Evaluation of Queries on Linked Distributed XML Data

Dissertation
zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultäten
der Georg-August-Universität zu Göttingen

vorgelegt von
Erik Behrends
aus Itzehoe

Göttingen, 2006

D7
Referent: Prof. Dr. Wolfgang May
Korreferent: Prof. Dr. Jens Grabowski
Tag der mündlichen Prüfung: 18. Dezember 2006

To Gabi

Abstract

XML (eXtensible Markup Language) is the de-facto standard for exchanging information and for representing data in the World Wide Web. In contrast to the document-centric perspective given by the well-known language HTML which defines the human-readable content and the layout of web pages, XML offers more flexibility and expressiveness.

XML documents are not required to be self-contained but may rather have links to other XML resources. For expressing such links between XML documents, the W3C (World Wide Web Consortium) proposed XLink – but mainly for browsing purposes. If the linked documents are considered from the data-centric viewpoint, it shows that XLink does not specify how the referenced instances should be handled. Especially, it is not possible to query along links though the W3C XML Query (XQuery) Requirements explicitly state that this has to be guaranteed.

In order to cope with these issues, an XLink extension "*dbxlink*" has been proposed. It allows for modeling interlinked XML instances as integrated views where XLinks are resolved in a transparent way. In particular, it is possible to query these instances with XPath and XQuery.

In this work, the dbxlink model is described and it is investigated how to query distributed XML instances interlinked with a simple kind of XLinks according to this approach. Different strategies are analyzed and emerging problems like the handling of cyclic instances are treated. It is shown how to extend XPath-based query systems in order to be able to handle queries wrt. dbxlink. Furthermore, optimizing techniques like special caching strategies are proposed. The results of these investigations have been used to conduct a proof-of-concept implementation of the dbxlink approach as an extension to the open source XML database system eXist.

Zusammenfassung

XML (eXtensible Markup Language) ist der de-facto Standard, um im World Wide Web Informationen auszutauschen und Daten zu repräsentieren. Im Gegensatz zu der dokumentenzentrierten Sichtweise der bekannten Sprache HTML, welche den visuell lesbaren Inhalt und das Layout von Webseiten definiert, bietet XML mehr Flexibilität und Ausdruckskraft.

XML-Instanzen müssen nicht notwendigerweise in sich abgeschlossen sein, sondern können Verknüpfungen (Links) zu anderen XML-Quellen enthalten. Um solche Links zwischen XML-Dokumenten beschreiben zu können, hat das W3C (World Wide Web Consortium) XLink spezifiziert – jedoch hauptsächlich zur Anwendung im Browser. Werden die miteinander verbundenen Dokumente vom datenzentrierten Gesichtspunkt betrachtet, so zeigt sich, dass XLink nicht festlegt, wie die verknüpften Dokumente behandelt werden sollen. Insbesondere ist es nicht möglich, die Links bei Anfragen zu berücksichtigen obwohl dies explizit von den W3C XML Query (XQuery) Requirements gefordert wird.

Die XLink-Erweiterung "*dbxlink*" wurde vorgestellt, um diese Probleme zu bewältigen. Sie ermöglicht es, durch Links verbundene XML-Instanzen als integrierte Sichten zu modellieren, in der die XLinks auf transparente Art und Weise verarbeitet werden. Dadurch ist insbesondere das Beantworten von XPath- und XQuery-Anfragen auf den miteinander verbundenen Dokumenten möglich.

In dieser Arbeit wird zunächst das dbxlink-Modell beschrieben und es wird erläutert, wie Anfragen an XML-Instanzen (die durch eine einfachen Sorte von XLinks miteinander verbunden sind) mit diesem Ansatz beantwortet werden können. Verschiedene Strategien werden untersucht und dabei entstehende Probleme wie z.B. der Umgang mit zyklischen Instanzen werden behandelt. Es wird gezeigt, wie XPath-basierte Systeme erweitert werden können, um Anfragen gemäß dbxlink beantworten zu können. Weiterhin werden Methoden zur Optimierung wie z.B. spezielle Caching-Strategien vorgestellt. Die Ergebnisse dieser Untersuchungen wurden dazu genutzt, einen Konzeptnachweis in Form einer Implementierung des dbxlink Ansatzes als Erweiterung des Open-Source XML Datenbanksystems eXist durchzuführen.

Acknowledgements

First of all, I would like to thank Prof. Dr. Wolfgang May for giving me the possibility to be part of his research group and to conduct a dissertation in the LinXIS project under his supervision. Having always time for discussions and for the questions that came up during the compilation of this thesis, he provided the necessary background in both technical and practical issues related to this work. Also, I thank all colleagues at the Institute for Informatics of the Göttingen University for the interesting time and for the fruitful discussions. Finally, I would like to thank my parents, my wife Gabi and all friends in Göttingen.

Table of Contents

List of Figures

1 Introduction

Today, more than fifteen years after its invention, the *World Wide Web (WWW, or simply "Web")* can be considered as one of the most influencing innovations of the last century whose significant impacts can not be measured. It is omnipresent and has become indispensable as communication platform and for exchanging information in private, economic, social, political and research areas.

Most users perceive the Web from the *document-centric* perspective: usually, they are interested in the *content* of a web page (or, more generally, a resource) which is *displayed* to them in *human-readable* form in a browser. Until today, most web pages have been specified with *HTML* (*Hypertext Markup Language*) [HTM99]. For instance, consider the following HTML document:

```
<html>
 <head>
   <title>Example</title>
 </head>
 <body>
   <h1>Just an example for browsing</h1>
   Please click <a href="http://www.example.org/index.html">here</a>
 </body>
</html>
```

This simple document defines *both* the layout and the content of a web page and is almost self-describing by its hierarchical structure consisting of nested *tags* (e.g. <html>...</html>). Here, it specifies that it should be displayed with a certain title (given as "<title>" element) and its main part ("<body>") consists of some text. The text contains a heading which is given between the "<h1>" tags and some ordinary text ("Please click here") which embeds a hyperlink represented by an element "<a>". This anchor element will be rendered by browsers as a "clickable" character string ("here"). While *browsing* the Web, users navigate from one web page to another by clicking on such hyperlinks. If the link in the example document is activated (i.e. "clicked") by a user, the currently displayed document is replaced in the browser with the resource located at the *URI* (*Uniform Resource Identifier*) "http://www.example.org/index.html" as specified in the anchor's "href" reference attribute. Figure 1.1 shows the example document displayed in a browser with the emphasized link that points to the referenced resource which replaces the current document if the link is activated by the user. Any user of the Web is familiar with these "explicit" hyperlinks while there exist also "implicit" links which are activated automatically by browsers, e.g. for embedding images to be displayed as part of a web page.

In contrast to this *document-centric* viewpoint of the Web, it can be considered in a *data-centric* way where it is a network of interconnected resources providing arbitrary *information* or *data*. Besides web pages defined in HTML, various kinds of resources can be found in the Web, e.g. multimedia files (movies, games, MP3 songs, etc.), printable documents (for instance given as plain text or in PDF) or arbitrary binary files (executable programs, CD images for Linux distributions, etc.). Especially, in some cases data has to be *exchanged* between resources without human interaction. For instance, a retailer might use *electronic data interchange (EDI)* for sending orders to a manufacturer who could return an electronic bill. In order to be able to exchange data electronically, it has to be *represented* in an agreed format. The restricted constructs of HTML are not sufficient to achieve this and this is one reason why *XML* (*eXtensible Markup Language*) [XML06] has been defined by the *World Wide Web Consortium (W3C)* [W3C].

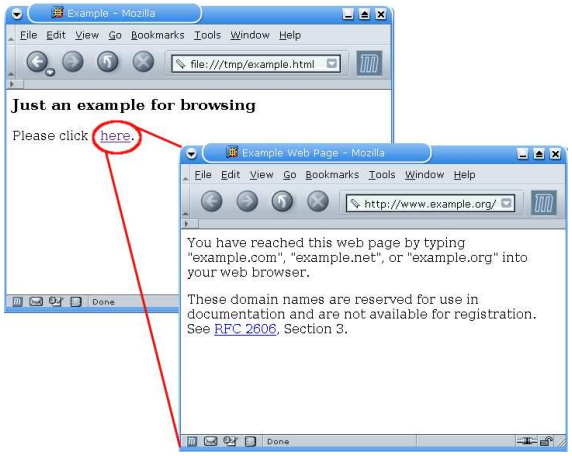


Figure 1.1: Browsing the Web

Since it has been proposed in 1996, XML is increasingly used as a data format for storing or representing information and it is the de-facto standard language for exchanging information in the Web. For instance, XML can be applied in the area of news feeds and Web Services. In fact, being a meta language, many special purpose languages are defined with XML. To illustrate this, consider *XHTML* [XHT00], the successor of HTML which is an application of XML. Thus, any document based on XHTML *is* an XML document. Still, if represented in ASCII format, XML documents are human-readable but the main purpose of XML is to define formats to be used for exchanging information in the Web. From this data-centric viewpoint, the Web can be considered as a huge database containing big amounts of (XML) *data* resources.

Similar to HTML documents, these XML resources might not be necessarily self-contained but rather may have links to remote XML data possibly residing on other servers. In order to express links between XML documents, the W3C proposed the *XML Linking Language* (*XLink*) [XLi01]. Figure 1.2 depicts an excerpt of the distributed version of the MONDIAL XML database [Mon01]. One document contains information

about all countries where each country defines e.g. a link to its capital which can be found in another XML document, namely in the one that contains all cities for a country.

The XLink specification is more flexible than the hyperlink concept of HTML. For instance, any element can be defined as a link (which in the XML context are sometimes also called XLinks): in Figure 1.2, the capital element has an xlink:href attribute and is thus given as a link. In this case, it is a simple kind of XLink defining a reference to a fragment of an XML resource. This is achieved by the xlink:href attribute which contains a URI that precisely defines a server location, a path to a specific XML resource on that server and a fragment identifier that contains an *XPointer* [XPt03b] for addressing the desired document parts. Consider again the capital element:

<capital xlink:href="http://linxis03/cities-D.xml#xpointer(/cities/city[name='Berlin'])"/>

It defines a reference to the document "cities-D.xml" which contains information about German cities and which can be found on the server "linxis03" in the local network. Then, the XPointer "/cities/city[name='Berlin']" defines that inside the XML document cities-D.xml the city element representing Berlin should be selected as referenced target.

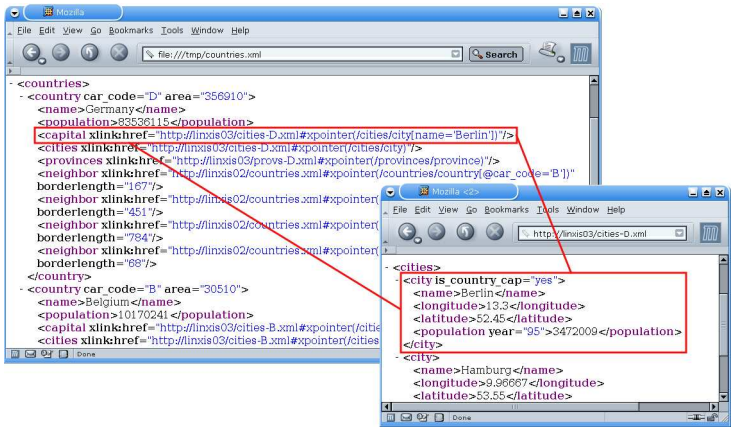


Figure 1.2: XML Documents with XLinks

In Figure 1.2, the XML documents are displayed in browsers in order to show their content in ASCII representation. However, it is obvious that there is no rendering of the content apart from a hierarchical structure and appropriate indentation. Here, in case of the MONDIAL documents, abstract information about countries is given without a specification how it should be displayed to users. Instead, this data could be processed by arbitrary applications. Especially, XML data can be queried like a database.

Considering links in HTML, they are used to navigate from one document to another. In contrast to this, for XML the situation is different. In case of the example given in Figure 1.2, if country data is interlinked with city data by XLinks, applications could use the link e.g. for building integrated views on the distributed data. Unfortunately, if a set of XML resources connected by XLinks is seen from a *data-centric* viewpoint, the XLink specification does not state *how* links should be handled.

Mapping of Interlinked XML Instances

When resolving such an XLink, there are several possibilities how the referenced XML fragment should be mapped into the referencing document. Besides others, the target fragment (in the example given above, the city element representing Berlin) could replace the XLink (the referencing capital element) or it could be appended to it as a child.

Consider Figure 1.3 for illustration. In the upper half, the document on the left (outer triangle) contains a link (inner triangle) that references a specific part of another document depicted as separate triangle on the upper right. Below, two possible mappings are sketched: on the lower left, the referenced part is embedded into the link element and on the lower right the whole link has been replaced.

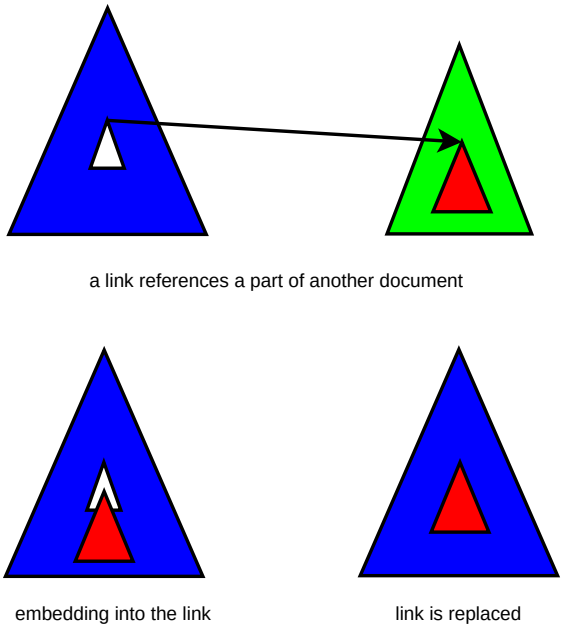


Figure 1.3: Different Mapping Options for a Referenced Fragment

In order to describe various useful mapping possibilities systematically, a logical model has been proposed in [May02, MM03, BFM06a]. It specifies how XML fragments referenced by XLinks can be embedded into the referencing resource and has been defined as an extension to XLink called "*dbxlink*".

Querying Interlinked XML Instances

With the dbxlink model, interlinked XML sources induce a *virtual instance* (cf. Chapter 3) or a *view* on the data. When considering XML *data*, the question arises, how this view can be queried. The XML Query (XQuery) Requirements [XMQ04] stated by the W3C include the handling of links (cf. [XMQ04], Sections 3.3.4 and 3.4.12):

"The XML Query Data Model MUST include support for references, including both references within an XML document and references from one XML document to another."

"Queries MUST be able to traverse intra- and inter-document references."

Each requirement has a corresponding status. While this thesis has been written, the status for both of the above mentioned requirements was described as follows: "*this requirement has been partially met*". Detailed investigations, as exposed in Section 4.1.1, showed that with the W3C language XQuery [XQu06], which is most likely to become the standard XML query language, it is *not* possible to query along link references even with an explicit link dereferencing operator which could be given as user-defined function. To overcome this limitation was another motivation to introduce the logical model.

While querying instances that are mapped according to this model, XLinks are resolved *transparently*. The XLink elements are seen as view definitions that integrate the referenced XML data within the referencing instance (where the XLink element specifies the referenced nodes, and how they are mapped into the surrounding instance). This virtual instance can then be processed by standard XML query and processing languages like XPath, XQuery and XSLT as depicted in Figure 1.4. Additionally, it follows that no explicit dereferencing operator or function is required.

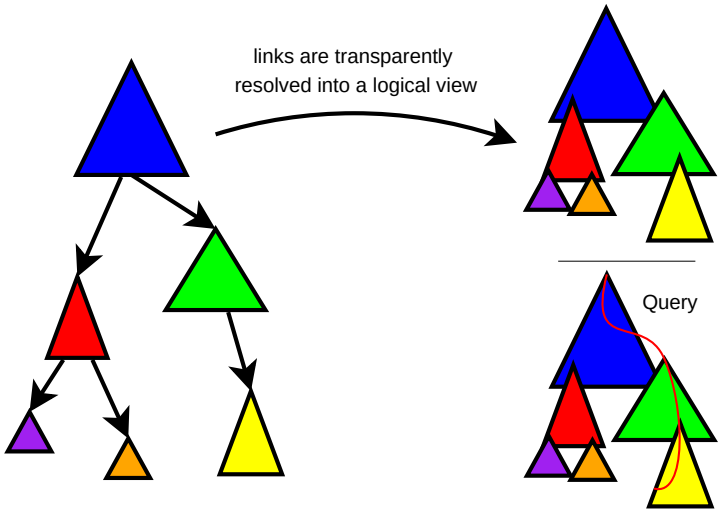


Figure 1.4: Querying the Logical View in a Transparent Way

Proof-of-Concept Implementation

The dbxlink functionality described in this thesis has been implemented as an extension to the Java-based XML database system *eXist* [exi]. It is an open source project with an active development team and it has received the *2006 Technology of the Year Awards* of the San Francisco based magazine *InfoWorld*[1] in the category "*Best XML Database*".

[1] `http://www.infoworld.com/`

The following characteristics were significant for choosing eXist as a basis for a proof-of-concept implementation.

- Open source: eXist is an open source project and thus, all modifications and extensions can be integrated based on the methods described in this thesis.
- Conformance of standards: besides XML, eXist supports the standard query languages XPath and XQuery.
- Web access: eXist offers different networking interfaces allowing for accessing whole documents or for stating queries (in XPath and XQuery) including *HTTP* [HTT99] and *SOAP*. Thus, eXist servers extended with dbxlink functionality will be able to communicate which each other and with any server on the Web allowing for setting up an appropriate testbed.

Testbed and Demonstrator

For testing the functionality and experimenting with different strategies, a network of dbxlink-enabled eXist servers on different hosts is used. The main demonstrator is based on a distributed version of the MONDIAL database [Mon01] as illustrated in Figure 1.5. The distributed scenario can be queried via a public interface which is reachable via `http://www.dbis.informatik.uni-goettingen.de/linxis/`.

- mondial-root.xml (serves as a root that contains only a mondial element with links to countries, organizations, and geographical objects). As an entry point, its external schema coincides with the one of the non-distributed Mondial database.
- countries.xml (all countries) and continents.xml (all continents)
- cities-XX.xml (cities for each country, where XX is the car code of the respective country)
- organizations.xml (organizations)
- geo.xml (mountains, waters etc.)

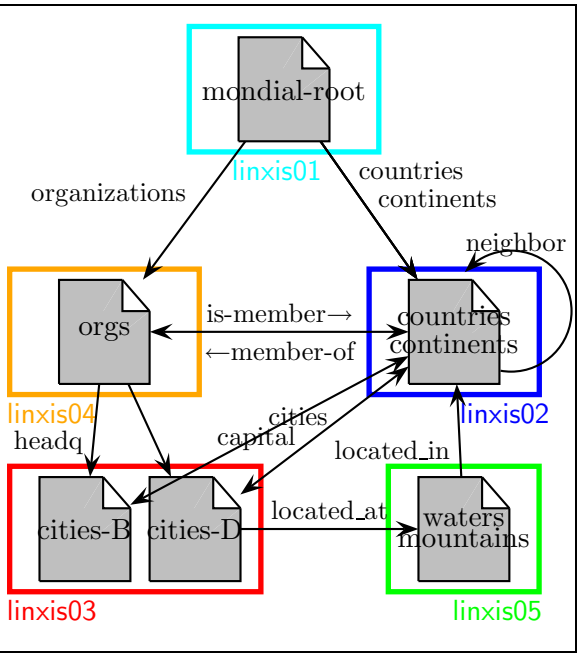


Figure 1.5: Distribution of the Mondial Database over Several Hosts

Applications for dbxlink

The dbxlink approach can be used both for building a distributed database by splitting an XML document, and for building (virtual) XML documents by combining autonomous sources which can then be queried by XPath/XQuery:

(i) distribution of existing data over several instances, and

(ii) integration of autonomous sources according to the *virtual* approach (i.e., not materializing them, but defining a global view).

(i) When XML documents grow, it is sometimes preferable or necessary to split them over several documents or even servers. For instance, the distributed Mondial instance used in the testbed has been created from the non-distributed one [Mon01].

Often, in case of a distribution of data, the original schema should be kept as *external* schema which is a *virtual* schema that provides the user with a view over the actual data. Here, in the data splitting scenario, all queries against this view still yield the same answers as before. From the data *integration* point of view, the logical model can be seen as a *Global as View (GAV)* [Len02] view over the –now distributed– data.

(ii) An integrated view over distributed, autonomous data can be defined according to a given target schema. In this case, the integration approach is also realized by the GAV approach, i.e., queries are answered by *view unfolding* which in this case amounts to integrating the data referenced by an XLink into the surrounding structure.

Also, calls to Web Services, data requests via the XML interfaces of database systems and XHTML sources can be integrated via XLink, as depicted in Figure 1.6. Thus, for a local XML database with dbxlink functionality, XLinks can be used to reference arbitrary remote XML data which is then included during the evaluation of queries.

Summary of Contributions

- A refinement of the proposal in [May02] for mapping linked XML sources to an integrated view based on an extension to XLink (cf. Chapter 3 and [BFM06a]).
- A transparent and flexible mechanism to query interlinked XML instances according to the proposed model is described in an abstract way. It allows for integrating dbxlink functionality in XML query systems which rely on the standard stepwise evaluation strategy for XPath, including
  - the handling of cyclic instances during query evaluation (cf. Chapters 4),
  - the analysis of several query shipping strategies (cf. Chapters 5 and 6), and
  - optimization and caching strategies for query processing over distributed interlinked XML sources (cf. Chapter 7).
- A proof-of-concept implementation of the functionality necessary for querying over XLinks according to the W3C XML Query (XQuery) Requirements using the dbxlink approach as an extension to eXist (cf. Chapter 5).

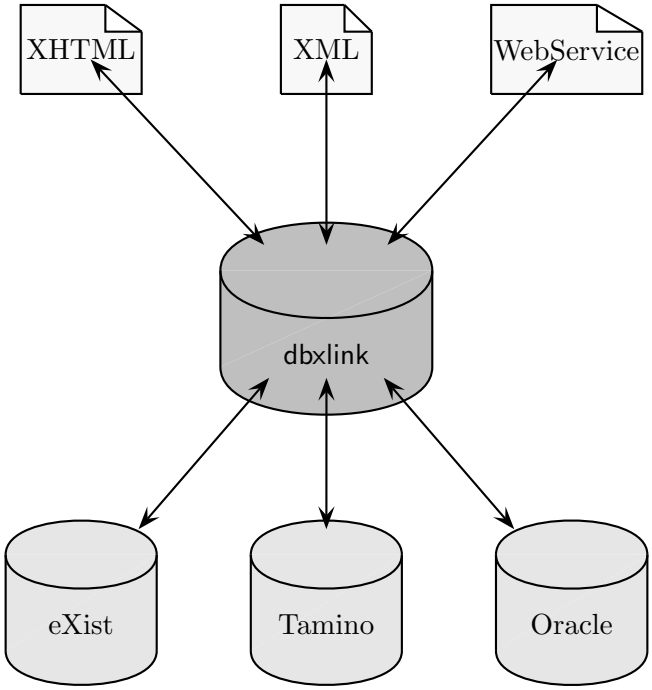


Figure 1.6: Integrating Heterogeneous Data Sources

Outline of this Thesis

This thesis is structured as follows. In order to equip the reader with the notions and concepts that build the basis to understand this work, some preliminaries about XML and relevant accompanying standards are given in Chapter 2. Then, the logical model on which the investigations contained in this thesis are based is discussed in Chapter 3. The following chapters contain the main contributions of this dissertation. Chapter 4 describes how to query XLink-ed XML sources. Emerging problems like querying cyclic instances are examined and according strategies how to cope with them are proposed. Then, Chapter 5 discusses the proposed querying facilities in detail and the proof-of-concept implementation is outlined. Many special issues arise when investigating query shipping, as shown in Chapter 6. Optimization techniques like caching are discussed in Chapter 7. In Chapter 8 related work is discussed and an outlook on further work is given. The final chapter concludes this dissertation.

2 XML Preliminaries

Since its advent in 1996, XML has become ubiquitous in the Web. For instance, XML is used as a data format for electronic data interchange (EDI) in business-to-business applications, as communication protocol for web services (namely in terms of the SOAP [SOA03] specification), for representing news feeds (cf. RSS 0.91 [RSS99] or Atom [Ato05]) and for defining the markup language XHTML [XHT00], the successor of the well-known *Hypertext Markup Language (HTML)* [HTM99].

This thesis investigates the evaluation of queries on linked distributed XML data. Thus, in this chapter, an informal introduction to XML is given in Section 2.1. Additionally, as *linked* XML data shall be *queried*, we then consider the standard XML query languages XPath and XQuery in Section 2.2, and in Section 2.3 we discuss the linking mechanisms of the XML Linking Language (XLink) which in turn is based on XPointer.

2.1 XML

XML is an acronym for *eXtensible Markup Language* [XML06] and has been first proposed as a working draft by the *World Wide Web Consortium (W3C)* [W3C] in November 1996. In 2006, the XML recommendation has reached its fourth edition. As indicated by its name, XML could be considered as a *markup language*, i.e. a language for specifying the *layout* of documents like HTML (by using *optical markup*) or to define its *logical structure* (cf. LaTeX). This viewpoint is, however, not suitable for XML because XML is a *meta* markup language that can be used to *define* special purpose markup languages that can then serve for various applications.

The original goal for designing XML was "*to meet the challenges of large-scale electronic publishing*"[1]. These requirements could have already been achieved by *SGML (Standardized Markup Language)* [SGM86], a meta language that evolved out of IBM's *Generalized Markup Language (GML)* and then became an ISO standard in 1986 (ISO 8879). Before XML has been proposed, SGML has been widely used for information processing and electronic data interchange, mainly in printing and publishing industries. It is a very generic and powerful language but has been considered as being too complex for the daily use in the Web.

On the other hand, the *Hypertext Markup Language (HTML)* [HTM99] has been extensively used in the Web in several versions since the early 1990s for publishing hypertext documents (i.e. documents containing hyperlinks enabling users to navigate to other documents). HTML has been specified as an SGML application with a fixed

[1] `http://www.w3.org/XML/`

set of markup elements. It is easy to use but its limited vocabulary restricts it to being only suitable for defining the optical layout of web pages.

Considering SGML and HTML, it is obvious that a language bridging between these two well-established technologies would have been very useful for the Web. As a consequence, XML has been derived from SGML in order to keep its flexibility and expressiveness while being simplified for achieving a similar straightforwardness as offered by HTML. In fact, XML is a subset of SGML while HTML's successor XHTML has been defined using XML, i.e. XHTML is an XML application. Thus, XML can be considered as a language situated "between" HTML and SGML.

Today, XML is the de-facto standard for exchanging data on the Web. Increasingly, it has also been used as a semistructured data model for representing "mixed" data.

2.1.1 Semistructured Data

Until today, in 2006, in both research and industry areas the most common way to store, manipulate and query data is still given by relational databases with the well-known SQL language. They are based on the *relational model* [Cod70], a well-founded data model for storing *structured* data[2]. This rigid model requires that a schema has to be given a priori and that this schema has to be satisfied by the data to be stored. Many database systems implement the relational model, among them commercial products like the databases offered by IBM, Microsoft and Oracle, and open source projects like MySQL and PostgreSQL.

In contrast to that structured kind of data, data on the Web often has an *irregular* structure. Consider for example a web site offering world news. Its content (and also the structure of the data) might change several times per hour and besides headlines and text sections it might also contain images in different parts. Sometimes, data on the Web is also *self-describing*, e.g. for a news page there might be special labels (*tags*) indicating headlines and images. Thus, for data on the Web, meta data is often contained in the data itself.

In general, these characteristics also apply to *semi-structured* data [QRS+95], a notion that intuitively covers the range between structured data (e.g. data in relational databases) and unstructured data (like raw text) while it is self-describing. Though XML has originally not been defined as a language for representing semistructured data, it can be considered as such and research on semistructured data has influenced the further development of XML and its related standards in the late 1990s.

2.1.2 Components of XML Documents

In order to give a first understanding of XML, it is useful to recall the syntax of HTML, because the ASCII representations of both XML and HTML documents are based on *elements* enclosed by *start* and *end tags*, possibly containing *attributes* and *text* content. The nesting of elements induces a hierarchical structure.

[2] There also exist other data models for storing structured data that have never reached the same acceptance as the relational model, e.g. the network model and the object oriented data model.

Example 2.1 (HTML Document)
The nested, hierarchical structure of HTML documents illustrated by a simple example:

```
<html>
 <head>
   <title>Example HTML page</title>
 </head>
 <body>
   <h1>This is a headline</h1>
   A list follows:
   <ul type="square">
     <li>First item,</li>
     <li>last item.</li>
   </ul>
 </body>
</html>
```

The outermost element (here: html*) is called the* root *element of this document. Like all other elements, it is enclosed by its start tag (*<html>*) and its end tag (*</html>*). Elements can be nested inside each other. If an element* e *is directly embedded between the start and end tag of another element* p*, then* e *is called a* child *of* p *which in turn is the* parent *of* e *(note that all elements except for the root element have exactly one parent). For instance, the* title *element is a child of the* head *element. The root element (*html*) is the parent of both* head *and* body*. The element* ul *is equipped with an attribute (*type*) indicating that the list items should be dashed with a square. Thus, it shows that HTML is an optical markup language for defining the layout of web pages.*

In order to give an introduction to XML, we will first start to discuss the ASCII representation of XML documents. Please note that this is just one possible *representation* format for XML *data* (cf. Section 2.1.5). However, XML's ASCII format is the most widely used representation, especially for electronic data exchange. In the remainder of this work, most examples will be based on this format.

Well-formed XML Documents. An XML document has to be *well-formed*. This means that it must have a *document prolog* and at least one element, and it must meet several constraints like the fact that start and end tags of different element must not be interleaved. In addition to this required property of being well-formed, an XML document may be *valid* (this notion will be discussed in Section 2.1.3). We now continue with discussing the structural parts of XML documents.

Document Prolog. The XML document prolog usually consists of the XML declaration and an optional document type declaration (DTD, cf. Section 2.1.3). Omitting the DTD by now, we just consider the XML declaration:

```
<?xml version="1.0" encoding="utf-8"?>
```

The XML declaration shown above specifies the XML version being used[3] and the encoding of the document (here, "utf-8", the 8-bit Unicode Transformation Format[4]).

Elements. The main components of XML documents are elements. A well-formed XML document has to contain at least one element, namely the *root* element which is always unique in the whole document. Elements are labelled by a *tag name* (often referred to as the *element name*). It is part of its start and end tag which enclose the *element content*:

<elem_name> content</elem_name>
start tag end tag

The content of an element may consist of other elements which are called *children*. As these may also have children, we refer to all elements found in the content of an element as its *subelements*. As a consequence, all elements are naturally subelements of the root element. The content of an element may also contain text which may be mixed with the elements. An element may have attributes (they are discussed below) which are given inside the element's start tag. Elements that don't have any children or text content are called *empty elements* and are denoted without end tag while the start tag closes with a slash (/). An empty element with n attributes could look like this:

<elem_name $attr_1$="$value_1$" ... $attr_n$="$value_n$"/>

Attributes. All elements may have attributes which are defined inside the element's start tag. Attributes are key-value pairs having an attribute name (key) and an attribute value. If an element has more than one attribute, the attributes must have different attribute names. An attribute is specified as follows: attr_name="attr_value". The attribute value is thus enclosed in (single or double) quotes.

Comments. Sometimes, it is useful to add comments to an XML document. Comments may appear anywhere in an XML document, except for element tags or other markup. They may contain arbitrary text, but for compatibility reasons, "--" is not allowed. Like in HTML, an XML comment is always encapsulated by "<!--" and "-->":

<!-- Here, we can enter arbitrary text as comment ...-->

Remark. XML also consists of several further components like processing instructions, entities, and CDATA sections. These are not considered in this work. Please refer to the W3C XML recommendation [XML06] for further information.

[3] Besides XML 1.0 there also exists a newer version (1.1) that is not considered in this work because it adds only technical details to its more widely used predecessor.

[4] `http://www.unicode.org`

Example 2.2 (XML Document)
The MONDIAL *database [Mon01] contains geographical data including information about countries, their provinces and cities, organizations and geographical entities like mountains, seas, rivers and lakes. In Figure 2.1, an excerpt of its XML version is depicted. It starts with the usual XML declaration followed by a doctype declaration which references a DTD (cf. the next section). Then, the root element* mondial *follows. It has children that represent two countries, Belgium and Germany. For each country, there is a corresponding element having attributes for its car code, area and capital. Countries have subelements for the name, population, borders with other countries (including the border length) and provinces, that may also have city children.* MONDIAL *will serve as an example throughout this work.*

In order to give an intermediate summary, we can state that XML documents in the ASCII representation have to be well-formed, i.e. they have to start with a document prolog followed by the mandatory root element that contains all other elements. Elements may have content consisting of text and properly nested subelements, and they also may have attributes.

2.1.3 DTD

A *document type definition (DTD)* specifies further constraints on XML documents. In this section, we briefly explain the syntax of DTDs.

Valid XML Documents. As stated in the preceding section, all XML documents have to be well-formed. Additionally, they may be *valid*. For that, an XML document has to be associated with a DTD which defines constraints that have to be satisfied by the document. For a specific XML document, a DTD is always supplied as a *document type declaration* that has to be given in the document prolog, directly after the XML declaration:

<!DOCTYPE *name dtd_spec*>

The doctype declaration used for the MONDIAL excerpt depicted in Figure 2.1 references a DTD that is stored as "mondial.dtd" in the local file system. A DTD describes the logical structure of XML documents and is defined in terms of a grammar. It may contain four types of declarations: element type, attribute-list, entity and notation declarations. For this work, only element type and attribute-list declarations are relevant and will thus be discussed briefly.

Element type declarations. For all elements that occur in an XML document of a certain document type, the corresponding DTD must define element types:

<!ELEMENT *element-type-name element-content*>

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE mondial SYSTEM "mondial.dtd">
<mondial>
  <country car_code="B" area="30510" capital="cty-Belgium-Brussels">
    <name>Belgium</name>
    <population>10170241</population>
    <border country="D" length="167" />
    <province id="prov-Belgium-Antwerp" capital="cty-Belgium-Antwerp">
      <name>Antwerp</name>
      <population>1610695</population>
      <city id="cty-Belgium-Antwerp">
        <name>Antwerp</name>
        <population year="95">459072</population>
      </city>
    </province>
    <province id="prov-Belgium-Brabant" capital="cty-Belgium-Brussels">
      <name>Brabant</name>
      <population>2253794</population>
      <city id="cty-Belgium-Brussels">
        <name>Brussels</name>
        <population year="95">951580</population>
      </city>
    </province>
  </country>
  <country car_code="D" area="356910" capital="cty-Germany-Berlin">
    <name>Germany</name>
    <population>83536115</population>
    <border country="B" length="167" />
    <province id="prov-Germany-Berlin" capital="cty-Germany-Berlin">
      <name>Berlin</name>
      <population>3472009</population>
      <city id="cty-Germany-Berlin">
        <name>Berlin</name>
        <population year="95">3472009</population>
      </city>
    </province>
    <province id="prov-Germany-Hamburg" capital="cty-Germany-Hamburg">
      <name>Hamburg</name>
      <population>1705872</population>
      <city id="cty-Germany-Hamburg">
        <name>Hamburg</name>
        <population year="95">1705872</population>
      </city>
    </province>
  </country>
  ⋮
</mondial>
```

Figure 2.1: Excerpt of the MONDIAL XML Database

An element type has a name which is also the name of the element instances of that type in an XML document satisfying the DTD. The content of an element type can be defined using one of the following options:

- EMPTY
  This keyword denotes that the element is empty, i.e. it may have attributes but no children or text content.

- (*contentmodel*)
  The *contentmodel* can consist of sequences and choices of children which can be combined by using a syntax similar to regular expressions with the well-known operators "*", "+" and "?". Children listed in sequences are delimited with "," while choices use the "|" character. Example 2.3 illustrates the usage of these constructs. All element types contained in sequences or choices must be declared somewhere in the DTD as appropriate element types.

- Mixed element types
  The keyword "#PCDATA" is used in order to indicate that arbitrary text content is allowed. For mixed-content declarations, character data may be combined with any number of subelements using the choice operator ("|") and hence, in case that subelements are given, the outer asterisk ("*") is mandatory:

  <!ELEMENT mixed_elem (#PCDATA|subelem|next)*>.

  If for an element only text content should be allowed, the following syntax has to be used:

  <!ELEMENT text_elem (#PCDATA)>.

- ANY
  Any element content is allowed:

  <!ELEMENT any_elem ANY>.

Example 2.3 (DTD with Intertwined Sequences and Choices)
A more complex example for an element having content composed of a sequence and a choice of children is given by this DTD fragment:

<!ELEMENT complex (a,(b|c)*,d?)+>
<!ELEMENT a EMPTY>
<!ELEMENT b EMPTY>
<!ELEMENT c EMPTY>
<!ELEMENT d EMPTY>

An element of type complex *must have at least one sequence of child elements which is indicated by "+". Each such sequence must consist of exactly one child element "*a*", arbitrary ("*") many choices of "*b*" or "*c*" elements and an optional element "*d*" (indicated by "?"). Below, we show an appropriate element that has children consisting of three sequences:*

```
<complex>
    <!-- first sequence (each element occurs once): -->
    <a/><b/><c/><d/>

    <!-- second sequence with several b and c elements, no d element: -->
    <a/><b/><c/><c/><c/><b/>

    <!-- third sequence (only the mandatory a): -->
    <a/>
</complex>
```

All kinds of element types, including empty elements, may have attributes. These are declared as corresponding attribute-lists.

Attribute-list declarations. In order to declare the attribute-list for an element type in a DTD, the following syntax is used:

<!ATTLIST *element-type-name* *attribute-name*$_1$ *attribute-type*$_1$ *attribute-qualifier*$_1$
⋮
attribute-name$_n$ *attribute-type*$_n$ *attribute-qualifier*$_n$>

For a specific element type (given by *element-type-name*), a list of arbitrary many attributes may be specified. Each attribute has a name, a type and a qualifier. Possible attribute types are[5]:

- CDATA (character data, i.e. text content),
- NMTOKEN (name token, i.e. restricted characters without whitespaces),
- NMTOKENS (one or more NMTOKEN values separated by whitespaces),
- ID (identifying value, unique for the document (at most one ID per element)),
- IDREF (a value referencing an ID attribute given somewhere in the same document),
- IDREFS (one ore more IDREFs separated by whitespaces).

It is also possible to use enumerations for attribute types. In that case, instead of using one of the attribute types given above, the type is explicitly defined as a choice of possible values. For instance, if an attribute type is given as

[5] The attribute types ENTITY, ENTITIES and NOTATION are not relevant for this work.

(value$_1$|value$_2$|value$_3$),

it follows that for attributes of this type only these three values are allowed.

The qualifier corresponding to an attribute type declaration can be given using the following options:

- #REQUIRED (the attribute is mandatory for all element instances of this type),
- #IMPLIED (indicates an optional attribute),
- #FIXED *"value"* (here, a fixed attribute value that must fit to the attribute type is supplied; if the #FIXED directive is omitted, *"value"* is considered as default value[6]).

Example 2.4 (DTD)
For the excerpt of MONDIAL *shown in Figure 2.1, the DTD could have the following form. Please note that* MONDIAL*'s original DTD (cf. [Mon01]) is more complex.*

```
<!ELEMENT mondial (country*)>
<!ELEMENT country (name, population?, border*, province*)>
<!ELEMENT province (name, population?, city*)>
<!ELEMENT city (name, population?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT population (#PCDATA)>
<!ELEMENT border EMPTY>
<!ATTLIST country car_code ID #REQUIRED
                  area CDATA #IMPLIED
                  capital IDREF #IMPLIED>
<!ATTLIST province id ID #REQUIRED
                   capital IDREF #REQUIRED>
<!ATTLIST city id ID #REQUIRED>
<!ATTLIST border country IDREF #REQUIRED
                 length CDATA #REQUIRED>
```

The root element mondial *can have arbitrary many* country *children. These must have an* ID *attribute* car_code *while the* area *and* capital *attributes are optional. Countries again must have exactly one* name *child, which is defined as* PCDATA*. Note that one limitation of DTDs is that element types are defined globally and thus, only one* name *element type for countries, provinces and cities can be specified. Countries, provinces and cities have an optional* population *child which is also given as* PCDATA*. DTDs don't distinguish numerical types which would be useful here. Countries have arbitrarily many provinces which in turn can have arbitrarily many cities. The elements of type* border

[6] Note that for enumerations the default value must be contained in the defining choice.

can occur as children of country *elements and are given as empty elements with mandatory attributes for indicating the neighboring country (*IDREF*) and the border length. Unfortunately,* IDREF *can not be restricted to refer to specific element types. Here, it would be appropriate to indicate that only* ID*s of countries are allowed in* border *elements. Finally, also provinces and cities have required* ID *attributes while* city *elements have to be nested inside* province *elements.*

Remark. Besides DTDs, which are rather limited in expressiveness, there are other technologies for specifying the schema of an XML document, e.g. XML Schema [XML04b] and Relax NG [REL01] (which are, however, not relevant for this work).

2.1.4 Namespaces

In case that several XML documents have to be processed in the same application area, it is possible that naming conflicts arise. This may happen if the same name is used for different element types. For instance, the element type name can occur as the name of persons, having subelements first and last, and also as the name of countries which is defined as #PCDATA. In order to be able to distinguish these different concepts, the W3C defined *namespaces* [Nam06].

A namespace is specified by a URI and, as a consequence, it is globally unique. Thus, when using namespaces for specific elements, they can be distinguished from all other elements having a different namespace. A namespace is declared in the start tag of an element by assigning a *namespace prefix* to it. Also, syntactically, the directive "xmlns" has to be given, as illustrated in the following example for the MONDIAL database where a namespace with prefix "mon" is defined:

```
<mondial xmlns:mon="http://dbis.informatik.uni-goettingen.de/Mondial">
    ⋮
</mondial>
```

Then, in order to use this namespace for a specific element, the namespace prefix (in this case "mon") is prepended to the element's name in both its start and end tag:

```
<mon:country car_code="D" area="356910" capital="cty-Germany-Berlin">
    <name>Germany</name>
    <population>83536115</population>
       ⋮
</mon:country>
```

The namespace context is valid for the whole subtree defined by the element that is associated with a namespace. Thus, all descending children inherit the namespace, i.e., in the above example, the name, population and all other subelements of Germany are labeled with the namespace "mon" implicitly.

2.1.5 XML Data Models

In the preceding sections, XML has been introduced using its ASCII representation in order to show how XML documents can be defined. The syntactic structure of XML documents and their corresponding DTDs has been explained. When considering XML *documents* from the data-centric viewpoint, usually the term XML *instance* is used. As we will mainly investigate XML data, the notions of XML *documents* and *instances* will be used synonymously in the remainder of this work. While XML data can be *represented* and *serialized* in the above mentioned ASCII format, the general XML *data model*, however, is tree-based using the same notions of elements, attributes, etc. In the following paragraphs, several XML data models are discussed.

XML as Tree-Based, Ordered Data Model. Because of their nested element structure, XML instances can be considered as *trees*. From that viewpoint, an XML instance is given as a *document node* which is a *virtual* parent node of the root node. The tree's root node represents the root element of an XML instance. Non-empty XML elements are the inner nodes of the tree, while empty elements and text content (in this context considered as *text nodes*) correspond to the leaves. The nesting of the elements corresponds to the tree structure. The attributes are loosely coupled with their corresponding elements. We sometimes will refer to arbitrary parts of XML trees as (XML) *fragments*.

In Figure 2.2, a part of the MONDIAL excerpt shown in Figure 2.1 is illustrated as a tree. Because of limited space, the element representing Germany is just sketched but Belgium is shown as a whole. The root node of the tree corresponds to the mondial element while the leaves are in most cases text nodes except for the empty border element. Obviously, inner nodes correspond to the non-empty elements and elements are equipped with their attributes. In this work, we will sometimes use similar tree structures for illustration.

Document Order. Considering this tree-based model, the order of the element and text nodes in a tree is relevant. The *document order* describes in which order all nodes are found in an XML document. According to the *XQuery 1.0 and XPath 2.0 Data Model (XDM)* [XMQ06a], in an XML tree the document order is defined in terms of a depth-first search. This means that the root node is the first node and that for every node it holds that it occurs before its children and subelements which in turn occur before following siblings. In document order, attributes follow immediately their corresponding element but all attributes of an element are considered as unordered.

XML Data Considered as Graph. When ID and IDREF attributes are taken into account, the references between elements having an IDREF attribute pointing to elements with a corresponding ID attribute value can be represented as additional edges between nodes. As these references and thus their representing edges can be found between arbitrary elements, the plain XML tree turns into a directed, possibly cyclic graph. The same holds for XML instances containing XLinks (cf. Section 2.3).

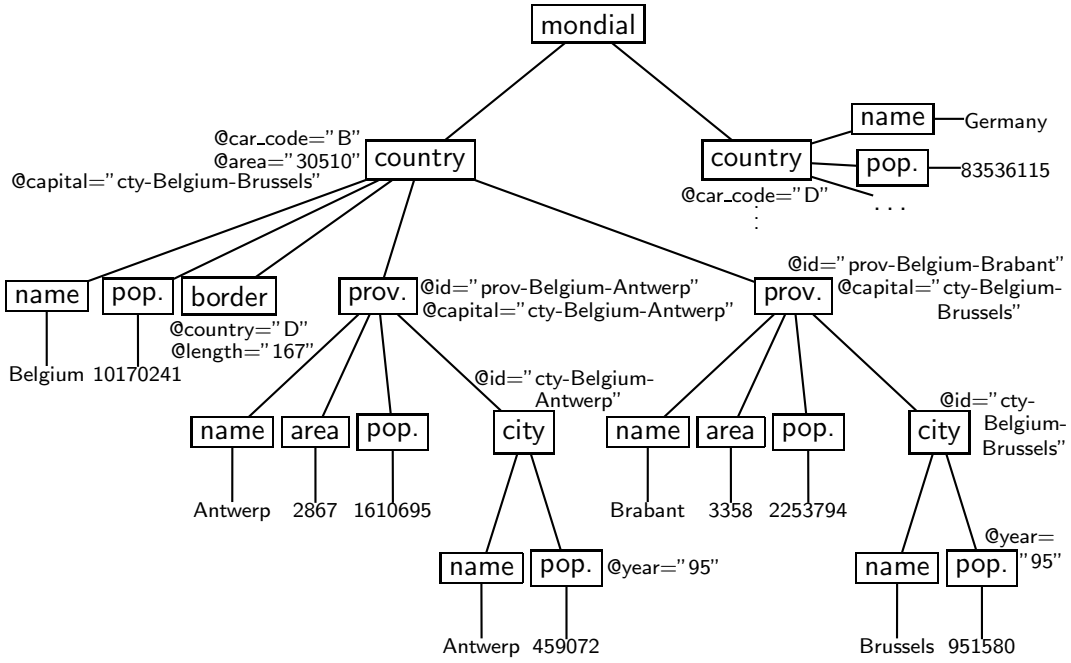


Figure 2.2: XML Document as Tree

XML Information Set. Besides describing the structure of XML data by a tree-based model, it is sometimes useful to directly refer to the information contained in XML documents in a straightforward way. For this purpose, the W3C proposed the *XML Information Set (Infoset)* [XML04a]. Given an XML instance, its Infoset defines its information in terms of *information items* having specific *properties*. Thus, all structural parts of a document like elements and attributes are represented by corresponding items while their properties yield the actual data (e.g. in order to refer to the children of an element item, its children property is accessed).

Other XML Data Models. The *document object model (DOM)* [DOM98] proposed by the W3C is a platform- and language-independent application programming interface (API) for managing documents. It was mainly designed for XML and (X)HTML documents but its specification avoids to refer directly to the technical details of XML.

DOM documents can be considered as hierarchies of nodes. For XML nodes (e.g. elements, attributes and text nodes), specialized subtypes are supplied. There are methods for accessing parts of a document, for navigating through the hierarchical structure, for creating or deleting nodes and for manipulating node properties (e.g. element names) or node contents.

The W3C proposed the *XQuery 1.0 and XPath 2.0 Data Model (XDM)* [XMQ06a] as a standard data model for *querying* XML data. With this model, the input values for queries and the output of XPath and XQuery expressions can be defined.

Instances of the XDM have to be *sequences*. A sequence is an *ordered* collection consisting of arbitrary many *items*. As the XDM relies on the XML Infoset, it also defines a straightforward mapping from the Infoset to the XDM and vice versa.

2.2 XML Querying

For querying XML data, the W3C proposed the *XML Query Language (XQuery)* [XQu06] which has reached the status of a candidate recommendation in late 2005. Due to its wide support by various leading IT companies (among them IBM, Microsoft, Oracle and AT&T), XQuery is most likely to become the standard XML query language. Regarding our investigations on querying distributed interlinked XML data, we focus on XQuery's subset for addressing nodes in an XML tree, called XPath.

Remark. As mentioned already in previous sections, we do not consider namespace nodes, processing instructions, comments and CDATA sections in this thesis. Thus, when querying XML data, we only take document, element, attribute and text nodes into account and querying concepts that are not relevant for these kinds of nodes are not mentioned in the following.

2.2.1 XPath

The *XML Path Language (XPath)* [XPa06] offers mechanisms for addressing nodes in an XML tree conforming to the XQuery and XPath Data Model (XDM) (cf. the preceding section). Note that there are two versions of XPath (1.0 and 2.0). In this work, we will only consider XPath 2.0 which totally covers its predecessor. Though XPath has been specified in its own W3C recommendation, it is a subset of XQuery. The XPath 2.0 recommendation states ([XPa06], Section 1 (Introduction)): "*XQuery Version 1.0 is an extension of XPath Version 2.0. Any expression that is syntactically valid and executes successfully in both XPath 2.0 and XQuery 1.0 will return the same result in both languages.*" Besides XQuery, XPath is also used as a basis for other languages in the XML world, e.g. for XPointer (cf. Section 2.3).

XPath, being a language for addressing and selecting parts of XML trees, is mainly based on *path expressions*. However, the XPath specification also defines arithmetic, logical and comparison expressions similar to other programming languages, and primary expressions like literals, variable references and function calls. Thus, arithmetic expressions (e.g. "8 + 21") or function calls like "doc(*uri*)" (for retrieving an XML document at *uri*) are valid XPath expressions. Additionally, there exist other kinds of expressions like conditional or quantified expressions which will be discussed in connection with XQuery.

For the remainder of this work, with *XPath expressions* we implicitly mean path expressions. These consist of arbitrary many *steps* that traverse an XML tree for selecting specific nodes. The syntax is similar to the *cd* ("change directory") command of the Linux and Unix operating systems for navigating through the file system. Considering for example the excerpt of the MONDIAL XML database depicted in Figure 2.1, the XPath expression

//country[@car_code= "B"]/population

traverses the whole tree and selects all population children of country elements in arbitrary depth with a car_code attribute having the value "B". As there is only one corresponding country element (*Belgium*), it yields its <population>10170241</population> child.

In general, an XPath expression consists of n steps and has the following form:

$step_1/step_2/.../step_n$

If an XPath expression starts with a leading slash, the evaluation will start with the document node of the document to which the expression is applied. In case that it starts with "//", all descendant nodes from the document node serve as initial input nodes. If a relative expression is given (i.e. the expression does not start with a slash), the expression is applied to the currently processed nodes. This makes sense only if the expression occurs as part of an XPath predicate (see below) or if it is used in a dynamic context in XQuery. During an evaluation of an XPath expression, for each step, the currently processed nodes are called *context nodes*. The nodes selected in that step are then the context nodes for the next step.

Each step specifies an *axis* for the navigation direction relative to the current position in the tree and specifies the nodes to be selected by a *node test* and arbitrary many *predicates*. Basically, an XPath step is of the following form:

axis::*nodetest*[*predicate*]

Alternatively, in simple Extended Backus-Naur Form, an XPath step looks as follows:

step ::= *axis* "::" *nodetest* ("[" *predicate* "]")*

In the following paragraphs, the usage of the different kinds of axes in combination with node tests and predicates will be explained.

XPath Axes. Depending on the current nodes' position in the XML tree, the axis specifies which nodes should be selected. There are two kinds of axes that are distinguished wrt. the direction in which they navigate in the current context nodeset. From the viewpoint of an XML tree, *forward* axes select nodes that occur after the context node in document order while *reverse* axes select nodes that can be found before the context node in document order. The different axes types and their semantics are given in Figures 2.3 and 2.4.
Note that in XPath 1.0, a step given as "." is an abbreviation for the step "self::node()". In XPath 2.0, however, the expression "." is considered as primary expression and evaluates to the context item. Thus, it can also be applied to atomic values and not just for nodes.

For some constructs, there exists an abbreviated syntax. For instance, "//" is short for "/descendant-or-self::node()/"[7]. If no axis is given, the child axis is applied as default,

[7] To be more precise, only non-initial occurrences of "//" should be replaced in this manner (cf. [XPa06]).

Forward Axis	Description
child	the children (element and text nodes) of the context node; non-empty only for document and element nodes
descendant	transitive closure of the child axis, i.e. all element and text nodes contained in the subtree
attribute	the attributes of the context node (only for elements)
self	the context node
descendant-or-self	combination of descendant and self
following-sibling	the text and element siblings of the context node that occur after the context node in document order (empty for attributes and document node)
following	all text and element nodes that are not descendants of the context node and occur after it in document order (empty for document node)
namespace	this axis is deprecated in XPath 2.0; before, it could be used to select the associated namespaces of nodes

Figure 2.3: XPath Forward Axes

i.e. "country/city"="country/child:city"[8]. The attribute axis can be given as "@", and ".." is an abbreviation for "parent::node()".

Node Tests. To all nodes that have been selected by the corresponding axis directive, a *node test* is applied. This can either be a *kind test* or a *name test*. For a kind test, there are several options which include the following that are relevant for this work:

- node() (selects all nodes of the current set),
- text() (all text nodes),
- element() (all element nodes),
- element(*name*) (all element nodes of type *name*),
- document-node() (matches the document node),
- document-node(element(*name*)) (matches the document node that has a single element of type *name* as content),
- attribute() (all attribute nodes),
- attribute(*name*) (all attributes nodes of type *name*).

[8] An exception to this is given if the step contains an attribute node test (attribute(), cf. next paragraph). If so, then the attribute axis will be used as default for obvious reasons.

Reverse Axis	Description
parent	the parent document or element node of the context node (empty for document node)
ancestor	transitive closure of the parent axis (empty for document node)
preceding-sibling	the text and element siblings of the context node that occur before the context node in document order (empty for attributes and document node)
preceding	all text and element nodes that are not ancestors of the context node and occur before it in document order (empty for document node)
ancestor-or-self	combination of ancestor and self

Figure 2.4: XPath Reverse Axes

Alternatively, instead of a kind test, a node test can be given as a name test which also requires that the context nodes match the *principal node kind* for the step axis. The W3C XPath recommendation lists the principal node kinds of the different axes as follows [XPa06, Sec. 3.2.1.1 (Axes)] (here, we ignore the deprecated namespace axis):

- For the attribute axis, the principal node kind is attribute.
- For all other axes, the principal node kind is element.

Then, the name test restricts the selected nodes by their name. Here, also namespace prefixes can be taken into account and wildcards are allowed:

- *name* (selects all nodes without namespace prefix and with a name matching *name*),
- *namespace*:*name* (all nodes of type *name* that are qualified with prefix *namespace*),
- *:*name* (all nodes with name matching *name*, no matter if they are qualified with a namespace prefix or not),
- *namespace*:* (all nodes qualified with prefix *namespace*),
- * (all nodes).

Example 2.5 (XPath Node Tests)
The examples given below rely on the aforementioned Mondial *XML database. The following XPath expression consists only of steps* child::*somenodename. It selects the name of all cities, i.e. the text content of the* name *elements:*

```
/child::mondial/child::country/child::province/child::city/child::name/text()
```

The child *axis can be abbreviated simply by omitting it. Thus, the preceding expression can be expressed as follows:*

```
/mondial/country/province/city/name/text()
```

For the child *axis, the principal node kind is element. As a consequence, the following expression containing a name test with a wildcard yields only element children of* city *elements:*

```
/mondial/country/province/city/child::*
```

while the similar query

```
/mondial/country/province/city/child::node()
```

that uses a kind test would produce a result set consisting of both element and text node children (in MONDIAL*, however,* city *elements do not have text content).*

In order to select all name *elements located in arbitrary depth as subelements of countries, the* descendant *axis, abbreviated by "//" can be used:*

```
/mondial/country//name
```

The abbreviated syntax for the attribute *axis("@") is applied in the following expression. It selects all* year *attributes of the cities'* population *children:*

```
//city/population/@year
```

Predicates. Finally, after applying the node test, predicates can be defined for further filtering of the sequence computed by an XPath step. A predicate is always enclosed in square brackets and it is allowed to supply arbitrary many predicates which are then evaluated subsequently. These are again supplied as XPath expressions. Relative expressions (without "/") are applied to the current context nodes selected by *axis*::*nodetest* and absolute expressions (starting with "/" or "//") are evaluated to the document node. The corresponding inner result is then converted to a boolean value. Only those items that evaluate to "true" are selected.

Example 2.6 (XPath Expressions with Predicates)
The following example selects all cities with more then 10 million inhabitants:

```
//country//city[population>10000000]
```

XPath and XQuery Functions and Operators. The accompanying W3C recommendation *XQuery 1.0 and XPath 2.0 Functions and Operators* [XPQ06] defines many additional functions and operators for the use in XPath and XQuery. It specifies many utility functions like common arithmetic operators, functions on sequences and various string functions. Some of these functions are worth mentioning:

- position() results in the context position of the current context node,
- last() returns the size of the context,
- doc(*uri*) retrieves the document node for the XML document found at *uri*,
- id(*id-value*) yields the element that has an ID attribute matching *id-value*. Here, also a sequence of ID values may be supplied as argument. In that case, a sequence of elements matching the IDs is returned.

If a numeric value n is given as predicate, it is interpreted as the expression "position()=n". Thus, only the item that has a context position n evaluates to "true". For instance, when querying MONDIAL, we might only be interested in the first city of each country:

```
//country/descendant::city[1]
```

The doc() function can especially be used for defining an "entry point" for XPath expressions. Also, if a remote document shall be queried with XPath, this function allows for the desired addressing. For instance, the following query selects the element representing Germany in the document "mondial.xml" hosted at a remote server:

```
doc("http://dbis.informatik.uni-goettingen.de/Mondial/mondial.xml")
                                        /mondial/country[@car_code="D"]
```

Example 2.7 (Dereferencing with id())
In MONDIAL*, the* capital *attribute of* country *elements is modeled as* IDREF *attribute. For each country, it contains the id of the* city *element that represents its capital. Here, we query the population of Germany's capital:*

```
/mondial/country[name="Germany"]/id(@capital)/population
```

XPath is the addressing mechanism used in several XML technologies. Besides others, it is part of XQuery which allows for stating more complex queries on XML data.

2.2.2 XQuery

The *XML Query Language (XQuery)* [XQu06] is an extension of XPath. In fact, any XPath expression is also a valid XQuery expression. Similar to SQL [SQL03], its syntax is clause-based and it is "relationally complete" wrt. XML data (e.g., joins can be expressed). Instead of operating directly on XML documents, XQuery is based on the

XPath/XQuery Data Model (XDM) and thus relies on the concept of sequences (cf. Section 2.1.5). It has reached the status of a W3C candidate recommendation in November 2005.

The main syntax construct of XQuery is the *FLWOR* expression (pronounced "flower"). FLWOR is a combination of the main keywords used for this kind of expressions: for, let, where, order by and return. In general, an XQuery FLWOR expression has the following structure:

for $\$var_1$ in $expr_1$, $\$var_2$ in $expr_2$, ..., $\$var_n$ in $expr_n$
let $\$var'_1 := expr'_1$, $\$var'_2 := expr'_2$, ..., $\$var'_m := expr'_m$
where *conditions*
order by *order_by_expr*
return *result*

The for and let clauses produce *tuple streams* consisting of variable bindings. Each variable in the for clause iterates over the sequence produced by the expression it is associated with. In practice, these expressions are usually XPath expressions. Thus, in case that the for clause consists of several variables, each variable iterates over its binding sequence and the resulting tuple stream then contains combinations of all variable bindings. In contrast to that iterative evaluation, the let clause is not processed iteratively. It binds its variables to the whole resulting sequences of their corresponding expressions. The tuples generated by the for and let clauses are then filtered by the where expression. Similar to SQL, for each tuple, the where expression is evaluated and a tuple is kept only if the where expression evaluates to "true". Finally, the resulting tuples can be ordered (order by) and are then used in the return clause for generating the result. FLWOR expressions must at least have a for or let clause and a return expression. The where and order by parts are optional.

Example 2.8 (XQuery)
The following example is a simple XQuery expression that shows the usage of all FLWOR keywords. For all countries, it dereferences the capital IDREF *attribute, filters all capitals with more than one million inhabitants, orders the result by ascending country names and returns an XML fragment consisting of the resulting country names.*

```
for $country in doc("http://.../mondial.xml")//country
let $capital := id($country/@capital)
where $capital/population > 1000000
order by $country/name
return <result>{$country/name}</result>
```

*We could obtain the same result with the equivalent, more concise expression depicted below. Thus, depending on the use case, it shows that XQuery offers various ways where to restrict the resulting sequence (*for *vs.* let *vs.* where*).*

```
for $country in
    doc("http://.../mondial.xml")//country[id(@capital)/population > 1000000]
order by $country/name
return <result>{$country/name}</result>
```

Besides XPath expressions, also arithmetic, logical, comparison and conditional expressions (if ... then ... else ...) can be used in a way similar to imperative programming languages. Additionally, XQuery defines *quantified expressions* that allow for testing sequences (produced by XPath expressions) wrt. quantification. For instance, if applied to MONDIAL, the quantified expression

```
every $country in //country satisfies $country/@area
```

evaluates to "true" if, and only if, all countries found in MONDIAL are equipped with an "area" attribute. Analogously, the quantifier "some" tests if there exists at least one item which satisfies the test.

As shown in the examples above, the return clause produces XML fragments. It uses *direct constructors* for generating XML in a straightforward way by mixing XML syntax (the <result> wrapper element) with variable contents (surrounded by curly brackets). An alternative is given by *computed constructors* for creating XML nodes in a more dynamic way (e.g. attribute id {81 + 27} would create an attribute node "id" with value "108").

User-defined Functions. In XQuery, it is possible to declare *user-defined functions* (example taken from [XQu06]):

```
declare function local:depth($e as node()) as xs:integer
{
        (: A node with no children has depth 1 :)
        (: Otherwise, add 1 to max depth of children :)
        if (empty($e/*)) then 1
        else max(for $c in $e/* return local:depth($c)) + 1
};
```

This recursive function can be declared in advance of the actual query to be processed (i.e. in the prolog). It takes a node as argument and recursively computes the depth of the XML subtree that has node as root node. Note that in XQuery, comments are encapsulated in "(:" and ":)". The following function call (which is a valid XPath expression that can be used stand-alone or in combination with more complex expressions) applies this function to the whole MONDIAL document:

```
local:depth(doc("mondial.xml"))
```

If combined with recursive user-defined functions, XQuery is a computationally complete programming language. It is a very powerful, versatile and flexible XML query language.

Discussing all the facets of XQuery would go beyond the scope of this thesis, for more information please refer to the XQuery specification at [XQu06] or to a dedicated book (e.g. [KCD+03], [Bru04] or [LS04]).

2.3 XML Linking

Most of the documents that can be found in the Web of today are given as HTML documents. In HTML, hyperlinks are simple directed connections between two documents, where one document references another. Though it is possible to specify anchors for pointing to a certain element inside a referenced document, the linking mechanisms of HTML are rather limited. These have been extended in XHTML 2.0, where any element may be equipped with an href attribute in order to be specified as a link *source*. Thus, it is possible to define arbitrary elements as link sources referencing arbitrary elements of remote documents.

Obviously, these linking techniques are designed for *browsing* aspects where users navigate via hyperlinks "by clicking" from one resource to another. Concerning the XML world from the database viewpoint, more sophisticated features for creating links between XML resources would be useful. For instance, besides linking from one resource to another (this can also be considered as *1:1* relation between resources on the Web), it could be useful to connect arbitrary many resources with each other (e.g. for modeling *n:m* relations). Additionally, it might be required to address arbitrary parts of an XML resource like specific subtrees or attributes. Concepts to achieve these sophisticated addressing and linking mechanisms are offered by XPointer and XLink (both proposed by the W3C). We also discuss XInclude which provides basic means for merging XML data into an XML document during the parsing process.

2.3.1 XPointer

In Section 2.2.1, the XML query language XPath has been discussed and it has been shown how an XML tree can be traversed in order to select specific nodes. In addition to the basic addressing mechanisms defined in XPath, some XML applications like XLink [XLi01] (cf. Section 2.3.3) and XInclude [XIn04] (cf. Section 2.3.2) need more sophisticated techniques for special addressing purposes. For instance, with XPath, it is not possible to address ranges of XML trees with arbitrary start and end points. These and further addressing mechanisms are specified in the W3C *XPointer Framework* [XPt03b] and its accompanying scheme specifications.

The XPointer Framework introduces the basic constructs for addressing parts of XML documents. XPointers are usually part of URIs in form of fragment identifiers in order to address parts of specific documents:

http://www.example.org/file.xml#$\underbrace{\text{xpointer–expression}}_{\text{fragment identifier}}$

The XPointer specification distinguishes between *shorthand pointers* and *schema-based pointers*.

Shorthand Pointers

Using DTDs or XML Schema, elements can be associated with an ID. With shorthand pointers, it is possible to address at most one element having a specific ID. On the other hand, it results in an error if no element is found corresponding to a shorthand pointer. In order to define a shorthand pointer, the user simply has to supply the identifying value. This concept is similar to anchors in HTML and the following URI shows how to reference the element representing Germany in the MONDIAL database using the shorthand pointer "D":

http://dbis.informatik.uni-goettingen.de/Mondial/mondial.xml#D

Note that in HTML, anchors have to be explicitly defined in a remote document in order to be referenced by external URIs while in XML it suffices to supply a schema (DTD or XML Schema) that specifies the IDs. Also, IDs are stable wrt. any restructuring of the XML document, i.e. IDs can be kept unchanged while the document and its structure evolves.

Schema-based Pointers

In addition to the straightforward addressing properties of shorthand pointers, the XPointer Framework also offers more flexible pointers. These *schema-based* pointers can be given as follows:

schema-name(*schema-expr*)

The *schema-name* indicates the name of the XPointer schema to be used while the *schema-expr* inside the parentheses has to conform to the given schema. The W3C proposes three schemas: element(), xmlns() and xpointer() which will be discussed in the following paragraphs. Additionally, users may define their own schemas and thus achieve even more flexibility for their applications. In this thesis, however, we will focus on the predefined schemas.

XPointer element() Scheme. For addressing *one* specific XML *element*, the *XPointer element() Scheme* [XPt03a] offers basic mechanisms. Similar to shorthand pointers as described above, a single ID value can be given in combination with element() expressions. Thus, the element with the given ID is located, e.g. the expression

element(D)

addresses the "Germany" element of the MONDIAL database. Note that in contrast to shorthand pointers, it is allowed to specify element() expressions with ID values that do not locate any element.

These simple expressions based on ID values can be extended with so-called *child sequences*, i.e. sequences of integers separated by slashes (/)[9]. Given in combination with an ID value, the first integer n addresses the nth child of the element corresponding to the ID and each following integer locates the appropriate child of the previously addressed element. For instance, if applied to the MONDIAL database, the expression

element(D/8/1)

addresses the first child of the 8th child of the element identified by "D".

Without ID value, and applied to an XML document, a child sequence always has to start with /1 in order to address the root element. Then, the remaining integers will recursively locate the corresponding children in document order.

XPointer xpointer() Scheme. The most sophisticated scheme-based pointers can be specified using the *XPointer xpointer() Scheme* [XPt02]. It is based on XPath. Expressions conforming to the xpointer() scheme have the following form:

xpointer(*xpointer-expr*).

For example, the XPointer

http://.../Mondial/mondial.xml#xpointer(//country[@car_code="D"])

addresses the node that represents Germany in http://.../Mondial/mondial.xml. Thus, the concepts of XPath can be applied in a straightforward way in xpointer() expressions for selecting parts of XML documents. In this thesis, we consider only XPath expressions in place of *xpointer-expr*. Expressions given in the other schemes can be easily mapped to the xpointer() scheme using appropriate XPath constructs, e.g. the id() function for selecting a specific element via its id.

On the other hand, the specification defines additional constructs for addressing strings, points and ranges. For instance with range-to(*xpath-expr*), the range from the context location to the point defined by *xpath-expr* is returned. Note that with these constructs, the addressed parts of XML documents do not necessarily conform to well-formed XML. We do not discuss these concepts here because they have been defined mainly for browsing purposes or for the document-centric viewpoint of XML instances.

Evaluation of XPointers. Given the different XPointer schemes, an XPointer expression has the following structure:

xptr-expr$_1$xptr-expr$_2$...xptr-expr$_n$

The evaluation takes place from left to right and the first expression that evaluates to a non-empty node sequence supplies the result of the XPointer expression. Consider for example the following expression:

[9]Note that in XML, ID values are not allowed to start with a digit.

xpointer(//country[name='Deutschland'])xpointer(//country[@car_code='D'])

When applying the above XPointer expression to the MONDIAL database, the first pointer part results in the empty sequence because all countries are given with English names. Thus, the second part is applied and will return the element representing Germany. For simplicity reasons, in the remainder of this work, we base our investigations on the evaluation of XPointers that contain only one expression of the xpointer scheme. Our results can be iteratively applied for several successive expressions in a straightforward way.

XPointer xmlns() Scheme. The *XPointer xmlns() Scheme* [XPt03c] declares namespace prefixes for XPointer expressions:

xmlns(mon=http://.../Mondial/)xpointer(//mon:country)

Here, the namespace prefix "mon" is bound to the MONDIAL namespace given by the URI "http://.../Mondial/". This URI is used to identify the namespace in a unique way. It is not considered as a target to be selected. Then, the declared namespace prefix is used directly in the XPointer expression to the right of the xmlns() directive for selecting all country elements.

2.3.2 XInclude

With XInclude [XIn04], simple inclusion options can be defined for XML documents (similar to "\input{}" statements in LaTeX). Thus, it is possible to split up big XML instances into smaller parts which are then merged into the main document. This straightforward yet restricted approach is based on XML-friendly directives of the form

<xi:include href="*uri*" xpointer="*xpointer*"/>

where the xi namespace is given by the URI "http://www.w3.org/2001/XInclude". They define a fixed XML-to-XML transformation where the xi:include elements are replaced by the referenced XML fragments. The href attribute specifies the XML document located at *uri* of which the parts to be included are given by an XPointer (*xpointer*) in the corresponding attribute.

Example 2.9 (Splitting Mondial with XInclude)
With XInclude, a straightforward way to split the monolithic MONDIAL *XML instance into several smaller files can be achieved. Here, in this example, all main elements (e.g. countries and organizations) are contained in a corresponding file. An XInclude-aware XML parser would thus materialize the original* MONDIAL *XML document.*

```
<mondial xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="http://.../countries.xml" xpointer="//country"/>
  <xi:include href="http://.../continents.xml" xpointer="//continent"/>
  <xi:include href="http://.../organizations.xml" xpointer="//organization"/>
  <xi:include href="http://.../rivers.xml" xpointer="//river"/>
  <xi:include href="http://.../seas.xml" xpointer="//sea"/>
  <xi:include href="http://.../lakes.xml" xpointer="//lake"/>
  <xi:include href="http://.../mountains.xml" xpointer="//mountain"/>
  <xi:include href="http://.../islands.xml" xpointer="//island"/>
  <xi:include href="http://.../rivers.xml" xpointer="//desert"/>
</mondial>
```

2.3.3 XLink

In order to define links between XML resources, the W3C introduced the *XML Linking Language (XLink)* [XLi01]. It offers ways for defining different types of links. In order to specify a link with the XLink mechanism, a *link element* has to be created. In the remainder of this work, we will also refer to link elements as *XLink elements* or simply *XLinks*. They are represented by XML elements with additional attributes of the xlink namespace (http://www.w3.org/1999/xlink/). Among these attributes, the xlink:type attribute is mandatory and indicates the type of the XLink[10]. Basically, one distinguishes between *simple* and *extended* XLinks.

Remark. We consider XLinks for connecting distributed XML *data* sources and thus we do not investigate the mechanisms designed for browsing or hypertext processing.

Simple XLinks. HTML's Hyperlinks are similar to simple XLinks in XML. A simple XLink connects two resources, one resource referencing the other. It is defined by an xlink:type="simple" attribute and the referenced remote resource is given by its URI in an attribute xlink:href="*uri*". Those URIs may be equipped with a fragment identifier containing an XPointer for addressing parts of remote resources.

Example 2.10 (Simple XLinks)
The distributed MONDIAL *database (which is also illustrated in Figure 1.5) consists of several XML documents connected by appropriate XLinks (cf. the next section). All countries are stored in the resource* countries.xml. *Each country has a* neighbor *XLink child for each neighboring country. An XPointer is used in the simple XLink for referencing the neighboring country by its* car_code *attribute.*

[10] Note that in the new XLink 1.1 candidate recommendation published in early 2006 [XLi06] it is stated that if no xlink:type attribute is supplied for an XLink, it is assumed to have an attribute xlink:type="simple" as default.

```
<country car_code="D" area="30510">
  <name>Germany</name>
  ⋮
  <neighbor xlink:type="simple" borderlength="167"
    xlink:href="http://.../countries.xml#xpointer(//country[@car_code='B'])" />
  <neighbor xlink:type="simple" borderlength="451"
    xlink:href="http://.../countries.xml#xpointer(//country[@car_code='F'])" />
  ⋮
</country>
```

Simple XLinks can be considered as *1:1* or *1:n* relations between resources on the Web, where a source document references arbitrary parts of another resource. For expressing more flexible links, extended XLinks have to be used. Note that simple XLinks are just a shorthand for an often used specific kind of extended links. Thus, the semantics induced by the simple XLink syntax could also be achieved by appropriate extended XLinks.

Extended XLinks. With simple XLinks, resources can be connected in a straightforward way. In addition to that, extended XLinks allow for expressing more sophisticated link relationships between arbitrary many XML resources. From the modeling perspective, they offer an option to specify *n:m* or *many-to-many* relationships.

An extended XLink must have an attribute xlink:type="extended" and may have several children for specifying local and remote resources, and traversal rules (*arcs*) for connecting these:

- *Local resources* are defined *locally* as elements in the content of the XLink.
- *Remote resources* are given similar to simple XLinks.
- Between local or remote resources, *traversal rules*, or *arcs*, are specified.

As we focus on simple XLinks in this thesis, we will only sketch briefly the anatomy of extended XLinks. Example 2.11 illustrates the structure of extended XLinks.

Example 2.11 (Extended XLink for Sister Cities)
The following extended link defines locators for several European cities contained in the Mondial *database. As the city of Padua is not contained in* Mondial*, it is defined as local resource while Freiburg, Innsbruck, Sarajevo, Barcelona and Boston are referenced by locators pointing to* Mondial*. Some of these cities are sister cities. For specifying these relations, appropriate arcs in both directions are given.*

```
<sisterCities xmlns:xlink="http://www.w3.org/1999/xlink" xlink:type="extended">
  <city xlink:label="padua" xlink:type="resource">
    <name>Padua</name>
    <population year="2004">211985</population>
  </city>
```

```
    ⋮
  <cityLocator xlink:type="locator" xlink:label="freiburg" xlink:href=
    "mondial.xml#xpointer(//country[@car_code='D']//city[name='Freiburg'])" />
  <cityLocator xlink:type="locator" xlink:label="innsbruck" xlink:href=
    "mondial.xml#xpointer(//country[@car_code='I']//city[name='Innsbruck'])" />
  <cityLocator xlink:type="locator" xlink:label="sarajevo" xlink:href=
    "mondial.xml#xpointer(//country[@car_code='BIH']//city[name='Sarajevo'])" />
  <cityLocator xlink:type="locator" xlink:label="barcelona" xlink:href=
    "mondial.xml#xpointer(//country[@car_code='E']//city[name='Barcelona'])" />
  <cityLocator xlink:type="locator" xlink:label="boston" xlink:href=
    "mondial.xml#xpointer(//country[@car_code='USA']//city[name='Boston'])" />
    ⋮
  <sisterCityProgram xlink:type="arc" xlink:from="freiburg" xlink:to="innsbruck" />
  <sisterCityProgram xlink:type="arc" xlink:from="innsbruck" xlink:to="freiburg" />
  <sisterCityProgram xlink:type="arc" xlink:from="innsbruck" xlink:to="sarajevo" />
  <sisterCityProgram xlink:type="arc" xlink:from="sarajevo" xlink:to="innsbruck" />
  <sisterCityProgram xlink:type="arc" xlink:from="sarajevo" xlink:to="barcelona" />
  <sisterCityProgram xlink:type="arc" xlink:from="barcelona" xlink:to="sarajevo" />
  <sisterCityProgram xlink:type="arc" xlink:from="barcelona" xlink:to="boston" />
  <sisterCityProgram xlink:type="arc" xlink:from="boston" xlink:to="barcelona" />
  <sisterCityProgram xlink:type="arc" xlink:from="boston" xlink:to="padua" />
  <sisterCityProgram xlink:type="arc" xlink:from="padua" xlink:to="boston" />
  <sisterCityProgram xlink:type="arc" xlink:from="padua" xlink:to="freiburg" />
  <sisterCityProgram xlink:type="arc" xlink:from="freiburg" xlink:to="padua" />
</sisterCities>
```

Remark. The XLink recommendation defines several concepts by specific attributes that don't need to be considered in this work. For instance, the attribute "xlink:show" specifies how the target of XLinks has to be presented in browsing applications: with xlink:show="embed" the application has to embed the referenced resource into the document on link traversal, similar to images in HTML documents.

Additionally, semantic attributes are defined in the XLink namespace for providing optional extra information about links. Using "xlink:title", a link element can be equipped with a human-readable text for describing the meaning of the link. Furthermore, links can be associated with a role by the "xlink:role" and "xlink:arcrole" attributes. Roles are given as URIs and denote certain properties. No specific semantics is given for these concepts and thus we do not consider them in the following.

2.3.4 XLinks for Distributed XML Documents

XLink provides a technology to interlink arbitrary XML instances. This can be useful for splitting big XML instances into several smaller parts which are then connected by XLinks. Basically, one way to achieve this is given by XInclude as illustrated in

Example 2.9. On the other hand, for defining more sophisticated dependencies between XML sources, XInclude is not suitable. For instance, with XInclude, it is not possible to express bidirectional relationships like the "neighbor" relation in MONDIAL where two countries are neighbors of each other. Here, XLink provides more flexibility and expressiveness as illustrated by the following examples.

Example 2.12 (Mondial Distributed)
In its stand-alone version, the MONDIAL *XML database consists of one big file. Using XLinks, we can split this monolithic instance into a distributed version as already illustrated in Figure 1.5. Figure 2.5 shows the structure of the split version of* MONDIAL *in more detail. It consists of a top-level document ("*`mondial`*") that contains only the links to the files where the actual data is situated: a file ("*`geo`*") for all rivers, lakes, seas, deserts, islands and mountains, and documents that contain data about all countries ("*`countries`*"), continents ("*`continents`*") and organizations ("*`orgas`*"). There are several links between these documents, e.g. all countries reference their cities, capitals and provinces in appropriate files. The distributed version of* MONDIAL *can be downloaded via [Mon01].*

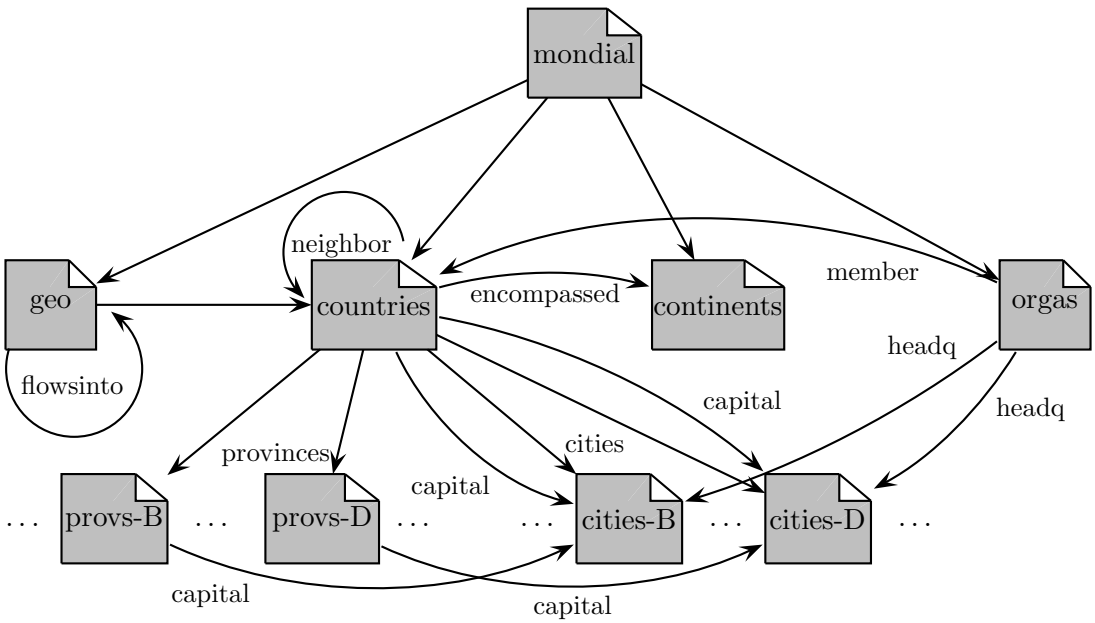


Figure 2.5: A Distributed Version of MONDIAL

External Schema. The classical three-level database architecture [TK78] consists of a physical level (or storage view) which concerns the storage of the data, a logical (or conceptual) level that in relational databases is given by the relational schema and an

external level where users are provided with specific views on the data. As different classes of users usually work with different views, for each view an *external schema* is given which describes the structure of the data.

The preceding example illustrates how the MONDIAL XML database can be distributed over several XML documents. Thus, both the stand-alone and the distributed versions of MONDIAL are possible representations of the same data. Concerning the external level, after splitting MONDIAL, the original schema (given by its DTD) should be kept as external schema which means that the interlinked XML instances should induce a view that conforms to the original DTD. Chapter 3 will explain in detail how this can be achieved.

Example 2.13 (Mondial Distributed: Countries)
In order to illustrate some of the links of the distributed MONDIAL *database, consider again the excerpt of the* MONDIAL *XML database depicted in Figure 2.1. It consists of two countries and some of their subelements. In Figure 2.6, the distributed version of this excerpt is shown*[11]*. The links are graphically emphasized. For instance, the country element that represents Belgium has a* cities *XLink child. It contains a pointer "*http://. . . /Cities/cities-B.xml#xpointer(/cities/city)*" which references all elements that represent Belgian cities. Belgium's* capital *child links only to Brussels. Then, there is a* neighbor *child pointing to Germany. It has an additional attribute to express that the border between Germany and Belgium is 167 km long.*

2.3.5 XLink Usage

In the application area, XLink has not been widely adopted. Though being a W3C recommendation, to the best of our knowledge, it has never been implemented to its full extent in browsing applications. Also in the context of data-centric XML applications it is used only rarely. Here, we try to sketch some possible reasons for XLink's failed acceptance.

Browsing. In the context of the document-centric view on the Web, links are mainly used to connect web pages with other web pages in order to enable users to navigate from one resource to another by clicking on hyperlinks. In addition to that, links are also used for embedding images into web pages. The linking functionality of HTML is rather simple and straightforward, and resembles simple XLinks. Syntactically, only href attributes are needed to express linking semantics in HTML's <a> links or <img> elements. In XHTML 2.0 this feature is extended in a way that any element may be equipped with an href attribute in order to be considered as a link by user agents and browsers. Here, the XLink technique has not been used and the W3C *HTML and XHTML Frequently Asked Questions* state that "*XLink and XHTML had different requirements for linking that turned out not to be reconcilable.*"[12]

[11] For the sake of simplicity, the provinces have been omitted.
[12] `http://www.w3.org/MarkUp/2004/xhtml-faq#xlink`

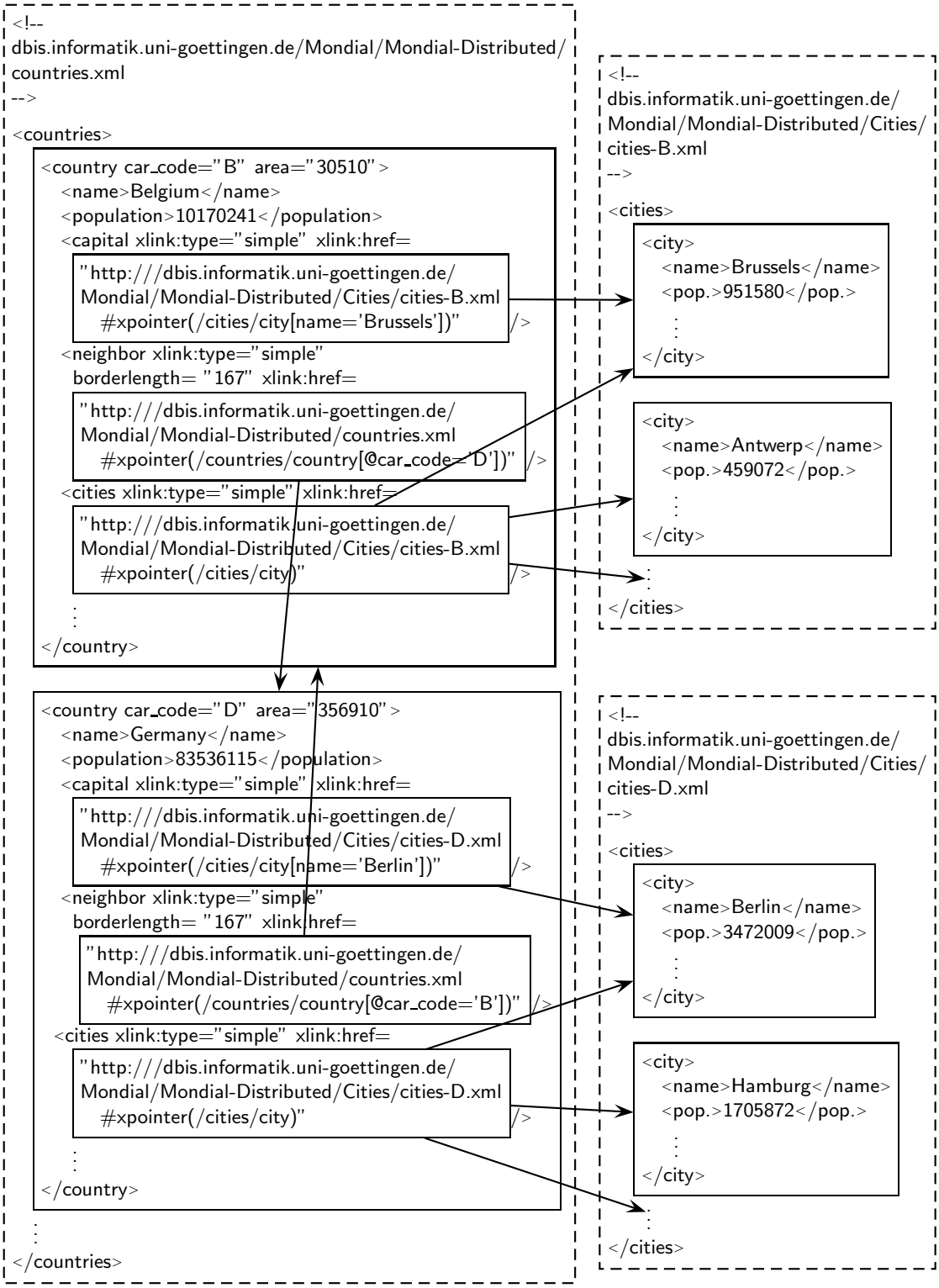


Figure 2.6: Excerpt of the Distributed Mondial XML Database

Another reason for XLink's lack of adoption might be its cumbersome syntax that imposes much overhead (cf. the XML syntax of extended XLinks). However, for most users that browse the web, it might be sufficient to be able to navigate between web sites via hyperlinks. For this purpose, the common linking functionality is fairly adequate and there is no need to extend or replace it with XLink's powerful concepts.

Data. Considered from the data-centric viewpoint on XML, it is more remarkable that XLink has not found its way into the application area. One reason for this might be that XLink has been mainly designed for hypermedia applications as implied by the xlink:show attribute that aims solely for browsing purposes. On the other hand, it is still common to process distributed XML data "by hand". Usually, applications in the Web are realized with server side programming methods (e.g. Java/J2EE or .NET based). If XML data has to be processed, then all sources are accessed explicitly and the desired computations are executed inside the runtime engine.

In contrast to that, XLink has to be used directly on the data level where connections between XML sources can be specified. Unfortunately, XLink does not supply a semantics for embedding XML instances into a resource from the data-centric viewpoint.

2.4 Summary

In this chapter, we discussed the technologies involved in this thesis. Here, XML, the de-facto standard for representing and exchanging data in the web, was considered from the data-centric perspective. We also showed the basic concepts of XPath and XQuery, the most widely used XML query languages. Finally, the XInclude and XLink/XPointer specifications for connecting distributed XML instances have been explained.

In the next chapter, we investigate how distributed XML instances interlinked with XLinks can be mapped to an integrated view. XInclude offers one possible mapping which is not flexible enough for general data integration tasks: it lacks the required flexibility to combine data according to a given external schema.

Having defined the mapping, how can the resulting view be queried with XPath? We propose and discuss appropriate techniques concerning the evaluation of queries on this view in Chapter 4.

To our best knowledge, these issues have not been examined yet.

3 The dbxlink Model for Mapping XLinked XML Sources

In this chapter, we describe a model for mapping interlinked XML instances to a logical view. This model is specified by a flexible and expressive extension "dbxlink" for XLink and provides a basis for the investigations conducted in the remainder of this work.

3.1 Motivation

XML documents are not required to be self-contained but may rather have links to remote XML sources. As shown in Section 2.3.3, such references to *autonomous* resources can be defined with the W3C XLink specification [XLi01] in terms of a *syntactical representation* as XML elements. In contrast to the context of browsing and navigating to remote documents via links, from the data-centric viewpoint, a set of distributed interlinked XML documents induces a *logical view* that can be considered as a *virtual XML instance*. Figure 3.1 illustrates how this scenario is related to the classical three-level database architecture [TK78].

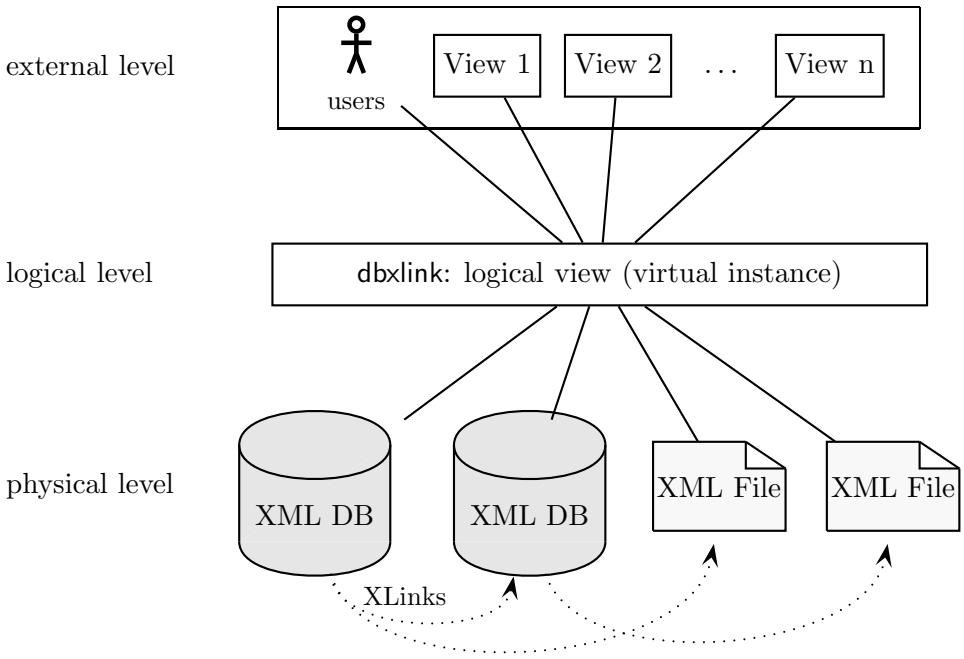


Figure 3.1: Three-Level Database Architecture

On the physical level, interlinked XML sources (e.g. provided by database systems or stored as plain files) are given. These are then mapped to an integrated view (the logical level) which in turn serves as a basis for defining further views on the external level from where it is also accessed by users.

Unfortunately, XLink does not specify *how* the referenced fragments should be mapped into the virtual instance. Thus, the XLink mechanism has to be extended with *semantics* that defines how to actually handle instances with references. The primary goal is that XLink references are mapped to a *logical model* that is (or at least provides the look-and-feel of) a plain XML instance that can be subject to the application of standard languages from the XML area. Especially, XPath as the basic addressing mechanism underlying XQuery must be applicable. Thus, a transparent modeling as an XML-to-XML transformation where the XLink elements are present only on the syntactical level, but queries navigate in the virtual instance along semantic notions is desirable.

Given an XML instance with XLink references, the actual specification of the *logical model* must be flexible enough to cope with data integration issues. For instance, if for a distributed XML scenario a target schema is given that has to be met by the virtual instance, the model should allow versatile mapping options. Considering simple XLinks, the naive mapping approach would be to replace an XLink element with the target of its "xlink:href" attribute as done by XInclude (cf. Section 2.3.2). However, it might be useful or even necessary wrt. data integration to have alternative mapping options like the merging of the XLink's local data (i.e. non-XLink-attributes and subelements) with the referenced nodes. In addition to that deficiency, the XLink specification solely specifies linking semantics for the context of hypermedia systems while the data-centric viewpoint is not considered. Thus, two questions arise:

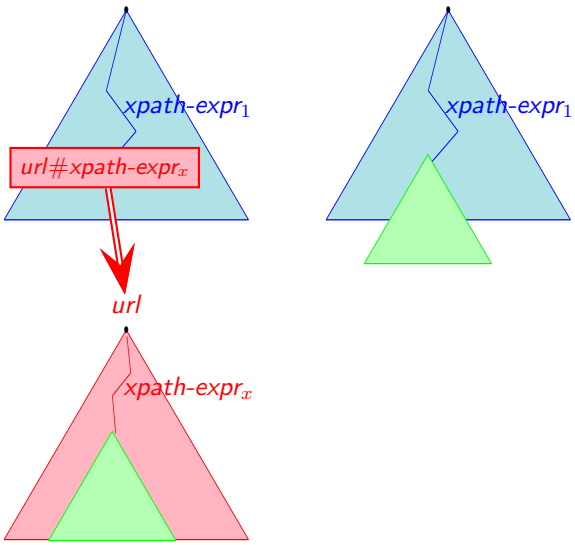
1. What kind of modeling options are useful for (simple) XLinks?
2. How can interlinked XML data instances be queried while navigating across links?

In order to propose a solution to these issues, we introduced additional modeling and querying directives as an extension to the XLink technology by the dbxlink namespace in [May02, MM03, BFM06a]. Similar to the xlink namespace, the dbxlink namespace offers several attributes that specify the database-specific semantics of XLink elements. Also, XLink's attribute xlink:actuate is interpreted for the data-centric viewpoint on XML. To give a first intuition, we shortly mention the relevant attributes used for simple XLinks:

- dbxlink:transparent is used to specify how the referenced data is mapped into the referencing instance,
- xlink:actuate supplies the time-point for evaluating the reference (during parsing or query answering),
- for querying, dbxlink:eval specifies how the evaluation of the XPointer expression contained in the xlink:href attribute is distributed between the local and remote server, and

- with dbxlink:cache, it can be specified which intermediate results (of both the XPointer and query results) are cached for reuse.

The dbxlink:transparent directives are the main issue in this chapter while the other directives will be explained in Section 4.1.3. In order to get an intuition of the logical model which is specified by the dbxlink:transparent attributes of XLinks compared to the real data model, see Figure 3.2. We thus have specified a model that *transparently* resolves and embeds XLinks into a virtual instance.



(real DM) (transparent DM)

Figure 3.2: Extended XML Data Model with XLink Elements

Remark. In this thesis, all investigations related to querying interlinked XML instances are restricted to simple XLinks. For extended XLinks, the modeling issues are more complex. However, the results presented in this work can serve as a basis for dealing with extended links.

Therefore, in the following, we assume any XLink to be a simple XLink. According to XLink 1.1 [XLi06], simple XLinks are an application default. Thus, we will omit the attribute xlink:type="simple" occasionally and assume that it is given implicitly.

3.2 Mapping Distributed XML Instances

The dbxlink namespace is an extension to the xlink namespace. It introduces an attribute "dbxlink:transparent" that specifies the mapping of the data referenced by an XLink. For illustration, consider the following XML fragment that contains a simple XLink element:

<*linkelement* xlink:type="*simple*" xlink:href="*uri#xpointer*"
dbxlink:transparent="*to be described*" *non-(db)xlink-attributes*>
content
</*linkelement*>

Basically, the *linkelement* consists of a start and an end tag that embed some attributes and content (here, we focus on subelements and text content). The tags can also be seen as element "*hull*" while the attributes and the contents can be denoted as element "*body*". Analogously, each of the nodes referenced by *uri#xpointer* found in the xlink:href attribute also consists of a "hull" and a "body". In order to describe the linking semantics of the *linkelement*, i.e. the structure of the XML fragment after evaluating the XLink, we have to take both the link element and its referenced nodes into account. Thus, the *virtual model* of this fragment can be seen as a combination of both its "local" and "remote" parts. In order to specify (i) the mapping of the result set of *uri#xpointer* (remote parts), and (ii) the mapping of the simple XLink element itself (local parts), the value of the dbxlink:transparent policy attribute consists of appropriate keywords. In the following, we describe the directives for simple XLinks. Also, several examples are given for illustration.

Remark. Note that the *logical* model possibly induces an infinite tree (e.g. due to the neighbor relation in our example), but as long as only a finite set of Web sources is involved, it has a finite representation as a graph. Thus, the actual query evaluation wrt. this (abstract) model must then care that it does not run into cycles. These issues will be discussed in Section 4.5.

3.2.1 Directives for Simple XLinks

For simple XLinks, the dbxlink:transparent attribute consists of two keywords, namely the *left-hand-* (*link-*) and *right-hand-directive* (*result-directive*):

dbxlink:transparent="*left-hand-directive* *right-hand-directive*"

The left-hand-directive describes how the XLink element should be treated while the right-hand-directive specifies what parts of the referenced nodes should be mapped into the logical model (e.g. the complete nodes or just their bodies).

Right-hand-directives ("R"-directives):

- insert-nodes (the nodes contained in the result set of the XPointer shall be inserted "as a whole", i.e. without changing their structure), and
- insert-bodies (for element nodes, their subelements, text children and attributes are taken; for text and attribute nodes, the "body" is considered empty).

If no "R"-directive is given, then "insert-nodes" is used as default because it is the intuitive, straightforward option which keeps the structure of the referenced nodes. The

"R"-directive "insert-bodies" selects the body of the nodes addressed by the XPointer, namely their subelements, text children and attributes.

Left-hand-directives ("L"-directives):

- drop-element: the XLink element is dropped and replaced with the result set,
- keep-body: the hull of the XLink element is dropped and only the information of its body (i.e. its subelements, text children and (non-xlink- and non-dbxlink-) attributes) is used for enriching the referenced nodes,
- group-in-element: all referenced nodes are embedded in the link element,
- duplicate-element: each referenced node is placed in a duplicate of the link element,
- make-attribute: the link element is dropped and an attribute (with the same name as the link element) is added to the link's parent. If text and attribute nodes are contained in the result set of the XPointer after applying the "R"-directive, they are used as attribute value. In case of element nodes, an IDREF(S) attribute is added to the link's parent which references auxiliary elements representing the referenced elements. The latter are enriched with the link element's body.

The evaluation of the right-hand-directive results in a sequence of element nodes, text nodes and attributes. For each left-hand-directive, we now describe how this result set is processed wrt. the link element.

The examples are based on an excerpt of countries.xml, the document of the distributed MONDIAL database that contains information about all countries[1]. Each country is represented by an XML element "country" that has children for the name, population, etc. It also has simple XLink children for referencing the country's capital and its cities: both point to cities located in the given country which can be found in the remote XML document cities-XX.xml where "XX" stands for the country's car code.

drop-element. This directive is the simplest and most straightforward choice. It is thus also used as a default if no left-hand-directive is given for a link. It forces the link element to be dropped completely and be replaced by the nodes of the result set obtained after applying the right-hand-directive. If the result set contains attributes, they are added to the link's parent element (which is the canonic behaviour as induced by element constructors in common XML manipulation languages).

For instance, in the distributed version of MONDIAL, each country has a cities link element which references the appropriate cities:

[1] For simplicity reasons, we don't include other elements like province or neighbor here. The neighbor elements will be used for illustrating cyclic structures in Chapter 4.

```
<country car_code="D" area="356910">
  <name>Germany</name>
  <population>83536115</population>
  ⋮
  <cities dbxlink:transparent="drop-element insert-nodes"
    xlink:href="http://. . . /cities-D.xml#xpointer(/cities/city)" />
  ⋮
</country>
```

In the logical model, the link element is dropped and replaced with the referenced nodes which are not modified because insert-nodes is given as "R"-directive:

```
<country car_code="D" area="356910">
  <name>Germany</name>
  <population>83536115</population>
  ⋮
  <city>
    <name>Berlin</name>
    <population>3472009</population>
  </city>
  <city>
    <name>Hamburg</name>
    <population>1705872</population>
  </city>
  ⋮
  ⋮
</country>
```

In order to illustrate the special case that occurs if attributes are contained in the result set which replaces the link, consider the following example:

```
<country car_code="D" area="356910">
  <name>Germany</name>
  <population>83536115</population>
  ⋮
  <climate dbxlink:transparent="drop-element insert-nodes"
    xlink:href="http://. . . #xpointer(//country[@id='GER']/@climatic_zone)" />
  ⋮
</country>
```

Assume that we have access to a server hosting XML files with meteorologic data. Here, the XPointer of the climate link addresses the climatic_zone attribute of an element containing the meteorological data for Germany. In the resulting structure, the link element is dropped and the resulting attribute is added to the link's parent, namely country:

```
<country car_code="D" area="356910" climatic_zone="temperate">
  <name>Germany</name>
  <population>83536115</population>
  ⋮
</country>
```

keep-body. This "L"-directive can be considered as an extension of drop-element. Instead of dropping the link element, its body is kept and the element nodes contained in the XPointer result set are enriched (attributes and text are kept unchanged): to each of these element nodes, add all non-xlink- and non-dbxlink-attributes, element and text children of the XLink element. Finally, replace the XLink element with the enriched result elements. Note that keep-body is equivalent to drop-element if the link has no additional data.

Assume that to the cities link from above a country attribute has been added:

```
<country car_code="D" area="356910">
  <name>Germany</name>
  <population>83536115</population>
  ⋮
  <cities country="D" dbxlink:transparent="keep-body insert-nodes"
      xlink:href="http://. . . /cities-D.xml#xpointer(/cities/city)" />
  ⋮
</country>
```

With keep-body, this additional attribute will be kept and added to all referenced elements. Thus, in the induced view, each city element obtains an attribute that indicates the country it is located in[2]:

[2] Usually, this information is derived from the fact that cities are subelements of the appropriate countries.

```
<country car_code="D" area="356910">
  <name>Germany</name>
  <population>83536115</population>
  ⋮
  <city country="D">
    <name>Berlin</name>
    <population>3472009</population>
  </city>
  <city country="D">
    <name>Hamburg</name>
    <population>1705872</population>
  </city>
  ⋮
  ⋮
</country>
```

group-in-element. The link element is basically kept and it is modified in the following way. All attributes belonging to the dbxlink and xlink namespaces are dropped. Any element or text node contained in the result set obtained by resolving the XPointer and applying the "R"-directive is inserted into the logical view as child of the link element. Attributes are added to the kept link element and if an existing attribute and a result set attribute coincide wrt. their names, their values are concatenated. This "L"-directive is useful for embedding the referenced data into the link element as will be illustrated with a modified cities link:

```
<country car_code="D" area="356910">
  <name>Germany</name>
  <population>83536115</population>
  ⋮
  <cities dbxlink:transparent="group-in-element insert-nodes"
      xlink:href="http://.../cities-D.xml#xpointer(/cities/city)"/>
  ⋮
</country>
```

In the fragment induced by the example given above, the cities element contains all referenced city elements:

```
<country car_code="D" area="356910">
  <name>Germany</name>
  <population>83536115</population>
  ⋮
  <cities>
    <city>
      <name>Berlin</name>
      <population>3472009</population>
    </city>
    <city>
      <name>Hamburg</name>
      <population>1705872</population>
    </city>
    ⋮
  </cities>
  ⋮
</country>
```

duplicate-element. For each node of the referenced result set the link element is duplicated. Thus, each result node is embedded in its own local element which stems from the link that is treated like in the group-in-element case (drop dbxlink- and xlink-attributes, then insert attributes, text nodes and elements).

This directive is useful if the included result elements shall not be grouped together (as in the case of group-in-element) but rather be inserted separately. For instance, additional data about cities could be included from a server that provides a file germantowns.xml which consists of town elements:

```
<country car_code="D" area="356910">
  <name>Germany</name>
  <population>83536115</population>
  ⋮
  <city source="not approved"
      dbxlink:transparent="duplicate-element insert-bodies"
      xlink:href="http://.../germantowns.xml#xpointer(//town)" />
  ⋮
</country>
```

With this modeling, the body of each referenced town element is embedded into a separate city element having a source attribute indicating that the included data has to be checked for correctness:

```
<country car_code="D" area="356910">
  <name>Germany</name>
  <population>83536115</population>
  ⋮
  <city source="not approved">
    <name>Göttingen</name>
    <population>129051</population>
  </city>
  <city source="not approved">
    <name>Braunschweig</name>
    <population>245273</population>
  </city>
  ⋮
  ⋮
</country>
```

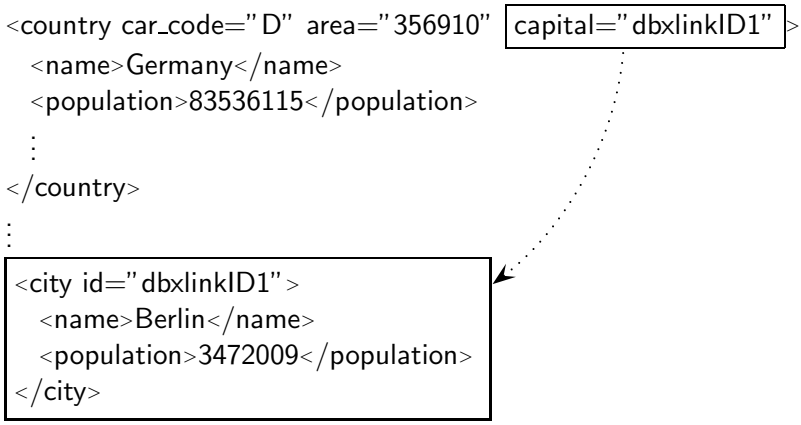
make-attribute. This directive is different from the aforementioned ones in a sense that the result elements are enriched with the link element's body and the link element itself is "transformed" into a reference attribute. The link's parent element gets a new attribute with the name of the link element. If text and attribute nodes are among the result nodes, their values are added to the new attribute's value. For element nodes, the new attribute is of type IDREF(S). If the result set contains several element nodes, then an IDREFS attribute is added with a value for each element and IDREF otherwise. This reference attribute points to new auxiliary elements, one per result element. The new elements have to be equipped with ID attributes corresponding to the values of the IDREF attribute of the link element. These elements can be inserted somewhere in the virtual instance and they can be directly addressed via the id() function that dereferences the corresponding IDREF attribute.

In MONDIAL's distributed version, the capital of countries is modeled with make-attribute as shown below.

```
<country car_code="D" area="356910">
  <name>Germany</name>
  <population>83536115</population>
  ⋮
  <capital dbxlink:transparent="make-attribute insert-nodes"
     xlink:href="http://. . . /cities-D.xml#xpointer(/cities/city[name='Berlin'])" />
  ⋮
</country>
```

Thus, the link element is dropped and a capital IDREF attribute is added to the country which references an auxiliary element representing the referenced data. Here, only one

element is referenced ("Berlin") which has been added somewhere to the local virtual instance.



It is worth mentioning that the modeling of make-attribute is "orthogonal" to the other "L"-directives because the included data is only accessible via the attribute axis or via the dereferencing function id(). Thus, if XPath queries should traverse the data referenced by an XLink with make-attribute they have to contain an appropriate step involving the id() function.

Summarizing Example. In Figure 3.3, parts of the elements representing Belgium and Germany are given (the population elements are omitted). They contain the links for their respective cities and capital. Figure 3.4 gives an intuition of the corresponding node references. The resulting XML tree and fragment for the induced logical model can be seen in Figures 3.6 and 3.5.

Remark. Note that in the logical model, there is a redundancy concerning the element representing a country's capital. It can be found as auxiliary element referenced by the capital attribute and as a city child of the country. With respect to the data model, this is not a problem, because the model is considered as *virtual* instance. On the other hand, while querying, it might be useful to avoid such redundancies in order to reduce both computational and network resources.

To put it all together, mapping an XLink element according to the dbxlink:transparent directive consists of two steps:

1. processing the XPointer's result set according to the right-hand- or "R"-directive (yielding a set of nodes ("insert-nodes"), or a set of bodies ("insert-bodies")),
2. mapping the XLink element itself, as specified by the left-hand- or "L"-directive.

The resulting nodeset is then added to the parent element as new children and/or attributes.

```
<countries>
 <country car_code="B" area="30510">
  <name>Belgium</name>
  <capital xlink:type="simple" dbxlink:transparent="make-attribute insert-nodes"
   xlink:href="http://.../Cities/cities-B.xml#xpointer(/cities/city[name='Brussels'])" />

  <cities xlink:type="simple" dbxlink:transparent="drop-element insert-nodes"
   xlink:href="http://.../Cities/cities-B.xml#xpointer(/cities/city)" />
 </country>
 <country car_code="D" area="356910">
  <name>Germany</name>
  <capital xlink:type="simple" dbxlink:transparent="make-attribute insert-nodes"
   xlink:href="http://.../Cities/cities-D.xml#xpointer(/cities/city[name='Berlin'])" />

  <cities xlink:type="simple" dbxlink:transparent="drop-element insert-nodes"
   xlink:href="http://.../Cities/cities-D.xml#xpointer(/cities/city)" />
 </country>
  ⋮
</countries>
```

Figure 3.3: Distributed Version of MONDIAL with Additional dbxlink Directives

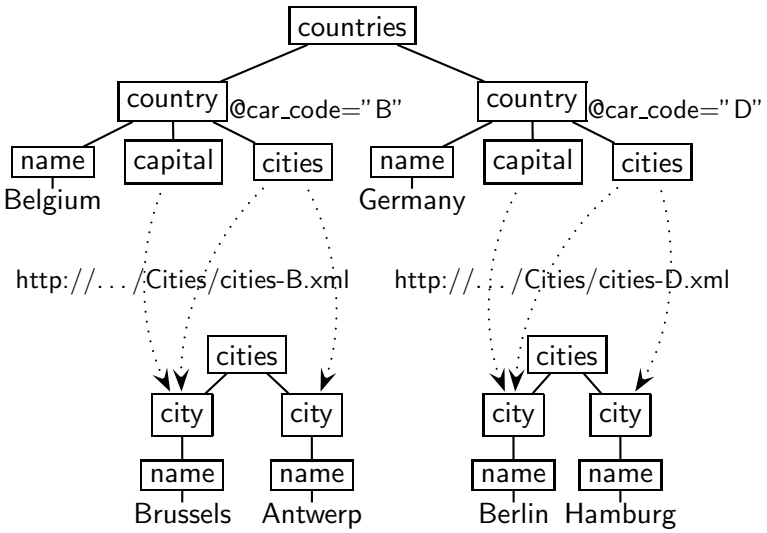


Figure 3.4: Original Document Trees with XLink References

```
<countries>
  <country car_code="B" area="30510" capital="cty-brus">
    <name>Belgium</name>
    <city><name>Brussels</name></city>
    <city><name>Antwerp</name></city>
  </country>
  <country car_code="D" area="356910" capital="cty-berlin">
    <name>Germany</name>
    <city><name>Berlin</name></city>
    <city><name>Hamburg</name></city>
  </country>
  ⋮
  <city id="cty-brus">
    <name>Brussels</name>
  </city>
  <city id="cty-berlin">
    <name>Berlin</name>
  </city>
</countries>
```

Figure 3.5: Resulting Logical Model in XML ASCII Representation

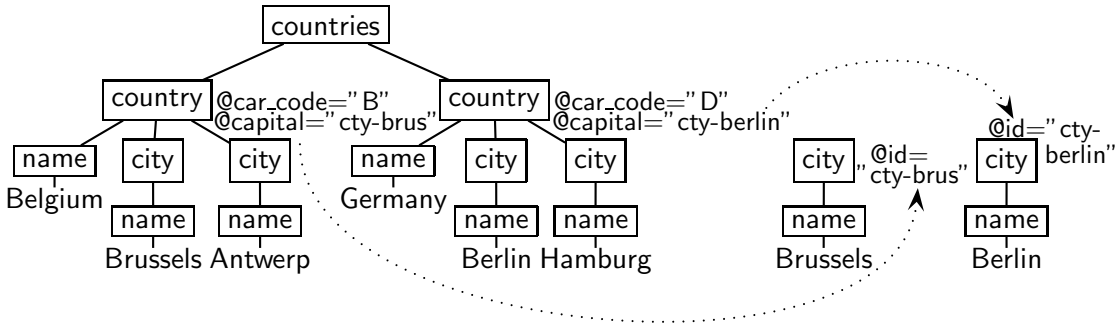


Figure 3.6: Resulting Logical Model with ID/IDREF References in Tree Representation

3.2.2 Relative XLinks

The XPath expressions contained in the XPointer part of an link element's xlink:href attribute do not necessarily have to address element nodes. Instead, they may select atomic values which can be used for enriching the link element's parent with additional data. Especially, in combination with *relative* references this is an option for deriving properties. Consider the following example:

```
<country car_code="D" area="356910">
  <name>Germany</name>
  <population>83536115</population>
<density dbxlink:transparent="make-attribute insert-nodes"
          xlink:href="population div @area" />
  ⋮
</country>
```

When the density link element is resolved, it adds a density attribute to the country element:

```
<country car_code="D" area="356910" density="234">
  <name>Germany</name>
  <population>83536115</population>
  ⋮
</country>
```

Thus, in contrast to fully specified URIs given in the XLink's xlink:href attribute, also *local, relative* XPointer expressions are allowed. We define the completion of the relative expressions given in the xlink:href attribute as follows:

- a complete URI is considered as it is;
- an absolute XPath p is appended to the URI of the current document as fragment identifier (embedded in an XPointer expression "xpointer(p)");
- a relative path expression is evaluated wrt. the XLink element's *parent* node. This strategy is consistent with the modeling because the XLink can be considered as "virtual" in a sense that it is replaced by referenced data items which are then children or attributes of the parent element.

3.3 Relationships with W3C Concepts

The dbxlink proposal extends the functionality of XLink with several modeling options and evaluation strategies. The approach is based on and related to various technologies introduced by the W3C as will be discussed in this section.

3.3.1 XML Infoset

Usually, XML documents induce an XML Information Set (XML Infoset). As a document might come along with an XML Schema, there also exists an augmented XML Infoset for this document, e.g. consisting additionally of all default values supplied in the schema. In [XML04c], this augmented infoset is called the *post-schema-validation infoset (PSVI).*

From this perspective, the mapping of interlinked XML sources to the model defined by the dbxlink approach happens after the schema has been evaluated. Intuitively, given an XML document, we first have to process any given schema and obtain thus a new instance conforming to the PSVI. Then, we can continue with the link evaluation for getting the virtual instance. This instance conforms to a *post-dbxlink-validation infoset (PDBXI)*. If in a data integration application a target schema is given, this PDBXI has to conform to the target schema.

3.3.2 XLink for Browsing

Up to now, the XLink approach is primarily used in the context of hypermedia systems, i.e. for browsing. Therefore, the W3C XLink Recommendation [XLi01] defines several attributes for link elements which specify the *behavior* of the link element during browsing. For instance, the xlink:show="embed" directive requires the referenced XML fragment to replace the local link element upon traversing the link. This specific behavior of XLink can be seen as one special case of the dbxlink approach, specified by dbxlink:transparent="drop-element insert-nodes" which is thus considered as the default option if no dbxlink:transparent directive is given.

3.3.3 XInclude

As described in Section 2.3.2, with XInclude [XIn04], big XML documents can be split into smaller ones by straightforward inclusion mechanisms. Similar to simple XLinks, xi:include elements contain a *uri* and an *xpointer*. The semantics of these inclusion rules corresponds to dbxlink:transparent="drop-element insert-nodes", i.e., the referenced target is included when the document is loaded/parsed, materializing the model completely.

In contrast to the XLink and XInclude models, the dbxlink approach allows for fine-tuning the logical model. It is more flexible wrt. the model because various combinations of mapping directives are allowed. Additionally, we have directives for query evaluation and caching strategies (dbxlink:eval and dbxlink:cache, cf. next chapter) which are not given in XLink or XInclude.

The XInclude handling of references is also not suitable for database environments possibly containing huge amounts of data. In general, the XInclude parse-time evaluation materializes all referenced contents especially when storing documents. In contrast to that, dbxlink allows to keep the links in the database, only evaluating them on-demand, always using the latest contents of the referenced sources.

4 Querying XML Sources along XLinks with dbxlink

Considering XLink from the data-centric viewpoint of XML, its specification does not offer constructs for describing the modeling semantics of referenced XML fragments. Thus, as described in Chapter 3, we proposed an extension to the XLink technique called "dbxlink". It provides necessary modeling directives in terms of dbxlink:transparent attributes to be added to link elements in order to flexibly integrate interlinked XML sources into one virtual logical view. This view then serves as a basis for querying distributed XML sources connected by XLinks as will be described in this chapter.

First, we analyze how distributed XML sources containing XLinks can be queried and show the benefits of our approach. We will then explain why we focus our investigations on a fragment of the XML query language XPath using only forward axes. Then, we describe two methods for evaluating queries on interlinked XML sources: a straightforward but naive approach that is not suitable in practice, and a preferable dynamic evaluation method handling XLinks on demand during query evaluation. Afterwards, we show how we can detect and handle cyclic instances during query processing and how to deal with non-terminating queries.

4.1 Querying Linked XML Instances

In this section, we describe the situation regarding XQuery's capabilities to handle XLinks. We also illustrate the benefits of our approach and discuss additional querying directives of the dbxlink approach for specifying evaluation and caching strategies. Parts of this section have been published in [BFM06b].

4.1.1 XQuery and XLinks

Although the W3C *XML Query (XQuery) Requirements* [XMQ04, Sec. 3.3.4/3.4.12 ("References")] explicitly state that

"the XML Query Data Model MUST include support for references, including both references within an XML document and references from one XML document to another"

and that

"queries MUST be able to traverse intra- and inter-document references",

neither XPath nor XQuery can handle XLinks to their full extent. This is mentioned in the XML Query Requirements (for each of the quoted requirements, it states that "*this*

requirement has been partially met") and will be analyzed in a general way below.

Considering intra-document references, i.e. IDREF attributes referencing ID attributes in the same document, the id() function allows for dereferencing them in order to access the referenced elements. On the other hand, for accessing remote documents, the doc() function can be used but in XPath or XQuery there is not yet complete support for evaluating XPointers remotely. For a simple XLink element, users can select the XPointer found in the xlink:href attribute with

```
for $pointer in
doc("http://.../countries.xml")//country[@car_code="B"]/capital/@xlink:href
```

but XQuery cannot be told to resolve it because a *data item* (i.e., the value of the xlink:href attribute) can not be considered as an XPath query. This is currently not possible in the base language, nor can it be achieved by the functions and operators given in the *XQuery 1.0 and XPath 2.0 Functions and Operators* specification [XPQ06]. The following paragraphs show that a solution for handling shorthand pointers based on XQuery exists while xpointer() expressions consisting of XPath queries can not be resolved by XQuery in general.

Simple XPointers for ID-Dereferencing. Simple XPointers actually consisting of an id() function application of the form *url*#xpointer(id(*string*)) for dereferencing elements by their ID can be resolved by combining the doc() and id() functions. These pointers are equivalent to "shorthand pointers" like http://.../country.xml#D (pointing to Germany) in [XPt03b] (cf. Section 2.3.1). In [LS04, Section 7.4.2], a solution by an XQuery user-defined function is given which is restricted to such simple XPointers:

```
declare namespace func="http://www.example.org/functions";
declare function func:follow-xlink($href as xs:string) as item()*
{
  let $docValue := substring-before($href,"#")
  let $x := substring-after($href,"#xpointer(id('")
  let $idValue := substring-before($x,"')")
  return
    doc($docValue)/id($idValue)
};
```

The URI supplied as $href argument is split into its host part containing the path to the referenced document and its fragment identifier which is separated from the host part by "#". The solution in [LS04] assumes the fragment identifier to contain an XPointer that consists of an id() function call for dereferencing an element by its id. Thus, general XPointers based on arbitrary XPath expressions can not be handled. In order to return the referenced element, an XPath expression is dynamically constructed and evaluated. It consists of two steps: the first resolves the remote document with the doc() function and the second consists of the id() function for dereferencing the appropriate element. Note that here, the relevant computed variables are passed to XPath functions as arguments.

XPath Expressions in XPointers. In the general case, instead of id($idValue), the xpointer() scheme allows for the use of arbitrary XPath expressions. Similar to the solution proposed above, a naive approach might look as follows:

```
declare namespace func="http://www.example.org/functions";
declare function func:follow-xlink($href as xs:string) as item()*
{
  let $docValue := substring-before($href,"#")
  let $x := substring-after($href,"#xpointer(")
  let $path := substring-before($x,")")
  return
    doc($docValue)/ $path
};
```

The remote document is resolved in the same way like above. However, instead of extracting a simple id value, we have to access the XPath expression contained in the xpointer() directive using the auxiliary variable $x. Intuitively, we then concatenate the doc() function call and the obtained path expression but this will result in an error. This dynamically constructed XPath expression is not accepted by XQuery because it is not allowed to use a variable as location step. This can also not be programmed using the current *XQuery 1.0 and XPath 2.0 Functions and Operators* [XPQ06]. Note that in case that the XPointer is not based on an id() function call, the proposed solution given in [LS04, Section 7.4.2] explicitly returns a message "*XPointer Syntax nicht unterstützt*" (XPointer syntax not supported). However, there exist XPath implementations and other proposals that allow for this dynamic evaluation.

Dynamically Evaluating Data Items as Queries. For the dynamic evaluation of an XPath expression supplied as a string variable, the XPath/XQuery implementation Saxon [SAX] offers an *extension function* saxon:evaluate(*string*). This function also works on remote documents. The above function can be expressed as

```
declare namespace func="http://www.example.org/functions";
declare function func:follow-xlink($href as xs:string) as item()*
{
  let $docValue := substring-before($href,"#")
  let $path := replace($href, "^.*#xpointer.(.*).$","$1")
  let $expr := concat("doc('", $docValue, "')", $path)
  return
    saxon:evaluate($expr)
};
```

Thus, the data item $path containing the XPath expression used in the XPointer is dynamically evaluated on the remote document.

The XQuery extension proposed in [RBHS04] introduces a mechanism for evaluating XPath/XQuery expressions on remote servers that offer an appropriate interface. The syntax reads as "execute at *uri* xquery { *xquery* }". Using this technique, the above query could be stated as follows:

```
declare namespace func="http://www.example.org/functions";
declare function func:follow-xlink($href as xs:string) as item()*
{
  let $docValue := substring-before($href,"#")
  let $path := replace($href, "^.*#xpointer.(.*).$","$1")
  return
    execute at $docValue xquery { $path }
};
```

Then, queries use –similar to the id() function– an *explicit* dereferencing. For instance, in order to query the population of Belgium's capital in the distributed MONDIAL database that models the capital relationship for countries by an XLink to the appropriate city:

```
doc("http://.../countries.xml")//country[@car_code="B"]/
                    capital/func:follow-xlink(@xlink:href)/population
```

In contrast to that explicit approach,we argue that, if XLink references are used, implicit dereferencing is preferable. Seeing interlinked XML instances as one integrated view (that e.g. has to conform to a target schema), we want to provide a way to use original XPath/XQuery *without* any syntactical extensions thus enabling users to *transparently* query the data while XLinks are dereferenced *implicitly* like

```
doc("http://.../countries.xml")//country[@car_code="B"]/capital/population.
```

In this section, we showed that it is not (yet) possible to query XML instances containing simple XLinks with XPath/XQuery. As simple XLinks are just a syntactic variant of specific extended XLinks, it also follows that XQuery can not handle extended XLinks.

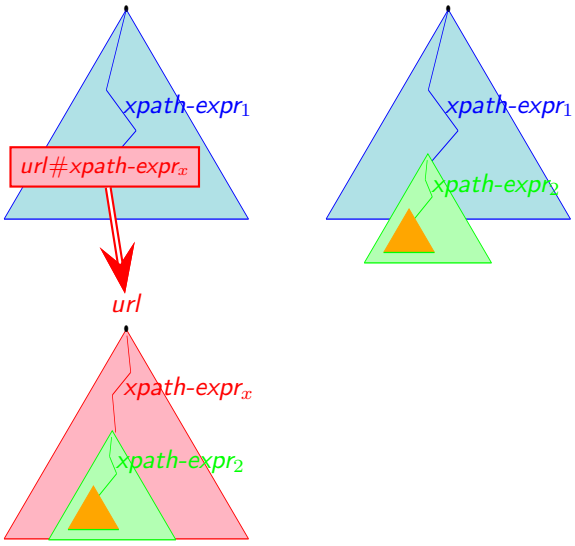
4.1.2 Querying Distributed XML the dbxlink Way

As our model defines the semantics of XLinks in terms of an XML-to-XML mapping, linked XML sources can be seen as a virtual XML instance. In the following, we illustrate how this virtual instance can be queried.

As described in the preceding section, when querying the real data model, we explicitly have to "jump" from the local document to the referenced instance. In contrast to that, with the dbxlink approach, interlinked XML instances are considered as *transparent*. For all link elements, the result set defined by their xlink:href attributes is silently mapped into the referencing XML structure according to the dbxlink directives.

Figure 4.1 shows how a given XPath query is processed wrt. the real and the transparent model. Consider an XPath query *xpath-expr*$_1$/*xpath-expr*$_2$ which for simplicity

reasons should consist only of steps along the child axis. The first part of the query (*xpath-expr*$_1$) yields a node sequence on which the remaining query (*xpath-expr*$_2$) has to be evaluated. As only the child axis is involved in this example scenario, *xpath-expr*$_2$ would continue on the children of all elements found in the intermediate node sequence generated by *xpath-expr*$_1$. If the sequence contains an element with an XLink child, assume that this link element has an xlink:href attribute that includes an XPath-based XPointer expression *xpath-expr*$_x$ referencing some XML fragment of a remote document. The left hand side of Figure 4.1 shows how in the real data model a "jump" to the remote document is required. Using the dbxlink approach, the referenced fragment is silently mapped into the logical view which can then be queried as indicated on the right hand side of Figure 4.1. This simplified illustration serves for giving a first intuition while Chapter 5 describes the querying process in detail and for arbitrary XPath expressions.



(real DM) (transparent DM)

Figure 4.1: Querying over XLink Elements

Consider again the distributed Mondial XML database, i.e. its excerpt shown in Figure 3.3 that contains countries and cities. As depicted in Figure 3.6, the resulting model allows for the following queries:

- the capital is modeled as an IDREF attribute:
  doc("...")//country[@car_code="B"]/id(@capital)/population
- each referenced city is modeled as a subelement, dropping the "auxiliary" cities link element:
  doc("...")//country[@car_code="B"]/city/name

4.1.3 Additional Directives

In the dbxlink context, the XLink directive xlink:actuate specifies the time-point for evaluating links. Besides the modeling directives given by the dbxlink:transparent attribute, the dbxlink namespace defines additional directives:

- dbxlink:eval indicates the preferred evaluation strategy, and
- dbxlink:cache can be used for supplying caching modes.

Evaluation Time-Point. The W3C XLink recommendation [XLi01] specifies an attribute xlink:actuate that can be used for indicating the desired time-point of a link traversal. Like the whole recommendation, this attribute is tailored for the browsing context of hypermedia systems. Thus, the directive xlink:actuate="onRequest" is equivalent to the requirement that a user has to click on a hyperlink in order to navigate to the referenced resource. Analogously, xlink:actuate="onLoad" requires that the link is traversed while loading the document.

As the dbxlink extension has been developed for the data-centric viewpoint of XML, xlink:actuate is used for indicating the desired time-point for evaluating an XLink wrt. the mapping directives given in the dbxlink:transparent attribute. For xlink:actuate, in the dbxlink context, the same options can be distinguished:

- xlink:actuate="onLoad" forces an XLink to be evaluated (i.e. resolved and materialized) during the parsing of the document,
- xlink:actuate="onRequest" denotes that a link has to be evaluated when it is used by a query.

If all XLinks are equipped with xlink:actuate="onLoad", the whole virtual instance will be materialized when added to a database. In contrast to that, "onRequest" guarantees that always the current state of the referenced resources is queried. If no xlink:actuate attribute is given, then "onRequest" is the default.

Evaluation Strategies. For the evaluation of a link, there are three different strategies that can be supplied with the dbxlink:eval directive:

- dbxlink:eval="local" accesses the complete remote document and processes both the XPointer and the remaining query ($xpath\text{-}expr_2$) locally in two successive steps,
- dbxlink:eval="distributed" evaluates the pointer remotely and the remaining query locally (note that this is the default option if dbxlink:eval is omitted), and
- dbxlink:eval="remote" combines the pointer and $xpath\text{-}expr_2$ to a new query to be processed remotely.

The option "local" should be used if the remote server is not able to answer XPath queries (including XLink resolving). Then, the whole document is requested and stored locally, and query processing takes place on the local copy. According to the classification proposed in [FJK96], the "local" strategy resembles *data shipping* because the whole referenced document is transferred to the local host.

When using the evaluation mode "distributed", it is assumed that the remote server can handle XPath queries. The XPointer expression (*xpath-expr*$_x$ according to Figure 4.1) is submitted to the target host which then returns the XML fragment referenced by the pointer. This fragment is then temporarily added to the virtual instance according to the dbxlink:transparent directives. Afterwards, the remaining query *xpath-expr*$_2$ is evaluated against the temporarily extended virtual instance. This evaluation mode relates to *hybrid shipping* as described in [FJK96] because the evaluation of the XPointer takes place remotely and the remaining XPath query is processed locally.

Finally, the option "remote" delegates both the XML fragment computation and the remaining XPath query to the remote host. To achieve this, the expressions *xpath-expr*$_x$ and *xpath-expr*$_2$ are rewritten and combined to a new XPath query in a suitable way. As this is a complex task with many special cases and restrictions, it is discussed in detail in Chapter 6. Considering the classifications of [FJK96], this strategy resembles *query shipping*.

Note that for both the "distributed" and "remote" methods, the remote host must be able to process XPath queries, and it also has to cope with the dbxlink approach in order to handle further XLinks contained in the referenced XML document. Chapters 5 and 6 describe how these evaluation modes affect the query process.

Caching Modes. During the evaluation of queries, in order to resolve an XLink, it is usually necessary to access a remote server. Then, the obtained data may be cached for reuse in subsequent queries. For individual XLink elements, caching modes can be specified by the dbxlink:cache attribute. There are several options:

- dbxlink:cache="complete" indicates to store the whole referenced document in the local XML database,
- dbxlink:cache="pointer" denotes to keep the referenced XML fragment (the result of the XPointer) in a local cache,
- dbxlink:cache="answer" caches the query result obtained by applying the user's XPath query to the referenced fragment,
- dbxlink:cache="on" combines several caching strategies and includes fallbacks depending on dbxlink:eval, and
- dbxlink:cache="none" can be used for turning caching off.

Disabled caching ("none") is the default option if no dbxlink:cache attribute is supplied for a link. Thus, it is guaranteed that always the latest data is queried. While

"complete" forces the whole remote document referenced by the XLink to be cached locally, the "pointer" option can be used to cache the XML fragment that is defined by the XPointer expression. After the remaining user query has been processed on the fragment, "answer" denotes to cache the final query result for the subtree defined by the fragment. Evaluation and caching options can not be combined arbitrarily, e.g. it makes no sense to require a caching of the whole document if we have shipped a query for reducing network traffic. Thus, designers of linked documents must be aware that based on a specific evaluation strategy, appropriate explicit caching options ("complete", "pointer" and "answer") are chosen. In order to offer a caching option that works with any evaluation mode, we provide dbxlink:cache="on". With this option, the appropriate caching mode is used automatically. dbxlink:cache="on" is defined to work incrementally, e.g. if a whole document has been cached, also the computed fragment and the query result will be kept for later use. All details and further investigations concerning caching issues can be found later in Section 7.1.

4.2 Focus on XPath without Reverse Axes

XLinks are defined as XML elements (cf. Section 2.3.3) and thus correspond to nodes of the XML tree. Because in XQuery, the addressing of specific nodes is only possible via XPath expressions, it is sufficient to consider XPath queries in order to treat XQuery expressions correctly on XML instances interlinked with the dbxlink approach. Thus, in the remainder of this work, we will focus our investigations on XPath.

It should be noted, that as a consequence, any XML query language which bases its node location facility exclusively on XPath (e.g. XSLT [XSL06] and XPathLog [May04]) can be enabled for handling XLinks according to the dbxlink approach. This can be achieved by integrating the techniques proposed in this thesis into the query engine of systems that implement this kind of XML query languages.

The work in [OMFB02] shows that any XPath expression can be rewritten into an equivalent query without reverse axes and it proposes several algorithms to achieve this. Thus, without loss of generality, we assume here that a given XPath query to be processed does not contain any reverse axis (parent, ancestor, ancestor-or-self, preceding and preceding-sibling). This can also be assumed for all XLinks, i.e. any XPointer expression consists solely of forward axes: while parsing a document any XPath-based XPointer will be rewritten accordingly.

4.3 Naive Querying Approach

In [BFM06a] we proposed a formal specification for our dbxlink approach based on a function ϕ. It recursively maps an XML document *doc* including all XML fragments which are reachable via XLinks from *doc* to the single virtual XML instance obtained by resolving all XLinks according to the contained dbxlink:transparent directives. Thus, a query that enters a linked network of XML documents at a *uri* should yield the same answer as it would if applied against $\phi(\mathsf{doc}(uri))$. In other words, an intuitive, naive

approach to query an interlinked set of XML instances would consist of two steps (cf. Figure 4.2 for an intuition):

1. Materialize the whole virtual instance induced by all interlinked XML instances wrt. the contained XLinks and their dbxlink directives.
2. Evaluate the given XPath expression on this new instance.

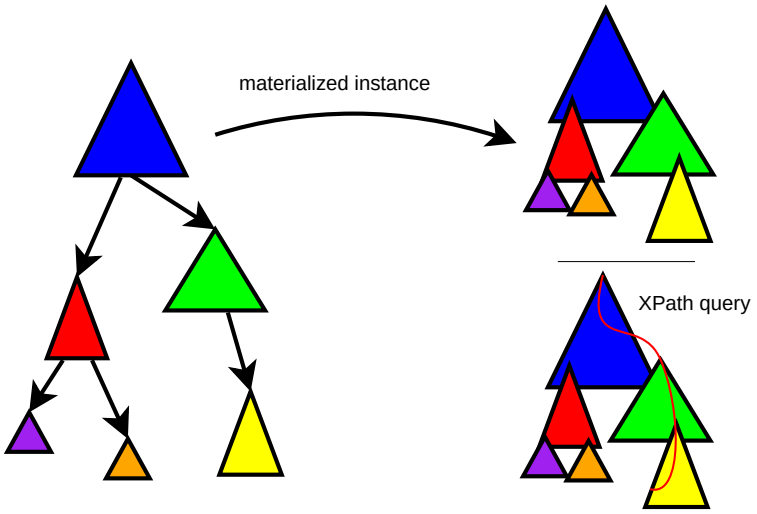


Figure 4.2: Querying the Materialized Virtual Instance

However, this solution is not suitable in practice for several reasons. If many links and thus many distributed documents are involved, it might be time-consuming to fetch all partaking XML documents and to compute the virtual instance. Also, the virtual instance can be of huge size. This overhead can be avoided because usually not the whole instance is needed to answer the given query.

For illustration, consider the following scenario based on the distributed MONDIAL database (cf. Section 2.3.4). One server contains information about all countries stored in a file "countries.xml" and another one hosts all files containing the cities of a specific country (e.g. "cities-D.xml" for all German cities). We want to query the names of all German cities using the following XPath expression which is stated on the server hosting the data about countries:

doc('http://.../countries.xml')/countries/country[name='Germany']/city/name.

In order to build the virtual instance for MONDIAL, the city files of all 260 countries have to be accessed and processed resulting in 260 network requests to the remote server[1]. Then, after fetching the "cities-XX.xml" documents to the local server host-

[1]Many more XML instances are interlinked in the distributed MONDIAL database and have to be requested to materialize the whole view. For the sake of clarity we don't consider them here.

ing "countries.xml", the virtual instance has to be built resulting in an XML instance consisting of all country elements each having subelements for all its cities.

Obviously, for evaluating the above query, instead of resolving all links to cities-XX.xml documents in advance, it would be sufficient to resolve only the link referencing German cities. This link element is the only context node computed by the intermediate step country[name='Germany']. This *partial instance* of the logical view, corresponding to the parts that are emphasized by the ellipse in Figure 4.3, is much smaller and only a few network accesses are needed for its computation. Therefore, it is not necessary to build up the whole virtual instance before query evaluation.

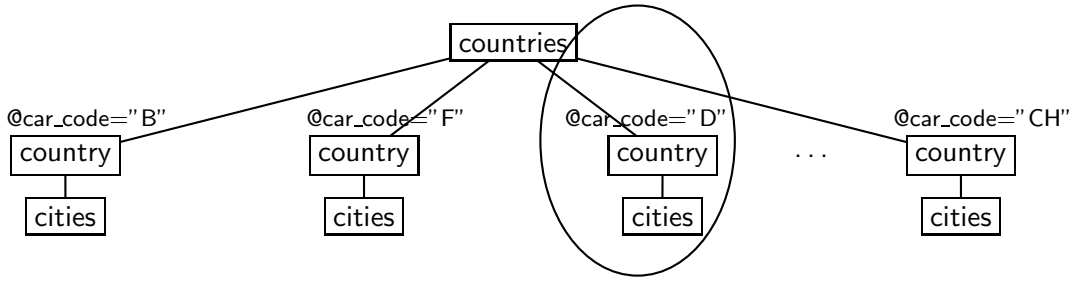


Figure 4.3: Relevant XLink References

Even worse, it is possible that the interlinked instances contain cycles and thus the straightforward materialization process might not terminate. For example, in the distributed version of MONDIAL, the neighbor children of the country elements are link elements pointing to neighboring countries. As this is a symmetric relation, the neighbor children of a country point to the corresponding neighboring countries and vice versa as depicted in Figure 4.4 for Germany and France.

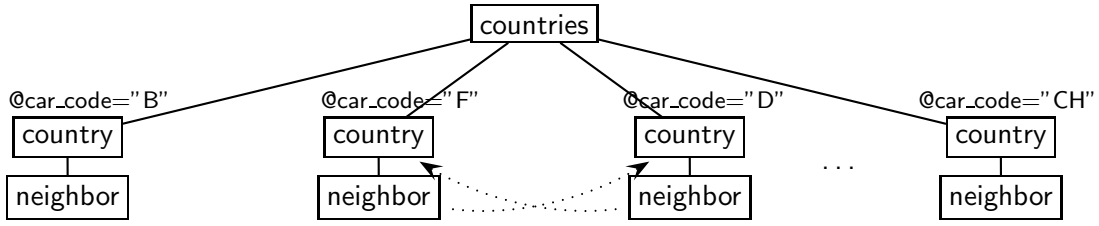


Figure 4.4: Cyclic XLink References between two Elements

A naive materialization process would replace a neighbor link with the appropriate country which again has the "inverse" neighbor link that has to be expanded. This results in an infinite expansion process which is sketched in Figure 4.5.

Hence, in order to evaluate a query, it is in general neither necessary nor possible to create the whole virtual instance in advance. It is sufficient to expand relevant links *on demand* during query evaluation. Using this approach, also cyclic instances can be handled without the need of materializing the whole virtual instance. This is achieved by

detecting cycles during query evaluation as described later in Section 4.5. We describe this dynamic evaluation process in the following section.

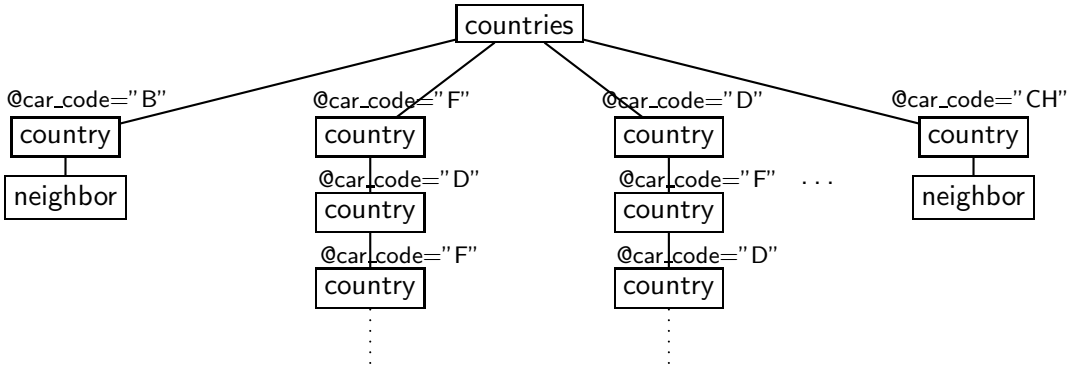


Figure 4.5: Infinite Expansion Process for Cyclic Instances

Remark. The directive xlink:actuate specifies the desired time-point for evaluating an XLink. In simple scenarios with only a few links, it can be appropriate to specify xlink:actuate="onLoad" for all links. Thus, the whole virtual instance would be materialized once, assuming it is not too big and does not contain cycles. On the other hand, we consider general, non-trivial cases where we could have cyclic and possibly huge virtual instances. Also, if caching is disabled, fresh data shall be accessed on remote servers, which is not possible for a static materialized instance. Hence, in the remainder of this thesis, we assume that for all links the directive xlink:actuate="onRequest" is specified, which is the default XLink evaluation time-point option in the dbxlink approach anyway.

4.4 Dynamic Query Evaluation

As shown in the preceding section, the generation of a complete virtual instance is often inefficient or even not possible due to cycles. Instead, for processing queries, we dynamically compute the relevant parts of the virtual instance during query evaluation. In order to describe which parts are the relevant ones, we first have to analyze the XPath evaluation process. Usually, XPath query processors evaluate each step successively.

4.4.1 Stepwise Result Set Evaluation

Given an XPath expression, we assume that it has the following form consisting of n *location steps*:

$$step_1/step_2/\ldots/step_n$$

The most common and intuitive method for evaluating XPath queries is to subsequently apply all location steps. This method is also induced by the semantics definition given

by the W3C in [XPa06] and [XMQ06b]. In each step, the set of nodes selected by the previous step is called the current *context*. In the first step, if an absolute path expression is given, the document node is the initial context. In case that a relative path is given, the initial context is for instance given by an XQuery variable containing a node sequence. Then, for each node n of the current context the current step is evaluated, selecting a sequence of matching items relative to n. This results in attribute or element nodes, or atomic values that form the context for recursively applying the next step, and so forth. Note that not complete intermediate results are materialized, but only local contexts on the way to the next step. Most XPath engines like Saxon [SAX], Xalan [Xal], and eXist [exi], the native XML database system we chose for a proof-of-concept implementation (cf. Chapter 5), use this strategy. Thus, by showing how this strategy has to be extended in order to be able to handle XLinks during query evaluation, any such XPath query engine can be turned XLink aware by integrating these changes.

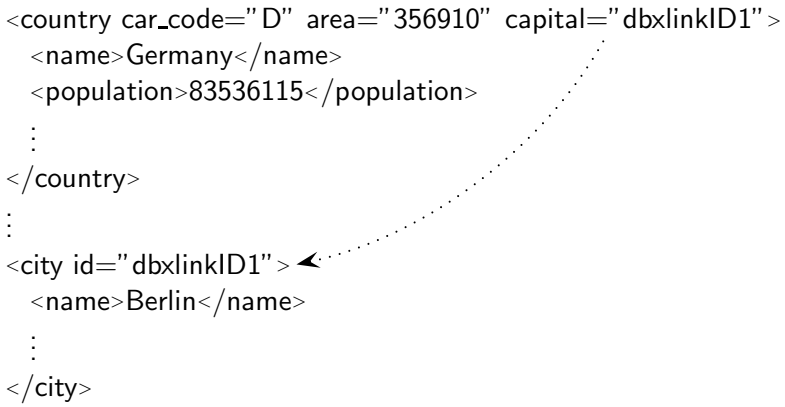
4.4.2 Extension of the Stepwise Evaluation

The stepwise evaluation process of an XPath query consists of isolated steps, where each step produces a new intermediate result set for a given context. As XLinks are not expanded in advance, it has to be checked if any XLink induces nodes that contribute to the next step to be applied. Hence, in order to implicitly replace all link elements relevant for a context node, a kind of look-ahead approach is required.

Consider again the excerpt of MONDIAL's distributed version depicted in Figure 3.3. It is used in the following example which illustrates the stepwise evaluation. Assume that we have a current context consisting of all country elements and have to apply a next step "@capital" (which is the abbreviation of "attribute::capital"). This step produces a new context that contains the countries' capital attributes. Now, consider the country element which represents Germany:

```
<country car_code="D" area="356910">
  <name>Germany</name>
  <population>83536115</population>
  <capital dbxlink:transparent="make-attribute insert-nodes"
    xlink:href="http://.../cities-D.xml#xpointer(/cities/city[name='Berlin'])" />
  ⋮
</country>
```

According to the dbxlink model, in the virtual instance, the capital link element is dropped and an IDREF attribute capital is added to the country parent element because "make-attribute insert nodes" is specified as dbxlink:transparent attribute. This new reference attribute then points to a local, auxiliary city element:



As we do not materialize the whole instance in advance, we have to be sure that the link is resolved and the resulting element is matched by the next step "attribute::capital". As a consequence, we need to resolve all relevant links *before* applying the next step. Afterwards, the navigation could continue with an id() function call for selecting the local element representing Berlin.

The basic idea for extending the stepwise evaluation of XPath expressions in order to respect links is as follows. To each node contained in a current context the next step has to be applied. The set of nodes selected by the next step's axis might contribute to the next context (the nodetest and predicates have to be applied afterwards). Thus, any relevant link element among these nodes selected by the axis has to be resolved. Which link is relevant depends on its dbxlink:transparent attribute, and on the axis and nodetest to be applied. The entire process of determining the relevant links wrt. the different modeling directives is described in Section 5.2.

4.5 Cyclic Instances and Non-Terminating Queries

As described in Section 4.3, we have to cope with cyclic instances while querying interlinked XML sources. In this section, this issue is discussed. First, we describe two different types of cycles. Then, we discuss how cycles can be detected during query evaluation. Finally, we describe the issue of non-terminating queries in a scenario of interlinked XML instances.

4.5.1 Ordinary Cycles

We consider cyclic structures induced by links with non-XLink-elements in-between as *ordinary cycles*. The cycles produced by the neighbor elements illustrated in Figures 4.4 and 4.5 can be used as an example for this kind of cycles. These cycles are dangerous only if we transitively navigate through a subtree, i.e. if any of the descendant, descendant-or-self or following axis is involved. They imply the need for a recursive traversing of the XML tree in arbitrary depth. All other axes are harmless because they describe a traversing that is finite and they don't select nodes of arbitrary depth, i.e. even in a scenario which contains such ordinary cycles, the query evaluation process will terminate.

Example 4.1
Consider again the Example illustrated in Figures 4.4 and 4.5. The modeling is in some issues problematic, since e.g. the query

/countries/country[@car_code='B']//city

*does not only return cities in Belgium, but, as e.g. France is a (*neighbor*) subelement of Belgium, also the cities in France (and all other countries that are reachable by land from Belgium). The borders relation is symmetric and hence France again has Belgium as subelement. Thus, the straightforward evaluation process for this query won't terminate.*

On the other hand, if we consider the slightly modified query

/countries/country[@car_code='B']/neighbor/neighbor/neighbor/name

that returns the names of all countries that can be reached from Belgium by passing three times some border, e.g., Belgium itself (via France and Germany) or France (via going to France, going back to Belgium, and again to France). Any such query that uses non-recursive steps can use a cycle in the logical model, but only for different situations *of the evaluation.*

In contrast,

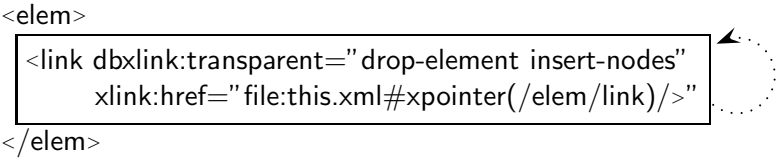
/countries/country[@car_code='B']//neighbor/name,

which is the union *of the above answers, would run into a lot of cycles, and run infinitely with the naive strategy.*

Hence, ordinary cycles are harmful only for specific queries. However, modelings can be constructed that produce situations where also queries without recursive axes might not terminate. This issue will be discussed in the following section.

4.5.2 Vicious Cycles

We now discuss a special "vicious" scenario. Consider a document of the following form:

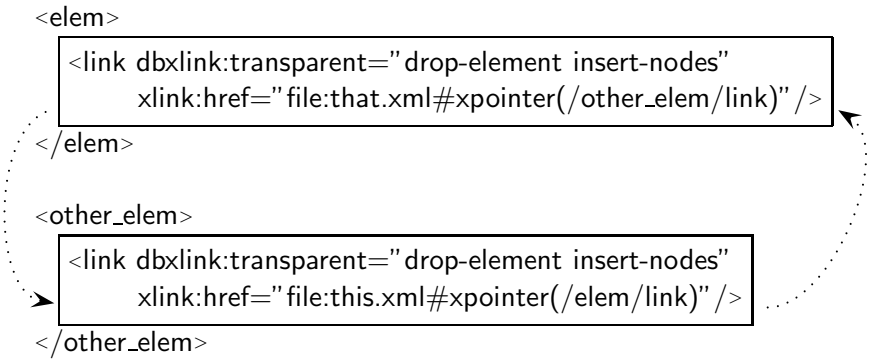


Here, the link element is pointing at itself. Intuitively, if the link is resolved, it is replaced with itself. For generating the virtual instance, this process would not terminate because it would infinitely try to resolve the referenced link element which is always replaced by itself.

Considering the dynamic query evaluation method described in the preceding sections, executing the XPath query /elem/link on this document would result in an infinite loop,

because the directive "drop-element insert-nodes" would always require to replace the link element with a copy of itself.

While the scenario above might be detected during parsing the document, the following example shows a more complex scenario. Here, we have two elements in two different files (that could also be hosted on different servers), referencing each other:



We just need to state the same query (/elem/link) on the first document in order to get caught in a loop again: because the link child of the node elem points to the link subelement of other_elem and vice versa, we will infinitely jump from one link element to the other during the evaluation of the considered query.

Note that we don't have to state queries involving the descendant(-or-self) axis to end up in infinite computations. An XPath expression consisting of two steps each using the child axis does the job. Such *vicious cycles* can only occur if we have a chain of n link elements ℓ_1, ℓ_2, ..., ℓ_n such that ℓ_i references ℓ_{i+1} and $\ell_1{=}\ell_n$. This scenario can only occur if drop-element or keep-body[2] is used for all n link elements, as described in the example above. For links with group-in-element and duplicate-element this can not happen because the link element is kept (and thus an explicit next step selecting this element is necessary) and xlink and dbxlink attributes are removed from both the local link element and from the result nodes. Thus, for these two "L"-directives, it is not possible that a link element is replaced by a new link. This is similar for make-attribute because the referenced fragment is integrated as auxiliary element which is then referenced locally by an IDREF attribute.

Any evaluation directive can be involved in a scenario where vicious cycles occur: in the local and distributed modes, a link element is replaced locally with the referenced node which is again a link element. In case of remote evaluation, a query would be shipped from one document to the other without termination.

4.5.3 Detection of Cycles

According to the two kinds of cycles described above, two situations can occur where we have to detect cycles:

[2]Note that the semantics of keep-body is equivalent to that of drop-element if the link element has no children and no non-XLink-attributes.

1. The next step to be evaluated is recursive, i.e. the descendant, descendant-or-self or following axis is involved. We thus have to be aware of ordinary cycles.

2. The link to be handled is equipped with drop-element or keep-body and hence, vicious cycles have to be detected no matter what query is given.

Considering the first case (ordinary cycles), whenever the same link is visited twice during the recursive evaluation, there is a cycle. Secondly, a vicious cycles occurs if a chain of links with drop-element or keep-body is given and – similar to ordinary cycles – the same link has to be handled more than once.

Thus, in both cases, a cycle can be detected by appropriate book-keeping using the element ids: here, in this abstract analysis, we assume that every XML element can be globally identified in a unique way by the URI pointing to the document it is contained in combined with an internal id (e.g. built according to the document order). The book-keeping functions as follows. If we have to detect cycles, i.e. any of the two situations described above occurs, then for each link to be handled, we keep its element id in a dedicated list. Depending on the chosen shipping strategies, appropriate auxiliary information must be generated or provided for controlling the evaluation process.

In case of local or distributed evaluation modes (data shipping or hybrid shipping), the book-keeping can be done locally. Thus, we generate a local list that contains the element ids of the currently visited links. For each new link to be resolved, we check if its id has already been stored in the list. If so, a cycle has been detected and the query evaluation process on this subtree terminates. The process also terminates if no more link has to be resolved or, in case of vicious cycles, if only link elements with "L"-directive different from drop-element and keep-body are integrated locally. This means that no cycle has been found. For ordinary cycles, the evaluation can then be pruned and a "partial result" is returned for the link structure that has been expanded during this process. In case of vicious cycles, the detected link element is simply dropped and an empty answer will be returned because it is not feasible to determine any node that replaces the link element in a reasonable way.

If query shipping (remote evaluation) is applied, in case of a cycle, the same query is shipped around this cycle. If we have to start a cycle detection according to the two kinds of cycles, we assume that the currently processed query is assigned with a globally unique id. This could e.g. be achieved by an appropriate timestamp. The book-keeping list is now also mapped to this query id. Thus, we will know to which query evaluation process a cycle detection has been started. Then, if a query has to be shipped[3], it is attached with the appropriate query id. For any link that will be visited, we can thus check if it has already been stored in the book-keeping query on a specific server during the evaluation of a given query.

In this section, we described the detection of cycles in an abstract way. Later, in Section 5.6.3 it is explained how this detection strategies are realized in the proof-of-concept implementation.

[3]Recall that this will be a combination of an XPointer with the remaining query part.

4.5.4 Non-Terminating Queries

For evaluating queries that use the descendant(-or-self) or the following axis, in general all XLink elements (except those with "make-attribute" where the navigation continues along the attribute axis) must (recursively) be resolved. This can lead to searching huge parts of the Web. Since this *is* necessarily the case for guaranteeing completeness, this problem is not special to our approach, but applies to any approach that allows for including views on distributed XML resources. Since we cannot forbid to use the axes that traverse subtrees in arbitrary depth in queries, we propose the following handling: make the model as robust as possible, support pruning the search space by metadata knowledge, and let the evaluation detect cycles:

- Design: use a modeling with "make-attribute" in all cases where the resulting structure is not inherently nested – here, the considered axes end.
- Metadata about the element and attribute names and paths contained in a document (including recursive views) can help not only for detecting cycles, but also for pruning the search space.

4.6 Summary

In this section, we discussed on an abstract level how queries on distributed interlinked XML instances can be processed according to the dbxlink approach. First we argued that it suffices to focus on XPath queries without reverse axes. As the straightforward approach for querying, namely materializing the whole virtual instance in advance, is not suitable, we proposed a dynamic evaluation method. Also, we described how to cope with cyclic instances.

Although the basic idea for extending XPath query engines in order to respect XLinks is quite intuitive, there are numerous restrictions and special cases that have to be dealt with and show that this is not a trivial task. The following chapter discusses these issues and describes the proof-of-concept implementation based on the native XML database system *eXist*.

5 Detailed Querying and Implementation Issues

In the preceding chapter, the process of querying along interlinked XML sources has been described in an abstract way. Here, in this chapter, we discuss the details that have to be respected during this task.

First, we describe how a partial instance of the logical view evolves during query processing. We continue with a detailed discussion of the computation of relevant links for a given context as this is the fundamental part for extending the standard iterative XPath evaluation process. This also includes different fallback strategies. Then, we briefly describe how to handle ID and IDREF attributes for which it has to be guaranteed that IDs remain unique in the logical view and IDREFs still point to the referenced elements. Afterwards, it is explained how to ensure that equivalent queries deliver the same result wrt. XLink expansion. We conclude this chapter with a sketch of the proof-of-concept implementation based on our investigations.

5.1 Partial Instance

As described in the preceding chapter, for a given XPath query, only the relevant links that are needed to answer the query appropriately are resolved. This happens on demand if we have to handle a link during query evaluation. Hence, only a *partial instance* of the logical view induced by the interlinked resources has to be materialized. This is sufficient to answer the user's XPath query.

Consider Figure 5.1 for illustration. On the left, a distributed scenario of interlinked XML instances is shown. The small inner triangles symbolize link elements that reference parts of remote documents. Also, a query is shown which navigates through documents that are depicted on the lower right of the top document. Then, in the middle of the figure, only the parts of the distributed scenario that are relevant for the query are shown and the query has to resolve two links (assume that hybrid shipping is applied to all links). Now, we describe what happens while the query is processed:

1. We start to evaluate the query in the document on the top and during this process we have to traverse a link. Initially, the partial instance equals to the top document as indicated on the right side of Figure 5.1.

2. Then, we have to handle the first link and fetch the fragment it references. It is integrated into the partial instance. Now, as can been seen on the right, the partial instance has grown.

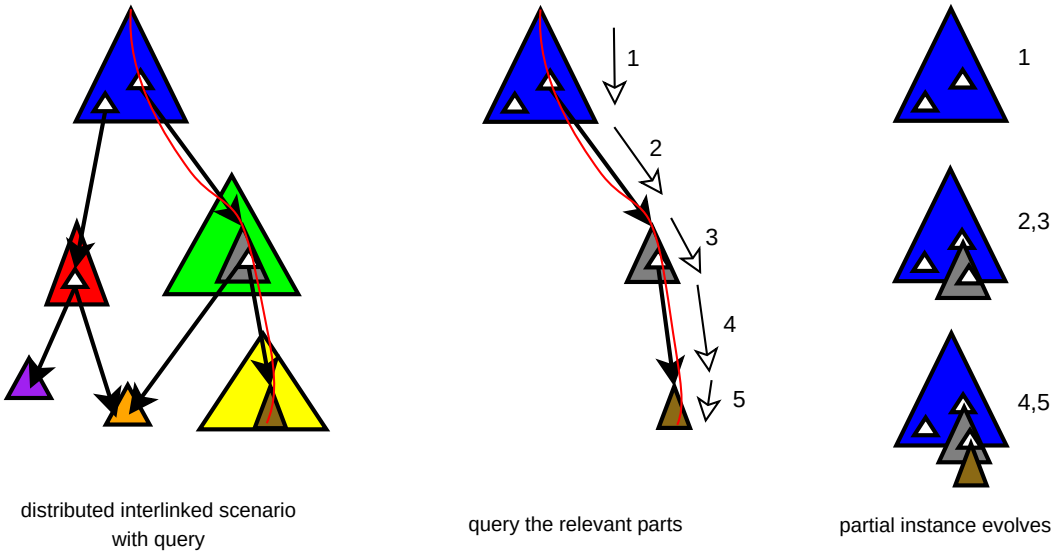


Figure 5.1: A Partial Instance is Materialized

3. The navigation continues in this integrated fragment until another link element has to be resolved.
4. The corresponding fragment is requested and again integrated accordingly.
5. Finally, inside this fragment, the query result is found.

Thus, the partial instance evolves during the query evaluation process. After a relevant link requiring hybrid shipping or data shipping has been resolved, the partial instance grows because XML fragments are integrated locally. If query shipping has been applied for a link, then we don't have to integrate fragments into the partial instance because the remote server will return nodes that contribute to the final query result. This will be discussed later in Chapter 6.

5.2 Extending the Stepwise Path Evaluation

As described in the previous chapter, an XPath expression consisting of n location steps is usually evaluated iteratively (step by step). Each step consists of an *axis*, a *nodetest* and an optional *predicate* (please refer to Section 2.2.1 for a short introduction to XPath):

axis::*nodetest*[*predicate*]

Given a current context, the *axis*, *nodetest* and *predicate* are evaluated successively: first, for each context node, the *axis* determines the "navigation direction", i.e. which nodes have to be selected next. For instance, the child axis selects the children of all context

nodes and following-sibling returns all sibling nodes that occur after a context node in document order. To these nodes, the *nodetest* (kind or name test) is applied and only matching nodes are kept which are then further filtered by the *predicate*. A *predicate* is basically considered as an expression yielding a boolean value for each node to be filtered determining if the node contributes to the next context. It can contain path expressions itself and in that case, the XPath query processor will evaluate the predicate's path expressions separately on the current context in the same stepwise manner as for an absolute path expression (this is also termed *branching*).

Given an XPath query *xpath-expr*, we assume that it has the following form:

$$\textit{xpath-expr} = \textit{xpath-expr}_1/\textit{step}_x/\textit{xpath-expr}_2$$

where $\textit{xpath-expr}_1$ produces a current context to which a next step $\textit{step}_x$ shall be applied, i.e. the steps contained in $\textit{xpath-expr}_1$ have already been processed. Then, $\textit{step}_x/\textit{xpath-expr}_2$ is the XPath query that remains to be evaluated.

The adaption to XLink processing for XPath query engines that evaluate XPath expressions stepwise (e.g. eXist, which is used for the proof-of-concept implementation) is achieved as follows. During query evaluation, we call processRelevantLinks() before the next location step ($\textit{step}_x$) is applied to the current context. This procedure is crucial as it enables the query processor to handle relevant XLinks. Any query engine that implements the stepwise XPath evaluation approach can thus be enabled for handling XLinks according to the dbxlink specification if its implementation is extended with code equivalent to processRelevantLinks() and the corresponding helper procedures and functions which will be explained in the following.

Procedure 5.1: processRelevantLinks

Input: A current context (set of nodes) C, $\textit{step}_x$, $\textit{xpath-expr}_2$.
Result: Relevant links resolved in advance for the next step.

```
1  begin
2      foreach element e ∈ C do
3          L ⟵ getRelevantLinks(e, step_x)
4          while L ≠ ∅ do
5              for ℓ ∈ L do
6                  resolveXLink(ℓ, step_x, xpath-expr_2)
7              end
8              L ⟵ getRelevantLinks(e, step_x)
9          end
10     end
11 end
```

The procedure processRelevantLinks() processes all relevant links for a given context (a set of nodes) depending on the next step to be applied. It takes the current context and the XPath expression parts $\textit{step}_x$ and $\textit{xpath-expr}_2$ as arguments. Iterating over all context nodes (line 2), for each context node the relevant links are computed by

getRelevantLinks() which is explained in the following section. These links are then each resolved by resolveXLink().

In case that the local or distributed evaluation mode is given for a link (i.e. data shipping or hybrid shipping shall be applied), the processing of links according to the dbxlink:transparent directives changes the partial instance as described in the preceding section: XML fragments are embedded into the local view and with these fragments, new relevant link elements have possibly been integrated. Hence, we have to repeat this loop until no more new link elements are found (lines 4-8). For instance, if the next axis to be applied (i.e. the axis contained in $step_x$) equals to the descendant axis, there might be link elements in some integrated XML fragment. These links will be detected by the call to getRelevantLinks() in line 8 as will be described in the following section. Thus, there are still some links to be resolved in this step (as checked in line 4) for which resolveXLink() will be called due to the inner loop. When processRelevantLinks() has finished its task, the partial instance has been extended with the relevant parts and the next step can be applied to the current context.

If remote evaluation has to be applied for a link, we combine the XPointer and the remaining XPath to a new query that is shipped to the referenced server as will be explained later in Section 6.2.2.

5.2.1 How to Obtain Relevant Link Elements for a Given Axis

We now show, for an arbitrary context element, which link elements have to be taken in account wrt. each axis, i.e. we explain getRelevantLinks(). This function takes a current context element and the next location step to be applied as arguments, and it returns the list of relevant links for that element wrt. the axis used in the next step. We assume that for element nodes there exists a method getLDirective() which returns the link's "L"-directive if the element is an XLink and an empty string otherwise. Note that we can assume that the XPath user query has been normalized according to the methods proposed in [OMFB02] (cf. Section 4.2) and thus does not contain any reverse axes. The only axes we need to consider are self, child, descendant, descendant-or-self, following-sibling, following and attribute.

self

The self axis can be considered as trivial because this axis selects all current context nodes. No new nodes are selected during such a navigation. Thus, the set of relevant links is empty and we do not handle this axis in getRelevantLinks().

child, **descendant**, **descendant-or-self**, **following-sibling**, **following**

These axes all select elements or text nodes. As shown in getRelevantLinks(), we assume that all nodes addressed by the given axis are stored in the variable "*tmpList*" of type node list. Which links contained in this node list have to be resolved? Generally, we can make the following basic observations:

- Any link element having "make-attribute" as "L"-directive can be ignored because such a link produces attributes in the virtual instance which are never selected by the considered axes.

Function 5.2: getRelevantLinks

```
   Input: An element e, step_x.
   Output: The links relevant for e wrt. step_x.
 1 begin
 2     links, tmpList ⟵ emptyList
 3     axis_x ⟵ the axis of step_x;     nodetest_x ⟵ the nodetest of step_x
 4     switch axis_x do
 5         case child
 6             tmpList ⟵ e.getChildren()
 7         case descendant
 8             tmpList ⟵ e.getDescendants()
 9         case descendant-or-self
10             tmpList ⟵ e.getDescendants()
11         case following-siblings
12             tmpList ⟵ e.getFollowingSiblings()
13         case following
14             tmpList ⟵ e.getFollowing()
15         case attribute
16             foreach link ∈ e.getChildren() do
17                 if link.getLDirective() = "make-attribute" and link matches nodetest_x then
18                     links.add(link)
19                 else if link.getLDirective() ∈ {"drop-element", "keep-body"} then
20                     links.add(link)
21             end
22     end
23     foreach link ∈ tmpList do
24         if link.getLDirective() ∈ {"drop-element", "keep-body"} then
25             links.add(link)
26         else if link.getLDirective() ∈ {"group-in-element", "duplicate-element"} then
27             if axis_x ∈ {"child", "following-sibling"} and link matches nodetest_x then
28                 links.add(link)
29             else if axis_x ∈ {"descendant", "descendant-or-self", "following"} then
30                 links.add(link)
31     end
32     return links
33 end
```

- For those link elements that are equipped with the directives "group-in-element" or "duplicate-element" we have to distinguish the considered axes.

  In case that child or following-sibling are given, such links are only relevant if they match the nodetest (name or kind test as described in Section 2.2.1) of the next step to be applied. The reason for this is that the modeling semantics of "group-in-element" and "duplicate-element" requires that the link element itself is kept and has to match the nodetest of the next step.

  The descendant(-or-self) or following axes can be considered as "recursive", i.e. they traverse complete subtrees. Thus, for these axes, links with "duplicate-element" and

"group-in-element" have to be taken in account in any case because their subtrees might contain matching nodes.

- Links having the directives "drop-element" or "keep-body" are replaced by the nodes referenced by the XPointer which are further processed according to the right-hand-directive. We do not know in advance if they qualify for the next step. Thus, no matter which of the considered axes is given, all links having any of these two kinds of directives are potentially relevant.

This means that for the axes that select element or text nodes, we can ignore any link having "make-attribute" as "L"-directive. Considering the other left-hand-directives, we first select the nodes wrt. the semantics of the next axis to be applied (lines 5-14) and handle any link element according to the observations given above (lines 23-31).

attribute
The attribute axis selects all attributes of context elements. Hence, we have to check if any link element produces new attributes. This is the case if a link child element contains a dbxlink:transparent attribute with an "L"-directive "make-attribute". Also, if a child is a link having "drop-element" or "keep-body", it can produce a new attribute for its parent (which is the context node) in case that attributes are selected by the XPointer. Link children with "group-in-element" or "duplicate-element" can be ignored because these elements are kept and hence never produce new attributes for their parents. Thus, the relevant links for this axis are found among the child elements that are equipped with "drop-element" or "keep-body". Obviously, also links with "make-attribute" are relevant. These links have to match the nodetest of the next step to be applied because for "make-attribute", the created attribute obtains the name of the link element (lines 15-21).

5.2.2 Special Case: Links that Turn their Parent into an XLink

The formal specification proposed in [BFM06a] allows attributes of the xlink or dbxlink namespaces to be contained in the result set of the XPointer. This can lead to scenarios as described with the following example:

```
<elem1>
  <elem2>
    <elem3>
     <dbxlink_attr dbxlink:transparent="drop-element insert-nodes"
       xlink:href="http://...#xpointer(//trans[@id='42']/@dbxlink:transparent)"/>
     <xlink_attr dbxlink:transparent="drop-element insert-nodes"
       xlink:href="http://...#xpointer(//link[@id='23']/@xlink:href)"/>
    </elem3>
  </elem2>
</elem1>
```

Two link elements (dbxlink_attr and xlink_attr) are children of the element elem3. If the result set of their XPointers each contains appropriate dbxlink:transparent and xlink:href

attributes, we will obtain an "*intermediate*" virtual instance in which the element elem3 has become a link element (recall that drop-element causes attributes to be added to the link's parent element):

```
<elem1>
  <elem2>
    <elem3 dbxlink:transparent="drop-element insert-nodes"
        xlink:href="http://...#xpointer(//target[@id='108']/@attr")/>
  </elem2>
</elem1>
```

This instance can be considered "intermediate" because it still contains a link element that has to be processed according to the formal dbxlink specification. After resolving the remaining link, the resulting virtual instance might look as follows:

```
<elem1>
  <elem2 attr="some_value"/>
</elem1>
```

Using the dynamic evaluation method based on the stepwise XPath evaluation proposed in this section, the query /elem1/elem2/@attr would not be processed correctly. Considering the context produced by /elem1/elem2, the next step to be applied is @attr (attribute::attr). According to the specifications described in the preceding paragraph, only link children of elem2 elements would be processed and thus the result would be empty.

Another related scenario that is even worse can be constructed as follows. Consider an XML instance located at the URI "http://www.example.org/linkbomb.xml" that consists of a single element:

```
<bomb dbxlink:transparent="drop-element insert-bodies"
        xlink:href="http://www.example.org/linkbomb.xml#xpointer(/bomb)"/>
```

The bomb is "fired" in another instance by simply referring to it in some element – it will "explode" all its ancestors, and finally the root node as explained below:

```
<a>
  <b>
    <detonator dbxlink:transparent="drop-element insert-nodes"
        xlink:href="http://www.example.org/linkbomb.xml#xpointer(/bomb)"/>
  </b>
</a>
```

The detonator link element is replaced by the bomb element, which is again an XLink element. After resolving the bomb element, i.e. after applying the "R"-directive "insert-bodies" we obtain a nodeset that contains the attributes of the bomb, and because of the "L"-directive "drop-element" they are added to the b parent (as the link itself is dropped):

```
<a>
  <b dbxlink:transparent="drop-element insert-bodies"
     xlink:href="http://www.example.org/linkbomb.xml#xpointer(/bomb)" />
</a>
```

Now, the b element is an XLink element, and the evaluation of the element's pointer does the same with the a element:

```
<a dbxlink:transparent="drop-element insert-bodies"
   xlink:href="http://www.example.org/linkbomb.xml#xpointer(/bomb)" />
```

that "blows up" the root node: the "process" ends when the root element is turned into an XLink element that adds attributes to its "parent", which is the document element.

At latest here, the formal semantics would "end", since a document node cannot be turned into an XLink; instead, the document would consist only of a nodelist containing element and attribute nodes. Therefore, we argue that such scenarios should be avoided because they are hard to be handled in the general case. Also, the declarative nature of the dbxlink approach suffers from these effects. Thus, we require that all xlink or dbxlink attributes found in the result set produced by the right-hand-directive are dropped.

5.3 Resolving of XLinks

Here, continuing the previous sections of this chapter, we assume that an XPath query *xpath-expr*=*xpath-expr*$_1$/*step*$_x$/*xpath-expr*$_2$ is given where both *xpath-expr*$_1$ and *xpath-expr*$_2$ consist of arbitrary many location steps. Assume again that this query is currently being evaluated and the steps in *xpath-expr*$_1$ produced a current context. For that, a set of relevant links wrt. the axis found in *step*$_x$ can be determined as described before. We now show how such a relevant link is processed conforming to the dynamic query evaluation introduced in the previous section and how to continue with the evaluation of the remaining query *step*$_x$/*xpath-expr*$_2$ as shown in resolveXLink().

5.3.1 Data and Hybrid Shipping

As already described in Section 4.1.3, the local evaluation strategy resembles data shipping according to the classification proposed in [FJK96], and analogously, the directive dbxlink:eval="distributed" can be considered as hybrid shipping. For these evaluation modes, the procedure resolveXLink() implements the following steps.

(i) Getting the Referenced Fragment

Firstly, we get the XML fragment referenced by the XLink/XPointer by evaluating the pointer's XPath expression contained in the xlink:href attribute locally or remotely according to dbxlink:eval. In case of local evaluation, the whole referenced instance is accessed and stored locally. We then obtain the desired XML fragment by stating the XPath query contained in the XPointer expression against the local copy of the referenced

Procedure 5.3: resolveXLink

```
   Input: A link element ℓ, step_x, xpath-expr_2
   Result: ℓ has been resolved.
 1 begin
 2     switch ℓ.getAttribute("dbxlink:eval") do
 3         case "local"
 4             href ⟵ ℓ.getAttribute("xlink:href")
 5             doc ⟵ getReferencedDocument(href)
 6             frag ⟵ getLocalXMLFragment(doc, href.getSubstringAfter("#"))
 7             addXMLFragment(frag, ℓ.getAttribute("dbxlink:transparent"))
 8         case "distributed"
 9             frag ⟵ getXMLFragment(ℓ.getAttribute("xlink:href"))
10             addXMLFragment(frag, ℓ.getAttribute("dbxlink:transparent"))
11         case "remote"
12             if canShipQuery(ℓ, step_x, xpath-expr_2) then
13                 q ⟵ buildQueryToShip(ℓ, step_x, xpath-expr_2)
14                 shipQuery(q)
15                 removeElementFromPartialInstance(ℓ)
16             else
17                 ℓ ⟵ ℓ.setAttribute("dbxlink:eval","distributed")
18                 resolveXLink(ℓ, step_x, xpath-expr_2)
19     end
20 end
```

instance. If the "distributed" strategy is desired we request the remote server to evaluate the XPointer's XPath expression.

(ii) Mapping and Adding the Fragment

As described in Section 3.2, the result of the XPointer is preprocessed according to the right-hand-directive which is specified by the dbxlink:transparent attribute. For instance, if "insert-bodies" is specified, we have to extract the attributes, text and element children of the received result elements. These intermediate fragments are then processed together with the link element according to the dbxlink:transparent attribute's left-hand-directive. Afterwards, in the partial instance, the result is integrated accordingly (cf. Section 5.1 and Figure 5.1 for an illustration of the partial instance).

(iii) Continuing the Local Query Evaluation

For both the local and distributed evaluation modes, the partial instance has been extended locally with the result set of the XPointers according to the modeling directives. Thus, in this cases, the XPath query processor can continue its local evaluation process seamlessly.

If dbxlink:eval="local" (lines 3-7), the auxiliary function getReferencedDocument() is used to request the whole referenced document from the remote server using the URI found

in the XLink's xlink:href attribute. It is stored locally and getLocalXMLFragment() is executed for evaluating the XPointer expression on it. Finally, the procedure addXMLFragment() maps the referenced fragment into the partial instance as specified by the right-hand-directive.

The directive dbxlink:eval="distributed" (lines 8-10) is handled similarly with the sole distinction that the referenced fragment is requested directly from the remote server.

5.3.2 Query Shipping

For query shipping (dbxlink:eval="remote", lines 11-18), we first check if this strategy is possible (line 12). If so, an adapted XPath query based on a combination of the XPointer expression ($xpath\text{-}expr_x$) and the remaining query parts ($step_x/xpath\text{-}expr_2$) is built and sent to the remote server (lines 13 and 14). Additionally, as the link has been handled, it is removed from the partial instance with an auxiliary procedure removeElementFromPartialInstance() (line 15). If query shipping is not possible, we fall back to the distributed mode (line 16-18). The remote evaluation strategy has several restrictions and special cases that have to be considered. Here, in this chapter, we describe the basic algorithms that are needed for implementing a dbxlink framework. As the query shipping task is rather complex, it deserves to be discussed separately in Chapter 6.

5.3.3 Fallback Strategies

Designers of distributed XML instances using the dbxlink approach should be aware that the evaluation strategies given in dbxlink:eval attributes of XLink elements have to be chosen carefully. Thus, the desired evaluation mode for a specific link should fit the capabilities of the remote server which hosts the referenced document. For query shipping, where we expect that any returned result contributes to the final result set of the user query, the remote server has to be able to process XPath queries along XLinks. Also during hybrid shipping, where the XPointer is evaluated remotely, it can be necessary to resolve XLinks in order to compute the referenced fragment that should be returned to the local server. Typically, in both these cases, the remote server should implement the dbxlink approach described in this thesis. Thus, designers of distributed XML instances should prefer data shipping for links that point to servers that can not handle XLinks. The only requirement for data shipping is that the whole referenced resource is accessible via HTTP GET requests.

Considering data shipping, the requesting of the whole document might be refused or it could result in a network connection timeout. In that case, we try to perform hybrid shipping, i.e. we try to get the referenced XML fragment by sending the XPointer expression $xpath\text{-}expr_x$.

A hybrid or remote shipping request could possibly result in an error if the remote server is not able to answer XPath queries. In that case we switch to data shipping and try to request the referenced instance as a whole. A special scenario that we do not handle explicitly could be observed if a remote host is under heavy computation load and thus XPath queries could take too long to be processed. Here, data shipping might

be preferable because requesting a resource could be quicker.

For the sake of simplicity, we have not included these fallback strategies in the procedure resolveXLink(). However, their implementation is straightforward.

5.4 Handling ID/IDREF Attributes

Consider the case where the user query contains an application of the id() function. Given a string v as argument, the application of id(v) selects the element that has an ID attribute with a value v. As queries have to be evaluated on the logical view induced by the XLinks, the appropriate element can be located in any referenced XML source. In order to find this element, in the worst case the whole logical view has to be traversed. Thus, we state the following *IDREF locality or explicitness* condition on the modeling:

1. Consider an attribute *attr* that is declared as an IDREF attribute in the DTD of an XML document. If a value for *attr* is given explicitly, the referred value must be contained as an ID value in the same document (validity of the source documents).

2. If the *external* schema of a document (cf. Section 2.3.4) declares an attribute as an IDREF attribute, then all its instances in the logical model must be contributed by either

   - an instance of an IDREF attribute in the schema of the local document or of a referenced document (note that by (1) this guarantees the existence of a matching ID in the same source document), or
   - the attribute value is induced by an XLink element.

According to this requirement, an id() application to an IDREF attribute is processed inside the document where the attribute is located. Anticipating the discussion about actual evaluation strategies, the evaluation of an id() call in the context of its original document is always safe: either (i) the matching ID can be found in the same document, or (ii) an explicit XLink addresses the target node. Nevertheless, for case (ii), as illustrated in Figure 5.2, an id() application may lead outside the explicitly referenced subtree in the referenced document.

5.4.1 IDREF(S) in Referenced Documents

For data shipping (where the referenced document is fetched together with its DTD information) and query shipping (the evaluation is pushed to the referenced document) each dereferencing takes place in its local context and is thus executed correctly. For hybrid shipping, if the id() application is in the XPointer, it refers to the logical model of the referenced document, and the same holds.

Only during the evaluation of XLinks where hybrid shipping is applied, the result nodes are transferred from the evaluation of an XPointer to the evaluation of the surrounding XPath expression. If the remaining query part contains an id() function call to be applied

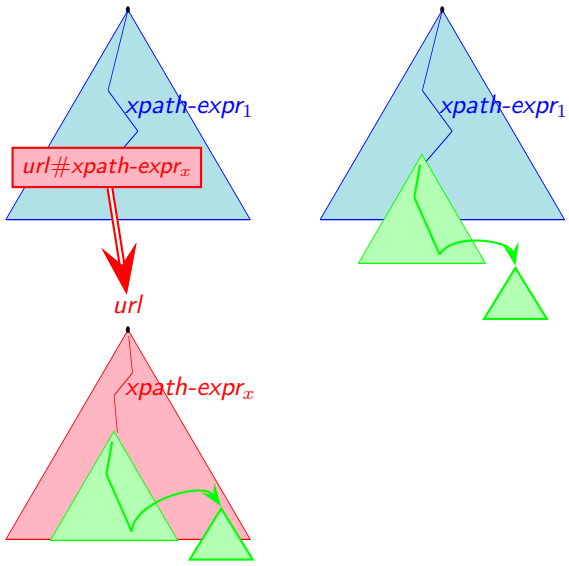


Figure 5.2: IDREF Step in the Referenced Fragment

to the *result* of the XPointer, the latter is already pulled out of its original context, and the IDREF-ID connection can be lost (cf. Figure 5.2). In such cases, either query shipping or data shipping must be applied.

A similar problem occurs with XPath expressions embedded in XQuery when selecting a reference attribute by an XPath query (e.g. in the for clause), and dereferencing it later. This can be circumvented by analyzing the query, and reformulating it equivalently by putting the dereferencing already in the XPath expression.

5.5 Result Set Normalization

The last step of XPath queries stated by users determines the nodes that are contained in the result set. For instance, the query

```
/countries/country/city[population>10000000]/name
```

generates a result set consisting of the name elements of all megacities contained in MONDIAL (i.e. cities with a population exceeding 10 million people). Usually, XLinks contained in the subtree of resulting nodes are not expanded. However, if the last step of a query includes predicates, it is possible that some links have been resolved in order to check the conditions of the predicates. Thus, depending on the user query, identical nodes could have a different structure. For instance, the two queries

```
/countries/country[id(@capital)/name='Berlin']
/countries/country[city/name='Berlin']
```

both select the country element representing Germany. Considering the queries, the result nodes differ such that for the first query, the capital link child has been mapped to an IDREF attribute of the same name:

```
<country car_code="D" area="356910" capital="dbxlinkID1">
  <name>Germany</name>
  <population>83536115</population>
  <cities dbxlink:transparent="drop-element insert-nodes"
      xlink:href="http://.../cities-D.xml#xpointer(/cities/city)" />
  ⋮
</country>
```

The same element returned by the second query would still contain the capital link child but the cities link would have been expanded to numerous city elements:

```
<country car_code="D" area="356910" >
  <name>Germany</name>
  <population>83536115</population>
  <capital dbxlink:transparent="make-attribute insert-nodes"
      xlink:href="http://.../cities-D.xml#xpointer(/cities/city[name='Berlin'])" />
  <city>
   <name>Berlin</name>
    <population year="95">3472009</population>
  </city>
  <city>
   <name>Hamburg</name>
  </city>
  ⋮
</country>
```

In order to avoid these differences for queries returning an equivalent result, we propose the following strategy. If the last step contains predicates, we evaluate these on a copy of the actual context node and check the boolean result value. Thus, the links are not expanded in the current context.

5.6 Implementation

In this section, we describe the proof-of-concept implementation of the dbxlink proposal which has been carried out while conducting this thesis. We sketch the extensions that have been integrated into the Java-based *eXist* [exi] system and show an example evaluation.

5.6.1 Extensions to eXist

We implemented a class org.exist.xquery.dbxlink.DBXLinkProcessor that contains all procedures and functions we discussed in this thesis. It provides a method process() which is the implementation of processRelevantLinks(). Thus, DBXLinkProcessor is the crucial extension that enables eXist for handling XLinks according to the dbxlink approach.

For extending the query engine according to the dynamic evaluation method, during the stepwise XPath processing the dbxlink behavior has to be added. eXist's class org.exist.xquery.PathExpr has a method eval() which iteratively processes the location steps of an XPath query. This method is applied to each step, starting with the document node. As described in Section 5.2, for a given context, before the next step is applied, processRelevantLinks() has to be called in order to integrate the dynamic evaluation of XPath queries. Thus, considering eXist, in the method org.exist.xquery.PathExpr.eval() an instance of class DBXLinkProcessor is created and its method process() which implements processRelevantLinks() is called. All necessary tasks including the resolving of relevant links and the appropriate materialization of the partial instance as described in Section 5.1 are then handled internally by private methods of DBXLinkProcessor.

Several further changes of specific classes have been undertaken in eXist:

- org.exist.dom.ElementImpl: The class representing an XML element has been extended with some auxiliary methods for accessing the values of dbxlink and xlink attributes. For instance, the method getXLinkHRef() returns the value of the xlink:href attribute.

- org.exist.dom.NodeSetHelper: For this class, two useful methods for manipulating attributes have been added.

- org.exist.xquery.XQuery: The algorithms proposed in [OMFB02] for eliminating reverse axes out of XPath expressions have been integrated into this class (cf. to Section 4.2). Thus, any XPath query stated by users is rewritten appropriately before it is processed.

- org.exist.xquery.util.DocUtils: Users usually state queries that start on a specific document which is initially copied to a temporary instance. This happens in this class. As explained in Section 5.1, a partial instance is materialized during the XPath query evaluation process. In the beginning, the copy of the starting document thus represents the partial instance which will be manipulated during query evaluation. The temporary document is deleted when the query evaluation is finished.

- org.exist.http.RESTServer: This class implements the HTTP interface of eXist servers. Special functionality for query shipping and cycle detection has been added to the HTTP GET interface.

- org.exist.xupdate: In this package, the classes Append, Insert, and Remove have been extended with special constructors that are needed for being able to use

these XUpdate [XML00] methods stand-alone for manipulating XML instances that represent the evolving partial instance. Section 5.1 showed that we have to integrate XML fragments into the partial instance and therefore we must be able to append fragments to existing nodes, insert them appropriately and remove obsolete link elements that have been resolved. This can be achieved by using the corresponding classes of the xupdate package.

5.6.2 Example Evaluation

We show an example evaluation in order to illustrate how we have implemented our results in the eXist system. For the most "intuitive" case, assume that the remote server is capable of answering XPath queries. Given an XLink with dbxlink:eval="distribute", the query *xpath-expr*$_x$ (representing the link's XPointer) is thus submitted to the remote server that transfers the result (not expanding XLink elements contained inside the result trees) which is then mapped according to the "L"-directive. Then, the local query evaluation continues with *step*$_x$ and *xpath-expr*$_2$. Consider the following example query whose evaluation is illustrated in Figure 5.3:

```
doc("/db/LinXIS/countries.xml")
                    /countries/country[@car_code="B"]/id(@capital)/population
```

This query returns the population of Belgium's capital, namely Brussels. Recall that we chose the modeling dbxlink:transparent="make-attribute insert-nodes" which turns the capital into a reference attribute to adhere to a "given" target DTD.

If stated on the server "linxis02", doc("/db/LinXIS/countries.xml") accesses the XML document that contains all countries. eXist offers collections (similar to directories in a file system) for storing XML files and in our setting, we store the files of MONDIAL's distributed version in distinguished collections "/db/LinXIS". Now, in order to prepare the dynamic evaluation process which will generate the partial instance, a copy of countries.xml is created. This temporary document will evolve into the partial instance needed for answering the stated query and thus we preserve the original file. All changes that are performed during this process are executed on the persistent file.

The next step countries then selects the root element of countries.xml (i.e. of its temporary copy). The subsequent step, country[@car_code="B"], actually consists of (i) an axis step country and (ii) the evaluation of the predicate for selecting Belgium. For (i), all subelements of the countries element have to be considered. These are only the country elements, which are non-XLink-elements.

Thus, the axis step results in all country elements. For evaluation of the predicate @car_code="B", all attributes of these elements *in the virtual model* have to be checked, searching for an attribute with name car_code and value "B". The country elements have already a car_code attribute.

Let's have a short look on the XLink subelements of the countries: cities has drop-element as "L"-directive, thus it could contribute an attribute. But its XPointer shows that it adds only city subelements. capital is equipped with make-attribute, thus it

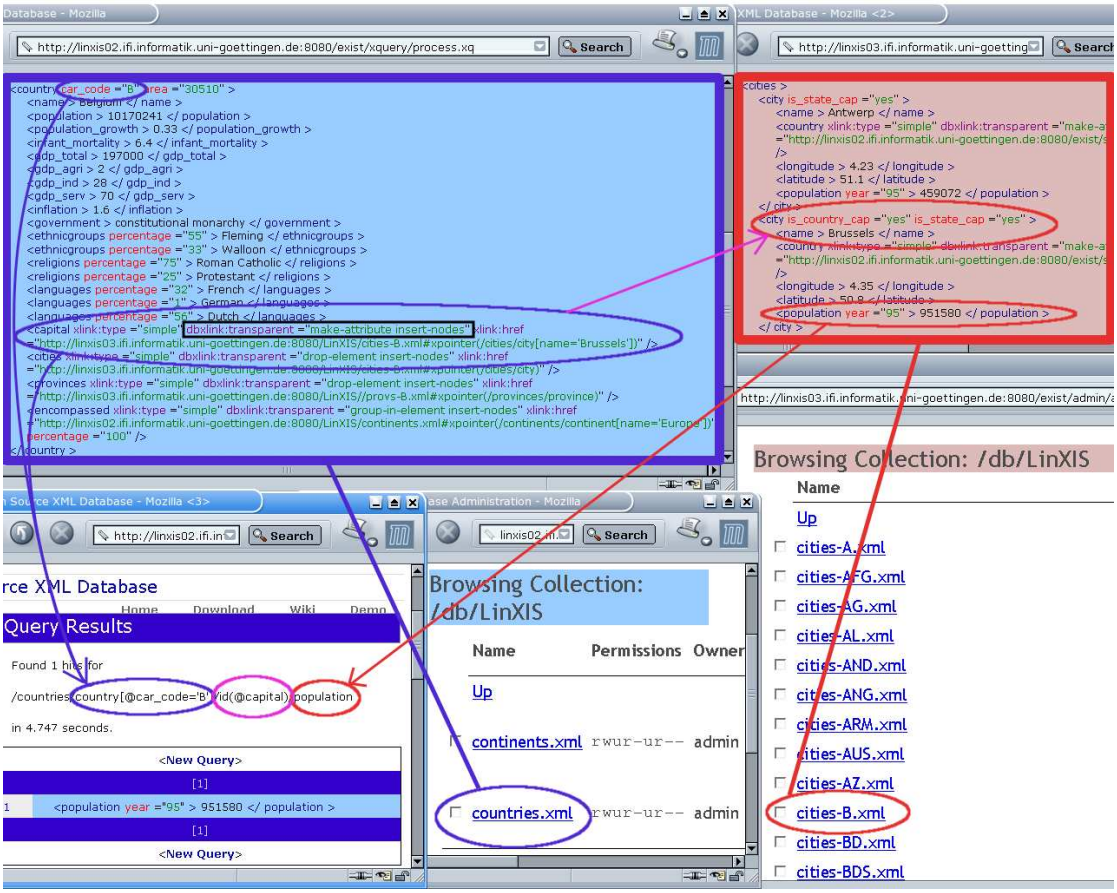


Figure 5.3: Querying the Distributed MONDIAL Database

contributes an attribute, with the name capital. For neighbor the "L"-directive duplicate-element is given, hence it stays as a subelement.

Thus, evaluating the predicate, only the country element for Belgium qualifies. The next step, @capital, again has to take into account the attributes of Belgium, and all its XLink subelements that contribute attributes. The capital XLink subelement is specified as make-attribute and thus has to be expanded: as illustrated in Figure 5.3, its XPointer

http://linxis03/cities-B.xml#xpointer(/cities/city[name='Brussels'])

is sent to the remote server which returns the city node for Brussels. The screenshot in Figure 5.4 illustrates the communication between the two servers, traced by the *Apache Axis TCPMonitor* [axi]. On the left hand side of the figure, the corresponding GET request for http://linxis03/cities-B.xml/cities/city[name="Brussels"] from the country server (linxis02) to the city server (linxis03) can be seen, whose result, i.e., the XML fragment representing Brussels, is shown on the right hand side.

Once the country server (linxis02) server has received the XML data for Brussels, it processes it according to dbxlink:transparent:="make-attribute insert-nodes": for the *Belgium*

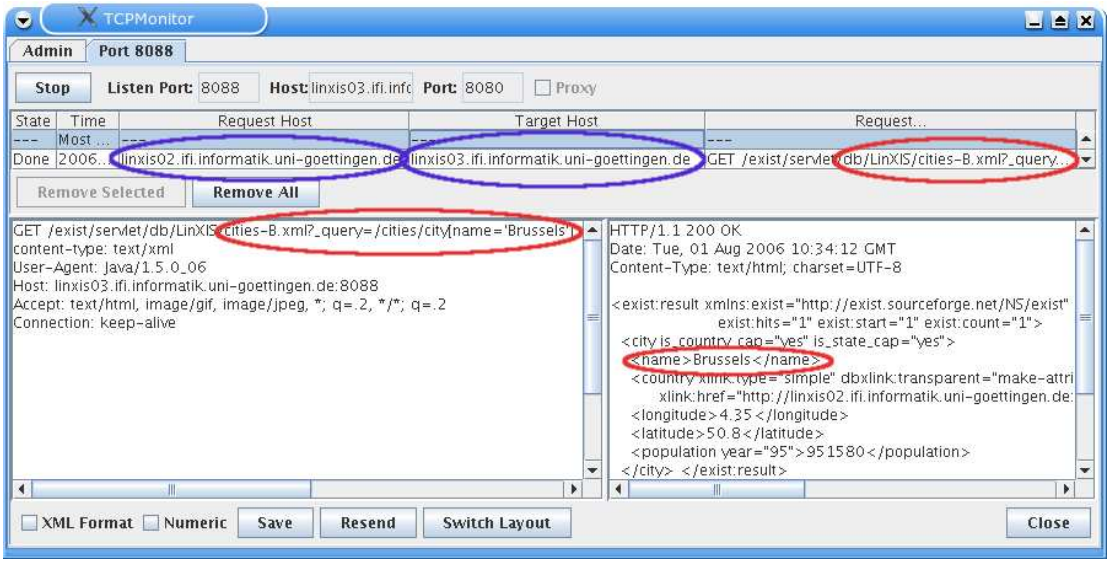


Figure 5.4: Communication: Answer Shipping

element, a capital reference attribute is created which points to the new, local *Brussels* node. These nodes are added to the document that represents the partial instance by appropriate XUpdate statements. The rest of the query is then id(@capital)/population, which dereferences the attribute and returns the population element of Brussels. No more link expansions are required.

5.6.3 Book-Keeping for Cycle Detection

In Section 4.5, an abstract description of the cycle detection strategies during query evaluation has been given. There, we assumed that any XML element can be globally identified by an id. However, there are cases where the ids of elements are not available. For instance, if XML fragments are requested for links where hybrid shipping has been applied, we only get a serialized ASCII stream of XML data. Thus, it is not possible to reconstruct the element ids. In this section, we will describe how we implemented an appropriate book-keeping that allows for detecting cycles in eXist.

Assume that during a given XPath query stated by the user a cycle detection process needs to be started for a specific link which has to be resolved (cf. Section 4.5 for a description of cycles). A new book-keeping list object is initialized and mapped to an actual timestamp which will represent the id of the current user query. Instead of the link element's internal id, the value of its xlink:href attribute is stored in the book-keeping list. Then the link is resolved according to its dbxlink:eval attribute.

In case of the local and distributed modes, the referenced XML fragments are integrated into the partial instance. Then, for each link contained in this new fragment to be resolved, we check if the book-keeping list contains the value of its xlink:href attribute. If not, then the value is added to the list and the link will be handled. If it is already

contained in the list, then we know that the same referenced fragment has already been integrated in this subtree. As the current link is part of this subtree, we can reason that we will have to handle the link again and thus a cycle has been detected. If there are no more links to be checked in the last fragment which has been added, then the corresponding value of the xlink:href attribute of the link which referenced this fragment can be removed from the book-keeping list, i.e. this subtree has been expanded appropriately. Thus, we always have an "implicit" list of the current processed "chain" of links in terms of the xlink:href attributes.

For query shipping, we have to attach the id of the user query to the query we have to ship. As we use the HTTP interface of eXist servers for delegating queries, the query id can be given as URI parameter in the HTTP GET request. For instance, in

$$\text{http://linxis03/cities-B.xml?}\underbrace{\text{\_query=/cities/city/name}}_{\text{shipped query}}\text{\&}\underbrace{\text{\_queryID=4210884000}}_{\text{user query id}}$$

two parameters are given as key-value pairs. In this example, the first parameter represents the query which is shipped to the remote server ("/cities/city/name"). The second parameter (_queryID) contains the query id. Then, the remote server knows that the query contained in the parameter _query shall be stated on cities-D.xml during the evaluation process of a user query with id "4210884000". Now, the server can check if it has a book-keeping list mapped to the given id and for each link that has to be handled, the same detection strategy as for the local and distributed modes is applied.

5.6.4 Results

For a proof-of-concept implementation, we extended the open source XML database system eXist. During the practical implementation process, many details and critical issues have been discovered and solved. As a result, we obtained a system which allows for querying along distributed XML instances connected by (simple) XLinks. At `http://www.dbis.informatik.uni-goettingen.de/linxis/`, a query interface for the MONDIAL testbed can be found. There, also some example queries are listed which allow for some straightforward tests.

The abstract strategies described in this work can be used to extend XML query engines that are based on the stepwise evaluation process of XPath queries. Thus, it is possible to enable them for handling XLinks during XPath queries according to the dbxlink approach.

6 Discussion of Query Shipping

This chapter is dedicated to the query shipping strategy used in the proof-of-concept implementation of dbxlink. Data and hybrid shipping have already been described in the two preceding chapters and it showed that they function in a way that a referenced XML fragment of an XLink (relevant wrt. a given user XPath query) is integrated into the local XML instance. Thus, according to its dbxlink:transparent attribute, the link is replaced by the referenced XML fragment leading to the materialization of a partial instance. However, the query shipping strategy for XLinks equipped with a dbxlink:eval="remote" attribute requires a different approach.

There are several cases where this strategy can not be applied or where it is not suitable to do so. These cases will be pointed out successively using examples for illustration. Then, we discuss some rewritings of query parts that are necessary for the query to be shipped. Also, we show how the final result set is obtained wrt. to query shipping results. The anatomy of a shipped query depends on the user query, the XPointer expression and the XLink's "L"-directive and we thus show how to construct an appropriate query to be shipped.

The results of the investigations outlined in this chapter are used in methods that have already been used in the procedure resolveXLink() (cf. Section 5.3). The function canShipQuery() contains the checks corresponding to the restrictions on query shipping and buildQueryToShip() then constructs the query to be shipped.

Here, in this chapter, we use the same terminology for XPath queries stated by users as in Chapter 5, i.e. $xpath\text{-}expr_1$, $step_x$ and $xpath\text{-}expr_2$ are given. Also, $step_x$ will be of particular interest and thus we assume that it is constructed as follows:

$$axis_x::nodetest_x[predicate_x].$$

6.1 Restrictions on Query Shipping

Basically, a shipped query consists of two parts, namely the XPointer expression and the remaining XPath query stated by the user. For query shipping, several restrictions exist and we first give a concise overview of the cases when not to ship a query.

- If local data is contained in the link and the "L"-directive is different from drop-element, then shipping is prohibited.
- We do not ship a query if the remaining query contains the following axis.
- The following-sibling axis and position checks must not be used in the first step of the remaining query.

The first restriction depends on the considered link element while the next two depend on the query stated by the user. These constraints are now described in detail and they are checked in the already mentioned function canShipQuery() which is given at the end of this section. If query shipping can not be applied, then the fallback strategy is given by hybrid shipping as implemented in the procedure resolveXLink() (cf. Section 5.3).

6.1.1 Local Data of Links

Consider a link which contains *local data*, i.e. non-xlink- and non-dbxlink-attributes, text children or subelements. If query shipping should be applied for this link element, intuitively, both the link's local data and the remote parts have to be respected. Due to this, several problems arise. For instance, both the local and remote data might contribute to the result set of the given user query. Also, if the query is shipped, the local data is not "reachable" anymore and sometimes it is necessary to have the data "in one place".

Only links with drop-element as "L"-directive do not have to be considered here. According to the modeling semantics of this directive, the local link element is dropped completely and thus its local data is not relevant for the query. However, for a link which has any of the other "L"-directives and if it also has local data, we argue that query shipping should not be applied and therefore give some illustrating examples.

Remark. One possibility to overcome this problem would be to ship the query and any local data contained in the link's subtree. Then, we would have all relevant data in one place, namely on the remote server. On the other hand, the idea of query shipping intends to reduce network traffic and not to ship any data which might not be accepted by remote servers anyway.

group-in-element. Any expression contained in $predicate_x$ has to be evaluated locally because if the link contains a body, then its local attributes or subelements might satisfy the predicate. However, there might be cases where it *also* has to be checked remotely. An example will clarify this issue.

The city elements could be modeled with an additional temperature link for integrating the actual temperature into the local data. For instance, consider Berlin:

```
<city>
  <name>Berlin</name>
  <population year="95">3472009</population>
  <temperature time="now" dbxlink:eval="remote"
    dbxlink:transparent="group-in-element insert-bodies"
    xlink:href="http://.../temp.xml#xpointer(//city[name='Berlin']/temp)" />
  ⋮
</city>
```

With this modeling, in the virtual instance, the city element's child temperature would be enriched with the actual temperature in degree Celsius obtained from a remote file "temp.xml". Assume that this file is structured as follows:

```
<temp_measurements>
  <city>
    <name>Berlin</name>
    <temp>18</temp>
  </city>
  ⋮
</temp_measurements>
```

Thus, the virtual instance contains the referenced temperature:

```
<city>
  <name>Berlin</name>
  <population year="95">3472009</population>
  <temperature time="now">18</temperature>
  ⋮
</city>
```

Now, consider the following example query which selects the name of all cities where the actual temperature is below 20 degrees:

```
//city[temperature/@time="now" and temperature/text()<20]/name
```

As Berlin qualifies wrt. the predicate, we expect its name to be contained in the result set. If evaluated via data or hybrid shipping, where a partial instance is materialized locally, the computation delivers the correct result in a straightforward way because all relevant nodes are found in one place. But in case of query shipping, we have to be aware that in general, the predicate has to be checked both locally and remotely. In the example above, a predicate containing a conjunction of predicates like "[$cond_1$ and $cond_2$]" is given where $cond_1$ is satisfied locally by the time attribute and $cond_2$ qualifies remotely by the referenced actual temperature.

This example shows why it is appropriate to fall back to hybrid shipping if links having local data and equipped with group-in-element as "L"-directive are given. The main reason for this is due to the necessity that relevant data should be accessible in one place, be it remotely or locally. The following paragraph shows similar examples for the other "L"-directives.

duplicate-element, keep-body and make-attribute. Recall the example already used in Section 3.2.1 for illustrating the modeling of duplicate-element where extra cities referenced as town elements in a file germantowns.xml are included for Germany:

```
<country>
  <name>Germany</name>
  ⋮
  <city source="not approved" dbxlink:eval="remote"
      dbxlink:transparent="duplicate-element insert-bodies"
      xlink:href="http://.../germantowns.xml#xpointer(//town)" />
  ⋮
</country>
```

The city element is duplicated according to the number of referenced elements representing these German towns. Thus, in the virtual instance induced by this fragment, all city elements are enriched with an source attribute which stems from the original link element:

```
<country>
  <name>Germany</name>
  ⋮
  <city source="not approved">
    <name>Göttingen</name>
    <population>129051</population>
  </city>
  <city source="not approved">
    <name>Braunschweig</name>
    <population>245273</population>
  </city>
  ⋮
</country>
```

If the city link would have an attribute dbxlink:transparent="keep-body insert-nodes", the induced virtual instance would have a similar structure consisting of town elements because keep-body forces the link element to be replaced by the referenced town elements to which the source attributes are added. Thus, a similar example can also be constructed for keep-body.

Now, assume that the following query is stated, querying for all source attributes:

```
//country[@car_code='D']/city/@source
```

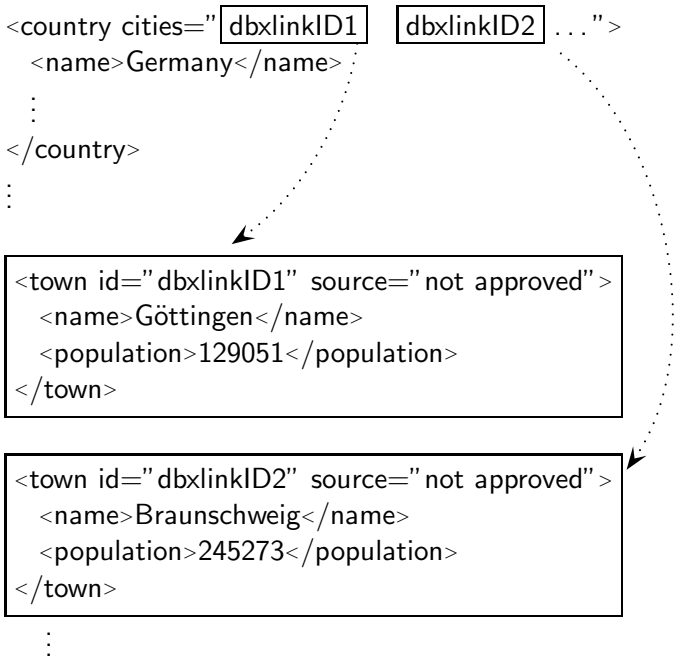
We have not yet discussed how the query to be shipped is built but the idea is to delegate a combination of the XPointer and the rest of the query to the remote server. For the example query given above, the final result set consists of source attributes which amount to the number of referenced towns. Obviously, these attributes would not be selected by a shipped query because remotely, no such attributes reside. In this example, we would have to request the remote server how many element nodes are returned by the

XPointer in order to add the appropriate amount of source attributes to the final query result set. This would be equivalent to a hybrid shipping which requests the fragment referenced by the link's XPointer. Thus, it shows already in this simple example that a query shipping is not feasible. We argue that in this case we should fall back to hybrid shipping in order to build the partial instance locally.

Links having local data and the "L"-directive make-attribute can create a similar situation. Consider an alternative modeling of the cities link for referencing towns with make-attribute:

```
<cities source="not approved" dbxlink:eval="remote"
    dbxlink:transparent="make-attribute insert-nodes"
    xlink:href="http://.../germantowns.xml#xpointer(//town)" />
```

All referenced city elements are enriched with the local data items. However, in the virtual instance, an appropriate IDREFS attribute references the town elements:



If for this fragment the query

```
//country[@car_code='D']/id(@cities)/@source
```

has to be evaluated, then it is obvious that the source attributes are not found remotely. Thus, also for links with make-attribute which have local data it is not feasible to apply query shipping.

In this section, several examples showed why not to ship a query for links having local data and an "L"-directive different from drop-element. Also from the modeling perspective it is inappropriate to ship a query in such situations because if a query has to be shipped for a link, we do not want to evaluate the link locally but expect the remote server to take care of this task.

6.1.2 Remaining Queries that Contain the following Axis

Given a context node c, the following axis selects all nodes that are not descendants of c and occur after it in document order. Thus, if shipped to a remote server and evaluated in a different document, this axis could select false result nodes that are not part of the virtual instance. On the other hand, matching nodes of the local view would not be selected if evaluated remotely. Consider the following XPath expression for illustration:

```
//country[@car_code='D']/city[name='Berlin']/population/following::name
```

As before, this query has to be executed on the distributed version of MONDIAL:

```
<country car_code="D" area="356910">
  <name>Germany</name>
  <population>83536115</population>
  <cities dbxlink:eval="remote" dbxlink:transparent="drop-element insert-nodes"
      xlink:href="http://.../cities-D.xml#xpointer(/cities/city)" />
  ⋮
</country>
```

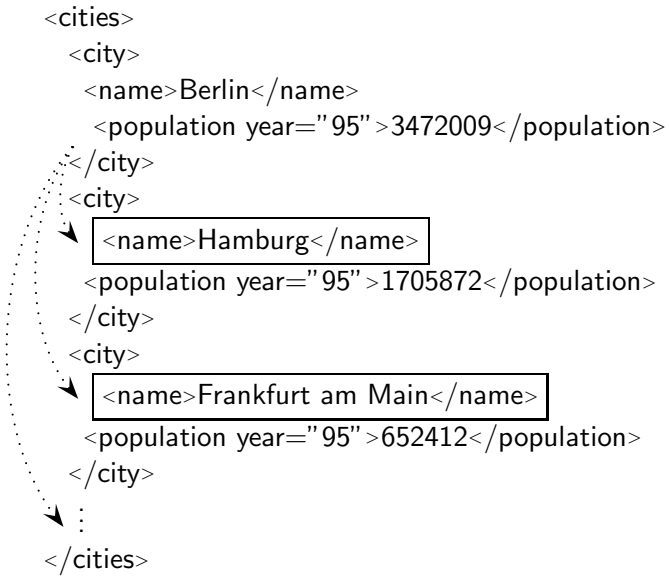
Thus, in the logical view, the referenced city elements replace the cities link element. In the virtual instance, the above XML fragment would look as follows:

```
<country car_code="D" area="356910">
  <name>Germany</name>
  <population>83536115</population>
  <city>
    <name>Berlin</name>
    <population year="95">3472009</population>
  </city>
  <city>
    <name>Hamburg</name>
    <population year="95">1705872</population>
  </city>
  ⋮
</country>
```

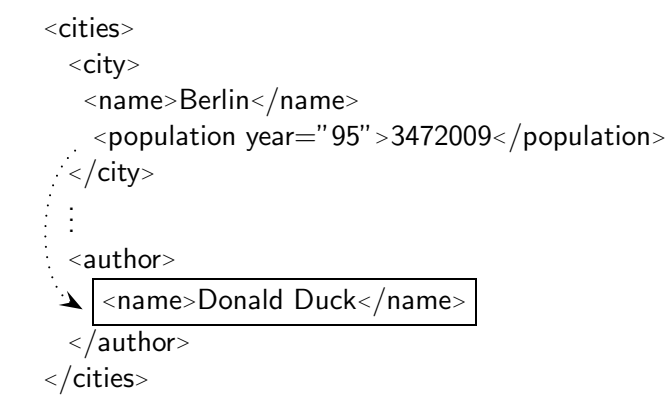
It is this virtual instance on which the query given above has to be evaluated. Considering data shipping and hybrid shipping, a partial instance is built locally and the data is

queried in one place yielding all name elements that follow the population element of Berlin. However, with query shipping, the situation is different.

While querying, we step on the cities link element and would ship a specific query. The remaining query (*xpath-expr*$_2$) ends with "population/following::name" and if involved into a combined query to be shipped, this part would be evaluated remotely on cities-D.xml. Thus, in the remote document, we would navigate to Berlin's population and from there, according to the following axis, we would select all following name elements as indicated by the dotted arrows below:



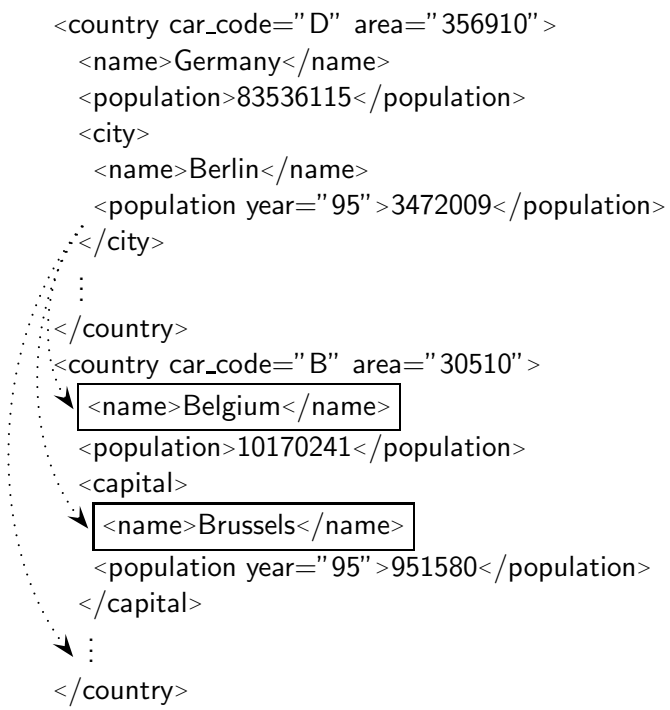
Here, the name elements of Hamburg, Frankfurt and all other cities would be selected because they are not contained in the subtree of Berlin's population element and occur after it in document. Now, assume that cities-D.xml is extended in the following way:



The author "Donald Duck" has added its name as a child of an author element to the document. This element is not part of the virtual instance induced by the country and city data because the cities link element's XPointer addresses only city elements. If the

query part following::name is processed on this modified document remotely, it would also select the author's name element because it is addressed by the following axis. Thus, a usage of the following axis in shipped queries can lead to false results.

On the other hand, any name element found in the local document or in any other included data would not be selected because the query has been delegated to a specific remote server. Here, any name element occurring in the virtual instance after Berlin's population element, like the names of several countries, should be contained in the result set. However, they could not be selected on the remote server because the local data is not "reachable" anymore after shipping the query. Some name elements that should have been selected in the logical view as result nodes are marked in the following fragment:



In order to summarize, we can state that we prohibit a shipping of a query if we have to combine the XPointer with a remaining query that contains the following axis because false nodes could be selected on the remote server and matching nodes in the local instance are mistakenly omitted.

6.1.3 Considering following-siblings and Position Checks

As we have seen in the previous section, a query that is shipped to the remote server can not "return" to the local instance for selecting matching nodes. This means that a query shipping usually delegates the remaining query computation on an XML subtree to another host. In case that the remaining query which will be part of the query to be shipped *starts* with a step consisting of the following-sibling axis, the local siblings of the XML tree which replaces or enriches the link element have to be tested if they match the next step. Again, we use the cities link element as an example:

```
<country car_code="D" area="356910">
  <name>Germany</name>
  <population>83536115</population>
  <cities dbxlink:eval="remote" dbxlink:transparent="drop-element insert-nodes"
      xlink:href="http://.../cities-D.xml#xpointer(/cities/city)" />
  ⋮
</country>
```

Now, assume we have stated the following query for selecting the name of the cities that have a neighbor element among their following siblings:

```
//country[@car_code='D']/city[following-sibling::neighbor]/name
```

If a remaining query starting with city[following-sibling::neighbor] would be shipped to the remote server that hosts the cities file for Germany, it would be executed after the XPointer has been evaluated – here, this would be an XML instance that does not contain any neighbor elements (cf. cities-D.xml in the distributed version of MONDIAL). Thus, the returned query result would be empty because the predicate that contains the following-sibling axis would not be satisfied. On the other hand, we omitted matching elements locally. There are neighbor elements in the logical view that follow the German city elements and thus the predicate is satisfied by these (ignore by now that neighbor is an XLink itself):

```
<country car_code="D" area="356910">
  <name>Germany</name>
  <population>83536115</population>
   <city>
    <name>Berlin</name>
    <population year="95">3472009</population>
   </city>
   ⋮
   <city>
    <name>Jena</name>
    <population year="95">102204</population>
   </city>
   <neighbor dbxlink:transparent="..." xlink:href="..." borderlength="167" />
   ⋮
</country>
```

Hence, we can not ship a query that starts with following-sibling because, similar to the case concerning the following axis, false nodes might be selected and relevant nodes could be omitted. Also, we prohibit shipping if the remaining query starts with a step that

contains a position expression. We then have to evaluate the remaining query locally because the position has to be evaluated wrt. the document order in the local view. The same holds for the context functions position() and last().

Note that the following-sibling axis and position checks are not harmful if they are not part of the next step to be executed. In that case, the tests take place in the context of the link's subtree which has no local data and which can thus be located entirely on the remote server. All other axes (child, descendant(-or-self), attribute) are harmless if executed as part of the next step wrt. a considered link element. The reason for this is that, from the viewpoint of the XML tree, these axes either navigate to a deeper level in the referenced subtree (child and descendant(-or-self)) or are directly related to the current link element (attribute).

Remark. Consider the following query:

/countries/country[@car_code='D']/descendant-or-self::node()/. . .
. . . /descendant-or-self::node()/following-sibling::country.

It consists of several steps "descendant-or-self::node()" applied to the country element representing Germany. Due to the repeated implicit "self" step, the following-sibling::country location step at the end of this query selects the country that follows Germany in document order. We also forbid such a usage of following-sibling for queries to be shipped but for the sake of clarity, we don't test these in the function canShipQuery(). However, it is just a technical issue to implement these kind of tests that have to analyze the given query syntactically.

Function 6.1: canShipQuery

Input: The link ℓ and the remaining query parts ($step_x$ and $xpath\text{-}expr_2$).
Output: false, if the query can not be shipped, true otherwise.

```
1  begin
2      if xpath-expr2 is empty then
3          return false
4      else if ℓ has local data and ℓ.getLDirective() ≠ "drop-element" then
5          return false
6      else if step_x.predicate or xpath-expr2 contain following then
7          return false
8      else if step_x.predicate or xpath-expr2 start with following-sibling or position checks then
9          return false
10     else
11         return true
12 end
```

6.1.4 Summary

In this section, we have shown several restrictions on query shipping. These are checked in the function canShipQuery() wrt. a given user query and a current link to be resolved.

Remark. We assume that the remaining query *xpath-expr*$_2$ is not empty. Thus, only if the navigation continues on the referenced subtree, the query is shipped. Otherwise, it is preferable to apply any of the other shipping methods.

There are no other cases where query shipping has to be avoided but in some situations we have to rewrite the query. The related cases are discussed in the following section.

6.2 Rewritings and Result Integration

Here, we describe several cases where a query to be shipped has to be adapted slightly and we discuss how result nodes obtained by query shipping have to be integrated wrt. the final result. These issues are only briefly sketched because they are rather obvious.

6.2.1 Absolute Document References

In XPath, semi-joins can be expressed by using absolute paths in a predicate expression. For example, the following query selects the names of all cities that have more inhabitants than Germany:

$$\texttt{//city[population>}\underbrace{\texttt{//country[name='Germany']/population}}_{\text{absolute path expression}}\texttt{]/name}$$

From the modeling perspective, we assume that any absolute path expression contained in predicates has to be applied to the same document where the query starts. Any relevant subtree should be reachable from that document. Thus, before the query is evaluated, any absolute path expression found as part of a predicate is extended with a doc() function call containing the URI of the document in which the query evaluation starts.

In our testbed (cf. Chapter 1 for a description how MONDIAL is distributed in our test scenario), we would state the example query against the file mondial-root.xml. Thus, we would extend the absolute path in the predicate with the URI of the server that hosts this file (linxis01) including the path to the file itself:

```
//city[population>doc('http://linxis01/.../mondial-root.xml')
                    //country[name='Germany']/population]/name
```

If a query containing the predicate has to be shipped, it will then be sent to linxis01 to be executed on mondial-root.xml from where the countries are reachable via an appropriate XLink that points to linxis02 (the server hosting countries.xml).

6.2.2 Local and Remote Result Nodes of Links

Generally, query shipping results must respect document order. Thus, if for a link query shipping has been applied and a non-empty result set is obtained, these items must occur in the final result wrt. the document order. In order to achieve this, for each link that has

been treated with query shipping, the shipping result is marked with the link's position in document order. At the end of the query evaluation, any partial result obtained by query shipping can thus be integrated appropriately.

6.3 Building the Query to be Shipped

In this section, we discuss how the query to be shipped has to be built. The results of these investigations can be found in the function buildQueryToShip() at the end of this section.

First, we can draw some some consequences out of the restrictions described in the preceding sections. For a query to be shipped, the following constraints hold:

- If the "L"-directive is different from drop-element, we can assume that the link has no local data (non-xlink- and non-dbxlink-attributes, or subelements). Thus, for query shipping we do not have to take local data of links into account.
- In the remaining query to be combined with the XPointer, the first step does not contain the following-sibling axis and position checks.
- The remaining query does not include the following axis.
- *xpath-expr*$_2$ is not empty.

Now, we assume that an XLink with dbxlink:eval="remote" is given and all conditions that allow for query shipping are satisfied. In order to show how the adapted query that will be shipped for this link is obtained, we discuss all dbxlink:transparent directives in separate paragraphs. Recall that *step*$_x$= *axis*$_x$::*nodetest*$_x$[*predicate*$_x$].

(i) drop-element

The modeling semantics of this directive requires the link element to be dropped completely and to be replaced by the nodes obtained after applying the "R"-directive (insert-nodes or insert-bodies) to the referenced data.

For this "L"-directive, as the link element is dropped, we can assume that *step*$_x$ continues on the nodes obtained after applying the "R"-directive to the referenced fragment because these replace the link.

In order to combine the XPointer and the remaining XPath query stated by the user to a new XPath query that can be shipped, we have to distinguish between the two possible "R"-directives and show how the next axis to be applied (*axis*$_x$) affects the structure of the shipped query.

insert-nodes:
The nodes referenced by the XPointer expression *xpath-expr*$_x$ are not manipulated and hence replace the link element in a straightforward way. Thus, in case that *axis*$_x$ equals to child or attribute, we first have to get the appropriate nodes among the referenced

nodes according to *axis*$_x$. In order to achieve this, if *axis*$_x$=child (or *axis*$_x$=attribute), a step self::element() (or self::attribute()) is appended to *xpath-expr*$_x$ for getting the relevant nodes wrt. the kind of nodes that is required by *axis*$_x$. Afterwards, the node test and the predicate of *step*$_x$ is checked for these nodes and then we continue with *xpath-expr*$_2$[1]:

axis$_x$=child:
xpath-expr$_x$/self::element()/self::*nodetest*$_x$[*predicate*$_x$]/*xpath-expr*$_2$

axis$_x$=attribute:
xpath-expr$_x$/self::attribute()/self::*nodetest*$_x$[*predicate*$_x$]/*xpath-expr*$_2$

If *axis*$_x$ is given by the descendant axis, which recursively traverses the whole referenced subtree, we have to ensure that both the nodes addressed by the pointer are tested and the navigation continues (which can be achieved by using descendant-or-self). Here, an explicit step selecting only the elements is not needed because implicitly, descendant-or-self does this task:

axis$_x$=descendant:
xpath-expr$_x$/descendant-or-self::*nodetest*$_x$[*predicate*$_x$]/*xpath-expr*$_2$

In case that *axis*$_x$=descendant-or-self, a different approach is needed. Consider a query that addresses all German cities with more than 500000 inhabitants for selecting their names:

/countries/country[name='Germany']//city[population>500000]/name,

which is equivalent to the rewritten query

/countries/country[name='Germany']/descendant-or-self::node()
/city[population>500000]/name.

The cities link has an XPointer /cities/city (ignoring the host and file parts of the URI) and thus, if we would apply the same strategy like for *axis*$_x$=descendant given above, then the following query would be shipped:

/cities/city/descendant-or-self::node()/city[population>500000]/name.

Remotely, evaluated on the referenced file cities-D.xml, this query addresses any city element that occurs as subelement of the city elements that are nested in the cities root element (though, however, there are no such elements). Unfortunately, due to the part ...city/descendant-or-self::node()/city..., the "real" city elements are not selected because in the query assumed to be shipped, city elements are required to have other city descendants. Thus, in order to apply *step*$_x$ and *xpath-expr*$_2$ directly to the nodes selected by *xpath-expr*$_x$, an document element constructor is used:

[1] In XPath, if the self axis is combined with a name test, only element nodes are selected. Thus, we assume that in dbxlink systems the query shipping strategy allows for using self in combination with name tests also for attributes.

axis$_x$=descendant-or-self:
document {*xpath-expr*$_x$/self::element()}/*step*$_x$/*xpath-expr*$_2$

With this approach, the remote server (which we assume to be able to handle such XQuery expressions) first computes the nodes selected by the pointer (*xpath-expr*$_x$), among which the elements are selected because only for them the navigation can continue with descendant-or-self. A new document node is constructed (to which no validation is applied, cf. [XQu06]) that serves as "input" for the remaining query (which is *step*$_x$/*xpath-expr*$_2$). Note that for this approach, two evaluations (one for the pointer and one for *xpath-expr*$_x$) are needed. Thus, for the other axes, we preferred a concatenation of *xpath-expr*$_x$ and *xpath-expr*$_2$ without document constructor in order to require just a single straightforward XPath computation.

For the example query selecting German cities with more than 500000 inhabitants, the following query would be shipped when Germany's cities link has to be resolved:

document {/cities/city/self::element()}//city[population>500000]/name

Thus, remotely, first the following XML fragment containing all German cities is created by the document constructor:

```
<city>
  <name>Berlin</name>
  <population year="95">3472009</population>
</city>
<city>
  <name>Hamburg</name>
  <population year="95">1705872</population>
</city>
  ⋮
```

Note that no root element is created. However, the XQuery specification [XQu06] allows for such constructs and the remaining query //city[population>500000]/name can be applied to this fragment yielding the desired nodes.

insert-bodies:
Using this "R"-directive, only the bodies (i.e. the text and element children, and the non-XLink attributes) of all element nodes contained in the result set of the XPointer are kept. These are then used for replacing the link element. This means that the child and attribute axes have been applied implicitly to each referenced element, yielding their children and attributes. In case that *axis*$_x$ is among child, attribute or descendant, we therefore can apply *step*$_x$ and *xpath-expr*$_2$ directly to *xpath-expr*$_x$ because *step*$_x$ can be considered as the implicit step for selecting the body:

axis$_x$ ∈ {child, attribute, descendant}:
xpath-expr$_x$/*step*$_x$/*xpath-expr*$_2$

Assuming that the cities link has insert-bodies as "R"-directive and an XPointer /cities instead of /cities/city, we give some examples how the shipped query is constructed:

user query: /countries/country[@car_code='D']/city/name
shipped query: /cities/city/name

user query: /countries/country[@car_code='D']/descendant::city/name
shipped query: /cities/descendant::city/name

For descendant-or-self we have to be aware that only the element children of the referenced nodes are used as "input" for the remaining query and thus we again use an appropriate document constructor:

$axis_x$=descendant-or-self:
document {$xpath\text{-}expr_x$/child::element()}/$step_x$/$xpath\text{-}expr_2$

Below, examples similar to those from above (using the modified cities link) are given.

user query: /countries/country[@car_code='D']//city/name
= /countries/country[@car_code='D']/descendant-or-self::node()/city/name
shipped query: document {/cities/child::element()}//city/name

user query: /countries/country[@car_code='D']/descendant-or-self::city/name
shipped query: document {/cities/child::element()}/descendant-or-self::city/name

(ii) keep-body

This directive is similar to drop-element but instead of dropping the whole link element, its body is kept, i.e. its subelements, text nodes and attributes have to be added to the nodes that are going to replace the link element. As stated in Section 6.1.1, links with local data which are equipped with this "L"-directive forbid query shipping. If query shipping can be applied, i.e. the link element has no local data, we ship in the same way as described for drop-element because in this case, the modeling semantics is the same.

(iii) group-in-element

In the logical model, the "L"-directive group-in-element embeds *all* referenced nodes in the link element hull. The link element has no local data and thus, we let the remote server "reconstruct" this element in order to have the embedded nodes in one place. Therefore, we assume that the remote server is able to map the referenced nodes into the surrounding element appropriately wrt. the result mappings given in the dbxlink approach, e.g. it should concatenate values of attributes having the same name.

insert-nodes:
If executed remotely, the following usage of a document constructor creates an element with name *link* that "reconstructs" the link element. The nodes referenced by the XPointer are embedded into it and thus the remaining query can be applied in a straightforward way:

document {element *link* {*xpath-expr*$_x$}}/*step*$_x$/*xpath-expr*$_2$

For illustration, consider some example queries involving the cities link which is assumed to be given with dbxlink:transparent="group-in-element insert-nodes":

user query: /countries/country[name='Germany']//city/name
shipped query: document {element cities {/cities/city}}//city/name

user query: /countries/country[name='Germany']/cities/city/name
shipped query: document {element cities {/cities/city}}/cities/city/name

On the remote server, the expression document {element cities {/cities/city}} creates the following XML fragment to which the remaining query can be applied yielding the desired result:

```
<cities>
  <city>
    <name>Berlin</name>
    <population year="95">3472009</population>
  </city>
  <city>
    <name>Hamburg</name>
    <population year="95">1705872</population>
  </city>
  ⋮
</cities>
```

insert-bodies:
We ship the same query as for insert-nodes and assume that the remote server selects the referenced nodes' bodies and embeds them in the surrounding element appropriately. This can e.g. be achieved by a special flag given in the HTTP GET request used to ship the query.

(iv) duplicate-element

As explained in Section 6.1.1, we only ship a query for links that are equipped with this "L"-directive if there is no local data (i.e. elements or non-dbxlink- and non-dbxlink-attributes) contained in the link. With duplicate-element, any referenced node will be embedded into its "own" element which is derived from the link. If the remote server constructs the desired element structure, i.e. each referenced nodes is embedded in a separate element, we can thus apply the same strategies for shipping a query as shown above for group-in-element.

(v) make-attribute

Links equipped with this modeling directive are dropped and the link's parent element gets a new IDREF attribute with the same name as the link element. In the virtual instance, this attribute references auxiliary elements containing the nodes referenced by the link's XPointer which are enriched with the link element's body that is empty if query shipping shall be applied.

Function 5.2 (getRelevantLinks()) shows that links having the "L"-directive make-attribute are only relevant if $step_x$ uses the attribute axis, i.e. the navigation continues along an attribute.

A remaining query $xpath\text{-}expr_2$ which is not empty is given and it won't contain the following or following-sibling axis and thus, the only possible way to continue the navigation is given by the id() function because an attribute does not have children or descendants. Additionally, it follows that $predicate_x$ has to be empty. Thus, we can assume that $step_x$ has the following form:

$$id(attribute::nodetest_x)[predicate_{id}]$$

which is equivalent to

$$attribute::nodetest_x/id(.)[predicate_{id}].$$

Because we want to ship the remaining query, we do not create temporary local elements addressed by the IDREF attribute as described in Section 3.2. Instead, we assume that we implicitly "jump" to the referenced elements and thus we can combine the XPointer expression and the remaining query directly:

insert-nodes:

$$xpath\text{-}expr_x/\text{self::element()}[predicate_{id}]/xpath\text{-}expr_2$$

insert-bodies:

$$xpath\text{-}expr_x/\text{child::element()}[predicate_{id}]/xpath\text{-}expr_2$$

The following example shows a query for selecting the population of Germany's capital which is required to have an attribute is_state_cap with value "yes":

/countries/country[@car_code='D']/id(@capital)[@is_state_cap='yes']/population
=/countries/country[@car_code='D']/@capital/id(.)[@is_state_cap='yes']/population

In combination with the capital XPointer (/cities/city[name='Berlin']), the query given below is shipped to the remote server:

/cities/city[name='Berlin']/self::element()[@is_state_cap='yes']/population

Function 6.2: buildQueryToShip

Input: A link element ℓ, $step_x$, $xpath\text{-}expr_2$.
Output: The query to ship for the given link.

```
1  begin
2      axis_x ⟵ the axis of step_x;      nodetest_x ⟵ the nodetest of step_x
3      if ℓ.getLeftHandDirective() ∈ {"drop-element","keep-body"} then
4          if ℓ.getRightHandDirective()="insert-nodes" then
5              if axis_x="child" then
6                  return
                   "xpath-expr_x/self::element()/self::nodetest_x[predicate_x]/xpath-expr_2"
7              else if axis_x="attribute" then
8                  return
                   "xpath-expr_x/self::attribute()/self::nodetest_x[predicate_x]/xpath-expr_2"
9              else if axis_x="descendant" then
10                 return "xpath-expr_x/descendant-or-self::nodetest_x[predicate_x]/xpath-expr_2"
11             else if axis_x="descendant-or-self" then
12                 return "document {xpath-expr_x/self::element()}/step_x/xpath-expr_2"
13         else if ℓ.getRightHandDirective()="insert-bodies" then
14             if axis_x ∈ {"child", "attribute", "descendant"} then
15                 return "xpath-expr_x/step_x/xpath-expr_2";
16             else if axis_x="descendant-or-self" then
17                 return "document {xpath-expr_x/child::element()}/step_x/xpath-expr_2"
18     else if ℓ.getLeftHandDirective() ∈ {"group-in-element", "duplicate-element"} then
19         dbx ⟵ ℓ.getTransparent()
20         link ⟵ ℓ.getName()
21         return dbx . "%document {element " . link . " {xpath-expr_x}}/step_x/xpath-expr_2"
22     else if ℓ.getLeftHandDirective()="make-attribute" then
23         if ℓ.getRightHandDirective()="insert-nodes" then
24             return "xpath-expr_x/self::element()[predicate_id]/xpath-expr_2"
25         else if ℓ.getRightHandDirective()="insert-bodies" then
26             return "xpath-expr_x/child::element()[predicate_id]/xpath-expr_2"
27 end
```

The function buildQueryToShip() contains the construction of the query to be shipped as described above. For the cases that handle the "L"-directives group-in-element and duplicate-element, the string value of the link's dbxlink:transparent attribute is concatenated with the returned query. Thus, the remote server can decide how the referenced data should be mapped into the surrounding elements as described in the preceding paragraphs. Note that comments on caching follow in the next chapter.

7 Optimizing Query Processing for Interlinked XML Documents

The preceding chapters of this thesis described (XPath) query processing for XML instances connected by XLinks in an XML database system according to the dbxlink approach. Here, in this chapter, we discuss methods for optimizing this task.

Several issues related to caching in the dbxlink scenario are discussed. We then describe the projection of XML documents wrt. an XPath query (stated by the user or given as an XPointer) for reducing the size of the transmitted data.

7.1 Caching in dbxlink

Any XLink can be equipped with caching options by specifying the dbxlink:cache attribute as already described in Section 4.1.3. Firstly, we now describe how these options are interpreted during querying wrt. the evaluation modes. Then, in order to improve these basic caching mechanisms, we discuss methods that could be used additionally.

7.1.1 Caching for XLinks using dbxlink:cache Attributes

Recall the caching options that can be supplied by the dbxlink:cache attribute for XLinks:

- dbxlink:cache="complete" caches the whole referenced document,
- dbxlink:cache="pointer" caches only the referenced XML fragment,
- dbxlink:cache="answer" caches the query result,
- dbxlink:cache="on" combines the options above, and
- dbxlink:cache="none" caches nothing.

We call "complete", "pointer" and "answer", *explicit caching modes* because they explicitly define the desired caching behavior. The mode "on" can be considered as *combined* mode that includes all explicit options and applies the appropriate one depending on the evaluation strategy that is actually used for a concrete link resolving process. It is also interpreted in a way that *any* valuable data is cached. In the following, we describe the different options in detail.

Caching the Document (dbxlink:cache="complete"). If a link is equipped with this caching option, it indicates that for this link the whole referenced document shall be cached. In case of data shipping, we thus cache the whole document that is received from the remote server.

Concerning links for which hybrid or query shipping has been applied, this caching option is not suitable and will thus be ignored because we consider the evaluation strategy to be more important than the caching mode. It would be inadequate to additionally request the whole referenced document if only a fragment (in case of hybrid shipping) or a set of resulting nodes (for query shipping) is transmitted to the local server after resolving the link. On the other hand, if we have to fall back to data shipping as described in Section 5.3, this caching option will be applied. In other words, this means that if we receive a whole XML document after a link has been resolved, the document shall be cached.

In systems implementing the dbxlink proposal, there should be a dedicated cache for preserving local copies of referenced documents. In our proof-of-concept implementation, local copies of remote documents are stored in the database backend of eXist. As any referenced document can be uniquely identified by its URI, the URI is a part of the local copy's file name in order to be able to access the file for later use. Using this approach, we are able to query this instance for obtaining fragments referenced by other links in a straightforward way by using eXist's XPath interface.

Caching XML Fragments (dbxlink:cache="pointer"). Using this caching mode, we request the XML fragments specified by the link's XPointer to be cached for later reuse. This mode can be applied for links where data or hybrid shipping is desired. For query shipping, we don't want a fragment to be transmitted to the local host if we only expect parts of the final query result and hence this option has no effect on this evaluation strategy. In case of hybrid shipping, we receive the referenced fragment by the remote host which is then stored locally. If data shipping has been applied and we have obtained the whole referenced document, we have to evaluate the XPointer on it anyway (cf. Section 5.3) and thus the computed fragment can be cached afterwards. The document itself is not kept for subsequent queries. This is, for instance, useful if the whole document is considered too big for caching.

In order to cache these fragments, we use an associative array (a data structure also known as "(hash) map" or "dictionary"). The key to find a fragment in the array is given by the xlink:href attribute which is a URI consisting of the host part and an XPointer expression. For links that are processed locally (data and hybrid shipping) we can thus use the fragments found in the cache. If a fragment has a certain size (e.g. several megabytes) it will be stored in the backend.

Caching the Results of Query Answers (dbxlink:cache="answer"). This is the only explicit caching option that can be used in combination with all evaluation strategies. Here, for a given link that has to be resolved during an XPath query is processed, the result of the remaining XPath query is cached. Recall that in the dynamic query

evaluation proposed in Section 4.4, for any link that has to be resolved, there exists a query part that has already been processed while a query rest remains. Thus, the caching strategy "answer" designates that we should cache the result of the remaining query evaluated on the fragment that is specified by the link. In query shipping, this will be the returned answer. For hybrid and data shipping we process the XPath query locally and thus have to cache the nodes that are obtained by applying the query rest to the integrated XML fragment.

Like for dbxlink:cache="pointer", we cache the computed partial result sets in a dedicated associative array. Here, the combination of the xlink:href attribute and the applied query rest serves as cache key. This means that we can only use a cached answer if it has been applied for a specific query rest on the actual document fragment. Thus, we have three caches in total, one for each explicit caching mode.

Automatic Caching (dbxlink:cache="on"). The explicit caching modes are rather strict, i.e. they are only applied if this is possible according to the corresponding link's evaluation strategy. Thus, if we have to apply fallback strategies as described in Section 5.3, it might happen that no data is cached. For instance, if a link is equipped with dbxlink:cache="pointer" and dbxlink:eval="distributed" (hybrid shipping), the remote server might refuse to answer XPath queries. In that case, we fall back to data shipping and request the whole document which won't be cached. Only the XML fragment specified by the XPointer will be kept in memory. Also, if query shipping is not possible for a specific link and a given user XPath query, and hybrid shipping is applied, the received fragment will not be cached. Thus, in order to always cache *any* data that has been received on the local host, we also offer a caching mode that combines all explicit modes and automatically chooses the appropriate caching procedure.

This means that for each evaluation mode automatically *all possible* caching strategies are applied:

- for data shipping, we cache the document, the fragment and the answer,
- in case of hybrid shipping, caching is applied for the pointer and the result of the query rest, and
- for query shipping, the partial result is kept in memory.

For a given link, in the dbxlink:eval attribute the desired evaluation strategy is specified for which the directive dbxlink:cache="on" determines to chose the appropriate caching option. However, also if due to a fallback another strategy is chosen by the system, we still can be sure that caching is applied in the way described above. Using this caching mode, designers of interlinked XML instances are thus relieved from the task of specifying an appropriate explicit caching strategy.

No Caching (dbxlink:cache="none"). For links equipped with this attribute or without a dbxlink:cache attribute, we do not cache anything at all. Thus, it is guaranteed that always the latest data from the remote server is used.

Combinations of dbxlink:eval and dbxlink:cache. As we have seen above, it is not possible or reasonable to combine dbxlink:eval and dbxlink:cache in an arbitrary way. In order to summarize the appropriate combinations, Figure 7.1 gives an overview.

cache \ eval	local	distributed	remote	on	none
complete	✓	–	–	✓	✓
pointer	✓	✓	–	✓	✓
answer	✓	✓	✓	✓	✓

Figure 7.1: Possible Combinations of Evaluation and Caching Directives

Cache Lookup and Replacement. For the explicit caching modes, the lookup procedure is straightforward, i.e. before the link is resolved (and if the caching mode fits the evaluation strategy as described above), we first check if we can find any useful data in the appropriate local cache:

- dbxlink:cache="complete": check if the answer of the XPath query stated by the user wrt. the given link has been cached previously. If this has not been found, then try to get the fragment referenced by the link's XPointer from the appropriate cache and use it for the current query. In case that also the fragment has not been cached, we try to find the whole remote document locally.
- dbxlink:cache="pointer": first look for the answer of the query. If it has not been cached, then check if the referenced fragment can be found.
- dbxlink:cache="answer" tries to get the answer for the query locally.

For dbxlink:cache="on" the lookup is more sophisticated. Given an XLink to be handled having such an attribute, we first check if the answer can be found in the cache and we would use it to answer the query rest. If it is not found, then we look for the XML fragment specified by the XPointer. In case that the fragment is found, we compute the answer, cache it and return it to the query evaluator. Analogously, if the fragment is not in the cache, we look for the whole document and request it if a cache miss occurs. But if it is found, then we compute and cache both the fragment and the answer. Hence, we always try to get the most specific data portion first and cache any freshly computed data. For illustration, this process is implemented in the function combinedCaching().

The appropriate cache replacement strategy is not an issue in this thesis; thus, in an implementation, any cache algorithm can be used. In our proof-of-concept implementation, we apply the well-known and straightforward *LFU (least frequently used)* algorithm, i.e. if a cache is full and a new item should be inserted, we discard the least frequently used entry.

Function 7.1: combinedCaching

Input: A link ℓ with dbxlink:cache="on" to be resolved.
Output: The best fitting data item from the cache or, if a cache miss occurs, the remote data.

```
1  begin
2      if answer found then
3          return answer
4      else if pointer found then
5          compute answer
6          cache answer
7          return answer
8      else if localDocument found then
9          compute pointer
10         cache pointer
11         compute answer
12         cache answer
13         return answer
14     else
15         if eval="remote" then
16             answer ⟵ shipQuery(buildQueryToShip(...))
17             cache answer
18             return answer
19         else if eval="local" then
20             doc ⟵ getReferencedDocument(ℓ.getAttribute("xlink:href"))
21             cache doc
22             compute pointer
23             cache pointer
24             compute answer
25             cache answer
26             return answer
27         else
28             pointer ⟵ getXMLFragment(ℓ.getAttribute("xlink:href"))
29             cache pointer
30             compute answer
31             cache answer
32             return answer
33 end
```

7.1.2 Implicit Caching during Query Evaluation

In contrast to the dbxlink:cache options given by designers of linked instances, an implicit caching strategy is applied during query evaluation in an opaque way.

Consider the following situation. An XML document containing several XLinks is queried. Assume that there are two different links ℓ_1 and ℓ_2 that reference the same remote document. For instance, cities-D.xml is referenced by both the capital and the cities link children of the element representing Germany. During querying the considered document, at time-point t_1, ℓ_1 is resolved, later at time-point t_2, we have to handle ℓ_2.

Now, assume that in the meantime the remote document has changed. Thus, during a single query evaluation, we could get different results for the same document. Though in MONDIAL this should happen very rarely, this is still an issue in the general case.

However, this problem is not only relevant for our approach. It occurs in any scenario where distributed XML documents are involved and it requires a distributed transaction concept for XML. An investigation of this topic would lead us beyond the focus and scope of this thesis but we mention some points that could be useful for the proof-of-concept implementation.

In case of query shipping, we can not guarantee that remotely, the same data is supplied for different shipped queries. The remote server gets the queries and handles them autonomously. We have no possibility to check if the document changes between different queries and also, there is no general mechanism to "*lock*" a certain remote document during the evaluation of a specific user query.

For hybrid shipping, where we request the XML fragment addressed by an XPointer, we can store any such fragment during the evaluation of a query in a special cache. In case that an XPointer occurs in different links, the same (implicitly stored) data is used. However, if two different XPointers reference "overlapping" fragments (e.g. one fragment containing all German cities and another one consisting only of Berlin) which are requested at different time-points, it might happen that the same data items have changed in the meantime (e.g. Berlin's population gets updated).

Considering data shipping, copies of remote documents can be kept locally. Thus, during we process a query, we can favor any locally stored fragment or document over requesting the corresponding remote data. By this approach, for a specific query, we assure that any referenced fragment or document of links requiring hybrid or data shipping is at most requested once remotely. Also, using this approach, we save some network traffic and parsing of redundant data.

Note that any data cached for a link having an appropriate dbxlink:cache attribute is used for *any* future query that traverses this link while the implicit or automatic caching proposed in this paragraph is only relevant for a *single* query. After its evaluation, the data that has been cached for links with disabled caching options is removed.

7.2 Projection of XML Documents and Fragments

Main memory XQuery processors like Saxon [SAX], Xalan [Xal] or Galax [Gal] are not able to process queries on XML documents of arbitrary size because the documents have to be loaded completely before evaluating the query. Hence, for overcoming this limitation, the work in [MS03] proposes a method for reducing XML documents to relevant parts wrt. a query, called *projection*. For a given XPath query, a set of relevant *projection paths* is computed at compile time. These paths describe the parts of an XML document that are needed to answer the query. Then, before loading a document, it can thus be reduced in size according to these paths. Experiments in [MS03] have shown that, using this method, the memory requirements can be reduced to 5% on average.

If we want to apply this approach in our implementation, we have to discuss the

relationship to the different query evaluation modes, i.e. query shipping, data shipping and hybrid shipping.

Query Shipping (dbxlink:eval="remote"). We delegate the evaluation of the XLink and the ongoing query evaluation to the remote server. The final result for this part is then sent back to the originating server. Projection methods are not required here because the result set can not be reduced. However, the remote server might exploit these methods autonomously for its internal XPath evaluation processes.

Data Shipping (dbxlink:eval="local"). In case of dbxlink:eval="local", the referenced document is requested for a local evaluation, i.e. the XPointer and the remaining query are processed subsequently on a local copy. If we want to avoid big documents to be stored in the local database backend, we can apply projection (based on the XPointer of the currently processed link) to the received document before it is stored. This means that the requested document is projected wrt. the XPath expression contained in the pointer. However, note that this is only useful if projection and the query evaluation on the smaller fragment is faster than storing the whole document and evaluating the query on the big fragment.

Hybrid Shipping (dbxlink:eval="distributed"). If this shipping strategy is applied, the XPointer expression is evaluated remotely returning the referenced XML fragment to which the remaining query is applied locally. Thus, the remote server could apply projection to the returned fragment according to the remaining query. Consider the following informal example which illustrates how this approach could be used in our implementation for hybrid shipping.

Example 7.1
Assume that the following query has to be evaluated on a variant of MONDIAL*'s distributed version where for all links dbxlink:eval="distributed" (i.e. hybrid shipping) is set:*

```
//organization[@abbrev="EU"]/member/id(@capital)/population
```

The XPointers of the member elements (which are children of elements representing organizations) reference the countries that are members of the actual organization. When these pointers are resolved, not the whole country elements are transmitted, but only the element "hull" with the capital XLink element (that due to the "L"-directive as "make-attribute" will contribute the required capital attribute of the next step). Then, resolving the capital link, not the whole corresponding city elements are transmitted, but only the element hulls with the population subelements that are needed to answer the last step of the query.

As sketched in the example above, given an XLink and an XPath user query, the idea is to project the referenced XML fragment according to the XPath query rest. This would be an "extended" hybrid shipping leading to less data traffic over the network

but it requires the remote server to support this operation. Here, the question comes into mind if it is better to let the remote server process the whole query, i.e. to switch to query shipping. However, there are some cases where hybrid shipping is preferable compared to query shipping, e.g. there might be servers that do not support query shipping according to the dbxlink approach but, besides XPath processing facilities, they could offer projection methods.

8 Related and Further Work

In this section, some related work is discussed and we also sketch some open issues that might lead to further work.

8.1 Related Work

Active XML. A general approach for integrating remote access functionality into XML documents is proposed by *Active XML* [ABM+02]: <axml:sc> elements allow for embedding *service calls* into XML documents. Active XML and dbxlink differ significantly wrt. generality (Active XML) and specialization (dbxlink) and in the degree of integration with the database functionality. While the dbxlink approach is an incremental extension to the existing concepts of XLink and XPointer, targeting to provide a transparent data model and support XPath/XQuery for them from the database point of view, Active XML is a generic extension of functionality towards Web Services. Nevertheless, as described below, dbxlink and Active XML can be used to implement each other.

Active XML has no processing directives (the left-hand- or "L"-directives in our approach) specifying how the results of Web Services should be integrated. Especially, in Active XML, it is not possible to create attributes or duplicate elements. Additionally, the dbxlink proposal provides explicit processing strategies (cf. the dbxlink:eval and dbxlink:cache directives). Because there are some similarities between our approach and Active XML we show how these techniques are related to each other wrt. Web Services.

In Active XML, axml:sc elements represent Web Service calls which are then replaced by the result of the service call. It follows an example Active XML document[1]:

```
<directory>
    <dep name="Toy">
        <axml:sc>toy.xyz.com/GetToyPersonnel()</axml:sc>
    </dep>
    <dep name="DVD">
        <axml:sc>dvd2000.com/GetDVDPersonnel()</axml:sc>
    </dep>
</directory>
```

Note that there are many extra parameters that have to be supplied in order to fully specify a Web Service call but they are generally omitted by the Active XML authors for clarity reasons.

[1] Taken from `http://www.activexml.net`.

On one hand, an Active XML service that implements the dbxlink modeling and takes a dbxlink-extended XLink element could return the appropriate XML fragment which is then integrated into the Active XML document conforming to dbxlink:transparent="drop-element insert-nodes". Other mappings are not possible with Active XML. On the other hand, in order to embed Active XML into our proposal, evaluating XLink elements that refer to Web Services by a dbxlink-extended XPath/XQuery engine covers the basic Active XML functionality. Depending on the given Web Service, we can build appropriate calls as described later in Section 8.2.1.

Decomposing Queries on Distributed XML Data. Several approaches dealing with strategies for decomposing queries on distributed XML data have been investigated whose results can also be used for the implementation of the dbxlink specification. In [Suc02], distributed query evaluation for general semistructured data graphs is investigated. The approach assumes that a fixed community of sites agrees on sharing their data and answering queries. They split the query into a *decomposed query*, evaluate its parts independently at each site, and assemble the result fragments. In contrast to this, the scenario of our approach considers XLink references between *arbitrary* sources, and the specification for mapping the linked fragments to a virtual instance and querying it. The logical modeling of [Suc02] is similar to XInclude. In [BG03], the distribution of XML repositories is investigated, focusing on index structures. Both these approaches are orthogonal to ours (where the focus is on the modeling and handling of the interplay of links seen as views) and could probably be applied for a more efficient implementation.

SXLink. The XLink processor SXLink [LL05] implemented in Scheme aims for offering methods in order to be able to obtain all information about XLinks contained in a set of XML documents. In this system, queries are supported by an XPath extension called "XPathLink" implemented as "SXPathLink" in Scheme. This language introduces an additional XPath axis traverse for traversing XLinks explicitly in XPath queries. As already discussed in Section 4.1.1, we showed that our approach which handles queries over linked XML instances transparently is preferable in the general case.

8.2 Further Work

The approach for querying interlinked XML instances proposed in this thesis can be used for many different scenarios like data integration and distributed XML data management. In addition to these applications, with some further extensions, dbxlink would even become more versatile. Thus, in this section, we sketch some ideas that could result in useful extensions, e.g. for integrating Web Service calls and additional optimization strategies.

8.2.1 Integrating Web Service Calls

The Web of today consists of a huge amount of web sites. Users usually request information via a web browser by navigating to desired web pages while computers also

exchange data between each others in an autonomous way. In order to facilitate the machine-to-machine interaction for this task, *Web Services* [WSW02] supply versatile concepts. Many popular web sites offer Web Services for various purposes, e.g. the *Google SOAP Search API* [GOO] can be used to access the popular web search engine for submitting search requests. Thus, web searches can be integrated into arbitrary programs.

In this section, we discuss how Web Service Calls can be integrated into the dbxlink approach. As Web Services are an abstract concept and can be implemented using different approaches like *SOAP* and plain *HTTP*, we consider both these approaches.

HTTP/REST. *REST (Representational State Transfer)* [Fie00] is a concept based on *HTTP (Hypertext Transfer Protocol)* [HTT99] that defines several principles for modeling distributed hypermedia systems in a straightforward way. In the context of Web Services, it can be used for defining *RESTful* Web Services that are accessed by simple HTTP GET requests. In order to supply parameters for a Web Service call, key-value pairs can be given in the URI of the Web Service:

```
http://www.example.org/RESTwebservice?key1=value1&key2=value2
```

Here, the reference to the remote resource is described without XPointer. Thus, RESTful Web Services are accessed by standard URIs and assuming that only Web Services that return XML data to the client are considered, it is straightforward to embed these calls into dbxlink. In fact, such a Web Service can be considered as any other static web resource providing XML data. We just have to access the data and process an XPointer to extract the relevant information locally. The following example illustrates how a fictitious weather forecast service can be used to enrich the data about cities:

```
<city>
  <name>New York</name>
  <longitude>-74</longitude>
  <latitude>40.4</latitude>
  <population year="96">7380906</population>
  <weather dbxlink:eval="local"
       dbxlink:transparent="group-in-element insert-bodies"
       xlink:href="http://.../forecast?long=-74&lat=40.4#xpointer(//data)"/>
</city>
```

SOAP. Many Web Services can be accessed with the *SOAP* protocol [SOA03]. SOAP messages are usually sent in XML via HTTP POST requests and consist of a *SOAP Envelope* that contains a *SOAP Body* which is the most important part. It provides the desired method to call plus its necessary parameters. Because of this overhead, SOAP calls can not be embedded into dbxlink as straightforward as calls to RESTful services.

We could use the dbxlink:eval="soap" directive to indicate that a SOAP call has to be built. The xlink:href attribute should supply the URI of the Web Service. Additional attributes in a dbxlinksoap namespace could then be used to specify the desired

SOAP method (dbxlinksoap:method), any necessary parameter (dbxlinksoap:param) and an XPointer expression (dbxlinksoap:xpointer) for extracting the desired information. The basic structure of an embedded SOAP call looks as follows:

```
<link xlink:href="soap-service-uri"
      dbxlink:transparent="transparent directive" dbxlink:eval="soap"
      dbxlinksoap:method="soap-method" dbxlinksoap:xpointer="xpath-expr"
    <dbxlinksoap:param name="key1" value="value1" />
      ⋮
    <dbxlinksoap:param name="keyn">
      <!-- data for this parameter -->
    </dbxlinksoap:param>
</link>
```

The obtained XML result is then stored locally in a temporary document on which the XPointer-like expression is processed in order to extract desired information. Finally, the result is mapped into the partial instance according to the dbxlink:transparent directive. Below, an example similar to the one given above is shown. It illustrates how to embed a weather forecast into the city element via SOAP:

```
<city id="cty-cid-cia-United-States-2">
    <name>New York</name>
    <longitude>-74</longitude>
    <latitude>40.4</latitude>
    <population year="96">7380906</population>
    <weather xlink:href="http://www.weather.gov/forecasts/xml/
                                              SOAP_server/ndfdXMLserver.php"
            dbxlink:transparent="group-in-element insert-bodies"
            dbxlink:eval="soap"
            dbxlinksoap:method="NDFDgenByDay"
            dbxlinksoap:xpointer="//weather-conditions">
      <dbxlinksoap:param name="longitude" value="-74" />
      <dbxlinksoap:param name="latitude" value="40.4" />
      <dbxlinksoap:param name="startDate" value="TODAY" />
      <dbxlinksoap:param name="numDays" value="3" />
      <dbxlinksoap:param name="format" value="24 hourly" />
    </weather>
</city>
```

8.2.2 XPath Query Containment for XPointers

During the evaluation of an XPath user query on a scenario consisting of interlinked XML documents, a set of relevant links has to be resolved. To achieve this, the links' XPointers have to be processed. Basically, these XPointers are again XPath expressions (cf. Section 2.3.1).

For some pointers, it is possible that they have been computed for a previous user query and hence can be found in a local cache. Thus, the corresponding XML fragments can be used without the need of accessing the remote server. By contrast, for XPointers that might not have been or should not be cached, we might want to check if they are included in some other XPointer that has been evaluated before. For instance, the element representing the city of Berlin which is addressed in the XML document citiesD.xml by the expression /cities/city[name='Berlin'] is already contained in the fragment consisting of all German cities (/cities/city). Thus, if we have computed the latter fragment and have cached it locally, then we could save the remote request by extracting Berlin from this fragment. Note that this is only possible if we have accessed a fragment during the evaluation of a query, i.e. if we have received actual data.

The task to check if the result of an XPath expression can be found in the result set of another XPath expression is called *XPath query containment* problem [MS04]. There exist several investigations and proposals related to this problem. Unfortunately, it has been shown that even for an XPath fragment that is restricted to the child and descendant axes, wildcards (*) and predicates, the XPath query containment problem is co-NP complete [MS04]. However, [BOB+04] proposes a polynomial time algorithm also for comparison predicates and disjunctions that is sound and complete for most useful cases. Also, in absence of one of the three operators, i.e. any combination of two operators, there exist polynomial algorithms for the containment problem. Containment detection and evaluation can be useful for the following cases:

- check if two pointers are contained in each other,
- containment of an XPath query rest in a previously computed rest.

After storing and parsing a new XML document, we have computed a set of XLinks contained in that document. We know (i) where an XLink is contained in the XML tree and (ii) the structure of the XPointer. Next, as XPointers are basically XPath expressions, we can use the exponential but complete query containment algorithm proposed in [MS04] in order to detect containment between the XLinks. This might be expensive but is done only once for static XML documents. Thus, we will know which links are contained in others and use this information for a more efficient link evaluation strategy during query processing. For any link that has to be resolved during query evaluation it is then checked if there is a fragment that contains its pointer.

Given a set of relevant links that have to be handled we propose the following treatment:

- each pointer is checked if it is contained in some other pointer,
- big fragments are requested before the smaller ones (e.g. cities before capital).

On the other hand, if a user states a query, the question arises if it is possible to decide in advance which links need to be evaluated. If we know which links have to be handled and if we can detect if their results have already been cached (using a hash map), then

we can materialize the view and run the query directly on it instead of iterating over all element nodes. The hash map used for mapping link elements to referenced XML fragments could also be equipped with a path index leading to that link. Here, this structure also has to respect the dbxlink:transparent directive.

8.2.3 XML Indexing

XML index structures are intended to expedite the processing of queries, especially of path queries. Thus, in this section, we briefly describe how XML indexing techniques could improve our implementation of an XLink-aware querying system.

General Requirements for Indexing. In our dbxlink approach, an important aspect is that queries "traverse" *autonomous* documents, coming in via an XLink/XPointer, and probably leaving it via another XPointer. Especially, the referenced documents are usually not aware which references use them. Furthermore, it is preferable that the knowledge about a referenced document can optionally be combined with the knowledge of the referencing document (e.g., the index of countries.xml could be enhanced with the knowledge about the structure of the linked cities-XX.xml documents). For that, the combination of local indexes with a more lightweight structure for handling structural information of the referenced sources as proposed in [BMCJ04] seems appropriate.

For the adaptation of indexing to the requirements of the dbxlink scenario the following issues have to be considered: (i) autonomous documents (or "closely related" sets of documents) maintain their own indexes, that must be "open" to adapt to the linked sources, (ii) maintenance of combined index structures along references, preferably using *operators* on such structures, and (iii) the possibility to extend the covered area on demand.

Another use case is to build such indexes for all incoming XML fragments that are received when links are processed with the local and distributed evaluation strategies. As these fragments have to be parsed, we can generate appropriate indexes on-the-fly. Here, if we do not take caching into account, the index can directly be tailored to the remaining query $xpath\text{-}expr_2$ which is not possible for stored XML instances which are parsed once. Thus, for all documents the local instance points to, additional information is obtained incrementally, e.g. for the corresponding links we know which progressing paths we can expect. As a consequence, for query rests of XPath expressions that navigate along links, we can decide if the link has to be resolved, i.e. if the query would yield a non-empty result.

Data Guides. Data guides [GW97] have been developed for providing some benefits of schema information in schema-free *semistructured data* environments. A data guide can thus be adopted to a concise and accurate *structural summary* of an XML instance. Such a structure can be generated in linear time. Considered from our point of view, data guides can be used a priori for deciding if a given path (query) exists in an instance, i.e., if the answer set of an XPath query against an XML document must be empty or

not. *Strong data guides* [MAG+97] even allow not only to check if a path exists, but also to return its answers.

Data guides can not only be applied for answering user queries, but especially for deciding if an XLink provides a relevant answer. Consider hybrid shipping or query shipping. A server that provides XML documents that are frequently queried (for evaluating XPointers or user queries) can maintain a data guide, and match each query first against the data guide before actually evaluating it. Still, *communication* of the query, including establishing of the connection is necessary. As an alternative, the *referencing* document can keep (either by polling, or by publish-subscribe mechanisms) the data guide of the referenced documents, and *check* the emptiness of the answer even before doing any communication. Using data shipping, the data guide can be generated during the parsing process of the referenced document.

Note that in case of references, the data guides either must include the referenced documents (which leads to a fixpoint process for generating all of them in case of a cyclic network), or end with an "ANY" where XLinks are present.

9 Conclusion

The dbxlink proposal specifies how interlinked XML instances are mapped to a virtual instance. Links are considered as *transparent*, i.e. they define a view on referenced XML data which is silently mapped into a virtual instance in a flexible way. In this thesis, it has been analyzed how this virtual instance can be queried with XPath and it has been described in an abstract way how to achieve this. In order to evaluate queries on an interlinked scenario, it showed that it is not feasible to materialize the complete virtual instance in advance. Instead, during query evaluation, only relevant links are resolved on demand leading to the materialization of a partial instance that covers the parts of the view which are necessary to answer the query appropriately.

Three different strategies can be applied for the evaluation of XLinks, namely data shipping, hybrid shipping and query shipping. It has been discussed in detail how these strategies are integrated into the query evaluation process. Also, how to cope with cyclic instances has been investigated and some useful caching and optimization strategies have been given. A proof-of-concept implementation of the dbxlink approach has been undertaken as part of the open source XML database system *eXist* [exi].

This thesis provides results that fill the gap in the W3C XML Query (XQuery) Requirements and that can be applied to different XML query languages.

W3C XML Query (XQuery) Requirements. Recall the aforementioned W3C XML Query (XQuery) Requirements [XMQ04, Sec. 3.3.4/3.4.12 ("References")] which explicitly state that

> *"the XML Query Data Model MUST include support for references, including both references within an XML document and references from one XML document to another"*

and that

> *"queries MUST be able to traverse intra- and inter-document references",*

but which have only been partially met by standard XQuery. Additionally, in Section 4.1.1, it has been shown that it not possible to query along XLinks in a general way. A description has been given how the dbxlink approach allows even for XQuery's subset XPath to query interlinked scenarios. Especially, the above mentioned W3C requirements have been met for simple XLinks in both an abstract and practical way in terms of a proof-of-concept implementation.

Transferability and Applicability. The query strategies proposed in this thesis are described in an abstract way and it has been shown how to implement dbxlink functionality as an extension to XML query systems that rely on the stepwise evaluation strategy for XPath: the link expansion has to be integrated into the axis evaluation. Thus, these descriptions allow for enabling any XML query language based on XPath like XQuery, XPathLog [May04] or XSLT [XSL06] for handling simple XLinks. Additionally, for querying interlinked XML instances which contain the more complex extended XLinks, the investigations conducted in this work can serve as a basic foundation.

Online Demonstration. The results of the investigations outlined in this thesis have been used for conducting a proof-of-concept implementation of the dbxlink approach as an extension to the open source XML database system *eXist* [exi]. The MONDIAL testbed (cf. Chapter 1) has been used to set up an online demonstration that is reachable via `http://www.dbis.informatik.uni-goettingen.de/linxis/` where some example queries are provided.

Bibliography

[ABM+02] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: Peer-to-Peer Data and Web Services Integration. In *VLDB*, pp. 1087–1090, 2002.

[Ato05] The Atom Syndication Format. `http://www.ietf.org/rfc/rfc4287.txt`, 2005.

[axi] Apache Axis: an Implementation of the SOAP Protocol. `http://ws.apache.org/axis/`.

[BFM06a] E. Behrends, O. Fritzen, and W. May. Handling Interlinked XML Instances on the Web. In *EDBT*, pp. 792–810, 2006.

[BFM06b] E. Behrends, O. Fritzen, and W. May. Querying along XLinks in XPath/XQuery: Situation, Applications, Perspectives. In *11th FMLDO Workshop: QLQP*, pp. 662–674, 2006.

[BG03] J.-M. Bremer and M. Gertz. On Distributing XML Repositories. In *WebDB*, pp. 73–78, 2003.

[BMCJ04] A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain. XPath lookup queries in P2P networks. In *WIDM*, pp. 48–55, 2004.

[BOB+04] A. Balmin, F. Ozcan, K. S. Beyer, R. Cochrane, and H. Pirahesh. A Framework for Using Materialized XPath Views in XML Query Processing. In *VLDB*, pp. 60–71, 2004.

[Bru04] M. Brundage. *XQuery: The XML Query Language*. Addison-Wesley, 2004.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[DOM98] Document Object Model (DOM). `http://www.w3.org/DOM/`, 1998.

[exi] eXist: Open Source Native XML Database. `http://exist-db.org/`.

[Fie00] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Dissertation, University of California, Irvine, 2000.

[FJK96] M. J. Franklin, B. T. Jónsson, and D. Kossmann. Performance Tradeoffs for Client-Server Query Processing. In *SIGMOD Conference*, pp. 149–160, 1996.

[Gal] Galax: An Implementation of XQuery. `http://www.galaxquery.org/`.

[GOO] Google SOAP Search API. `http://www.google.com/apis/`.

[GW97] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB*, pp. 436–445, 1997.

[HTM99] HTML 4.01 Specification. `http://www.w3.org/TR/html401/`, 1999.

[HTT99] Hypertext Transfer Protocol – HTTP/1.1, *Requests for Comments: 2616*. `http://www.ietf.org/rfc/rfc2616.txt`, 1999.

[KCD+03] H. Katz, D. Chamberlin, D. Draper, M. Fernandez, M. Kay, J. Robie, M. Rys, J. Simeon, J. Tivy, and P. Wadler. *XQuery from the Experts: A Guide to the W3C XML Query Language*. Addison-Wesley, 2003.

[Len02] M. Lenzerini. Data Integration: A Theoretical Perspective. In *PODS*, pp. 233–246, 2002.

[LL05] D. A. Lizorkin and K. Y. Lisovsky. Implementation of the XML Linking Language XLink by Functional Methods. *Programming and Computer Software*, 31(1):34–46, 2005.

[LS04] W. Lehner and H. Schöning. *XQuery*. dpunkt.verlag GmbH, 2004.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3):54–66, 1997.

[May02] W. May. Querying Linked XML Document Networks in the Web. In *11th. WWW Conference*, 2002. Available at `http://www2002.org/CDROM/alternate/166/`.

[May04] W. May. XPath-Logic and XPathLog: A Logic-Programming Style XML Data Manipulation Language. *TPLP*, 4(3):239–287, 2004.

[MM03] W. May and D. Malheiro. A Logical, Transparent Model for Querying Linked XML Documents. In *BTW*, pp. 147–156, 2003.

[Mon01] The MONDIAL Database. `http://www.dbis.informatik.uni-goettingen.de/Mondial/`, 2001.

[MS03] A. Marian and J. Siméon. Projecting XML Documents. In *VLDB*, pp. 213–224, 2003.

[MS04] G. Miklau and D. Suciu. Containment and Equivalence for a Fragment of XPath. *Journal of the ACM*, 51(1):2–45, 2004.

[Nam06] Namespaces in XML 1.0 (Second Edition). `http://www.w3.org/TR/REC-xml-names/`, 2006.

[OMFB02] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *EDBT Workshops*, pp. 109–127, 2002.

[QRS+95] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying Semistructured Heterogeneous Information. In *DOOD*, pp. 319–344, 1995.

[RBHS04] C. Re, J. Brinkley, K. P. Hinshaw, and D. Suciu. Distributed XQuery. In *IIWeb*, pp. 116–121, 2004.

[REL01] RELAX NG. `http://www.relaxng.org/spec-20011203.html`, 2001.

[RSS99] Rich Site Summary (RSS) 0.91 Spec, revision 3. `http://my.netscape.com/publish/formats/rss-spec-0.91.html`, 1999.

[SAX] SAXON: The XSLT and XQuery Processor. `http://saxon.sourceforge.net/`.

[SGM86] Standard Generalized Markup Language (SGML). ISO (International Organization for Standardization), ISO 8879:1986, 1986.

[SOA03] SOAP Version 1.2 Part 0: Primer. `http://www.w3.org/TR/soap12-part0/`, 2003.

[SQL03] Information Technology–Database Language–SQL. ISO (International Organization for Standardization), ISO 9075:2003, 2003.

[Suc02] D. Suciu. Distributed Query Evaluation on Semistructured Data. *TODS*, 27(1):1–62, 2002.

[TK78] D. Tsichritzis and A. C. Klug. The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Dabatase Management Systems. *Inf. Syst.*, 3(3):173–191, 1978.

[W3C] W3C – The World Wide Web Consortium. `http://www.w3.org/`.

[WSW02] W3C Web Services Activity. `http://www.w3.org/2002/ws/`, 2002.

[Xal] The Apache Xalan Project. `http://xalan.apache.org/`.

[XHT00] XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). `http://www.w3.org/TR/xhtml1/`, 2000.

[XIn04] XML Inclusions (XInclude). `http://www.w3.org/TR/xinclude/`, 2004.

[XLi01] XML Linking Language (XLink) Version 1.0. http://www.w3.org/TR/xlink/, 2001.

[XLi06] XML Linking Language (XLink) Version 1.1. http://www.w3.org/TR/xlink11/, 2006.

[XML00] XML:DB. XUpdate - XML Update Language. http://xmldb-org.sourceforge.net/xupdate/, 2000.

[XML04a] XML Information Set (Second Edition). http://www.w3.org/TR/xml-infoset/, 2004.

[XML04b] XML Schema Part 0: Primer Second Edition. http://www.w3.org/TR/xmlschema-0/, 2004.

[XML04c] XML Schema Part 1: Structures Second Edition. http://www.w3.org/TR/xmlschema-1/, 2004.

[XML06] Extensible Markup Language (XML) 1.0 (Fourth Edition). http://www.w3.org/TR/REC-xml/, 2006.

[XMQ04] XML Query (XQuery) Requirements. http://www.w3.org/TR/xquery-requirements/, 2004.

[XMQ06a] XQuery 1.0 and XPath 2.0 Data Model (XDM). http://www.w3.org/TR/query-datamodel/, 2006.

[XMQ06b] XQuery 1.0 and XPath 2.0 Formal Semantics. http://www.w3.org/TR/xquery-semantics/, 2006.

[XPa06] XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20/, 2006.

[XPQ06] XQuery 1.0 and XPath 2.0 Functions and Operators. http://www.w3.org/TR/xquery-operators/, 2006.

[XPt02] XPointer xpointer() Scheme. http://www.w3.org/TR/xptr-xpointer/, 2002.

[XPt03a] XPointer element() Scheme. http://www.w3.org/TR/xptr-element/, 2003.

[XPt03b] XPointer Framework. http://www.w3.org/TR/xptr-framework/, 2003.

[XPt03c] XPointer xmlns() Scheme. http://www.w3.org/TR/xptr-xmlns/, 2003.

[XQu06] XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, 2006.

[XSL06] XSL Transformations (XSLT) Version 2.0. http://www.w3.org/TR/xslt20/, 2006.

Curriculum Vitae

Erik Behrends

Persönliche Daten

Geburtsdatum	8. März 1974
Geburtsort	Itzehoe
Staatsangehörigkeit	deutsch
Familienstand	verheiratet

Wissenschaftlicher Werdegang

1980-1984	Breitenaugrundschule Plön
1984-1993	Gymnasium Schloss Plön; Abschluss: Abitur
1995-1997	Studium der Informatik und Mathematik an der Christian-Albrechts-Universität zu Kiel
1997-2001	Studium der Informatik mit Nebenfach Mathematik an der Albert-Ludwigs-Universität Freiburg Abschluss: Diplom-Informatiker
1998-2001	Studentische wissenschaftliche Hilfskraft am Institut für Informatik der Albert-Ludwigs-Universität Freiburg
2001-2003	Software-Entwickler bei der Inxmail GmbH in Freiburg
seit 2003	Wissenschaftlicher Mitarbeiter am Institut für Informatik der Georg-August-Universität Göttingen