SCALE-Element. Der zugeordnete Datentyp ist float.

Nachdem der Datentyp zugeordnet wurde, wird in einer weiteren if-Abfrage die Ausgabe des Elements <dbs:length> geregelt. Diese ist bei Strings, Dezimalzahlen und Float-Zahlen vorgesehen. Bei Strings findet man die entsprechende Angabe im LENGTH-Element, bei den numerischen Werten im PRECISION_NUM-Element.

In der letzten if-Abfrage wird nun noch der scale-Wert für Dezimalzahlen ausgegeben, der im gleichnamigen Element des Rohdokuments gespeichert ist.

Zum Schluss wird geprüft, ob die betrachtete Spalte Teil eines Fremdschlüssels ist. Dazu wird eine folgendermaßen geschachtelte for-Schleife benutzt:

- Die erste Schleife läuft über alle Fremdschlüssel.
- Die zweite Schleife läuft über alle Spalten, die dieser Fremdschlüssel beinhaltet.
- Die dritte Schleife läuft über alle Spalten, auf die dieser Fremdschlüssel verweist.

In der zugehörigen `where`-Klausel wird geprüft, ob der Name der aktuellen Spalte mit dem Namen einer Fremdschlüsselspalte übereinstimmt. Falls dem so ist, muss man darüber hinaus wissen, auf welche Spalte verwiesen wird. Dies lässt sich über den abgefragten `POS_NUM`-Wert feststellen, der bei zusammengehörenden Spalten beider Fremdschlüssellisten identisch ist. Falls eine Fremdschlüsselabhängigkeit vorliegt, wird das entsprechende `<dbs:references>`-Element ausgegeben. Der Wert des Zielattributs setzt sich zusammen aus dem Namen der Tabelle, auf die verwiesen wird, sowie dem Namen der entsprechenden Spalte. Für eine Spalte aus der Beispieldatenbank sieht das Ergebnis so aus:

```
<...>
<owl:DatatypeProperty rdf:ID="PERSON.LANDID">
 <rdf:type rdf:resource=
  "http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#Column" />
 <rdfs:domain rdf:resource="#PERSON" />
 <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#integer" />
 <dbs:references rdf:resource="#LAND.ID" />
</owl:DatatypeProperty>
</...>
```

Nach dem Schließen der noch offenen Tags ist die Anfrage vollständig. Dem in der Variablen `return_doc` gespeicherten Dokument werden nun noch die Prozessierungsinformationen mit XML-Version und verwendetem Zeichensatz durch Konkatenation hinzugefügt. Die Angabe der Versionsnummer „1.0“ ist dabei obligatorisch, es gibt (noch) keine andere XML-Version. Der verwendete Zeichensatz sollte idealerweise zur Laufzeit aus der Datenbank ausgelesen werden. Da in der Oracle 10g der Zeichensatz jedoch kodiert gespeichert wird und ein „Übersetzen“ der weit über 100 Codes (vgl. [Lan04], Anhang A) das Programm enorm aufgebläht hätte, ohne echten Mehrwert zu bringen, wird der Einfachheit halber ein konstanter Wert für den in Westeuropa üblichen Zeichensatz verwendet.

Das Einfügen durch Konkatenation ist notwendig, da XQuery, genau wie XSLT, das Einfügen derartiger Informationen in der Anfrage selbst nicht zulässt (vgl. [BCF+05], Abschnitt 3.7.3.5). Wenn das fertige Dokument diese Informationen enthalten soll, muss man also diesen Umweg gehen. Als Ausgangspunkt für diese Modifikation dient die Repräsentation der XML-Dokuments als CLOB, die von der Funktion `getClobVal()` zurückgeliefert wird. Anschließend wird dieses modifizierte CLOB mit dem entsprechenden Konstruktor wieder in `XMLType` umgewandelt, bevor das Dokument zurückgegeben wird.

4.2.2 Die Funktion DATA_AS_OWL()

Diese Funktion (Listing 4) erwartet als Argumente den Speicherort der zugehörigen Schemaontologie und den Namen der zu verarbeitenden Tabelle, der nach dem aus dem letzten Abschnitt bekannten Muster direkt an die Funktion zur Extraktion der Rohdaten weitergegeben wird. Das von dieser Funktion zurückgegebene „Rohdokument“ wird an eine XQuery-Abfrage übergeben, die wir nun näher betrachten wollen.

Zunächst wird das öffnende `<rdf:RDF>`-Tag ausgegeben, mit den erforderlichen Namensräumen als Attributen. An der Stelle, an der im fertigen Dokument der

Speicherort der Schemaontologie stehen soll, wird jedoch zunächst ein Platzhalter (`SCHEMA_PLACEHOLDER`) eingefügt. Man könnte den Speicherort zwar alternativ wie im vorigen Abschnitt beschrieben als `XMLType` „verpackt“ übergeben, dies würde das grundlegende Problem jedoch nicht lösen. Der XQuery-Prozessor lässt nämlich nicht zu, dass Namensraumattribute zur Laufzeit bestimmt werden, hier muss bereits zur Kompilierzeit ein konstanter Wert eingetragen werden. Also wird das Problem mit dem Platzhalter umgangen, der dann in einem letzten Arbeitsschritt durch den konkreten Wert ersetzt werden wird.

Als nächstes wird ein verschachtelter FLWOR-Ausdruck benutzt: In der aüßeren `for`-Schleife werden alle Tabellen durchlaufen (je ein `TABLE`-Element steht für eine Tabelle), in der inneren werden alle `ROW`-Elemente der jeweiligen Tabelle durchlaufen, die jeweils eine Zeile bzw. ein Tupel repräsentieren. Mit der `let`-Klausel wird der Name der Tabelle, der wie in Abschnitt 4.1.2 beschrieben im entsprechenden Attribut des `TABLE`-Elements steht, an die Variable `tab_name` gebunden. Da es sich um einen einzelnen Wert handelt, könnte man hier auch eine `for`-Klausel verwenden.

In der Schleife wird für jede Zeile ein Element angelegt, welches als Bezeichner den Bezeichner der Tabelle (welchen wir gerade in der Variablen `tab_name` gespeichert hatten) mit vorangestelltem Namensraum erhält. Der Inhalt dieses Elements wird in einer weiteren Schleife festgelegt, die über alle Spalten der Zeile läuft. Für jede Spalte wird ein neues Element konstruiert, dessen Name zusammengesetzt ist aus dem Namensraum, dem Tabellennamen sowie dem Namen der Spalte selbst. Der Inhalt jedes dieser Elemente besteht aus dem Wert, der in der jeweiligen Spalte gespeichert ist und mit der Funktion `data()` ausgelesen wird. Ein Ausschnitt des Ergebnisses für die Beispieldatenbank:

```
<...>
<dbinst:LAND>
 <dbinst:LAND.ID>1</dbinst:LAND.ID>
 <dbinst:LAND.NAME>Deutschland</dbinst:LAND.NAME>
</dbinst:LAND>
<dbinst:LAND>
 <dbinst:LAND.ID>2</dbinst:LAND.ID>
 <dbinst:LAND.NAME>Spanien</dbinst:LAND.NAME>
</dbinst:LAND>
</...>
```

Nach dem Durchlaufen dieser Schleifenkonstruktion wird noch der zu Beginn geöffnete Tag geschlossen, was die Anfrage beendet. Im letzten Verarbeitungsschritt wird mit der Funktion `replace()` der Platzhalter für den Speicherort der Schemaontologie durch den korrekten Wert ersetzt, und die Informationen über Version und Zeichensatz werden eingefügt. Das Dokument ist damit fertig.

4.3 Umwandlung der Dokumente in Relational.OWL mit XSLT

Die zweite Möglichkeit, die vorliegenden Rohdaten zu transformieren, ist die Verwendung eines XSLT-Stylesheets. Dazu ist lediglich der Aufruf der Funktion

`XMLType.transform()` erforderlich, welche als Argument das entsprechende Stylesheet im Format `XMLType` erwartet. Des Weiteren ist es möglich, in diesem Stylesheet deklarierten Parametern Werte zu übergeben. Dies geschieht durch einen String als zweitem Argument, der die Zuweisungen in der Form „parametername=wert“ enthält. Im Folgenden wollen wir uns den Funktionen widmen, die von `XMLType.transform()` Gebrauch machen.

4.3.1 Die Funktion METADATA_AS_OWL_WITH_XSLT()

Wie die entsprechende Variante mit XQuery erwartet diese PL/SQL-Funktion (Listing 5) den Datenbanknamen und die zu verarbeitende Tabelle als Argument. Sie beginnt ebenfalls mit der Extraktion der Rohdaten. Danach wird ein `XMLType`-Objekt konstruiert, in dessen Konstruktor wir unser Stylesheet übergeben.

Dieses ist nun wie folgt aufgebaut: Nach dem `<xsl:stylesheet>`-Tag folgt die Deklaration des Parameters `DBNAME`, an welchen der Datenbankname übergeben werden wird. Danach folgt das erste (und einzige) `template`, welches die Verarbeitung des gesamten Dokuments vornimmt. Als `match`-Paramter dient daher das `DATABASE`-Element, von dem wir wissen, dass es das Wurzelelement ist. Alle weiteren Anweisungen können im weiteren Verlauf darauf Bezug nehmen.

Der folgende Teil, die Beschreibung der Datenbank-Klasse, ist der XQuery-Lösung sehr ähnlich, wobei hier eine `for-each`-Schleife zum Einsatz kommt, um die `hasTable`-Elemente für jede Tabelle zu erzeugen. Auch das anschließende Anlegen der entsprechenden Klassen für jede einzelne Tabelle ist weitgehend aus der XQuery-Variante bekannt. Auffälligste Unterschiede sind der Verzicht auf Bindung von Pfaden an Variablen, was zu längeren Pfadausdrücken führt, sowie die Tatsache, dass der Teil zur Behandlung des Primärschlüssels einfach in ein `<xsl:if>`-Element mit der entsprechenden Bedingung geschrieben wird, was leichter zu lesen ist als das entsprechende XQuery-Konstrukt mit leerem `else`-Teil. Trotzdem sind die Verfahren natürlich von der Semantik identisch.

Beim letzten Teil des Dokuments, der Beschreibung der einzelnen Spalten, besteht die erste nennenswerte Auffälligkeit in der Verwendung des in Abschnitt 3.5.4 vorgestellten `<xsl:choose>`-Konstrukts zur Bestimmung des Datentyps und des korrekten `length`-Wertes, wodurch auch hier eine bessere Lesbarkeit und Nachvollziehbarkeit der Abfrage ermöglicht wird.

Bei der Bestimmung der Fremdschlüsselabhängigkeiten hingegen stößt XSLT an seine Grenzen, was die Übersichtlichkeit betrifft. Die Adressierung der notwendigen Knoten in der vorliegenden Dreifach-Schleife (die der aus der XQuery-Lösung entspricht) wäre zwar auch hier über die jeweils gültigen Kontextknoten möglich. Dies wäre aber nur noch schwer nachzuvollziehen. Daher werden die aktuell betrachtete Spalte sowie Quell- und Zielspalte des Fremdschlüssels hier an Variablen gebunden, um die Navigation zu erleichtern, womit die Lösung wiederum fast identisch zu der in XQuery ist. Die Feststellung, ob bzw. welche Abhängigkeit vorliegt, wird dann mit einer `if`-Abfrage getätigt, die semantisch der `where`-Klausel aus der XQuery-Lösung entspricht.

Nach dieser Schleifenkonstruktion werden nur noch die offenen Tags geschlossen, und das Stylesheet ist komplett. Abschließend wird dieses mit einem Aufruf von

`transform()` auf das Rohdokument angewendet, wobei der Datenbankname wie oben beschrieben an den Parameter `DBNAME` übergeben wird. Des Weiteren werden die Prozessierungsinformationen hinzugefügt, womit das Dokument fertig ist und zurückgegeben werden kann.

4.3.2 Die Funktion DATA_AS_OWL_WITH_XSLT()

Diese Funktion (Listing 6) erwartet die gleichen Argumente wie ihr XQuery-Pendant, nämlich den Speicherort der Schemaontologie und den Namen der zu verarbeitenden Tabelle. In gleicher Weise werden auch zunächst die erforderlichen Rohdaten extrahiert, die transformiert werden sollen.

Die XSLT-Stylesheet beginnt dann mit dem `<xsl:stylesheet>`-Tag und dem `template`, welches das `DATABASE`-Element als Einstiegspunkt liefert. Den Beginn des eigentlichen Dokuments markiert das öffnende `<rdf:RDF>`-Tag mitsamt Namensräumen. Wie bei XQuery ist auch hier zu beachten, dass anstelle des Speicherorts der Schemaontologie zunächst ein Platzhalter angegeben wird, da auch XSLT für die Definition von Namensräumen nur konstante Werte zulässt. Danach folgen zwei verschachtelte `for-each`-Schleifen, eine über alle Tabellen und eine über alle Zeilen der jeweiligen Tabelle. Zu beachten sind hier die jeweiligen Pfadausdrücke: Sie sind nicht abhängig von Variablen, sondern nehmen Bezug auf den jeweils aktuellen Kontextknoten. Dies gilt im Folgenden auch für die Konstruktion der Elemente, die hierbei verwendeten Pfadausdrücke kommen ebenfalls ohne Variablen aus.

Für jede Zeile wird ein Element mit dem Namensraumpräfix und dem zugehörigen Tabellennamen angelegt, und zusätzlich wird auch noch ein Attribut `namespace` angelegt, mit dem Ort auf den das Präfix verweist. Das ist zunächst der bereits angesprochene Platzhalter. Das Attribut `namespace` ist, ähnlich wie `name`, ein reserviertes Attribut von `<xsl:element>` und ist in diesem Falle notwendig, da ansonsten im fertigen Dokument das Namensraumpräfix des Elements unterdrückt werden würde.

Der Inhalt des Elements wird nun wiederum in einer `for-each`-Schleife über alle Spalten der Zeile festgelegt. Für jede Spalte wird ein neues Element angelegt (auch wieder mit erwähntem `namespace`-Attribut), der Inhalt des Elements wird mit einem Aufruf von `<xsl:value-of>` für den aktuellen Knoten bestimmt. Nach dem Schließen aller offenen Tags ist das Stylesheet komplett. Nach der Anwendung desselben auf die Rohdaten und den bereits bekannten abschließenden Modifikationen kann das Dokument zurückgegeben werden.

4.4 Ausgabe der fertigen Dokumente

Nachdem wir die Dokumente im gewünschten Format erstellt und als `XMLType` vorliegen haben, bleibt noch zu klären, wie wir diese Dokumente weiterverarbeiten bzw. für die „Außenwelt“ verfügbar machen können. Dazu stellen wir zwei Prozeduren vor, die die Dokumente in Dateien schreiben und darüberhinaus eine simple Fehlerbehandlung auf oberster Ebene durchführen. Sie bilden sozusagen die „Schnittstelle“ für den Benutzer des Datenbanksystems. Da sich diese beiden Prozeduren

(`EXTRACT_METADATA_INTO_FILE()`, Listing 7 und `EXTRACT_DATA_INTO_FILE()`, Listing 8) sehr ähnlich sind, werden sie in dieser Besprechung gemeinsam betrachtet.

Beide Prozeduren erwarten fünf Argumente: Den Datenbanknamen (bei der Metadaten-Extraktion) bzw. den Speicherort der Schemaontologie (bei der Datenextraktion), ein Verzeichnis, einen Dateinamen, einen booleschen Wert (der steuert, ob XQuery oder XSLT benutzt wird) sowie den Namen der zu verarbeitenden Tabelle, welcher wie bereits aus den vorgestellten Funktionen bekannt optional ist. Aufrufe dieser Prozeduren aus der SQL-Kommandozeile mit Hilfe des Befehls `exec` können z.B. wie folgt aussehen:

```
exec EXTRACT_METADATA_INTO_FILE('MEINEDB','DIR','schema.owl',false);

exec EXTRACT_DATA_INTO_FILE('c:\schema.owl','DIR','data.owl',true,'LAND');
```

Dabei ist jedoch zu beachten, dass das Verzeichnis dem Datenbanksystem bekannt sein muss, um einen Laufzeitfehler zu vermeiden. Dies kann man mit einem entsprechenden SQL-Befehl erreichen:

```
CREATE OR REPLACE DIRECTORY DIR AS 'c:\';
```

Die Prozeduren beginnen zunächst mit der Deklaration einiger Variablen:

- `xml_clob` dient als Zwischenspeicher für die CLOB-Repräsentation des Dokuments. Diese benötigen wir für das Schreiben in die Datei.
- `buffer` dient als Puffer für das Lesen aus dem CLOB und das Schreiben in die Datei. Die Größe ist die maximal mögliche für ein `VARCHAR`, nämlich 32767 Bytes (vgl. [FP03], Kapitel 8).
- `amount` speichert, wieviele Zeichen aus dem CLOB gelesen werden.
- `offset` speichert die Stelle, an der aus dem CLOB gelesen wird.
- `e_invalid_table` ist eine selbst definierte Ausnahme, die bei ungültigem Tabellennamen ausgelöst werden wird.
- Die Variable `filehandle` wird benötigt, um den Zugriff auf die zu erstellende Datei zu steuern.
- `xml_document` ist das Dokument, welches in die Datei geschrieben werden soll.
- `temp_table_name` und `table_ok` sind Hilfsvariablen für die Prüfung des übergebenen Tabellennamens.
- Die Variable `name_cursor` speichert einen Cursor für die bereits bekannte Tabelle `tabs`, die Informationen über alle Tabellen der Datenbank erhält.

Zunächst wird überprüft, ob der übergebene Tabellenname gültig ist. Dazu wird in einer mit Hilfe des Cursors realisierten Schleife geprüft, ob der angegebene Name dem Defaultwert „RETURN_ALL_TABLES“ oder einem Tabellennamen aus der Datenbank

entspricht. Falls nicht, wird nach dem in Abschnitt 3.3.1 vorgestellten Prinzip die selbstdefinierte Exception ausgelöst, und die Ausführung der Prozedur wird an dieser Stelle abgebrochen.

Die nächste `if`-Abfrage (Listings 7 und 8, Zeile 62) verwendet das Argument `use_xslt`. Abhängig davon wird das zu schreibende Dokument entweder mit der XSLT- oder der XQuery-gestützten Funktion erstellt.

Anschließend wird mit dem entsprechenden Befehl aus dem Paket `UTL_FILE` (das für das Schreiben und Lesen von Dateien gedacht ist, vgl. [Rap03], Kap. 155) die Datei mit maximaler Puffergröße (32767) zum Schreiben ('W') geöffnet, wobei hier die als Argument übergebenen Werte für den Dateinamen und das Verzeichnis verwendet werden.

Die Zeichenmenge (`amount`), die gelesen werden soll, wird ebenfalls auf den maximalen Wert gesetzt, und der `offset`, an dem gelesen wird, ist zu Beginn natürlich 1. Das CLOB, aus dem gelesen werden soll, ist die CLOB-Repräsentation des vorliegenden `XMLType`-Dokuments, die wir uns mit der Funktion `getClobVal()` beschaffen.

Die folgende Schleife (Listings 7 und 8, Zeile 77) wird über die Variable `amount` gesteuert. Dies ist zunächst irritierend, da die Variable in der Schleife offenbar nicht verändert wird. Es ist jedoch so, dass die Funktion `DBMS_LOB.READ()` (welche zum Auslesen des CLOBs dient, vgl. [Rap03], Kap. 45) in der Variablen nicht nur die Zeichenmenge übergeben bekommt, die gelesen werden soll. Nach erfolgtem Aufruf wird in dieser Variablen auch gespeichert, wie viele Zeichen tatsächlich gelesen wurden, d.h. wenn weniger Zeichen gelesen wurden, als der Puffer fasst bzw. als gelesen werden sollten, ist das ein Indikator für das Erreichen des Endes der Zeichenkette.

Der gelesene Teil des CLOBs wird in der Variablen `buffer` gespeichert und mit der Funktion `UTL_FILE.PUT()` in die zu Beginn geöffnete Datei geschrieben. Der entsprechende Schreibpuffer der Datei wird in jeder Iteration geleert, und der `offset` für das Lesen aus dem CLOB wird entsprechend der gelesenen Zeichenmenge angepasst, damit in der nächsten Runde an der richtigen Stelle weitergelesen wird. Nach Ende der Schleife wird der Dateihandle geschlossen, und die Prozeduren enden.

Im `EXCEPTION`-Block werden nun noch passende Fehlermeldungen definiert, die bei den zu erwartenden Problemen ausgegeben werden sollen. Dazu zählt auch eine Meldung für die von uns definierte Ausnahme, dass der Tabellenname nicht den Erwartungen entspricht.

Natürlich stellt sich bei diesen Prozeduren die Frage, ob es keine Alternative zu diesem recht komplizierten Vorgehen gibt. Die Begründung für den vorgestellten Ansatz findet sich in der beschränkten Länge von Strings. Man könnte sich mit der entsprechenden Funktion das `XMLType`-Dokument statt als CLOB auch als String zurückgeben lassen und diesen direkt in die Datei schreiben. Da die Länge von Strings bzw. `VARCHAR`-Variablen aber auf 32767 Bytes beschränkt ist und nicht garantiert werden kann, dass das XML-Dokument diese Beschränkung einhält, ist diese schrittweise Verarbeitung notwendig, um Laufzeitfehler durch zu große Dokumente zu vermeiden.

5 Vergleich der Verfahren

Nachdem wir alle zur Extraktion der Daten bzw. Metadaten notwendigen Prozeduren bzw. Funktionen eingeführt haben, wollen wie in diesem Abschnitt einen Vergleich der beiden vorgestellten Verfahren durchführen. Des Weiteren wollen wir einen Vergleich mit der eingangs erwähnten Relational.OWL-Implementierung in Java ziehen, denn schließlich war es ein Ziel dieser Arbeit, Alternativen hierzu aufzuzeigen. Dabei interessiert uns natürlich in erster Linie die Leistung der Verfahren im praktischen Einsatz. Weitere interessante Aspekte werden in einem zusätzlichen Abschnitt erläutert.

5.1 Geschwindigkeit

Für die Messung der Geschwindigkeit der Verfahren bei der Extraktion der Metadaten wurden Beispieldatenbanken mit unterschiedlicher Anzahl von Tabellen bzw. Spalten angelegt, für die Extraktion der Datenbankausprägung wurden Datenbanken mit unterschiedlichen Anzahlen von Tupeln bzw. Zeilen gefüllt. Da Relational.OWL Oracle-Datenbanken (noch) nicht unterstützt, basieren diese Messungen jeweils auf einer entsprechenden MySQL-Datenbank. Die Zeiten beziehen sich auf die Ausführungsdauer der im vorigen Abschnitt vorgestellten Prozeduren zum Schreiben der Dokumente in eine Datei bzw. die Ausführungsdauer der entsprechenden Funktionalität bei Relation.OWL. Alle Zeitangaben sind die Mittelwerte von 50 Durchläufen, die Werte sind hierbei in Sekunden angegeben. Messungen der reinen Transformationsgeschwindigkeit (d.h. ohne das abschließende Schreiben in eine Datei) brachten beinahe identische Ergebnisse, daher sind diese hier nicht aufgeführt.

Tabellen/Spalten	XQuery	XSLT	Relational.OWL
3/9	0,59	0,18	0,05
6/18	0,82	0,20	0,09
12/36	1,27	0,23	0,17
24/72	2,20	0,34	0,33
48/144	4,07	0,51	0,84
96/288	7,94	0,81	1,72

Tabelle 1: Messung Metadatenextraktion.

Zeilen	XQuery	XSLT	Relational.OWL
100	0,69	0,17	0,03
500	2,83	0,33	0,05
1000	5,53	0,49	0,10
2000	11,33	0,83	0,26
5000	30,35	2,06	0,71
10000	61,67	3,95	1,41

Tabelle 2: Messung Datenextraktion.

Es fällt auf, dass in beiden Fällen die Extraktion bzw. Transformation mit Hilfe von XSLT deutlich schneller ist als das entsprechende XQuery-Pendant. Eine Begründung dafür könnte man darin sehen, dass bei der Implementierung der XQuery-Anfragen noch Optimierungspotenzial vorhanden ist. Dieses Argument dürfte jedoch nur schwer zu halten sein, da selbst bei der in der Umsetzung vergleichsweise „einfachen“ Extraktion der Datenbankausprägung der Geschwindigkeitsvorsprung von XSLT beachtlich ist und mit steigender Anzahl von Informationen immer deutlicher wird. Während dieser Umstand bei der Verarbeitung der Metadaten noch verschmerzbar wäre, da dies in der Regel ein einmaliger Vorgang ist, wird die Datenbankausprägung in der Praxis wohl wesentlich häufiger extrahiert werden müssen. Folglich ist hier von einer Verwendung von XQuery eindeutig abzuraten.

Die tatsächliche Begründung für den augenfälligen Unterschied dürfte zunächst der unterschiedliche Ansatz von XQuery und XSLT sein. XQuery verwaltet im Hintergrund ein im Vergleich zu XSLT komplexeres und vielseitigeres Datenmodell, das bei der Navigationsgeschwindigkeit über die Dokumente einen negativen Einfluss haben dürfte. Des Weiteren erlaubt XQuery, wie bereits erwähnt, auch komplexe Operationen, die in XSLT so nicht oder nicht ohne Weiteres umzusetzen sind. Daher dürften die zugehörigen Verarbeitungsroutinen ihren „schlankeren“ XSLT-Pendants unterlegen sein, da schlichtweg mehr Dinge überprüft bzw. beachtet werden müssen. Ein weiterer Grund könnte auch einfach darin liegen, dass der XSLT-Standard schon wesentlich älter ist, und dementsprechend die dazugehörigen Verarbeitungsmechanismen von stetiger Optimierung in den letzten Jahren profitiert haben, während bei einem im Vergleich dazu jungen Standard wie XQuery sicherlich noch einiges an Potenzial ungenutzt bleibt.

Der Vergleich mit Relational.OWL fördert ein Ergebnis zutage, mit dem man nicht unbedingt rechnen konnte. Das Programm liegt auf einem Niveau mit der XSLT-gestützten Datenextraktion, wobei bei der Daten- bzw. Metadatenextraktion jeweils eines der Verfahren einen leichten Vorteil hat, der sich praktisch aber nicht gravierend auswirken dürfte. Dieses Ergebnis ist insofern überraschend, als dass die von uns vorgestellten Prozeduren und Funktionen relativ kurz und spezialisiert sind, während Relational.OWL ein vergleichsweise großes und komplexes Programm ist. Der relative Gleichstand mag darin begründet liegen, dass Relational.OWL sich dem Problem von einer anderen Seite nähert als unsere Lösung. Während in dieser von Anfang an auf XML-Dokumenten gearbeitet wird, werden in Relational.OWL zeitintensive Aktionen wie das Bestimmen von Fremdschlüsseln über entsprechende SQL-Anfragen gelöst, deren Ergebnisse analysiert werden. Das Erstellen von XML-Dokumenten findet erst spät statt. Insofern könnten die relativ aufwändigen Operationen, welche direkt auf der Struktur der Dokumente ausgeführt werden, ein besseres Ergebnis verhindern. Ein weiterer beachtenswerter Punkt ist sicherlich die Frage, wie effizient PL/SQL in der Datenbank ausgeführt wird, im Vergleich zu einem Java-Programm in entsprechender Laufzeitumgebung. Hier müssten noch Messungen angestellt werden, um diese Frage befriedigend zu klären.

5.2 Weitere Aspekte

Neben dem Hauptaspekt der Geschwindigkeit ist sicherlich auch interessant, inwiefern die vorgestellten Lösungen modifizierbar bzw. wiederverwendbar sind.

Relational.OWL ist in seiner Implementierung so angelegt, dass es verhältnismäßig leicht modifiziert werden kann. Es bleibt dennoch ein relativ komplexes Programm, d.h. für die schnelle Umsetzung zukünftiger Erweiterungen oder Veränderungen der zu extrahierenden Informationen wären entsprechende Vorkenntnisse bzw. Einarbeitungszeit erforderlich. Im Gegensatz dazu sind die von uns vorgestellten Lösungen sehr kurz, d.h. eine Anpassung an zukünftige Erfordernisse durch Analyse der XML-Rohdaten und eine Umsetzung in entsprechende Anfragen bzw. Transformationen auf Grundlage der bestehenden Lösungen dürfte relativ zügig zu erledigen sein. Doch auch bei den von uns vorgestellten Lösungen lassen sich noch einmal Unterschiede feststellen, was die Modifizierbarkeit betrifft.

Ein XSLT-Stylesheet ist selbst ein XML-Dokument, welches ohne Weiteres außerhalb des Datenbanksystems gespeichert und bearbeitet werden kann. Die datenbankeigenen Funktionen erlauben es dann, das Stylesheet zur Laufzeit aus einer externen Datei einzubinden. Es ist damit nicht zwingend erforderlich, die Funktionen bei Veränderungen des Stylesheets neu zu kompilieren. Dass in der vorgestellten Lösung das Stylesheet als konstante Zeichenkette direkt in der Funktion angegeben wird, geschieht lediglich der Einfachheit halber.

Im Gegensatz dazu erlaubt es die Funktion zur Verarbeitung von XQuery-Anfragen nicht, dass eine Anfrage von „außen“ übergeben wird. Sie muss bereits zur Kompilierzeit als konstante Zeichenkette feststehen, was eine Veränderung oder Erweiterung des Programms erschwert, da jede Modifikation eine Neuübersetzung nötig macht.

5.3 Fazit

Wie wir gesehen haben, sind in diesem Anwendungsszenario Relational.OWL und XSLT in punkto Geschwindigkeit in etwa gleichauf, während XQuery deutlich abfällt. Was im direkten Vergleich jedoch für XSLT spricht, ist die Tatsache, dass die Funktionalität direkt in der Datenbank bereitsteht und nicht von einem externen Programm bereitgestellt werden muss. Dies kann, wie zu Beginn erwähnt, wünschenswert oder gar erforderlich sein. Weiterhin punktet XSLT durch die relativ einfache Durchführung von Anpassungen und Erweiterungen, weswegen insgesamt der XSLT-basierten Lösung der Vorzug zu geben sein dürfte.

6 Weiterführende Überlegungen

Nachdem die Lösung vorgestellt und diskutiert wurde, werden nun noch einige weitere Punkte angesprochen, die einer Überlegung wert sind und möglicherweise die Basis für weiterführende Arbeiten darstellen.

6.1 Erweiterungen der Funktionalität

Mit der vorgestellten Lösung wird die Oracle 10g in die Lage versetzt, ihre Daten und Metadaten in einer der Relational.OWL-Ontologie entsprechenden Form zu exportieren. Für die praktische Anwendung ist es aber ebenso wichtig, diese Informationen wieder in die Datenbank zu integrieren bzw. zu importieren. Das Programm Relational.OWL leistet dies bereits, und der nächste logische Schritt wäre eine Umsetzung dieser Funktionalität in der 10g. Erforderlich ist dafür eine „Übersetzung" der entsprechenden XML-Dokumente in SQL-Statements, wobei XQuery und XSLT die naheliegendsten Hilfsmittel zur Umsetzung sein dürften.

Weiterhin interessant ist die Umsetzung möglicher Erweiterungen der Relational.OWL-Ontologie, die in der entsprechenden Arbeit [PC05a] in Aussicht gestellt werden. Dazu zählt die Darstellung zusätzlicher Metadaten (z.B. Trigger), für die dann die jeweiligen Rohdaten-Dokumente (siehe Abschnitt 4.1) einer weiteren Analyse unterzogen werden müssten.

Eine weiterer Aspekt betrifft in erster Linie die Leistungsfähigkeit der Lösung im praktischen Einsatz. Mit den vorgestellten Prozeduren bzw. Funktionen ist es bei jeder Änderung an den Daten bzw. Metadaten erforderlich, die XML-Dokumente neu zu generieren, um ein aktuelles Abbild der Datenbank zu erhalten. Sinnvoller wäre es jedoch, bei einer Änderung an der Datenbank (über SQL) die Änderungen parallel an den Dokumenten vorzunehmen. Hier wäre zu überlegen, wie so etwas zu steuern wäre (z.B. über Trigger), und inwiefern XSLT bzw. XQuery für derartige Änderungsoperationen benutzt werden können. Ein weiterer untersuchenswerter Ansatz hierfür ist auch die XQuery Update Facility [CFR06], an der momentan beim W3C gearbeitet wird, und die explizit für das konsistente Durchführen von Änderungsoperationen auf XML-Dokumenten gedacht ist.

6.2 Andere Lösungsansätze

Wie in Abschnitt 5 angesprochen, basiert unsere Lösung auf der direkten Manipulation von XML-Dokumenten, was möglicherweise zu Einbußen bei der Geschwindigkeit führt. Eine Alternative hierzu wäre es, die notwendigen Informationen mit Hilfe von SQL aus der Datenbank anzufragen, und die Ergebnisse dieser Anfragen in XML zu „übersetzen". Dabei stellt sich jedoch die Frage, wie dies umzusetzen wäre. Die in [Hig03], Kapitel 15 vorgestellten Funktionen, die auf dem in SQL 2003 vorgestellten SQL/XML-Standard basieren, ermöglichen zwar das Generieren von XML aus Anfrageergebnissen. Die Möglichkeiten dieser Funktionen (z.B. bei der Benennung von Elementen) sind jedoch sehr begrenzt. Eine andere denkbare Möglichkeit wäre die „naive" Erstellung von XML-Dokumenten z.B. durch simple Konkatenation von Strings, wobei Nachvollziehbarkeit bzw. Wiederverwendbarkeit einer solchen Lösung wahrscheinlich zu wünschen übrig ließen.

6.3 Umsetzung auf andere Systeme

Nachdem von uns exemplarisch gezeigt wurde, wie die Extraktion einer Oracle 10g-Datenbank umgesetzt werden kann, ist natürlich auch die Umsetzung der Lösung auf

vergleichbare Datenbanksysteme von Interesse. Zu beachten ist hierbei in erster Linie, ob und wie XML-Verarbeitung vom Datenbanksystem unterstützt wird. Das weitverbreitete MySQL-System z.B. bietet von sich aus nur eine rudimentäre Unterstützung für XML (vgl. [SQL06], Abschnitt 12.9). Hier müsste man auf den im letzten Abschnitt beschriebenen Ansatz zurückgreifen, sich dem Problem von SQL-Seite zu nähern.

In kommerziellen Datenbanksystemen hingegen scheint sich XML angesichts seiner wachsenden Bedeutung langsam durchzusetzen. IBM z.B. wird in seiner nächsten DB2-Version (DB2 9, auch „Viper“ genannt) umfangreiche XML-Funktionalität integrieren, inkl. Unterstützung für XQuery (vgl. [IBM06]). Auch in Microsoft SQL Server 2005 ist die Verarbeitung von XQuery-Anfragen vorgesehen (vgl. [MIC06]). Zu klären wäre allerdings, inwiefern diese Systeme eine Extraktion der Daten und insbesondere der Metadaten unterstützen. Wenn es möglich ist, diese Informationen ähnlich wie im Oracle-Datenbanksystem vollständig in XML zu extrahieren, sollte ein Umsetzung der vorliegenden Lösung auf die entsprechenden Systeme relativ leicht möglich sein.

A Quelltexte

A.1 Die Funktion METADATA_AS_RAW_XML()

```
1  CREATE OR REPLACE FUNCTION METADATA_AS_RAW_XML
2  (
3  -- Tabelle, die zurückgegeben werden soll. Bei fehlendem Argument
4  -- wird der Default-Wert verwendet.
5  table_name STRING DEFAULT 'RETURN_ALL_TABLES'
6  )
7  -- Rückgabewert ist ein XMLType-Dokument
8  RETURN XMLType
9  AS
10 -- Handle für den Metadaten-Export
11 handle NUMBER;
12 -- Handle für das Festlegen der Metadaten-Transformation
13 transform_handle NUMBER;
14 -- temporäres CLOB
15 temp_clob CLOB;
16 -- CLOB für das fertige Dokument
17 final_clob CLOB;
18 BEGIN
19 -- Festlegen, dass man Metadaten von Tabellen haben möchte
20 handle := DBMS_METADATA.OPEN('TABLE');
21 -- Festlegen, dass man XML-Daten möchte(nicht DDL)
22 transform_handle := DBMS_METADATA.ADD_TRANSFORM(handle,'MODIFY');
23 -- Wenn nicht alle Tabellen geliefert werden sollen...
24 if table_name != 'RETURN_ALL_TABLES'
25  -- ...filtern nach dem entsprechenden Namen
26  then DBMS_METADATA.SET_FILTER(handle,'NAME', table_name);
27 end if;
28 LOOP
29  -- Holen der Informationen über eine Tabelle
30  temp_clob:=DBMS_METADATA.FETCH_CLOB(handle);
31 -- Verlassen der Schleife, wenn alle Daten geholt wurden
32 EXIT WHEN temp_clob IS NULL;
33  -- Anhängen der geholten Daten an bisherige Daten
34  final_clob:=final_clob||temp_clob;
35 END LOOP;
36 -- Schließen des Handles
37 DBMS_METADATA.CLOSE(handle);
38 -- Einführen des Database-Tags
39 final_clob:='<DATABASE>'||final_clob||'</DATABASE>';
40 -- Konstruktion und Rückgabe des fertigen XML-Dokuments
41 return XMLType(final_clob);
42 end;
```

Listing 1: METADATA_AS_RAW_XML()

A.2 Die Funktion DATA_AS_RAW_XML()

```
1  CREATE OR REPLACE FUNCTION DATA_AS_RAW_XML
2  (
3  -- Tabelle, die zurückgegeben werden soll. Bei fehlendem Argument
4  -- wird der Default-Wert verwendet.
5  table_name STRING default 'RETURN_ALL_TABLES'
6  )
7  -- Rückgabewert ist ein XMLType-Dokument
8  RETURN XMLType
9  AS
10 -- Schleifenvariable für den Cursor
11 temp_table_name varchar(255);
12 -- temporäres CLOB
13 temp_clob CLOB;
14 -- CLOB für das fertige Dokument
15 final_clob CLOB;
16 -- CLOB für das Ergebnis der SQL-Anfrage
17 query_result CLOB;
18 -- Cursor für die Iteration über alle Tabellennamen
19 CURSOR name_cursor IS select table_name from tabs;
20 begin
21 -- Wenn die gesamte DB geliefert werden soll...
22 if table_name='RETURN_ALL_TABLES'
23  then
24  -- Öffnen des vorher definierten Cursors
25  OPEN name_cursor;
26  LOOP
27   -- Extraktion des Tabellennamens
28   FETCH name_cursor INTO temp_table_name;
29   -- Verlassen der Schleife nach Extraktion aller Namen
30  EXIT WHEN name_cursor%NOTFOUND;
31   -- Extraktion des XML-Dokumentes
32   query_result:=DBMS_XMLQuery.getXML('select * from '||temp_table_name);
```

```
33   -- Konkatenation des bisher vorhandenen CLOBs mit dem Ergebnis der
34   -- Anfrage, dazu Einführung des TABLE-Tags
35   temp_clob:=temp_clob||
36   '<TABLE name="'||temp_table_name||'">'||
37   query_result||
38   '</TABLE>';
39  END LOOP;
40  -- Schließen des Cursors
41  CLOSE name_cursor;
42  -- Wenn nur eine Tabelle angefragt wurde...
43  else
44  -- Extraktion des XML-Dokumentes
45  query_result:=DBMS_XMLQuery.getXML('select * from '||table_name);
46  -- Einführung des TABLE-Tags
47  temp_clob:=temp_clob||
48  '<TABLE name="'||table_name||'">'||
49  query_result||
50  '</TABLE>';
51 end if;
52 -- Löschen der redundanten Prozessierungsanweisungen
53 temp_clob:=replace(temp_clob,'<?xml version = ''1.0''?>');
54 -- Einführen des DATABASE-Tags
55 final_clob:='<DATABASE>'||temp_clob||'</DATABASE>';
56 -- Konstruktion und Rückgabe eines XMLType-Dokuments
57 return XMLType(final_clob);
58 END;
```

Listing 2: DATA_AS_RAW_XML()

A.3 Die Funktion METADATA_AS_OWL()

```
1
2  CREATE OR REPLACE FUNCTION METADATA_AS_OWL
3  (
4  -- Name der Datenbank
5  database_name STRING,
6  -- Tabelle, die zurückgegeben werden soll. Bei fehlendem Argument
7  -- wird der Default-Wert verwendet.
8  table_name STRING DEFAULT 'RETURN_ALL_TABLES'
9  )
10 -- Rückgabewert ist ein XMLType-Dokument
11 RETURN XMLType
12 AS
13 -- Variable für fertiges Dokument
14 return_doc XMLType;
15 -- Variable für "Rohdaten"
16 meta_data_doc XMLType;
17 -- Variable für das "Verpacken" des Datenbanknamens
18 dbname_doc XMLType;
19 BEGIN
20 -- Extrahieren der "Rohdaten"
21 meta_data_doc:=METADATA_AS_RAW_XML(table_name);
22 -- "Verpacken" des Datenbanknamens
23 dbname_doc:=XMLType('<dbname value="'||database_name||'"/>');
24 -- Beginn der XQuery-Anfrage
25 SELECT
26 XMLQuery(
27 '
28 (:Wurzelelement mit Namensraumfestlegung:)
29 <rdf:RDF
30 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
31 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
32 xmlns:owl="http://www.w3.org/2002/07/owl#"
33 xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
34 xmlns:dbs="http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#"
35 >
36 (:Beschreibung der Datenbank mit ihren Tabellen:)
37 <owl:Class rdf:ID="{data($dbn_doc/dbname/@value)}">
38 {
39 (:Typangabe:)
40 <rdf:type rdf:resource="http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#Database"/>,
41 (:Für jeden Tabellennamen...:)
42 for $tab_name in $md_doc/DATABASE/ROWSET/ROW/TABLE_T/SCHEMA_OBJ/NAME
43 return
44  (:...Wird die entsprechende "hasTable"-Eigenschaft hinzugefügt:)
45  <dbs:hasTable rdf:resource="{concat("#",data($tab_name))}" />
46 }
47 </owl:Class>
48 {
49 (:Für jede Tabelle...:)
50 for $table in $md_doc/DATABASE/ROWSET/ROW/TABLE_T
51 return
52  (:...wird eine entsprechende Beschreibung ausgegeben:)
53  <owl:Class rdf:ID="{data($table/SCHEMA_OBJ/NAME)}">
54  (:Typangabe:)
55  <rdf:type rdf:resource="http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#Table"/>
56  {
```

```
 (:Für jedes Spalte der Tabelle...:)
 for $col_name in $table/COL_LIST/COL_LIST_ITEM/NAME
 return
  (:...Wird die entsprechende "hasColumn"-Eigenschaft hinzugefügt:)
  <dbs:hasColumn rdf:resource=
  "{concat(concat(concat("#",data($table/SCHEMA_OBJ/NAME)),"."),data($col_name))}"/>
 }
 {
 (:Wenn die Tabelle einen Primärschlüssel hat...:)
 if ($table/CON1_LIST/CON1_LIST_ITEM[CONTYPE="2"])
  then
  (:...ausgeben der entsprechenden Tags:)
  <dbs:isIdentifiedBy>
   <dbs:PrimaryKey>
   {
   (:Für jede Spalte des Primärschlüssels...:)
   for $pkey_col_name in $table/CON1_LIST/CON1_LIST_ITEM[CONTYPE="2"]/COL_LIST/COL_LIST_ITEM/COL/NAME
   return
    (:...Wird die entsprechende "hasColumn"-Eigenschaft hinzugefügt:)
    <dbs:hasColumn rdf:resource=
    "{concat(concat(concat("#",data($table/SCHEMA_OBJ/NAME)),"."),data($pkey_col_name))}"/>
   }
   </dbs:PrimaryKey>
  </dbs:isIdentifiedBy>
  else ""
 }
</owl:Class>
}
{
(:Für jede Tabelle...:)
for $table in $md_doc/DATABASE/ROWSET/ROW/TABLE_T
 (:...für jedes Spalte der Tabelle...:)
 for $column in $table/COL_LIST/COL_LIST_ITEM
 return
  (:...wird eine entsprechende Beschreibung ausgegeben :)
  <owl:DatatypeProperty rdf:ID="{concat(concat(data($table/SCHEMA_OBJ/NAME),"."),data($column/NAME))}">
  (:Typangabe:)
  <rdf:type rdf:resource="http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#Column"/>
  (:"Domain" ist die übergeordnete Tabelle:)
  <rdfs:domain rdf:resource="{concat("#",data($table/SCHEMA_OBJ/NAME))}"/>
  {
  (:Festlegung des Datentyps:)
  (:Wenn es sich um eine Zeichenkette handelt...:)
  if($column[TYPE_NUM="1"] or $column[TYPE_NUM="96"])
   then <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
   (:Wenn es sich um eine Datumsangabe handelt...:)
   else if ($column[TYPE_NUM="12"])
         then <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#date"/>
         (:Wenn es sich um eine Dezimalzahl handelt...:)
         else if($column[TYPE_NUM="2"] and $column/SCALE and $column/PRECISION_NUM)
               then <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#decimal"/>
               (:Wenn es sich um eine Integerzahl handelt...:)
               else if($column[TYPE_NUM="2"] and $column/SCALE)
                     then <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#integer"/>
                     (:Wenn es sich um eine Float-Zahl handelt...:)
                     else if($column[TYPE_NUM="2"] and $column/PRECISION_NUM)
                          then <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#float"/>
                          (:Wenn es sich um eine gemischte Datums-Zeit-Angabe handelt...:)
                          else if($column[TYPE_NUM="180"])
                                then <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#dateTime"/>
                                else ""
  }
  {
  (:Festlegung der Länge:)
  (:Wenn es sich um eine Zeichenkette handelt...:)
  if($column[TYPE_NUM="1"] or $column[TYPE_NUM="96"])
   (:...steht die Länge im "LENGTH"-Element:)
   then <dbs:length>{data($column/LENGTH)}</dbs:length>
   (:Wenn es sich um eine Float- oder Dezimalzahl handelt...:)
   else if ($column[TYPE_NUM="2"] and $column/PRECISION_NUM)
         (:...steht die Länge im "PRECISION_NUM"-Element :)
         then <dbs:length>{data($column/PRECISION_NUM)}</dbs:length>
         else ""
  }
  {
  (:Festlegung der Skalierung bei Dezimalzahlen :)
  if($column[TYPE_NUM="2"] and $column/PRECISION_NUM and $column/SCALE)
   then <dbs:scale>{data($column/SCALE)}</dbs:scale>
   else ""
  }
  {
  (:Hier werden die Fremdschlüsselabhängigkeiten behandelt:)
  (:Iteration über alle Fremdschlüssel:)
  for $constraint in $table/CON2_LIST/CON2_LIST_ITEM[CONTYPE="4"]
   (:Iteration über alle Spalten dieses Fremdschlüssels:)
   for $fkey_col in $constraint/SRC_COL_LIST/SRC_COL_LIST_ITEM
    (:Iteration über alle Zielspalten:)
    for $target_col in $fkey_col/../../TGT_COL_LIST/TGT_COL_LIST_ITEM
    (: Wenn der Name der aktuellen Spalte mit dem Namen einer Fremdschlüsselspalte übereinstimmt...:)
    (: ...und die Position dieser Fremdschlüsselspalte mit der Position der Zielspalte übereinstimmt...:)
```

```
147     (: ...dann verweist die aktuelle Spalte auf diese Zielspalte! :)
148     where $column/NAME eq $fkey_col/COL/NAME and $target_col/POS_NUM eq $fkey_col/POS_NUM
149     return
150     <dbs:references rdf:resource=
151     "{concat(concat(concat("#",data($fkey_col/../../SCHEMA_OBJ/NAME)),"."),data($target_col/COL/NAME))}"/>
152    }
153    </owl:DatatypeProperty>
154  }
155  </rdf:RDF>
156  '
157  PASSING meta_data_doc AS "md_doc", dbname_doc AS "dbn_doc" -- Übergabe der "Rohdaten" und des DB-Namens
158  RETURNING CONTENT) INTO return_doc FROM DUAL; -- Festlegung der Ausgabevariablen
159  -- Einfügen der Prozessierungsinformationen sowie Konstruktion des Dokuments
160  return_doc:=XMLType('<?xml version = "1.0" encoding = "ISO-8859-1"?>'||
161  XMLType.getClobVal(return_doc));
162  -- Rückgabe des Dokuments
163  return return_doc;
164  END;
```

Listing 3: METADATA_AS_OWL()

A.4 Die Funktion DATA_AS_OWL()

```
1   CREATE OR REPLACE FUNCTION DATA_AS_OWL
2   (
3   -- Speicherort der zugehörigen Schemaontologie
4   schema_url STRING,
5   -- Tabelle, die zurückgegeben werden soll. Bei fehlendem Argument
6   -- wird der Default-Wert verwendet.
7   table_name STRING default 'RETURN_ALL_TABLES'
8   )
9   -- Rückgabewert ist ein XMLType-Dokument
10  RETURN XMLType
11  AS
12  -- Variable für Rückgabedokument
13  return_doc XMLType;
14  -- Variable für "Rohdaten"
15  table_data_doc XMLType;
16  begin
17  -- Extraktion der "Rohdaten"
18  table_data_doc:=DATA_AS_RAW_XML(table_name);
19  -- Begin der XQuery-Anfrage
20  SELECT
21  XMLQuery(
22  '
23  (: Wurzel-Tag mit Namensraum-Festlegung :)
24  <rdf:RDF
25  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
26  xmlns:dbinst="SCHEMA_PLACEHOLDER#"
27  >
28  {
29  (: Iteration über alle Tabellen :)
30  for $table in $data_doc/DATABASE/TABLE
31   (: Iteration über alle Zeilen :)
32   for $row in $table/ROWSET/*
33   (: Bindung des Tabellennamens an eine Variable:)
34   let $tab_name:=$table/data(@name)
35   (:Bildung des Elementnamens durch Konkatenation :)
36   return
37   element {concat("dbinst:", $tab_name)}
38   {
39    (: Iteration über alle Spalten :)
40    for $column in $row/*
41    (:Bildung des Elementnamens durch Konkatenation :)
42    return
43    element{concat(concat(concat("dbinst:",$tab_name),"."),name($column))}
44    {
45     data($column) (: Elementinhalt :)
46    }
47   }
48  }
49  </rdf:RDF>
50  '
51  PASSING table_data_doc AS "data_doc"     -- Übergabe der "Rohdaten"
52  RETURNING CONTENT) INTO return_doc FROM DUAL; -- Festlegung der Zielvariablen
53  -- Einfügen der Verarbeitungsinformationen und des Schemaspeicherortes
54  return_doc:=XMLType('<?xml version = "1.0" encoding = "ISO-8859-1"?>'||
55  replace(XMLType.getClobVal(return_doc),'SCHEMA_PLACEHOLDER',schema_url));
56  -- Rückgabe des Dokuments
57  return return_doc;
58  END;
```

Listing 4: DATA_AS_OWL()

A.5 Die Funktion METADATA_AS_OWL_WITH_XSLT()

```
1  CREATE OR REPLACE FUNCTION METADATA_AS_OWL_WITH_XSLT
2  (
3  -- Name der Datenbank
4  database_name in STRING,
5  -- Tabelle, die zurückgegeben werden soll. Bei fehlendem Argument
6  -- wird der Default-Wert verwendet.
7  table_name in STRING DEFAULT 'RETURN_ALL_TABLES'
8  )
9  -- Rückgabewert ist ein XMLType-Dokument
10 RETURN XMLType
11 AS
12 -- Variable für das fertige Dokument
13 return_doc XMLType;
14 -- Variable für die "Rohdaten"
15 meta_data_doc XMLType;
16 -- Variable für das Stylesheet
17 XSLT_doc XMLType;
18 BEGIN
19 -- Extrahieren der "Rohdaten"
20 meta_data_doc:=METADATA_AS_RAW_XML(table_name);
21 -- Erstellen des Stylesheets
22 XSLT_doc:=XMLType(
23 '
24 <!--Beginn des Stylesheets mit Namensraumfestlegung -->
25 <xsl:stylesheet version="1.0"
26 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
27 >
28 <!--Parameter für den Datenbanknamen-->
29 <xsl:param name="DBNAME"/>
30 <!--Als "Einstiegspunkt" dient das DATABASE-Element -->
31 <xsl:template match="/DATABASE">
32 <!--Wurzelelement mit Namensraumfestlegung -->
33 <rdf:RDF
34 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
35 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
36 xmlns:owl="http://www.w3.org/2002/07/owl#"
37 xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
38 xmlns:dbs="http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#"
39 >
40 <!-- Beschreibung der Datenbank mit ihren Tabellen -->
41 <owl:Class rdf:ID="{$DBNAME}">
42 <!-- Typangabe -->
43 <rdf:type rdf:resource="http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#Database" />
44 <!--Für jeden Tabellennamen... -->
45 <xsl:for-each select="ROWSET/ROW/TABLE_T/SCHEMA_OBJ/NAME">
46  <!-- ...wird die entsprechende "hasTable"-Eigenschaft hinzugefügt-->
47  <dbs:hasTable rdf:resource="#{.}" />
48 </xsl:for-each>
49 </owl:Class>
50 <!--Für jede Tabelle... -->
51 <xsl:for-each select="ROWSET/ROW/TABLE_T">
52  <!--...wird eine entsprechende Beschreibung ausgegeben -->
53  <owl:Class rdf:ID="{SCHEMA_OBJ/NAME}">
54  <!--Typangabe-->
55  <rdf:type rdf:resource="http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#Table"/>
56  <!-- Für jede Spalte...-->
57  <xsl:for-each select="COL_LIST/COL_LIST_ITEM/NAME">
58   <!-- ...wird die entsprechende "hasColumn"-Eigenschaft hinzugefügt-->
59   <dbs:hasColumn rdf:resource="#{../../../SCHEMA_OBJ/NAME}{.{.}"/>
60  </xsl:for-each>
61  <!-- Wenn die Tabelle Primärschlüssel hat..-->
62  <xsl:if test ="CON1_LIST/CON1_LIST_ITEM/CONTYPE=2">
63   <!--Ausgeben der entsprechenden Tags-->
64   <dbs:isIdentifiedBy>
65    <dbs:PrimaryKey>
66     <!--Für jede Primärschlüsselspalte...-->
67     <xsl:for-each select="CON1_LIST/CON1_LIST_ITEM[CONTYPE=2]/COL_LIST/COL_LIST_ITEM/COL/NAME">
68      <!-- ...wird die entsprechende "hasColumn"-Eigenschaft hinzugefügt-->
69      <dbs:hasColumn rdf:resource="#{../../../../../../SCHEMA_OBJ/NAME}{.{.}"/>
70     </xsl:for-each>
71    </dbs:PrimaryKey>
72   </dbs:isIdentifiedBy>
73  </xsl:if>
74  </owl:Class>
75 </xsl:for-each>
76 <!-- Für jede Tabelle...-->
77 <xsl:for-each select="ROWSET/ROW/TABLE_T">
78  <!--Für jede Spalte dieser Tabelle...-->
79  <xsl:for-each select="COL_LIST/COL_LIST_ITEM">
80   <!--...wird eine entsprechende Beschreibung ausgegeben -->
81   <owl:DatatypeProperty rdf:ID="{../../SCHEMA_OBJ/NAME}{.{NAME}">
82   <!--Typangabe-->
83   <rdf:type rdf:resource="http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#Column"/>
84   <!--"Domain" ist die übergeordnete Tabelle-->
85   <rdfs:domain rdf:resource="#{../../SCHEMA_OBJ/NAME}"/>
86   <!--Festlegung des Datentyps-->
87   <xsl:choose>
```

```
88     <!-- Wenn es sich um eine Zeichenkette handelt...-->
89     <xsl:when test="TYPE_NUM=1 or TYPE_NUM=96">
90      <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
91     </xsl:when>
92     <!-- Wenn es sich um eine Datumsangabe handelt...-->
93     <xsl:when test="TYPE_NUM=12">
94      <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#date"/>
95     </xsl:when>
96     <!-- Wenn es sich um eine Dezimalzahl handelt...-->
97     <xsl:when test="TYPE_NUM=2 and SCALE and PRECISION_NUM">
98      <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#decimal"/>
99     </xsl:when>
100    <!-- Wenn es sich um eine Integerzahl handelt...-->
101    <xsl:when test="TYPE_NUM=2 and SCALE">
102     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#integer"/>
103    </xsl:when>
104    <!-- Wenn es sich um eine Float-Zahl handelt...-->
105    <xsl:when test="TYPE_NUM=2 and PRECISION_NUM">
106     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#float"/>
107    </xsl:when>
108    <!-- Wenn es sich um eine kombinierte Datum-Zeit-Angabe handelt..-->
109    <xsl:when test="TYPE_NUM=180">
110     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#dateTime"/>
111    </xsl:when>
112   <!-- Ende der Datentypfestlegung-->
113   </xsl:choose>
114   <!-- Festlegung der Länge-->
115   <xsl:choose>
116    <!-- Wenn es sich um eine Zeichenkette handelt...-->
117    <xsl:when test="TYPE_NUM=1 or TYPE_NUM=96">
118     <!--...ist die Länge im "LENGTH"-Element gespeichert-->
119     <dbs:length>
120      <xsl:value-of select="LENGTH"/>
121     </dbs:length>
122    </xsl:when>
123    <!-- Wenn es sich um eine Float- oder Dezimalzahl handelt...-->
124    <xsl:when test="TYPE_NUM=2 and PRECISION_NUM">
125     <!--...ist die Länge im "PRECISION_NUM"-Element gespeichert-->
126     <dbs:length>
127      <xsl:value-of select="PRECISION_NUM"/>
128     </dbs:length>
129    </xsl:when>
130   <!-- Ende der Längenfestlegung-->
131   </xsl:choose>
132   <!-- Festlegung der Skalierung bei Dezimalzahlen-->
133   <xsl:if test="TYPE_NUM=2 and PRECISION_NUM and SCALE">
134    <!-- Der Wert ist im "SCALE"-Element gespeichert-->
135    <dbs:scale>
136     <xsl:value-of select="SCALE"/>
137    </dbs:scale>
138   </xsl:if>
139   <!--Hier beginnt die Behandlung der Fremdschlüsselverweise-->
140   <!--Die aktuell behandelte Spalte wird in der Variablen festgehalten-->
141   <xsl:variable name = "column" select ="."/>
142    <!--Iteration über alle Fremdschlüssel -->
143   <xsl:for-each select="../../CON2_LIST/CON2_LIST_ITEM[CONTYPE=4]">
144    <!--Iteration über die Spalten dieses Fremdschlüssels-->
145    <xsl:for-each select="SRC_COL_LIST/SRC_COL_LIST_ITEM">
146     <!--Die aktuelle Spalte des Fremdschlüssels wird in der Variablen festgehalten-->
147     <xsl:variable name = "fkey_col" select ="."/>
148     <!--Iteration über die Spalten, auf die der Fremdschlüssel verweist-->
149     <xsl:for-each select="../../TGT_COL_LIST/TGT_COL_LIST_ITEM">
150      <!--Die aktuelle "Zielspalte" des Fremdschlüssels wird in der Variablen festgehalten-->
151      <xsl:variable name = "target_col" select ="."/>
152      <!--Wenn der Name der aktuellen Spalte mit dem Namen einer Fremdschlüsselspalte übereinstimmt...-->
153      <!-- ...und die Position dieser Fremdschlüsselspalte mit der Position der Zielspalte übereinstimmt...-->
154      <!-- ...dann verweist die aktuelle Spalte auf diese Zielspalte! -->
155      <xsl:if test="$column/NAME=$fkey_col/COL/NAME and $target_col/POS_NUM=$fkey_col/POS_NUM">
156       <dbs:references rdf:resource="#{$fkey_col/../../SCHEMA_OBJ/NAME}.{$target_col/COL/NAME}"/>
157      </xsl:if>
158     <!--Ende der Iteration über die Zielspalten-->
159     </xsl:for-each>
160    <!--Ende der Iteration über die Fremdschlüsselspalten-->
161    </xsl:for-each>
162   <!--Ende der Iteration über die Fremdschlüssel-->
163   </xsl:for-each>
164   </owl:DatatypeProperty>
165  <!--Ende der Iteration über die Spalten-->
166  </xsl:for-each>
167 <!-- Ende der Iteration über die Tabellen-->
168 </xsl:for-each>
169 </rdf:RDF>
170 </xsl:template>
171 </xsl:stylesheet>
172 '
173 );
174 -- Anwenden des Stylesheets. Der DB-Name wird als Parameter übergeben
175 return_doc:=meta_data_doc.transform(XSLT_doc,'DBNAME=""'||database_name||'""');
176 -- Einfügen der Prozessierungsinformationen sowie Konstruktion des Dokuments
177 return_doc:=XMLType('<?xml version = "1.0" encoding="ISO-8859-1"?>'||
```

```
XMLType.getClobVal(return_doc));
-- Rückgabe des Dokuments
return return_doc;
END;
```

Listing 5: METADATA_AS_OWL_WITH_XSLT()

A.6 Die Funktion DATA_AS_OWL_WITH_XSLT()

```
CREATE OR REPLACE FUNCTION DATA_AS_OWL_WITH_XSLT
(
-- Speicherort der zugehörigen Schemaontologie
schema_url STRING,
-- Tabelle, die zurückgegeben werden soll. Bei fehlendem Argument
-- wird der Default-Wert verwendet.
table_name STRING default 'RETURN_ALL_TABLES'
)
-- Rückgabewert ist ein XMLType-Dokument
RETURN XMLType
AS
-- Variable für Rückgabedokument
return_doc XMLType;
-- Variable für das Stylesheet
XSLT_doc XMLType;
-- Variable für "Rohdaten"
table_data_doc XMLType;
BEGIN
-- Extraktion der "Rohdaten"
table_data_doc:=DATA_AS_RAW_XML(table_name);
-- Konstruktion des Stylesheets
XSLT_doc:=XMLType(
'
<!-- Beginn des Stylesheets mit Festlegung des Namensraumes -->
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 >
<!--Als "Einstiegspunkt" dient das DATABASE-Element -->
<xsl:template match="/DATABASE">
<!-- Wurzeltag mit Namensraumfestlegung -->
<rdf:RDF xmlns:dbinst="SCHEMA_PLACEHOLDER" xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
<!-- Iteration über alle Tabellen -->
<xsl:for-each select="TABLE/ROWSET">
 <!-- Iteration über alle Zeilen -->
 <xsl:for-each select="ROW">
  <!-- Konstruktion des Elementnamens -->
  <xsl:element name="dbinst:{../../@name}" namespace="SCHEMA_PLACEHOLDER">
   <!-- Iteration über alle Spalten-->
   <xsl:for-each select="*">
    <!-- Konstruktion des Elementnamens -->
    <xsl:element name="dbinst:{../../../@name}{.{name(.)}" namespace="SCHEMA_PLACEHOLDER">
     <!-- Elementinhalt ist Inhalt der aktuellen Spalte -->
     <xsl:value-of select="."/>
    </xsl:element>
   <!-- Ende der Spalten-Schleife -->
   </xsl:for-each>
  </xsl:element>
 <!-- Ende der Zeilen-Schleife -->
 </xsl:for-each>
<!-- Ende der Tabellen-Schleife -->
</xsl:for-each>
</rdf:RDF>
</xsl:template>
</xsl:stylesheet>
'
);
-- Anwenden des Stylesheets
return_doc:=table_data_doc.transform(XSLT_doc);
-- Einfügen der Verarbeitungsinformationen und des Schemaspeicherortes
return_doc:=XMLType('<?xml version = "1.0" encoding = "ISO-8859-1"?>'||
replace(XMLType.getClobVal(return_doc),'SCHEMA_PLACEHOLDER',schema_url));
-- Rückgabe des Dokuments
return return_doc;
END;
```

Listing 6: DATA_AS_OWL_WITH_XSLT()

A.7 Die Prozedur EXTRACT_METADATA_INTO_FILE()

```
CREATE OR REPLACE PROCEDURE EXTRACT_METADATA_INTO_FILE
(
-- Name der Datenbank
database_name string,
-- Verzeichnis, in dem gespeichert werden soll. Muss vorher angelegt werden!
directory_name string,
```

```
7  -- Dateiname, unter dem gespeichert werden soll
8  file_name string,
9  -- Gibt an, ob XSLT benutzt werden soll. Alternativ wird XQuery benutzt
10 use_xslt boolean,
11 -- Tabelle, die zurückgegeben werden soll. Bei fehlendem Argument
12 -- wird der Default-Wert verwendet.
13 table_name string default 'RETURN_ALL_TABLES'
14 )
15 AS
16 -- Variable für die CLOB-Repräsentation des XML-Dokuments
17 xml_clob CLOB;
18 -- Puffer für das Schreiben in die Datei
19 buffer VARCHAR2(32767);
20 -- Speichert, wieviel Zeichen aus dem CLOB gelesen werden
21 amount INTEGER;
22 -- Offset für das CLOB, aus dem gelesen wird
23 offset NUMBER;
24 -- Exception, für den Fall, dass der Tabellenname ungültig ist
25 e_invalid_table EXCEPTION;
26 -- Handle für das Schreiben in die Datei
27 filehandle UTL_FILE.FILE_TYPE;
28 -- Auszugebendes Dokument
29 xml_document XMLType;
30 -- temporäre Variable für den Test, ob der Tabellenname gültig ist
31 temp_table_name varchar(255);
32 -- Gibt an, ob der Tabellenname gültig ist
33 table_ok boolean;
34 -- Cursor über alle Tabellennamen in der DB
35 CURSOR name_cursor IS select table_name from tabs;
36 BEGIN
37 -- Initialisierung der "Gültigkeitsvariablen"
38 table_ok:=FALSE;
39 -- Öffnen der Cursors
40 OPEN name_cursor;
41  -- Schleifenbeginn
42 LOOP
43  -- Tabellennamen holen
44  FETCH name_cursor INTO temp_table_name;
45 -- Beenden, wenn der letzte Name verarbeitet wurde
46 EXIT WHEN name_cursor%NOTFOUND;
47  -- Wenn der übergebene Name ok ist...
48  if table_name=temp_table_name OR table_name='RETURN_ALL_TABLES'
49   then
50    -- ...wird dies in der Variablen gespeichert und die Schleife verlassen
51    table_ok:=TRUE;
52    EXIT;
53  end if;
54 END LOOP;
55 -- Cursor schließen
56 CLOSE name_cursor;
57 -- Wenn der Tabellenname nicht in Ordnung ist...
58 if NOT table_ok
59  -- ...Exception auslösen
60  then RAISE e_invalid_table;
61 end if;
62 -- Wenn XSLT benutzt werden soll...
63 if use_xslt
64  -- Dokument mit XSLT-Funktion erstellen
65  then xml_document:=METADATA_AS_OWL_WITH_XSLT(database_name,table_name);
66  -- Sonst die XQuery-Funktion benutzen
67  else xml_document:=METADATA_AS_OWL(database_name,table_name);
68 end if;
69 -- Datei zum Schreiben öffnen
70 filehandle:=UTL_FILE.FOPEN(directory_name,file_name,'W',32767);
71 -- Die zu lesende Zeichenmenge ist zunächst maximal
72 amount := 32767;
73 -- Der Offset ist zunächst am Beginn der Datei
74 offset := 1;
75 -- Das XML-Dokument wird als CLOB gespeichert
76 xml_clob:=XMLType.getClobVal(xml_document);
77 -- Solange mehr gelesen wurde, als der Puffer fasst...
78 WHILE amount >= 32767
79 LOOP
80  -- Nächsten Teil des CLOBs in den Puffer lesen
81  DBMS_LOB.READ(xml_clob, amount, offset,buffer);
82  -- Offset entsprechend der gelesenen Menge anpassen
83  offset := offset + amount;
84  -- Puffer in die Datei schreiben
85  UTL_FILE.PUT(filehandle,buffer);
86  -- Schreibpuffer der Datei leeren
87  UTL_FILE.FFLUSH(filehandle);
88 END LOOP;
89 -- Handle schließen
90 UTL_FILE.FCLOSE(filehandle);
91 -- Hier werden die Ausnahmen behandelt
92 EXCEPTION
93 WHEN e_invalid_table THEN
94  DBMS_OUTPUT.PUT_LINE('Ungültiger Tabellenname');
95 WHEN UTL_FILE.ACCESS_DENIED THEN
96  DBMS_OUTPUT.PUT_LINE('Kein Zugriff auf Datei möglich');
```

```
97  WHEN UTL_FILE.INVALID_FILENAME THEN
98   DBMS_OUTPUT.PUT_LINE('Ungültiger Dateiname');
99  WHEN UTL_FILE.INVALID_PATH THEN
100  DBMS_OUTPUT.PUT_LINE('Ungültiger Systempfad');
101 WHEN UTL_FILE.WRITE_ERROR THEN
102  DBMS_OUTPUT.PUT_LINE('In Datei kann nicht geschrieben werden');
103 WHEN UTL_FILE.FILE_OPEN THEN
104  DBMS_OUTPUT.PUT_LINE('Datei ist bereits geöffnet');
105 -- Hier werden alle anderen Ausnahmen behandelt
106 WHEN OTHERS THEN
107  DBMS_OUTPUT.PUT_LINE('Unbekannter Fehler');
108 END;
```

Listing 7: EXTRACT_METADATA_INTO_FILE()

A.8 Die Prozedur EXTRACT_DATA_INTO_FILE()

```
1   CREATE OR REPLACE PROCEDURE EXTRACT_DATA_INTO_FILE
2   (
3   -- Speicherort der Schemaontologie
4   schema_url string,
5   -- Verzeichnis, in dem gespeichert werden soll. Muss vorher angelegt werden!
6   directory_name string,
7   -- Dateiname, unter dem gespeichert werden soll
8   file_name string,
9   -- Gibt an, ob XSLT benutzt werden soll. Alternativ wird XQuery benutzt
10  use_xslt boolean,
11  -- Tabelle, die zurückgegeben werden soll. Bei fehlendem Argument
12  -- wird der Default-Wert verwendet.
13  table_name string default 'RETURN_ALL_TABLES'
14  )
15  AS
16  -- Variable für die CLOB-Repräsentation des XML-Dokuments
17  xml_clob CLOB;
18  -- Puffer für das Schreiben in die Datei
19  buffer VARCHAR2(32767);
20  -- Speichert, wieviel Zeichen aus dem CLOB gelesen werden
21  amount INTEGER;
22  -- Offset für das CLOB, aus dem gelesen wird
23  offset NUMBER;
24  -- Exception, für den Fall, dass der Tabellenname ungültig ist
25  e_invalid_table EXCEPTION;
26  -- Handle für das Schreiben in die Datei
27  filehandle UTL_FILE.FILE_TYPE;
28  -- Auszugebendes Dokument
29  xml_document XMLType;
30  -- Temporäre Variable für Test, ob der Tabellenname gültig ist
31  temp_table_name varchar(255);
32  -- Gibt an, ob der Tabellenname gültig ist
33  table_ok boolean;
34  -- Cursor über alle Tabellennamen in der DB
35  CURSOR name_cursor IS select table_name from tabs;
36  BEGIN
37  -- Initialisierung der "Gültigkeitsvariablen"
38  table_ok:=FALSE;
39  -- Öffnen der Cursors
40  OPEN name_cursor;
41  -- Schleifenbeginn
42  LOOP
43   -- Tabellennamen holen
44   FETCH name_cursor INTO temp_table_name;
45  -- Beenden, wenn der letzte Name verarbeitet wurde
46  EXIT WHEN name_cursor%NOTFOUND;
47   -- Wenn der übergebene Name ok ist...
48   if table_name=temp_table_name OR table_name='RETURN_ALL_TABLES'
49    then
50     -- ...wird dies in der Variablen gespeichert und die Schleife verlassen
51     table_ok:=true;
52     EXIT;
53   end if;
54  END LOOP;
55  -- Cursor schließen
56  CLOSE name_cursor;
57  -- Wenn der Tabellenname nicht in Ordnung ist...
58  if NOT table_ok
59   -- ...Exception auslösen
60   then RAISE e_invalid_table;
61  end if;
62  -- Wenn XSLT benutzt werden soll..
63  if use_xslt
64   --Dokument mit XSLT-Funktion erstellen
65   then xml_document:=DATA_AS_OWL_WITH_XSLT(schema_url,table_name);
66   -- Sonst die XQuery-Funktion benutzen
67   else xml_document:=DATA_AS_OWL(schema_url,table_name);
68  end if;
69  -- Datei zum Schreiben öffnen
70  filehandle:=UTL_FILE.FOPEN(directory_name,file_name,'W',32767);
```

```
71  -- Die zu lesende Zeichenmenge ist zunächst maximal
72  amount :=32767;
73  -- Der Offset ist zunächst am Beginn der Datei
74  offset := 1;
75  -- Das XML-Dokument wird als CLOB gespeichert
76  xml_clob:=XMLType.getClobVal(xml_document);
77  -- Solange mehr gelesen wurde, als der Puffer fasst...
78  WHILE amount >= 32767
79  LOOP
80   -- Nächsten Teil des CLOBs in den Puffer lesen
81   DBMS_LOB.READ(xml_clob, amount, offset,buffer);
82   -- Offset entsprechend der gelesenen Menge anpassen
83   offset := offset + amount;
84   -- Puffer in die Datei schreiben
85   UTL_FILE.PUT(filehandle,buffer);
86   -- Schreibpuffer leeren
87   UTL_FILE.FFLUSH(filehandle);
88  END LOOP;
89  -- Handle schließen
90  UTL_FILE.FCLOSE(filehandle);
91  -- Hier werden die Ausnahmen behandelt
92  EXCEPTION
93  WHEN e_invalid_table THEN
94   DBMS_OUTPUT.PUT_LINE('Ungültiger Tabellenname');
95  WHEN UTL_FILE.ACCESS_DENIED THEN
96   DBMS_OUTPUT.PUT_LINE('Kein Zugriff auf Datei möglich');
97  WHEN UTL_FILE.INVALID_FILENAME THEN
98   DBMS_OUTPUT.PUT_LINE('Ungültiger Dateiname');
99  WHEN UTL_FILE.INVALID_PATH THEN
100  DBMS_OUTPUT.PUT_LINE('Ungültiger Systempfad');
101 WHEN UTL_FILE.WRITE_ERROR THEN
102  DBMS_OUTPUT.PUT_LINE('In Datei kann nicht geschrieben werden');
103 WHEN UTL_FILE.FILE_OPEN THEN
104  DBMS_OUTPUT.PUT_LINE('Datei ist bereits geöffnet');
105 -- Hier werden alle anderen Ausnahmen behandelt
106 WHEN OTHERS THEN
107  DBMS_OUTPUT.PUT_LINE('Unbekannter Fehler');
108 END;
```

Listing 8: EXTRACT_DATA_INTO_FILE()

B Kurzanleitng für SourceForge

B.1 Introduction: What is this package all about?

The target of this project was to extract relational data and schema information for the Semantic Web from an Oracle 10g database without using any third party programs, i.e. using stored procedures/functions. The representation format of choice is Relational.OWL [PC05a], which allows to extract the schema and data of a relational database into an OWL ontology.

A sample schema export might look like this:

```
<...>
<owl:Class rdf:ID="MYDATABASE">
 <rdf:type rdf:resource=
  "http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#Database" />
 <dbs:hasTable rdf:resource="#COUNTRY" />
</owl:Class>
<owl:Class rdf:ID="COUNTRY">
 <rdf:type rdf:resource=
  "http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#Table" />
 <dbs:hasColumn rdf:resource="#COUNTRY.ID" />
 <dbs:hasColumn rdf:resource="#COUNTRY.NAME" />
 <dbs:isIdentifiedBy>
  <dbs:PrimaryKey>
    <dbs:hasColumn rdf:resource="#COUNTRY.ID" />
  </dbs:PrimaryKey>
 </dbs:isIdentifiedBy>
</owl:Class>
<owl:DatatypeProperty rdf:ID="COUNTRY.NAME">
 <rdf:type rdf:resource=
  "http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#Column" />
 <rdfs:domain rdf:resource="#COUNTRY" />
 <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string" />
 <dbs:length>30</dbs:length>
</owl:DatatypeProperty>
</...>
```

The corresponding data export could look like this:

```
<...>
<dbinst:COUNTRY>
 <dbinst:COUNTRY.ID>1</dbinst:COUNTRY.ID>
 <dbinst:COUNTRY.NAME>Germany</dbinst:COUNTRY.NAME>
</dbinst:COUNTRY>
<dbinst:COUNTRY>
 <dbinst:COUNTRY.ID>2</dbinst:COUNTRY.ID>
 <dbinst:COUNTRY.NAME>Spain</dbinst:COUNTRY.NAME>
</dbinst:COUNTRY>
</...>
```

B.2 Functionality

The first step in representing the database is the extraction of data and schema information as raw XML documents. This is done by the functions `DATA_AS_RAW_XML()` and `METADATA_AS_RAW_XML()`, respectively. These functions are invoked by the functions which provide the actual representation, they are not meant to be invoked manually.

The transformation from raw data into Relational.OWL can either be done using XQuery (`DATA_AS_OWL()`/`METADATA_AS_OWL()`) or using XSLT (`DATA_AS_OWL_WITH_XSLT()`/`METADATA_AS_OWL_WITH_XSLT()`). The transformation method does not affect the result of the extraction. However, XSLT turned out to be a lot faster than XQuery.

All of these functions return an `XMLType` document, which is the type used by the Oracle database system to handle XML documents. Such a document may be processed by other functions and procedures, or it may be stored in the database.

If actual XML files need to be generated, the procedures `EXTRACT_DATA_INTO_FILE()` and `EXTRACT_METADATA_INTO_FILE()` may be used in order to write the result of the extraction into files outside the database system. They rely on the functions mentioned above and provide an user interface to easily access the schema and data representation.

All functions and procedures are written in PL/SQL, and they have been implemented and tested on Oracle 10g Releases 10.2.0.1.0 and 10.2.0.2.0. However, there is no warranty that this package will work on these database systems in all cases. It is highly improbable that the said package will run on any release earlier than Oracle 10g Release 1.

B.3 Installation and Usage

The functions and procedures of this package may be entered into the database by simply invoking the given SQL-DDL statements (`CREATE OR REPLACE FUNCTION ...`), e.g. by using the SQL command line. It is strongly recommended that this is done by a database administrator. Other users would need `CREATE ANY PROCEDURE` authorization. In order to invoke any of the functions/procedures, `EXECUTE ANY PROCEDURE` authorization is required.

B.3.1 Schema export

The functions `METADATA_AS_OWL()` and `METADATA_AS_OWL_WITH_XSLT()` expect the name of the database as first argument. The second argument allows the user to specify the name of a table which shall be extracted. If no second argument is given, the whole schema will be extracted. As mentioned before, the return value is a document of `XMLType`.

The procedure `EXTRACT_METADATA_INTO_FILE()` expects five arguments: The database name, a directory name, a file name, a boolean value and a table name (which, as above, is optional). The directory and file name specify the location and name of the file which is to be created. The boolean value specifies whether XSLT (true) or

XQuery (false) should be used. Note that the directory must be known to the database system, which means it must have been declared using an SQL statement such as `CREATE DIRECTORY name AS 'path'`.

An execution of this procedure from the command line might look like this:

```
exec EXTRACT_METADATA_INTO_FILE('MYDB','DIR','schema.owl',false,'COUNTRY');
```

B.3.2 Data export

The functions `DATA_AS_OWL()` and `DATA_AS_OWL_WITH_XSLT()` expect an URL as first argument which specifies the location of the corresponding schema document. As above, the optional second argument allows to extract only a specific table. The return value is of `XMLType`.

The procedure `EXTRACT_DATA_INTO_FILE()` expects five arguments: The URL of the schema document, a directory name, a file name, a boolean value and an optional table name. It basically works like the corresponding procedure for metadata extraction.

A sample execution:

```
exec EXTRACT_DATA_INTO_FILE('c:\schema.owl','DIR','data.owl',true);
```

Literatur

[BCF+05] BOAG, Scott ; CHAMBERLIN, Don ; FERNÁNDEZ, Mary F. ; FLORESCU, Daniela ; ROBIE, Jonathan ; SIMÉON, Jérome: *XQuery 1.0: An XML Query Language*. http://www.w3.org/TR/xquery/, 2005

[CD99] CLARK, James ; DEROSE, Steve: *XML Path Language (XPath) Version 1.0*. http://www.w3.org/TR/xpath, 1999

[CFR06] CHAMBERLIN, Don ; FLORESCU, Daniela ; ROBIE, Jonathan: *XQuery Update Facility*. http://www.w3.org/TR/xqupdate/, 2006

[Cla99] CLARK, James: *XSL Transformations (XSLT) Version 1.0*. http://www.w3.org/TR/xslt, 1999

[Dra05] DRAKE, Mark: *Oracle Database 10g Release 2 XML DB*. http://download-west.oracle.com/otndocs/tech/xml/xmldb/TWP_XML_DB_10gR2_long.pdf, 2005

[DS04] DEAN, Mike ; SCHREIBER, Guus: *OWL Web Ontology Language Reference*. http://www.w3.org/TR/owl-ref/, 2004

[FP03] FEUERSTEIN, Steven ; PRIBYL, Bill: *Oracle PL/SQL Programmierung*. Köln : O'Reilly Verlag, 2003

[GB04] GUHA, R.V. ; BRICKELY, Dan: *RDF Vocabulary Description Language 1.0: RDF Schema*. http://www.w3.org/TR/rdf-schema/, 2004

[Gru93] GRUBER, Thomas R.: *A Translation Approach to Portable Ontology Specifications*. http://ksl-web.stanford.edu/KSL_Abstracts/KSL-92-71.html, 1993

[Hig03] HIGGINS, Shelley: *Oracle XML DB Developer's Guide*. http://www.stanford.edu/dept/itss/docs/oracle/10g/appdev.101/b10790.pdf, 2003

[IBM06] *DB2 Viper Release Candidate 1 XML Guide*. ftp://ftp.software.ibm.com/software/data/pubs/papers/db2xmlguide.pdf, 2006. – White paper

[Lan04] LANE, Paul: *Oracle Database Globalization Support Guide*. http://www.stanford.edu/dept/itss/docs/oracle/10g/server.101/b10749.pdf, 2004

[LS04] LEHNER, Wolfgang ; SCHÖNING, Harald: *XQuery - Grundlagen und fortgeschrittende Methoden*. Heidelberg : dpunkt.verlag, 2004

[MH04] MILLER, Eric ; HANDLER, Jim: *Web Ontology Language (OWL)*. http://www.w3.org/2004/OWL/, 2004

[MIC06] *What's New in SQL Server 2005*. http://www.microsoft.com/sql/prodinfo/overview/whats-new-in-sqlserver2005.mspx, 2006

[MM05] MELTON, Jim ; MURALIDHAR, Subramanian: *XML Syntax for XQuery 1.0 (XQueryX)*. http://www.w3.org/TR/xqueryx/, 2005

[MSB04] MILLER, Eric ; SWICK, Ralph ; BRICKELY, Dan: *Resource Description Framework (RDF)*. http://www.w3.org/RDF/, 2004

[PC05a] PÉREZ DE LABORDA, Cristian ; CONRAD, Stefan: *Relational.OWL - A Data and Schema Representation Format Based on OWL*. Newcastle, Australia : Australian Computer Society, Inc., 2005

[PC05b] PÉREZ DE LABORDA, Cristian ; CONRAD, Stefan: *Relational.OWL Ontology*. http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl, 2005

[Pér06] PÉREZ DE LABORDA, Cristian: *Relational.OWL*. http://sourceforge.net/projects/relational-owl/, 2006

[Rap03] RAPHAELY, Den: *PL/SQL Packages and Types Reference*. http://download-west.oracle.com/docs/cd/B14117_01/appdev.101/b10802.pdf, 2003

[SQL06] *MySQL 5.1 Reference Manual*. http://dev.mysql.com/doc/refman/5.1/en/index.html, 2006

[Urm02] URMAN, Scott: *PL/SQL-Programmierung*. München : Carl Hanser Verlag, 2002

Abbildungsverzeichnis

Tabellenverzeichnis

Listings