Modeling and Querying of Distributed XML Data in Presence of 3rd Party Links

Dissertation

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

vorgelegt von

Oliver Fritzen
aus Trier

Göttingen
im November 2007

Referent: Prof. Dr. rer. nat. Wolfgang May, Georg-August-Universität Göttingen
Korreferent: Prof. Dr.-Ing. habil. Kai-Uwe Sattler, Technische Universität Ilmenau

Abstract

XML (short for *eXtensible Markup Language*) is a meta-language for the representation of digital data. XML has had an enormous impact on modern computer science and IT industry since its advent in 1997, for several reasons: XML is simple and easily accessible. Using Unicode as encoding, XML can be viewed and authored/edited with common text editors, and due to the context-free and well-formed structure of XML document types, it is easy to provide efficient parsers for processing XML documents. Also, XML's concept of definable document types enables for a structured representation of almost arbitrary digital data, with the document type modeling the domain of the data, which makes XML a very powerful and flexible standard for data representation, particularly regarding the Web.

The *XLink* standard is an extension to XML for defining references between XML documents, inspired by the hyperlink concept from hypertext. XLink defines two types of links: *Simple Links* are unidirectional links from one document to another, similar to HTML hyperlinks. *Extended Links* create graph-based relationships (*arcs*) between portions of XML (*resources*) over multiple XML documents.

Within the *LinXIS* project, models and query evaluation for *XLink* have been investigated: in a *logical data model*, a *Simple Link* is given the semantics of an *embedded view* that "imports" the referenced data from a remote document into the link-defining document. The participating XML data, together with the *Simple Links* define a *virtual instance* (a single-document view on the distributed data) according to the *logical data model.*

Extended Links define relations between *XML resources*, but in contrast to *Simple Links*, they are not defined inside the participating resources but *apart of them.* This allows to define a semantics for *Extended Links*, with an *Extended Link* defining views that combine and extend the participating resources from a *3rd party perspective*, without need for write access to them, and thus extending the *Simple Links logical data model.*

The above described *logical data model* provides a semantics for the evaluation of XPath queries over distributed XML data: A query may be evaluated not on a (physical) XML document, but on the *virtual instance* defined by the given *Simple* and *Extended Links.* The query evaluation may "follow" along a Simple Link, continuing the evaluation process on the referenced, physically remote data. For Extended Links, queries can be evaluated on the integrated view combining the sources referenced by an *Extended Link*, based on the *3rd party semantics* of the link.

A previous PhD thesis, which also emerged from the *LinXIS* project, introduced the data model for *Simple Links* and investigated tech-

niques and algorithms for XPath query evaluation on the linked XML data. As part of the work, the data model was implemented on base of the Open Source XML database system *eXist*, thus creating a Simple-Link-enhanced XML database prototype.

The present work extends the focus from *Simple* to *Extended Links*: The work includes a formal description of both *Simple Link* and *Extended Link* semantics, based on a specification as an *abstract data type (ADT)*, and providing *Extended Links* with a *3rd Party Link semantics*. Also, the basic concepts for query evaluation with respect to *3rd Party Links* are investigated. The algorithms as well as the logical data model for *3rd Party Links* are implemented by further enhancement of the *eXist*-based prototype, providing the query evaluation unit with that semantics. The prototype is tested within a case study, evaluating the prototype's functional behavior and performance. The case study is followed by a discussion of the proposed *3rd Party Link* approach, addressing its applicability in terms of its design, performance and its relevance within a rapidly evolving Web infrastructure. The work is completed by a conclusion addressing the previously discussed issues, and giving an overview over related research as well as over perspectives and further work.

Abstract (Deutsch)

XML (für *eXtended Markup Language*) ist eine Metasprache zur Darstellung digitaler Daten, die seit ihrer Standardisierung 1997 in kurzer Zeit extrem populär geworden ist, sowohl im akademischen als auch im industriellen Anwendungskontext. Dafür gibt es eine Anzahl von Gründen: XML hat eine niedrige Einstiegsschwelle in Bezug auf die notwendige Tool-Infrastruktur und Lesbarkeit durch Menschen, da XML Unicode als Darstellungsform benutzt und mit üblichen Texteditoren geschrieben und gelesen werden kann. Außerdem ist die kontextfreie Grammatik eines Dokumenttyps zusammen mit der strikten Serialisierungsvorschrift der *Wohlgeformtheit* effizient mit Parsern verarbeitbar. Darüber hinaus bietet das Konzept der frei definierbaren Dokumenttypen die Möglichkeit, nahezu beliebige Daten strukturiert darzustellen und über die Grenzen von Plattformen, Softwareinfrastrukturen oder bestimmter Formate hinweg auszutauschen, insbesondere über das Web.

XLink ist eine vom World Wide Web Consortium standardisierte Syntax um XML-Dokumente mit einer Hyperlink-Funktionalität auszustatten. *XLink* definiert zwei Arten von Links: *Simple Links* sind unidirektionale Verbindungen von einem Dokument in ein anderes, vergleichbar mit dem HTML-Hyperlink. Die komplexeren *Extended Links* verknüpfen *Resourcen* innerhalb verschiedener XML-Dokumente miteinander durch gerichtete Graphstrukturen (*arcs*).

Der *XLink*-Standard definiert für solche Links lediglich eine *Syntax*; eine *Semantik* bekommt ein Link erst im Kontext einer Anwendung wie der Anfragesprache *XQuery* oder der Transformationssprache *XSL*. Für beide existiert allerdings keine – zumindest keine standardisierte – *XLink*-Semantik.

Im Rahmen des Forschungsprojektes *LinXIS* sind Modelle und die Auswertung von Anfragen in Bezug auf *XLink* untersucht worden: Man kann einen *Simple Link* als Definition einer eingebetteten Sicht betrachten, die Daten aus einem referenzierten Dokument in das einbettende Dokument einbindet. Die beteiligten – physischen – Dokumente und die *Simple Links* werden durch diese Link-Semantik zu einer einzigen *virtuellen Instanz* verschmolzen; die Link-Semantik definiert also ein *logisches Datenmodell*.

Extended Links repräsentieren Beziehungen zwischen *XML-Resourcen*, nur sind *Extended Links* nicht wie *Simple Links* im einbettenden Dokument definiert, sondern unabhängig von den referenzierten Resourcen in einem eigenen Dokument. So ist es möglich, von dritter Seite aus Sichten über mehrere beteiligte XML-Dokumente zu beschreiben, ohne auf die Dokumente selber Schreibrechte haben zu müssen. Das *logische Datenmodell für Simple Links* wird also erweitert auf *Extended Links*, denen so eine *3rd Party Link*-Semantik zugeeignet wird.

Das oben beschriebene *logische Datenmodell* liefert eine Semantik zur Auswertung von *XPath*-Anfragen über räumlich verteilte, mit *XLink* verknüpfte XML-Daten: Anfragen werden nicht auf dem physischen Datenmodell des angefragten Dokumentes, sondern auf der zugehörigen, durch die beteiligten Dokumente und Links definierten *virtuellen Instanz* ausgewertet. Die Auswertung "läuft" entlang eines *Simple Links* in ein anderes Dokument und wird dort fortgesetzt. Ebenso können Anfragen auf einer integrierten Sicht ausgewertet werden, die durch *Extended Links* mit 3rd-Party-Semantik definiert wird.

In einer früheren, im Rahmen von *LinXIS* entstandenen Dissertation wurde ein Datenmodell für *Simple Links* beschrieben und Verfahren und Algorithmen zur Auswertung von *XPath*-Anfragen auf diesem Datenmodell untersucht. Als Teil der Arbeit entstand eine Implementierung des Datenmodells auf Basis des Open-Source-XML-Datenbanksystems *eXist*, so dass der dadurch entstandene Prototyp in der Lage ist, Anfragen auf mit *Simple Links* verknüpften Daten gemäß dem Datenmodell auszuwerten.

In der vorliegenden Arbeit wird der Fokus auf *Extended Links* erweitert: Teil der Arbeit ist eine formale Beschreibung eines gemeinsamen Datenmodells für *Simple Links* und *Extended Links* (letztere versehen mit 3rd-Party-Link-Semantik), spezifiziert in Form eines *abstrakten Datentypen (ADT)*. Darüber hinaus werden grundlegende Aspekte der Anfrageauswertung in Bezug auf die 3rd-Party-Link-Semantik untersucht. Die beschriebenen Techniken und Algorithmen die das obige *logische Datenmodell* implementieren, werden prototypisch umgesetzt. Dazu wird der bereits vorhandene *Simple Link*-fähige, *eXist*-basierte Prototyp weiterentwickelt. Der so entstandene neue Prototyp wird in einer Fallstudie auf Funktion und Leistung hin untersucht. Anschließend wird der gesamte 3rd-Party-Link-Ansatz kritisch diskutiert in Bezug auf Design, Leistungsfähigkei und Relevanz im Kontext einer sich kontinuierlich verändernden Web-Infrastruktur. Schließlich wird diese Diskussion in einem Fazit abschließend bewertet. Hier wir auch ein zusammenfassender Überblick über andere Arbeiten auf dem Gebiet sowie über Perspektiven zur Weiterentwicklung und Umsetzung gegeben.

Contents

Chapter 1

Introduction

XML – One Among Many

For representing and modeling data, lots of languages, specifications, standards, formalisms and notations exist. UML class diagrams, for example, are used for modeling object-oriented software systems. Entity-Relationship diagrams provide a data model for describing general-purpose data, which can be easily transformed into the relational schema of a relational database system. There are semistructured data models like XML. RDF serves for expressing relations between Web resources, OWL is a language for defining and using ontologies. In these terms, XML is one among many.

XML – One Among Few

After its publication in 1998, XML quickly gained widespread acceptance as well in the research community as among commercial and private software creators. On one hand, XML is very simple to use: an XML document can be written using a plain text editor, since its file format is Unicode. Which means: authoring access is simple. It has a fixed structure consisting of well-formed tags and attributes, similar to HTML, but without HTML's syntactic fault tolerance. With that, it is easy to create simple, performant and highly customizable XML parsers (which makes reading/processing simple). On the other hand, XML offers a high degree of flexibility, since each document type can be customized to a specific data domain: the vocabulary's items represent the basic concepts of the given domain, rules define relationships between these concepts. Consider e.g. a domain "book". A document type book could have a vocabulary covering the basic concepts of a book: book, chapter, section, paragraph, author, etc., as well as a set of rules describing that a book has one or more authors, it has a number of chapters, each chapter has a number of sections, each section consists of a number of paragraphs, each paragraph contains portions of the book's literal text body. Each document type together with the XML syntax defines a language for describing data from a certain domain, with a “word” in such a data domain language being called an *XML instance* or *document*.

Summarizing the above, XML serves as a *meta language* for data representation with highly complex – and as well performant — querying and manipulation mechanisms, as well as with low requirements regarding the essentially needed tool infrastructure. Which makes XML one among *few*.

XML – Syntax and Data Model

XML documents have a strict syntax, based on a hierarchical structure of well-formed tags, attributes and literal values. The concepts of *elements* (e.g. book) having text and/or other elements, e.g. title, chapter) as their *contents*, of attributes (attributed to an element) and literal values (inside attributes or *text children* of an element), along with some additional types as entities, comments and namespaces altogether imply a hierarchical data model: an XML document represents a tree data structure, with the tree's nodes being elements, attributes, text nodes etc.

XML documents have also a text representation: Elements are represented by their name, given in pointy brackets: <book>...</book> denotes a book element, with the element's *content* being enclosed between the *opening tag* <book> and the *closing tag* </book>. Attributes assigned to an element are written as key-value pairs as part of the opening element:
<book isbn="978-3518188187">...</book>.

When accessing an XML document, one can follow the textual representation of the document (e.g. in a file), or one can follow the tree data model of the document[1]. When accessing a graph-based data model (such as a tree), the access is no more sequential, as for text files, but navigation-based on notions as neighbor (graph) or parent-child (tree) relationships. Query or transformation engines as well as certain parsers[2] operate on basis of the XML data model instead of its textual representation. The *XPath* [XPa99] Data Model extends the XML data model by introducing so-called *axes*, which enable for navigation inside the document tree. E.g. the child axis of an element yields all element and text children as result. Other axes are attribute, parent, descendant or self.

Linking XML – The XLink Standard

XML Documents are monolithic: one single document can be seen as a single file. HTML documents are also monolithic in some way, since each HTML

[1]The textual representation induces the data model tree, and vice versa, the textual representation can be obtained by a pre-order traversal of the tree structure.

[2]For XML, two families of parsers exist. *DOM* [DOM98] parsers adhere to XML's *Document Object Model*, where *SAX (Simple API for XML)* parsers refer to XML's serialized textual representation.

document is located in a single HTML file[3]. But within hypertext, connections between documents can be expressed using *hyperlinks*. Everyone has an intuitive idea of the concept of a hyperlink, since hyperlinks are a part of HTML, the document language of the ubiquitous World Wide Web. Hyperlinks are followed by clicking them in a browser. So, would it make sense to adopt the concept of the hyperlink for the XML world? What would be its benefits? What would be its properties? What *syntax* and what kind(s) of *semantics* would a hyperlink have?

In 1999, the World Wide Web Consortium [W3C] published the XLink recommendation [XLi01b], which defines a link as "an explicit relationship between resources or portions of resources". *Simple Links* reference an *XML resource* from a document (more precisely: from the *linking element*, which contains the Simple Link markup). An XML resource is another document, or part(s) of another document. *Extended Links* represent complex relationships between resources. Resources can be either locally defined inside the Extended Link, or via *locators* using XPointers pointing to remote resources. The relationships itself are modeled by *arcs*, which are unidirectional connections between resources.

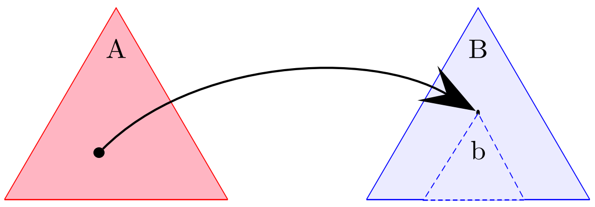


Figure 1.1: Simple Link – reference from instance A to resource b in instance B

Simple Links (see Figure 1.1) always "start" in the document where they are defined, and point to some remote resource. Extended Links may contain locally defined resources, locators that point to to remote resources, and arcs connecting these resources. In contrast to Simple Links, Extended Links are not defined inside the documents that they link together, but outside of these in a *linkbase* document (see Figure 1.3). This can be useful for linking remote XML data with no authoring / write access granted.

Link Expansion and Logical Data Model

When bringing Simple Links into play, the question comes up how to integrate Simple Links into the XML data model. Are links integrated into the data model as a novel kind of relationship, or will they rather be mapped to existing relationships as `child`, `attribute` etc.? The first option demands an explicit way

[3]For the sake of simplicity, techniques like *HTML frames*, which bring together multiple HTML documents in one screen presentation, are not considered here.

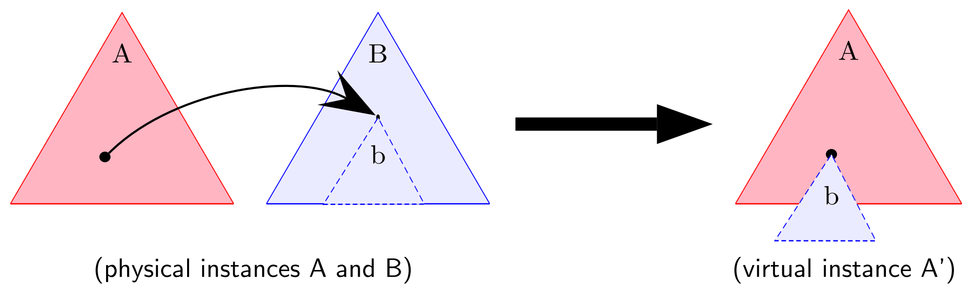


Figure 1.2: Simple Link – Mapping from physical to logical data model

of navigation along links. E.g. XPath needs to be equipped with an additional XLink axis, or some kind of dereferencing function, for being able to follow and evaluate XLink references.

The second option is to blend the link results *transparently* into the current data model, right into the position where the Simple Link element was defined. The *linking relation* to a (previously remote) resource is mapped into a regular relation in the XML data model, such as child-of, neighbor-of or attribute-of. This resembles cutting the referenced material out of the linked remote tree and pasting it into the currently navigated tree, thereby *expanding* the Simple Link element (see Figure 1.2).

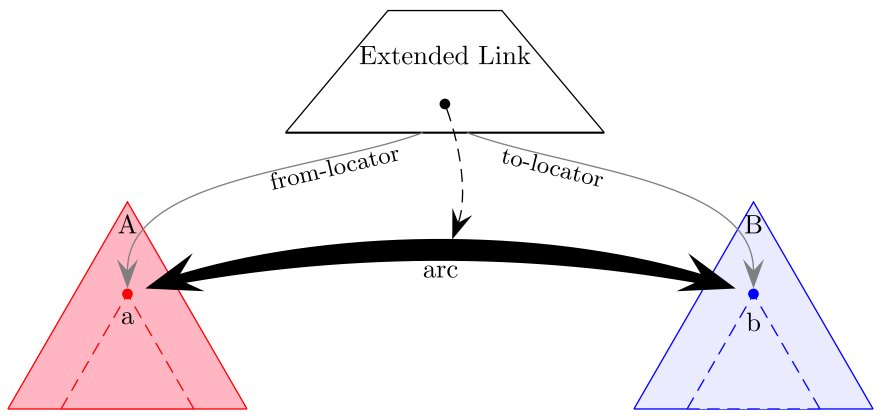


Figure 1.3: Extended Link with two locators identifying remote resources, and one arc connecting both resources

The first option, explicit navigation, brings an additional notion of relationship into the XML modeling. In contrast to that, the approach involving transparent expansion of links makes it possible to reference objects from remote instances as if they were locally defined. In terms of data integration and data distribution scenarios, the latter approach seems the more sophisticated and promising one, since it enables for sharing XML data across multiple places without regarding the concrete location of a requested piece of data. This can be useful in scenarios of distributed authoring or data fragmentation. Generally, it seems to be a more flexible and superior approach toward the modeling capabilities, to distinguish between data items in terms of their intrinsic properties instead of in terms of their physical locations on the Web.

Note that the "transparent approach" implies a mapping from XML to XML: XLink's Simple Links are syntactically described in XML. The structure induced by the original XML plus the link information is also XML, since all links are transformed into plain XML constructs. This motivates the definition of the following terms:

transparent link expansion: an XLink element expresses a link relation to some remote XML resource. When traversing such a link element, the remote result is *transparently* blended into the currently navigated instance, with the remote data being seamlessly integrated into the traversed instance. The link is said to be *expanded*.

physical and virtual instance: when traversing an XML document containing XLinks (the *physical instance*) from top down, expanding every found XLink, the completely expanded result instance is called *virtual instance*[4].

physical data model and logical data model: The rules of how to map sets of physical instances to a virtual instance by expanding the contained XLinks provide the semantics of the *logical XLink data model*.

Extended Links and 3rd Party Semantics

In [BFM06a], the "transparent approach" was described including both a specification of its logical data model and the description of a prototypical implementation for XLink Simple Links. For Extended Links, the situation is different due to their different structure. Since Extended Links refer to resources in remote documents, and since an Extended Link's arc is a directed connection with a from and a to resource, an arc's impact on the logical data model depends on the traverser's perspective.

- The document containing the from-resource is traversed (document A in Figure 1.4). When data of the from-resource is traversed, the to-resource data is transparently blended into the traversed document. Thereby, the from-resource's document, the to-resource and the linkbase together specify a virtual instance (document A' in Figure 1.4). This perspective is called the forward perspective.

[4]Note that such a virtual instance is not necessarily finite, since it may contain cycles. More on this issue can be found in Section 5.2.

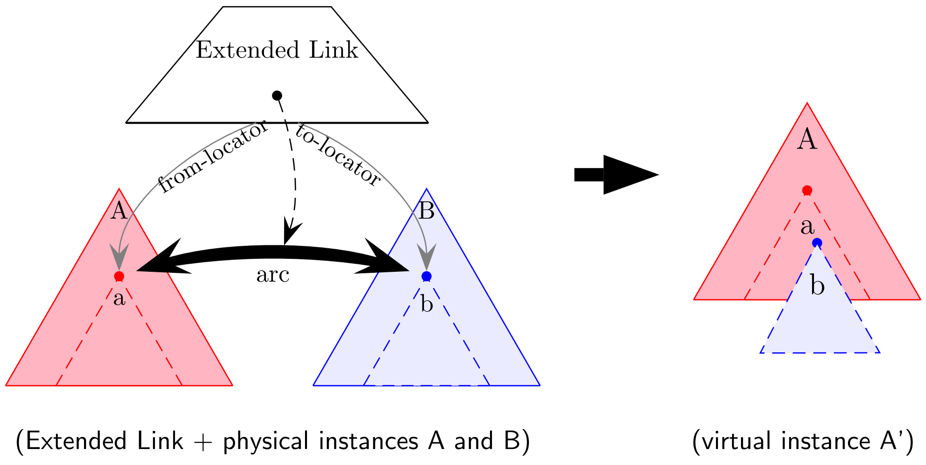


Figure 1.4: Extended Link (i) – logical data model in forward perspective

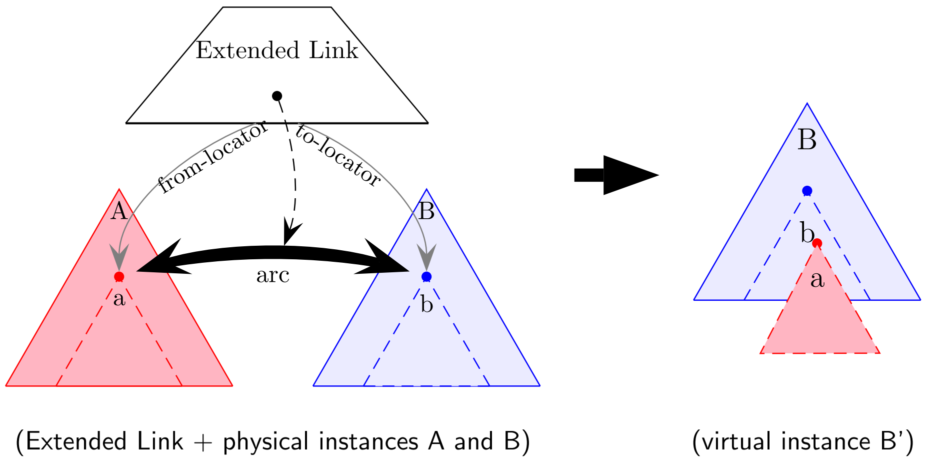


Figure 1.5: Extended Link (ii) – logical data model in inverse perspective

- The document containing the to-resource is traversed (document B in Figure 1.5). Here, the from-resource data is blended into the to document, analogue to forward perspective. Since both perspectives can be considered symmetrical to each other (with interchanged from and to ends), this perspective is named inverse perspective.
- A third perspective is anchored to the linkbase: When the linkbase itself is traversed, arcs inside Extended Links can be expanded by blending the referenced remote resources into the arc element. Since this perspective creates a view based on the relation between the from and to resources that the arc establishes, it is called the relation perspective (Figure 1.6).

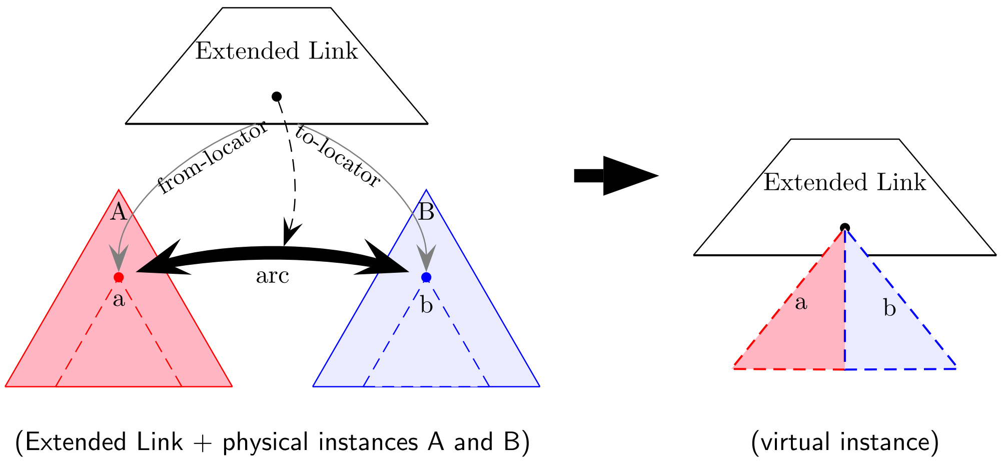


Figure 1.6: Extended Link (iii) – logical data model in relation perspective

Perspectives forward and inverse are well-suited for creating views (as they are known from relational database systems) on remote, read-only data sources on the Web, while the perspective relation can be seen as an extension of the logical model induced by Simple Link semantics, since the link information is located in the traversed document itself.

This Work as a Part of the LinXIS Project

This work is embedded into the LinXIS project [Lin], which focuses on semantics for XLink-connected XML data regarding the evaluation of queries. A number of publications exist which present and document the research work and the achieved results throughout the LinXIS project, with two of them being of particular relevance for this work: "Handling Interlinked XML Instances on the Web" [BFM06a] contains a formal description of the logical data model for Simple Links, and "Querying along XLinks in XPath/XQuery: Situation, Applications, Perspectives" [BFM06b] describes evaluation techniques for Simple Links, representing two essential building blocks in the scope of this work.

Another work which emerged from the LinXIS project, the PhD thesis of my then-coworker Erik Behrends [Beh06], is strongly linked to this one: in Erik's thesis, the semantics and evaluation techniques for Simple Link-connected, distributed data was investigated. Part of his work was a prototype implementing the Simple Link data model by extending the Open Source XML database system *eXist* [exi].

This work extends the previous research by (i) giving a formal description of the logical data model as an abstract data type, covering Simple Links as well as Extended Links, by (ii) specifying a 3rd Party Link semantics for Extended Links, and by (iii) investigating query evaluation techniques for 3rd Party Links, validated by a proof-of-concept implementation of the data model and query evaluation, with an implementation based on the already existing Simple Link-aware prototype.

This work, as part of the LinXIS project [Lin], has been supported by the *Deutsche Forschungsgemeinschaft (DFG).*

Outline

In 2, an short recapitulation of the basic notions of XML, XPath, XLink and XPointer is given. 3 conceptually describes the semantics for navigating along XLink Simple Links. 4 does the same for XLink Extended Links. 5 specifies the logical data model for Simple Links formally by describing it as an abstract data type. In the same manner, 6 defines the logical data model for XLink Extended Links. 7 describes the algorithmic concepts of processing Extended Links in an XML database system, and describes the software prototype implementing these concepts. 8 contains a small case study which applies the Extended Link approach to a real-world example in shape of an airline schedule containing worldwide flight connections, and delivers some statistical query runtime results. 9 analyzes and discusses the 3rd-Party-Link approach critically considering its design, its performance behavior, and its function regarding the appropriateness and competitiveness in the context of modern Web infrastructure. The thesis is concluded by 10, pointing out the contribution of the proposed 3rd Party Link approach in terms of its concept and of its realization and giving an overview over related research done in that area, as well as giving an outlook over further work and perspectives.

Chapter 2

Preliminaries

The purpose of this chapter is to give a brief introduction to the XML-related concepts XPath and XPointer, which are necessary for understanding the XLink language. Also, the concept of links in HTML is shortly revisited, with a focus on the similarities between HTML Hyperlinks and XLink, since historically the idea of the HTML hyperlink served as a blueprint for the XLink concept.

2.1 XML for Documents and Data

Since its publication in 1998, XML [XML98] has quickly become a central means for data integration and exchange, especially in application areas with heterogeneous data sources, with the most heterogeneous application of all being the World Wide Web itself.

XML is a meta language for representing data in a *semi-structured* fashion. The term *semi-structured* means that the data has a less rigid structure than e.g. a relational database (whose structure is given in the database *schema*), but it has more structure than raw data (for example a plain text file containing the complete text of a book, but without any markup or formatting structure denoting chapters, pages etc.[1]). This intermediate approach makes XML an appropriate choice for exchanging data between data sources as diverse as relational databases (with database schemas), any kind of Web Services (with a result adhering to some return type specification), or raw character data (adhering to no schema at all). So, XML enables for data exchange between heterogeneous data sources.

On the other hand, XML has initially been designed as an easier manageable alternative to the *Structured General Markup Language* (*SGML*), a document description meta language[2]. From an abstract point of view, a document consists of a sequence of atomic data items, as characters and numeric values,

[1]For a profound definition of the term "semi-structured", please refer to [Abi97]

[2]The design of XML can be seen as a stripped-down version of SGML, refining the rich, but extremely complex SGML to an essential subset.

together with some kind of a structure, denoted as *markup* adhering to a *document type*. A quite prominent markup language (and an application of SGML) is HTML (see Section 2.2), with HTML documents being the syntactical basis of what is commonly denominated as "the Web".

So, XML can be seen on one hand as a data representation meta language, and on the other hand as a meta language for document processing. The terms *schema* and *document type* describe more or less the same concept.

2.2 XML, HTML and Hyperlinks

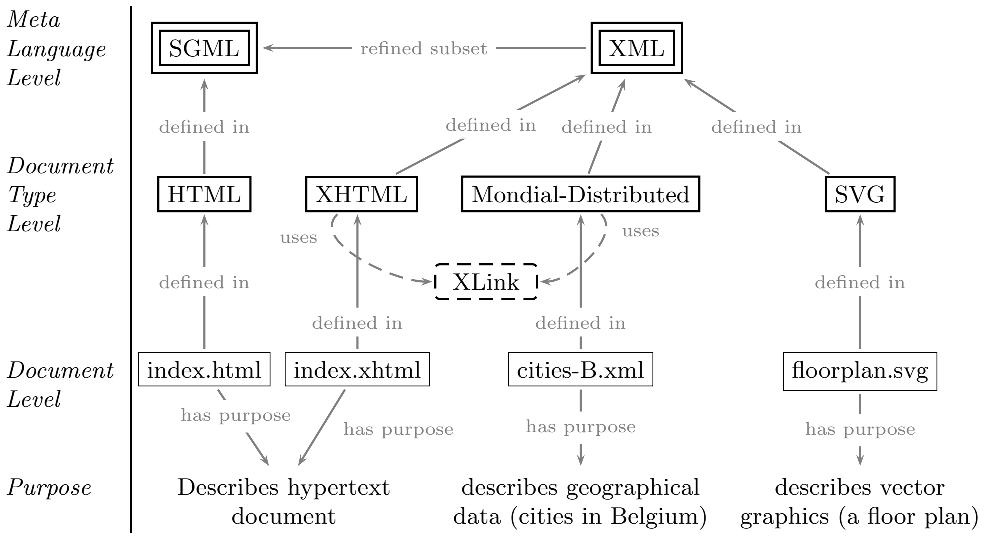


Figure 2.1: Relation of XML, SGML, HTML and XLink

XML and HTML are often (and some say, erroneously) considered similar because of their similar appearance: serialized as a data format, both contain textual information (*PCDATA*) interspersed with markup elements, given as *tags* in pointy brackets. However, there are differences in syntax and concept. HTML provides a fixed vocabulary of markup elements for describing hyperlinked Web documents with the purpose of being read ("browsed") online with a Web browser. Thus, HTML is a *document type*. XML, in contrast, is a *meta language*, allowing to define its own document types. Conceptually, XML is a follow-up to SGML rather than to HTML.

Nevertheless is the perception of XML as a successor of HTML still quite common, which is to some part owed to the historical background: one design goal for XML was to supply a markup language for the Web which was more flexible than HTML (since HTML had a fixed vocabulary), and which had a strict separation of content and layout (since HTML intermixes both, which does not enable for a clean conceptual modeling). Thus, XML documents were initially considered to be "consumed" in a browsing context by an XML browser software. However, the practitioner's perspective on the usage of XML has shifted quite a bit away from the browsing context since these days.

HTML and Hyperlinks: *Hyperlinks* enable the author of an HTML document to place references to other Web resources in the document, which then can be followed by clicking on the textual link representation in the browser. In this manner, HTML documents on the Web are connected by unidirectional edges or *links*. With hyperlinks, HTML contains a simple and robust mechanism for representing links from one document to either another document, or to some other resource, as text, image, video or audio files, which can possibly reside on a remote server. A resource is located by its URL, its *Uniform Resource Locator* [URL]. URLs to documents can be enhanced with a fragment identifier pointing to a pre-defined anchor inside the referenced document (see Figure 2.2).

An overview over the relationships between XML, SGML, HTML, XML and XLink, together with some example document and document types motivating the meta structure, is given in Figure 2.1.

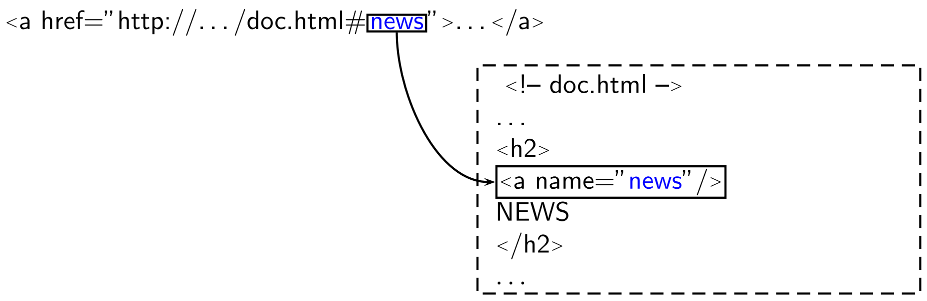


Figure 2.2: HTML Link with fragment identifier to anchor element

2.3 Linking XML Data

In contrast to HTML, native XML documents are self-contained without built-in features for creating links to other XML resources. Since XML initially was thought to be used also in a browsing context, the need was seen to equip XML also with a concept for defining hyperlinks. As the *eXtensible* in XML suggests, the hyperlink functionality was not built into the XML standard, but *XLink* was defined as a syntactical extension[3], which could be adopted by any XML document type to express hyperlinks between XML documents.

A first draft on “Linking XML” was formulated in 1997 [XLD97]. The XLink specification itself reached recommendation status in 2001, presenting a framework for linking of XML documents, featuring the notions of *Simple Links* and *Extended Links*. Simple Links are similar to HTML links in the point that they provide a unidirectional reference into another XML document, specified with a URI. But where for HTML links, fragment identifiers support only navigation to a pre-defined anchor, XLink takes advantage of a number of more sophisticated fragment identifier mechanisms, the most expressive one being XPointer [XPt02a].

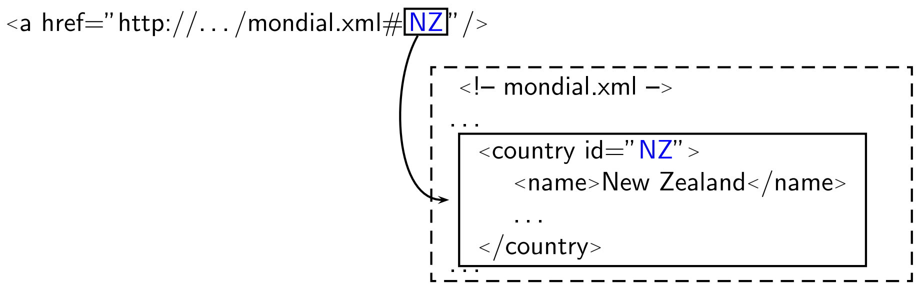


Figure 2.3: XLink with XPointer using shorthand addressing

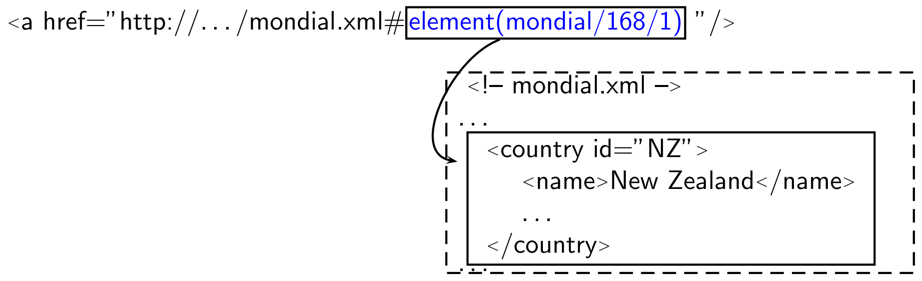
2.3.1 XPointer

XPointer [XPt02a] is a W3C standard for identifying fragments inside XML data instances. This enables for creating links to complete XML documents, to document fragments (which can even be contiguous text regions inside a document). There are three ways for identifying XML fragments: via ID, via child positions, or with XPath-based navigation:

- *shorthand pointers* (formerly *“barenames”*) identify a single element by the – unique – value of its ID attribute (see Figure 2.3).
- The element() scheme (formerly *“child sequences”*) identifies a single element by the position of its ancestors. E.g. doc.xml/1/7/2 means the 2nd child of the 7th child of the root element of document doc.xml (see Figure 2.4). Starting point is either the document root node or a single element identified by a shorthand pointer expression.

[3]The term “syntactical extension” shall *not* suggest that XLink extends the XML syntax – it does not. Instead, it denotes that XLink is not an own document type, but rather “some portion of syntax”. In few words, its just a number of attribute definitions, which can be adopted by every document type. When added to an XML element, the attributes describe a hyperlink *syntactically*, but not semantically. In Section 2.3.2, the XLink syntax is described in detail.

- with the xpointer() scheme, a fragment is identified by a XPath-based navigational expression (see Figure 2.5).



```
<a href="http://.../mondial.xml#element(mondial/168/1)"/>

<!– mondial.xml –>
...
<country id="NZ">
    <name>New Zealand</name>
    ...
</country>
```

Figure 2.4: XLink with XPointer using element() addressing scheme

The xpointer() scheme is in some sense superior to the other schemes, since it functionally comprises the other two: every element() or shorthand pointer expression can be rewritten into an xpointer() expression, but not vice versa. xpointer() in its function and syntax is an extension to XPath [XPa99], which was developed as a generic navigation mechanism for XML Query and Transformation Languages.

XPointer (or, to be precise, the xpointer() scheme defined as part of the XPointer language [XPt02a]) and XPath differ in two significant points:

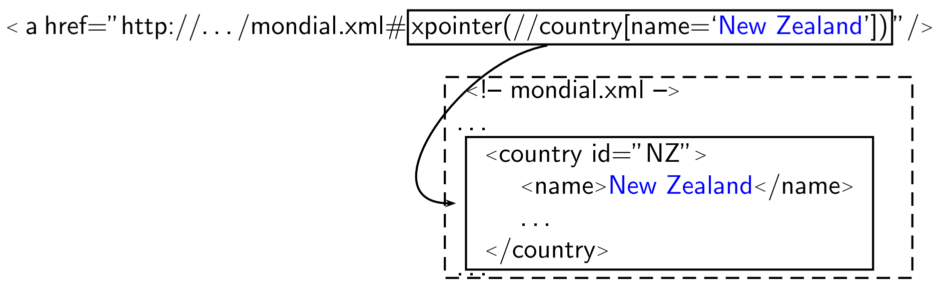
- **location versus node**: in XPointer, the concept of *nodes*, *node types* and *node-sets* is generalized to *locations*, *location types* and *ranges*, to enhance navigation inside and across neighboring text nodes (e.g. for marking contiguous text regions inside an XML document and referencing them with an XPointer).

- **root nodes**: in contrast to XPath, XPointer allows the root node of the referenced XML data instance to have arbitrary types and numbers of node children, instead of a single root element node, in order to allow expressions to address location sets inside arbitrary external parsed entities (which are not necessarily in tree structure) as well as well-formed documents.

2.3.2 XLink Syntax

XLink enables for creating links between *XML resources*. The XLink language is expressed in XML itself. The two existing linking constructs are *Simple Links* representing unidirectional connections between an XLink element and a data from a remote instance, and *Extended Links* using *arcs* to connect *local resources* and/or *remote resources* which are identified with *locators*[4].

[4]Further reading: [WL02] give a comprehensive overview on the XLink/XPointer area. The official W3C XLink recommendation [XLi01a] serves as authoritative source on the XLink



```
< a href="http://. . . /mondial.xml#xpointer(//country[name='New Zealand'])" />

<!– mondial.xml –>
. . .
<country id="NZ">
    <name>New Zealand</name>
    . . .
</country>
. . .
```

Figure 2.5: XLink with XPointer using xpointer() addressing scheme

Simple Links

Simple Links are similar to HTML <a> *(= anchor)* elements with href attributes. An XLink *Simple Link* element is equipped with the additional XLink attributes xlink:type and xlink:href. xlink:type is the attribute which makes a regular XML element be an XLink element. therefore, it is mandatory for all kinds of XLink elements. The xlink:href attribute contains a URI identifying a remote XML resource.

Example 1 *Consider an XML Element* country *which contains data about a specific country, e.g. New Zealand. Consider a remote XML instance* cities-NZ.xml *containing geographical data about cities in New Zealand. Then, the country element could contain a reference to the city data of its own capital Wellington, which is residing at the remote* cities-NZ.xml *instance. So, the information that Wellington is the capital of New Zealand can be expressed by referencing the* city *element of Wellington from the* country *Element of New Zealand, without need of duplicating the* city *data:*

```
<country car_code="NZ">
  <name>New Zealand</name>
    <capital  xlink:type="simple"
              xlink:href="http://. . . /cities-NZ.xml#xpointer(//city[name='Wellington'])" />
  . . .
</country>
```

The XLink attributes are:

- xlink:type="simple" indicates that capital is a Simple Link,
- xlink:href="cities-NZ.xml#xpointer(//city[name='Wellington'])" indicates a reference to the city element inside cities-NZ.xml with a child element name with the text content "Wellington".

standard.

Simple XLinks can have the following XLink attributes:

name	**function**	**allowed values**
xlink:type	indicating “this is a link element”	“simple”
xlink:href*	contains XPointer reference	URI / XPointer expression
xlink:role*	declares role of the XLink element	URI referring to role resource
xlink:title*	human-readable title	CDATA
xlink:show*	determines browsing behavior	{new,replace,embed,other,none}
xlink:actuate*	determines browsing behavior	{onLoad,onRequest,other,none}

Starred(*) attributes are optional, all others are mandatory.

Note that Simple Links (as well as all other XLink elements) are not identified as Simple Links by their name. Any arbitrary XML element, without regard of its name, its attributes or child nodes, can be made an XLink element by adding an xlink:type attribute with one of the values simple, extended, arc, locator, resource or title.

Extended Links

As the name suggests, Extended Links differ more from HTML links concerning their modeling functionality, which makes also the syntax more complex. An Extended Link contains:

- zero or more XML resources, either local (contained child elements etc.) or remote (specified by a URI given in a locator element),
- zero or more directed arcs connecting these resources,
- optionally a title, and
- optionally some none-XLink-related content.

Local resources: an Extended Link can contain local XML data in form of zero or more resource elements, which may contain arbitrary XML data. Each resource element has an attribute xlink:label by that arcs can refer to it.

Remote resources are XML data existing outside the Extended Link element. A remote resource is identified (and thereby defined) by a locator element.

Locators: An Extended Link contains zero or more locator elements. A locator contains a URI pointing to a *resource*[5] outside the link location, and – just as local resource elements – an xlink:label attribute.

[5]The term *resource* in that context refers to any kind of XML data which can be described by a URI expression identifying a document and using any of the available schemes of the xpointer language as given in Section 2.3.1. A resource defined that way can consist of a single XML node, or of multiple XML nodes, which do not necessarily form a contiguous document fragment, but can be single, isolated nodes spread over a document, or even over multiple documents.

Arcs represent directed connections between resources. An Extended Link defines a graph with resources as vertices, and arcs as edges. Arcs from local resources to remote resources are called *outbound* arcs, arcs from remote resources to local resources are called *inbound* arcs, arcs connecting remote resources are called *3rd party arcs.* Arcs have an xlink:from attribute and an xlink:to attribute, denoting the start and the end resource of the arc[6]. Local resources are identified by the value of their xlink:label attribute. Remote resources are identified by their locator element, which in turn is addressed by its xlink:label attribute's value.

Example 2 *Consider an Extended Link element* flightplan *which contains data about flight connections of the airline* Y.A.A.[7]*. The cities are modeled as resources. Most cities referenced by the Extended Link can be taken from the* Mondial *XML database. These remote resources are described by locator elements. Other, less prominent cities (e.g.* Anytown *located in* Somecountry*) are modeled inside the link as local resources. Cities are connected with flight routes, modeled as arcs, establishing direct connections from one city to another (see Figure 2.6).*

Extended Links have the following XLink attributes:

name	**function**	**allowed values**
xlink:type	indicating "this is a link element"	"extended"
xlink:role*	declares role of the XLink element	URI referring to role resource
xlink:title*	declares human-readable title	CDATA

Except for Simple Links and Extended Links, the other XLink elements as arcs, resources, locators and titles may be children of an Extended Link element. Like Simple and Extended Links, they are identified by an xlink:type attribute with the respective value.

title:

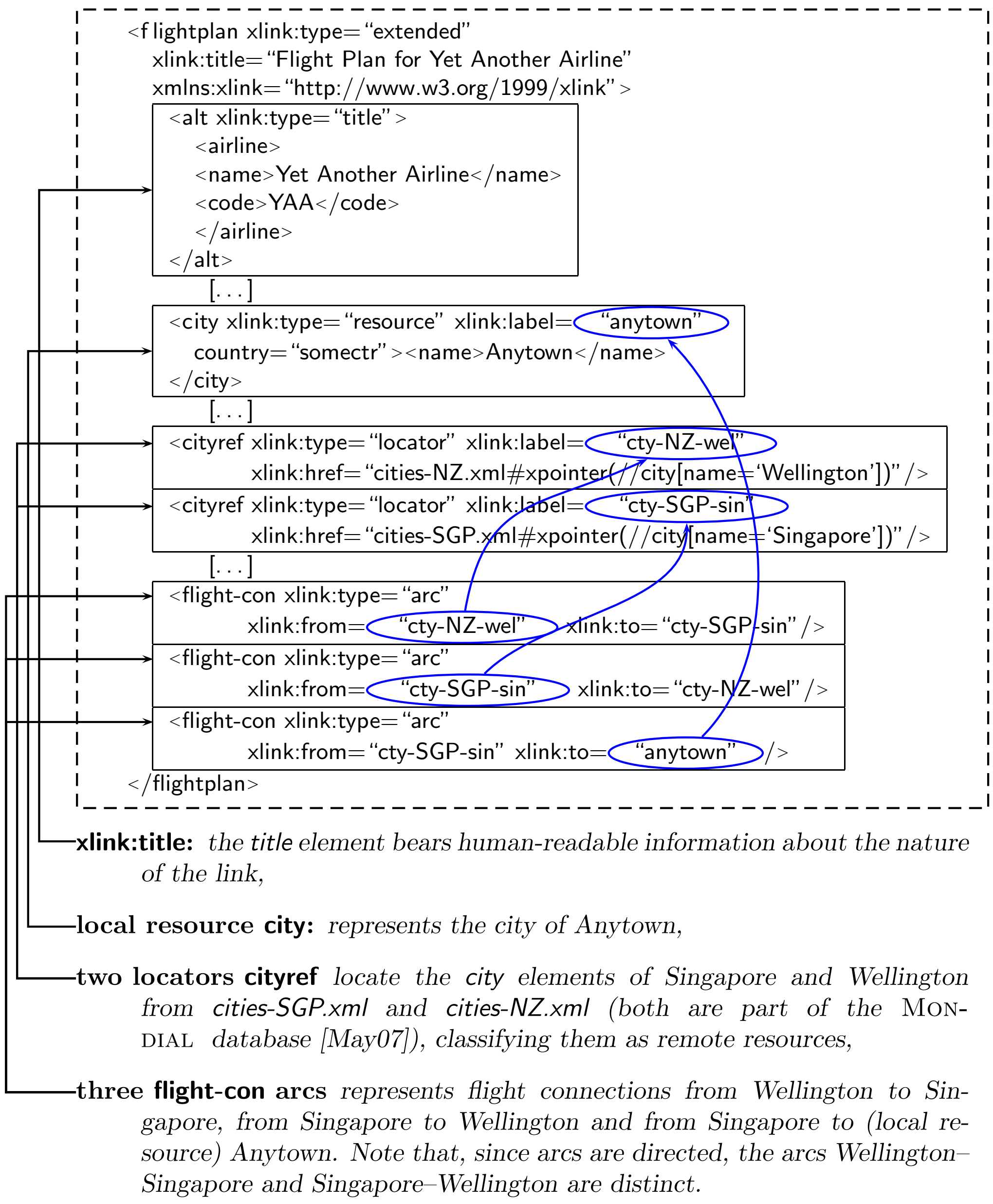
name	**function**	**allowed values**
xlink:type	indicating "this is a link element"	"title"

resource:

name	**function**	**allowed values**
xlink:type	indicating "this is a link element"	"resource"
xlink:role*	declares role of the XLink element	URI referring to role resource
xlink:title*	human-readable title	CDATA
xlink:label	for identification by arc(s)	NMTOKEN

[6]The "regular" and most intuitive case is an arc connecting exactly one resource (*from*) with exactly one other resource (*to*). But arcs can also associate multiple resources: the xlink:label values inside an Extended Link are not necessarily unique. Hence, one arc can address multiple *from* and *to* resources by one single label.

[7]could stand for "Yet Another Airline", national Airline of the *Republic of Somecountry.*



```
<flightplan xlink:type="extended"
  xlink:title="Flight Plan for Yet Another Airline"
  xmlns:xlink="http://www.w3.org/1999/xlink">
  <alt xlink:type="title">
    <airline>
    <name>Yet Another Airline</name>
    <code>YAA</code>
    </airline>
  </alt>
      [...]
  <city xlink:type="resource" xlink:label="anytown"
    country="somectr"><name>Anytown</name>
  </city>
      [...]
  <cityref xlink:type="locator" xlink:label="cty-NZ-wel"
           xlink:href="cities-NZ.xml#xpointer(//city[name='Wellington'])"/>
  <cityref xlink:type="locator" xlink:label="cty-SGP-sin"
           xlink:href="cities-SGP.xml#xpointer(//city[name='Singapore'])"/>
      [...]
  <flight-con xlink:type="arc"
           xlink:from="cty-NZ-wel" xlink:to="cty-SGP-sin"/>
  <flight-con xlink:type="arc"
           xlink:from="cty-SGP-sin" xlink:to="cty-NZ-wel"/>
  <flight-con xlink:type="arc"
           xlink:from="cty-SGP-sin" xlink:to="anytown"/>
</flightplan>
```

xlink:title: *the title element bears human-readable information about the nature of the link,*

local resource city: *represents the city of Anytown,*

two locators cityref *locate the city elements of Singapore and Wellington from cities-SGP.xml and cities-NZ.xml (both are part of the MONDIAL database [May07]), classifying them as remote resources,*

three flight-con arcs *represents flight connections from Wellington to Singapore, from Singapore to Wellington and from Singapore to (local resource) Anytown. Note that, since arcs are directed, the arcs Wellington–Singapore and Singapore–Wellington are distinct.*

Figure 2.6: Extended Link containing the flightplan of "Yet Another Airline"

locator:

name	function	allowed values
xlink:type	indicating "this is a link element"	"locator"
xlink:href	contains XPointer reference	URI / XPointer expression
xlink:role*	declares role of the XLink element	URI referring to role resource
xlink:title*	human-readable title	CDATA
xlink:label	for identification by arc(s)	NMTOKEN

arc:

name	**function**	**allowed values**
xlink:type	indicating “this is a link element”	“arc”
xlink:from	specifies connection’s starting point	label value/NMTOKEN
xlink:to	specifies connection’s end point	label value/NMTOKEN
xlink:arcrole*	declares role of the arc	URI referring to role resource
xlink:title*	human-readable title	CDATA

Starred(*) attributes are optional, all others are mandatory.

All XLinks, Simple Links as well as Extended Links, can also have non-XLink attributes and children, with no XLink-specific meaning for the XLink element.

2.3.3 Remarks

Since the W3C and IETF standards and recommendations for XPath, XPointer, XQuery – and almost all other XML-related technologies – are quickly evolving, there is an obvious need to specify the version / state of the art of these technologies as they are used, understood and cited in scope of this work. This work refers to:

- *XPath*: XML Path Language (XPath) Version 1.0, W3C Recommendation 16 November 1999 [XPa99]
- *XPointer*:
  - XML Pointer Framework (XPointer), W3C Recommendation 25 March 2003 [XPt03b]
  - XML XPointer element() Scheme, W3C Recommendation 25 March 2003 [XPt03a]
  - XPointer xmlns() Scheme, W3C Recommendation 25 March 2003 [XPt03c]
  - XPointer xpointer() Scheme, W3C Working Draft 19 December 2002 [XPt02b]
- *XLink*: XML Linking Language (XLink) Version 1.1, W3C Recommendation 27 June 2001 [XLi01a]

The namespace for the XML Linking Language is http://www.w3.org/1999/xlink. Throughout the examples in this work, The namespace is always bound to the namespace prefix xlink, if not stated otherwise.

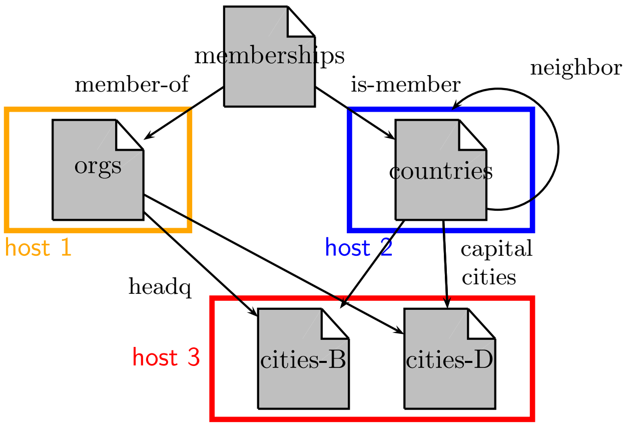
Chapter 3

Querying XML Data with Simple Links

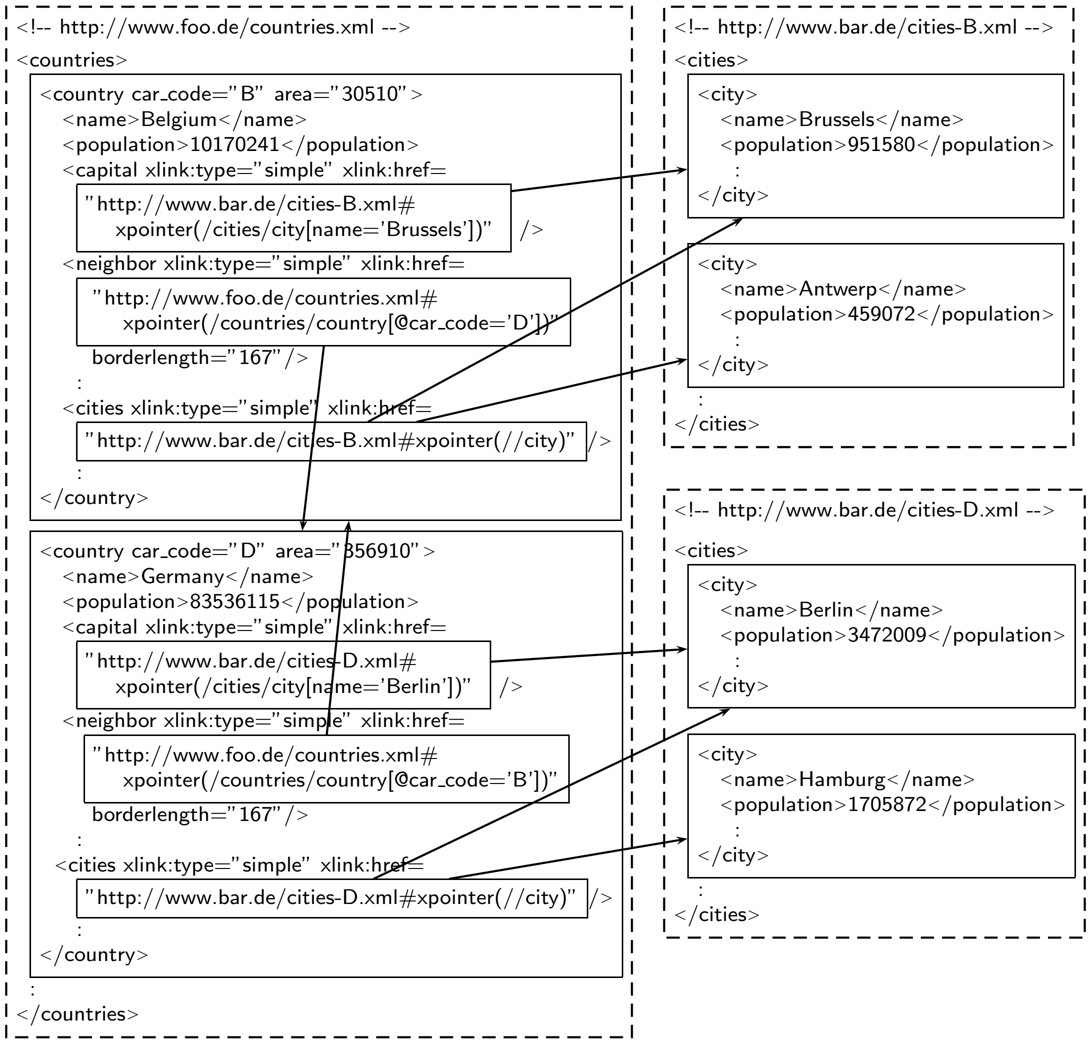
3.1 Query Support for XLinks

Consider the following XLink example: The geographical database MONDIAL is split up into several instances and distributed over a number of host locations. An instance countries.xml contains country data, instances cities-UK.xml, cities-B.xml and cities-D.xml contain data about all cities of a specific country (here, cities in the U.K., in Belgium and in Germany).

The fact that Antwerp is in Belgium is expressed via a Simple Link from inside the Belgium element in countries.xml to Antwerp's city element in the cities-B.xml document (at Figure 3.1). The fact that global organizations have members (countries) is represented with one Extended Link, containing one arc for each country↔organization membership relation:



How can XML documents linked in this way be queried? Many relations in the modeled data are expressed with XLinks. E.g. for finding out how many



```
<!-- http://www.foo.de/countries.xml -->
<countries>
  <country car_code="B" area="30510">
    <name>Belgium</name>
    <population>10170241</population>
    <capital xlink:type="simple" xlink:href=
      "http://www.bar.de/cities-B.xml#
        xpointer(/cities/city[name='Brussels'])"  />
    <neighbor xlink:type="simple" xlink:href=
      "http://www.foo.de/countries.xml#
        xpointer(/countries/country[@car_code='D'])"
      borderlength="167"/>
    :
    <cities xlink:type="simple" xlink:href=
      "http://www.bar.de/cities-B.xml#xpointer(//city)" />
    :
  </country>
  <country car_code="D" area="356910">
    <name>Germany</name>
    <population>83536115</population>
    <capital xlink:type="simple" xlink:href=
      "http://www.bar.de/cities-D.xml#
        xpointer(/cities/city[name='Berlin'])"  />
    <neighbor xlink:type="simple" xlink:href=
      "http://www.foo.de/countries.xml#
        xpointer(/countries/country[@car_code='B'])"
      borderlength="167"/>
    :
   <cities xlink:type="simple" xlink:href=
      "http://www.bar.de/cities-D.xml#xpointer(//city)" />
    :
  </country>
  :
</countries>
```

```
<!-- http://www.bar.de/cities-B.xml -->
<cities>
  <city>
    <name>Brussels</name>
    <population>951580</population>
      :
  </city>
  <city>
    <name>Antwerp</name>
    <population>459072</population>
      :
  </city>
  :
</cities>
```

```
<!-- http://www.bar.de/cities-D.xml -->
<cities>
  <city>
    <name>Berlin</name>
    <population>3472009</population>
      :
  </city>
  <city>
    <name>Hamburg</name>
    <population>1705872</population>
      :
  </city>
  :
</cities>
```

Figure 3.1: Excerpt of the Distributed MONDIAL XML Database [May07]

inhabitants the capital of Belgium has, it would be necessary to gather data from two different documents – countries.xml and cities-B.xml, possibly on two different hosts – during a single query execution.

The *XML Query Requirements* [XMQ03][1] explicitly state that querying

[1]The XML Query Requirements led to the specification of the *XML Query Language (XQuery)* by the World Wide Web Consortium. *XPath* is an XML navigation language based on path expressions, and is an integral part of XQuery. Thus, all XPath functions can be used within XQuery; that's why for the scope of this work there is no distinction between XPath and XQuery functions, using the term "XPath/XQuery function" instead. In the specification *XQuery 1.0 and XPath 2.0 Functions and Operators* [XPQ07], the distinction

along references, both within an XML document and between documents, must be supported. Intra-document references are modeled in XML using the ID-IDREF construct. In XQuery, these references can be explicitly dereferenced with the XPath/XQuery function id(). Inter-document references in XML documents can be expressed with XLink constructs. How can they be queried? Can they be queried at all?

With the XPath/XQuery function document(), a remote document can be identified in a query, and with
let $pointer :=

doc("http://.../countries.xml")//country[name="Belgium"]/capital/@href/string(),

one can select the URI value of the capital element's href attribute:
"http://.../cities-B.xml#xpointer(/cities/city[name='Brussels'])", which references the city document of Brussels. But inside XQuery, that attribute value is just a string, which cannot be resolved in order to dereference the capital Simple Link. Hence, inter-document xlink:href references as the above cannot be resolved in XQuery, at least not in general.

However, there exist some exceptions: If the URI's XPointer expression is a shorthand pointer, as "http://.../countries.xml#B", or an XPointer scheme with an explicit ID value given, as in "http://.../countries.xml#xpointer(id(B))", the URI can be resolved by combining the document() and the id() functions. Also, there exist XML processing applications that provide proprietary functions which can be used to supply that functionality. E.g., the Saxon XML processing software [Kay] provides an XSLT extension function saxon:evaluate() which can be used to evaluate an XPath expression within a remote document specified by Saxon's doc function. Furthermore, [RBHS04] propose an XQuery extension with"execute at *uri* xquery { *xquery* }".

These solutions either work only on restricted URIs, or within non-XQuery-standard software solutions. Within the scope of standard XQuery functions as given in *XQuery 1.0 and XPath 2.0 Functions and Operators*, the described dereferencing functionality cannot be made available for the general case.

Apart from being insular, the above approaches for querying in the presence of XLink references require *explicit* link dereferencing. Preferable to this would be an approach for handling distributed XML data where the links are *transparent* in the sense that they are seamlessly embedded into the common XML / XPath data model, so that queries could follow the links *implicitly* to the referenced nodes in other documents without "minding the gap" between two linked documents. This leads to a logical data model where distributed, XLinked XML documents represent a *single, virtual, integrated XML instance*, as shown in Figure 3.2. The XLink elements are seen as view definitions that integrate the referenced XML data into the referencing XML instance. The XLink element specifies the referenced nodes, and how they are mapped seamlessly into the surrounding instance. Of special interest is here, how the *link relation* is

between XQuery and XPath functions also has been given up.

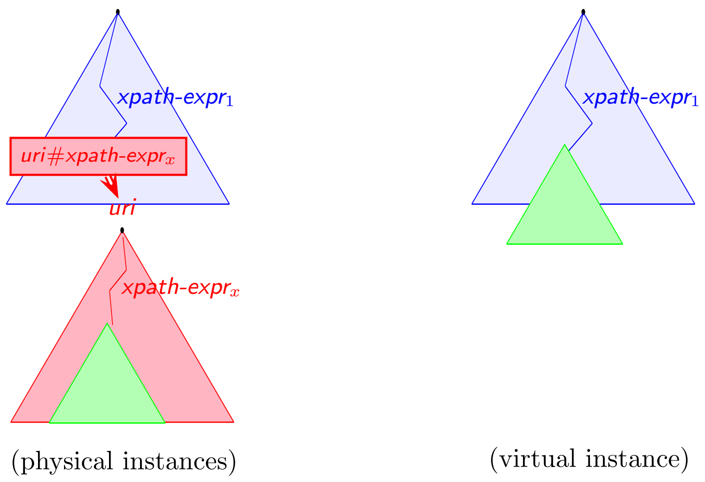


Figure 3.2: Extended XML Data Model with XLink Elements

mapped to a standard XML data model relation (e.g. *child* or *attribute* relation). The virtual instance can then be processed with standard languages like XPath, XQuery, or XSLT without need for specific link dereferencing operators.

3.2 Applications: Data Integration and Splitting Documents

The usage of linked XML information occurs mainly in two situations:

- Data integration: building (virtual) XML documents by combining autonomous resources. The referenced resources may be given as remote documents on the Web without write access.

- Splitting and distributing documents: An XML document can be split up into parts and distributed over multiple servers. With the use of XLinks, these parts can be interconnected to form a distributed database. In this case, it is intended to keep the *external schema*[2] unchanged, i.e., the virtual instance of the linked documents should be valid wrt. the original document's DTD/Schema. The idea is, to get the same answers from the distributed database as from the original one *for each query*. This requires the links' "cutting edges" – that can be between elements and their subtrees, or between elements and their attributes – to be reassembled flexibly.

[2] *External schema* in the context of relational database systems means the schema of a view defined over a database. Here, it is the schema of the original XML instance before splitting it up.

For an example for document splitting, have a look at Figure 3.1, where the countries Belgium and Germany are depicted, each referencing their cities via XLink from remote locations. For providing flexibility in fine-tuning the logical model of the linked data, XLink elements are extended with *modeling directives* for designing an external schema by defining the “cutting edges” of the instance in different ways.

3.3 Handling Simple Links

3.3.1 Modeling Directives: dbxlink:transparent

In [May02], a logical model was proposed that transparently resolves XLinks into one virtual XML instance, defining a semantics for Simple Links. For that purpose, the XLink specification is extended with attributes from the dbxlink namespace, in order to specify the “behavior” of a Simple Link element when it is traversed:

- dbxlink:transparent: mapping of the linked resources to a virtual instance according to the logical model,
- dbxlink:actuate: point in time when the XLinks are evaluated to generate the view (materialization at parse time, or on-demand for answering a query),
- dbxlink:eval: location where the XPointers and query expressions are evaluated (locally at the server hosting the referencing document, or at the remote site, where the referenced document is located),
- dbxlink:cache: caching strategies for views and intermediate results.

The most important dbxlink attribute in terms of the data model and towards fine-grained modeling of linked XML data is the dbxlink:transparent attribute. The attribute’s value contains the modeling directives that determine how exactly the remote resource is mapped into the document context, and what happens to the linking element.

Consider again the “Belgium” example from Figure 3.1, taken from the MONDIAL database, which contains references from the country element of Belgium to the city elements for (a) all cities and (b) Belgium’s capital Brussels, all located at a remote instance cities-B.xml. The first intuition of the above idea is to simply “copy” the target of the XPointer and to “paste” it into the XLink element, replacing the XLink element thereby. But also, other options can be thought of: a referenced resource can be made subelement(s) of the Simple Link element, or could be made a reference attribute, depending on the intended external schema. A possible mapping of the resources shown in Figure 3.1 could e.g. result in a model that allows for the following XPath queries:

- model the capital as an attribute:
  doc("http://.../countries.xml")//country[name="Belgium"]/id(@capital)/population
- model cities as subelements, dropping the "auxiliary" cities element:
  doc("http://www.foo.de/countries.xml")//country[name="Belgium"]/city/name
- model neighbors as subelements that contain the referenced country data and the link's borderlength attribute:
  doc("...")//country[name="Belgium"]/neighbor[name="Germany"]/@borderlength
  (note that the virtual substructure that matches the latter part is obtained from combining the country element with its name subelement "Germany", and the neighbor subelement of Belgium with its attribute borderlength).

In the resulting virtual instance, the Belgium element would look like:

```
<country car_code="B" capital="cty-B-brussels">
  <name>Belgium</name>
  <city id="cty-B-antwerp">
    <name>Antwerp</name>
    ...
  </city>
  <city id="cty-B-brussels">
    <name>Brussels</name>
    <population year="1995">951580</population>
  </city>
  ...
  <neighbor car_code="D" capital="cty-D-berlin" borderlength="167">
    <name>Germany</name>
    <city id="cty-D-aachen">
      <name>Aachen</name>
      ...
    </city>
    ...
  </neighbor>
  ...
</country>
```

The exact node-to-node mapping for Simple Links is based on two modeling directives:

- the *L-directive* (Left-hand or Link directive)
- the *R-directive* (Left-hand or Link directive)

Right-hand / Result Directive: the XPointer expression from the xlink:href attribute is evaluated, yielding a set of XML nodes. The nodeset is mapped to an intermediate result depending on (i) whether complete elements are to be inserted, or (ii) only the referenced element's contents (its attributes, text and child elements) are to be inserted. Since the directive is – by convention – being written on the right hand, and since it takes charge of the XPointer *result set*, it is called *Right-Hand Directive* or *Result Directive*. From now on, the term *R-directive* is used for short.

Left-hand / Link Directive: The intermediate result is mapped again, now into the referencing instance. Here, the question arises how to treat the Simple Link element containing the xlink:href and dbxlink:transparent attributes. Whether it is removed and replaced by the result set, whether it is grouped around the result set, whether it is wrapped around each result set node, or being transformed into a reference attribute of its parent, referencing the XLinked nodeset: all this is determined by the *Link Directive* or *Left-Hand Directive*. From now on, the term *L-directive* is used for short.

3.3.2 L-Directive and R-Directive

(i) Consider a result set containing a number of nodes (and implicitly their child elements, text and attribute nodes). The following R-directive alternatives exist for embedding the result nodes in the virtual instance:

- insert-nodes: Each result node is inserted "as a whole".
- insert-bodies: Insert only the result nodes' *bodies*, namely their element and text children and attributes. For text and attribute nodes, the body is considered empty.
- insert-nothing: Drop the result nodes and – you might have guessed it – insert just nothing.

(ii) The treatment of the XLink element itself also influences the structure of the virtual instance. Here, the following options for the L-directive exist:

- drop-element: Drop the XLink element, replacing it with the result.
- keep-body: Drop the XLink element's hull and use only the information of its body (its non-XLink attributes and content) for enriching the referenced nodes[3].
- group-in-element: Keep the XLink element and "wrap it around" the referenced nodes (only once around all referenced nodes).
- duplicate-element: Duplicate the XLink element and wrap it around *each* referenced node.
- make-attribute: Replace the link element with an IDREF(S) attribute with the same name as the link element, referencing the result set element(s).

To put it all together, mapping a Simple Link element consists of three steps that determine the structure of the virtual instance: (i) mapping the result set,

[3]Note that drop-element is a specialization of keep-body, since drop-element can be replaced with a keep-body with an XLink element with no non-XLink children and attributes.

yielding a set of nodes (“insert-nodes”), or a set of bodies (“insert-bodies”), (ii) mapping the XLink element itself, and (iii) mapping the result to a nodeset which is then added to the parent element as new children and/or attributes. The mapping in the two first steps is determined by the user amongst the above alternatives; the step (iii) is then solely a mapping into well-formed XML.

The dbxlink:transparent attribute for a Simple Link element thus contains two keywords: (i) the R-directive determining the mapping of the result set. *R-dir* ∈ {insert-nodes, insert-bodies, insert-nothing}, and (ii) the L-directive determining the mapping of the link element itself. *L-dir*∈ {duplicate-element, group-in-element, drop-element, keep-body, make-attribute}.

A prototypical Simple Link element looks like this:

```
<linkelement xlink:type="simple" xlink:href="xpointer"
             dbxlink:transparent="L-dir R-dir" non-xlink-attributes>
    non-xlink children
</linkelement>
```

Default values for dbxlink:transparent Values:

Simple Link	*L-dir R-dir*	drop-element insert-nodes

Example 3 *Figure 3.3 shows an excerpt of doc("http://www.foo.de/countries.xml") where the cities XLink element has been extended with a dbxlink:transparent attribute containing L-directive drop-element and R-directive insert-nodes. The virtual instance is obtained by dropping the cities link element and inserting the referenced city nodes instead.*

3.3.3 XLinks and Querying

The semantics for Simple Links as “embedded views”, that include the linked data into the virtual instance, is defined for allowing to query interlinked XML data that is distributed over the Web. XQuery, the standard query language for XML, is based on XPath for identifying nodesets inside the queried instance. Hence, for providing a Simple Link semantics for querying, it is sufficient to describe the Simple Link’s “behavior” with respect to XPath navigation.

In the following 4, the “behavior” of Extended Links with respect to XPath queries is introduced, in order to provide an Extended Link semantics for querying. Then, s 5 and 6 define the Simple Link and Extended Link querying semantics formally by algebraically specifying a logical linking data model, and how it is compliant with XPath queries.

```
<countries>
  <country car_code="B"> <name>Belgium</name>
  ...
   <cities xlink:type="simple"
            xlink:href="http://www.bar.de/cities-B.xml#xpointer(/cities/city)"
            dbxlink:transparent="drop-element insert-nodes" />
  </country>
  <country car_code="D"> ... </country>
</countries>
```

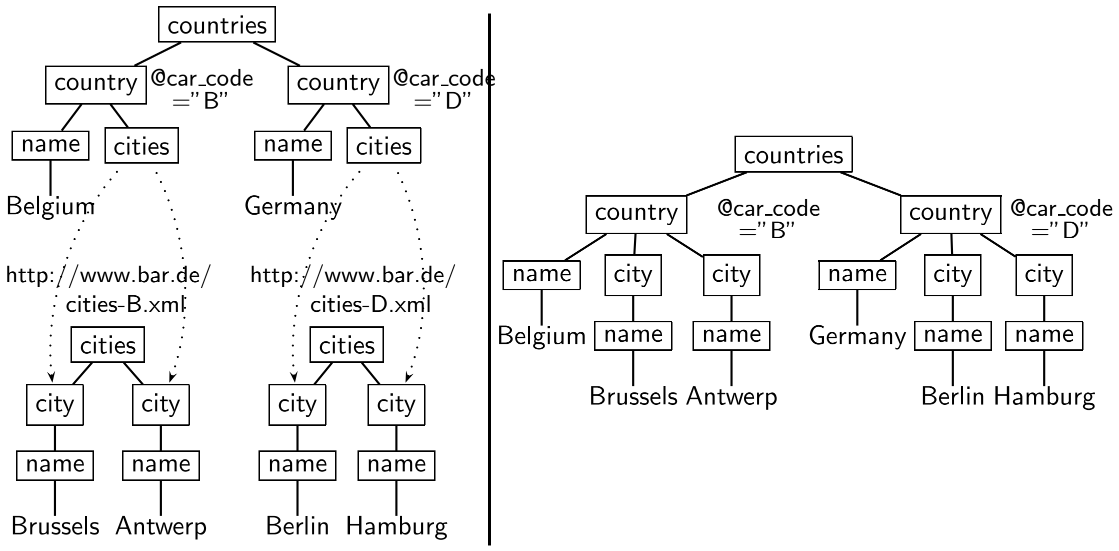


Figure 3.3: Illustration of the mapping – Above: Fragment Extended with dbxlink:transparent. Left: Physical instances with XLink References. Right: Induced Virtual Instance.

Chapter 4

Querying XML Data with Extended Links

Both Simple Links as well as Extended Links define directed point-to-point connections between XML resources. In Figure 4.1, examples for both Simple Links and Extended Links are given, comparing them side by side: where for Simple Links, the connection always starts at the Simple Link element, the situation is different for Extended Links, since they are not defined *inline*, as part of the involved remote resources. Instead, the linking metadata is maintained and kept *out of line* (in a separate linkbase document), which is the reason why Extended Links are often referenced as *Out-of-Line Links*[1].

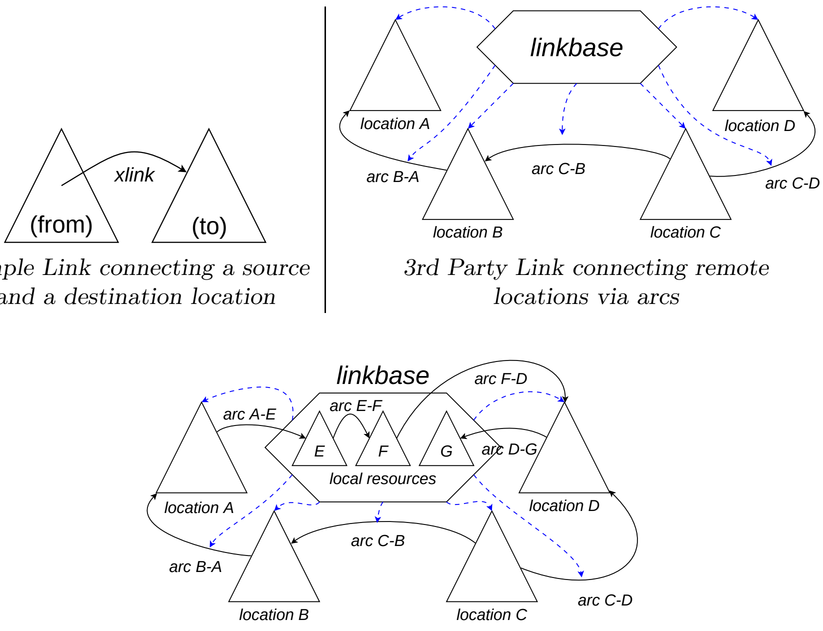
The out-of-line definition has further implications: the three involved data locations – the *from-location* with the from-resource where the link starts, the *to-location* with the to-resource where the link points to, and the linkbase – can be authored and maintained by three completely autonomous parties. The only requirement is that both resources (maintained by the 1st, the *from* party, and the 2nd, the *to* party) are visible to the linkbase (3rd party). Whenever it is intended to stress this 3rd party aspect of Extended Links within this work, the term *3rd Party Links* is used.

4.1 A Matter of Perspective: forward, inverse and relation Perspectives

As pointed out in Section 2.3, an Extended Link consists of:

- zero or more local XML resources, defined inside the link element,

[1]In earlier XLink language specifications there existed the definition of "Inline Extended Links". Inline Extended Links were formally Extended Link elements, but – as the name suggests – defined inline, realizing some kind of a multi-target Simple Link. Inline Extended Links were removed from the candidate release of the specification in the year 2000.



Simple Link connecting a source and a destination location

3rd Party Link connecting remote locations via arcs

3rd Party Link connecting remote locations (A-D) and local resources (E-G)

Figure 4.1: Simple Link, 3rd Party Link, 3rd Party Link with local resources

- remote resources, specified with a locator element containing a URI / XPointer reference,
- zero or more directed arcs connecting from-resources with to-resources,
- optionally a title, and
- optionally some none-XLink-related content.

Without loss of generality, let us focus on arcs connecting one single from-resource with one single to-resource[2]. Considering the 3-parties notion for Extended Links that is given above in the text, three perspectives have to be considered:

- the *forward* perspective from the viewpoint of the 1st (the from-resource) party,
- the *inverse* perspective from the viewpoint of the 2nd (the to-resource) party, and
- the *relation* perspective from the viewpoint of the 3rd (linkbase) party.

[2]An set of multiple resources $\{r_1, \ldots, r_n\}$ at one of the ends of an arc can be considered to be equivalent to one resource r' consisting of the union of nodes from $r_1, \ldots, r_n$

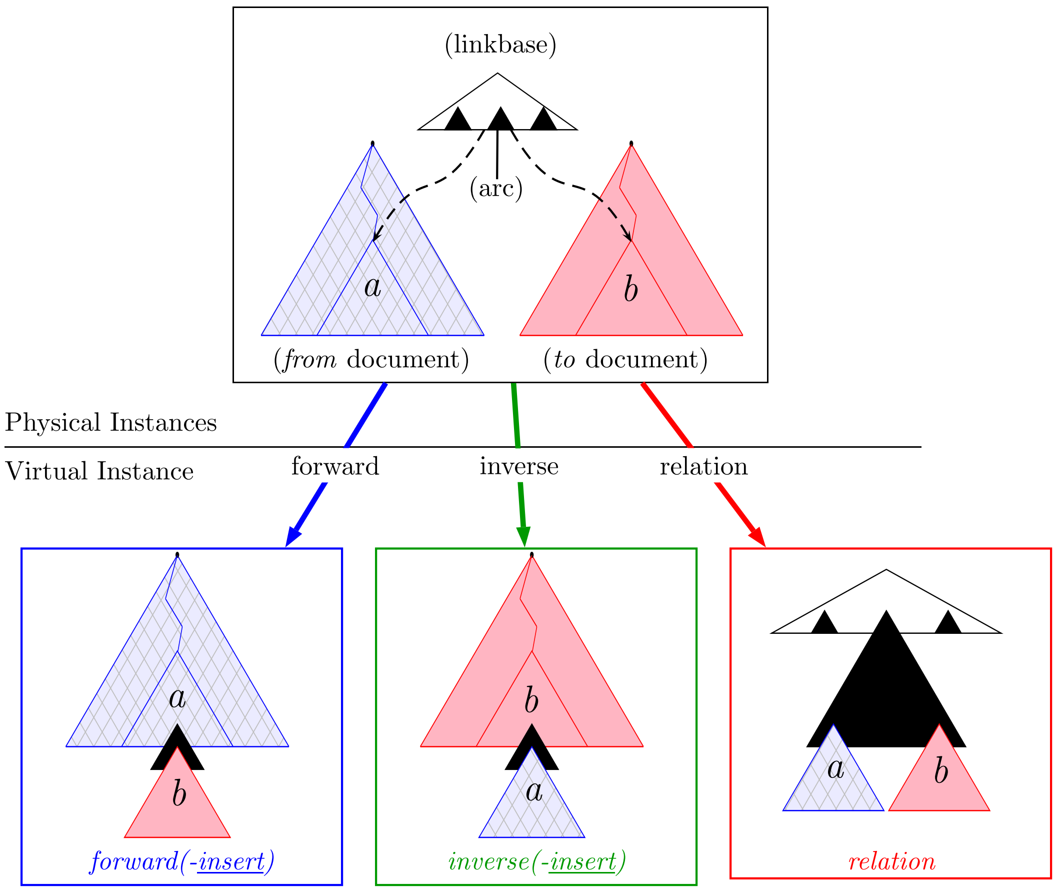
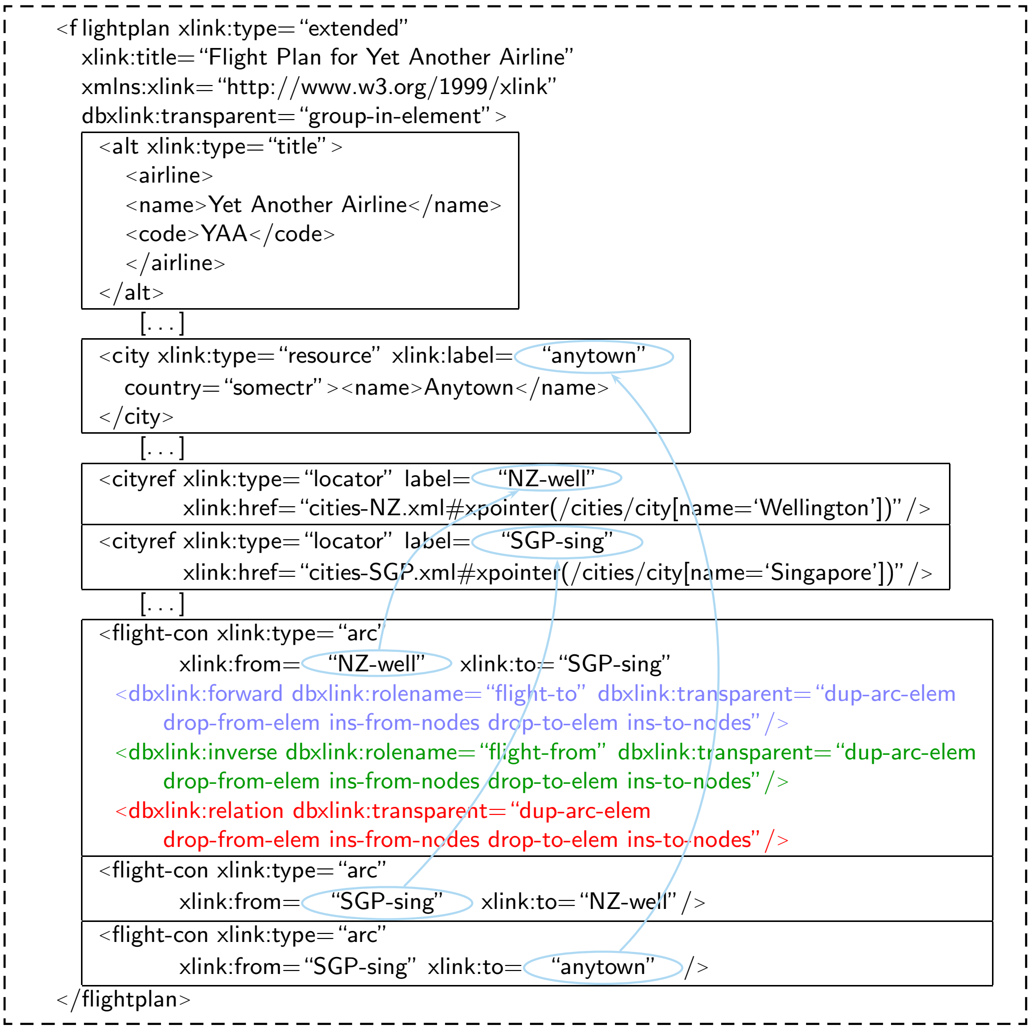


Figure 4.2: forward, inverse, and relation: physical and virtual instances for three perspectives

For the logical data model, this implies that each perspective defines its own virtual instance. Hence, for a single arc and its two referenced resources, three different virtual instances exist, with the composition depending on the supposed perspective. The different perspectives' virtual instances are schematically depicted in Figure 4.2.

Example 4 *Consider again the flightplan example of* "Yet Another Airline"*, given in Figure 4.3. The from-resource is located inside* cities-NZ.xml*, supplying data about New Zealand's cities. The to-resource is located inside* cities-SGP.xml *about Singapore's cities. The linkbase is* linkbase.xml*, supplying data about flight connections between cities. It defines a view on the data for each of the three perspectives (with the resulting virtual instances depicted in Figure 4.4):*

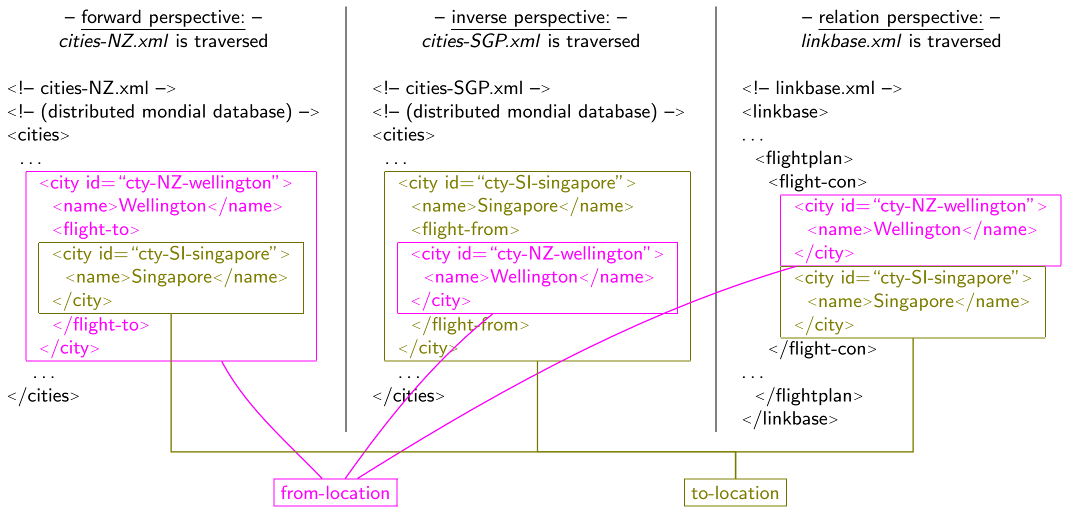


```
<flightplan xlink:type="extended"
   xlink:title="Flight Plan for Yet Another Airline"
   xmlns:xlink="http://www.w3.org/1999/xlink"
   dbxlink:transparent="group-in-element">
   <alt xlink:type="title">
      <airline>
      <name>Yet Another Airline</name>
      <code>YAA</code>
      </airline>
   </alt>
      [. . . ]
   <city xlink:type="resource" xlink:label="anytown"
      country="somectr"><name>Anytown</name>
   </city>
      [. . . ]
   <cityref xlink:type="locator" label="NZ-well"
         xlink:href="cities-NZ.xml#xpointer(/cities/city[name='Wellington'])"/>
   <cityref xlink:type="locator" label="SGP-sing"
         xlink:href="cities-SGP.xml#xpointer(/cities/city[name='Singapore'])"/>
      [. . . ]
   <flight-con xlink:type="arc"
         xlink:from="NZ-well" xlink:to="SGP-sing"
     <dbxlink:forward dbxlink:rolename="flight-to" dbxlink:transparent="dup-arc-elem
         drop-from-elem ins-from-nodes drop-to-elem ins-to-nodes"/>
     <dbxlink:inverse dbxlink:rolename="flight-from" dbxlink:transparent="dup-arc-elem
         drop-from-elem ins-from-nodes drop-to-elem ins-to-nodes"/>
     <dbxlink:relation dbxlink:transparent="dup-arc-elem
         drop-from-elem ins-from-nodes drop-to-elem ins-to-nodes"/>
   <flight-con xlink:type="arc"
         xlink:from="SGP-sing" xlink:to="NZ-well"/>
   <flight-con xlink:type="arc"
         xlink:from="SGP-sing" xlink:to="anytown"/>
</flightplan>
```

Figure 4.3: Flightplan example with dbxlink:transparent attributes

The forward perspective *(left, in Figure 4.2) has an impact on the XML instance that contains the from-resource (here the city element for Wellington in cities-NZ.xml): The to-resource (the Singapore city element inside cities-SGP.xml) is mapped into the virtual instance, defining a modified view over cities-NZ.xml.*

The inverse perspective *(mid, in Figure 4.2) is symmetrically inverse to the forward perspective. Here, the linkbase defines a modified view over the to-resource, blending the Wellington city element into cities-SGP.xml, according to the modeling directives and rolename given for inverse.*



– forward perspective: –
cities-NZ.xml is traversed

```
<!– cities-NZ.xml –>
<!– (distributed mondial database) –>
<cities>
 . . .
  <city id="cty-NZ-wellington">
    <name>Wellington</name>
    <flight-to>
    <city id="cty-SI-singapore">
      <name>Singapore</name>
    </city>
    </flight-to>
  </city>
 . . .
</cities>
```

– inverse perspective: –
cities-SGP.xml is traversed

```
<!– cities-SGP.xml –>
<!– (distributed mondial database) –>
<cities>
 . . .
  <city id="cty-SI-singapore">
    <name>Singapore</name>
    <flight-from>
    <city id="cty-NZ-wellington">
      <name>Wellington</name>
    </city>
    </flight-from>
  </city>
 . . .
</cities>
```

– relation perspective: –
linkbase.xml is traversed

```
<!– linkbase.xml –>
<linkbase>
. . .
  <flightplan>
    <flight-con>
      <city id="cty-NZ-wellington">
        <name>Wellington</name>
      </city>
      <city id="cty-SI-singapore">
        <name>Singapore</name>
      </city>
    </flight-con>
. . .
  </flightplan>
</linkbase>
```

Figure 4.4: Resulting views for forward, inverse and relation perspective

In the relation perspective *(right, in Figure 4.2), the linkbase itself is expanded: arcs and locators are expanded in-place when traversed by a query, according to the given modeling directives, including the results from cities-NZ.xml and cities-SGP.xml into the linkbase document.*

The query
document("cities-NZ.xml")/cities/city[name="Wellington"]/flight-to/city *returns the Singapore* city *element as result, since Singapore is the destination of the flight blended into the virtual instance (and may be more, since there can be more flight connections to other cities starting in Wellington).*
The query
document("cities-SGP.xml")/cities/city[name="Singapore"]/flight-from/city *returns (at least) the Wellington* city *element from cities-NZ.xml.*
A query
document("flightplan.xml")/linkbase/flightplan/flight-con[city[1]/name="Singapore"]/city[2]
returns all cities that can be reached from Singapore by plane.
The query
document("flightplan.xml")/linkbase/flightplan/flight-con[city[2]/name="Singapore"]/city[1]
returns all cities that have outgoing flight connections straight to Singapore.

Throughout all queries, the relevant perspective is determined by the leading document() function's argument, which determines a query's entry point: queries with a leading document("cities-NZ.xml") (1st party) are evaluated wrt. the forward perspective. Queries with a leading document("cities-SGP.xml") (2nd party) are evaluated wrt. the inverse perspective. Queries with a leading document("linkbase.xml") (3rd party) are evaluated wrt. the relation perspective.

4.2 Arc Roles in Different Perspectives

An arc describes a relationship between resources. The "meaning" of a relationship established by an arc depends not only on the connected resources and the information given with the arc itself (e.g. its name), but it also changes with the perspective from where the relationship is seen. E.g., the relationship between a parent and its child can be seen from an "above perspective" as a binary "parent-child" relationship, with one argument being the parent, the other argument being the child. From the perspective of the parent, the same relationship can be seen as a unary predicate "has-child" with the child being the predicate's argument. From the perspective of the child, the relation looks like a unary "has-parent" predicate with the parent being the predicate's single argument. A relationship hence may have different *roles* according to different perspectives. To allow for a semantically meaningful modeling, it is desirable to enable the link designer to assign names to a relationship's roles within the relationship-defining arc.

Regarding again the flight connection scenario from Example 4, the arc has the "flight-to" role for the forward perspective, where the arc points to another city with an outgoing flight connection going there. In the inverse perspective, the arc points from the destination city to the start city of an incoming flight connection, fulfilling a "flight-from" role for the to-resource. The relation perspective assigns the "flight-con(nection)" role to the arc. The role of the arc for each perspective is assigned with the dbxlink:rolename attribute[3]. Within the arc's syntax, the perspective-specific role name is given as the value of dbxlink:rolename for each perspective (see Figure 4.5). If dbxlink:rolename is omitted, the name of the arc element is used as default rolename.

4.3 3 Perspectives – 3 Modeling Directives

Since in the presence of 3rd Party Links, three perspectives exist, and since each perspective needs its own modeling directives for mapping the physical to the virtual instance, an arc needs to be equipped with three sets of directives. For that, each arc element has dbxlink:relation, dbxlink:forward and dbxlink:inverse subelements, each containing the dbxlink:transparent attribute for the according perspective, as well as a dbxlink:rolename. The arc element's dbxlink:transparent attribute can still serve as a default value for the three dbxlink subelements, if any of them is omitted. This leads to an enhanced Extended Link syntax, shown in Figure 4.5, which will be explained in detail throughout the following sections.

[3]not to be confused with the xlink:role or xlink:arcrole attribute, that are defined by the W3C XLink standard, and that have no dbxlink-specific semantics.

```
<ext dbxlink:transparent='ext-L-dir'>
  <arc xlink:type='arc'>
    <dbxlink:relation dbxlink:rolename='rolename'
      dbxlink:transparent='card-dir arc-L-dir from-L-dir from-R-dir to-L-dir to-R-dir'/>
    <dbxlink:forward dbxlink:rolename='rolename'
      dbxlink:transparent='card-dir placement-dir arc-L-dir to-L-dir to-R-dir'/>
    <dbxlink:inverse dbxlink:rolename='rolename'
      dbxlink:transparent='card-dir placement-dir arc-L-dir from-L-dir from-R-dir'/>
    arc content
  </arc>
  <loc xlink:type='locator'
    dbxlink:transparent='loc-L-dir loc-R-dir'>
    locator content
  </loc>
</ext>
```

Figure 4.5: Extended Link Syntax with dbxlink:transparent and dbxlink:rolename directives for each perspective

4.4 Modeling Directives for the relation Perspective

Although the relation perspective is anchored to the linkbase (and thus is assigned to the 3rd party), it is addressed first here, due to its similarity with the already familiar Simple Links modeling. In this manner, the modeling issues concerning Extended Links can be explained on basis of the "least abstract" perspective, so that they are already familiar, and can be referred to during the sections covering the forward and inverse perspectives.

The relation perspective creates a modified view over the linkbase containing the Extended Link elements. Both from-locator and to-locator elements are expanded, the expansion results are concatenated ("from" first, "to" last). The concatenated result is combined with the traversed arc element, and finally, the expanded arcs are combined with the link element itself. In terms of perspective, relation resembles more of a complex kind of Simple Links: when link information is found inline, it is expanded in-place.

4.4.1 The dbxlink:transparent Directives for **relation** perspective

The modeling directives for relation are given in the dbxlink:transparent attributes of the arc element and the locator elements. The following modeling directives, marked up in Figure 4.7, apply:

- *ext-L-dir* determines how the surrounding Extended Link element is processed (duplicated, kept, dropped, etc.). As for a Simple Link's *L-directive*,

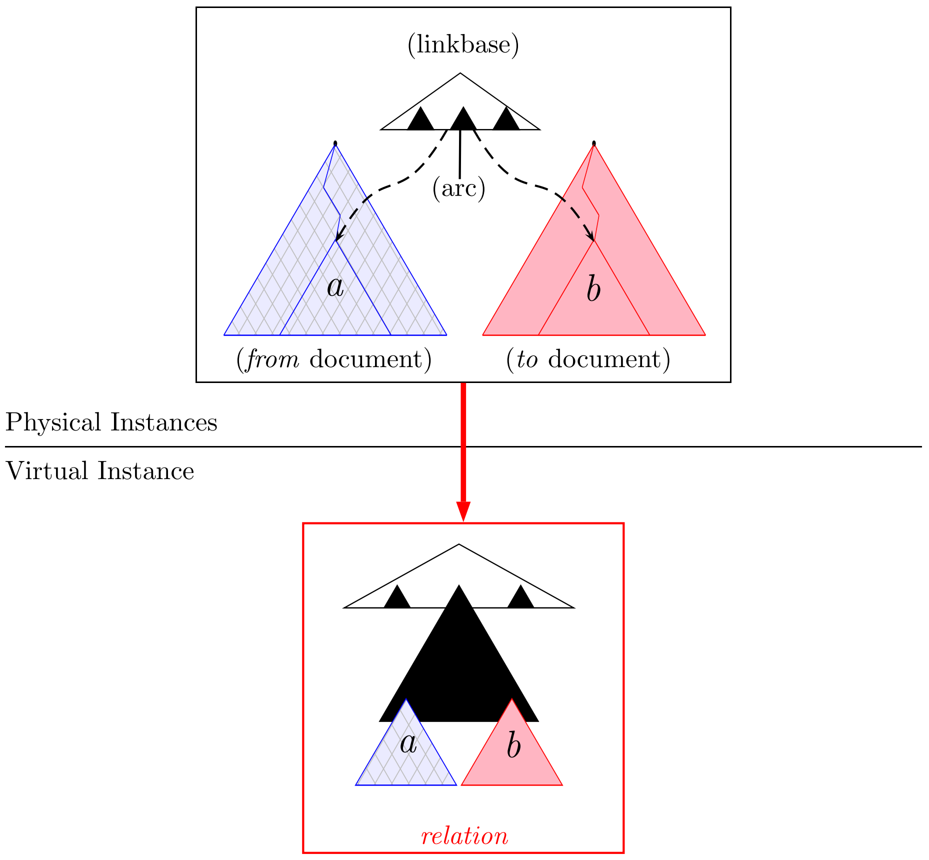


Figure 4.6: The relation Perspective - Physical and Virtual Instance

its possible values are: duplicate-element, group-in-element, drop-element, make-attribute, and keep-body.

For duplicate-element, the Extended Link element is multiple times duplicated and wrapped around every expanded arc and/or locator. The link element's non-XLink nodes are duplicated with their parent, and the arc's/locator's result is inserted in document order right in the former position of the arc/locator.

Note that *ext-L-dir* is only relevant for the relation perspective. It has no meaning for either forward or inverse perspectives.

- *arc-L-dir* determines how the arc element is processed (duplicated, kept, dropped, etc.). Again, its possible values are: dup-arc-elem, group-arc-elem, drop-arc-elem, make-arc-attr, and keep-arc-body.

- *from-L-dir* and *from-R-dir* determine how the nodeset that is identified by the from-locator is processed. As for Simple Links L-directive and R-directive, the possible values are: dup-from-elem, group-from-elem, drop-from-elem, make-from-attr, keep-from-body (*from-L-dir*) and ins-from-elem, ins-from-bodies, ins-from-nothing (*from-R-dir*).

- *to-L-dir* and *to-R-dir* determine how the nodeset that is identified by the to-locator is processed. It is exactly the same as for Simple Links. Hence, the possible values are: dup-to-elem, group-to-elem, drop-to-elem, make-to-attr, keep-to-body (*to-L-dir*) and ins-to-elem, ins-to-bodies, ins-to-nothing (*to-R-dir*).

- *loc-L-dir* and *loc-R-dir* are the left and right hand side directives for each locator. Note that in principle, each locator can be given the same mapping functionality as a Simple Link. When traversed over an arc (which is the regular case), the arc's *from-L/R-dir*/*to-L/R-dir* directives supersede the locator's own directives. But if a locator is traversed *directly*, the locator is expanded by its own L- and R-directive, just like a Simple Link. There is only one difference: to avoid unwanted expansion of locators by accidentally traversing them (e.g. with imprecise XPath queries), a locator's default transparent directive is drop-element insert-nothing. So the locator won't contribute anything (unwanted) to the virtual model, except explicitly stated different by the linkbase designer.

  Since the linkbase is only directly accessed in relation perspective, a locator's dbxlink:transparent directives are only relevant for the relation perspective. They have no meaning for the forward and inverse perspectives, since those contribute to the virtual instance only by their arcs.

```
<ext dbxlink:transparent='ext-L-dir'>
  <arc xlink:type='arc'>
    <dbxlink:relation dbxlink:rolename='rolename'
      dbxlink:transparent='card-dir arc-L-dir from-L-dir from-R-dir to-L-dir to-R-dir'/>
    <dbxlink:forward dbxlink:rolename='rolename'
      dbxlink:transparent='card-dir placement-dir arc-L-dir to-L-dir to-R-dir'/>
    <dbxlink:inverse dbxlink:rolename='rolename'
      dbxlink:transparent='card-dir placement-dir arc-L-dir from-L-dir from-R-dir'/>
    arc content
  </arc>
  <loc xlink:type='locator'
    dbxlink:transparent='loc-L-dir loc-R-dir'>
    locator content
  </loc>
</ext>
```

Figure 4.7: The directives for relation perspective, marked in the link structure

Allowed dbxlink:transparent directives for relation perspective[a]:

arc-L	from-L	from-R	to-L	to-R
*	*	*	*	*

[a]The star (*) stands for wildcard: for an *L-dir*, all five *L-dir* directives are allowed, for an *R-dir*, all three *R-dir* directives are allowed.

Default values for dbxlink:transparent in relation perspective:

Simple Link	*L-dir R-dir*	drop-element insert-nodes
Extended Link element: Extended Link's arc element	*ext-L-dir* *arc-L-dir* *from-L-dir from-R-dir* *to-L-dir to-R-dir*	group-in-element dup-arc-elem drop-from-elem ins-from-nodes drop-to-elem ins-to-nodes
Extended Link's Locator element	*L-dir R-dir*	drop-element insert-nothing

Example 5 *To visualize the effect of the modeling directives from Example 4, look again at the parameters for the relation perspective:* transparent="dup-arc-elem drop-from-elem ins-from-nodes drop-to-elem ins-to-nodes"

- dup-arc-elem*: the arc element* flight-con *is kept, and is wrapped around the locator results*[4]*.*
- drop-from-elem*: the locator element* cityref *is dropped and replaced by the referenced node(s).*
- ins-from-nodes*: the referenced node(s) are completely inserted.*
- drop-to-elem*,* ins-to-nodes*: equivalent.*

Result:

```
<linkbase>
  <flightplan>
    <flight-con>
      <city id="cty-NZ-wellington">
        <name>Wellington</name>
      </city>
      <city id="cty-SI-singapore">
        <name>Singapore</name>
      </city>
    </flight-con>
  </flightplan>
</linkbase>
```

Now consider the same relationship, still in the relation *perspective, but let's assume now different mapping directives, given in dbxlink:transparent:* dbxlink:transparent="dup-arc-elem dup-from-elem ins-from-bodies make-to-attr ins-to-nodes"

[4]Since both locations only yield a single node each, cardinality options have no effect.

- *dup-arc-elem: again, the arc element flight-con is kept, and is wrapped around the locator results.*
- *dup-from-elem: the locator element cityref is kept, eventually duplicated (not here in the example) and inserted.*
- *ins-from-bodies: the referenced nodes' bodies are are extracted and inserted into the surrounding element (here: the locator element cityref.)*
- *make-to-attr ins-to-nodes: the referenced "to" element – here the city element of Singapore – is taken, and an IDREF attribute with the locator's element name is created, referencing the Singapore element. For multiple result elements, an IDREFS attribute is created, containing a set of IDREF tokens, each one identifying one result element.*

Result:

```
<linkbase>
  <flightplan>
    <flight-con city="cty-SI-singapore">
      <cityref id="cty-NZ-wellington">
        <name>Wellington</name>
      </cityref>
    </flight-con>
  </flightplan>
    ...
      <city id="cty-SI-singapore">
        <name>Singapore</name>
      </city>

    ...

</linkbase>
```

4.4.2 Cardinality Directives for relation

Algebraically, an arc defines a relationship between resources. Conceptually, a resource can be seen either as an atomic unit or as a multi-valued set of nodes (a nodelist). Following the latter concept, the arc – as a relationship – has a notion of *cardinality*. Like a relationship in Entity-Relationship Modeling for relational databases, an arc can relate two node sets as $1:1$, $1:n$, $n:1$ or $m:n$ relations.

1:1 means that the cartesian product of both nodesets is formed (the set of all ordered pairs with one item from nodeset A and one item from nodeset B). By convention, the node from the from-nodeset is first in document order, the node from the to-nodeset is second. The modeling directive is card-1-1.

1:n means that a from-node is followed by all to-nodes. This is done sequentially for all from-nodes. The modeling directive is card-1-n.

m:1 similarly means that all from-nodes are followed by one to-node. Accordingly, this is done with all to-nodes. The modeling directive is card-m-1.

m:n simply concatenates the from-locator's and to-locator's results. The modeling directive is card-m-n.

See Figure 4.8 for illustration of the cases above. The cardinality directive is, just like all other modeling directives, denoted in the dbxlink:transparent attribute of the according dbxlink:relation child element of the concerned arc. For later sections, please keep in mind that the cardinality aspect is as well of relevance for the forward and inverse perspectives.

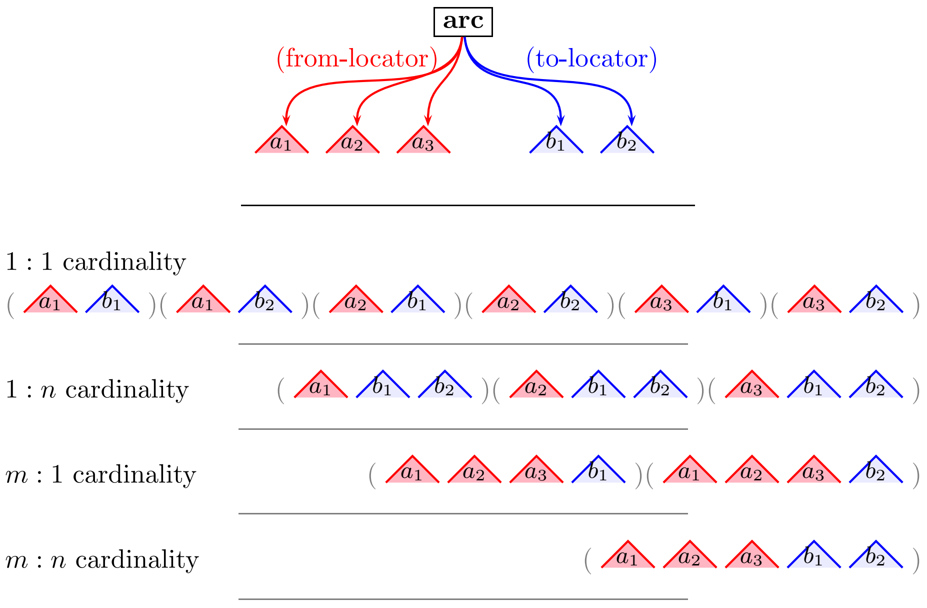

arc
(from-locator)
(to-locator)
a_1 a_2 a_3 b_1 b_2
1 : 1 cardinality
(a_1 b_1)(a_1 b_2)(a_2 b_1)(a_2 b_2)(a_3 b_1)(a_3 b_2)
1 : n cardinality
(a_1 b_1 b_2)(a_2 b_1 b_2)(a_3 b_1 b_2)
m : 1 cardinality
(a_1 a_2 a_3 b_1)(a_1 a_2 a_3 b_2)
m : n cardinality
(a_1 a_2 a_3 b_1 b_2)

Figure 4.8: Multi-valued locators mapped with different cardinalities

4.5 Modeling Directives for the forward and inverse Perspectives

The forward perspective is anchored to the *from*-document, which means that a 3rd Party Link has an impact to a traversed *from*-document. The *to*-resource is combined with the arc information, and this intermediate result is blended into the from-document. In detail: the to-locator and the to-resource at the

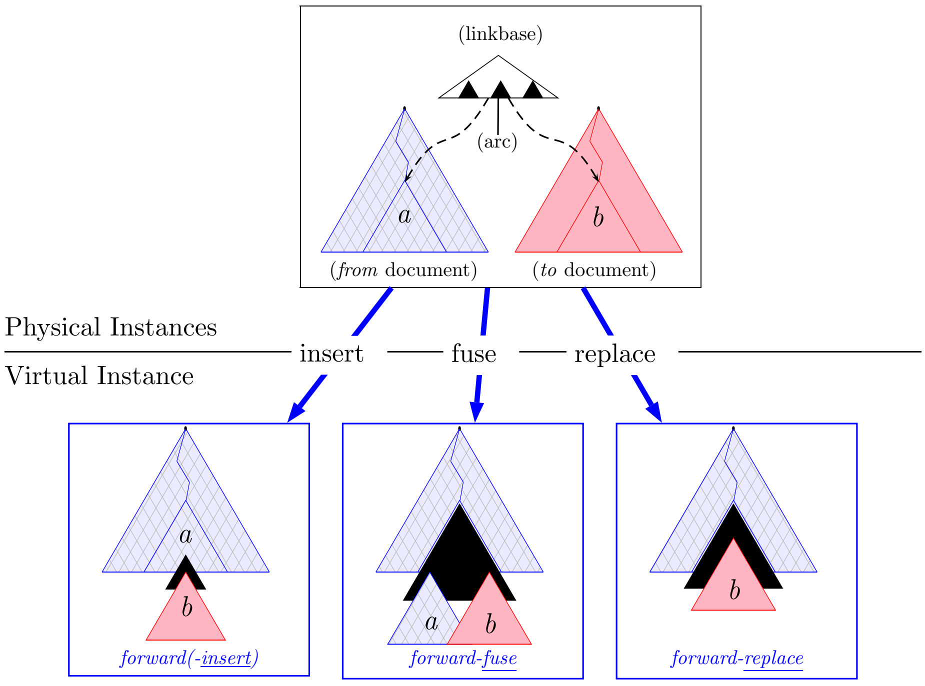


Figure 4.9: The three Placing Modes for forward perspective: fuse, replace and insert(default)

remote end point of the arc are processed with *to-L-dir* and *to-R-dir* before the result is blended into the from-resource. Hence, the transparent attribute of dbxlink:forward has the directives *arc-L-dir*, *to-L-dir* and *to-R-dir* (and also placement and cardinality directives, see Sections 4.5.1 and 4.5.2), but no *from-L-dir* or *from-R-dir*.

The from-locator serves only for defining the from-resource where the arc starts, but does *not* influence the modeling of the virtual instance. Hence, the from-directives are omitted.

The inverse perspective is defined vice versa: here, the from-resource is blended into the to-document. Hence, *from-L-dir* and *from-R-dir*, as well as *arc-L-dir* (and, same as within forward, placement and cardinality directives) define the result. *to-L-dir* and *to-R-dir* are omitted.

4.5.1

Placement Directives for the forward and inverse Perspectives For forward perspective (and inverse as well), an additional issue comes into focus: after the to-resource is processed together with its locator and arc information (yielding an intermediate result), where is it placed within the from-resource? For this

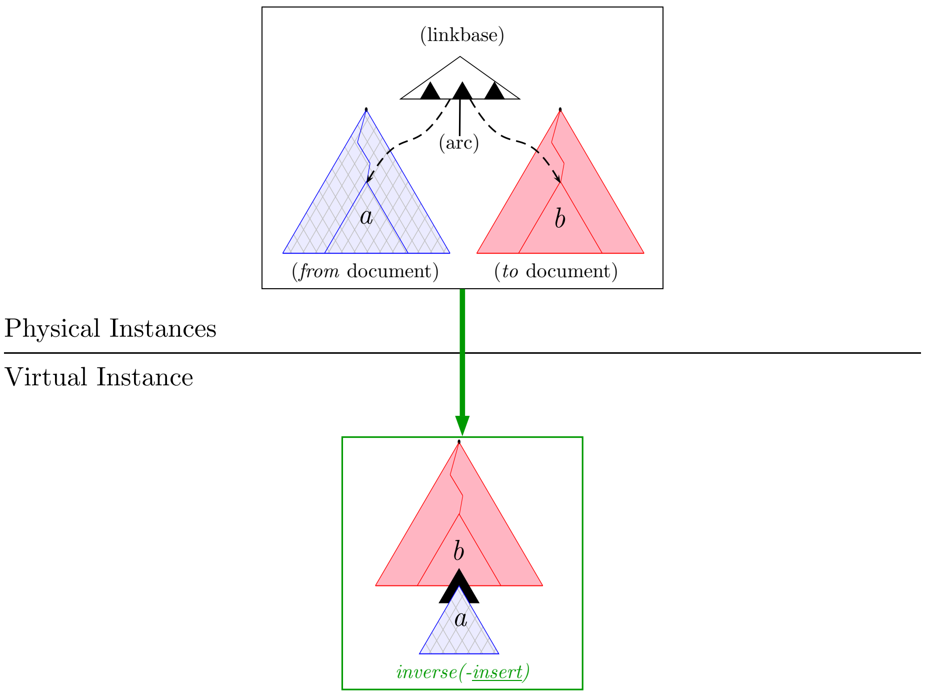


Figure 4.10: The inverse perspective - physical and virtual data model

notion, called *placement*, three customization options (*placement* options) exist for how to combine from-location, to-location and arc element:

- insert placement: after combining the to-locator, the to-locator's results and the arc element with each other, the thereby created intermediate result is inserted as child/attribute nodes of each node that is identified by the from-locator (see Figure 3.2).

- replace placement: each node located by the from-element is replaced with the intermediate result from arc, to-locator and to-locator's result. See Figure 4.9.

- fuse placement: concatenating and inserting the bodies of the intermediate results from both from-locator and to-locator into the arc, fusing them together. See Figure 4.9 also.

For the inverse perspective, the same options apply with swapped *from* and *to*-resources.

For fuse placement, the *from-L/R* and *to-L/R* directives are restricted to "drop-from-elem ins-from-bodies" / "drop-to-elem ins-to-bodies", since fuse determines two nodes to be fused in a certain way that also determines the proper

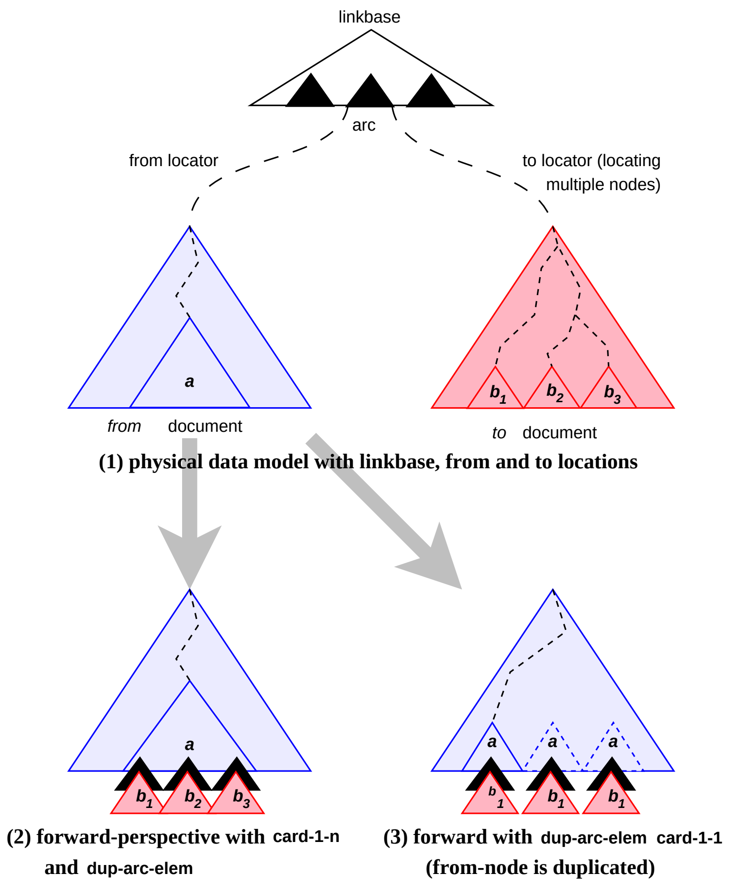

linkbase
arc
from locator
to locator (locating multiple nodes)
a
b_1
b_2
b_3
from document
to document
(1) physical data model with linkbase, from and to locations
(2) forward-perspective with card-1-n and dup-arc-elem
(3) forward with dup-arc-elem card-1-1 (from-node is duplicated)

Figure 4.11: Example for $1 : n$ and $1 : 1$ cardinalities in forward perspective

handling of element hull and element bodies: both nodes' bodies of each pair of nodes are concatenated, and the arc element is wrapped around each pair in a dup-arc-elem manner.

These placement options only apply for the forward and inverse perspectives. The reason is obvious: both perspectives are *asymmetric* in some way. Nodes from a remote resource have to be blended into nodes from a local resource, and the placement determines the mode of this intertwinement. For the relation perspective, both locations are remote, since the local part is the linkbase itself. So, both remote locations can be treated equally, since they only have to be intertwined with the linkbase's arc, but not with each other.

```
<ext dbxlink:transparent='ext-L-dir'>
  <arc xlink:type='arc'>
    <dbxlink:relation dbxlink:rolename='rolename'
      dbxlink:transparent='card-dir arc-L-dir from-L-dir from-R-dir to-L-dir to-R-dir'/>
    <dbxlink:forward dbxlink:rolename='rolename'
      dbxlink:transparent='card-dir placement-dir arc-L-dir to-L-dir to-R-dir'/>
    <dbxlink:inverse dbxlink:rolename='rolename'
      dbxlink:transparent='card-dir placement-dir arc-L-dir from-L-dir from-R-dir'/>
    arc content
  </arc>
  <loc xlink:type='locator'
    dbxlink:transparent='loc-L-dir loc-R-dir'>
    locator content
  </loc>
</ext>
```

Figure 4.12: The directives for forward perspective, marked up in the link structure

4.5.2

Cardinality Directives for the forward and inverse Perspective

The cardinality is also of relevance for the forward and inverse perspectives. In the forward perspective, the to-result (with the already processed arc) can be inserted into the from-node as a complete fragment list ($1 : n$ relationship), or the from-node can be duplicated in-place around each result fragment (thereby creating a $1 : 1$ relationship). Accordingly, in the inverse, the $1 : 1$ and $m : 1$ cardinalities are available. Other alternatives (as $m : n$) are not available, because in both forward and inverse perspectives, one of the two related sets is already “glued” to the surrounding instance – the one that is navigated – and therefore cannot be modified (except being duplicated in-place). See Figure 4.11 for an example in forward perspective. Figure 4.13 depicts a single-steps-example also in forward perspective, with multiple to-nodes, a duplicated to-locator and a duplicated arc element, with cardinality $1 : 1$.

4.5.3

Allowed and Default dbxlink:transparent Values for forward and inverse

The allowed dbxlink:transparent values for the forward and inverse perspectives, depending on the given placement, are given here in a table for overview:

Allowed dbxlink:transparent values for forward perspective:

	arc-L	from-L	from-R	to-L	to-R	placement
insert	*	drop-element	insert-nodes	*	*	{1:1,1:n}
replace	*	drop-element	insert-nodes	*	*	{1:1,1:n}
fuse	dup-arc-elem	drop-element	insert-bodies	drop-element	insert-bodies	{1:1,1:n}

Allowed dbxlink:transparent values for inverse perspective:

	arc-L	from-L	from-R	to-L	to-R	placement
insert	*	*	*	drop-from-elem	ins-from-nodes	{1:1,1:n}
replace	*	*	*	drop-to-elem	ins-to-nodes	{1:1,1:n}
fuse	dup-arc-elem	drop-from-elem	insert-bodies	drop-to-elem	insert-bodies	{1:1,1:n}

Default dbxlink:transparent values for forward and inverse perspectives:

	arc-L	from-L	from-R	to-L	to-R	placement
insert	dup-arc-elem	drop-from-elem	ins-from-nodes	drop-to-elem	ins-to-nodes	1:n
replace	dup-arc-elem	drop-element	insert-bodies	drop-element	insert-bodies	1:n

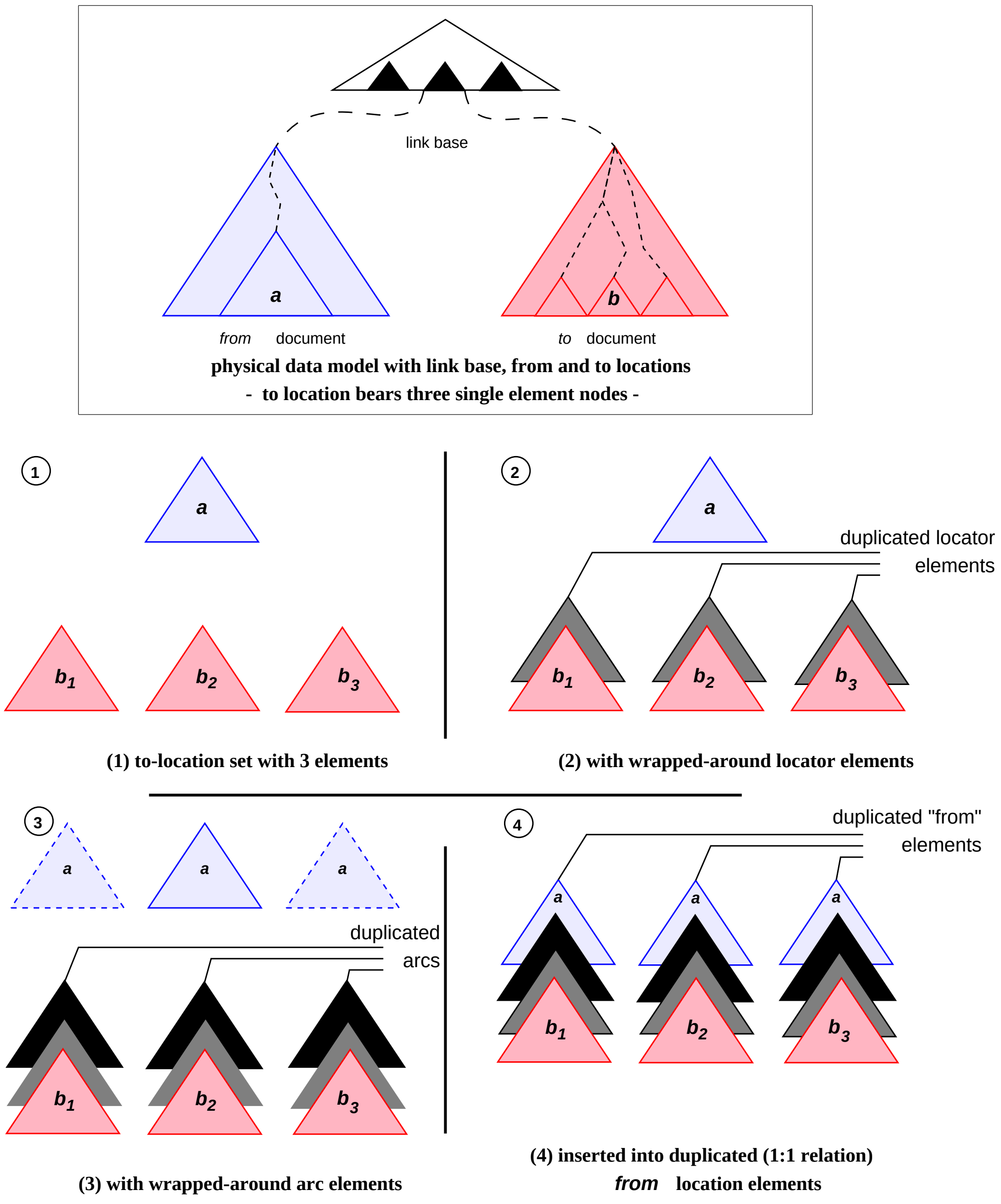

link base
a
b
from document
to document
physical data model with link base, from and to locations
- to location bears three single element nodes -
1
a
b_1
b_2
b_3
(1) to-location set with 3 elements
2
a
duplicated locator elements
b_1
b_2
b_3
(2) with wrapped-around locator elements
3
a
a
a
duplicated arcs
b_1
b_2
b_3
(3) with wrapped-around arc elements
4
duplicated "from" elements
a
a
a
b_1
b_2
b_3
(4) inserted into duplicated (1:1 relation) from location elements

Figure 4.13: Stepwise mapping from physical to virtual data model with "dup-arc-elem dup-to-elem card-1-1" in forward perspective

Chapter 5

The Logical Data Model for Simple Links

In s 3 and 4, a proposal for a data model for querying distributed XML instances is sketched and motivated on base of examples. In this chapter, this data model is formally specified for Simple Links by describing an XML-to-XML mapping from a number of interlinked physical XML instances to a virtual instance. The virtual instance is defined as the result of mapping the inter-linked physical XML instances to one single instance, thereby replacing the links.

The mapping itself is defined in terms of an abstract data type, which uses a reduced XML data model that is based on the XML information set. The XML infoset describes the essential data model concepts that are common to all XML-related data model standards. Since the mapping given below is based on (a reduced) XML infoset, it is compliant with all common data / document / information models for XML as DOM, or the XPath/XQuery data model.

5.1 The Data Model as an Abstract Data Type

Expanding a Simple XLink element is done in three steps:

1. processing the XPointer's result,
2. embedding the result into the XLink element, and
3. embedding this intermediate result set into the document context of the originating XLink element.

These three steps are executed each time a Simple Link element has to be expanded, e.g. during the evaluation of an XPath query against XLink-interlinked documents.

For specifying the behavior of the Simple Link expansion, a custom XML data model is used, based on the notions of *element*, *attribute* and *text nodes*.

Other nodes, namely namespace, processing instruction or comment nodes are omitted, as they are of no interest for link expansion as it is described here. During processing, we also need the terms of *nodelists* and *nodelist lists*. Nodelists are lists (ordered tuples) of nodes, nodelist lists are lists of nodelists.

In the presented model, there is no formal separation between attributes (unordered) and element and text nodes (ordered), since all nodes – including attributes – are considered to be ordered. This makes the definition of the semantics much simpler and easier readable, since all nodes associated to an element – attributes as well as text children or subelements – can be treated in a syntactically uniform way by the mapping functions ϕ and γ.

The data model is described as an *abstract data type*, using *constructors* for basic terms like elements, attributes, text nodes, nodelists etc., and *axioms* for describing the behavior of the data type. The axioms can be distinguished into *accessors* for accessing components of composed data structures (e.g. $\mathsf{Name}(node)$ accesses a node's name), *modificators* and *combinators*, such as $\mathsf{InsertAttr}(elem, attr)$, which adds an attribute to an element, and *transformations*: The mapping function γ and the traversal function ϕ map a set physical XML instances containing XLinks to its link-induced virtual instance by transparently following and expanding each XLink[1]. γ and ϕ are specified as operations on the abstract data type. Hence, they're are *transformations* according to the ADT terminology, transforming an XML structure with links into an XML structure without links.

In the following, the abstract data type is specified, starting with the axioms' signatures together with a textual description, with the axioms themselves given in the following section.

5.1.1 Signatures

Here, the signature and a short explanation is given for each data type axiom, starting with the basic types, and continuing with constructors, accessors and modifiers / combinators (except for lists which are presented in a coherent block, since they are type-generic). The axioms itself are defined afterwards in Section 5.1.2.

basic types:

STRING	is the set of all unicode strings[2]
QNAME	is the set of all qualified XML names (QNAME $\subseteq$ STRING)
ELEM	is the set of all elements
ATTR	is the set of all attributes

[1]This applies to Simple Link as well as 3rd Party Link elements. The latter are described in the following chapter, "on top" of the Simple Link definitions.

[2]where STRING represents all type of XML string values, like CDATA, PCDATA, NMTOKEN, NMTOKENS etc.

TEXT	the set of all text nodes
NODE = ELEM ∪ ATTR ∪ TEXT	the set of all nodes
NODELIST = LIST<NODE>	the set of all lists(ordered tuples) of nodes.
NODELISTLIST = LIST<NODELIST>	the set of all lists of nodelists.

constructors:

Elem : QNAME × NODELIST → ELEM	constructs a new element with given name ∈ QNAME, attrs + children (both intermixed in the nodelist).
Attr : QNAME × STRING → ATTR	constructs a new attribute node with a name and a value.
Text : STRING → TEXT	constructs a new text node. Text("*abc*") is abbreviated "*abc*".
Null : ∅ → NODE	constructs a null value (needed as return value for certain operations)

accessors:

Name : NODE → QNAME	returns the name of an element or attribute node
Attrs : NODE ∪ NODELIST → NODELIST	returns the list of attributes of an element, or the concatenation of all lists of attributes of elements in a nodelist.
Children : NODE ∪ NODELIST → NODELIST	returns the list of children of an element, or the concatenation of all lists of children of elements in a nodelist.
Body : NODE → NODELIST	returns the element body of an element node as a list of element, text and attribute nodes. If the argument is a non-element node, the empty list is returned.
StripXLinkAttrs : ELEM → ELEM	removes all XLink attributes of an XLink element.
StripXLinkChildren : ELEM → ELEM	removes all XLink element children of the given element.
StripXLinkNodes : ELEM → ELEM	removes all XLink nodes of the given element.
Value : NODE ∪ NODELIST → STRING	returns the text value of an element, the attribute value of an attribute, the text value of a text node, and the concatenation of values of a nodelist. (similar to XPath's string() function)

AttrsValue : NODELIST $\rightarrow$ STRING	returns the concatenation of values of the attributes in a nodelist, separated with blanks.
NodesByName : NODELIST $\times$ QNAME $\rightarrow$ NODELIST	returns a list of named nodes from a nodelist by their names.
AttrsByName : NODELIST $\times$ QNAME $\rightarrow$ NODELIST	returns a list of attributes from a nodelist by their names.

modifiers / combinators:

Flat : NODELISTLIST $\rightarrow$ NODELIST	Turns a NODELISTLIST into a NODELIST by concatenating the nodelists.
Unflat : NODELIST$\rightarrow$ NODELISTLIST	Turns a nodelist into a list of nodelists, each containing one single node.
CombineBodies: NODE$\times$NODELIST$\rightarrow$NODELIST	Combines two element bodies (resp. nodelists).
AddBody: ELEM$\times$NODELIST$\rightarrow$ELEM	Adds the given element body to the other element's body.
InsertAttr: ATTR$\times$NODELIST$\rightarrow$NODELIST	Inserts an attribute into a nodelist.
InsertAttrs: LIST<ATTR>$\times$NODELIST$\rightarrow$NODELIST	Inserts a list of attributes into a nodelist.
EvalXPointer: STRING$\rightarrow$NODELIST	returns a nodeset representing the result of an XPointer evaluation against a virtual instance[3].

lists:

List of type T: $\emptyset \rightarrow$ LIST<T>	*create* $\rightarrow []$
List containing n items:	$[a_1, \ldots, a_n]$
Concatenation: $[a, b, c] \circ [b, c, d] \equiv [a, b, c, b, c, d]$	Lists are concatenated with $\circ$
$[]$	the empty list

[3]Note the "virtual" part. This means that an XPointer expression is not evaluated within a physical instance, but may reference nodes in document parts that are "not really there", but that are already the product of another XLink expansion! Hence, XLink involves not only the two layers physical/virtual, but some superior "levels of virtuality". See Section 6.3 for more details.

5.1.2 Data Model: Axioms

Here, the axioms, that have been introduced above, are specified as operators of the abstract data type. Most operators are more or less self-describing, and the definition of their behavior is rather straightforward. Some terminology and a-priori abbreviations are given first:

XLink attributes are attribute nodes belonging to the namespaces

- xmlns:xlink="http://www.w3.org/1999/xlink" or
- xmlns:dbxlink="http://www.dbis.informatik.uni-goettingen.de/linxis".

XLink Elements are XML elements containing one or more XLink Attributes.

XLink nodes are XLink Elements or XLink Attributes.

URIs [Uri98] reference a document or a document fragment. A URI has the form *doc-ref#fragment-identifier*, with *doc-ref* identifying a specific document, and *fragment-identifier* being an XPointer expression specifying a fragment of the XML document.

XPointers [XPt02a] specify a *nodelist* inside of a specific document by a URI reference. See Section 2.3.1.

- $\mathsf{Name} : \mathsf{NODE} \to \mathsf{QNAME}$ gives the name of an element or attribute. Name of a text node is the empty string.
  - $\mathsf{Name}(\mathsf{Elem}(elem\_name, body)) = elem\_name$
  - $\mathsf{Name}(\mathsf{Attr}(attr\_name, attr\_val)) = attr\_name$
  - $\mathsf{Name}(\mathsf{Text}(text\_val)) = \mathsf{Null}$.
- $\mathsf{Attrs} : \mathsf{NODE} \cup \mathsf{NODELIST} \to \mathsf{NODELIST}$ returns the list of attributes of an element node, or the concatenation of the lists of attributes of the nodes in a nodelist.
  $\mathsf{Attrs}(arg) = ?$
  - $arg \in \mathsf{TEXT} : \mathsf{Attrs}(arg) = []$.
  - $arg \in \mathsf{ATTR} : \mathsf{Attrs}(arg) = []$.
  - $arg \in \mathsf{ELEM} : \mathsf{Attrs}(arg) = \mathsf{Attrs}^*(\mathsf{Body}(arg))$.
  - $arg \in \mathsf{NODELIST}, arg = [n_1, \ldots, n_k]$:
    $\mathsf{Attrs}([n_1, \ldots, n_k]) = \mathsf{Attrs}(n_1) \circ \ldots \circ \mathsf{Attrs}(n_k)$.

  Auxiliary function Attrs^*:
  $\mathsf{Attrs}^* : \mathsf{NODELIST} \to \mathsf{NODELIST}$.
  $\mathsf{Attrs}^*([n_1, \ldots, n_k]) =?$
  - $n_1 \in \mathsf{ATTR} : \mathsf{Attrs}^*([n_1, \ldots, n_k]) = n_1 \circ \mathsf{Attrs}^*([n_2, \ldots, n_k])$
  - $n_1 \in \mathsf{ELEM} \cup \mathsf{TEXT} : \mathsf{Attrs}^*([n_1, \ldots, n_k]) = \mathsf{Attrs}^*([n_2, \ldots, n_k])$

$\mathsf{Attrs}^*([]) = []$.

- $\mathsf{Children} : \mathsf{NODE} \cup \mathsf{NODELIST} \rightarrow \mathsf{NODELIST}$ returns the list of element and text children of an element node, or the concatenation of the lists of children of the nodes in a nodelist.
  $\mathsf{Children}(arg) = ?$
  - $arg \in \mathsf{TEXT} : \mathsf{Children}(arg) = []$.
  - $arg \in \mathsf{ATTR} : \mathsf{Children}(arg) = []$.
  - $arg \in \mathsf{ELEM} : \mathsf{Children}(arg) = \mathsf{Children}^*(\mathsf{Body}(arg))$.
  - $arg \in \mathsf{NODELIST}, arg = [n_1, \ldots, n_k] :$
    $\mathsf{Children}([n_1, \ldots, n_k]) = \mathsf{Children}(n_1) \circ \ldots \circ \mathsf{Children}(n_k)$.

  Auxiliary function $\mathsf{Children}^*$:
  $\mathsf{Children}^* : \mathsf{NODELIST} \rightarrow \mathsf{NODELIST}$.
  $\mathsf{Children}^*([n_1, \ldots, n_k]) = ?$
  - $n_1 \in \mathsf{ELEM} \cup \mathsf{TEXT} : \mathsf{Children}^*([n_1, \ldots, n_k]) = n_1 \circ \mathsf{Children}^*([n_2, \ldots, n_k])$
  - $n_1 \in \mathsf{ATTR} : \mathsf{Children}^*([n_1, \ldots, n_k]) = \mathsf{Children}^*([n_2, \ldots, n_k])$

  $\mathsf{Children}^*([]) = []$.

- $\mathsf{Body} : \mathsf{NODE} \rightarrow \mathsf{NODELIST}$:
  - An element's body is the nodelist of its element, text and attribute nodes:
    $\mathsf{Body}(\mathsf{Elem}(name, nodelist)) = nodelist$
  - Text and attribute nodes have no body:
    $\mathsf{Body}(text\_or\_attr) = []$

- $\mathsf{Value} : \mathsf{NODE} \cup \mathsf{NODELIST} \rightarrow \mathsf{STRING}$:
  - value of a text node is its text value (as a string):
    $\mathsf{Value}(\mathsf{Text}(stringval)) = stringval$
  - value of an attribute node is the attribute value:
    $\mathsf{Value}(\mathsf{Attr}(attr\_name, attr\_value)) = attr\_value$
  - value of an element node is the concatenation of values of its element and text children:
    $\mathsf{Value}(element) = \mathsf{Value}(\mathsf{Children}(element))$
  - value of a nodelist is the concatenation of the values:
    $\mathsf{Value}([n_1, \ldots, n_k]) = \mathsf{Value}(n_1) \circ \mathsf{Value}([n_2, \ldots, n_k])$
    $\mathsf{Value}([n]) = \mathsf{Value}(n)$
    $\mathsf{Value}([]) = ""$ (empty string)

- $\mathsf{AttrsValue} : \mathsf{NODELIST} \rightarrow \mathsf{STRING}$:

- $\mathsf{AttrsValue}([n_1, \ldots, n_k]) =$
  - if n_1 is an attribute:
    $\mathsf{AttrsValue}([n_1, \ldots, n_k]) = \mathsf{Value}(n_1) \circ " \; " \circ \mathsf{AttrsValue}([n_2, \ldots, n_k])$
  - if n_1 is an element or text node:
    $\mathsf{AttrsValue}([n_1, \ldots, n_k]) = \mathsf{AttrsValue}([n_2, \ldots, n_k])$
- $\mathsf{AttrsValue}([n_1]) = \mathsf{Value}(n_1)$
- $\mathsf{AttrsValue}([]) = ""$

- $\mathsf{NodesByName} : \mathsf{NODELIST} \times \mathsf{QNAME} \rightarrow \mathsf{NODE}$ selects a named node by its name from a nodelist.
  $\mathsf{NodesByName}([n_1, \ldots, n_k], qname) =?$
  - $\mathsf{Name}(n_1) = qname$:
    $\mathsf{NodesByName}([n_1, \ldots, n_k], qname) = n_1 \circ \mathsf{NodesByName}([n_2, \ldots, n_k], qname)$
  - $\mathsf{Name}(n_1) \neq qname$:
    $\mathsf{NodesByName}([n_1, \ldots, n_k], qname) = \mathsf{NodesByName}([n_2, \ldots, n_k], qname)$
  - $\mathsf{NodesByName}([], qname) = []$.

- $\mathsf{AttrsByName} : \mathsf{NODELIST} \times \mathsf{QNAME} \rightarrow \mathsf{NODELIST}$:
  AttrsByName selects an attribute by its name from a nodelist.
  $\mathsf{AttrsByName}([n_1, \ldots, n_k], qname) = \mathsf{AttrsByName'}(\mathsf{Attrs}^*([n_1, \ldots, n_k]), qname)$

  Auxiliary operator $\mathsf{AttrsByName'}{:}\mathsf{LIST{<}ATTR{>}} \times \mathsf{QNAME} \rightarrow \mathsf{NODELIST}$:
  $\mathsf{AttrsByName'}([a_1, \ldots, a_n], qname) =?$
  - $\mathsf{Name}(a_1) = qname$:
    $\mathsf{AttrsByName'}([a_1, \ldots, a_n], qname) = a_1 \circ \mathsf{AttrsByName'}([a_2, \ldots, a_n], qname)$
  - $\mathsf{Name}(a_1) \neq qname$:
    $\mathsf{AttrsByName'}([a_1, \ldots, a_n], qname) = \mathsf{AttrsByName'}([a_2, \ldots, a_n], qname)$
  - $\mathsf{AttrsByName'}([], qname) = []$.

- InsertAttr: $\mathsf{NODELIST} \times \mathsf{ATTR} \rightarrow \mathsf{NODELIST}$ inserts an attribute into a nodelist. If there is already an attribute of the same name, the old attribute's value and the new attribute's value are concatenated.
  $\mathsf{InsertAttr}(nodelist, attr) = ?$:
  - if $\mathsf{AttrsByName}(nodelist), \mathsf{Name}(attr)) = []$:
    $\mathsf{InsertAttr}(nodelist, attr) = nodelist \circ attr$
  - $\mathsf{AttrByName}(nodelist, \mathsf{Name}(attr)) = [a_1, \ldots, a_k]$ with j being a_1's position in $nodelist$. Then let $n'_j = \mathsf{Attr}(\mathsf{Name}(attr), \mathsf{AttrsValue}([a_1, attr]))$. Then,
    $\mathsf{InsertAttr}([n_1, \ldots, n_j, \ldots, n_k], attr) = [n_1, \ldots, n'_j, \ldots, n_k]$.[4]

[4] If a nodelist already contains multiple attributes with the same name, then it is already to some degree inconsistent, since it can't have been created using InsertAttrs. For in that sense consistent nodelists, it is sufficient to fuse the to-be-inserted attribute with the first attribute of the same name found in the list.

- $\mathsf{InsertAttrs}$: $\mathsf{NODELIST}\times\mathsf{LIST<ATTR>}\rightarrow\mathsf{NODELIST}$ inserts a list of attributes into a nodelist.
  - $\mathsf{InsertAttrs}([n_1,\ldots,n_k],[a_1,\ldots,a_n]) =$ $\mathsf{InsertAttrs}(\mathsf{InsertAttr}([n_1,\ldots,n_k],a_1),[a_2,\ldots,a_n])$.
  - $\mathsf{InsertAttrs}([n_1,\ldots,n_k],[]) = [n_1,\ldots,n_k]$.

- $\mathsf{CombineBodies}$: $\mathsf{NODELIST}\times\mathsf{NODELIST}\rightarrow\mathsf{NODELIST}$ combines two bodies (resp. nodelists) by concatenating element and text nodes and by combining the attributes.
  $\mathsf{CombineBodies}(body_1, body_2) =?$
  - $\mathsf{CombineBodies}(body_1,[]) = body_1$.
  - Let $body_1 = [b_1,\ldots,b_n]$, and $body_2 = [n_1,\ldots,n_k]$.
    $\mathsf{CombineBodies}([b_1,\ldots,b_n],[n_1,\ldots,n_k]) = ?$
    * $n_1 \in \mathsf{ELEM} \Rightarrow$
      $\mathsf{CombineBodies}([b_1,\ldots,b_n],[,n_1,\ldots,n_k]) =$
      $\mathsf{CombineBodies}([b_1,\ldots,b_n,n_1],[n_2,\ldots,n_k])$
    * $n_1 \in \mathsf{TEXT}$:
      · if $b_n \in \mathsf{ELEM}\ \cup\ \mathsf{ATTR} \Rightarrow$
        $\mathsf{CombineBodies}(body_1, body_2) = \mathsf{CombineBodies}(body_1 \circ [n_1],[n_2,\ldots,n_k])$
      · if $b_n \in \mathsf{TEXT} \Rightarrow$
        Let $b' = \mathsf{Text}(\mathsf{Value}([b_n,n_1]))$. $\mathsf{CombineBodies}(body_1, body_2) =$
        $\mathsf{CombineBodies}([b_1,\ldots,b_{n-1},b'],[n_2,\ldots,n_k])$
    * $n_1 \in \mathsf{ATTR}$: $\mathsf{CombineBodies}(body_1, body_2) =$
      $\mathsf{CombineBodies}(\mathsf{InsertAttr}(body_1,n_1),[n_2,\ldots,n_k])$.

- $\mathsf{AddBody}$: $\mathsf{ELEMENT}\times\mathsf{NODELIST}\rightarrow\mathsf{ELEM}$ adds a new body (simply a nodelist) to the element's body.
  $\mathsf{AddBody}(\mathsf{Elem}(ename, ebody), newbody) =$
  $\mathsf{Elem}(ename, \mathsf{CombineBodies}(ebody, newbody))$.

- $\mathsf{StripXLinkAttrs}$: $\mathsf{ELEM} \rightarrow \mathsf{ELEM}$: Removes all *XLink attributes* from the element.
  $\mathsf{StripXLinkAttrs}(\mathsf{Elem}(ename,[b_1,\ldots,b_n])) = \mathsf{Elem}(ename, \mathsf{StripXLinkAttrs*}([b_1,\ldots,b_n]))$
  Auxiliary function $\mathsf{StripXLinkAttrs*}$: $\mathsf{NODELIST} \rightarrow \mathsf{NODELIST}$:
  - $\mathsf{StripXLinkAttrs*}([b_1,\ldots,b_n]) =$
    * $\mathsf{Name}(b_1)$ belongs to one of the namespaces xlink or dbxlink:
      $\Rightarrow \mathsf{StripXLinkAttrs*}([b_2,\ldots,b_n])$
    * else:
      $\Rightarrow [b_1] \circ \mathsf{StripXLinkAttrs*}([b_2,\ldots,b_n])$
  - $\mathsf{StripXLinkAttrs*}([]) = []$.

- $\mathsf{StripXLinkElems} : \mathsf{ELEM} \rightarrow \mathsf{ELEM}$: Removes all *XLink Element Children* from the element.
  $\mathsf{StripXLinkElems}(\mathsf{Elem}(ename, [b_1, \ldots, b_n])) = \mathsf{Elem}(ename, \mathsf{StripXLinkElems*}([b_1, \ldots, b_n]))$
  Auxiliary function $\mathsf{StripXLinkElems*} : \mathsf{NODELIST} \rightarrow \mathsf{NODELIST}$:
  - $\mathsf{StripXLinkElems*}([b_1, b_2, \ldots, b_n]) = ?$
    - $b_1 \in \mathsf{ELEM}, b_1 = \mathsf{Elem}(ename, [n_1, \ldots, n_{k_1}])$.
      - $\exists\ \ 1 \leq i \leq k_1 : \mathsf{Name}(n_i) \in \{$"xlink:type", "dbxlink:type"$\}$:
        $\Rightarrow \mathsf{StripXLinkElems*}([b_2, \ldots, b_n]) \Rightarrow [b_1] \circ \mathsf{StripXLinkElems*}([b_2, \ldots, b_n])$
    - $b_1 \in \mathsf{TEXT}\ \cup\ \mathsf{ATTR}$:
      $\Rightarrow [b_1] \circ \mathsf{StripXLinkElems*}([b_2, \ldots, b_n])$
  - $\mathsf{StripXLinkElems*}([]) = []$.

- $\mathsf{StripXLinkNodes} : \mathsf{ELEM} \rightarrow \mathsf{ELEM}$: Removes all *XLink Attributes* and *XLink children* from the element.
  $\mathsf{StripXLinkNodes}(elem) = \mathsf{StripXLinkElems}(\mathsf{StripXLinkAttrs}(elem))$.

- $\mathsf{Flat} : \mathsf{NODELISTLIST} \rightarrow \mathsf{NODELIST}$:
  $\mathsf{Flat}([nl_1, \ldots, nl_k]) = nl_1 \circ \mathsf{Flat}([nl_2, \ldots, nl_k])$.
  $\mathsf{Flat}([]) = []$.

- $\mathsf{Unflat} : \mathsf{NODELIST} \rightarrow \mathsf{NODELISTLIST}$:
  $\mathsf{Unflat}([n_1, \ldots, n_k]) = [[n_1] \circ \mathsf{Unflat}([n_2, \ldots, n_k]]$.
  $\mathsf{Unflat}([]) = []$.

- $\mathsf{EvalXPointer} : \mathsf{STRING} \rightarrow \mathsf{NODELIST}$
  returns a nodeset representing the result of an XPointer evaluation against a virtual instance with the URI containing the XPointer given as string. A URI as given in an href-attribute value of an XLink element is usually composed of the following parts:

  $$\underbrace{\mathsf{www.foo.org/}}_{\text{server}}\underbrace{\mathsf{mondial\text{-}countries.xml}}_{\text{path}}\#\underbrace{\mathsf{xpointer}}_{\substack{\text{xpointer}\\\text{scheme}}}\underbrace{\mathsf{("//country[car\_code='B']")}}_{\text{xpath expression}}$$

  The URI contains:
  - server and path information which indicate the queried XML document instance,
  - and an XPointer expression identifying a nodelist.

  It should be stressed that the XPath expression is not evaluated against the (physical) target document, but against the virtual instance induced by the target document plus XLinks[5].

[5]For details on this aspect, please refer to Section 6.3 where the notion of different degrees of transparency is explained.

5.1.3 Operators ϕ and ϕ^*

The operators ϕ (phi) and ϕ^* (phi-star) traverse the physical instance, recursively following its tree structure. Each found XLink is expanded with the operator γ. Other nodes remain unchanged. These operators describe the mapping of the physical to the virtual instance, thereby defining the logical data model for Simple Links.

An XPointer expression, given within a single Simple Link element, can reference multiple nodes in a remote instance. Thus, γ and ϕ algebraically map single nodes to nodelists.

The *bucket* collects certain elements which have been addressed with an XLink element with a dbxlink:transparent="make-attribute" directive. The make-attribute directive is covered in more detail later.

The Operator ϕ

Signature:

$$\phi : \mathsf{NODE} \rightarrow \mathsf{NODELIST} \times \mathsf{NODELIST}$$
$$\phi(node) \mapsto (newnodelist, newbucket)$$

Definition:

$\phi(node) = ?$

- $node \in \mathsf{TEXT} \cup \mathsf{ATTR} : \phi(node) = ([node], [])$
- $node \in \mathsf{ELEM}$ and $node$ is not an XLink element node:
  $\phi(node) = ([\mathsf{Elem}(\mathsf{Name}(node), elembody)], bucket)$
  with $(elembody, bucket) = \phi^*(\mathsf{Body}(node))$
- if ϕ's argument is an XLink element node, map the element with γ:
  $\phi(xlinkelem) =$
  $(phi\_elements, phi\_bucket \circ gamma\_bucket)$
  $(phi\_elements, phi\_bucket) = \phi^*(gamma\_children)$
  $(gamma\_children, gamma\_bucket) = \gamma(xlinkelem)$.

The Operator ϕ^*

Signature:

$$\phi^* : \mathsf{NODELIST} \rightarrow \mathsf{NODELIST} \times \mathsf{NODELIST}$$
$$\phi^*(nodelist) \mapsto (newnodelist, bucket)$$

Definition:

$\phi^*([node_1, \ldots, node_k]) = (result_1 \circ \ldots \circ result_k, bucket_1 \circ \ldots \circ bucket_k)$
with $(result_i, bucket_i) = \phi(node_i)$ for all $1 \leq i \leq k$.

5.1.4 Transformation Start

The initial transformation starts with *root_node* as the root node of the physical instance which is the entry point to the virtual instance.

$$(\mathit{virtual\_instance}, \mathit{bucket}) = \phi(\mathit{root\_node}).$$

5.1.5 Signature and Definition of γ

The γ operator expands Simple Link elements: applied to an Simple Link element, γ expands the Simple Link according to its xlink:type attribute, the XPointer expression in attribute xlink:href and transparent directive in attribute dbxlink:transparent.

As described in Section 3.3, dbxlink:transparent contains the (*R-directive* and the *L-directive*), the first one determining the processing of the link element itself, and the latter one determining the processing of the remote result. L-directives can have the values {duplicate-element | group-in-element | drop-element | keep-body | make-attribute }. Result directives have values { insert-nodes | insert-bodies | insert-nothing }.

Signature:

$$\gamma : \mathsf{ELEM} \rightarrow \mathsf{NODELIST} \times \mathsf{NODELIST}$$

$$\gamma : (\mathit{xlink}) \mapsto (\mathit{xlink\_result}, \mathit{newbucket})$$

Definition:

- xlink is of type "simple":
  $\gamma(\mathit{xlink}) = (\mathit{xlink\_result}, \mathit{bucket})$
  with
  $\mathit{xlink\_result} = \mathsf{Flat}(\mathit{nodelistlist}_{LR})$
  $(\mathit{nodelistlist}_{LR}, \mathit{bucket}) = \gamma_{LR}(\mathit{xlink})$

- xlink is of type "extended":
  $\gamma(\mathit{xlink}) = \gamma_X(\mathit{xlink})$

For γ, the sub-operators γ_L, γ_R, γ_{LR} (for Simple Links) and γ_X (for Extended Links) exist. γ_R does the *result set mapping*, where the result set from the XPointer evaluation is mapped according to the given R-directive. γ_L does the *link mapping*, where the mapped result set is mapped into the link element according to the L-directive. γ_{LR} performs the composition of both γ_L and γ_R. γ_X is relevant for Extended Link processing, which is addressed in Section 6.1.

5.1.6 Signature and Definition of γ_{LR}

Signature:

$$\gamma_{LR} : \mathsf{ELEM} \rightarrow \mathsf{NODELISTLIST} \times \mathsf{NODELIST}$$

Definition:

$$\gamma_{LR}(xlink) = (nodelist\_list_L, bucket)$$

with
$(nodelist\_list_L, bucket) = \gamma_L(xlink, nodelist\_list_R)$
$nodelist\_list_R = \gamma_R(xlink, \mathsf{EvalXPointer}(xpointer))$
$xpointer = \mathsf{Value}(\mathsf{AttrByName}(\mathsf{Body}(xlink), \text{“xlink:href”}))$ is the XPointer expression from the xlink element's xlink:href-attribute.

5.1.7 Signature and Definition of γ_L

Signature:

$$\gamma_L : \mathsf{ELEM} \times \mathsf{NODELISTLIST} \to \mathsf{NODELISTLIST} \times \mathsf{NODELIST}$$

$$\gamma_L : (xlink, result_R) \mapsto (result_L, bucket)$$

with $result_R = [r_1, \ldots, r_k]$ being a list of nodelists, which is the intermediate result of the R-directive applied to the Simple Link's XPointer's result.

The behavior of γ_L is determined by the L-directive in the dbxlink:transparent attribute.

Definition:

- L-directive is drop-element :
  The link element is dropped and replaced with the result from γ_R:
  $\gamma_L(xlink, [r_1, \ldots, r_k]) =$
  $\mathsf{CombineBodies}(r_1, \mathsf{CombineBodies}(r_2, \ldots, \mathsf{CombineBodies}(r_{k-1}, r_k) \ldots))$.

  Note that r_i is not a single node, but each r_i stands for a nodelist containing nodes[6].

- L-directive is group-in-element :
  All element bodies from the result are gathered and added to the body of the XLink element:
  $\gamma_L(xlink, [r_1, \ldots, r_k]) =$
  $([\mathsf{AddBody}(\mathsf{AddBody}(\ldots$
  $\mathsf{AddBody}(\mathsf{AddBody}(\mathsf{StripXLinkAttrs}(xlink), r_1), r_2), \ldots, r_{k-1}), r_k)], [])$.

- L-directive is duplicate-element :
  The XLink element is duplicated for each element body from the result, forming its new “element hull”:
  $\gamma_L(xlink, [r_1, \ldots, r_k]) = ([[e_1], \ldots, [e_k]], [])$ with
  $e_i = \mathsf{AddBody}(\mathsf{StripXLinkAttrs}(xlink), r_i)$ for all $1 \le i \le k$.

[6]If the R-directive specifies insert-bodies, each nodelist contains one element's body. If the R-directive specifies insert-nodes, the result contains each result node unmodified, but packed into a single nodelist.

- L-directive is keep-body :
  The link element is dropped, and its body is mapped into each of the elements inside every body in the result of γ_R:
  $\gamma_L(xlink, [r_1, \ldots, r_k]) = ([r'_1, \ldots, r'_k], [])$ with
  (let $r_i =: [n_{i1}, \ldots, n_{ik_i}]$ be the single nodelists)
  $r'_i := [n'_{i1}, \ldots, n'_{ik_i}]$,

  $$n'_{ij} := \begin{cases} \mathsf{AddBody}(n_{ij}, \mathsf{Body}(\mathsf{StripXLinkAttrs}(xlink))), \text{ if } n_{ij} \in \mathsf{Elem} \\ n_{ij}, \text{ else.} \end{cases}$$

  with $n_{i1}, \ldots, n_{ik_i}$ being the nodes in the result's i-th element body.

- L-directive is make-attribute :
  The link element is dropped and replaced by a *virtual attribute* with the name of the link element. The virtual attribute's value depends on the type of the nodes in the result:

  (i) for a text or attribute note in the result, the virtual attribute's value is the node's string value. If there are multiple result nodes, the single string values are concatenated to form the virtual attribute's value.

  (ii) For an element node in the result, the element is equipped with a newly generated, virtual-instance-wide unique ID value, and inserted into the bucket. The virtual attribute obtains the newly generated value as an IDREF value.

  If there are multiple elements in the result node, all are added to the bucket, and the virtual attribute obtains an IDREFS value, referencing all newly added elements[7]. If an element node already has an ID attribute, the present one is re-used instead of generating a new one.

  $\gamma_L(xlink, [r_1, \ldots, r_k]) =$
  $([\mathsf{Attr}(\mathsf{Name}(xlink), \mathsf{AttrsValue}([v_{11}, \ldots, v_{1n_1}, \ldots\ldots, v_{k1}, \ldots, v_{kn_k}])],$
  $b_{11} \circ \ldots \circ b_{1n_1} \circ \ldots\ldots \circ b_{k1} \circ \ldots \circ b_{kn_k})$ with

  $$v_{ij} := \begin{cases} \mathsf{Attr}(\text{"dbxlink:id"}, id_{ij}) \text{ if } n_{ij} \text{ is an element node,} \\ n_{ij}, \text{else.} \end{cases}$$
  $$b_{ij} := \begin{cases} [\mathsf{AddBody}(n_{ij}, [\mathsf{Attr}(\text{"dbxlink:id"}, id_{ij})]) \circ \mathsf{StripXLinkNodes}(\mathsf{Body}(xlink))], \\ \qquad\qquad \text{if } n_{ij} \text{ is an element node,} \\ [] \text{ else.} \end{cases}$$

  with each id_{ij} being a virtual instance-wide unique ID value. Note that the $\mathsf{Attr}(\mathsf{Name}(xlink), [\ldots])$ attribute is of the type IDREF/IDREFS, and

[7]Note that make-attribute delivers a "sensible" mapping only if the referenced nodes are *either* elements (resulting in an IDREF or IDREFS value) *or* attribute and text nodes (resulting in a CDATA value). Mapping a mixture of element and other nodes with *make-attribute* results in a concatenation of both IDREFs and regular CDATA values, which is utterly useless, since at least the IDREFs cannot be resolved anymore.

that the $\mathsf{Attr}(\text{“}\mathsf{dbxlink{:}id}\text{”}, id_{ij})$ attributes inside the b_{ij}'s are of the type ID.

5.1.8 Signature and Definition of γ_R

Signature:

$$\gamma_R : \mathsf{ELEM} \rightarrow \mathsf{NODELISTLIST}$$

$$\gamma_R(xlink) = ?$$

Let $[n_1, \dots, n_k] = \mathsf{Eval}(\mathsf{Value}(\mathsf{GetAttrByName}(\mathsf{Attrs}(xlink), \text{"}\mathsf{xlink{:}href}\text{"})))$ be the resulting nodes from evaluating the XPointer expression in the XLink element. Definition:

- result directive is insert-nodes : $\gamma_R(xlink) = \mathsf{Unflat}([n_1, \dots, n_k])$
- result directive is insert-bodies : $\gamma_R(xlink) = [\mathsf{Body}(n_1), \dots, \mathsf{Body}(n_k)]$
- result directive is insert-nothing : $\gamma_R(xlink) = []$.

5.2 Finite Data Model, Cycle Detection and Link Bombs

The regular XML data model defines an XML instance as a monolithic tree structure with the document root element as the tree's root. The logical data model discussed in this work changes the picture: link structures may be cyclic, so that a complete serialization would be infinite. Consider the following Mondial neighbor example:

Example 6 *In the distributed* MONDIAL*, each country element has a number of neighbor Simple Link elements as its children that reference the country's neighboring countries (except countries that are islands). E.g. France has one neighbor element to each of its neighbors Belgium, Germany etc. So, the query*

.../country[name="Belgium"]/neighbor/name

yields all names of France's neighbor countries, as there are Belgium, Luxemburg, Germany, Switzerland, Italy, Monaco, Spain and Andorra.
Belgium again references France as its neighbor. So, a query

.../country[name="Belgium"]/neighbor/name/neighbor/name

will yield all France's neighbors' neighbors' names, including France itself.
Consider now a query

.../country[name="Belgium"]//neighbor/name.

Due to the use of the descendant-or-self*-axis ("//"), the query would not terminate, if evaluated naïvely: Since each of Belgium's neighboring countries again has a number of neighboring countries, and each of the neighbors' neighboring countries again has neighbors and so on, the query would try to expand and to follow each neighbor link again and again. This process won't terminate, since neighborhood relations are cyclic: if A is B's neighbor, then B is also A's neighbor.*

Nevertheless, the query is well-defined because it has a fixed result, consisting of the transitive closure of the neighbor relation between all countries reachable from France crossing borders on land.

As shown in the example above, the virtual instance may have an infinite serialization. Nevertheless, the data model itself is finite, since a regular XML instance in the XLink-transparent data model – even if infinite in its materialization – still has, or at least, may have a finite representation. The tree data model of regular XML is expanded to a more general graph model, which may contain cycles, but still may be navigated and evaluated using XPath expressions. But, as the above example suggests, when implementing an XPath/XQuery engine, one has to deal with cycle detection, especially in the presence of queries containing the descendant or descendant-or-self axes. Cycle detection is a well understood problem in graph theory, and the results from there can be applied for implementing a query engine. For more details on cycle detection in XPath engines over the above data model, please refer to [Beh06].

5.2.1 Not Well-Defined Instances

However, there are link constructs that have a finite representation, but do induce virtual instances that are not well-defined, either in terms of a finite expansion, or in terms of evaluating them in finite time. E.g. a Simple Link referencing its parent element with drop-element, adding xlink:href, xlink:type and dbxlink:transparent attributes to its parent element, again referencing the parent's parent, and so on, leading straight to the root element, would "consume" the complete instance. There are several kinds of link constructs showing this or similar pathologic behavior. To avoid these, it is sufficient to forbid "self-modifying" virtual instances: the creation of new XLink elements by links that add xlink and dbxlink attributes to some element is generally not allowed. For a detailed treatment of the "link bomb" phenomenon, please refer again to [Beh06].

Chapter 6

The Logical Data Model for 3rd Party Links

For Simple Links, a formal description of the mapping of the physical to the virtual data model has been given in 5. It is based on a set of operators (i) traversing the physical data model and searching for XLinks (ϕ, ϕ^*), and (ii) expanding an XLink *in place* if found ($\gamma, \gamma_L, \gamma_R, \gamma_{LR}$).

In Section 4.1, it was pointed out that there exist three different perspectives on 3rd Party Links, defining three different semantics. Thus, the forward and inverse perspective demand a different way of specifying Extended Link semantics than the relation perspective: the relation perspective of an arc defines the mapping from where the arcs are *defined*, namely inside the linkbase document. This is in some way the same as with Simple Links: a Simple Link defined in a document A embeds parts of a remote document B into A (described in Section 4.4). In contrast, arcs in the forward and inverse perspective define a physical-to-virtual mapping from the perspective where the arc *starts* (forward), or where it *ends* (inverse, both described in Section 4.5).

The chapter starts with the relation perspective in Section 6.1. Section 6.2 gives a specification of the mapping for forward and inverse perspectives. All three types of mappings are described using the $\phi, \phi^*, \gamma, \gamma_L, \gamma_R$ and γ_{LR} from the Simple Links mapping, familiar from 5.

6.1 Description of the Mapping for the relation Perspective

As for Simple Links, a set of operators traverse a physical instance (the linkbase / the Extended Link element), finding and expanding XLink elements: Extended Link, arc, locator and dbxlink:relation elements define the mapping for the relation perspective. The traversal operators are the same for both Simple and Extended Links, namely ϕ and ϕ^*. The expansion operators are given with

γ_X (Extended Link elements) and the sub-operators ϕ_X, ϕ^*_X, γ_{ext}, γ_{arc}, γ^*_{LR}, FilterXLinks, Arc2Simple, Loc2Simple, FlatCardList, GetRoleName, DoCard and GetArcDirectives which will be introduced alongside.

6.1.1 Definition of γ_X

γ_X maps the entire Extended Link structure to the virtual instance of the relation perspective, using a number of sub-operators: γ_{ext} combines the Extended Link element (and its physical children and attributes) with the result of the inner arc and locator elements. Arc and locator elements are mapped using γ_{arcloc}.

$$\gamma_X(extlink) = \gamma_{ext}(extlink, \phi_X(extlink))$$

Definition of γ_{ext}:

The operator γ_{ext} expands the Extended Link element according to the given L-directive, which is the same as γ_L does with Simple Links. Nevertheless, the expansion semantics for the Extended Link element is a little different from the semantics for Simple Links. To motivate why an additional operator γ_{ext} is needed, consider the following example of an Extended Link with one arc element and some non-XLink children. The arc element arc is expanded to two element nodes, "x" and "y" (and some content):

```
<extlink xlink:type="extended" dbxlink:transparent="duplicate-element">
   <pre1/><pre2/><pre3/>
   <arc xlink:type="arc" .../>  ──────────→
   <post1/><post2/>
</extlink>
                                [[<x>content of x</x>],[<y>content of y</y>]]
```

The result will look like:

```
<extlink>
   <pre1/><pre2/><pre3/><x>content of x</x><post1/><post2/>
</extlink>
<extlink>
   <pre1/><pre2/><pre3/><y>content of y</y><post1/><post2/>
</extlink>
```

For duplicating the extlink element, the locator is evaluated (which yields two nodelists, each containing one "x" element), and extlink is "wrapped around" each locator's result, inserting the result itself *in place* at the arc's previous position in the physical instance: after the <pre*/>, but before the <post*/> elements. The plain γ_L operator cannot be made to process arc results which are spatially spread over a link element's child nodes. This special in-place treatment of arc/locator result requires a specific γ_{ext} operator:

$$\gamma_{ext} : \mathsf{ELEM} \times < (\mathsf{NODELIST} \times \mathsf{ELEM} \times \mathsf{NODELIST}) > \rightarrow \mathsf{NODELIST} \times \mathsf{NODELIST}$$

$\gamma_{ext}(extlink, [(pre_1, link_1, post_1), \ldots, (pre_n, link_n, post_n)]) = (\mathsf{Flat}(result), bucket)$

Let $transval$ =Value(AttrByName(Body($extlink$),"dbxlink:transparent")).

- if $transval$ =drop-element, then:
  $result = result_1 \circ result_{rest}$
  $bucket = bucket_1 \circ bucket_{rest}$
  $(result_1, bucket_1) = \gamma_{arcloc}(extlink, link)$
  $(result_{rest}, bucket_{rest}) = \gamma_{ext}(extlink, [(pre_2, link_2, post_2), \ldots, (pre_n, link_n, post_n)])$

- if $transval$ =keep-body, then:
  $result = result_1 \circ result_{rest}$
  $bucket = bucket_1 \circ bucket_{rest}$
  $(tmp\_result, tmp\_bucket_1) = \gamma_L(new\_extlink, arclocresult)$
  $new\_extlink = \mathsf{Elem}(\mathsf{Name}(extlink), [\mathsf{Attr}(\text{xlink:type}, \text{"simple"}),$
  $\mathsf{Attr}(\text{dbxlink:transparent}, \text{"keep-body"})] \circ pre_1 \circ post_1)$

  $(arclocresult, tmp\_bucket_2) = \gamma_{arcloc}(extlink, link_1)$
  $tmp\_bucket = tmp\_bucket_1 \circ tmp\_bucket_2.$
  $(result_{rest}, bucket_{rest}) = \gamma_{ext}(extlink, [(pre_2, link_2, post_2), \ldots, (pre_n, link_n, post_n)])$

- if $transval$ =group-in-element, then:
  $(result, bucket) = ([[\mathsf{Elem}(\mathsf{Name}(extlink), elembody)]], bucket)$
  $(elembody, bucket) = \gamma^*_{ext}(extlink, \mathsf{Body}(extlink)).$
  Auxiliary function γ^*_{ext} : ELEM×NODELIST → NODELISTLIST × NODELIST
  $\gamma^*_{ext}(extlink, [n_1, \ldots, n_k]) =?$

  - $n_1 \in$ ATTR ∪ TEXT :
    $\gamma^*_{ext}(extlink, [n_1, \ldots, n_k]) = ([[n_1]] \circ result, bucket)$
    $(result, bucket) = \gamma^*_{ext}(extlink, [n_2, \ldots, n_k]).$
  - $n_1 \in$ ELEM and n_1 is a non-XLink element: same as above.
  - $n_1 \in$ ELEM and n_1 *is* an XLink element:
    $\gamma^*_{ext}(extlink, [n_1, \ldots, n_k]) = (result_1 \circ result_{rest}, bucket_1 \circ bucket_{rest})$
    $(result_1, bucket_1) = \gamma_{arcloc}(extlink, n_1)$
    $(result_{rest}, bucket_{rest}) = \gamma^*_{ext}(extlink, [n_2, \ldots, n_k])$
  - $\gamma^*_{ext}(extlink, []) = ([], []).$

- if $transval$ =duplicate-element, then:
  $result = result_1 \circ result_{rest}$
  $bucket = bucket_1 \circ bucket_{rest}$
  $result_1 = \mathsf{Elem}(\mathsf{Name}(extlink), pre_1 \circ \mathsf{Flat}(linkresult) \circ post_1)$
  $(linkresult, bucket_1) = \gamma_{arcloc}(extlink, link_1)$
  $(result_{rest}, bucket_{rest}) = \gamma_{ext}(extlink, [(pre_2, link_2, post_2), \ldots, (pre_n, link_n, post_n)])$

- if $transval$ =make-attribute, then:
  $\gamma_{ext}(extlink, [(pre_1, link_1, post_1), \ldots, (pre_n, link_n, post_n)]) =$
  $\mathsf{Attr}(\mathsf{Name}(extlink), \mathsf{Eval}(\gamma^{(attr)}_{ext}([link_1, \ldots, link_n])))$

Auxiliary operator $\gamma_{ext}^{(attr)}$: $\mathsf{NODELIST} \rightarrow \mathsf{NODELIST}$:
$\gamma_{ext}^{(attr)}([link_1, \ldots, link_n]) = (result_1 \circ result_{rest}, bucket_1 \circ bucket_2 \circ bucket_{rest})$
$(result_1, bucket_1) = \gamma_L(newlink, linkresult)$

$$\begin{aligned} newlink = {} & \mathsf{Elem}(\text{"dbxlink:id"}, [\mathsf{Attr}(\text{xlink:type}, \text{"simple"}), \\ & \mathsf{Attr}(\text{dbxlink:transparent}, \text{"make-attribute"})]) \end{aligned}$$
$(linkresult, bucket_2) = \gamma_{arcloc}(extlink, link_1)$
$(result_{rest}, bucket_{rest}) = \gamma_{ext}(extlink, [(pre_2, link_2, post_2), \ldots, (pre_n, link_n, post_n)])$

and finally: $\gamma_{ext}(extlink, []) = ([], [])$.

Definition of ϕ_X:

Operator ϕ_X traverses an Extended Link element's children. It yields a list consisting of one tuple $(pre, link, post)$ for each found arc/locator with *link* being the arc/locator element itself, and *pre* / *post* denoting all preceding / following non-XLink sibling elements of *link*. Thus, it produces the input list for the γ_{ext} operator.

ϕ_X : $\mathsf{ELEM} \rightarrow \mathsf{LIST} < (\mathsf{NODELIST} \times \mathsf{ELEM} \times \mathsf{NODELIST}) >$

$\phi_X(extlink) = \phi_X^*([], \mathsf{Children}(extlink))$
$\phi_X^*([n_1, \ldots, n_{i-1}], [n_i, \ldots, n_k]) =?$

- $n_i \in \mathsf{TEXT}$: $\phi_X^*([n_1, \ldots, n_{i-1}], [n_i, \ldots, n_k]) = \phi_X^*([n_1, \ldots, n_i], [n_{i+1}, \ldots, n_k])$
- $n_i \in \mathsf{ELEM}$ and n_i is no arc or locator:
  $\phi_X^*([n_1, \ldots, n_{i-1}], [n_i, \ldots, n_k]) = \phi_X^*([n_1, \ldots, n_i], [n_{i+1}, \ldots, n_k])$
- $n_i \in \mathsf{ELEM}$ and n_i *is* an arc or locator:
  $\phi_X^*([n_1, \ldots, n_{i-1}], [n_i, \ldots, n_k]) =$
  $[([n_1, \ldots, n_{i-1}], n_i, \mathsf{FilterXLinks}([n_{i+1}, \ldots, n_k]))] \circ \phi_X^*([n_1, \ldots, n_{i-1}], [n_{i+1}, \ldots, n_k])$
- $\phi_X^*([n_1, \ldots, n_k], []) = []$.

Auxiliary operator FilterXLinks: $\mathsf{NODELIST} \rightarrow \mathsf{NODELIST}$:
$\mathsf{FilterXLinks}([n_1, \ldots, n_k]) =?$

- $n_1 \in \mathsf{TEXT}$: $\mathsf{FilterXLinks}([n_1, \ldots, n_k]) = [n_1] \circ \mathsf{FilterXLinks}([n_2, \ldots, n_k])$
- $n_1 \in \mathsf{ELEM}$ and n_1 is a non-XLink element (arc or locator):
  $\mathsf{FilterXLinks}([n_1, \ldots, n_k]) = [n_1] \circ \mathsf{FilterXLinks}([n_2, \ldots, n_k])$
- $n_1 \in \mathsf{ELEM}$ and n_1 *is* an XLink element (arc or locator):
  $\mathsf{FilterXLinks}([n_1, \ldots, n_k]) = \mathsf{FilterXLinks}([n_2, \ldots, n_k])$
- $\mathsf{FilterXLinks}([]) = []$.

Definition of γ_{arcloc}:

γ_{arcloc} maps a link element – either an arc or a locator element – to its result.

$$\gamma_{arcloc} : \mathsf{ELEM} \times \mathsf{ELEM} \rightarrow \mathsf{NODELISTLIST} \times \mathsf{NODELIST}$$

$\gamma_{arcloc}(extlink, link) =?$

- *link* is an arc element:
  $\gamma_{arcloc}(extlink, link) = \gamma_{arc}(extlink, link)$
- *link* is a locator element:
  $\gamma_{arcloc}(extlink, link) = \gamma_{LR}(link)$

Definition of γ_{arc}:

$\gamma_{arc}(extlink, arc) = \gamma_L(\mathsf{Arc2Simple}(arc, perspective), \mathsf{FlatCardList}(\mathsf{DoCard}(fromresult, toresult)))$
$(fromresult, frombucket) = \gamma^*_{LR}(\mathsf{Loc2Simple}(extlink, arc, \text{"xlink:from"}))$[1]
$(toresult, tobucket) = \gamma^*_{LR}(\mathsf{Loc2Simple}(extlink, arc, \text{"xlink:to"}))$
Auxiliary operators:

- $\gamma^*_{LR} : \mathsf{NODELIST} \rightarrow \mathsf{NODELISTLIST} \times \mathsf{NODELIST}$
  $\gamma^*_{LR}([link_1, \ldots, link_n]) = (result_1 \circ result_{rest}, bucket_1 \circ bucket_{rest})$
  $(result_1, bucket_1) = \gamma_{LR}(link_1)$
  $(result_{rest}, bucket_{rest}) = \gamma^*_{LR}([link_2, \ldots, link_n])$
- Arc2Simple transforms an arc element to a Simple Link element, so that it can be processed with γ_L according to its *ext-L-dir* value.

  $\mathsf{Arc2Simple} : \mathsf{ELEM} \times \mathsf{STRING} \rightarrow \mathsf{ELEM}$
  $\mathsf{Arc2Simple}(arc, perspective) =$
  $\quad \mathsf{Elem}(\mathsf{GetRolename}(arc, perspective),$
  $\quad\quad [\mathsf{Attr}(\text{"xlink:type"}, \text{"simple"}), \mathsf{Attr}(\text{"dbxlink:transparent"}, arcdir)]$
  $\quad\quad \circ \mathsf{StripXLinkNodes}(\mathsf{Body}(arc)))$
  with $(arcdir, card, place) = \mathsf{GetArcDirectives}(arc, perspective)$

- Loc2Simple transforms an arc's locators to a list of Simple Links, which then can be processed with γ^*_{LR}.

  $\mathsf{Loc2Simple} : \mathsf{ELEM} \times \mathsf{ELEM} \times \mathsf{STRING} \rightarrow \mathsf{NODELIST}$
  $\mathsf{Loc2Simple}(extlink, arc, fromto) = [simple_1, \ldots, simple_n]$
  Let $locs = \mathsf{ArcGetLocs}(extlink, arc, fromto) = [loc_1, \ldots, loc_n]$
  Let $\forall 1 \leq i \leq n : simple_i =$
  $\quad \mathsf{Elem}(\mathsf{Name}(loc_i), \mathsf{StripXLinkNodes}(\mathsf{Body}(loc_i) \circ [type, href, newtrans]))$
  with

[1]The definition of γ^*_{LR} is necessary, since an Extended Link element may have multiple locator elements with the same xlink:label value. So, an arc can reference multiple from-locators, with each of them being processed with γ_{LR}. γ^*_{LR} applies γ_{LR} to a nodelist of locators.

$type =$ Attr("xlink:type","simple")
$href =$ AttrByName(loc_i,"xlink:href")
$trans =$ Value(AttrByName(loc_i,"dbxlink:transparent"))
$newtrans =$ Attr("dbxlink:transparent",$newtransval$)
$newtransval = lparam \circ rparam$

with:
<u>if $fromto =$ "xlink:from":</u>

“drop-from-elem” $\in trans \Rightarrow lparam =$ “drop-element”
“group-from-elem” $\in trans \Rightarrow lparam =$ “group-in-element”
“dup-from-elem” $\in trans \Rightarrow lparam =$ “duplicate-element”
“keep-from-body” $\in trans \Rightarrow lparam =$ “keep-body”
“make-from-attr” $\in trans \Rightarrow lparam =$ “ make-attribute”

“insert-from-nodes” $\in trans \Rightarrow rparam =$ “ insert-nodes”
“insert-from-bodies” $\in trans \Rightarrow rparam =$ “ insert-bodies”
“insert-from-noth” $\in trans \Rightarrow rparam =$ “ insert-nothing”

<u>if $fromto =$ "xlink:to":</u>

“drop-to-elem” $\in trans \Rightarrow lparam =$ “drop-element”
“group-to-elem” $\in trans \Rightarrow lparam =$ “group-in-element”
“dup-to-elem” $\in trans \Rightarrow lparam =$ “duplicate-element”
“keep-to-body” $\in trans \Rightarrow lparam =$ “keep-body”
“make-to-attr” $\in trans \Rightarrow lparam =$ “ make-attribute”

“insert-to-nodes” $\in trans \Rightarrow rparam =$ “ insert-nodes”
“insert-to-bodies” $\in trans \Rightarrow rparam =$ “ insert-bodies”
“insert-to-noth” $\in trans \Rightarrow rparam =$ “ insert-nothing”

- ArcGetLocs: ELEM $\times$ ELEM $\times$ STRING $\rightarrow$ NODELIST
  ArcGetLocs($extlink, arc, fromto$) = ArcGetLocs*(Body($extlink$),
  Value(AttrsByName(Body(arc), $fromto$)))

  ArcGetLocs*($[n_1, \ldots, n_k], label$) =?

  - $n_1 \in$ TEXT $\cup$ ATTR :
    ArcGetLocs*($[n_1, \ldots, n_k], label$) = ArcGetLocs*($[n_2, \ldots, n_k], label$)
  - $n_1 \in$ ELEM and AttrByValue(Body(n_1), $label$) = Attr("xlink:label", $label$):
    ArcGetLocs*($[n_1, \ldots, n_k], label$) = $[n_1] \circ$ ArcGetLocs*($[n_2, \ldots, n_k], label$)
  - $n_1 \in$ ELEM and AttrByValue(Body(n_1), $label$) $\neq$ Attr("xlink:label", $label$):
    ArcGetLocs*($[n_1, \ldots, n_k], label$) = ArcGetLocs*($[n_2, \ldots, n_k], label$)
  - ArcGetLocs($[], label$) = [].

- GetArcDirectives : ELEM $\times$ STRING $\rightarrow$ STRING $\times$ STRING $\times$ STRING
  GetArcDirectives($arc, perspective$) = $lparam \circ$ " " $\circ$ $rparam$

with:
Let $trans$ = Value(AttrByName(Body(arc),"dbxlink:transparent").

"drop-arc-elem" $\in trans$	$\Rightarrow lparam =$ "drop-element"
"group-arc-elem" $\in trans$	$\Rightarrow lparam =$ "group-in-element"
"dup-arc-elem" $\in trans$	$\Rightarrow lparam =$ "duplicate-element"
"keep-arc-body" $\in trans$	$\Rightarrow lparam =$ "keep-body"
"make-arc-attr" $\in trans$	$\Rightarrow lparam =$ "make-attribute"
"1-1" $\in trans$	$\Rightarrow card =$ "1-1"
"1-n" $\in trans$	$\Rightarrow card =$ "1-n"
"n-1" $\in trans$	$\Rightarrow card =$ "n-1"
"n-m" $\in trans$	$\Rightarrow card =$ "n-m"
"insert" $\in trans$	$\Rightarrow place =$ "insert"
"replace" $\in trans$	$\Rightarrow place =$ "replace"
"fuse" $\in trans$	$\Rightarrow place =$ "fuse"

- AttrByValue : NODELIST $\times$ STRING $\rightarrow$ ATTR
  AttrByValue($[n_1, \ldots, n_k], value$) =?
  - $n_1 \in$ ELEM $\times$ TEXT :
    AttrByValue($[n_1, \ldots, n_k], value$) = AttrByValue($[n_2, \ldots, n_k], value$)
  - $n_1 \in$ ATTR and Value(n_1) $\neq value$:
    AttrByValue($[n_1, \ldots, n_k], value$) = AttrByValue($[n_2, \ldots, n_k], value$)
  - $n_1 \in$ ATTR and Value(n_1) $= value$:
    AttrByValue($[n_1, \ldots, n_k], value$) $= n_1$
  - AttrByValue($[], value$) = Null

- GetRolename($arc, perspective$) =
  Value(AttrByName(Body(NodeByName(Body(arc),
  $perspective$)),"dbxlink:rolename"))

- DoCard :
  NODELISTLIST $\times$ NODELISTLIST $\times$ STRING $\rightarrow$ LIST < NODELIST $\times$ NODELIST >
  DoCard($fromloc, toloc, card$) =?
  let $fromloc = [fl_1, \ldots, fl_n]$
  let $toloc = [tl_1, \ldots, tl_k]$
  - $card =$ 1-1 $\Rightarrow$
    $$\text{DoCard}(fromloc, toloc, \text{1-1}) = \begin{array}{l}[(fl_1, tl_1), \ldots, (fl_1, tl_k), \\ \quad \vdots \qquad\qquad \vdots \\ (fl_n, tl_1), \ldots, (fl_n, tl_k)]\end{array}$$
  - $card =$ 1-n $\Rightarrow$
    DoCard($fromloc, toloc,$ 1-n) =

$[(fl_1, toresult), \ldots, (fl_n, toresult)]$

- $card = \text{n-1} \Rightarrow$
  $\mathsf{DoCard}(fromloc, toloc, \text{n-1}) =$
  $[(fromresult, tl_1), \ldots, (fromresult, tl_n)]$
  with $fromresult =$
  $\mathsf{CombineBodies}(fl_1, \ldots, \mathsf{CombineBodies}(fl_{n-1}, fl_n) \ldots)$

- $card = \text{n-m} \Rightarrow$
  $\mathsf{DoCard}(fromloc, toloc, \text{n-m}) = [(fromresult, toresult)]$
  with $fromresult = \mathsf{CombineBodies}(fl_1, \ldots, \mathsf{CombineBodies}(fl_{n-1}, fl_n) \ldots)$
  and $toresult = \mathsf{CombineBodies}(tl_1, \ldots, \mathsf{CombineBodies}(tl_{k-1}, tl_k) \ldots)$

- $\mathsf{FlatCardList} : < \mathsf{NODELIST} \times \mathsf{NODELIST} > \rightarrow \mathsf{NODELISTLIST}$
  $\mathsf{FlatCardList}([(a_1, b_1), \ldots, (a_n, b_n)]) =$
  $[\mathsf{CombineBodies}(a_1, b_1)] \circ \mathsf{FlatCardList}([(a_2, b_2), \ldots, (a_n, b_n)])$
  $\mathsf{FlatCardList}([]) = []$.

- $\mathsf{DoPlacement} : \mathsf{LIST} < \mathsf{NODELIST} \times \mathsf{NODELIST} > \times \mathsf{STRING} \rightarrow \mathsf{NODELISTLIST}$
  $\mathsf{DoPlacement}([(A_1, B_1), \ldots, (A_n, B_n)], place) =$
  $[\mathsf{DoPlacementSingle} : (A_1, B_1, place)] \circ \mathsf{DoPlacement}([(A_2, B_2), \ldots, (A_n, B_n)], place)$
  $\mathsf{DoPlacement}([], place) = []$.

- $\mathsf{DoPlacementSingle} : \mathsf{NODELIST} \times \mathsf{NODELIST} \times \mathsf{STRING} \rightarrow \mathsf{NODELISTLIST}$
  $\mathsf{DoPlacementSingle}(A, B, place) = result$
  with: $A, B \in \mathsf{NODELIST}$,
  $A = [a_1, \ldots, a_m], B = [b_1, \ldots, b_k], a_i, b_j \in \mathsf{NODE}$ for $1 \leq i \leq m, 1 \leq j \leq k$

  - $= \mathsf{insert} \Rightarrow$
    $result = [[a'_1] \ldots, [a'_m]]$ with $a'_i \in \mathsf{NODE}$
    $$\forall 1 \leq i \leq m : a'_i = \begin{cases} \mathsf{AddBody}(a_i, B), & \text{if } a_i \in \mathsf{ELEM} \\ a_i, & \text{if } a_i \in \mathsf{ATTR} \cup \mathsf{TEXT} \end{cases}$$

  - $place = \mathsf{replace} \Rightarrow result = [\underbrace{B, \ldots, B}_{m \text{ times}}]$

  - $place = \mathsf{fuse} \Rightarrow result = [\mathsf{CombineBodies}(A, B)]$.

6.2 Extended Links – **forward** and **inverse** Perspective

The out-of-line definition and 3rd party semantics of linkbases for forward and inverse perspectives as described in Section 4.1 cannot be mapped with the ADT operators defined up to here, since γ and γ_X are based on mapping links *inline*. One idea is to expand the traversal functions ϕ and ϕ^* with an additional linkbase parameter LB (to provide it with the necessary linkbase information), and to apply them to the embedding resource (*from* for forward, *to* for inverse). $\phi_{(LB)}$ and $\phi^*_{(LB)}$ traverse an XLinked document, expanding Simple Links, and verifying at the same time if the traversed node is referenced by a locator in the linkbase LB. If the locator is an arc's from-locator, evaluate the arc's forward perspective. If the locator is an arc's to-locator, do the same with the arc's inverse perspective.

6.2.1 Placement Value determines Processing Order

Now, with the knowledge of the mapping operators γ_L and γ_R, DoPlacement, and DoCard and with knowledge of the meaning of L/R, placement and cardinality directives, but before starting with the formal details of the mapping, let's have a short – and less formal[2] – look at the placement directive, and its impact on the order of operator applications to the data model, since the value of the *placement directive* (insert, fuse, replace) determines the order of processing operations. Let *node* be a single node identified by the from-locator $fromloc$, *arc* the arc element, and *toloc* the to-locator.

- For forward perspective with $place \in \{\mathsf{insert}, \mathsf{replace}\}$ the processing order is:

  $$\mathsf{DoPlacement}(\mathsf{DoCard}([[node]], \gamma_L(arc, \gamma_L(toloc, \gamma_R(toloc))), card), place)$$

  In words:

  1. evaluate the to-locator's XPointer result and process it with the R-directive (γ_R).
  2. process the to-locator's result with the to-locator element (γ_L).
  3. process the previous result with the arc element (γ_L).
  4. apply the cardinality to the node identified by the from-locator *node* and the previous result (DoCard).

     1-1 builds pairs of *node* one node/element body from the previous result (duplicating *node*). 1-n builds one tuple consisting of *node* and the complete previous to side result.

     Allowed combinations: for insert, 1-1 and 1-n (default) is allowed. For replace, only 1-n is allowed.
  5. apply the placement to each pair. The placement operator DoPlacement concatenates each of these pairs (of nodelists). Each pair consists of a list of from-nodes and a list of to-nodes. Let's assume the perspective is forward: then, for each pair, the to-nodes are then (i) made children of, (ii) fused with, or (iii) replacing the from-nodes. In

[2]For reasons of readability, buckets are not considered in the results of γ_L and γ_R as they are given here. For the formal definition of the mapping involving also the bucket results, see Section 6.2.3 later in this chapter.

inverse perspective, the same is done with from and to interchanged. In relation perspective, no placement is done at all.

- For $place$ = fuse, let again $node$ be a single node identified by the from-locator. The processing order is:

$$\gamma_L(arc,\mathsf{DoPlacement}(\mathsf{DoCard}([\mathsf{Body}(node)],\\ \gamma_L(toloc,\gamma_R(toloc)),card),place))$$

  In words:

  1. evaluate the to-locator's XPointer result and apply the R-directive (γ_R).
  2. process the to-locator's result with the to-locator element (γ_L). For fuse, L/R-directives are fixed to drop-element insert-bodies.
  3. apply the cardinality to the from-node's body and the previous result (DoCard). Allowed combinations for fuse: 1-1 and 1-n (default).
  4. implement the fuse placement: process the previous result with the arc element (γ_L), with its L/R-directive set to dup-arc-elem. The effect is, the arc element is wrapped around each pair of bodies (Body($node$),$tolocresultbodies$)).
  5. the original from-node $node$ is replaced with the previous results.

After having the above as a "big picture" in mind, we can plunge into the gory details of the mapping. In the following, the signatures and definitions for $\phi_{(LB)}$ and $\gamma_{(LB)}$ are given for the *forward* perspective: a physical document instance is traversed with $\phi_{(LB)}$. If a traversed node is part of a from-locator referenced by an arc, and if both the arc and the locator are contained in the linkbase LB, then $\gamma_{(LB)}$ is applied to that node, which combines the node with the arc result (using insert, replace or fuse), and blends the combined result back into the instance, replacing the original node.

The *inverse* perspective is completely symmetric to *forward* (with only from- and to-locators interchanged). Thus, it is omitted here.

6.2.2 Signature and Definition of $\phi_{(LB)}$

Signature:

$$\phi_{(LB)} : \mathsf{NODE} \rightarrow \mathsf{NODELIST} \times \mathsf{NODELIST}$$

$$\phi_{(LB)}(node) \rightarrow (newnodelist, bucket)$$

Definition:
$\phi_{(LB)}(node) =?$

- $node$ is a – yet unresolved – Simple Link element:
  $\phi_{(LB)}(node) = (result, bucket)$ with
  $(result, tmpbucket_1) = \phi_{(LB)}(tmpresult)$
  $(tmpresult, tmpbucket_2) = \phi(node)$
  $bucket = tmpbucket_1 \circ tmpbucket_2$.

- *node* is *not* an unresolved Simple Link element AND *node* is contained in the xlink:href XPointer expression of from-locator *loc* (in forward perspective) or to-locator *loc* (in inverse perspective) of at least one arc *arc* in the Extended Link element *extlink* in the linkbase LB[3]:

  $\phi_{(LB)}(node) = (newnodelist, bucket)$
  with $(newnodelist, tmpbucket_1) = \phi^*_{(LB)}(\mathsf{Flat}(nodelistlist))$
  $(nodelistlist, tmpbucket_2) = \gamma_{(LB)}(extlink, arc, loc, node)$
  $bucket = tmpbucket_1 \circ tmpbucket_2$

- else:

  – $node \in \mathsf{ELEM}$:
    $\phi_{(LB)}(node) = ([newnode], bucket)$
    $newnode = \mathsf{Elem}(Name(node), result)$
    $(result, bucket) = \phi^*_{(LB)}(\mathsf{Body}(node))$

  – $node \in \mathsf{TEXT} \cup \mathsf{ATTR}$:
    $\phi_{(LB)}(node) = (node, [])$.

Definition of $\phi^*_{(LB)}$: NODELIST → NODELIST × NODELIST:

$\phi^*_{(LB)}([n_1, \ldots, n_k]) = (result_1 \circ result_{rest}, bucket_1 \circ bucket_{rest})$ with
$(result_1, bucket_1) = \phi_{(LB)}(n_1)$
$(result_{rest}, bucket_{rest}) = \phi^*_{(LB)}([n_2, \ldots, n_k])$

$\phi^*_{(LB)}([]) = ([], [])$.

6.2.3 Signature and Definition of $\gamma_{(LB)}$

Signature:

$$\gamma_{(LB)} : \mathsf{ELEM} \times \mathsf{ELEM} \times \mathsf{NODE} \times \mathsf{STRING} \rightarrow \mathsf{NODELISTLIST} \times \mathsf{NODELIST}$$

$$\gamma_{(LB)}(extlink, arc, node, perspective) \mapsto (nodelistlist, bucket)$$

Definition:
$\gamma_{(LB)}(extlink, arc, node, perspective) = (nodelistlist, bucket)$
with:

- if $perspective = \mathsf{dbxlink{:}forward}$:
  $loc = \mathsf{Loc2Simple}(extlink, arc, \text{"xlink:to"})$

[3]If there are multiple arcs referencing the node, or one single arc referencing the node via multiple locators, the view defined by the linkbase is said to be *concurrent*. For the treatment of concurrent view definitions, please refer to Section 6.3.

- if $perspective = \mathsf{dbxlink{:}inverse}$:
  $loc = \mathsf{Loc2Simple}(extlink, arc, \text{"}\mathsf{xlink{:}from}\text{"})$

$(locresult, locbucket) = \gamma_{LR}(loc)$
$(arcparam, card, place) = \mathsf{GetArcDirectives}(arc)$
$newarc = \mathsf{Arc2Simple}(extlink, arc, perspective)$
$nodelistlist = ?$

- $place \in \{\mathsf{insert}, \mathsf{replace}\} :\Rightarrow$
  - if $node \in \mathsf{TEXT} \cup \mathsf{ATTR} \Rightarrow nodelistlist = [[node]], bucket = []$.
    (you can't *insert* anything into a text or attribute node)
  - if $node \in \mathsf{ELEM}$: $\Rightarrow$
    $nodelistlist =$
    $\mathsf{DoPlacement}(\mathsf{DoCard}([[node]], arcresult, card), place)$ with
    $(arcresult, bucket_1) = \gamma_L(newarc, locresult)$
    $(locresult, bucket_2) = \gamma_{LR}(loc)$
    and
    $bucket = bucket_1 \circ bucket_2$.

    Allowed cardinality directive values: 1-n (default) and 1-1 for $place =$ insert, 1-n (fixed) for $place =$ replace.

- $place = \mathsf{fuse} :\Rightarrow$
  $(nodelistlist, bucket_1) = \phi_{(LB)}(arcresult)$
  with

  $$(arcresult, bucket_2) = \begin{cases} \gamma_L(newarc, placementresult) \text{ if } placementresult \neq [] \\ ([], []) \text{ else} \end{cases}$$

  $placementresult = \mathsf{DoPlacement}(\mathsf{DoCard}([\mathsf{Body}(node)], locresult, card), place)$
  $(locresult, bucket_3) = \gamma_{LR}(loc)$
  and
  $bucket = bucket_1 \circ bucket_2 \circ bucket_3$.

6.3 Three Kinds of Transparency for 3rd Party Links

For querying in presence of 3rd Party Links, it needs to be clarified what "transparent" for linkbases in the implementation context can mean. Consider a locator's or a Simple Link's xlink:href XPointer expression referencing nodes in a (virtual) instance that are "not really there", but blended into the instance by Simple or by 3rd Party Links. Remember that in 5, it was stated that the function EvalXPointer, which serves for evaluating XPointer references in XLink elements, evaluates XPointer/XPath expression with respect to the *virtual instance.* This implies that Simple Links or locators can reference nodes that are not part of the referenced physical instance, but that are blended into the virtual instance by XLink semantics, either by Simple Links or by Extended / 3rd Party Links. The evaluation of an XPath expression needs to be "forwarded" along an XLink reference. With that effect in mind, three different notions, or "levels", of transparency can be distinguished:

Physical Addressing: all from-resources contain nodes in the *physical* data model only. XPath expressions are evaluated directly on the document that the URI – which contains the XPath – specifies. *It is not allowed to address nodes that are mapped into the virtual instance either by Simple Links or by 3rd Party Links.*

Physical addressing is the simplest, least "abstract" form of transparency for 3rd Party Links, and is – usually – the easiest one to implement.

Simple XLink-aware Addressing: An intermediate level of abstraction displays the Simple XLink-aware addressing. Here, from-locators can address nodesets in the virtual data model, where the navigation starts at the starting document, but can follow Simple Link references (and only those) into other documents. *It is not allowed to address nodes that are added to the virtual instance by 3rd Party Links.*

The server's location can be found during evaluation of the locator's XPointer expression, by starting at the entry document and following Simple XLink references out of the entry document. Since an XPath expression on the virtual data model can address nodes in multiple documents (example: the query

/country[@car_code="B" or @car_code="NL"]//city

locates *city* elements from both *cities-B.xml* and *cities-NL.xml*), a locator's result might be located on multiple physical instances on different servers.

Simple and 3rd Party Link-aware Addressing: locators address nodes in the *complete* virtual data model, including Simple XLinks and 3rd Party Links. This implies that one arc can add new nodes to a document by creating a view over it, and another arc can reference portions of these newly added nodes, again modifying or replacing them, and so on (*views over views over views etc.*).

6.3.1 Concurrent View Definition

Within the Simple and 3rd Party Link-aware Addressing, it is possible to define arcs originating on nodes that have been added by other arcs itself: views over views over views can thus be created. Note that there is no natural order given over the arcs of a linkbase. If multiple arcs reference the same node, or a single arc references a node via multiple from-locators (forward case) or multiple to-locators (inverse case), the view definition is called *concurrent*.

Concurrent view definitions are to some degree problematic for the XLink semantics, since there is no notion of an "execution order" among the arcs or locators of a linkbase. If multiple arcs expand or modify the same node, what should be the result? For certain kinds of concurrent view definitions, conflict-free mappings can be thought of. Consider e.g. an element X, with multiple arcs enriching X with "insert" placement: all arcs could add their results to the body of element X inside the virtual instance. But in which document order? Or what, if one of the from-locators' XPointer expression does no more address node X due to the modifications of one of the previous arcs? Even more obvious is the problem in the following constellation: two arcs reference node X as their from side, one with insert placement, one with replace placement. Is node X first modified and then replaced? Or first replaced and then modified? In such situations, multiple ordering of mapping the arcs may result in multiple different results. Which ordering is the right one? And how to find it? Some approaches for finding the right mapping:

- find the "most conservative" mapping: from all combinatorially available mappings, take the mapping wich destroys the fewest information within the virtual instance. E.g. first evaluate arcs with replace semantics, then evaluate arcs with insert semantics, then arcs with fuse semantics, etc.
- Postulate that the appearance of the arcs in the linkbase document determines the order of the mapping: arcs coming first in document order are evaluated first.
- Evaluate the arcs in arbitrary ordering.

The first alternative – and this can already be guessed from the rather imprecise formulation – is not really formally definable, since (1) there is no canonical measure for information within a linkbase (at least, no *trivial* one), and (2) even supposing to have an efficiently computable (or heuristically approximated) canonical measure, many steps would involve problems as query containment, which often cannot be sufficiently solved with algorithmic methods. So, this alternative seems not to be an alternative at all.

The second alternative, taking the document order as mapping order, seems technically feasible, but relatively random, since it seems not quite evident how the rather technical aspect of the position of an arc in the linkbase should have any impact on the algebraic properties of the logical data model.

The third alternative is a generalization of alternative 2, and bears – seeming rather desperate than reasonable – few attraction, for similar reasons as alternative 2.

In absence of an intuitive and consistent interpretation of concurrent (and conflicting) view definitions, it seems reasonable to simply ignore concurrency in linkbases as invalid, leaving their evaluation to the application on a random or heuristic best-effort basis (indeed, in the general case it seems quite non-trivial to decide upon the concurrency of a linkbase, since this involves – among other things – deciding about things like query containment).

Nevertheless it might be useful in certain scenarios to have multiple arcs referencing one node. For *Simple Link and 3rd-Party-Link-aware addressing*, it is possible to explicitly declare a mapping order for the linkbase designer with means of the regular XLink semantics. Consider the following example, using the fuse placement:

Example 7 *Consider a linkbase with an arc* c*, two locators* loc*, from-document* fromdoc.url *and to-document* todoc.url *in* forward *direction, with* fuse *semantics and* 1 : 1 *cardinality:*

```
<linkbase>
   <c dbxlink:transparent="dup-arc-elem place-fuse card-1-1"
      xlink:from="get-x" xlink:to="get-y" strat="1"/>
   <loc xlink:label="get-x" xlink:href="fromdoc.url#xpointer('//x')"/>
   <loc xlink:label="get-y" xlink:href="todoc.url#xpointer('//y')"/>
   ...
</linkbase>
```

```
fromdoc.url                                         todoc.url
   ...                                                 ...
   <x><a/></x>                                         <y><b1/></y>
                                                       <y><b2/></y>
```

The fromdoc.url *instance is expanded with the given arc* c*, producing the cartesian product of the children of nodes* x *(from) and* y *(to):*

```
fromdoc.url                     fromdoc.url
   ...                             ...
<x><a/></x>          ⇒             <c strat="1"><a/><b1/></a>
                                   <c strat="1"><a/><b2/></a>
```

1. *Now, the linkbase is expanded with yet another arc* c *and two more locators (the old, already evaluated parts are displayed in gray):*

```
<linkbase>
   <c dbxlink:transparent="dup-arc-elem place-fuse card-1-1"
      xlink:from="get-x" xlink:to="get-y" strat="1"/>
   <loc xlink:label="get-x" xlink:href="fromdoc.url#xpointer('//x')"/>
   <loc xlink:label="get-y" xlink:href="todoc.url#xpointer('//y')"/>
   <c dbxlink:transparent="dup-arc-elem place-fuse card-1-1"
      xlink:from="get-c1" xlink:to="get-z" strat="2"/>
   <loc xlink:label="get-c1" xlink:href="fromdoc.url#xpointer('//c[@strat="1"]')"/>
   <loc xlink:label="get-z" xlink:href="todoc2.url#xpointer('//z')"/>
   ...
</linkbase>
```

```
fromdoc.url                                     todoc2.url
   ...                                             ...
   <c strat="1"><a/><b1/></c>                      <z><d1/></z>
   <c strat="1"><a/><b2/></c>                      <z><d2/></z>
```

Expanding again fromdoc.url *results in the following virtual instance, producing the cartesian product of the three participating locators' results, the children of* x*,* y*, and* z*:*

```
fromdoc.url
   ...
   <c strat="1"><a/><b1/></c>
   <c strat="1"><a/><b2/></c>
```

⇒

```
fromdoc.url
   ...
   <c strat="2"><a/><b1/><d1/></c>
   <c strat="2"><a/><b1/><d2/></c>
   <c strat="2"><a/><b2/><d1/></c>
   <c strat="2"><a/><b2/><d2/></c>
```

2. *Now, a third arc* c *is added to the linkbase, expanding the given set 3-tuples to a (even larger) set of 4-tuples:*

```
<linkbase>
   <c dbxlink:transparent="dup-arc-elem place-fuse card-1-1"
      xlink:from="get-x" xlink:to="get-y" strat="1"/>
   <loc xlink:label="get-x" xlink:href="fromdoc.url#xpointer('//x')"/>
   <loc xlink:label="get-y" xlink:href="todoc.url#xpointer('//y')"/>
   <c dbxlink:transparent="dup-arc-elem place-fuse card-1-1"
      xlink:from="get-c1" xlink:to="get-z" strat="2"/>
   <loc xlink:label="get-c1" xlink:href="fromdoc.url#xpointer('//c[@strat="1"]')"/>
   <loc xlink:label="get-z" xlink:href="todoc2.url#xpointer('//z')"/>
   <c dbxlink:transparent="dup-arc-elem place-fuse card-1-1"
      xlink:from="get-c2" xlink:to="get-l" strat="3"/>
   <loc xlink:label="get-c2" xlink:href="fromdoc.url#xpointer('//c[@strat="2"]')"/>
   <loc xlink:label="get-l" xlink:href="todoc3.url#xpointer('//l')"/>
   ...
</linkbase>
```

```
fromdoc.url
   <c strat="2"><a/><b1/><d1/></c>
   <c strat="2"><a/><b1/><d2/></c>
   <c strat="2"><a/><b2/><d1/></c>
   <c strat="2"><a/><b2/><d2/></c>
   ...
```

```
todoc3.url
   ...
   <l><k1/></l>
   <l><k2/></l>
```

Expanding again yields:

```
fromdoc.url
   <c strat="2"><a/><b1/><d1/></c>
   <c strat="2"><a/><b1/><d2/></c>
   <c strat="2"><a/><b2/><d1/></c>
   <c strat="2"><a/><b2/><d2/></c>
   ...
```

⇒

```
fromdoc.url
   <c strat="3">
      <a/><b1/><d1/><k1/>
   </c>
   <c strat="3">
      <a/><b1/><d1/><k2/>
   </c>
   <c strat="3">
      <a/><b1/><d2/><k1/>
   </c>
   <c strat="3">
      <a/><b1/><d2/><k2/>
   </c>
   <c strat="3">
      <a/><b2/><d1/><k1/>
   </c>
   <c strat="3">
      <a/><b2/><d1/><k2/>
   </c>
   <c strat="3">
      <a/><b2/><d2/><k1/>
   </c>
   <c strat="3">
      <a/><b2/><d2/><k2/>
   </c>
   ...
```

3. *Now, a finally last arc* x *is added, producing the 5-tuple cartesian product over the sets of elements* {a},{b1/b2}, {d1/d2}, {k1/k2} *and* {r}, *removing the auxiliary* **strat** *attribute and re-installing the initial* x *element surrounding the generated tuples:*

```
<linkbase>
   ...
   <c dbxlink:transparent="dup-arc-elem place-fuse card-1-1"
      xlink:from="get-last" xlink:to="get-f" strat="2"/>
   <loc xlink:label="get-last" xlink:href="fromdoc.url#xpointer('//c[@strat="3"]')"/>
   <loc xlink:label="get-f" xlink:href="todoc4.url#xpointer('//f')"/>
   ...
</linkbase>
```

```
fromdoc.url
   ...
```

⇒

```
fromdoc.url
   <x><a/><b1/><d1/><k1/><r/></x>
   <x><a/><b1/><d1/><k2/><r/></x>
   <x><a/><b1/><d2/><k1/><r/></x>
   <x><a/><b1/><d2/><k2/><r/></x>
   <x><a/><b2/><d1/><k1/><r/></x>
   <x><a/><b2/><d1/><k2/><r/></x>
   <x><a/><b2/><d2/><k1/><r/></x>
   <x><a/><b2/><d2/><k2/><r/></x>
   ...
```

Please note that

- *instead of creating a cartesian product over n sets, it is also possible to model with* $1 : n$ *cardinality,* adding *new nodes to a from-node's body each time. Thereby, it is possible to model a sort of "sum" of nodes instead of a (cartesian) product.*

- *with the* strat *(= stratum) attribute in the arc and in the locator's XPointer, it is possible to explicitly define an "execution order" (or better: mapping priority) over a linkbase's arcs. With this technique, it is possible for the linkbase designer to explicitly define a mapping order over the concurrent arc definitions, based alone on the proposed XLink semantics.*

6.4 The Flight Schedule Example

In the following, three small examples for the three perspectives *relation*, *forward*, *inverse* are given, situated in the flight connection scenario from Example 4 and Example 5. The scenario includes the distributed XML geo-database MONDIAL, containing information about countries and cities, and a linkbase containing information about flights from city to city worldwide; both are given in Figure 6.4. The linkbase contains an Extended Link representing the flight plan, identifying – among others – the cities Wellington (New Zealand) and Singapore (capital of the country Singapore) by locators, and representing the flight connection from Wellington to Singapore with an arc connecting both locators.

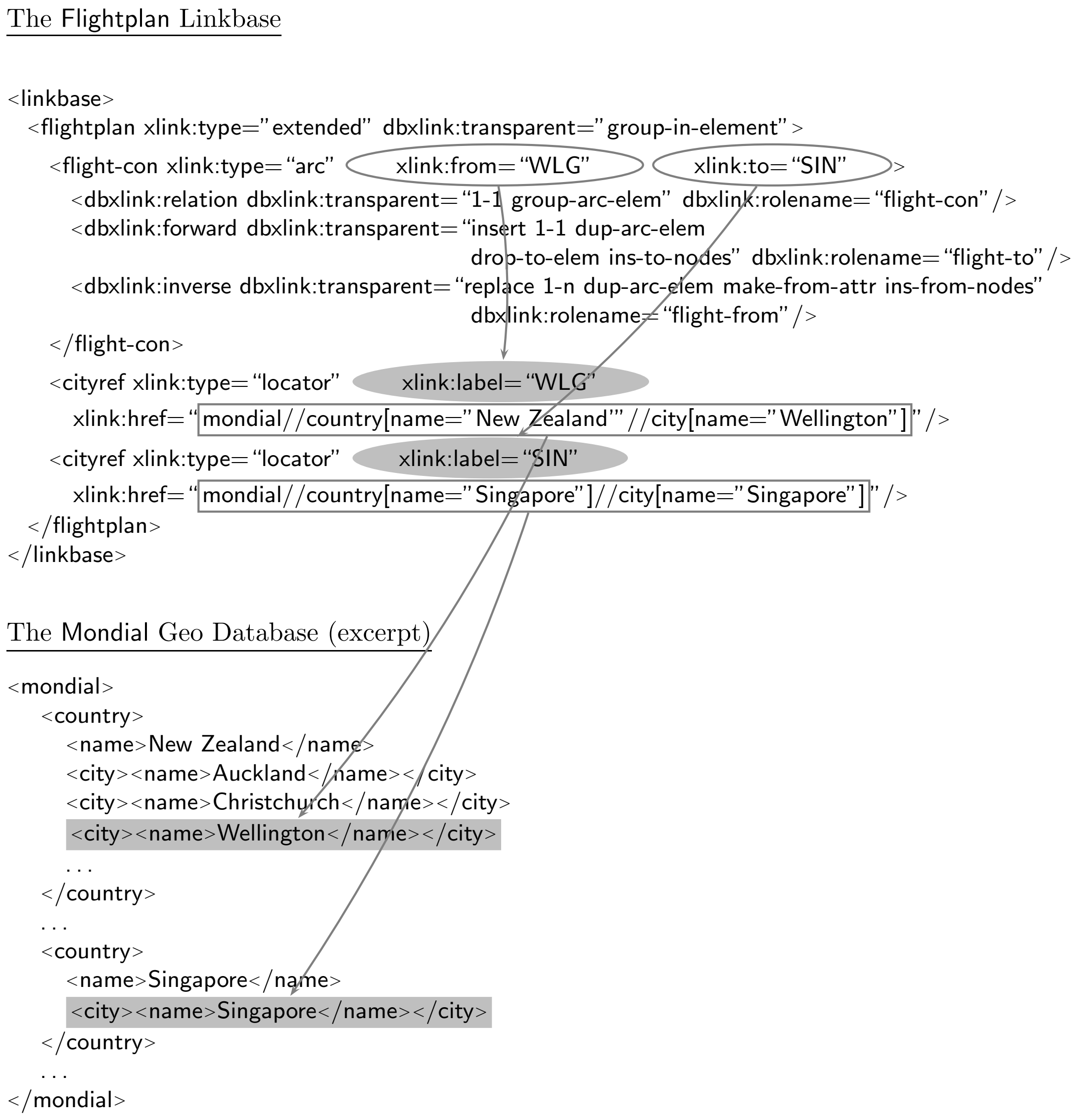
In "relation" perspective, the linkbase itself is queried, connecting and integrating country and city data into the flight plan data. In "forward" and "inverse" perspective, a view over the *Mondial* country and city data is created, modifying the geographical data with flight plan information.

6.4.1 Relation Perspective

The mapping for the relation perspective is started by applying ϕ to the linkbase given in Figure 6.4:

```
$\phi$(Elem("linkbase", [
   Elem("flightplan", [Attr("xlink:type","extended"), Attr("dbxlink:transparent", "group-in-element"),
     Elem("connection",[Attr("xlink:type","arc"), Attr("xlink:from","WLG"), Attr("xlink:to","SIN"),
       Elem("dbxlink:relation", [Attr("dbxlink:transparent","1-1 group-arc-elem"),
         Attr("dbxlink:rolename","flight-con")]),
       Elem("dbxlink:forward", [Attr("dbxlink:transparent","insert 1-1 dup-arc-elem drop-from-elem ins-from-nodes"),
         Attr("dbxlink:rolename","flight-to")]),
       Elem("dbxlink:inverse", [Attr("dbxlink:transparent","replace dup-arc-elem make-from-attr ins-from-nodes
         drop-to-elem ins-to-nodes"),Attr("dbxlink:rolename","flight-from")])]),
     Elem("cityref", [Attr("xlink:type","locator"), Attr("xlink:label","WLG"),
       Attr("xlink:href","//country[name='New Zealand']/city[name='Wellington']")]),
     Elem("cityref", [Attr("xlink:type","locator"), Attr("xlink:label","SIN"),
       Attr("xlink:href","//country[name='Singapore']/city[name='Singapore']")])])])])) =

([Elem("linkbase", result)], bucket) (1)
with
(result, bucket) =
$\phi^*$([ Elem("flightplan",[Elem("connection",[...]),Elem("cityref",[...]),Elem("cityref",[...])])]) =
$\phi$(Elem("flightplan", [Elem("connection",[...]),Elem("cityref",[...]),Elem("cityref",[...])])) =
$\gamma_X$(Elem("flightplan", [Elem("connection",[...]),Elem("cityref",[...]),Elem("cityref",[...])])) =
$\gamma_{ext}$(Elem("flightplan", [...]), $\phi_X$(Elem("flightplan", [...]))) =
```



The Flightplan Linkbase

```
<linkbase>
  <flightplan xlink:type="extended" dbxlink:transparent="group-in-element">
    <flight-con xlink:type="arc"  xlink:from="WLG"  xlink:to="SIN">
      <dbxlink:relation dbxlink:transparent="1-1 group-arc-elem" dbxlink:rolename="flight-con"/>
      <dbxlink:forward dbxlink:transparent="insert 1-1 dup-arc-elem
                                  drop-to-elem ins-to-nodes" dbxlink:rolename="flight-to"/>
      <dbxlink:inverse dbxlink:transparent="replace 1-n dup-arc-elem make-from-attr ins-from-nodes"
                                  dbxlink:rolename="flight-from"/>
    </flight-con>
    <cityref xlink:type="locator"  xlink:label="WLG"
      xlink:href="mondial//country[name="New Zealand"]//city[name="Wellington"]"/>
    <cityref xlink:type="locator"  xlink:label="SIN"
      xlink:href="mondial//country[name="Singapore"]//city[name="Singapore"]"/>
  </flightplan>
</linkbase>
```

The Mondial Geo Database (excerpt)

```
<mondial>
   <country>
      <name>New Zealand</name>
      <city><name>Auckland</name></city>
      <city><name>Christchurch</name></city>
      <city><name>Wellington</name></city>
      . . .
   </country>
   . . .
   <country>
      <name>Singapore</name>
      <city><name>Singapore</name></city>
   </country>
   . . .
</mondial>
```

Figure 6.1: The flightplan example

$$
\begin{aligned}
&\gamma_{ext}(\text{Elem("flightplan", [...]), } \phi^*_X(\text{[],Children(Elem("flightplan", [...])))) } = \\
&\gamma_{ext}(\text{Elem("flightplan", [...]),}\phi^*_X(\text{[],[Elem("connection",[...]), Elem("cityref",[...]),Elem("cityref",[...])])) } = \\
&\gamma_{ext}(\text{Elem("flightplan", [...]),[([],Elem("connection",[...]),[])]} \circ \\
&\qquad\qquad\qquad\qquad \phi^*_X(\text{[], [Elem("cityref",[...]),Elem("cityref",[...])])) } = \\
&\gamma_{ext}(\text{Elem("flightplan", [...]), [([],Elem("connection",[...]),[]),([],Elem("cityref",[...]),[])]} \circ \\
&\qquad\qquad\qquad\qquad \phi^*_X(\text{[], [Elem("cityref",[...])])) } = \\
&\gamma_{ext}(\text{Elem("flightplan", [...]), [([],Elem("connection",[...]),[]),([],Elem("cityref",[...]),[]),}
\end{aligned}
$$

([],Elem("cityref",[...]),[])]) =

(Flat(extresult),extbucket) **(2)**
with
transval = group-in-element:
(extresult,extbucket) =

([[Elem(Name(Elem("flightplan", [...])),Flat(elembody))]],extbucket) **(3)**
with
(elembody,extbucket) = γ^*_{ext}(Elem(flightplan,[..]),Body(Elem(flightplan,[..])) =
γ^*_{ext}(Elem(flightplan,[..]),[Elem("connection",[...]),Elem("cityref",[...]),Elem("cityref",[...])]) =

(result$_1$ $\circ$ result$_{rest}$,bucket$_1$ $\circ$ bucket$_{rest}$) **(4)**
with
(result$_1$, bucket$_1$) = γ_{ext}(Elem(flightplan,[..]),[Elem("connection",[...])) =
γ_{arcloc}(Elem(flightplan,[..]),Elem("connection",[...])) =
(* because connection is an arc *)
γ_{arc}(Elem(flightplan,[..]),Elem("connection",[...])) =
γ_L(Arc2Simple(Elem(connection,[..])),FlatCardList(DoCard(fromresult,toresult))) =
γ_L(Elem(GetRolename(Elem("connection",[...]),"dbxlink:relation"),
        [Attr("xlink:type","simple"), Attr("dbxlink:transparent","group-in-element")]),
    FlatCardList(DoCard(fromresult,toresult))) =

γ_L(Elem("flight-con", [Attr("xlink:type","simple"), Attr("dbxlink:transparent","group-in-element")]),
    FlatCardList(DoCard(fromresult,toresult))) **(5)**
with
(fromresult,frombucket) =
γ^*_{LR}(Loc2Simple(Elem(flightplan,[..]),Elem("connection",[...]),"xlink:from")) =
γ^*_{LR}([Elem(flightplan,[Attr("xlink:type","simple"),Attr("dbxlink:transparent","group-in-element",
        Attr("xlink:href",'//country[name="New Zealand"]/city[name="Wellington"]')])]) =
[γ_L(γ_R(Elem(flightplan,[Attr("xlink:type","simple"),Attr("dbxlink:transparent", "group-in-element",
        Attr("xlink:href",'//country[name="New Zealand"]/city[name="Wellington"]')])))] =
[γ_L([[Elem("city",[Elem("name",[Text("Wellington")])])]])] =
([[Elem("city",[Elem("name",[Text("Wellington")])])]], [])

and
(toresult,tobucket) =
... =
([[Elem("city",[Elem("name",[Text("Singapore")])])]], [])

Insert fromresult, toresult, frombucket,tobucket into **(5)**:
(result$_1$, bucket$_1$) =
γ_L(Elem("flight-con", [Attr("xlink:type","simple"), Attr("dbxlink:transparent","group-in-element")]),
    FlatCardList(DoCard(fromresult,toresult))) =
γ_L(Elem("flight-con", [Attr("xlink:type","simple"), Attr("dbxlink:transparent","group-in-element")]),
    FlatCardList([[Elem("city",[Elem("name",[Text("Wellington")])])]],[Elem("city",[Elem("name",[Text("Singapore")])])]]
γ_L(Elem("flight-con", [Attr("xlink:type","simple"), Attr("dbxlink:transparent","group-in-element")]),
    [[Elem("city",[Elem("name",[Text("Wellington")])])]],[Elem("city",[Elem("name",[Text("Singapore")])])]]]) =
([[Elem("flight-con", [Elem("city",[Elem("name",[Text("Wellington")])]),
        Elem("city",[Elem("name",[Text("Singapore")])])])]], [])

and
(result$_{rest}$, bucket$_{rest}$) = γ^*_{ext}([Elem("cityref",[...]),Elem("cityref",[...])]) =
... (* since default transparent value for locators is "drop-element insert-nothing", thus adding nothing *)
([],[])

Insert result$_1$,result$_{rest}$, bucket$_1$, bucket$_{rest}$ into (**(4)**):

```
(elembody,extbucket) =
([[Elem("flight-con", [
      Elem("city",[Elem("name",[Text("Wellington")])]),
      Elem("city",[Elem("name",[Text("Singapore")])])])]] ∘ [], [] ∘[])=
([[Elem("flight-con", [
     Elem("city",[Elem("name",[Text("Wellington")])]),
     Elem("city",[Elem("name",[Text("Singapore")])])])]], [])
```

Insert elembody,extbucket into ((**3**)):

```
(extresult,extbucket) =
([[Elem(Name(Elem("flightplan", [...])),Flat([[Elem("flight-con", [Elem("city",[Elem("name",[Text("Wellington")])]),
     Elem("city",[Elem("name",[Text("Singapore")])])])]]))]],[]) =
([[Elem("flightplan",[
      Elem("flight-con", [
         Elem("city",[Elem("name",[Text("Wellington")])]),
         Elem("city",[Elem("name",[Text("Singapore")])])])])]],[])
```

Insert extresult, extbucket into ((**2**)):

```
(result, bucket) =
(Flat([[Elem("flightplan", [
             Elem("flight-con",[
                Elem("city",[Elem("name",[Text("Wellington")])]),
                Elem("city",[Elem("name",[Text("Singapore")])])])])]]),[])
= ([Elem("flightplan",[
         Elem("flight-con", [
            Elem("city",[Elem("name",[Text("Wellington")])]),
            Elem("city",[Elem("name",[Text("Singapore")])])])])],[])
```

Insert result, bucket into (**1**):

ϕ(...) = ([Elem("linkbase", result)], bucket) =

```
([Elem("linkbase",[
     Elem("flightplan", [
        Elem("flight-con", [
           Elem("city",[Elem("name",[Text("Wellington")])]),
           Elem("city",[Elem("name",[Text("Singapore")])])])])])],[])
```

---

in XML:

```
<linkbase>
  <flightplan>
    <flight-con>
      <city><name>Wellington</name></city>
      <city><name>Singapore</name></city>
    </flight-con>
  </flightplan>
</linkbase>
```

The bucket remains empty.

6.4.2 Forward **Perspective**

Again, consider the Flightplan Extended Link element from Figure 6.4. Now, the Mondial instance is traversed, expanding/modifying each node that participates the Extended Link. In forward perspective, this is the Wellington city element. In the arc element's dbxlink:transparent attribute, the insert placing denotes that the Singapore city element is – after being processed with the surrounding locator and arc elements – placed inside the Wellington city element as an additional child element.

The Mondial instance is traversed by the $\phi_{(LB)}$ operator, applying the $\gamma(LB)$ operator to elements referenced by arcs/locators in the linkbase LB, the γ operator to Simple Links, and leaving all other nodes unmodified. Starting point is the above Mondial instance in its data model description:

```
φElem("mondial",[
   . . .
   Elem("country",[
     Elem("name",[Text("New Zealand")])
     Elem(" city ",[Elem("name",[Text("Auckland")])])
     Elem("city",[Elem("name",[Text("Christchurch")])])
     Elem("city",[Elem("name",[Text("Wellington")])])
     . . .
   ])
   . . .
   Elem("country",[
     Elem("name",[Text("Singapore")])
     Elem("city",[Elem("name",[Text("Singapore")])])
   ])
   . . .
])
```

Starting with the application of $\phi_{(LB)}$ to the instance (leaving away the unmodified Singapore part, as well as the "..." sections):

```
φ_(LB)(Elem("mondial",[
          Elem("country",[
            Elem("name",[Text("New Zealand")])
            Elem("city ",[Elem("name",[Text("Auckland")])])
            Elem("city",[Elem("name",[Text("Christchurch")])])
            Elem("city",[Elem("name",[Text("Wellington")])])
          ])
        ]) =

φ_(LB)(Elem("mondial",[
          Elem("country",[
          Elem("name",[Text("New Zealand")]),
          Elem("city",[Elem("name",[Text("Auckland")])]),
          Elem("city",[Elem("name",[Text("Christchurch")])]),
          Elem("city",[Elem("name",[Text("Wellington")])])])])) =
```

([Elem("mondial", *mondialbody*)],*mondialbucket*) **(1)**

with
(*mondialbody*,*mondialbucket*) =
$\phi^*_{(LB)}$(Body(Elem("mondial",[...]))) =
$\phi^*_{(LB)}$([Elem("country",[...])]) =
$\phi_{(LB)}$(Elem("country",[
          Elem("name",[Text("New Zealand")]),
          Elem("city",[Elem("name",[Text("Auckland")])]),

Elem("city",[Elem("name",[Text("Christchurch")])]),
Elem("city",[Elem("name",[Text("Wellington")])])])])) =

[Elem("country", $countryresult$)],$countrybucket$) **(2)**

with
($countryresult$,$countrybucket$) =
$\phi^*_{(LB)}$(Body(Elem("country",[. . .]))) =
$\phi^*_{(LB)}$([Elem("name",[Text("New Zealand")]),
Elem("city",[Elem("name",[Text("Auckland")])]),
Elem("city",[Elem("name",[Text("Christchurch")])]),
Elem("city",[Elem("name",[Text("Wellington")])])]) =

$(result_{name} \circ result_{rest1}, bucket_{name} \circ bucket_{rest1})$ **(3)**

with:
$(result_{name}, bucket_{name}) =$
$\phi_{(LB)}$(Elem("name",[Text("New Zealand")])) = ([Elem("name",[Text("New Zealand")])],[])

and
$(result_{rest1}, bucket_{rest1}) =$
$\phi^*_{(LB)}$([Elem("city",[Elem("name",[Text("Auckland")])]),
Elem("city",[Elem("name",[Text("Christchurch")])]),
Elem("city",[Elem("name",[Text("Wellington")])])]) =

$(result_{auck} \circ result_{rest2}, bucket_{auck} \circ bucket_{rest2})$ **(4)**

with:
$(result_{auck}, bucket_{auck}) =$
$\phi_{(LB)}$(Elem("city",[Elem("name",[Text("Auckland")])]) =
. . .
([Elem("city",[Elem("name",[Text("Auckland")])])],[])

and
$(result_{rest2}, bucket_{rest2}) =$
$\phi^*_{(LB)}$([Elem("city",[Elem("name",[Text("Christchurch")])]),
Elem("city",[Elem("name",[Text("Wellington")])])]) =

$(result_{chri} \circ result_{rest3}, bucket_{chri} \circ bucket_{rest3})$ **(5)**

with:
$(result_{chri}, bucket_{chri}) =$
$\phi_{(LB)}$(Elem("city",[Elem("name",[Text("Christchurch")])]) =
. . .
([Elem("city",[Elem("name",[Text("Christchurch")])])],[])

and
$(result_{rest3}, bucket_{rest3}) =$
$\phi^*_{(LB)}$([
Elem("city",[Elem("name",[Text("Wellington")])])]) =

$(result_{well} \circ result_{rest4}, bucket_{well} \circ bucket_{rest4})$ **(6)**

with:
$(result_{well}, bucket_{well}) =$
$\phi_{(LB)}$(Elem("city",[Elem("name",[Text("Wellington")])])) =

(* since the Wellington city element is referenced by a locator: *)
$(newnodelist, bucket)$ **(7)**

with

$(newnodelist,\ tmpbucket_1) = \phi^*_{(LB)}$(Flat($nodelistlist$)) **(8)**

$(nodelistlist, tmpbucket_2) = \gamma_{LB}(extlink,\ arc,\ node,$ "dbxlink:forward") **(9)**

```
extlink = Elem("linkbase",[...])
arc = Elem("connection",[
          Attr("xlink:type","arc"), Attr("xlink:from","WLG"), Attr("xlink:to","SIN"),
          Elem("dbxlink:relation", [...]),
          Elem("dbxlink:forward", [...]),
          Elem("dbxlink:inverse", [...])])
node = Elem("city",[Elem("name",[Text("Wellington")])])
```

Inserted into **(9)**:
$(nodelistlist, tmpbucket_2) = \gamma_{LB}(extlink,\ arc,\ node,$ "dbxlink:forward") =

```
γLB(Elem("linkbase",[...]),
      Elem("connection",[
        Attr("xlink:type","arc"), Attr("xlink:from","WLG"), Attr("xlink:to","SIN"),
        Elem("dbxlink:relation", [...]),
        Elem("dbxlink:forward", [...]),
        Elem("dbxlink:inverse", [...])]),
      Elem("city",[Elem("name",[Text("Wellington")])]), "dbxlink:forward") =
```

$(gammaresult,\ gammabucket)$ **(10)**

with:
$(gammaresult, bucket_1) =$

DoPlacement(DoCard([[Elem("city",[Elem("name",[Text("Wellington")])])]], $arcresult$,"1:1"),"insert") **(11)**

with

$(arcresult,\ bucket_2) = \gamma_L(newarc, toresult)$ **(12)**

with
$newarc$ = Arc2Simple($extlink$,arc,"dbxlink:forward") =

```
Elem("flight-to",[
  Attr("xlink:type","simple"), Attr("dbxlink:transparent","duplicate-element")])
```

and

$(toresult,\ bucket_3) = \gamma_{LR}$*(toloc) **(13)**

with
$toloc$ =
Loc2Simple(Elem("connection",[...]),Elem("cityref",[...],"xlink:to") =

```
[Elem("cityref", [
   Attr("xlink:type","simple"), Attr("xlink:href","//country[name='New Zealand']/city[name='Wellington']"),
   Attr("xlink:dbxlink:transparent","drop-element insert-nodes")])]
```

Inserted into **(13)**:
$(toresult,\ bucket_3) =$
γ_{LR}*([Elem("cityref", [

```
          Attr("xlink:type","simple"), Attr("xlink:href","//country[name='New Zealand']/city[name='Wellington']"),
          Attr("xlink:dbxlink:transparent","drop-element insert-nodes")])]) =
```
γ_{LR}(Elem("cityref", [
```
        Attr("xlink:type","simple"), Attr("xlink:href","//country[name='New Zealand']/city[name='Wellington']"),
        Attr("xlink:dbxlink:transparent","drop-element insert-nodes")])) =
([[Elem("city",[Elem("name",[Text("Wellington")])])]],[])
```

Inserted into **(12)**:
($arcresult$, $bucket_2$) = γ_L($newarc$, $toresult$) =
γ_L(Elem("flight-to",[Attr("xlink:type","simple"),
```
      Attr("dbxlink:transparent","duplicate-element")]),
      [[Elem("city",[Elem("name",[Text("Wellington")])])]]) =
([[Elem("flight-to",[Elem("city",[
    Elem("name",[Text("Wellington")])])])]],[])
```

Inserted into **(11)**:
($gammaresult$,$bucket_1$) =
```
DoPlacement(DoCard([[Elem("city",[Elem("name",[Text("Wellington")])])]],
```
$arcresult$,"1:1"),"insert") =
```
DoPlacement(DoCard([[Elem("city",[Elem("name",[Text("Wellington")])])]],
  [[Elem("flight-to",[Elem("city",[Elem("name",[Text("Singapore")])])])]],"1:1"),"insert") =
DoPlacement(
  [([Elem("city",[Elem("name",[Text("Wellington")])])],
  [Elem("flight-to",[Elem("city",[Elem("name",[Text("Singapore")])])])])],"insert") =
DoPlacementSingle(
  [Elem("city",[Elem("name",[Text("Wellington")])])],
  [Elem("flight-to",[Elem("city",[Elem("name",[Text("Singapore")])])])],"insert") =
DoPlacementSingle(
  [[AddBody(
  Elem("city",[Elem("name",[Text("Wellington")])]),
  [Elem("flight-to",[Elem("city",[Elem("name",[Text("Singapore")])])])])]] =
[[Elem("city",[Elem("name",[Text("Wellington"),
 Elem("flight-to",[Elem("city",[Elem("name",[Text("Singapore")])])])])])]] =
[[Elem("city",[
   Elem("name",[
     Text("Wellington"),
          Elem("flight-to",[
            Elem("city",[
              Elem("name",[
                Text("Singapore")])])])])])])]
```

Inserted into **(10)**:
(nodelistlist,$tmpbucket_2$) = γ_{LB}(arc,loc,$node$,"dbxlink:forward") =
```
([[Elem("city",[
  Elem("name",[
    Text("Wellington"),
      Elem("flight-to",[
        Elem("city",[
          Elem("name",[
            Text("Singapore")])])])])])])],[])
```

Inserted into **(8)**:
($newnodelist$, $tmpbucket_1$) =
$\phi^*_{(LB)}$(Flat([[Elem("city",[
```
                 Elem("name",[
                   Text("Wellington"),
                     Elem("flight-to",[
                    Elem("city",[
                      Elem("name",[
                        Text("Singapore")])])])])])])]])) =
φ*(LB)([Elem("city",[
         Elem("name",[
           Text("Wellington"),
             Elem("flight-to",[
               Elem("city",[
                 Elem("name",[
                   Text("Singapore")])])])])])])]) =
([Elem("city",[
   Elem("name",[
     Text("Wellington"),
             Elem("flight-to",[
               Elem("city",[
                 Elem("name",[
                   Text("Singapore")])])])])])])],[])
```

Inserted into **(7)**:

$(result_{well}$,$bucket_{well}) =$
$\phi^*_{(LB)}$(Flat(
[[Elem("city",[Elem("name",[Text("Wellington"),
 Elem("flight-to",[Elem("city",[Elem("name",[Text("Singapore")])])])])])])]])) =
$\phi^*_{(LB)}$([Elem("city",[Elem("name",[Text("Wellington"),
 Elem("flight-to",[Elem("city",[Elem("name",[Text("Singapore")])])])])])])])) =
([Elem("city",[Elem("name",[Text("Wellington"),
 Elem("flight-to",[Elem("city",[Elem("name",[Text("Singapore")])])])])])])],[])

and
$(result_{chri}$,$bucket_{chri}) =$
([Elem("city",[Elem("name",[Text("Christchurch")])])],[])

and
$(result_{rest3}$,$bucket_{rest3}) =$
$\phi^*_{(LB)}$([Elem("city",[
 Elem("name",[Text("Wellington")]),
 Elem("flight-to",[Elem("city",[Elem("name",[Text("Singapore")])])])])])]) =
$(result_{well} \circ result_{rest4},\ bucket_{well} \circ bucket_{rest4})$

and
$(result_{rest4}$,$bucket_{rest4}) =$
$\phi^*_{(LB)}$([]) = ([],[]).
Inserting into **(6)**:
$(result_{rest3},\ bucket_{rest3}) =$
$(result_{well} \circ result_{rest4},\ bucket_{well} \circ bucket_{rest4}) =$
([Elem("city",[Elem("name",[Text("Wellington")]),
 Elem("flight-to",[Elem("city",[Elem("name",[Text("Singapore")])])])])] ∘ [],[] ∘ []) =
([Elem("city",[
 Elem("name",[Text("Wellington")]),
 Elem("flight-to",[Elem("city",[Elem("name",[Text("Singapore")])])])])], [])
Inserting into **(5)**:
$(result_{rest2},\ bucket_{rest2}) =$
$(result_{chri} \circ result_{rest3},\ bucket_{chri} \circ bucket_{rest3)} =$
([Elem("city",[Elem("name",[Text("Christchurch")])])] ∘
 [Elem("city",[Elem("name",[Text("Wellington")])])] ∘ [], [] ∘ []) =
([Elem("city",[Elem("name",[Text("Christchurch")])]),

```
  Elem("city",[
     Elem("name",[Text("Wellington")]),
     Elem("flight-to",[Elem("city",[Elem("name",[Text("Singapore")])])])])], []).
```
Inserting into **(4)**:
$(result_{rest1},\ bucket_{rest1}) =$
$(result_{auck} \circ result_{rest2},\ bucket_{auck} \circ bucket_{rest2}) =$
```
([Elem("city",[Elem("name",[Text("Auckland")])])] ∘
 [Elem("city",[Elem("name",[Text("Christchurch")])]),
 Elem("city",[
    Elem("name",[Text("Wellington")]),
    Elem("flight-to",[Elem("city",[Elem("name",[Text("Singapore")])])])])],
 [] ∘ []) =
([Elem("city",[Elem("name",[Text("Auckland")])]), Elem("city",[Elem("name",[Text("Christchurch")])]),
  Elem("city",[
     Elem("name",[Text("Wellington")]),
     Elem("flight-to",[Elem("city",[Elem("name",[Text("Singapore")])])])])], []).
```
Inserting into **(3)**:
$(countryresult,countrybucket) =$
$(result_{name} \circ result_{rest1},\ bucket_{name} \circ bucket_{rest1})$ **(3)**
```
([Elem("name",[Text("New Zealand")])] ∘
 [Elem("city",[Elem("name",[Text("Auckland")])]),
  Elem("city",[Elem("name",[Text("Christchurch")])]),
  Elem("city",[
     Elem("name",[Text("Wellington")]),
     Elem("flight-to",[Elem("city",[Elem("name",[Text("Singapore")])])])])], [] ∘ []) =
([Elem("name",[Text("New Zealand")]),
  Elem("city",[Elem("name",[Text("Auckland")])]),
  Elem("city",[Elem("name",[Text("Christchurch")])]),
  Elem("city",[
     Elem("name",[Text("Wellington")]),
     Elem("flight-to",[Elem("city",[Elem("name",[Text("Singapore")])])])])], []).
```
Inserting into **(2)**:
$(mondialbody,mondialbucket) =$
([Elem("country", $countryresult$)],$countrybucket$) =
```
([Elem("country", [
     Elem("name",[Text("New Zealand")]),
     Elem("city",[Elem("name",[Text("Auckland")])]),
     Elem("city",[Elem("name",[Text("Christchurch")])]),
     Elem("city",[
        Elem("name",[Text("Wellington")]),
        Elem("flight-to",[Elem("city",[Elem("name",[Text("Singapore")])])])])])],[])
```

Inserted into **(1)**:
$\phi_{(LB)}$(Elem("mondial",[...])) =
([Elem("mondial", $mondialbody$)],$mondialbucket$) =
```
([Elem("mondial", [
     Elem("country", [
        Elem("name",[Text("New Zealand")]),
        Elem("city",[Elem("name",[Text("Auckland")])]),
        Elem("city",[Elem("name",[Text("Christchurch")])]),
        Elem("city",[
           Elem("name",[Text("Wellington")]),
           Elem("flight-to",[Elem("city",[Elem("name",[Text("Singapore")])])])])])]),[]).
```

in XML:

```
<mondial>
   <country>
      <name>New Zealand</name>
      <city><name>Auckland</name></city>
      <city><name>Christchurch</name></city>
      <city>
         <name>Wellington</name>
         <flight-to>      <!-- newly inserted surrounding arc element -->
            <city><name>Singapore</name></city>      <!-- inserted to-nodes -->
         </flight-to>
      </city>
   </country>
</mondial>
```

The bucket remains empty.

6.4.3 Inverse Perspective

Again, consider the Flightplan Extended Link element from Figure 6.4. Same as for the forward perspective, the Mondial database is traversed, expanding/modifying each node that participates in the Extended Link (Wellington and Singapore city elements).

Here, the given directives are replace, which means that the Singapore city node is replaced with the processed arc origin. dup-arc-elem denotes that the arc's origin (the Wellington city element) is kept, with the arc element wrapped around the – single – origin element Wellington. make-from-attr determines the from-locator element being turned into an IDREFS attribute referencing the origin. ins-from-nodes determines the Wellington element to remain unchanged (so the IDREFS attribute contains one reference to the Wellington city element).

As within forward perspective, the Mondial instance is traversed by the $\phi_{(LB)}$ operator, applying the $\gamma(LB)$ operator to Extended Links (and its components).

```
φ(LB)(Elem("mondial",[
        Elem("country",[Elem("name",[Text("New Zealand")],...)]),
      ...
        Elem("country",[
          Elem("name",[Text("Singapore")]),
          Elem("city",[Elem("name",[Text("Singapore")])])])
        ])
      ...
      ]) =

(mondialresult,mondialbucket) =
φ(LB)(Elem("mondial",[
        Elem("country",[Elem("name",[Text("New Zealand")],...)]),
      ...
        Elem("country",[
          Elem("name",[Text("Singapore")]),
```

Elem("city",[Elem("name",[Text("Singapore")])])
])
. . .
]) =

(dropping the New Zealand and the [. . .] parts)

$\phi_{(LB)}$(Elem("mondial",[
Elem("country",[
Elem("name",[Text("Singapore")]),
Elem("city",[Elem("name",[Text("Singapore")])])
])
]) =
([Elem(Name(Elem("mondial",[. . .])),*mondialinnerresult*)], *mondialinnerbucket*) =
([Elem("mondial", *mondialinnerresult*)],*mondialinnerbucket*) **(1)**

with:
(*mondialinnerresult*,*mondialinnerbucket*) =$\phi^*_{(LB)}$(Body(Elem("mondial",[. . .]))) =
$\phi^*_{(LB)}$([Elem("country",[
Elem("name",[Text("Singapore")]),
Elem("city",[Elem("name",[Text("Singapore")])])])]) =
($countryresult \circ result_{rest}$,$countrybucket \circ bucket_{rest}$) **(2)**

with:
(*countryresult*, *countrybucket*) =
$\phi_{(LB)}$(Elem("country",[
Elem("name",[Text("Singapore")]),
Elem("city",[Elem("name",[Text("Singapore")])])])) =
([Elem(Name(Elem("country",[. . .])), *countryinnerresult*)],*countryinnerbucket*) **(3)**

with:
(*countryinnerresult*,*countryinnerbucket*) =
$\phi^*_{(LB)}$([Elem("name",[Text("Singapore")]), Elem("city",[Elem("name",[Text("Singapore")])])]) =
($nameresult \circ cityresult$, $namebucket \circ citybucket$) **(4)**

with:
(*nameresult*,*namebucket*) = $\phi_{(LB)}$(Elem("name",[Text("Singapore")])) =
([Elem("name",[Text("Singapore")])],[]) **(5)**

and:
(*cityresult*,*citybucket*) =
$\phi_{(LB)}$(Elem("city",[Elem("name",[Text("Singapore")])])) =
(* since the Singapore city element is touched by to-locator cityref *)
($gammaresultflat$,$gammabucket_1 \circ gammabucket_2$) **(6)**

with:
($gammaresultflat$,$gammabucket_1$) = $\phi^*_{(LB)}$(Flat(*gammaresult*))
($gammaresult$,$gammabucket_2$) = $\gamma_{(LB)}$(*extlink*,*arc*,*node*,"dbxlink:inverse") **(7)**
extlink = Elem("linkbase",[. . .])
arc = Elem("connection",[
Attr("xlink:type","arc"), Attr("xlink:from","WLG"), Attr("xlink:to","SIN"),
Elem("dbxlink:relation", [. . .]),
Elem("dbxlink:forward", [. . .]),
Elem("dbxlink:inverse", [. . .])])
node = Elem("city",[Elem("name",[Text("Singapore")])])

```
Inserted into (7):
($gammaresult$,$gammabucket_2$) =
$\gamma_{(LB)}$(Elem("linkbase",[. . . ]),
         Elem("connection",[
            Attr("xlink:type","arc"), Attr("xlink:from","WLG"),Attr("xlink:to","SIN"),
            Elem("dbxlink:relation", [. . . ]),
            Elem("dbxlink:forward", [. . . ]),
            Elem("dbxlink:inverse", [. . . ])]),
            Elem("city",[Elem("name",[Text("Singapore")])]),
         "dbxlink:inverse") =

(DoPlacement(
     DoCard(
          [[Elem("city",[Elem("name",[Text("Singapore")])])]],
          $arcresult$,"1-n"),
     "replace"),
     $arcbucket \circ locbucket$) (8)

with:
($arcresult$,$arcbucket$) =
$\gamma_L$(Arc2Simple($extlink$,$arc$,"dbxlink:inverse"),$locresult$) (9)

with:
loc = Loc2Simple($extlink$, $arc$,"xlink:from") =
Elem("cityref",[
   Attr("xlink:type","simple"),
   Attr("xlink:href","//country[name='New Zealand']/city[name='Wellington']"),
   Attr("dbxlink:transparent","make-attribute insert-nodes")])

and:
($locresult$, $locbucket$) = $\gamma_{LR}(loc)$ =
$\gamma_{LR}$(Elem("cityref",[
          Attr("xlink:type","simple"),
          Attr("xlink:href","//country[name='New Zealand']/city[name='Wellington']"),
          Attr("dbxlink:transparent","make-attribute insert-nodes")])) =
(
 [[Attr("cityref","id0001")]],                        (* locresult *)
 [Elem("city",[Attr("dbxlink:id","id0001"),Elem("name,[Text("Wellington")])])] (* locbucket *)
)

Inserted into (9):
($arcresult$,$arcbucket$) =
$\gamma_L$(Arc2Simple($extlink$,$arc$,"dbxlink:inverse"),$locresult$) =
$\gamma_L$(Arc2Simple(
       Elem("linkbase",[. . . ]),
       Elem("connection",[
         Attr("xlink:type","arc"), Attr("xlink:from","WLG"), Attr("xlink:to","SIN"),
         Elem("dbxlink:relation", [. . . ]),
         Elem("dbxlink:forward", [. . . ]),
         Elem("dbxlink:inverse", [. . . ])]),
       "dbxlink:inverse"),
       [[Attr("cityref","id0001")]]) =

$\gamma_L$(Elem("flight-from",[
        Attr("xlink:type","simple"), Attr("dbxlink:transparent","duplicate-element")]),
     [[Attr("cityref","id0001")]]) =
```

([[Elem("flight-from",[Attr("cityref","id0001")])]],[]).

Inserted into **(8)**:
($gammaresult$,$gammabucket_2$) =
(DoPlacement(
    DoCard([[Elem("city",[
        Elem("name",[Text("Singapore")])])]], $arcresult$,"1-n"),"replace"),
        $arcbucket \circ locbucket$) =

(DoPlacement(
    DoCard([[Elem("city",[Elem("name",[Text("Singapore")])])]],
            [[Elem("flight-from",[Attr("cityref","id0001")])]],
            "1-n"),
    "replace"),
    [] ∘ [Elem("city",[Attr("dbxlink:id","id0001"),Elem("name,[Text("Wellington")])])]) =

(DoPlacement(
    [([[Elem("city",[Elem("name",[Text("Singapore")])])]],
    [[Elem("flight-from",[Attr("cityref","id0001")])]])],
    "replace"),
 [Elem("city",[Attr("dbxlink:id","id0001"),Elem("name,[Text("Wellington")])])]) =

([[Elem("flight-from",[Attr("cityref","id0001")])]],
 [Elem("city",[Attr("dbxlink:id","id0001"),Elem("name,[Text("Wellington")])])])

Inserted into **(6)**:
($cityresult$,$citybucket$) =
$\phi_{(LB)}$(Elem("city",[Elem("name",[Text("Singapore")])])) =
($gammaresultflat$,$gammabucket_1 \circ gammabucket_2$)

with:
($gammaresultflat$,$gammabucket_1$) = $\phi^*_{(LB)}$(Flat($gammaresult$)) =
$\phi^*_{(LB)}$(Flat([[Elem("flight-from",[Attr("cityref","id0001")])]])) =
$\phi^*_{(LB)}$([Elem("flight-from",[Attr("cityref","id0001")])]]) =
([Elem("flight-from",[Attr("cityref","id0001")])]],[]).

Back to **(6)**:
($cityresult$,$citybucket$) =
$\phi_{(LB)}$(Elem("city",[Elem("name",[Text("Singapore")])])) =
($gammaresultflat$,$gammabucket_1 \circ gammabucket_2$) =
([Elem("flight-from",[Attr("cityref","id0001")])],
[Elem("city",[Attr("dbxlink:id","id0001"),Elem("name,[Text("Wellington")])])])

Inserted into **(4)**:
($countryinnerresult$,$countryinnerbucket$) =
($nameresult \circ cityresult$, $namebucket \circ citybucket$) =
($nameresult$ ∘ [Elem("flight-from",[Attr("cityref","id0001")])],
 $namebucket$ ∘ [Elem("city",[Attr("dbxlink:id","id0001"),Elem("name,[Text("Wellington")])])]) =

(now, inserting **(5)** here:)
([Elem("name",[Text("Singapore")])] ∘ [Elem("flight-from",[Attr("cityref","id0001")])],
 [] ∘ [Elem("city",[Attr("dbxlink:id","id0001"),Elem("name,[Text("Wellington")])])]) =

([Elem("name",[Text("Singapore")]),Elem("flight-from",[Attr("cityref","id0001")])],
 [Elem("city",[Attr("dbxlink:id","id0001"),Elem("name,[Text("Wellington")])])])

Inserted into **(3)**:

$(countryresult,\ countrybucket) =$

([Elem("country",[Elem("name",[Text("Singapore")]), Elem("flight-from",[Attr("cityref","id0001")])])],
[Elem("city",[Attr("dbxlink:id","id0001"),Elem("name,[Text("Wellington")])])])

Inserted into **(2)**:
$(mondialinnerresult, mondialinnerbucket) =$

$(countryresult \circ result_{rest1}, countrybucket \circ bucket_{rest1}) =$

(* $result_{rest}$ and $bucket_{rest}$ being [] each *)

```
([Elem("country",[
    Elem("name",[Text("Singapore")]),
    Elem("flight-from",[Attr("cityref","id0001")])])] ∘ [],
 [Elem("city",[Attr("dbxlink:id","id0001"),Elem("name,[Text("Wellington")])])] ∘ []) =
```

```
([Elem("country",[
    Elem("name",[Text("Singapore")]),
    Elem("flight-from",[Attr("cityref","id0001")])])],
    [Elem("city",[Attr("dbxlink:id","id0001"),Elem("name,[Text("Wellington")])])])
```

Inserted into **(1)**:
$(mondialresult, mondialbucket) =$

([Elem(Name(Elem("mondial",[. . .])),$mondialinnerresult$)], $mondialinnerbucket$) =

([Elem("mondial", $mondialinnerresult$)],$mondialinnerbucket$)

```
([Elem("mondial",[
 Elem("country",[
    Elem("name",[Text("Singapore")]),
    Elem("flight-from",[Attr("cityref","id0001")])])])],
 [Elem("city",[Attr("dbxlink:id","id0001"),Elem("name,[Text("Wellington")])])])
```

which is equivalent to

```
<mondial>
    ...
    <country>
       <name>Singapore</name>
       <flight-from cityref="id0001"/>      <!-- former Singapore city element, now replaced -->
                                           <!-- by reference to Wellington city element -->
    </country>
    ...
</mondial>
```

and the bucket containing

```
<city dbxlink:id="id0001">
   <name>Wellington</name>
</city>
```

.

Chapter 7

Algorithms for Query Evaluation with 3rd Party Links

7.1 Evaluating 3rd Party Links in Distributed XML Environments

In s 5 and 6, a logical data model for querying distributed XML data in presence of Simple Links and Extended Links has been formally defined. With that, a formal foundation is provided to explore the algorithmic side of querying XLinked data. In [Beh06], algorithmic and implementation aspects of Simple Links have been investigated, including a detailed investigation of the issues of Query Shipping / Data Shipping. Using this knowledge about Simple Links as basis, this chapter focuses on methods, techniques and infrastructure for evaluating XPath queries in presence of 3rd Party Links, within a distributed XLink-aware XML environment.

Since the relevant perspectives for 3rd Party Links are forward and inverse, and since both are symmetric to each other, only the forward perspective is covered here.

The evaluation techniques given in this chapter are applied in the context of a technical infrastructure, which is described first. The proposed infrastructure model is an abstraction of the XLink-aware XML server infrastructure as it was developed throughout the LinXIS [Lin] project.

7.1.1 Server Infrastructure

The chosen scenario for demonstrating the XLink features of the prototype consists of a number of XLink-aware eXist servers. Each server contains XML data, which again may contain XLink references to data on other XML servers.

A server can be abstractly seen as a repository of XML files with additional Web server functionality – supplying data on http requests, requesting data from remote resources etc. – and with XPath support. XPath expressions can be submitted (via http) to an XML server, and are evaluated on the *virtual instance*, induced by the XLink-connected instances. In that way, XPath expressions may be evaluated across several linked XML instances on multiple hosts.

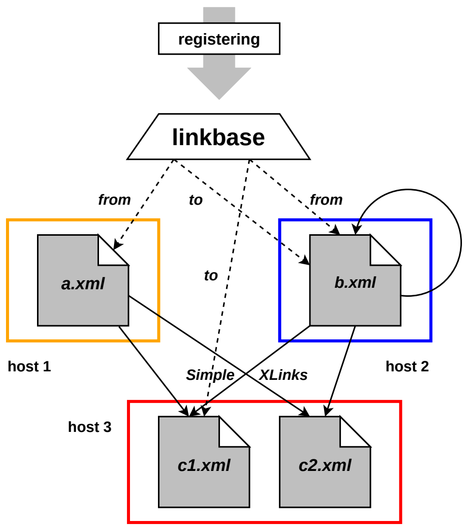
Note that since both, XSLT and XQuery are based on XPath, XSL transformations as well as the unmodified XQuery expressions can be applied to the virtual instance without modification. In the LinXIS implementation, the modified eXist servers execute XQuery expressions on the virtual instance. The general concept of XPath evaluation is based on the following notion:

- Each query – at least basically – consists of a number of location steps.
- there exists a *context*, which serves as the input for the next location step, which again produces a new context as output, which is passed on to the next location step, and so on.
- When the last location step is processed, the resulting context is the processed result of the expression.
- The initial context is empty (note that a leading "/" identifies a document's root node).

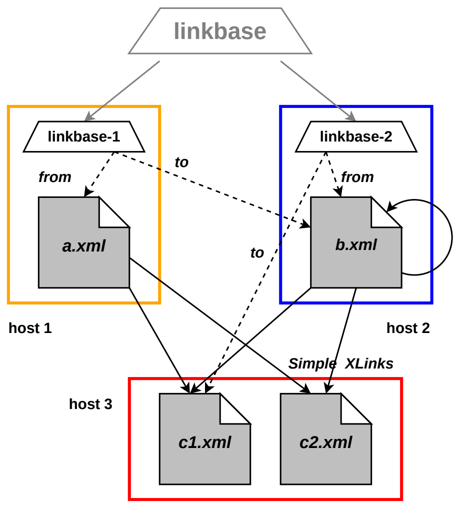
A location step consists of three items: an axis identifier, a node test, and zero or more predicates. For each location step, and for each node in the context, the set of those nodes is computed that can be reached from the node by the given axis. The node test is applied to each of the result nodes, and – if present – the predicates are evaluated, which can reduce the number of result nodes. Then, the current node is removed from the context and replaced by the computed nodes. This is performed for each location step, where one step's result is the next step's input context. The obtained result is the last location step's result.

In the described XLink setting, two cases deserve special treatment during evaluation of an XPath query:

- The evaluation passes a Simple Link element:
  each time an axis result set is computed, it has to be considered if the nodeset can contain nodes generated by an XLink. So, it must be checked if the axis result set contains XLink elements, or contains nodes with XLink element children which might produce XLink results to be added to the axis result set.
- The evaluation passes nodes that are part of a resource referenced by an arc's from-locator:
  In that case, the arc's result is processed by first combining the to-resource with the arc information, according to the modeling defined in the arc's dbxlink:transparent attribute (*arc-L-dir*, *to-L-dir*, *to-R-dir*). Then, this



(a) registering and distributing a linkbase



(b) with distributed linkbase indexes *linkbase-1* and *linkbase-2*

Figure 7.1: Registering and distributing linkbase information

intermediate result is combined with the currently traversed from-nodes, again according to the keywords in dbxlink:transparent (cardinality and placement directives).

In most cases (depending on the given mapping directives), the given link semantics can be emulated by rewriting the current from-nodes, replacing or enriching them with Simple Link structures (see Function 7.7). In the few cases where the 3rd Party Link cannot be "compiled down" to Simple Links, it is computed directly, but still reusing the already given Simple Link processing logic described by ϕ, γ etc.

After that, the regular processing continues.

In the following, the basic concepts for implementing these notions in the given infrastructure scenario are discussed, along with application perspectives, as well as bringing the attention to some pitfalls raising their ugly heads.

7.2 Implementing Transparency

7.2.1 Outline: Three Steps

The basic procedure for using 3rd Party Links in the DBXLink scenario includes the following steps:

1. Registering a linkbase by introducing it to all affected servers. These are those servers that contain XML data which is referenced by locators from

Links inside the linkbase.

2. The linkbase data is distributed among the servers. An arc with a locator referencing data on a server *a* will be stored on the server *a*. The created sub-linkbases are termed *linkbase indexes*. Each linkbase index contains entries for arcs originating on the server where the index is located (an arc originates at the nodes located by its from-locator). This is relevant for evaluation wrt. forward perspective. Each linkbase index also contains entries for arcs with their destinations on the server where the index is maintained. Hence, this is of relevance for evaluation in inverse perspective.

3. When an XPath query is issued to the whole virtual data model, each traversed node is checked whether it is affected by the server's linkbase index, means: in each location step evaluation, it is checked whether an arc from the local linkbase index could influence the traversed nodes, by adding, enriching or modifying the current context by referenced nodes. If so, the evaluation might be spread over the 3rd-party-linked nodes on remote servers.

Level of Transparency

Note that the three steps which are given above, and which are depicted in detail below, are only applicable for *Physical Addressing* and *Simple-Link-aware Addressing*. Step 1, the registering, and step 2, the distribution of a linkbase are a form of preprocessing, resulting in having split up a linkbase in several parts on several hosts. Which part is sent to which host depends on the location of the referenced nodes: an arc that references nodes on a host X with its from-locator will be sent to that host X. If the node is either referenced directly on physical level, or if the XPointer expression is forwarded to host X during registry, then the physical location of that node can be determined. But considering *3rd-Party-Link-aware addressing*, this technique meets its limits: for physical or Simple-Link-aware addressing, it is relatively easy to find out if a node is influenced by an arc: simply evaluate the from-locator's XPath expression, and potentially follow one (or more) Simple Link reference(s). At some point, a node can be obtained which is part of the from-resource, together with the node's physical document location. For 3rd-party-link-aware addressing, it would be necessary *for each traversed node* to find out if it is influenced by any of the linkbase's arcs, which might add evaluation-relevant nodes to the virtual instance as it is known by now. If so, the nodes blended into the virtual instance by the arc have to be checked as well if they in turn are touched by some arc, and so on. Computationally, this can be reduced to the problem of finding a transitive closure over 3rd Party mapping rules, which seems theoretically feasible, but prohibitively expensive regarding the runtime. Moreover, it would not be compliant with the design decision to distribute a linkbase's parts to the server that hosts the document bearing the influenced node, since this information cannot be obtained a priori, but only during the XPath evaluation, which makes the precomputing / registering of linkbase information impossible. Due to this, the prototype described here supports only "level 2" transparency, allowing linkbases to have from-locators pointing to nodes via a "Simple Link *detour*".

7.2.2 Creating and Using Linkbase Indexes

In the following, the 3 steps – registering a linkbase, distributing a linkbase, and evaluation XPath expressions wrt. the linkbase – are described in detail, with an algorithmic notation following later.

Registering a Linkbase.

A linkbase is registered at some server in the network. For each arc in the linkbase, it is determined where the arc's locators are pointing to. Then, the arc is transmitted to the servers that host the nodes identified by the locators. Hence, the registered linkbase is split up and distributed among the servers containing nodes referenced by the arc's locators.

Building an Index.

Each server may receive a number of arcs with their locators pointing into one of the server's hosted documents. The arcs and locators are locally registered in the index. The locator results are precomputed: nodes in one of the database's local documents that are identified by one of the received locators are associated with the index's representation of the locator. See Figure 7.2 for an example index entry.

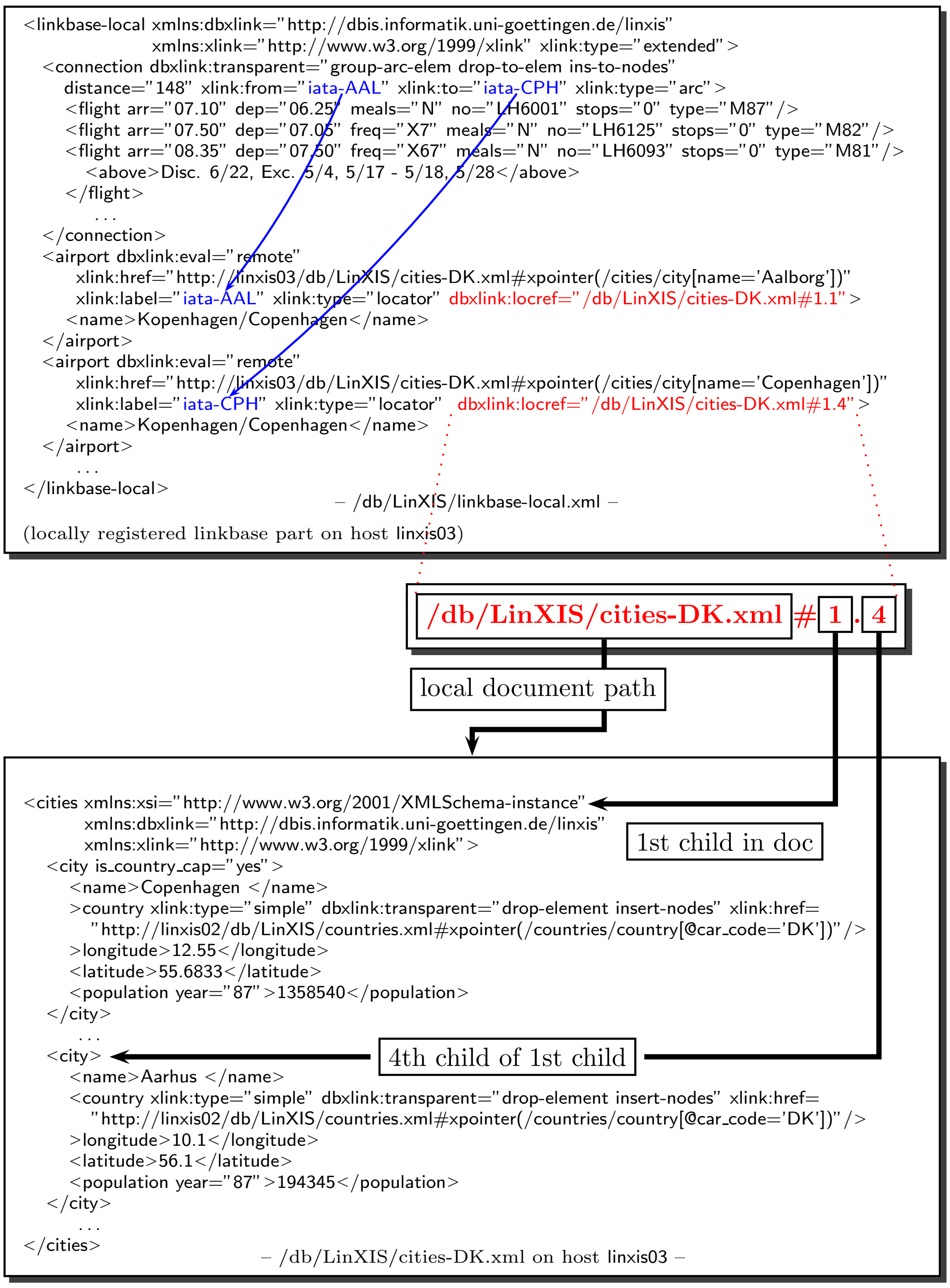
XPath Evaluation in Presence of a Linkbase.

An XPath query is evaluated on a server, with each location step producing an intermediate context. If such a context contains a node influenced by an arc's locator (the query *traverses* that node), the node is modified / replaced by the arc's result, the new result nodes are included in the intermediate context, and the next location step is evaluated with the modified context as input.

Example.

In Figure 7.3, an example is given for a number of linked XML documents on multiple hosts, and a linkbase with physical addressing. host0 hosts the document countries.xml containing data about all countries. host1 hosts the document cities-NZ.xml containing data about New Zealand's cities. host2 hosts the document cities-SGP.xml containing data about Singapore's cities (which is only Singapore itself). Consider now the query

```
/countries/country[@car_code="NZ"]//city[name="Wellington"]/flight-to/city/name
```



```
<linkbase-local xmlns:dbxlink="http://dbis.informatik.uni-goettingen.de/linxis"
                xmlns:xlink="http://www.w3.org/1999/xlink" xlink:type="extended">
  <connection dbxlink:transparent="group-arc-elem drop-to-elem ins-to-nodes"
    distance="148" xlink:from="iata-AAL" xlink:to="iata-CPH" xlink:type="arc">
    <flight arr="07.10" dep="06.25" meals="N" no="LH6001" stops="0" type="M87"/>
    <flight arr="07.50" dep="07.05" freq="X7" meals="N" no="LH6125" stops="0" type="M82"/>
    <flight arr="08.35" dep="07.50" freq="X67" meals="N" no="LH6093" stops="0" type="M81"/>
      <above>Disc. 6/22, Exc. 5/4, 5/17 - 5/18, 5/28</above>
    </flight>
      ...
  </connection>
  <airport dbxlink:eval="remote"
      xlink:href="http://linxis03/db/LinXIS/cities-DK.xml#xpointer(/cities/city[name='Aalborg'])"
      xlink:label="iata-AAL" xlink:type="locator" dbxlink:locref="/db/LinXIS/cities-DK.xml#1.1">
    <name>Kopenhagen/Copenhagen</name>
  </airport>
  <airport dbxlink:eval="remote"
      xlink:href="http://linxis03/db/LinXIS/cities-DK.xml#xpointer(/cities/city[name='Copenhagen'])"
      xlink:label="iata-CPH" xlink:type="locator" dbxlink:locref="/db/LinXIS/cities-DK.xml#1.4">
    <name>Kopenhagen/Copenhagen</name>
  </airport>
      ...
</linkbase-local>
```

– /db/LinXIS/linkbase-local.xml –

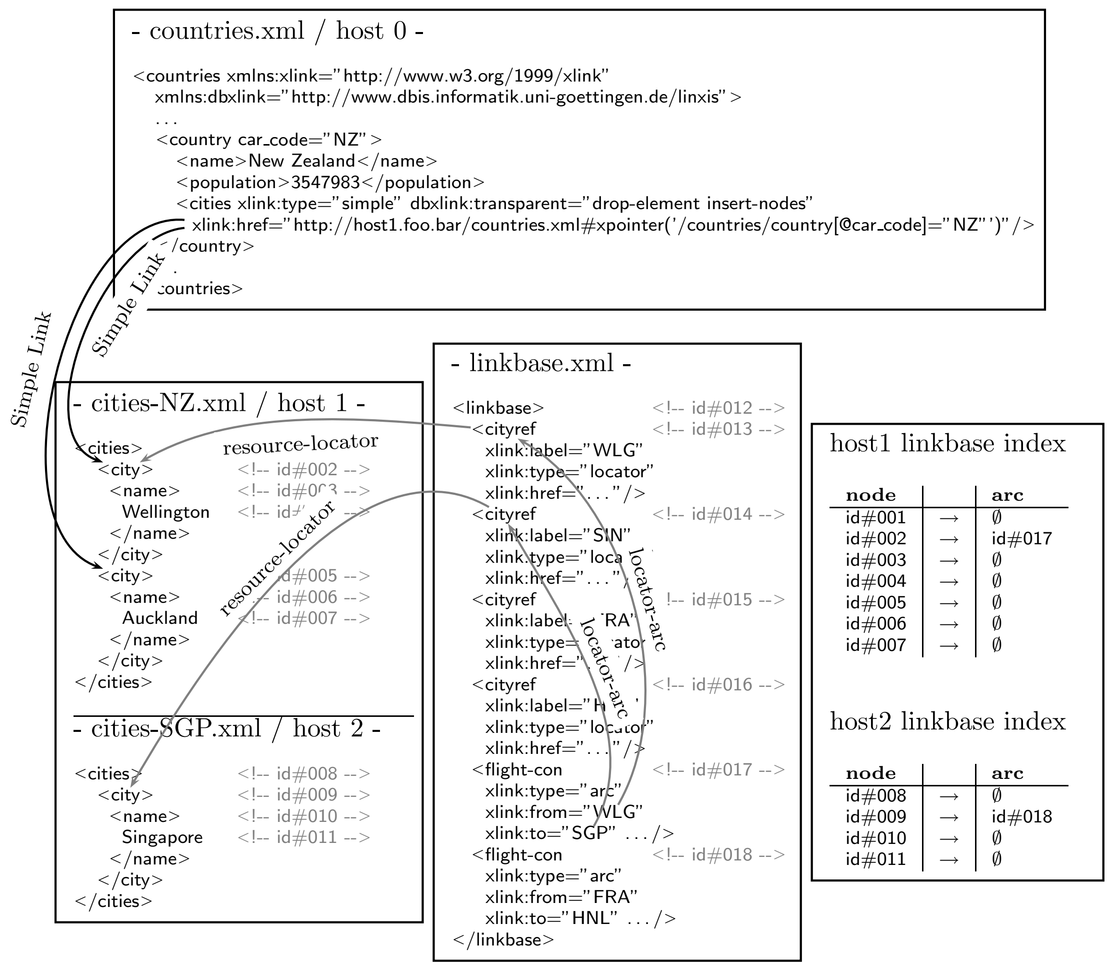
(locally registered linkbase part on host linxis03)

```
<cities xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:dbxlink="http://dbis.informatik.uni-goettingen.de/linxis"
        xmlns:xlink="http://www.w3.org/1999/xlink">
  <city is_country_cap="yes">
    <name>Copenhagen </name>
    >country xlink:type="simple" dbxlink:transparent="drop-element insert-nodes" xlink:href=
      "http://linxis02/db/LinXIS/countries.xml#xpointer(/countries/country[@car_code='DK'])"/>
    >longitude>12.55</longitude>
    <latitude>55.6833</latitude>
    <population year="87">1358540</population>
  </city>
    ...
  <city>
    <name>Aarhus </name>
    <country xlink:type="simple" dbxlink:transparent="drop-element insert-nodes" xlink:href=
      "http://linxis02/db/LinXIS/countries.xml#xpointer(/countries/country[@car_code='DK'])"/>
    >longitude>10.1</longitude>
    <latitude>56.1</latitude>
    <population year="87">194345</population>
  </city>
    ...
</cities>
```

– /db/LinXIS/cities-DK.xml on host linxis03 –

Figure 7.2: Excerpt of a registered local linkbase

being evaluated on host0. The evaluation of an XPath query is segmented into a



- countries.xml / host 0 -

```
<countries xmlns:xlink="http://www.w3.org/1999/xlink"
   xmlns:dbxlink="http://www.dbis.informatik.uni-goettingen.de/linxis">
   ...
   <country car_code="NZ">
      <name>New Zealand</name>
      <population>3547983</population>
      <cities xlink:type="simple" dbxlink:transparent="drop-element insert-nodes"
       xlink:href="http://host1.foo.bar/countries.xml#xpointer('/countries/country[@car_code]="NZ"')"/>
   </country>
   ...
</countries>
```

- cities-NZ.xml / host 1 -

```
<cities>
  <city>              <!-- id#002 -->
    <name>            <!-- id#003 -->
      Wellington      <!-- id#004 -->
    </name>
  </city>
  <city>              <!-- id#005 -->
    <name>            <!-- id#006 -->
      Auckland        <!-- id#007 -->
    </name>
  </city>
</cities>
```

- cities-SGP.xml / host 2 -

```
<cities>              <!-- id#008 -->
  <city>              <!-- id#009 -->
    <name>            <!-- id#010 -->
      Singapore       <!-- id#011 -->
    </name>
  </city>
</cities>
```

- linkbase.xml -

```
<linkbase>                  <!-- id#012 -->
  <cityref                  <!-- id#013 -->
    xlink:label="WLG"
    xlink:type="locator"
    xlink:href="..."/>
  <cityref                  <!-- id#014 -->
    xlink:label="SIN"
    xlink:type="locator"
    xlink:href="..."/>
  <cityref                  <!-- id#015 -->
    xlink:label="FRA"
    xlink:type="locator"
    xlink:href="..."/>
  <cityref                  <!-- id#016 -->
    xlink:label="HNL"
    xlink:type="locator"
    xlink:href="..."/>
  <flight-con               <!-- id#017 -->
    xlink:type="arc"
    xlink:from="WLG"
    xlink:to="SGP" .../>
  <flight-con               <!-- id#018 -->
    xlink:type="arc"
    xlink:from="FRA"
    xlink:to="HNL" .../>
</linkbase>
```

host1 linkbase index

node		**arc**
id#001	→	∅
id#002	→	id#017
id#003	→	∅
id#004	→	∅
id#005	→	∅
id#006	→	∅
id#007	→	∅

host2 linkbase index

node		**arc**
id#008	→	∅
id#009	→	id#018
id#010	→	∅
id#011	→	∅

Figure 7.3: Evaluation of an XPath Query wrt. a Linkbase

number of steps. The steps correspond to the XPath expression's location steps. Each processing step is described in terms of its pre- and post condition (intermediate evaluation context before and after), and of a textual description of what happens in that very step. The initial context contains only the document node of countries.xml on host0 at address http://host0.foo.bar.

1. evaluate location steps /countries/country[@car_code="NZ"]
   context before:
   [doc-node("countries.xml")]
   via child axis from the document node and the countries *node test, first the root element "countries" is selected. From here, via child axis and with a* country *node test, all "country" elements are selected. Now, the predicate [@car_code="NZ"] is applied, which selects of all "country" elements that one with car code NZ (New Zealand).*

context after:
[country-element(NZ)]

2. evaluating //city[name="Wellington"]

   (a) evaluating axis "//"
   context before:
   [country-element(NZ)]
   from the New Zealand country element, the "//" axis (descendant-or-self) select the New Zealand country element itself, and all descendants.
   intermediate context:
   [country-element(NZ), name-element("New Zealand"), population-element, cities-element]
   cities is a Simple Link element. Resolving it yields all city-elements of New Zealand from the cities-NZ.xml document on host1.
   Part of the evaluation continues on host1 (due to Query Shipping), since the cities-NZ.xml instance is hosted there. Note that on host1, a linkbase index exists that is relevant for evaluation.
   context after:
   [country-element(NZ), name-element("New Zealand"), population-element, city-element(id#002), name-element(id#003), textnode(id#004), city-element(id#005), name-element(id#006), textnode(id#007)]

   (b) evaluating nodetest "city"
   context before:
   [country-element("New Zealand"), city-element(id#002), name-element(id#003), textnode(id#004), city-element(id#005), name-element(id#006), textnode(id#007)]
   all elements are eliminated except the two city elements.
   context after:
   [city-element(id#002), city-element(id#005)]

   (c) evaluating predicate [name="Wellington"]
   context before:
   [city-element(id#002), city-element(id#005)]
   predicate deletes the "Auckland" element (id#005) from the context, keeping just the "Wellington" city element.
   context after:
   [city-element(id#002)]

3. check linkbase index
   context before:
   [city-element(id#002)]
   consulting the linkbase index table reveals the fact that the Wellington city element is referenced as a from-locator from the arc with id#017, which is the flight-con arc representing the flight connection from Wellington to Singapore.
   The arc has the placement directive insert, which means that the arc's result is inserted into each from-node, here the Wellington city node. The arc's result is composed of the from-locator result (which is the city element of Singapore from cities-SGP.xml on host2) and wrapped into the flight-con arc element, which is renamed to its rolename in dbxlink:forward perspective, "flight-to".

```
<city>
  <flight-to>
    <city><name>Singapore</name></city>
  </flight-to>
  <name>Wellington</name>
</city>
```

The new, enriched element is denoted city-element(id#002)*.
context after:
[city-element(id#002)*]

4. evaluate /flight-to location step
   context before:
   [city-element(id#002)*]
   Navigates into the flight-to child element of the city-element(id#002).* context after:
   [flight-to-element(SGP)]
5. evaluate /city location step
   context before: [flight-to-element(SGP)]
   Navigates into the flight-to's child city-element. context after:
   [city-element(SGP)]
6. evaluate /flight-to location step
   context before:
   [city-element(SGP)]
   Navigates into the name child element of Singapore's city element. context after:
   [name-element("Singapore")]

So, the after-all result is the following element node:

`<name>Singapore</name>`

7.2.3 Query Shipping versus Data Shipping

The above example exhibits *that* the evaluation of a single XPath query can spread over multiple hosts, but it does not point out explicitly *how* this may happen. E.g. step 1 focuses on nodes on host0. Steps 2 and 3 deal with nodes on host1, step 4 and 5 deal with nodes originally on host2. Are these query parts evaluated at the host where the affected nodes are situated? Or are the remote nodes copied to the host where the query evaluation started from? Or a mixture of both?

If an XPath expression is evaluated with respect to a registered linkbase, and one location step crosses a Simple Link border, three options can be figured out for continuing the evaluation process.

Query Shipping: the unprocessed rest of the XPath expression is forwarded from the current server to the remote document and evaluated there, with the remote server respecting its local linkbase index table. The results from the remote server are serialized into XML data sent over the network, deserialized again and inserted as a copy into the local evaluation context.

Data Shipping: the referenced remote document is serialized and transferred completely to the server where the evaluation currently is taking place. Here, a copy is created and the query is evaluated locally on the copy.
Respecting the linkbase here would imply – since the linkbase indexes are stored locally at the server hosting the document where the physical instance of the from-locator-referenced nodes resides – not only to receive the complete remote document, but also the complete remote linkbase index, which then could be evaluated locally at the current server. This would result in even higher communication overhead. Also, for most straightforward implementations in imperative programming languages, the indexes would become invalid, since their addressing schema, hash tables etc. probably will be based on object IDs or similar, which are limited in scope to their original runtime environment. The tables could be restructured and transformed *before* sending them, but nevertheless the efforts would be enormous and the implementation would be cumbersome. So, this approach seems conceptually inappropriate.

Hybrid Shipping: the XPointer expression is transferred to the remote document and evaluated there; the resulting XML is transferred to the current server and copied there, and the XPath expression is evaluated to the local copy.
Here, the problem is basically the same: not only the regular result trees have to be sent back, but also the remote linkbase index, with a need to abstract from the local runtime environment, which would be barely more efficient than Data Shipping.

The above-mentioned aspects lead to the conclusion that the only conceptually promising evaluation approach here is Query Shipping, which also makes sense when analyzing potential scenarios: Consider a scenario which consists of a number of XLink-aware servers (Simple Links as well as Extended Link bases) on one hand (e.g. a distributed Mondial [May07] instance), and of a number of external non-XLink-aware XML sources (as remote Web Services, XML Databases, RSS newsfeeds, or plain XML files) on the other hand. The sources are referenced via Simple Links or as "to" locators from the linkbase's arcs. The first is addressed as the "own domain", the latter one may be called the "alien domain". Data shipping and Hybrid Shipping are offered alternatives for "weaker" XML services from the alien domain that are not XLink-aware and not capable of Query Shipping.

Since it seems more common to enrich the own domain by incorporating "alien" data instead of enriching "alien" data, the assumption of Query Shipping among the own-domain servers and of data or Hybrid Shipping among the alien data sources seems to be valid at least within a reasonably large subset of real-world data integration scenarios.

7.2.4 The Algorithm

It has been discussed in Section 6.3 that there exist three different notions of introducing 3rd Party Links to the DBXLink data model: with physical addressing (1), with Simple-Link-aware addressing (2), and with 3rd-Party-Link-aware addressing (3). Assuming that Query Shipping is used throughout the scenario, alternatives (1) and (2) seem quite feasible. Alternative (3) is significantly more complex. It would e.g. involve the processing of transitive closures for multiple arcs referencing the same node, which can become quite delicate. The issues of circular links and link bombs would become relevant in an even more weird fashion (due to the more complex modeling), and the introduced linkbase indexing by precomputing the locator results would not be applicable here (since nodes that are virtually blended into the document can't be indexed at the document's server site). Thus, the 3rd-Party-Link-aware addressing is considered to be rather of formal and theoretical interest than of practical impact. For the following, this work focuses on (2), the Simple-Link-aware addressing. Algorithms for processing are given below.

Only forward Perspective.

The given algorithms below implement the forward perspective, since inverse is symmetrically to forward. So, the from-locator always locates the nodes that are modified, replaced or fused. The to-locator always references the remote nodes that are inserted into / fused with / replace the from-nodes.

Part 1 - Registering the Linkbase.

The linkbase is preprocessed before evaluating an XPath expression on the distributed XML data with respect to the linkbase. For that, the XPointer expressions inside the from-locators' xlink:href attributes are evaluated. Since the sources may reference each other with Simple Links, and a from-locator's XPointer expression may cross such a Simple Link, the result of such an evaluation might be spread over multiple (physical) documents on multiple hosts. On each host, a *linkbase index table* is created that provides a mapping from the evaluated node to the arc containing the from-locator referencing that node. So, each server knows about the arcs that start from one of the nodes in one of the documents that the server is hosting.

In the following, the process of registering a linkbase over multiple servers is described in detail, starting with a description of the assumed server infrastructure.
Infrastructure / Prerequisites:

- We assume to have a number of servers $host_1, \dots host_n$, that are XLink-aware (have the logic to evaluate queries traversing Simple Links as well as queries over data referenced by 3rd Party Links).

- We assume to have a linkbase to be registered. The linkbase has arcs $arc_1 \ldots, arc_m$, each arc with one from-locator and one to-locator[1].

- Let EvalXPointer(xptr) = $[(node_1, host_1), \ldots, (node_n, host_n)]$ be the function that evaluates an XPointer expression, returning a set of nodes. The nodes may be distributed over different servers, with $host_i$ denoting the server where the node is originally located[2].

- Let InsertIntoIndex($host$,$node$,arc) insert the mapping ($node \rightarrow arc$) into the local linkbase index located at server $host$.

Procedure 7.1 (registerLinkbase), Procedure 7.2 (registerArc)

In Procedure 7.1, the linkbase is parsed, each arc is extracted. In Procedure 7.2, the arc's from-locator's XPointer expression is evaluated with respect to Simple Links. The result is the set of nodes identified by the arc's from-locator. These nodes might be spread over multiple servers/hosts. The arc itself is then sent to each of these hosts, to be stored there inside the local linkbase index table.

```
Procedure RegisterLinkbase
  Input: {$arc_1, \ldots, arc_m$} from linkbase
  Result: Each arc is sent to servers hosting nodes from the arc's "from"
          locator
1 (executed at the registering server)
2 begin
3     foreach $arc \in \{arc_1 \ldots, arc_m\}$ do
4         RegisterArc($arc$)
5     end
6 end
```

[1]As done earlier in this work (e.g. in Section 4.1), we focus without loss of generality on arcs identifying exactly one from- and exactly one to-node.

[2]The necessary informations are in detail: (1) the node information as name, node type, document element, children etc. as given in the XML Infoset, (2) the host where the – physical – document is hosted, and (3) some kind of local physical ID of the node on the host machine.

```
Procedure RegisterArc
  Input: $arc$ from linkbase
  Result: $arc$ is sent to server hosting nodes from $arc$'s from-locator
1 (executed at the registering server)
2 begin
3     $fromloc \longleftarrow arc$'s from-locator
4     $xpointer \longleftarrow fromloc$'s XPointer
5     $result = [(node_1, host_1), \ldots, (node_k, host_k)] \longleftarrow \text{Eval}(xpointer)$
6     (the nodes identified by $xpointer$, along with their host locations)
7     foreach $(node, host) \in result$ do
8         InsertIntoIndex($host, node, arc$)
9     end
10 end
```

Part 2 - Evaluating XPath over XLinked XML with Linkbases.

The linkbase has now been registered, and its arc information has been spread over the participating hosts' local linkbase indexes. Now, an XPath expression is issued to the XLinked data. Simple Links as well as Linkbase information might add nodes to the virtual instance. The children/attributes must be checked whether they are XLink-relevant or not, and – if so – if their outcome is relevant for the current XPath location step.

The check "are there any relevant arcs for node X?" is performed by lookup in the linkbase index that is created during the registering. The linkbase index is a pre-computed hashtable, with a node's local address inside the storage system of the hosting server's database system as key value. If there are arcs outgoing from X, they can be found with X's local address.

As described in Section 7.2.3, Query Shipping is the only evaluation strategy which is compliant with the pursued linkbase approach. Thus, Query Shipping is assumed to be used for all links.

- Let $arc \longleftarrow$ GetIndex($node$) return the arc associated with the given node from the local linkbase index of server *host*, when executed there.
- *xpath-expr*$_1$/*step*$_x$/*xpath-expr*$_2$ are three parts of an XPath expression, with *xpath-expr*$_1$ being the part that has already been processed, *step*$_x$ being the current location step, and *xpath-expr*$_2$ being the yet unprocessed part.

Note that the following algorithms for evaluating XPath expressions over XML with Simple XLinks are adaptions from the dissertation of Erik Behrends [Beh06], with logic added for including linkbases into the evaluation.

Procedure 7.3 (processRelevantLinks)

For each element in the current evaluation context, its text children, element children, and attributes are checked if they are (1) Simple XLink Elements, or (2) nodes that are touched by an arc. In both cases, they have to be expanded to satisfy the virtual data model.

```
Procedure processRelevantLinks
  Input: A current context (set of nodes) $C$, xpath-expr$_1$, step$_x$,
         xpath-expr$_2$.
  Result: Relevant links resolved in advance for the next step.
begin
    foreach element $e \in C$ do
        $L \longleftarrow$ getRelevantLinks($e$, step$_x$)
        while $L \neq \emptyset$ do
            for $\ell \in L$ do
                if isSimpleLink($\ell$) then
                    resolveSimpleLink($\ell$, step$_x$, xpath-expr$_2$)
                else if isArc($\ell$) then
                    resolveExtendedLink($\ell$, $L$.next(), step$_x$, xpath-expr$_2$)
                    $L$.remove($L$.next())
            end
            $L \longleftarrow$ getRelevantLinks($e$, step$_x$)
        end
    end
end
```

Procedure 7.4 (getRelevantLinks)
Here, the XLink-relevant children of a given context element are determined. Relevant are Simple Link elements, since they might add elements to the virtual instance, and children or attributes addressed by a 3rd Party Link's from-locator. They can be replaced or modified by the arc's result, and by that also add nodes to the virtual instance.

Function 7.5 (performNodetest)
performNodetest checks if the name of an arc result enriching the current document can be precomputed from inspecting the arc's transparent values. If so, and if the resulting node name does not match the nodetest from the given location step, then the arc is not to be included into the evaluation (returns "false"). If the name of the result node cannot be guessed, or if the location step doesn't include a name test (e.g. wildcard "*"), or if the name matches the name test, then "true" is returned.

Procedure 7.6 (resolveSimpleLink)
resolves Simple Link Elements either by (1) Query Shipping, (2) Data Shipping, or (3) Hybrid Shipping (these terms are defined in Section 7.2.3).

Procedure 7.7 (resolveExtendedLink)
resolveExtendedLink takes a node, an arc that references the node via its from-locator, the current location step, and the remaining, yet unevaluated XPath expression. The node is replaced / modified with the arc's result, which is acquired by replacing the arc with a construct involving Simple Links, and adding the construct to the current evaluation context. The construct itself is then evaluated as a regular Simple Link.

Function getRelevantLinks

```
Input: An element $e$, $step_x$, the local $LinkbaseIndex$.
Output: The $links$ children relevant for $e$ wrt. $step_x$.
begin
    $links, tmpList \longleftarrow emptyList$
    $axis_x \longleftarrow$ the axis of $step_x$;
    $nodetest_x \longleftarrow$ the nodetest of $step_x$
    switch $axis_x$ do
        case self
            // do nothing
        case child
            $tmpList \longleftarrow e.getChildren() \circ e.getAttributes()$
        case descendant
            $tmpList \longleftarrow e.getDescendants() \circ e.getAttributes() \circ$
            $e.getDescendants().getAttributes()$
        case descendant-or-self
            $tmpList \longleftarrow e.getDescendants() \circ e.getAttributes() \circ$
            $e.getDescendants().getAttributes()$
        case following-siblings
            $tmpList \longleftarrow e.getFollowingSiblings()$
        case following
            $tmpList \longleftarrow e.getFollowing() \circ e.getFollowing().getAttributes()$
        case attribute
            $tmpList \longleftarrow e.getChildren() \circ e.getAttributes()$
    end
end
foreach $node \in tmpList$ do
    if isSimpleLink(node) then
        $link \longleftarrow (SimpleLink)node$
        if $axis_x$ = attribute then
            if link.getLDirective() = "make-attribute" and link matches
            $nodetest_x$ then
                $links$.add($link$)
            else if link.getLDirective() ∈{"drop-element", "keep-body"} then
                $links$.add($link$)
        else if link.getLDirective() ∈ {"group-in-element",
        "duplicate-element"} and link matches $nodetest_x$ then
            $links$.add($link$)
        else if link.getLDirective() ∈ {"drop-element", "keep-body"} then
            $links$.add($link$)
    else if LinkbaseIndex.contains(node) and performNodetest(arc, $step_x$) then
        $links$.add($arc$), $links$.add($node$)
end
return $links$
```

Function 7.8 (arc2SimpleLink)
Translates the given arc and node from arc's from-locator into an equivalent

```
Function performNodetest
   Input: $arc$,$step_x$
   Output: $false$ if $arc$'s prospective result doesn't match nodetest; else $true$.
1  begin
2      $nametest \longleftarrow step_x$.getNametest()
3      if $nametest = null$ then
4          return true
5      if $arc.getPlacingDirective() = $ "place-insert" then
6          return $nametest = node$.getName()
7      else
8          // placing is "place-replace"
9          if $arc.getLDirective() \in$
           {"dup-arc-elem","group-arc-elem","make-arc-attr"} then
10             return $nametest = arc$.getName()
11         else if $arc.getLocator().getLDirective() \in$
           {"dup-arc-elem","group-arc-elem","make-arc-attr"} then
12             return $nametest = arc$.getToLocator().getName()
13     return true
14 end
```

```
Procedure resolveSimpleLink
   Input: A Simple Link element $\ell$, $step_x$, $xpath\text{-}expr_2$
   Result: $\ell$ has been resolved.
1  begin
2      switch $\ell.getAttribute($"dbxlink:eval"$)$ do
3          case "local"
4              $href \longleftarrow \ell.getAttribute($"xlink:href"$)$
5              $doc \longleftarrow$ getReferencedDocument($href$)
6              $frag \longleftarrow$ getLocalXMLFragment($doc$,
               $href.getSubstringAfter($"#"$)$)
7              addXMLFragment($frag$, $\ell.getAttribute($"dbxlink:transparent"$)$)
8          case "distributed"
9              $frag \longleftarrow$ getXMLFragment($\ell.getAttribute($"xlink:href"$)$)
10             addXMLFragment($frag$, $\ell.getAttribute($"dbxlink:transparent"$)$)
11         case "remote"
12             if canShipQuery($\ell$, $step_x$, $xpath\text{-}expr_2$) then
13                 $q \longleftarrow$ buildQueryToShip($\ell$, $step_x$, $xpath\text{-}expr_2$)
14                 shipQuery($q$)
15             else
16                 $\ell \longleftarrow \ell.setAttribute($"dbxlink:eval","dist ributed"$)$
17                 resolveSimpleLink($\ell$, $step_x$, $xpath\text{-}expr_2$)
18     end
19 end
```

structure containing only Simple XLinks. Example 7.4 presents an overview on how to translate an arc into Simple Link structures, depending on the arc's

Procedure resolveExtendedLink

```
Input: $arc$,$node$,$step_x$,$xpath\text{-}expr_2$
Output: $node$ has been resolved with respect to $arc$.
1 begin
2     $newlinks \longleftarrow$ arc2SimpleLink($arc$,$node$)
3     foreach $link \in newlinks$ do
4         resolveSimpleLink($link$,$step_x$,$xpath\text{-}expr_2$)
5     end
6 end
```

transparent and placement directives.

Note that for the (arc's L-Directive ; to-locator's L-Directive) tuples, the combinations "dup-arc-elem ; dup-to-elem" and "dup-arc-elem ; dup-to-elem" cannot be replaced by Simple Link constructs. For both, the L-Directives behavior of the arc element AND the to-locator element cannot be simulated using a single Simple Link, since the result structure and size is a priori unknown, but would be needed to simulate the correct "wrapping" of the result elements. Hence, Query Shipping cannot be supported, since the cardinality of the surrounding arc element depends on the – yet uncomputed – result. Instead, the arc's result must be evaluated by using γ_R and γ_L directly: γ_R and γ_L are applied to the locator element, then γ_L is used for the arc element. Finally, the results are put together with γ_{LR}.

7.2.5 Resolving an Arc

Example 8 *For the following examples, assume the prefix* x *to be bound to the w3c* xlink *namespace, and* d *to the linxis* dbxlink *namespace.* d:transparent *is abbreviated to* d:trans.

```
<somearc x:type="arc" d:trans="group-arc-elem dup-to-elem ins-to-nodes" non-xlink-arcs/>
<somearcs-tolocator x:type="locator" d:trans="..." x:href="toloc-href"/>
```

The arc element somearc *and its* to-locator somearcs-tolocator *are transformed as described in Figure 7.4:*

```
<somearc non-xlink-arcs>
   <somearcs-tolocator x:type="simple" x:href="toloc-href" d:trans="duplicate-element insert-nodes"/>
</somearc>
```

To take advantage of the already existing infrastructure for evaluating Simple Links, a promising approach is to evaluate an arc element by rewriting it into a result-equivalent structure consisting of Simple Links and non-XLink nodes, and add it into the current result context for further processing.

Due to the nature of dup-to-elem, where the number of duplications depends on the – by time of expansion unknown – size of the locator's result set, and due to keep-to-body spreading the locator's non-XLink body among also that result

Function arc2SimpleLink

```
Input: $arc$, $node$
Output: a replacement for $node$ is returned in shape of a Simple Link structure,
        emulating $arc$'s outcome.
begin
    if $arc.getPlacingDirective() = "insert"$ then
        if $isElement(node)$ then
            $arc$.setPlacingDirective("replace")
            $node$.addChild(arc2SimpleLink($arc$))
        return $node$
    else
        // replace
        $locator \longleftarrow arc$.getToLocator()
        $locator$.setXLinkType("simple")
        $locator$.setLDirective($arc$.getToLocatorLDirective())
        $locator$.setRDirective($arc$.getToLocatorRDirective())
        $arc \longleftarrow arc$.removeXLinkProperties()
        switch $arc.getArcLDirective()$ do
            case $"drop-element"$
                return $locator$
        end
        case $"keep-body"$
            return $\{locator\} \cup arc$.getBody()
        case $"group-in-element"$
            $arc$.addChild($locator$)
            return $arc$
        case $"duplicate-element"$
            $locatorresult \longleftarrow gamma_{LR}(locator)$
            $result \longleftarrow$ emptySet()
            foreach $nodelist \in locatorresult$ do
                $result \longleftarrow result \cup \{arc.addBody(nodelist)\}$
            end
            return $result$;
        case $"make-attribute"$
            $locatorresult \longleftarrow gamma_{LR}(locator)$
            $arc$.setXLinkType("simple")
            $arc$.setLDirective("make-attribute")
            $(result, newbucket) \longleftarrow gamma_L(arc, locatorresult)$
            $bucket \longleftarrow bucket \cup newbucket$ // side effect: bucket update
            return $result$
end
```

set, the combinations 1.1 and 1.4 from Example 8 can't simply be replaced with Simple Link constructs. The solution here would be either

materializing: First the locator's result is materialized (which can be enormous), then the arc's left-hand directive is applied to the intermediate result. In this case, the XLink engine cannot take advantage of Query Shipping, even if the remote system supports Query Shipping.

modify Query Shipping: the other alternative involves recoding and enhancing the Query Shipping unit. With document constructors like document" + innerQuery + "/xpath, complex XQueries, including also information from the arc element, can be shipped to the other side and can be evaluated there (Although, this approach also has its shortcomings: e.g. a link element can contain non-XLink subtrees, that theoretically would need to be shipped to the remote host, together with the rest of the query, to be included in the result evaluation there).

For the scope of the implementation prototype, the first of these alternatives was chosen. If possible, arcs are compiled into Simple Links which are given back into the regular Simple Link evaluation process, and for those few cases where this is not feasible, the arc is evaluated by materializing.

7.3 Implementation of the Prototype

7.3.1 The eXist Database System

For the implementation of the evaluation techniques from 7, a matter of choice was (1) to build a complete new XLink-aware XPath/XQuery processor from scratch, using only basic XML APIs as SAX, DOM, Xerces etc., or to (2) enhance an already existing XPath/XQuery system to process 3rd Party XLinks in the described way. Alternative 1 would include the benefit of a conceptually clean design with the *a priori* design goal of XLink/3rd Party Link evaluation, where the second alternative would imply to modify an already existing – and probably complex – system which was originally not designed to follow XLink references, which could make it necessary not only to modify the XPath/XQuery evaluation unit, but also the storage model, indexing algorithms etc.

In the end, the choice was made for alternative 2: enhancing an already existing XPath/XQuery system. In the present case, this was the XML database system eXist [exi]. eXist offers – among other advantages – complete XQuery support. Also, my co-researcher Erik Behrends decided earlier to use eXist as a basis for his own work, which was focussed on implementing Simple Link functionality into an XML database. So, I could benefit from the already present Simple Link modifications, and I could take advantage from his insight and – overall positive – experiences with the interna of the eXist system.

Hence, choosing eXist as a code basis seemed the most promising solution in terms of reusability, user-friendliness during testing and during the construction of a case study, and transferability of the results to industrially relevant systems.

The eXist database system is a native XML database system, completely written in Java. Since the software is open source, it is maintained and developed by a vivid developer community. It features a number of XML-related and Web-related standards as XPath/XQuery, XUpdate, XSL, XML Schema, XInclude,

	transparent value		resulting structure
	arc part	to-locator part	
1.1	dup-arc-elem	dup-to-elem	—
1.2	dup-arc-elem	group-to-elem	<somearc *arc-nonxlink-attrs*> <somearcs-tolocator x:type="simple" d:trans="group-element .." x:href=".." *loc-nonxlink-attrs*/> </somearc>
1.3	dup-arc-elem	drop-to-elem	<somearc x:type="simple" d:trans="group-element .." x:href=".." *arc-nonxlink-attrs*/>
1.4	dup-arc-elem	keep-to-body	—
1.5	dup-arc-elem	make-to-attr	<somearc *arc-nonxlink-attrs*> <somearcs-tolocator x:type="simple" d:trans="make-attribute .." x:href=".." *loc-nonxlink-attrs*/> </somearc>
2.1	group-arc-elem	dup-to-elem	<somearc *arc-nonxlink-attrs*> <somearcs-tolocator x:type="simple" d:trans="duplicate-element .." x:href=".." *loc-nonxlink-attrs*/> </somearc>
2.2	group-arc-elem	group-to-elem	<somearc *arc-nonxlink-attrs*> <somearcs-tolocator x:type="simple" d:trans="group-element .." x:href=".." *loc-nonxlink-attrs*/> </somearc>
2.3	group-arc-elem	drop-to-elem	<somearc *arc-nonxlink-attrs*> <someloc x:type="simple" d:trans="drop-element .."x:href=".."/> <*other arc's children . . .* /> </somearc>
2.4	group-arc-elem	keep-to-body	<somearc *arc-nonxlink-attrs*> <someloc x:type="simple" d:trans="keep-body .." x:href=".." *to-loc-nonxlink-attrs*/> </somearc>
2.5	group-arc-elem	make-to-attr	See 1.5
3.1	drop-arc-elem	dup-to-elem	<somearcs-tolocator x:type="simple" d:trans="duplicate-element .." x:href=".." *loc-nonxlink-attrs*/>
3.2	drop-arc-elem	group-to-elem	<somearcs-tolocator x:type="simple" d:trans="group-element .." x:href=".." *loc-nonxlink-attrs*/>
3.3	drop-arc-elem	drop-to-elem	<somearcs-tolocator x:type="simple" d:trans="drop-element .." x:href=".." />
3.4	drop-arc-elem	keep-to-body	<somearcs-tolocator x:type="simple" d:trans="keep-body .." x:href=".." *loc-nonxlink-attrs*/>
3.5	drop-arc-elem	make-to-attr	<somearcs-tolocator x:type="simple" d:trans="make-attribute .." x:href=".." />
4.1	keep-arc-elem	dup-to-elem	<somearcs-tolocator x:type="simple" d:trans="duplicate-element .." x:href=".." *arc-nonxlink-attrs loc-nonxlink-attrs*/>
4.2	keep-arc-elem	group-to-elem	See 4.1
4.3	keep-arc-elem	drop-to-elem	<somearc x:type="simple" d:trans="keep-body .." x:href=".." *arc-nonxlink-attrs*/>
4.4	keep-arc-elem	keep-to-body	<somearc x:type="simple" d:trans="keep-element .." x:href=".." *arc-nonxlink-attrs loc-nonxlink-attrs*/>
4.5	keep-arc-elem	make-to-attr	See 3.5

5.1	make-arc-attr	dup-to-elem	somearc="*idref*"[3]
			bucket := [<somearcs-tolocator x:type="simple" dbxlinkID="*idref*"
			d:trans="duplicate-element .." x:href=".." *loc-nonxlink-attrs*/>,..]

5.2	make-arc-attr	group-to-elem	somearc="*idref*"
			bucket := [<somearcs-tolocator x:type="simple" dbxlinkID="*idref*"
			d:trans="group-element .." x:href=".." *loc-nonxlink-attrs*/>,..]

5.3	make-arc-attr	drop-to-elem	<somearc x:type="simple" d:trans="make-attribute .." x:href=".." />

5.4	make-arc-attr	keep-to-body	somearc="*idref*"
			bucket := [<somearcs-tolocator x:type="simple" dbxlinkID="*idref*"
			d:trans="keep-body .." x:href=".." *loc-nonxlink-attrs*/>,..]

5.5	make-arc-attr	make-to-attr	<somearc x:type="simple" d:trans="make-attribute .." x:href=".."/>

Figure 7.4: Results for Example 8

HTTP as means for interaction and data querying/manipulation. With its JSP/Servlet architecture, the application adopts an up-to-date Web application paradigm, and since it is quickly evolving and adopting new features, it seemed like a promising infrastructure for the LinXIS prototype implementation.

7.3.2 Software Architecture

The eXist database system is client-server based, with the options to run it either in a stand-alone server mode (accessible via http), as well as in JSP/servlet Web application context. The latter option offers interaction with the eXist database via HTTP-GET, HTTP-POST, WebDav and XML-RPC. Along with the server, there comes an interactive Web interface for setting off XQuery/XUpdate commands, an interactive client for querying, updating and administering databases, together with a number of jar files containing the jetty application server [jet07], a lightweight[4] servlet application engine in which the eXist server can be run.

7.3.3 Database Architecture

An eXist database system consists of the eXist server – as mentioned above either running in stand-alone mode or as a java servlet application in a servlet engine – together with a single database instance. The instance consists (in terms of its data model) of a number of "collections", with each collection containing a number of XML documents. Physically, the XML data is organized in B*-Trees which are stored in persistent files, completed with a number of files containing several kinds of index structures.

Metadata concerning user and permission management is stored in a dedicated XML collection called "system". Here, write/read and authoring permissions as well as collection metadata are maintained. The database can be accessed and operated by multiple users/operators in parallel using multiple access methods as HTTP, WebDav etc. Transaction ACID criteria are granted for XQuery and XUpdate.

The database is capable of processing DTD and XML Schema by validating each XML document during uploading. However, for updating an existing XML document, there is no new conformance checking of the updated document with its associated DTD or schema. Also, the database seems to do no kind of post schema validation (e.g., #FIXED attributes from DTDs are not included in the XQuery evaluation).

Data and Storage Model

The internal data model for XML nodes is based on the DOM data model; the classes ElementImpl, AttrImpl etc. implement the DOM interfaces for Element, Attribute etc. Due to the storage model, not all DOM methods are implemented for the node types. Insertion, updating and deletion operations as well as many

[4]"lightweight" in opposition to large-scale, highly customizable engines such as Apache Tomcat

operations concerning attributes are subject to dedicated insert/update operations that are performed by a database broker. The database broker gathers metadata about collections and XML instances, and coordinates concurrent access to the database.

The storage model differs in some points from the DOM data model: e.g., attributes are considered as "children" of their hosting element, which means that they appear intermixed with other element or text children of an element, when retrieving actual children or inserting/appending new ones.

Indexing Schema

eXist in the versions $\geq$ 1.0 uses the *dynamic level numbering (DLN)* [BR04] schema for indexing nodes proposed by Böhme and Rahm in 2004. The schema is based on a hierarchical notation. Each node – element, attribute, text or other node – exception: the document node – is assigned an index key consisting of a number of numeric values and "." (dot) as separators. When parsing a new XML document into a collection, the root element receives the DLN index 1. The first child in document order (which might also be an attribute) receives the 1.1, the second child gets 1.2 and so on. So, 1.4.7.2 stands for the 2nd child of the 7th child of the 4th child of the root element.

How does insertion of nodes into a document work with the numbering scheme depicted above? Consider the (element) node 1.1 with child nodes $1.1.1, \ldots, 1.1.9$. If between nodes 1.1.7 and 1.1.8, a sequence of 3 new nodes are to be inserted as children of 1.1, the inserted nodes have to be given the numbers $1.1.8, 1.1.9, 1.1.10$, and the former nodes 1.1.8 and 1.1.9 must be renumbered to 1.1.11 and 1.1.12. This is compliant with the numbering scheme, but frequent renumbering is very inefficient (since e.g. all child nodes of renumbered nodes have to be renumbered as well).

So, the new nodes are assigned the keys 1.1.7/1, 1.1.7/2, 1.1.7/3 etc. If now another node is inserted between 1.1.7/1 and 1.1.7/2, it is assigned the key 1.1.7/1/1. Hence, the dot separator can be seen as a "vertical" separator (separating the layers of the tree), and the slash can be seen as a "horizontal" separator, adding new neighbors to a node without disturbing the already present neighbors.

From time to time, the storage structures are reorganized, which goes along with a restructuring of the index keys (removing "/" steps by "pulling up" the node to their predecessor's layer, renumbering all following sibling nodes) and with updating the index structures to the new key identifiers. eXist supports – among others – an element index (by name, also including attributes), and a value index (by text content, text nodes and attributes), plus methods for searching children and descendants of given nodes by their QNames or values.

Remark: the "id" values mentioned in Section 7.2 are implemented using a concatenation of the hosting document's URL and the concerned node's current DLN index key. In the prototype, the "local" database, which circumvents all those documents containing nodes referenced by the local linkbase's from-locators, is assumed to remain unmodified. Theoretically, it would be possible to make the linkbase infrastructure update-compliant by restructuring the linkbase's index keys each time the other indexes are restructured too. Since this would have involved lot of tiny code work, going along with minimal academic relevance, I refrained from implementing update stability for linkbase indexing.

7.3.4 XPointer/XInclude Support

eXist in its original state offers support for XLink/XPointer together with XInclude [XIn06]. Since XInclude is a – less expressive – subset of the introduced XLink semantics, and since it lacks a proper data model, the XInclude features of eXist were not used for adding the XLink/DBXLink functionality to the eXist query engine.

7.3.5 Version

The LinXIS-aware prototype is based on eXist version 1.0-beta from May 2006, revision 3595. The current version of the original eXist is 1.1 (final release), with version 1.1.1 in development.

Chapter 8

Case Study: the "Flightbase"

This chapter is a case study for some "real world" application involving 3rd Party Links, in order to study the effects on creation, use and maintenance of a 3rd Party Link application with the prototype implementation based on the eXist Database [exi] described in the previous chapters.

As a business domain for the case study, the flight schedule of the commercial airline Lufthansa was chosen. One reason was the almost allegoric similarity between a flight schedule and a linkbase in terms of the modeling: a flight schedule can be seen as a directed graph between airports, and such a graph can be modeled – almost without any abstraction – as a 3rd Party Link, with airports as locators, and flight connections as arcs.

The second reason for choosing the flight schedule scenario was the availability and accessibility of the needed data sources. For geographical aspects, the approved Mondial database could provide the necessary base data about countries and cities, delivering the targets for the linkbase's locator pointers. The linkbase itself is based on the – publicly available – flight schedule of the Lufthansa airline company [LH07].

The chapter is organized as follows: First, the overall structure and design of the distributed Mondial linkbase is sketched.
Second, the process of generating, extracting and integrating data from the flight schedule (plus some additional Web sources) into a linkbase is described in detail.
Next, the hardware configuration where the application operates on is depicted.
Then, a comparison is performed between 3 example queries, (1) expressed and evaluated on the 3rd Party Link flight schedule example, and (2) on the same set of distributed XML files, but without use of 3rd Party Links, instead with *explicit navigation* over XLinks/XPointers using the Saxon XQuery processor.

Finally, an evaluation is given on the implementation in terms of performance, usability and practical relevance, together with a number of possible future improvements.

8.1 Distributing the Mondial Database

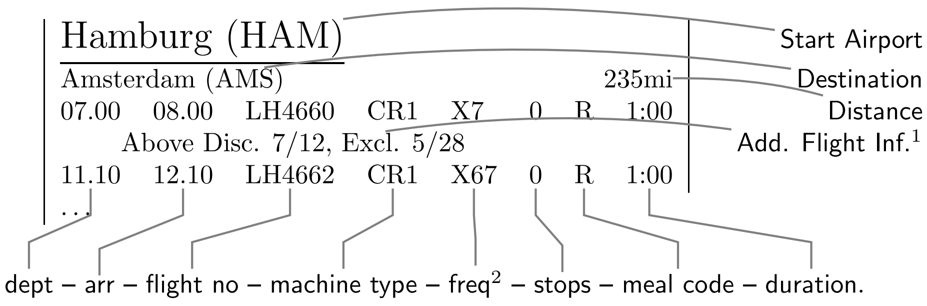
The Mondial Database does already exist in its distributed version (see [May07]). For the use case, one minor change had to be made: the dbxlink:eval attributes of all occurring XLinks had to be set to “remote” to enable query shipping, since both local evaluation and hybrid evaluation perform relatively bad with respect to queries against the virtual data model. In its distributed version, the monolithical mondial.xml file is split into a number of XML files and spread over a number of hosts, here given as linxis01.ifi.informatik.uni-goettingen.de, . . . , linxis05.ifi.informatik.uni-goettingen.de:

- mondial-root.xml at linxis01:
  starting document, contains references to countries, continents, organizations as well as geographical entities.
- countries.xml, continents.xml at linxis02:
  contains information about all countries and continents. Each country with international car code XX contains references to its cities in cities-XX.xml.
- cities-A.xml . . . cities-ZW.xml and provs-A.xml . . . provs-ZW.xml at linxis03:
  each file contains information about the respective countries’ cities in cities-XX.xml. Same with provinces.
- organizations.xml, flightbase.xml at linxis04:
  contains data about international organizations. Each organization has 0 or more countries as members, as well as a headquarter city. Also, the flightbase.xml linkbase (see below) is situated here.
- geo.xml at linxis05:
  contains data about geographical entities, as lakes, mountains, rivers, seas, and deserts.

8.2 Generating the Linkbase

The flight schedule scenario involves the Lufthansa flight schedule data, containing all flight connections offered by Lufthansa plus partner airlines, and the geographical data from Mondial, providing the cities as endpoints of the flight connections. Hence, the flight schedule had to be transformed into a linkbase (*"flightbase"*) with airports as locator elements referencing MONDIAL's cities, and arcs representing the flight connections between airports/cities.

For generating the flightbase, certain portions of information had to be identified in both MONDIAL and the flight schedule, in order to be mapped correctly onto each other. In the flight schedule, the schedule looks like follows:



```
Hamburg (HAM)
Amsterdam (AMS)                                  235mi
07.00   08.00   LH4660   CR1   X7    0   R   1:00
     Above Disc. 7/12, Excl. 5/28
11.10   12.10   LH4662   CR1   X67   0   R   1:00
...
```

Start Airport
Destination
Distance
Add. Flight Inf.^{1}
dept – arr – flight no – machine type – freq^2 – stops – meal code – duration.

The available information about the airport is the given name, which is often the name of the city nearby the airport (here: Hamburg), sometimes together with the airport's name - in case a city has more than one airport, e.g. London (Heathrow) and London City Airport. In some cases, the name is neither a city's nor the airport's name (e.g. Funchal Airport on Madeira Island is simply denoted as "Madeira"). Together with the name goes the three-letter IATA-Code [IAT07] of the airport.

8.2.1 IATA-Code

The IATA-Code is a trigram code identifying each international airport. E.g. San Francisco is SFA, Frankfurt am Main is FRA. At [IAT07], a list of airports can be obtained, together with city and country information, sometimes region information, as well as the complete name of the airport (e.g. "JFK (KJFK) John F. Kennedy International Airport (formerly Idlewild Airport) Jamaica, New York (New York City), United States"). The documents are in XHTML, which makes the automatic data extraction easier, since I could use XPath/XQuery directly on the XHTML documents. But since there is no fixed schema for these webpages (except that is has to be valid XHTML), and since the descriptions of each airport differ from case to case, the generated data has to be checked for consistency and be manually refined afterwards. Another problem is that the code words are not unique: 323 of the 17,576 triples are used by more than one airport, e.g. *AUH = Abu Dhabi Airport at the Persian Gulf* and *AUH = Aurora Municipal Airport, Aurora, Nebraska, United States.*

8.2.2 Description of the Integration Process

In the following, the data integration process involving the 3 data sources:

- Lufthansa flight schedule (LH.pdf),

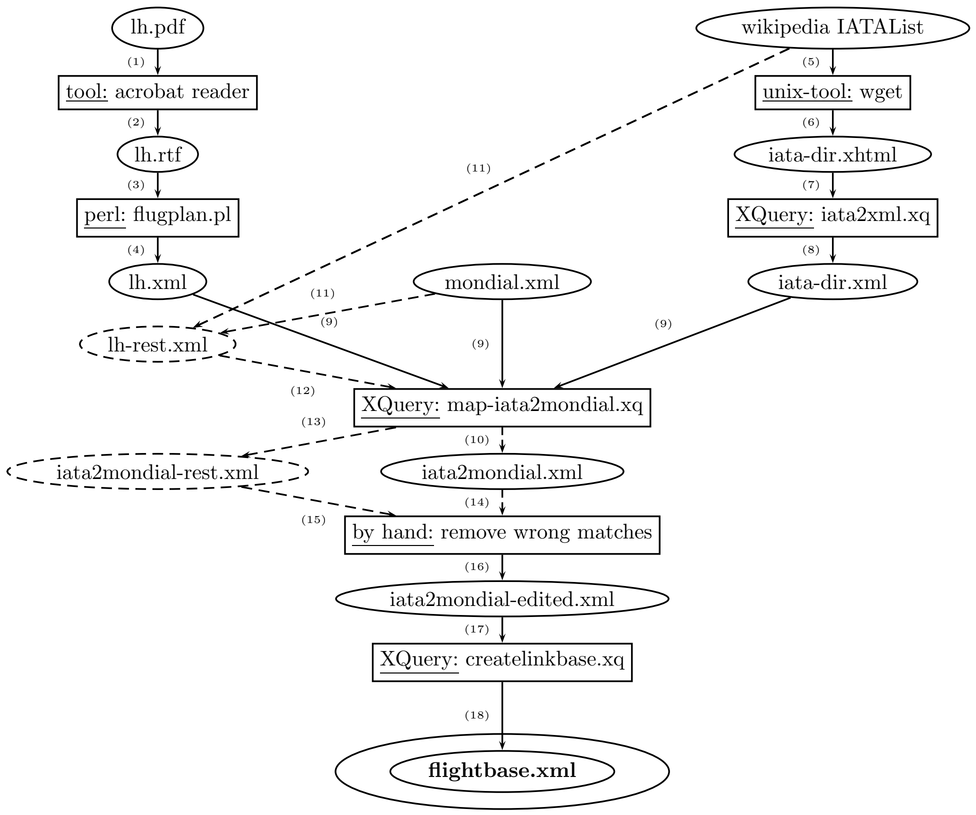


Figure 8.1: diagram of the integration process

- MONDIAL (Mondial.xml), and
- the IATA data from Wikipedia

is described in order of the numbering of the steps from (1) to (18) in Figure 8.1.

- **acrobat reader:** in order to make the Lufthansa flight schedule Schedule PDF (about 600kb in size) machine-readable, the Adobe® Reader®'s copy&paste option was used, obtaining a RTF (Rich Text Format) version of the schedule. Rich Text Format [RTF07] is a cross-platform markup language for text documents. Since RTF is ascii-based, it can be processed using scripting languages such as awk, sed, perl etc.
- **flugplan.pl:** flugplan.pl is a perl script, generating XML data from the flight schedule data in lh.rtf. The result is an XML instance lh.xml containing 1473 flight connections and 416 airports.
- **wget:** wget is a common unix tool for automatically receiving multiple HTML files from the Web, either by recursion or by specifying them in

a file. iata-dir.xml is a huge file containing all IATA information from the english Wikipedia site [IAT07] (which is spread among 26 sub-pages according to their starting letters) as one huge XHTML document.

- **iata2xml.xq** is an XQuery program that collects and refines data from the Wikipedia XHTML data in iata-dir.xml into XML data which associates each IATA airport code with its airport, city and - if present - region name.

- **map-iata2mondial.xq:**

  (1) This XQuery program joins the schedule data containing references to airports (via airport / city name and IATA code) and the MONDIAL data containing cities using the IATA code information from Wikipedia. First, the IATA codes from lh.xml are extracted. Then, the corresponding data from the Wikipedia-driven iata-dir.xml is received, consisting of airport name, city's/cities' name(s), country name, and sometimes region or state name. Then, some heuristic matching algorithm is performed, consisting of refining, splitting and combining the known airport information, in order to receive (possibly multiple) matches among the MONDIAL cities. Since all three the Wikipedia data, the MONDIAL data and the Lufthansa data are not formalized in terms of the nomenclature and spelling of cities, this process is heuristic and hence error-inflicted to some degree. This may lead to a number of false positives: cities that are assigned to an IATA code by mistake. Example: Abu Dhabi airport is assigned Abu Dhabi in MONDIAL, but also Aurora Municipal Airport in Aurora, Nebraska, since Aurora and Abu Dhabi share the same IATA AUH (see above). As well, false negatives can appear: cities that are present in MONDIAL, but that aren't recognized by the heuristics. Example: Frankfurt, Germany is named as "Frankfurt" in the Lufthansa Schedule, but in MONDIAL it is named "Frankfurt am Main", to distinguish it from the German city *Frankfurt an der Oder*. So both strings won't match – even not partially, since the Wikipedia airport names are split up and sought after in MONDIAL, but not the other way round – and "Frankfurt" cannot be assigned to the MONDIAL element for Frankfurt. A frequent source of such misalignments are different transcriptions of German Umlauts, as well as the diacritic accents used in languages as French, Spanish, Polish etc.
  (2) After the creation of iata2mondial.xml in the previous step, both lh.xml and mondial.xml have to be scanned (manually, using regexp search) for false negatives: airport data from lh.xml that reference cities that are present in MONDIAL, but have not been matched, and thus have not been included in iata2mondial.xml. When finding such a city, the airport elements that did not match in the previous step were manually modified, so that they would now (e.g. the name of the airport of South Corean city "Busan" was changed to "Busan/Pusan", so that the heuristics could match it with the "Pusan" entry in MONDIAL). The modified airport elements were collected in lh-rest.xml. Then, the last step was repeated

with lh-rest.xml instead of lh.xml, producing a iata2mondial-rest.xml file containing 19 previously false negatives.

- **removing wrong matches:** having eliminated false negatives in the previous step, the iata2mondial.xml and iata2mondial-rest.xml files were merged by hand. False positives were removed by hand: duplicate IATA codes as the Aurora - Abu Dhabi case, duplicate cities, or some airports that have multiple cities assigned. For example, the Köln-Bonn Airport is assigned to both Cologne and Bonn. In these cases, usually the bigger city was selected, supposing that the bigger city usually is the more relevant one.

- **createlinkbase.xq:** finally, the XQuery program createlinkbase.xq creates the flightbase from the iata2mondial-edited.xml by creating xpointers pointing to the MONDIAL cities (in the *distributed* MONDIAL version), by creating locator elements using the XPointers as xlink:href attribute values, and by creating arcs using the locators as xlink:from and xlink:to. The original LH.pdf contained 1473 flight connections between 416 airports, the final flightbase.xml contains 1274 connections between 348 cities. So, the flightbase is not complete with respect to the original flight schedule, which in most cases can be tracked to airports that are in the schedule, but not present in MONDIAL (e.g. Faro, Portugal).
A short excerpt of the produced linkbase flightbase.xml is given in Figure 8.2.

8.3 Hardware Configuration and Test Setup

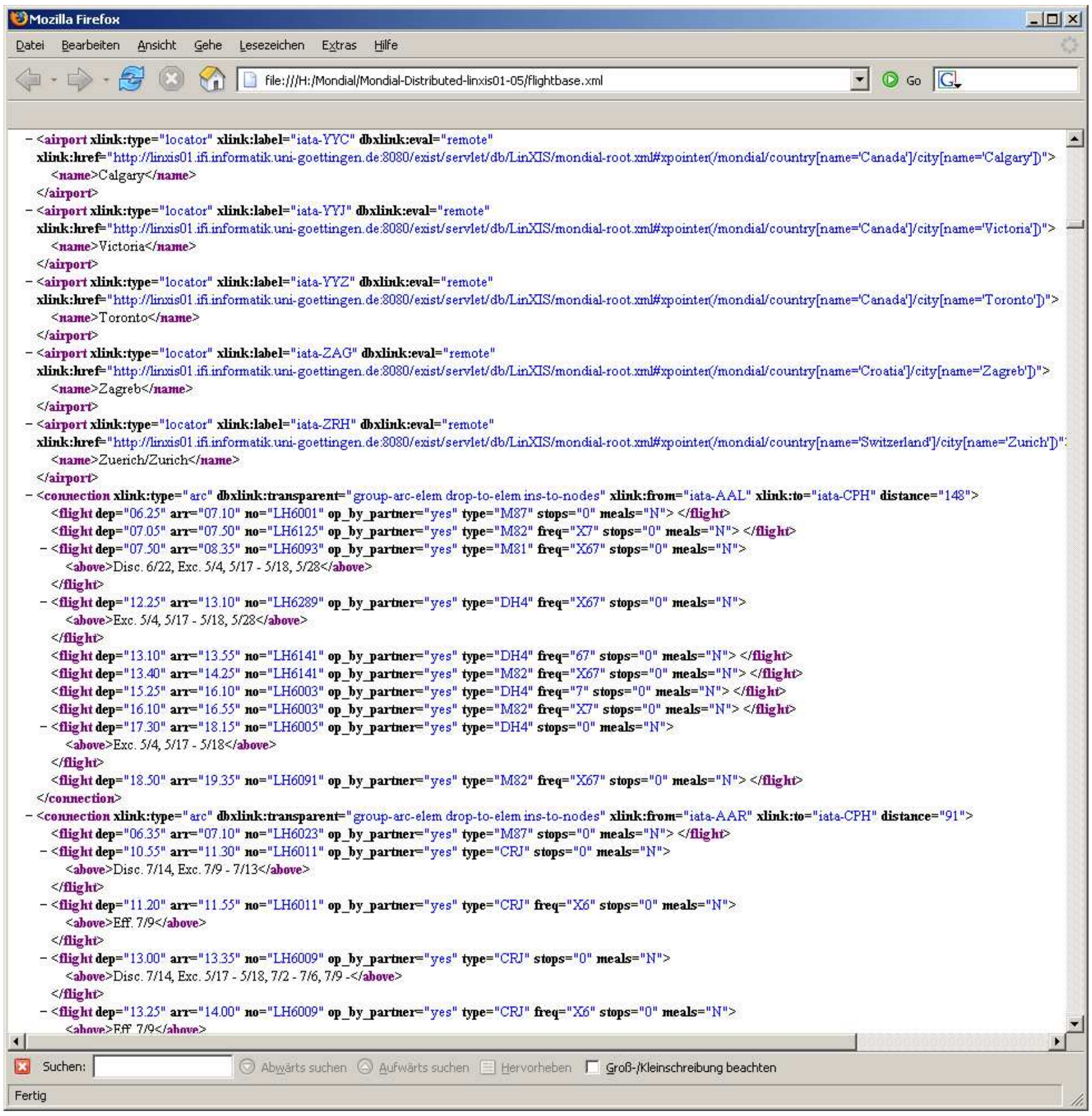
8.3.1 Hardware

For testing the scenario consisting of the distributed MONDIAL version plus the flightbase, the following hardware configuration was used:

A Debian Sarge Linux box, with 2x Intel® Xeon™ CPUs with 2.80GHz, 6GB of memory and a RAID 5 3×75GB Disk Drive with netto 146.5GB. On the Linux box, a VMWare®Server is running with 5 virtual hosts named linxis01...linxis05.

8.3.2 Test Setup

The idea of the case study was, to motivate the XLink/Linkbase concept based on some real-world example, together with a small set of queries on that example. On one hand, the virtual model is queried, including linkbase and Simple Link evaluation with the modified eXist system. One the other hand, the same queries are issued against the physical system using a non-server-based, lightweight XQuery engine as Saxon® in order to produce the equivalent result


Mozilla Firefox
Datei Bearbeiten Ansicht Gehe Lesezeichen Extras Hilfe
file:///H:/Mondial/Mondial-Distributed-linxis01-05/flightbase.xml
Go
Suchen:
Abwärts suchen
Aufwärts suchen
Hervorheben
Groß-/Kleinschreibung beachten
Fertig

```
- <airport xlink:type="locator" xlink:label="iata-YYC" dbxlink:eval="remote"
  xlink:href="http://linxis01.ifi.informatik.uni-goettingen.de:8080/exist/servlet/db/LinXIS/mondial-root.xml#xpointer(/mondial/country[name='Canada']/city[name='Calgary'])">
    <name>Calgary</name>
  </airport>
- <airport xlink:type="locator" xlink:label="iata-YYJ" dbxlink:eval="remote"
  xlink:href="http://linxis01.ifi.informatik.uni-goettingen.de:8080/exist/servlet/db/LinXIS/mondial-root.xml#xpointer(/mondial/country[name='Canada']/city[name='Victoria'])">
    <name>Victoria</name>
  </airport>
- <airport xlink:type="locator" xlink:label="iata-YYZ" dbxlink:eval="remote"
  xlink:href="http://linxis01.ifi.informatik.uni-goettingen.de:8080/exist/servlet/db/LinXIS/mondial-root.xml#xpointer(/mondial/country[name='Canada']/city[name='Toronto'])">
    <name>Toronto</name>
  </airport>
- <airport xlink:type="locator" xlink:label="iata-ZAG" dbxlink:eval="remote"
  xlink:href="http://linxis01.ifi.informatik.uni-goettingen.de:8080/exist/servlet/db/LinXIS/mondial-root.xml#xpointer(/mondial/country[name='Croatia']/city[name='Zagreb'])">
    <name>Zagreb</name>
  </airport>
- <airport xlink:type="locator" xlink:label="iata-ZRH" dbxlink:eval="remote"
  xlink:href="http://linxis01.ifi.informatik.uni-goettingen.de:8080/exist/servlet/db/LinXIS/mondial-root.xml#xpointer(/mondial/country[name='Switzerland']/city[name='Zurich'])"
    <name>Zuerich/Zurich</name>
  </airport>
- <connection xlink:type="arc" dbxlink:transparent="group-arc-elem drop-to-elem ins-to-nodes" xlink:from="iata-AAL" xlink:to="iata-CPH" distance="148">
    <flight dep="06.25" arr="07.10" no="LH6001" op_by_partner="yes" type="M87" stops="0" meals="N"> </flight>
    <flight dep="07.05" arr="07.50" no="LH6125" op_by_partner="yes" type="M82" freq="X7" stops="0" meals="N"> </flight>
  - <flight dep="07.50" arr="08.35" no="LH6093" op_by_partner="yes" type="M81" freq="X67" stops="0" meals="N">
      <above>Disc. 6/22, Exc. 5/4, 5/17 - 5/18, 5/28</above>
    </flight>
  - <flight dep="12.25" arr="13.10" no="LH6289" op_by_partner="yes" type="DH4" freq="X67" stops="0" meals="N">
      <above>Exc. 5/4, 5/17 - 5/18, 5/28</above>
    </flight>
    <flight dep="13.10" arr="13.55" no="LH6141" op_by_partner="yes" type="DH4" freq="67" stops="0" meals="N"> </flight>
    <flight dep="13.40" arr="14.25" no="LH6141" op_by_partner="yes" type="M82" freq="X67" stops="0" meals="N"> </flight>
    <flight dep="15.25" arr="16.10" no="LH6003" op_by_partner="yes" type="DH4" freq="7" stops="0" meals="N"> </flight>
    <flight dep="16.10" arr="16.55" no="LH6003" op_by_partner="yes" type="M82" freq="X7" stops="0" meals="N"> </flight>
  - <flight dep="17.30" arr="18.15" no="LH6005" op_by_partner="yes" type="DH4" stops="0" meals="N">
      <above>Exc. 5/4, 5/17 - 5/18</above>
    </flight>
    <flight dep="18.50" arr="19.35" no="LH6091" op_by_partner="yes" type="M82" freq="X67" stops="0" meals="N"> </flight>
  </connection>
- <connection xlink:type="arc" dbxlink:transparent="group-arc-elem drop-to-elem ins-to-nodes" xlink:from="iata-AAR" xlink:to="iata-CPH" distance="91">
    <flight dep="06.35" arr="07.10" no="LH6023" op_by_partner="yes" type="M87" stops="0" meals="N"> </flight>
  - <flight dep="10.55" arr="11.30" no="LH6011" op_by_partner="yes" type="CRJ" stops="0" meals="N">
      <above>Disc. 7/14, Exc. 7/9 - 7/13</above>
    </flight>
  - <flight dep="11.20" arr="11.55" no="LH6011" op_by_partner="yes" type="CRJ" freq="X6" stops="0" meals="N">
      <above>Eff. 7/9</above>
    </flight>
  - <flight dep="13.00" arr="13.35" no="LH6009" op_by_partner="yes" type="CRJ" stops="0" meals="N">
      <above>Disc. 7/14, Exc. 5/17 - 5/18, 7/2 - 7/6, 7/9 -</above>
    </flight>
  - <flight dep="13.25" arr="14.00" no="LH6009" op_by_partner="yes" type="CRJ" freq="X6" stops="0" meals="N">
      <above>Eff. 7/9</above>
```

Figure 8.2: Excerpt from flightbase.xml

sets. Since Saxon cannot deal with XLinks, the queries had to be rewritten towards explicit XLink navigation using the non-XQuery standard saxon:evaluate function for evaluating an XPath expression to some remote XML resource into a nodeset.

eXist comes in a from-the-shelf distribution containing the XML server, plus some additional classes, including the Jetty® servlet engine in which the eXist server is usually run in. The Saxon queries use the same MONDIAL files, using

Jetty only as a webserver supplying the Mondial files via Web. It is the only running application on the Linux box, but inside the VMWare® , there are some other virtual hosts running. Hence, it might be possible that some test queries can run longer, depending on the workload of the physical machine. I tried to eliminate this effect to a minimum by running each query three times and taking the arithmetic mean of the three runtimes as resulting runtime. This technique was applied to the eXist queries as well as to the Saxon queries.

8.4 Query Comparison

All queries are more or less based on the assumption that a person – a traveling salesman, an airline employee or a customer – asks for a certain connection, or a set of connections, presenting a structured set of single connections as result. The queries are “exhaustive” in a sense that the queries traverse large portions of the linkbase, sometimes even multiple times (especially the queries in Sections 8.5.2 and 8.5.3).

8.4.1 Query I: Germany to India

Task: Find all flight connections from German cities to India.

In the Virtual Model:

```
for $fromcity in /cities/city,
    $tocity in $fromcity/connection/city
  let $countryname := $tocity/country/name
  where $countryname="India"
  return <con>
           <from>{$fromcity/name/text()}</from>
           <to>{$tocity/name/text()}</to>
           <country>{$countryname/text()}</country>
          </con>
```

In the Physical Model:

```
declare namespace xlink="http://www.w3.org/1999/xlink";

declare function local:normalize-space($s as xs:string) as xs:string {
  fn:replace($s,"%20"," ")
};

declare function local:get-hrefparts($href as xs:string) as xs:string* {
  let $hrefparts := fn:tokenize($href,'#')
  let $hrefhost := $hrefparts[1]
  let $hrefxptr := fn:substring($hrefparts[2],11,fn:string-length($hrefparts[2])-11)
  let $tokens := fn:tokenize($hrefxptr,"/","m")
  return ($hrefhost, $tokens)
};

declare function local:get-locators($flightbasedoc as document-node()) as element()* {
  $flightbasedoc/*/airport
};


declare function local:get-airportcities($locators as element()*) as element()* {
  for $locator in $locators
    let $href := fn:string($locator/@xlink:href)
    let $hrefparts := local:get-hrefparts($href)
    let $cities-host := $hrefparts[1]
    let $citiesstep := $hrefparts[2]
    let $citystep := local:normalize-space($hrefparts[3])
    let $city :=
      doc($cities-host)/saxon:evaluate(fn:concat("/",$citiesstep,"/",$citystep))
  return <pair>{($locator,$city[population=max($city/population)])}</pair>
};

let $host := 'http://linxisXX.ifi.informatik.uni-goettingen.de:8080/exist/servlet/db/LinXIS/'
let $host01 :=  fn:replace($host,"linxisXX","linxis01")
let $host03 :=  fn:replace($host,"linxisXX","linxis03")
let $host05 :=  fn:replace($host,"linxisXX","linxis05")
let $citiesD:= doc(fn:concat($host03,"cities-D.xml"))
let $Germancities := $citiesD/cities/city
let $locators := local:get-locators(doc(fn:concat($host05,"flightbase.xml")))
let $loc-city-pairs := local:get-airportcities($locators)
let $German-loc-city-pairs :=
  for $p in $loc-city-pairs
    let $l := $p/airport
    let $c := $p/city
    where ($c = $germancities) return $p

  let $indian-loc-city-pairs :=
  for $p in $loc-city-pairs
     let $l := $p/airport
     let $c := $p/city
     let $country-hrefparts := local:get-hrefparts(fn:string($c/country/@xlink:href))
     let $host := $country-hrefparts[1]
     let $country := doc($host)/saxon:evaluate(fn:concat("/",
          $country-hrefparts[2],"/",$country-hrefparts[3]))
     where $country/name="India"
     return $p

let $g2i-connections :=
  for $con in doc(fn:concat($host05,"flightbase.xml"))/*/connection
  where $con/@xlink:from = $german-loc-city-pairs/airport/@xlink:label
    and $con/@xlink:to =   $indian-loc-city-pairs/airport/@xlink:label
  return $con

let $result-tuples :=
   for $c in $g2i-connections
     let $fromcity := $german-loc-city-pairs/self::pair[airport/@xlink:label = $c/@xlink:from]/city
     let $tocity := $indian-loc-city-pairs/self::pair[airport/@xlink:label = $c/@xlink:to]/city
   return <con>
     <from>{$fromcity/name/text()}</from>
     <to>{$tocity/name/text()}</to>
   </con>

return <result
  xmlns:xlink="http://www.w3.org/1999/xlink"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:dbxlink="http://dbis.informatik.uni-goettingen.de/linxis">
 {$result-tuples}
</result>
```

In the virtual model, the query is relatively compact. Though, the processing in the background is complex, since in the query clause, a number of XLink “edges” have to be traversed. The city elements have virtual child elements of type connection, which are added by the flightbase. The connection elements each have a Simple Link child that evaluates to a city element – the destination city of the connection. Each city element has a – virtual – country attribute, that evaluates to the country element where the city belongs to. All these “linked axes” are traversed in the above XPath expression.

In the physical model, the query is relatively long and complicated, with much efforts being put in processing the xlink:href XPointer values and producing the necessary joins. It should be stressed that in terms of comparability, it is important not to use implicit knowledge – e.g. that all Indian city elements are in some document cities-IND.xml – but rather use only the information given in the xlink:href attributes. *The system should behave exactly the same if the distribution across files and hosts was a completely different one.*

8.4.2 Query II: All Connections from Hannover to Lisbon

Since Lufthansa doesn’t offer direct connections from Hannover, Germany to Lisbon, Portugal, the system must be queried for indirect connections including one transit. Also, there should be a valid chance for a transfer in so far as connection A to airport X should have at least one flight with an arrival time earlier than the departure time of at least one flight from airport X to the destination Lisbon.

In the Virtual Model:

```
for $con1 in doc('cities-D.xml')/cities/city[name='Hannover']/connection
   let $city := $con1/city
   return
     for $con2 in $con1/city/connection[city/name='Lisbon']
     where ($con2/flight/@arr > $con1/flight/@dep)
     return
       <con>{($con1,$con2)}</con>
```

In the Physical Model:

```
declare namespace xlink="http://www.w3.org/1999/xlink";

declare function local:normalize-space($s as xs:string) as xs:string {
  fn:replace($s,"%20"," ")
};

declare function local:get-hrefparts($href as xs:string) as xs:string* {
  let $hrefparts := fn:tokenize($href,'#')
  let $hrefhost := $hrefparts[1]
  let $hrefxptr := fn:substring($hrefparts[2],11,fn:string-length($hrefparts[2])-11)
  let $tokens := fn:tokenize($hrefxptr,"/","m")
  return ($hrefhost, $tokens)
};

declare function local:get-locators($flightbasedoc as document-node()) as element()* {
  $flightbasedoc/*/airport
};


declare function local:get-airportcities($locators as element()*) as element()* {
  for $locator in $locators
    let $href := fn:string($locator/@xlink:href)
    let $hrefparts := local:get-hrefparts($href)
    let $cities-host := $hrefparts[1]
    let $citiesstep := $hrefparts[2]
    let $citystep := local:normalize-space($hrefparts[3])
    let $city :=
      doc($cities-host)/saxon:evaluate(fn:concat("/",$citiesstep,"/",$citystep))
    return <pair>{($locator,$city[population=max($city/population)])}</pair>
};

let $host := 'http://linxisXX.ifi.informatik.uni-goettingen.de:8080/exist/servlet/db/LinXIS/'
let $host01 :=  fn:replace($host,"linxisXX","linxis01")
let $host03 :=  fn:replace($host,"linxisXX","linxis03")
let $host05 :=  fn:replace($host,"linxisXX","linxis05")
let $citiesD:= doc(fn:concat($host03,"cities-D.xml"))
let $locators := local:get-locators(doc(fn:concat($host05,"flightbase.xml")))
let $loc-city-pairs := local:get-airportcities($locators)
let $hannover := $loc-city-pairs[city/name="Hannover"]
let $lisbon := $loc-city-pairs[city/name="Lisbon"]
let $han2lis-connections :=
  for $con1 in doc(fn:concat($host05,"flightbase.xml"))/*/connection,
      $con2 in doc(fn:concat($host05,"flightbase.xml"))/*/connection
  where $con1 != $con2
    and $con1/@xlink:from = $hannover/airport/@xlink:label
    and $con2/@xlink:to = $lisbon/airport/@xlink:label
    and $con1/@xlink:to = $con2/@xlink:from
  return <con>{($con1,$con2)}</con>

return
<result xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:dbxlink="http://dbis.informatik.uni-goettingen.de/linxis">
  {$han2lis-connections}
</result>
```

8.4.3 Query III: All Connections from Munich, Germany to Auckland, New Zealand without Stopover in the U.S.

Similar to query II, the database is searched for a for a two-flight connection. Starting point is Munich in Germany, endpoint is Auckland, New Zealand. Additional constraint is that the flight has no stopover in any city in the United States. For each pair of single connections, the overall distance is calculated as the sum of the distances of the two single connections. The result connections are sorted by distance in ascending order.

In the Virtual Model:

```
for $con1 in /cities/city[name='Munich']/connection
  let $transitcountryname := $con1/city/country/name
  let $dist1 := $con1/@distance
  return
    for $con2 in $con1/city/connection
      let $destcityname := $con2/city/name
      let $distance := $dist1 + $con2/@distance
      where (($transitcountryname !="United States") and ($destcityname = "Auckland"))
      order by $distance
      return <con>{attribute distance {$distance},$con1,$con2}</con>
```

In the Physical Model:

```
declare namespace xlink="http://www.w3.org/1999/xlink";

declare function local:normalize-space($s as xs:string) as xs:string {
  fn:replace($s,"%20"," ")
};

declare function local:get-hrefparts($href as xs:string) as xs:string* {
  let $hrefparts := fn:tokenize($href,'#')
  let $hrefhost := $hrefparts[1]
  let $hrefxptr := fn:substring($hrefparts[2],11,fn:string-length($hrefparts[2])-11)
  let $tokens := fn:tokenize($hrefxptr,"/","m")
  return ($hrefhost, $tokens)
};

declare function local:get-locators($flightbasedoc as document-node()) as element()* {
  $flightbasedoc/*/airport
};

declare function local:get-airportcities($locators as element()*) as element()* {
  for $locator in $locators
    let $href := fn:string($locator/@xlink:href)
    let $hrefparts := local:get-hrefparts($href)
    let $cities-host := $hrefparts[1]
    let $citiesstep := $hrefparts[2]
    let $citystep := local:normalize-space($hrefparts[3])
    let $city :=
      doc($cities-host)/saxon:evaluate(fn:concat("/",$citiesstep,"/",$citystep))
      return <pair>{($locator,$city[population=max($city/population)])}</pair>
    };


let $host := 'http://linxisXX.ifi.informatik.uni-goettingen.de:8080/exist/servlet/db/LinXIS/'
let $host01 :=  fn:replace($host,"linxisXX","linxis01")
let $host03 :=  fn:replace($host,"linxisXX","linxis03")
let $host05 :=  fn:replace($host,"linxisXX","linxis05")
let $citiesD:= doc(fn:concat($host03,"cities-D.xml"))
let $locators := local:get-locators(doc(fn:concat($host05,"flightbase.xml")))
let $loc-city-pairs := local:get-airportcities($locators)
let $munich := $loc-city-pairs[city/name="Munich"]
let $auckland := $loc-city-pairs[city/name="Auckland"]

let $us-cities :=
  for $p in $loc-city-pairs
    let $country-hrefparts := local:get-hrefparts(fn:string($p/city/country/@xlink:href))
    let $countrydoc := doc($country-hrefparts[1])
    let $country := $countrydoc/saxon:evaluate(fn:concat("/",$country-hrefparts[2],"/",$country-hrefparts[
3]))
    where $country/name="United States"
    return $p

let $muc2auck-connections :=
  for $con1 in doc(fn:concat($host05,"flightbase.xml"))/*/connection,
      $con2 in doc(fn:concat($host05,"flightbase.xml"))/*/connection
  let $distance := $con1/@distance + $con2/@distance
  where
    (: check connections :)
    $con1/@xlink:from = $munich/airport/@xlink:label
    and $con2/@xlink:to = $auckland/airport/@xlink:label
    and $con1/@xlink:to = $con2/@xlink:from
    and (not (fn:exists($us-cities/airport[@xlink:label=$con1/@xlink:to])))
  order by $distance
  return <con>{(attribute distance {$distance},$con1,$con2)}</con>


return <result xmlns:xlink="http://www.w3.org/1999/xlink"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:dbxlink="http://dbis.informatik.uni-goettingen.de/linxis">
          <allcons>{$muc2auck-connections}</allcons>
        </result>
```

Here, the basic problem is the same as in query two: find a connection from A to B consisting of two basic connections $A \rightarrow X$ and $X \rightarrow B$ using some stopover airport X. The only differences are (1) the negation in describing the set of possible stopover airports (*'no city in the U.S.'*), and (2) the sorting of the result connections by their summed-up distances. Obviously, the latter one has no impact on the approach on neither the virtual model query nor the physical model query. For the stopover airport restriction, the "country" link has to be evaluated.

8.4.4 Query Environment

Virtual vs. Physical

Both groups of queries, the virtual-model queries via eXist and the physical-model queries via Saxon, were issued via http from the same machine, a unix workstation connected to the same local area network as the linxis0X host machines. For Saxon, the internal http client was used using the Saxon doc() function. For the virtual queries, a small java wrapper was used, taking the query string as input (the start document was hardcoded into the wrapper code). Reassuringly, each virtual-physical pair of queries produced the same output in all cases[3]. For the exact query results, see below.

Query Running Conditions

The queries I, II and III were issued 6 times each, divided in two runs with the sequence I-II-III-I-II-III-I-II-III. Since, in the virtual model, answering XLinked queries produces additional “virtual documents”, increasing the indexes’ sizes and thereby decreasing the database performance, the database was reset after each query, so that the order of the queries did not influence the query answering time. For issuing the queries and for automation of the experiment, I used a configuration of cron jobs on linxis01-linxis05, cooperating with a perl script executed on a remote host (s2.ifi.informatik.uni-goettingen.de, in the same local network), issuing the queries. Inter-host synchronization was performed via file locking over a common AFS filesystem.

8.5 Evaluation and Summary

8.5.1 Query Results

The issued queries yielded the following results:

Query I: Frankfurt to India

```
<con>
  <name>Bangalore</name>
  <name>India</name>
</con>
<con>
  <name>Madras</name>
  <name>India</name>
</con>
<con>
  <name>Hyderabad</name>
  <name>India</name>
</con>
<con>
  <name>Calcutta</name>
  <name>India</name>
</con>
<con>
  <name>Mumbai</name>
  <name>India</name>
</con>
```

[3]Some visual differences, however, may be traced back to eXist’s serialization functionality. In the virtual model, there exist no XLink elements. Albeit if a query result never contains XLink elements, a non-XLink element may have XLink children, e.g. the airport link inside each connection element. Due to the straightforward XML serialization of eXist, these children are output straight from their physical representation, instead of being filtered out by the serializer. Nevertheless, the *querying itself* performs correct, since the direct query results are XLink-free. Outputting the result nodes *including their child and attribute nodes* is more a custom than a convention, and does not touch the validity of the query answer.

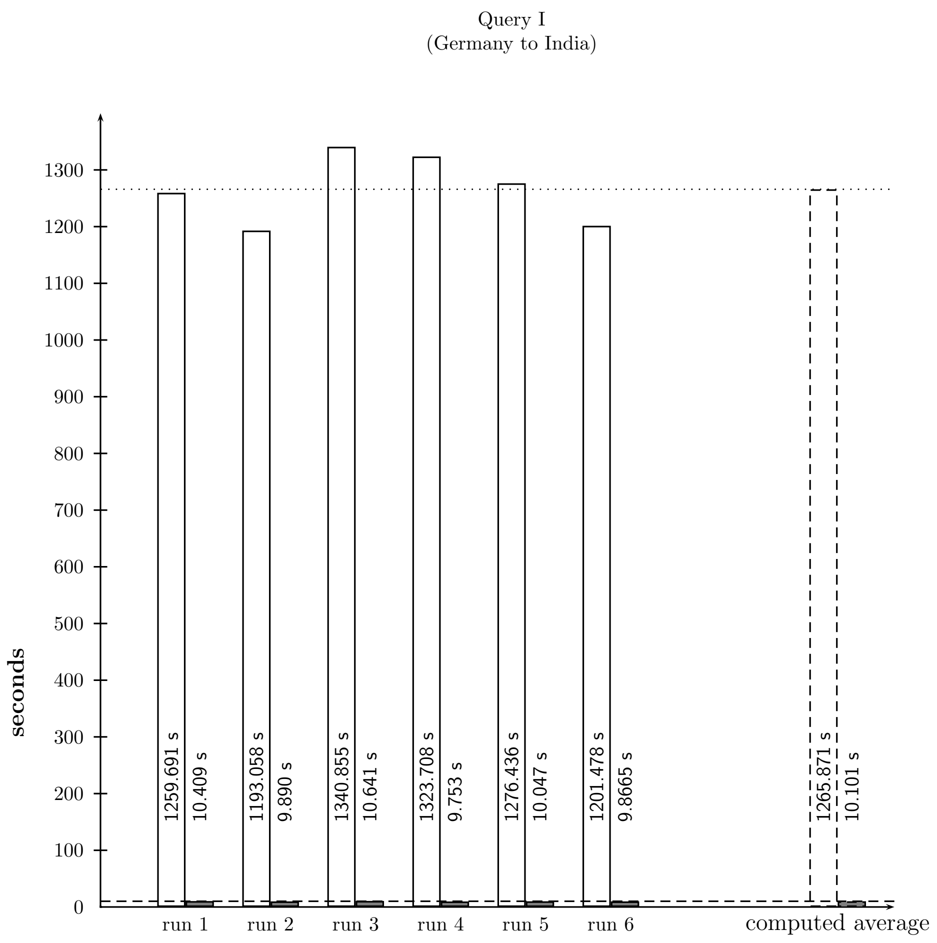


Figure 8.3: Runtime of Query I in the virtual LinXIS model (white) and in the physical model using Saxon (gray)

In words: the query returned five flight connections from Frankfurt to India:

1. Frankfurt → Bangalore
2. Frankfurt → Madras
3. Frankfurt → Hyderabad (India)
4. Frankfurt → Calcutta
5. Frankfurt → Mumbai

8.5.2 Query II: All Connections from Hannover to Lisbon

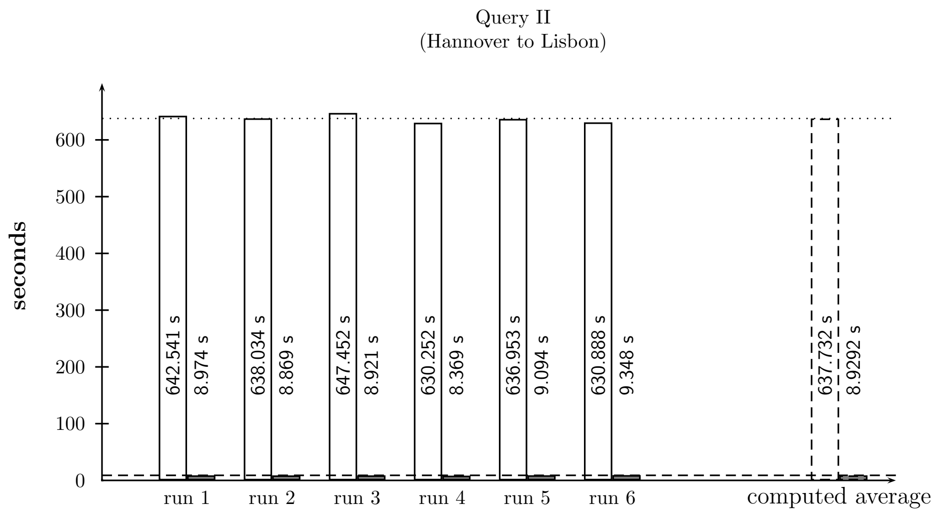


Figure 8.4: Runtime of Query II in the virtual LinXIS model (white) and in the physical model using Saxon (gray)

```
<con>
  <connection xlink:type="arc" dbxlink:transparent="group-arc-elem drop-to-elem ins-to-nodes"
     xlink:from="iata-HAJ" xlink:to="iata-FRA" distance="174">
    <flight dep="05.40" arr="06.40" no="LH1001" .../>
    <flight dep="06.15" arr="07.15" no="LH1001" .../>
    <flight dep="07.00" arr="08.00" no="LH1003" .../>
    <flight dep="07.15" arr="08.15" no="LH1003" .../>
    <flight dep="09.55" arr="10.55" no="LH1005" ...>
        <above>Exc. 5/17 - 5/20, 5/27 - 5/28</above>
    </flight>
    <flight dep="11.30" arr="12.30" no="LH1007" .../>
    <flight dep="13.20" arr="14.20" no="LH1009" .../>
    <flight dep="13.40" arr="14.40" no="LH1009" ...>
        <above>Eff. 5/2, Exc. 5/18, 5/27 - 5/28</above>
    </flight>
    <flight dep="14.30" arr="15.30" no="LH1011" .../>
    <flight dep="18.50" arr="19.50" no="LH1013" .../>
  </connection>
  <connection xlink:type="arc" dbxlink:transparent="group-arc-elem drop-to-elem ins-to-nodes"
     xlink:from="iata-FRA" xlink:to="iata-LIS" distance="1165">
    <flight dep="06.45" arr="08.40" no="LH2174" .../>
    <flight dep="09.30" arr="11.25" no="LH4530" .../>
    <flight dep="13.30" arr="15.25" no="LH2172" .../>
    <flight dep="13.40" arr="15.35" no="LH4532" .../>
    <flight dep="19.15" arr="21.10" no="LH2176" .../>
    <flight dep="21.55" arr="23.50" no="LH4536" .../>
  </connection>
</con>
```

```
   <con>
      <connection xlink:type="arc" dbxlink:transparent="group-arc-elem drop-to-elem ins-to-nodes"
                  xlink:from="iata-HAJ" xlink:to="iata-MUC" distance="298">
        <flight dep="07.05" arr="08.15" no="LH1017" .../>
        <flight dep="08.50" arr="10.00" no="LH1021" .../>
        <flight dep="11.10" arr="12.20" no="LH1025" ...>
            <above>Exc. 5/27 - 5/28</above>
        </flight>
        <flight dep="13.05" arr="14.15" no="LH1027" .../>
        <flight dep="15.10" arr="16.20" no="LH1019" ...>
            <above>Exc. 5/27, 6/8</above>
        </flight>
        <flight dep="17.15" arr="18.25" no="LH1029" .../>
        <flight dep="19.00" arr="20.10" no="LH1033" .../>
        <flight dep="20.35" arr="21.45" no="LH1035" ...>
            <above>Exc. 5/17 - 5/18, 6/7 - 6/8</above>
        </flight>
      </connection>
      <connection xlink:type="arc" dbxlink:transparent="group-arc-elem drop-to-elem ins-to-nodes"
                  xlink:from="iata-MUC" xlink:to="iata-LIS" distance="1234">
        <flight dep="06.35" arr="08.40" no="LH2212" .../>
        <flight dep="11.10" arr="13.20" no="LH4540" .../>
        <flight dep="14.10" arr="16.15" no="LH2170" .../>
        <flight dep="19.05" arr="21.15" no="LH4544" .../>
      </connection>
   </con>
</result>
```

In words: The database contains two connections from Hannover to Lisbon: one going via Frankfurt am Main, the other one via Munich.

8.5.3 Query III: All Connections from Munich, Germany to Auckland, New Zealand without Stopover in the U.S.

```
<result xmlns:dbxlink="http://dbis.informatik.uni-goettingen.de/linxis"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xlink="http://www.w3.org/1999/xlink">
   <allcons>
      <con distance="11314">
         <connection xlink:type="arc" xlink:from="iata-MUC" xlink:to="iata-HKG" distance="5610"
                     dbxlink:transparent="group-arc-elem drop-to-elem ins-to-nodes">
            <flight dep="21.40" arr="15.10+1" no="LH730" type="343" stops="0" meals="MM"/>
         </connection>
         <connection xlink:type="arc" xlink:from="iata-HKG" xlink:to="iata-AKL" distance="5704"
                     dbxlink:transparent="group-arc-elem drop-to-elem ins-to-nodes">
            <flight dep="18.10" arr="09.00+1" no="LH9810" op_by_partner="yes" type="744" stops="0"
                    meals="DB"/>
         </connection>
      </con>
      <con distance="11416">
         <connection xlink:type="arc" xlink:from="iata-MUC" xlink:to="iata-BKK" distance="5455"
                     dbxlink:transparent="group-arc-elem drop-to-elem ins-to-nodes">
            <flight dep="21.10" arr="12.40+1" no="LH9716" op_by_partner="yes" freq="346" stops="0"
                    meals="M"/>
         </connection>
```

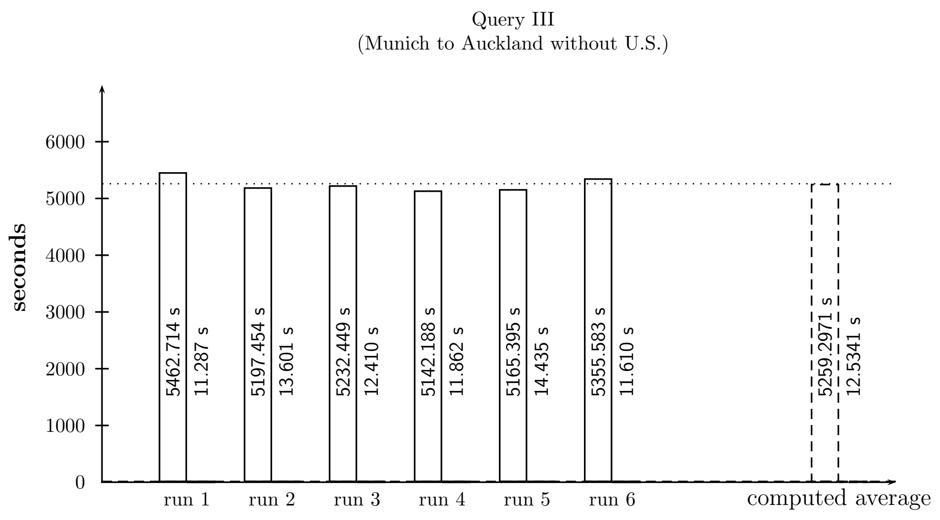


Figure 8.5: Runtime of Query I in the virtual LinXIS model (white) and in physical model using Saxon (gray)

```
            <connection xlink:type="arc" xlink:from="iata-BKK" xlink:to="iata-AKL" distance="5961"
                        dbxlink:transparent="group-arc-elem drop-to-elem ins-to-nodes">
               <flight dep="19.40" arr="11.40+1" no="LH9738" op_by_partner="yes" freq="346"
                       stops="0" meals="M"/>
            </connection>
      </con>
   </allcons>
</result>
```

In words: The database contains two connections from Munich to Auckland: one via Hong Kong (11314 miles) and one via Bangkok (11416 miles). Two more results exist over San Francisco and Los Angeles, but they are filtered from the result since the cities are in the United States.

8.5.4 Performance Evaluation

Non-concurrent query evaluation:

in eXist, the query evaluation is non-concurrent. For evaluating a single location step on a given context nodeset, the axis, the nodetest and (eventually) the predicates are applied to the context's first node, then the second, and so on. The $n + 1$th node is not evaluated until the nth node's result is evaluated. For regular XML/XQuery this seems to be a reasonable approach. But consider now the first node of the context set being an XLink: the location step has to "cross" the XLink connection via query or data shipping, the data has to be transmitted via a http connection, which is of course significantly slower. Hence, the effort of following an XLink during evaluation is enormous in comparison to a following a “regular axis”. In the modified eXist implementation, this means that the evaluation is halted until the XLink part is done. *Then*, the evaluation can continue.

Here, an opportune optimization would be to concurrently start the evaluation on all context nodes, and merging the results in document order afterwards until the last result has arrived. This would reduce the theoretical complexity for a node context with nodes $\{node_1, \ldots, node_n\}$ from

$$T(\{node_1, \ldots, node_n\}) = T(node_1) + \ldots + T(node_n)$$

to

$$T(\{node_1, \ldots, node_n\}) = MAX(\{T(node_1), \ldots, T(node_n)\})$$

plus some constant overhead.

When implemented naïvely, this would result into a large number of parallely open http connections for large context nodesets. The effort of opening, closing and maintaining these connections would again be enormous. To avoid this, connection pooling could be an option, so that – at least, inside a single query evaluation – only *one* steady http connection is established between two participating XML servers, with multiple threads sharing one physical http connection.

These modifications alone could supply an enormous speedup, since my experiences during debugging revealed that the significant part of the runtime is used for establishing connections to remote hosts and idle waiting for the remote results. At least, the idle waiting could be reduced using that technique.

Linkbase Representation: Native vs. XML

In the prototype, linkbases are represented using XML documents containing references into the index structure of the local server, stored as attribute values. This was a relatively simple solution, since it mainly involved the usage of already existing functionality as inserting and updating XML data. On the other hand, this is a relatively inefficient way for maintaining a hash map (remember that a linkbase is nothing more than a partial mapping from nodes to from-locators affecting that very node with its xlink:href attribute), since the linkbase data is stored in the XML datamodel, which is again stored in eXist-internal index structures (B+-trees etc.). A more efficient way would be to store the linkbases directly in a similar index/B-tree structure. This would evade numerous serializing and deserializing operations, executed on the materialization of each XML node during a location step evaluation.

8.5.5 Functionality Evaluation

Apart from the performance issues addressed above, the prototype implementation has some shortcomings in evaluation of XQuery statements. This has mainly to do with the embedding of the XLink functionality into eXist’s software architecture: the class DBXLinkProcessor, which is the central class for expanding XLinks during query evaluation, is coupled with the XPathExpr class, which represents an XPath expression from an incoming query. DBXLinkProcessor checks for relevant XLink elements, initializes their expansion / evaluation using query shipping, data shipping or hybrid shipping, and integrates their results into the XPath expressions "regular", which means: non-XLink results. So, all XLink-relevant aspects are processed in the scope of a single XPath expression. Hence, Simple Link and Extended Link evaluation work fine for single, flat XPath expressions.

Problems may arise however, if a query consists of more than a flat XPath expression, especially

- inside nested XPath queries,
- concerning data and hybrid shipping behavior, and
- for variable definition and evaluation.

Example: Nested XPath Expressions

A quite common XPath construct is the id() expression. Consider an expression /a/b/id(c/d/@e). The whole expression is an (absolute) XPath expression, and the argument of if id() is again a (relative) XPath expression.

The XLink expansion is tied to the XPathExpr class, and does not consider XLink information tied to surrounding or included XPath expressions. If, e.g., the c location step would match an XLink child element, which is present in the virtual model, but not in the physical model, the evaluation unit has no chance to find that element. Let us consider just having evaluated the b location step[4]. If the next evaluation step would be another location step, the engine would now check for XLink elements, that possibly could contribute to the next context set. But since the next step is a function call, with a completely independent XPath expression as its argument, the eventual XLink element is not considered relevant. The inner expression /c/d/@e gets the previous context set from its surrounding expression as start context, but none of its XLink meta-information, and has therefore no chance to find possibly relevant XLinks, neither Simple Links nor Extended Links.

This behavior applies to each kind of inner XPath expressions, especially inside function calls, as well as in predicates. Note that this forbids the usage of make-attribute as L-directive for the *to*-locator of an arc, since these can be evaluated only within the id() function call.

Example: Variables

Consider the following XQuery example, applied to the familiar airline scenario:

[4]For a more detailed look into the Simple Link-aware evaluation of XPath queries within the modified eXist version, please have a look at [Beh06]

```
for $con1 in /cities/city[name='Hannover']/connection
   let $city := $con1/city
   for $con2 in $city/connection
   where $con2/city/name = 'Lisbon'
return <con>{$con1,$con2}</con>
```

Let us examine each variable appearance in detail:

- **$con1:** The variable iterates over all connections of the "Hannover" city element, which are given by the linkbase and blended into the virtual instance the usual way. Everything's fine until here.
- **let $city := $con1/city:** the nodeset represented by the $con1 variable binding is the start context for the XPath expression $con1/city. The city location step is evaluated, the evaluation-relevant airport link element is found and expanded. As a result, the link element is evaluated, and the result – the city element where the airport element points to – is evaluated by query shipping and copied into the local instance[5]. So, the city element is expanded in the local virtual instance. Still, everything seems alright up to now.
- **for $con2 in $city/connection:** Now, the nodeset represented by $city is scanned for possible XLink elements. The problem is: the city elements are copied and thereby deprived of their original host, that also maintains the relevant part of the linkbase. In the local linkbase, no information can be found about the new elements. So, the produced result set is empty, which of course is wrong in terms of the logical data model.

8.5.6 Summary

Focusing alone on *performance issues*, the results of the virtual datamodel querying may appear somewhat humbling on first sight, since their runtimes all exceed the physical datamodel queries' runtimes by the factor 100. As depicted above, some reasons for that could be determined:

- non-concurrent query evaluation
- internal linkbase representation as XML

The above described *functional anomalies* concerning XQuery evaluation can be traced to one software design decision:

- XLink processing is bound to the evaluation of XPathExpr.

Let's discuss these points now one by one in order of appearance in the text.

[5]This is not a contradiction to the query shipping directive. Even when evaluating the rest of a query in a remote place, the final results have to be integrated in the local virtual instance, to be used for further evaluation steps.

Non-Concurrent Query Evaluation

When thinking about making eXist’s query evaluation concurrent, one has to spot the code regions that involve active, non-concurrent waiting. The code region where this has an enormous impact on performance is the expansion of Simple Links in DBXLinkProcessor.process():

```
...
while(!relevantLinks.isEmpty()) {
  for (int j = 0; j < relevantLinks.getLength(); j++) {
    ElementImpl link = (ElementImpl) relevantLinks.item(j);
    NodeListImpl xpointerresult = new NodeListImpl();
    // get all nodes to be appended specified by the dbxlink rules
    xpointerresult.addAll(resolveXLink(link));
    // store all collected referenced nodes, mapped to the referencing link
    nodesToBeInserted.put((StoredNode)link, xpointerresult);
  }
  //  changes have to be made persistent
  try {
    context = applyChanges(currentElement);
  } catch (XPathException e) {
    e.printStackTrace();
  }
  // get remaining links that have to be resolved
  relevantLinks = getRelevantLinks(currentElement);
}
...
```

– DBXLinkProcessor.process() –

The creation of an http connection and the shipping of the data / query results over that connection is done in resolveXLink(link) in a strictly sequential way: the connection is opened, the data is sent, the result is received, the connection is closed. The program execution is halted until the remote host has fulfilled his workload and has answered the http request.

A possible way of evading that bottleneck would be to encapsulate the connection opening, query/data transport and connection closing into another class extending the java class java.lang.Thread (or implementing the interface java.lang.Runnable), delegating the task into a new thread for each single connection. java.lang.Thread brings along the functionality for synchronization using monitoring. Synchronization is needed e.g. for waiting until the last thread has completed and *then* gathering the single XLinks’ results.

Since the above described optimizations have no impact on the basic functionality of executing queries over XLinked XML instances, I decided to leave these modifications as subject to further research and optimization work.

Internal Linkbase Representation

Let’s shortly reconsider the linkbase registration process: the central linkbase file (e.g. flightbase.xml) is registered by traversing all from-locators, taking their xlink:href XPointer values, and evaluate these via query shipping. If an XPointer is shipped to a remote host and terminates there – which means, it identifies some nodes there as the XPointer's result – then, the relevant linkbase portion, namely the locator and all arcs having the locator as their from-locator, are transferred to the remote host, copied into the local linkbase part, and endowed with an additional attribute dbxlink:locref. The attribute's value consists of multiple entries, each one consisting of (1) the local document path and (2) the concerned node's nodeID. Later, on evaluating queries on that host with respect to the local linkbase data, each traversed context node is checked if in the local linkbase there is a locator containing the node's document path and nodeID. If so, the arc is evaluated. Of course, storing the local linkbase data is significantly slower than to store it directly in some file-based indexing structure, or in a in-memory hash structure. The problem with the in-memory solution is, that it is not persistent to database reboot[6]. The other solution would result in a complete re-write of the storage unit, completely with B-tree support, indexing schemes, concurrency control, transaction management etc. Even if many of the features could be copied from the regular storage unit, this would result in an amount of effort which seems not to bear a reasonable cost-benefit ratio, neither in terms of scientific relevance nor regarding the effective speedup.

XLink Processing tied to PathExpr

Most functional disabilities of the prototype's evaluation unit have to do with the "location" in the code, where XLink processing takes place. The processing is done mainly in class DBXLinkProcessor, which is a field of class PathExpr. PathExpr represents the XQuery construct of a path expression. Considering the "Variables" example above, this leads to the insight that not only the path expression should be aware of all XLink information, but also its subsequent location steps. Also, a path expression might contain inner path expressions (e.g. a function call or a predicate are parts of a path expression, and may contain zero or more path expression as arguments). For several tasks, especially involving query shipping, it would be necessary to pass information from the inner path expression's DBXLinkProcessor instance to the DBXLinkProcessor instance of the surrounding path expression (see the example: "Nested XPath Expressions" above). This again would have led into serious refactoring of the XLink processing, which I considered to be of subordinate relevance, especially since many queries – at least most of the "common" ones – can be rewritten towards not containing nested path expressions or variable expressions going over XLink boundaries.

[6]In fact, the in-memory variant can be used as an additional feature, for caching some information and storing it in shared memory, e.g. in the servlet context, which is accessible to all servlet instances in the servlet engine. In the prototype, this variant is implemented to accelerate linkbase evaluation.

Chapter 9

Analysis and Discussion

The general idea behind the XLink/LinXIS approach was basically to connect two main concepts of data engineering in a benefit-bringing way: (i) information integration from autonomous sources on the Web, and (ii) the definition of views over XML data, as known from relational databases, where views are defined over the relational data model. On the Web, lots of autonomous data sources exist, maintained by autonomous content supplying parties, supplying data about general aspects of public interest, such diverse as the weather, estate prices, flight plans, movie critics, etc., which could be combined with views. A multitude of scenarios can be thought of that would profit from the definition and implementation of a data model which allows for querying / navigating these views[1].

The World Wide Web is evolving quickly in various aspects: the number and customs of its users, the available bandwidth, the used infrastructure and technologies. What impact do these rapid changes have on the idea of XLink, and on the data model and evaluation techniques presented in this work?

The next sections try to give an analysis of the XLink + LinXIS approach as it is proposed in this work, regarding its relevance and competitiveness concerning its efficiency, functional behavior and applicability, regarding today's Web infrastructure. What was the Web supposed to look like today, back when the XLink standard was developed? And what *does* it look like today? Where is XLink still relevant, and where is it not? In what kinds of scenarios is the use of 3rd Party Link-created views useful and applicable? Does the approach (XLink as well as LinXIS) still make sense at all?

This chapter describes the vision of today's Web infrastructure as it was en vogue during the late 90ies, when XLink was developed (denominating it *the XML Web*) in the context of browsing the Web (Section 9.1) and querying the

[1]It does not need too much fantasy to figure out that, for certain peer groups, there would be an added value in combining data from a weather forecast website with online flightplan data from an airline into a view which then could be queried: "give me all flight connections from my hometown to any place with a sunny weather forecast for the next two weeks", to give a very trivial example.

Web (Section 9.2). Moreover, the then-postulated *XML Web* is compared to present Web technologies as the *Semantic Web* and the *Social Web*. Throughout these sections, some arguments *against* the validity and relevance of the XLink + LinXIS approach are pointed out in form of *objections*. These objections are discussed in detail in Section 9.3. In Section 9.4, an improved software architecture is proposed, based on the preceding discussion and the experiences made with the actual prototype implementation.

9.1 Browsing the Web

One goal throughout the development of XML and XLink was to define a standard which sooner or later should replace, or at least should embrace the HTML as the primary data representation format on the Web. Instead of having lots of – mostly manually edited and maintained – HTML documents containing both content and layout intermixed, the idea was to have an XML *data body* providing the data of a specific data domain in a clear, domain-specific modeling, and to use XSL transformations and CSS layout information for creating a customized, but generic representation for being viewed in a browser. Assuming such an infrastructure, 3rd Party Linking would be a powerful concept for aggregating such data bodies into views, thereby creating new data bodies which then could be queried or browsed. Although this is not a commonly agreed term, let us refer to that infrastructure as *the XML Web*.

The reality today looks a bit different: indeed, most websites are no more written by hand, but created from data models by automated work flows. But the underlying data model is not always the above mentioned XML data body, but the internal data model of some content management system. Which, again, can be XML + XSL, or a relational database system, or any other data model, all with one thing in common: the model is – usually – not available to the public. (X)HTML pages are generated from the internal model and published, but the underlying data model is not made public.

Even websites authored in XHTML are not really adequate for being addressed with XPointer expressions: though XHTML is an XML document type, and all nodes in an XHTML document can be addressed with the `xpointer()` scheme, the XHTML contains data not in a domain-driven representation, but in a representation focusing mainly on layout aspects , which makes it harder and more imprecise to reference the interspersed relevant domain data. Also, Web layout is often subject to frequent change (website relaunches/redesigns etc.), additionally to eventual changes of the underlying domain data. Subsuming the above in a catchphrase, it can be stated:

$$\boxed{\textbf{Objection 1: there is no such thing as an } \textbf{\textit{XML Web.}}}$$

9.2 Querying the Web

9.2.1 Searching the Web: State of the Art

The common approaches for seeking relevant information in the World Wide Web have not changed significantly in the past 15 years. The weapons of choice still are search engines, searching and sorting the Web by creating huge indexes over character sequences, plus more or less efficient ranking heuristics in an effort to present the – often vast – amount of results for a query ordered by their (supposed) relevance for the user. Key mechanism for finding data in the Web is mainly a sophisticated kind of character matching over several millions (if not billions) of hypertext documents and other media with no common data model.

9.2.2 The Semantic Web

The *Semantic Web* is based on the idea to associate data on the Web with a notion of explicit, machine-processable semantics. The semantics of a certain data domain is described with an *ontology.* An ontology consists of *classes*, which represent central *concepts* from the data domain, and of *individuals*, which represent concrete, existing *instances* among the data, with each instance being an instance of – at least – one class. Classes can be described by defining *attributes* for them (attributes describe certain properties, features or characteristics of the classes' individuals), and by relationships to other classes, as *subsumption* (subclass), *partition*, or *disjoint partition.*

The idea is, to formalize such an ontology with an *ontology language* such as OWL [OWL04], and to be able to use *reasoning* over the given ontology concepts as well as individuals with means of *description logic* (a specific subset of first order logic). With the help of a *reasoner* (a software that "understands" description logic), applied to a given ontology, the ontology can be queried for *intensional* knowledge, which is knowledge about classes and individuals that is not directly expressed in the ontology description, but which is *inferred* from the *extensional* (= directly specified) knowledge plus the ontology rules and constraints, defined in description logic. The idea behind the idea is, to be able to formulate a query based on the domain concepts instead of character/string matching. Consider the following example:

Example 9 *Imagine yourself trying to remember details from a recent party talk, where someone else recommended you a book, but you can't remember either the title nor the author's name. You only remember some vague details: that it was the author's 3rd book, and that the author's sister was an actress. To a Semantic Web, you could issue a query like "Give me all books by an author who has published more than 3 books, and whose sister is an actress"*[2].

Data modeling and search techniques using ontologies and description logic are a vivid area of research. Despite the conceptual impressiveness of the Se-

[2]Try to google that!

mantic Web approach, a number of problems arise. One is, that reasoning on ontologies in presence of large data amounts in general is computationally expensive, so that queries based on the ontology using solely reasoning often are not an option at all, when considering problems of a real-world size.

Another problem is the integration of ontologies. Most problem areas involve heterogeneous data from multiple data domains. Even if for each of those domains, an agreed ontology exists, integrating such overlapping domains demands an explicit matching of concepts across the different ontologies, which – by now – can only be done manually by a domain expert, which is a time-consuming and possibly error-prone process.

In spite of their complexity and inherent computational intractabilities, and in spite of unsolved problems in the area of ontology integration, semantic technologies for querying the Web are yet of – at best – moderate relevance outside the scientific community.

9.2.3 The Social Web

In absence of common information models,ontologies tc., applications from the *Social Web* or *Web 2.0* use the domain knowledge of user communities to automatically create implicit *taxonomies*. A simple example are interactive recommender systems e.g. for music ("if you like the music you are listening right now, you will probably also like the following artists:"), with their measure of relevance based on the behavior of a sufficiently large user community. Other examples can be found in *communities of practice*, as Flickr [FLI07] for publishing and annotating photos, or *social tagging* applications for annotating and sharing bookmarks. The resulting *folksonomies* (= folks + taxonomy) are necessarily imprecise and not canonic, since they are created by a community, in contrast to "standards", which a community – hopefully – agrees on, but which are created by single persons or boards with institutional character. Folksonomies, despite being imprecise and non-canonical, offer a way of querying the Web for concepts, rather than for string occurrence.

Newer research efforts aim on combining technological Semantic Web approaches with community-driven Social Web concepts, sometimes referred to as "Web 3.0".

9.2.4 XPath – The Right Choice?

Enhancing the *XML Web* with linking has the striking advantage of XML as a common data model, enabling for querying the Web with query languages based on a precise algebraic data model and bearing precise results, instead of imprecise search based on string matching or ad-hoc community-generated taxonomies.

Let us forget for a moment objection 1, assuming there was an XML Web consisting of distributed, XLink-connected, public XML documents. Would then XLink-enhanced XPath be adequate for exploring, traversing and querying the Web?

XPath consists of location steps, with a location step consisting of an axis identifier, a node test testing the name or type of the context nodes, and zero or more predicates over the traversed XML data. XPath, and especially the axes, have been designed to meet the requirements for navigating single, self-contained, hierarchical XML documents, which, if seen in the algebraic way, are tree structures.

When adding Simple Links to the model, the tree model changes to a graph or network model, which may contain cycles. The phenomenon is known from retrieving HTML pages from the Web, e.g. with the Unix tool wget: each page contains numerous links to remote pages, which might degenerate, when following each link to an arbitrary depth, to the retrieval of all (linked) HTML documents of the World Wide Web.

In XPath, one of the axes is the self-or-descendant axis "//", which identifies all nodes which are, in direct or transitive relation, children of a context node. Which means, all children of the context node, all the grandchildren, all the children's children, and so on. A tree's depth is an upper bound for the length of this line of ancestors. In an arbitrary graph network, there is no such upper bound, since a graph has no depth. In the presence of cycles, there only might be a "longest non-cyclic path".

Consider now an XPath query //*, issued to any document in our "XML Web". The query would return every element in the document from the root element down to each leaf element, following also every Simple Link defined in the document (except Simple Links specifying make-attribute), following every Simple Link given in the linked documents, and so on. The query would result in returning a sufficiently large part of the known XML Web. Even cycle detection and the search for a transitive closure would not change the fundamental absurdness of formulating such a query, which is nevertheless completely legal XPath. Other problems come up when regarding backward axes as the parent axis (are the physical oder the virtual parents to be considered the correct ones?), which is discussed in [Beh06]. Which leads to

Objection 2: (Full) XPath is not adequate for querying the Web.

Note that even if objection 2 is based on the assumption of having only Simple Links, it is trivial to conclude that, if it holds for Simple Links, it might as well hold for Extended Links.

9.2.5 Implementation Aspects

When regarding the implementation of the XLink + LinXIS approach as it is described in 7, a few things can be figured out which conflict with the idea of querying the Web in presence of 3rd Party Links. One is: the introduced infrastructure is based on distributing and registering (= precomputing) a linkbase's information across all affected document locations. Assuming now the initial scenario of thousands (or millions) of users creating personal linkbases and registering them across the XML Web in hundreds of XLink-enhanced XML servers, each server would have to maintain and evaluate thousands of linkbases at the same time. Also, each server would need the functionality to authenticate each user who sends a query, ensuring that *his* query is evaluated with *his* linkbase, and keeping all other users' linkbases secret for him. This would result in an enormous workload for each Web server, which scales at least linear to the (increasing) number of users in the Web, which is a very disadvantageous performance behavior for a distributed system. Still, the prototype's performance with even one user is far below competitive, as can be derived from Figures 8.3, 8.4 and 8.5. Which leads to

Objection 3: Evaluating XPath wrt. 3rd Party Links is prohibitively expensive, dumping most effort on the server side.

9.3 Discussion: Facing the Objections

In the sections above, a number of objections have been given, which will be discussed here, together with some perspectives how to solve the described problems, and how to adopt the XLink approach – and the own perspective to it – in order to be compliant with today's rapidly changing Web infrastructure.

9.3.1 Objection 1: No, there is no XML Web. But:

Today's Web infrastructure is a heterogeneous mixture of techniques and data models, which is far from the *XML Web* as referred to above. This is no surprise at all, since heterogeneity lies in the nature (and is one of the reasons of the success) of the World Wide Web. But where the World Wide Web in the early to mid 90ies was populated mainly by humans providing (and consuming) content in shape of HTML resources, the Web is more and more evolving to a medium where data is not only produced and consumed by persons, but where the users are – with increasing rates – goal-driven agents or peer-to-peer software components exchanging data autonomously. These actors – consider e.g. the quickly emerging sector of Web Services and grid computing applications – provide and exchange data in domain-specific representations, often using RDF or XML as interchange format.

Examples for XML-based data exchange standards are newsfeed techniques as *RSS* (*Really Simple Syndication*, see [RSS07]) or *Atom* [ATO07]. Moreover, Linking XML can make sense in smaller contexts than the complete World Wide Web, e.g. in smaller subsets such as corporate intranets or *private virtual networks* (*VPNs*), with a closed and controlled domain of information sources and users. The conclusion that can be drawn is: there is no *XML Web*, but there is a lot (and a growing amount) of XML on the Web.

9.3.2 Objection 2: No, full XPath is not Adequate for Querying the Web. But:

Some XPath concepts, such as the descendant(-or-self) axes, make only sense inside non-interlinked XML document trees, but not within open and structurally unrestricted graph structures, possibly traversing the whole Web. Here, an option is to use an XPath subset for "Web queries". A reduced XPath language can be thought of, containing only the self, child, attribute, and namespace axes. The restriction from 12 to 4 axes is not that harsh as it seems on the first glance: Backward axes (parent, ancestor, ancestor-or-self, preceding-sibling, preceding) can be replaced by forward axes (see [OMFB02]), and following-sibling and following axes are only relevant when considering the document order of XML documents. For data-centric XML, the document order is not of much concern (if necessary, it can be represented explicitly in the data anyway). So, full XPath might not be adequate for querying the Web, but it also might be sufficient to restrict it to an as well essential as viable subset, still enabling for issuing expressive queries over linked XML data.

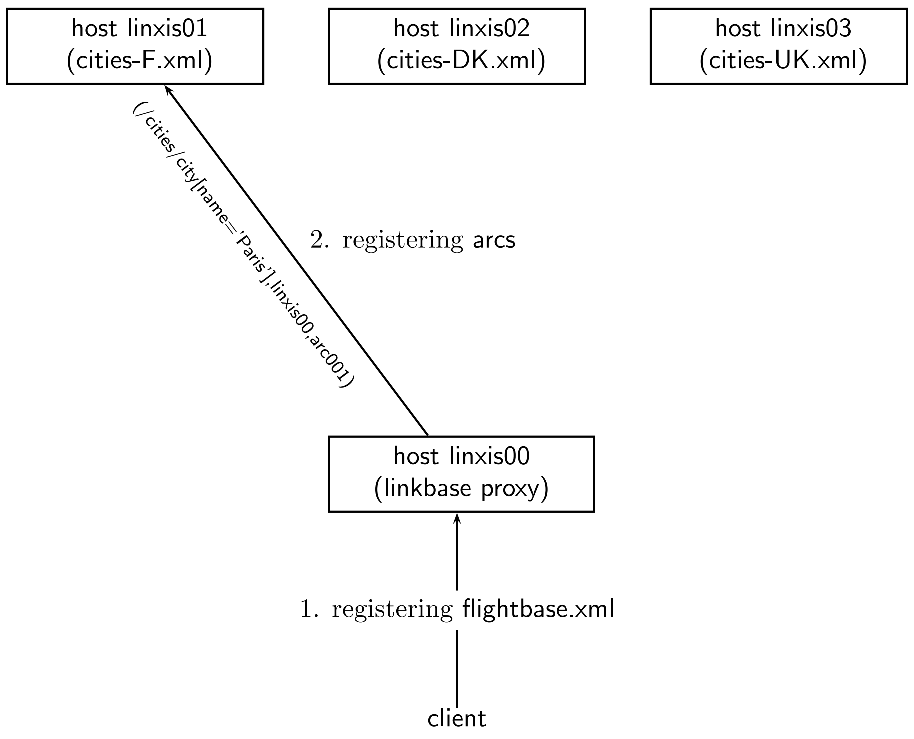
9.3.3 Objection 3: Yes, 3rd Party Links are prohibitively expensive. But:

The introduced XLink-database prototype is based on the eXist database system. eXist was chosen as code basis since it is (a) a native XML database system written in Java, offering full XPath and XQuery support, (b) it supports a wide range of protocols as HTTP, REST, SOAP, XML-RPC etc., and (c) it is Open Source. However, eXist is not designed as a distributed 3rd-Party-Link-aware database system, and the software architecture of the prototype was a compromise between an implementation of the XLink evaluation strategies, and between the given eXist software architecture. Designing a completely new prototype from scratch could result in a much more efficient and consequent architecture (but would take significantly more time and cost than available). A snapshot of such an architecture is described in the following section.

9.4 Proposal for an Improved Architecture

With the experiences and observations that have been made during the development of the prototype, a few things can be said: the software' performance is rather weak, and scales badly with the number of linkbase users (discussed in the previous section). This raises the question: can we do any better? Can we think of an improved prototype with better performance, and how could such a prototype look like? Which would be the requirements? It should be able to process Simple Links as well as linkbases. Which should be the design principles? For a competitive overall performance, a strict peer-to-peer approach would be favorable, involving strict query-shipping, to minimize the effort of transmitting large portions of XML data, aiming at processing as much of a query as possible right at the server where the queried data is located. In order to keep as much workload as possible away from common XML servers (which provide XML data, and which can answer XPath queries by Query Shipping), the efforts for checking for node traversals on one hand – this *must* be done by the common servers – and answering and linkbase management on the other hand should be disseminated.

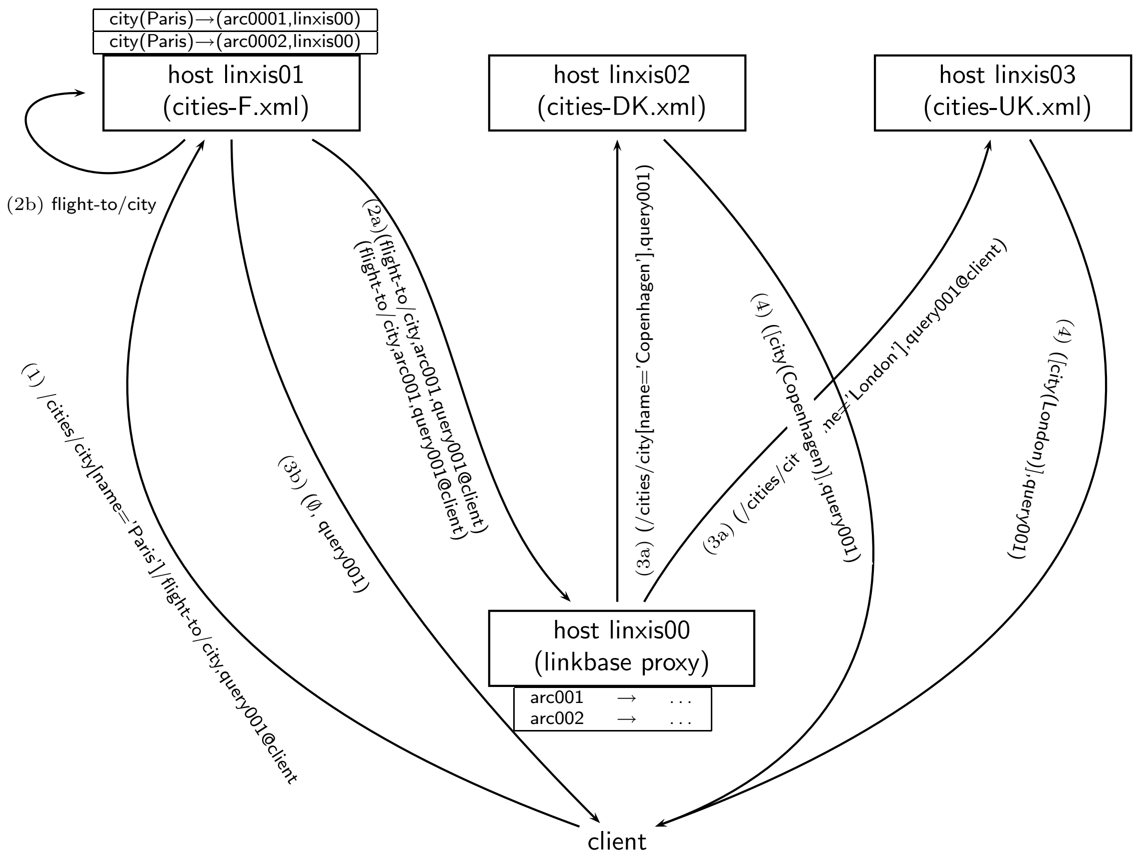
The proposed software architecture is sketched in Figures 9.1 and 9.2. The necessary components are a client that is able to merge single XPath results, coming from distributing a query by Query Shipping, back together, a linkbase proxy for maintaining, brokering and evaluating the linkbase information, and regular, XPath-aware XML servers where XPath expressions can be "registered" in a way that a message is sent to the linkbase if a certain registered node has been traversed by a query. The linkbase server continues the evaluation along the XLink "axis", sending the results back to the client (where the results are possibly merged with the non-XLink results of the common XML server(s)).



1. A flightplan linkbase with connections Paris (F) → Copenhagen (DK) and Paris (F) → London (UK) (and others) is registered at the linkbase proxy by a client.

2. For each arc, it's from-locator is registered at the servers.

Figure 9.1: Peer-to-peer architecture part 1: registering a linkbase


city(Paris)→(arc0001,linxis00)
city(Paris)→(arc0002,linxis00)
host linxis01 (cities-F.xml)
host linxis02 (cities-DK.xml)
host linxis03 (cities-UK.xml)
(2b) flight-to/city
(2a) (flight-to/city,arc001,query001@client) (flight-to/city,arc001,query001@client)
(3a) (/cities/city[name='Copenhagen'],query001)
(3a) (/cities/city[name='London'],query001@client)
(4) ([city(Copenhagen)],query001)
(4) ([city(London)],query001)
(1) /cities/city[name='Paris']/flight-to/city,query001@client
(3b) (∅, query001)
host linxis00 (linkbase proxy)
arc001 → ...
arc002 → ...
client

1. the client issues the XPath query /cities/city[name='Paris']/flight-to/city to server linxis01, together with a unique query id (which is always transmitted together with the query or parts of it).
2. server linxis01 evaluates location steps /cities/city[name='Paris']. On finding the city node for Paris in its linkbase index (with the arc marked with insert placement), linxis01 ships the remaining query part to the linkbase proxy, together with an id for the arc (at the proxy) and the query (at the client) (2a), and evaluates the remaining query part also locally (2b). (for replace placement, only (2a) is performed)
3. The rest query flight-to/city is evaluated relatively to the 2 arcs registered for the Paris node (representing the connections to Copenhagen and London), and combined with their to-locators' addresses, http://linxis02/cities-DK#xpointer(/cities/city[name='Copenhagen']) and http://linxis03/cities-UK#xpointer(/cities/city[name='London']), resulting in the queries /cities/city[name='Copenhagen'] and /cities/city[name='London'] being shipped to linxis02 and linxis03 (3a).
   The query flight-to/city yields no result when issued locally on linxis01, and an empty nodeset is sent to the client (3b).
4. The new rest queries /cities/city[name='Copenhagen'] and /cities/city[name='London'] are evaluated at linxis02 and linxis03. The results (the Copenhagen and London city elements) are sent back to the client.
5. The received city nodes are merged with other results that have the same query id (here, it is only the empty nodeset from linxis01), in order to assemble the final query result:
   [<city><name>Copenhagen</name>... </city>, <city><name>London</name>... </city>]

Figure 9.2: Peer-to-peer architecture part 2: evaluating an XPath query

Chapter 10

Conclusions and Outlook

In this work, a logical data model for Extended Links has been introduced. The idea was to present a semantics for integrating XML data sources distributed over the Web into a single, personalized view (i) by means of a precise, formal and coherent data model, which (ii) provides means for a modeling which is expressive enough to be useful, and simple enough to be realizable, and (iii) which is compliant with the syntax of the W3C XLink standard for expressing inter-document links.

This chapter gives a brief overview over the related research work over linking and querying XML in Section 10.1. Section 10.2 presents a résumé on the main scientific contributions of this work, with respect to the discussion in the previous chapter. Finally, Section 10.3 highlights some aspects of the work that still deserve deeper investigation, and opens perspectives on further work in the area of 3rd Party Links.

10.1 Related Work

10.1.1 Views in XML

Numerous publications exist on the issue of defining, querying and maintaining XML views: in [AMR+98], update maintenance of materialized views in OEM (a semistructured data language, considered by many as the main precursor of XML) is investigated; views are defined in Lorel, a query language with similarities to XQuery. In [SKS+01], techniques are described for expressing and querying XML views over relational data, especially regarding how to translate XQuery expressions on the view to SQL expressions on the underlying relational data. [BDH04] presents algorithms for propagating updates made to an XML view to its underlying set of relational views. In [BLP+98], requirements for an XML view definition language are formulated, based on a (pre-XQuery) XML mediator infrastructure. In [AMN+01], typechecking of XML views over relational data is investigated regarding as well its technical as its computational aspects.

10.1.2 Querying Distributed XML Data

In [Suc02], the distributed evaluation of path expressions on XML data is investigated, considering an architecture of a (fixed) set of federated repositories, using an algebraic representation based on bisimulation, and focusing on computational aspects such as upper bounds for runtime complexity. The graph-based data model includes ϵ-edges, which have the same functionality as (and can be expressed by) Simple Links.

Active XML [AXM02] is a framework extending XML by introducing active elements, which enable including Web Services to provide dynamic content for XML documents. Considering a Web Service which offers access to XML data by XPath queries, this provides the functionality of resolving and materializing Simple Links. In [ABC$^+$04], a method for lazy evaluation is described for materializing only those service calls that are traversed within a *tree pattern query* (a construct considered to subsume XPath queries). *Active XML* is focused on providing means and infrastructure for integrating dynamic (but typed) Web Service calls into static XML data by materialization (where the LinXIS approach, in contrast, is focused on delivering a logical data model for stepwise evaluation for Simple Links).

10.1.3 XLink

In [CFRV02], an HTTP proxy server for processing external linkbases using XLink's Extended Links is proposed. The *XLinkProxy* annotates Web resources in XML with respect to a given linkbase by merging the relevant linkbase information into the annotated document, and supplies the assembled document to a browser. The focus is on browsing XML hypertext.

In [LL03], the XML representation *SXML* is developed on the basis of the functional language *scheme*, together with the extension *SXLink* for dereferencing Simple Links as well as linkbases with Extended Links, based on explicit dereferencing operators. Here, the focus is set on describing a DOM-like in-memory structure representing multiple, interlinked XML instances with functional programming.

[NCEF02] describes the framework *XLinkIt* for defining constraints and discovering inconsistencies among XML data. The results are represented using Extended Links.

Amaya [Ama] is an experimental Web browser and editor. It is an Open Source project hosted by the W3C. It supports RDF and Simple Links as well as linkbases defined by Extended Links within the context of browsing XML / XHTML documents.

10.1.4 Summary on Related Work

Much of the work found about *views in XML* focuses on coupling concepts of querying semistructured data with common techniques for view definition and maintenance in relational databases, or to solve these problems (e.g. view maintenance wrt. updates) for semistructured views by mapping them to the domain of relational databases, where view management is a common and well-understood research area. The remaining publications, addressing pure XML views, mostly consider the management of materialized views defined by XQuery expressions or other, proprietary, kinds of view definition.

For publications about *querying distributed XML data*, two main focuses can be determined: one is the algebraic and computational aspects of distributed query evaluation, another one is about infrastructural aspects towards providing dynamic XML contents by embedding service calls. These approaches all incorporate a notion analogue to XLink's Simple Links.

The literature dealing explicitly with the *XLink standard* uses it mainly for browsing purposes, representation format or navigation on the materialized data structure in DOM-style.

Summarizing the above, it can be said that the research efforts that have been done for explicitly bringing together the concepts (and standards) of linking XML (with W3C's XLink as standard) and querying XML (with W3C's XPath/XQuery) are relatively modest. Much work is centered around creating, querying and maintaining views in XML, but none of them is based on linkbases with 3rd Party Links in XLink syntax for describing a view's properties. The (rare) use of linkbases is mostly limited to the context of browsing.

10.2 Contribution

XLink, introduced by the W3C as the designated standard for Linking XML documents, and XQuery, the W3C's designated (and widely-used) standard for querying XML data, exist independently from each other, despite in the W3C XML Query Requirements [XMQ05][1], it has been stated as an explicit requirement that

> "3.4.12 References: Queries *must* be able to traverse intra- and inter-document references."

Up to date, this requirement has only partially been met. An explicit dereferencing operator has been dropped. XPointer support was considered, but not provided by the XQuery work group.

The presented work aims at closing the conceptual gap between these two – up to date largely isolated – main XML issues: between XML querying and XML linking. A logical data model is presented, describing a mapping from linked XML to plain XML, which enables for processing XLink-connected XML data, providing rich modeling features for customizing the "cutting edges" between two XML resources. The logical data model thus provides a semantics for querying linked distributed XML data. A prototype implementation of a query engine is presented, implementing as well Simple Link as 3rd Party Link

[1]The XML Query Requirement from 2005 has been superseded by a new version in spring 2007, which still contains the above quoted requirement for inter-document traversal.

semantics, validating the introduced data model by providing a proof of concept, finding that the semantics is adequate for splitting up and integrating distributed XML data across multiple Web locations, and enables for creating 3rd Party views.

Based on the discussion in 9, it can be said that the LinXIS approach is of conceptual relevance for a number of reasons:

- it is based on the W3C standards *XLink* and *XQuery*,
- a formal definition and a prototype implementation are provided, demonstrating the basic validity of the LinXIS approach, and
- LinXIS represents an approach for data integration on the Web based on linked XML.

LinXIS might not be the only approach for XML-based data integration on the Web. Other approaches exist, but up to now, none of them succeeded in defining a "standard of practice", legitimated by commercial relevance and/or approval by a large user community. Still, XML is the most common way of publishing data on the Web with a structural, machine-processable data model (even regarding the steadily growing amount of RDF data on the Web).

Apart from speaking of LinXIS as of closing a conceptual gap: with creating a technique for connecting previously isolated XML data repositories on the Web based on W3C technology, enabled for being queried – and in the absence of any other solution of significantly higher relevance – with LinXIS, we're supposed to be as good as it gets.

10.3 Outlook

As stated above in Section 9.3, the presented prototype serves as a proof-of-concept implementation, but is – in terms of performance – not appropriate for as a basis for a large-scale Web-wide-distributed infrastructure. Considering the sketch of an alternative implementation architecture from Section 9.4 in the previous chapter, it can be stated that some performance bottlenecks specific to the prototype's design (which was driven by pure feasibility considerations and dictated by the time schedule) can be evaded. Thus, with the architecture described there, and with a reduced version of XPath, 3rd Party Links as well as Simple Links can be implemented, assuming an infrastructure of XML servers as described above, resulting in a significantly faster software infrastructure for Simple Links as well as 3rd Party Links. Other possible areas of research are the investigation of query optimization, query caching, schema processing and link typing in context of the described LinXIS approach, as well as discovering and investigating new application areas for evaluating queries in presence of 3rd Party Links.

Bibliography

[ABC$^+$04] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy query evaluation for active xml. In *SIGMOD*, pages 227–238, 2004.

[Abi97] S. Abiteboul. Querying Semi-Structured Data. In *Intl. Conference on Database Theory (ICDT)*, number 1186 in LNCS, pages 1–18. Springer, 1997.

[Ama] W3c's editor/browser. `http://www.w3.org/Amaya/`.

[AMN$^+$01] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. Typechecking xml views of relational databases. In *LICS*, pages 421–430, 2001.

[AMR$^+$98] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. L. Wiener. Incremental maintenance for materialized views over semistructured data. In A. Gupta, O. Shmueli, and J. Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 38–49. Morgan Kaufmann, 1998.

[ATO07] The ATOM Syndication Format. `http://www.atompub.org/rfc4287.html`, 2007.

[AXM02] Active XML Primer, 2002. `http://www-rocq.inria.fr/gemo/Gemo/Projects/axml/`.

[BDH04] V. P. Braganholo, S. B. Davidson, and C. A. Heuser. From xml view updates to relational view updates: old solutions to a new problem. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, editors, *VLDB*, pages 276–287. Morgan Kaufmann, 2004.

[Beh06] E. Behrends. *Evaluation of Queries on Linked Distributed XML Data.* PhD thesis, Universität Göttingen, Germany, 2006.

[BFM06a] E. Behrends, O. Fritzen, and W. May. Handling Interlinked XML Instances on the Web. In *Intl. Conference on Extending Database Technology (EDBT)*, number 3896 in LNCS, pages 792–810, 2006.

[BFM06b] E. Behrends, O. Fritzen, and W. May. Querying along XLinks in XPath/XQuery: Situation, Applications, Perspectives. In *EDTB Workshops. WS Query Languages and Query Processing (QLQP)*, number 4254 in LNCS, pages 662–674, 2006.

[BLP$^+$98] C. K. Baru, B. Ludäscher, Y. Papakonstantinou, P. Velikhov, and V. Vianu. Features and requirements for an xml view definition language: Lessons from xml information mediation. In *QL*, 1998.

[BR04] T. Böhme and E. Rahm. Supporting Efficient Streaming and Insertion of XML Data in RDBMS. In *DIWeb*, pages 70–81, 2004.

[CFRV02] P. Ciancarini, F. Folli, D. Rossi, and F. Vitali. Xlinkproxy: external linkbases with xlink. In *ACM Symposium on Document Engineering*, pages 57–65. ACM, 2002.

[DOM98] Document object model (DOM). `http://www.w3.org/DOM/`, 1998.

[exi] eXist: an Open Source Native XML Database. `http://exist-db.org/`.

[FLI07] `http://www.flickr.com` Photosharing. `http://en.wikipedia.org/wiki/Flickr`, 2007.

[IAT07] The International Air Transport Association (IATA) airport code. `http://en.wikipedia.org/wiki/IATA_Airport_Code`, 2007.

[jet07] Jetty. Jetty, a lightweight webserver written in Java, `http://jetty.mortbay.com/`, 2007.

[Kay] M. Kay. SAXON: an XSLT processor. `http://saxon.sourceforge.net/`.

[LH07] LH.pdf. The Lufthansa Time Table / Der Lufthansa-Flugplan, `http://www.lhtimetable.com/LH.pdf`, 2007.

[Lin] The LinXIS Project. `http://dbis.informatik.uni-goettingen.de/LinXIS/`.

[LL03] K. Lisovsky and D. Lizorkin. XSLT and XLink and their implementation with functional techniques. *Russian Digital Libraries Journal*, 6(5), 2003.

[May02] W. May. Querying linked XML document networks in the web. In *11th. WWW Conference*, 2002. Available at `http://www2002.org/CDROM/alternate/166/`.

[May07] W. May. The Mondial database, 1999–2007. `http://dbis.informatik.uni-goettingen.de/Mondial/`.

[NCEF02] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a consistency checking and smart link generation service. *ACM Trans. Internet Techn.*, 2(2):151–185, 2002.

[OMFB02] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *EDBT Workshop "XML-Based Data Management" (XMLDM)*, pages 109–127, 2002.

[OWL04] OWL Web Ontology Language. `http://www.w3.org/TR/owl-features/`, 2004.

[RBHS04] C. Re, J. Brinkley, K. Hinshaw, and D. Suciu. Distributed XQuery. In *Workshop on Information Integration on the Web (IIWEB)*, 2004.

[RSS07] The RSS (Really Simple Syndication) Specification 2.0.10. `http://www.rssboard.org/rss-specification`, 2007.

[RTF07] The Microsoft Rich Text Format 1.7. `http://support.microsoft.com/kb/86999`, 2007.

[SKS+01] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. E. Funderburk. Querying xml views of relational data. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *VLDB*, pages 261–270. Morgan Kaufmann, 2001.

[Suc02] D. Suciu. Distributed query evaluation on semistructured data. *ACM Transactions on Database Systems (TODS)*, 27(1):1–62, 2002.

[Uri98] Uniform Resource Identifier. `http://www.ietf.org/rfc/rfc2396.txt`, 1998.

[URL] IETF-URL – the uniform resource locator (url). `http://tools.ietf.org/html/rfc1738`.

[W3C] W3C – The World Wide Web Consortium. `http://www.w3.org/`.

[WL02] E. Wilde and D. Lowe. *XPath, XLink, XPointer, and XML: A Practical Guide to Web Hyperlinking and Transclusion.* Addison Wesley, 2002.

[XIn06] XML Inclusions (XInclude). W3C Recommendation, `http://www.w3.org/TR/xinclude/`, 2006.

[XLD97] XLink first Draft. `http://www.w3.org/TR/WD-xml-link-970731`, 1997.

[XLi01a] XML Linking Language (XLink). `http://www.w3.org/TR/xlink`, 2001.

[XLi01b] XML Linking Language (XLink) Version 1.0. W3C Recommendation, `http://www.w3.org/TR/xlink`, 2001.

[XML98] Extensible Markup Language (XML). `http://www.w3.org/XML/`, 1998.

[XMQ03] XML Query Requirements. `http://www.w3.org/TR/xmlquery-req`, 2003. Work in progress.

[XMQ05] XML Query Requirements. W3C Working Draft, `http://www.w3.org/TR/xmlquery-req`, 2005. Work in progress.

[XPa99] XML Path Language (XPath) Version 1.0. W3C Recommendation, `http://www.w3.org/TR/xpath`, 1999.

[XPQ07] XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Recommendation, `http://www.w3.org/TR/xquery-operators`, 2007.

[XPt02a] XML Pointer Language (XPointer). `http://www.w3.org/TR/xptr`, 2002. Work in progress.

[XPt02b] XPointer xpointer() Scheme. `http://www.w3.org/TR/xptr-xpointer`, 2002. Incorporated into [XPt03b].

[XPt03a] XPointer element() Scheme. `http://www.w3.org/TR/xptr-element`, 2003. Incorporated into [XPt03b].

[XPt03b] XPointer Framework. W3C Recommendation, `http://www.w3.org/TR/xptr-framework`, 2003.

[XPt03c] XPointer xmlns() Scheme. `http://www.w3.org/TR/xptr-xmlns`, 2003. Incorporated into [XPt03b].

Curriculum Vitae

Personal Data

Date and place of birth:	19.11.1974 in Trier, Germany
Nationality:	German

[professional]	
1.10.2003 – present	Research Assistant (*Wissenschaftlicher Mitarbeiter*) in the Database and Information Systems group at the Universität of Göttingen, Germany.
1.3.2002 – 30.9.2003	Research Assistant at the Chair of Computer Science 5 (Information Systems & and Database Technology), Rheinisch-Westfälische Technische Hochschule Aachen, Germany
[studies]	
24.9.2001	Graduation in Computer Science (Diplom-Informatiker) from the University of Trier
1999 – 2000	Student Researcher (*Studentische Hilfskraft*) for project SB-PRAM (Parallel Programming in Fork)
1998 – 1999	Student Researcher (*Studentische Hilfskraft*) for project DBLP (Bibliographic Information System for Computer Science Publications)
1997 – 1998	Teaching Assistent (*Studentische Hilfskraft*) for Automata Theory and Formal Languages
1994 – 2001	Student of Computer Science (*Informatik*) at the University of Trier, Germany
[school]	
10.6.1994	Graduation *(*Abitur)
1985 – 1994	Secondary school Friedrich-Wilhelm-Gymnasium in Trier.
1981 – 1985	Elementary school Grundschule Trier-Olewig in Trier.

Languages:

German:	native
English:	fluent
French:	basic knowledge