Indexing Collections of XML Documents with Arbitrary Links

Dissertation

zur Erlangung des Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

vorgelegt dem Fachbereich 5

Institut für Informatik und Wirtschaftsinformatik

der Universität Duisburg-Essen (Campus Essen)

von

Awny Abd El Hady Ahmed Sayed, geboren in El Minia, Ägypten

September 2005

Datum der mündlichen Prüfung: 27. September 2005

Erstgutachter: Prof. Dr. Rainer Unland

Zweitgutachter: Prof. Dr. Michael Goedicke

I dedicate this work to my parents,
my wife and my Children

Acknowledgement

First of all and the greatest important, I would like to thank my GOD for all his blessings without which nothing of my work would have been done.

I would like to thank my supervisor Prof. Dr. Rainer Unland, for his inspiring and encouraging way to guide me to a deeper understanding of knowledge work, and his invaluable comments during the whole work with this dissertation. Without his encouragement and constant guidance, I could not have finished this dissertation. He was always there to meet and talk about my ideas, to proofread and mark up my papers and chapters, and to ask me good questions to help me think through my problems. Beside my supervisor, I would like to thank the other members of my thesis : Prof. Dr. Michael Goedicke and Prof. Dr. Bruno Müller-Clostermann. I am also grateful to Dr. Ralf Schenkel (Max-Plank Institute for Computer Science, Saarbrücken) for his contributions and suggestions on this work.

Thanks also to all my colleagues at Institute for Computer Science and Business Information Systems for providing a good working atmosphere. Thanks also to the Egyptian government for providing me with the financial support for my scholarship.

Last, but not least, I thank my family: my parents, my brother, my sisters, my beloved wife, my kids, who serve as an inspiration for me to move on against all odds on my way.

Awny Sayed

Abstract

In recent years, the popularity of XML has increased significantly. XML is the extensible markup language of the World Wide Web Consortium (W3C). XML is used to represent data in many areas, such as traditional database management systems, e-business environments, and the World Wide Web. XML data, unlike relational and object-oriented data, has no *fixed schema known in advance* and is stored separately from the data. XML data is self-describing and can model heterogeneity more naturally than relational or object-oriented data models. Moreover, XML data usually has XLinks or XPointers to data in other documents (e.g., global-links). In addition to XLink or XPointer links, the XML standard allows to add internal-links between different elements in the same XML document using the ID/IDREF attributes.

The rise in popularity of XML has generated much interest in query processing over graph-structured data. In order to facilitate efficient evaluation of path expressions, structured indexes have been proposed. However, most variants of structured indexes ignore global- or interior-document references. They assume a tree-like structure of XML-documents, which do not contain such global-and internal-links. Extending these indexes to work with large XML graphs considering of global- or internal-document links, firstly requires a lot of computing power for the creation process. Secondly, this would also require a great deal of space in which to store the indexes. As a latter demonstrates, the efficient evaluation of ancestors-descendants queries over arbitrary graphs with long paths is indeed a complex issue.

This thesis proposes the HID index (2-Hop cover path Index based on DAG) is based on the concept of a two-hop cover for a directed graph. The algorithms proposed for the HID index creation, in effect, scales down the original graph size substantially. As a result, a directed acyclic graph (DAG) with a smaller number of nodes and edges will emerge. This reduces the number of computing steps required for building the index. In addition to this, computing time and space will be reduced as well. The index also permits to efficiently evaluate ancestors-descendants relationships. Moreover, the proposed index has an advantage over other comparable indexes: it is optimized for descendants- or-self queries on arbitrary graphs with link relationship, a task that would stress any index structures. Our experiments with real life XML data show that, the HID index provides better performance than other indexes.

Contents

List of Figures

List of Tables

1

Introduction

1.1 Motivation

Recently, XML (eXtensible Markup Language) has become the universal data exchange format for integrating and exchanging data over Intranets and the Internet. It is called “extensible” because its tags are unlimited, self-describing, and irregular. An XML document consists of a set of elements, which are hierarchically structured. The hierarchy is defined by the user. Each element has a name (e.g., “A” for Author), which is also defined by the user. The data an element contains can be stored inside the element, delimited by its start end tags. It can also be stored as a value in the element’s attribute. Certain attribute value types are specifically reserved for referencing (e.g., ID/IDREF). An XML element is typically accessed using the XPath language. XPath queries traverse its input document using a number of location steps. For each step, an axis describes which document nodes form the intermediate result for this step. For example, the path expression */film/actor/name* accesses the *name* node from the root node *film*, and the child node *actor* (parent-child relationship).

An Internet search engine (e.g., Google, Altavista or Infoseek) returns thousands of so-called “matched documents” from a single query, some of which are relevant and others irrelevant to the query. End users usually have problems with organizing and digesting such vast quantities of information, in which much of the information retrieved, is likely to be irrelevant. XML holds the promise that searching can be done more precisely because structural, self-describing information and meta-data (e.g., RDF) are available to allow for context-based and/or category-based searches. In addition to this, XML offers the possibility of modeling heterogeneous data, generated from databases. In turn, this would enable search engines to locate and process heterogeneous documents or records. It is for this reason, that XML documents were previously used for exchanging data within different applications. The complete information was contained in a single document. Nowadays, XML is used as a replacement for HTML on the Web, or Intranets instead. For this reason, documents usually have XLinks or XPointers to data in other documents. The type of links referred to above are called "global-link". In addition to "global-link", the XML standard allows for the addition of "internal-document" links. Internal-document links relate elements within a single document (e.g., using attributes of type ID and IDREF). As a result, the information needed to evaluate a query may be spread over several linked documents. That is to say, when evaluating path expressions in queries, the XML search engine should treat elements that serve as references and follow these references to other documents. In future, it will be the big challenge for the XML retrieval to efficiently evaluate queries with path expressions and promise the end-user(s) to avoid the problem of the irrelevant information to their query.

XML permits a specification, which can facilitate data exchange and that can be used by multiple applications. However, XML documents that come from different applications do not have the same structure and vocabulary, even if they refer to similar domains. A tag <computer> in one XML document might be <laptop> in another XML document. This demonstrates that until now, no universal standard exists in which to represent data in XML on the web. The absence of a general schema in XML leads to the employment of structural summaries derived from the data. This in turn facilitates the tasks such as indexing and querying that would benefit from such a schema. These structural summaries can play an important role in query evaluation, because they answer queries from the summaries, instead of considering the original data. DTDs and XML Schemas are such tools for describing the structural of the XML documents. A DTD serves as a "grammar" for the underlying XML data and it is part of the XML language. Most notably, it is a context-free grammar for XML documents.

1.2 Problem Description

As mentioned previously, XML data has two important characteristics. Firstly, there is no general schema known in advance. Secondly, XML documents usually have pointers to the data in other documents (e.g., XLinks and XPointers), as well as IDs and IDREFS between elements within the same XML document. These two characteristics play an important role in efficient XML query processing. Thus, it has become important to address the question of how one can efficiently index, query, and search large collections of XML documents.

To understand what exactly the problem is when no schema is present. One can first consider traditional relational and object-oriented database systems to see how they query data. Database management systems of that kind force all data to adhere to an explicitly specified schema or a predefined schema. This predefined schema has the following purposes:

1. A predefined schema, in form of either tables and their attributes or class hierarchies, helps users to have all necessary information about the underlying database in order to form significant queries over it.
2. A query processor that based on the schema can compute efficient plans to get the query results.

These two tasks become very difficult when the schema is absent. Moreover, a lack of information about the structure of the database causes a query processor to resort to exhaustive searches.

Apart from that, the main structure of the XML document is tree-like (comprising element-subelement relationships). However, there is also a way to define the element-element relationships in that tree using an ID/IDREF links, or between several trees using XLinks and XPointers. Such link relationships transform the tree-like structure into a graph-like structure, which may even contain cycles. Indeed, when taking ID/IDREF attributes or XLink and XPointer constructs into account two problems arise:

1. The structure of an XML document is no longer tree-like but instead forms a directed graph, which may contain arbitrary cycles. In fact, due to the possible inter- document references a number of smaller XML documents may merge into a larger XML document so the cycles can exist on the level of this spanning.
2. Links generate large sets of connected elements, which may have long paths between them. Thus, to efficiently evaluate path expression queries (especially those with Wildcard "//") an appropriate index structure is needed.

From the above discussion, three important questions arise. The first question is, *what is the best way to test the reachability between two given nodes over this large graph with long paths?* This is also known as *ancestor-descendant structural* relationship. The second question is, *how can path expression queries along the descendant-or-self axis ("//" axis) be efficiently evaluated?* The third question is, *how can path expression queries along the ancestor-or-self axis ("//" axis) be efficiently evaluated?* Especially, in the case of the XML graph which has cycles that can stress any path index. Thus, an appropriate index structure is needed.

This thesis proposes an efficient path index called HID index (2-Hop cover based on path Index for DAG). It can efficiently address these three questions and handle path queries with wildcards on complex XML documents with arbitrary links.

1.3 Solution Overview

The query processing in XML database always involves determining ancestor-descendant structural relationships in addition to parent-child relationships, since the user may not know exactly the structure of the XML document in the absence of the schema. In the "navigation-based query processing", the nodes that match with the ancestor have to be kept for a long time to wait for matching the descendants. In the "index-based query processing", there are two options: one is to maintain the parent-child relationships and compute ancestor-descendant relationships through repeated joins. This however, takes too much time. The other is to maintain all ancestor-descendant relationships, which will lead to much space cost.

This thesis proposes a HID index that makes the checking of the ancestor-descendant structural relationships as easy as checking parent-child relationships. It can easily evaluate all ancestors or descendants of a given node over large graphs with long paths.

The HID index is based on the idea that, if a label is assigned to each node in the XML graph in a highly compact way, such that the assigned labels of two nodes are given, one can be determined by simply looking at the labels if there is a reachability between these two given nodes. Based on this technique, the HID index can avoid time-/space consuming problems as previously discussed. The HID index is also optimized for evaluating descendants-or-self queries and ancestors-or-self queries on arbitrary graphs for a given node based on its labels. The index structure is stored in a relational database that makes it possible to use SQL statements to evaluate XPath queries.

1.4 Outline of this Dissertation

The remainder of the thesis is divided into 7 chapters:

Chapter 2 introduces an overview of XML and related technologies. It presents a short introduction to XML. It describes two ways to define the structure of the XML documents i.e. DTD and XML schema. It explains XPath and its different navigation axes. In addition to the above, it briefly discusses the different kinds of links (e.g., XLink, XPointer, IDs, and IDREFS) with their semantics as proposed by W3C.

Chapter 3 focuses on the different kinds of index structures that are proposed for indexing structured and semistructured documents. It describes the general architecture of a retrieval system. It divides indexes into two categories, the first category dealing with structured documents, and the second category considering semistructured documents. For each category, several indexing strategies are discussed. Since an index plays such an important role in efficient query processing, this chapter further describes several strategies proposed for XML query processing.

Chapter 4 presents related work and the contribution of our work compared to it. It classifies the related work into three approaches: structure indexes, path indexes, and connection indexes. It describes all the proposed indexes with their advantages and disadvantages for each approach. At the end, it discusses the problem of these indexes and explains how our HID index can overcome these problems.

Chapter 5 focuses on the HID index. At first, it gives an overview about all the basic terms and definitions, especially the 2-hop cover algorithm [CHKZ02] as a basis of the HID index. Then it explains in detail and with help of examples all the algorithms that are used to build the HID index. At the end, it provides a detailed discussion on how to efficiently evaluate different types of path expression queries with the aid of the HID index.

Chapter 6 describes how to implement algorithms that are used to build the HID index. It investigates the database schema that is used to store information about the index.

Chapter 7 covers an experimental evaluation of the HID index. It presents two experiments; the first experiment uses a small fragment of data to proof our concepts. The second experiment uses a large subset of data to study the efficiency of the HID index against other indexes. For each experiment, it compares the creation time and the space requirements for HID index against other indexes.

Chapter 8 summaries the contribution of the work and outlines the future areas of research. Preliminary results of our work have been published in [SU03, SU04, SU05].

1.5 Limitation of This Work

The HID index described in this thesis, focuses on XML data graphs that contain cycles (e.g., Internet Movies Database (IMDB) as a real life XML data [IMDB]). If the XML data is represented as a very large tree or as a large graph without cycles, other approaches may indeed prove to be more efficient that the HID index.

2

XML and Related Technologies

In this chapter, we give a brief introduction to XML (eXtensible Markup Language), its context and the role it plays in a Web environment. We briefly discuss the syntax of XML documents and the semantics it can represent. Since we do not have much more space to exhaust all details of technical definition of XML, we restrict ourselves to the presentation of the most prominent features that form the backbone of XML and make a challenge for database management systems, especially in case of querying and indexing data.

The remainder of this chapter is organized as follows. In Section 2.1, we give a general overview of XML. In Section 2.2, we introduce the description of document type definitions (DTD) and XML schema. In Section 2.3, the necessary concepts for processing XML data are briefly reviewed. In Section 2.4, we discuss advanced technologies related to XML.

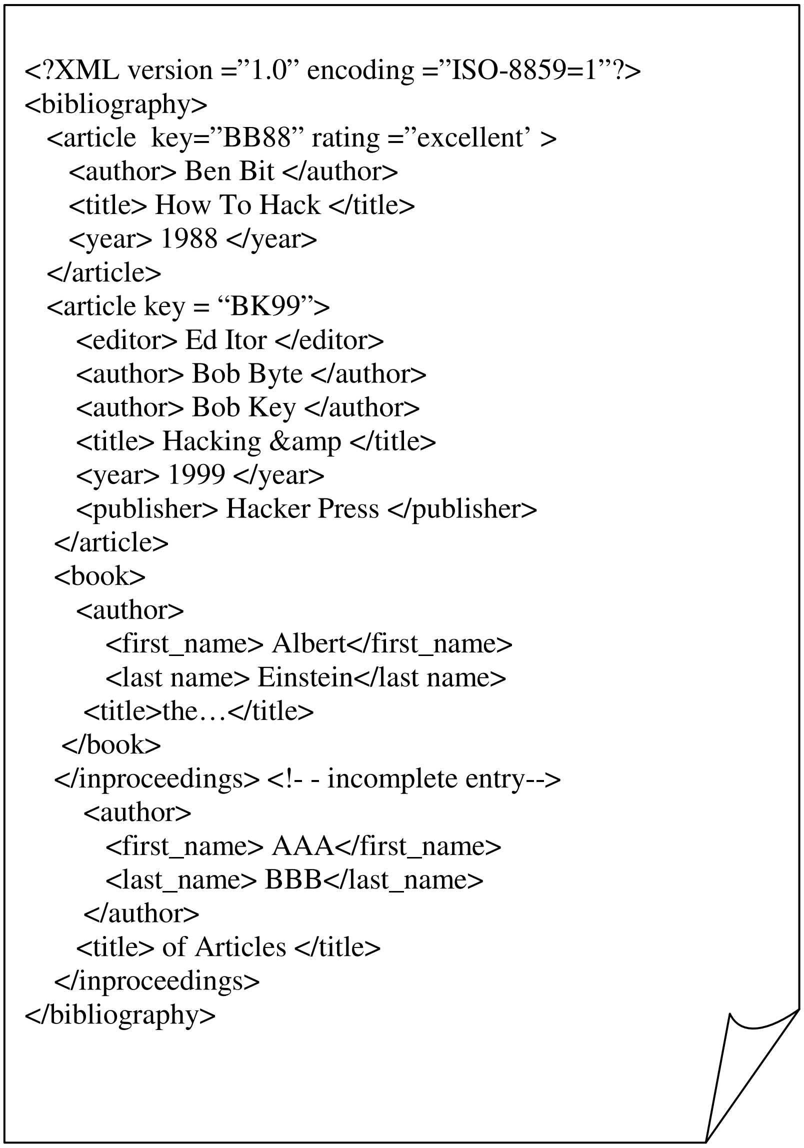
2.1 The eXtensible Markup Language (XML)

When asked for a concise definition of XML, one could say that XML is a markup language for structured documents. It appeared 10 years ago by the XML Working Group which has been working in cooperation with the W3C (World Wide Web Consortium) [W3C98X] and serves as a syntactical framework for data interchange on the Internet. Because it was backed by the leading players in the software and communication industry, it quickly become of the ubiquitous and universal data interchange format.

Structured documents are documents, which not only contain *particles elements*, e.g. plain text, but also feature syntactic annotation of what the semantic relationships between these particles are. XML gives the user two basic mechanisms to structure documents: elements and attributes. Elements are named lists, which again contain elements or text; each element having a name, called tag, and an associated list of attributes; i.e., a list that consists of name-value (attribute name-attribute value) pairs. The XML standard defines two special kinds of attributes: identifiers (IDs) and identifier references (IDREFs) (more details in Section 2.4); XML parsers also have to check certain semantic constraints that the presence of these attributes may require. Identifiers have to be unique throughout the documents, i.e., no two elements may carry the same ID, and identifier references must point to identifiers which exist in the documents; otherwise, a parser is supposed to issue an error massage and abort. It should be noticed that for most documents there is more than nature way to group content particles, an issue that will be discussed later in more detail.

A markup language is a mechanism to single out the document structure from the content particles usually by employing two different but not necessarily disjoint alphabets. One of the main contributions of XML is to provide a standard way of doing this, thus enabling the development of one parser for many different applications. Historically, XML can be seen as a mixture of regular right-part grammars and parenthesis grammars [GH76] [LaL77].

Definition 2-1 (Semistructured Documents): Semistructured document is defined as a data that has no fixed schema known in advance (e.g. XML). Recently, semistructured data arises from a wide range of applications such as integration of heterogeneous sources, digital libraries, and the World Wide Web. In general, semistructured data can be neither stored nor queried in relational or object-oriented database management systems easily and efficiently. Most semistructured data models organize data into directed graphs (or tree-like graphs). The data in the graph is self-describing, where each node represents an object and each edge represents the relationship between objects. We used through the thesis the Object Identifiers (OID) to reference to nodes in the XML graph. They are usually pointers to either a memory or a disk location where nodes are stored in the database.



```
<?XML version =”1.0” encoding =”ISO-8859=1”?>
<bibliography>
  <article  key=”BB88” rating =”excellent’ >
    <author> Ben Bit </author>
    <title> How To Hack </title>
    <year> 1988 </year>
  </article>
  <article key = “BK99”>
     <editor> Ed Itor </editor>
     <author> Bob Byte </author>
     <author> Bob Key </author>
     <title> Hacking &amp </title>
     <year> 1999 </year>
     <publisher> Hacker Press </publisher>
   </article>
   <book>
     <author>
        <first_name> Albert</first_name>
        <last name> Einstein</last name>
      <title>the…</title>
    </book>
   </inproceedings> <!- - incomplete entry-->
      <author>
        <first_name> AAA</first_name>
        <last_name> BBB</last_name>
      </author>
     <title> of Articles </title>
   </inproceedings>
</bibliography>
```

Figure 2.1: A bibliography as an instance for semistructured document

It is linguistically unfortunate that mark-up documents are called structured documents whereas the graph representation of those documents is called semi-structured data (see Definition 2-1). In this sense, we can say that, for our purposes, semi-structured data are abstract representations of structured documents. Some

database researches also call structured documents *semistructured* documents to highlight the difference between the flexible structure of documents and the very regular nature of (relational) databases.
To illustrate the previous concepts, we now present an XML example (see Figure 2.1). Historically, bibliographies are one of the favorite instances of semistructured documents since they come from a domain, namely digital libraries.

We call an XML documents well formed if it adheres to a tree structure and obeys other syntactic and semantic restriction, which we will mention when, appropriate. Therefore, XML document comprise text and tags. As shown in the example (see Figure 2.1), *Bin Bit, Bit, How To Hack, or AAA* constitutes the text part or the element contents, whereas <bibliography>, <author>, <article> are tags: tags are enclosed in an angle brackets (< >), which must not appear as such in text, and may carry an association list. The first element of tags (article, author,…) is also called element or element name. Each element may carry an association list of name-value or symbol-value pairs, called attributes; for example, the association list for the fist article element assigns the value BB88 to the symbol key and the value excellent to the symbol rating. XML entities, which are either a privation (Section 2.2.1) or representations for characters, which are not legal in XML documents. Entities do not extend the expressiveness of the language. The title in the second article record in the example document in Figure 2.1 contains the entity &amp; to represent the character &, which is not legal in XML documents.

There are additional features of XML, which were not included into the abstract model because they may be expressed by elements and attributes in combination with application specific knowledge, which they would require anyway when they are evaluated. Notations are a way to link entities in the document-to-document-external entitles by means of Uniform Resource Identifiers (URI) [BLFIM98]. Processing Instruction is used to pass information to external application, style-sheets are a popular example, and determine their behavior. Comments finally are used to annotate documents for human readers; they should not influence the behavior of (automated) applications.

2.2 Structure of XML Documents

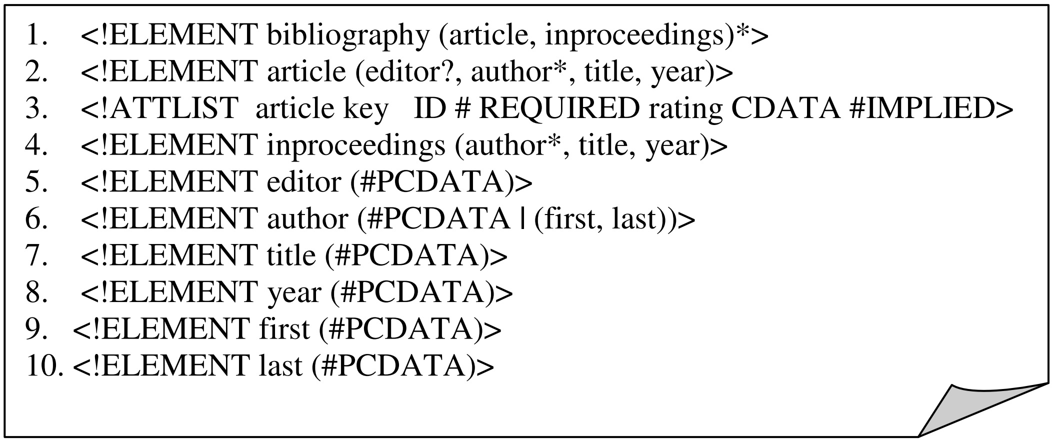
XML is an attractive data representation standard because it offers a simple, intuitive, and uniform text based syntax. In addition, it is extensible, which means that new structure can be added by creating and nesting new tags. Because of these characteristics, the XML representation provides unlimited potential in representing any kind of data. This unlimited potential is one of XML's most important strengths, but it turns out that it is too difficult for application to mange it. Therefore, some structural constraints are needed to bound the unlimited XML data representation. In order to specify and enforce XML structure, Document Type Definitions (DTDs) and XML Schema have been used. They describes in more details as follows:

2.2.1 Document Type Definitions (DTD)

The Document Type Definition (DTD) [Har99] describes the structure of XML documents and acts as a schema for the XML documents. A DTD specifies the structure of the XML element by specifying the names of its sub-elements and attributes. Sub-element structure is specified using the operators * (set with zero or more elements), **+** (set with one or more elements)? (Optional), and | (or). All values are assumed to be string values unless the type is ANY, in which case the value can be an arbitrary one XML. Each element can also have an arbitrary number of attributes. There is a special attribute of type ID, which can occur at most once for each element. The ID attribute uniquely identifies an element within a document and can be references through an IDREF attribute from another element. IDREFs are untyped in the sense that they can point to the ID field of any element. There is no concept of a root of a DTD. Figure 2.2 shows an example DTD specification, which specifies the schema for the XML document example in Figure 2.1.

As shown in the DTD specification, the bibliography element has articled and in proceedings as sub-elements (line 1). The article sub-element has editor, author, title, and year sub-elements (line 2). The editor sub-element is optional, as specified by the

"?" in the DTD. In addition, the article sub-element contains the attribute key of type ID (line 3).



```
1.   <!ELEMENT bibliography (article, inproceedings)*>
2.   <!ELEMENT article (editor?, author*, title, year)>
3.   <!ATTLIST  article key   ID # REQUIRED rating CDATA #IMPLIED>
4.   <!ELEMENT inproceedings (author*, title, year)>
5.   <!ELEMENT editor (#PCDATA)>
6.   <!ELEMENT author (#PCDATA | (first, last))>
7.   <!ELEMENT title (#PCDATA)>
8.   <!ELEMENT year (#PCDATA)>
9.  <!ELEMENT first (#PCDATA)>
10. <!ELEMENT last (#PCDATA)>
```

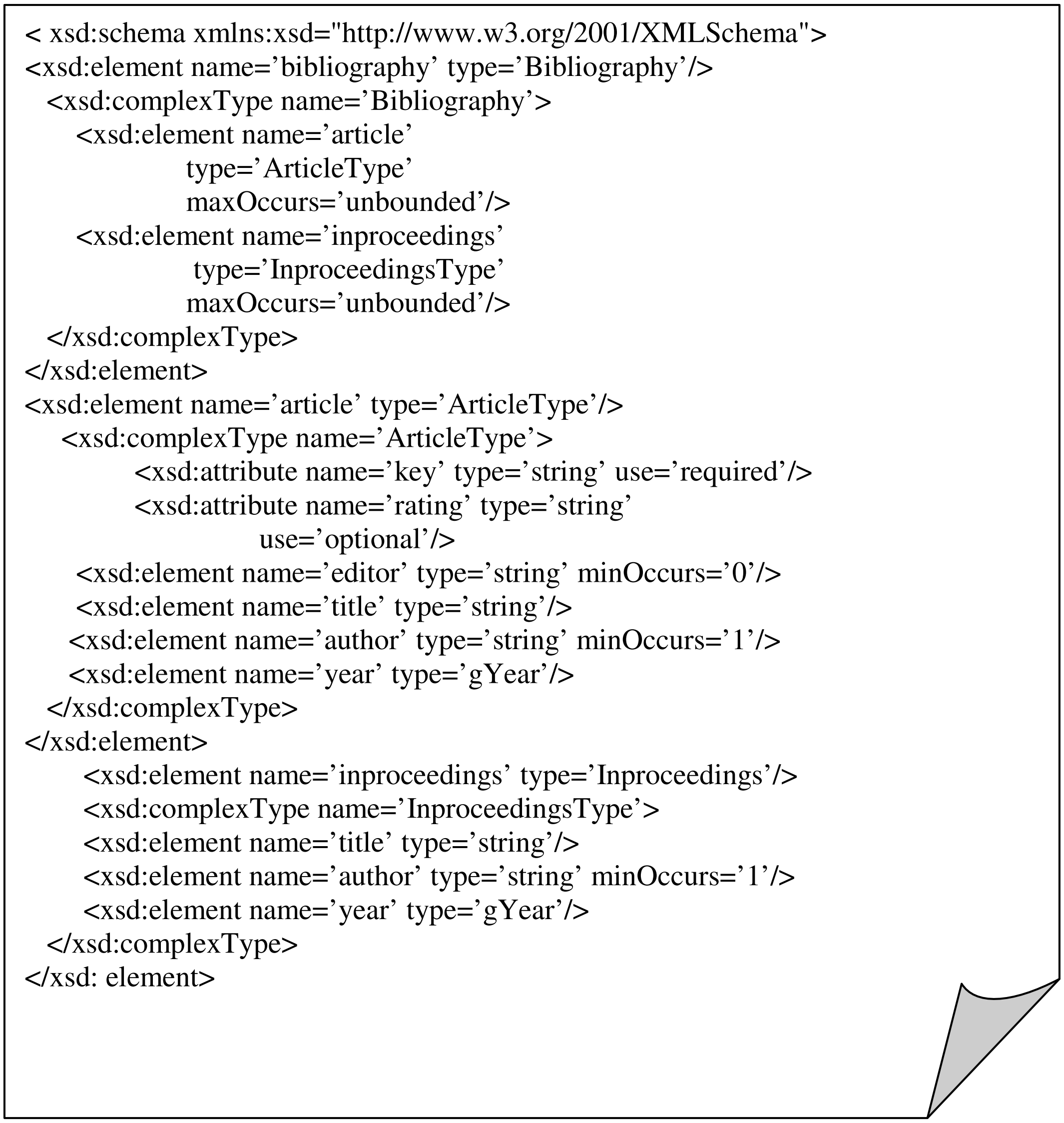
Figure 2.2: DTD for the example document of Figure 2.1

A DTD is a context-free grammar that specifies the optional structure of an XML document. It can be used to a certain extend, as a schema; however, DTD has some limitations in managing type information (e.g., there does not exist the notion of atomic types, unique type per element name, etc.). Due to these limitations, DTDs were not able to cope with the excitations for a proper schema language. Therefore, new schema languages have been proposed. Among them, XML schema has become the recommendation of the World Wide Web Consortium.

2.2.2 XML Schema

Although DTDs have served well for several years as the primary mechanism for describing structural information in the SGML and HTML communities, they are too limited for many data-interchange applications. For example, DTDs can only specify that elements are text strings, text strings mixed with other child elements or child elements without text.

Furthermore, they are not formulated in XML syntax and provide only very limited support of types or namespaces. XML Schema tries to overcome some of the deficiencies, and, like DTDs, is being developed and standardized by the World Wide Web Consortium.



```
< xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name=’bibliography’ type=’Bibliography’/>
  <xsd:complexType name=’Bibliography’>
     <xsd:element name=’article’
                type=’ArticleType’
                maxOccurs=’unbounded’/>
     <xsd:element name=’inproceedings’
                 type=’InproceedingsType’
                maxOccurs=’unbounded’/>
  </xsd:complexType>
</xsd:element>
<xsd:element name=’article’ type=’ArticleType’/>
    <xsd:complexType name=’ArticleType’>
           <xsd:attribute name=’key’ type=’string’ use=’required’/>
           <xsd:attribute name=’rating’ type=’string’
                      use=’optional’/>
     <xsd:element name=’editor’ type=’string’ minOccurs=’0’/>
     <xsd:element name=’title’ type=’string’/>
    <xsd:element name=’author’ type=’string’ minOccurs=’1’/>
    <xsd:element name=’year’ type=’gYear’/>
  </xsd:complexType>
</xsd:element>
      <xsd:element name=’inproceedings’ type=’Inproceedings’/>
      <xsd:complexType name=’InproceedingsType’>
      <xsd:element name=’title’ type=’string’/>
      <xsd:element name=’author’ type=’string’ minOccurs=’1’/>
      <xsd:element name=’year’ type=’gYear’/>
  </xsd:complexType>
</xsd: element>
```

Figure 2.3: An XML schema for the example XML document in Figure 2.1

Figure 2.3 shows an XML Schema for the example document in Figure 2.1. Clearly, XML Schema is a more comprehensive and complex description language than DTD. It offers more control over the document structure and text data by introducing types such as generic string but also more specific ones like *gYear* for Gregorian year. Since XML Schema is too complex to even give an overview of all its features here, we just mention some general principles of the language. More information about XML schema can be found at [W3CS01].

Predefined Types. The standard provides a set of commonly used so-called *simple types* and allows the definition of complex types, which are composed of simple types. Regular expressions, list and set constructs even permit sophisticated structures in text types.

Type Inheritance. XML Schema encourages the reuse of previously defined structures, no matter whether they are user-defined or pre-defined in the standard. Subtypes can add more elements to a supertype but may also only represent a subset or values. Groups of attributes and elements can be named for later re-use.

Documentation. To allow for specification of both domain specific and application-Specific knowledge, XML Schema provides dedicated elements like *appinfo*, documentation and notations elements for annotating schemes for both human readers and machines.

Uniqueness Constraints, Key, and References. It is possible to declare uniqueness constraints on certain attributes of child elements. This mechanism enables keys and references, too.

Namespaces. Sometimes it is desirable that documents conform to more than one schema. To achieve this, XML Schema contains the necessary tools to enable fine-grained control over namespaces. The process of establishing whether a document conforms to a schema is called *schema validation*. Note that even despite the terminology used, XML Schema is not so much about defining data types like integers or zip codes and hence defining semantics but more about restricting documents. Therefore, the set of permissible (parsed) character data in a content particle looks exactly like an integer or zip code. It does not assign semantics to documents. Data types in the XML world are only introduced by query languages or when documents are to be processed by an application.

2.3 Processing XML Documents

There are two major standardized ways for users to get access to the content of XML documents: event-based parsing APIs (SAX) and tree-based parsing APIs (DOM). While the former enable extremely resource-efficient processing of documents, the later provide a natural view of documents which is convenient when higher-level applications like query languages are implemented.

2.3.1 Event-based Parsing (SAX)

An event-based API reports parsing events, such as the start and end of elements, directly to a host application through callbacks of user-registered functions for the different events, and usually does not build an internal parse tree. The simple API for XML events and (SAX) [Meg02] is the best-known example of event-based parsing is a *de facto* standard.

The first step of a SAX parser usually consists of splitting up the source document into tokens. The simple syntax of XML does not require a sophisticated laxer to do this. The most basic way to tokenize a document is to use the occurrences of the brackets <and> as an orientation. For our example document in Figure 2.1, a tokenization could produce the following stream:

1. ‘‘<bibliography>’’
2. ‘‘\n ’’
3. ‘‘<article key="BB88" rating="excellent">’’
4. ‘‘\n ’’
5. ‘‘<author>’’
6. ‘‘Ben Bit’’
7. ‘‘</author>’’
8. ‘‘\n ’’
9. ‘‘<title>’’
10. ‘‘How To Hack’’
11. ‘‘</title>’’

12....

In general, these token stream events are not immediately suited for storage in databases. For example, a SAX parser might choose to return ‘‘Ben Bit’’ as two tokens ‘‘Ben B’’ and ‘‘it’’. The author conjectures that this is to make parsing faster and easier.

Additionally, SAX parsers de-entities documents and tokenizes processing instructions *etc*. Furthermore, the programmer has some control over low-level features like character sets used in the document. Before parsing can begin, users have to register callback functions with the parser. On encountering a token of a specific type, say a start token, the parser calls the function the user previously registered for, in this case, processing start tokens.

2.3.2 Document Object Model (DOM)

Tree-based APIs (or DOM) convert the document into an internal tree structure. Applications can then navigate through this tree via a standard interface. For most navigating applications, the Document Object Model (DOM) [W3CD98] is the API of choice. Often, tree-based APIs use schema information to spare the user from having to write code that dispatches the problem control flow according to the element type of the current node, therefore, allow more declarative programming styles.

While the event-based and stream-oriented perspective of XML documents represents the lowest logical layer in XML processing, tree representations provide more views that are intuitive. Tree-based XML processors are usually build on top of stream-based processors. The final goal of the DOM standard is to provide a programmatic and language-nature interface for XML and HTML, which means that the same interface can be used to access documents from different programming languages. The DOM specifications come in three parts: Core, HTML, and XML. The first layer, the Core DOM, provides base classes that can be used to represent parse trees of any structured document written in any markup language. Like stream-based representations, this layer can represent any document in a generic way; the API it defines is minimal and compact and used to navigate through the document content generically. However, to support more specific and adaptive programming, the DOM comes in more application-oriented interfaces: the HTML interface is oriented towards visual representation and, for example, includes access to style sheets and events. The XML interface focuses on higher-level data-centric processing, which means that documents are primarily considered data structured, and enables traversals on the parse tree; furthermore, it defines an event model and support for namespaces along with other useful extensions.

To support the evaluation of XML processing. The W3C wants the DOM to evolve on several levels. The first level is the core as described in the last paragraph. Its main objective is to provide support for document navigation and manipulation. The second level additionally features a style sheet object model, and defines functionality for manipulating the style information which a user may provide to annotate a document or, for example, for the visual representation of the document. It also enables traversals of the parse tree, defines an event model, and provides access to namespace information. The third level will address the persistence issues like document loading and saving. As well, it aims at validation of content models as defined in DTDs and XML schemas. In addition, it will also address document views and formatting, key events and event group

```
Package org.w3c.dom;
public interface Document extends Node
{
   public DocumentType getDoctype();
   public DOMImplementation getImplementation();
   public Element getDocumentElement();
   public Element createElement(String tagName)
            throws DOMException;
   public DocumentFragment createDocumentFragment();
   public Text createTextNode(String data);
   public Comment createComment(String data);
   public CDATASection createCDATASection(String data)
           throws DOMException;
   public ProcessingInstruction
            createProcessingInstruction(String target, String data)
            throws DOMException;
  public Attr createAttribute(String name)
           throws DOMException;
  public EntityReference
           createEntityReference(String name)
           throws DOMException;
  public NodeList getElementsByTagName(String tagname);
}
```

Figure 2.4: Snippet from the DOM specification

Future levels may concentrate on interactive features like interaction with a windows system, such as X-Windows and user input. There are also plans to include the query language XPath, and to address multi-threading, synchronization, security, and repository storage. Note that the DOM does not specify physical data structure; instead, an implementer is expected to use it as an interface to proprietary data structures.

To give an impression of the flavor of the DOM specification, Figure 2.4 shows a small fraction of the java version of the XML part of the DOM interface as is found in [W3CD98]. Note that the interface documents contain XML-specific functions, e.g., one for the creation of attributes of syntax nodes, but that the arguments of the function are generic, i.e., strings. It is possible to create a generic attribute but not to create a specialized key attribute as in the bibliography example in Figure 2.1. Schema languages can be used to create more application-specific interface.

2.3.3 XML Query Languages

Compared to the lower-level APIs of the previous section, XML query languages are another way to navigate through documents. They are usually declarative, i.e., users specify what information they want to extract from documents rather than how it should be extracted algorithmically. For example, if a user wants to have a list of all people who is listed as authors of articles in our example bibliography (Figure 2.1), he could write the query language XPath (section 2.4.1):

document (“bibliography.xml”) /bibliography/article/author/text ()

This query reads as follows, in the document “bibliography.xml” finds and returns all the text strings at the end of the tag bibliography, article, and author. Tag sequences are also called path expressions. The result is the list “Ben Bit”, “Bon Byte”, “Ken Key”. If the user is interested in all authors, no matter in what kind of publications they are listed, he can write.

document (“bibliography.xml”) //author/text ( )

The two simple queries show the basic way of how to access information in XML documents: the element hierarchies play a role similar to relations in query languages like SQL [Ame86]. Path expressions (Section 2.4.1) are used to specify which parts of a document the query is supported to return and store in the result variable. The second example demonstrates how wildcard ("//") can be used to denote any sequence of tags rather than a specific element.

Several query languages for extracting and restructuring the XML content have been proposed. Some are in the tradition of SQL and OQL [Cat94]; others are more closely, inspired by XML. No standard query language for XML has yet been decided; even XQuery [XQuery] the discussion is ongoing within the World Wide Web up till now. A comparison between several XML query languages is proposed in [BC00]. For example, XML-QL [DFFL+99] query language which was designed at AT&T Labs and it extends the SQL with an explicit CONSTRCUT clause for building the document resulting from the query and used the element pattern to match data in the XML document. XML-GL [CCDF+98] graphical query language, it was designed at Polytechnic di Milano, relying on graphical representation of XML documents and DTDs by means of labeled XML graphs. All the elements of XML-GL are displayed visually; XML-GL is suitable for supporting a user-friendly interface. XSL query language [Ber03], it was designed by the W3C XSL working group, its program consists of a collection of template rules; each template rule has two parts: pattern which is matched against nodes in the source tree and a template which form part of the result tree. XQL [RLS98] query language it was designed by Microsoft, it is a notation for selecting and filtering the elements, and text of the XML documents, it can be considered a natural extension to the XSL pattern syntax. Quilt query language [CRF00], it was designed by Software AG, borrows from many languages like, XML-QL, XQL and XPath, it is able to express queries based on document structure and to produce query results that either preserve the original document structure or generate a new structure. XQuery [XQuery] language, it was designed by W3C query language working group, it designed specifically for XML, where documents and their portions are ordered hierarchies of typed objects, each with properties and based on the type system of the XML schema and it also designed to be compatible with other XML-related standards. ApproXQL [Sch01] approximate pattern matching language for XML, it supports hierarchical, Boolean-connected patterns. Queries are tree-shaped search patterns with existence semantic: a document part matches a query if the names and keywords of the query exist and fulfill the desired relationships. XXL [TW02] search engine, present the concepts, which combine the capabilities of XML query languages with ranked retrieval, discuss how regular path expression can be recognized, it returns results ranked by textual similarity, show how ontology and suitable index structures used to improve the efficiency.

2.4 XML Technologies

The World Wide Web Consortium (W3C) represents the leading body for channeling and standardizing the development efforts in all areas concerning the internet. To find all relevant XML technologies at one place W3C Website [W3C] is the place to look at. This section describes three XML specifications, which improve the capabilities of defining links in XML documents.

2.4.1 XML Path Language (XPath)

XPath [W3C99P] [WL02] defines the basic addressing mechanism in XML documents, which is employed by most XML query languages. The expressions, which are defined by XPath, are called *location paths*. Every location path is composed of one or more location steps and declaratively selects a set of nodes from a given XML document. One important expression in XPath is the location expression, which consists of the following three parts:-

- An axis, which specifies the tree relationship between the nodes selected by the location step and the context node.
- A node test**,** which specifies the node type and expanded-name of the nodes selected by the location step,
- Zero or more predicates, which further refine the set of nodes selected by the location step.

Definition 2-2 (XPath Axes): The axes that can be expressed with XPath are:

- *Navigating along child axis*

The child:: axis selects nodes that are immediate children of the context node. For example, child::author means return all the direct children for the context node author.

- *Navigating along parent axis*

The parent:: axis is the inverse of the child:: axis. It selects the node in the document tree that is immediately above the context node in the hierarchical order.

- *Navigating along descendant axis*

The descendant:: axis of a given context node locates all element nodes which the context node contains, that is, a child or a child of a child, and so on. For example /descendant::* locates all elements descendant form the root node, and therefore locates all elements from the given XML document.

- *Navigating along descendant-or-self axis*

The descendant-or-self::axis, like the descendant:: axis, locates all nodes that are descended from the context node, its children, its children's children, and so on. Unlike descendant::axis, however, descendant-or-self:: also selects the context node itself.

- *Navigating along ancestor axis*

The ancestor:: axis locates all nodes in the document hierarchy above the context node. It locates the parent of the context node, the parent's parent, and so on up to the root node. The root node is an ancestor of all other nodes in the document. It has no ancestors.

- *Navigating along ancestor-or-self axis*

The ancestor-or-self:: axis, like the ancestor:: axis, locates all nodes in the document hierarchy above the context node. It locates the parent of the context node, the parent's parent, and so on up to the root node. Unlike ancestor::, however, ancestor-or-self:: also selects the context node itself.

- *Navigating along following axis*

All nodes that follow the context node, in document order, lie along the following:: axis. This specifically excludes attribute and namespace nodes, as well as nodes that are descendants of the context node.

- *Navigating along preceding axis*

The preceding:: axis is the inverse of the following:: axis. It locates all nodes that come before the context node in document order. Ancestors, attribute nodes, and namespace nodes cannot be located along the context node's preceding:: axis.

- *Navigating along following-sibling axis*

The following-sibling:: axis locates all nodes, elements only by default, which share the same parent as the context node itself, and which appear in their entirety after the context node, in document order. Since the root node has no parent, it also has no following-siblings. If the context node is an attribute or namespace node, the following-sibling:: node locates an empty node-set.

- *Navigating along preceding-sibling axis*

Similar to the following-sibling:: axis, the preceding-sibling:: axis locates all nodes, elements, by default, which share the same parent node and which appear before the context node, in document order. Since the root node has no parent, it also has no preceding-siblings. If the context node is an attribute or namespace node, the preceding-sibling:: node locates an empty node-set.

- *Self axis*

The self:: axis always locates the context node. It is often used to retrieve the value of the current node, particularly as the select attribute of the <xsl: value-of> element.

2.4.2 XML-Links

The fundamental nature of the Web is that it consists of documents linked together, generally with the ubiquitous <A href= "…" > element (in case of HTML [LJ99]). This type of link is one pointer direction. Recently W3C propose more powerful links that deal with XML documents. We divide XML links into two parts as follows:

- *Interior-document link*: Cross references between elements within a document by means of ID/IDREF(S) references.
- *Global-documents link*: Cross references from an element in an XML document to the root element of another XML document by means of XLink [W3C01XL] [Meg98] and cross references from elements inside one XML document to another element inside another XML document by means of XPointer

[W3C02XP] [Meg98]. In the next two sections, we discuss briefly the syntax and semantics of these XML linkages. In the following, these types of Links shall be discussed in details.

2.4.2.1 ID and IDREF(S) References

As explained previously in Section 2.2, XML documents may be accompanied by schemes that specify their structure, and impose restrictions on the values of some elements or attributes. The two languages (DTD or XML scheme) that were proposed to describe the structure of the XML documents, both allow the specification of the ID and IDREF(S) attributes. ID attributes are unique identifiers for the elements that bear them; IDREF attributes are logical pointers to ID attributes; IDREF(S) attributes are pointers to sets of ID attributes. IDREF(S) attributes establish the reference among the elements in the XML document, turning an XML tree into a graph. ID attributes serve as primary keys for the elements, while IDREF(S) attribute serve as foreign keys.

ID and IDREF(S) Semantics: Values of type ID must be single-valued. A name must not appear more than once in an XML document as a value of this type; i.e., ID values must unique by identifying the elements which bear them. Moreover, no element type may have more than one ID attribute specified. This element name serves as a source element of the link. For IDREF attributes, values of type IDREF(S) may be multi-values. Each value must match the ID attribute of some element in the XML document. This element name of these types serves as a target element of the link.

2.4.2.2 XLink and XPointer

A link represents a connection or relationship between two or more entities. Links may be explicit (direct reference) or implicit, that is, the entities are linked by applying a set of rules. The entities linked together are referred to as anchors. Thus, the simplest link consists of two anchors connected by a unidirectional reference. The entity from which the link starts is the source anchor, where it ends is the target anchor. If traversal is possible in both directions the link is called “bi-directional” [Hom99].

The HTML <A href=”...”> construct represents a simple link. Nevertheless, it may address several target items.

XLink :

The XML Linking Language or XLink for short supports two types of links: “simple XLink” and “extended XLink”. Simple XLink complies with the HTML linking model, while extended XLink is more powerful. It is meant to be used by applications requiring a more elaborate linking technique than the one provided by HTML. XPointer can be used within XLink for expressing links between XML documents. Arbitrary elements can be declared to have link semantics by equipping them with an “XLink: type” attribute and suitable additional descriptive attributes from the “XLink: namespace” that permit to specify different kinds of properties for a link (see Table 2.1). All this information specifies the behavior or semantics of a link.

The type attribute may have one of the following values:

- Simple: a simple link, about the semantics known from < A href= “…”>.
- Extended: an extended link, possibly multi-resource, link that may reference to external resources.
- Locator: a pointer to an external resource
- Resource: an internal resource
- Arc: a traversal rule between resources; defines how one can get from one anchor to the other
- Title: a descriptive title for another linking element

Type definition attribute	type
Locator attribute	href
Semantic attributes	role, arcrole, title
Behavior attributes	show, actuate
Traversal attributes	label, from, to

Table 2.1: Properties of XLink

Simple XLink [Meg98] is comparable to HTML links. It realizes a unidirectional reference. However, the target may be one item as well as a set of items. Example 2.1 explains a simple link.

Example 2.1: Simple XLink

```
<mylink XLink: type="simple" title="Computer Science" href="http://www.cs.uni-duisburg-essen.de"
    show="new" Content-role="information about the database group and their research projects">
    Data management system and knowledge representation group
  </mylink>
```

The above link is of type "simple" (unidirectional link, as normal HTML link). The local resource is the page of "the data management system and knowledge representation group" which has the role ="information about the database group and their research projects". The source element name is "mylink".

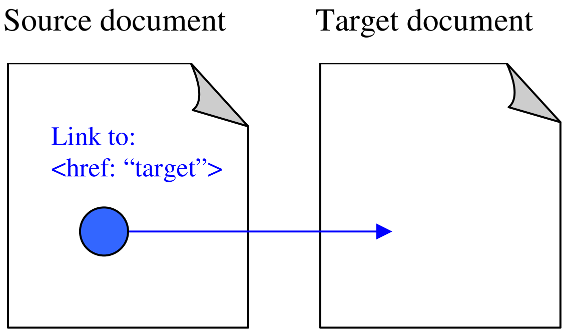


Figure 2.5: A simple XLink example

The target resource is the href = http://www.cs.uni-duisburg-essen.de. The title of this target resource is "Computer Science". This target resource will be displayed in a new window (show = "new"). Figure 2.5 describes a link from a source XML document to a target XML document as a simple XLink.

Extended XLink [Meg98] covers everything what simple XLink covers. Additionally, it can be used to create relationships between multiple resources (e.g., documents, images…). Extended links contain one or more locator elements, each of which is a reference of its own. Example 2 gives an example of an extended XLink.

Example 2.2: Extended XLink

```
<mylink XLink: type ="extended "inline = "false">
```

```
    <mytarget XLink: type=“locator” inline = “false” role=“Essen” title =“Duisburg-
            Essen University” href = “http://www.uni-duisburg-essen.de”/>
    <mytarget XLink: type =“locator“ inline = “false” role=“professor in computer
            science“ title = “computer science” href =“http://www.cs.uni-essen.de”/>
    <mytarget XLink: type =“locator“ inline = “false” role=“research project” title
            =“XML and database” href = “http://www.cs.essen-sb.de/dawis”/>
</mylink>
```

The above link is of type “extended” and is a multidirectional link. Each of these links has some attributes that reveal more information about the remote resource. For example, “locator” means that the link points to exactly one document or one resource within the extended link. Inline is *false* means that this link is an *out of line link*. The two attributes “role” and “title” give additional information about the remote documents. Figure 2.6 shows an example of extended Xlink. It has one-source XML document points to four target documents.

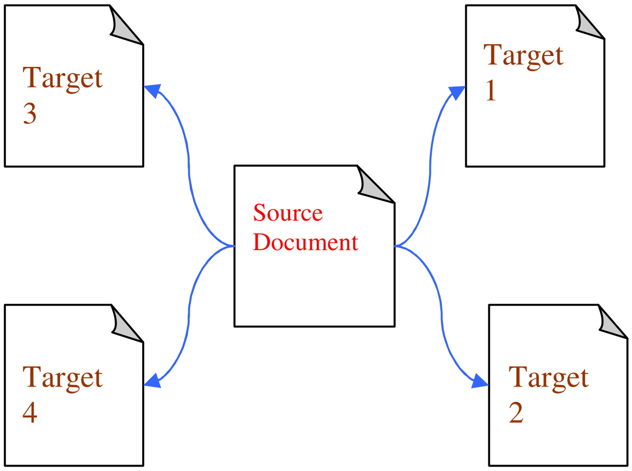


Figure 2.6: An Extend XLink example

XPointer:

Xpointer is an extension of the XML Path Language (XPath) [W3C99P]. It permits referencing into the internal structures of another XML document. The examination of internal parts may be based on various properties, such as element types, attribute values, character content, and relative position. The parts that are to be examined are addressed by a set of expressions. An expression is evaluated with respect to the current context (which includes aspects such as bindings between variables and values and the context element within the given XML document). An expression can be used to select children, siblings, and nodes with the given attributes. The general form of the XPointer expression is *URL # XPointer-expression.* Such expressions are evaluated as follows:

- The actual XPointer portion begins after the URL and starts with the wildcard "#"
- Start at root ( ) or id (element_id) or origin ( ) or html (anchor_name)
- Navigate with child, descendant, ancestor, preceding, Psibling and Fsibling
- Four arguments:
    - ➥ Number (i-th instance found).
    - ➥ Element name sought after.
    - ➥ Attribute name.
    - ➥ Value of the attribute.

In the following, several examples shall be presented to explain the characteristics of XPointer:

Example 2.3: XPointer

URL # Child (1, element)

Selects the first child from the source at the specified location (URL) independently of the existence of some attribute and its value.

Example 2.4: XPointer

URL # ID (DBS)

This example selects the element from the source at the specified location (URL) with an ID attribute whose value is "DBS".

Example 2.5: XPointer

URL # ID (DBS). Child (1, Unland, professor, "database").

This example selects the first element with name "Unland" that has an attribute "professor" whose value is "database". This attribute has to occur within an element with an ID attribute whose value is "DBS".

2.4.2.3 XML Links Semantics

XLink information is a special kind of metadata. One of the most remarkable features of the XLink is that it supports third-party links, which can be aggregated in so-called link base. XLink feature make it possible to realize the vision of the Web as an "open hypermedia system", commonly called the "open Web". Then, the question of the open Web is, how the browser communicates with the open link bases?

The semantics of attributes for XLink would describe as follows: Since there is no predefined set of "link meaning" in the XLink, there are some attributes to describe the link semantics. For example, a link for the book you are currently reading may associate resources such as the author's personal Web pages, the publishers Web site, a number of online stores selling the book, several web pages with the reviews of the book, and the web site of the book itself. While XLink makes it possible to create such a link, there is no standardized way to describe the actual semantics of the linked resources. XLink follows the path of many web technologies and defines a way in which semantic information may be specified.

There are two types of semantics attributes. Attributes ("*role*" and *"arcrole"*) are two machine-readable semantics attributes, which carry semantic information that can be interpreted by applications. In addition to the two machine-readable attributes, there is one attribute (title), which presents semantics in a human-readable way. The role and arc-role attributes always contain an URI, which references some meta-data which describe the meaning of the link. These attributes can be described as follows:-

"Role" attributes:

This attribute describes the role that an item plays. It describes the role of the link (appearing in simple and extended element). Describe the role of the resources (appearing in locator or resource element). It also describes the role of the particular resources within the link (e.g., the book link may associate resources describing, people, publishers,).

"Arcrole" attributes:

Arcrole attribute describes the role of an arc. In XLink, arcs are represented by *arc* elements or by simple elements. There is only one special case of the *arcrole attribute* defined in XLink specifications. This is a special case of a link-base “third-party link”, which identifies a link-base for a particular resource.

Title attributes. The title attribute contains human-readable information about the element in which it appears. It is allowed for all elements that identify the links and/or resources (simple, extended, locator, arc…).

3

Indexing Structured and Semistructured Data

Index structures are transparent for the user, but play a key role in database performance. When searching for relevant data, the straight-forward approach is to scan through all data and test every data item. This can be inefficient and cumbersome, especially, if huge volumes of data are involved. An index is an auxiliary structure, which can be viewed as a collection of data entries designed to speed up access to data by content. The subject of indexing is a hot research topic, especially in case where the underlying data is XML data. In Section 3.1, the relation between data and queries, indexes and information retrieval, and the general structure of indexes are discussed. We divide indexes into two categories, one proposed for indexing structured documents (Section 3.2) and the other proposed for indexing semistructured documents (Section 3.3). In Section 3.4, several strategies proposed for XML query processing are discussed also. Since the way of storing XML in a database systems has a direct impact on indexing and querying, different storing techniques are shortly discussed in Section 3.5.

3.1 Motivation

XML intimidate to expand beyond its document markup origins to become the basis for data interchange on the internet. One of the most expected application of XML is the interchange of electronic data (EDI). Compared to existing Web documents, electronic data is primarily designed for computer, not human, computation. For example, businesses could publish data about their products and services, this data can be compared and processed automatically by potential customers; business partners have the ability to build the index and exchange operational data between their information systems on secure channels; search reboot could automatically integrate information from related sources, (like stock quotes from financial sites and sports scores from news sites). New opportunities will arise for third parties to add value by integrating, transforming, cleaning, and aggregating XML data. Once it becomes spreading, it's not hard to imagine that many information sources will structure their external views as a repository of XML data, no matter what their internal storage mechanisms are. Data exchange between applications will then be in XML format.

What is the role of an index and query language in this world? The main goal of indexes is to speed up query evaluation and query languages are a local adjust to browsing capabilities, providing a more expressive "find" command over one or more retrieved documents. Alternatively, it may serve as a soaped-up version of XPointer [Har99], allowing richer forms of logical reference to portions of documents. From the database viewpoint, the enticing role of a XML query language is as tool for structured and content-based queries that allows the application to extract of precisely, the information it needed from one or several XML data sources.

XML Data is fundamentally different from relational and object-oriented data, so the traditional query languages like, SQL [UW97] or OQL [Cat94] are not appropriate for querying XML data. The key distinction between data in XML and data in traditional models is that XML is not rigidly structured, but sensitively structured. In the relational and object-oriented models, every data instance has a schema that is separated from and independent from the data. In XML, the schema exists with the data (e.g., DTD) or maybe no schema at all. Thus, XML data is self-describing and can naturally model irregularities that cannot be modeled by relational or object-oriented schemes. For example, data item may have missing elements or multiple occurrences of the same element; elements may have atomic values in some data items and structured values in others; and collections of elements can have heterogeneous structure. Even if

XML data is associated with a DTD, it is self-describing (e.g., the data can still be parsed, even if the DTD is removed); and except for restrictive forms of DTDs, may have all the irregularities described above. This flexibility is crucial for many EDI applications.

3.1.1 Indexes and Information Retrieval (IR)

An index is a data structure used to locate specific elements of a collection of data entries. Again, the main objective of an index is to speeds up query evaluation, allowing the access of the relevant data directly without a sequential parsing of all databases. The benefit of accelerated query evaluation comes at a price, though indexes require additional storage space and depending on the data structures involved, this storage overhead may be significant. Hence, some indexing approaches focus on memory space reduction or come with elaborated paging strategies for efficient disk access.

Another problem of indexes is their capacity to reflect on updates. That is, an index may be more or less capable to adapt to change in the database, e.g., insertion or deletion of records. If the proposed index structure has the ability to reflect on such changes, and the modifications are restricted to only a small part of the index, this proposed index is said to realize "*incremental updates*". Other indexes need to rebuild from scratch after any update of the underlying database.

Finally, there are *exact* and *inexact* indexes. Intuitively, an index is *exact* if it retrieves all relevant data for a given query. *Exact* indexes are those with optimal *precision* and *recall*, (these two performance measures for retrieval systems are defined formally in the next section). However, indexes are considered *exact* if they have optimal *precision* and *recall*. Here, we would like to point out that the notation of exactness is more intricate than it appears at the first glance. An index may well perform with optimal *precision* and *recall* [BR99] yet return more items than expected. An index is *inexact* if it retrievals more data than actually relevant to the given query.

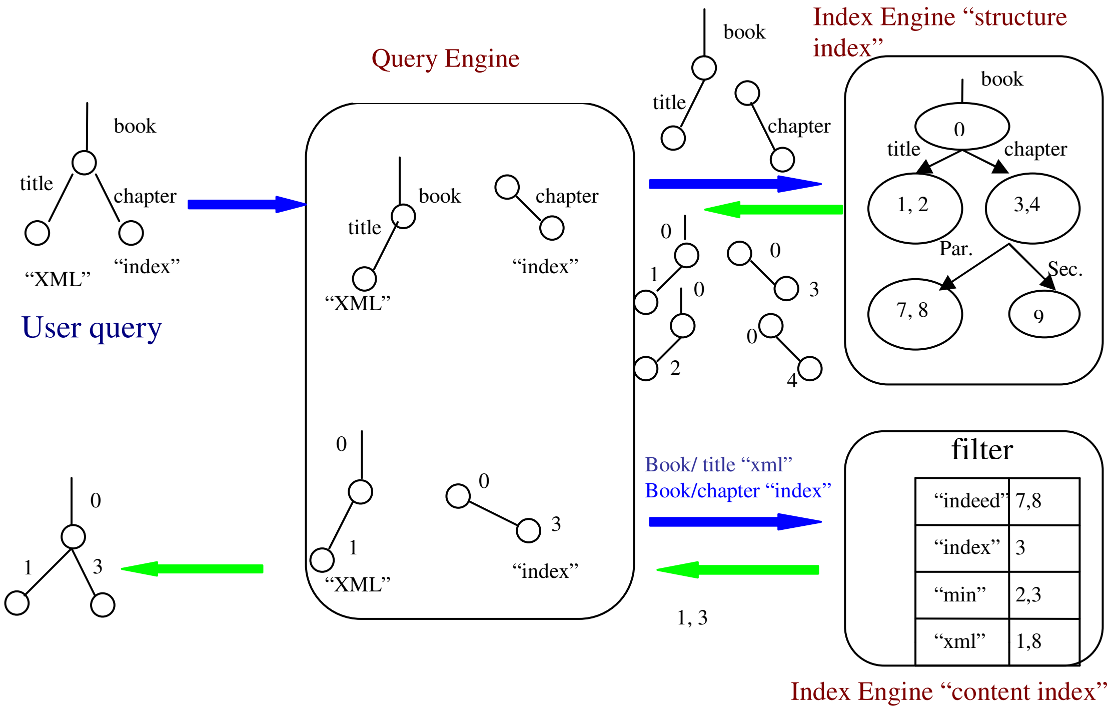


Figure 3.1: General architecture of a retrieval system

3.1.2 An Information Retrieval System

In this section, we briefly explain a generic architecture for a general retrieval system. As shown in Figure 3.1 that a generic retrieval system consists of two kinds of components: the *query engine* and a set of *index engines*. The *query engine* is responsible for accepting user queries and creating *query plans* to evaluate them. The *query plan* may split the user query into multiple look-up queries, each retrieving hints for only a part of the user query that finally need to be rejoined.

As illustrated in Figure 3.1, the *query engine* consults both a structure index and content index, to evaluate path and text queries. Here, we distinguish a "top-down" query plan, which looks up the structure part first and then matches the keywords, from a "bottom-up" approach, which inverts the look-up order.

A Query plan and look-up strategies must not be confused. While a query plan specifies which index looks up need to be carried out in which order in the behalf of the *query engine*, on the other hand, a look up strategy which describes how a single index look-up is organized by the index engine.

Retrieval Performance: As described previously, the main concepts for assessing the quality of search results for a given query are *precision* and *recall*. They can be measure to the performance of a retrieval system as a whole.

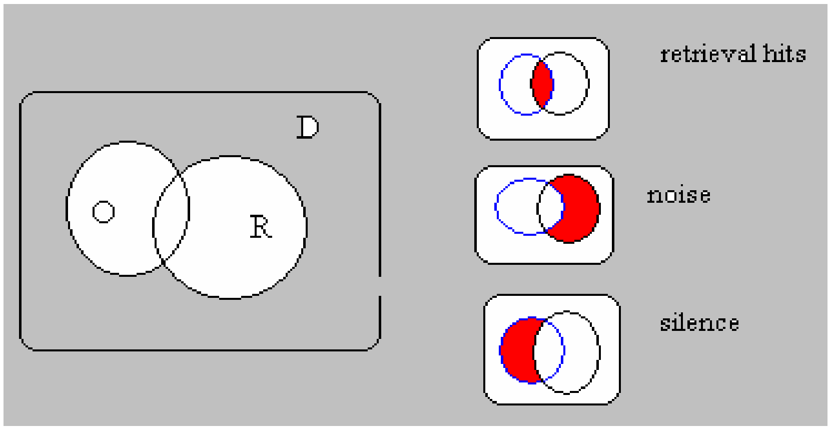


Figure 3.2: Retrieval precision and recall

Consider Figure 3.2 for example, where *D* refers to the set of documents, *O* be the subset of documents matching a given query, and *R* be the set of hits actually returned by the system. The *precision* of the retrieval process for a given query can be computed by using the following formula $\frac{|O \cap R|}{|R|}$. This formula defines the proportion of relevant search results $(O \cap R)$ computed to all search results *(R)*. The *recall* of the retrieval process for a given query can be computed as $\frac{|O \cap R|}{|O|}$: Defines the proportion of relevant search results $(O \cap R)$ computed to all documents actually matching the query *(O)* [BR99]. The *recall noise* (or simply, the noise) contained in the search results for a given query is the set (*R\O)*, i.e. the set of false hits. The amount of missed hits (e.g., the set *O\R)* is called the *silence*. To conclude, *recall* and *precision* are tools to measure the effectiveness. Effectiveness is purely a measure of the ability of the system to satisfy the user in terms of the relevance of documents retrieved.

3.2 Indexing Structured Data

In this section, we introduce indexes proposed for structured documents (e.g., Standard Generalized Markup Language (SGML)) without going into deeper detail (as our work is concentrated on semistructured documents (e.g., XML)). A structured document is a document that complies with a fixed schema known in advance. Indexing structured documents is generally referred to as a "domain of database systems" (DBs). Moreover, in this section we also survey several major classes of structured indexing techniques, focusing on different types of applications and systems.

Before exploring the different types of indexes, we shall first review the familiar DBs. For all major classes of DB structures: hierarchical DB, relational DB (RDB), and object-oriented DB (OODB), the basic mechanisms concerning indexing are similar. In all types of DBs, there is the notion of an entity (following the well-accepted entity-relationship design). In the older types of DBs, an entity is represented by a record. In an RDB, an entity is represented by a tuple in a relation. In addition, in an OODB, it is represented by an object. One of the most important indexing capabilities is to locate for an entity, given its characteristic description, using a "key" or an object ID. By means of the key or the object ID, the entity is uniquely identified. Such key values can be stored in different search structures that allow for their efficient retrieval. Likewise, other attributes can also be stored in such search structures. In the following, different strategies proposed for indexing structured documents are presented.

3.2.1 Tree-structured Indexing

A variety of index structures has been proposed for indexing database systems. Here, we describe a few of these indexes shortly, in order to determine the difference between indexes that have been proposed for traditional databases, and indexes proposed for XML documents. The oldest indexing structure with sufficient efficiency which is still in common use is the *Indexing sequential Acess Method (ISAM)* [RG00]. An ISAM tree is a static index structure that is effective when the frequently file is not updated, but it is not suitable when files that grow and shrink a lot. This kind of index is based on the "sequential file" which contains a collection of entities or records in DBs, and each of them is separable addressable. An index contains pointers to the physical location of entities. The index is sorted in order to facilitate binary searches and range searches. There may be multiple indexes, each maintained by a separate index. However, the main index can only be ordered according to at most one index, which is called a *primary index*. Other indexes are referred to as *secondary indexes* [RG00].

Queries operating on this primary index can enjoy the clustering effect of the entities at reduced retrieval cost, because consecutive entities are stored on the same physical page. Following an increase of information to be stored, the index file itself can become very large. Adjacent indices can then be grouped together to be indexed by yet another index. Thus, multiple levels of indexes are formed. However, insertion and deletion of entities with dense indexes would not be as efficient. Thus, *ISAM* has been generalized into *Virtual sequential file organization VSAM*, in which a *B+ tree* [CLRS01] is maintained for the records, indexed under the primary index or a key attribute. The use of a *B+ tree* is very useful and efficient in handling queries and in particular, in the existence of frequent insertion and deletion of records. The indexing structure is dynamic enough to adapt to the changes. *B+ tree* is a variant of *B-tree*, which is also commonly in use. Although *B+ tree* and multilevel indexing schemes are useful, the access efficiency is still of *O (log n)* [Com79] [RG00], where *n* is the total number of entities in the system.

3.2.2 Path-based Indexing

The evolution of OODB systems has led to the invention of more efficient ways of processing queries, e.g., by making use of the inherent navigational nature of the OODB. In particular, objects and their attributes are often accessed via a sequence of "paths". Such paths are defined by one-anchor points and following object identifiers (ID) of a sequence of intermediate objects. Querying, therefore, basically involves the traversal of the object hierarchy. Standard queries that can be answered readily are those that involve traversal from the top to the bottom of the hierarchy by following the ID chain forward. To process queries involving "traversal" in the reversed direction (implemented by a join in RDB), reverse links can be inserted in order to avoid the need of an actual join (or searching through hashed values). This approach, while easy to implement, involves the cost of having to access intermediate objects even if they are not involved in the queried attributes and predicates.

Moreover, this type of indexing can be extended to deal with XML documents. Such that, for path-based indexing, two indices are maintained separately: one for structure and one for content. In this case, each element in the XML document has an associated XPath. A unique identifier, say XID, is assigned to each unique XPath and the XID is regarded as a document identifier in conventional IR systems. Hence, the retrieval result for a token is a set of XIDs. Consider this example, "<*Author*> <*name*> *Rainer Unland* </*name*></*Author*>" the XPath */Author/first_name* is indexed as a document with two words: *Author* and *first_name*. The information stored for each index term is the set of XIDs that contains these index terms. Each query is the conjunction of Boolean queries or vector space queries [BR99] for XPaths and for the content of the documents. Hence, a query is evaluated in the same way as a Boolean query and the result is a set of XIDs, mapped to a set of document identifiers. For example, the query (*RUnland*) & (*Authors / name*), searches for documents with the word *RUnland* in the XPath *Authors/name*, where & is the conjunction of the expression for content followed by the expression for XPaths.

3.3 Indexing Semistructured Data (e.g., XML)

In semistructured indexing, it is not necessary to use all the structure information for indexing. In some approaches, a predefined structure is provided and information is fed into the structure provided. In other approaches, documents are allowed to have specific structure types (such as trees or segments) for indexing. Here, we concentrated on the XML documents as a well-known example of the semistructured data. The marriage between XML and relational data models has led to the need to index and retrieve information which is stored in XML format but which should be retrieved in a similar structure as with RDBs. It is therefore necessary to map a relational schema to the XML definition via DTD or XSL. The Agora [MFKX+00] is one that integrates XML data into an RDB through a generic schema. The text index is built as relational tables to improve the performance of user queries over the data, especially over the semistructured part.

As we described in Section 3.1, the key distinction between data in XML and data in traditional models is that XML data is not rigidly structured but a sensitively structured. In the relational and object-oriented models, every data instance has a schema, which is separated from, and independent of the data. In XML, the schema exists with the data (e.g., DTD) or may be no schema at all. From these points, the complexities of indexing XML documents are arise. According to this reason, we first describe several different approaches that propose to extract the schema from the data. Then, we describe several strategies for indexing XML data.

3.3.1 Schema Extraction Techniques

Query evaluation techniques based on traversing of the complete underlying graph are usually inefficient since there is no fixed schema known in advance. In traditional database systems, a query processor can only process queries targeted to specific schema.

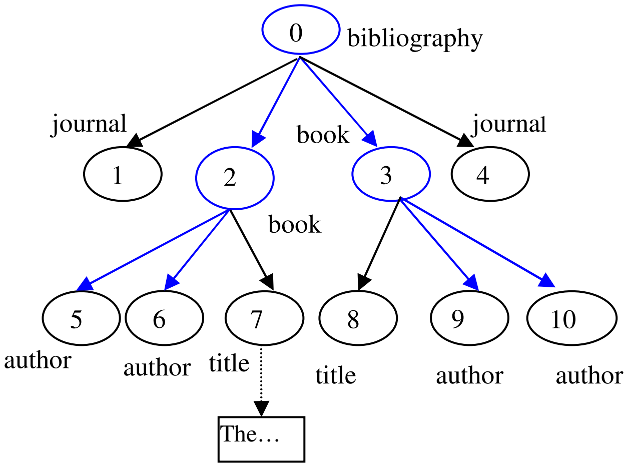


Figure 3.3: A sample OEM database

On the other hand, in XML data models, the entire data graph should be processed by a query processor. So, optimization techniques for queries by scheme extraction are proposed. We divide these techniques into two groups: one using *Automata* [ABS00] and the other using *simulation*. Moreover, through this section, XML data is represented as a graph structure, which is similar to Object Exchange Model (OEM). This model defines as follows:

Definition 3-1 (Object Exchange Model (OEM)): OEM is a simple data model. In the essence, an OEM database can be considered a directed, labeled graph. In OEM, each object contains an object identifier (oid) and a value. A value may be atomic values or complex values. Atomic values may be integers, reals, strings, images, programs, or any other considered indivisible. A complex OEM value is a collection of one or more OEM subobjects, each linked to a parent via a descriptive textual label. Note that a single OEM object may have multiple parent objects. Furthermore, cycles are allowed. For more details on OEM and its motivation, see [RG00].

Figure 3.3 presents a simple OEM database. Each object has an integer oid. This figure has one complex root node with oid "0". This root has four sub-objects: two books, and two journals. Each book node is a complex object. Each journal is an atomic object. Each book has an atomic object author and complex object title.

3.3.1.1 Schema Extraction Using Automata

Schema extraction techniques using automata are proposed in [GW97] [NUWC97] [TW02]. DataGuide in [GW97] [NUWC97] is a structural summary of semistructured databases. The technique regards a source database as Non-deterministic Finite Automaton (NFA) [TW02] and creates a DataGuide that is the corresponding deterministic finite automaton. Formally, a NFA has five parameters $(Q,\Sigma,\delta, q_0, F)$ corresponds to an object "*o*" in an OEM. NFA $(Q, \Sigma, \delta, q_0, F)$ is constructed as follows.

- $Q = \mathit{state}\ (D) \cup \{\text{end}\}$.
- $\Sigma = L \cup \{\perp\}$
- $\delta(\mathit{state}\ (c), l\ ) = \text{state (object (oid))}$ for all $c \in C$ and $<l, \text{oid}> \in \text{value}(c)$.
- $\delta(\mathit{state}\ (a), \perp\ ) = \text{end}$ for all $a \in A$.
- $q_0 = \text{state (o)}$.
- $F = Q$.

The function *state* maps every object within *o* to a unique automaton state corresponding to it and maps a set of objects within *o* to the set of automaton state corresponding to them. *A* denotes the set of all atomic objects within *o*. *C* is the set of all complex objects within *o*, *D* is the set of all objects within *o*. In addition, *l* is the set of all labels of object references within *o*.

To illustrate the equipping, consider the data graph in Figure 3.3. If all nodes are regarded as *states* in an automaton and all edges as *transitions*, Figure 3.3 represents a non-deterministic automaton. This means, for example, that there are two transitions labeled “book” out of root *state* (0), one going to *state* (2) and another going to *state* (3), and two transitions labeled “Journal” going to root *state* (0) and *state* (4).

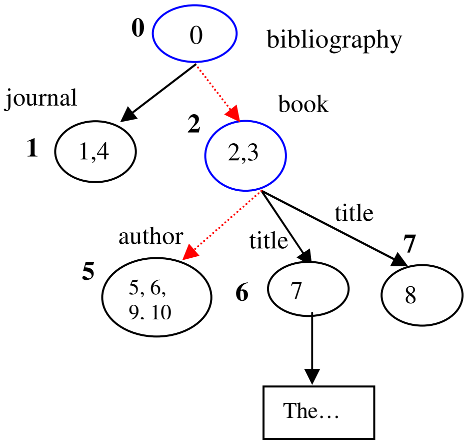


Figure 3.4: A DataGuide of Figure 3.3

Figure 3.4 shows a DataGuide for Figure 3.3. Since a DataGuide is a deterministic finite automaton, its size is small compared to the size of the source graph. This means that this technique can speed up query processing for semistructured queries, since the DFA can be used as a path index.

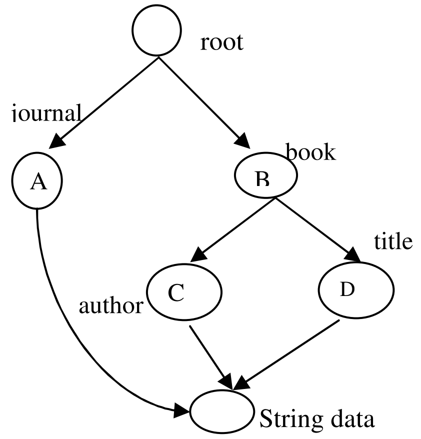
To explain in more detail why this technique is more powerful, consider the regular path expression *“bibliography.book.author”*. For evaluating this path query without a DataGuide, query processing starts from the root node *bibliography* and then would examine every *book* element. Then the *author* elements of each *book* would be investigated. Finally, the object would be return.

In the DataGuide technique, every object has a target set that denotes all objects reachable by a given label path source in the original graph. For example, the target set of object (**2**) in Figure 3.4 is the set {2, 3} in Figure 3.3. Therefore, the result of evaluating the above path expression using the DataGuide would be all objects in the target set of *"bibliography.book.author"*, i.e. The target set of object (**5**) is {5, 6, 9, 10} will be returned. The *dash lines* in Figure 3.4 explain this process, and it is clear that the object search space is reduced.

However, while this technique is more efficient when evaluating single path expression, but it cannot deal with complex path queries with several regular path expressions.

3.3.1.2 Schema Extraction Using Simulation

In this section, another technique to reduce the search space in query processing shall be explained. This technique depends on the idea of constructing a schema graph to which a data graph conforms; this way can reduce a data graph and create a new schema graph using the concepts of *simulation* [ABS00]. A simulation relation between two graphs G_1 and G_2 exists, if every edge in G_1 has a corresponding edge in G_2. Formally, given G_1 = (E_1, V_1) and G_2= (E_2, G_2), a relation R on V_1 and V_2 is a simulation if it satisfies:



Data node	Schema node
0	root
2 , 3	A
1, 4	B
5, 6, 9 ,10	C
7, 8	D

Figure 3.5: Schema graph with table containing data node and schema node

$\forall l \in L,\ \forall x_1, y_1 \in V_1,\ \forall x_2 \in V_2\, (x_1[l]y_1 \wedge x_1 R x_2 \Rightarrow y_2 \in V_2 (y_1 R y_2 \wedge x_2[l]y_2)).$

Where *L* represents the set of edge labels, *[l]* is a binary relation on $V=V_1 \cup V_2$. The notation *x[l]y* means that there is an *l*-labeled edge from *x* to *y*. The formal definition of a simulation between a semistructured data graph and a schema graph can be explained in the following:

- The root of the OEM data (source data, see Figure 3.3) and the root of the schema graph must be the same.
- An edge $x_1[l]y_1$ in the data graph can be simulated by some edges $x_1[l']y_1$ where *l'* is an alternation containing the label *l*.

For more explanation about this technique, see Figure 3.5. It shows the schema graph corresponding to Figure 3.3 and the table shows the simulation relation between the nodes. As we see, the size of the schema graph smaller compared with the data source. Hence, the search space in query processing can be reduced by traversing a schema graph extracted by simulation instead of the original graph.

It is noted from the above discussion that these two proposed techniques for schema extraction have the same objective to reduce the search space. These techniques are nearly the same based on the idea of minimizing the original graph into a small graph, and then evaluate queries using the small graph instead of the original graph. However, these techniques are still inefficient to evaluate the complex path expressions (e.g., paths with more simple path expressions) with wildcard ("//") on long paths.

3.3.2 Indexing Strategies for XML Documents

In this section, different strategies for indexing XML documents are described. The main difference between indexing XML documents and traditional databases (that have global schema) is that XML documents have no predefined schema like traditional data. The absence of the schema of XML data makes the indexing process more difficult. These strategies are divided into two types as follows:

3.3.2.1 Database Strategies

Quite large numbers of indexing strategies have been proposed for XML documents. One of the most common strategies is to map an XML structure to a relational database schema. This can be done by mapping the root element of an XML document to the name of a table residing within a RDB. The child elements of the document are then mapped to the particular attributes of that entity modeled by the table structure. However, this approach will only deal with less complicated XML documents. A more sophisticated scheme to map a XML document to a relational database can be achieved by using an object-oriented approach. In that case, the structure of an XML document mapped to classes and the resulting objects hierarchy is used to permanently store the data from the XML documents into a relational or object-oriented database structure for future retrieval. Once a mapping of either type is available, both approaches allow indexing and retrieval of the content of XML documents under the guises of the well-established relational model (see Section 3.5).

3.3.2.2 Information Retrieval (IR) Strategies

Most of the database techniques that map XML data to a RDB or to an object-oriented database have serious limitation when one considers their prospects for supporting rich IR applications, e.g., those are able to provide “relevance measures” to matching documents [BR99], divide XML indexing strategies for XML documents in IR applications into the following categories.

- *Field-based indexing:* Perhaps the simplest semistructured indexing method for XML documents is to represent a document as a set of fields. For example, each document may have an author field, a title field, and a publisher field. To allow searching restricted to certain fields, index terms are constructed by combining the field name with the terms from the content. For example, the index terms for *<Author> Albert Einstein </Author>* may be represented as Author: *Albert* and Author: *Einstein*, where the prefix Author: specifies the field in which the information appeared. This type of field-based indexing is actually common. For example, the ACM digital library and IEEE electronic library have various fields for users to specify keywords in titles, authors, and/or date, etc. This field data can be obtained in various ways. Firstly, the field data may be encoded as a meta-data in XML, for example, using RDF. Secondly, the field data may be information in the original document, being enclosed in XML tags. For example, legal documents have a header, which contains information about court cases (e.g., court level, case type, and parties). By adding XML tags for this header information, the authenticity of the legal documents are maintained and field-based indexing enables searching based on this header information. Finally, this field information may be derived automatically, using some information extraction techniques (e.g., [GMV00]). For instance, information for each field of job advertisements can be extracted from the original text descriptions, originating from various sources. The user can use a unified search interface, like a DB view, to find all jobs related to certain criteria.

- *Segment-based Indexing*: In segment-based indexing, XML document is divided up into regions (or segments) (e.g., all the chapter segments are selected and the content within these segments is indexed). An index of these contributions treats each region as a unique document to be queried and retrieved (e.g., [ST93] introduced the PAT expressions model algebra for text search). Unfortunately, it does not allow regions to overlap. XML elements that allow inclusion of an arbitrary number of elements and that allow arbitrary references may cause some concern with this model. Although the overlapping list model proposed in [CCB95] allows overlap, arbitrary nesting, which is needed for XML documents, is not fully supported. The indexing technique of the list of references model [Mac91] enables querying of hypertext using linkages, attribute management, and external procedures. This model handles hierarchical document structures, which makes it suitable for XML documents because XML documents have a basic tree structure (with references or links). However, the answers to queries always return to the top-level element of a document (i.e., search result nesting not is supported) and all returned elements must be of the same type.

- *Tree-based Indexing*: In a tree-based indexing (see Section 3.2.1), document structures are trees or hierarchies. Most XML documents can be considered to have this basic structure (optionally with *references, pointers,* and *link*). Most of the proposed indexes based on the tree-based are proposed for indexing structured documents. However, they can easily extend to deal with XML documents. A schema that indexes the actual tree structure of XML documents includes parent-child and sibling-relationships among elements. Navigation of the index is accomplished with XPath -like expressions (see Section 3.4.1). In the following, we present an example to illustrate this technique: In [Yoo96] has studied tree-based indexing schema for SGML documents, which is also relevant to XML documents. This schema is based on the model of a document tree as a complete k-ary tree. In this way a unique identifier (OID), which is an integer, is assigned to each node on the tree.

The parent of the current node can be found by looking for a node with the OID_p calculated based on the OID_c of the current node, i.e.

$$p = \left\lceil \frac{c-2}{k} - 1 \right\rceil$$

Since the OID for every node is stored, there is no need to store the complex tree structures, as the parent– child relationship can be recreated. They found that the inverted index scheme for *All Nodes With Replication (ANOR*) is the best in terms of retrieval speed and storage. The inverted index of ANOR has a set of inverted lists for each word. The inverted list stores items that are tuples of the form *(DID, EID)*, where *DID* is the document identifier and *EID* is the element identifier. When an item *(D, E)* is inserted into the inverted list of word *w*, then all the descendent elements of *E* must have *w* in their contents.

- *Path-based Indexing* : When using Path-based indexing (we already explained it in Section 3.2.2), two different kinds of index files are used. One for indexing the structure of the XML document and another for indexing the content of the XML document. Each unique XPath that occurs in a document is indexes and is given an identifier. The text that appears in that XPath is mapped to this identifier in the structure index.
- *Position-based Indexing* : In position-based indexing the document is regarded as a two-dimensional object. Different element tags within that object are

regarded as rectangular regions. In addition, the contents within those regions is indexed, and mapped to that particular region. This has been regards as advantageous because both a rendered version of the document instance and the original can be indexed using the same indexing schema.

There are at least two ways that these regions could be defined for each element. Since the basic structures for XML documents are trees, only regions nested one inside another is allowed. The advantage is, that the spatial relationships between regions mirror those logical relationships between XML elements. In this way, the logical relationship between XML elements can be readily translated into spatial relationships for matching. Another way to define regions is based on the smallest rectangles (or bounding box) enclosing an element, which may overlap with rectangular regions of its child elements.

3.4 XML Query Processing

As described previously in chapter 2, one of the main features of XML query languages is their ability to reach to arbitrary depths in the XML data graph. To do this, they exploit in some form, the notation of path expressions (see chapter 2). In addition, the HID index that is proposed in this thesis, is based on the idea of path expressions to evaluate several types of queries. So, in this section we explain in detail, how different kinds of path expressions can be evaluated over a large graph with a long path. Firstly, the following definitions for different types of path expressions that are used through this section and also through the thesis shall be introduced.

Definition 3-2 (Path Expression). A path expression is a sequence of labels (e.g., $l_1.l_2…$ $l_{n)}$ belongs to nodes, which together forms a path in a given graph. The result of the path expression is a set of nodes from the given data graph.

Definition 3-3 (Absolute Path Expression): An absolute path expression is a path starting at the root of a given graph.

Definition 3-4 (Simple Path Expression): A simple path expression is an absolute path expression without wildcards ("//' or "*"), usually it is a parent-child relationship.

3.4.1 Path Expressions Evaluations

Path expressions can be straight forwardly defined as a sequence of element type names connected by connectors such as "/" wildcards, which defines a parent-child (e.g., simple path expression) relationship or "//" wildcards, which defines an ancestor-descendant relationship (e.g., complex path expression).

For example, Figure 3.6 represents the XML data graph of digital libraries with additional XLink between documents. The path expression, *"/book//authors"* is used to find all titles of authors whose root element is book.

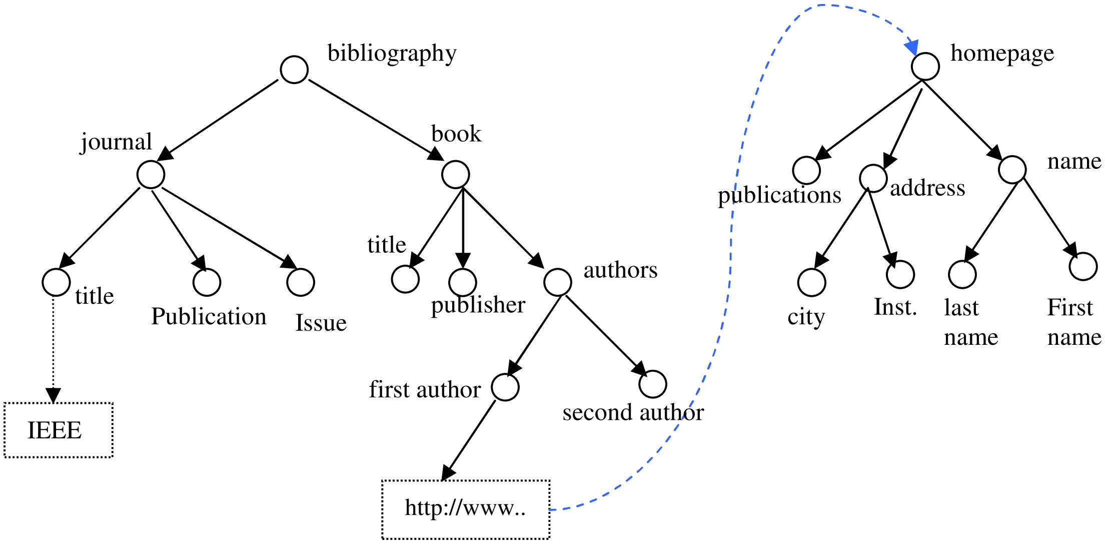


Figure 3.6: An Example graph-structured data

As can see, the path expression consists of two parts, *path steps,* and *connectors*. *Path steps* can be of two kinds, names, and wildcards "*". Path step names match element instances with a given tag names at a given step. Wildcards "*" on the other hand match any element, no matter which type it belongs to or in what depth in the graph it is. Between two path steps, a *connector* is found which determines the relationships between them. A connector "/" located at the beginning of the path expression means that the path begins at the root element; the following path step designates the root element name. A connector "//" means that the following path step is the descendant of the current element. For example, in Figure 3.6 all descendants of the *authors* element are designated by expression *"authors //"*, and all ancestors are designated by path expression *"// authors"*. The two connectors can be located between two path steps in order to specify a parent-child relationship using a connector "/" (e.g., book/title), or an ancestor-descendant relationship using connector "//" (e.g., book//authors).

Generally, the result of the path expression $p_1,p_2,\ldots,p_n$ on a XML data graph D is the set of nodes V_n such that there is exist edges (r, p_1,v_1), $(v_1, p_2,v_2),\ldots,(v_{n-1}, p_n,v_n)$ in the given XML graph, where r is the root. Path expression results are sets of nodes.

An important question arise here, *how can path expressions be evaluated on a given XML data graph?* There are several ways to answer this question. For example, many authors have tried to optimize the required evaluation steps. Nevertheless, most of them depend on the general idea of Finite State Automaton (FSA) [ABS00]. Given a path expression p and a XML data graph D, the result of p on D is computed as follows. First, construct the automaton A that corresponds to the given path expression p. Let $\{x_1, x_2, \ldots, x_n\}$ be the set of nodes in D (x_1 being the root element of the graph), and let $\{s_1, s_2,\ldots\}$ be the set of states in A (s_1 being the initial state). Compute a set of *closure* as follows: Initially, compute *closure* $\{(x_1, s_1)\}$, doing so until the *closure* does not change anymore. Then, choose a pair $(x, s) \in$ *closure*, and consider some edges $x \rightarrow x'$ in the data graph D and some transition $s \rightarrow s'$ in A, labeled with the same label. Add the pair (x', s') to the *closure* and repeat these steps. After the *closure* has reached a fix point, the resulting expressions consist of the set of nodes x for which the closure contains a pair (x, s), with s being a terminal state in A.

The major disadvantage of this technique is, in its worst case, that it has to visit each node x in the data graph D as many times as there are states in the automaton. However, in practically, it visits most nodes, at most ones, and may not visit portions of the given graph at all [ABS00].

3.5 Storing XML Documents in Database Systems

In this section, we want to answer the following question: *What is the best way of storing XML documents?* This is relevant since the performance of the underlying storage representation has a significant impact on efficient query processing. In the following, we will start with a discussion of the requirements of storing XML documents. Then we provide a classification of the different strategies used to store XML documents. The classification is based on the underlying system used for it (e.g., relational systems, object-relational systems, or native systems). To the best of our knowledge, there has no comprehensive performance study comparing these strategies. Consequally, it is still an open question *which of these strategies is the best?*

3.5.1 Requirements of XML Storage

The design decision to build XML applications on top of general-purpose repository systems using an XML interface raises the question *how* to physically organize the storage of the XML documents in the underlying system. A straightforward answer to this question would be that changes to the physical storage design are not necessary; only "XML view" with a one-to-one mapping onto the underlying storage structure is needed. An alternative is to design storage structures specifically tailored to XML. The motivation behind this approach is that the storage mechanisms of database systems heavily rely on the fact that the stored data has a rigid structure. This is the case, for instance, for today's hierarchical and relational database systems, which both strictly enforce the schema for any data stored. However, the semistructured data model of XML is more flexible, it is difficult to find a mapping schema to databases that would suit the needs of XML processing in general.

The following overview addresses what exactly are the requirements for *efficient XML storage management.* A storage management schema must cover the following aspects efficiently:

- lossless storage of XML documents,
- complete and efficient reconstruction of decomposed XML documents,
- support for processing path expressions on the XML document structure,
- support for processing of precise and vague predicates on XML content,
- navigation in XML documents,
- online updates of XML documents.

To cover these requirements is challenging since XML is a semistructured data. Consequently, the data and its structure are not independent. Instead, both of them are defined in the XML documents. In the following section, we will describe three strategies that are used to store XML in database systems.

3.5.2 Classification of XML Storage Techniques

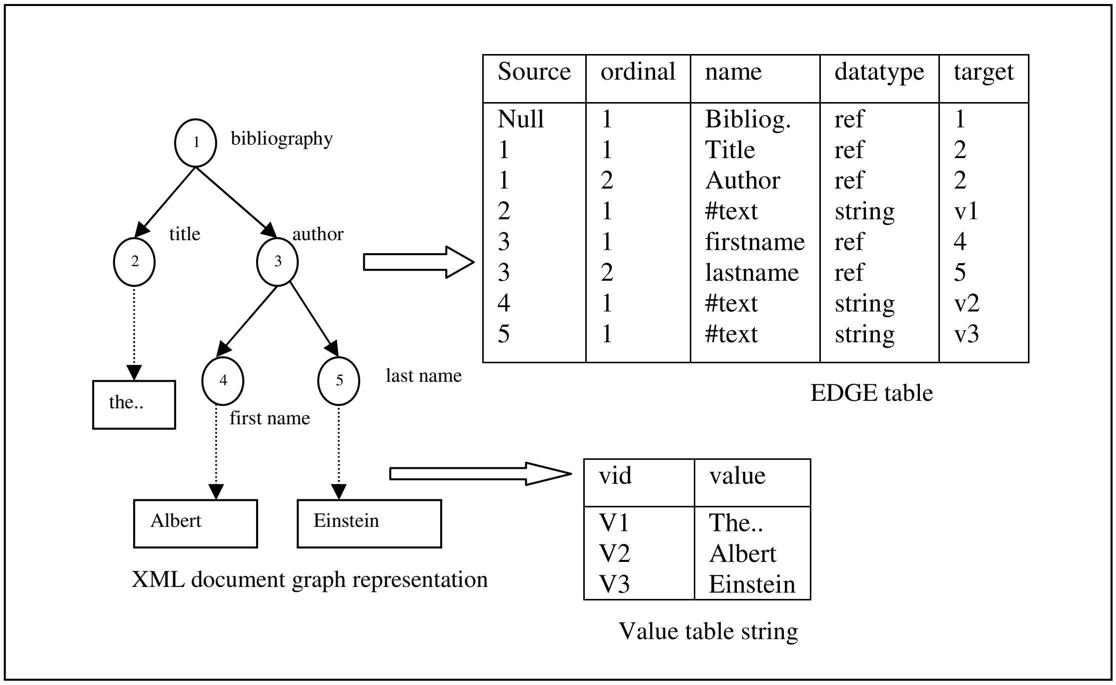
In this section, we describe different strategies that can be used to store XML in databases: storing XML in a Relational Database Management System (RDBMS), in an Object-Relational Database (ORDB), or in a Native Database.

3.5.2.1 Storing XML Documents in Relational Databases

Storing XML documents to database systems is attractive since important functionalities such as indexing and buffer management comes free. However, it is not obvious how to map XML data on tables. In literature, several approaches have been proposed to automatically map XML contents to database tables. Most of these approaches scan the XML documents first and then store all the information in relational tables. The schema design of these tables depends on the approaches chosen. In the following we will discuss in more detail the different fundamental approaches, namely EDGE model and BINARY model [FK99a] [FK99b], STORED [DFS99], and XPath Accelerator [Gru02]. Our database schema (see chapter 7) that we proposed for storing XML documents based on these approaches.

The EDGE Model

EDGE [FK99a] [FK99b] follows the intuition of storing the data according to the tree representation of XML document. A database table called EDGE stores the generic structure of XML documents, i.e. the edges of the tree representation of the document.



Source	ordinal	name	datatype	target
Null	1	Bibliog.	ref	1
1	1	Title	ref	2
1	2	Author	ref	2
2	1	#text	string	v1
3	1	firstname	ref	4
3	2	lastname	ref	5
4	1	#text	string	v2
5	1	#text	string	v3

EDGE table

vid	value
V1	The..
V2	Albert
V3	Einstein

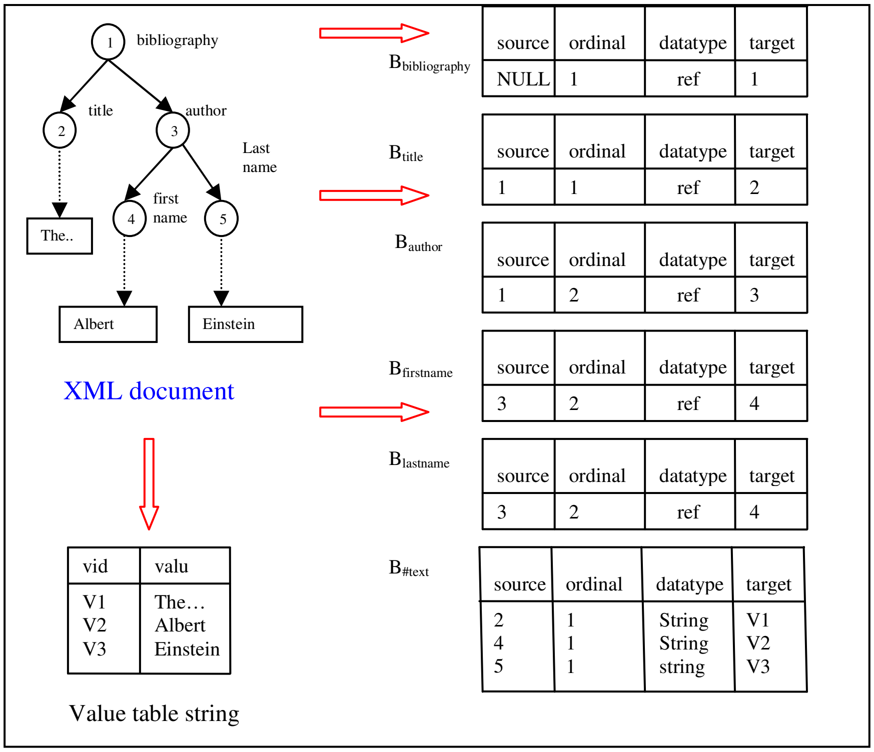
Value table string

Figure 3.7: XML document mapped with the EDGE model

As shown in Figure 3.7, the EDGE table has the attributes *source, ordinal, name, datatype,* and *target.* These columns store the identifiers of the source and the target nodes of each edge, keeping track of the order of the outgoing edges, and storing the label of the edges. The data type indicates whether the target node is an internal node or a leaf node. If the target is leaf node, a separate table stores the values of the leaf node. The major disadvantage of this approach is that many queries perform poorly since they require many joins on large EDGE table.

The BINARY Model

Similar to the EDGE model, the BINARY approach [FK99a] [FK99b] materializes the generic tree structure of XML documents in database tables. Hence, it is a model-mapping approach as well. As shown in Figure 3.8, this approach has separate tables for each element type. Nevertheless, the layout of the tables is identical to the EDGE approach (except for the elements *name* attribute that has been moved to the schema level).



$B_{bibliography}$

source	ordinal	datatype	target
NULL	1	ref	1

B_{title}

source	ordinal	datatype	target
1	1	ref	2

B_{author}

source	ordinal	datatype	target
1	2	ref	3

$B_{firstname}$

source	ordinal	datatype	target
3	2	ref	4

$B_{lastname}$

source	ordinal	datatype	target
3	2	ref	4

$B_{\#text}$

source	ordinal	datatype	target
2 4 5	1 1 1	String String string	V1 V2 V3

vid	valu
V1 V2 V3	The… Albert Einstein

Value table string

Figure 3.8: XML document mapped with BINARY model

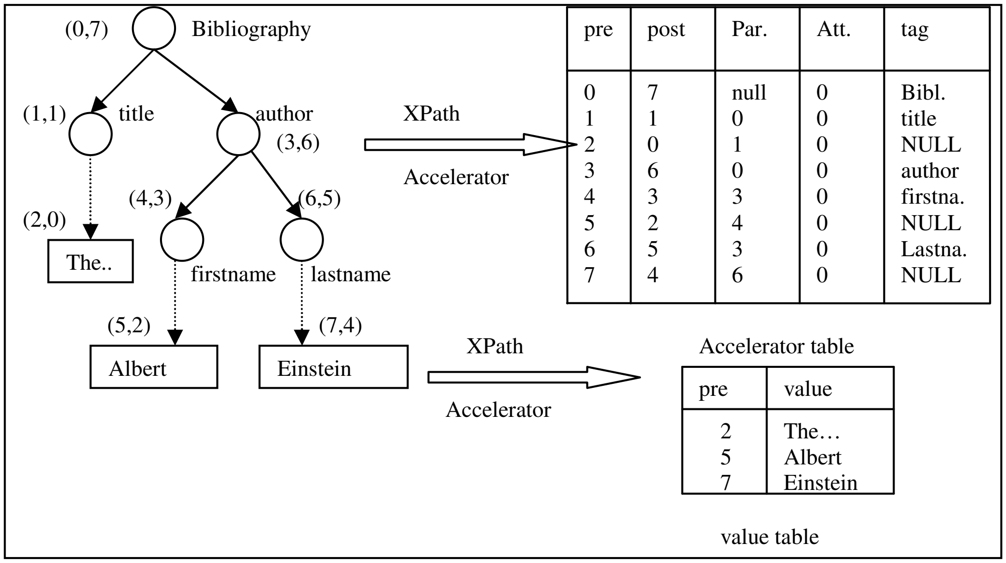
The main difference between the EDGE and the BINARY approach is that the table size in the BINARY approach is much smaller than the table size in the EDGE approach. But the amount of joins required to process path expression queries is the same.

The STORED Model

The STORED model [DFS99] automatically derives a relational schema from the given XML document using *data mining techniques* [BR99]. The mining algorithm used with STORED takes additional constraints into account, for example the request workload and resource limitations. It computes a mapping on a relational schema that considers constraints. With the STORED specification, there is an *overflow graph* to store XML content if it does not match any of the patterns defined by STORED. This makes the mapping lossless, i.e., the original XML documents can be reconstructed completely from the mapped data. The overflow graph can be implemented by using the EDGE approach. Hence, STORED is a combination of structure-mapping and model-mapping approaches.

XPath Accelerator

XPath Accelerator supported by [Gru02] is one of the most important index structures for efficiently evaluation of path expression queries (in particular, W3C XPath expressions). With simple additional measures included in the following explanation, it may also serve as a lossless relational mapping for XML documents. Briefly, the intuition behind the XPath Accelerator is as follows: when loading a new XML document, the XPath Accelerator performs a *pre-/post-order traversal* of the tree representation of the document. During the traversal process, every visited node in the tree has two ranks one for the pre-order traversal and the other for the post-order traversal (as shown in Figure 3.9). According to that mapping, the nodes are placed into a two-dimensional plane based on the pre-/-post coordinates (as shown in Figure 3.10).



pre	post	Par.	Att.	tag
0	7	null	0	Bibl.
1	1	0	0	title
2	0	1	0	NULL
3	6	0	0	author
4	3	3	0	firstna.
5	2	4	0	NULL
6	5	3	0	Lastna.
7	4	6	0	NULL

pre	value
2	The…
5	Albert
7	Einstein

Figure 3.9: XML document mapped with XPath Accelerator model

The XPath Accelerator can evaluate the path expressions for a given node *v* as follows:

- *all ancestors of v* are to the upper left of *v's* position in the plane,
- *all descendants* are to the lower right,
- *all preceding* nodes in document order are to the lower left, and
- the upper right partition of the plane comprises all following nodes.

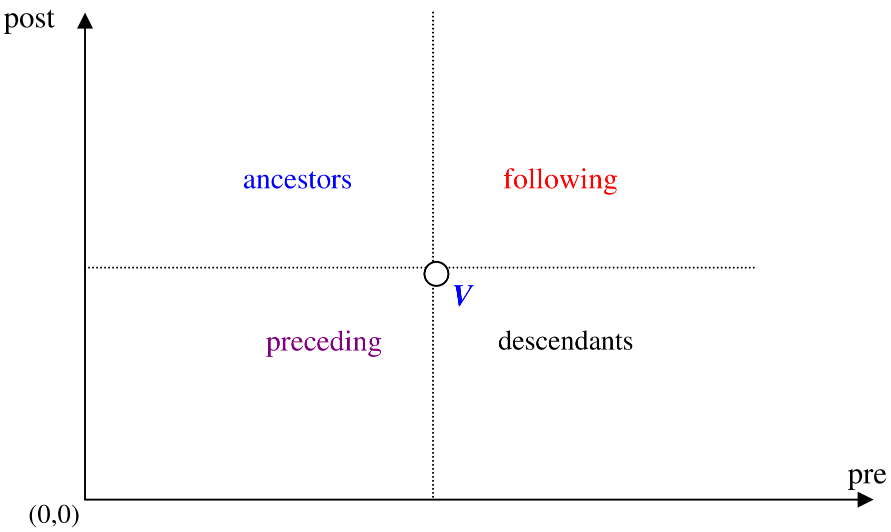
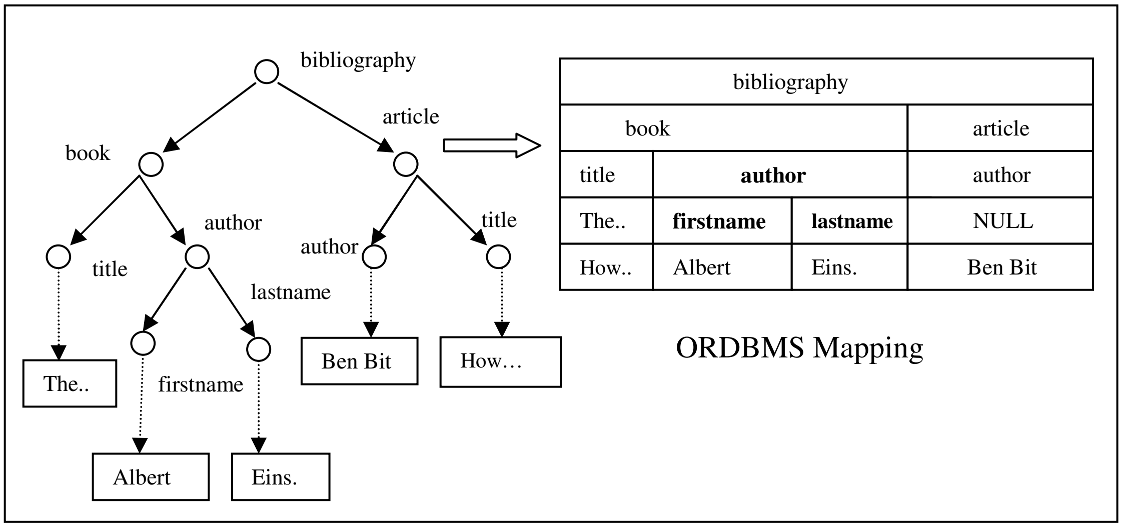


Figure 3.10: Node v distributions in pre-/post plane

The storing of additional information improves efficient evaluation of XPath expressions. Examples are the preorder rank of the parent node, an attribute flag, or the tag name of each node. Because XPath Accelerator works with a generic tree structure of XML documents, it is considered as a model-mapping approach.

The implementation of XPath Accelerator depends on B-tree and R-tree index structures. When comparing XPath Accelerator with other approaches it can be see that the implementation of XPath Accelerator on top of an RDBMS using B-tree indexes improves the responses times by at least a factor of 5, as compared with the EDGE approach. Using R-tree indexes instead of B-tree indexes yields additional advantages of the multi-dimensional index structures (see above). For instance, queries on the two-dimensional pre-/post order plane map can be put efficiently on multi-dimensional operations of the R-tree. This explains the performance improvements as compared to B-tree indexes. XPath Accelerator works very efficiently in case of a tree structure therefore. It is difficult to extend this to graph structure, though; the aim of this thesis is to propose an efficient index structure that can deal efficiently with XML graph structure, as well.



bibliography			
book			article
title	**author**		author
The..	**firstname**	**lastname**	NULL
How..	Albert	Eins.	Ben Bit

Figure 3.11: XML document mapped to an ORDBMS table

3.5.2.2 Storing XML Documents in Object-Relational Databases

A prerequisite for storing XML documents in object-relational databases is the definition of a data structure that sufficiently reflects the document structure. [STHZ+99] discuss the limitations and opportunities of mapping XML structures such as the ones defined by XML DTD, for instance to object databases. In addition to their proposal, the functionality of object-oriented database systems allows including further semantics of the XML structure into the database schema [KM00]. To give an example, complex XML element structures can be mapped to complex row types or nested database tables. SQL lists and sets can represent XML elements that appear repeatedly, i.e. those with a quantifier "+" or "*" in the XML DTD.

However, similar to relational database systems, object-relational database approaches cannot completely represent the flexibility of the XML format since XML relies on a semi-structured data model and XML structures are usually not strict. Consider Figure 3.11, which explains in more detail how XML documents can be stored in an ORDBMS. As shown in the Figure, we have two content models for the *author* element. The first one has two sub-elements *firstname* and *lastname*. The second *author* element does not have further sub-elements. It has only text content. The schema of the

ORDBMS table, which the document is mapped to, reflects this with two different columns for the *author* data. As the figure shows, one of these representations must be NULL depending on the content model chosen for a particular mapped element.

3.5.2.3 Storing XML Documents in a Native Database

Finally, we should have a look shortly at so-called native XML databases, which are specialized to store and process XML documents. Native storage schemas aim at efficient support for loading and storage complete documents as well as efficient navigation in documents.

A native XML storage system store XML documents as flat files, i.e., it uses a t*ext-based mapping*. However, evaluation of queries requires reconstructing the complete XML documents, which is not efficient when only parts of the documents are evaluated by the given query. As a result, most native XML storage schemas store XML documents as a tree structures based on the tree data model of XML. These particular approaches are model-mapping approaches. Usually, native XML storage systems rely on the DOM tree representation of XML documents.

4

Related Work

This chapter gives an overview on related work to the topic of this thesis. First, we introduce some necessary notation in Section 4.1. In Section 4.2, we discuss in detail several approaches for indexing XML data and divide these approaches according to the underlying XML model (e.g., tree-structure or graph-structure). In Section 4.3, several problems with the existing index structures are discussed. Since the way of storing XML documents has a direct impact in the way of indexing and querying XML, several schemas for mapping XML to relational database management systems are proposed in Section 4.4.

4. 1 Notation

In this section, we introduce all the definitions that we use through this chapter.

Definition 4-1 (Equivalence Relation "≡"): For each node *u* in the data graph, let the set $L_u = \{w \mid \exists$ a path from the root to node u labeled $w\}$. The set L_u may be infinite when the graph has cycles; however, it is always a regular set. Given two nodes *u* and *v* in the data graph we say that they are *language-equivalent* in notation $u \equiv v$, if $L_u = L_v$.

Definition 4-2 (Index Graph) : Index graph means that we reduced the graph that summarized all the paths from the root in the data graph, the nodes that have the same label from root are collected into one node called *index node*. The index graph is smaller than the data. Path expressions can be directly evaluated from the index graph and can retrieval label-matching nodes without referring to the original data graph.

Definition 4-3 (Bisimilarity)*:* Two nodes in the data graph are bisimilar (≈) if all label paths into them are the same. In other words, if node *u'* is parent of node *u*, node *v'* is the parent of node *v*. If the two nodes *u* and *v* have the same label, then, u ≈ *v* if u'≈ *v'*.

Definition 4-4 (*K*-Bisimilarity): For any two nodes *u* and *v* in the data graph. $u \approx^{K} v$ if and only if $\mathrm{u} \approx^{K-1} v$ and for every parent *u'* of *u*, there is a parent *v'* of *v* such that $\mathrm{u'} \approx^{K-1} v'$, and vice versa. Where *k* is the length of the label path to the node from the root. If nodes *u* and *v* are k-bisimilar, then the set of label paths of length *k* into them are the same.

Definition 4-5 (Deterministic): The relationships between two nodes can be quickly determined by examining their labels.

4.2 Classification of Index Structures

A number of research efforts have been made to introduce and investigate index structures, access methods suitable for efficient querying, retrieval of semistructured data collection, and XML databases. In this section, we start with a short classification of structures indexes for semistructured data by the navigational axes (more information about XPath axes is introduced in chapter 2) they support. *Structure index* supports all navigational for XPath axes. *Connection index* supports the XPath axes that are used as wildcards in path expressions (ancestor (descendant)-or-self-relationship and ancestor-descendant relationship). *Path index* supports only the following kinds of XPath axes (parent-child relationship,ancestor-descendant relationship, ancestor-or-self relationship, and descendant-or-self relationship*)*.

4.2.1 Structure Indexes

With the rapidly increasing popularity of the XML for data representation, there is a lot of interesting in a query processing on data that conforms to a labeled- tree or labeled-graph model. To summarize, the structure of such data in the absence of a schema and to support path expressions evaluation, several structure indexes have been proposed for semistructure data described as follows :

The structure index proposed in [Gur02] [GK03] presents a database index structure designed to support path expressions evaluation on trees. It has the capability to support all XPath axes and start traversal from any arbitrary nodes in an XML document. Building the index takes O (|E|), and space consumption is O (|V|), where V denotes the number of nodes in the XML tree and E the number of edges.

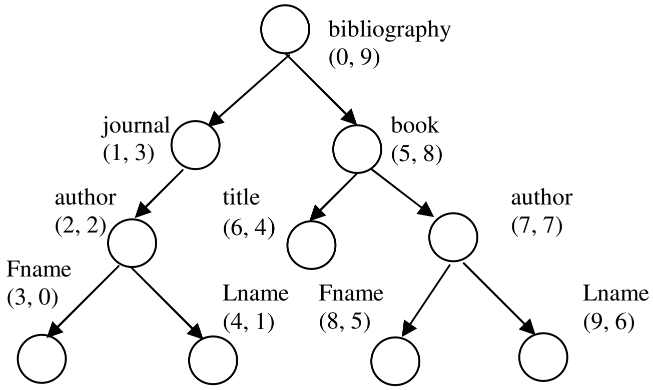


Figure 4.1: Pre/post schema encoding XML tree-structure

The main idea of this index depends on the numbering schema. It computes two numbers for each element name in the XML data tree, one representing the *pre-order* and the other representing the *post-order.* These numbers are the result of a depth-first search [CLRS01] on the XML data tree. Starting with the root element, the *pre-order* numbers are assigned in the order in which the nodes are visited during this search. *The post-order* defines the order in which the nodes are left. The authors explain that XPath axes (like ancestor and descendant axes) can be evaluated using these numbers. The tree shows in Figure 4.1 gives an example for this numbering schema. This index based on the following property for evaluating path expressions:

For any two given nodes A and B in the tree, an arbitrary node B is a descendant of a node A, if and only if this condition is satisfied:

$$pre(A) < pre(B) \text{ and } post\ (A) > post(B)$$

If we want to evaluate all descendants of a given node using this schema, then the result is the set of all nodes that satisfies the above-discussed condition.

The *pre-/post-order* approach can be determined in a constant time by examining the *pre-and post-order* variable of the corresponding tree nodes.

The drawback of this approach is its lack of flexibility in case of changes to the structure of the XML-document. That is, the *pre-/post-order* variables need to be recomputed for a number of tree nodes if a new node is inserted or an existing one is deleted.

XISS (**X**ML **I**ndexing and **S**torage **S**ystem) [LM01] is a system for indexing and querying XML data for regular path expressions. It also depends on the interval-encoding schema for each node in the XML tree. This schema uses an *extended pre-order* traversal that speeds up the processing of path expression queries and evaluates the ancestor-descendant relationship between two arbitrary nodes in the XML tree.

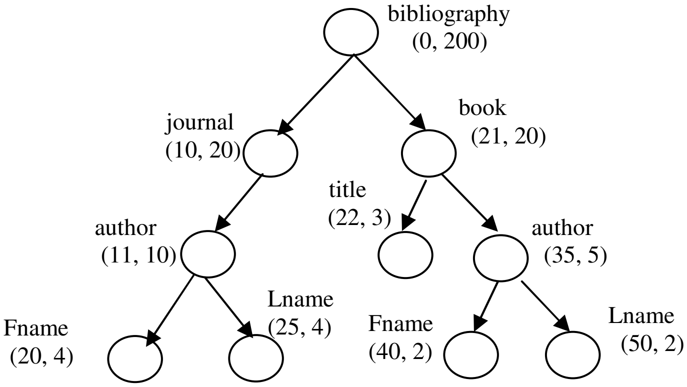


Figure 4.2: Extended Pre-/Post schema using (order, size) encoding

In this system, each node in the XML tree encoded with two numbers (*order, size*) (see Figure 4.2), with the following properties:

1. Every node is described by two variables. The first variable (*order*) is related to the *pre-order* of the nodes; i.e., it provides a total order on the nodes (in *pre-order*). The second variable (*size*) is a variable that fulfills the following condition. The sum of the size of all descendants of the node is smaller than the size of the node.
2. For a node *B* and its parent *A* the following holds: *order(A) < order(B)* and *order(B) + size (B) ≤ order(A) + size(A)*. In other words, the interval [order(B), order(B) + size(B)] is contained in the interval [order(A), order(A) + size(A)].

3. For two sibling nodes *A* and *B*, if *A* is the predecessor of *B* in a preorder traversal, then *order(A) + size(A) < order(B).*

4. Size() can be an arbitrary integer number larger than the total of the sum-variables of all descendants of A (*size(A)* $\geq \sum$ *size(B)*), for all *B's* that are direct children of *A*).

From Figure 4.2, we note that the schema is equivalent to the *pre-order* traversal. The schema guarantees that for a pair of tree nodes A and B, *order(A) < order(B)*, iff *A* comes before *B* in a *pre-order* traversal then the *ancestor-descendant* relationship for a pair of nodes can be determined by examining the *order* and *size* variables.

The interval of a child-node is contained in the interval of its parent. For example, the node (20, 4) is contained in nodes (11, 10) and (0, 200). It means that the node of *order* 20 is descendant of the nodes with *orders* 11 and 0.

Moreover, several index structures are also proposed in the literature depend on the labeling schema for rooted trees, for example, the authors in [ZAR03] [ZADR03] propose a new index structure called tree signatures for efficient tree navigation and twig pattern matching. They represent the XML tree as a sequence of preorder and *post-order* ranks. [CKM02] proposed algorithms to label the nodes of an XML tree. They design a persistence structural labeling schema by labeling each node immediately when it is inserted and this label remains unchanged. Using the information of two labels one can decide whether one node is ancestor of the other without traversing the entire XML tree. The *persistence structural labeling schema* is a pair *<P, L>* where, *P* is a 2-ary predicate over binary string and *L* is a labeling function that is given a tree T assigns a distinct binary string *L(v)* for each node v$\in$ T. The labeling function *L*, however, rather than getting an input a full tree, gets a sequence of insertion of nodes into an initially empty tree. The root is the first to be inserted. Each subsequence insertion is of the form "insert node *u* as a child of node *v*". (So when *u* is inserted, its parent *v* must already be in the tree). The function label *L* does not know the insertion sequence in advance, but receives it online. As each node is inserted, *L* assigns it a binary string. This label cannot be changed subsequently. The predicate *p* and the labeling function *L* are such that for every tree *T* and every two nodes *u*, *v* $\in$ *T, P(L(v), L(u))* evaluates to TRUE iff *v* is an ancestor of *u*.

4.2.2 Connection Indexes

Several approaches are proposed to evaluate all the ancestors of a given node and test the reachability between two given nodes. For example, labeling schema proposed in [KMS02] is called a *prefix-labeling schema* to handle a dynamic XML tree. The nodes in the XML tree are labeled such that the ancestor relationship is determined by whether one label is a prefix of the other. New nodes can be inserted without affecting the labels of the existing nodes. They define an assignment of binary strings to the edges of the tree, such that, the collection of strings associated with the outgoing edges from any node is prefix free, *a prefix free assignment*.

At the first, the simple prefix schema finds a prefix free assignment to the tree. Then, it is label every node *v* with the concatenation strings assigned to the edges of the path from the root node to *v*. For every assignment, labels are unique. Node *u* is ancestor of node *v*, iff the label of *u* is a prefix of the label of *v*. One major problem related to this approach is how to find an assignment that minimizes the sum of the lengths of the labels, unfortunately this problem is NP-hard [KMS02] means no optimal solution to this problem. The main goal of the work in [KMS02] is to find an assignment that minimizes the maximum length of the labels by using Huffman's algorithm [Huff02].

Several labeling schemes are proposed using the above technique, for example, [AKM01] [AR02] proposed a labeling schema for rooted trees that supports ancestor queries by assigning to each node in the tree a label which is a binary string. Given the labels of two nodes *u* and *v* it can be determined in a constant time whether *u* is an ancestor of *v* only by looking at the labels. Another labeling schema proposed on [WLH04], it takes the advantages of the unique property of prime numbers to meet this need. Answering the ancestor-descendant queries for a given two nodes by only looking at the labels (based on prime numbers). An analytical study of the size requirements of the prime numbers indicates that this schema is compact and hardly affected. Moreover, the authors introduced several optimization techniques to reduce the size of the schema.

Unfortunately, these indexing techniques were supposed to handle tree-structure data. Extension of these techniques to the context of graph data could be very difficult because of the possibly exponential number of paths in the graph. Moreover, it may require a lot of computing power for the creation process and a lot of space to store the index.

4.2.3 Path Indexes

Several XML path indexes have been proposed recently to deal with the query evaluation problem, from relatively simple way to highly efficient and space-effective structures. Most of these indexes are quite efficient in evaluating simple path queries. These indexes widely differ in space utilization, support for paths with wildcards (wildcard means the arbitrary long paths from source point to targets in XML graph). These path indexes depend on the *structure summaries* of the XML graph. *Structure summary* is an important technique for indexing XML arbitrary graph, in case the general schema of the information is missing. Using this summary of the data, one can evaluate the path expression queries without looking at the original data. In the following, we will describe these indexes in details.

DataGuide: An Index for Semistructure Data

The most influential and well-known index structure for semistructure data is the *DataGuide* [GW97]. It's main data structure is a tree- or a graph-shaped documents, consisting of all label paths which occur in the indexed document collection. Each label path from the document collection appears exactly once in the index tree (or graph). In other words, when two label paths in the document collection share the same prefix (which may comprise the whole path), the part, which is common to both, appears only once in the *DataGuide*, as if the document paths had been merged during the index creation. It further simplifies path matching. The *DataGuide* is intended to be *concise*, *accurate*, and *convenient* summary of the structure of the database. The source database is a database that we summarize. It is identified by its root object. Since the *DataGuide* describes every unique label path of a source database exactly once, so the *DataGuide* achieves *conveniess*. To ensure *accuracy*, specify that the *DataGuide* encode no label path that not appears in the source database. Finally, for *convenience*, the data structure of the *DataGuide* itself is an Object-Exchange Model (see Definition 3-1) so they can store and access it using the same techniques available for the OEM database.

The major disadvantage of the *DataGuide* is that, it's restricted to simple path expression and is not useful for complex path expression (like ancestor-descendant relationship) with long paths. The *DataGuide* allows only for top-down navigation starting from the root in XML document and cannot execute *backward navigation*.

A LORE System for Indexing Semistructured Data

A LORE System [MWAL+98] is a query language proposed by Stanford university for indexing and querying semistructured data that has lack of information about the schema. The Lore index try to avoid some problems related to *DataGuide*. To speed up query processing and path expression evaluation in a LORE database, they built four different types of index structures that can help to avoid the problem of scanning the complete database to evaluate the query. The first two indexes identify the objects that have specific values; the next two indexes are used to efficiently traverse the database graph. In the following, we will discuss these indexes in details.

1. *Value index (Vindex)* is built over all atomic objects of base type integer or real, that has an incoming edges with a given label *l*. This value index allows the query engine to quickly locate all objects reachable by an *l* edge and matching a comparison predicate. While we could have chosen to support a single label-independent *Vindex*, a specific describes incoming usually is known at a query processing time, so it is useful to partition the *Vindex* by labels.

2. *Text index (Tindex)* as we discussed above, *Vindex* is useful for finding values that satisfies comparison like =, <, *etc*. However, for string values, an information retrieval style keyword search can be very useful, especially for strings containing a significant amount of text. In this case, the *Vindex* is not powerful enough and a different indexing structure, called *Tindex* is proposed.

Text index is implemented using inverted lists [BR99] which map a given word *w* and a label *l* to a list of atomic values with incoming edge *l* that contains word *w*. Like *Vindex*

, *Tindexes* are created by the administrator for a given label, for the reason outline earlier, but the label can always be omitted for a full search.

3. *Link index (Lindex)* this index provides a mechanism for retrieving the parents of an object via a given label. A *Lindex* looks up a "child" object *c* and a label *l*, and returns all parents *p*, such that there is an *l*-labeled edge from *p* to *c*. The *Lindex* also supports looks up with no label, in which case all parents and their labels are returned.

4. *Path index (Pindex)* finds all objects reachable by a given labeled path through the database is an important part of query processing. A *Pindex* looks up for a path *p* return a set of objects *o* reachable *v* is *p*.

Indexing Rooted Paths with Values: Index Fabric

The *index Fabric* has been proposed in [CSFH+01]. It includes an efficient implementation of the *DataGuide* and a clever extension with values in a place of identifiers. To shorten the paths, labels are first encoded with one or more latter. All paths from the root to a leave containing data are prearranged as a sequence of encoding labels followed by the value as a string. To store the encoded strings, the method uses an efficient index for string, i.e., a Patricia trie [Knut98]. A Patricia trie is a simple form of compressed tree that merges child nodes with their parents. A balancing mechanism is added to Patricia trie to guarantee constant access time when searching for paths of length N. Path expressions including predicates or values for elements are performed as a string search. The *Index Fabric* does not keep information of non-terminal nodes. It keeps all label paths starting from the root element. The *Index Fabric* encodes each label path to each XML element with a data value as a string. Then inserted the encoded label path and data value into an efficient index for strings. The index block and XML data are both stored in relational database systems. The *Index Fabric* losses the parent-child relationships so this index is not efficient for processing partial matching queries.

ToXin: An Indexing Schema for XML

The index proposed in [RM01] is a main-memory indexing mechanism inspired by *DataGuide*, LORE index, and Access Support Relations [KM90]. It was designed for allowing fast access to elements in an XML document both for forward and backward navigations. It consists of two separate indexes: *path index*, for allowing evaluation of regular path expressions, and a *value index*, used for efficiently locating nodes in the document that satisfies the certain criteria. One disadvantage of *ToXin* index is compared with DataGuide or Lore index, the total size of the index is always linear respect to the size of the document.

All these indexes discussed above are suitable for simple path queries and cannot efficiently evaluate complex path expressions (ancestors or descendants queries) with wildcard. In addition to the above, most of them depends on tree-structure data and ignore the link information between documents. In the following, we will discuss several indexes proposed to deal with complex path expressions without wildcards and based on tree-/-graph structure data.

Indexing Template-compliant Paths: T-index

Like *DataGuide* [GW97], *1-index* [MS99] is intended to be used by queries that search the database from the root for nodes matching some arbitrary path expressions. *1-index* therefore, represents the same set of paths from the root like *DataGuide*. The main idea behind the index construction is the generation of a *non-deterministic automaton (NFA)* (see Section 3.3.1) to get more compact structure than the *DataGuide*. To construct the *1-index* of a data graph, the authors compute for each node the equivalence class (see Definition 4-1) using a bisimulation as equivalence relation.
Using bisimulation (see Definition 4.3) to deal with the index size and the construction cost problems that *DataGuide* index yields. Where the size of the *DataGuide* may be large as the database itself, while *1-index* is at most linear. Figure 4.3 explains the way to construct *1-index* (Figure 4.3 (b)) from the database graph (Figure 4.3 (a)).

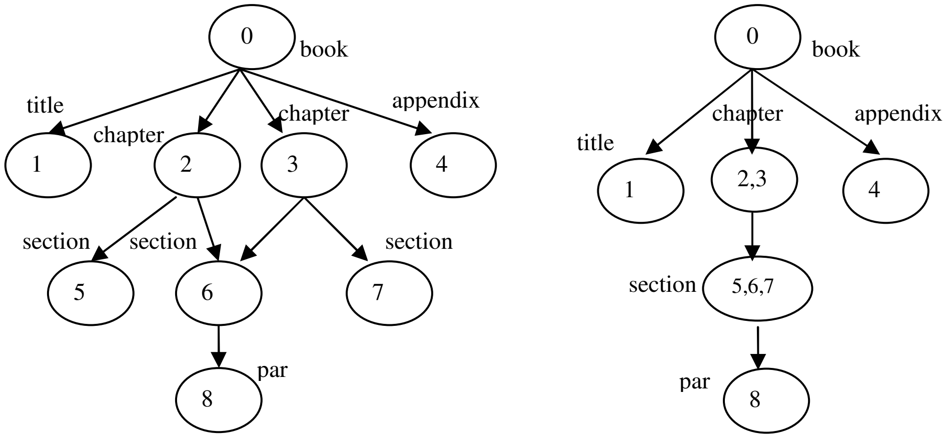


Figure 4.3: (a) Database graph (b) 1-index for Figure 4.3 (a)

The advantage of *1-index* and its family (*2-index* and *T-index* [MS99]) is that, it can be used to evaluate any path expressions accurately without accessing the data graph. However, the size of *1-index* can be quit larger for irregular XML data. Moreover, not all structures are interesting and most queries probably only involve short path expressions.

A(K)-Index:

Based on the above observations, *A(k)-index* [KSBG02] introduced the notation of *k-bisimilarity* (see Definition 4.4) to capture the local structures of the data graph. The *A(k)-index* can accurately supports all expressions of length up to *k*. Path expressions longer than *k* must be validated in the data graph. Taking advantages of local similarity [ABS00], the *A (k)-index* can be substantially smaller than *1-index* [MS99]. The parameter *k* control the "resolution" of the entire A (k)-index; all index nodes have the same local similarity of *k*. If *k* is too smaller, the index cannot support long path expressions accurately. If *k* is too large, the index may become so large. At this case, evaluating any path expression over this index will be expensive. The time required to build the index is *O(km)* where *m* is the number of edges in the data graph. Furthermore, not all path expressions of length *k* are equally common. The *A(k)-index* lacks the ability to make certain parts have higher resolution than the others do, so it can not be optimized for complex path expressions with wildcards.

D(k)-Index:

The *D(k)-index* is an adaptive summary structure for the general graph-structured data proposed recently in [QLO03]. It allows different index nodes to have different local similarity requirements that can tailored to support a given set of frequently used path expressions and to avoid the *A(k)-index* drawbacks. For parts of the data graph targeted only by longer path expressions, a larger *k* can be used for finer partitioning. For parts targeted only for shorter path expressions, a smaller *k* can be used for coarser partitions. However, as a generalization of *1-index* and *A(k)-index*, the *D(k)-index* processes the adaptive ability to adjust its structure according to the current query loads. *D(k)-index* has a very nice property compared with *1-index* and *A(k)-index* because of dynamics. The author provides an efficient algorithms to update the *D(k)-index* with changes in the source data . The general approach of the *D(k)-index* is flexible and powerful, but the index design still has several limitations that need to overcome. For example of these limitations, the construction procedure of the *D(k)-index* forces all index nodes with the same label to have the same local similarity, which is unnecessary and restrictive. The *D(k)-index* also proposes a promoting procedure that incrementally refines the index to support a given set of frequently used path expressions. This procedure increases the local similarity of an index node if it reached by a given set of frequently used path expressions in the index graph (see Definition 4-2). This index node will be partitioned into smaller nodes, all with the same increased local similarity. However, the problem is that in general the index node to be refined also points to data nodes that are irrelevant to the given set of frequently used path expressions.

M(k)-Index:

To overcome these limitations for the *D(k)-index*, A *M(k)-index* (for "*Mixed-k*") is proposed in [HY04]. Like *D(k)-index*, *M(k)-index*, it uses a k-bisimilarity equivalence relation but allows different *k* values for different nodes; it is also incrementally refined to support the new given set of frequently used path expressions extracted from the query workload. Unlike the *D(k)-index*, however, *M(k)-index* is never over-refined for irrelevant index or data node.

Another path index follows the same idea is *UD(k, l)-index* [WWYZ+02] generalization the *A(k)-index* by extending local bisimilarity to up-bisimulation and down-bisimulation, corresponding to upward and downward paths respectively. The index is especially efficient for branching path expressions. However, it also inherits the static nature of the *A(k)-index*.

Forward and Backward Index: F&B Index:

The *F&B-Index* proposed in [KBNK02] based on the notation of the *F&B-index* which it introduces in [ABS00] in the context of structural summaries. The main idea is to show that *F&B-index* is useful as a basic construct in the context of covering indexes (e.g., the indexes that have the ability to answer query without consult the base data) for branching path expressions (rather than simple path expressions) and to proposed techniques to control the size of the covering index.

Indexing Frequently Used Paths: APEX

The APEX index is an Adaptive Path indEx for XML data [CMS02]. *APEX* index keeps all paths of length two and utilizes frequently used paths to improve query performance. In contrast traditional indexes such as *DataGuide*, *1-indexes*, and the *index fabric*, it is constructed by utilizing the data-mining algorithm to summaries paths that appears frequently in the query workload. APEX has a nice property that it can incrementally updates to minimize the overhead of construction whenever the query workload changes. APEX keeps all paths of length two, so that any label path expressions can be evaluated by joins of extents in APEX without scanning the original graph. To support efficient query processing, APEX consists of two structures: the graph structure G_{APEX} and the hash table H_{APEX}. G_{APEX} represents the structure summary of the XML data extents. H_{APEX} keeps the information for frequently used paths and their corresponding nodes in G_{APEX} . Given a query, *APEX* uses H_{APEX} to locate the nodes of G_{APEX} that have extended require to evaluate the query.

Most of these indexes have mostly focused on constructing index structures for paths without wildcard ("//"), with poor performance for evaluating wildcard queries. In addition, they have not taken into account the internal-document link (ID/IDREF(S)) or the global-documents link (XLink). Recently HOPI index [STW04] proposed to deal with linked XML document; it makes use of compact representation of reachability and distance queries information in graphs proposed in [CHKZ02]. HOPI index provides a divide-and-conquer algorithm for index creation that is reasonably fast as long as the XML document collection is too large. The major problem of this index is; its size increases with the size of the XML documents.

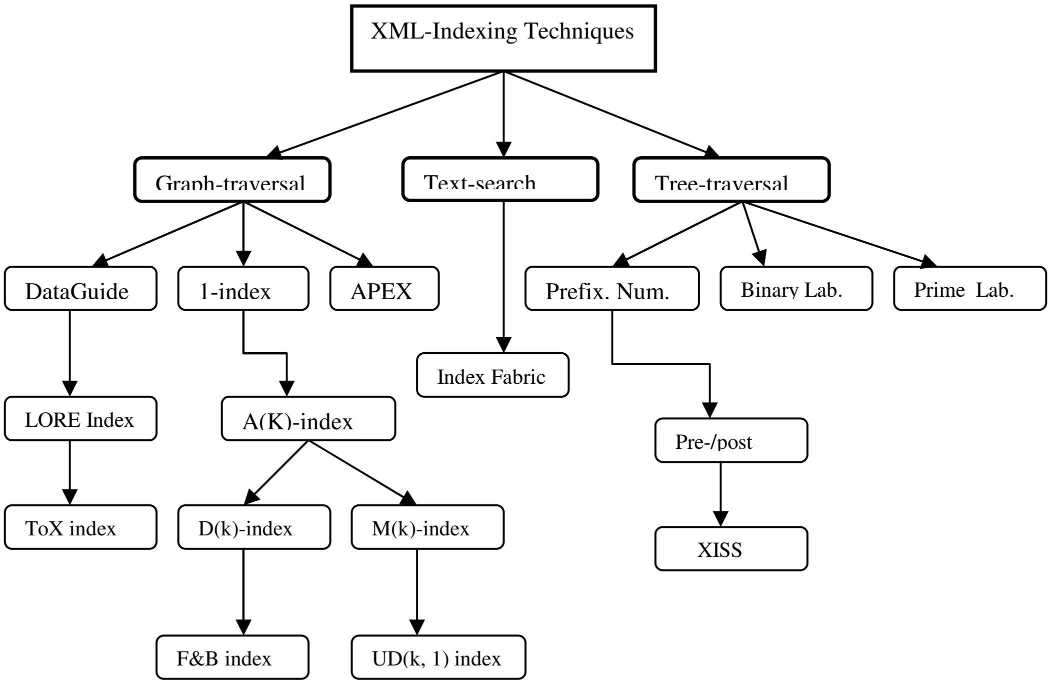


Figure 4.4: Classifications of XML indexing techniques

To conclude, we classify all the indexing XML strategies that we discussed above in Figure 4.4. The first strategy is “Tree-traversal”, based on the single rooted XML tree. They usually traverse XML tree and give every node in the tree two numbers based on Pre-/-Post numbering schema. This technique usually needs additional join algorithms to evaluate path expressions. The second strategy is ” Graph-traversal”, based on the idea of minimizes the underlying XML graph to small graph without change the structures of the original graph, and then evaluate path expressions over this small graph. This technique ignores link information between XML documents and cannot handle all XPath expressions. The third strategy proposed for “Text-search” based on the index Fabric.

4.3 Problems of Existing Index Structures

1. Structure indexes have been proposed to deal with tree-structure data and cannot be extended to work with graph-structure data. Such that these approaches depend on encoding each node in the XML tree-structure with numbers. They need join algorithms to evaluate path expressions. Several structural join algorithms are proposed [KJKP+02] [CVZT+02] [JLWO03] that help to evaluate path queries, these algorithm are more costs in time and in size. Moreover, these indexes are efficiently evaluate path expression of type parent-child relationship and they mostly fail to evaluate path expressions with wildcard “*” and “//” (arbitrary long path query). They usually start from the root node during the evaluation of the path expressions and ignore the references between nodes inside or between XML documents.
2. For *Connection indexes*, these indexes deal with XML documents represents as tree-structure. Moreover, they cannot extend to deal with XML graph structure with long paths. These indexes have not the capability to evaluate the descendant-or-self axis with wildcards.
3. *Path indexes* have been proposed to evaluate path queries for XML graph structure data. They can efficiently evaluate path expressions without wildcards (e.g., “*’ and “//”). However, they also have not the capability to evaluate descendant-or-self axis over long path. In addition, these approaches cannot deal with linked XML graph that have many cycles. Such that, all the proposed indexes cannot deal with XML graph that have many cycles.
4. If we take references between nodes into account, the structure of the XML document is no longer tree but becomes a directed graph with long paths. Thus, to efficiently evaluate path expressions queries (especially those with wildcards) with a descendant-or-self axis an appropriate index structure is needed, in case of most of

the above indexes are not efficient. The objective of the thesis is to propose an index, which can deal efficiently with these problems.

4.4 Mapping XML Data into Databases

Type of storage selected for XML documents greatly influences the querying and the indexing of the stored data. In the recent times, a lot of effort has been applied to study storage alternatives for XML documents [TDCZ00]. Techniques ranging from storing XML data as files (in the file system) to employing relational or object-relational databases have been developed. In addition, several attempts to develop native storage mechanisms for XML data have been made. However, a long time will pass until this native technology reaches a mature point of developments, and therefore, alternatives strategies have to be exploited. Among these alternatives, relational databases are one of the strongest candidates. In this work, we will focus on several strategies that used to map XML data into relational and object-relational databases.

4.4.1 Mapping XML Data into Relational Databases

In this section, we discuss and compare previous work on mapping XML to relational database. Research projects such as SilkRoute [FMS01] [FMST01] and XPERANTO [CFIL+00] [SSBC+00] have proposed techniques for efficiently publishing relational data as XML. Commercial database such as SQL server 2000 [Oracle] OPENXML statement can insert or update records of a relational table by specifying meta-properties. Oracle XML SQL Utilities [Oracle] can extract data from XML document, then do insert, update, delete to a relational table, and IBM DB2 XML Extender [DB2] either can store a whole XML document in one column of a relational table or can decomposed XML documents into a set of tables at load time with the mapping from DTD to relational table defined by DAD (Data Access Definition).

The database researchers also support several approaches to store XML data into relational databases have been proposed, [FK99a] [FK99b] describe various alternatives to store XML into RDBMS. They represent XML documents as ordered edge-labeled graph, they divided the mapping problem into: a) mapping elements and subelements and b) mapping values. They depend on the following three Approaches: The *Edge Approach* (see Section 3.5), each edge in the edge-labeled graph is stored as a tuple, in a single relational table. The *Binary Approach* (see Section 3.5), groups all edges with the same label into the same relational table. Finally, the *Universal Approach*, stores all edges in the same table, which is corresponding to the result of the outer join of the binary tables obtained by *Binary Approach*. The authors proposed also two different approaches to store XML values in relational tables: the first one distinguishes XML values by data types, storing each type in different relational table. The second technique stores XML values in the same relational tables, together with their corresponding elements and subelements, using different columns for each data type. Some known drawbacks of all the proposed mappings are the number of the join required to querying XML documents and in some cases, it required the schema to evaluate the path queries, moreover, these strategies have no information about link elements in-or-between XML documents that we considered in our thesis.

Moreover, in [STHZ+99] proposed an approach examined how to map XML to a relational database given the DTD of the file. This approach is used the number of join operations performed as its performance matrix and not response times for running real queries against real XML data sets. [SYU99] proposed the method that decompose the XML documents into nodes, and stored them in relational tables according to the nodes types. They defined a user data type to store a region of each node within a document. This data type keeps positions of nodes, and the method (associated with the data types) determine ancestor-descendant and element order relationships.

4.4.2 Mapping XML into Object-relational Databases

In this section, we discuss and compare previous work on mapping XML to object-relational database. In [RP02], present an algorithm called *XORator (XML to OR Translator)* for mapping XML to tables in an Object-Relational Database Systems by using the DTD of the XML documents. An important part of this mapping is the assignment of a fragment of an XML documents to a new data type, called *XADT* (*XML*

Abstract Data Type). In [SYU99], an XML document is decomposed into simple paths, and stored in an object-relational database. The [SYU99] mapping can be classified as a node-oriented approach, because it maintains nodes rather than edges. This approach stores for each node in the XML graph, a single path and a pair of numbers associated with its starting and ending positions. The pair is called a *region*, and maintains the containment relationship (ancestors and descendant relationships). The author using four relational tables for storing XML: *Element*, *Attribute*, *Text*, and *Path*. The first three tables store nodes of type element, attribute, and text respectively. A path tables stores information about simple paths. Due to this storage mechanism, answering regular path queries can become very inefficient, because each simple path has to be tested to determine if it satisfied the regular path query. Then, all paths that satisfied the regular path query have to be retrieval. [DFS99] proposed the STORED system for mapping between semistructured data model and the relational data model. They adapted a data-mining algorithm to identify highly supported patterns for storage in relations.

5

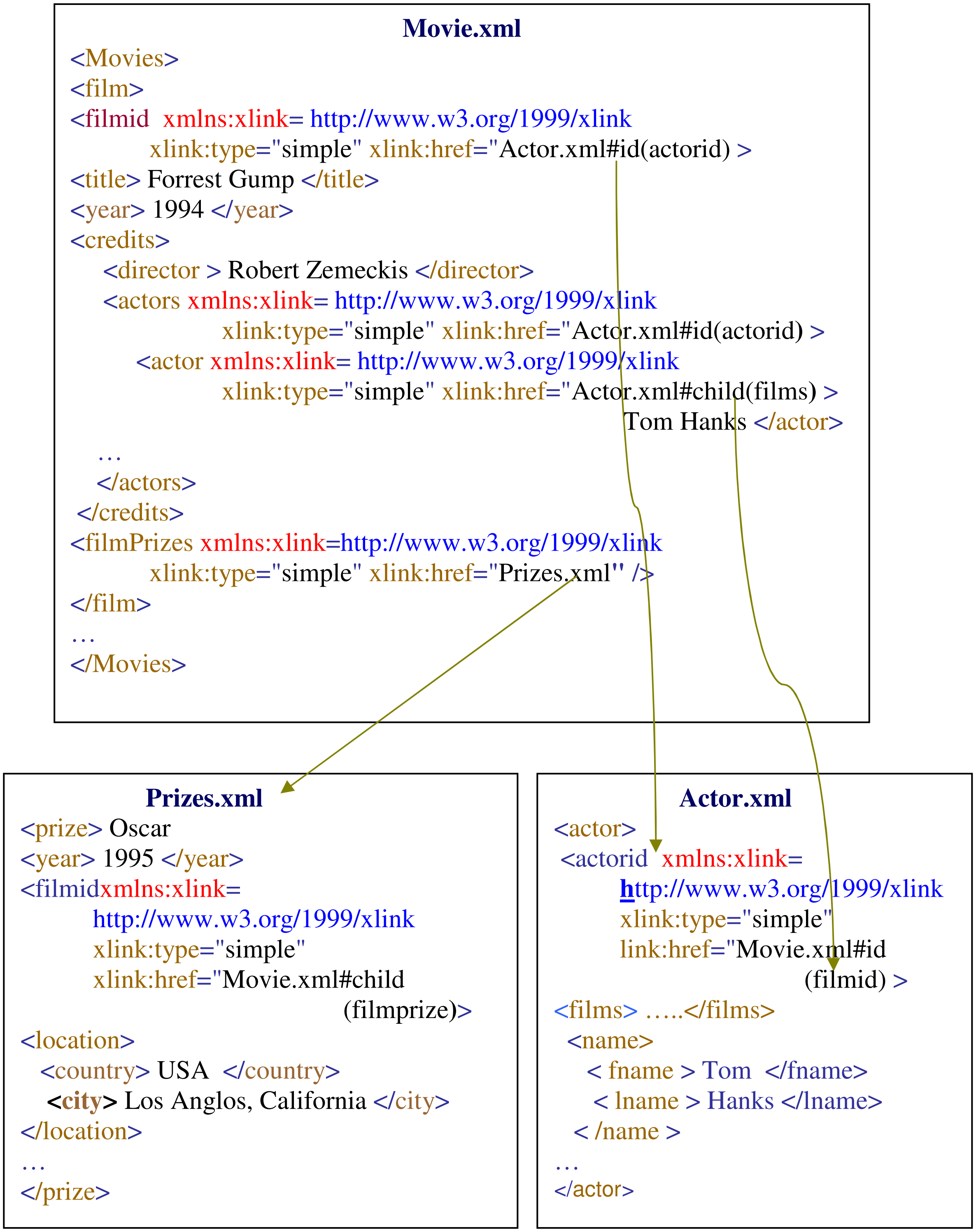
An Efficient Path Index for XML Documents with Arbitrary Links

5.1 Motivation

The problem of efficiently managing and querying XML documents poses interesting challenges on database research. Not only can XML documents have a rather complex internal structure with (ID, IDREF) link relationships but they can also be connected to other XML documents via links, (XLink and XPointer). Nowadays, HTML documents are more commonly being replaced by XML documents. These internal-document links are increasing on popularity. Link relationships replace tree-like structures by graph-like structures, which may contain *cycles*. XML query languages usually deploy regular *path expressions* to query data by traversing the entire XML document. To efficiently evaluate path queries, algorithms are utilized. Moreover, these algorithms rely on index structures. Whilst many techniques exist to efficiently construct and maintain such index structures on *trees*, these same techniques usually are not suited to fulfill the same task on *graphs*.

The reason for this is that, too much time is required to construct the index, and a great deal of space is required to store and work with the information. Consequently, queries like *ancestor-or-self* and *descendant-or-self* relationships with wildcards cannot usually be answered in reasonable time.

In this chapter, the HID index (an efficient path index for highly connected XML collections) shall be examined in detail.



Movie.xml

```
<Movies>
<film>
<filmid  xmlns:xlink= http://www.w3.org/1999/xlink
         xlink:type="simple" xlink:href="Actor.xml#id(actorid) >
<title> Forrest Gump </title>
<year> 1994 </year>
<credits>
    <director > Robert Zemeckis </director>
    <actors xmlns:xlink= http://www.w3.org/1999/xlink
                  xlink:type="simple" xlink:href="Actor.xml#id(actorid) >
        <actor xmlns:xlink= http://www.w3.org/1999/xlink
                  xlink:type="simple" xlink:href="Actor.xml#child(films) >
                                                              Tom Hanks </actor>
   …
   </actors>
 </credits>
<filmPrizes xmlns:xlink=http://www.w3.org/1999/xlink
         xlink:type="simple" xlink:href="Prizes.xml" />
</film>
…
</Movies>
```

Prizes.xml

```
<prize> Oscar
<year> 1995 </year>
<filmidxmlns:xlink=
        http://www.w3.org/1999/xlink
        xlink:type="simple"
        xlink:href="Movie.xml#child
                                        (filmprize)>
<location>
  <country> USA  </country>
   <city> Los Anglos, California </city>
</location>
…
</prize>
```

Actor.xml

```
<actor>
 <actorid  xmlns:xlink=
         http://www.w3.org/1999/xlink
         xlink:type="simple"
         link:href="Movie.xml#id
                            (filmid) >
<films> …..</films>
  <name>
    < fname > Tom  </fname>
     < lname > Hanks </lname>
   < /name >
…
</actor>
```

Figure 5.1: Three XML Documents from IMBD

5.1.1 Motivating Example

Let us consider this representation of XML data on which comments, processing instructions, and namespaces were removed. Then, an XML document can be modeled as a directed graph *GD= (VD, ED*). *VD* is a set of nodes that represents element names or attribute values. *ED* is a set of edges, which contains element/subelement or element/ attribute edges. Each node in *VD* is assigned a string label and has a unique identifier.

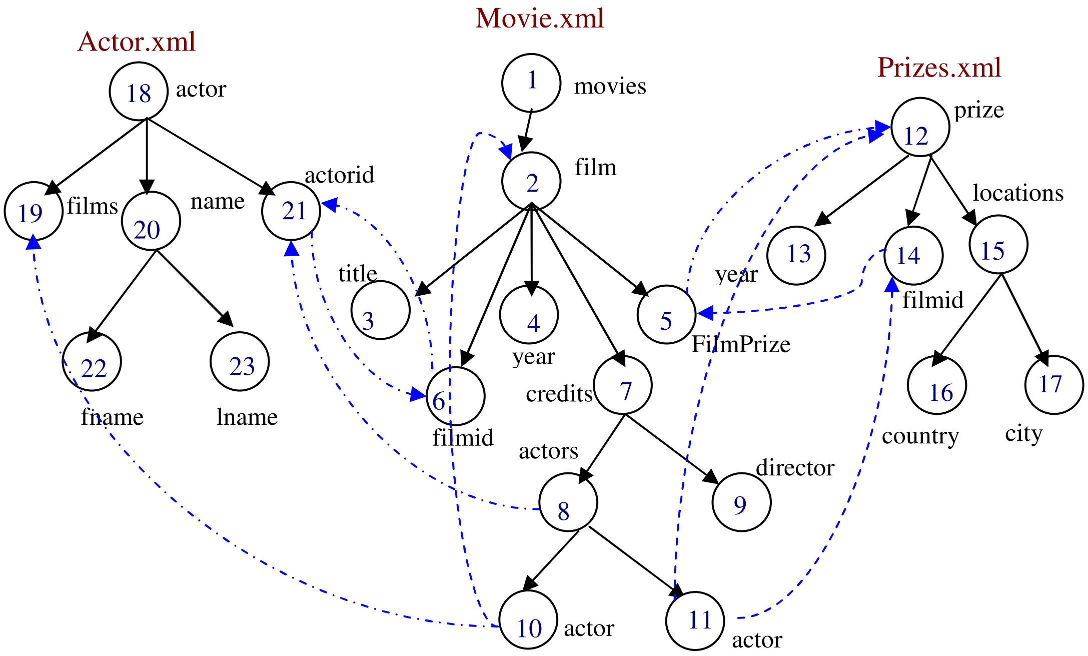


Figure 5.2: XML-graph representations of Figure 5.1

The union of all XML document graphs *G1,…, Gn* form a large XML graph *G= (V, E, EA, EL). V* and *E* are defined as above. *EA* is the set of all directed edges, where an edge represents the relationship between an element and a value. This representation is in turn expressed by an XML attribute. *EL* is the set of directed edges that represents element/element relationships via *ID* and *IDREFS* attributes as defined in the XML schema or DTD. Each node in the graph G represents an element, an attribute name, or an attribute value. An element node is an object that contains additional information. Each element node has a label, the URL of the document (in which it occurs), and its identifier. The diagrams in Figure 5.1 describe the structure of three XML documents from a real world scenario (an Internet Movie Database (IMDB)), which is in effect, a set of highly connected XML document collections.

To understand the problem of indexing data that has no prior schema, the traditional relational and object-oriented database systems shall firstly be examined, in order to see how these systems query data. In brief, these traditional systems force all data to adhere to a specified schema [GW97]. This schema serves two important purposes:

- A schema, in the form of either tables and their attributes or class hierarchies, helps users to have all the information about the underlying database, and thus attain better results from queries.
- A query processor relies on the schema to devise efficient plans for computing the results of the particular query.

These two tasks become very difficult, if the schema is "absent". Furthermore, a lack of information about the structure of the database will cause a query processor to carry out exhaustive searches.

From Figure 5.2 (e.g., represented linked XML documents) the following characteristics can be noted:

- XML documents are represented as large graphs with long paths.
- Graphs may have cycles, which may hinder or even prevent the evaluation of path queries.

From this, an important question arises. This is "How can path queries (with wildcard //) be efficiently evaluated within this instance?"

For this, the HID index is proposed, which deals with three types of path queries:

1. *The descendant-or-self axis* with wildcards (// axis), considering a child axis ("/") as a special case over long paths.
2. *The ancestor-or-self axis* with wildcards (// axis) over long paths.
3. *Ancestor-descendant queries* or reachability queries (*a*//*b*). This relationship is satisfied if there is a path from the ancestor node *a* to the descendant node *b* over long paths.

To illustrate the above, consider the following path expressions: At first, *"all the credits for the film Forrest Gump"* (see Figure 5.2) are sought. The normal path expressions to evaluate this query results in the following: “/credit/director”, “/credit/actors”, and “/credits/actors/actor”. Therefore, we need three path expressions to evaluate this query. This means that normal path expressions are not sufficient in the case of complex XML graphs with long paths. Here, dealing with the *descendant-or-self axis* is needed. We can replace these three path expressions with a single path expression with wildcards (// axis) where “*all the descendant of a credit*” (e.g.,” credit //”) is sought. From this example, it can be concluded that *descendant-or-self axis* is important in minimizing the evaluation steps of path expressions over large graphs with long paths.

The second path expression determines whether there is a path between the ancestor node “actorid” and the descendant node “locations” (e.g., “actorid//locations”). The traditional way is to start from the “actorid” node and to follow all available paths until the “locations” node is reached. This traditional method contains many problems. For example, we can follow all the paths from the “actorid” node in the same fashion, but perhaps not arrive at the “locations” node, meaning that there is no path between them. Moreover, in the worst case, if the “actorid” node is the root of the graph and the “locations” node is a leaf node in the graph, we need to traverse the entire XML graph to test the reachability. This shows that the method in question is not suited to fulfill the objective. However, the HID index deals with this problem efficiently.

Finally**,** an XML graph may have *cycles*, as demonstrated in Figure 5.2. These *cycles* may prevent the evaluations of path expressions. As can be seen from Figure 5.2 running in cycles is to be avoided, one would never attain answers to queries such as, “/filmprize/filmid” or “/actorid/filmid”. However, if we permit moving in *cycles* we will inevitably incur complications with the following path query: *“is there a path between actor node and filmid node”*. Therefore, *cycles* in the graph can stress any path indexes and prevent to evaluate queries. Therefore, the removal of these cycles from the XML graph before querying is an important task.

The following definitions explain the two most basic possible ways to represent XML graphs in database systems

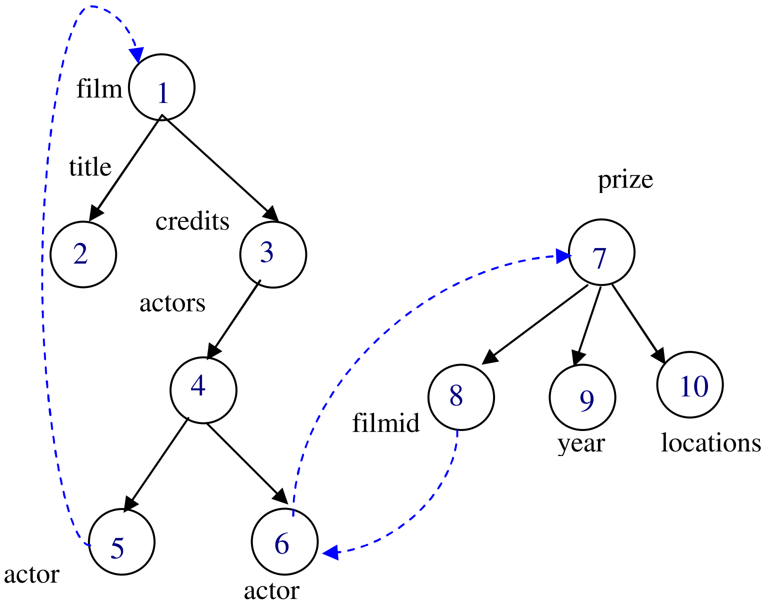


Figure 5.3: Part of the large XML graph in Figure 5.2

Definition 5-1 (Adjacency-list):

There are two possible ways to represent a graph G: As a collection of *adjacency-lists* or as an *adjacency-matrix*. Usually the *adjacency-list* representation is preferred if the underlying graph G is sparse (a graph with relatively few edges). *The adjacency-matrix* representation is preferred in case of dense graphs (a graph with many edges). Furthermore, *the adjacency- matrix* may be used to determine quickly if there is an edge between two given nodes.

At first, we explain how to represent a graph G = (V, E), where V is a set of nodes and E a set of edges using *adjacency-lists*. This representation consists of an array *Adj* of |V| lists, one for each node in the graph. For each node *u ∈ V*, *the adjacency list Adj [u]* contains all nodes *v* where there is an edge *(u, v) ∈ E*. Therefore, *Adj [u]* consists of all the nodes adjacent to *u* in graph *G*. An XML graph is usually a directed graph. Thus, the sum of the length of all adjacency lists is |E|, since an edge of the form *(u, v)* is represented by having *v* appears one in *Adj[u]*. Figure 5.4 (a) describe the adjacency list representation of the XML graph in Figure 5.3 [CLRS01]**.**

1	2	3	
2			
3	4		
4	5	6	
5	1		
6	7		
7	8	9	10
8	6		
9			
10			

	1	2	3	4	5	6	7	8	9	10
1	0	1	1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0
4	0	0	0	0	1	1	0	0	0	0
5	1	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	1	1	1
8	0	0	0	0	0	1	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0

Figure 5.4: (a) Adjacency-list of Figure 5.3 (b) Adjacency-matrix of Figure 5.3

Definition 5-2 (Adjacency-matrix):

The *adjacency-matrix* representation is preferred for large graphs (e.g., the XML graph in Figure 5.2 that has many edges). The *adjacency-matrix* representation of a graph *G= (V, E)* consists of a (V×V)-matrix A= (a_{ij}) therefore,

$$a_{ij} = \begin{cases} 1 & \mathit{if}\ (i,\ j) \in E, \\ 0 & \mathit{otherwise} \end{cases}$$

We use this data structure to represent an XML graph structure based on the identifier that is assigned to each element node of the XML graph structure. Figure 5.4 (b) explains *the adjacency-matrix* representation of the XML graph in Figure 5.3. The *adjacency-matrix* of the graph requires $O(V^2)$ memory space independent of the number of edges in the graph [CLRS01].

5.2 Basic Terms and Definitions

In this section, the 2-hop cover algorithm as a basis of our HID index will be described in detail. In addition, the basic definitions that we will use throughout this chapter will be introduced also.

Definition 5-3: (Transitive Closure TC)

The transitive closure of a graph G = (V, E) is a graph G’= (V, E’) such that E’ contains an edge (*u, v*) if and only if G contains a non-null path <*u* … *v*>. The size of the TC is denoted by |E’|.

Finding the TC of a directed graph is an important subproblem in many computing tasks. For example, it is essential in the reachability analysis of transition networks that represent distributed and parallel systems. Recently, efficient transitive closure computation has been recognized as a significant subproblem in evaluating database queries. This is because almost all practical recursive queries are transitive.

Figure 5.5 is an example, which demonstrates how to construct the TC from a given graph.

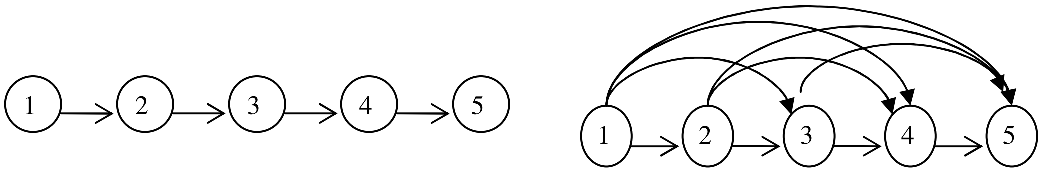


Figure 5.5: (a) An example of graph G = (V, E) (b) TC of graph G of Figure 5.5 (a)

5.2.1 Two-hop Covers

The concept of a 2-hop cover was first proposed in [CHKZ02]. It calculates and expresses the Transitive Closure (TC) (see Definition 5.3) more efficiently, thus being an order of magnitude more space-efficient and less time-consuming in calculating the index. Instead of storing the two sets of all ancestors and all descendants of a node directly, they store only a far smaller subset (this being L_{IN} and L_{OUT}). Thus, for each node *u* two label sets $L_{IN}(u)$ and $L_{OUT}(u)$ are maintained. $L_{IN}(u)$ is a set containing an arbitrary number of ancestors. These nodes are connected to *u* by at least one path. In the same way, $L_{OUT}(u)$ defines a set of descendants of a node *u*. These nodes can be reached from *u* by at least one path. The calculation of these sets has to be done in a way, which guarantees that it can be derived from the subsets of both nodes if there is a path from node *u* to *v*. The idea is based on the concept of a "center" node. The center node is a member of the descendants set of *u* as well as of the ancestors set of *v* . Thus, the center node is a node on the path from *u* to *v*. In order to calculate the descendant property between *u* and *v*, the descendant's subset of *u* is "intersected" with the ancestor's subset of *v*, if and only if there is an overlap between both sets (in the ideal case comprising one node only) a path between both nodes *u* and *v* exists $(L_{OUT}(u) \cap L_{IN}(v) \neq \emptyset)$.

The name 2-hop cover stems from the fact that this solution requires two stages to complete. The first step is to get from the node *u* to the center node *m* and then, in a second step, from *m* to the final node *v*, which in fact, is a 2-hop proceeding. The challenge now is to keep the two subsets of descendants and ancestors as small as possible. Unfortunately, their calculation is an *NP-hard problem* [CHKZ02].

Definition 5-4: (Two-hop Reachability Labeling)

A 2-hop reachability labeling for a directed graph G = (V, E), assigns to each node $u \in$ V a label $L(u)=(L_{IN}(u), L_{OUT}(u))$, where $L_{IN}(u), L_{OUT}(u) \subseteq V$ and there is a path between every $x \in L_{IN}(u)$ to *u* and from *u* to every $x \in L_{OUT}(u)$. Furthermore, for any two nodes $u, v \in V$, the following is true: There is a path between *u* and *v* if and only if $L_{OUT}(u) \cap L_{IN}(v) \neq \emptyset$. The size of this labeling is defined to be $\sum_{v \in V} |L_{IN}(v)| + |L_{OUT}(v)|$ [CHKZ02].

Definition 5-5: (2-hop Cover)

Let G = (V, E) be a directed graph with nodes V and edges E. A *2-hop cover* is a *2-hop reachability labeling* (Definition 5-4) that covers all paths (i.e. all connections) of G. Therefore, there is a path between a node *u* and a node *v* if and only if: $L_{OUT}(u) \cap L_{IN}(v) \neq \emptyset$ [CHKZ02].

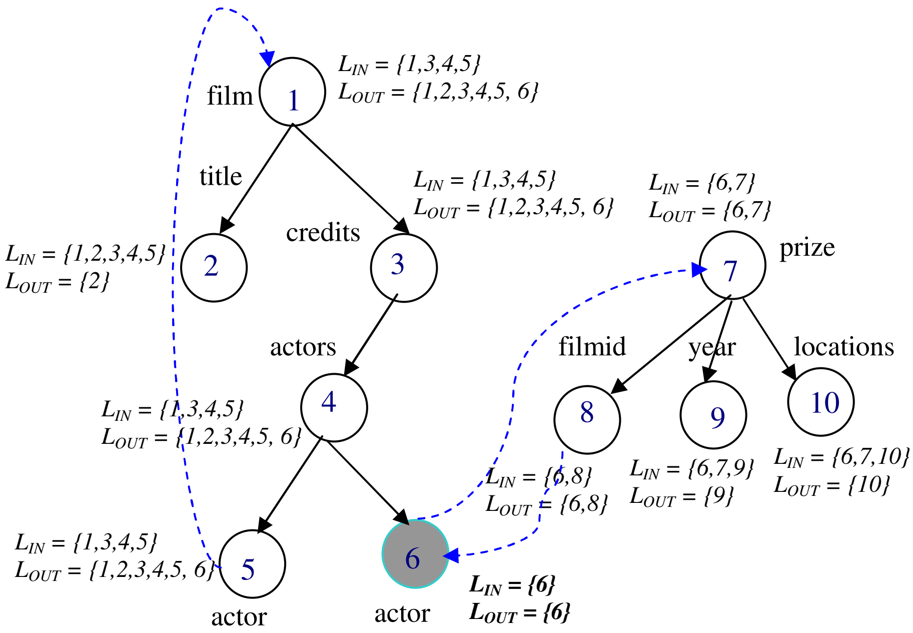


Figure 5.6: XML graph with 2-hop reachability labels for each node

For example, Figure 5.6 shows the information needed for the 2-hop cover: each node has two sets, a set of ancestors L_{IN} and a set of descendants L_{OUT}. The node *actor* (having the object identifier **6**) is considered the center node of the graph. There is a reachability between the two nodes *x= (3, Credits)* with $L_{OUT}(x) = \{3, 4, 5, \mathbf{6}\}$ and *y= (10, location)* with $L_{IN}(y) = \{\mathbf{6}, 7, 9\}$ because the intersection of $L_{OUT}(x)$ and $L_{IN}(y)$ is not empty ($L_{OUT}(x) \cap L_{IN}(y) = \{6\}$). However, there is no reachability between the two nodes y1= *(2, title)* and *y2= (10, location)* because the intersection between $L_{OUT}(y1)$ and $L_{IN}(y2)$ is empty ($L_{OUT}(y1) \cap L_{IN}(y2) = \varnothing$).

Note that Cohen et al. [CHKZ02] studied the concepts of 2-hop covers from a theoretical point of view; they have neglected several implementation and scalability issues and do not consider XML-specific issues either. We adapt their definitions to meet our purposes with XML applications.

5.2.2 Evaluation of 2-hop Covers

When we want to represent the transitive closure of a graph in a reasonable way, we are interested in a 2-hop cover with minimum size. Cohen el al. [CHKZ02] are casting the problem of finding a minimum 2-hop cover as a minimum set cover problem. In this case, they are facing an NP-hard problem [CHKZ02]. Therefore, they introduce a polynomial-time algorithm that evaluates a 2-hop cover for a given XML graph G = (V, E) whose size is at most larger than the optimal cover size by at most a logarithmic factor (*O log (|V|)*). In the following we explain how this algorithm works.

Given a directed graph G = (V, E) and the transitive closure G' = (V, E') as an input parameter where E' contains all combinations of nodes between which a path exists. For a node $x \in$ V, $C_{IN}(x) = \{v \in V \mid (v, x) \in E'\}$ is the set of all nodes for which there is a path from *v* to *x* in G (i.e., ancestors of the node *x*). In the same way $C_{OUT}(x) = \{v \in V \mid (x, v) \in E'\}$ defines the set of all descendants of node *x*, such that there is a path from *x* to *v* in G.

For each node *x* and two sets $C_{IN}(x)$, $C_{OUT}(x) \in$ V, we have $S(C_{IN}(x), x, C_{OUT}(x)) = \{(u,v) \in E' \mid u \in C_{IN}(x) \wedge v \in C_{OUT}(x)\} = \{(u, v) \in E' \mid (u, x) \in E' \wedge (x, v) \in E'\}$ denotes the set of all paths in G that contain *x*. The goal is to find a collection of such sets that cover E' (note that the collection of sets is an exponential in size).

For a given 2-hop labeling that is still not yet a 2-hop cover, let E" be the part of E' (i.e., E"$\subseteq$ E'). Thus, the set $S(C_{IN}(x), x, C_{OUT}(x)) \cap E''$ contains all connections within G that contain *x* and are not yet covered. The relationship between the number of connections via *x* that are not yet covered and the total number of nodes that lie on such connections is described by this ratio:

$$r(x) = \frac{\left| S\,(C_{IN}(x), x, C_{OUT}(x)) \bigcap E'' \right|}{\left| C_{IN}(x) \mid + \mid C_{OUT}(x) \right|}$$

The algorithm that used to compute the 2-hop cover starts with E"=E' and the 2-hop labels for each node in the underlying graph G are empty. In each step of the algorithm, the set E" contains all connections that are not yet covered. In a greedy manner the algorithm frantically selects the "best" $x \in V$; the "best" x is the highest value for $r(x)$. In a directed case we are looking for a set $S(C_{IN}(x), x, C_{OUT}(x))$ for which the ratio $r(x)$ is maximized and covers as many connections as possible using a small number of nodes. In this case, the node x is called the "center node" for the set $S\ (C_{IN}(x), x, C_{OUT}(x))$.

After the set $S(C_{IN}(x), x, C_{OUT}(x))$ is selected with its "center node" x, its nodes are used to update the 2-hop labels according to the following rules [CHKZ02]:

For a center node x

$$L_{IN}(x) = L_{OUT}(x) = \{x\}$$

For every node $u \in C_{IN}(x) \wedge u \notin C_{OUT}(x)$

$$L_{IN}(u) = C_{IN}(u) \cup \{u\}$$

$$L_{OUT}(u) = C_{OUT}(u) \setminus C_{OUT}(x) \cup \{x, u\}$$

For every node $u \in C_{OUT}(x) \wedge u \notin C_{IN}(x)$

$$L_{IN}(u) = C_{IN}(u) \setminus C_{IN}(x) \cup \{x, u\}$$

$$L_{OUT}(u) = C_{OUT}(u) \cup \{u\}$$

For every node $u \in C_{IN}(x) \wedge u \in C_{OUT}(x)$

$$L_{IN}(u) = L_{OUT}(u) = \{x, u\}$$

Then, the set $S(C_{IN}(x), x, C_{OUT}(x))$ is removed from E''. The algorithm terminates when E'' is empty. This happens when all connections in E' are covered. At this case, the 2-hop labeling is a 2-hop cover (i.e. figure 5.6).

To compute the 2-hop covers for a given set E' the above algorithm requires exponential time [CHKZ02]. This is because in each computation step for the algorithm, there are an exponential number of subsets $C_{IN}(x), C_{OUT}(x) \subseteq V$. This means that the above algorithm needs an exponential time for evaluating a 2-hop cover for a given set E' [CHKZ02].

The problem of finding the sets $C_{IN}(x)$ and $C_{OUT}(x)$ for a node $x \in V$ that maximize the ratio $r(x)$ is exactly the same problem of finding the *densest subgraph* of the *center graph* of a node x. An auxiliary bipartite graph $G_x = (V_x, E_x)$ is constructed, which we call the *center graph* of the center node x in the following way. The nodes sets

V_x contains two nodes V_{IN} and V_{OUT} for every node $x \in V$ of the original XML graph G. We have the undirected edges $(u_{out}, v_{in}) \in E_x$ if and only if $(u, v) \in E'$ is still not covered and $u \in C_{IN}(x)$ and $v \in C_{OUT}(x)$. Many of the nodes in the *center graph* may be isolated and can therefore be removed from the graph. Figure 5.7 (right) shows the *center graph* of node 5 (as a center node) in Figure 5.7 (left).

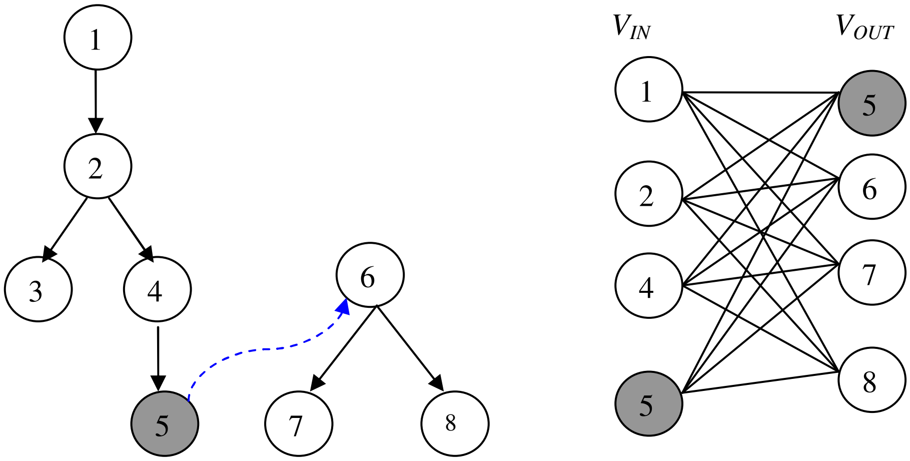


Figure 5.7: Center-graph of node 5

Definition 5-6: (Center Graph)

The construction of the *center graph* is defined as follows: Let G = (V, E) be given. Let $E'' \subseteq E'$ be the set of connections that are not yet covered in G. For a node $x \in V$, the *center graph* $G_x = (V_x, E_x)$ of *x (center node)* is an undirected bipartite graph with two node sets $V_{IN}(x)$ and $V_{OUT}(x)$. $V_{IN}(x)=\{u|u \in V: \exists v \in V: (u, v) \in E'' \wedge u \in C_{IN}(x) \wedge v \in C_{OUT}(x)\}$ contains all the nodes from which a path to *x* exists. This not only includes incoming and outgoing edges but also the ancestors of *x* in G. $V_{OUT}(x) = \{w|w \in V: \exists y \in V: (y, w) \in E'' \wedge y \in C_{IN}(x) \wedge w \in C_{OUT}(x)\}$ contains all the descendants of *x* in G. Moreover, the current node itself is added to the two sets.

Definition 5-7: (Density of the Center Graph)

The *density* of the *center graph* is defined as the number of edges divided by the number of nodes. In Figure 5.7 the density of the *center-graph* is 16 (edges) / 8 (nodes) =2.

Definition 5-8: (Construction of the Densest Sub-graph)

To compute the *densest sub-graph* from the *center-graph*, [CHKZ02] introduced the *2-approximation algorithm* that needs linear time. This algorithm iteratively removes a node of minimum degree (that has minimum edges going out) from the graph. This generates a sequence of *n* subgraphs. The algorithm computes the *density* of each resulting sub-graph using Definition 5-7. The result of the algorithm is a set of sub-graphs together with their *densities*. Then, the algorithm returns the sub-graph with the highest density.

Given is an undirected *center graph* G_x= (V_x, E_x). The task is to find a subset S $\subseteq V_x$ for which the average degree in the sub-graph induced by S is maximized, i.e., a set that maximizes the ratio |E(S)| / |S|, where E(S) is the set of edges connecting two vertices of S.

Computing the 2-hop cover has time complexity $O(|V|^3)$, this is because Cohen at el. [CHKZ02] based on the Floyd-Warshall algorithm [CLRS01] for computing the transitive closure of the given XML graph G, which needs time $O(|V|^3)$ and also for computing the 2-hop cover from the given transitive closure the algorithm needs times $O(|V|^3)$. (the first step of the algorithm computes the densest subgraphs for V-nodes, the second step computes the densest subgraphs for V-nodes, etc. yielding $O(|V|^2)$ computation each with worst-case complexity $O(|V|\ )$) [CHKZ02] [STW04]. Moreover, the 2-hop cover algorithm requires at most space $O(|V|.|E|^{\frac{1}{2}})$, yielding $O(|V|^2)$ in the worst case.

In the following we introduce a complete example to illustrate how the 2-hop cover algorithm works:

5.2.2.1 Example Scenario

Step (1): Given a directed graph G= (V, E) (in Figure 5.3), where V represents the set of nodes and E represents the set of edges. Additionally, we consider the links from 6 to 7, from 8 to 6, and from 5 to 1 as edges in G. From now on, we don't make a difference between links and edges.

Step (2): Built the transitive closure G'=(V, E') of graph G.

	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1
2	0	0	0	0	0	0	0	0	0	0
3	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1	1	1
6	0	0	0	0	0	1	1	1	1	1
7	0	0	0	0	0	1	1	1	1	1
8	0	0	0	0	0	1	1	1	1	1
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0

Figure 5.8: Transitive closure of Figure 5.3

The columns in the above matrix represent the set C_{IN} of each node in G, while the rows represent the set C_{OUT}.

Step (3): For each node x in G we build the center graph G_x= (V_x, E_x). It is a bipartite undirected graph with two node sets, $V_{IN}(x)$ and $V_{OUT}(x)$, where $V_{IN}(x)$ contains all the nodes from which a path to x exists in G (i.e., the ancestors of x in G), and $V_{OUT}(x)$ contains the descendants of x in G.

Let E" be the part of E' that is still uncovered. Initially, E"= E' (i.e. all connections in G), and the 2-hop labels for each node in G are empty.

The algorithm has E" as input, it starts at node 1 (as the first node in the node list), compute the two sets $C_{IN}(1)$ and the $C_{OUT}(1)$ from E".

<u>Start at node "1"</u>

✓ *Construction of the center graph G_1=(V_1 , E_1)*

The center graph G_1=(V_1 , E_1) of node 1 is constructed as follows:

C_{IN}(1) = {1,3,4,5} and C_{OUT}(1) = {1,2,…,10}. The node itself is also added to each of the two sets, so

G_1 = (V_1 = (1_{in}, 3_{in}, 4_{in}, 5_{in}, 1_{out}, 2_{out}, 3_{out}.4_{out}, 5_{out}, 6_{out}, 7_{out}, 8_{out}, 9_{out}, 10_{out}), E_1={}). In the following we explain how the edges E_1 shall be constructed using the following code:

```
For each $u \in C_{IN}(1)$ do
    For each node $v \in C_{OUT}(1)$
        If ($v \in C_{OUT}(u)$) add an edge between u and v; otherwise not.
```

u = 1

v =1

C_{OUT}(u) = C_{OUT}(1) = {1,2,…,10} then v = 1 ∈ C_{OUT}(1), this means that there is an edge between *u* and *v,* then G_1= (V_1, {(1,1)}).

u = 1

v =2

C_{OUT}(u) = C_{OUT}(1) = {1,2,…,10} then v = 2 ∈ C_{OUT}(1), this means that there is an edge between *u* and *v* , then, G_1= (V_1, {(1,1), (1,2)}).

u = 1

v =3

C_{OUT}(u) = C_{OUT}(1) = {1,2,…,10} then v = 3 ∈ C_{OUT}(1), this means that there is an edge between *u* and v, then, G_1= (V_1, {(1,1), (1,2), (1, 3)}).

u = 1

v =4

C_{OUT}(u) = C_{OUT}(1) = {1,2,…,10} then v = 4 ∈ C_{OUT}(1), this means that there is an edge between *u* and *v*, then, G_1= (V_1, {(1,1), (1,2), (1, 3), (1, 4)}).

u = 1

v =5

C_{OUT}(u) = C_{OUT}(1) = {1,2,…,10} then v = 5 ∈ C_{OUT}(1), this means that there is an edge between *u* and v, then, G_1= (V_1, {(1,1), (1,2), (1, 3), (1, 4), (1, 5)}).

u = 1

v =6

C_{OUT}(u) = C_{OUT}(1) = {1,2,…,10} then v = 6 ∈ C_{OUT}(1), this means that there is an edge between *u* and v, then, G_1= (V_1, {(1,1), (1,2), (1, 3), (1, 4), (1, 5), (1, 6)}).

$u = 1$

$v = 7$

C_{OUT}(u) = C_{OUT}(1) = {1,2,…,10} then v = 7 ∈ C_{OUT}(1), this means that there is an edge between *u* and v, then, G_1= (V_1, {(1,1), (1,2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7)}).

$u = 1$

$v = 8$

C_{OUT}(u) = C_{OUT}(1) = {1,2,…,10} then v = 8 ∈ C_{OUT}(1), this means that there is an edge between *u* and *v*, then, G_1= (V_1, {(1,1), (1,2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1,8)}).

$u = 1$

$v = 9$

C_{OUT}(u) = C_{OUT}(1) = {1,2,…,10} then v = 9 ∈ C_{OUT}(1), this means that there is an edge between *u* and v, then, G_1= (V_1, {(1,1), (1,2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1,8), (1,9)}).

$u = 1$

$v = 10$

C_{OUT}(u) = C_{OUT}(1) = {1,2,…,10} then v =10 ∈ C_{OUT}(1), this means that there is an edge between *u* and v, then, G_1= (V_1, {(1,1), (1,2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1,8), (1,10)}).

By the same way, these steps are repeated for u = 3 , u =4, and u = 5. The result in Figure 5.9 is the center graph G_1 = (V_1, E_1) of node 1.

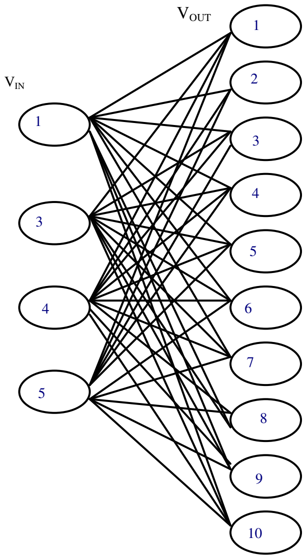


Figure 5.9: Center-graph G_1 = (V_1, E_1) of node “1”

Step (4): For each center graph $G_x = (V_x, E_x)$, we compute the density of its densest subgraph G'_x and keep the maximum density along with its corresponding densest subgraph.

- ✓ *Construction of the densest sub-graphs for the center graph $G_1=(V_1 , E_1)$*

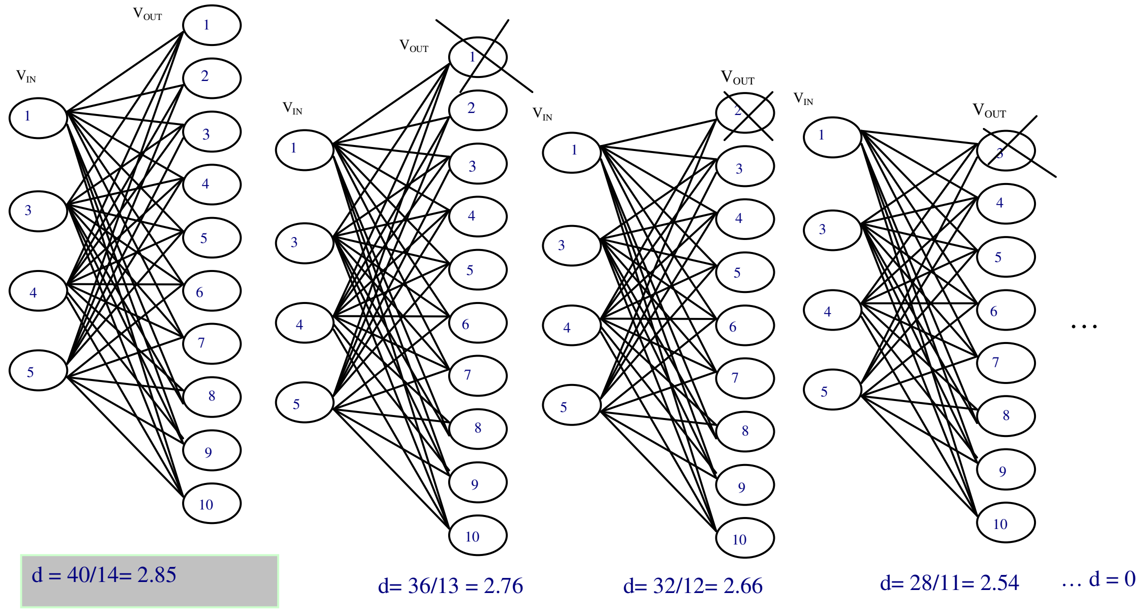


Figure 5.10:(a) center-graph (b) removing node 1 (c) removing node 2 (d) removing node 3

Compute the densest sub-graphs from the center-graph $G_1 = (V_1, E_1)$ as follows: Remove a node of minimum degree (i.e. that has minimum edges going out) from $G_1 = (V_1, E_1)$ and compute the density of each resulting densest sub-graph by using this relation d = |edges|/|nodes|.

Figure 5.10 shows the densest sub-graphs for the center-graph G_1 of node 1. Figure 5.10(a) shows the center graph itself with its density. We notice from G_1 that all nodes in V_{OUT} have the same number of out-edges, therefore choosing any node from V_{OUT} has the same property. Figure 5.10(b) illustrates the densest subgraph after node 1 is removed. Figures 5.10(c) and 5.10(d) show the densest sub-graphs with their densities after nodes 2 and 3 are removed. The algorithm works iteratively until the density of the resulting subgraph is zero. The algorithm keeps the maximum density along with its corresponding densest subgraph (i.e. the maximum density d = 2.85).

Node "2"

- ✓ *Construction of the center graph $G_2=(V_2, E_2)$*

```
For each $u \in C_{IN}(1)$ do
    For each node $v \in C_{OUT}(1)$
        If ($v \in C_{OUT}(u)$) add an edge between u and v; otherwise not.
```

$C_{IN}(2) = \{1,3,4,5\}$ and $C_{OUT}(2) = \{\}$. The node itself is also added to the two sets
$G_2 = (V_2 = (1_{in}, 2_{in}, 3_{in}, 4_{in}, 5_{in}, 2_{out}),\ E_2=\{\})$.

u=1
v= 2
$C_{OUT}(u) = C_{OUT}(1) = \{1,2,\ldots,10\}$ then $v = 2 \in C_{OUT}(1)$, this means that there is an edge between *u* and *v* , then, $G_2= (V_2, \{(1,2)\})$.
u=2
v= 2
$C_{OUT}(u) = C_{OUT}(2) = \{\}$ then $v = 2 \notin C_{OUT}(2)$, this means that there is no edge between *u* and *v*, then, $G_2= (V_2, \{(1,2)\})$.
u=3
v= 2
$C_{OUT}(u) = C_{OUT}(3) = \{1,3,\ldots,10\}$ then $v = 2 \in C_{OUT}(3)$, this means that there is an edge between *u* and v, then, $G_2= (V_2, \{(1,2), (3,2)\})$.
u=4
v= 2
$C_{OUT}(u) = C_{OUT}(4) = \{1,3,\ldots,10\}$ then $v = 2 \in C_{OUT}(4)$, this means that there is an edge between *u* and *v*, then, $G_2= (V_2, \{(1,2), (3,2), (4, 2)\})$.
u=5
v= 2
$C_{OUT}(u) = C_{OUT}(5) = \{1,3,\ldots,10\}$ then $v = 2 \in C_{OUT}(5)$, this means that there is an edge between *u* and *v* . $G_2= (V_2, \{(1,2), (3,2), (4, 2), (5,2)\})$. The center graph of node 2 shows as follows

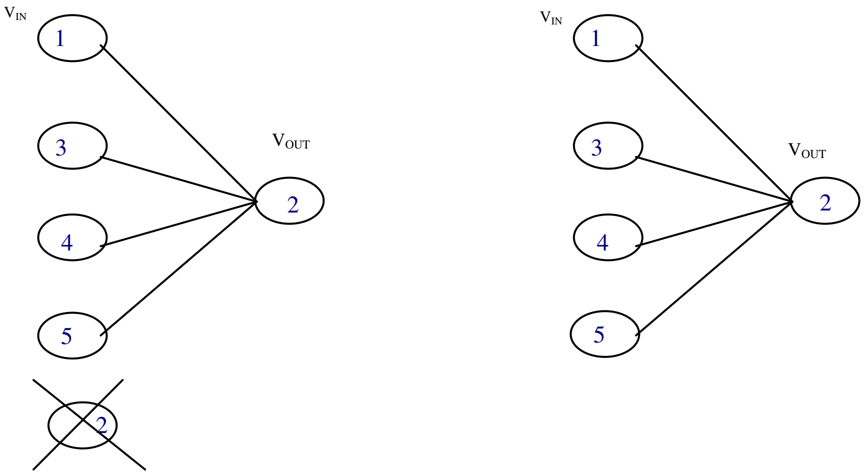


Figure 5.11:(a) center graph of node 2 (b) center graph after deleting the isolated node

As explained in section 5.2.2, the isolated nodes can be removed from the center graph. Therefore, node 2 in Figure 5.11(a) is removed. The result is a center graph without isolated nodes (i.e. Figure 5.11(b)).

- ✓ *Construction of the densest sub-graphs for the center graph $G_2=(V_2 , E_2)$*

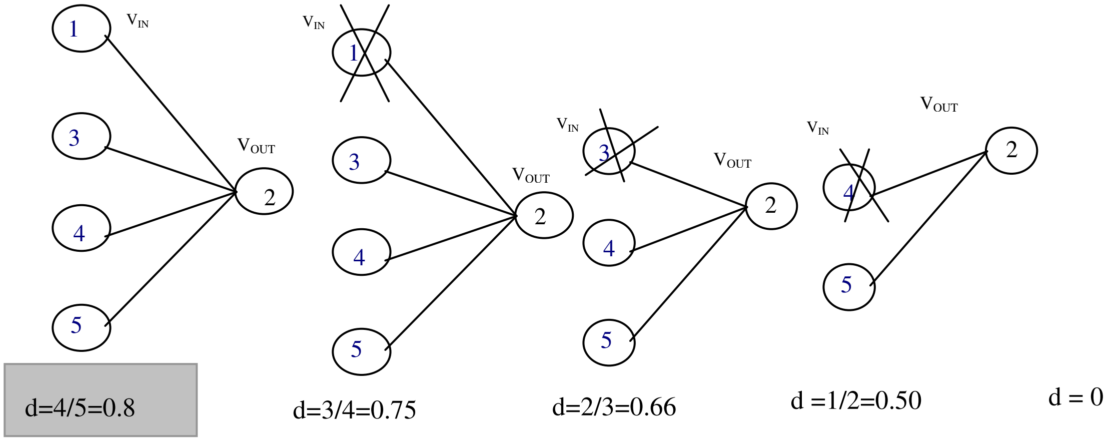


Figure 5.12:(a) Center-graph (b) removing node 1 (c) removing node 3 (d) removing node 4

Figure 5.12 shows the densest sub-graphs for the center graph $G_2=(V_2 , E_2)$ of node 2 with their corresponding densities. Figure 5.12 (a) illustrates the density of the center graph itself. Figure 5.12(b) illustrates the densest subgraph after node 1 is removed. Figures 5.12 (c) and 5.12 (d) show the densest sub-graphs with their densities after nodes 3 and 4 are removed. The algorithm works iteratively until the density of the resulting subgraph is zero. The result of the algorithm is the maximum density along with its corresponding densest subgraph (i.e. the maximum density d = 0.8).

We repeat this process with every node in the node list (i.e. in this example with nodes (3,4,5,6,7,8,9,10)). For each node the algorithm returns the maximum density along with its corresponding densest sub-graph. As a lot example node = 6 is discussed.

Node “6”

✓ *Construction of the center graph $G_6 =(V_6 , E_6)$*

$C_{IN}(6) = \{1, 3, 4, 5, 6, 7,8\}$ and $C_{OUT}(1) = \{6, 7, 8, 9, 10\}$
$G_6 = (V_6 = (1_{in}, 3_{in}, 4_{in}, 5_{in}, 6_{in}, 7_{in}, 8_{in}, 6_{out}, 7_{out}, 8_{out}, 9_{out},10_{out}),\ E_6=\{\})$.

```
For each u∈ C_IN (6) do
    For each node v ∈ C_OUT (6)
        If (v∈ C_OUT (u)) add edge between u and v; otherwise not.
```

$u = 1$
$v = 6$
$C_{OUT}(u) = C_{OUT}(1) = \{1,2,\ldots,10\}$ then $v = 6 \in C_{OUT}(1)$, this means that there is an edge between u and v , then, $G_6 = (V_6 ,\ E_6=\{(1,6)\})$.
$u = 1$
$v = 7$
$C_{OUT}(u) = C_{OUT}(1) = \{1,2,\ldots,10\}$ then $v = 7 \in C_{OUT}(1)$, this means that there is an edge between u and v, then, $G_6 = (V_6 ,\ E_6=\{(1,6), (1, 7)\})$.
$u = 1$
$v = 8$
$C_{OUT}(u) = C_{OUT}(1) = \{1,2,\ldots,10\}$ then $v = 8 \in C_{OUT}(1)$, this means that there is an edge between u and v, then, $G_6 = (V_6,\ E_6=\{(1,6), (1,7), (1,8)\})$.
$u = 1$
$v = 9$
$C_{OUT}(u) = C_{OUT}(1) = \{1,2,\ldots,10\}$ then $v = 9 \in C_{OUT}(1)$, this means that there is an edge between u and v , then, $G_6 = (V_6 ,\ E_6=\{(1,6), (1,7), (1,8), (1,9)\})$.

$u = 1$

$v = 10$

$C_{OUT}(u) = C_{OUT}(1) = \{1,2,\ldots,10\}$ then $v = 10 \in C_{OUT}(1)$, this means that there is an edge between u and v, then, $G_6 = (V_6, E_6=\{(1,6), (1,7), (1,8), (1,9),(1,10)\})$.

$\underline{u = 3}$

$v = 6$

$C_{OUT}(u) = C_{OUT}(3) = \{1,2,\ldots,10\}$ then $v = 6 \in C_{OUT}(3)$, this means that there is an edge between u and v , then, $G_6 = (V_6, E_6=\{(1,6), (1,7), (1,8), (1,9),(1,10), (3,6)\})$. And so on as above.

These steps are repeated for u = 4 , u =5, u = 6, u = 7, and u = 8. The result is the following center-graph of node 6.

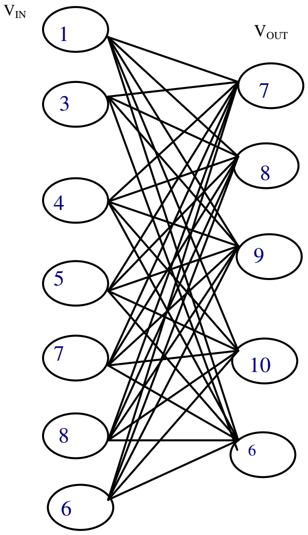


Figure 5.13: Center-graph of node 6

- ✓ *Construction of the densest sub-graphs $G_6=(V_6 , E_6)$*

In the same way the densest sub-graphs of nodes 1 and 2 are constructed, the densest sub-graphs of node 6 can be constructed.

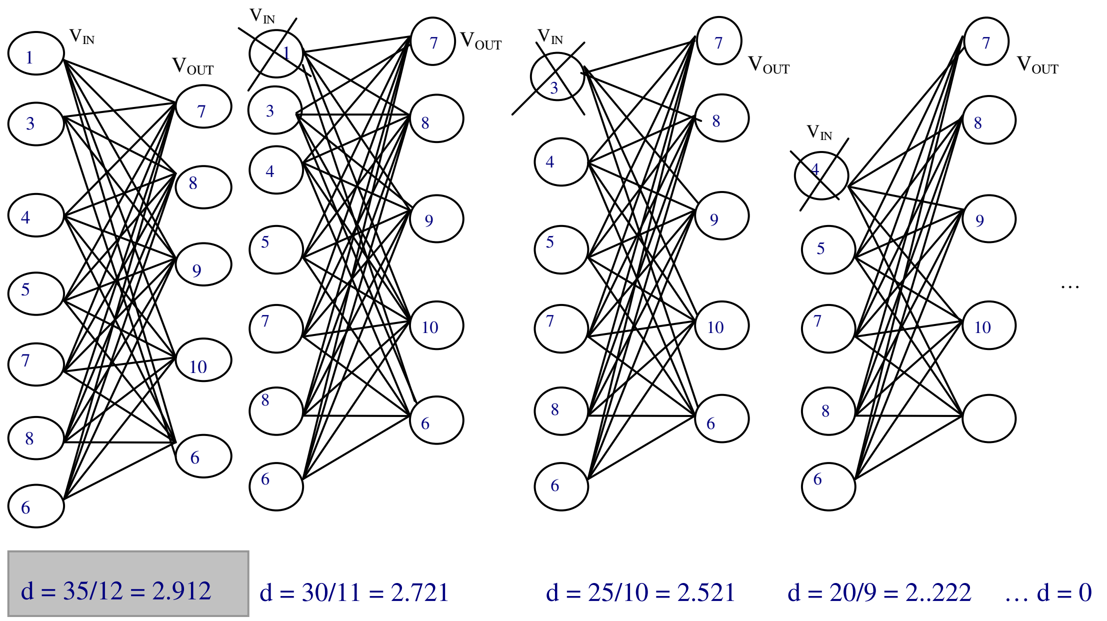


Figure 5.14: (a) center-graph of node 6 (b) removing node 1 (c) removing node 3 (d) removing node 4

Figure 5.14 shows the densest sub-graphs for the given center-graph G_6. Node 6 has the maximum density = 2.912. We keep the densest subgraph of node 6 along with its density.

Step (5): We choose the node $x \in V$, where its densest subgraph has the maximum density among all center graphs.

Table 5.1 shows the density calculated for each node in the graph G.

Node	1	2	3	4	5	**6**	7	8	9	10
Density	2.851	0.8	2.851	2.851	2.851	**2.912**	2.912	2.912	0.875	0.875

Table 5.1: Densities table

Table 5.1 shows that, nodes 6, 7, and 8 have the same maximum density. The algorithm randomly chooses node 6 as the center node.

Step (6): We remove all the connections from the transitive closure E" that are covered by the center node x (in this example $x = 6$). We notice that the center graph of node 6 covers all connections of E". Therefore, the algorithm will stop here. Please note that the algorithm stops at the first iteration, because all connections are covered. Otherwise the algorithm works iteratively from step 3 until all connections are covered.

Step (7) : Updates the 2-hop labels as follows:

After the center graph of node 6 (i.e. the set $S_6 = (C_{IN}(6), 6, C_{OUT}(6)))$, is selected, its nodes are used to update the 2-hop labeling as follows: for each node *u* we add a label $L(u) = (L_{IN}(u), L_{OUT}(u))$ according to the following rules (see Figure 5.15).

For the center node *6*

$$L_{IN}(6) = L_{OUT}(6) = \{6\}$$

For every node $u \in C_{IN}(6) \wedge u \notin C_{OUT}(6)$

$$L_{IN}(u) = C_{IN}(u) \cup \{u\}$$

$$L_{OUT}(u) = C_{OUT}(u) \setminus C_{OUT}(6) \cup \{6, u\}$$

For every node $u \in C_{OUT}(6) \wedge u \notin C_{IN}(6)$

$$L_{IN}(u) = C_{IN}(u) \setminus C_{IN}(6) \cup \{6, u\}$$

$$L_{OUT}(u) = C_{OUT}(u) \cup \{u\}$$

For every node $u \in C_{IN}(6) \wedge u \in C_{OUT}(6)$

$$L_{IN}(u) = L_{OUT}(u) = \{6, u\}$$

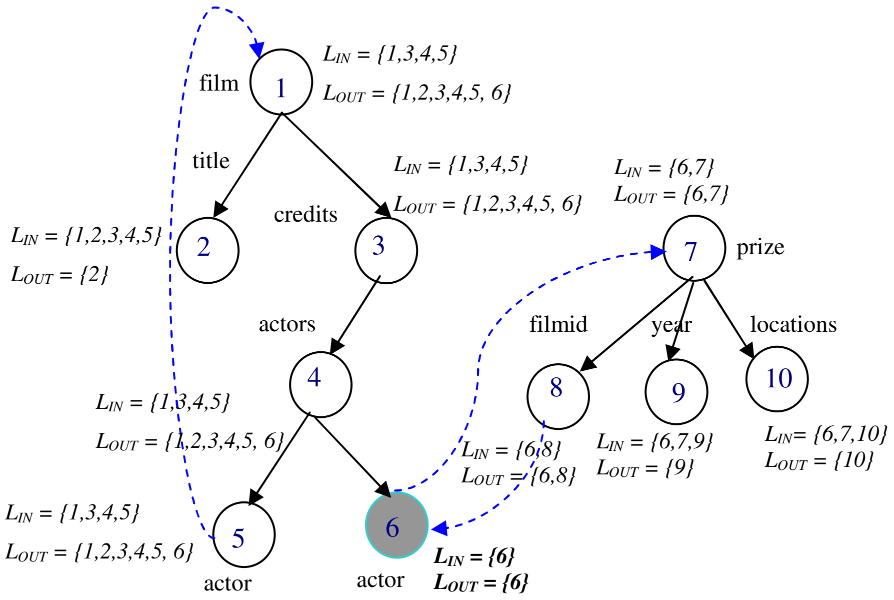


Figure 5.15: Graph G with 2-hop labels for each node

5.2.3 Two-hop Cover Problems

As mentioned previously, Cohen et al. have studied the 2-hop cover from a theoretical approach and applied the algorithm to directed and undirected graphs. However, they do not consider the XML-specific issues that we described in Section 5.1. Originally, we applied the 2-hop cover for XML graphs as it is (without any modification [SU03]). However, this leads to problems when the underlying XML documents is too large. The reason for this is that the input parameter for the 2-hop cover is the pre-computed transitive closure of the input graph (which can be a very space-consuming problem). The HOPI [STW04] index solves this problem by proposing a *divide-and-conquer* algorithm that is based on partitioning the original graph into several graphs. It computes the transitive closure and the 2-hop cover for each part. Then, it merges the 2-hop cover of all partitions. By this technique, they overcome the space-consuming problem. However, the HOPI index still has the time problem, as it needs more time to join the covers. Moreover, it cannot efficiently deal with graphs that have *cycles*.

Taking the later into account we proposed a HID index [SU04] [SU05] that can efficiently deal with graphs that have cycles.

5.3 HID Path Index

The HID path index has the following particular characteristics to solve the problems previously discussed, and to improve the performance when processing queries.

- *Efficient processing of complex queries* (i.e**.,** "//" condition in XPath): As discussed in chapter 4, most traditional path indexes evaluate all label paths starting from the root element first. They are suitable for simple path expressions that start from the root of the XML document. However, these indexes are not efficient for evaluating complex path expressions like, *descendant-or-self axis* and *ancestor-descendant axis* with *wildcards*.
- *Reachability tests*, it can efficiently test the reachability between two given nodes in large XML graph with cycles over long paths.
- *Incremental Update***:** This HID path index has the following advantageous

property: One of the important factors in building a path index is that it must be consistent with the underlying data (which may mean that it has to rebuild it after each update).

5.3.1 HID Index Framework

HID index relies on three factors: Firstly, the HID index; it is constructed by shrinking every strongly connected component (see Definition 5-9) of the input XML graph G = (V, E) to a single node. Therefore, the resulting graph is a DAG (Directed Acyclic Graph) with a minimal number of nodes and edges. These strongly connected components are the most likely reason for redundant computations in the transitive closure algorithms, since it does not detect and eliminate them. In fact, if two nodes belong to the same strongly connected component they are automatically ancestors of each other. Our experimental results show that this "shrinking" technique minimizes the size of the labels (L_{IN} and L_{OUT}) compared with the 2-hop cover algorithm.

Secondly, to avoid the time-consuming problem, we use the Nuutila algorithm [NS93] (see Section 5.3.2) rather than the Floyed-Warshall algorithm [CLRS01] (which is used in the 2-hop cover and in the HOPI index) for computing the transitive closure. Nuutila suggests an algorithm for computing the transitive closure that first detects the strongly connected components of the directed graph. This algorithm has a quadratic time complexity in the number of nodes compared with the Floyed-Warshall algorithm that has a cubic time complexity.

Note from Section 5.2.2 that the 2-hop cover algorithm computes the densest subgraph for all center graphs in each step of the algorithm. This is very time-consuming. As a result, in Sections 5.3.3 and 5.3.4, one can draw the conclusion that the achieved results can reduce the densest sub-graph computation dramatically during the creation of the HID index.

Definition 5-9: (Strongly Connected Components (SCCs))

The problem of determining the strongly connected components of an input graph is a "classical" one. A common solution is based on two *depth first algorithm* [CLRS01].

The HID index uses this technique to reduce the size of the XML graph. To detect the SCCs two traverses of the underlying graph are to be performed. The first is a *depth-first search*, which traverses all the edges and constructs a *depth first* spanning forest. Once the so-called root (topmost node) of a strongly connected component is found, all its descendants that are not elements of previously found components become marked elements of these components. This second traversal is implemented by a "stack" that holds each node in a depth first order (computed in the first step). Before the root of a component is pushed from the stack, all nodes down the root are removed from the stack. These nodes form the component in question.

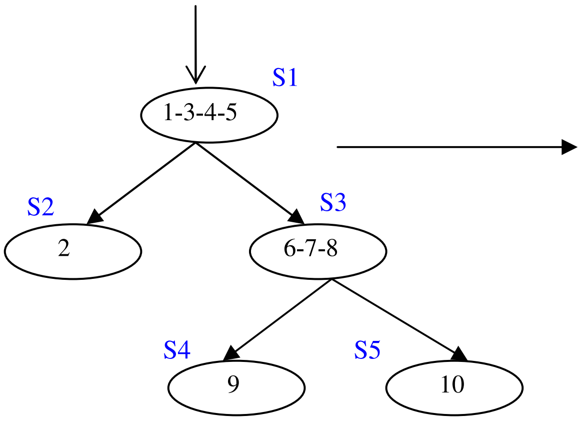
```
Procedure visit (v)
begin
    Root (v) = v; Comp (v) = Null
    push (v, stack)
    for each (node u such that $(u, v) \in E$ ) do
        if  u is not visited then visit (u)
        if Comp (u) = Null then Root (v) = MIN (Root (v), Root (u))
      end
      if Root (v) = v then
          Create a new component SCC
        repeat
            u = POP (stack)
            Comp (u) = SCC
            Insert u into Component SCC
        until u = v
       end if
      end for
 begin // main program
   stack = $\varphi$
 for each node $v \in V$ do
       if  v is not already visited then visit (v)
 end procedure
```

Figure 5.16: Strongly Connected Component detection algorithm for graph G

Figure 5.16 describes the algorithm that is used to compute the strongly connected component of the input graph G = (V, E). It consists of a recursive procedure *visit ()* and a main program that apply this method in the depth first order to each node that has not already been visited. In the case of each strongly connected component SCC, the first node of the SCC is called the root of the SCC. The *MIN operation* at line 7 compares the nodes using the order in which *visit()* has entered them. The *Comp operation* at line 13 is used to distinguish between nodes belonging to the same component as well as nodes belonging to other components. The time complexity needed to evaluate the SCC is *O (V+E)* (where V represents the set of nodes in the underlying graph and E represents the edges): The size of the Strongly Connected Component is defined as the number of nodes it contains. Any node of the underlying graph that is not contained in cycles forms an SCC of size one. In the following, example 5.1 will be used to show that the strongly connected components technique, in fact, it reduces the redundant computation of the transitive closure. It will help to determine a reachability query in an efficient way.

Example 5.1

Consider the directed XML graph in Figure 5.6. It consists of ten nodes with labels {1, 2, 3,…, 10}. By shrinking every strongly connected component of this graph, the result is a DAG with a minimum number of nodes.



Node	set
1	S1
2	S2
3	S1
4	S1
5	S1
6	S3
7	S3
8	S3
9	S4
10	S5

Figure 5.17: (a) The result DAG of Figure 5.6 (b) Nodes table of (a)

Figure 5.17 (a) shows the resulting DAG of Figure 5.6. It consists of five SCCs: S1= {1, 3, 4, 5}, S2= {2}, S3= {6, 7, 8}, S4= {9}, and S5= {10}. S2, S4, and S5 are strongly connected components of size one. S1 is the root node of the DAG (because it contains the root node of the original graph). It is evident that by using this technique, the number of nodes is reduced by half (see Figure 5.17 (a)). Therefore, the redundant computation of the transitive closure is reduced. The experiments show that this technique minimizes the size of the label sets (L_{IN} and L_{OUT}) since there is a direct relationship between the size of the labels and the number of connections of the transitive closure.

Additionally, to evaluate reachability queries an SCC table is set up (see Figure 5.17) (b)), which stores the set that each node belongs to. For example, consider this path expression "*is a node film that has identifier "1" ancestor of the node actor that has identifier "5"" (i.g.,"1//5")?* First, we check the table to see whether these two nodes belong to the same set. The answer is yes, since these two nodes belong to S1. Therefore, this reachability query is efficiently determined in linear time. On the other hand, if the two nodes belong to separate strongly connected components, the HID index has to be used. In our experiment, we found out that about 30% of the reachability queries could be determined in a linear time without needing to use the HID index.

Example 5.1 shows the benefits of our first optimization technique that is based on the principle of strongly connected components. The aim was to reduce the number of nodes and the number of connections of the TC, to minimize the size of the labels (L_{IN} and L_{OUT}), and to test efficiently the reachability between two nodes in linear time. In the next section, the second optimization technique that is used to evaluate the transitive closure based on the idea of SCCs shall be explained in detail.

5.3.2 Efficient Computation of the Transitive Closure using SCCs

Since the transitive closure is needed as an input parameter for the 2-hop cover algorithm, it needs to be materialized and stored in main memory during the construction of the index. Thus, the question arises: how the transitive closure of a linked forest of XML documents can be computed efficiently, especially, if the underlying XML graph has many cycles (e.g., Figure 5.2). The HOPI connection index deals with this problem by proposing the divide-and-conquer algorithm. Since it partitions the original XML graph the transitive closure needs to be materialized for each partition separately. The divide and conquer partitioning of a graph is known to be NP-hard problem [STW04]. Therefore, it is difficult to find a good partition for a large XML graph in reasonable time. Moreover, two nodes that are connected in the original graph may be disconnected after the partitioning (due to the partition process). This affects the efficiency of the query evaluation.

The HID index constructs the transitive closure from a directed acyclic graph (DAG), which has fewer edges and vertices than the original graph. Working with a DAG instead of the original XML graph has the following benefits:

1. Materializing the transitive closure of the computed DAG (e.g., Figure 5.9 (a), having five nodes) instead of the original XML graph (e.g., Figure 5.6, having ten nodes) can avoid the problem of extensive space-consumption. Furthermore, this may make a main memory-based computation of the cover feasible.
2. The reachability queries can be efficiently evaluated in a linear time O(n) if the two nodes are located in the same SCC.

The authors in [CHKZ02] and [STW04] use the Floyed Warshall [CLRS01] algorithm to compute the transitive closure for the input XML graph. It needs $O(|V|^3)$ time. In our work the transitive closure algorithm introduced in [NS93] is utilized. This algorithm is considered the "best" algorithm for the computation of the transitive closure of directed graphs based on the detection of strongly connected components. The main strength of this algorithm is that it scans the input graph only once (i.e., directed graph may have pairs of nodes that are connected via multiple paths) without generating "partial successor sets" (containing all nodes reachable to the current node) for each node of the SCC. The algorithm therefore only needs $O(|V|^2)$ time compared with $O(|V|^3)$ time that is needed by the Floyed Warshall [CLRS01] algorithm. In addition to this, an important issue to be taken into consideration for an efficient transitive closure computation is the choice of an appropriate representation for the transitive closure. The representation should be compact to reduce memory space. Therefore, we can take advantage of operating on the DAG rather than on the underlying XML graph, since there are several issues with which it is not necessary to counter the computation of the transitive closure of the large XML graph. For example:

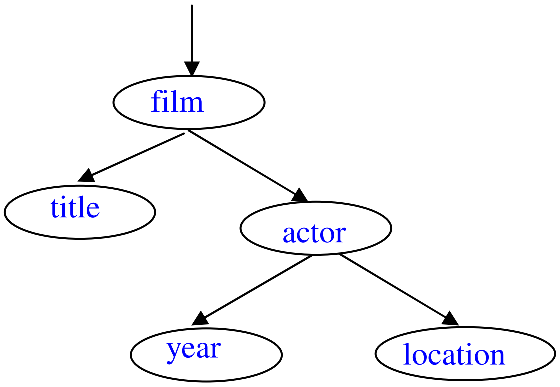


Figure 5.18: DAG with the root nodes of SCCs

- Edges inside the strongly connected component can be ignored entirely. This example deals solely with the root node of each SCC as a deputy node to construct the DAG. This implies that edges are not equally important in a transitive closure computation. For example, in Figure 5.17 (a) the strongly connected component S1 has the nodes (film, credits, actors, and actor) with four edges. The root of S1 ("film" topmost node of the component S1) is the primary goal here, and so, all nodes and edges that construct it are ignored. S3 follows in the same manner. Figure 5.18 shows the DAG with the root nodes of SCCs of Figure 5.6.

- Compute the successor sets (containing all nodes reachable from the current node), for the components instead of the nodes that construct the component (i.e., Succ[root[v]] instead of Succ[v] in Figure 5.19) [CLRS01]. This makes the successor sets smaller and minimize the number of connections that used as an input of the HID index.

- Figure 5.19 explains the algorithm for the transitive closure computation of directed graphs based on the strongly connected components in a pseudo code. The main procedure is SCC-TC( ). As we described previously, for each SCC, the first node of SCC that is entered is the root of SCC. Here, the main goal is to find the component roots. For this purpose, we define a variable root [*v*] for each node *v*. Initially, at line 2, node *v* is itself root [*v*]. The MIN operation at line 7 compares the nodes with respect to the order in which the procedure SCC-TC( ) has entered them, i.e., *MIN (x, y) = x* if the procedure SCC-TC( ) entered node *x* before it entered node *y*; otherwise *MIN (x, y) = y*. The forward edge *(x, y)* (line 8) means that, *y* is a descendant of *x* in the spanning forest induced by the depth-first traversal, buy *y* is not a direct child of *x* in the spanning forest. There must be a path leading from *x* to *y* that consists solely of edges of spanning tree. At line 9, it inserts a component SCC[*u*] adjacent from node *v* and the successor set of SCC[*u*] into the successor set of the current node *v*.

```
(1)  Procedure SCC-TC (v);
(2)  Begin
(3)     root [v] = v;  SCC[v] = 0
(4)     push (v, stack)
(5)     for each node u such that $(v, u) \in E$ do
(6)        if u is not already visited then SCC-TC(u)
(7)        if SCC[u] = NULL then root [v] = MIN (root [v], root [u])
(8)        else if (v, u) is not a forward edge and $SCC[u] \notin Succ[root[v]]$ then
(9 )          $Succ [root[v]] = Succ[root[v]] \cup \{SCC[u]\} \cup Succ [SCC[u]]$
(10)      end
(11)      if  SCC[v] = v then begin
(12)             create new component SCC
(13)             if   TOP (stack) = v then Succ [SCC] = Succ [v]
(14)             else $Succ[SCC] = Succ[v] \cup \{SCC\}$
(15)             repeat
(16)                u = POP(stack)
(17)                SCC[u] = SCC
(18)                insert u into Component  SCC
(19)              if $u \neq v$ and $Succ[u] \neq 0$ then $Succ [SCC] = Succ [SCC] \cup Succ[u]$
(20)             until u = v
(21)         end
(22)  end procedure
(23)  begin  /* Main program*/
(24)     stack = 0
(25)     for each node $v \in V$ do
(26)         if v is not already visited  then  SCC-TC (v)
(27)  end program
```

Figure 5.19**:** Transitive closure algorithm based on SCCs of graph G = (V, E)

When a component is fully detected, the non-empty successor sets of its nodes are combined. If the component is nontrivial, i.e. it contains more than one node the component is included into its own successor set (line 14). After the execution of the algorithm, Succ[x] can be found in Succ[SCC[x]], which means that the result is one successor set for each SCC. By this way, this procedure eliminate the redundancy caused by the strongly connected components more efficiently than the transitive closure algorithm used in [CHKZ02] and [STW04].

Moreover, each node is stored on a stack at the beginning of the procedure SCC-TC (*v*). When the component is fully detected the nodes belongs to it are on top of the stack. Then, procedure SCC-TC (*v*) removes them from the stack. In the following an example is given to explain the benefits of the above algorithm.

Example 5.2

The benefits of this optimization algorithm become more obvious if one compares the number of connections in the transitive closure of the DAG in Figure 5.17 (a) with the number of connections of the original graph in Figure 5.6.

	S1	S2	S3	S4	S5
S1	0	1	1	1	1
S2	0	0	0	0	0
S3	0	0	0	1	1
S4	0	0	0	0	0
S5	0	0	0	0	0

	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1
2	0	0	0	0	0	0	0	0	0	0
3	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1	1	1
6	0	0	0	0	0	1	1	1	1	1
7	0	0	0	0	0	1	1	1	1	1
8	0	0	0	0	0	1	1	1	1	1
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0

Figure 5.20:(a) Transitive closure of Figure 5.18 (a) (b) Transitive closure of Figure 5.6

Figure 5.20 (b) describes the adjacent matrix that represents the transitive closure of the XML graph (having cycles) in Figure 5.6. The number of connections is 55 ("connection" in this case means that there is an edge between two nodes (e.g., "1" means there is an edge, "0" no edge)). Figure 5.20 (a) shows the adjacent matrix of the transitive closure of the DAG in Figure 5.17 (a). It has six connections. Therefore, the result is about 9-times more compact than the transitive closure of the original graph. Moreover, there is a direct relationship between the number of connections of the transitive closure and the size of the label sets (L_{IN} and L_{OUT}), i.e., the smaller the number of connections, the smaller the result.

As previously demonstrated, several optimization techniques have been introduced to overcome the memory space-consuming problem. In the next sections, several optimization techniques to overcome the time-consuming problem will be discussed also.

5.3.3 Efficient Computation of Densest Sub-graph Revisited

Another question that needs consideration is how much time it takes for the algorithm to create the index. As mentioned previously, one of the disadvantages of the 2-hop cover algorithm is its time consumption for computing the densest subgraph (Definition 5.8) for all center graphs (Definition 5.6) in each step of the algorithm.

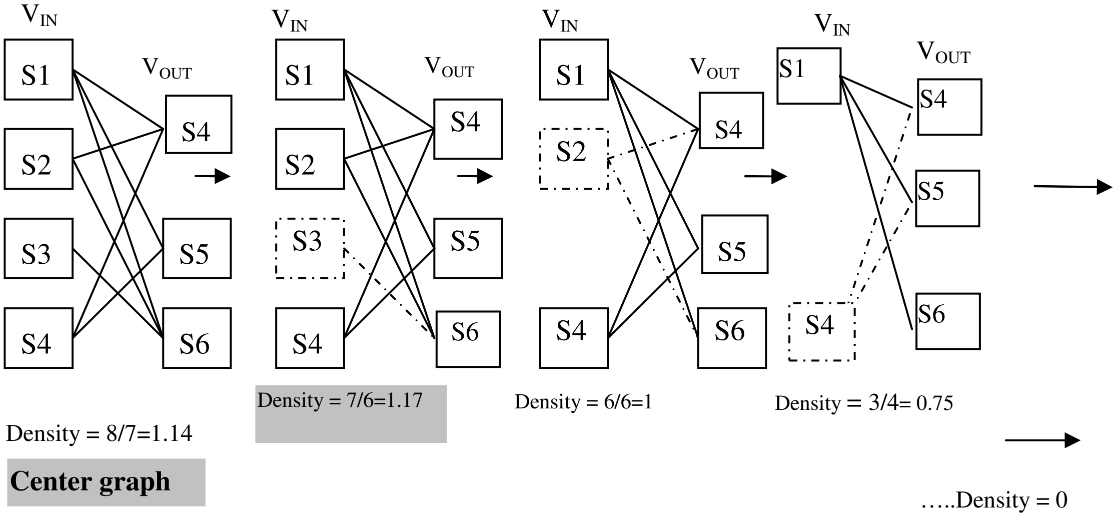


Figure 5.21: Computation of densest sub-graph from a center graph

Figure 5.21 explains how the *density subgraph* can be evaluated from a given *center graph*. The algorithm works in iterations, it removes node S3 from the given *center graph*, because S3 has a minimum degree. It then evaluates the *density* for the resulting densest subgraph (e.g., 1.17). Then it removes node S2 and evaluates the *density* of the resulting densest subgraph (e.g., 1) and so on until the density is equal to zero.

This process generates a sequence of several subgraphs of the given *center graph* (as shown in Figure 5.21). The algorithm returns the *densest subgraph* with the highest *density*. In Figure 5.21, the algorithm returns the *densest subgraph* of node S2, because it has a high *density* among the others (e.g., it is density equal 1.17). It is O ($|V|^2$) in the worst case in each step, a *polynomial-time algorithm* is used. This is considered very time-consuming, especially, with large XML document graphs.

If a DAG is given, the HID index requires building the transitive closure of a DAG as an input. Then, for each SCC in the DAG the center graph G_x has to be constructed. The algorithm proceeds in iterations, where the basic operation in each iteration is the selection of the densest sub-graph. The nodes of the selected sub-graph are removed from the transitive closure. This procedure is repeated until all the connections in the transitive closure are covered. For each SCC in the DAG, during the 2-hop labeling computation, this SCC is added to the two sets L_{IN} and L_{OUT}. Figure 5.22 (a) shows the resulting DAG from the original XML graph (Figure 5.6) using the SCC algorithm. It has five nodes instead of the original ten.

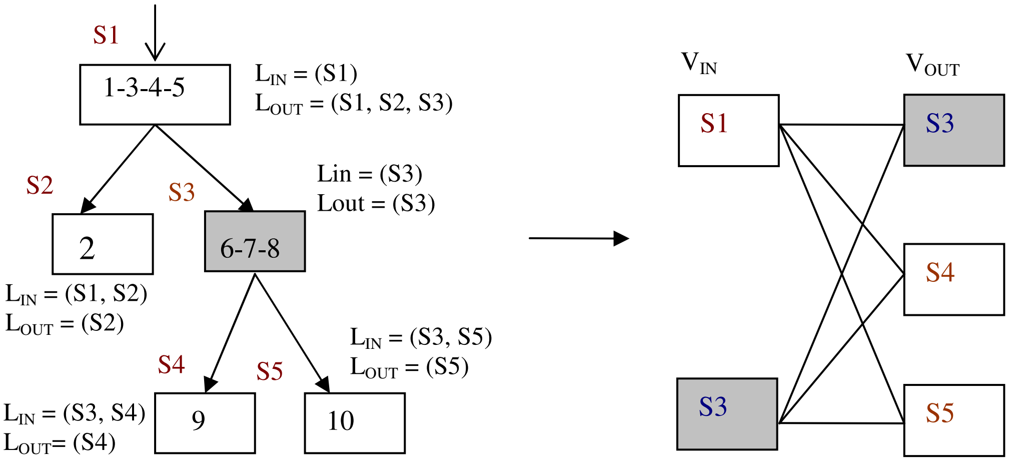


Figure 5.22**:** (a) 2-hop labeling of DAG (b) the center graph of node S3 in Figure (a)

Figure 5.22 illustrates that this approach (based on SCC) minimize the size of the label *L(u)* given for each node *u* as much as possible compared with the size of the label given for each node in Figure 5.6. Thus, the size of the HID index and the time required to build it are reduced significantly. In the next section, several other techniques to overcome the time-consuming problem during the implementation process will be introduced.

5.3.4 Other Optimization Issues

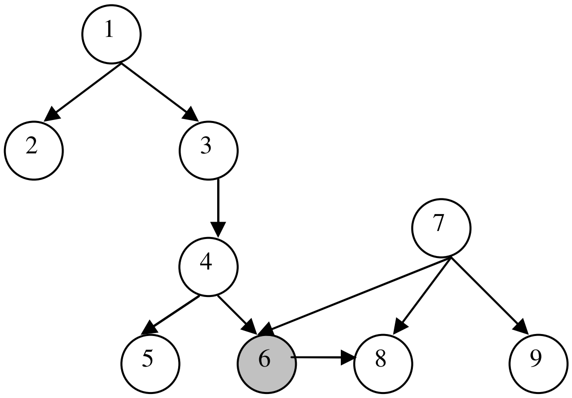
As previously described, Cohen studied the 2-hop cover algorithm from a mainly theoretical point of view. We adapted this algorithm so that it is as simple as possible (concentrated on our application on XML collections). There are several issues (i.e., see Section 5.2.3) which we do not need when this algorithm is applied on XML graphs that have cycles. These issues make the implementation of the 2-hop algorithm more complicated. In this section, the pseudo codes for the 2-hop cover shall be rewritten, and therefore, all unnecessary issues that we do not need in the XML graph application shall be disregarded.

To begin with, the 2-hop cover evaluates the *densest subgraph* for all *center graphs* in each step of the algorithm in order to return the center graph that has the highest *density* among all these graphs. It was noted in section 5.2.2.1 (Example Scenario) that the *densities* of most *center graphs* do not change in every step of the algorithm. This implies that there is no need to recompute the *densest subgraphs* in each step of the algorithm if the *center graph* does not change. To deal with this problem, one would suggest computing the *density* d_u of the densest subgraph of a center graph of each node u at the beginning of the algorithm. Each node u is stored in a priority queue together with its corresponding *density* that was evaluated as the first step of the algorithm [STW04]. After this, we do not need the transitive closure at all. In each step of the algorithm the node x with the current maximum density from the queue was extracted. Then the validity of the stored density was verified. This can be tested by recomputing the *density* of the node x (using Definition 5.8).

If it occurs that the maximum *density* (that we extracted from the priority queue) and the newly computed *density* are different, for example, the extracted density is larger than d_x (the newly evaluated density), then the node x with its newly computed density in the priority queue is to be reinserted. Then, a new current maximum *density* from the queue is sought. This process is repeated until the extracted *density* is less than or equal to the newly computed (see Example 5.3). Cohen et al. [CHKZ02] discusses a similar idea to keep the *density* of the densest subgraphs in a heap, but their technique requires more space in case all center graphs are stored in main memory. In the following, an example shall be given in order to demonstrate this optimization technique.

Example 5.3

In this example, it shall be demonstrated how the *density* for each node can be evaluated and stored in a priority queue. Consider the XML graph in Figure 5.23.



Nodes	C_{IN}	C_{OUT}	density
1	0	6	2.3
2	1	0	1.6
3	1	4	1.5
4	2	3	2.3
5	3	0	1.8
6	***4***	***1***	***3.2***
7	0	3	2.7
8	6	0	2.1
9	1	0	1.3

Figure 5.23: (a) XML graph representation (b) Densities table

As mentioned previously, the *density* can be evaluated by this relation (Edges / nodes (see Definition 5-7)). Figure 5.23(b) shows the table of C_{IN} and C_{OUT} for each node in Figure 5.23(a) with its corresponding density. It was already noticed that node 6 has a maximum density in the table (e.g., this density is called the "*extracted density*" *d* = 3.2). Thus, node 6 together with its corresponding *density* is to be extracted from the table. Then the *center graph* for this node and its *densest subgraph* is to be constructed. Then, its corresponding density (in the same way that described in section 5.2.2.1 (example scenario)) *d1*= 12/7 = 1.71 is computed. We call this "*computed density*". Then the *extracted value d* (3.2) from the table and the computed value d1 (1.71) are to be compared. If *d1*≤ *d*, then replace *d* and *d1* in the priority queue. The density of node 6 becomes 1.71. Now the current maximum density from the queue (e.g., node 7 with *density* v*alue* 2.7) is extracted. This method until all densities in the priority queue are completely scanned is to be repeated.

For more optimization we say that a *center graph* is a full graph [STW04] if and only if ($|C_{IN}| \times |C_{OUT}| = |E|$), where $|C_{IN}|$ and $|C_{OUT}|$ represent the number of in-/out nodes and $|E|$ represents the number of edges.

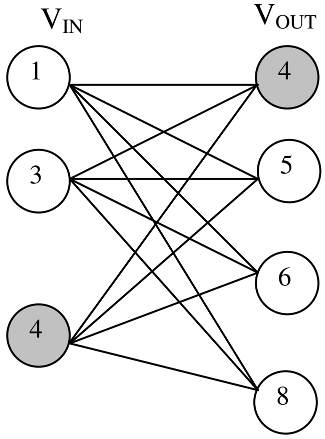


Figure 5.24: Center graph of node 4 (in Figure 5.15 (a))

For example, the center graph in Figure 5.24 is a full graph because $|C_{IN}| \times |C_{OUT}|$ = 3×4 = 12 and the number of edges is $|E|$ = 12. Therefore, from Definition 5-7, the density = Edges / nodes = $|E| / (|C_{IN}| + |C_{OUT}|)$. When the *center graph* becomes a full graph, then the *density* = $(|C_{IN}| \times |C_{OUT}|) / (|C_{IN}| + |C_{OUT}|)$ which is considered the maximum density we can get (this can also be noted from the example scenario in Section 5.2.2.1). This means that any full graph from the center graph is always its densest subgraph with maximum density.

Figures (5.25, 5.26, and 5.27) show the most important algorithms needed to build the HID index in a pseudo code. We drop all the complexity issues from the 2-hop cover algorithm that we do not need for the XML graphs with cycles.

```
Procedure CenterGraph ($x \in V$)
$C_{IN}$ = In($x$) // set of in_degree for each node (from the transitive closure)
$C_{OUT}$ = Out($x$) // set of out_degree for each node (from the transitive closure)
for all ($u \in C_{IN}$) do
    for all ($v \in C_{OUT}$) do
        if ($v \in$ Out($u$)) then
            $E_x = E_x \cup \{(u, v)\}$    //add edge between u and v
        end if
    end for all
end for all
for all ($u \in C_{IN}$) do
    if (DEGREE ($u$) = 0) then // remove the isolated node from center graph
        $C_{IN} = C_{IN} \setminus \{u\}$
    end if
end for all
for all ($v \in C_{OUT}$) do
    if (DEGREE ($u$) = 0) then //this node has no edges
        $C_{OUT} = C_{OUT} \setminus \{u\}$         // remove from the Cout
    end if
end for all
RETURN ($C_{IN}$, $C_{OUT}$, $E_x$) // return bipartite center graph
```

Figure 5.25: Compute center graph for a given node in the XML graph

```
Procedure Densest Sub-graph ($x \in V$)
if ($|C_{IN}| \times |C_{OUT}| = E_x$) then    //Test if the center graph is full graph
    $FullG_x$ = true
else
    $FullG_x$ = false
endif
$dmax = |E_x| / (|C_{IN}| + |C_{OUT}|)$        //compute the maximum density
while ($|E_x| > 0$ and $FullG_x$ = false) do
    ($C_{IN}$ , $C_{OUT}$ , $E_x$) = RemoveMin ($C_{IN}$ , $C_{OUT}$ , $E_x$) ) //remove nodes from TC
    $d" = |E_x| / (| C_{IN} | + |C_{OUT}|)$         // evaluate the new density
    if (d" > dmax) then                // compare between densities
        $d" = d_{max}$                        //replace the values
    endif
    if ($|C_{IN}| . |C_{OUT}| = E_x$) then
        $fullG_x$ = true
    endif
endwhile
RETURN ($d_{max}$, $C_{IN}$ , $C_{OUT}$ )   // return the densest subgraph with max. density
```

Figure 5.26: Method to evaluate a densest sub-graph for all center graphs

The procedure in Figure 5.25 is used to efficiently evaluate the *center graph* of every node *w* in the underlying graph. Step (1) and step (2) compute the *C_{IN} (x)* and *C_{OUT} (x)* sets from the transitive closure. Step (6) adds edges between nodes; step (12) removes the nodes that have no edges (i.e. isolated nodes). This procedure returns a *center graph* for a every node *x*. The resulting *center graph* is used as an input to the procedure in Figure 5.26 to evaluate its densest subgraphs. Steps (1) and (2) test if the center graph is a full graph (which means its density is the maximum density; then we do not need to run the algorithm to find the densest subgraphs). Step (6) computes its density. Step (7), step (8), and step (9) remove the node with minimum edges and evaluate a new density. Step (10) compares this new density with the old one. This procedure returns the *densest subgraph* with maximum density.

```
Procedure 2-Hop Labels (G = (V, E))
(1)  forall ($x \in$ G) do
(2)   $L_{IN}(x) = L_{OUT}(x)$ = { } // at the start the 2-hop labels are empty for each node
(3)   $C_{IN}(x) = \{v \in V| P_{vx} \neq \varphi\}$   // $P_{vx}$ represents all the shortest paths from v to x
(4)   $C_{OUT}(x) \{v \in V | P_{xv} \neq \varphi\}$ // $P_{xv}$ represents all the shortest paths from x to v
(5)   $d_x = |C_{IN}(x)| . |C_{OUT}(x)|$     // density before constructing the densest subgraph
(6)    INSERT (Q, ($x$ , $d_x$))      // insert each node with its density in the queue
(7)    end forall
(8)    while (|V| 〉 φ)
(9)    repeat
(10)   ($x$, $d_x$) = ExtractMax (Q)   //extract the maximum density from the queue
(11)   ($C_{IN}(x)$, $C_{OUT}(x)$ , $E_x$)  = CenterGraph (x) //call the center graph
(12)    d' , ($C_{IN}(x)$, $C_{OUT}(x)$)  = Densest Sub-graph ($C_{IN}(x)$, $C_{OUT}(x)$, $E_x$)
(13)     if (d' < $d_x$) then            // test the extract density and the computed density
(14)     INSERT (Q, ($x$, d'))       // insert the new max. density into the queue
(15)     until (d' = $d_x$)
(16)    forall v $\in C_{IN}(x)$
(17)       $L_{IN}(v) = C_{IN}(v) \cup \{v\}$
(18)       $L_{OUT}(v) = C_{OUT}(v) \setminus C_{OUT}(x) \cup \{v, x\}$
(19)    forall v $\in C_{OUT}(x)$
(20)         $L_{IN}(v) = C_{IN}(v) \setminus C_{IN}(x) \cup \{v\}$
(21)         $L_{OUT}(v) = C_{OUT}(v) \cup \{v, x\}$
(22)     $L_{IN}(x) = L_{OUT}(x) = \{x\}$   // label the center node x
(23)    Remove ($C_{IN}(x)$, $x$, $C_{OUT}(x)$)   //remove the center graph from TC
(24)    end while
(25)   end
```

Figure 5.27: Method to evaluate 2-hop labeling for input graph G

The procedure in Figure 5.27 utilizes the main 2-hop labels algorithm. The input to the algorithm is the set of connections (TC) that are not yet covered. Step (3) and step (4) compute for every node *x* the set of C_{IN} *(x)* and the set of C_{OUT} *(x)*. Step (5) and step (6) compute the *density* for every node *x* and store it in a queue at the beginning of the algorithm. It extracts the maximum *density* from the queue with its corresponding node *w* (step (10)). Steps (16-22) update the two-hop labels by adding to each node the label sets. Step (23) remove the node *x* along with its two sets C_{IN}(x) and C_{OUT}(x) from the transitive closure. The algorithm works in iteration, until all connections are removed from the transitive closure.

5.3.5 Query Evaluation with the HID Index

As mentioned in chapter 3, path expressions are at the core of all XML query languages. In the literature exists neither an XML (or semistructured) algebra nor a standard XML query language. XML queries are often expressed by regular expressions that have an SQL–like syntax. XML queries may consist of one or more path expressions, because there is often a need to express multiple search conditions. Here, a *wildcard* is used in queries that were submitted to the HID Index (as the schema of the underlying XML documents is not known in advance or it may change often). The HID index can efficiently evaluate the following path expressions:

- *Reachability query (e.g. u // v)*: the HID index tests the reachability between two nodes *u* and *v* as follows. First, look at the SCC table (Figure 5.17 (b)) by running SQL queries against this table. If both nodes belong to the same SCC, the reachability between *u* and *v* is already proven. This process needs linear time without any support by the HID index. In the case that the two nodes *u* and *v* do not belong to the same SCC, the HID index has to be used. As mentioned previously, all relevant information is stored in database tables. Our precise database scheme is introduced in the implementation chapter (next chapter). Then, an SQL statement against the HID index in order to attain the two labels (L_{IN} and L_{OUT}) that contain both nodes is to be executed. If and only if the “intersection” between two labels is not NULL, a path between these two nodes exists.

- *Searching for descendants / ancestors*: To evaluate all descendants or ancestors for a given node (a task that can stress any index structures), the HID index first looks up the SCC table to get the corresponding nodes. It then submits an SQL statement against the HID index tables to get the L_{OUT} set label. In the same way, all ancestors for a given element name can be computed. For more information about query evaluation techniques see chapter 7.

6

Implementation Study

This chapter explains in detail the different methods that are used to implement the HID index. The index structure is stored in a relational database that makes it possible to use SQL statements to evaluate path queries.

The structure of the chapter is as follows. Section 6.1 explains how to deal with XML links during the parsing process. Section 6.2 describes our database schema. Section 6.3 explains the implementation of the HID index. Section 6.4 explains the path expression evaluation with the help of the HID index.

6.1 How to Deal with Linked XML Documents

As explained previously, in addition to the unidirectional hyperlinks of HTML (Hypertext Markup Language) a specific language XLink is used for specifying advanced links between XML documents. The link characteristics are described by so-called linking elements. In XLink, a link is a relation between two or more resources. These resources can be described within an XML document by using specific XML elements with specific attributes and nested elements. Resources can be either: (i) local: an element inside the document; (ii) remote: an external resource usually identified by its URI. Each XLink element represents either a resource involved in the link or the link itself. XLink attributes support, among others, the specification of: (i) the type of the link (attribute type); (ii) the resource involved in the link using their URI address; (iii) information concerning how linked XML documents must be presented to the user; (iv) Information concerning when the link must be traversed; (v) information concerning the source at the destination of the link.

6.1.1 Search Algorithm for Internal-links (ID/IDREF(S))

An element instance in an XML document cannot directly be derived from the appearance of this element, even though it realizes a link. Such information can only be deduced from the schema description of the XML document (DTD or XSD). Thus, an XML document needs to rely on an explicit schema description in order to be processable. Therefore, it can be derived whether an attribute is of type ID. An ID or IDREF(S) appears in the schema description as follows:

1. <!ATTLIST address name ID#REQUIRED>
2. <!ATTLIST publication author IDREFS#REQUIRED>

In the first schema description, the element *address* with the attribute *name* serves as an ID while in the second description the element *publication* with the attribute *author* constitutes a reference (IDREF) to an ID element that was defined elsewhere in the document.

How to parse an XML document and how to get the information about the links during the parsing process is explained as follows:

1. The source XML document is to be parsed together with its schema description. For this purpose, the two prevalent models exist: DOM (Document Object Model) [DOM] and SAX (Simple API for XML parsing) [SAX]. SAX is an event driven API that calls event methods while parsing the document. It provides methods that can react to specific data in an XML document while reading data. SAX makes it possible to read just required data sequentially, not the complete document. DOM parses the complete document, creates an object for the document, and saves available memory. It is not useful when the document is very large and the available memory is relatively small. DOM

provides a programmatic access to the complete document in a non-linear fashion. Though the DOM model is easier to use, the SAX model allows faster parsing and requires less memory.

2. The parser detects the element name, which has the value (ID#) in the schema description. Then it executes the extended element constructor function, which adds the information that this local element name realizes a link. It finds all element names that have the value (IDREF(S)) in their schema description. These element names represent the target element names.

6.1.2 Search Algorithm for XLink

A link defined by XLink may realize a one-to-one link (from one source XML document to one target XML document) or one-to-many links (from one source XML document to several target XML documents). Considering these links, the search algorithm works as follows:

1. The source XML document is parsed.
2. The parsing process finds all the target documents or anchors that have the attribute “href” in the parsing source document. Then, the parsing process finds the local element name that has this attribute. After that, it finds the root element for every target document.

6.1.3 Search Algorithm for XPointer

XPointer provides a general way for selecting fragments of an XML document by writing a set of expressions. An expression can be used to select children, siblings, and nodes with given attributes. There are two types of *“absolute terms”*, firstly, *URL# id()* secondly, *URL # root(),* are used here. These expressions can be extended to the *URL # id().children (number, name, attribute, value)* or *URL # root().* Children (number, name, attribute, value) (more information about XPointer is already discussed in chapter 2). The algorithm works as follows:

1. The source XML document is parsed.

2. The parsing process finds all XML target documents that contain the XPointer expressions (doc.xml# XPointer). This XPointer expression points to an element name (target element name) inside of the target XML document.

In every above-discussed case, the relevant information about the source XML document and the target XML documents is stored in the Link-table of the database. The next section describes the database schema in detail.

6.2 Database Schema

This section describes the main database schema that is used to implement the HID index. After XML documents are parsed, they are represented as a large graph. Then, the cycles are fetched and a new DAG is constructed. Next, the TC for a DAG is evaluated. The result of the TC is a set of connections, which are used as input parameters for the HID index. The information about the HID is stored in two tables. The main database tables are described as follows:

- XML collections are represented as a graph in the database

  URLS (URLid, URL, Lastmodified)

  NODES (Eid , Ename, URLid)

  EDGES (Eid1, URLid1, Eid2, URLid2)

- Strongly Connected Components table (Directed Acyclic Graph representation)

  SCC (SCCid, Eid)

- HID index tables

  ANCES (SCCid, INid)

  DESCS (SCCid, OUTid)

The underlined attributes are the primary keys of the respective tables. The URLS, NODES, and EDGES tables represent the actual XML collections. Each element in the XML document represents a node in the corresponding XML graph with an object identifier (Eid). Each XML document is stored in the URLS table with its identifier (URLid) and the last modified time. Each node in the XML graph is stored in a table NODES with its object identifier (Eid), its name, and its source XML document identifier (URLid). The EDGES table has the information about edges between the individual nodes. An edge represents a connection between two nodes located in the same XML document or located in different documents. Each edge in the XML graph is represented by its source (Eid1) and target (Eid2) object identifiers; with its source XML document integer (URLid1) and target document integer (URLid2). The SCC table is used to compress the large XML graph to its DAG with minimum number of nodes and edges. Each SCC is defined by an integer (SCCid) and the set of element identifiers (Eid) that construct the SCC. At the end, the HID index tables (ANCES and DESCS) are constructed. These tables are the main tables for the HID index that capture L_{IN} and L_{OUT} sets. Here, SCCid represents the identifier of the SCC. For each entry of the SCC, INid and OUTid sets that correspond to the L_{IN} and L_{OUT} sets are stored.

Since XML documents are very large, it is not efficient to look up all the actual data stored in the tables every time to evaluate path queries. To avoid this problem and to evaluate queries efficiently the following two database indexes are built on ANCES and DESCS tables (F&B-index) [ABS00]. "A forward database index" (or top-down approach) of the concatenation of SCCid and INid for the ANCES table and the concatenation of SCCid and OUTid for the DESC table ((SCCid, INid) and (SCCid, OUTid) as primary keys) (more information about this approach was already discussed in chapter 3). "A backward database index" (or bottom-up approach) on the concatenation of INid and SCCid for the ANCES table and on the concatenation of OUTid and SCCid for the DESCS table (more information about this approach was already discussed in chapter 3 too). Moreover, a B+ tree index [RG00] on the NODES and SCC tables is built. The B+ tree is an "efficient" data structure that can search quickly huge quantities of data. In the following section, more information about the B+ tree index is given and its implementation is described.

6.2.1 B+ Tree Index

A B+ tree index is a page-oriented tree that has the following properties: First, it is a balanced leaf search tree (actual keys are presented only in the leaf pages, and all paths from the root to a leaf are of the same length). A B+ tree is said to be of order *d* if every node has at most *2d* separators (a key usually implies that associated information for that value exists in the index. A *separator* defines one-step in a search path to leaf pages that contain actual keys and associated information). Each node except the root has at least *d* separators. The root has at least two children. The leaves of the tree are at the lowest level of the tree (level 1) and the root is at the highest level. The number of levels in the tree is termed as the tree height. A non-leaf node with *j* separators contains *j+1* pointer to children. A <pointer, separator> pair is termed as an index entry. Thus, a B+ tree is a multi-level index with the topmost level being the single root page and the lowest level consisting of the set of leaf pages. Figure 6.1 summarizes these concepts [RG00].

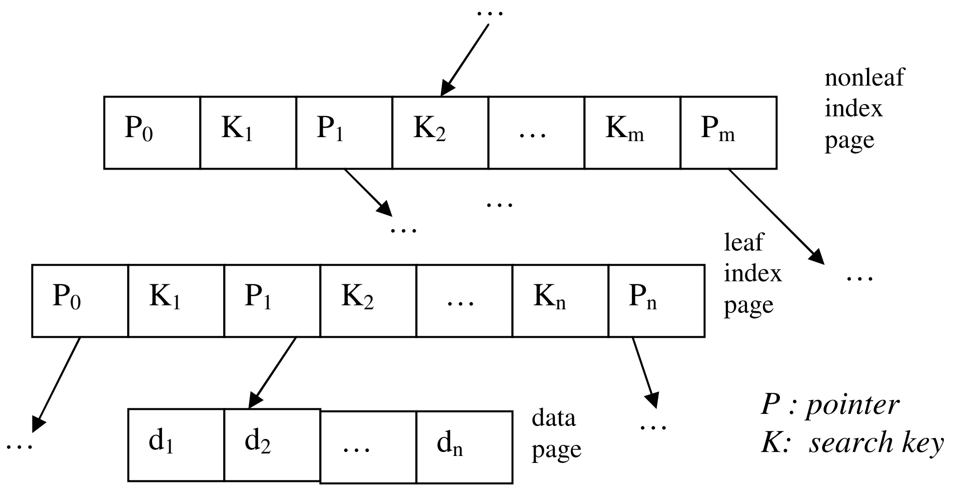


Figure 6.1: Structure of a B+ tree

The index is stored on disk and the search, insert, or delete operation starts by searching the root to find the page at the next lower level that contains the subtree having the search key in its range. The next lower level page is examined, and so on, until a leaf is reached. The leaf is then examined and the appropriate action is performed.

Search algorithm of B+ tree: The algorithm finds the leaf node to which the given data entries belong. Figure 6.2 shows the pseudocode for the B+ tree search algorithm [RG00], i.e., the search for a record with a search-key value *k*. The search process begins with the root node, looking for the smallest search-key value greater than *k*. Let us assume that this search-key value is k_i. The search process follows pointer p_i to another node. If $k \langle k_1$ then the search follows p_1 to another node. If the tree has *m* pointers in the node and $k \langle k_m$, then the search follows p_m to another node. Once again, the search process will look for the smallest search-key value greater than *k*, and follows the corresponding pointer. Eventually, the search process will reach a leaf node, at which point the pointer directs it to the desired data record. Thus, in the processing of a search process a path is traversed in the tree from the root to a leaf node.

```
func find (search key value k) returns nodepointer
    //Given a search key value, finds its leaf node
return tree-search(root, k)     //search from the root
endfunc
func tree-search (nodepointer, search key value k)
      returns nodepointer
if nodepointer is a leaf then    // search tree for entry
return nodepointer
   else
        if ($k \langle k_i$ ) then
         return tree_search ($p_0$ , k); // p  pointer
         else
              if ($k \langle k_m$ ) then
              return tree_search ($p_m$ , k);
              else
                   find i  such that $k_i \le k \le k_{i+1}$;
                   return tree_search ($p_i$, k)
end func
```

Figure 6.2: Search algorithm of a B+ tree

6.3 HID Index Implementation

As explained previously, in [CKM02] the concepts of the 2-hop cover algorithm from a theoretical point of view are described. Thus, they considered several implementation and scalability issues and did not consider the XML applications with a very large graph. We simplified this algorithm as much as possible to work efficiently with a large XML graph by leaving out several issues, which we do not need in the XML applications. The overall architecture of the HID index is described in Figure 6.3. From this figure, it is easy to understand the structure of the index and its implementation. The steps that are used to implement the HID index are as follows.

- The Transitive Closure (TC) (i.g., G = (V, E')) of a DAG is built. It consists of all connections E' in DAG. TC is an input parameter to the HID index. In the initiation step, all connections E' are not yet covered and the 2-hop labels for each node are empty.
- For each node *x* in a DAG, a *center graph* $G_x = (V_x, E_x)$ is built. This center graph is a bipartite undirected graph with node sets $V_{IN}(x)$ and $V_{OUT}(x)$, where $V_{IN}(x)$ contains all the nodes from which a path to *x* exists in DAG (e.g., all ancestors of node *x)*, and $V_{OUT}(x)$ contains all the descendants of *x* in DAG. An edge between each node in $V_{IN}(x)$ and each node in $V_{OUT}(x)$ is added.
- For each center graph G_x the problem of evaluating the two sets $V_{IN}(x)$ and $V_{OUT}(x)$ for each node *x* is exactly the same problem of evaluating the *densest subgraph* of the center graph of node *x*. In practice, the 2-approxiamation algorithm [CHKZ02] for constructing the densest subgraph from the underlying center graph in a linear time is used. This algorithm iteratively removes a node of minimum degree from the center graph. This generates a sequence of *m* subgraphs of the original center graph. The algorithm returns a sequence of subgraphs with their densities. The density of a subgraph is the average degree of its nodes.
- The node *x* is chosen where the center graph has high density among all nodes of the center graph.
- Now, the 2-hop cover is updated. Let G'_x be the densest subgraph of the center graph G_x. For each node *w* in $V_{IN}(x)$, the node *x* to the set $L_{OUT}(w)$ (descendants) is inserted, and for each node w in V_{OUT}, the node *x* to the set $L_{IN}(w)$ (all ancestors) is inserted.
- All connections from the TC that are covered by this center graph are removed; the result node labels are called 2-hop labeling, which is still not 2-hop cover. Now the procedure is iterated from the start until all the connections are covered. At this state, the 2-hop label is the 2-hop cover.

All algorithms that are used to implement HID index are discussed in chapter 5.

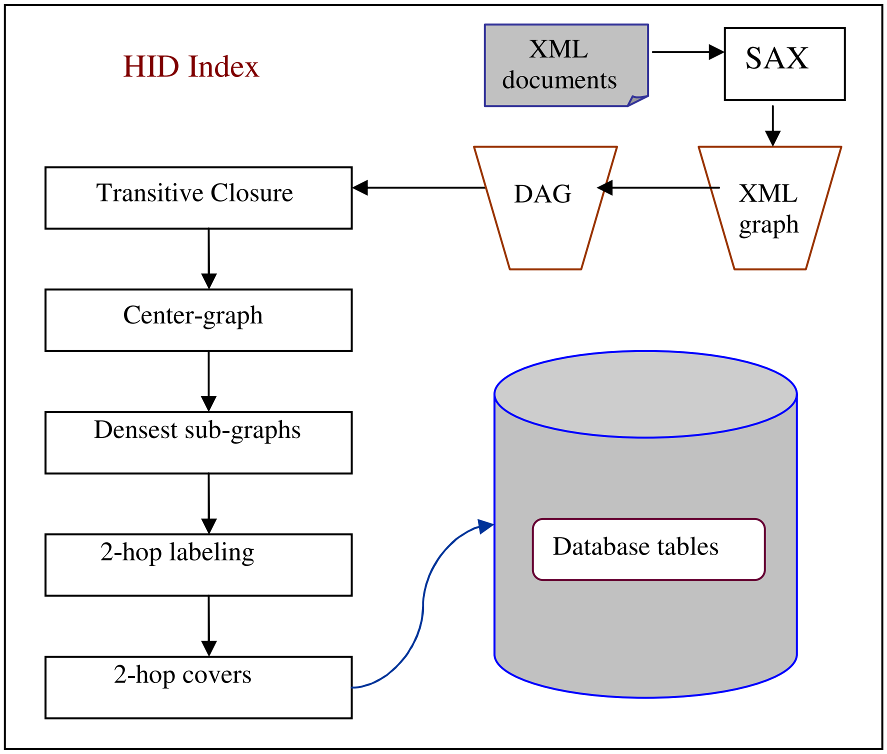


Figure 6.3: The structure of HID Index

6.4 Query Evaluation Techniques

This section explains the details how to evaluate different types of queries using the HID index. The overall architecture of query-processing framework is shown in Figure 6.4. Firstly, XML documents are parsed. The parsing layer is the place where the document is transformed from a string of Unicode characters into a conveniently accessible data structure. The “SAX parser” is updated in a way that it takes XML documents as an input and produces a graph as output. The resulting graph is stored into a database. Secondly, HID index as a database-package is constructed in order to store all information in database tables. Finally, “query evaluation” and “query output” that evaluate the SQL statements against the data are stored in the database and return the result of the current query to the user.

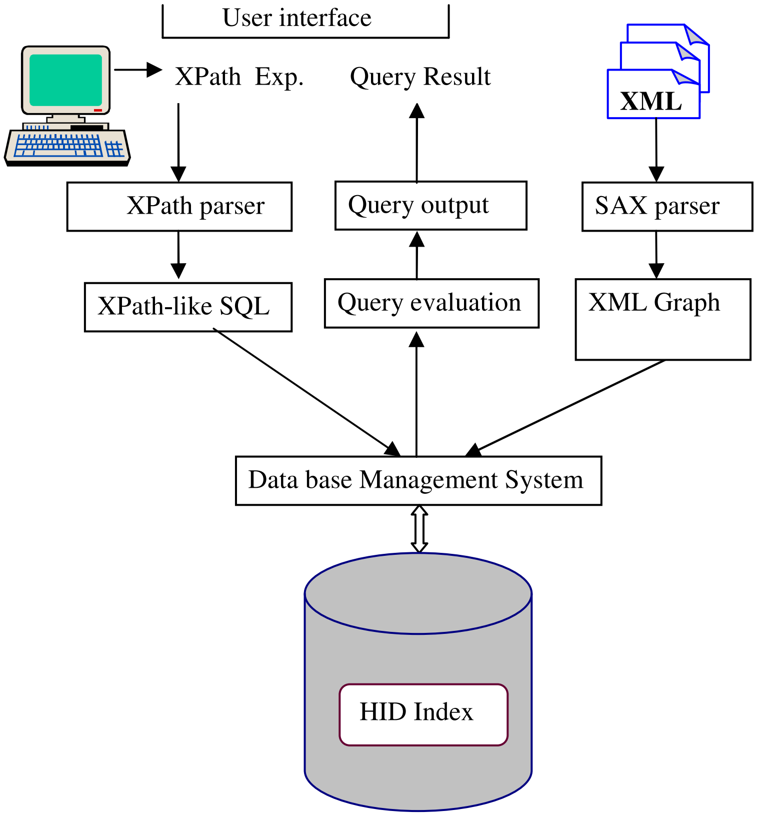


Figure 6.4: Framework of Query evaluation with HID index

As described before, the HID index is optimized for three types of path expressions. Firstly, ancestor-descendant query (a//b) which tests the reachability between two nodes. Secondly, descendant-or-self axis, for examples, (a//Y) which is used to find all the successor elements are of type Y, (a//*) to find all the successors of the current element a. Finally, ancestor-or-self axis, for example (//a) to find all the ancestors of element a. Now the way to evaluate these path expressions over HID index is described in next section.

6.4.1 Reachability Tests

The query processing in XML database always involves determining ancestor-descendant structural relationship in addition to parent-child structure relationship. Users may not know the exact structure of the XML document in absence of a schema. In a “navigation-based query processing”, the node that matched with the ancestor must be kept for a long time to wait for the matching descendants. In the "index-based query processing", there are two options: one is to maintain only parent-child node pairs and obtain ancestor-descendant pairs through repeated joins, which will take too much query processing time; the other is to maintain all ancestor-descendant relationship that will lead too much space costs.

To handle the difficulty of determining the ancestor-descendant relationship (or reachability query), HID index is proposed (that uses labels to represent the positions of the elements in XML graph) which makes the checking of ancestor-descendant structural relationships as easy as checking the parent-child structural relationships. HID index can determine the ancestor-descendant relationship (or the reachability) between two given nodes as follows:

1. For any two given nodes *a* and *b*, the SCC table is looked up. If the two nodes are located at the same SCC, the reachability query between *a* and *b* is true. It needs linear time. The HID index is not fired necessarily and the traversing graph problems are avoided. The experiments show that 30% of the submitted reachability queries to database are determined from the SCC table without needing to fire the HID index.
2. There are four steps to determine the reachability between two nodes located in different SCCs using the HID index. First step, it is easy to look at NODES table by a simple SQL query to get the corresponding identifiers for the two nodes. Suppose these nodes have two identifiers "Eid1" and "Eid2", a B+ tree index that build on NODES table will speedup the scanning of this table to get the identifiers. Second step, the SCC table is looked up to get the corresponding SCCids (e.g. "SCCid1" and "SCCid2"), here also a B+ tree index that is built on SCC table evaluates this query efficiently. Third step is to get the corresponding L_{OUT} set from DESCS table corresponding to "SCCid1" and L_{IN} set from ANCES table corresponding to "SCCid2". In Fourth step, the **intersection** of the two sets is performed, if *L_{OUT} (SCCid1) INTERSECT L_{IN} (SCCid2)* is not empty, there is a reachability between the given two nodes, otherwise, the nodes are not connected.

Since the two label sets L_{IN} and L_{OUT} are stored in tables, the reachability between two nodes can be tested in $O(\min\{|L_{IN}|\} + \{|L_{OUT}|\})$. This means there is a direct relationship between the size of the labels and the required time to execute the query. The size of the labels is a NP-hard problem [CHKZ02] the optimal size cannot be computed efficiently. The optimization techniques that are proposed in HID index minimize this size as much as possible. In the following, the XPath-like SQL statements, which can be submitted to the database to test the reachability between two given nodes "n1" and "n2" located in different SCCs are explained.

Firstly, the NODES table is looked up to get the corresponding identifiers using the following SQL statement:

```
SELECT   Eid
FROM     NODES
WHERE    Ename ='n1' AND Ename = 'n2';
```

This SQL query returns two identifiers that correspond to "n1" and "n2" (e.g., values of these identifiers are "Eid1" and "Eid2").

Secondly, the following SQL query is submitted to SCC table to get the SCCids that correspond to the node identifiers (from first step):

```
SELECT    SCCid
FROM      SCC
WHERE     Eid = 'Eid1' AND Eid = 'Eid2';
```

This SQL query returns two values (e.g., "SCCid1" and "SCCid2") that correspond to the two node identifiers. The B+ index that is built on both NODES and SCC tables helps to evaluate the shown two SQL queries efficiently. Moreover, it is proposed that each of the two given nodes "n1" and "n2" has a unique name (which means that the first step returns only two identifiers). In general, the node name may be repeated several times in database, so the COUNT function must be used in the first step (e.g., SELECT COUNT (*)). This function returns the number of occurrences of n1 and n2 in database (see more information in the next chapter).

Thirdly, look up at the HID index tables (ANCES and DESCS tables) to find the set of L_{OUT} for the identifier “SCCid1” and L_{IN} for the identifier “SCCid2”. The following SQL query is used to do this:

```
SELECT  OUTid
 FROM    DESCS
 WHERE  SCCid = ‘SCCid1’
                INTERSECT
                     (SELECT  INid
                       FROM    ANCES
                        WHERE  SCCid = ‘SCCid2’);
```

The above SQL query performs the intersection between the L_{OUT} set (OUTids) of the node “n1” and the L_{IN} (INids) set of node “n2”. If the result of this query has a non-zero value, “n1” and “n2” are connected. Otherwise, “n1” and “n2” are not connected (e.g., our system returns this statement, “*No rows selected*”).

6.4.2 Finding Descendants /Ancestors for a Given Node Identifier

HID index has the ability to evaluate the descendant (ancestor)-or-self-axis. This type of query is more complex especially in case of large graphs with long paths and the user has no knowledge about the schema of the underlying data. The HID index can efficiently evaluate these queries by writing an XPath-like SQL query against the database tables. For example, consider that all descendants of an element, which contains the object identifier “Eid1”, have to be evaluated. The corresponding SQL query is as follows: First, the SCC table is looked up to get the corresponding “SCCid” for the current given identifier “Eid1” as,

```
 SELECT    SCCid
 FROM       SCC
WHERE    Eid =’Eid1’;
```

This query returns the SCC identifier that contains the given element identifier (e.g., SCCid1). Then the HID index tables are looked up to get the result, using these SQL statements.

```
SELECT  OUTid
FROM    DESCS
WHERE   SCCid ='SCCid1';
```

The above query returns a set of SCCids, which contains the descendants of the underlying “Eid1”. In the same way, all ancestors of a given identifier can be evaluated.

6.4.3 Finding Descendants/Ancestors for a Given Node Name

Suppose that the descendants of a given element name “N” have to be evaluated. First, the NODES table is looked up for the corresponding identifiers. The following SQL query is used.

```
SELECT   Eid
FROM     NODES
WHERE    Ename='N';
```

The result of the above query may be one identifier or a set of identifiers (which means that this node “N” is repeated several times in the database). Then, for each identifier, its corresponding “SCCid” was to be found using the following query:

```
SELECT    SCCid
FROM      SCC
WHERE     Eid='Eid1';
```

Then the HID index is looked up to get all descendants/ancestors of the given node “N”.

```
SELECT   OUTid
 FROM     DESCS
WHERE   SCCid ='SCCid1';
```

Note, if there are several Eid1’s (e.g., a node N is repeated several times in database), all descendants/ancestors for each identifier are evaluated and the results are sorted by the number of descendent for each identifier.

To summarize, after the HID index is created, the evaluation of path expressions is an easy task. The reachability can be easily tested; compute all the ancestors and all descendants of a given node by submitting a simple SQL query against the HID index tables. The additional indexes (B+ tree and F&B-index) are helpful to increase the performance. In the next chapter, the experimental work is explained and HID index is compared with other indexes. The comparison depends on three parameters: time required creating the index, space requirements, and time required to return the query.

7

Experimental Results

In this chapter, we empirically compare the performance of the HID index against the HOPI index and the transitive closure as a connection index. The comparison concentrates on the memory space requirements to build each index, the time needed to construct each index, and the performance of the query processing. A series of performance experiments is done.

This chapter is structured as follows. Section 7.1 describes the experimental setup. Section 7.2 explains the proof of our concepts. Section 7.3 applies the HID index over large XML document collections. Section 7.4 provides the discussion of the experimental results.

7.1 Experimental Setup

7.1.1 Experimental Platform

The experiments are performed under two environments. The first experiment is performed on a Pentium IV-2GHz platform with windows-XP and 788 MB of RAM. Oracle-OracHom 9.2 is used as a database server. The second experiment is performed on a Pentium 1V-3 GHz platform with Linux Suse 9.1 and 3GB of RAM (e.g., Doppy server). IBM DB2 8.1 [DB2] is used as a database server and a single hard disk with 120 GB. All strategies of the HID index are implemented as a database application (set of tables), using a java-based application to store the information into tables.

7.1.2 Data Set Description

As a real-life example of XML data with links, the Internet Movie Database (IMDB) [IMDB] (see Appendix A) is used. The general characteristics of this data are as follows. The center file of the IMDB has a list of movies, each with a unique identifier. The actors of those movies are listed with their roles in a distinct file. All directors are listed in an independent file, with a number of important producers, writers, and cinematographers. The IMDB is used because it was identified as a highly cyclic database likely to stress the path-indexing algorithms.

Now, the generation of linked XML documents from this data (see Appendix B) is described. A small subset of movies and all people (actor, directors…) associated with these movies is randomly chosen. One XML document for each *movie* is generated and an XLink to the *actors* and the *director* for the underlying movie is added. The portion of the used database is organized around *movie* elements and elements for classes of people who appear in movie credits, for example, *actor*, *director*, *composer*, etc, as well as a wide variety of information about movies. Cyclicity arises since each movie element is serviced as ID references and has pointers to individuals who acted in the movie, and each element representing an individual pointer to the movies in which she or he acted. This datasets consists of 40,211 nodes and 44,349 edges, among which 2,718 are of type IDREF.

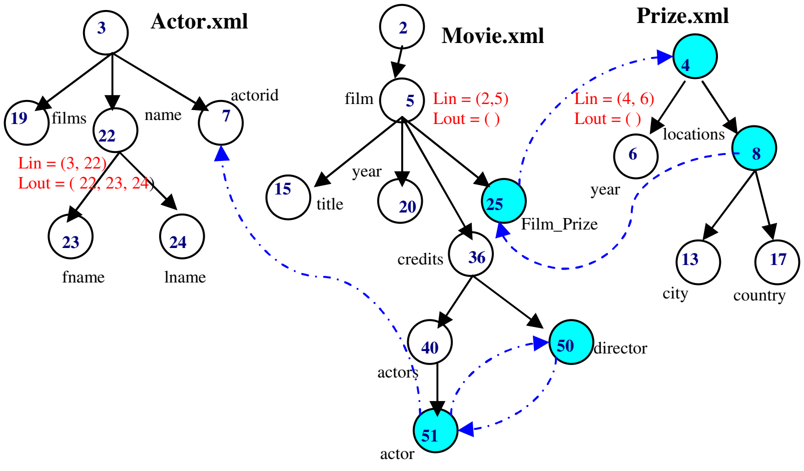


Figure 7.1: Part of the XML document used in Experiment 1

Figure 7.1 illustrates the structure of three XML documents, Actor.xml, Movie.xml, and Prize.xml, which are connected together using XLink, XPointer, and ID/IDREF(S)-references. In this figure, each node has a label with a unique object identifier; nodes with identifiers 5, 6, 22 have information about HID index.

7.2 Experiment 1: "Proof of Concepts"

At first, the concepts are studied by using a small fragment of the movies database from the generated set as described above. With this fragment, the efficiency of the HID index against the HOPI index is proved and the transitive closure is used as a connection index.

7.2.1 Space Requirements

This small fragment has 1, 235 nodes, 1, 411 edges, 53 Xlinks, and 123 IDREFS. The TC for this fragment has 17, 213 connections; each connection needs 2x4 Bytes for storage. Therefore, the TC for this fragment needs about 137, 704 KB. Then the 2-hop cover for the underlying graph (or the HOPI index without partition) is evaluated. The number of entries in L_{IN} and L_{OUT} tables is about 2,103 entries. Each entry needs 2x4 Bytes of space. Therefore, the storage requirement for the HOPI index is about 16 KB and the same amount for the backward index (see Section 6.2).

XML Graph	# Nodes	1,235
	# Edges	1,411
	# Global Links	46
	# Internal links	130
TC	# Connections in TC	17, 213
	Space required for TC	137, 704 KB
HOPI index	# Entries in L_{IN}	963
	# Entries in L_{OUT}	1,140
	Cover size	2,103 entries
	Space required for HOPI	32 KB

Table 7.1: Storage requirements for TC and HOPI index for the original graph

Table 7.1 shows the information about the small XML fragment and the storage requirements for the HOPI index and the TC as a connection index.

For the HID index, first, the cycles from the original XML graph are removed. The underlying small fragment has 24 cycles. The maximum number of nodes in one cycle is 15 nodes. The result is a DAG with 782 nodes (SCCs). The transitive closure for the DAG has 10,117 connections and needs about 79 KB for storage. The size of the covers is about 1,323 entries (2× 4 bytes per entry). The storage required for the HID index is about 10 KB for the underlying small fragment. Table 7.2 shows the results.

DAG	# Nodes	782
	# Edges	862
	# Global links	25
	# Internal links	55
	# Cycles	24
	Max. No. of nodes in cycle	15
TC	# Connections in TC	10,117
	Space required for TC	79 KB
HID index	# Entries in L_{IN}	610
	# Entries in L_{OUT}	713
	Cover size	1,323 entries
	Space required for HID	10 KB

Table 7.2: Storage requirements for HID index

The different experiments using different XML document sizes from the subset, which we created from IMDB with different number of nodes, are performed to see the effect of cycles on indexes sizes. A table 7.3 shows the results.

7.2.1.1 Discussion

Tables 7.1 and 7.2 illustrate the difference between space requirements for the HOPI index, the HID index, and TC as a connection index. First, the storage needed for the TC of the original graph and the TC of a DAG is compared. The TC from the original graph needs about 137,704 KB for storage and the TC from the DAG needs about 79 KB for storage.

#Nodes	#Global links	# Internal links	# Cycles	# Nodes in cycle	Indexes size		
					TC	**HOPI**	**HID**
2,500	94	141	30	15	388 KB	52 KB	41,15 KB
3,200	110	20	95	15	441 KB	87 KB	50,22 KB
5,000	137	196	132	187	0.5 MB	293,3KB	70,931KB
6,240	163	210	140	152	0.83 MB	316 KB	85,36 KB
7,100	213	392	174	195	0.98 MB	361 KB	97,9 KB
8,312	287	401	213	286	1.01 MB	398 KB	110 KB
9,530	392	470	242	364	1.87 MB	411 KB	127,41 KB

Table 7.3: Index sizes

This means that the optimization techniques based on the principle of the SCC, reduce the storage requirements for the TC as an input parameter to the HID index about 57 %. The storage requirements for the HOPI index and the HID index are compared. The comparison shows that the HID index reduces the memory space needed to store the covers about 16 % as compared to the HOPI index. With the HOPI index, space was dominated by the space needed for storing the transitive closure of the original graph, which means that the space is increased if the document size is increased. The proposed techniques that build the HID index depend on the strongly connected component techniques to deal with the redundancy problem and to avoid excessive memory space consumption.

Table 7.3 shows the database storage requirements for the indexes (Indexes size). The structure of this table can be described as follows: (#Nodes) the number of nodes, the number of global links (#global links (XLink and XPointer)), the number of internal links (#internal links (ID/IDREF)), the number of cycles (#cycles), and the maximum number of nodes in a cycle (#Nodes in cycle). It is noteworthy that the idea of considering strongly connected components of the XML graph instead of the XML graph itself sounds promising. Table 7.3 illustrates that the cycles and the number of nodes that construct each cycle have a direct effect on the storage required for the indexes. For example, if the underlying graph has one cycle constructed from 187 nodes, the TC has 34,969 (187×187) connections only for this cycle, and each connection needs eight Byte for storage; so this cycle needs 273.195 KB. The index minimizes this cycle to only one node (SCC) which need only 8 Bytes for storage.

Table 7.3 shows that the HID index is about 5-times more compact than the HOPI index (in case of the number of nodes that construct the SCC are large) and 13-times more compact than the transitive closure as a connection index. Therefore, the scalable HID index can save a lot of space as compared with the HOPI index (without partition and without distance information), and it can represent compactly connections in a highly interlinked XML collection where the indexes mostly fail to deal with this kind of data.

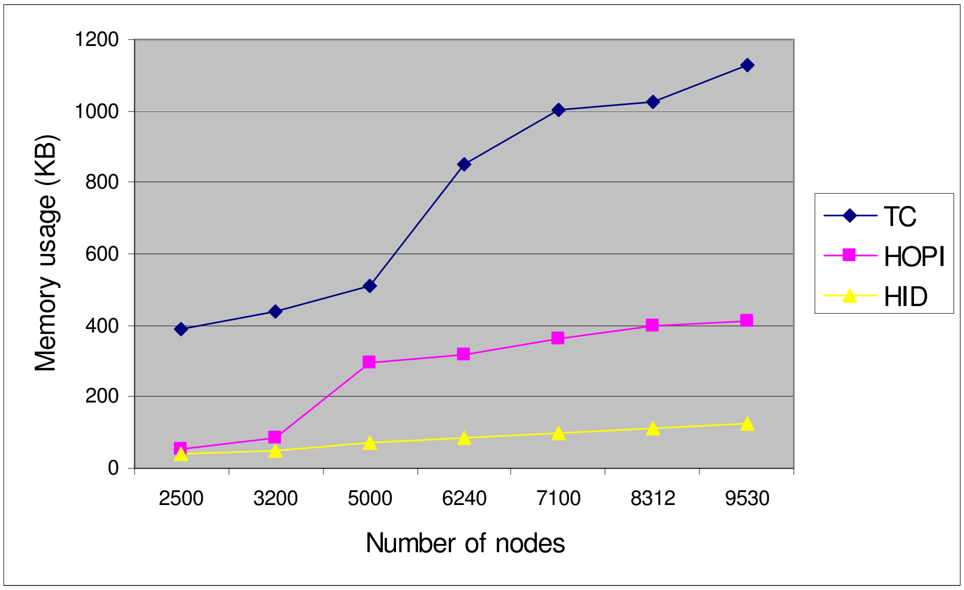


Figure 7.2: Index sizes

Figure 7.2 shows the memory usage for indexes (TC, HOPI, HID). The X-axis represents the number of nodes and Y-axis represents the memory usage (see table 7.3). This figure shows that the HID index needs less memory for storage than TC as a connection index and the HOPI index. It is also noteworthy to remark that the higher the number of cycles in the graph, the lower is the storage needed by the HID index. It shows, if the data has very few cycles (and the maximum number of nodes that construct these cycles are very few) the storage required for the HID index is nearly the same as for the HOPI index (this is obvious in Figure 7.2 when the number of nodes are 2,500 and 3,200 nodes).

# Nodes	Indexes built time	
	HOPI (S)	**HID (S)**
1,235	11	9.4
2,500	18	15.9
3,200	25	22.2
5,000	41.3	27
6,240	49.6	31.1
7,100	59.07	43.01
8,312	69.5	50.7
9,530	77.04	57

Table 7.4: Indexes built time

7.2.2 Index Construction Time

In this section, the time needed to build the HID index and the HOPI index is compared. The same fragment of Internet Movies databases (IMDB) is used. Firstly, the build time for the HOPI index, which is based on the original XML graph with cycles, is measured. Secondly, the cycles from the XML graph are removed and then the HID index is applied on the resulting graph (DAG). Table 7.4 shows the time required to build the indexes.

7.2.2.1 Discussion

Figure 7.3 illustrates, if the underlying XML graph has no cycles the time needed to build the HOPI index and the HID index is nearly the same (this is shown by the figure when the tested nodes are 1,235 and 2,500 nodes). In case the underlying graph has more cycles, the HID index saves about (25%) of the time as compared to the HOPI index (without partition and distance queries; note that HOPI index with partitions and distance queries needs more time and space [STW04]). This is related to the following reasons:

- The HID index reduces the redundant computation of the transitive closure by shrinking every strongly connected component of the input graph, which results in a DAG with a minimum number of nodes (SCCs). The time needed to compute the covers is less than the time needed to evaluate the covers from the original XML graph.
- The HID index is based on the Nuutila algorithm [NS93] instead of the Floyd-Warshall algorithm [CLRS01] (as HOPI index and 2-hop covers) for computing the transitive closure, which has quadratic time complexity in a number of nodes compared with a cubic time complexity by the Floyd-Warshall algorithm.

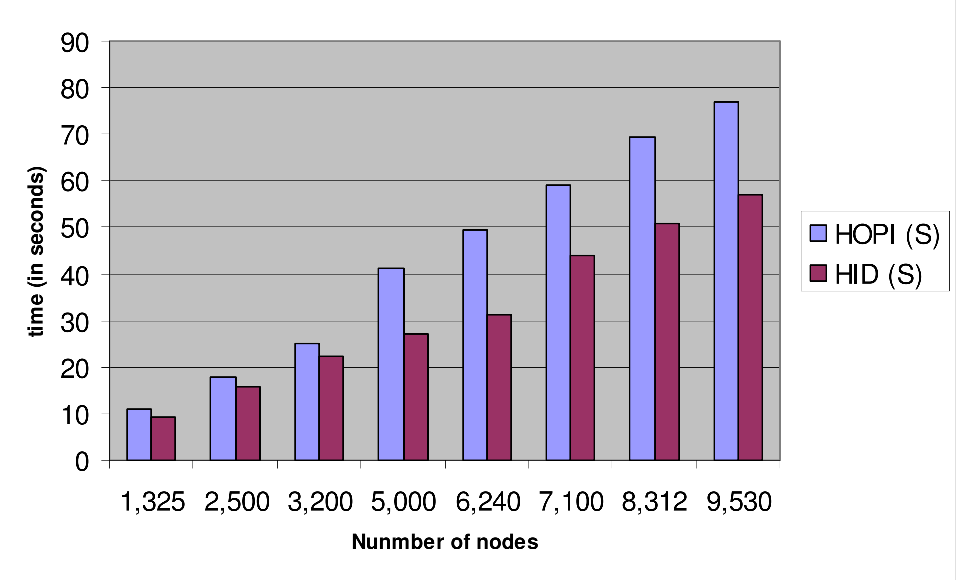


Figure 7.3: Indexes built time

7.2.3 Query Evaluation

This section focuses on three types of XPath queries:

- Reachability tests or so called *"ancestor-descendant query"* to test whether there is a path between an ancestor node and a descendant node in a complex XML graph with long paths. There are two possible cases. First, if a node *u* and a node *v* are located in the same strongly connected component, this means that, they are connected. In this case, the reachability test can be determined in linear time. Second, if node *u* and node *v* are located in different strongly connected components, then the reachability test can be determined by comparing their label sets.

- All descendants of a given node *u* (or d*escendants-or-self axis*) are to be found.
- All ancestors of a given node *u* (or *ancestors-or-self axis*) are to be found.

Reachability Tests

To test the reachability several queries are submitted to the small subset fragment of IMDB. First, a single query is submitted to the stored database to measure the required time needed by the HID and the HOPI indexes. Then, a collection of queries is submitted to examine the average query execution time.

To assess the performance of the HID index against the HOPI index the path query “*actorid // film_prize” (e.g., is there a reachability between actorid and film_prize)* is submitted to the database. As mentioned in chapter 6, there are two cases to test the reachability. The first case is based on the nodes name and the second case is based on the identifiers. The above path query can be tested as follows:

- **First**: Test the reachbility between two nodes by names.

From table NODES the object identifiers for these nodes are obtained. The following SQL query is submitted to the stored database.

```
SELECT  Eid                      SELECT Eid
FROM    NODES                    FROM    NODES
WHERE   Ename = “actorid”;       WHERE    Ename = “film_prize”;
```

The above query returns “14” identifiers of “actorid” node and “3” identifiers of film_prize node. For each “acotid” the identifier tests if there is reachability with each film_prize identifier. After wards, several SQL queries are submitted to the database. It is noted that about 30% of the reachability queries could be tested in a linear time without help of the HID index (this means that they are located in the same SCC). One example is to test the reachability between two identifiers and then in the same way test the reachability between all identifiers that were returned by the above SQL query. Let us consider an identifier “7” of “actorid” and an identifier “25” of “film_prize”. These identifiers are used to test whether these nodes are located in the same strongly connected component or not by executing the following SQL queries against the SCC table.

```
SELECT   SCCid               SELECT   SCCid
FROM     SCC                 FROM     SCC
WHERE   Eid = “7”;           WHERE  Eid= “25”;
```

If the above SQL queries return the same SCCid, then the names “actorid” and “film_prize” are located in the same SCC. So, the reachability is proven. This process needs linear time. As mentioned in chapter 6, the B+ tree index that is built on NODES and SCC tables helps to evaluate these SQL queries efficiently. Otherwise, the two nodes are located in different SCCs. The results of the above SQL queries are two identifiers “7 “and “5”. This means that the “actorid” node itself is a SCC of size one and “film_prize” located in a SCC with identifier “5”.

In this case, the help of the HID index is needed to test the reachability. The following SQL query is submitted to the HID index tables.

```
SELECT  OUTid AS IDs
   FROM    DESCS
   WHERE  SCCid = ‘7’
              INTERSECT
                   (SELECT  INid AS IDs
                    FROM    ANCES
                         WHERE  SCCid = ‘5’);
```

This SQL statement returns “*No rows selected*” (e.g., the intersection between the two label sets is ϕ). Then, there is no reachability between “actorid” and “film_prize”.

Table 7.5 shows the average time needed by the HID and the HOPI indexes to test the reachability between “actorid” node and “film_prize” node. It illustrates that the time measured to execute the query is nearly the same for both indexes. This is because the two nodes are located in different strongly connected components and the underlying small fragment of IMDB has very few cycles.

# Nodes	HOPI-index	HID-index
1,235	0.03 (sec)	0.02 (sec)

Table 7.5: Average execution time for the query “actorid//film_prize”

- **Second**: Test the reachability between two nodes by identifiers.

To test the reachability between two nodes by their identifiers is straight forward. We repeat all the above steps that determine the reachability between two nodes by their names without needing to look at the NODES table to get the identifiers.

Ten SQL queries are submitted to the database using different identifiers. One example query is explained. All the queries are evaluated in the same way. Consider the query “5//40” (e.g., test the reachability between the identifier “5” and the identifier “40”). The intersection is a non-zero value. This means that there is a reachability between the two given identifiers “5” and “40”. The average execution time for this query is (0.01) for the HID index and (0.04) for the HOPI index.

Find all descendants /ancestors

As described above, the HID index is also optimized for descendants/ancestors-or-self axis. Two cases to evaluate all descendants/ancestors of a given node are described as follows:

- **First:** All descendants of a given node will have to be found by name.

This query ***“find all the descendants of actor node” or “actor//”*** is submitted to the HOPI index and then to the HID index. First, the identifiers corresponding to the “actor” node are obtained by executing the following SQL statement on the stored database.

```
SELECT  Eid                        SELECT SCCid
FROM    NODES                      FROM SCC
WHERE   Ename = “actor”;           WHERE  Eid=’Eid1’;
```

Using the additional indexes that are built on the tables NODES and SCC, the underlying query can be efficiently evaluated. This query returns all identifiers that have a name “actor”. For each identifier its descendants are evaluated by using the following SQL statement.

```
SELECT  OUTid
FROM    DESCS
WHERE   SCC_id = “n1”;
```

Running this query against the HOPI index returns “21” and against HID index returns “16”. Looking at the NODES table returns 21 identifiers that have the name “actor”. Using these identifiers to get the corresponding SCCid returns 16 identifiers (two actor nodes are located in the same SCC). Then the results are sorted according to the number of descendants for each identifier.

Table 7.6 shows the total time that the HOPI and the HID indexes need to return all the results for the “actor// *”. The HID index is 2-times faster than the HOPI index.

#Nodes	**HOPI-index**		**HID-index**	
	Time (s)	# results	Time (s)	#results
1,235	0.4	21	0.2	16

Table 7.6: Average execution time for the query “actor //”

For the HOPI index, more duplicated results for this query are found. Duplicated results may occur, because there are a number of cycles in the link structure on the fragment database. Our optimized techniques used in the HID index overcome this problem and achieves a higher evaluation performance than HOPI index.

Figure 7.4 shows the query average execution time to compute all the descendants of a given node actor as a function of a number of the descendants. The X-axis in Figure 7.4 represents the number of results in ascending order and the Y-axis represents the query execution time for each result.

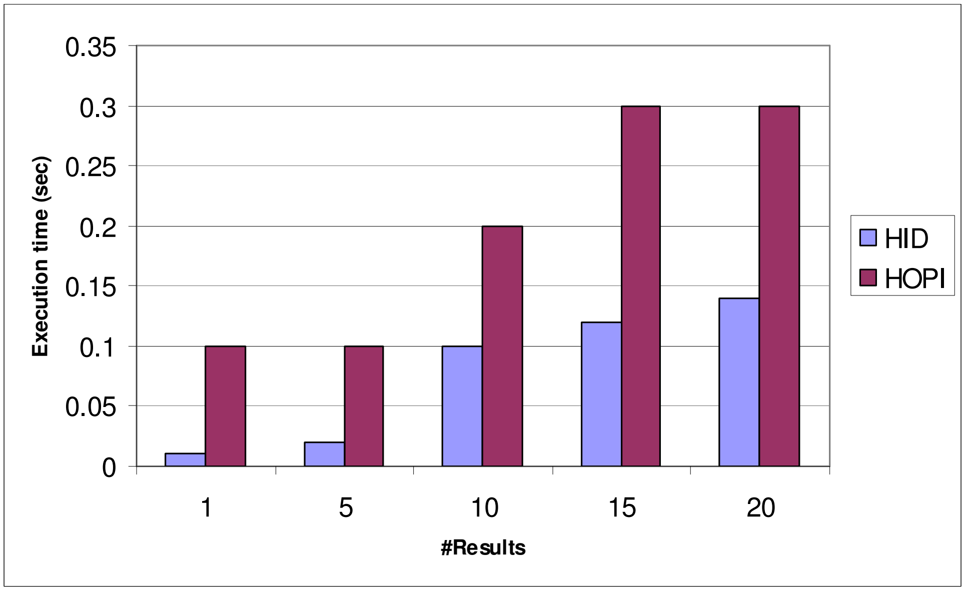


Figure 7.4: Number of results and time to compute all the descendants of "actor //"

Figure 7.4 illustrates that the HOPI index has the problem of duplicating results. However, the HID index avoids this problem. In addition, the HID index needs less query execution time than the HOPI index.

- Second: "Find all descendants of a given node by its identifier ID"

This query "*find all descendants of a given node that has identifier 100*" (e.g., "*100//")* is submitted to the HOPI index and then to the HID index. This path query is more easily compared to the above path query that is based on the name of the node. However, the evaluation of the descendants for only one node that has a unique identifier is needed. The HOPI and HID indexes need nearly the same time to return the result. The following SQL query is submitted to evaluate all the descendants for a given identifier.

```
SELECT  OUTid
FROM    DESCS
WHERE   SCC_id = "100";
```

In the same way all ancestors of a given node can be evaluated by its name or identifier.

7.3 Experiment 2: "Benchmark"

This section discusses the performance of the HID index for large XML graphs with many cycles. The complete subset is used, which is generated from IMDB. This subset consists of 712 XML documents of size 4.71 MB. Table 7.7 shows the statistics of the XML data model. The database server is used as a platform for this experiment.

Documents	Nodes	Edges	Links	Global-links	Internal-links
712	40,211	44,349	4,839	2,121	2,718

Table 7.7: XML data model

7.3.1 Space Requirements

First, the TC (Transitive Closure) for the result graph is computed. This TC has 1, 983, 216 connections. Each connection needs 2×4 Bytes for storage. Therefore, the TC for this subset needs 15, 1 MB memory storage. The HOPI index is constructed for this subset without partition (e.g., corresponding to the standard 2-hop covers) and without distance information. The covers typically get larger if they include distance information. The number of entries of L_{IN} is equal to 90,220 and the number of entries of L_{OUT} is equal to 73,048 entries. This means that the total number of entries for the HOPI index is 163,268 entries (which represent the size of the covers). Since both the L_{IN} and L_{OUT} tables have two columns (see Section 6.2), each entry needs 2×4 Bytes for storage. The storage requirement for the HOPI without partitions is about 2.49 MB.

Working with a DAG: Optimization techniques are applied to the underlying subset. The result is a DAG with a minimum number of nodes. The resulting DAG has 25, 622 SCCs and 23,331 edges with 2034 cycles. The maximum number of nodes in one cycle is 899 nodes. The TC for the DAG has 127,160 connections. Each connection needs 2×4 Bytes for storage. Therefore, the TC for this DAG needs 1.01 MB memory storage.

The total number of entries for the HID index is 18,324 entries (L_{IN} table has 9, 500 rows (entries) and L_{OUT} has 8,724 rows (entries)). Each entry needs 2×4 Bytes for storage, so the storage requirement for the HID index is about 0.27 MB. Table 7.8 shows these statistics.

HOPI index					**HID index**						
#Nodes	#L_{IN}	#L_{OUT}	#entries	Size (MB)	#SCCs	#Cycles	#Nodes in SCC	#L_{IN}	#L_{OUT}	#Entries	Size (MB)
40,211	90,220	73,048	163,268	2.49	25,622	2034	899	9,500	8,724	18,324	0.37

Table 7.8: Index sizes

7.3.1.1 Discussion

Table 7.8 shows that the idea of considering the strongly connected component technique sounds promising. The HID index saves more space compared to the HOPI index or with the TC as a connection index. The TC of the DAG, which is used as an input parameter to the HID index, is about 37-times more compact than the TC of the original XML graph, which is used as an input to the HOPI index. The reason is that the original XML graph has many cycles. In this experiment, one SCC is constructed from 899 nodes. This means that the number of connections for this SCC is (899×899 = 808,201 connections). The optimization techniques minimize these connections to only one connection. It is also evident that the HID index is about 9-times more compact than the HOPI index. Therefore, the HID index can save a lot of space in comparison to the HOPI index and the TC as connection index.

Now, the size of the underlying data (subsets of IMDB) is increased to 90,000 nodes. It is observed that the HOPI index cannot efficiently compute the cover for the densely connected collection. It is turned out that there is a strongly connected component of about 10, 000 elements, thus, the transitive closure has at least 100, 000, 000 (10,000 ×10,000) entries, which cannot be handled. The HID index deals with this problem efficiently. However, there is a direct relationship between the efficiency of the HID index and the number of elements that construct the strongly connected component. Figure 7.5 explains the relationship between the storage needed for the indexes with the number of nodes that construct the SCC. The X-axis in Figure 7.5 represents the number of nodes that construct the SCC and the Y-axis represents the storage needed for every index. The figure illustrates that the HID index is more efficient and needs less memory storage than the HOPI index, in case the number of nodes, which construct an SCC, is increased. In addition, the HOPI index becomes huge and needs more memory spaces in comparison to the HID index if the number of nodes, which construct the cycle increases.

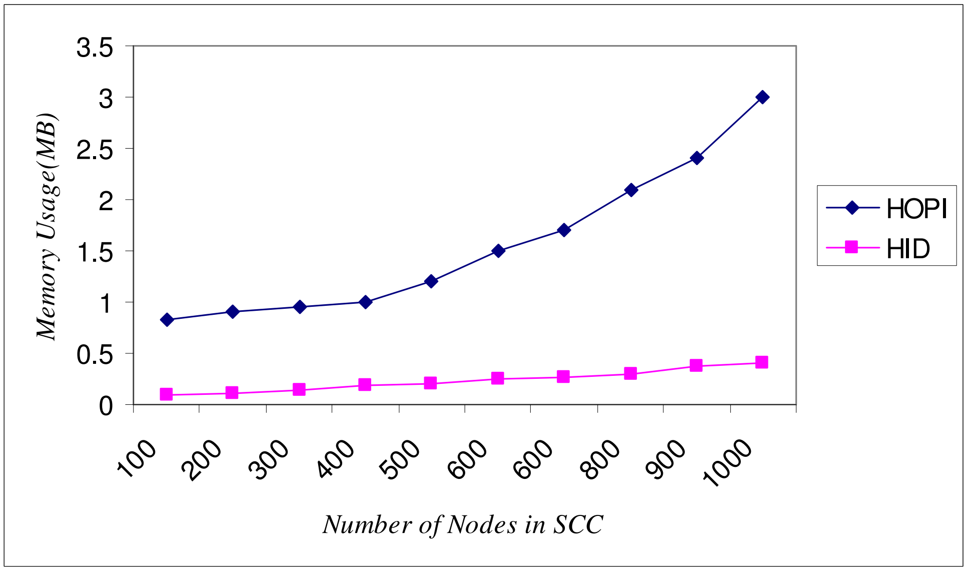


Figure 7.5: Index sizes based on the number of nodes in SCC

7.3.2 Index Construction Time

This section describes the time needed to build the HID and the HOPI indexes over the subset of movie database (see table 7.7). The series of experiments is repeated using different numbers of nodes to discuss the effect of the number of cycles and the number of nodes, which construct each cycle, on the building time of the indexes. Table 7.9 shows the building time for both indexes. It consists of two main parts. The first part is related to the HOPI index, which has the following columns: Number of nodes (#Nodes), number of connections (‘connections) as an input to the HOPI index, and the index building time. The second part for the HID index has the same columns as the HOPI index, but instead of number of nodes, there is number of SCCs (# SCC).

HOPI index			HID index		
#Nodes	#Connections	Build time(m)	#SCCs	#Connections	Build time(m)
40,211	163,268	15	25,622	18,324	4.9

Table 7.9: Indexes build time

7.3.2.1 Discussion

Table 7.9 illustrates that the time needed to build the HID index is much less than the time needed to build the HOPI index. This is obvious because the number of connections of the DAG that is used as an input to the HID index (see optimization techniques described in chapter 5) is much smaller than the number of connections of the original XML data, which is used as an input to HOPI index. Therefore, the HID index saves about (32%) of the time compared to the HOPI index.

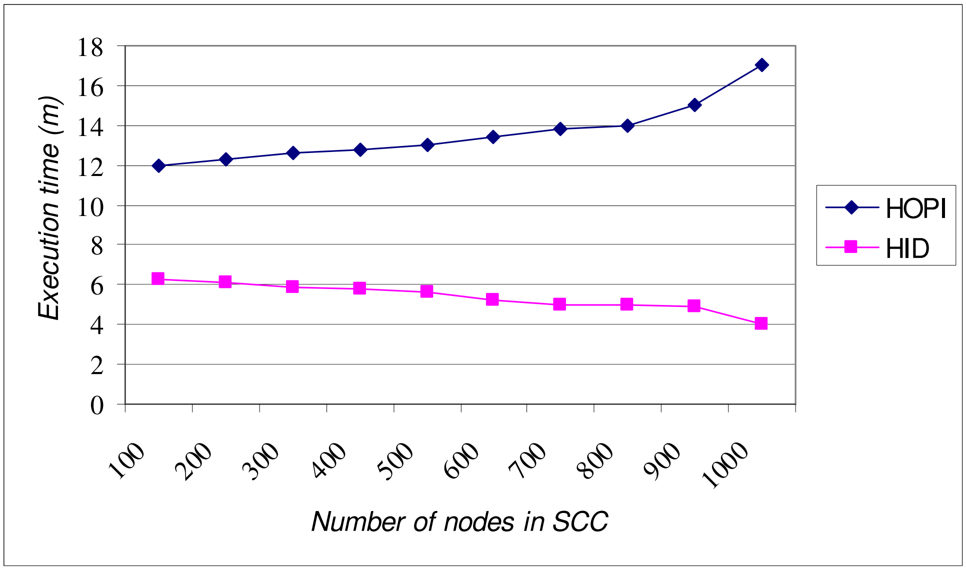


Figure 7.6: The effects of cycles on indexes build time

The effect of the cycles and the number of nodes, which construct each cycle during the building time is discussed. In Figure 7.6, the X-axis represents the number of nodes in each SCCs in each underlying subset and the Y-axis represents the execution time. It is noted that when the number of cycles in the underlying subset data is increased, the time needed to build the HID index decreases. Then, the time needed to build the HOPI index is increased. It is also noted that the number of nodes that construct the cycles has a direct effect on the indexes building time.

This proves that the HID index is more efficient than the HOPI index where dealing with data that has many cycles. In addition to this, the time needed to build the HID index is less than the time needed to build the HOPI index for the underlying subset, which has a large graph with cycles.

Query	Query in English	Path Expression	**HID** time (sec)	**HOPI** time (sec)
Q1	Test the reachability between movie and actor	movie//actor [name =”Hak”]	0.4	0.67
Q2	Test the reachability between movie and actor	movie//director[name=”Nissim”]	1.2	1.92
Q3	Test the reachability between locations and film_prize	locations//film_prize	linear time	0.98
Q4	Test the reachability between actor and director	actor[name=”Hanks”]//director [name=”Zemeckis”]	linear time	3.4
Q5	Test the reachability between casting and title	casting//title[“Animal Instinct”]	0.24	0.5
Q6	Test the connectivity between identifiers (8, 25)	8//25	0.01	0.01
Q7	Test the connectivity between identifiers (50, 90)	50//90	0.01	0.02
Q8	Test the reachability between identifiers (100,300)	100//300	0.01	0.03
Q9	Test the reachability between identifiers (8, 25)	1000//4000	linear time	2.4
Q10	Test the reachability between identifiers (200, 12100)	200//12100	3.1	5.3

Table 7.10: Execution time for reachability queries using HID and HOPI indexes.

7.3.3 Query Evaluation

The subset of IMBD described in table 7.7 is used to study the query performance of the HID index against the HOPI index. This section does not explain in detail the submitted SQL queries to the database (that is already discussed in section 7.2.3 and in chapter 6). First, reachability between two nodes is tested and then all descendants/ancestors are evaluated for a given node.

Reachability Tests

To assess the performance of the HID index ten path expression queries are submitted to the database. Table 7.10 shows these ten path expressions. The first five path expressions test the reachability by node names and the second five path expressions test the reachability by node identifiers. The last two columns explain the execution time for each path expressions using the HID and the HOPI indexes. The table shows that about "30%" of the reachability queries can be determined in a linear time without needing to use the HID index. It is observed that the HID index provides significantly better performance than the HOPI index. The HID index is more than an order of magnitude better than the HOPI index when testing reachability queries.

It is also observed that the execution time needed to test the reachability between two nodes depends on the distance between these nodes. For example, the time needed to execute the path query "200//12100" is 3.1 seconds, whereas, the time needed to execute the path query "8//25" is 0.01 seconds.

Find all descendants /ancestors

There are two possibilities to evaluate all the descendants/ancestors for a given node. First with it's name, and second with it's identifier. As the reachability between nodes is tested, ten queries are submitted to the database (that is described in section in table 7.7). In this experiment, all descendants for a given node are evaluated and similarly all ancestors can be evaluated. Table 7.11 shows the ten path expressions that are submitted to the database. The first five path expressions evaluate all the descendants for a given node by names and the second five path expressions evaluate all the descendants for a given node by identifiers. It is noted from table 7.11 that queries (Q1, Q2, and Q5) are without any additional conditions and the other queries are with additional conditions. These conditions have a direct effect on the number of results returned by the underlying query.

Query	Query in English	Path Expression	HID index		HOPI index	
			Time(s)	#Result	time (s)	#Result
Q1	Find all the descendents of a movie	movie//*	0.54	408	1.2	1064
Q2	Find all the descendents of actor	actor//*	0.61	651	1.4	2011
Q3	Find all the descendants of title with its name	title[“Animal Instinct”] //*	0.2	1	0.3	1
Q4	Find all the descendants of a director by its name	director [”Zemeckis”]//*	0.2	1		1
Q5	Find all the descendants of casting node	casting //*	0.45	209	1.1	879
Q6	Find all the descendants of identifier 8	8//*	0.1	1	0.1	1
Q7	Find all the descendants of identifier 100	100//*	0.11	1	0.16	1
Q8	Find all the descendants of identifier 300	300//*	0.2	1	0.28	1
Q9	Find all the descendants of identifier 1000	1000//*	0.2	1	0.2	1
Q10	Find all the descendants of identifier 4000	4000//*	0.31	1	0.4	1

Table 7.11: Execution time and the number of results to evaluate the descendants-axis

Table 7.11 shows that there is a difference between the execution time for path queries which have additional conditions and for path queries without additional conditions. For path queries that have additional conditions the execution time for the HID index and the HOPI index is nearly the same. However, for path queries without additional conditions the table shows that the HID index performs two or three orders of magnitude better than the HOPI index. For the HOPI index, we find more duplicate results (see the results column in table 7.11) for these queries. As mentioned before the duplicate results arise because the underlying data storage has many cycles.

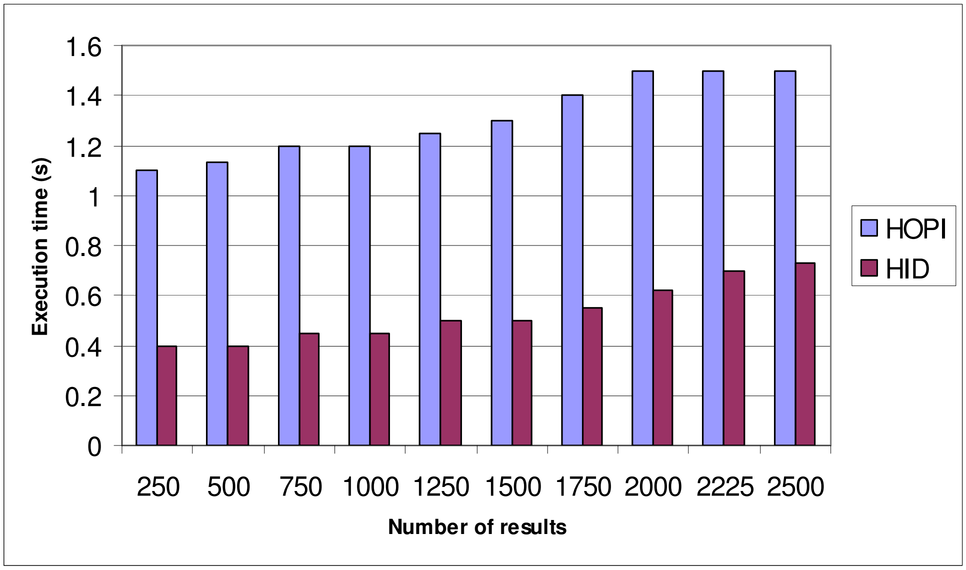


Figure 7.7: Time to compute all the descendants for a given node

To study the relationship between the number of results and the execution time the query "find all the descendants for actor" is performed. This query shows that there are "2011" nodes, which have a name actor. For each node *actor* all of its descendants are evaluated and the results are arranged according to the number of descendant's for each node. Figure 7.7 shows that the HID index performs two or three orders of magnitude better than the HOPI index and avoids the duplicated results..

7.4 Discussion of the Experimental Results

The experimental results discussed in this chapter show that the HID index minimizes space and time required to build the index in comparison to other indexes (e.g., the HOPI index). However, for the HOPI index, the space was dominated by the space needed for storing the TC of the original XML graph. So the size of the HOPI index is huge if the documents size is increased. The presented optimization techniques, which are used to build the HID index, deal with this problem properly and avoid the memory space-consuming problem. These optimization techniques help to avoid the time consuming problem and the query processing costs as compared to the HOPI index.

The experimental results show that the HID index can deal very efficiently with a highly interlinked XML collection, which has many cycles (e.g., Internet Movie Database). However, this data stresses any path index. It is not known, if any index structure proposed in the literature can deal with this highly connected data.

For path queries, all descendants/ancestors for a given node can easily be evaluated by only looking at the HID index tables using an SQL statement. The HID index avoids the duplicating results problem that is present in the HOPI index while evaluating such a query. The experimental results show that the HID index saves time to execute the descendants/ancestors-or-self query in comparison to the HOPI index. For testing the reachability between nodes, experimental results show that about 30% of these reachability queries are evaluated in a linear time without needing to use any index (e.g., only using the SCC table). It is also observed that the HID index achieves a higher evolution performance than the HOPI index. Therefore, these experiments show that the HID index performs two or three orders of magnitude better than the HOPI index and overcomes the problem of cycles in the graph that can stress every path index.

8

Conclusion and Future Work

This chapter concludes the thesis by summarizing the contributions (Section 8.1) and identifying directions for possible future work (Section 8.2).

8.1 Contributions

This thesis presents the HID index. This index supports efficient reachability tests as well as evaluations of the path queries along the descendant's axis on large and highly interlinked XML document collections with cycles. This is a task that most of today's existing index structures fail to achieve. The cycles in the graph can stress every path index, or may even prevent the evaluation of path queries. The HID index can efficiently evaluate different types of path queries. Firstly, the *ancestors-descendants* queries on large XML graphs with long paths. Secondly, the HID index is optimized to evaluate path queries of type *descendant/ancestor-or-self relationship* (with wildcard "//") on large XML graphs, a task that was ignored by all existing path indexes.

The HID index leverages the existing concept of a two-hop cover for a directed graph for highly efficient indexing of XML document collections. The main technique is to use strongly connected components for scalable index building. The strongly connected components are used to map the large original XML graph onto another graph (e.g., DAG) with a minimum number of nodes and edges, and to deal with possible large cycles that appear in the presence of IDs and IDREFs, when the path queries involve them.

The HID index is implemented as a database-backed index structure. Thus, all the information about the index is stored in database tables. That makes it possible to use SQL queries to evaluate path expressions without needing to traverse the complete XML graph.

The performed experiments with real life data (e.g., movie database as a highly *cyclic* database) show that the HID index can represent connections in highly interlinked XML documents very efficiently and can overcome the time-consumption problem as compared to other indexes. Moreover, it significantly improves query-processing costs and space requirements in comparison with the previous index structure. It is also noteworthy that about 30% of the reachability queries can be evaluated in linear time based on the SCC table without help of the HID index.

However, the HID index has one disadvantage. It can only deal efficiently with XML data graphs that have cycles like the underlying application about movie databases. Otherwise, if the XML documents' graph has no cycles (e.g., large tree), other approaches prove to be more efficient than the HID index in dealing with this data.

8.2 Possibilities for Future Work

There are several enhancements, which can improve the performance and the efficiency of the HID index.

- It is planed to apply additional algorithms to the HID index to handle all path expression axes.
- It is planed to deal efficiently with the updating problem.
- To further extend the HID index, it is planed to add additional information retrieval methods, i.e., "distance information" and "relevance feedback" to matching documents. Here, the system should not materialize all results (including the marginally ones), but compute only the most relevant results (namely rank-aware query evaluation).

- For heterogeneous XML documents in the Web (divided XML documents into several subcollections), a single index structure may not be appropriate. Therefore, it will be investigated whether it makes sense to combine several indexes as building blocks. This would allow for building an index for each subcollection and evaluating the proposed queries by “navigating” through the underlying subcollection only.

Appendix A

A small Example of Movies Database

The XML file below describes a very small example about movies database. Note we cannot show here the complete files, they are very large.

```
<? xml version="1.0" standalone="yes" ?>
<movies>
 <film id="H1">
<title>Always Tell Your Wife</title>
<year>1922</year>
<dname>Se.Hicks</dname>
<prods>Lasky</prods>
<studio>Famous, SD:*</studio>
<prc>sbw</prc>
<type>Dram</type>
<award>aw</award>
<lc />
<notes>CoD(Hitchcock)</notes>
</film>
 <film id="H2">
<title>Number Thirteen</title>
<year>1922</year>
<dname>Hitchcock</dname>
<prods>Hitchcock</prods>
<studio>Islington, SD:Famous</studio>
<prc>sbw</prc>
<type />
<award />
<lc />
<notes>Nt(unfinished)</notes>
</film>
<film id="H3">
<title>Woman to Woman</title>
<year>1922</year>
<dname>Hitchcock</dname>
<prods>Balcon</prods>
<studio>B-S-F, SD:Wardour</studio>
<prc>sbw</prc>
<type>Dram</type>
<award />
<lc>England</lc>
<notes>CoD(Cutts) Er(same as GCt27, 1926?)</notes>
</film>
<film id="H4">
<title>The Passionate Adventure</title>
<year>1924</year>
<dname>Hitchcock</dname>
<prods>Balcon</prods>
<studio>Gainsborough, SD:GaumontD</studio>
<prc>sbw</prc>
<type />
```

```
<award />
 <lc />
 <notes>CoD(Cutts)</notes>
 </film>
 <film id="H5">
 <title>The Blackguard</title>
 <year>1925</year>
 <dname>Hitchcock</dname>
 <prods>Balcon</prods>
 <studio>UFA, SD:Wardour</studio>
 <prc>sbw</prc>
 <type />
 <award />
 <lc />
 <notes>CoD(Cutts)</notes>
 </film>
 <film id="H7">
 <title>The Pleasure Garden</title>
 <year>1925</year>
 <dname>Hitchcock</dname>
 <prods>Balcon</prods>
 <studio>Gainsborough and Emelka, SD:Wardour</studio>
 <prc>sbw</prc>
 <type>Dram</type>
 <award>H</award>
 <lc>Pleasure Garden, theater</lc>
 <notes>Nt(released 1927) C(Ventimiglia) B(Oliver Sandys) W(Eliot Stannard)</notes>
 </film>
 <film id="H8">
 <title>The Mountain Eagle</title>
 <year>1926</year>
 <dname>Hitchcock</dname>
 <prods>Balcon</prods>
 <studio>Gainsborough, Emelka, SD:Wardour</studio>
 <prc>sbw</prc>
 <type>Dram</type>
 <award>H</award>
 <lc />
 <notes>Nt(no prints left) Nt(Released 1927) W(Eliot Stannard) C(Baron Ventimiglia) Alt(T:Fear O'God; USA)</notes>
 </film>
 <film id="H9">
 <title>The Lodger: A Story of The London Fog</title>
 <year>1926</year>
 <dname>Hitchcock</dname>
 <prods>Balcon</prods>
 <studio>Gainsborough, SD:(Wardour</studio>
 <prc>sbw</prc>
 <type>Susp</type>
 <award />
 <lc>London, England</lc>
 <notes>B(Mary Belloc Lowndes: The Lodger) Nt(Jack the Ripper) W(Hitchcock, Eliot Stannard) Cost(12K lbs) Nt(released 1927)
 C(Baron Ventimiglia) Alt(T:The Case of Jonathan Drew; USA)</notes>
 </film>
 <film id="H10">
 <title>Downhill</title>
 <year>1927</year>
 <dname>Hitchcock</dname>
 <prods>Balcon</prods>
 <studio>Islington; Gainsborough, SD:Wardour</studio>
 <prc>sbw</prc>
 <type>Susp</type>
 <award />
 <lc />
<notes>W(Eliot Stannard) W(Ivor Novello, Constance Collier) C(Baron Ventimiglia) Er(W))</notes>
</film>
<film id="H11">
<title>Easy Virtue</title>
<year>1927</year>
<dname>Hitchcock</dname>
<prods>Balcon</prods>
<studio>Gainsborough, SD:Wardour</studio>
<prc>sbw</prc>
<type>Susp</type>
<award />
<lc />
<notes>B(Noel Coward) W(Eliot Stannard) C(Claude McDonnell)</notes>
</film>
<film id="H12">
<title>The Ring</title>
<year>1927</year>
<dname>Hitchcock</dname>
<prods>J.Maxwell</prods>
<studio>BIP Elstree, SD:Wardour</studio>
<prc>sbw</prc>
<type>Susp</type>
<award />
<lc>theater, England</lc>
<notes>W(Hitchcock, Alma Reveille) C(Claude McDonnell)</notes>
</film>
<film id="H13">
<title>The Farmer's Wife</title>
<year>1928</year>
<dname>Hitchcock</dname>
<prods>J.Maxwell</prods>
<studio>BIP Elstree, SD:Wardour</studio>
<prc>sbw</prc>
<type>Susp</type>
<award />
<lc />
<notes>B(Eden Philpotts) W(Hitchcock) C(J.Cox)</notes>
</film>
<film id="H14">
<title>Champagne</title>
<year>1928</year>
<dname>Hitchcock</dname>
<prods>J.Maxwell</prods>
<studio>BIP Elstree, SD:Wardour</studio>
<prc>sbw</prc>
<type>Romt</type>
<award />
<lc />
<notes>W(Eliot Stannard) B(Walter Mycroft) C(J.Cox)</notes>
</film>
<film id="H15">
<title>Harmony Heaven</title>
<year>1929</year>
<dname>Hitchcock</dname>
<prods>N:</prods>
<studio>BIP, SD:France-Societ\'e des Cin\'e-Romans</studio>
<prc>col</prc>
<type>Susp</type>
<award>H</award>
<lc />
<notes>CoD(Thomas Bentley) W(Arthur Wimperis, Randall Faye) M(Eddie Pola, Edward Brandt) C(J.Cox)</notes>
</film>
<film id="H16">
```

```
  <title>The Manxman</title>
  <year>1929</year>
  <dname>Hitchcock</dname>
  <prods>J.Maxwell</prods>
  <studio>BIP Elstree, SD:Wardour</studio>
  <prc>sbw</prc>
  <type>Susp</type>
  <award>H</award>
  <lc>csd, Scotland</lc>
  <notes>B(Hall Caine) W(Eliot Stannard) C(J.Cox)</notes>
  </film>
 <film id="H17">
  <title>Blackmail</title>
  <year>1929</year>
  <dname>Hitchcock</dname>
  <prods>J.Maxwell</prods>
  <studio>BIP Elstree, SD:Wardour</studio>
  <prc>bnw</prc>
  <type>Susp</type>
  <award />
  <lc>British Museum, London, England</lc>
  <notes>Nt(first sound, added) W(Hitchcock, Benn W. Levy, Charles Bennett) C(J.Cox) Seen(9Apr1990)</notes>
  </film>
 <film id="H18">
  <title>Elstree Calling</title>
  <year>1929</year>
  <dname>none</dname>
  <prods>N:</prods>
  <studio>BIP Elstree</studio>
  <prc>bnw</prc>
  <type>Musc</type>
  <award />
  <lc>studio</lc>
  <notes>CoD(Brunel(supervisor), Hitchcock, Andr\'e Charlot, Jack Hulbert, Paul Murray) W(Ivor Novello)</notes>
  </film>
<film id="H19">
  <title>Juno And The Paycock</title>
  <year>1930</year>
  <dname>Hitchcock</dname>
  <prods>J.Maxwell</prods>
  <studio>BIP Elstree, SD:Wardour</studio>
  <prc>bnw</prc>
  <type>Susp</type>
  <award>H*</award>
  <lc>Dublin, Ireland</lc>
  <notes>B(Sean O'Casey) W(Hitchcock, Alma Reville) C(J.Cox)</notes>
  </film>
 <film id="H20">
  <title>Murder!</title>
  <year>1930</year>
  <dname>Hitchcock</dname>
  <prods>J.Maxwell, Alfred Abel: German version</prods>
  <studio>BIP Elstree, SD:Wardour</studio>
  <prc>bnw</prc>
  <type>Myst</type>
  <award>H**</award>
  <lc>theater, London, England</lc>
  <notes>W(Hitchcock and Alma Reville) B(Clarence Dane, Helen Simpson: Enter Sir John) V(Winifred Ashton) C(J.Cox)
   Seen(10Apr1990) Er(V)</notes>
  </film>
 <film id="H21">
  <title>The Skin Game</title>
  <year>1931</year>
```

```
<dname>Hitchcock</dname>
<prods>J.Maxwell</prods>
<studio>BIP Elstree, SD:Wardour</studio>
<prc>bnw</prc>
<type>Dram</type>
<award>aw</award>
<lc />
<notes>B(John Galsworthy) W(Hitchcock, Alma Reville) C(J.Cox, Charles Martin) Er(C)</notes>
</film>
<film id="H22">
<title>Rich and Strange</title>
<year>1932</year>
<dname>Hitchcock</dname>
<prods>J.Maxwell</prods>
<studio>BIP Elstree, SD:Wardour</studio>
<prc>bnw</prc>
<type>Susp</type>
<award />
<lc>London, England; Pars, France; Marseille, France; Singapore; ship, Pacific</lc>
<notes>W(Alma Reville, Val Valentine) B(Dale Collins) C(J.Cox, Charles Martin) Er(C) Seen(16Apr1990) Alt(T:East of Shanghai; USA)</notes>
</film>
<film id="H23">
<title>Number Seventeen</title>
<year>1932</year>
<dname>Hitchcock</dname>
<prods>J.Maxwell</prods>
<studio>BIP Elstree, SD:Wardour</studio>
<prc>bnw</prc>
<type>Myst</type>
<award>H*</award>
<lc>London, England; train, England; Dover, England</lc>
<notes>B(Jefferson Farejon) W(Jefferson Farejon, Rodney Auckland) C(J.Cox) Seen(16Apr1990) Er(W)</notes>
</film>
<film id="H24">
<title>Lord Camber's Ladies</title>
<year>1932</year>
<dname>Hitchcok</dname>
<prods>Hitchcock</prods>
<studio>BIP Elstree, SD:Wardour</studio>
<prc>bnw</prc>
<type>Susp</type>
<award />
<lc>castle, England</lc>
<notes>W(Hitchcock,,Jefferson Farejon) B(H.A. Vachell: The Case of Lady Camber) CoD(Benn~Levy)</notes>
</film>
<film id="H25">
<title>Waltzes From Vienna</title>
<year>1933</year>
<dname>Hitchcock</dname>
<prods>N:Tom Arnold</prods>
<studio>Lime Grove, SD:G.F.D.</studio>
<prc>bnw</prc>
<type>Romt,Comd</type>
<award>H</award>
<lc>Vienna, Austria</lc>
<notes>Nt(least favorite of Hitchcock) M{Johann Strauss~sr., Johann Strauss~jr. W(Alma Reveille, Bolton) Alt(T:Strauss's Great Waltz; USA)</notes>
</film>
<film id="H26">
<title>The Man Who Knew Too Much</title>
<year>1934</year>
<dname>Hitchcock</dname>
```

```
<prods>Balcon, Montagu</prods>
<studio>Gaumont Lime Grove, SD:G.F.D.</studio>
<prc>bnw</prc>
<type>Susp</type>
<award>H***</award>
<lc>St.Moritz, Switzerland; London, England</lc>
<notes>B(D.B. Wyndham-Lewis, Edwin Greenwood) W(A.R.Rawlinson, Charles Bennett, D.B. Wyndham-Lewis, Edwin
 Greenwood) M(Arthur Benjamin, Louis~Levy) C(Curt Courant)</notes>
</film>
 <film id="H27">
<title>The 39 Steps</title>
<year>1935</year>
<dname>Hitchcock</dname>
<prods>Balcon, Montagu</prods>
<studio>Gaumont Lime Grove, SD:G.F.D.</studio>
<prc>bnw</prc>
<type>Susp</type>
<award>H****</award>
<lc>theater, London, England</lc>
<notes>W(Charles Bennett, Alma Reveille) M(Louis Levy) B(John Buchan) C(Bernard Knowles)</notes>
</film>
<film id="H28">
<title>Secret Agent</title>
<year>1936</year>
<dname>Hitchcock</dname>
<prods>Balcon, Montagu</prods>
<studio>Gaumont Lime Grove, SD:G.F.D.</studio>
<prc>bnw</prc>
<type>Susp</type>
<award />
<lc>London, England; chocolate afctory, Switzerland</lc>
<notes>B(S.Maugham: Ashenden) W(Charles Bennett, Campbell Dixon) C(Bernard Knowles) Er(W)</notes>
</film>
<film id="H29">
<title>Sabotage</title>
<year>1936</year>
<dname>Hitchcock</dname>
<prods>Balcon, Montague</prods>
<studio>Lime Grove, SD:G.F.D.</studio>
<prc>bnw</prc>
<type>Susp</type>
<award>H***</award>
<lc>theater, London, England</lc>
<notes>B(Conrad: The Secret Agent) W(Charles Bennett) C(Bernard Knowles) Alt(T:The Woman Alone; USA)</notes>
</film>
<film id="H30">
<title>Young and Innocent</title>
<year>1938</year>
<dname>Hitchcock</dname>
<prods>Black</prods>
<studio>Lime Grove and Pinewood, SD:G.F.D.</studio>
<prc>bnw</prc>
<type>Susp</type>
<award />
<lc />
<notes>B(Josephine Tey) C(Bernard Knowles) W(Charles Bennett, Alma Reveille) Alt(T:The Girl Was Young; USA)</notes>
</film>
<film id="H31">
<title>The Lady Vanishes</title>
<year>1938</year>
<dname>Hitchcock</dname>
<prods>Black</prods>
<studio>Lime Grove Gainsborough, SD:MGM</studio>
```

```
<prc>bnw</prc>
<type>Susp</type>
<award />
<lc>train; Austria</lc>
<notes>B(Ethel Lina White: `The Wheel Spins') C(J.Cox) W(Gilliat, Launder) Seen(11Jul1988) W(Charles Bennett)</notes>
</film>
<film id="H32">
<title>Jamaica Inn</title>
<year>1939</year>
<dname>Hitchcock</dname>
<prods>Pommer, Laughton</prods>
<studio>Elstree, SD:Associated British, Paramount</studio>
<prc>bnw</prc>
<type>Dram</type>
<award>W50</award>
<lc>seashore, Cornwall, England</lc>
<notes>C(Bernard Knowles, Harry Stradling) Er(C)</notes>
</film>
<film id="H33">
<title>Rebecca</title>
<year>1940</year>
<dname>Hitchcock</dname>
<prods>Selznick</prods>
<studio>Selznick, SD:U.A.</studio>
<prc>bnw</prc>
<type>Dram</type>
<award>AA, AAN dir, H****</award>
<lc>castle, England</lc>
<notes>C(George~Barnes; AA) W(Robert E. Sherwood, J.Harrison; AAN) B(Daphne duMaurier) M(Waxman; AAN) Seen(1989)
VT(N:HT5) V(Lyle~Wheeler)</notes>
</film>
<film id="H34">
<title>Foreign Correspondent</title>
<year>1940</year>
<dname>Hitchcock</dname>
<prods>Wanger</prods>
<studio>U.A.</studio>
<prc>bnw</prc>
<type>Susp</type>
<award>H****, AAN</award>
<lc>London, England; flying boat, Atlantic; T(1940)</lc>
<notes>W(Charles Bennett, J.Harrison, James Hilton, Robert Benchley; AAN) B(Vincent Sheehan: `Personal History') C(Mate;
 AAN) Seen(3Jan1991) VT(N:HT1)</notes>
</film>
<film id="H35">
<title>Mr.~and Mrs.~Smith</title>
<year>1941</year>
<dname>Hitchcock</dname>
<prods>Eddington</prods>
<studio>RKO</studio>
<prc>bnw</prc>
<type>Romt</type>
<award />
<lc>NYC, NY</lc>
<notes>Seen(23Oct1988) VT(N:HT4) C(Harry Stradling) V(Polglase?)</notes>
 </film>
<film id="H36">
<title>Suspicion</title>
<year>1941</year>
<dname>Hitchcock</dname>
<prods>Raphaelson, Hitchcock</prods>
<studio>RKO</studio>
<prc>bnw,cld</prc>
```

```
<type>Susp</type>
<award />
<lc>mansion, England</lc>
<notes>C(Harry Stradling) B(Francis Iles) Prof(440K)</notes>
</film>
<film id="H37">
<title>Saboteur</title>
<year>1942</year>
<dname>Hitchcock</dname>
<prods>F.Lloyd, Skirball</prods>
<studio>Universal</studio>
<prc>bnw</prc>
<type>Susp</type>
<award>H***</award>
<lc>national monuments, WY; statue of liberty, NY</lc>
<notes>Seen(20Sep1989) C(Valentine) VT(N:HT2)</notes>
</film>
<film id="H38">
<title>Shadow of a Doubt</title>
<year>1943</year>
<dname>Hitchcock</dname>
<prods>Skirball</prods>
<studio>Universal</studio>
<prc>bnw</prc>
<type>Susp</type>
<award />
<lc>San Rafael, CA; T(1938)</lc>
<notes>C(Valentine) W(Thornton Wilder) Seen(10Jul1991) M(Tiomkin, Previn, Strauss: Merry Widow Waltz) Nt(two of
everything [Rohmer])</notes>
</film>
<film id="H39">
<title>Lifeboat</title>
<year>1943</year>
<dname>Hitchcock</dname>
<prods>MacGowan</prods>
<studio>Fox</studio>
<prc>bnw</prc>
<type>Susp</type>
<award>H**, AAN dir</award>
<lc>sea</lc>
<notes>B(John Steinbeck; AAN) W(John Steinbeck, Jo Swerling) C(Glen MacWilliams; AAN) Seen(18Sep1989)
VT(N:HT4)</notes>
</film>
<film id="H42">
<title>Spellbound</title>
<year>1945</year>
<dname>Hitchcock</dname>
<prods>Selznick</prods>
<studio>Selznick, SD:U.A.</studio>
<prc>bnw</prc>
<type>Susp</type>
<award>H**, AAN, AAN dir</award>
<lc>mental hospital, Vermont</lc>
<notes>M(Rozsa; AA) C(George~Barnes; AAN) W(Hecht) V(Salvador Dali) B(Francis Beeding: ``The House of Dr.Edwardees")
 Seen(3Dec1989), 15May90</notes>
</film>
<film id="H43">
<title>Notorious</title>
<year>1946</year>
<dname>Hitchcock</dname>
<prods>Hitchcock</prods>
<studio>RKO</studio>
<prc>bnw</prc>
```

```xml
<type>Susp</type>
<award>H***</award>
<lc>RioDe Janeiro, Brazil</lc>
<notes>W(Ben Hecht; AAN) C(Tetzlaff) M(Roy Webb) Seen(20Jan1990, 30Jun1997)</notes>
</film>
<film id="H44">
<title>The Paradine Case</title>
<year>1947</year>
<dname>Hitchcock</dname>
<prods>Selznick</prods>
<studio>Selznick, SD:U.A.</studio>
<prc>bnw</prc>
<type>Susp</type>
<award>H**</award>
<lc>London, England B(Robert Hichens)</lc>
<notes>W(Selznick) C(Lee Garmes) M(Waxman)</notes>
</film>
<film id="H45">
<title>Rope</title>
<year>1948</year>
<dname>Hitchcock</dname>
<prods>Bernstein, Hitchcock</prods>
<studio>Transatlantic, SD:Warners</studio>
<prc>col=Tcol</prc>
<type>Susp</type>
<award>H**</award>
<lc>penthouse, NYC, NY</lc>
<notes>M(Francis Poulenc: Mouvement Perpetuel) Nt(first color Hitchcock) Nt(all shot in long, 10 minute takes) C(William V.
Skall, Valentine) Seen(25Nov1992)</notes>
</film>
<film id="H46">
<title>Under Capricorn</title>
<year>1949</year>
<dname>Hitchcock</dname>
<prods>Bernstein, Hitchcock</prods>
<studio>Transatlantic, MGM British, SD:Warners</studio>
<prc>col=Tcol</prc>
<type>Dram</type>
<award />
<lc>Sidney, Australia</lc>
<notes>W(James Bridie, Hume Cronyn) B(Helen Simpson) C(Jack Cardiff, Ian Craig, David McNeilly)
Seen(15May1990)</notes>
</film>
<film id="H47">
<title>Stage Fright</title>
<year>1950</year>
<dname>Hitchcock</dname>
<prods>Hitchcock, Ahern</prods>
<studio>Elstree, SD:Warners, SL:(\Ge, \GB)</studio>
<prc />
<type>Susp</type>
<award />
<lc>theater, London, England</lc>
<notes>Nt(filmed in Eng,Ger) C(Wilkie Cooper) Alt(T:Die rote Lola; \Ge)</notes>
</film>
<film id="H48">
<title>Strangers on a Train</title>
<year>1951</year>
<dname>Hitchcock</dname>
<prods>Hitchcock</prods>
<studio>Warners</studio>
<prc>bnw</prc>
<type>Susp</type>
```

```
  <award>H***</award>
  <lc>Washington, DC; NYC, NY; train; merry-go-round; csd</lc>
  <notes>Seen(10Jun1989, 2Mar1998) C(Burks; AAN) M(Tiomkin) VT(N:HT8) Sy(crossings) B(Patricia Highsmith)</notes>
  </film>
 <film id="H49">
  <title>I Confess</title>
  <year>1952</year>
  <dname>Hitchcock</dname>
  <prods>Hitchcock</prods>
  <studio>Warners</studio>
  <prc>bnw</prc>
  <type>Susp</type>
  <award />
  <lc>theater, Quebec City, Canada</lc>
  <notes>M(:Dies Irae) C(Burks)</notes>
  </film>
 <film id="H50">
  <title>Dial M for Murder</title>
  <year>1954</year>
  <dname>Hitchcock</dname>
  <prods>Hitchcock</prods>
  <studio>Warners</studio>
  <prc>\Wcol 3D</prc>
  <type>Susp</type>
  <award>H**</award>
  <lc>London, England</lc>
  <notes>M(Tiomkin) C(Burks) Seen(3Dec1989)</notes>
  </film>
 <film id="H51">
  <title>Rear Window</title>
  <year>1954</year>
  <dname>Hitchcock</dname>
  <prods>Hitchcock</prods>
  <studio>Paramount</studio>
  <prc>col=Tcol</prc>
  <type>Susp</type>
  <award>H***, AAN dir</award>
  <lc>town, East</lc>
  <notes>B(Woolrich) C(Burks; AAN) W(J.M.Hayes; AAN) VT(N:HC3; HT8, inc)</notes>
  </film>
 <film id="H52">
  <title>To Catch a Thief</title>
  <year>1955</year>
  <dname>Hitchcock</dname>
  <prods>Hitchcock</prods>
  <studio>Paramount</studio>
  <prc>col=Tcol</prc>
  <type>Susp</type>
  <award>aw</award>
  <lc>Riviera, France</lc>
  <notes>W(J.M.Hayes) C(Burks) B(David Dodge)</notes>
  </film>
 <film id="H53">
  <title>The Trouble with Harry</title>
  <year>1956</year>
  <dname>Hitchcock</dname>
  <prods>Hitchcock</prods>
  <studio>Paramount</studio>
  <prc>col=Tcol</prc>
  <type>Susp</type>
  <award />
  <lc>csd, Vt</lc>
  <notes>Seen(1957), 1989 C(Burks) W(J.M.Hayes)</notes>
 </film>
<film id="H57">
 <title>The Man Who Knew Too Much</title>
 <year>1956</year>
 <dname>Hitchcock</dname>
 <prods>Hitchcock</prods>
 <studio>Paramount</studio>
 <prc>col=Tcol</prc>
 <type>Susp</type>
 <award>H*</award>
 <lc>Indianapolis, IN; Morocco; Albert Hall, London</lc>
 <notes>VT(N:HT3) C(Burks) M(Bernard Herrman; Jay Livingston, Ray Evans; AA)</notes>
 </film>
<film id="H61">
 <title>The Wrong Man</title>
 <year>1957</year>
 <dname>Hitchcock</dname>
 <prods>Hitchcock</prods>
 <studio>Warners</studio>
 <prc>bnw</prc>
 <type>Susp,Docu</type>
 <award />
 <lc>USA</lc>
 <notes>C(Burks) W(John Michael Hayes, Angus McPhail) B(Charles Bennett, D.R. WyndhamLewis)</notes>
 </film>
<film id="H65">
 <title>Vertigo</title>
 <year>1958</year>
 <dname>Hitchcock</dname>
 <prods>Hitchcock, Robert Coleman</prods>
 <studio>Paramount</studio>
 <prc>\Tcol, \Vistavision</prc>
 <type>Susp</type>
 <award>H**</award>
 <lc>Nob Hill, S.F., CA; restaurant, Ernie's, S.F., CA; SanJuanBattista, CA</lc>
 <notes>M(Herrmann; "Liebestod") C(Burks) V(Henry Bumstead, Saul Bass "titles" ) VT(N:HC1)</notes>
 </film>
<film id="H69">
 <title>North by Northwest</title>
 <year>1959</year>
 <dname>Hitchcock</dname>
 <prods>Hitchcock</prods>
 <studio>MGM</studio>
 <prc>col=Tcol</prc>
 <type>Susp</type>
 <award>H****</award>
 <lc>train, Chicago, IL; plane (Northwest Airlines), US; prarie, IN; national monument, MT</lc>
 <notes>W(Ernest Lehman; AAN) C(Burks)</notes>
 </film>
<film id="H73">
 <title>Psycho</title>
 <year>1960</year>
 <dname>Hitchcock</dname>
 <prods>Hitchcock</prods>
 <studio>Shamley, Universal, SD:Paramount</studio>
 <prc>bnw</prc>
 <type>Susp</type>
 <award>AAN Dir, NFR, Z*</award>
 <lc>Tucson, AZ; Victorian house, csd</lc>
 <notes>M(Herrmann) R(PG) W(Joseph Stefano) B(Robert~Bloch) C(John L. Russell; AAN) V(Joseph Hurley, Robert Clatworthy;
 AAN) M(Beeethoven: Eroica) Cost(.8M) Inc(12M) VT(N:HT9, HC2;HT9)</notes>
 </film>
<film id="H79">
```

```
   <title>The Birds</title>
   <year>1963</year>
   <dname>Hitchcock</dname>
   <prods>Hitchcock</prods>
   <studio>Universal</studio>
   <prc>col=Tcol</prc>
   <type>Susp</type>
   <award>H***</award>
   <lc>bird shop, S.F., CA; Bodega Bay, CA</lc>
   <notes>W(Evan Hunter) B(Daphne duMaurier) C(Burks) M(Herrmann; sounds only) VT(N:HT5)</notes>
   </film>
  <film id="H80">
   <title>Marnie</title>
   <year>1964</year>
   <dname>Hitchcock</dname>
   <prods>Hitchcock</prods>
   <studio>Universal</studio>
   <prc>col</prc>
   <type>Susp</type>
   <lc>Philadelphia, PA</lc>
   <notes>B(Winston Graham)Seen(1Sep1988),23Jan89 M(Herrmann) W(J.P. Allen) C(Burks) VT(N:HT9)</notes>
 </film>
</movies>
```

Appendix B

Examples of Linked XML Documents from Movies Database

```
  <?xml version="1.0" encoding="ISO-8859-1" ?>
 <movie id="2">
 <title>#1 Fan: A Darkomentary</title>
 <production_year>2005</production_year>
 <type>Video</type>
 <production_country>USA</production_country>
 <production_language>English</production_language>
 <production_location>Los Angeles, California, USA</production_location>
 <production_location>San Diego, California, USA</production_location>
 <director xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/425/318.xml">Robertson, Dee
 Austin</director>
 <producer xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/1074/596.xml">
 <name>Mansfield, Raymond</name>
 <job>producer</job>
 </producer>
<cast order="credits">
 <casting>
 <actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/41/178.xml">Berger,
 Todd</actor>
 <role>Darryl Donaldson</role>
 </casting>
 <casting>
 <actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/140/570.xml">Duval,
 James</actor>
 <role>Himself</role>
 </casting>
 <casting>
 <actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/257/713.xml">Kelly, Richard
(II)</actor>
 <role>Himself</role>
 </casting>
<casting>
 <actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/330/175.xml">McKittrick,
Sean</actor>
 <role>Himself</role>
 </casting>
<casting>
 <actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/483/293.xml">Stone,
Stuart</actor>
 <role>Himself</role>
 </casting>
 <casting>
  <actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/676/14.xml">Grant,
Beth</actor>
 <role>Herself</role>
 </casting>
 </cast>
<genres>
 <genre>Documentary</genre>
 <genre>Short</genre>
 </genres>
 <colourinfo>Color</colourinfo>
 </movie>

 <?xml version="1.0" encoding="ISO-8859-1" ?>

<movie id="4">
 <title>#2: Drops</title>
 <production_year>2004</production_year>
 <production_country>Argentina</production_country>
 <production_language>Spanish</production_language>
  <genres>
 <genre>Short</genre>
 </genres>
 <colourinfo>Color</colourinfo>
 </movie>

  <?xml version="1.0" encoding="ISO-8859-1" ?>
  <movie id="5">
 <title>#7 Train: An Immigrant Journey, The</title>
 <alternative_title>#7 Train from Main Street</alternative_title>
 <production_year>2000</production_year>
 <production_countries>
 <production_country>South Korea</production_country>
 <production_country>USA</production_country>
 </production_countries>
 <production_language>English</production_language>
 <production_location>New York City subway, Manhattan, New York City, New York, USA</production_location>
 <production_location>Queens, New York City, New York, USA</production_location>
 <director xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/943/208.xml">Park, Hye
Jung</director>
 <genres>
 <genre>Short</genre>
 <genre>Documentary</genre>
 </genres>
 <keywords>
 <keyword>korean</keyword>
 <keyword>new-york-city</keyword>
 <keyword>pakistani</keyword>
 <keyword>street-vendor</keyword>
 <keyword>subway</keyword>
 <keyword>manhattan</keyword>
 </keywords>
 <colourinfo>Color</colourinfo>
  </movie>

 <?xml version="1.0" encoding="ISO-8859-1" ?>
 <movie id="7">
 <title>$1,000 Reward</title>
 <production_year>1913</production_year>
 <production_country>USA</production_country>
<cast order="credits">
<casting>
 <actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/39/681.xml">Bennett,
Charles</actor>
 </casting>
 <casting>
 <actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/328/511.xml">McGee,
  Morris</actor>
 </casting>
<casting>
 <actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/540/185.xml">Wilbur,
Crane</actor>
```

```
 </casting>
 <casting>
<actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/676/456.xml">Gray,
Betty</actor>
 </casting>
 </cast>
 <colourinfo>Black and White</colourinfo>
 <soundmix>Silent</soundmix>
 </movie>

<?xml version="1.0" encoding="ISO-8859-1" ?>

 <movie id="8">
 <title>$1,000 Reward</title>
 <production_year>1915</production_year>
 <production_country>USA</production_country>
  <cast order="credits">
  <casting>
  <actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/652/571.xml">Fairbanks,
  Madeline</actor>
 </casting>
 <casting>
 <actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/652/572.xml">Fairbanks,
  Marion</actor>
 </casting>
 <casting>
 <actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/687/159.xml">Hastings, Carey
 L.</actor>
 </casting>
 <casting>
 <actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/585/77.xml">Berlin,
Minnie</actor>
 </casting>
 </cast>
 <colourinfo>Black and White</colourinfo>
 <soundmix>Silent</soundmix>
  </movie>

    <?xml version="1.0" encoding="ISO-8859-1" ?>
<movie id="9">
 <title>$1,000 Reward</title>
 <production_year>1923</production_year>
 <production_country>USA</production_country>
 <director xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/951/661.xml">Seeling,
Charles R.</director>
 <cast order="credits">
 <casting>
 <actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/541/652.xml">Williams, Guinn
'Big Boy'</actor>
 </casting>
 </cast>
<genres>
 <genre>Western</genre>
 </genres>
 <colourinfo>Black and White</colourinfo>
 <soundmix>Silent</soundmix>
 </movie>

<?xml version="1.0" encoding="ISO-8859-1" ?>
<movie id="12">
 <title>$100 & a T-Shirt: A Documentary About Zines in the Northwest</title>
 <production_year>2004</production_year>
 <production_country>USA</production_country>
 <production_language>English</production_language>
 <production_location>Portland, Oregon, USA</production_location>
```

```
  <production_location>Seattle, Washington, USA</production_location>
  <director xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/903/604.xml">Biel,
 Joe</director>
 <producer xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/903/604.xml">
  <name>Biel, Joe</name>
  <job>producer</job>
  </producer>
  <plot author="Joe Biel">A cultural analysis of what causes zine makers to tick; what the hell zines are, why people make zines, the
 origin of zines, the resources and community available for zine makers, and the future of zines. Interviews with about 70 zine makers,
 ex-zine makers, and readers from the northwest. Featuring footage of the Portland Zine Symposium, other zine related events, and
 activities bringing zine culture to life. An original documentary with over 64 hours of footage for people with a new interest in zines
 as well as pros and novices. The video sparks untapped creativity and new interest into zine making and reading.</plot>
 <genres>
  <genre>Documentary</genre>
  </genres>
  <colourinfo>Color</colourinfo>
  </movie>
<?xml version="1.0" encoding="ISO-8859-1" ?>
  <movie id="13">
  <title>$100,000</title>
  <production_year>1915</production_year>
  <production_country>USA</production_country>
  <production_language>English</production_language>
  <director xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/297/867.xml">Lloyd, Frank
 (I)</director>
 <cast order="credits">
 <casting position="1">
  <actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/297/867.xml">Lloyd, Frank
 (I)</actor>

<role>The Bank Secretary</role>
  </casting>
 <casting position="2">
  <actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/735/993.xml">Leslie,
 Helen</actor>
  </casting>
 <casting position="3">
  <actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/561/246.xml">Adams,
 Mildred</actor>
  </casting>
  </cast>
 <genres>
  <genre>Drama</genre>
  </genres>
  <colourinfo>Black and White</colourinfo>
  <soundmix>Silent</soundmix>
  </movie>

<?xml version="1.0" encoding="ISO-8859-1" ?>
 <movie id="14">
  <title>$100,000 Pyramid, The</title>
  <production_year>2001</production_year>
  <type>TV Mini Series</type>
  <production_country>USA</production_country>
  <production_language>English</production_language>
 <genres>
  <genre>Family</genre>
  </genres>
 <keywords>
  <keyword>based-on-game-show</keyword>
  <keyword>game-show</keyword>
  </keywords>
  <colourinfo>Color</colourinfo>
  </movie>
```

Appendix B: Examples of Linked XMLL Documents from Movies Database

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
 <movie id="11">
  <title>$10,000 Under a Pillow</title>
  <production_year>1921</production_year>
  <production_country>USA</production_country>
  <director xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/939/319.xml">Moser,
Frank</director>
  <miscEntry>
  <person xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/939/319.xml">Moser,
Frank</person>
  <job>animator</job>
  </miscEntry>
 <genres>
  <genre>Animation</genre>
  <genre>Short</genre>
  <genre>Comedy</genre>
  </genres>
 <keywords>
  <keyword>domestic</keyword>
  <keyword>marriage</keyword>
  <keyword>sequel</keyword>
  </keywords>
  <colourinfo>Black and White</colourinfo>
  <soundmix>Silent</soundmix>
 <links>
 <link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../83/186.xml" linktype="FOLLOWED_BY">
 <movie>
  <title>Dashing North</title>
  <production_year>1921</production_year>
  </movie>
  </link>
 <link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../186/992.xml"
linktype="FOLLOWED_BY">
 <movie>
  <title>Kitchen, Bedroom, and Bath</title>
  <production_year>1921</production_year>
  </movie>
  </link>
 <link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../376/197.xml"
linktype="FOLLOWED_BY">
<movie>
  <title>Wars of Mice and Men, The</title>
  <production_year>1921</production_year>
  </movie>
  </link>
  <link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../51/67.xml" linktype="FOLLOWS">
  <movie>
  <title>Bud Takes the Cake</title>
  <production_year>1920</production_year>
  </movie>
  </link>
 <link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../51/61.xml" linktype="FOLLOWS">
 <movie>
  <title>Bud and Susie Join the Tecs</title>
  <production_year>1920</production_year>
  </movie>
  </link>
 <link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../51/62.xml" linktype="FOLLOWS">
 <movie>
  <title>Bud and Tommy Take a Day Off</title>
  <production_year>1920</production_year>
  </movie>
  </link>
```

```
<link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../53/417.xml" linktype="FOLLOWS">
<movie>
 <title>By the Sea</title>
 <production_year>1921</production_year>
 </movie>
 </link>
<link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../67/579.xml" linktype="FOLLOWS">
<movie>
 <title>Circumstantial Evidence</title>
 <production_year>1921</production_year>
 </movie>
 </link>
 <link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../96/983.xml" linktype="FOLLOWS">
 <movie>
 <title>Down the Mississippi</title>
 <production_year>1920</production_year>
 </movie>
 </link>
 <link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../118/58.xml" linktype="FOLLOWS">
 <movie>
 <title>Fifty-Fifty</title>
 <production_year>1920</production_year>
 </movie>
 </link>
 <link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../133/209.xml" linktype="FOLLOWS">
 <movie>
 <title>Getting Theirs</title>
 <production_year>1921</production_year>
 </movie>
 </link>
 <link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../140/233.xml" linktype="FOLLOWS">
 <movie>
 <title>Great Clean Up, The</title>
 <production_year>1920</production_year>
 </movie>
 </link>
 <link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../145/240.xml" linktype="FOLLOWS">
 <movie>
 <title>Handy Mandy's Goat</title>
 <production_year>1920</production_year>
 </movie>
 </link>
<link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../185/139.xml" linktype="FOLLOWS">
<movie>
 <title>Kids Find Candy's Catching, The</title>
 <production_year>1920</production_year>
 </movie>
 </link>
 <link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../208/277.xml" linktype="FOLLOWS">
 <movie>
 <title>Ma's Wipe Your Feet Campaign</title>
 <production_year>1921</production_year>
 </movie>
 </link>
<link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../222/889.xml" linktype="FOLLOWS">
<movie>
 <title>Mice and Money</title>
 <production_year>1920</production_year>
 </movie>
 </link>
<link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../241/567.xml" linktype="FOLLOWS">
<movie>
 <title>New Cook's Debut, The</title>
 <production_year>1920</production_year>
```

```xml
 </movie>
 </link>
<link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../246/356.xml" linktype="FOLLOWS">
 <movie>
 <title>North Pole, The</title>
 <production_year>1920</production_year>
 </movie>
 </link>
  <link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../269/690.xml" linktype="FOLLOWS">
 <movie>
 <title>Play Ball</title>
 <production_year>1920</production_year>
 </movie>
 </link>
 <link xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../294/45.xml" linktype="FOLLOWS">
 <movie>
 <title>Romance and Rheumatism</title>
 <production_year>1920</production_year>
 </movie>
 </link>
 </links>
 </movie>

 <?xml version="1.0" encoding="ISO-8859-1" ?>
<movie id="19">
<title>$2500 Bride, The</title>
<production_year>1912</production_year>
<production_country>USA</production_country>
<production_language>English</production_language>
<cast order="credits">
<casting position="1">
<actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/671/800.xml">Glaum,
Louise</actor>
</casting>
<casting position="2">
<actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/118/868.xml">De Grasse,
Joseph</actor>
</casting>
<casting position="3">
<actor xmlns:xlink="http://www.w3.org/1999/xlink/" xlink:type="simple" xlink:href="../../people/35/341.xml">Beatty, George
W.</actor>
</casting>
</cast>
 <genres>
<genre>Drama</genre>
<genre>Romance</genre>
<genre>Short</genre>
</genres>
<colourinfo>Black and White</colourinfo>
<soundmix>Silent</soundmix>
</movie>
```

Bibliography

[ABS00] S. Abiteboul, P. Buneman, and D. Suciu: *Data on the Web – From Relations to Semistructured Data and XML.* San Francisco: Morgan Kaufmann Publishers, 2000.

[AKM01] S. Abiteboul, H. Kaplan, and T. Milo: Compact Labeling Schemes for Ancestor Queries. In: *Proceedings of the 12th Annual ACM/SIAM Symposium on Discrete Algorithms (SODA), pages 547–556, Washington, USA, January 2001.*

[Ame86] American National Standard Institute: *The Database Language SQL*, *1986.*

[AR02] S. Alstrup and T. Rauhe: Improved labeling Scheme for Ancestor Queries. In: *Proceedings of the 13th Annual ACM/SIAM Symposium on Discrete Algorithms (SODA)*, *pages 947–953, San Francisco, CA, USA, January 2002.*

[BC00] A. Bonifati and S. Ceri: Comparative Analysis of Five-XML Query Languages. In: *SIGMOD Record, 29(1), 68-79, 2000.*

[BLFIM98] T. Berners-Lee, R. Fielding, U. Irvine, and L. Masinter: *Uniform Resources Identifier* (URI). Generic Syntax. *Available at http://www.ietf.org/rfc2396.txt, 1998.*

[BR99] Richard Baeza-Yates and Berthier-Neto: *Modern Information Retrieval. Addison-Wesley, 1999.*

[CAM02] G. Cobena, S. Abiteboul, and A. Marian: Detecting Changes in XML Documents. In: *the 18th Int. Conference on Data Engineering (ICDE). California, USA, 2002.*

[Car75] A. F. Cardenas: Analysis and performance of inverted database structure. *Communications of the ACM, 18(5), 253–258., 1975.*

[Cat94] R. G. Cattell: The Object Database Standard: ODMG-93. In : *Morgen Kaufmann, San Francisco, 1994.*

[CCB95] C. A. Clarke, G.V. Cormack , and F.J. Burkowski : An algebra for Structured text search and a framework for its implementation, *Computer Journal 38 (1), pages 43-56, (1995).*

[CCDF+98] S. Ceri, S. Comai, E. Damiani, P. Fratenali, S. Paraboschi, and L. Tanca: XML– GL: A Graphical Language for Querying and Reshaping XML Documents. In : *W3C Workshop QL98*, *USA, 1998.*

[CFI00] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian: XPERANTO: Publishing Object-Relational Data as XML. In: *Workshop on Web and Databases (WebDB), 2000.*

[CHKZ02] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick: Reachability and Distance Queries via 2-hop Labels. In: *Proceedings of the 13th Annual ACM/SIAM Symposium on Discrete Algorithms (SODA)*, *pages 937-946, San Francisco, CA, USA, January 2002.*

[CKM02] E. Cohen, H. Kaplan, and T. Milo: Labeling Dynamic XML Trees. In: *Proceedings of the 21st Symposium on Principles of Database Systems 2002 (PODS), pages 271–281, Madison, Wisconsin, USA, June 2002.*

[CLRS01] T. H. Coremen, C. E. Leiserson, R. L. Rivrest, and C. Stein: *Introduction to Algorithms*. *The Massachusetts Institute of Technology. 2001.*

[CMS02] C.–W. Chung, J.–K. Min, and K. Shim: APEX: An Adaptive Path Index for XML data. In: *Proceedings of the International Conference on Management of Data (SIGMOD) 2002, pages 121–132, Madison, Wisconsin, USA, June 2002.*

[Com79] D. Comer. : The Ubiquitous B-tree. *ACM Computing Surveys, 11(2), 121–137., 1979.*

[CRF00] D. Chamberlin, J. Robie, and D. Florescu : Quilt: An XML Query Language for Heterogeneous Data Sources". In : *The World Wide Web and Database (WebDB), Dallas, USA, Pages 53-62, 2000.*

[CSFH+01] B. Cooper, N. Sample, M. Franklin, G.R. Hjaltason, and, M. Shadmon: A fast index for semistructured data. In: *Proceedings of 27th International Conference on Very Large Data Bases (VLDB), September 11-14, 2001, Roma, Italy. Morgan Kaufmann , 2001.*

[CVZT+02] S-Y Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo: Efficient Structural Joins on Indexed XML Documents. In: *28th International Conference on Very Large Data Bases (VLDB), Hong Kong, China, 2002.*

[DB2] IBM DB2 Extender : http://www-306.ibm.com/software/data/db2

[DFFL+99] A. Deutsch, M. Fernandez, D. Florescu, A.Levy, and D. Suciu: XML-QL: A query language for XML. In: *International World Wide WebConference (www99), 1999.*

[DFS99] A. Deutsch, M. F. Fernandez, and D. Suciu: Storing Semistructured Data in STORED. In: *Proceedings on ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Pennsylvania, USA, pages 431-442. ACM Press, 1999.*

[DOM] Avlaible at: http://www.w3.org/DOM/.

[ES81] S. Even and Y. Shilloach: An on-line Edge-deletion Problem. *Journal of ACM, 28(1), 1-4, 1981.*

[FK99a] D.Florescu and D. Kossamnn: Storing and Querying XML Data using RDBMS. In: *IEEE Data Engineering Bulletin, 22(3): 27-34, 1999.*

[FK99b] D. Florescu and D. Kossamnn: A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. *Technical Report, IRIA, 1999.*

[FMST01] M. F. Fernandez, A. Morishima, D. Suciu, and W. Tan: Publishing Relational Data as XML: The SilkRoute Approach. In: *IEEE Data*

Engineering Bulletin 24(2), 2001.

[GH76] S. Gindburg and M. Harrison: Bracketed Context-Free Languages. In: *Journal of Computer and System Sciences*, *1(1):1-23, 1976.*

[GK03] T. Grust and M. Keulen: Tree Awareness for Relational DBMS Kernels: Staircase Join. In: *Intelligent Search on XML Data , Springer Verlag, September 2003.*

[GMV00] A. Gutierrez, R. Motz, and D. Viera : Building databases with Information extracted from web documents. In: *International conference of the Chilean computer science society, pages 41–49, 2000.*

[Gru02] T. Grust: Accelerating XPath Location Steps. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, *Wisconsin, USA, pages 109-120, 2002.*

[GW97] R. Goldman, and J. Widom: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In: *Proceedings of the 23rd Very Large Databases Conference* (*VLDB)*, *Pages 436–445, Athen, Greece, 1997.*

[Har99] E. R. Harold: *XML Bible*. *USA, IDG Book World Wide, Inc., 1999.*

[Hom99] A. Homer*: XML IE5: Programmer's References*. *Canda, Worx Press Ltd., 1999.*

[Huf02] D. Huffman: A Method for the Construction of Minimum Redundancy Codes. In: *Proceedings of the IRE, 40, 1098-1101 (1952).*

[HY04] H. He and J. Yang: Multiresoultion Indexing of XML Frequent Queries. In: *Proceedings of the 20th International Conference on Data Engineering (ICDE)*, *Boston, USA, 2004.*

[Ita88] G. F. Italiano: Finding Paths and Deleting Edges in Directed Acyclic Graphs. In: *Information Processing Letters*, *pages 5-11, 1988.*

[IMDB] The Internet Movie Database: *available at http://www.imdb.com*

[JLWO03] H. Jiang, H. Lu, W. Wang, and B.C. Ooi: XR-Tree: Indexing XML Data for Efficient Structural Joins. In: the 19th Int. *Conference on Data*

Engineering (ICDE), pages 253-263, Bangalore, India, 2003.

[KBNK02] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth: Covering Indexes for Branching Path Queries. In: *Proceedings of the 21th International ACM Conference on Management of Data (SIGMOD), pages 133–144, Madison, Wisconsin, USA, June 2002.*

[KBNS02] R. Kaushik, P. Bohannon, J. Naughton, and P. Shenoy: Updates for Structure Indexes. In: *Proceedings of the 28th Inter. Conference of Very Large Data Base (VLDB), China, 2002.*

[KJKP+02] S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Partel, D. Srivastava, and Y. Wu: Structural Joins: A Primitive for Efficient XML Queries Pattern Matching. In: *18th Inter. Conference on Data Engineering (ICDE). California, USA, 2002.*

[KM00] M. Klettke and H. Meyer: XML and Object-Relational System-Enhancing Structure Mappings Based on Statistics. In: *The World Wide Web and Database (WebDB), Dallas, USA, pages 151-170, 2000.*

[KM90] A. Kempfer and G. Moerkotte: Advances Query Processing in Object Based Using Access Support Relations. In: *Proceedings of the 16th Very Large Data Base (VLDB) Conference, Australia, 1990.*

[KMS02] H. Kaplan, T. Milo, and R. Shabo: A comparison of labeling schemes for Ancestor Queries. In: *Proceedings of the 13th Symposium on Discrete Algorithms 2002 (SODA), pages 954–963, San Francisco, California, USA, 2002.*

[Knu98] Donald Knuth. The Art of Computer Programming*, Vol.3, Sorting and Searching, Third Edition. Addison Wesley, Reading, MA, 1998.*

[KS99] V. King and G. Sagert: A Fully Dynamic Algorithm for Maintaining the Transitive Closure. In: *Proceedings of Annual Symposium on Theory of Computing (STOC), 1999.*

[KSBG02] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes: Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In: *Proceedings* *in the 18th Int. Conference on Data Engineering (ICDE), San Jose, California, 2002.*

[LaL77] W. LaLonde: Regular Right Part Grammars and Their Parsers. In: *Communications of ACM, 20(10): 731-741. 1977.*

[LJ99] A. Le-Hors and I. Jacobs: HTML 4.01 Specification. *http://www.w3.org/TR/html4/, W3C Recommendation 24 December 1999.*

[LM01] Q. Li and B. Moon: Indexing and Querying XML Data for Regular Path Expressions. In: *Proceedings of the 27th International Very Large Database Conference (VLDB), pages 361–370, Roma, Italy, September 2001.*

[Mac91] I. MacLeod: A query language for retrieving information from hierarchical text structures. *The Computer Journal, 34(3), 254–264, 1991.*

[Meg02] D. Megginson. SAX 2.0: The simple API for XML. *Available at http://www.saxproject.org/ 2002.*

[Meg98] D. Megginson: *Structuring XML Docuemnts. USA : Pentice Hall PTR, 1998.*

[MFKX+ 00] I. Manolescu, D. Florescu, D. Kossmann, F. Xhumari, and D. Olteanu : Agora: Living with XML and relational database. In: *Proceedings of the 26th International Very Large Database Conference VLDB, pages 623–626. Cairo, Egypt. 2000.*

[MS99] T. Milo and D. Suciu: Index Structures for Path Expressions: In: *Proceedings of the 7th International Conference on Database Theory (ICDT) 1999, pages 277–295, Jerusalem, Israel, 1999.*

[MWAL+98] J .McHugh, J. Wisdom, S. Abiteboul, Q. Luo, and A. Rajaraman: Indexing Semistructured Data. *Technical Report Stanford University, 1998.*

[NS93] E. Nuutila and Soisalon-Soininen. Efficient Transitive Closure

Computation. *Technical Report TKO-B113*, *1993.*

[NUWC97] S, Nestorov, J. Ullman, J. Wiener, and S. Chaeathe: Representative Objects: concise representation of semistructured, hierarchical data. In: *IEEE International Conference on Data Engineering, 1997.*

[Oracle] Oracle 9i Release 2 (9.2.0.1) Oracle Corporation: *http://www.oracle/com/.*

[QLO03] C. Qun, A. Lim, and W. Ong: D(K)-Index: Adaptive Structural Summaries for Graph-based Data. In: *ACM SIGMOD Inter. Conference on Management of Data, California*, *USA, pages 134-144, 2003.*

[RG00] R. Ramakrishnan and J. Gehrke: *Database Management Systems*. *The McGraw-Hill Companies. 2000.*

[RLS98] J. Robie, J. Lapp, and D. Schah: XML Query Language (XQL). In: *The Query Language Workshop (QL'98), Des.1998.*

[RM01] F. Rizzolo and A. Mendelzon: Indexing XML Data with ToXin. In: *4th International Workshop on the Web and Databases (in conjunction with ACM SIGMOD 2001). Santa Barbara, CA. May 2001.*

[RP02] K. Runapongsa and J. M. Patel: Storing and Querying XML Data in Object-Relational DBMS. In: *International Conference on Extending Database Technology (EDBT) Workshops*, *Prague, Czech Republic, March 24-28, 2002.*

[SAX] SAX Project: *http://www.saxproject.org*.

[Sch01] T. Schlieder: ApproXQL: Design and Implementation of an Approximate Pattern Matching Language for XML. *Technical Report B 01–02, FU Berlin, May 2001.*

[SSBC+00] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald: Efficiently Publishing Relational Data as XML Documents. In: *Proceedings of International Conference on Very Large Database (VLDB*), *Cairo, Egypt, 2000.*

[ST93] A Salminen and F. Tompa: PAT expressions: an algebra for fast text search. In: *Proceedings of the 2th International Conference on Computational Lexicography*, *COMPLEX '92, (pp. 309-332). Budapest: Linguistics Institute, Hungarian Academy of Sciences, 1993.*

[STHZ+99] J. Shanmugasundaram, K. Tuffe, G. He, C. Zhang, D. DeWitt, and J. Naughton: Relational Database for Querying XML Documents: Limitations and Opportunities. In: *Proceedings of International Conference on Very Large Database (VLDB)*, *September 1999, UK, pages 302-314. Morgan Kaufmann, 1999.*

[STW04] R. Schenkel, A. Theobald, and G.Weikum: HOPI: An Efficient Connection Index for Complex XML Document Collections. In: *Proceedings of the 9th International Conference on Extending Database Technology (EDBT) 2004, pages 237–255, Crete, Greece, March 2004.*

[SU03] A. Sayed and R. Unland: Index-Support on XML Documents Containing Links" In: *the 46th International IEEE Symposium on Circuiles and System. Dec. 2003.*

[SU04] A. Sayed and R. Unland: Indexing and Querying Heterogeneous XML Collections" In: *14th International Conference on Computer Theory and Applications*, *28-30 Sep. 2004, Alex, Egypt.*

[SU05] A. Sayed and R. Unland: HID: An Efficient Path Index for Complex XML Collections with Arbitrary Links" In: *4th International Conference on Databases in Networked Information Systems (DNIS 2005), March 28 - 30, 2005, Aizu, Japan. Pages 78-91. In Springer Lecture Notes in Computer Science (LNCS).*

[SYU99] T. Shimura, M. Yoshikawa, and S. Uemura: Storage and Retrieval of XML Documents using Object-Relational database. In: *Database and Expert System Applications*, *pages 206-217, 1999.*

[TDCZ00] F. Tain, D. DeWitt, J. Chen, and C. Zhang: The Design and Performance Evaluation of Alternative XML Storage Strategies. In: *Technical Report, department of computer science, university of Wisconsin. Available at; www.cs.wisc.edu.*

[TW02] A. Theobald, G. Weikum: The Index–based XXL Search Engine for Querying XML Data with Relevance Ranking. In: *the 8th International Conference on Extending Database Technology (EDBT) 2002, Prague, Czech Republic, pages 477–495, March 2002.*

[UW97] J. D. Ullman and J. Wisdom: A First Course in Database Systems. *New Jersey, Penttice-Hall, Inc., 1997.*

[W3C] World Wide Web. Available at http://www.w3.org/.

[W3C01XL] XML Linking Language (XLink) Version 1.0, *W3C Recommendation (27 June 2001), available at http://www.W3.org/TR/xlink.*

[W3C02XP] XML Pointer Language (XPointer), *W3C Working Draft (16 August 2002), see http://www.w3.org/TR/xptr.*

[W3C98X] Extendible Markup Language (XML). *Available at http://www.w3.org/REC- XML. 1998.*

[W3C99P] XML Path Language (XPath): *available at http://www.w3.org/TR/xpath, 1999.*

[W3C99P] XML Path Language (XPath): *available at http://www.w3.org/TR/xpath, 1999.*

[W3CD98] Document Object Model (DOM). *Available at http://www.w3.org/DOM/, 1998.*

[W3CS01] XML Schema. *Available at http://www.w3.org/xmlschema/, 2001.*

[WBDG+02] K. Williams, M. Brundage, P. Dengler, J. Gabriel, A. Hoskinson, M.Kay, T. Maxwell, M. Ochoa, J. Papa, and M. Vnmane: *Professional XML Database. Wrox Press Ltd., 2000.*

[WL02] E. Wilde and D. Lowe: *XPath, XLink, XPointer, and XML – A Practical Guide to Web Hyper linking and Translations. Boston: Addison Wesley, 2002.*

[WLH04] X. WU, M. Li Lee, and W. Hsu: A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In: *Proc. of the 20th Inter. Conference on Data Engineering (ICDE), Boston, USA, 2004.*

[WWYZ+02] H. Wu, Q. Wang, J.X. Yu, A. Zhou, and S. Zhou: UD (k, l)-index An

Efficient Approximate Index for XML Data. In: *The Fourth Inter .Conference on Web-Age Information Management, China, 2003.*

[XQuery] XQuery 1.0: A query Language for XML. *W3C Working Draft, 2001. Available at http://www.w3.org/TR/xquery/.*

[YHSY04] K. Yi, H. He, I. Stanoi, and J. Yang: Incremental Maintenance of XML Structural Indexes. In: *Proceedings of the International Conference on Management of Data (SIGMOD), Pairs, 2004.*

[Yoo96] J. Yoon: Extracting Database Knowledge from Query Trees, *Journal of Electrical Engineering and Information Science, Vol.1, No.2, pp.145-156, 1996.*

[ZADR03] P. Zezula, G. Amato, F. Debole, and F. Rabitti: Tree Signatures for XML Querying and Navigation. In: *1st international XML Database Symposium, Pages 149-163, 2003.*

[ZAR03] P. Zezula, G. Amato, and F. Rabitti: Processing XML Queries with Tree Signatures. In: *Intelligent Search on XML Data. Pages 247–258, Springer–Verlag: Berlin, 2003.*