UNIVERSITÀ DELLA CALABRIA

Dipartimento di Elettronica,
Informatica e Sistemistica

Dottorato di Ricerca in
Ingegneria dei Sistemi e Informatica
XX ciclo

Tesi di Dottorato

XML and Web Data Management

Bettina Fazzinga

UNIVERSITÀ DELLA CALABRIA

Dipartimento di Elettronica,
Informatica e Sistemistica

Dottorato di Ricerca in
Ingegneria dei Sistemi e Informatica
XX ciclo

Tesi di Dottorato

XML and Web Data Management

Bettina Fazzinga

[illegible]

Coordinatore
Prof. Domenico Talia

[illegible]

Supervisore
Prof. Sergio Flesca

[illegible]

DEIS

DEIS – Dipartimento di Elettronica, Informatica e Sistemistica
Novembre 2007

Settore Scientifico Disciplinare: ING–INF/05

To Simona and Alessandra

Acknowledgments

First of all, I wish to gratefully acknowledge my Ph.D. supervisor, Sergio Flesca, the point of reference for my work, for his support, his valuable suggestions and various insightful discussions. Special thanks go to my dear friend and precious colleague Andrea Pugliese, for everything he taught me, for the useful and funny discussions and for his patience when I was stressful. I am deeply grateful to my dearest friend Fabio, for his constant support and for having been always present during these years, even in the hardest moments. A big thank you goes to Ale, for her moral support and for all the funny moments spent together. Great thanks go to Filippo, for having made my studies more pleasant and for having made friend with me. Finally, I express my gratitude to all the friends of my department, that made my Ph.D. course a special experience.

Contents

List of Figures

List of Tables

1 Introduction

Internet is a worldwide, publicly accessible series of interconnected computers. It is a “network of networks” that consists of millions of smaller domestic, academic, business, and government networks, which together provide various information and services, such as electronic mail, online chat, file transfer, and the interlinked web pages and other resources of the World Wide Web [108].

Nowadays, millions of persons and corporations exploit Internet and networks for several purposes, among which retrieving, divulging and sharing information. From the company point of view, the development of network related technologies has carried out several changes. Initially, corporate information systems only managed local data. Therefore, as all analysis and researches were limited to their own data, it was not possible to provide enough information to support management decision. The proliferation of networks has boosted the amount and the completeness of data available, thus companies have been stimulated to develop applications suitable for retrieving data from other places of the world and handling them.

Since up to few years ago, all the information available in Internet was codified using *Hypertext Markup Language* (HTML) [107], the first need was the development of applications that retrieve data embedded in Web pages and process them automatically. Unfortunately, HTML documents are not suitable for being automatically processed by applications, since HTML is mainly aimed at describing and displaying data, but it does not support the use of a meta-linguistic level describing the semantics of Web documents. HTML is not suitable for sharing data among applications in a network, as HTML is a “human readable” format, whereas the automatic processing of information requires a “machine readable” format. To this aim, the *Extensible Markup Language* has been adopted [105]. XML is a general-purpose markup language, classified as an extensible language because it allows its users to define their own tags. Differently from HTML, XML is a meta-language suitable for documents wherein data are associated with a structure and a semantics. Using XML for sharing data on the networks the process of collecting and integrating data from several sources has been hastened, since XML is suitable for automatic processing. However, other difficulties remain to be solved, mainly due to the fact that data in the sources can be organized in different way. Specifically, since in the general case sources are totally autonomous, data stored in different sources can be in accordance with different schemas and can be "partial", in the sense that sources can be interested to the same data, but from different point of views, thus storing only partial information.

For copying with these difficulties, several approaches have been proposed, mostly aimed at building a mediator system that hides to the user the amount of sources and the differences among them and interacts with all sources for providing the user with a meaningful answer. Specifically, mediator systems provide uniform access to multiple data sources encapsulating proper mechanisms for handling the difference among the schemas of the sources and for allowing the user to pose queries on the overall amount of data [11, 14, 15, 24, 48, 49, 55, 56, 62, 66, 76, 94]. In Figure 1.1 a simple architecture for retrieving data from several sources using a mediator is shown.

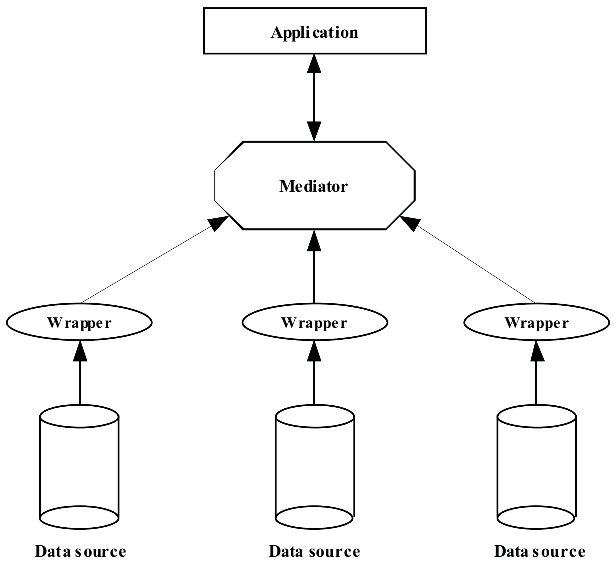


Fig. 1.1: A simple architecture of system for integrating/querying data

In this architecture, wrappers are used for providing a unique format for source data, i.e. data are extracted and translated from the original format to a proper one uniform for all sources. Wrappers output data in a specific format, such as XML, and these data are further managed for answering queries and collecting results. Several approaches have been proposed for providing a uniform access to different data, but the most common is the usage of global views over data. In such an approach, a virtual schema summarizing and representing all the data stored at the sources is available and users are allowed to pose queries on it. Therefore, for these kinds of systems, it is required to:

- build proper wrappers for data extraction;
- build mappings between the schemas of the source data and global view;
- rewrite queries posed on the global view for being evaluated on source data;

In the rest of this section, we discuss the main issues related to wrappers, mappings and techniques for query rewriting.

The purpose of a wrapper is that of extracting the content of a particular information source and delivering the relevant content in a self-describing representation. Although wrappers are not limited to handle Web data, most of their current applications belong to this domain. In the Web environment, a wrapper can be defined as a processor that converts information implicitly stored in an HTML document into information explicitly stored in a proper format for further processing. Web pages can be ranked in terms of their format from structured to unstructured. Structured pages follow a predefined and strict, but usually unknown, format where information presents uniform syntactic clues. In semistructured pages, some of these constraints are relaxed and attributes can be omitted, multivalued or changed in its order of occurrence. Unstructured pages usually consist of free text merged with HTML tags not following any particular structure. Most existing wrapping systems are applied to structured or semistructured Web pages. Knowledge is encoded in rules that are applied over the raw text (in a pattern matching process) or over a more elaborated or enriched data source (sequence of tokens, set of predicates, HTML tree, etc.). Generally, tokens include not only words but also HTML tags. This fact has important consequences. On the one hand, HTML tags provide additional information that can be used for extraction; on the other hand, the presence of HTML tags makes it difficult to apply linguistic based approaches to extraction.

The main problem in building Web wrappers is to make the wrapper design process as automatized as possible, since each source does not only frequently update its data but also its layout. Several wrapping systems have been proposed (see Chapter 3 for details), classifiable in automatic, semi-automatic and manual tools, according to the degree of automation in the wrapper design phase. The main component of a wrapper is the set of extraction rules used for extracting data. Existing wrapping systems can be classified in three main categories: HTML-aware tools, NLP-based tools and wrapper induction tools. The first essentially aim at analyzing the tree structure of the HMTL page and generate extraction rules in a semi-automatically or automatically way. The second usually apply techniques such as part- of-speech tagging and lexical semantic tagging to build relationship between phrases and sentences elements, so that extraction rules can be derived. Such rules are based on syntactic and semantic constraints that help to identify the relevant information within a document. The NLP-based tools are usually more suitable for Web pages consisting of grammatical text, possibly in telegraphic style, such as job listings, apartment rental advertisements, seminar announcements, etc. Wrapper induction tools generate delimiter-based extraction rules derived from a given set of training examples. The main distinction between these tools and those based on NLP is that they do not rely on linguistic constraints, but rather in formatting features that implicitly delineate the structure of the pieces of data found. This makes such tools more suitable for HTML documents than the previous ones.

As regards the problem of building mappings between the schemas of the source data and global view and the problem of rewriting queries posed on the global view, various approaches have been proposed in the data integration literature [11, 14, 15, 20, 24, 33, 34, 49, 55, 56, 62, 72, 74, 76, 92, 94, 102]. Data integration systems provide a uniform query interface to a multitude of autonomous data sources, which may reside within an enterprise or on the World-Wide Web. Data integration systems free the user from having to locate sources relevant to a query, interact with each one in isolation, and combine data from multiple sources. Users of data integration systems do not pose queries in terms of the schemas in which the data is stored, but rather in terms of a global schema. The global schema is designed for a specific data integration application, and it contains the salient aspects of the domain under consideration. This approach is used both in the relational and in the semistructured context (see Chapter 4 for details). However, here we give an overview of this technique without focusing on a specific context[1]. Data represented in the global schema are not actually stored in the data integration system. The system includes a set of source descriptions that provide semantic mappings between concepts in the source schemas and concepts in the global schema. Two main approaches are used for building mappings between global schema and source schema: global-as-view (GAV) and local-as-view (LAV). In the former, concepts in the global schema are defined as views over the concepts in the sources, whereas in the latter concepts in the sources are specified as views over the global schema. In the GAV approach, translating the query on the global schema into queries on the local schemas is a simple processing of unfolding. In the LAV approach, the query on the global schema needs to be reformulated in the terms of the local sources schemas. This process is known as "rewriting queries using views" and it is an NP-hard problem [72], in the case that equivalent or maximally-contained rewritings are desired. On the other hand, in a GAV architecture, to handle modifications in local source schemas or insertion of new sources the global schema needs to be redesigned.

In many applications, the schemas of the source data are so different from one another that a global schema would be almost impossible to be built, and very hard to maintain over time. Hence, a solution is to provide architectures for decentralized data sharing. In this context, mappings between disparate

[1] We use the term "concept" for intending relation or XML elements

schemas are provided locally between pairs or small sets of sources. When a query is posed at a source, relevant data from other sources can be obtained through a semantic path of mappings. The key step in query processing in this kind of architectures is reformulating a query over other sources on the available semantic paths. Broadly speaking, the process starts from the querying source and the query is reformulated over its immediate neighbors, then over their immediate neighbors, and so on. In these architectures joining the network can be done opportunistically, i.e., a source can provide a mapping to the most convenient (e.g., similar) source(s) already in the network, and a source can pose a query using its own schema without having to learn a different one. On the other hand, when a source joins the network it is necessary to build mappings with several neighbors in order to be reachable through a "chain" of semantic mappings for query processing. Therefore, the volatility of sources and the dynamism of the network become a critical aspect, since the building of mappings is not immediate and simple.

Finally, in all the discussed cases, sources are forced to provide information about their own schemas in order to build mappings, thus source autonomy is violated.

In this thesis, we address all the phases needed for providing a system for querying heterogeneous multiple data sources, firstly tackling the problem of extracting XML data from Web pages and secondly dealing with the problem of retrieving XML data spread across several sources.

As the first issue, we propose a wrapping system using the schema of the information to be extracted in both the design and evaluation steps. The main advantages of this approach range from the capability of easily guiding and controlling the extraction and integration of required data portions from HTML documents, to the specification of structured yet simple extraction rules.

As regards the second issue, we consider the scenario where the user is not aware of the local data schemas and no global schema is provided, thus avoiding all the problems regarding building and maintenance of mappings and guaranteeing source autonomy. Our approach enables the retrieval of meaningful answers from different sources with a limited knowledge about their local schemas, by exploiting vague querying and approximate join techniques. It essentially consists in first applying transformations to the original query, then using transformed queries to retrieve partial answers and finally combining them using information about retrieved objects.

Plan of the thesis

The rest of this thesis is organized as follows. Chapter 2 briefly introduces XML and describes the main characteristics of the most widely adopted XML query languages. Chapter 3 describes our wrapping technique and presents some experimental results of our wrapping system. Finally, Chapter 4 introduces our approach to the querying of XML heterogeneous sources and describes a P2P system implementing the proposed techniques.

2

Preliminaries

2.1 Semistructured data and XML

Relational and object-oriented data models have been widely adopted for modeling the large amounts of data managed by modern database systems. Nowadays, less rigid data models are needed in order to deal with new kinds of data that are *irregular* (also w.r.t. time), and *self-describing*, i.e., the schema is embedded in data themselves. Data with the above-mentioned characteristics are commonly used in Web systems, scientific databases, and data integration systems.

Examples of data models suitable for these new data are *semistructured* models [2, 3, 22, 25, 32, 99, 103]. In these models, data are node-labeled trees. Nodes in the trees are viewed as objects, and carry along the concept of identity; data values are associated with leaf nodes.

Nowadays, *Extensible Markup Language* (*XML*) [105] is considered the standard format for semistructured data. Though XML was invented as a syntax for data, its characteristics make it usable as a logical data model as well. An XML *document* consists of nested *elements*, with *ordered* sub-elements or value child nodes. Each element is delimited by a *start-* and *end-tag* pair, has an associated name, and may have associated name-value pairs called *attributes* (written in the start tag). An example XML document, drawn from [112], is shown in Figure 2.1. In a *well-formed* XML document, tags are always properly nested; thus, each element may correspond to a tree node, and an XML document can be entirely mapped to a tree, as it is typical for semistructured data. Figure 2.2 shows an extract of a tree corresponding to the example document of Figure 2.1.

XML *processing instructions* are of the form "`<?target data?>`", where *target* is the application that is expected to process the enclosed *data*. The XML *prolog* ("`<?xml version="1.0"?>`" in the example document) is a particular processing instruction that identifies the document as an XML one and may specify the value of some pre-defined attributes (among which, the attribute "`version`" reported in the example). *Comments* are delimited by

```
<?xml version="1.0"?> <bib>
    <book year="1994">
        <title>TCP/IP Illustrated</title>
        <author> <last>Stevens</last> <first>W.</first> </author>
        <publisher>Addison-Wesley</publisher>
        <price> 65.95</price>
    </book>

    <book year="2000">
        <title>Data on the Web</title>
        <author> <last>Abiteboul</last> <first>Serge</first> </author>
        <author> <last>Buneman</last> <first>Peter</first> </author>
        <author> <last>Suciu</last> <first>Dan</first> </author>
        <publisher>Morgan Kaufmann Publishers</publisher>
        <price>39.95</price>
    </book>

    <book year="1999">
        <title>The Economics of Technology and
                 Content for Digital TV</title>
        <editor>
            <last>Gerbarg</last> <first>Darcy</first>
            <affiliation>CITI</affiliation>
        </editor>
        <publisher>Kluwer Academic Publishers</publisher>
        <price>129.95</price>
    </book>
</bib>
```

Fig. 2.1: Example XML document.

special markers, "`<!--`" and "`-->`", whereas "`CDATA`" sections, containing free unparsed data (i.e., data that is directly forwarded to the application), are delimited by markers "`<![CDATA[`" and "`]]>`". Finally, empty elements may follow the shortcut "`<elementName/>`".

The main drawback of the flexibility of semistructured data is the loss of schema. Schemas describe data, ease the formulation of queries, and allow for efficient storage and querying. In general, adding a schema to semistructured data means constraining the paths that can be found in the tree, and the type of data values. Two different languages can be used for specifying the schema of an XML document, namely the *Document Type Definitions* (*DTD*) [105] and *XMLSchema* [109]; in both cases, a document is said to be *valid* w.r.t. a given schema if it complies with the constraints stated in the schema. DTDs express simple schemas for XML data; there is almost no distinction among value types, and structural constraints may be specified for basic tree structures. XMLSchema is much more expressive: it embeds a richer set of scalar

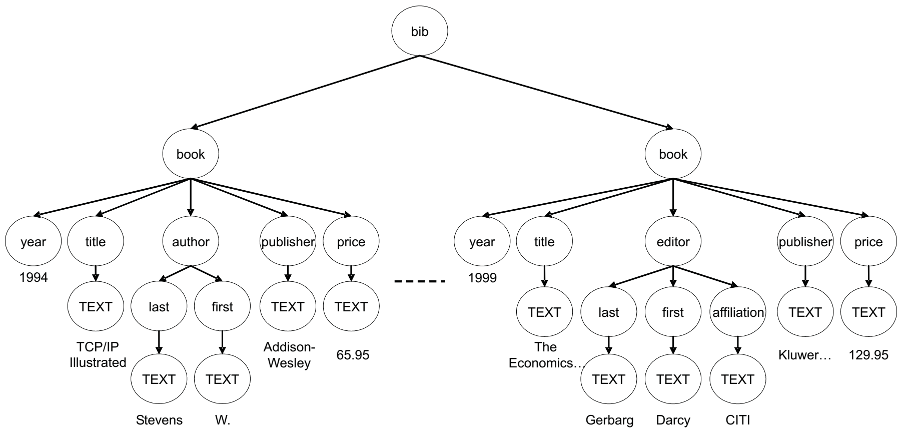


Fig. 2.2: Tree corresponding to the example document.

types, allows user-defined simple types, subtyping, and more expressive constraints on XML tree structures. However, we concentrate here on DTDs, that are extensively used in this thesis.

A DTD is essentially a set of document and attribute declarations, allowing to specify, for non-leaf nodes, the (ordered) structure of children, and for leaf nodes, their type. For example, the declaration `<!ELEMENT bib (book)*>` dictates that `bib` nodes may have an arbitrary number of `book` children, whereas the declaration `<!ELEMENT editor (last,first,affiliation)>` constrains `editor` nodes to have a sequence of a `last`, a `first`, and an `affiliation` child nodes. Element structure declarations may also embed disjunction ("`|`") and cardinality constraints ("`*`" indicates any number of instances, "`+`" one or more instances, and "`?`" zero or one instance). The special symbol "`#PCDATA`" indicates textual content, whereas "`EMPTY`" constrains a node to be empty.

Attribute declarations constrain attributes, indicating the nodes they must be associated with, their "type", and default values. For instance, the declaration `<!ATTLIST book year CDATA #REQUIRED>` associates a mandatory ("`#REQUIRED`") `year` attribute to `book` elements, with textual content ("`CDATA`"). The main attribute types and attribute specification parameters are shown in Table 2.1; Figure 2.3 depicts a possible DTD for the `bib` document of Figure 2.1.

Finally, two approaches are widely adopted for parsing XML data. *Document Object Model* (*DOM*) [106] parsers build the XML tree in memory, associating a list of children with every node. The DOM API supports navigation in the tree and creation/modification of nodes. The main drawback of the DOM approach is that it may be expensive to materialize whole trees for large XML data collections. In contrast, *Simple API for XML Parsing* (*SAX*) [93] is event-driven, that is an event is fired and notified to the application for each node found during document scans; this obviously corresponds

Attribute type	Meaning	Specification	Meaning
`(v1, ..., vn)`	Enumeration	`#REQUIRED`	Mandatory
`CDATA`	Text string	`#IMPLIED`	Optional
`ID`	Unique identifier	`"v"`	Default value is `v`
`IDREF`	Reference to an ID	`#FIXED "v"`	Fixed value is `v`
`IDREFS`	References to IDs		

Table 2.1: Main DTD attribute types and specification parameters.

```
<!ELEMENT bib (book)*>
<!ELEMENT book (title,(author+|editor+),publisher,price?)>
<!ATTLIST book year CDATA #REQUIRED>
<!ELEMENT author (last,first)>
<!ELEMENT editor (last,first,affiliation)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT affiliation (#PCDATA)>
```

Fig. 2.3: DTD of the example document.

to a preorder visit of the corresponding trees. The main advantage of SAX parsing is that it consumes much less memory than DOM, and can be significantly faster; moreover, full parsing is not needed, as XML data are read in a "streaming" way. Obviously, with SAX it is not possible to create or modify tree nodes; therefore, it is more used in conjunction with LIFO data structures to support loading into storage systems or validation.

2.2 Querying XML data

The main query languages for XML data available today are *XPath* [110] and *XQuery* [111]. The need for specific query languages for XML arises for dealing with its specificities, e.g., its ordered hierarchical structure and the potential absence of a schema. Both XPath and XQuery can be used in a declarative fashion, and their semantics is described in terms of *abstract data models*, independently of the physical data storage.

2.2.1 XPath

XPath is aimed at addressing parts of XML documents; the result of an XPath query is indeed a sequence of XML nodes. From the point of view of XPath, every element, comment, attribute, processing instruction or text string is a node in the XML tree; in particular, attributes are children of their corresponding elements, and comments, PIs and text strings are children of their enclosing elements.

During the XML tree traversal, moving from a "current" node, XPath allows to choose among current node's children as well as among other sequences of nodes having a "structural" relationship with it. These sequences of nodes are called *axes*. Axes are thus always referred to the current node, and the *child* axis, comprising its direct children, is the default. Table 2.2 shows the main XPath axes.

Axis	Nodes addressed w.r.t. current node n
ancestor	Parent of n, grandparent of n, etc.
ancestor-or-self	As *ancestor*, but including n
attribute (also denoted as "@")	Attributes of n
child	Children of n
descendant (also denoted as "/")	Children of n, grandchildren of n, etc.
descendant-or-self	As *descendant*, but including n
parent	Parent of n

Table 2.2: Main XPath axes.

To select particular nodes in an axis on the basis of their type and/or name, XPath uses *node tests*. Node tests return subsequences of the current axis. For instance, the *element*() test selects element nodes, whereas *text*() and *attribute*() select text and attribute nodes, respectively. If the axis contains elements, *element*() is the default type, and if it contains attributes, the default is (obviously) *attribute*(). To identify subsequences of named nodes (elements and attributes), the test is denoted by the name itself ("*" matches all named nodes).

XPath *filters*, enclosed in "`[`" and "`]`" brackets, select in turn subsequences of the nodes returned by node tests. Filters are built by combining XPath expressions or constants by means of comparison, logic, mathematical and set operators. A filters f is either a boolean or an arithmetic expression. In the first case (f is a boolean expression), the result of applying f to a node sequence s consist of those nodes $n \in s$ such that f evaluates to true on n (a node sequence is seen as boolean expressions itself, evaluating to *true* if it is not empty). In the latter case (f is an arithmetic expression), the result of applying f to a node sequence s is the node having the corresponding position in s. A rich set of functions is available for expressing filters, mainly working on node sequences, strings, and numerical/boolean values (see Table 2.3).

Finally, an XPath *expression* consists of a sequence of *steps*, each step comprising an axis, a node test, and a sequence of filters (meant as a conjunction of filters). Steps are written in their order, separated by a "/". Each step operates on the node sequence obtained at the previous step. The first step

Function	Meaning
count(*ns*)	Cardinality of sequence *ns*
last()	Index of the last node in the current sequence
contains(*s*, *t*)	*true* if string *s* contains string *t*
starts-with(*s*, *t*)	*true* if string *t* is a prefix of string *s*
round(*x*)	Integer nearest to *x*
sum(*ns*)	Sum of numbers in *ns*
not(*e*)	*true* iff *e* is *false*

Table 2.3: Some XPath functions.

operates on a context that is defined by the "environment", or is the root of a document (in this case, the expression begins with a "/"). Table 2.4 shows some example XPath expression, referred to the document in Figure 2.1.

Expression	Meaning
`/bib/book/author`	Book authors
`/bib/*`	Children of `bib`
`/bib/book[title="Data on the Web"]/@year`	Year of the book titled "Data on the Web"
`/bib/book[@year="2000"]/author`	Authors of books in 2000
`/bib/book[author and not(price)]`	Books having an author and not a price
`/bib/book[1]/title`	Title of first book

Table 2.4: Example XPath expressions.

2.2.2 XQuery

XQuery can express arbitrary XML queries. It includes XPath as a sub-language, and guarantees logical/physical data independence. The abstract model adopted in XQuery is based on *item sequences*, where items are XML nodes or atomic values. XQuery is strongly typed; using types from XMLSchema, it assigns a static type to every expression, and a dynamic type to every XML instance. This strong typing brings several benefits: errors can be detected statically, the type of results can be inferred, and the result of a query can be guaranteed to be of a given type based on the input data type.

A query in XQuery is in the form of a (freely nested) `for-let-where-order by-return` (*FLWOR*) expression. `for` and `let` clauses build variable bindings to node sequences; the difference between the two is that the semantics of the former additionally requires *iteration* through the sequences, that is a result is produced for each node, instead that for each sequence. The `where` clause prunes variable bindings according to some selection criterion; finally, the `return` clause builds outputs. For instance, the query

```
for $b in document("bib.xml")//book
order by $b/@year
return $b
```

returns the `book` nodes in document *bib.xml* (Figure 2.1), sorted by year, whereas the query

```
for $b in document("bib.xml")//book
where $b/author/last="Stevens"
return <book-title>{$b/title}</book-title>
```

returns a sequence of `book-title` nodes, where each node contains the title of a book in *bib.xml* written by an author whose last name is “Stevens”. Moreover, XQuery allows users to define functions (user-defined functions), using a very rich set of language primitives; a comprehensive set of operators is also provided to work on expressions and item sequences. It is easy to see that XQuery can also express joins. For instance, the query

```
for $b1 in document("bib1.xml")//book,
   $b2 in document("bib2.xml")//book
where $b1/title=$b2/title
return <book><title>$b1/title</title>
          <price1>{$b1/price}</price1>
          <price2>{$b2/price}</price2>
      </book>
```

yields a `book` element for each pair of books having the same title in documents *bib1.xml* and *bib2.xml*, and pairs their prices. A sample set of XQuery queries using other language features is shown in Table 2.5.

Query	Result

`for $b in document(...)//book`
`where contains($b/title/text(),"Web")`
`return $b/author`
	Authors who wrote a book about the Web

`for $b in document(...)//book,`
`  $t in $b/title, $a in $b/author`
`return <result>{$t}{$a}</result>`
	All title-author pairs

`for $a in distinct-values(`
`         document(...)//book/author)`
`return <author last-name={$a/last}>`
`      {for $b in document(...)//book`
`            [author=$a]`
`       order by $b/title`
`       return $b/title}`
`      </author>`
	For each book author, the author's last name and ordered book titles

`for $y in document(...)//book/@year,`
`let $b:=document(...)//book`
`where $y=$b/@year`
`return <sales year={$y}>`
`        {count($b)}`
`       </sales>`
	Book sales per year

Table 2.5: Example XQuery queries.

3

Schema-based web wrapping

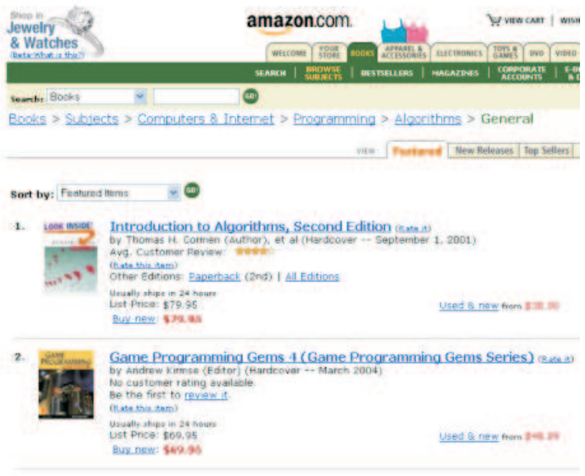
3.1 Introduction

In this chapter we deal with the information extraction from HTML pages by means of wrappers. Wrappers represent an effective solution to capture information of interest from a source-native format and encode such information into a machine-readable format suitable for further application-oriented processing.

Domain-specific extraction patterns, or rules, represent the basic requirement shared between wrappers and traditional IE systems. However, wrappers are designed for reliably extracting information from HTML documents in a structured way. Moreover, in most cases, a wrapper is able to associate an HTML page with an XML document. The wrapping task is typically accomplished by exploiting structural information and formatting instructions within documents in order to define delimiter-based *extraction rules*.

The schema of the information contained in the XML document, called *extraction schema*, is also considered in some wrapping approaches. However, unlike extraction rules, the schema is typically seen as a minor aspect of the wrapping task, since it is usually considered only in the wrapper design phase. In particular, a schema specifies how the output of extraction rules is to be mapped to some (XML) element types; mappings are usually defined in a declarative way (e.g. [65, 73, 79]), or in some cases (e.g. [95]) using a programmatic specification. However, in general, such element types are constrained to the structure of the output provided by the extraction rules, whereas the vice versa does not hold necessarily.

Most existing wrapping systems ignore the potential advantages coming from the exploitation of the extraction schema during the extraction process. The extraction schema can be used as both a guide and a means for recognizing, extracting and integrating semantically structured information. In particular, using the extraction schema simplifies the identification and discarding of irrelevant or noisy information and, most importantly, supports



Books > Subjects > Computers & Internet > Programming > Algorithms > General

1. Introduction to Algorithms, Second Edition (Rate it)
by Thomas H. Cormen (Author), et al (Hardcover -- September 1, 2001)
Avg. Customer Review: ★★★★
(Rate this item)
Other Editions: Paperback (2nd) | All Editions
Usually ships in 24 hours
List Price: $79.95
Buy new: $79.95
Used & new from [illegible]

2. Game Programming Gems 4 (Game Programming Gems Series) (Rate it)
by Andrew Kirmse (Editor) (Hardcover -- March 2004)
No customer rating available.
Be the first to review it.
(Rate this item)
Usually ships in 24 hours
List Price: $69.95
Buy new: $69.95
Used & new from [illegible]

Fig. 3.1: Excerpt of a sample *Amazon* page

the design of extraction rules to improve the wrapper accuracy and robustness. Furthermore, it simplifies the use of extracted information in several application scenarios such as data integration.

As a running example, consider an excerpt of *Amazon* page displayed in Fig. 3.1. We would like to extract the title, the author(s), the customer rate (if available), the price proposed by the Amazon site, and the publication year, for any book listed in the page. The relating schema is represented by the DTD shown in Fig. 3.2.

```
<!ELEMENT doc (store)>
<!ELEMENT store (book+)>
<!ELEMENT book (title, author+, (customer_rate | no_rate), price, year)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT customer_rate (rate)>
<!ELEMENT no_rate EMPTY>
<!ELEMENT rate (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT year (#PCDATA)>
```

Fig. 3.2: Extraction schema for Amazon wrappers

Schema-based wrappers allow the extraction of structured information with multi-value attributes (operator +), missing attributes (operator ?), and variant attribute permutations (operator |). This is particularly useful for Web wrapping, since page templates with disjunctions and optionals hold for many collections of Web pages. Moreover, many documents that reflect different semantics of data to be extracted may in principle be extracted from the same page conforming to the same extraction schema. For example, an XML document conforming to the above DTD may have many elements `author`, or an element `customer_rate` that occurs alternatively to element `no_rate`.

Exploiting the extraction schema not only enables an effective use of the extracted documents for further successful processing, but also allows the specification of structured, simple extraction rules. For instance, to extract `customer_rate` from a book, a standard approach should express a rule extracting the third row of a book table only if this row contains an image displaying the "rate". The presence of the extraction schema enables the definition of two simple rules, one for `customer_rate` and one for its sub-element `rate`: the former extracts the third row of the book table, while the latter extracts an image.

The possibility of specifying simpler rules plays an important role with respect to the issue of *wrapper maintenance.* Indeed, if the wrapper designer is too tight in the specification of extraction rules, the resulting wrapper may become sensitive to even small changes in the source HTML documents, and frequent updates to the extraction rules are needed to guarantee that the wrapper continues to work on the changed pages. Moreover, extraction rules in the original wrapper may also contain redundant conditions. For example, it may happen that the extraction rule for element `book` requires the presence of an emphasized heading, corresponding to the book title, whereas the extraction rule for `title` selects all the emphasized headings. Removing the above constraint from the extraction rule for `book` makes wrapper maintenance easier, since changes in the layout of the book title only require to be reflected in the extraction rule for `title`.

Maintaining wrappers may become a labor-intensive and error-prone task when high human involvement is needed. Thus, the problem of wrapper maintenance could be overcome by making wrappers *robust* with respect to structural changes occurring in the source HTML documents. From this perspective, the main effort should be defining an inductive mechanism that automatically produces extraction rules more "general" and, hence, resilient to changes in the document layout.

We address the problem of extracting information from Web pages by proposing a novel Web wrapping approach, which combines both extraction rules and extraction schema in wrapper generation and evaluation.

Specifically, we propose:

- the notion of *schema-based wrapper* and a clean declarative semantics for schema-based wrappers. Intuitively, this semantics guarantees that a part of a document is extracted only if the information contained satisfies the extraction schema.
- a wrapper evaluation algorithm, which runs in polynomial time with respect to the size of a source document if the extraction schema is not recursive. This algorithm is the core of a Web wrapping system, called *SCRAP - SChema-based wRAPper for web data* [40, 42].

- an inductive learning method to speed up the specification of schema-based wrappers and improve their robustness with respect to structural changes occurring in source HTML documents.

Experimental evaluation conducted on significant Web sites during a long-term period gives evidence that SCRAP wrappers are not only able to accurately extract data with the aid of extraction schema, but also resilient to minor changes that may occur in source Web pages. Moreover, generalizing SCRAP wrappers really improves the wrapper robustness to deal with more evident changes in page layouts.

3.1.1 Related work

Since Web documents consist of a mixture of markup tags and natural language text passages, extracting information from these documents is ever attractive and has been recognized as a significant, non-trivial task for Web data integration and management. Unfortunately, traditional IE tasks and methods are fit only for structure-free text. This has raised the need for extraction mechanisms suitable for the semistructured or fairly structured nature of Web sources.

Nowadays, the best way to perform information extraction from Web sources is provided by *wrappers* [70]. A wrapper is a program designed to extract specific contents of interest and deliver them in a structured representation. Web wrappers hence represent an effective solution for capturing information encoded into HTML pages and translate it into a (semi)structured and self-describing format, like XML, suitable for further application-oriented processing. Moreover, a Web environment imposes that wrappers are sufficiently flexible to deal with dynamic and unstable sources, which may contain ill-formed documents and frequently change their layout. Some applications also require that wrappers are able to efficiently perform “on the fly” in response to users’ queries.

Domain-specific extraction patterns represent the basic requirement shared between wrappers and traditional IE systems. However, the latter use linguistic patterns which are based on a combination of syntactic and semantic constraints: such patterns are usually not sufficient, or even not applicable, to Web documents. Indeed, Web sources often do not exhibit the rich grammatical structure linguistic patterns are designed to exploit. Moreover, NLP techniques are too slow to allow for on-line extraction from possibly a large number of documents.

In order to handle Web documents, wrapper generation systems exploit structure information and formatting instructions within documents to define *delimiter-based* extraction patterns. This new kind of pattern usually does not use linguistic knowledge, but in some cases—as we shall describe later—syntactic and semantic constraints can be combined with delimiters that bound the text to be extracted.

Depending on the degree of human effort required, wrapper generation can be accomplished by following three different approaches, namely manual, semi-automatic, and automatic generation.

Manual wrapper generation

Manual wrapper generation involves a human expert to analyze the document source and code ad-hoc extraction rules. Programming information extraction procedures by hand can be simpler for semistructured Web pages than for structure-free text. One of the first examples of manual construction of wrappers was provided by *TSIMMIS* (The Stanford-IBM Manager of Multiple Information Sources) [49, 57], a historic framework for accessing multiple information sources in an integrated fashion. In general, hand-coding implies a high level of knowledge of the structure of the documents being wrapped, thus may be tedious, time-consuming, and error-prone.

To speed up the manual construction of wrappers, there have been proposed tools for generating extraction patterns based on expressive grammars that describe the structure of a Web document. Unfortunately, the above mentioned disadvantages are in practice transmitted to the grammar specification. Furthermore, manually generated Web wrappers would require high maintenance costs, as the sources of interest are often in large number and their content and structure may vary significantly.

Automatic wrapper generation

Fully *automatic wrapper generation* is really supported by only a few systems. We mention here *RoadRunner* [30] and *ExAlg* [10]. By examining the structure of sample Web pages, these systems automatically generate a page template for the data to be extracted. This is mainly accomplished by exploiting methods for automatic structure detection and for separation of content from structure. Specifically, RoadRunner works by comparing the HTML structure of two (or more) given sample pages belonging to a same class, generating as a result a schema for the data contained in the pages. From this schema, a grammar is inferred which is capable of recognizing instances of the attributes identified for this schema in the sample pages (or in pages of the same class). To accurately capture all possible structural variations occurring on pages of a same page class, it is possible to provide more than two sample pages. All the extraction process is based on an algorithm that compares the tag structure of the sample pages and generates regular expressions that handle structural mismatches found between the two structures. In this way, the algorithm discovers structural features such as tuples, lists, and variations.

Automatic wrappers rely strictly on syntactic constraints and, in general, do not exploit any knowledge on the specific domain, thus they are not able to semantically label the extracted data. Other drawbacks are specific of system. For instance, RoadRunner may fail to generate the page template if HTML tags are contained in the source page as data values, and assumes that the page template is union-free. Our approach, instead, does not pose any assumption on the nature of textual data encoded in a page, and is able to specify an extraction schema with optionals and disjunctions. ExAlg supports templates with optionals and disjunctions, but handles only template-generated pages, that is pages that are regularly rendered using a common structured template by plugging-in values coming from a database source.

In general, fully automatic wrappers require a large number of sample input pages, which are structurally almost identical. Therefore, automatic wrapper generation approach is mainly justified for Web pages generated from an on-line database and having little structural deviations.

Semi-automatic wrapper generation: Wrapper induction

Semi-automatic wrapper generation refers to the broadest class of wrapping systems. In this class, the outstanding approach is represented by *wrapper induction*, which aims to generate extraction rules by using machine learning (ML) methods. In general, a wrapper induction method performs a search through the space of possible wrappers (hypothesis space) until the wrapper with the highest accuracy with respect to a training set of labeled examples is found. Extraction rules are thus learned from these examples. The accuracy of extraction rules usually relies on both the number and the "quality" of the examples. It is worthy noticing that wrapper induction would be really automatic if it did not require a training phase, which is typically human-supervised.

In wrapper induction, the problem of wrapper generation is mapped to one of *inductive learning*, which is accomplished through inductive inference (i.e. reasoning from specific instances to generalized rules). The form of training data and the way a learned theory is expressed determine the type of inductive learning method. Zero-order, or propositional, methods formulate theories in the form of production rules or decision trees that relate the class of an instance to its attribute values. However, these learners lack expressiveness as they cannot define hierarchical relationships, but can deal only with instances represented by attribute-value pairs.

Inductive logic programming (ILP) [83] performs induction based on first-order logic, and enables learning relational and recursive concepts, thus allowing the extraction of information from documents with more complex structures. An ILP method adopts either a *bottom-up* (generalization) or a *top-down* (specialization) approach to induction. A bottom-up method selects an example set of instances and learns a hypothesis to cover those examples, then generalizes the hypothesis to explain the remaining instances. A historic system called *GOLEM* [82] performs a greedy covering based on the construction of least general generalizations of more specific clauses. In general, bottom-up methods fare well with a very small training set, but induce only a restricted class of logic programs. By contrast, top-down methods can induce a larger class of programs, but need a relatively large number of (positive and negative) training examples. These methods start with the most general hypothesis available, making it more specific by introducing negative examples. The well-known *FOIL* [90] searches for a clause that covers some positive examples and no negative examples, then all the covered instances are removed from the training set until there are no positive examples in the training set.

Anyhow, independently from the kind of adopted inductive inference, ML-based wrappers must be tailored with respect to the type and degree of structure of the target information source. In the following, we give a short description of relevant methods and systems for semi-automatic wrapper generation.

Several wrapping systems have been developed to extract data specifically from Web pages that are generated on-line, and hence are regularly rendered or, at worst, fairly structured. Such systems produce delimiter-based extraction rules that do not exploit syntactic or semantic information.

ShopBot [35] is an agent designed to extract information from pages available on e-commerce sites. ShopBot works in two main stages by combining heuristic search, pattern matching, and inductive learning techniques. In the first phase, a symbolic representation containing the product descriptions is learned for each site. The induced formats of product description are then subject to a ranking process to identify the best description format available. As a main limitation, ShopBot is unable to semantically label the extracted data.

WIEN (Wrapper Induction ENvironment) [68] is the earliest tool for inductive wrapper generation. WIEN operates only on structured text containing information organized in a tabular fashion, thus it does not deal with missing values or permutations of attributes. The system uses only delimiters that immediately precede or follow the data to be extracted (HLRT organization), by applying a unique multi-slot rule on all the documents of the source. In the attempt of automating the learning phase, WIEN exploits domain-specific heuristics to label the training documents, then learns the best covering HLRT wrapper using bottom-up induction.

SoftMealy [59] is a system based on non-deterministic finite-state automata, and it is conceived to induce wrappers for semistructured pages. In contrast to WIEN, SoftMealy can handle missing values, allowing the use of semantic classes and disjunctions. The system produces extraction rules by means of a bottom-up inductive learning algorithm that uses labeled training pages represented by an automaton defined on all the permutations of the input data. The states and the state transitions of the automaton correspond to the data to be extracted and the resulting rules, respectively. SoftMealy is more expressive than WIEN, although it has an efficiency limit as it considers each possible permutation of the input data.

Multi-slot rules are not suitable for extraction tasks on documents with multiple levels of embedded data. *STALKER* [84] is a wrapper induction system that addresses this issue by performing *hierarchical extraction*. The key idea is the introduction of a formalism, called Embedded Catalog Tree (ECT), to describe the hierarchical organization of the documents. An ECT represents the structure of a page as a tree in which the root corresponds to the complete sequence of tokens and each internal node is associated with portions of the complete sequence. Each node specifies how to identify the associated subsequences, using Simple Landmark Grammars capable of recognizing the delimiters of such sequences. Since the ECT is an unordered tree, STALKER does not rely on the order of items, therefore can extract information from documents containing missing items or items that may appear in varying order. Despite its high flexibility, STALKER seems to suffer the generation of complex extraction rules when, in many cases, a simple extraction rule based on the order of items would be sufficient to extract the desired information.

More sophisticated systems for wrapper generation are able to work on different types of text, including unstructured text. In this case, delimiter-based patterns can be combined with linguistic ones. These systems are closely related to ILP methods.

RAPIER [23] is a wrapper induction system for (semi)structured pages. It works on pairs of pages and user-provided templates with the information to be extracted. Then, it learns (using GOLEM) single-slot extraction patterns that include constraints on syntactic information (e.g. the output of a part-of-speech tagger) and semantic class information (e.g. hypernym links from WordNet) of the items surrounding a candidate filler. An extraction pattern is composed of a filler pattern, which describes the structure of the target information, a pre- and a post-filler patterns, which play the role of left and right delimiters, respectively.

SRV (Sequence Rules with Validation) [44, 45] is a top-down relational learning algorithm that generates single-slot first-order logic extraction rules for (semi)structured pages. Besides a training set of labeled pages, SRV takes as input a set of simple and relational features, such that a simple feature maps a token to a categorical value (e.g. length, orthography, part-of-speech, lexical meaning), while relational features represent syntactic/semantic links among tokens. The algorithm starts with the whole set of examples, including negative examples (i.e. fragments that are not labeled as slot-filler instances). Induction proceeds as in FOIL: iteratively, all positive examples that match a currently learned rule—a rule that covers only positive examples or cannot be further specialized—are removed from the training set. SRV generates extraction rules using all the possible instances, regarded as sentences, in the document. Each instance is presented to a classifier, and associated with a score indicating its suitability as a filler for the target slot. Extracted rules are highly expressive and can handle missing items and different item orderings.

Unlike all the above systems, the WHISK system [98] is able to deal also with structure-free text when used in conjunction with a syntactic analyzer and a semantic classifier. Given a training set of pages, WHISK generates, in a top-down fashion, special regular expressions that are used to recognize the context of the relevant instances (sentences) and the delimiters of such instances. Despite its completeness and ability in handling missing values, WHISK needs to be trained on documents containing all the possible orderings of fields.

In general, the wrapper induction approach suffers from the negative theoretical results on the expressive power of learnable extraction rules, which are mostly specified by regular expressions. Moreover, this approach still requires manual construction of training data for each information source, thus inherits some of the problems as the manual wrapper generation approach. Also, using examples to generate wrappers relies often on an implicit knowledge or a predefined description of the structure of the page containing the target data.

Semi-automatic wrapper generation: Visual support tools

Another approach to semi-automatic wrapper generation consists in providing support tools to assist wrapper designing. These tools are usually equipped with demonstration-oriented graphical interfaces in which the user visually specifies what information the system has to extract. This allows users having even limited familiarity of HTML to easily work with the wrapper generator.

An example of wrapper generation with visual support is *XWRAP* [73, 58], which can be classified both as unstructured and semistructured data wrapper generation. A syntax tree describing the HTML structure of a target page is presented to the user for browsing and selecting portions of interest. Heuristic analysis of HTML is used to identify which parts of the page can be headings. The extracted data is formatted using XML, but may derive structure from the HTML source. XWRAP suffers from several disadvantages. It lacks powerful semi-automatic generalization mechanisms for the specification of more similar patterns at once while marking only a single pattern of the desired type. It does not support sufficient facilities for imposing inherent or contextual conditions to an extraction pattern. The employed levels of structure description and the automatic hierarchical structure extractor severely limit the expressiveness of extraction patterns. Finally, the XWRAP approach suffers from an explicit use of the HTML syntax; by contrast, our approach allows a user to focus on the target structure of interest without having concern of the structure underlying a whole source HTML page.

W4F [95] is an advanced programming environment for an SQL-like wrapper language, called *HEL* (HTML Extraction Language), designed for semistructured documents. HEL query construction is supported by a specialized visual extraction wizard. However, except for certain trivial extraction tasks, the wizard is not able to generate a full query, which instead must be hand-coded by generalizing tree paths generated by the wizard and adding further HEL language constructs. HEL is clearly more expressive than, for example, the visual pattern definition method of XWRAP, but is a rather complex language hard to use. W4F also contains a wizard that helps the user to define a translation of the query output into XML, and a visual interface for testing and refining the wrapper interactively before its deployment.

In *DEByE* [69], information extraction is fully based on examples specified by a user, which is guided by a graphical interface adopting a visual paradigm based on nested tables. Extraction patterns, generated from the examples, are used to perform bottom-up extraction which has been proved very effective on some data collections. Extracted data is outputted in XML.

The most widely known tool for visual and interactive wrapper generation is perhaps *Lixto* [13]. In this system, extraction patterns operate on the HTML parse tree of a page following two paradigms of tree and string extraction. For tree extraction, Lixto identifies elements with their corresponding tree paths, which are generalized based upon visual example selection. For extracting strings (from the text of leaf nodes), both regular expressions and predefined concepts can be used. Lixto enables the generation of highly expressive wrappers, thanks to hierarchical extraction—based on surrounding landmarks, HTML attributes, semantic and syntactic concepts—and a number of advanced features, such as disjunctive pattern definition, pattern specialization, pattern hierarchy construction by aggregating information from various Web pages, and recursive wrapping [12].

The aforementioned features are internally reflected by a declarative extraction language called *Elog*, which uses Datalog-like logic syntax and semantics. It is ideally suited to represent and successively increase the knowledge about extraction patterns described by designers. Elog is flexible, intuitive and easily extensible. In [51], the expressive power of a kernel fragment of Elog has been studied, and it has been demonstrated that it captures monadic second-order logic, hence is very expressive while at the same time easy to use due to visual specification.

The visual wrapper generation approach has recently attracted significant attention mainly due to the increasingly availability of Web browser components that allow users to highlight representative examples of an extraction pattern. Moreover, expertise in wrapper coding is not required at this stage. On the other hand, visual wrapper generators do not induce the structure of a Web source, thus they must be updated according to eventual changes occurring in the source.

Schema-based wrapping

Most existing wrapper designing approaches focus mainly on the specification and use of powerful extraction languages, whereas they ignore that the target structure for the data to be extracted, or *extraction schema*, could be profitably exploited during wrapper designing as well as the extraction process. For instance, a wrapper based on extraction schema is more likely to be capable of recognizing and discarding irrelevant or noisy data from extracted documents, thus improving the extraction accuracy.

Some full-fledged systems describe a hierarchical structure of the information to be extracted [13, 84], and they are mostly capable of specifying constraints on the cardinality of the extracted sub-elements. However, no such system allows complex constraints to be expressed.

Two preliminary attempts of exploiting extraction schema have been proposed in the information extraction [65] and wrapping [79, 80] research areas. In the latter work, schemas represented as tree-like structures do not allow alternative subexpressions to be expressed. Moreover, a heuristic approach is used to make a rule fit to other mapping rule instances: as a consequence, rule refinement based on user feedback is needed. In [65], DTD-style extraction rules exploiting enhanced content models are used in both learning and extracting phases.

In *NoDoSE* [4] the user may mark portions in the whole input document and then visually map the contents to some output, possibly hierarchical, structure. Actually, there is no need for a wrapper to understand the structure of the whole source page: what a user wants to extract from a Web document should be carefully specified at the time of wrapper generation. The most noticeable distinction between our system and all preexisting wrapping systems, including NoDoSE, is the way the schema is used: not only as the target structure of the extracted data but, most importantly, as a valuable, key means to guide and control both the wrapper design and evaluation stages.

[16, 37, 36] are related to a particular direction of research: turning the schema matching problem to an extraction problem based on inferring the semantic correspondence between a source HTML table and a target HTML schema. This approach differs from the previous ones related to schema mapping since it entails elements of table understanding and extraction ontologies. In particular, table understanding strategies are exploited to form attribute-value pairs, then an extraction ontology performs data extraction. However, the approach relies on the manual creation of an ontology by a domain expert, and may fail when dealing with Web sources related to a domain where an ontology is hard to be created.

Wrapping maintenance

In comparison to wrapper design, wrapper maintenance has received less attention. This is an important problem, because even slight changes in the Web page layout can break a wrapper that uses landmark-based rules and prevent it from extracting data correctly. Two major problems are relevant to reduce the human effort required to maintain or repair a wrapper: *i)* automatically checking wrapper validity and *ii)* providing examples that allow for automatically inducing a new correct wrapper.

[27] presents an automatic-maintenance approach to repair wrappers using content features of extracted information. Two sequential maintenance

steps are distinguished: information extraction recovery and wrapper repairing. The former aims at extracting as much information as possible from the new pages; the latter is invoked only if the extraction went successfully, otherwise the wrapper cannot be repaired automatically and it is necessary the user intervention.

In [67] the "wrapper verification" problem is studied, which consists in checking if a wrapper stops extracting correct data. To verify a wrapper, the proposed system invokes it on a page returned in response to a given query, and also on an earlier page obtained by the same query when the wrapper was known to be correct. The system declares the wrapper is not broken if the two outputs are similar with respect to a set of predefined features.

Among recent works dealing with the problem of repairing, or maintaining wrappers, [78] is a schema-guided wrapping approach that proposes a maintenance solution based on some assumptions on the features of extracted data. More precisely, syntactic and hypertextual features and annotations of extracted data items are assumed to be preserved after page variations. Such features are then exploited to recognize the desired information in the changed pages and induce new rules.

In [71] both wrapper verification and wrapper maintenance problems are addressed. Similarly in [67], a vector of features of earlier data (correct data) and a vector of features of new data are compared, and the wrapper is judged to be correct if the two vectors are statistically similar; otherwise, it is judged broken and a wrapper maintenance phase is employed. The algorithm for wrapper maintenance exploits patterns learned during the ordinary working of the wrapper for identifying examples of data fields in the new pages. The final step is to re-induce the wrapper using as training examples the data fields discovered by the algorithm.

All described approaches are based on features of data extracted to be used for detecting data fields in the new pages. However, such approaches may fail if new data is heavily changed.

[91] proposes a strategy to identify previously wrapped examples in pages whose structure has been updated. The authors assume that a good number of unlabeled examples contained in the original page are still present in the updated page. Actually, this is a major limitation of that approach; indeed, a wrapper would fail with respect to sources where the above assumption does not ever hold: this is the case of pages whose contents are time-changing with respect to a fixed query, for example "Search for the top-n weekly sellers books".

3.2 Preliminaries

3.2.1 XML DTDs

Any XML document can be associated with a document type definition (DTD) that defines the structure of the document and what tags might be used to encode the document. We provide a simplified formal definition of DTD, which is sufficiently general and, at the same time, well-suited to meet most needs in this thesis. In particular, we refer to DTDs that do not contain attribute list; as a consequence, we consider a simplified version of XML documents, whose elements have no attributes.

A DTD is a tuple $\mathcal{D} = \langle El, P, e_r \rangle$ where: *i)* El is a finite set of element names, *ii)* P is a mapping from El to *element type definitions*, and *iii)* $e_r \in El$ is the root element name. An element type definition α is a one-unambiguous regular expression [21] defined as follows:[1]

- $\alpha \rightarrow \ \alpha_1 \ || \ \alpha_2$,
- $\alpha_1 \rightarrow \ (\alpha_1) \ || \ \alpha_1 \,|\, \alpha_1 \ || \ \alpha_1, \alpha_1 \ || \ \alpha_1? \ || \ \alpha_1 * \ || \ e$,
- $\alpha_2 \rightarrow$ `ANY` || `EMPTY` || #`PCDATA`,

where $e \in El$, #`PCDATA` is an element whose content is composed of character data, `EMPTY` is an element without content, and `ANY` is an element with generic content. An element type definition specifies an element-content model that constrains the allowed types of the child elements and the order in which they are allowed to appear.

The "one-unambiguous" property for a regular expression allows for determining uniquely which position of a symbol in the expression should match a symbol in a target word, without looking beyond that symbol in the target word. For this reason, it is worth emphasizing that there is only one way by which a string can be parsed using a content model.

Given a DTD $\mathcal{D} = \langle El, P, e_r \rangle$, an element chain over $\mathcal{D}$ is a sequence of (not necessarily distinct) elements $e_0, \ldots, e_n$ belonging to El, such that $e_0 = e_r$ and e_i is a child of e_{i-1}, for each $i \in [1..n]$. A DTD $\mathcal{D}$ is said to be *recursive* if there exist an element chain $e_0, \ldots, e_n$ and indexes $i, j \in [0..n]$, with $i \neq j$, such that $e_i = e_j$. Two elements e', e'' are said to be *mutually recursive* if there exist an element chain $e_0, \ldots, e_n$ and indexes $i, j, k \in [0..n]$, with $i \leq k \leq j$ and $i < j$, such that $e' = e_i = e_j$ and $e'' = e_k$. An element chain $e_0, \ldots, e_n$ is said to be *non-recursive* if there not exist two indexes $i, j \in [0..n]$, with $i \neq j$, such that $e_i = e_j$. Let $C_{\mathcal{D}}$ denote the set of all the non-recursive element chains over $\mathcal{D}$. The maximum nesting level of $\mathcal{D}$ (denoted as *depth*$(\mathcal{D})$) is the maximum length of an element chain in C, that is $depth(\mathcal{D}) = max_{c \in C_{\mathcal{D}}} |c|$.

Given a DTD $\mathcal{D} = \langle El, P, e_r \rangle$ and an element $e \in El$, we denote as $N(e)$ the set of element names in $P(e)$. Moreover, we denote as $C(El)$ the maximum number of child elements of an element belonging to El, that is $C(El) = max_{e \in El} |N(e)|$.

3.2.2 Ordered regular expressions

The application of a wrapper to a source document may produce several candidate extracted documents. However, an order among such candidates

[1] Symbol || denotes different productions with the same left part. Here we do not consider *mixed content* of elements [105].

is required to identify the preferred extracted document. This requirement can be successfully satisfied by exploiting an extension of regular expressions where a *partial order* is defined among strings.

Definition 3.1 (Partially ordered language). *A* partially ordered language *on a given alphabet Σ is a pair $\langle L, >_L \rangle$, where $L \subseteq \Sigma^+$ is a (standard) language over Σ and $>_L$ is a partial order on the strings of L.*

Partially ordered regular expressions [41] are defined by adapting classic operations for standard languages to partially ordered languages. In particular, operations such as *prioritized union*, *concatenation*, and *prioritized closure* can be defined for partially ordered languages.

Prioritized union. Let $O_1 = \langle L_1, >_{L_1} \rangle$ and $O_2 = \langle L_2, >_{L_2} \rangle$ be two partially ordered languages. The prioritized union of O_1 with O_2, denoted by $O_1 \oplus O_2$, is a language $\langle L_3, >_{L_3} \rangle$ such that $L_3 = L_1 \cup L_2$ and $>_{L_3}$ is defined as follows:

- given two strings $a, b \in L_1$, if $a >_{L_1} b$ then $a >_{L_3} b$;
- given two strings $a, b \in L_2$, if $a >_{L_2} b$ and $b \notin L_1$ then $a >_{L_3} b$;
- given two strings a, b, if $a \in L_1$, $b \in L_2$ and $b \notin L_1$ then $a >_{L_3} b$.

The rationale behind the prioritized union $O_1 \oplus O_2$ is that the strings of O_1 are "preferred" with respect to the strings of O_2. Note that the prioritized union operator is not commutative (i.e. $O_1 \oplus O_2 \neq O_2 \oplus O_1$).

Concatenation. Let $O_1 = \langle L_1, >_{L_1} \rangle$ and $O_2 = \langle L_2, >_{L_2} \rangle$ be two partially ordered languages. The concatenation of O_1 with O_2, denoted by $O_1\, O_2$, is a language $\langle L_3, >_{L_3} \rangle$, such that $L_3 = L_1\, L_2$ and $>_{L_3}$ is defined as follows. Let a and b be two strings; then, $a >_{L_3} b$ if, for each $b_1 \in L_1$ and for each $b_2 \in L_2$ such that $b_1 b_2 = b$, there exist $a_1 \in L_1$ and $a_2 \in L_2$ such that $a_1 a_2 = a$ and either *i)* $a_1 >_{L_1} b_1$ or *ii)* $a_1 = b_1$ and $a_2 >_{L_2} b_2$.

Prioritized closure. The prioritized (positive) closure of a partially ordered language O, denoted by $O^{\triangleright}$, is defined by using concatenation and prioritized union of ordered languages, that is $O^{\triangleright} = \bigoplus_{i=\infty}^{0} O^i$. Observe that $\bigoplus$ is not commutative and $O^{\triangleright}$ is equal to $(... \oplus O^{i+1} \oplus O^i \oplus ... \oplus O^1 \oplus \{\epsilon\})$. The main difference between the standard closure operator $+$ and operator $\triangleright$ is that the latter gives preference to longer strings.

Definition 3.2 (Ordered regular expression). *Let Σ be an alphabet. The* ordered regular expressions *over Σ and the sets that they denote are defined recursively as follows:*

1. *$\emptyset$ is a regular expression and denotes the empty language $\langle \emptyset, \emptyset \rangle$;*
2. *for each $a \in \Sigma$, a is a regular expression and denotes the language $\langle \{a\}, \emptyset \rangle$;*
3. *if α_1 and α_2 are regular expressions denoting the languages $L(\alpha_1)$ and $L(\alpha_2)$, respectively, then $\alpha_1 + \alpha_2$ denotes the language $L(\alpha_1) \oplus L(\alpha_2)$, $\alpha_1 \alpha_2$ denotes the language $L(\alpha_1) L(\alpha_2)$, and $\alpha_1^{\triangleright}$ denotes the language $L(\alpha_1)^{\triangleright}$.*

Proposition 3.3. *Let α be a one-unambiguous ordered regular expression. The language $L(\alpha)$ is linearly ordered.*

3.3 The schema-based wrapping framework

A *schema-based wrapper* is essentially composed of: *i)* an extraction schema and *ii)* a set of extraction rules. Formally:

Definition 3.4 (Schema-based wrapper). *Let $\mathcal{D} = \langle El, P, e_r \rangle$ be a DTD, $\mathcal{R}$ be a set of extraction rules, and $w : El \times El \to \mathcal{R}$ be a function that defines a one-to-one correspondence between each pair of elements $e_i, e_j \in El$ and a rule $r \in \mathcal{R}$. A* schema-based wrapper *is defined as $\mathcal{WR} = \langle \mathcal{D}, \mathcal{R}, w \rangle$.*

We assume any source HTML document is represented by its parse tree, also called as XHTML document, although a tree-based model for HTML data is not a strong requirement and could be easily relaxed. Our desired extraction behavior is that each extraction rule applies to a (parent) sequence of nodes to yield a (target) sequence of sequences of nodes.

Definition 3.5 (Extraction rule). *Given an HTML parse tree doc and a sequence s_p of nodes in doc, an* extraction rule *r is a function associating s_p with a sequence S of node sequences in doc such that:*

1. *for each pair of sequences $s' = [n'_1, \ldots, n'_k]$ and $s'' = [n''_1, \ldots, n''_h]$ in S, there must be $n'_1 \neq n''_1$ or $n'_k \neq n''_h$;*
2. *for each sequence s in S ($s \in S$), there not exist two nodes $n, n' \in s$ such that n is an ancestor of n';*
3. *for each sequence $s \in S$, there not exist two nodes $n \in s$ and $n' \in s_p$ such that n' is an ancestor of n.*

The above notion of extraction rule is quite close to the notion of extraction filter introduced in Lixto [13]. However, unlike Lixto, our extraction rules allow non-contiguous portions to be extracted from a single HTML page. This is particularly useful when semantically cohesive pieces of information are scattered across the page and, instead of extracting them separately, we want to put together them as a whole. HEL rules [95] permit to identify and extract surrounding pieces in a structured way, but do not support disjunctions.

In Def. 3.5, condition *(1)* states that a rule cannot yield overlapping sequences, whereas condition *(2)* states that an extraction rule cannot yield a sequence containing pairs of elements with an ancestor-descendant relationship. Also, condition *(3)* states that the output of an extraction rule is limited to the nodes that are descendants of the input nodes, i.e., it extracts information in a hierarchical way. These restrictions are assumed by most of the state-of-the-art wrapping approaches.

In the following, we refer to *non-self extraction rule* as an extraction rule r such that, for each node sequence s_p, there not exists a sequence $s \in r(s_p)$ in which there is a node $n \in s$ which belongs to s_p as well.

It is worth emphasizing that our schema-based wrapping is substantially conceived to be an extension of hierarchical wrapping in which the extraction schema can be profitably exploited. That is, the extraction of the desired information proceeds in a hierarchical way (such as, e.g., [13, 84]) but under the control of the extraction schema.

Roughly speaking, a wrapper associates the root element e_r of the extraction schema with the root of the HTML parse tree to be processed, then it builds the content of e_r by exploiting the extraction rules to identify the sequences of nodes that should be extracted. In other terms, once an element e has been associated with a sequence s of nodes of the source document, an extraction rule r is applied to s to identify the sequences that can be associated with the children of e.

3.3.1 XPath extraction rules

In principle, our schema-based wrapping approach does not rely on a specific form of extraction rules. Nevertheless, in order to make the approach "XML-enabled" (i.e. extracted data is modeled as an XML document), we propose an effective implementation of extraction rules based on the *XPath* language [110].

The primary XPath syntactic construct is the *expression*. An expression is evaluated to yield a sequence of nodes, that is an ordered collection of nodes without duplicates. Nodes in the sequence are ordered according to the source document parsing (document order).[2]

The evaluation of an XPath expression with variables occurs with respect to a *context* and a *variable binding*. A context refers to a sequence of nodes, and variable bindings represent mappings from variable names to sequences of nodes. Given a variable binding θ and a variable name $\$v$, we denote with $\theta(\$v)$ the sequence associated to $\$v$ by θ. Moreover, given a variable binding θ and a variable name $\$v$, if θ does not associate $\$v$ to any sequence then the application of θ to $\$v$ returns $\$v$, i.e., $\theta(\$v) = \$v$. Finally, the application of a variable binding θ to a sequence of nodes s returns s itself, i.e., $\theta(s) = s$. Given two disjoint variable bindings θ_1 and θ_2, we denote with $\theta_1 \circ \theta_2$ the variable binding obtained composing θ_1, with θ_2, i.e. $\theta_1 \circ \theta_2$ applied to a variable name $\$s$ returns $\theta_1(\theta_2(\$v))$. Observe that, since θ_1 and θ_2 are disjoint, for each variable name $\$v$ we have $\theta_1(\theta_2(\$v)) = \theta_2(\theta_1(\$v))$.

Given an XPath expression p, a set of variable names $\{\$v_1, \ldots, \$v_n, \$c, \$t\}$ and a variable binding θ for $\{\$v_1, \ldots, \$v_n, \$c, \$t\}$, the application of θ to p ($\theta(p)$) yields an XPath expression obtained from p by replacing every occurrence of a variable name $\$v_i$ in p with $\theta(\$v_i)$, for each $i \in [1..n]$.

We can now define the evaluation of an XPath expression with variables with respect to a variable binding θ and a node sequence (XPath context) s in terms of the standard XPath semantics. Given an XPath expression p, an

[2] In the standard XPath semantics, the evaluation of an XPath expression yields a node-set (an unordered collection of nodes without duplicates). However, several applications using XPath-based languages (e.g. XQuery) employ sequences of nodes ordered with respect to the document order. We follow this behavior.

XHTML document *doc*, a sequence of nodes s, and a variable binding θ, we denote with $p(s, \theta, doc)$ the sequence of nodes yielded by evaluating $\theta(p)$ on *doc*, starting from s.

A key role in XPath extraction rules is played by XPath predicates. An *XPath predicate* takes a *context* sequence, i.e. a sequence of nodes, as input and applies an XPath expression to yield *target* node sequences. Specifically, we distinguish two kinds of XPath predicates: sequence XPath predicates ($\rightarrow$) and subsequence XPath predicates ($\twoheadrightarrow$). XPath predicates are used to define XPath atoms, using an infix notation.

Definition 3.6 (XPath Atoms). *Given a set* $\{\$v_1, \ldots, \$v_n, \$c, \$t\}$ *of variables and an XPath expression* p *using the variables* $\$v_1, \ldots, \$v_n$*, a* sequence XPath predicate *is denoted as* $\$c : p \rightarrow \$t$*. Moreover, a* subsequence XPath predicate *is denoted as* $\$c : p \twoheadrightarrow \$t$*.*
Given an XHTML document doc and a variable binding θ*, an XPath predicate* $\$c : p \rightarrow \$t$ *is true with respect to* θ *if* $\theta(\$t) = p(\theta(\$c), \theta, doc)$*. Analogously, a subsequence XPath predicate* $\$c : p \twoheadrightarrow \$t$ *is true with respect to* θ *if* $\theta(\$t)$ *is a subsequence of* $p(\theta(\$c), \theta, doc)$*.*

Given two node sequences $s' = [n'_1, \ldots, n'_k]$ and $s'' = [n''_1, \ldots, n''_h]$, we say that s' *precedes* s'' ($s' \prec s''$) if there exists an index i such that $n'_i < n''_i$ and, for each $j < i$, $n'_j = n''_j$, or if s' is a strict suffix of s''.

Given an XHTML document *doc*, a variable binding θ, and a subsequence XPath predicate $\$c : p \twoheadrightarrow \$u$, we denote with $eval(\$c : p \twoheadrightarrow \$u, \theta)$ the sequence of node sequences $[s_1, \ldots, s_k]$ such that $s_i \prec s_j$, for each $i < j$, and $\$c : p \twoheadrightarrow \$u$ is true with respect to $\theta \circ \{\$u/s_i\}$, for each $i \in [1..k]$.

XPath predicates are the basis of extraction filters which, in turn, are the constituent elements of extraction rules. An *XPath extraction filter* is defined over a target predicate and a conjunction of atoms, referred to as *condition atoms*, that act as filter conditions on the target predicate.

Definition 3.7 (XPath extraction filter). *Given a set of variables* $\{\$v_1, \ldots, \$v_n, \$u\}$*, an* XPath extraction filter *is defined as a tuple* $f = \langle tp, \mathcal{P} \rangle$*, where:*

- *tp is a target predicate, that is a subsequence XPath predicate defining variable* $\$u$ *on the empty set of variables;*
- $\mathcal{P}$ *is a conjunction of condition atoms defined on variables* $\{\$v_1, \ldots, \$v_n, \$u\}$*.*

Atoms in $\mathcal{P}$ allow the specification of conditions that reflect structural and content constraints to be satisfied from subsequences obtained by applying f to a sequence s, i.e. $f(s)$. In principle, $\mathcal{P}$ may be formed by both XPath and built-in atoms. The latter are particularly important as they capture the designer needs for evaluating complex structural relations between elements, or for manipulating regular expressions in order to extract specific substrings within text elements. Several syntactic and semantic predicates have been

Table 3.1: Sample types of extraction and condition predicates

predicate type	*arguments*	*description*
XPath predicate	*expr*, *context*	returns the result of evaluation of *expr* applied to the *context*
After	*context*, e_s, e_t	returns the elements e_t placed after e_s with respect to the *context*
Before	*context*, e_s, e_t	returns the elements e_t placed before e_s with respect to the *context*
TextHandler	*regexpr*, *context*	returns the result of evaluation of *regexpr* applied to the *context*

defined to be plugged in the schema-based wrapping framework, in order to extend the expressiveness of extraction rules. Table 3.1 summarizes the features of some main predicates used in our schema-based wrapping system.

Besides XPath extraction filters, our extraction rules may use filters that specify restrictive constraints on the size of the extracted sequences, named *external filters*. Before introducing such filters, we need to briefly explain how inclusion between node sequences is here intended. Given two node sequences $s' = [n'_1, \ldots, n'_k]$ and $s'' = [n''_1, \ldots, n''_h]$, we say that s'' *includes* s' ($s' \sqsubset s''$) if and only if

- $n''_1 < n'_1$ and $n''_h \geq n'_k$, or
- $n''_1 = n'_1$ and $n''_h > n'_k$.

We consider two kinds of external filters:

- an *absolute size condition* filter as, which is specified by bounds (min, max) on the size of a node sequence s, that is $as(s)$ is true if $min \leq size(s) \leq max$;
- a *relative size condition* filter rs, which is specified by policies {minimize, maximize, all}. In the case $rs =$ minimize (resp. $rs =$ maximize) the meaning is as follows: given a sequence S of node sequences and a sequence $s \in S$, $rs(s, S)$ is true if there not exists a sequence $s' \in S, s' \neq s$, such that $s' \sqsubset s$ (resp. $s' \sqsupset s$). In the case $rs =$ all, given a sequence S of node sequences and a sequence of node s, $rs(s, S)$ is true if $s \in S$.

Let $EF = f_1 \vee \ldots \vee f_m$ be a disjunction of extraction filters and s be a node sequence. The application of EF to s returns a sequence of node sequences $EF(s) = [s_1, \ldots, s_k]$, such that

- $k \leq \sum_{i=1}^{m} |f_i(s)|$, where $|f(s)|$ denotes the number of node sequences contained in $f(s)$,
- $s_h \prec s_{h+1}$, for each $h \in [1..k\text{-}1]$,
- $s_h \neq s_t$, for each $h, t \in [1..k], t \neq h$,
- for each $h \in [1..k]$ there exists $i \in [1..m]$ such that $s_h \in f_i(s)$.

Definition 3.8 (XPath extraction rule). *An* XPath extraction rule *is defined as $r = \langle EF, as, rs\rangle$, where $EF = f_1 \vee \ldots \vee f_m$ is a disjunction of extraction filters, as and rs are external filters.*

According to Def. 3.5, the application of an XPath extraction rule $r = \langle EF, as,\ rs\rangle$ to a node sequence s yields a sequence of node sequences $r(s)$, which is constructed as follows.

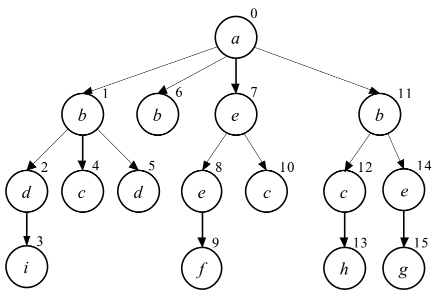
1. Each extraction filter $f = \langle tp, \mathcal{P}\rangle \in EF$ applied to s possibly selects a sequence S of node sequences. Specifically:
   a) the XPath expression contained in tp is evaluated on s and a node sequence $s' = [n_1, \ldots, n_t]$ is obtained. Starting from s' the sequence $S = [s_1, \ldots, s_m]$ of node sequences is built, such that each node sequence $s_i = [n_{i1}, \ldots, n_{in}] \in S$ is contained in s', that is there exists a subsequence $[n_{h1}, \ldots, n_{hn}]$ of s' such that $n_{ij} = n_{hj}$, for each $j \in [1..n]$;
   b) for each sequence $s_i \in S$, condition atoms in $\mathcal{P}$ are evaluated and s_i is removed from S if $\mathcal{P}$ is false on s_i. Specifically, $\mathcal{P}$ is true on a sequence s if, for each condition atom p in $\mathcal{P}$, there exists at least a node $n \in s$ such that p is true on n, otherwise $\mathcal{P}$ is false.
2. All the non-empty sequences of nodes selected by extraction filters are merged, that is the ordered sequence $EF(s)$ is computed.
3. A new sequence $S' = [s'_1, \ldots, s'_h]$, with $h \le |EF(s)|$, is derived from $EF(s)$ by filtering out all the sequences $s_i \in EF(s)$ such that $as(s_i)$ is false.
4. All the sequences $s'_i \in S'$ such that $rs(s'_i, S')$ is false are removed to finally obtain $r(s)$.

Observe that, in the respect of the first item of Def. 3.5, any two extraction filters should not be able to select two sequences, $s' = [n'_1, \ldots, n'_k]$ and $s'' = [n''_1, \ldots, n''_h]$, with $n'_1 = n''_1$ and $n'_k = n''_h$, that is two possibly different sequences having identical nodes at their respective bounds: in fact, both absolute and relative condition filters would not be sufficient to make a choice between a sequence or the other one. An appropriate solution to this special case can be provided by choosing the sequence s' such that $s' \prec s''$.

Example 3.9. Consider an extraction rule $r = \langle f_1 \vee f_2 \vee f_3, (1,2), \mathsf{maximize}\rangle$, where filters f_1, f_2 and f_3 are defined respectively as:

$$f_1 = \langle \$c : \texttt{//c} \twoheadrightarrow \$t,\ \{\$t : \texttt{[child::h]} \rightarrow \$v_1,\ \texttt{after(\$t,h,g)}\}\rangle,$$
$$f_2 = \langle \$c : \texttt{//e} \twoheadrightarrow \$t,\ \{\texttt{before(\$t,e,f)}\}\rangle.$$
$$f_3 = \langle \$c : \texttt{/d} \twoheadrightarrow \$t,\ \{\$t : \texttt{[child::i]} \rightarrow \$v_1\}\rangle.$$

Suppose that r is applied to the node sequence $s = [1, 7, 11]$ of the document tree shown in the figure below.



The sequence of nodes yielded by the evaluation of the XPath expression of the target predicate of f_1 is $[4, 10, 12]$*, thus the sequence of node sequences outputted by the target predicate is* $[[4], [4, 10], [4, 10, 12], [10], [10, 12], [12]]$. *Conditions in f_1 narrow this sequence in* $[[4, 10, 12], [10, 12], [12]]$*, as only node* 12 *satisfies the conditions, that is it is the only one having a child of type h followed by a node g with respect to the context* \$t*. The evaluation of the XPath expression of target predicate of f_2 produces* $[8, 14]$*, thus the target predicate of f_2 yields* $[[8], [8, 14], [14]]$*. The outputted sequence becomes* $[[8, 14], [14]]$ *by applying the relating condition, as* {`before($t,e,f)`} *is true on node* 14 *and it is false on node* 8*. The sequence of nodes yielded by the evaluation of the XPath expression of the target predicate of f_3 is* $[2, 5]$*, thus the target predicate of f_3 produces* $[[2], [2, 5], [5]]$*. The relating condition narrows the sequence in* $[[2], [2, 5]]$*, as only node* 2 *has a child of type d.*

Finally, the sequence associated with the disjunction of the filters is: $[[2], [2, 5], [4, 10, 12], [8, 14], [10, 12], [12], [14]]$*. The absolute external filter yields the sequence* $[[2], [2, 5], [8, 14], [10, 12], [12], [14]]$*, which is further simplified in* $[[2, 5], [8, 14], [10, 12]]$ *by the relative external filter.*

Now, suppose that filter f_3 in the example rule is replaced with the following:

$$f_3 = \quad \langle \$c : \texttt{/*[child::d]} \twoheadrightarrow \$t \rangle.$$

In this case, the evaluation of the target predicate of f_3 would yield $[[2]]$*. This points out that the result obtained using a native XPath predicate may be quite different from that obtained by using an XPath condition atom.* ◁

3.3.2 Wrapper semantics

In this section we introduce a clean declarative semantics for schema-based wrappers. The key element is the notion of *extraction model* for source HTML documents with respect to a given (i.e. already designed) wrapper. An extraction model is essentially a collection of information fragments hierarchically organized as *extraction events.* Intuitively, an extraction event models the extraction of a sequence of nodes by applying the appropriate extraction rule to a certain context.

Actually, not all the possible extraction events turn out to be useful for generating an XML document devoted to contain the extracted data. Extraction models enable the identification of those events that can be profitably exploited to extract the required information.

Extraction models

We refer to *information fragment* as an atomic piece of information, available from the source document, with an assigned semantic label. Formally, an information fragment η is a pair (e, s), where $e \in El$ is an element name and s is a sequence of nodes of the source XHTML document.

As previously discussed, each extraction rule is associated to a specific pair of element names, and applies to a node sequence to yield a sequence of node sequences. Information fragments represent input and output information for extraction rules. Given a wrapper $\mathcal{WR} = \langle \mathcal{D}, \mathcal{R}, w \rangle$ and two information fragments, $\eta_p = (e_p, s_p)$ and $\eta_t = (e_t, s_t)$, where e_t is a child element of e_p, η_t is extracted starting from η_p if and only if $r(s_p)$ yields a sequence of node sequences that contains s_t, where $r = w(e_p, e_t)$. We say that extracting a target information fragment η_t from a parent information fragment η_p generates an *extraction event*, denoted as $\varepsilon = \eta_t \vdash \eta_p$.

In order to make the construction of an output XML document easier, we devise a schema-based wrapper working on a tree of information fragments. In such a kind of tree, edges are just extraction events. A *(well-formed) extraction tree* $\mathcal{E}$ is a tuple $\langle E, <, \vdash, \eta_r \rangle$ such that:

- E is a partially ordered set of information fragments,
- $<$ is a partial order on E,
- η_r is the root information fragment,
- $\vdash$ is a binary relation representing extraction events that defines a tree of information fragments, whose root is η_r, and
- given $\eta_1, \eta_2 \in E$, if $\eta_1 < \eta_2$ then there exists an information fragment η such that $\eta_1 \vdash \eta$ and $\eta_2 \vdash \eta$.

Observe that the root information fragment is unique since an HTML parse tree has a unique root node (the document root). Moreover, as usual for ordered trees, we assume that a total order between information fragments is induced by the order between sibling information fragments in $\mathcal{E}$, that is we consider the order induced by the prefix visit of $\mathcal{E}$.

Given an extraction tree $\mathcal{E} = \langle E, <, \vdash, \eta_r \rangle$ and an information fragment η, we say that η belongs to $\mathcal{E}$ ($\eta \in \mathcal{E}$) if $\eta \in E$. Moreover, we denote with $\mathcal{E}(\eta) = \{\eta_t \mid \eta_t \in \mathcal{E} \wedge \ \eta_t \vdash \eta\}$ the partially ordered set containing all the information fragments in $\mathcal{E}$ that have η as parent information fragment.

An *ordered extraction tree* $\mathcal{E} = \langle E, <, \vdash, \eta_r \rangle$ is an extraction tree such that $<$ denotes a linear order on $\mathcal{E}(\eta)$, for each $\eta \in \mathcal{E}$. Let us introduce now some useful notations on ordered extraction trees.

We denote linearly ordered sets by lists of the form $[\eta_1, \ldots, \eta_n]$. Given an information fragment η and an ordered extraction tree $\mathcal{E}$, we denote with $elnames(\mathcal{E}(\eta))$ the sequence of element names corresponding to $\mathcal{E}(\eta)$. Formally, let $\mathcal{E}(\eta) = [(e_0, s_0), \ldots, (e_k, s_k)]$; then $elnames(\mathcal{E}(\eta)) = [e_0, \ldots, e_k]$. Moreover, we denote with $\mathcal{E}^*(\eta) \subseteq \mathcal{E}$ the subtree of $\mathcal{E}$ rooted in η.

Information fragments need to be characterized with respect to their conformance to a given regular expression specifying an element type. Given a regular expression α on an alphabet of element names, and an information fragment η, we say that $\mathcal{E}(\eta)$ is *valid* for α if $elnames(\mathcal{E}(\eta))$ spells α, i.e. the string formed by concatenating element names in $elnames(\mathcal{E}(\eta))$ belongs to language $L(\alpha)$.

Let $\mathcal{D} = \langle El, P, e_r \rangle$ be a DTD and $\mathcal{WR} = \langle \mathcal{D}, \mathcal{R}, w \rangle$ be a wrapper. An ordered extraction tree $\mathcal{E}$ is *valid* for an element name $e \in El$ if the following conditions hold:

- $P(e) = \texttt{EMPTY}$, or $P(e) = \#\texttt{PCDATA}$, or
- for each information fragment $\eta = (e, s) \in \mathcal{E}$:
  - $\mathcal{E}(\eta)$ is valid for $P(e)$, and
  - for each information fragment $(e_t, s_t) \in \mathcal{E}(\eta)$, $s_t \in w(e, e_t)(s)$, and
  - there not exist two information fragments (e_t, s_t) and (e_t, s'_t) in $\mathcal{E}(\eta)$ such that $(e_t, s_t) < (e_t, s'_t)$ and s_t does not precede s'_t in $w(e, e_t)(s)$.

An *extraction model* is an ordered extraction tree that conforms to the definition of all the elements in the extraction schema.

Definition 3.10 (Extraction Model). *Let $\mathcal{D} = \langle El, P, e_r \rangle$ be a DTD, $\mathcal{WR} = \langle \mathcal{D}, \mathcal{R}, w \rangle$ be a wrapper, doc be an XHTML document, and $\mathcal{E} = \langle E, <, \vdash, \eta_r \rangle$ be an ordered extraction tree. $\mathcal{E}$ is said to be an* extraction model *of doc with respect to $\mathcal{WR}$ (for short, $\mathcal{E}$ is an extraction model of $\mathcal{WR}(doc)$) if and only if*

- *$\eta_r = (e_r, s_r)$, and*
- *for each information fragment $\eta = (e, s) \in \mathcal{E}$, $\mathcal{E}(\eta)$ is valid for e.*

Example 3.11. Consider the Amazon *page displayed in Fig. 3.1, and suppose that such a page is subject to a wrapper whose extraction schema is the DTD shown in the Introduction. For the sake of simplicity, we focus on a portion of the parse tree associated with the page (Fig. 3.3), thus we consider only some extraction events, according to the portion of page we have chosen. Table 3.2 reports extraction events (parent and target information fragments) and associated extraction rules. We assume that* $(1, 1)$ *and* minimize *are adopted as default external filters.*

Looking at the page, we find that books are stored into one table, which is preceded by a simpler table containing a selection list. To extract the book table, we define an appropriate filter, f_{store}, to compose the extraction rule that triggers the extraction event for the book table. This event (ε_1) occurs when

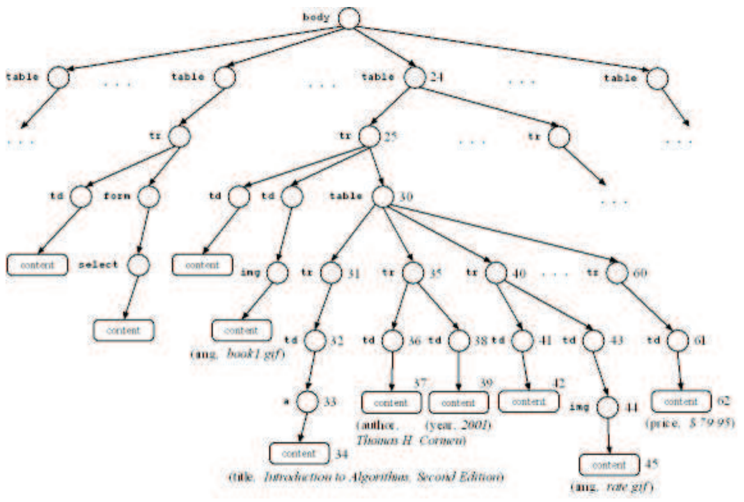


Fig. 3.3: Sketch of the HTML parse tree of page in Fig. 3.1

the information fragment defined on the book table (η_1) is extracted from the root information fragment (η_r).

Information about any book is stored into a separate table, which consists of two parts: the first part contains a book picture, while the second part is another table divided into a certain number of rows, one for each specific information about the book. Let us consider the first book instance, whose subtree is rooted in node 25 *of the parse tree. The book, which is identified by event $\varepsilon_2 = \eta_2 \vdash \eta_1$ using rule r_{book}, has information on title, (one) author, year, customer rate, and price. The set of information fragments that are extracted from of η_2 is built as $\mathcal{E}(\eta_2) = \{\eta_3, \eta_4, \eta_5, \eta_6, \eta_8, \eta_9\}$. Although information on customer rate is available from the first instance of book, we can observe that event $\varepsilon_8 = \eta_8 \vdash \eta_2$ happens for element* `no_rate`*: however, such an event cannot appear in the model, because $\mathcal{E}(\eta_2)$ would not be a valid content for an element of type* `book`*.*

It is worth noting that rules for extracting information on both availability and unavailability of customer rate have been intentionally defined as identical in this example. However, both kinds of extraction events occur only in books having customer rate, while only event for element `no_rate` *is extracted from books not having customer rate. This happens since it is not possible that an event for* `rate` *occurs as a child of an event for* `no_rate`*.* $\triangleleft$

An extraction model is implicitly associated with one only XML document, which is valid with respect to the extraction schema. Given a wrapper $\mathcal{WR} = \langle \mathcal{D}, \mathcal{R}, w \rangle$, an XHTML document *doc*, and an extraction model $\mathcal{E}$ of $\mathcal{WR}(doc)$, we define a function called *buildDoc* that takes $\mathcal{E}$ and an information fragment $\eta \in \mathcal{E}$ as input and yields the XML document portion relative to η. Let $text(s)$

Table 3.2: Extraction events and rules of a wrapper for *Amazon* pages

extrac. event	*parent info. fragment*	*target info. fragment*	*extrac. rule*
ε_1	η_r = (doc, [0])	η_1 = (store, [24])	$f_{store} = \langle \$doc$: /table $\twoheadrightarrow$ \$store, {\$store : preceding-sibling::*[1]//select $\longrightarrow$ \$list}$\rangle$ $r_{store} = \langle f_{store}, (1, 1)$, minimize$\rangle$
ε_2 ε_{10} ε_{17} ...	η_1 η_1 η_1 ...	η_2 = (book, [30]) η_{10} = (book, [68]) η_{17} = (book, [106]) ...	$f_{book} = \langle$\$store : /tr/table $\twoheadrightarrow$ \$book, {\$book : preceding-sibling::*[1]//img $\longrightarrow$ \$image}$\rangle$ $r_{book} = \langle f_{book}, (1, 1)$, minimize$\rangle$
ε_3 ε_{11} ...	η_2 η_{10} ...	η_3 = (title, [32]) η_{11} = (title, [70]) ...	$f_{title} = \langle$\$book : /tr/td $\twoheadrightarrow$ \$title, {\$title : //a $\longrightarrow$ \$anchor_text}$\rangle$ $r_{title} = \langle f_{title}, (1, 1)$, minimize$\rangle$
ε_4 ε_{12}	η_2 η_{10}	η_4 = (author, [36]) η_{12} = (author, [74])	$f_{author} = \langle$\$book : /tr/td $\twoheadrightarrow$ \$author, {\$author : .[contains(content.text(),'author') or contains(content.text(),'editor')] $\longrightarrow$ \$v}$\rangle$ $r_{author} = \langle f_{author}, (1, \infty)$, maximize$\rangle$
ε_5 ε_{13} ...	η_2 η_{10} ...	η_5 = (year, [38]) η_{13} = (year, [76]) ...	$f_{year} = \langle$\$book : /tr/td $\twoheadrightarrow$ \$year, {\$year : .[contains(content.text(),'year')] $\longrightarrow$ \$y}$\rangle$ $r_{year} = \langle f_{year}, (1, 1)$, minimize$\rangle$
ε_6 ε_{21} ...	η_2 η_{17} ...	η_6 = (customer_rate, [43]) η_{21} = (customer_rate, [120]) ...	$f_{crate} = \langle$\$book : /tr[3]/td $\twoheadrightarrow$ \$customer_rate$\rangle$ $r_{crate} = \langle f_{crate}, (1, 1)$, minimize$\rangle$
ε_7 ε_{22} ...	η_6 η_{21} ...	η_7 = (rate, [44]) η_{22} = (rate, [121]) ...	$f_{rate} = \langle$\$customer_rate : /img $\twoheadrightarrow$ \$rate$\rangle$ $r_{rate} = \langle f_{rate}, (1, 1)$, minimize$\rangle$
ε_8 ε_{15} ...	η_2 η_{10} ...	η_8 = (no_rate, [43]) η_{15} = (no_rate, [120]) ...	$f_{norate} = \langle$\$book : /tr[3]/td $\twoheadrightarrow$ \$no_rate$\rangle$ $r_{norate} = \langle f_{norate}, (1, 1)$, minimize$\rangle$
ε_9 ε_{16} ...	η_2 η_{10} ...	η_9 = (price, [61]) η_{16} = (price, [99]) ...	$f_{price} = \langle$\$book : /tr/td $\twoheadrightarrow$ \$price, {\$price : .[contains(content.text(),'Buy new')] $\longrightarrow$ \$p}$\rangle$ $r_{price} = \langle f_{price}, (1, 1)$, minimize$\rangle$

denote the concatenation of the string values of the nodes in a sequence s, and let symbol '+' indicate the concatenation of strings. For any information fragment $\eta = (e, s)$, $buildDoc(\mathcal{E}, \eta)$ is recursively defined as follows:

- if $P(e) =$ `EMPTY` then $buildDoc(\mathcal{E}, \eta) =$ `<e/>`;
- if $P(e) =$ `#PCDATA` then $buildDoc(\mathcal{E}, \eta) =$ `<e>`$text(s)$`</e>`;
- if $P(e)$ is a regular expression then $buildDoc(\mathcal{E}, \eta) =$ `<e>`$buildDoc(\mathcal{E}, \eta_1) + \ldots + buildDoc(\mathcal{E}, \eta_k)$`</e>`, where $\mathcal{E}(\eta) = [\eta_1, \ldots, \eta_k]$.

Moreover, we simply denote with $buildDoc(\mathcal{E})$ the application of *buildDoc* to the root information fragment in $\mathcal{E}$.

Definition 3.12 (Extracted XML document). *Let $\mathcal{D} = \langle El, P, e_r \rangle$ be a DTD, $\mathcal{WR} = \langle \mathcal{D}, \mathcal{R}, w \rangle$ be a wrapper, and doc be an XHTML document. An XML document xdoc is extracted from doc by applying $\mathcal{WR}$ (hereinafter referred to as $\mathcal{WR}(doc) \rightsquigarrow xdoc$) if there exists an extraction model $\mathcal{E}$ of $\mathcal{WR}(doc)$ such that $xdoc = buildDoc(\mathcal{E})$. Moreover, we denote with $XDoc(\mathcal{WR}(doc))$ the set of all the XML documents xdoc such that $\mathcal{WR}(doc) \rightsquigarrow xdoc$.*

Observe that, given a wrapper $\mathcal{WR}$ and a document *doc*, it could be possible that an extraction model $\mathcal{E}$ of $\mathcal{WR}(doc)$ contains an infinite number of information fragments. For instance, consider a wrapper $\mathcal{WR}^{\infty}$ with the extraction schema below

```
<!ELEMENT a (a | b)>
<!ELEMENT b (#PCDATA)>
```

Suppose that the rule to recursively extract `a` elements is $\langle\langle \$a : .[body] \rightarrow \$a1, \{\}\rangle, (1,1), \mathsf{all}\rangle$. In the case that $\mathcal{WR}$ is applied to the root element of an XHTML document, since it contains a `body` child, an infinite chain of information fragments relating to `a` elements is yielded. In the rest of the thesis, we shall refer to *safe* wrappers, that is wrappers that cannot yield extraction models of infinite size.

Definition 3.13. *Let $\mathcal{D} = \langle El, P, e_r\rangle$ be a DTD and $\mathcal{WR} = \langle \mathcal{D}, \mathcal{R}, w\rangle$ be a wrapper. $\mathcal{WR}$ is said* safe *if, for each pair of mutually recursive elements e, e' in $\mathcal{D}$ such that e' appears in the definition of e, the extraction rule $w(e, e')$ is a non-self extraction rule.*

It is easy to see that the above described $\mathcal{WR}^{\infty}$ wrapper is not a safe wrapper. We now prove that every extraction model yielded by a safe wrapper is of finite size. Specifically, we first introduce a preliminary result characterizing the maximum number of sequences of node sequences which can be yielded by an extraction rule, and next prove that every extraction model yielded by a safe wrapper is of finite size.

Lemma 3.14. *Let doc be an XHTML document, s a sequence of nodes in doc, and r an extraction rule. The size of sequence $r(s)$, $|r(s)|$, is bounded by $|doc|^2$, where $|doc|$ is the number of nodes in doc.*

Proof. It straightforwardly follows from the definition of extraction rule. □

Theorem 3.15. *Let $\mathcal{D} = \langle El, P, e_r\rangle$ be a DTD, $\mathcal{WR} = \langle \mathcal{D}, \mathcal{R}, w\rangle$ be a safe wrapper, and doc be an XHTML document. Each extraction model $\mathcal{E}$ of $\mathcal{WR}(doc)$ is finite.*

Proof. Reasoning by contradiction, assume there is an extraction model $\mathcal{E}$ of $\mathcal{WR}(doc)$ such that the number of information fragments in $\mathcal{E}$ is not finite. For any extraction rule $r \in R$ and for each sequence of nodes s, Lemma 3.14 guarantees that the number of sequences in $r(s)$ is bounded by $|doc|^2$. Thus, for each information fragment $\varepsilon \in \mathcal{E}$ the number of children of η is less then or equal to $|doc|^2$. Therefore, since $\mathcal{E}$ is not finite, the latter implies that the depth of $\mathcal{E}$ must be infinite. It is easy to see that if the depth of $\mathcal{E}$ is infinite then there is at least a pair of information fragments $\eta_1 = (e_1, s_1)$ and $\eta_2 = (e_2, s_2)$ such that the following conditions hold:

1. η_1 is an ancestor of η_2;
2. $e_1 = e_2$;
3. the sequence of nodes associated to η_1 and the sequence of nodes associated to η_2 share a common node, i.e., there is a node n such that $n \in s_1$ and $n \in s_2$.

The first condition implies that η_2 is extracted from η_1. Moreover, let $\eta_p = (e_p, s_p)$ be the parent information fragment of η_2 (possibly coinciding with η_1). Since η_2 is extracted from η_1 and η_p is the parent information fragment of η_2, it holds that e_p is mutually recursive with e_2. As $\mathcal{WR}$ is a safe wrapper, the latter implies that the extraction rule $w(e_p, e_2)$ must be a non-self extraction rule. As s_2 is in $w(e_p, e_2)$ it holds that no node appearing in s_p appears in s_2. The latter implies that every node in s_2 is a descendant of a node in s_p (it follows from the definition of extraction rule). Moreover, since every node in s_p is a descendant of a node in s_1 or a node in s_1, every node in s_2 is a descendant of a node in s_1. As s_1 does not contain two nodes that are one descendant of the other, the latter implies that there is no node n such that $n \in s_1$ and $n \in s_2$, thus contradicting that s_1 and s_2 share a common node. $\square$

We now characterize the cardinality of extraction models in the cases of non-recursive DTD and safe wrapper.

Theorem 3.16. *Let $\mathcal{D} = \langle El, P, e_r\rangle$ be a DTD, $\mathcal{WR} = \langle \mathcal{D}, \mathcal{R}, w\rangle$ be a wrapper, and doc be an XHTML document.*

1. *if $\mathcal{D}$ is not recursive then the cardinality of every extraction model $\mathcal{E}$ of $\mathcal{WR}(doc)$ is $\mathcal{O}((C(El) \times |doc|^2)^{depth(\mathcal{D})})$.*
2. *if $\mathcal{WR}$ is a safe wrapper then the cardinality of every extraction model $\mathcal{E}$ of $\mathcal{WR}(doc)$ is $\mathcal{O}((C(El) \times |doc|^2)^{depth(\mathcal{D})+H})$, where H is the depth of doc.*

Proof. Let $\mathcal{E}$ be an extraction model of $\mathcal{WR}(doc)$ and $\eta = (e, s)$ be an information fragment in $\mathcal{E}$. The number of information fragments in $\mathcal{E}$ that have η as parent fragment is $|\mathcal{E}(\eta)| \leq \sum_{e' \in N(e)} |w(e, e')(s)|$.
According to Lemma 3.14, we have $|\mathcal{E}(\eta)| \leq |N(e)| \times |doc|^2$. Thus, each node (information fragment) $\eta = (e, s)$ in $\mathcal{E}$ has at most $|N(e)| \times |doc|^2$ nodes as children. Therefore, the number of information fragments in $\mathcal{E}$ is less than or equal to $\frac{(C(El)\times|doc|^2)^{L+1}-1}{(C(El)\times|doc|^2)-1}$, where L is the depth of $\mathcal{E}$. We now consider distinctly the cases of non-recursive DTD and safe wrappers, providing a bound for the depth of $\mathcal{E}$.

1. In this case, it is trivial to see that the depth of $\mathcal{E}$ is less than or equal to $depth(\mathcal{D})$. Therefore, the cardinality of $\mathcal{E}$ is $\mathcal{O}((C(El) \times |doc|^2)^{depth(\mathcal{D})})$.
2. We now prove that the maximum depth of an extraction model $\mathcal{E}$ of $\mathcal{WR}(doc)$ is $depth(\mathcal{D}) + H$, reasoning by contradiction. Assume that there exists an extraction model $\mathcal{E}$ of $\mathcal{WR}(doc)$ such that the depth of $\mathcal{E}$ is greater than $depth(\mathcal{D}) + H$. Let $p = \eta_0, \ldots, \eta_n$ be the longest root-to-leaf path in $\mathcal{E}$. Obviously, n is greater that $2 \times depth(\mathcal{D}) + H$. Let $p_1, \ldots, p_k$ and $q_1, \ldots, q_{k-1}$ be sub-paths of p such that:
   a) p is of form $p_1, q_1, \ldots, p_{k-1}, q_{k-1}, p_k$;

b) for each $i \in [1..k\text{-}1]$, the first and the last information fragments in q_i are associated to the same element;
c) the element chain $el(p_1), el^*(q_1), el(p_2), \ldots, el(p_{k-1}), el^*(q_{k-1}), el(p_k)$ is not recursive, where $el(p_i)$ is the element chain obtained by replacing every information fragment in p_i with its associated element and $el^*(q_i)$ is the element associated to the first information fragment in q_i.

From the last condition, it follows that the size of the element chain $el(p_1), el^*(q_1), el(p_2), \ldots, el(p_{k-1}), el^*(q_{k-1}), el(p_k)$ is less than or equal to $depth(\mathcal{D})$. Therefore, the length of $p_1, \eta_{q_1}, p_2, \ldots, p_{k-1}, \eta_{q_{k-1}}, p_k$, where η_{q_1} is the first information fragment in q_i, is less than or equal to $depth(\mathcal{D})$. Therefore, contradiction hypothesis implies that the length of $q_1, q_2, \ldots, q_{k-2}, q_{k-1}$ is greater than $H + (k-1)$.
Let q_i be of the form $(e^i_0, s^i_0), (e^i_1, s^i_1) \ldots, (e^i_{h^i}, s^i_{h^i})$, where $e^i_0 = e^i_{h^i}$, for each $i \in [1..k\text{-}1]$. Let l_h be the level of the shallowest node belonging to $s^i_{h^i}$ and l_0 be the level of the shallowest node belonging to s^i_1. From Def. 3.13, it follows that $l_h \geq l_0 + h^i$, since for each pair of mutually recursive elements (e_j, e_{j+1}), nodes belonging to s_j are at least children of the nodes appearing in s_{j+1}.
Considering that the extraction context for the first information fragment of q_1 could be the root of the document, each q_i moves down the context for the following extractions of a number of levels of the document that is greater than or equal to $h^i - 1$. Therefore, the sum of the size of all q_i cannot be greater than H since the context reaches the end of the document, i.e. $\sum_{i=1}^{k-1} h^i \leq H$, that yields a contradiction.
Thus, the number of information fragments in $\mathcal{E}$ is bounded by $\frac{(C(El) \times |doc|^2)^{depth(\mathcal{D})+H+1} - 1}{(C(El) \times |doc|^2) - 1}$, which completes the proof.

□

Corollary 3.17. *Let $\mathcal{WR} = \langle \mathcal{D}, \mathcal{R}, w \rangle$ be a safe wrapper and doc be an XHTML document. The set $XDoc(\mathcal{WR}(doc))$ is finite.*

Preferred extraction models

Extraction models provide a characterization of the set of XML documents that encode information extracted by a wrapper $\mathcal{WR}$ from a source XHTML document *doc*, that is the set $XDoc(\mathcal{WR}(doc))$. Each document in this set represents a candidate output of extraction from *doc* by means of $\mathcal{WR}$. However, providing multiple extraction results is not a desirable property for a wrapping framework. Therefore, we have to investigate the requirements for identifying an extraction model that is *preferred* to all possible extraction models of $\mathcal{WR}(doc)$. The preferred extraction model is the objective of the wrapper evaluation task, as we shall discuss in Sect. 3.3.3, and consequently allows for computing a unique document that is preferred to all the candidate XML extracted documents.

We assume that the notion of preferred extraction model is based on a precedence relation between extraction models, and this, in turn, relies on an order relation between lists of extraction events having the same parent information fragment.

Definition 3.18 (Precedence between extraction models). *Let $\mathcal{WR} = \langle \mathcal{D}, \mathcal{R}, w\rangle$ be a safe wrapper, doc be an XHTML document, $\mathcal{E}_1$ and $\mathcal{E}_2$ be two extraction models of $\mathcal{WR}(doc)$, and $\eta = (e, s)$ be an information fragment such that $\eta \in \mathcal{E}_1$ and $\eta \in \mathcal{E}_2$. We say that $\mathcal{E}_1(\eta)$ precedes $\mathcal{E}_2(\eta)$ ($\mathcal{E}_1(\eta) \prec \mathcal{E}_2(\eta)$) if and only if one of the following conditions holds:*

1. $\mathrm{elnames}(\mathcal{E}_1(\eta))$ *precedes* $\mathrm{elnames}(\mathcal{E}_2(\eta))$ *in language $L(P(e))$.*
2. $\mathrm{elnames}(\mathcal{E}_1(\eta))$ *is equal to* $\mathrm{elnames}(\mathcal{E}_2(\eta))$*, and there exists an integer $i \geq 0$ such that:*
   - $\mathcal{E}_1(\eta) = [(e_0, s_0), \ldots, (e_{i-1}, s_{i-1}),\ (e_i, s_i^{(1)}), \ldots,\ (e_k, s_k^{(1)})]$, *and*
   - $\mathcal{E}_2(\eta) = [(e_0, s_0), \ldots, (e_{i-1}, s_{i-1}),\ (e_i, s_i^{(2)}), \ldots,\ (e_k, s_k^{(2)})]$, *and*
   - $s_i^{(1)}$ *precedes* $s_i^{(2)}$.
3. $\mathcal{E}_1(\eta) = \mathcal{E}_2(\eta) = [\eta_0, \ldots, \eta_k]$*, and there exists an integer $i \geq 0$ such that, for each $j < i$, $\mathcal{E}_1^*(\eta_j) = \mathcal{E}_2^*(\eta_j)$ and $\mathcal{E}_1(\eta_i) \prec \mathcal{E}_2(\eta_i)$.*

Moreover, let $\mathcal{E}_1 = \langle E_1, <_1, \vdash_1, \eta_r\rangle$ and $\mathcal{E}_2 = \langle E_2, <_2, \vdash_2, \eta_r\rangle$ be two extraction models of $\mathcal{WR}(doc)$ having the same root information fragment. $\mathcal{E}_1$ precedes $\mathcal{E}_2$ ($\mathcal{E}_1 \prec \mathcal{E}_2$) if and only if $\mathcal{E}_1(\eta_r) \prec \mathcal{E}_2(\eta_r)$.

Definition 3.19 (Preferred extraction model). *Let $\mathcal{WR} = \langle \mathcal{D}, \mathcal{R}, w\rangle$ be a safe wrapper, doc be an XHTML document, and $\mathcal{E}$ be an extraction model. $\mathcal{E}$ is* preferred *with respect to $\mathcal{WR}(doc)$ if and only if there not exists an extraction model $\mathcal{E}'$ of $\mathcal{WR}(doc)$ such that $\mathcal{E}' \prec \mathcal{E}$.*

Precedence relation between extraction models enables us to devise a similar relation between extracted documents. However, while an extraction model is implicitly associated with one only XML document, an extracted document may be in principle generated from multiple extraction models.

Definition 3.20 (Precedence between extracted XML documents). *Let $\mathcal{WR} = \langle \mathcal{D}, \mathcal{R}, w\rangle$ be a safe wrapper, doc be an XHTML document, and let $xdoc_1$ and $xdoc_2$ be two extracted XML documents in $XDoc(\mathcal{WR}(doc))$. $xdoc_1$ precedes $xdoc_2$ ($xdoc_1 \prec xdoc_2$) if and only if, for each model $\mathcal{E}_2$ of $xdoc_2$, there exists a model $\mathcal{E}_1$ of $xdoc_1$ such that $\mathcal{E}_1 \prec \mathcal{E}_2$.*

Intuitively, the *preferred extracted document* is the favorite among all the XML documents that can be generated from extraction models.

Definition 3.21 (Preferred extracted XML document). *Let $\mathcal{WR} = \langle \mathcal{D}, \mathcal{R}, w\rangle$ be a safe wrapper, doc be an XHTML document, and xdoc be an*

XML document in $XDoc(\mathcal{WR}(doc))$. $xdoc$ is preferred *in $XDoc(\mathcal{WR}(doc))$ if and only if $xdoc \prec xdoc'$, for each document $xdoc' \in XDoc(\mathcal{WR}(doc))$, with $xdoc' \neq xdoc$.*

Theorem 3.22. *Let $\mathcal{D} = \langle El, P, e_r \rangle$ be a DTD, $\mathcal{WR} = \langle \mathcal{D}, \mathcal{R}, w \rangle$ be a safe wrapper, and doc be an XHTML document. The preferred extracted XML document in $XDoc(\mathcal{WR}(doc))$ is unique.*

Proof. Let $\langle \mathbf{E}, \prec \rangle$ be the (ordered) set of all the extraction models of $\mathcal{WR}(doc)$. Since $\mathcal{WR}$ is a safe wrapper then it is easy to see that Theorem 3.15 implies that $\langle \mathbf{E}, \prec \rangle$ is finite. Therefore, the uniqueness of the preferred extracted document can be shown by proving that $\prec$ is a total order on $\mathbf{E}$. We will have to prove that antisymmetry, transitivity, and comparability hold for $\prec$.[3] Let us denote the depth of an extraction model $\mathcal{E}$ as $depth(\mathcal{E})$, and $\eta_r = (e_r, s)$. Let $\mathcal{E}_1 = \langle E_1, <_1, \vdash_1, \eta_r \rangle$, $\mathcal{E}_2 = \langle E_2, <_2, \vdash_2, \eta_r \rangle$, and $\mathcal{E}_3 = \langle E_3, <_3, \vdash_3, \eta_r \rangle$ be extraction models of $\mathcal{WR}(doc)$. Let us show first the comparability axiom for $\prec$.

Comparability. Reasoning by induction on the depth of $\mathcal{E}_1$ and $\mathcal{E}_2$, we show that if $\mathcal{E}_1 \neq \mathcal{E}_2$ then either $\mathcal{E}_1 \prec \mathcal{E}_2$ or $\mathcal{E}_2 \prec \mathcal{E}_1$.

1. *Induction base.* Assume that $depth(\mathcal{E}_1) = depth(\mathcal{E}_2) = 1$. Since $\mathcal{E}_1(\eta_r) \neq \mathcal{E}_2(\eta_r)$ then:
   - if $elnames(\mathcal{E}_1(\eta_r)) \neq elnames(\mathcal{E}_2(\eta_r))$, since $L(P(e_r))$ is a linearly ordered language, then either $elnames(\mathcal{E}_1(\eta_r)) >_{L(P(e_r))} elnames(\mathcal{E}_2(\eta_r))$ or $elnames(\mathcal{E}_2(\eta_r)) >_{L(P(e_r))} elnames(\mathcal{E}_1(\eta_r))$, otherwise
   - if $elnames(\mathcal{E}_1(\eta_r)) = elnames(\mathcal{E}_2(\eta_r))$ then there exists an integer i such that $\mathcal{E}_1(\eta_r) = [\eta_0, \ldots, \eta_{i-1}, \eta_i^{(1)}, \ldots, \eta_k^{(1)}]$ and $\mathcal{E}_2(\eta_r) = [\eta_0, \ldots, \eta_{i-1}, \eta_i^{(2)}, \ldots, \eta_k^{(2)}]$, where $\eta_i^{(1)} \neq \eta_i^{(2)}$. Let $\eta_i^{(1)} = (e, s_1)$ and $\eta_i^{(2)} = (e, s_2)$. Assume without loss of generality that s_1 precedes s_2 in the sequence $w(e_r, e)(s)$; then $\mathcal{E}_1(\eta_r) \prec \mathcal{E}_2(\eta_r)$ holds due to the second point of Def. 3.18.
2. *Induction.* Let $depth(\mathcal{E}_1) = depth(\mathcal{E}_2) \leq l$. Because of the inductive hypothesis, for each pair of extraction models $\mathcal{E}'$ and $\mathcal{E}''$, with $depth(\mathcal{E}') < l$ and $depth(\mathcal{E}'') < l$, if $\mathcal{E}' \neq \mathcal{E}''$ then either $\mathcal{E}' \prec \mathcal{E}''$ or $\mathcal{E}'' \prec \mathcal{E}'$.
   If $\mathcal{E}_1(\eta_r) \neq \mathcal{E}_2(\eta_r)$ then, by reasoning as in the induction base, we have that either $\mathcal{E}_1 \prec \mathcal{E}_2$ or $\mathcal{E}_2 \prec \mathcal{E}_1$. Assume that $\mathcal{E}_1(\eta_r) = \mathcal{E}_2(\eta_r) = [\eta_0, \ldots, \eta_k]$. Let $i \in [0..k]$ be an integer such that, for each $j < i$, $\mathcal{E}_1^*(\eta_j) = \mathcal{E}_2^*(\eta_j)$ and $\mathcal{E}_1^*(\eta_i) \neq \mathcal{E}_2^*(\eta_i)$. Since $depth(\mathcal{E}_1^*(\eta_i)) < l$, $depth(\mathcal{E}_2^*(\eta_i)) < l$, and $\mathcal{E}_1^*(\eta_i) \neq \mathcal{E}_2^*(\eta_i)$ then, according to the inductive hypothesis, either $\mathcal{E}_1^*(\eta_i) \prec \mathcal{E}_2^*(\eta_i)$ or $\mathcal{E}_2^*(\eta_i) \prec \mathcal{E}_1^*(\eta_i)$. Thus, we must have either $\mathcal{E}_1 \prec \mathcal{E}_2$ or $\mathcal{E}_2 \prec \mathcal{E}_1$, which concludes the proof that comparability holds for $\prec$ in $\mathcal{WR}(doc)$.

[3] Actually, a total order should also satisfy the reflexivity axiom; however, since $\prec$ is defined as a strict inequality, the reflexivity property has no sense for $\prec$.

Antisymmetry. The proof for antisymmetry can be straightforwardly derived from the proof for comparability: indeed, the comparability proof states it is not possible that, given two extraction models $\mathcal{E}_1$ and $\mathcal{E}_2$, $\mathcal{E}_1 \prec \mathcal{E}_2$ and $\mathcal{E}_2 \prec \mathcal{E}_1$ hold at the same time, unless $\mathcal{E}_1$ is equal to $\mathcal{E}_2$.

Transitivity. Reasoning by induction on the depth of the extraction models $\mathcal{E}_1, \mathcal{E}_2$ and $\mathcal{E}_3$, we show that if $\mathcal{E}_1 \prec \mathcal{E}_2$ and $\mathcal{E}_2 \prec \mathcal{E}_3$ then $\mathcal{E}_1 \prec \mathcal{E}_3$. Let $\mathcal{E}_1(\eta_r) = [\eta_0^{(1)}, \ldots, \eta_{k_1}^{(1)}]$, $\mathcal{E}_2(\eta_r) = [\eta_0^{(2)}, \ldots, \eta_{k_2}^{(2)}]$, and $\mathcal{E}_3(\eta_r) = [\eta_0^{(3)}, \ldots, \eta_{k_3}^{(3)}]$.

Induction base ($\mathit{depth}(\mathcal{E}_1) = 1$, $\mathit{depth}(\mathcal{E}_2) = 1$, and $\mathit{depth}(\mathcal{E}_3) = 1$). In order to prove that if $\mathcal{E}_1 \prec \mathcal{E}_2$ and $\mathcal{E}_2 \prec \mathcal{E}_3$ then $\mathcal{E}_1 \prec \mathcal{E}_3$, consider the following cases:

1. $\mathit{elnames}(\mathcal{E}_3(\eta_r)) >_{L(P(e_r))} \mathit{elnames}(\mathcal{E}_2(\eta_r))$ and $\mathit{elnames}(\mathcal{E}_2(\eta_r)) >_{L(P(e_r))} \mathit{elnames}(\mathcal{E}_1(\eta_r))$. Then, $\mathit{elnames}(\mathcal{E}_3(\eta_r)) >_{L(P(e_r))} \mathit{elnames}(\mathcal{E}_1(\eta_r))$ since $L(P\ (e_r))$ is a linearly ordered language. Thus, $\mathcal{E}_1(\eta_r) \prec \mathcal{E}_3(\eta_r)$.
2. $\mathit{elnames}(\mathcal{E}_3(\eta_r)) >_{L(P(e_r))} \mathit{elnames}(\mathcal{E}_2(\eta_r))$ and $\mathit{elnames}(\mathcal{E}_2(\eta_r)) = \mathit{elnames}\ (\mathcal{E}_1(\eta_r))$. Thus, since $\mathit{elnames}(\mathcal{E}_3(\eta_r)) >_{L(P(e_r))} \mathit{elnames}(\mathcal{E}_1(\eta_r))$, $\mathcal{E}_1(\eta_r) \prec \mathcal{E}_3(\eta_r)$ holds due to the first point of Def. 3.18.
3. $\mathit{elnames}(\mathcal{E}_2(\eta_r)) >_{L(P(e_r))} \mathit{elnames}(\mathcal{E}_1(\eta_r))$ and $\mathit{elnames}(\mathcal{E}_3(\eta_r)) = \mathit{elnames}\ (\mathcal{E}_2(\eta_r))$. The proof is analogous to that of the previous case.
4. $\mathit{elnames}(\mathcal{E}_3(\eta_r)) = \mathit{elnames}(\mathcal{E}_2(\eta_r)) = \mathit{elnames}(\mathcal{E}_1(\eta_r))$ and $\mathcal{E}_1(\eta_r) \neq \mathcal{E}_2(\eta_r) \neq \mathcal{E}_3(\eta_r)$. Then, there exist $i_1, i_2 \geq 0$ such that:
   a) for each $j < i_1$, $\eta_j^{(2)} = \eta_j^{(1)}$ and $\eta_{i_1}^{(2)} \neq \eta_{i_1}^{(1)}$; let $\eta_{i_1}^{(1)} = (e, s_{i_1}^{(1)})$ and $\eta_{i_1}^{(2)} = (e, s_{i_1}^{(2)})$, then $s_{i_1}^{(1)}$ precedes $s_{i_1}^{(2)}$ in the sequence $w(e_r, e)(s)$ since $\mathcal{E}_1 \prec \mathcal{E}_2$;
   b) for each $h < i_2$, $\eta_h^{(3)} = \eta_h^{(2)}$ and $\eta_{i_2}^{(3)} \neq \eta_{i_2}^{(2)}$; let $\eta_{i_2}^{(3)} = (e, s_{i_2}^{(3)})$ and $\eta_{i_2}^{(2)} = (e, s_{i_2}^{(2)})$, then $s_{i_2}^{(2)}$ precedes $s_{i_2}^{(3)}$ in the sequence $w(e_r, e)(s)$ since $\mathcal{E}_2 \prec \mathcal{E}_3$.

   Let $i = \min\{i_1, i_2\}$, $\eta_i^{(1)} = (e, s_i^{(1)})$, $\eta_i^{(2)} = (e, s_i^{(2)})$, and $\eta_i^{(3)} = (e, s_i^{(3)})$. Then, for each $j < i$, $\eta_j^{(3)} = \eta_j^{(2)} = \eta_j^{(1)}$.

   Assume that $i = i_1$ (resp. $i = i_2$). Since $s_i^{(1)}$ precedes $s_i^{(2)}$ in the sequence $w(e_r, e)(s)$, and $s_i^{(2)}$ precedes $s_i^{(3)}$ in the sequence $w(e_r, e)(s)$ or $s_i^{(2)} = s_i^{(3)}$ (resp. $s_i^{(2)}$ precedes $s_i^{(3)}$ in the sequence $w(e_r, e)(s)$, and $s_i^{(1)}$ precedes $s_i^{(2)}$ in the sequence $w(e_r, e)(s)$ or $s_i^{(1)} = s_i^{(2)}$), then $s_i^{(1)}$ precedes $s_i^{(3)}$. Thus $\mathcal{E}_1(\eta_r) \prec \mathcal{E}_3(\eta_r)$.
5. $\mathit{elnames}(\mathcal{E}_3(\eta_r)) = \mathit{elnames}(\mathcal{E}_2(\eta_r)) = \mathit{elnames}(\mathcal{E}_1(\eta_r))$, $\mathcal{E}_1(\eta_r) = \mathcal{E}_2(\eta_r) = [\eta_0^{(1,2)}, \ldots, \eta_{k_{1,2}}^{(1,2)}]$ and $\mathcal{E}_2(\eta_r) \neq \mathcal{E}_3(\eta_r)$. Then, there exists an integer $i \geq 0$ such that, for each $j < i$, $\eta_j^{(1,2)} = \eta_j^{(3)}$ and $\eta_i^{(1,2)} \neq \eta_i^{(3)}$. Let $\eta_i^{(1,2)} = (e, s_{1,2})$ and $\eta_i^{(3)} = (e, s_3)$. Since $s_{1,2}$ precedes s_3 in the sequence $w(e_r, e)(s)$, $\mathcal{E}_1(\eta_r) \prec \mathcal{E}_3(\eta_r)$ holds due to the second point of Def. 3.18.

6. $elnames(\mathcal{E}_3(\eta_r)) = elnames(\mathcal{E}_2(\eta_r)) = elnames(\mathcal{E}_1(\eta_r))$, $\mathcal{E}_1(\eta_r) \neq \mathcal{E}_2(\eta_r)$ and $\mathcal{E}_2(\eta_r) = \mathcal{E}_3(\eta_r) = [\eta_0^{(2,3)}, \ldots, \eta_{k_{2,3}}^{(2,3)}]$. The proof is analogous to that of the previous case.

Induction ($depth(\mathcal{E}_1) \leq l \wedge depth(\mathcal{E}_2) \leq l \wedge depth(\mathcal{E}_3) \leq l$). The inductive hypothesis states that, for each triple of extraction models $\mathcal{E}', \mathcal{E}'', \mathcal{E}'''$, with $depth(\mathcal{E}') < l$, $depth(\mathcal{E}'') < l$, and $depth(\mathcal{E}''') < l$, if $\mathcal{E}' \prec \mathcal{E}''$ and $\mathcal{E}'' \prec \mathcal{E}'''$ then $\mathcal{E}' \prec \mathcal{E}'''$.

In order to prove that if $\mathcal{E}_1 \prec \mathcal{E}_2$ and $\mathcal{E}_2 \prec \mathcal{E}_3$ then $\mathcal{E}_1 \prec \mathcal{E}_3$, the above cases (1)-(6) must be considered; however, the proof of such cases has been already reported in the induction base, thus here is omitted. We have only to consider the case when $\mathcal{E}_1(\eta_r) = \mathcal{E}_2(\eta_r) = \mathcal{E}_3(\eta_r) = [\eta_0, \ldots, \eta_k]$. Then, there exist $i_1, i_2 \geq 0$ such that:

1. for each $j < i_1$, $\mathcal{E}_2^*(\eta_j^{(2)}) = \mathcal{E}_1^*(\eta_j^{(1)})$ and, since $\mathcal{E}_1 \prec \mathcal{E}_2$, $\mathcal{E}_1(\eta_{i_1}^{(1)}) \prec \mathcal{E}_2(\eta_{i_1}^{(2)})$;
2. for each $h < i_2$, $\mathcal{E}_3^*(\eta_h^{(3)}) = \mathcal{E}_2^*(\eta_h^{(2)})$ and, since $\mathcal{E}_2 \prec \mathcal{E}_3$, $\mathcal{E}_2(\eta_{i_2}^{(2)}) \prec \mathcal{E}_3(\eta_{i_2}^{(3)})$.

Let $i = \min\{i_1, i_2\}$. Then, $\mathcal{E}_3^*(\eta_j^{(3)}) = \mathcal{E}_2^*(\eta_j^{(2)}) = \mathcal{E}_1^*(\eta_j^{(1)})$ holds for each $j < i$. Consider the following cases:

- $\mathcal{E}_1(\eta_i^{(1)}) \prec \mathcal{E}_2(\eta_i^{(2)})$ and $\mathcal{E}_2^*(\eta_i^{(2)}) = \mathcal{E}_3^*(\eta_i^{(3)})$. Then, $\mathcal{E}_1(\eta_i^{(1)}) \prec \mathcal{E}_3(\eta_i^{(3)})$ holds.
- $\mathcal{E}_2(\eta_i^{(2)}) \prec \mathcal{E}_3(\eta_i^{(3)})$ and $\mathcal{E}_1^*(\eta_i^{(1)}) = \mathcal{E}_2^*(\eta_i^{(2)})$. Then, $\mathcal{E}_1(\eta_i^{(1)}) \prec \mathcal{E}_3(\eta_i^{(3)})$ holds.
- $\mathcal{E}_1(\eta_i^{(1)}) \prec \mathcal{E}_2(\eta_i^{(2)})$ and $\mathcal{E}_2(\eta_i^{(2)}) \prec \mathcal{E}_3(\eta_i^{(3)})$. Since $depth(\mathcal{E}_1(\eta_i^{(1)})) < l$, $depth\ (\mathcal{E}_2(\eta_i^{(2)})) < l$, and $depth(\mathcal{E}_3(\eta_i^{(3)})) < l$, by applying the inductive hypothesis we derive that $\mathcal{E}_1(\eta_i^{(1)}) \prec \mathcal{E}_3(\eta_i^{(3)})$.

Thus, we must have that if $\mathcal{E}_1 \prec \mathcal{E}_2$ and $\mathcal{E}_2 \prec \mathcal{E}_3$ then $\mathcal{E}_1 \prec \mathcal{E}_3$, which concludes the transitivity proof, and finally proves that $\prec$ is a total order for $\mathcal{WR}(doc)$. □

Given a wrapper $\mathcal{WR}$ and an HTML document *doc*, the result of applying $\mathcal{WR}$ to *doc* is the unique preferred extracted XML document in $XDoc(\mathcal{WR}(doc))$. Hereinafter we denote $xdoc_{\mathcal{WR}}(doc)$ as the preferred extracted XML document in $XDoc(\mathcal{WR}(doc))$.

3.3.3 Wrapper evaluation

In our setting, wrapper evaluation can be stated as follows: Given a wrapper $\mathcal{WR}$ for an input XHTML document *doc*, compute the preferred extraction model $\mathcal{E}$ of $\mathcal{WR}(doc)$. The output XML document containing the extracted data can be directly obtained by applying the *buildDoc* function to the computed extraction model $\mathcal{E}$.

Figure 3.4 shows the *PreferredExtractionModel* algorithm for wrapper evaluation, which is devised into two main stages. In the first stage (function buildTree) all the information fragments are computed in a top-down fashion: starting from the root of the input XHTML document, an information fragment is added to the extraction tree being constructed until there is a sequence of nodes that can be constructed using the extraction rules specified by the wrapper. This task is performed by the computeFragments function, which is recursively invoked on an information fragment $\eta = (e, s)$ adding to the input extraction tree the information fragment (e_1, s_1) such that e_1 and s_1 appear in $P(e)$ and $w(e, e_1)(s)$, respectively. Finally, all the leaf information fragments in $\mathcal{E}$ are marked as ready.

For the sake of presentation of the algorithm, we introduce the following attributes for representing the order between information fragments during the construction of the preferred extraction model. Given an extraction tree $\mathcal{E}$, any information fragment is associated with a position (pos), which follows the partial order on $\mathcal{E}$. All fragments are initially associated with an invalid position ($pos = -1$). Information fragments in $\mathcal{E}$ are marked as either "ready" (R) or "waiting" (W). An information fragment η is marked as "ready" when all its children (i.e all fragments in $\mathcal{E}(\eta)$) have a fully defined content in $\mathcal{E}$. We say that an information fragment has a fully defined content if all its children have valid positions (i.e. $pos \neq -1$). Obviously, leaf fragments are "ready" since they have no child.

The second stage is covered by the buildElements function. This exploits the extraction schema and the fragments previously extracted to generate the preferred extraction model, in a bottom-up fashion. The function iteratively builds the content model of the element of an information fragment selected from the current set of "waiting" fragments whose children are "ready". This set is computed by function *childrenReady*, which is used to control the main loop in buildElements. Finally, an extraction model is returned only if all the assigned information fragments are "ready".

The buildElementContent function is devoted to the construction of the content of each element e such that $\eta = (e, s) \in \mathit{childrenReady}(\mathcal{E})$. For this purpose, the function exploits two structures: CM, initially empty, which represents all current information fragments already processed (i.e. information fragments with a fully defined content model in $\mathcal{E}$), and RCM, initialized to $\mathcal{E}(\eta)$, which contains the remaining information fragments to be processed and included in the resulting extraction model. Initially, the type $\alpha = P(e)$ of e is checked. If $\alpha = e_t$ then an information fragment η for e_t is selected according to the predefined partial order, associated with the next available position in CM, and then inserted into CM (function *assignNextPosition*). Otherwise, if α is a complex expression then it is divided in subexpressions: according to the order of these subexpressions, buildElementContent checks whether they are satisfiable and recursively calls itself on the first subexpression that is recognized to be satisfiable.

```
Input:
  A wrapper $\mathcal{WR} = \langle \mathcal{D}, \mathcal{R}, w \rangle$, where $\mathcal{D} = \langle El, P, e_r \rangle$;
  An XHTML document $doc$.
Output:
  An extraction model $\mathcal{E}$ of $\mathcal{WR}(doc)$.
Method:
  $\mathcal{E} :=$ buildTree($\mathcal{WR}, doc$);
  $\mathcal{E} :=$ buildElements($\mathcal{WR}, \mathcal{E}$);
  return $unmark(\mathcal{E})$;          /* returns $\mathcal{E}$ discarding marks */
--------------------------------------------------------------------
Function buildTree($\mathcal{WR}, doc$) : $\mathcal{E}$;
  Let $\mathcal{E}$ be an extraction tree containing only the root information
    fragment $\eta_r$, where $\eta_r = (e_r, [r_{doc}])$ and $\eta_r.state = \mathtt{W}$;
  $\mathcal{E} := computeFragments(\mathcal{E}, \mathcal{WR}, \eta_r)$;
  $\mathcal{LE} := leafFragments(\mathcal{E})$;
  for each $\eta \in \mathcal{LE}$ do
    $mark(\mathcal{E}, \eta, \mathtt{R})$;          /* identifies $\eta$ as "ready" in $\mathcal{E}$ */
  return $\mathcal{E}$;
--------------------------------------------------------------------
Function buildElements($\mathcal{WR}, \mathcal{E}$) : $\mathcal{E}$;
  while ($childrenReady(\mathcal{E}) \neq \emptyset$) do
    let $\eta = (e, s) \in childrenReady(\mathcal{E})$;
    $CM := \emptyset$;        /* information fragments already processed */
    $RCM := \mathcal{E}(\eta)$; /* information fragments not yet processed */
    buildElementContent($\mathcal{WR}, P(e)$, '', $CM, RCM$);
    if ($CM \neq \emptyset$) then
      $\mathcal{E} := \mathcal{E} - \mathcal{E}(\eta) \cup CM$;
      $mark(\mathcal{E}, \eta, \mathtt{R})$;
    else
      $\mathcal{E} := remove(\mathcal{E}, \eta)$;     /* removes $\eta$ and all its children */
  if ($\eta_r \in \mathcal{E} \wedge \eta_r.state = \mathtt{W}$) then
    return $\emptyset$;
  return $\mathcal{E}$;
```

Fig. 3.4: Wrapper evaluation: The *PreferredExtractionModel* algorithm

Theorem 3.23. *Let $\mathcal{D} = \langle El, P, e_r \rangle$ be a DTD, $\mathcal{WR} = \langle \mathcal{D}, \mathcal{R}, w \rangle$ be a safe wrapper, and doc be an XHTML document.*

(a) *The PreferredExtractionModel algorithm computes the preferred extraction model in $\mathcal{WR}(doc)$.*
(b) *if $\mathcal{D}$ is not recursive then the PreferredExtractionModel algorithm works in polynomial time with respect to the size of doc.*

Proof. (*Correctness*)
We first prove that every extraction model in $\mathcal{WR}(doc)$ can be obtained from the extraction tree $\mathcal{E}$ returned by buildTree by removing some information fragments in $\mathcal{E}$. Let $\mathcal{E}_1$ be an extraction model in $\mathcal{WR}(doc)$. We prove the

```
Function buildElementContent($\mathcal{WR}, \alpha, \alpha_r, \mathcal{E}, \mathcal{NE}$);
Input:
  A wrapper $\mathcal{WR} = \langle \mathcal{D}, \mathcal{R}, w \rangle$, where $\mathcal{D} = \langle El, P, e_r \rangle$;
  Regular expressions $\alpha, \alpha_r$;
  Extraction trees $\mathcal{E}, \mathcal{NE}$, with root $\eta = (e, s)$.
Output:
  $\mathcal{E}, \mathcal{NE}$ modified.
Method:
  if $(\alpha = e_t)$ then
    let $\eta' = (e_t, s') \in \mathcal{NE}$, with $\eta_r.state = \mathtt{R}$ and $\eta_r.pos = -1$,
    and there not exists $\eta'' = (e_t, s'') \in \mathcal{NE}$ such that $s'$ precedes
    $s''$ in $w(e, e_t)(s)$;
    $assignNextPosition(\eta', \mathcal{E})$;
    $\mathcal{NE} := \mathcal{NE} - \{\eta'\}$;
  else if $(\alpha = \alpha_1|\alpha_2)$ then
    if $satisfiable(\alpha_1\alpha_r, \mathcal{NE})$ then
      buildElementContent($\alpha_1, \alpha_r, \mathcal{E}, \mathcal{NE}$);
    else if $satisfiable(\alpha_2\alpha_r, \mathcal{NE})$ then
      buildElementContent($\alpha_2, \alpha_r, \mathcal{E}, \mathcal{NE}$);
    else
      $\mathcal{E} := \emptyset$;  $\mathcal{NE} := \emptyset$;
  else if $(\alpha = \alpha_1\alpha_2)$ then
    if not $satisfiable(\alpha_1\alpha_2\alpha_r, \mathcal{NE})$ then
      $\mathcal{E} := \emptyset$;  $\mathcal{NE} := \emptyset$;
    else
      buildElementContent($\alpha_1, \alpha_2\alpha_r, \mathcal{E}, \mathcal{NE}$);
      buildElementContent($\alpha_2, \alpha_r, \mathcal{E}, \mathcal{NE}$);
  else if $(\alpha = \alpha_1^*)$ then
    while $satisfiable(\alpha_1\alpha_r, \mathcal{NE})$ do
      buildElementContent($\alpha_1, \alpha_r, \mathcal{E}, \mathcal{NE}$);
```

Fig. 3.5: Wrapper evaluation: The buildElementContent function

above statement showing that every information fragment η in $\mathcal{E}_1$ appears at the same level in $\mathcal{E}$ and the parent of η in $\mathcal{E}_1$ is the same parent of η in $\mathcal{E}$. We prove the last statement reasoning by induction on the level of $\eta = (s_\eta, e_\eta)$.

- *Base case*: η is an information fragment at level 0 in $\mathcal{E}_1$. In this case η is the root information fragment of $\mathcal{E}_1$, and thus has no parent. Here, the property trivially holds, since all the extraction models in $\mathcal{WR}(doc)$ have (s_r, e_r) as root information fragment, where s_r is a sequence of nodes containing the root element of *doc* and (s_r, e_r) is also the root information fragment of $\mathcal{E}$.
- *Induction*: η is an information fragment at level k in $\mathcal{E}_1$ $(k > 0)$. Let $\eta_p = (e_{\eta_p}, s_{\eta_p})$ be the parent of η in $\mathcal{E}_1$. From the induction hypothesis, since the level of η_p is $k-1$, it follows that η_p is in $\mathcal{E}$, and that the level of η_p in $\mathcal{E}$ is $k-1$. Moreover, since η_p is the parent of η in $\mathcal{E}$, it holds that

s_η is in $w(e_{\eta_p}, e_\eta)(s_{\eta_p})$. As η_p is in $\mathcal{E}$, *computeFragments* has been invoked on η_p. Since this implies, for each element e appearing in $P(e_p)$, all the information fragments (e, s) such that s is in $w(e_{\eta_p}, e)(s_{\eta_p})$ are added to $\mathcal{E}$, then η is the child of η_p in $\mathcal{E}$. This concludes the proof of the property since η appears at the same level in $\mathcal{E}_1$ and $\mathcal{E}$ and the parent of η in $\mathcal{E}_1$ is the same parent of η in $\mathcal{E}$.

We prove that the tree of information fragments $\mathcal{E}$ yielded by buildElements is the preferred extraction model. It is easy to see that it is an extraction model, since for each information fragment $\eta = (e, s)$ in $\mathcal{E}$, it holds that $elnames(\mathcal{E}(\eta))$ belongs to the language defined by $P(e)$, otherwise it will be removed by buildElements; indeed, in this case CN will be equal to $\emptyset$ after invoking buildElementContent.

Finally, we show that the extraction model $\mathcal{E}_1$ yielded by buildElements is the preferred extraction model. Reasoning by contradiction, assume that $\mathcal{E}_1$ is not the preferred extraction model. From Theorem 3.22, it follows that $\mathcal{E}_1$ is not the preferred extraction model if and only if there exists an extraction model $\mathcal{E}_2$ such that $\mathcal{E}_2 \prec \mathcal{E}_1$. Let $\eta_r = (e_r, s_r)$ be the root information fragment of $\mathcal{E}$. Since every extraction model in $\mathcal{WR}(doc)$ can be obtained from the extraction tree $\mathcal{E}$ by removing some information fragments in $\mathcal{E}$, then η_r is also the root information fragment of $\mathcal{E}_1$ and $\mathcal{E}_2$. Therefore, $\mathcal{E}_2 \prec \mathcal{E}_1$ implies that $\mathcal{E}_2(\eta_r) \prec \mathcal{E}_1(\eta_r)$, and then one of the conditions stated in Def. 3.18 is satisfied for $\mathcal{E}_1(\eta_r)$ and $\mathcal{E}_2(\eta_r)$. We consider separately the following cases:

- $\mathcal{E}_1(\eta_r) \neq \mathcal{E}_2(\eta_r)$:
  In this case, since $\mathcal{E}_2 \prec \mathcal{E}_1$ either it holds that $elnames(\mathcal{E}_2(\eta_r))$ precedes $elnames(\mathcal{E}_1(\eta_r))$ or it holds that $elnames(\mathcal{E}_2(\eta_r)) = elnames(\mathcal{E}_1(\eta_r))$.
  - $elnames(\mathcal{E}_2(\eta_r))$ precedes $elnames(\mathcal{E}_1(\eta_r))$: Since every information fragments belonging to $\mathcal{E}_2(\eta_r)$ or $\mathcal{E}_1(\eta_r)$ also belongs to $\mathcal{E}(\eta_r)$, $elnames(\mathcal{E}_2(\eta_r))$ precedes $elnames(\mathcal{E}_1(\eta_r))$ implies that, at one step of the recursive invocation of buildElementContent, it is possible to create a sequence of information fragments whose element names compose a string which is preferred to $elnames(\mathcal{E}_1(\eta_r))$ and still $\mathcal{E}_1(\eta_r)$ is selected. It is easy to see that buildElementContent inserts an information fragment η in the output according to the preference specified by $P(e_r)$. This contradicts the initial hypothesis that $elnames(\mathcal{E}_2(\eta_r))$ precedes $elnames(\mathcal{E}_1(\eta_r))$.
  - $elnames(\mathcal{E}_2(\eta_r)) = elnames(\mathcal{E}_1(\eta_r))$: In this case, $\mathcal{E}_2(\eta_r) \prec \mathcal{E}_1(\eta_r)$ implies that there exists an integer $i \geq 0$ such that:
    - $\mathcal{E}_1(\eta_r) = [(e_0, s_0), \ldots, (e_{i-1}, s_{i-1}),\ (e_i, s_i^{(1)}), \ldots,\ (e_k, s_k^{(1)})]$, and
    - $\mathcal{E}_2(\eta_r) = [(e_0, s_0), \ldots, (e_{i-1}, s_{i-1}),\ (e_i, s_i^{(2)}), \ldots,\ (e_k, s_k^{(2)})]$, and
    - $s_i^{(2)}$ precedes $s_i^{(1)}$.

    Since function buildElementContent selects the information fragment $(e_i, s_i^{(1)})$ if and only if there not exists another information fragment

$\eta' = (e_i, s')$ such that s' precedes $s_i^{(1)}$, then it cannot hold that $s_i^{(2)}$ precedes $s_i^{(1)}$, thus contradicting that $\mathcal{E}_2(\eta_r) \prec \mathcal{E}_1(\eta_r)$.

- $\mathcal{E}_1(\eta_r) = \mathcal{E}_2(\eta_r)$:
In this case $\mathcal{E}_1(\eta_r) = \mathcal{E}_2(\eta_r) = [\eta_0, \ldots, \eta_k]$, and there exists an integer $i \geq 0$ such that, for each $j < i$, $\mathcal{E}_1^*(\eta_j) = \mathcal{E}_2^*(\eta_j)$ and $\mathcal{E}_2(\eta_i) \prec \mathcal{E}_1(\eta_i)$. Applying inductively the same reasoning we applied to $\mathcal{E}_1(\eta_r)$ and $\mathcal{E}_2(\eta_r)$ to $\mathcal{E}_1(\eta_i)$ and $\mathcal{E}_1(\eta_i)$ it is easy to show that it is not possible that $\mathcal{E}_2(\eta_i) \prec \mathcal{E}_1(\eta_i)$, thus contradicting the initial hypothesis.

(*Complexity*)
The cost of the *PreferredExtractionModel* algorithm is determined by the cost of two functions, namely buildTree and buildElements.

- Function buildTree. Initially, the extraction tree $\mathcal{E}$ contains only the root information fragment $\eta = (e_r, s_r)$, where s_r is the context sequence that equals the root of *doc*. By Theorem 3.15, we know that each information fragment $\eta = (e, s)$ in $\mathcal{E}$ has at most $|El_e| \times |doc|^2$ nodes as children, where El_e is the set of element names in $P(e)$, and the maximum number of information fragments in $\mathcal{E}$ is $\frac{(E \times |doc|^2)^{L+1} - 1}{(E \times |doc|^2) - 1}$, where $E = \max_{(e,s) \in \mathcal{E}} \{|El_e|\}$ and L is the depth of $\mathcal{E}$ (i.e. the maximum nesting level of elements in $\mathcal{D}$). For each pair of information fragments, the parent $\eta_p = (e_p, s_p)$ and the target $\eta_t = (e_t, s_t)$, a valid extraction rule $r = w(e_p, e_t)$ is applied to the context s_p. Let $r = \langle EF, as, rs \rangle$. The cost of evaluating r is bounded by $C_r = |EF| \times \max_{\mathcal{P} \in f \in EF}\{|\mathcal{P}|\} \times |doc|$. Let R denote the maximum cost of rule evaluation, that is $R = \max_{r \in \mathcal{WR}}\{C_r\}$. Thus, the cost of buildTree is $\mathcal{O}\big((E \times |doc|^2 \times R)^L\big)$.
- Function buildElements. The main loop is repeated while the condition evaluated by function *childrenReady* holds, that is at most L times, where L is the depth of $\mathcal{D}$. At each iteration (i.e. at each level $l \in [0..L]$), in order to build the content of an element e, function buildElementContent is (recursively) called at most $|El_e| \times |doc|^2$ times, since there may exist elements in El_e having multiple occurrences in e.
Each recursive call for buildElementContent is preceded by a call for function *satisfiable*, which checks the satisfiability of a subexpression of $P(e)$ with respect to a set of information fragments not yet processed. Function *satisfiable* has a maximum cost of $|El_e| \times |doc|^2$. Thus, the cost of buildElements is $\mathcal{O}\big((E \times |doc|^2)^{2L}\big)$, where $E = \max_{(e,s) \in \mathcal{E}}\{|El_e|\}$.

This proves that the computation of the preferred extraction model in $\mathcal{WR}(doc)$ runs in polynomial time with respect to the size of *doc*. □

Example 3.24. Consider again the HTML source, with its corresponding parse tree, and the extraction rules from Example 3.11. The PreferredExtractionModel algorithm starts from the root information fragment labeled as "waiting", then generates all the remaining fragments with functions computeFragments and leafFragments: $\mathcal{E} = \{\eta_r, \eta_1, \eta_2, \eta_3, \eta_4, \eta_5, \eta_6,$

$\eta_7, \eta_8, \eta_9\}$, $\mathcal{LE} = \{\eta_3, \eta_4, \eta_5, \eta_7, \eta_8, \eta_9\}$. All the leaf fragments are assigned with position -1 and marked as "ready". At this point, the algorithm calls buildElements*, which performs the following steps.*

- *Calls childrenReady($\{\eta_r, \eta_1, \eta_2, \eta_3, \eta_4, \eta_5, \eta_6, \eta_7, \eta_8, \eta_9\}$), which returns $\{\eta_6\}$, as this is the only fragment having all the children (η_7) "ready". Then,* buildElementContent($\mathcal{WR}$, `'rate'`, `' '`, $\emptyset, \{\eta_7\}$) *recognizes the $\alpha = e_t$ case, so directly assigns the right valid position to η_7 (i.e. $\eta_7 =$ (`rate`, [44])) and sets its state to "ready", while RCM becomes empty and CM is set to be $\{\eta_7\}$. Moreover, η_6 becomes "ready" and $\mathcal{E} = \{\eta_r, \eta_1, \eta_2, \eta_3, \eta_4, \eta_5, \eta_6, \eta_7, \eta_8, \eta_9\}$.*
- *Calls childrenReady($\{\eta_r, \eta_1, \eta_2, \eta_3, \eta_4, \eta_5, \eta_6, \eta_7, \eta_8, \eta_9\}$), which returns $\{\eta_2\}$, then* buildElementContent($\mathcal{WR}$, `'title author+ (customer_rate|no_rate) price year'`, `' '`, $\emptyset, \{\eta_3, \eta_4, \eta_5, \eta_6, \eta_8, \eta_9\}$)*. The type of regular expression here is $\alpha = \alpha_1\alpha 2$, where $\alpha_1 =$ `'title'` while α_2 equals the rest of the expression. As a consequence, a recursive call of* buildElementContent *is required.*
  - buildElementContent($\mathcal{WR}$, `'title'`, `'author+ (customer_rate|no_rate) price year'`, $\emptyset, \{\eta_3, \eta_4, \eta_5, \eta_6, \eta_8, \eta_9\}$) *sets $\eta_3 =$ (`title`, [32]) with pos $= 1$ and state "ready", then $CM = \{\eta_3\}$ and $RCM = \{\eta_4, \eta_5, \eta_6, \eta_8, \eta_9\}$;*
  - buildElementContent($\mathcal{WR}$, `'author+ (customer_ rate|no_rate) price year'`, `' '`, $\{\eta_3\}, \{\eta_4, \eta_5, \eta_6, \eta_8, \eta_9\}$) *requires a new recursive call:*
    - buildElementContent($\mathcal{WR}$, `'author+'`, `'(customer_rate|no_rate) price year'`, $\{\eta_3\}$, $\{\eta_4, \eta_5, \eta_6, \eta_8, \eta_9\}$) *would require further recursive calls until all the information fragments having* `author` *as element are processed. In this example, however, there is one only fragment of type* `author`*, so* buildElementContent *yields in one step the following: $\eta_4 =$ (`author`, [36]), with pos $= 2$ and state "ready", $CM = \{\eta_3, \eta_4\}$ and $RCM = \{\eta_5, \eta_6, \eta_8, \eta_9\}$;*
    - buildElementContent($\mathcal{WR}$, `'(customer_rate|no_rate) price year'`, `' '`, $\{\eta_3, \eta_4\}, \{\eta_5, \eta_6, \eta_8, \eta_9\}$) *recursively calls:*
      - buildElementContent($\mathcal{WR}$, `'(customer_rate|no_rate)'`, `'price year'`, $\{\eta_3, \eta_4\}$, $\{\eta_5, \eta_6, \eta_8, \eta_9\}$)*, which involves a regular expression of type $\alpha = \alpha_1|\alpha_2$. As the satisfiable expression is $\alpha = \alpha_1$, the new recursive call* buildElementContent($\mathcal{WR}$, `'customer_rate'`, `'price year'`, $\{\eta_3, \eta_4\}, \{\eta_5, \eta_6, \eta_8, \eta_9\}$) *yields: $\eta_6 =$ (`customer_ rate`, [43]), with pos $= 3$ and state "ready", $CM = \{\eta_3, \eta_4, \eta_6\}$ and $RCM = \{\eta_5, \eta_8, \eta_9\}$;*
      - buildElementContent($\mathcal{WR}$, `'price year'`, `' '`, $\{\eta_3, \eta_4, \eta_6\}, \{\eta_5, \eta_8, \eta_9\}$)*, which requires other two recursive calls. The result is $\eta_5 =$ (`year`, [38]), with pos $= 4$ and state "ready", $\eta_9 =$ (`price`, [61]), with pos $= 5$ and state "ready", $CM = \{\eta_3, \eta_4, \eta_5, \eta_6, \eta_9\}$ and $RCM = \{\eta_8\}$.*

No other child of η_2 is to be processed, so we have $\eta_2 = (\texttt{book}, [30])$, with pos $= -1$ and state "ready", and $\mathcal{E} = \{\eta_r, \eta_1, \eta_2, \eta_3, \eta_4, \eta_5, \eta_6, \eta_7, \eta_9\}$.

- *Calls childrenReady($\{\eta_r, \eta_1, \eta_2, \eta_3, \eta_4, \eta_5, \eta_6, \eta_7, \eta_9\}$), which returns $\{\eta_1\}$.* *build-ElementContent*($\mathcal{WR}$, '`book+`', ' ', $\emptyset, \{\eta_2\}$) *yields in one step (i.e. there is one book in the portion of document under consideration) $\eta_2 = (\texttt{book}, [30])$, with pos $= 1$ and state "ready", and $\eta_1 = (\texttt{store}, [24])$, with pos $= -1$ and state "ready".*
- *Calls childrenReady($\{\eta_r, \eta_1, \eta_2, \eta_3, \eta_4, \eta_5, \eta_6, \eta_7, \eta_9\}$), which returns η_r.* *buildElementContent*($\mathcal{WR}$, '`store`', ' ', $\emptyset, \{\eta_1\}$) *yields $\eta_1 = (\texttt{store}, [24])$, with pos $= 1$ and state "ready", and $\eta_r = (\texttt{doc}, [0])$, with pos $= 0$ and state "ready".*

All the assigned information fragments are now "ready". The final result is the preferred model $\mathcal{E} = \{\eta_r, \eta_1, \eta_2, \eta_3, \eta_4, \eta_5, \eta_6, \eta_7, \eta_9\}$, which will be processed by buildDoc *in order to build the output extracted XML document.* $\lhd$

3.4 Wrapper generation

As previously mentioned in Sect. 3.1.1, most Web wrapper generation systems fall into one of the following categories: wrapper generation based on inductive learning methods, or wrapper generation based on visual support tools. Both categories require human intervention, although with a substantial difference: in the wrapper induction approach, a domain expert is usually required to perform the preliminary step of manually labeling as many examples as possible (training data); differently, in the visual support based approach, a user plays the role of wrapper designer as she/he employs tools (e.g. Web browser components with content rendering and displaying capabilities) for the specification of extraction rules.

It is worth emphasizing that, although semi-automatic wrapping approaches turned out to be more effective and flexible than fully automatic ones (cf. Sect. 3.1.1), both labeling examples and visual selection of exemplary portions of the input pages to generate wrappers rely substantially on an implicit knowledge or a predefined description of the structure of the page containing the target data.

Our approach to Web wrapper generation can be seen as a "hybrid" approach, in that it features a double action. Indeed, the core of our schema-based wrapping approach was implemented in a visual support based wrapping system, named *SCRAP*, which offers visual tools to assist the wrapper designer in specifying the extraction rules and the extraction schema. Nevertheless, an inductive learning method is also proposed to speed up the specification of SCRAP wrappers and make them robust with respect to structural changes occurring in source HTML documents.

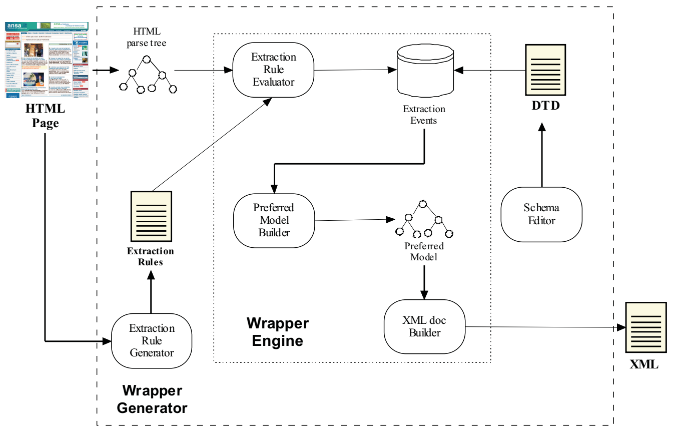


Fig. 3.6: Conceptual architecture of SCRAP

A major advantage of this hybrid approach is that the wrapper designer is only required to correctly specify a wrapper for a given set of HTML documents, but she/he does not need to care about detailed specification of extraction rules; rather, possible redundancy in the initial wrapper can be overcome by automatically generalize the early set of extraction rules. The generalized wrapper turns out to be much lesser constrained to layout details of source HTML documents and can be even effectively evaluated on documents structurally modified with respect to an initial source set.

In the following, we firstly present the SCRAP system and the conceptual framework for generalizing schema-based wrappers. In Sect. 3.4.2 we turn attention on our method for learning robust (i.e. generalized) schema-based wrappers.

3.4.1 The SCRAP system

The *SCRAP (SChema-based wRAPper for web data)* system is an XML-enabled wrapping system, in that it wraps data extracted from a source HTML page into an XML document [40, 42]. SCRAP is also able to allow text data handling for string-based extraction, providing a full support for condition atoms based on regular expressions.

The SCRAP architecture, depicted in Fig. 3.6, consists of two major components: a *Wrapper Engine* and a *Wrapper Generator* tool.

The Wrapper Engine is the SCRAP's core, as it provides a running environment for wrapper evaluation. It takes three documents as input: the parse tree of a source HTML page, a DTD representing the extraction schema, and an XML document containing the extraction rules. DTD and extraction rules can be specified using the Wrapper Generator tool. Wrapper Engine performs the following steps:

1. builds the tree of information fragments, in a top-down fashion, on the basis of the subsequences resulting from the evaluation of each extraction rule against the source HTML page;
2. computes the preferred extraction model conforming to the extraction schema, performing a bottom-up analysis of the extraction tree;
3. builds the XML document containing the extracted data.

The Wrapper Generator consists of two visual tools: the *Schema Editor* and the *Extraction Rule Generator*. They are devoted to specify, respectively, the extraction schema and the extraction rules for a source HTML page (Fig. 3.7).

Schema Editor is essentially a visual DTD editor that allows the wrapper designer to visually specify the extraction schema. This is represented as an acyclic graph, whose nodes are DTD elements and edges represent mappings from elements to element type definitions. Schema Editor also offers a simple XML editor equipped with a syntax highlighting component.

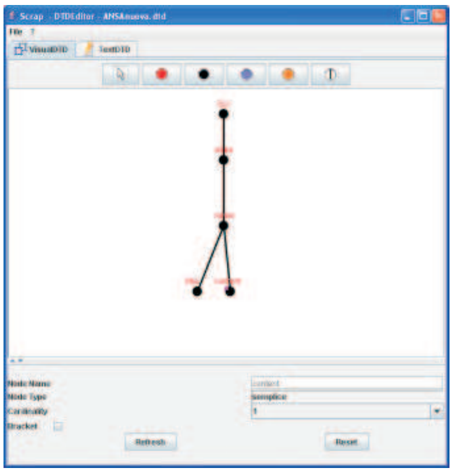
Extraction Rule Generator provides visual wizards that guide the designer through the specification of extraction rules and their constituents (i.e. filters and conditions). The module embeds a Web browser component which allows representative portions of page to be marked for the design of each extraction rule. The designer is thus completely shielded from the formatting specifics of the source page, since she/he is not required to know how a highlighted portion of interest is specified in the page: indeed, the location path of a representative instance of the target rule is automatically identified by accessing the parse tree of the page.

Figure 3.8 shows the SCRAP-based framework for learning robust wrappers. This framework is essentially composed of modules which are able to perform four tasks, once defined a specific HTML source and specified the relative extraction schema: 1) designing and evaluating an initial wrapper, 2) producing candidate generalized wrappers (i.e. their sets of extraction rules), 3) validating candidate generalized wrappers, 4) selecting a most general wrapper.

Task 1 is accomplished by the SCRAP's core (Wrapper Engine), with the support of the Schema Editor and Extraction Rule Generator modules previously described. The output is a set of extracted XML documents (named as 'early XML docs' in Fig. 3.8) for a given source of HTML documents.

The Extraction Rule Generalizer module is in charge of producing candidate generalized wrappers (task 2), that is a collection of generalized sets of extraction rules stored in the Generalized Rules repository. Each of such generalized sets of extraction rules is given in input to the Wrapper Engine, which correspondingly will produce a set of extracted XML documents. Thus, evaluating candidate generalized wrappers is an iterative step, represented in Fig. 3.8 by using bold dashed arrows.

Each candidate set of extracted XML documents is then compared to the earlier set of extracted XML documents by the Wrapping Validator module



(a)

(b)

Fig. 3.7: Sample snapshots of SCRAP tools: (a) Schema Editor, (b) Extraction Rule Generator

(task 3). This module marks the sets of generalized extraction rules corresponding to valid wrappers, and collects them into the Valid Generalized Rules repository. Finally, the Selector module selects non-deterministically a valid set of generalized rules, which represents the most general wrapper.

3.4.2 Wrapper generalization

Wrapper generalization is here intended as a task for learning *robust* schema-based wrappers. Given an example (initial) set of HTML documents and a

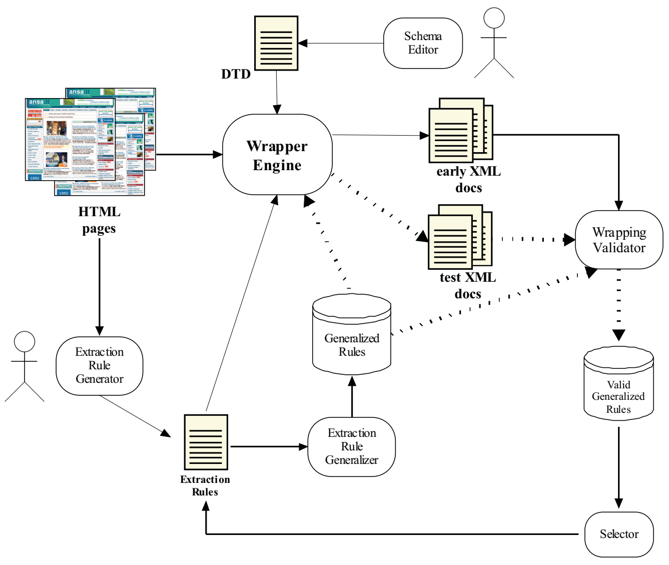


Fig. 3.8: Conceptual architecture of the SCRAP-based framework for wrapper generalization

valid wrapper visually designed for such documents, the wrapper generalization task automatically provides a new wrapper, with the same schema but more general (less redundant) extraction rules than the early wrapper. The generalized wrapper is required to behave in the same way as the early one, i.e. it is able to produce the same set of extracted XML documents when applied to the example set of HTML documents.

It is worth noticing that we do not need to perform time-consuming labeling of example documents, differently from traditional wrapper learning approaches, such as e.g. [46, 59, 68, 84].

Before going into the details of our wrapper generalization approach, we introduce useful definitions and notations. A set of extraction examples is meant as a set of pairs $\langle doc, xdoc \rangle$, such that doc is an HTML document and $xdoc$ is an extracted XML document. Given a set $\mathcal{S} = \{doc_1, \ldots, doc_n\}$ of HTML documents, the set of extraction examples defined by $\mathcal{WR}$ with respect to $\mathcal{S}$ is the set of pairs $Ex^{\mathcal{S}}_{\mathcal{WR}} = \{\langle doc, xdoc_{\mathcal{WR}}(doc)\rangle |\ doc \in \mathcal{S}\}$. Let $\mathcal{WR}$ be a wrapper and Ex be a set of extraction examples. We say that $\mathcal{WR}$ is valid with respect to Ex if and only if for each pair $\langle doc, xdoc \rangle \in Ex$ it holds that $xdoc_{\mathcal{WR}}(doc)$ is equal to $xdoc$. The set of wrappers that are valid with respect to Ex is denoted as $\mathcal{WR}_{Ex}$.

Extraction rule and wrapper entailment

Our approach of wrapper generalization is meant as a specific-to-general learning task in a standard logic fashion. Thus, we are initially interested in devising a suitable notion of wrapper entailment. The key idea is that a relationship of entailment between two any wrappers lies on a comparison, in terms of preference, between the corresponding extracted documents. It is easy to see that Def. 3.18 applies also to pairs of extraction models obtained using different wrappers sharing the same extraction schema.

Definition 3.25 (Wrapper entailment). *Let Ex be a set of extraction examples and $\mathcal{WR}' = \langle \mathcal{D}, \mathcal{R}', w' \rangle$ and $\mathcal{WR}'' = \langle \mathcal{D}, \mathcal{R}'', w'' \rangle$ be two wrappers in $\mathcal{WR}_{Ex}$. $\mathcal{WR}'$ entails $\mathcal{WR}''$ with respect to Ex ($\mathcal{WR}' \models_{Ex} \mathcal{WR}''$) if and only if, for each document $doc \notin Ex$, either $xdoc_{\mathcal{WR}''}(doc) \prec xdoc_{\mathcal{WR}'}(doc)$ or $xdoc_{\mathcal{WR}''}(doc) = xdoc_{\mathcal{WR}'}(doc)$.*

Clearly, since the set of possible input documents is infinite, the above notion of wrapper entailment cannot be directly exploited for the definition of a wrapper generalization framework. In fact, checking whether a wrapper $\mathcal{WR}$ is more general than an early one would require to check that, for each possible input HTML document, the output of $\mathcal{WR}$ is preferred to the output of the early wrapper.

An effective strategy for wrapper generalization can be devised by involving the constituents of a wrapper, that is its extraction schema and set of extraction rules. However, the extraction schema is assumed to be fixed as it represents the schema of information to be extracted. Thus, wrapper generalization involves only the set of extraction rules.

We now introduce the notion of extraction rule entailment, which poses the basis for wrapper generalization.

The notion of rule entailment lies on the containment of of node sequences containment. Given two node sequences $s' = [n'_1, \ldots, n'_k]$ and $s'' = [n''_1, \ldots, n''_h]$, we say that s'' contains s' ($s' \subseteq s''$) if and only if $n'_1 = n''_1$, $n'_k = n''_h$ and there exists a subsequence $s''_k = [n''_{i,1}, \ldots, n''_{i,k}]$ of s'' such that $n'_j = n''_{i,j}$, $\forall n'_j \in s'$ and $n''_{i,j} < n''_{i,j+1}$.

Definition 3.26 (Extraction rule entailment). *Let r' and r'' be two extraction rules. r' entails r'' ($r' \models r''$) if and only if, for each HTML document doc and for each node sequence s of doc, it holds that for each sequence $s' \in r'(s)$ there exists a sequence $s'' \in r''(s)$ such that $s' \subseteq s''$.*

Property 3.27. Let Ex be a set of extraction examples and $\mathcal{WR}' = \langle \mathcal{D}, \mathcal{R}', w' \rangle$ and $\mathcal{WR}'' = \langle \mathcal{D}, \mathcal{R}'', w'' \rangle$ be two wrappers in $\mathcal{WR}_{Ex}$. If it holds that $w'(e', e'') \models w''(e', e'')$ for each pair of elements e', e'' appearing in $\mathcal{D}$ then $\mathcal{WR}' \models_{Ex} \mathcal{WR}''$.

Proof. We prove that for each document *doc* if, for each pair of elements e', e'' appearing in $\mathcal{D}$ it holds that $w'(e', e'') \models w''(e', e'')$, then it holds that

$xdoc_{\mathcal{WR}''}(doc) \prec xdoc_{\mathcal{WR}'}(doc)$ or $xdoc_{\mathcal{WR}''}(doc) = xdoc_{\mathcal{WR}'}(doc)$, reasoning by contradiction. Assume that it holds that $w'(e', e'') \models w''(e', e'')$ for each pair of elements e', e'' appearing in $\mathcal{D}$ and there is a document *doc* such that $xdoc_{\mathcal{WR}'}(doc) \neq xdoc_{\mathcal{WR}''}(doc)$ and $xdoc_{\mathcal{WR}''}(doc) \not\prec xdoc_{\mathcal{WR}'}(doc)$. Then, the preferred extraction model $\mathcal{E}''$ of $\mathcal{WR}''(doc)$ is different from the preferred extraction model $\mathcal{E}'$ of $\mathcal{WR}'(doc)$ but $\mathcal{E}'' \not\prec \mathcal{E}'$, that is $\mathcal{E}''(\eta_r) \not\prec \mathcal{E}'(\eta_r)$ (Def. 3.18).

In this case, none of the conditions of precedence between extraction models in Def. 3.18 holds, that is one of the following cases occurs:

- $elnames(\mathcal{E}''(\eta_r)) = [e''_1, \ldots, e''_n]$ and $elnames(\mathcal{E}'(\eta_r)) = [e'_1, \ldots, e'_k]$ are different and the former does not precede the latter; thus, there exists an integer $i \geq 0$ such that $e'_j = e''_j$, for each $j < i$, $e'_i \neq e''_i$ and e''_i does not precede e'_i, therefore e'_i precedes e''_i. This implies that there not exists the fragment $\eta(e'_i, s)$ in $\mathcal{E}''(\eta_r)$, that is there not exists a sequence produced by $w''(e, e'_i)$, where e is parent of e'_i, otherwise $\mathcal{E}''$ would not be the preferred extraction model of $\mathcal{WR}''$.
  Let s' be the sequence associated with e'_i in $\mathcal{E}'(\eta_r)$. Since $w'(e, e'_i) \models w''(e, e'_i)$, there exists a sequence s'' produced by $w''(e, e'_i)$ such that $s' \subseteq s''$, thus yielding a contradiction.
- $elnames(\mathcal{E}'(\eta_r))$ is equal to $elnames(\mathcal{E}''(\eta_r))$ and there exists an integer $i \geq 0$ such that:
  - $\mathcal{E}'(\eta_r) = [(e_0, s_0), \ldots, (e_{i-1}, s_{i-1}),\ (e_i, s_i^{(1)}), \ldots,\ (e_k, s_k^{(1)})]$, and
  - $\mathcal{E}''(\eta_r) = [(e_0, s_0), \ldots, (e_{i-1}, s_{i-1}),\ (e_i, s_i^{(2)}), \ldots,\ (e_k, s_k^{(2)})]$, and
  - $s_i^{(1)}$ is different from $s_i^{(2)}$, but $s_i^{(2)}$ does not precede $s_i^{(1)}$.

  Let e be parent of e_i. Since $w'(e, e'_i) \models w''(e, e'_i)$, there exists a sequence s produced by $w''(e, e'_i)$ such that $s_i \subseteq s$. However, since $\mathcal{E}''$ is a preferred extraction model, $s_i^{(2)}$ precedes s, thus $s_i^{(2)}$ precedes $s_1^{(1)}$, which yields a contradiction.
- $\mathcal{E}'(\eta_r) = \mathcal{E}''(\eta_r) = [\eta_0, \ldots, \eta_k]$, and there exists an integer $i \geq 0$ such that, for each $j < i$, $\mathcal{E}'^*(\eta_j) = \mathcal{E}''^*(\eta_j)$ and $\mathcal{E}''(\eta_i) \neq \mathcal{E}'(\eta_i)$, but $\mathcal{E}''(\eta_i) \not\prec \mathcal{E}'(\eta_i)$. Applying inductively to $\mathcal{E}''(\eta_i)$ and $\mathcal{E}'(\eta_i)$ the same reasoning applied to $\mathcal{E}''(\eta_r)$ and $\mathcal{E}'(\eta_r)$ it can be proved that it is not possible that $\mathcal{E}''(\eta_i) \not\prec \mathcal{E}'(\eta_i)$ holds.

Therefore, $\mathcal{E}''(\eta_r) \prec \mathcal{E}'(\eta_r)$ and $\mathcal{E}'' \prec \mathcal{E}'$. It follows that, in the case $\mathcal{WR}'(d) \neq \mathcal{WR}''(d)$, $\mathcal{WR}''(d) \prec \mathcal{WR}'(d)$ holds, which concludes the proof. $\square$

Extraction rule and wrapper subsumption

In order to define a generalization strategy for XPath extraction rules, we seamlessly approximate wrapper entailment by exploiting the notion of *subsumption* [83] to define an efficient algorithm for wrapper generalization.

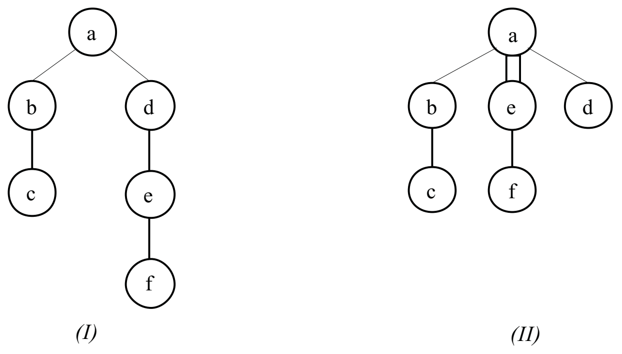


Fig. 3.9: Tree patterns

Firstly, we introduce some generalization operations to be applied to XPath expressions, secondly, we define the notion of atomic generalization between XPath expressions and, finally, we define the concepts of rule subsumption and wrapper subsumption.

Generalization steps.

In the following, we consider XPath expressions of the form $./s_1/\ldots/s_n$. Each s_i is an XPath steps of the form $a :: n\,\Pi$, where a is an XPath axis, n is an XPath node test and Π is a sequence of XPath predicates of the form $\Pi = [\pi_1]\ldots[\pi_t]$.

An XPath expression can be generalized by removing individual steps or applying relaxations to step components (i.e. node test, axis, predicates). Specifically, we can delete a predicate or a node test, and we can transform a child axis in a descendant one.

Moreover, we consider a transformation called *subtree promotion* as generalization operation. This transformation was originally introduced in [7] for relaxing a tree pattern query. Given a tree, a subtree promotion consists of moving up a subtree of one level, that is a subtree rooted at t is promoted if the edge between t and its parent p is removed and a descendant edge connecting the parent of p to t is created. As an example, consider the tree patterns shown in Fig. 3.9. Tree pattern 3.9(II) is obtained from tree pattern 3.9(I) by promoting the subtree rooted at e. Note that tree pattern 3.9(II) is more general than tree pattern 3.9(I). Finally, since an XPath predicate is also an XPath expression, all generalization operations can be applied to it. Therefore, the generalization of a predicate is also an atomic generalization step.

We consider the following operations as atomic generalization steps applicable to XPath expressions:

- deletion of a step;

- relaxation of a child axis;
- deletion of node test;
- deletion of a predicate;
- subtree promotion;
- generalization of a predicate.

Informally, an XPath expression l' is said to be an *atomic generalization* of an XPath expression l if and only if l' is obtained from l by applying one of the above atomic generalization steps.

Example 3.28. Consider the XPath expressions and the corresponding tree patterns depicted in Fig. 3.10. The XPath expression l reported inside the box of Fig. 3.10 is the expression to be generalized. Figure 3.10(1) shows the expression l_1 obtained from l by deleting the second last step, whereas Figg. 3.10(2), 3.10(3) and 3.10(4) show expressions obtained from l by applying relaxations to step components. Specifically, l_2 is yielded by relaxing the child axis of the second step, l_3 is provided by deleting the node test of the first step, and l_4 is obtained by deleting the last predicate of the second step.

Furthermore, Figg. 3.10(5) and 3.10(6) illustrate two examples of application of subtree promotion to l. Specifically, l_5 is yielded by promoting the subtree rooted at e, whereas l_6 is provided by promoting the subtree rooted as d. It can be noticed that the promotion performed in l_5 concerns a step located in the path conveying to the output node. Therefore, in order to not change the output node, it is necessary that the step preceding the node to be promoted becomes a predicate. We distinguish two cases of promotion: step subtree promotion *(used to provide l_5) and* predicate subtree promotion *(employed to obtain l_6). The latter concerns the case when the node to be promoted is the node test of the first step of an expression contained in a predicate, since it is necessary to move the whole predicate to the previous step.*

Finally, Fig. 3.10(7) shows the expression l_7 obtained from l by applying a generalization of a predicate*, that is the application of one of the operations described above to the XPath expression contained in the predicate. Specifically, l_7 is provided by applying the deletion of a node test to the first predicate of the second step of l.* ◁

We note two particular cases of application of step subtree promotion and predicate subtree promotion. Consider the XPath expression $./a/b[./d[./c][./e/f]]/g$. The promotion of the subtree rooted at f (step subtree promotion) produces the XPath expression $./a/b[./d[./c][.[./e]//f]]$, that has a redundant form since it can be simply expressed as $./a/b[./d[./c][.//f][./e]]$, wherein predicate $[.[./e]//f]]$ has been "split" in the two predicates $[.//f]$ and $[./e]$. Furthermore, consider again the XPath expression $./a/b[./d[./c][./e/f]]/g$ and suppose to apply a promotion to the subtree rooted at c (predicate subtree promotion). In this case, the expression $./a/b[.[.//c]/d[./e/f]]/g$ is obtained, which is the equivalent to the expression

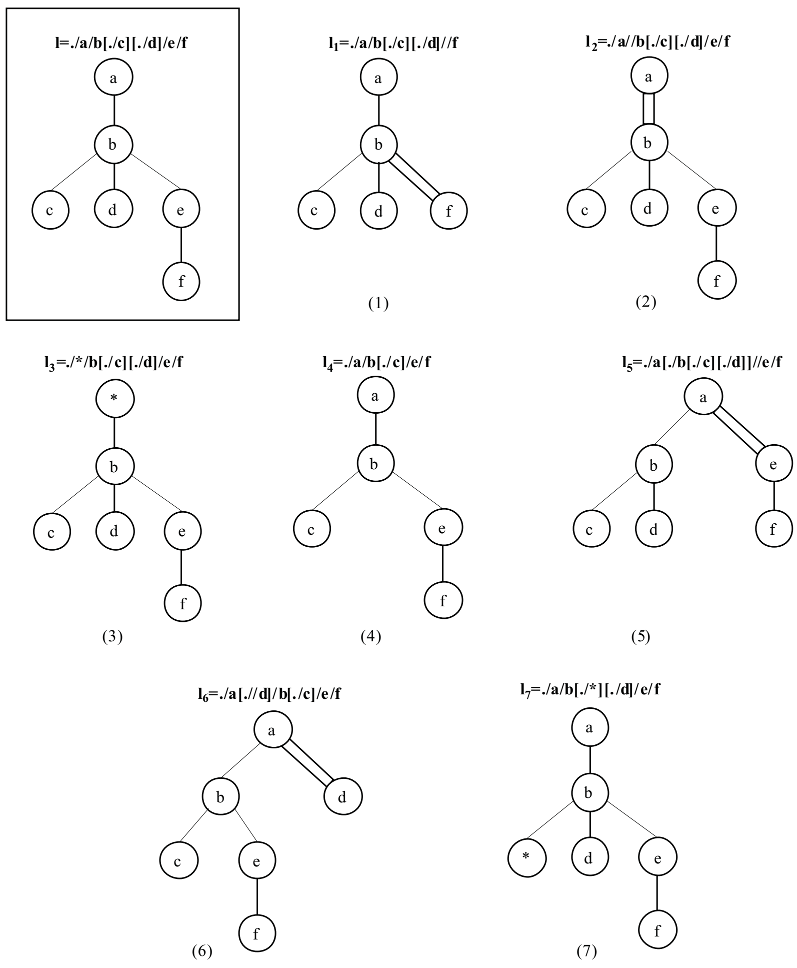


Fig. 3.10: A tree pattern and its atomic generalizations

$./a/b[.//c][./d[./e/f]]/g$. In the following, whenever an expression of the form $[.[.l]l']$ is obtained, we consider the equivalent form $[.l][.l']$.

In order to formally define the atomic generalization, we provide some notation. Given an XPath expression $l = ./s_1/\dots/s_n$, where $s_1 = a_1 :: n_1\Pi_1$, we denote as $desc(l)$ the XPath expression obtained from l by imposing a descendant axis as axis of first step, i.e. $desc(l) = ./descendant :: n_1\Pi_1/\dots/s_n$. Given two XPath predicates $p = [./s_1/\dots/s_n], p' = [./s'_1/\dots/s'_m]$, we denote as $unite(p, p')$ the XPath predicate $[.[./s_1/\dots/s_n]/s'_1/\dots/s'_m]$. Note that $unite(\perp, p') = p'$. Given an XPath expression $l = ./s_1/\dots/s_n$, we consider as s_0 the step containing the self axis ".", i.e. $s_0 = a :: n\Pi$, where $a = self$, $n = node()$ and $\Pi = \perp$.

Definition 3.29 (Atomic generalization). *Given two XPath expressions* $l = ./s_1/.../s_n$ *and* $\bar{l} = ./\bar{s}_1/\dots/\bar{s}_m$, *where* $s_i = a_i :: n_i \Pi_i$ *and* $\bar{s}_j = \bar{a}_j :: \bar{n}_j \bar{\Pi}_j$ *are XPath steps, with* $i \in [1..n]$ *and* $j \in [1..m]$, $\bar{l}$ *is an atomic generalization of* l ($l \rightarrow \bar{l}$) *if and only if one and only one of the following conditions holds:*

* *[*deletion of a step*]:* $m = n-1$ *and there exists* $k \in [1..n]$ *such that* $\bar{s}_h = s_h$, $\forall h \in [1..k-1]$ *and if* $k \neq n$ *it holds that* $\bar{s}_k = descendant :: n_{k+1}\Pi_{k+1}$ *and* $\bar{s}_r = s_{r+1}$, $\forall r \in [k+1..m]$*;*
* *[*relaxations of step components*]:* $n = m$ *and there exists* $k \in [1..n]$ *such that for each* $h \in [1..n]$ *with* $h \neq k$ *it holds that* $\bar{s}_h = s_h$*. Moreover, let* s_k *be the Xpath step* $a :: n\Pi$ *and* $\bar{s}_k$ *be the XPath step* $\bar{a} :: \bar{n}\bar{\Pi}$ *with* $\Pi = [\pi_1]\dots[\pi_y]$ *and* $\bar{\Pi} = [\bar{\pi}_1]\dots[\bar{\pi}_z]$*, exactly one of the following cases must occur:*
  - *[*generalization of a predicate*]:* $a = \bar{a}$, $n = \bar{n}$, *and there exists* $g \in [1..y]$ *such that* $\bar{\pi}_r = \pi_r$, $\forall r \in [1..y], r \neq g$ *and either:*
    - $z = y$ *and* $\pi_g \rightarrow unite(\perp, [\bar{\pi}_g])$ *or*
    - $z = y+1$ *and* $\pi_g \rightarrow unite([\bar{\pi}_z], [\bar{\pi}_g])$*;*
  - *[*deletion of a predicate*]:* $a = \bar{a}$, $n = \bar{n}$, $z = y-1$ *and there exists* $g \in [1..y]$ *such that:* $\bar{\pi}_w = \pi_w$, $\forall w \in [1..g-1]$ *and* $\bar{\pi}_v = \pi_{v+1}$, $\forall v \in [g..z]$*;*
  - *[*relaxation of child axis*]:* $a = child$, $\bar{a} = descendant$, $n = \bar{n}$ *and* $\Pi = \bar{\Pi}$*;*
  - *[*deletion of node test*]:* $a = \bar{a}$, $n \neq *$, $\bar{n} = *$, *and* $\Pi = \bar{\Pi}$*.*
* *[*step subtree promotion*]: there exists* $k \in [2..n]$ *such that* $m = n-1$, $\bar{s}_h = s_h$ *for each* $h \in [1..k-3]$, $\bar{s}_{k-2} = a_{k-2} :: n_{k-2}\Pi_{k-2}[./s_{k-1}]$, $\bar{s}_{k-1} = desc(s_k)$ *and* $\bar{s}_w = s_{w+1}$ *for each* $w \in [k+1..m]$*;*
* *[*predicate subtree promotion*]:*$m = n$ *and there exists* $k \in [1..n]$ *such that* $\bar{s}_h = s_h$ *for each* $h \in [1..k-2]$, $\bar{s}_z = s_z$ *for each* $z \in [k+1..n]$, $s_k = a_k :: n_k[\pi_k^1]\dots[\pi_k^r]$ *and there exists* $g \in [1..r]$ *such that:*
  - $\bar{s}_k = a_k :: n_k[\bar{\pi}_k^1]\dots[\bar{\pi}_k^{r-1}]$, *where* $\bar{\pi}_k^v = \pi_k^v$ *for each* $v \in [1..g-1]$ *and* $\bar{\pi}_k^w = \pi_k^{w+1}$ *for each* $w \in [g..r-1]$ *and*
  - $\bar{s}_{k-1} = a_{k-1} :: n_{k-1}\Pi_{k-1}[desc(\pi_k^g)]$.

Starting from an XPath expression and repeatedly applying atomic generalization steps, an XPath expression that is a generalization of the first one is obtained. Given two XPath expressions l and l', l' is a *generalization* of l ($l \rightarrow^* l'$) if and only if there exists a finite sequence of XPath expressions $l_0, \dots, l_n$ such that $l = l_0$, $l' = l_n$ and $l_{i-1} \rightarrow l_i$, $\forall i \in [1..n]$.

We now introduce the concept of atomic generalization of an XPath atom $\$c : l \twoheadrightarrow \$t$, which essentially consists in an atomic generalization of l. Formally, given two XPath atoms $p = \$c : l \twoheadrightarrow \$t$ and $p' = \$c' : l' \twoheadrightarrow \$t'$, p is an *atomic generalization* of p' ($p \rightarrow p'$) if and only if $\$c = \$c'$, $\$t = \$t'$ (p and p' use the same variables) and $l \rightarrow l'$ (l is an atomic generalization of l').

The atomic generalization of an extraction filter is yielded by applying an atomic generalization step to one of its components. Specifically, given two extraction filters $f = \langle tp, \mathcal{P} \rangle$ and $f' = \langle tp', \mathcal{P}' \rangle$, f' is an *atomic generalization* of f ($f \rightarrow f'$) if and only if one and only one of the following conditions holds:

1. $tp \rightarrow tp'$ and $\mathcal{P} = \mathcal{P}'$;
2. $tp = tp'$, $|\mathcal{P}| = |\mathcal{P}'|$ and there exist $p \in \mathcal{P}$ and $p' \in \mathcal{P}'$ such that $\mathcal{P} - \{p\} = \mathcal{P}' - \{p'\}$ and $p \rightarrow p'$;
3. $tp = tp'$ and there exists a condition atom $p \in \mathcal{P}$ such that $\mathcal{P}' = \mathcal{P} - \{p\}$;
4. $tp = tp'$ and there exists a substitution θ such that $\mathcal{P}'\theta = \mathcal{P}$ and:
   - $\theta = \{x/c\}$, such that $c \in \mathcal{C}(\mathcal{P}), x \in \mathcal{V}(\mathcal{P}')$, or
   - $\theta = \{y/x, z/x\}$ such that $y, z \in \mathcal{V}(\mathcal{P}'), x \in \mathcal{V}(\mathcal{P})$,

   where $\mathcal{C}(\mathcal{P})$ and $\mathcal{V}(\mathcal{P})$ denote the set of all constant and variable names, respectively, appearing in atoms in $\mathcal{P}$;

Let $r = \langle EF, as, all\rangle$ and $r' = \langle EF', as', all\rangle$ be two extraction rules[4], where $EF = \{f_1 \vee \ldots \vee f_m\}$, $EF' = \{f'_1 \vee \ldots \vee f'_m\}$, $as = [min, max]$ and $as = [min', \infty]$. r' is derived from r ($r \rightarrow r'$) by applying an atomic *step of generalization* if and only if one and only one of the following conditions holds:

1. there exist f_j and f'_j such that $f_j \rightarrow f'_j$ and $f_i = f'_i, \forall i \in [1..m], i \neq j$;
2. $EF = EF'$ and $min' = min$-1.

Definition 3.30 (Rule subsumption). *Given two extraction rules r and r', r' subsumes r ($r' \vdash r$) if there exists a sequence of extraction rules $r_0, \ldots, r_n$ such that $r_0 = r$, $r_n = r'$, and $r_{i-1} \rightarrow r_i, \forall i \in [1..n]$.*

Definition 3.31 (Wrapper subsumption). *Given two wrappers $\mathcal{WR}_1 = \langle \mathcal{D}, \mathcal{R}_1, w_1\rangle$ and $\mathcal{WR}_2 = \langle \mathcal{D}, \mathcal{R}_2, w_2\rangle$, $\mathcal{WR}_2$* subsumes *$\mathcal{WR}_1$ ($\mathcal{WR}_2 \vdash \mathcal{WR}_1$) if and only if, for each pair of elements names e_1, e_2 in $\mathcal{D}$, $w_2(e_1, e_2) \vdash w_1(e_1, e_2)$.*

Lemma 3.32. *Let l and l' be two XPath expressions. If $l \rightarrow^* l'$ then, for each HTML document doc, the set of nodes obtained by applying l' to doc contains the set of nodes obtained by applying l to doc.*

Proof. Let T and T' be the tree patterns corresponding to l and l', respectively. The statement is proved by showing that if $l \rightarrow^* l'$ then there exists a homomorphism containment between T and T'.[5]

We prove the existence of a homomorphism from T' to T in the case of $l \rightarrow l'$, since the general case $l \rightarrow^* l'$ is trivially deducible from it.

We first consider the case when l' is obtained by applying a subtree promotion generalization step to l. Assume without loss of generality that $n_T \in \mathcal{T}$ is the root of the subtree promoted in T'. Let $n^a_T \in \mathcal{T}$ be parent of the n_T's parent. We denote as $n_{T'} \in \mathcal{T}'$ the node corresponding to the root of the promoted subtree and as $n^a_{T'} \in \mathcal{T}'$ the parent of $n_{T'}$.

[4] We only apply generalization steps to extraction rules having *all* as relative filter, since generalizing rules having other kinds of relative filters can yield nonmonotonic rules.

[5] We recall that a homomorphism from a tree pattern T' to a tree pattern T is a total mapping from the nodes of T' to the nodes of T, such that node types and structural relationships are preserved [81].

The mapping from $n^a_{T'}$ to n^a_T and from $n_{T'}$ to n_T (and from all the other nodes of T' to the corresponding nodes in T) is a homomorphism since:

- the label of $n^a_{T'}$ is equal to the label of n^a_T, and the label of $n_{T'}$ is equal to the label of n_T;
- $n^a_{T'}$ is connected to $n_{T'}$ by means of a descendant edge and n_T is a descendant of n^a_T.

The case of predicate subtree promotion is analogous. The deletion of a step or a predicate from an expression l whose associated tree pattern is T produces a tree pattern that is obviously contained in T, since the obtained expression is a subexpression of l, then the existence of a homomorphism is straightforward. Other cases are trivial. □

Theorem 3.33. *Given two extraction rules r and r', $r' \vdash r \Rightarrow r \models r'$.*

Proof. Let r be of the form $\langle EF, as, all\rangle$ and r' be of the form $\langle EF', as', all\rangle$. We only prove the statement in the case of $r \rightarrow r'$, since the case $r \rightarrow^* r'$ is deducible from it. We list all the ways to derive r' from r by applying an atomic generalization step and we prove the thesis for each case:

* [*generalization of an internal filter*]: Let $f = \langle tp, \mathcal{P}\rangle$ be an internal filter of EF and $f' = \langle tp', \mathcal{P}'\rangle$ be an internal filter of EF' such that $f \rightarrow f'$. We recall that each filter f_i of EF with $f_i \neq f$ is equal to filter f'_i of EF' and $|EF| = |EF'|$.
  - [*generalization of the target predicate*]: In this case it holds that $tp \rightarrow tp'$. From Lemma 3.32 it holds that the set of nodes extracted from the XPath expression of tp is contained in the set of nodes extracted from the XPath expression of tp' for each document *doc*. Therefore, for each sequence s yielded by evaluating tp on a sequence s_c such that $s \in r(s_c)$, there exists a sequence s' yielded by evaluating tp' on s_c such that $s \subseteq s'$. For construction $s' \in EF'(s_c)$ and it holds that $\mathcal{P}'$ is true on s', since $\mathcal{P} = \mathcal{P}'$ and $\mathcal{P}$ is true on s. Since $as' = [min', \infty]$ and $|s'| \geq |s|$, it holds that $as'(s')$ is true, thus $s' \in r'(s_c)$;
  - [*generalization of an XPath atom*]: In this case it holds that $\mathcal{P}' - \{p'\} = \mathcal{P} - \{p\}$ and $p \rightarrow p'$. Since $tp = tp'$, the node sequences on which $\mathcal{P}'$ is evaluated are the same of that on which $\mathcal{P}$ is evaluated. Therefore, for each s yielded by evaluating tp on a sequence s_c such that p is true on s and $s \in r(s_c)$, from Lemma 3.32 it holds that p' is true on s, thus $s \in r'(s_c)$;
  - [*deletion of a condition atom*]: Since $tp = tp'$, the node sequences on which $\mathcal{P}'$ is evaluated are the same of that on which $\mathcal{P}$ is evaluated. Since $\mathcal{P}' = \mathcal{P} - \{p\}$, for each s yielded by evaluating tp on a sequence s_c such that $\mathcal{P}$ is true on s and $s \in r(s_c)$, it holds that for each $p' \in \mathcal{P}'$ p' is true on s, thus $s \in r'(s_c)$;

```
Input:
  A wrapper $\mathcal{WR} = \langle \mathcal{D}, \mathcal{R}, w \rangle$;
  A set $\mathcal{S} = \{doc_1, \ldots, doc_n\}$ of example HTML documents.
Output:
  A most general wrapper with respect to $\mathcal{WR}$ and $\mathcal{S}$.
Method:
  let $\mathcal{X} = \{xdoc_1, \ldots, xdoc_n\}$ be the set of XML documents extracted
  by applying $\mathcal{WR}$ on $\mathcal{S}$;
  $\mathcal{MW} := \mathcal{WR}$;
  repeat
    $NW := \emptyset$; /* the set of generalized wrappers at the current step */
    let $\mathcal{MW} = \langle \mathcal{D}, \mathcal{R}, w \rangle$;
    for each $r \in \mathcal{R}$ do
      $R' :=$ generalize($r$);
      for each $r' \in R'$ do
        $\mathcal{WR}' := \langle \mathcal{D}, \mathcal{R} - \{r\} \cup \{r'\}, \varphi(w, r, r') \rangle$;
        if (valid($\mathcal{WR}', \mathcal{X}, \mathcal{S}$))
          $NW := NW \cup \{\mathcal{WR}'\}$;
    if ($NW \neq \emptyset$)
      $\mathcal{MW} :=$ selectWrapper($NW$);
  until ($NW = \emptyset$)
  return $\mathcal{MW}$;
```

Fig. 3.11: The MostGeneralWrapper algorithm

- [*variable substitution*]: This case follows from the implication between subsumption and entailment in the case of logic rules. Indeed, it is easy to see that our notion of substitution is equivalent to the substitution in subsumption between logic rules.

* [*generalization of the absolute filter*] Let as be of the form $[min, max]$ and as' of the form $[min', \infty]$. In this case it holds that $EF = EF'$ and, thus, $EF(s_c) = EF'(s_c)$ for each s_c. Since $min' \leq min$ and $max' \geq max$, for each $s \in r(s_c)$ it holds that $as'(s)$ is true, thus $s \in r'(s_c)$.

□

Let Ex be a set of extraction examples and $\mathcal{WR}$ and $\mathcal{WR}'$ be two wrappers in $\mathcal{WR}_{Ex}$. $\mathcal{WR}'$ is said to be a *most general generalization* of $\mathcal{WR}$ (for short, most general wrapper) if $\mathcal{WR}' \vdash \mathcal{WR}$ and, for each wrapper $\mathcal{WR}'' \in \mathcal{WR}_{Ex}$ it holds that $\mathcal{WR}'' \neq \mathcal{WR}'$, $\mathcal{WR}'' \vdash \mathcal{WR}$ and $\mathcal{WR}' \vdash \mathcal{WR}''$.

An algorithm for wrapper generalization

The computation of a most general wrapper can be performed by repeatedly generalizing extraction rules, while guaranteeing that data are correctly extracted from the given set of source documents.

Our MostGeneralWrapper algorithm (Fig. 3.11) takes a wrapper and a set of example HTML documents as input and computes a most general wrapper. Function generalize computes the set of all extraction rules that can be obtained applying atomic generalization steps. We remark that, for any newly generalized wrapper $\mathcal{WR}'$, $\varphi(w, r, r')$ returns a function w' defined as follows:

- $w'(e_i, e_j) = r'$ for each $(e_i, e_j) \in El \times El$ such that $w(e_i, e_j) = r$, and
- $w'(e_i, e_j) = w(e_i, e_j)$ otherwise.

Function valid verifies that the set $\mathcal{X}' = \{xdoc'_1, \dots, xdoc'_n\}$ of XML documents extracted by a new wrapper $\mathcal{WR}'$ from the set $\mathcal{S}$ is such that $xdoc'_i = xdoc_i, \forall\ i \in [1..n], xdoc'_i \in \mathcal{X}', xdoc_i \in \mathcal{X}$. Finally, function selectWrapper non-deterministically chooses a wrapper from the set NW of general wrappers. The algorithm iterates until no generalized wrapper can be generated anymore.

Proposition 3.34. *Let $\mathcal{WR}$ be a wrapper. The MostGeneralWrapper algorithm correctly computes a most general generalization of $\mathcal{WR}$.*

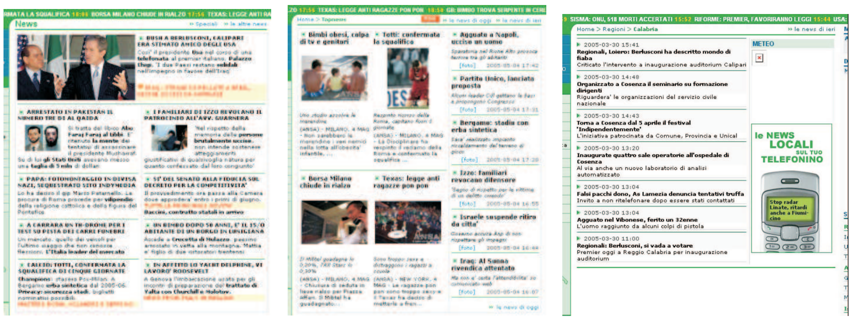
Proof. We prove the soundness of the algorithm reasoning by contradiction. Let $\mathcal{MW} = \langle \mathcal{D}, \mathcal{R}, w \rangle$ be the output wrapper, i.e. the wrapper returned by the algorithm when $NW = \emptyset$. Suppose that $\mathcal{MW}$ is not a most general wrapper with respect to the input wrapper $\mathcal{WR}$. In this case, there exists a wrapper $\mathcal{WR}' = \langle \mathcal{D}, \mathcal{R}', w' \rangle$ valid with respect to $\mathcal{S}$ that subsumes $\mathcal{MW}$, that is for each pair of elements names e_1, e_2 in $\mathcal{D}$ $w'(e_1, e_2) \vdash w(e_1, e_2)$. For Def. 3.30 it follows that there exists a sequence of extraction rules obtained by atomic generalization steps that provides $w'(e_1, e_2)$ starting from $w(e_1, e_2)$. Therefore, wrapper $\mathcal{MW}$ cannot be the output wrapper since it can be still generalized, i.e. NW cannot be empty, and then the hypothesis is contradicted. $\square$

It is easy to observe that the complexity of the MostGeneralWrapper algorithm is exponential with respect to the size of the initial wrapper, since evaluating a wrapper requires exponential time with respect to the size of the wrapper itself. However, the algorithm turns out to be useful in practice, as it performs a reasonable number of atomic steps of generalization in real cases.

3.5 Experimental evaluation

We devised experimental evaluation to pursue the following goals:

- assessing the ability of SCRAP wrappers in extracting data with the aid of extraction schemas,
- demonstrating the robustness of SCRAP wrappers with respect to minimal changes that may occur in Web pages,
- evaluating the impact of wrapper generalization to learn wrappers robust to major changes in page layouts.



(a) (b) (c)

Fig. 3.12: Excerpts of sample *ANSA* pages: (a) home page, (b) top-news page, (c) local news page

It is worth emphasizing that, differently from the case of wrapper induction systems, experimental validation of an interactive wrapper generation approach aims mainly at verifying its feasibility. Furthermore, the robustness of any semi-automatic wrapper generation system is typically affected by the human designer skills.

In the previous sections we have discussed an application of SCRAP on Amazon, illustrating the computation of a preferred extraction model. Here we gain an insight into interesting real case studies concerning the evaluation of SCRAP against Web sites having different presentation and content features.

3.5.1 Data description

We conducted experiments using three data-intensive real-world Web sites. Specifically, we monitored, during a 6-month period, pages available from selected URLs or resulting from a query issued on a site search engine.

- *Amazon* [6]. Pages listing books available at the popular online store have been chosen and discussed as our running example. For the experimental analysis, we considered the URL of a page listing the Top-25 seller books concerning a fixed subject (e.g. Computers & Internet-Programming), like that of Fig. 3.1.
- *IMDb* [61]. The biggest searchable on-line information repository about movies and TV shows. It offers reviews, plot summaries for movies, recommendation of top-movies as voted by users, filmographies, news about celebrities and daily studio briefing. We monitored the URLs of all movie pages referenced by the page listing the USA Weekly Top 50 Video Rentals.[6] This is an index page that points to movie pages changing every week.[7] We assumed to consider the following information in any movie

[6] http://www.imdb.com/boxoffice/rentals.

[7] http://www.imdb.com/title/.

page: title, year, director, genre (sequence of one or more movie categories), plot outline (a short movie summary), user comments (one or more long texts), user rating (optional), and cast overview (a list of pairs actor name and role).

- *ANSA* [9]. This is the site of an Italian news agency. We considered pages that fall into three different categories: home page, including the most relevant national and international news, top-news page,[8] containing further information about the most relevant news, and local-news page,[9] presenting news about a specific geographical region. Example ANSA homepage and top-news page are displayed in Fig. 3.12. Note that, besides heading and content, news may have also either date/time or site, and even a more complex schema including subheading or cross-heading.

Pages listing books on Amazon have typically a quite regular template, whereas movie pages on IMDB alternate short data values with even long texts and follow a template less regular than Amazon. The ANSA Web site represents the hardest test, since ANSA pages contain news items that are updated frequently and have irregular structure and different content features. In particular, a news may have an associated image, the content of a news may have formatting tags (elements) that could be not considered for extraction, more news may be contained within the same cell of a table, or a news may be structured over more table cells.

Figure 3.13 shows the extraction schemas used to design wrappers for ANSA and IMDB pages, in addition to the schema for Amazon wrappers previously introduced in Fig. 3.2.

3.5.2 Evaluation metrics

We adopted IR-based metrics, namely *precision* and *recall*, to evaluate the effectiveness of SCRAP wrappers.

Precision is a measure of the fraction of the extracted information that is correct, and recall is a measure of the fraction of information that has been correctly extracted. Loosely speaking, precision quantifies the reliability of the extracted information, whereas recall measures how much of the information to be extracted has been really extracted. Extracted information clearly refers to data items extracted by the wrapper, i.e. portions of the source HTML document which are associated with some elements of the extraction schema.

Let EI be the set of data items that have been extracted, and RI be the set of data items required to be extracted. Precision and recall are then defined as:

$$P = \frac{|EI \cap RI|}{|EI|} \qquad R = \frac{|EI \cap RI|}{|RI|},$$

where $|EI \cap RI|$ is the number of correctly extracted data items.

[8] `http://www.ansa.it/main/notizie/awnplus/topnews/topnews.html`.

[9] `http://www.ansa.it/main/notizie/regioni/`.

```
<!ELEMENT ansa-homepage (news+)>
<!ELEMENT news (headline, content, cross-heading?)>
<!ELEMENT headline (#PCDATA)>
<!ELEMENT content (#PCDATA)>
<!ELEMENT cross-heading (#PCDATA)>
```

(a) *ANSA* homepage

```
<!ELEMENT ansa-topNews (news+)>
<!ELEMENT news (headline, subheading, (site | date-time), content?)>
<!ELEMENT headline (#PCDATA)>
<!ELEMENT subheading (#PCDATA)>
<!ELEMENT site (#PCDATA)>
<!ELEMENT date-time (#PCDATA)>
<!ELEMENT content (#PCDATA)>
```

(b) *ANSA* top news

```
<!ELEMENT ansa-localNews (news+)>
<!ELEMENT news (date-time, headline, subheading)>
<!ELEMENT headline (#PCDATA)>
<!ELEMENT subheading (#PCDATA)>
<!ELEMENT date-time (#PCDATA)>
```

(c) *ANSA* local news

```
<!ELEMENT movie (title, year, director, genre+, plot-outline,
                user-comments*, user-rating?, cast-overview*)>
<!ELEMENT cast-overview (actor, role)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT director (#PCDATA)>
<!ELEMENT genre (#PCDATA)>
<!ELEMENT plot-outline (#PCDATA)>
<!ELEMENT user-comments (#PCDATA)>
<!ELEMENT user-rating (#PCDATA)>
<!ELEMENT actor (#PCDATA)>
<!ELEMENT role (#PCDATA)>
```

(d) *IMDB* movie page

Fig. 3.13: Extraction schemas associated with the test sites

The answer of any wrapper evaluation can be one of the following contingencies:

- complete result (CR): the wrapper correctly extracts all the desired data;
- partial result (PR): the wrapper correctly extracts only some desired data, but does not extract incorrect data;
- wrong result (WR): the wrapper extracts incorrect data;
- no result (NR): the wrapper does not work anymore.

A CR case corresponds to optimal precision and recall, i.e. all the correct data items that should be extracted from a page are really extracted by the wrapper.

A failure in recall is obtained in a PR case, since some correct data items have not been recognized. On the contrary, the wrapper fails in precision when a WR case occurs, since some undesired data items have been recognized as correct data items. It is worth emphasizing that, from the data extraction viewpoint, lower precision should be considered as more negative than lower recall: indeed, it is preferable to miss some correct data rather than extract incorrect data.

Finally, an NR case means that the wrapper cannot provide any result since it is not anymore able to compute an extraction model for a given source page.

3.5.3 Robustness evaluation

For each test site, we early designed a wrapper with SCRAP, according to the extraction schemas shown in Fig. 3.13. During the period of experimental evaluation, the pages on the selected Web sites underwent several changes in layout and content organization, therefore we planned the wrapper robustness evaluation as follows.

At a first stage, we monitored the behavior of the original wrappers and evaluated them on the newly fetched pages, whose structures could be changed at different degrees. At a second stage, we then performed a task of generalization for those wrappers, using as input all the pages which never caused a PR, WR or NR case during the monitoring period. Finally, we tried to perform new evaluations on the same pages of the first stage using the generalized wrappers. This led us to assess the effect of the generalization task on the wrapper robustness with respect to some kinds of changes occurring in HTML sources.

Specifically, we distinguished two kinds of page changes:

- *minor change (m-change)*, which refers to a marginal, not necessarily noticeable change of the page layout, such as deletion or insertion of a leaf node of the HTML tree. Moreover, a change is considered to be minor if it replaces the content of a leaf node with new child nodes (i.e. formatting tags are inserted instead of plain text). Also, insertion (or deletion) of fake internal HTML elements (e.g. `div` or `span` tags) is considered to be a minor change.
- *major change (M-change)*, which refers to any evident change of (a portion of) the page layout. Typical major changes may involve complete restructuring of an internal HTML node, for instance, the restructuring of an HTML list as an HTML table is considered a major change.

Table 3.3 reports statistics on the tests we conducted during a 6-month period. Specifically, for each HTML source, the table reports the following

Table 3.3: SCRAP robustness: statistics on the test HTML sources during a 6-month period

source	*page type*	*# URLs monitored*	*#fetches*	*#pages fetched*	*#m-changes / #M-changes*	*CR*	*NR*	*PR*	*WR*
ANSA	homepage	1	3/day	540	5 / 1	3 / 0	0 / 0	2 / 1	0 / 0
ANSA	top news	1	2/day	360	3 / 2	2 / 0	0 / 1	1 /0	0 / 1
ANSA	local news	20	20/day	3,600	2 / 1	2 / 0	0 /0	0 / 1	0 / 0
IMDb	movie list	51	51/week	1,224	3 / 1	3 / 0	0 / 0	0 / 1	0 / 0
Amazon	book list	1	25/day	4,500	3 / 2	2 / 0	0 / 1	1 / 0	0 / 1

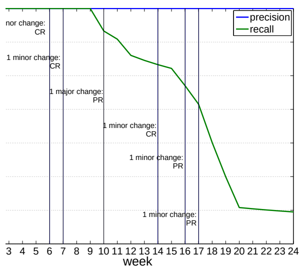
statistics on the pages chosen as input for the wrappers: the type of pages, the number of page URLs monitored, the frequency of page fetching over the monitored URLs, and the total number of pages fetched. For each monitored URL in a source, the number and type of changes occurred and relating details in terms of contingencies of wrapper answers are also reported.

As we can observe in the table, wrapper answers to change contingencies mostly provided correct or partial results, suggesting that the designed wrappers were able to still correctly extract data in most cases. However, when pages underwent major changes in their structure, wrappers returned partial results in some cases, but also wrong results or no results. The latter contingencies occurred in ANSA top news pages and Amazon pages due to evident changes to their layout. In particular, changes involving the location of entire news or books caused usually NR or WR cases, whereas changes involving portions of news or books (e.g. insertion and deletion of `span`, `div` or any other paragraph formatting tags) caused PR cases at worst.

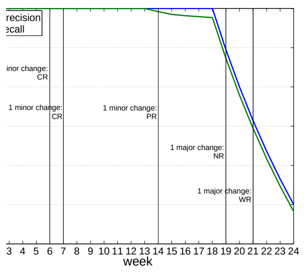
Accuracy results for the first stage of the experimental evaluation are displayed in Fig. 3.14. For each source, we first computed the average precision and recall obtained by the given wrapper over the set of pages fetched on a per-week basis. As extracted data items, we considered the leaf elements of news, movies and books respectively in ANSA, IMDb and Amazon. Each plot in Fig. 3.14 finally displays the average cumulative precision and recall during the 6-month period. It is worth noticing that NR cases resulted in zero precision and recall, and PR cases (resp. WR cases) were reflected by a decrease of recall (resp. precision).

At the end of the 6-month period, we started the second stage of the experimental evaluation in which the early designed wrappers were firstly generalized and then generalized wrappers are tested over the previously fetched HTML pages. Clearly, pages which caused a failure in an original wrapper were not used to learn the generalized wrappers. However, such pages were reconsidered in the new wrapping evaluation.

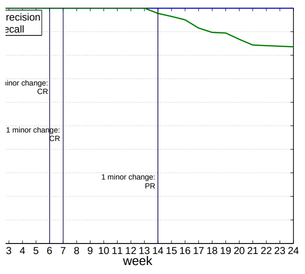
For the sake of brevity, we give here only few details about generalization of wrappers for ANSA local news page (Fig. 3.12 (c)). Figure 3.15 shows encoded in an XML document the extraction rules used for such wrappers,



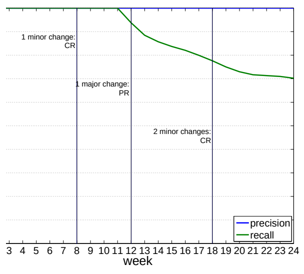
(a) ANSA homepage



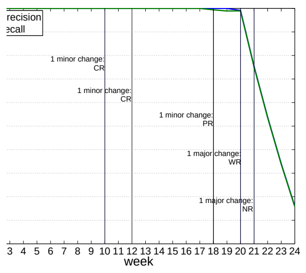
(b) ANSA top news



(c) ANSA local news



(d) IMDb movies



(e) Amazon books

Fig. 3.14: Accuracy results on the test HTML sources during a 6-month period

and Table 3.4 reports a summary of the generalized wrappers. As we can see in the table, rule generalization led to much simpler XPath expressions in target predicates.

Figure 3.16 shows accuracy results obtained by the generalized wrappers over the same input HTML pages used for the early wrappers. It is worth noting that all the minor/major changes causing PR and NR answers in the early wrappers did not cause any failure in the generalized wrappers. As we expected, only WR cases occurred yet: indeed, generalized rules allow for identifying larger portions in a page, thus reducing the probability of incurring

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE extractionRules SYSTEM "extractionRules.dtd">
<extractionRules>
  <rule>
    <input>ansa-localNews</input>
    <output>news</output>
    <filters>
      <filter>
        <targetPredicate>
          <binding-out>tg</binding-out>
          <xpath>/html/body/center/table/tr[2]/td[4]/table[1]/tr/td[1]/
                 table/tr[1]/td[1]/table</xpath>
        </targetPredicate>
        <conditions />
      </filter>
    </filters>
  </rule>
  <rule>
    <input>news</input>
    <output>headline</output>
    <filters>
      <filter>
        <targetPredicate>
          <binding-out>tg</binding-out>
          <xpath>./tr[2]//text()</xpath>
        </targetPredicate>
        <conditions />
      </filter>
      <as> <min>1</min> <max>100</max> </as>
      <rs>maximize</rs>
    </filters>
  </rule>
  <rule>
    <input>news</input>
    <output>subheading</output>
    <filters>
      <filter>
        <targetPredicate>
          <binding-out>tg</binding-out>
          <xpath>./tr[3]//text()</xpath>
        </targetPredicate>
        <conditions />
      </filter>
      <as> <min>1</min> <max>100</max> </as>
      <rs>maximize</rs>
    </filters>
  </rule>
  <rule>
    <input>news</input>
    <output>date-time</output>
    <filters>
      <filter>
        <targetPredicate>
          <binding-out>tg</binding-out>
          <xpath>./tr[1]/td[1]/node()</xpath>
        </targetPredicate>
        <conditions>
          <conditionAfter>
            <binding-in>tg</binding-in>
            <binding-out>$y</binding-out>
            <element>span</element>
          </conditionAfter>
          <conditionBefore>
            <binding-in>tg</binding-in>
            <binding-out>$f</binding-out>
            <element>span</element>
          </conditionBefore>
        </conditions>
      </filter>
      <as> <min>1</min> <max>100</max> </as>
      <rs>maximize</rs>
    </filters>
  </rule>
</extractionRules>
```

Fig. 3.15: Extraction rules of early wrappers for ANSA local news pages

Table 3.4: Generalization of extraction rules of wrappers for ANSA local news pages

rule	*generalized parts*
in: `ansa-localNews` out: `news`	in $targetPredicate$: `<xpath>//td[4]//table//*//*//*[1]//table</xpath>`
in: `news` out: `headline`	in $targetPredicate$: `<xpath>.//tr[2]//text()</xpath>` in as: `<min>1</min> <max>2147483647</max>`
in: `news` out: `subheading`	in $targetPredicate$: `<xpath>.//tr[3]//text()</xpath>` in as: `<min>1</min> <max>2147483647</max>`
in: `news` out: `date-time`	in $targetPredicate$: `<xpath>.//tr[1]//*//node()</xpath>` in as: `<min>1</min> <max>2147483647</max>`

in problems of partial extraction, but they do not provide any additional mechanism besides the schema to filter out isolated wrong results.

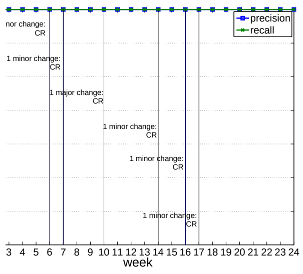
3.5.4 Time performances

Besides wrapping effectiveness, data extraction time is a useful indicator to characterize the behavior of a wrapper. Table 3.5 reports the times elapsed for carrying out the wrapper evaluation steps (i.e. building the extraction tree, computing the preferred extraction model, and generating the extracted XML document) and the wrapper generalization task (i.e. computing the most general wrapper). Specifically, wrapper evaluation times are averaged over the pages of each test site, whereas wrapper generalization times refer to times measured per page on average.[10]

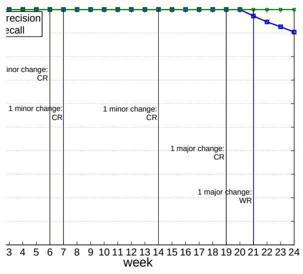
As we can observe in the table, most of time for wrapper evaluation is spent to parse the HTML tree and generate the extraction events. By contrast, computing the preferred extraction model and generating the extracted XML document take a few milliseconds on each test site. This suggests that SCRAP is able to meet even tight time requirements for data extraction.

Wrapper generalization turns out to be a time-consuming task, as we expected, requiring a few minutes per page on average. However, this is a task which should be requested much less frequently than the ordinary wrappings, and it is usually performed off-line.

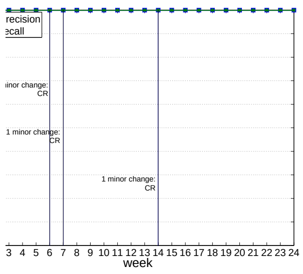
[10] Experiments were conducted on a platform Intel Pentium IV 2.8GHz with 512MB memory and running on Windows XP Pro.



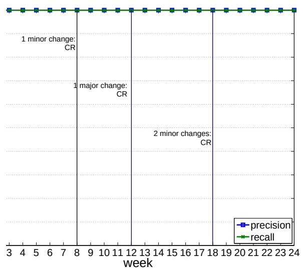
(a) ANSA homepage



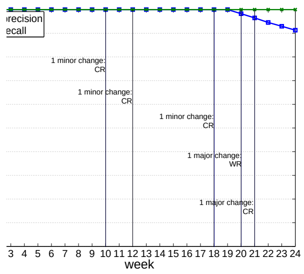
(b) ANSA top news



(c) ANSA local news



(d) IMDb movies



(e) Amazon books

Fig. 3.16: Accuracy results on the test HTML sources after wrapper generalization

Table 3.5: SCRAP time performances (in seconds)

site	***avg. page size (bytes)***	***parsing / buildTree***	***buildElements***	***buildDoc***	***generalization***
ANSA homepage	86,690	0.653 / 0.188	0.093	0.016	252.54
ANSA top news	74,143	0.537 / 0.142	0.087	0.013	202.76
ANSA local news	61,121	0.421 / 0.128	0.068	0.009	184.23
IMDb	51,016	0.305 / 0.11	0.052	0.005	140.35
Amazon	144,491	0.985 / 2.05	0.906	0.031	623.54

4

Retrieving XML data from heterogeneous sources through vague querying

4.1 Introduction

In this chapter we address the problem of retrieving XML data spread across different sources, each adopting its specific schema. For this context, many approaches require queries to be expressed with respect to a global schema which is related to the local ones by means of *mappings* (see, e.g., [11, 76, 113]). Therefore, queries expressed with respect to the global schema are translated to comply with the local schemas. Other approaches do not use a global schema but require schema mappings to be specified between pairs of data sources. Queries are expressed w.r.t. a local schema and then propagated to other sources through proper translations [34, 100]. In general, however, mapping-based approaches limit the autonomy of data sources, that in many cases can be a fundamental requirement.

We consider the scenario where the user is not aware of the local data schemas, and no inter-schema mapping is provided. This way, total autonomy is guaranteed to data sources and the main problem to be dealt with is information retrieval based on the specification of some properties of the objects to be retrieved. Obviously, since the classical way of evaluating a query used in traditional database systems is exact, it is expected to provide poor results in this setting. Several approaches have been proposed for approximate XML querying, that add flexibility to XPath by automatically adapting queries to the available data. Each of these approaches adopts different semantics and different sets of transformations for adapting queries [7, 8, 47, 97, 101].

Query transformation-based approaches proved useful to tackle the problem of query answering over single XML documents in the case that the schema of each document is unknown. The problem gets more complicated when different sources provide information on the same subject from different points of view, i.e., by considering different properties of the same objects. Here, besides isolating users from the possibly complex interaction with data sources, query evaluation mechanisms must be capable of combining "partial" information provided by the sources to obtain results as complete as possible. Example 4.1 shows a possible scenario where the needed information is available but spread across different sources.

Example 4.1. Fig. 4.1 shows a scenario where partial descriptions of books are provided by 3 XML data sources D_1, D_2, and D_3, and each source employs a different schema and focuses on different properties. In particular, D_1 focuses on book authors and titles, D_2 looks at titles and prices, and D_3 has more details about authors' names and surnames. In this scenario,

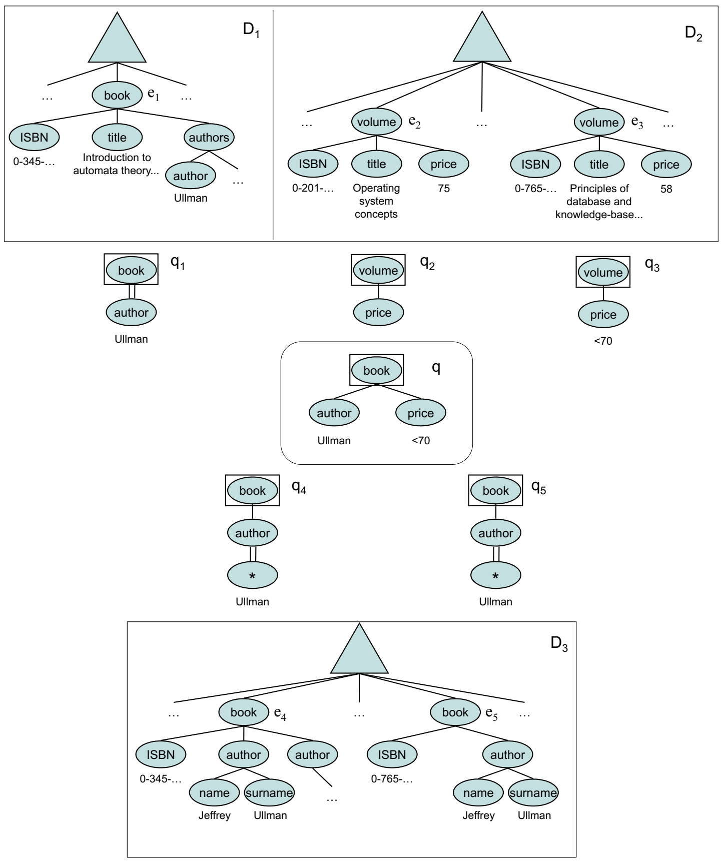


Fig. 4.1: Motivating example

a user interested in finding information about books written by Jeff Ullman and having a price lower than 70 may issue an XPath query q of the form `//book[author='Ullman'][price<70]`. The query is depicted in Fig. 4.1 (in a rounded box) as a *tree pattern* [81] where a box surrounds the output node. The exact evaluation of q over the fragments yields no result, and even adapting q to the available data would retrieve a set of XML elements each of which does not provide enough information by itself. However, the sources provide enough information to correctly characterize the searched books and therefore answer q.

Our aim and strategy are thus different from past work on querying heterogeneous XML data. Our objective is twofold: besides coping with the difference between query and data schemas, we aim at enabling the retrieval of objects that satisfy a query even if their descriptions are spread across different sources. As said before, we aim at providing a querying mechanism that does not require neither semantic schema mappings nor explicit knowledge of the local schema used by each data source. Our proposed technique, whose logical phases are depicted in Fig. 4.2 (for the scenario of Fig. 4.1), is based on the idea of "vaguely" evaluating a query, i.e., relaxing some of the conditions, executing these *transformed* versions, grouping the retrieved *partial answers* (joining), and finally returning those groups which satisfy the query (selection). We call this process *vague query evaluation.*

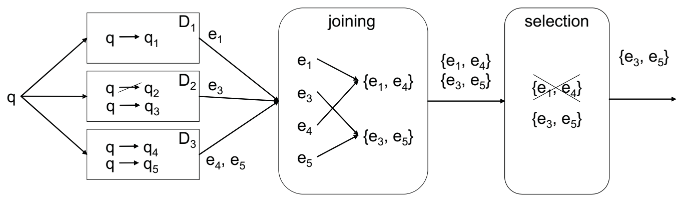


Fig. 4.2: Logical phases of vague query evaluation

The transformation process is performed locally at each data source and is driven by the available data, in the sense that proper transformations are chosen to match the data. Obviously, the transformation process should never produce queries that retrieve elements too different from those required by the user. Thus, *costs* can be associated with basic query transformation operations, and an answer is considered valid only if the overall transformation cost associated with the transformed query that retrieves it is under a certain (*local*) threshold. In the example, it is assumed that the transformation cost of query q_2 is above the threshold, thus q_2 is not actually executed on source D_2.

After evaluating transformed queries, retrieved elements that provide partial information about the same real-world objects are grouped. As it will be clearer in the following, this process is based on the evaluation of semantic similarity among XML elements. In the example, elements e_1 and e_4 are grouped, as well as elements e_3 and e_5.

Finally, the correspondence of the grouped elements with the original query is assessed, by comparing their "overall" transformation cost with a *global* threshold. In the example, it is assumed that the overall cost of $\{e_1, e_4\}$ exceeds the global threshold (as neither e_1 nor e_4 contain information about the book price), thus $\{e_1, e_4\}$ is not part of the final query result.

This evaluation mechanism allows us to retrieve the desired information even when disseminated over several sources. Indeed, by distinguishing two levels at which to apply query relaxation, we are able to collect, from each source, pieces of information that would not completely satisfy the query by themselves, but that can be subsequently completed with information coming from other sources. Therefore, it is reasonable to assume that the original query is allowed to be more deeply modified when it is evaluated locally at each source, although after joining the (partial) answers coming from each source, the elements obtained this way should better satisfy the conditions specified by users. Hence, the local threshold is assumed to be greater than or equal to the global one.

We investigate the problem of retrieving XML data from multiple heterogeneous data sources that do not share a global schema and do not specify schema mappings. Indeed, the main contribution of this proposal is the definition of an approximate querying approach based on XPath, which extends previous work on approximate tree patterns [7, 97] for querying multiple heterogeneous XML data sources.

Specifically, we propose:

- a new technique to combine partial results coming from different XML data sources, which uses approximate queries to check whether two XML elements coming from different sources refer to the same object;
- a complete algorithm for computing query answers on multiple heterogeneous XML data sources, which works in polynomial time w.r.t. the size of the data provided by each source, and in exponential time w.r.t. the number of sources;
- an incomplete algorithm for computing query answers which works in polynomial time w.r.t. both the size of the data and the number of sources. The completeness of this algorithm is proved in restricted cases.

Furthermore, we characterize the complexity of the problem of answering queries on multiple heterogeneous XML data sources and we present an experimental validation in a medical application scenario. Specifically, the proposed algorithms have been implemented in a P2P context, and queries have been done against sources containing clinical and diagnostical data.

4.2 Related Work

A large number of approaches for the problem of querying heterogeneous data sources focus on the differences between the schemas on which data are based, providing some form of mediated schema [11, 14, 15, 24, 48, 49, 55, 56, 62, 66, 76, 94] or some kind of mappings between pairs of sources [18, 29, 43, 52, 100]. Other approaches use specific techniques for directly translating queries w.r.t. source data [26, 75, 86]. In the following, we briefly describe all these solutions, both for relational data and for semistructured data. Finally, we discuss previous works about approximate XML query evaluation [7, 8, 28, 31, 47, 63, 64, 97, 101], consisting in transforming a given query in order to match data contained in an XML document.

4.2.1 Querying heterogeneous relational data

As far as the usage of a mediated schema is concerned, it is worth to note that the mediated schema is virtual in the sense that it is used for posing queries, but not for storing data. Mappings are established between the mediated schema and the relations at the data sources, forming a two-tier architecture in which queries are posed over the mediated schema and evaluated over the underlying source relations. Two main formalisms have been proposed for schema mediation on relational data. In the first, called global-as-view (GAV)[14, 15, 49, 55], the relations in the mediated schema are defined as views over the relations in the sources. In the second, called local-as-view (LAV) [48, 56, 62], the relations in the sources are specified as views over the mediated schema.

Example 4.2. Consider the following two sources storing relational data about movies:

$$s_1 : r_1(Title, Year, Director),\ s_2 : r_2(Title, Critique)$$

where s_1 stores data about directors of movies produced from 1960 on and s_2 contains information about titles and reviews of movies produced from 1990 on. Suppose that the mediated schema contains the following relations:

$$movie(Title, Year, Director),\ european(Director),\ review(Title, Critique)$$

A GAV mapping is:

$$movie(T, Y, D) : -r_1(T, Y, D);\ european(D) : -r_1(T, Y, D);\ review(T, R) : -r_2(T, R)$$

For providing a LAV mapping, we need to formulate relations in the sources as views on the mediated schema, then we obtain:

$$r_1(T, Y, D) : -movie(T, Y, D), european(D), Y >= 1960$$

$$r_2(T, R) : -movie(T, Y, D), review(T, R), Y >= 1990$$

Suppose that a user interested in selecting titles and reviews of movies produced in 1998 formulates the following query against the mediated schema:

$$mr(T, R) : -movie(T, 1998, D), review(T, R)$$

Using the GAV mapping, the query is easily formulated in terms of sources:

$$mr(T, R) : -r_1(T, 1998, D), r_2(T, R)$$

Using the LAV mapping, we can obtain the same formulation, but we need some kind of mechanism to express atoms of the global schema in terms of atoms of the sources.

The fundamental difference between the two formalisms is that GAV specifies how to extract tuples for the mediated schema relations from the sources, and hence query answering amounts to view unfolding. In contrast, LAV is sourcecentric, describing the contents of the data sources. Query answering requires algorithms for answering queries using views, but in exchange LAV provides greater extensibility: the addition of new sources implies no change of the mediated schema.

A mediator system for data integration that follows the GAV approach is proposed in [15]. The result of the integration process is a global schema obtained in a semi-automatically way, which provides a reconciled, integrated and virtual view of the underlying sources. The system is formed by tree main components: a common data model to describe source schemas for integration purposes; some wrappers, placed over each source, for translating metadata descriptions of the sources into the common representation, reformulating a global query expressed in the global query language into queries expressed in the sources languages and exporting query results; a mediator, which is composed of two modules: the SI-Designer and the Query Manager (QM). The SI-Designer module processes and integrates descriptions received from wrappers to derive the integrated representation of the information sources. The QM module performs query processing and optimization. The QM generates queries to be sent to wrappers starting from each query posed by the user on the global schema. QM automatically generates the translation of the query into a corresponding set of sub-queries for the sources and synthesize a unified global answer for the user.

SEWASIE system [14] is a multi-level agent-based architecture for querying heterogeneous data sources integrated by means of ontologies. In this system, each agent provides a the set of services to other agents, and there is a set of actions that the agent can invoke in response to a service. There are two main types of agents in the SEWASIE architecture: brokering agents (BAs) and query agents (QAs). BAs are the nodes responsible for maintaining a view of the knowledge handled by the network. This view is maintained in ontology mappings, that are composed by the information on the specific content of the SEWASIE Information Nodes (SINodes) which are under the direct control of the BA, and also by the information on the content of other BAs. QAs are the carriers of the user query from the user interface to the SINodes, and have the task of solving a query by interacting with the BAs network. Once a BA is contacted, it informs the QA a) which SINodes under its control contain relevant information for the query, and b) which other BAs may be further contacted. Therefore, the QA translates the query according to the ontology mappings of the BA, and directly asks the SINodes for collecting partial results. Also, it decides whether to continue the search with the other BAs. Once this process is finished, all partial results are integrated into a final answer for the user. SINodes are mediator-based systems, each including a global view of the overall information managed within. The managed information sources are heterogeneous collections of structured, semi-structured or unstructured data, e.g. relational databases, XML or HTML documents. SINodes are accessed by QAs in order to obtain data, and also by the managing BAs in order to build the ontology mappings. In order to create and maintain a global view of its information sources, SINodes require an ontology builder. This component performs in a semi-automatic way the enrichment process to create the SINode ontology. In turn, this SINode ontology is also integrated with other similar components into the BAs ontology mappings. In other words, each SINode combines the data residing at different sources, and provide the external user with a global schema representing a unified view of these data. Mappings between the sources and the global schema are obtained following the GAV approach.

In the Information Manifold [56], the user poses queries in terms of a global schema whose mappings with data sources are obtained using LAV approach. The system uses source descriptions to prune efficiently the set of information sources for a given query and to generate executable query plans. Authors present a practical mechanism to describe declaratively the contents and query capabilities of information sources. In particular, the contents of the sources are described as queries over a set of relations and classes. An algorithm that uses the source descriptions to create query plans that can access several information sources to answer a query is presented. The algorithm prunes the sources that are accessed to answer the query, and considers the capabilities of the different sources.

In [62] Tukwila data integration system is presented. A Tukwila user poses queries in terms of a mediated relational schema, where data sources are related by means of LAV mappings. A query reformulator converts the user's query into a union of conjunctive queries referring to the data source schema and a query optimizer takes a query from the reformulator and uses information from a source catalog to produce query execution plans for the execution engine. The optimizer does not always create a complete execution plan for the query. If essential requirements are missing or uncertain, the optimizer may generate a partial plan, deferring subsequent planning until sources have been contacted and critical metadata obtained.

Approaches that do not rely on a mediated schema are also proposed [1, 43, 52, 86].

Specifically, coDB [43] is a system for querying relational databases in P2P networks. Data sources, possibly having different schemas, are interconnected by means of coordination rules. Each node can be queried in its schema for data, which the node can fetch from its neighbors (acquaintances), if a coordination rule is involved. A global update in a P2P database network is a process of updating nodes databases using all definitions of coordination rules they maintain. coDB system only implements queries by first doing a global update and then by answering the query locally at the node at which the query itself was posed. A global update is started when some node sends to all its acquaintances global update requests, containing definitions of appropriate coordination rules. These acquaintances computes the queries, respond with the query results, and propagate the global update to their acquaintances, and so on. The global update request propagation is stopped at some node if that node has no acquaintances to propagate the request, or if that node has already received this request message.

PeerDB [86]is a P2P-based system for distributed data sharing without mediated schema. For each relation that is stored in a peer, associated meta-data (schema, keywords, etc) are stored in a Local Dictionary. Meta-data are essentially keywords provided by the users upon creation of the table, and serve as a kind of synonymous names. When a certain name for an attribute or a relation is asked in a query, each peer uses these keywords for trying to match the specified name with one of the attribute or relation names stored in its repository. There is also an Export Dictionary that reflects the meta-data of objects that are sharable to other nodes. Thus, only objects that are exported can be accessed by other nodes in the network. PeerDB adopts mobile agents to assist in query processing in the following way. When a user issues a query (SQL-like selection query), a master agent will be created to oversee the evaluation of the query. The agent parses the query to extract the list of relations and attributes names. The relation matching strategy is applied on the local dictionary. Promising relations can then be returned to the user immediately. At the same time, the master agent will clone relation matching agents and dispatch them to all neighbors of the node. The master agent will wait for the answers (relations schema) from remote nodes. Upon receiving any answers, they will be returned to the users for selection. For each relation selected by the user, the master agent will clone a data retrieval agent for that relation. The latter agent reformulates the query so that it matches the relation name and attributes at the target node and the answers will be returned to the user peer.

4.2.2 Querying heterogeneous semistructured data

Several approaches addressing the problem of querying heterogeneous XML data have been recently proposed [11, 18, 19, 24, 29, 34, 66, 75, 76, 87, 92, 94, 96, 100, 102, 113]. The classical LAV and GAV paradigms or some kinds of mapping between pairs of sources or some other techniques for matching involved schemas have been adopted.

[66] describes SQPeer middleware for routing and planning declarative queries in peer RDF/S bases by exploiting the schema of peers. Queries are formulated according to RDF/S schema known to each source; the *active schema* indicates the parts of the schemas populated by a peer. For each

query, the relative query pattern graph is extracted from the path expressions in the FROM clause. Each pattern in the graph is compared with all known active-schemas and, if an active-schema is subsumed by the selected pattern, the owner of the active-schema is contacted to answer it. A routing algorithm takes as input a query pattern and returns an annotated query pattern containing information about the peers that can actually answer it. A lookup service (i.e., function lookup), which strongly depends on the underlying P2P topology, is employed to find peer views relevant to the input pattern. A query/view subsumption algorithm is employed to determine whether a query can be answered by a peer view. Query planning in SQPeer is responsible for generating a distributed query plan according to the localization information returned by the routing algorithm. Peers are contacted based on the distributed query plan and results are collected.

[96] describes a zero-administration p2p system for sharing and querying XML data (XPeer). The system allows users to share XML data and to pose XQuery FLWR queries against them without any significant human intervention. The system, based on a hybrid p2p architecture, self-organizes its superpeer network, and allows for arbitrary changes in the network topology. Peers provide their superpeers with a description of their XML data in the form of a compact structure representing all the paths in the data. Super-peers are organized hierarchically; each super-peer stores a list of children schemas, and the union of the schemas in the list. When a user submits a query to a peer, the latter sends the query to its super-peer, that identifies relevant data sources using the schema list, and forwards the query to its own superpeer. Once the root of the hierarchy is reached by the propagated query, all peers containing the required data are discovered. After this, the issuing peer sends the query to the relevant peers and joins partial results. It is worth to note that no modifications of the original query are performed, i.e. peers are judged as relevant for a query only if their data are exactly compatible with those requested.

In [100] authors give an overview of Piazza, a peer data management system for XML data. The Piazza project focuses on the use of schemata, and, in particular, on the definition of schema integration and mapping techniques for p2p systems. The architecture of Piazza is basically a hierarchical p2p architecture, where peers are fully autonomous. Each peer stores the XMLSchema of its data and mappings between this schema and the ones of its neighbors; the mappings are expressed in XQuery. Query processing relies on the propagation of the reformulated query to the neighbors, according to the mappings. If the propagated query reaches a source containing relevant data, the query is evaluated there and the results are appended to the query result. Techniques for pruning paths in the reformulation process and for minimizing the reformulated query are also applied.

In [75] authors propose a new solution for approximate query evaluation in heterogeneous web document bases of which source schemas are available and are written using XMLSchema. In a preliminary schema matching process the similarities between the involved schemas are automatically identified and, thus, a query written on a source schema is automatically rewritten in order to be compatible with the other useful XML documents. This approach has been implemented in *XML* S^3MART system. This system supports a proper phase wherein each term used in document schema is disambiguated using WordNet [104], that is its meaning is made explicit as it will be used for the identification of the semantical similarities between the elements and attributes of the schemas. Term disambiguation is implemented by a semi-automatic operation where the operator is required to annotate each term used in each XML schema with the best candidate among the WordNet terms and, then, to select one of its synonym sets. The matching computation phase performs the actual matching operation between the annotated schemas made available by the previous step. For each pair of schemas, the best matchings between the attributes and the elements of the two schemas are identified by considering both the structure of the corresponding trees and the semantics of the involved terms. The involved schemas are first converted into directed labelled graphs following the RDF specifications, where each entity represents an element or attribute of the schema identified by the full path (e.g. /musicStore/location) and each literal represents a particular name (e.g. location) or a primitive type (e.g. xsd:string) which more than one element or attribute of the schema can share. From the RDF graphs of each pair of schemas a pairwise connectivity graph (PCG), involving node pairs, is constructed [77] in which a labelled edge connects two pairs of nodes, one for each RDF graph, if such labelled edge connects the involved nodes in the RDF graphs. Then an initial similarity score is computed for each node pair contained in the PCG using a linguistic approach. The initial similarities, reflecting the semantics of the single node pairs, are refined by an iterative fixpoint calculation as in the similarity flooding algorithm [77], which brings the structural information of the schemas in the computation. The intuition behind this computation is that two nodes belonging to two distinct schemes are the more similar the more their adjacent nodes are similar. In other words, the similarity of two elements propagates to their respective adjacent nodes. The fixpoint computation is iterated until the similarities converge or a maximum number of iterations is reached. Finally, a stable marriage filter which produces the best matching between the elements and attributes of the two schemas is applied. By exploiting the best matches provided by the matching computation, a given query, written w.r.t. a source schema, is easily rewritten on the target schemas. Each rewrite is assigned a score, in order to allow the ranking of the results retrieved by the query. Note that matching is total for the query, then no deletions are allowed.

[26] presents an inclusion mapping algorithm (using a compatibility factor) that decides how compatible the schema of the query and the schema of the target XML documents are. If the schemas are not identical but compatible, the generated mapping is then used to translate the query according to the target XML data schema. The query and data schemas are considered

compatible if the compatibility factor of a mapping is above or equal to a user specified or system default threshold. In order to estimate the compatibility between queries and data schemas, a tree similarity measure that takes into account both the semantic similarity between element names and the structural compatibility of the two schema structures is introduced. Specifically, given a query tree and a schema tree, the best match between them is computed trying to match the query tree in a subgraph of the schema tree. Semantic similarity of terms are taken in account by using WordNet [104]. It is worth to note that the inclusion mapping induces a subgraph where each node of the query tree is matched, then no deletions are allowed.

In [24] CXPath, an XPath based language for building queries over a conceptual schema that is an abstraction of several XML sources, is defined. A strategy for defining mapping information related to concepts and relationships of the conceptual schema that is also based on XPath views is presented. CXPath defines concepts and relationships between concepts. Two types of concepts are supported in this model: lexical and non-lexical. A lexical concept models information that has an associated textual content, like $\#PCDATA$ elements and attributes. A non-lexical concept models information that is composed by other information, like elements that have sub-elements. An association relationship is a binary relationship with cardinality constraints. Although CXPath is based on XPath, there several differences between the two languages: 1)in CXPath concept names are used instead of element names (a concept name refers to all instances of that concept in the conceptual base); 2)CXPath has not a root node, the navigation may start at any concept in the conceptual base; 3)in CXPath the navigation operator / means "navigate to the related elements" instead of "navigate to the child elements" and all the other navigation operators of XPath do not appear in CXPath; 4) a qualified navigation operator (relationship name) is introduced in CXPath, as it is necessary the identification of a specific relationship to be navigated when more than one relationship relates two concepts. Queries are formulated against the conceptual schema and translated in XPath queries to be evaluated on the sources by means of mappings. The CXPath to XPath translation applies a rewriting strategy, i.e., each reference to a concept as well as each relationship traversal found in the CXPath expression are replaced by their corresponding mapping information to the considered XML source. In this paper, the problem of query decomposition is not addressed, i.e., queries requiring information that is not completely contained in a source, but spread across multiple sources, are not discussed.

In [76] a methodology for integrating data sources of diverse formats, including relational and XML, under an XML global schema obtained using LAV approach is presented. The query language used for formulating queries on the global schema is XQuery. The execution of an XQuery query consists in a translation of the query into an SQL query on a virtual relational schema, independently from the data sources, and, then, in a rewriting of the SQL query into an other SQL query according to real data sources. The first step only gets the query across the language gap, whereas the second step provides a query executable on data sources using the LAV mappings. Not all features of XQuery can be translated to SQL, thus the query translation can fail is some cases. If the translation step succeeds, a query rewriting algorithm searching for maximally contained rewritings is used to produce a query to be sent to data sources, since it is reasonable to assume that not all required data are available.

To the best of our knowledge, none of the existing proposals for heterogeneous XML querying employs techniques that properly combine query answers obtained by applying query relaxations without schema knowledge to provide the user with answers as complete as possible.

4.2.3 Query Relaxation

Some approaches for the approximate evaluation of XML query propose query languages based on similarity conditions like those used in information retrieval, that is they only consider similarity between labels of node elements or between textual contents [28, 101]. In [101] a similarity operator is proposed for both element content (and also attribute value) comparisons, as well as approximate matching of element names. A score is assigned to each comparison by means of an ontology and the relevance assessments for all elementary conditions in a query are combined into an overall relevance of an XML path, and the result ranking is based on these overall relevance measures. In [28] a similarity operator is introduced for answering queries like "find books and CDs having similar titles". The proposed algorithm rewrites the original query into a series of sub-queries that generate intermediate relational data, and uses relational database techniques to evaluate the similarity operators on this intermediate data, yielding final answers ranked by similarity.

Other approaches take also into account structural approximation in the query evaluation [8, 31, 47, 63, 64]. [47] propose an XML query language, named XIRQL, that is based on XPath and incorporates the information retrieval approach of vagueness and imprecision for XML retrieval. In addition to features concerning similarity between textual content and node labels, XIRQL provides structural relaxations, but only concerning the transformation of attributes in elements and viceversa and the generalization of a parent-child relationship in an ancestor-descendant one. In [63] semi-flexible and flexible matching for a graph query on an XML document are defined. In the first, parent-child relationships are allowed to be relaxed in ancestor-descendant ones, and in the second it is also allowed the reversal of the parent-child relationship, but in both semantics no deletions are allowed, in the sense that all nodes have to be matched. In [64] XML queries with incomplete answers are investigated, allowing that not all the nodes of the query are matched on the data, but no other relaxations are provided.

The most similar approaches to the approximate query evaluation proposed in this thesis are [7, 97], that concern the tree pattern relaxation and use of transformation costs. In both approaches, a set of transformations applicable to a tree pattern query is defined and a strategy for evaluating the relaxed versions of the query is proposed.

In [7], allowed transformations are node renaming, leaf deletion, edge relaxation and subtree promotion. The last kind of relaxation allows a query subtree to be promoted so that the subtree is directly connected to its former grandparent by an ancestor-descendant edge. For each node and each edge of the query, user specifies two weights, an *exact* weight (denoted as *ew*) and a *relaxed* weight, where the first one is greater than the second one. The former is the score associated with an exact match of the node or the edge, whereas the latter is the score associated when a relaxation is applied to the node or the edge.

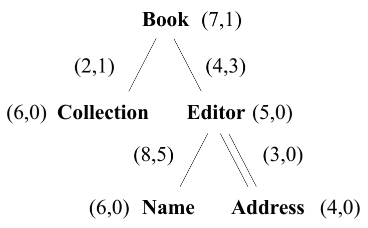


Fig. 4.3: Example of weighted tree pattern query

The score associated with a query is obtained by summing relaxed weights for nodes and edges on which some relaxations are applied and exact weights for non-relaxed nodes and edges. This means that a relaxed version of a query has an associated score that is lower than the score of the original one.

Example 4.3. For example, in the weighted tree pattern query of Fig.4.3, Book can be generalized to Document, but in this case the associated score is 1 rather than 7. The edge between Book node and Editor node can be generalized, allowing for books that have a descendant editor (but not a child editor) to be returned, with a score of 3. The leaf node Address can be promoted, allowing for books that have a descendant address to be returned, even if the address is not a descendant of the editor node. In this case, the score associated with the match is 0. The score of exact matches of the weighted query tree pattern in Fig.4.3 is equal to the sum of the exact weights of its nodes and edges, i.e., 45. If Book is generalized to Document, the score of an approximate answer that is a document (but not a book) is the sum of the relaxed weight of Book and the exact weights of the other nodes and edges in the weighted query, i.e., 39.

The key idea underlying this approach is to encode all possible query relaxations in a single query evaluation plan which is evaluated and only relevant

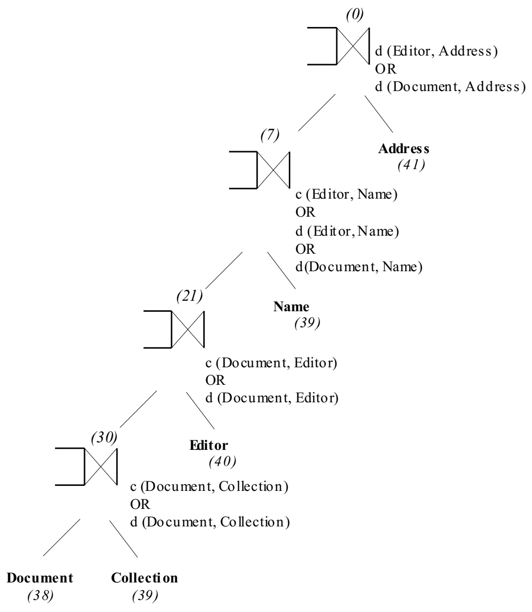


Fig. 4.4: Example of query plan

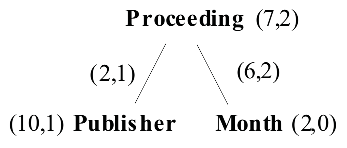
answers are selected. Fig.4.4 shows a translation of the query of Fig.4.3 in a left-deep join evaluation plan. An answer to a query is a tuple containing a value (possibly empty) for every leaf node in the query evaluation plan. Two predicates are used, $c(n_1, n_2)$ and $d(n_1, n_2)$, where the former checks for a parent-child relationship between n_1 and n_2 and the latter checks for an ancestor-descendant one.

Three algorithms are proposed: (1) *Thres* that takes a weighted query tree pattern and a threshold and computes all approximate answers whose scores are at least as large as the threshold; (2) *OptiThres*, an adaptive optimization to Thres that uses scores of intermediate results to dynamically "undo" relaxations encoded in the evaluation plan without compromising the set of answers returned; (3) *TopK*, that takes a weighted query tree pattern and finds the top-k approximate answers.

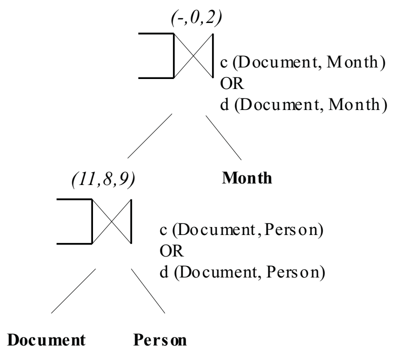
Algorithm Thres associates each node in the join evaluation plan with its maximal weight, denoted as maxW, defined as the largest value by which the score of an intermediate answer computed for that node can grow. Consider, for example, the evaluation plan in Fig.4.4. The maxW of the Document node is 38. This number is obtained by computing the sum of the exact weights of all nodes and edges of the query tree pattern, excluding the Document node itself. Similarly, maxW of the join node with Editor as its right child is 21. This is obtained by computing the sum of the exact weights of all nodes and edges of the query tree pattern, excluding those that have been evaluated as part of the join plan of the subtree rooted at that join node. By definition, maxW of the last join node, the root of the evaluation plan, is 0 since the query evaluation plan is executed in a bottomup fashion. At each node, intermediate results, along with their scores, are computed. If the sum of the score of an intermediate result and maxW at the node does not meet the threshold, this intermediate result is eliminated. The algorithms we use for inner joins and left outer joins are based on the structural join algorithms of [5].

The key idea behind OptiThres is that it is possible to predict, during evaluation of the join plan, if a subsequent relaxation produces additional matches that will not meet the threshold. In this case, this relaxation in the evaluation plan is undone. Undoing this relaxation (e.g., converting a left outer join back to an inner join, or reverting to the original node type) improves efficiency of evaluation since fewer conditions need to be tested and fewer intermediate results are computed during the evaluation. OptiThres uses three weights at each join node of the query evaluation plan. The first weight, relaxNode, is defined as the largest value by which the score of an intermediate result computed for the left child of the join node can grow if it joins with a relaxed match to the right child of the join node. This is used to decide if the node generalization of the right child of the join node should be undone. The second weight, relaxJoin, is defined as the largest value by which the score of an intermediate result computed for the left child of the join node can grow if it cannot join with any match to the right child of the join node. This is used to decide if the join node should remain a left outer join, or should go back to being an inner join. The third weight, relaxPred, is defined as the largest value by which the sum of the scores of a pair of intermediate results for the left and right children of the join node can grow if they are joined using a relaxed structural predicate. This is used to decide if the edge generalization and subtree promotion should be undone.

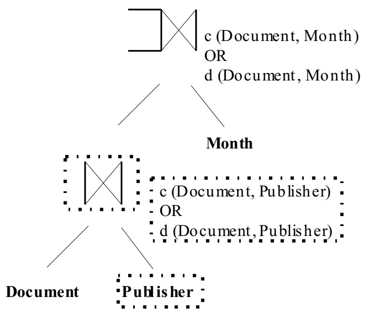
Example 4.4. Consider the weighted query tree pattern in Fig.4.5a. This query looks for all Proceedings that have as children subelements a Publisher and a Month. Exact and relaxed weights are associated with each node and edge in the query tree pattern. Proceeding is relaxed to Document, Publisher is relaxed to Person, the parent-child edges are relaxed to ancestor-descendant ones, and nodes Person and Month are made optional. The threshold is set to 14. First, weights are computed at each evaluation plan node statically. For example, at the first join node in the evaluation plan, relaxNode = 11 (ew(Month) + ew((Proceeding, Month)) + ew((Proceeding, Publisher)) + rw(Publisher)), relaxJoin = 8 (ew(Month) + ew((Proceeding, Month))), and relaxPred = 9 (ew(Month) + ew((Proceeding, Month)) + rw((Proceeding, Publisher))). Next, Algorithm OptiThres evaluates the annotated query evaluation plan in Fig.4.5b. Document is evaluated first. Assume that the maximal score in the list of answers we get is 2, i.e., there are no Proceeding's in the database. At the next join, relaxNode = 11, relaxJoin = 8, and relaxPred = 9. The sum relaxNode +2 = 13 is smaller than the threshold. In this case,



(a) Query



(b) Query evaluation plan



(c) Actual evaluation

Fig. 4.5: An OptiThres Example

OptiThres decides to unrelax Person to Publisher, and the plan is modified suitably. Next, Publisher is evaluated, and let the maximal score in the result list be 10 (i.e., exact matches were obtained). The sum relaxJoin +2 = 10 is also smaller than the threshold, and OptiThres decides to unrelax the left outer join to an inner join, since we cannot "afford to lose Publisher". The algorithm then checks whether to retain the descendant structural predicate. Since the sum relaxPred +2 + 10 = 21 is larger than the threshold, OptiThres decides to retain the relaxed structural join predicate d(Document, Publisher). During the evaluation of the first join, join results are pruned using maxW, as in Algorithm Thres. Assume that the maximal score of answers in the first join result is 14 (10+2+2). OptiThres then uses the weights at the second join node to determine whether any other relaxations need to be undone. Note that Month node has not been generalized, and this is reflected in the fact that relaxNode at the second join is not specified. Next, Month is evaluated, and matches have a score of 2. The sum relaxJoin +14 = 14, which meets the threshold. So the outer join is not unrelaxed. Similarly, relaxPred +14 + 2 = 18, which meets the threshold. So the join predicate is not unrelaxed. Finally, the second join is evaluated, and join results are pruned using maxW, as in Algorithm Thres. The query plan that has been effectively computed is shown in Fig.4.5c, where the dynamically modified portions of the evaluation plan are highlighted.

TopK is similar to Thres, unless it does not rely on a fixed threshold but on a dynamically computed one. At each step, intermediate results are ranked by their score and the score of the kth answer is used as current threshold for pruning purposes.

It is worth to note that the proposed querying mechanism only supports node renamings w.r.t. fixed name hierarchies that must be provided.

[97] proposes a query language that allows node insertion, deletion, and renaming. Costs are associated with labels and only node renamings are allowed that are chosen in advance and completely specified by the user, independently of the available data.

Two polynomial-time algorithms that find the best n answers to the query are presented: The first algorithm finds all approximate results, sorts them by increasing cost, and prunes the result list after the nth entry. The second algorithm is an extension of the first one. It uses the schema of the database to estimate the best k transformed queries, sorts them by cost, and executes them against the database to find the best n results. The evaluation of a query is based on an expanded representation of the query that implicitly includes all so-called semi-transformed queries. All embedding images of the semi-transformed query are computed using a bottom-up algorithm based on a list algebra.

4.3 Vague Queries on Multiple Heterogeneous XML Data Sources

In this section we define a query language, named *VXQL*, whose flexibility enables users to find the information they are interested in, even when such information is disseminated in different XML data sources. The language is essentially an extension of XPath; for the sake of conciseness, we do not consider attributes and use the graphical formalism of tree patterns to represent XPath expressions in the examples.

VXQL supports query relaxations similar to those proposed in [7, 97] and introduces transformations concerning textual predicates. Differently from these approaches, that adopt tree pattern formalisms to define query relaxations, our language is directly based on XPath. Let s be an XPath step of the form axis::$l[f]$. The language supports the following *basic transformations* applicable to the axis and to the *node test* l:

- *node renaming*: if $l \neq$'$*$', l is replaced with a different label l';
- *node deletion*: s is replaced with descendant-or-self::$*[f]$;
- *axis relaxation*: if axis is child, s is replaced with descendant::$l[f]$.

Moreover, transformations are also applicable to XPath predicates, which appear in the leaf nodes of the corresponding tree pattern (comparison predicates). Let f be an XPath predicate of the form `text()` *op* `'text'`, where *op* is a comparison operator; the language supports the following transformations:

- $*$*-node insertion*: f is replaced by descendant-or-self::$*[f]$;
- *relaxation of equality predicate*: if the comparison operator used in f is $=$, f is replaced with `contains(text(),'text')`;
- *predicate deletion*: f is deleted.

Example 4.5. In the scenario of Fig. 4.1 (where deleted nodes are not shown), we can transform q in 5 different ways:

- q_1 is obtained by relaxing the parent-child relationship between the `book` and `author` elements to an ancestor-descendant one, and removing the subtree rooted in the `price` element. This query captures element e_1.
- q_2 and q_3 are obtained by renaming the root element from `book` to `volume` and removing the subtree rooted in the `author` element. Moreover, the textual predicate on the `price` element is removed in q_2. These queries capture elements e_2 and e_3.
- q_4 and q_5 are obtained by removing the subtree rooted in the `price` element and adding a $*$-labeled descendant to the `author` element. These queries capture elements e_4 and e_5.

The costs for node renamings, node deletions, and axis relaxations are associated with query steps. In particular, a renaming cost represents the cost of renaming a node with a label having the maximum *semantic distance* from the original one. This cost is weighted by a semantic distance measure (such as the one provided in [104]) that evaluates the dissimilarity of a label in the query with respect to the labels in the data. The costs for the insertion of $*$-labeled nodes as descendants of leaf nodes, for the relaxation of equality predicates, and for the deletion of predicates are associated with predicates.

Observe that the possibility of specifying the cost of each transformation allows users to give "priority" to some of the conditions expressed in the query. That is, specifying a high cost for a certain transformation t_1 and a low cost for another transformation t_2 means that elements which satisfy t_1 are preferred over those satisfying t_2.

In general, transformation costs are expressed as natural numbers. Nevertheless, since when evaluating a VXQL query the overall costs of its transformed versions are compared with a given threshold, it is possible to define the cost of each transformation with respect to this threshold , and vice-versa. Therefore, we just assume three predefined cost levels, *high* (denoted as h), *medium* (m), and *low* (l).

In some cases, besides associating costs with applicable transformations, it can be useful to avoid that some of the allowed transformations, for instance those removing or modifying important constraints of the original query, are applied together, yielding query answers too different from the requested ones. VXQL features this by allowing users to *mark* the more relevant transformations and specify a maximum number of marked transformations applicable to the query.

Example 4.6. Query q of Fig. 4.1, augmented with transformation costs represented with labels attached to nodes and edges, takes the form of Fig. 4.6, where the output node is surrounded by a solid box and marked transformations are surrounded by dashed boxes.

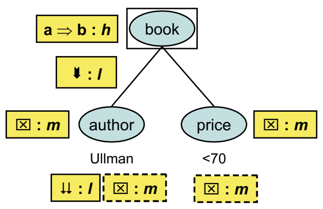


Fig. 4.6: VXQL query

The allowed relaxations specified in the query in Fig. 4.6 are:

1. renaming of the `book` element;
2. transformation of the child axis between `book` and `author` in a descendant one;
3. removal of the `author` and `price` elements;
4. insertion of a $*$-labeled node as a descendant of the `author` element;
5. deletion of the predicates on the content of `author` and `price` elements;

By specifying a low cost for transformations 2 and 4, a medium cost for transformations 3 and 5, and a high cost for transformation 1, the user essentially states that the queries obtained by relaxing the requirement that the text `Ullman` is directly contained inside the `author` element, or the requirement that the `author` element is a child of the returned `book` element, are preferred over the queries that remove `author` or `price` elements or their predicates. Moreover, the lowest preference is given to transformed queries which return elements named differently from "book". Since the deletions of `author` predicate and `price` predicate are marked, if the maximum number of allowed transformations is set to 1 the elements gathered from each document must satisfy either the condition on the author name or the condition on the book price.

VXQL allows a very fine-grained specification of transformation costs, e.g., a different cost can be specified for each transformation. In many practical cases, less-detailed cost specifications are enough for expressing users' needs. Thus, we introduce three kinds of simplified queries:

- *exact* queries, where an infinite cost is assumed for all transformations;
- *uniform-cost* queries, where the same cost is assumed for all transformations, and no transformation is marked;

- *count-based* queries, where the same cost is assumed for all transformations, but some of them are marked and a maximum number of marked transformations applicable is specified.

4.3.1 VXQL Syntax

VXQL is based on the concept of *weighted step*, that is an expression of the form $s[f]$, where s is a pair $(axis, node\ test)$, with $axis$ being child, descendant, or descendant-or-self; f is a predicate; both s and f have associated weights (we call s a *weighted simple step* and f a *weighted predicate*).[1] A *weighted XPath expression* is an expression of the form $ws_1/\cdots/ws_n$ where $ws_1, \cdots, ws_n$ are weighted steps.

As discussed above, VXQL admits 6 different kinds of transformation: axis relaxation (denoted as *AR*), node deletion (*ND*), node renaming (*NR*), $*$-node insertion (*SD*), relaxation of equality predicates (*EQ*), and predicate deletion (*PD*). The syntax of a *VXQL expression* is given by extending the Step nonterminal of the XPath grammar with the possibility to express the cost of each transformation, obtaining the nonterminal WeightedStep defined as follows:

```
WeightedStep        := WeightedSimpleStep (“[”Predicate“]” PredCosts?)*
WeightedSimpleStep  := AxisSpecifier ::NodeTest RelCosts?
RelCosts            := “{” (ARCost “*”? “,”)? (NDCost “*”? “,” )?
                       (NRCost “*”? )? “}” | “{*}”
PredCosts           := “{” (SDCost “*”? “,”)? (EQCost “*”? “,”)?
                       (PDCost “*”? )? “}” | “{*}”
ARCost              := “AR[” Cost “]”
NDCost              := “ND[” Cost “]”
NRCost              := “NR[” Cost “]”
SDCost              := “SD[” Cost “]”
EQCost              := “ER[” Cost “]”
PDCost              := “PD[” Cost “]”
```

where Cost is a natural number or one of the predefined constants h, m, and l. Moreover, PredCosts can be specified only for “comparison” predicates of the form text() op 'text' where op is a comparison operator. An infinite cost is assumed if a transformation cost is not specified. The symbol “*” after a transformation cost indicates that the transformation is marked. Moreover, using symbol “*” instead of a list of cost specifications means that all transformations are allowed (with all costs set to m) and marked.

A weighted XPath expression $ws_1/\cdots/ws_n$ is said to be *simple* iff $ws_1, \cdots, ws_{n-1}$ are weighted simple steps. A VXQL expression is a weighted simple XPath expression.

[1] Note that assuming the presence of a single predicate does not limit the language expressiveness because conjunctions and disjunctions of predicates can be rewritten as a single predicate in XPath.

Example 4.7. In its textual form, the VXQL expression corresponding to the query of Fig. 4.6 is expressed as

```
//book{NR[h]}
      [author{AR[l],ND[m]}[text()='Ullman']{SD[l],PD[m]*}]
      [price{ND[m]}[text()<70]{PD[m]*}]
```

Moreover, one of the simplified query types introduced before can be used.

Example 4.8. If in the query above we only want that at least one between the conditions on author name and book price must be satisfied, the VXQL expression can take the following form:

```
//book[author[text()='Ullman']{*}][price[text()<70]{*}]
```

requiring that only one among the marked transformations can be applied.

4.3.2 Relaxing VXQL Expressions

In this section we define the *relaxed expressions* obtainable from a VXQL expression and the transformation cost corresponding to the application of relaxations. The transformation cost incurred when relaxing a VXQL expression to capture an XML element e (that we will informally call *cost of* e) is a quality measure of how much e satisfies the original VXQL expression.

The application of one or more basic transformations to a weighted step ws yields a standard XPath step rs, called *relaxed step*. The cost of applying basic transformations to a weighted step ws (except for renaming) is the cost specified in ws for that transformation. For node renaming, if the renaming cost specified in ws is c, the transformation cost for obtaining rs applying node renaming is given by $semDist(l, l') \cdot c$, where *semDist* is a function evaluating the semantic dissimilarity between label l used as node test in ws and label l' used as node test in rs. Function *semDist* either returns a value in $[0..1]$, if l is considered “similar enough” to l', or ∞, if l is considered “too different” from l'.

The relaxed step rs can in general be obtained from ws by applying different sequences of basic transformations, each sequence entailing a cost equal to the sum of the costs of the single transformations. We define function $cost(ws, rs)$ as the minimum among the costs entailed by all possible sequences of transformations that yield rs from ws. The same applies to the number of marked transformations, denoted as $mrk(ws, rs)$.

Given a VXQL expression $xp = ws_1/\cdots/ws_n$, a *relaxed expression* rxp is obtained from xp (denoted as $xp \rightsquigarrow rxp$) by relaxing each weighted step ws_i in xp. rxp is therefore a standard XPath expression of the form $rs_1/\ldots/rs_n$ where $rs_i,\ i \in [1..n]$, is a relaxed step obtained from ws_i. The transformation cost of rxp is defined as $cost(xp, rxp) = \sum_{i\in[1..n]} cost(ws_i, rs_i)$. Analogously, the number of marked transformations applied to xp in order to obtain rxp is defined as $mrk(xp, rxp) = \sum_{i\in[1..n]} mrk(ws_i, rs_i)$.

The maximum cost of xp, given a maximum number of marked transformations applicable κ, is defined as follows. We denote as $\mathcal{T}(xp, \kappa)$ a function returning the set of relaxed expressions obtainable from xp (with a transformation cost less than ∞) by applying a number of marked transformations less than or equal to κ, i.e., $\mathcal{T}(xp, \kappa) = \{rxp \,|\, xp \rightsquigarrow rxp, mrk(xp, rxp) \leq \kappa, cost(xp, rxp) \leq \infty\}$. The maximum cost of xp given κ is $\eta_\kappa(xp) = max_{rxp \in \mathcal{T}(xp,\kappa)} cost(xp, rxp)$.

Example 4.9. Consider the VXQL expression xp_q corresponding to query q of Fig. 4.6 and the relaxed expressions $xp_1, \ldots, xp_5$, corresponding to queries $q_1, \ldots, q_5$ of Fig. 4.1. The transformation costs entailed are: $cost(xp_q, xp_1) = l + 2 \cdot m$; $cost(xp_q, xp_2) = 3 \cdot m + h$; $cost(xp_q, xp_3) = 2 \cdot m + h$; $cost(xp_q, xp_4) = cost(xp_q, xp_5) = l + 2 \cdot m$. The marked transformations applied are: $mark(xp_q, xp_1) = mark(xp_q, xp_3) = mark(xp_q, xp_4) = mark(xp_q, xp_5) = 1$; $mark(xp_q, xp_2) = 2$. Assuming $\kappa = 1$, the maximum cost of xp_q is $\eta_\kappa(xp_q) = 2 \cdot l + 3 \cdot m + h$.

Before defining the answer to a vague expression, we introduce some notation. We denote as $rxp(D)$ the set of answers obtained by applying a relaxed expression rxp to an XML document D. Given a VXQL expression xp, a cost threshold τ, a maximum number of marked transformations applicable κ, and an XML document D, we denote as $\mathcal{T}_D(xp, \tau, \kappa)$ a function returning the set of relaxed expressions obtainable from xp to capture elements in D, i.e., $\mathcal{T}_D(xp, \tau, \kappa) = \{rxp \,|\, xp \rightsquigarrow rxp, rxp(D) \neq \emptyset, mrk(xp, rxp) \leq \kappa, cost(xp, rxp) \leq \tau\}$. The application of function $\mathcal{T}$ to a single XML element e, denoted as $\mathcal{T}_D(xp, \tau, \kappa, e)$, returns the set of relaxed expressions obtainable from xp to capture e, i.e., $\mathcal{T}_D(xp, \tau, \kappa, e) = \{rxp \,|\, xp \rightsquigarrow rxp, e \in rxp(D), mrk(xp, rxp) \leq \kappa, cost(xp, rxp) \leq \tau\}$, where D is the XML document containing e.

Definition 4.10. ***(Vague expression answer)*** *Let xp be a VXQL expression, D an XML document, τ a transformation threshold and κ a marked transformation threshold. The* vague expression answer *of xp over D with respect to τ and κ, denoted as $xp_{\tau,\kappa}(D)$, is defined as*

$$xp_{\tau,\kappa}(D) = \bigcup_{rxp \in \mathcal{T}_D(xp,\tau,\kappa)} rxp(D)$$

4.3.3 VXQL Queries

As previously explained, the evaluation of a VXQL expression xp on a set of (heterogeneous) XML data sources requires to evaluate xp on each source, then to combine the (partial) answers obtained from this evaluation, and finally to filter out combined answers which do not satisfy xp. To capture this behavior, a *VXQL query* comprises, besides a VXQL expression, local and global cost thresholds (τ_l and τ_g) and the maximum number of marked transformations applicable locally and globally (κ_l and κ_g). Therefore, the formal definition of a VXQL query is the following.

Definition 4.11. ***(VXQL query)*** *A VXQL query is a tuple $\langle xp, \tau_g, \tau_l, \kappa_g, \kappa_l \rangle$, where xp is a VXQL expression, $\tau_g, \tau_l \in \mathbb{R}^+ \cup \{0\}$ and $\kappa_g, \kappa_l \in \mathbb{N}$.*

In order to facilitate choosing global and local cost thresholds, denoted as τ_g and τ_l, VXQL allows users to express them as percentages of the maximum allowed cost of a transformed query. Moreover, we consider three predefined (global) cost thresholds, *low*, *medium*, and *high*, corresponding to the 10%, 30%, and 50% of the maximum cost of a transformed query, respectively. The user may also avoid specifying the local transformation threshold, as it can be set by default by increasing the global value of a 25%. Therefore, κ_g can be selected by the user, and κ_l may be set as the smallest integer such that $\kappa_l \geq \kappa_g \cdot 1.25$.

The elements retrieved from different data sources can have a certain degree of dissimilarity even if they represent the same object. Therefore, when grouping them, we associate each group with a value representing the "overall dissimilarity" of the elements contained in it. The meaning and importance of this dissimilarity value will be made clearer in the following (see, in particular, Section 4.3.4). A *vague element* is thus a pair $v = \langle E_v, \gamma_v \rangle$, where E_v is a set of XML elements and γ_v is a dissimilarity value. Given an XML element e and a vague element $v = \langle E_v, \gamma_v \rangle$, we say that e belongs to v ($e \in v$) iff $e \in E_v$.

Definition 4.12. ***(Vague query answer on a single data source)*** *Given an XML document D and a VXQL query $q = \langle xp, \tau_g, \tau_l, \kappa_g, \kappa_l \rangle$, the* vague query answer *of q over D, denoted as $q(D)$ is defined as*

$$q(D) = \{\langle \{e\}, 0 \rangle \mid e \in xp_{\tau_l, \kappa_l}(D)\}.$$

Example 4.13. Consider again query q of Fig. 4.6 and the scenario depicted in Fig. 4.1, and assume that $\tau_l = 3 \cdot m + h$ and $\kappa_l = 1$. The vague query answers obtained by evaluating q on the data sources are $q(D_1) = \{\langle \{e_1\}, 0 \rangle\}$, $q(D_2) = \{\langle \{e_3\}, 0 \rangle\}$, and $q(D_3) = \{\langle \{e_4\}, 0 \rangle, \langle \{e_5\}, 0 \rangle\}$. Note that element e_2 of Fig. 4.1 is not part of the query answer because the number of marked transformations to be applied to retrieve it would be greater than κ_l.

4.3.4 Vague Join

Before providing the formal definition of the semantics of VXQL, it is necessary to investigate how answers *coming from different sources* are combined together. This problem requires to address the following issues: (i) how to check whether two XML elements (partial answers obtained by vaguely evaluating a VXQL expression on different sources) describe the same object; (ii) how to construct sets of elements which describe the same object.

As regards the first issue, in order to assess the dissimilarity between two XML elements e' and e'', we employ a *dissimilarity function* $\delta(e', e'')$ that measures the dissimilarity *of the objects the two elements describe*. Dissimilarity values range from 0 (e' and e'' definitely describe the same object) to ∞ (they do not refer to the same object). The particular dissimilarity function currently adopted is described in Section 4.4.2.

Although the problem of constructing sets of elements describing the same object could seem a generalization of the problem of checking dissimilarities, it has some peculiarity that makes it different. In particular, it may happen that two elements are actually a description of the same object, but to detect this we need to consider them as part of a set. For instance, consider elements e_1, e_2, e_3 in Figure 4.7.

```
<book>
  <ISBN>47</ISBN>
  <reviews>
    <review score= ''10''>
      ...
    </review>
    <review score= ''7''>
      ...
    </review>
  </reviews>
</book>
```
e_1

```
<book>
  <ISBN>47</ISBN>
  <title>T<title>
  <authors>
    <author>A1</author>
    <author>A2</author>
  </authors>
  <price> 100 </price>
</book>
```
e_2

```
<book>
  <title>T<title>
  <authors>
    <author>A1</author>
    <author>A2</author>
  </authors>
  <references>
    <reference>R1</reference>
    <reference>R2</reference>
  </references>
</book>
```
e_3

Fig. 4.7: Elements describing the same book

e_1, e_2, and e_3 describe different aspects of the same book. On one hand, by comparing e_1 and e_3 there is no way of detecting that they describe the same book, since their contents are completely different. On the other hand, e_1 and e_3 can be recognized to describe the same book described by e_2 since e_1 and e_2 share the same `ISBN` and e_2 and e_3 share the same `title` and `authors`.

To correctly build sets of elements describing the same object we introduce a *vague join* operator. This operator is essentially an outer join, where two vague elements are joined if and only if they refer to the same object, i.e., if their overall dissimilarity is under a *join threshold* λ. The overall dissimilarity between two vague elements v', v'' is given by a function $\delta(v', v'')$, defined in the following. Then, given two vague elements v' and v'', we denote with $v' \uplus v''$ the vague element obtained by joining the elements in v' and v'', that is $v' \uplus v'' = \langle E'_v \cup E''_v, \delta(v', v'')\rangle$.

A simple approach for assessing the overall dissimilarity between two vague elements $v' = \langle E_{v'}, \gamma_{v'}\rangle$ and $v'' = \langle E_{v''}, \gamma_{v''}\rangle$ is that of taking the maximum dissimilarity between every pair of elements in $E_{v'} \cup E_{v''}$. Unfortunately, this definition fails to capture all significant answers when facing particular (e.g., intransitive) behaviors of the function evaluating dissimilarity between XML elements. This case is discussed in the following example.

Example 4.14. Consider three vague elements $v_1 = \langle \{e_1\}, 0\rangle$, $v_2 = \langle \{e_2\}, 0\rangle$, and $v_3 = \langle \{e_3\}, 0\rangle$ where e_1, e_2, and e_3 are the XML elements reported in Figure 4.7, and assume that δ is intransitive, that is $\delta(e_1, e_2) = \delta(e_2, e_3) = 0$ and $\delta(e_1, e_3) = \infty$. If the overall dissimilarity is computed by taking the maximum between each pair, we have $\delta(v_1, v_2) = 0$ but $\delta(\langle \{e_1, e_2\}, 0\rangle, v_3) = \infty$, although e_1, e_2, and e_3 refer to the same object.

It is straightforward to see that choosing the minimum value among every pair would join unrelated elements. The overall dissimilarity notion we adopt solves these problems by adopting an *ad-hoc* approach. The dissimilarity between two vague elements $v' = \langle E_{v'}, \gamma_{v'}\rangle$ and $v'' = \langle E_{v''}, \gamma_{v''}\rangle$, denoted as $\delta(v', v'')$, is computed as follows:

1. if $\exists e' \in v'$ and $e'' \in v''$ such that $e' \neq e''$ and both e' and e'' come from the same source, then $\delta(v', v'') = \infty$;
2. otherwise, if $E_{v'} \cap E_{v''} = \emptyset$, then $\delta(v', v'')$ is the maximum among $\gamma_{v'}$, $\gamma_{v''}$ and $min_{\substack{e' \in v' \\ e'' \in v''}} \delta(e', e'')$;
3. otherwise, $\delta(v', v'') = max(\gamma_{v'}, \gamma_{v''})$.

Essentially, rule (1) states that vague elements containing different elements extracted from the same source do not refer to the same object. This is motivated by the assumption that the same source does not represent twice the same information. Rule (2) characterizes the dissimilarity of a vague element obtained by merging two disjoint vague elements according to the intuition that, since the elements in v' refer to an object o' and the elements in v'' refer to an object o'', it suffices that an element $e' \in v'$ and an element $e'' \in v''$ refer to the same object to conclude that $o' = o''$. Therefore, the dissimilarity of two vague elements v' and v'' is computed as the maximum among the dissimilarity of the elements in v' and v'', respectively, and the minimum dissimilarity between every pair of elements in v' and v''. Finally, rule (3) states that if v' and v'' contain a same element, then both refer to the same object and the dissimilarity of the vague element obtained by merging v' and v'' is the highest between $\gamma_{v'}$ and $\gamma_{v''}$, that is the maximum among the precomputed dissimilarity of the elements in v' and v'', respectively. It is worth noting that, if the XML element dissimilarity is correctly recognized by function δ, this approach allows us to join two vague elements iff they refer to the same object.

Example 4.15. Consider the vague elements of Example 4.14 and assume that e_1, e_2, and e_3 come from different sources. According to the definition of the overall dissimilarity, $\delta(v_1, v_2) = 0$ (rule (2)). Therefore v_1 and v_2 are joined obtaining the vague element $v_4 = \langle \{e_1, e_2\}, 0\rangle$. We also obtain $\delta(v_4, v_3) = 0$ (as $\delta(e_2, e_3) = 0$), thus recognizing that the three XML elements can be grouped together in a vague element $\langle \{e_1, e_2, e_3\}, 0\rangle$.

However, the overall dissimilarity of a set of elements computed using function δ may depend on the order elements are merged, that is $(v_1 \uplus v_2) \uplus v_3$

has an overall dissimilarity value different from $(v_1 \uplus v_3) \uplus v_2$. The following example shows this behavior.

Example 4.16. Consider the vague elements of Example 4.15 and the vague element $v_5 = \langle \{e_1, e_3\}, \infty \rangle$, obtained from $\delta(v_1, v_3) = \infty$. It is easy to see that $v_4 \uplus v_3 = \langle \{e_1, e_2, e_3\}, 0 \rangle$ while $v_5 \uplus v_2 = \langle \{e_1, e_2, e_3\}, \infty \rangle$.

We thus define the vague join operator in such a way that it combines vague elements in every possible way. Moreover, only vague elements whose overall dissimilarity is under the desired threshold λ are considered. We first introduce a "fusion operator" which works over a set of vague elements V, and expands it with the vague elements obtained by joining pairs of vague elements in V, if the resulting overall dissimilarity is smaller or equal to λ.

Definition 4.17. ***(Fusion operator)*** *Let V be a set of vague elements. We define the* fusion operator *Δ as*

$$\Delta(V) = V \cup \{v' \uplus v'' \mid v' \in V,\, v'' \in V, \delta(v', v'') \leq \lambda\}$$

The vague join operator is defined as the least fixpoint of the fusion operator.

Definition 4.18. ***(Vague join)*** *Let V be a set of vague elements. We define the* vague join *of V as the least fixpoint of operator Δ applied to V, denoted as $\bowtie^v (V)$.*

Obviously, since the fusion operator uses the join threshold, this threshold is essential for guaranteeing correct behaviors of the vague join operator. A possible choice is that of providing different thresholds that drive different behaviors, ranging from strictly conservative ($\lambda = 0$) to very approximate.

Example 4.19. Applying the vague join operator with $\lambda \neq \infty$ to the set of vague elements $V = \{\langle \{e_1\}, 0 \rangle,\ \langle \{e_2\}, 0 \rangle, \langle \{e_3\}, 0 \rangle\}$, where e_1, e_2, and e_3 are the XML elements of Example 4.14, we obtain

$$\begin{aligned} \bowtie^v (V) = \{&\langle \{e_1, e_2, e_3\}, 0 \rangle, \langle \{e_1, e_2\}, 0 \rangle, \langle \{e_2, e_3\}, 0 \rangle, \\ &\langle \{e_1\}, 0 \rangle, \langle \{e_2\}, 0 \rangle, \langle \{e_3\}, 0 \rangle\} \end{aligned}$$

thus capturing the fact that elements e_1, e_2, and e_3 refer to the same object.

Observe that, in general, the application of the vague join operator produces a set containing "redundant" vague elements that are strictly contained inside other vague elements.

4.3.5 VXQL Semantics

The semantics of VXQL takes into account the quality of the vague elements obtained, meant as their correspondence with the original query, as explained below. Given a vague element v, as each element in v describes a certain object o, it is reasonable to merge the elements in v to provide a single description of o. This can be achieved by simply concatenating the content of the elements in v. Function *merge* applied to a vague element v returns the concatenation of the contents of the elements $e \in v$.[2] Intuitively enough, the relevance of a vague element v w.r.t. a VXQL query q can be assessed by executing q on $merge(v)$.

Specifically, in order to assess whether a vague element v belongs to the answer of a VXQL query q, we define the transformation cost of v w.r.t. q. Given a vague element v and a VXQL query $q = \langle xp, \tau_g, \tau_l, \kappa_g, \kappa_l \rangle$, we define the cost of v w.r.t. q by considering separately, in $xp = s_1/s_2/\ldots/s_n[f]$, the expression $s_1/s_2/\ldots/s_n$, denoted as xp_s, and the predicate f. We evaluate xp_s on the original elements [3] belonging to v, and f on $merge(v)$.

The cost of v w.r.t. xp_s, denoted as $cost(xp_s, v)$, is evaluated by taking the minimum among the costs associated with the elements in v, i.e.,

$$cost(xp_s, v) = min_{\substack{e \in v \\ xp'_s \in \mathcal{T}_D(xp_s, \tau_g, \kappa_g, e)}} cost(xp_s, xp'_s)$$

The same applies for the number of marked transformations applied:

$$mrk(xp_s, v) = min_{\substack{e \in v \\ xp'_s \in \mathcal{T}_D(xp_s, \tau_g, \kappa_g, e)}} mrk(xp_s, xp'_s)$$

Moreover, as at least $mrk(xp_s, v)$ transformations have been applied to reach the elements in v, the cost associated with f considers only the relaxed expressions obtainable from f by applying a number of marked transformations which is less than or equal to $\kappa^* = \kappa_g - mrk(xp_s, v)$. That is, the cost of v w.r.t. f is defined as:

$$cost(f, v) = min_{rf \in \mathcal{T}_D(f, \tau_g, \kappa*, merge(v))} cost(f, rf)$$

Finally, the cost of v w.r.t. q is defined as $cost(q, v) = cost(xp_s, v) + cost(f, v)$. This function is employed to select the vague elements that satisfy q, that is those whose cost is lower or equal to the specified threshold. The set of *satisfying elements* is defined below.

[2] Observe that the same information can be reported several times in $merge(v)$ and that the relative order among the content of the elements merged together is irrelevant to compute their correspondence w.r.t. the original query.

[3] In the evaluation of an XPath expression $xp_s(e)$ we assume that xp_s is evaluated on the original document containing e.

Definition 4.20. ***(Satisfying elements)*** *Let V be a set of vague elements, and $q = \langle xp, \tau_g, \tau_l, \kappa_g, \kappa_l \rangle$ be a VXQL query. The set of elements in V satisfying q is*

$$\sigma_q(V) = \{v | v \in V,\, cost(q, v) \leq \tau_g\}.$$

As observed before, during the construction of the overall result, the application of the vague join operator may produce vague elements that are subsets of others and thus needless in the final result. Moreover, the same set of XML elements may correspond to one or more vague elements with different overall dissimilarity (due to the definition of vague join). To remove the unneeded vague elements, we employ the pruning operation defined below.

Definition 4.21. ***(Pruned set)*** *Let V be a set of vague elements. The pruned set of V is*

$$\rho(V) = \{v | v \in V,\ \nexists\ v' \in V\ s.t.\, (E_v \subset E_{v'})\,,\\ \nexists\ v'' \in V\ s.t.\, (E_v = E_{v''}, \gamma_v > \gamma_{v''})\}.$$

The overall result is obtained by vaguely joining the answers coming from different sources, selecting the vague elements whose cost is under the specified threshold, and pruning the resulting set of vague elements.

Definition 4.22. ***(VXQL semantics)*** *Let $\mathcal{D}$ be a set of XML data sources, and q be a VXQL query. The application of q to $\mathcal{D}$ is*

$$q(\mathcal{D}) = \rho\Big(\sigma_q\Big(\bowtie^v\ \big(\cup_{D\in\mathcal{D}}\, q(D)\big)\Big)\Big)$$

Example 4.23. Consider again query q of Fig. 4.6 in the scenario depicted in Fig. 4.1 and assume that $\tau_g = l + m$ and $\kappa_g = 0$. The set V of vague elements returned by the vague query evaluation process on the data sources (see Example 4.3.3) contains a vague element for each of the XML elements e_1, e_3, e_4, and e_5. Assuming that the employed function δ correctly identifies elements describing the same object, the application of operator $\bowtie^v$ on V adds to V two new vague elements corresponding to the sets $\{e_1, e_4\}$ and $\{e_3, e_5\}$, (with overall dissimilarity values γ_{14} and γ_{35}, respectively). The transformation costs of the vague elements in V w.r.t q are the following: $cost(q, \langle\{e_1\}, 0\rangle) = l + 2\cdot m$, $cost(q, \langle\{e_3\}, 0\rangle) = 2\cdot m + h$, $cost(q, \langle\{e_4\}, 0\rangle) = cost(q, \langle\{e_5\}, 0\rangle) = l + 2\cdot m$, $cost(q, \langle\{e_1, e_4\}, \gamma_{14}\rangle) = l + 2\cdot m$, $cost(q, \langle\{e_3, e_5\}, \gamma_{35}\rangle) = l$. Therefore, the query answer provided by the application of q on D_1, D_2, and D_3 is the vague element corresponding to $\{e_3, e_5\}$.

The following theorem characterizes the complexity of VXQL query evaluation under this semantics.

Theorem 4.24. *Let $\mathcal{D}$ be a set of XML data sources, and q a VXQL query. The problem of checking whether $q(\mathcal{D})$ is not empty is NP-complete.*

Proof. (*Sketch*) Membership in NP is straightforward. NP hardness is proved by showing a logspace reduction of SAT [50]. Let $\phi = \phi_1, \cdots, \phi_m$ be a boolean formula in conjunctive normal form defined on variables $x_1, \cdots, x_n$. We associate ϕ with a set of XML data sources $\mathcal{D}_\phi$. The set contains n sources D_i, with $i \in [1..n]$, each providing a document rooted by a `vars` tag with two `data` elements. The first `data` element contains an xi subelement with value `true` and the second one contains an xi subelement with value `false`. For instance, the XML document exported by source D_1 has the following structure:

```
<vars>
  <data>
    <x1>true</x1>
  </data>
  <data>
    <x1>false</x1>
  </data>
</vars>
```

Let $q = \langle xp, \tau_g, \tau_l, 0, \infty\rangle$ be a VXQL query where τ_g, $\tau_l \geq 0$ and $xp =$`/vars/data[`$f(\phi)$`]{PD[0]}` with $f(\phi)$ defined as follows:

$$f(\phi) = \begin{cases} \texttt{x}i\texttt{[text()=''true'']\{PD[0]*\}} & \text{if } \phi = x_i \\ \texttt{x}i\texttt{[text()=''false'']\{PD[0]*\}} & \text{if } \phi = \neg\, x_i \\ f(\phi_1) \text{ OR } f(\phi_2) & \text{if } \phi = \phi_1 \text{ OR } \phi_2 \\ f(\phi_1) \text{ AND } f(\phi_2) & \text{if } \phi = \phi_1 \text{ AND } \phi_2 \end{cases}$$

No cost is specified for the deletion of filter $f(\phi)$ from xp. However, as predicate deletion is a marked transformation, vague elements in the final result must satisfy $f(\phi)$, whereas partial answers coming from each source are allowed to violate it. We use a function δ which always evaluates to 0.

Every vague element v in $\bowtie^v_{D\in\mathcal{D}} q(D)$ is composed by taking at most one `data` element from each source in $\mathcal{D}$. Since the vague elements that do not contain at least one data element for each source cannot satisfy $f(\phi)$, $q(\mathcal{D})$ is not empty iff there is a truth assignment for $x_1, \cdots, x_n$ which satisfies ϕ.

We point out that Theorem 4.24 characterizes the complexity of VXQL query evaluation w.r.t. the size of the query and the number of XML data sources. Indeed, the problem of checking whether a query q returns a non-empty answer is NP-hard even if every source returns at most 2 elements as its partial answer to q. The problem is solvable in polynomial time if no source contains duplicate elements (i.e., key constraints are satisfied) and the dissimilarity function δ returns a value higher than the predefined join threshold λ iff applied on a pair of elements that do not describe the same object (see Section 4.4.3).

4.4 Vague Query Evaluation

In this section we first outline an algorithm that evaluates a VXQL query q on a set of XML data sources $\mathcal{D}$. Then, we detail how local vague evaluation is performed and the particular element dissimilarity function we adopt. Finally, we characterize the complexity of VXQL query evaluation and identify cases where it is tractable.

Algorithm 4.25, for each $D \in \mathcal{D}$, evaluates $q(D)$ (Step 1). Then, it combines the partial answers by joining those elements which refer to the same object (Step 2). Next, it eliminates the vague elements whose cost exceeds the specified threshold (Step 3) and, finally, redundant vague elements are pruned (Step 4).

Algorithm 4.25 ***(VXQL query evaluation)***
Input*: A VXQL query* $q = \langle xp, \tau_g, \tau_l, \kappa_g, \kappa_l \rangle$ *and a set of XML data sources* $\mathcal{D}$.
Output*: The set of vague elements* $q(\mathcal{D})$.

```
1. For each $D \in \mathcal{D}$, evaluate $q(D)$ (local evaluation);
2. $V \leftarrow \bowtie^v \left( \cup_{D \in \mathcal{D}} q(D) \right)$ (joining);
3. $V \leftarrow \sigma_q(V)$ (selection of satisfying elements);
4. $V \leftarrow \rho(V)$ (pruning);
5. Return $V$.
```

4.4.1 Local Evaluation

In this section we describe our approach to the evaluation of a VXQL query $q = \langle xp, \tau_g, \tau_l, \kappa_g, \kappa_l \rangle$ with respect to the local XML database of a source. W.l.g. we assume that the XML database of each source (denoted as D) has a unique root element r_D. The evaluation of $q(D)$ is performed by function $evQuery(q, D)$ and consists in evaluating $xp_{\tau_l, \kappa_l}(D)$.

We first introduce some notation. Given a weighted step ws and a transformation $t \in \{AR, ND, NR\}$, we define a function η_t that, applied to ws, returns the transformation cost specified in ws for t if t is allowed, and ∞ otherwise. Moreover, we define a function μ_t that evaluates to 1 if t is marked in ws and 0 otherwise. Finally, given a weighted simple step s, we denote as $desc(s)$ the weighted simple step obtained from s by replacing the axis in s with the descendant one, and denote as $axis(s)$ and $nTest(s)$ the axis and the node test in s, respectively.

The intermediate results consist of sets of tuples representing node bindings. Each tuple $\langle n, cost, tr \rangle$ means that node n has been bound through the application of a certain set of basic transformations entailing a total cost equal to $cost$, and applying tr marked transformations.

We now define operator $\odot$, which eliminates useless bindings from a set $\mathcal{B}$. More precisely, for each binding of an XML element n, it eliminates every other binding of n having worse associated costs. The formal definition of $\odot$ is:

$$\begin{aligned}\odot(\mathcal{B}) =& \{\langle n, cost, tr\rangle \in \mathcal{B} \mid \nexists \langle n, cost', tr'\rangle \in \mathcal{B},\\ &((cost' < cost,\ tr' \leq tr) \vee\\ &(cost' \leq cost,\ tr' < tr))\}.\end{aligned}$$

Note that if one or both local thresholds are equal to ∞, the filtering operation that in the general case is performed by operator $\odot$ is much simpler. For instance, if $\tau_l = \infty$, for each XML element, only one among the bindings associated with it (the one having the minimum number of applied marked transformations) must be kept in the set of current bindings.

Function *evStep* evaluates a weighted step w.r.t. a set of node bindings. Let $\mathcal{B}$ be a set of node bindings, and ws a weighted step of the form $s[f]$; $\mathit{evStep}(\mathcal{B}, ws)$ is defined as

$$\mathit{evStep}(\mathcal{B}, ws) = \odot(\phi(\sigma(\mathcal{B}, s), f))$$

where $\sigma(\mathcal{B}, s)$ is a function returning a new binding set which satisfies the weighted simple step s starting from the bindings in $\mathcal{B}$, and $\phi(\mathcal{B}, f)$ is a function evaluating predicate f on a binding set, that is filtering out those bindings which do not satisfy f w.r.t. κ_l and τ_l (the formal definition of σ and ϕ will be given later on).

Function *evExp* evaluates a VXQL expression w.r.t. a set of node bindings. Let $\mathcal{B}$ be a set of node bindings and xp a VXQL expression of the form $ws_1/\cdots/ws_n$. Function *evExp* essentially invokes *evStep* on each weighted step ws in xp; *evExp* is therefore recursively defined as follows:

$$\mathit{evExp}(\mathcal{B}, xp) = \begin{cases} \mathit{evStep}(\mathcal{B}, ws_1) & \text{if } n = 1\\ \mathit{evStep}(\mathit{evExp}(\mathcal{B}, ws_1/\cdots/ws_{n-1}), ws_n) & \text{if } n > 1\end{cases}$$

To define function σ, we first introduce a function *ext*, which takes as arguments a node binding $\langle n, cost, tr\rangle$ and a weighted simple step s, and computes a new binding set by evaluating s starting from the XML element n, considering a cost already paid equal to $cost$ and a number of marked transformations already applied equal to tr. Function *ext* also tries to apply node renamings that do not violate the given thresholds. The XML database is accessed by means of the function $\mathit{search}(n, s)$, that returns a new set of XML elements resulting from the evaluation of the XPath step s on the context XML element n. The formal definition of function *ext* is given in Fig. 4.8, where $\mathcal{L}_D$ denotes the set of element names in the XML database D containing node n.

We now define function $\sigma(\mathcal{B}, s)$, which evaluates a weighted simple step s from each binding $b \in \mathcal{B}$, by applying all possible transformations that do not exceed the thresholds. The function returns a new binding set obtained by uniting the bindings yielded by the exact evaluation of s from each single binding b, and those computed by functions $\sigma_{rel}(b, s)$ and $\sigma_{del}(b, s)$, that evaluate step s trying to apply axis relaxation and node deletion.

$$ext(\langle n, cost, tr\rangle, s) = \left\{ \langle n', c', tr'\rangle \;\middle|\; \begin{array}{l} l \in \mathcal{L}_D \text{ s.t. } c' = cost + \\ \quad +semDist(nTest(s), l) \cdot \eta_{NR}(s) \leq \tau_l, \\ n' \in search(n, axis(s) :: l), \\ tr' = tr + \mu_{NR}(s) \leq \kappa_l \end{array} \right\}$$

Fig. 4.8: Definition of function *ext*

Specifically, function σ_{rel} calls function *ext* imposing a descendant step, i.e., *ext* is invoked on $desc(s)$. σ_{del} invokes *search* with a step of the form descendant-or-self::$*$. The transformation cost and the number of applied marked transformations are also updated. The formal definitions of functions σ_{rel} and σ_{del} are shown in Fig. 4.9.

$$\sigma_{rel}(\langle n, cost, tr\rangle, s) = \begin{cases} ext(n, c', tr', \langle desc(s)\rangle) & \text{if } tr' = tr + \mu_{AR}(s) \leq \kappa_l, \\ & \quad c' = cost + \eta_{AR}(s) \leq \tau_l; \\ \emptyset & \text{otherwise.} \end{cases}$$

$$\sigma_{del}(\langle n, cost, tr\rangle, s) = \begin{cases} \left\{ \begin{array}{c} \langle n', c', \; tr'\rangle \, | n' \in search(n, \\ \text{descendant-or-self} :: *) \end{array} \right\} & \text{if } tr' = tr + \mu_{ND}(s) \leq \kappa_l, \\ & \quad c' = cost + \eta_{ND}(s) \leq \tau_l; \\ \emptyset & \text{otherwise.} \end{cases}$$

Fig. 4.9: Definitions of functions σ_{rel} and σ_{del}

Finally, $\sigma(\mathcal{B}, s)$ is defined as

$$\sigma(\mathcal{B}, s) = \odot \left(\bigcup_{b \in \mathcal{B}} (\sigma^*(b, s)) \right)$$

where $\sigma^*(b, s) = \sigma_{rel}(b, s) \cup \sigma_{del}(b, s) \cup ext(b, s)$.

Function $\phi(\mathcal{B}, f)$ filters out the bindings in $\mathcal{B}$ that do not satisfy filter f. For a filter consisting of a single VXQL expression $\phi(\mathcal{B}, f)$ is defined as follows:

$$\phi(\mathcal{B}, f) = \odot \left(\left\{ \langle n, cost', tr'\rangle \;\middle|\; \begin{array}{l} \langle n, cost, tr\rangle \in \mathcal{B}, \\ \langle n', cost', tr'\rangle \in evExp(\{\langle n, cost, tr\rangle\}, f) \end{array} \right\} \right)$$

For a filter of the form $f = f_1 \; AND \; f_2$, function ϕ is applied to $\mathcal{B}$ using f_1, then the result is filtered using f_2:

$$\phi(\mathcal{B}, f_1 \; AND \; f_2) = \phi(\phi(\mathcal{B}, f_1), f_2))$$

whereas for a filter of the form $f = f_1\ OR\ f_2$, function ϕ is applied to $\mathcal{B}$ using the two subfilters separately, then the results are united:

$$\phi(\mathcal{B}, f_1\ OR\ f_2) = \odot(\phi(\mathcal{B}, f_1) \cup \phi(\mathcal{B}, f_2)).$$

For simplicity we do not detail the behavior of function ϕ when evaluating a filter by trying to apply predicate relaxations.

Finally, the evaluation of a VXQL query $q = \langle xp, \tau_g, \tau_l, \kappa_g, \kappa_l \rangle$ on a source exporting a document D is performed by function *evQuery*, which is defined as follows:

$$evQuery(q, D) = evExp(\{\langle r_D, 0, 0\rangle\}, xp).$$

4.4.2 Assessing "Semantic" Dissimilarity

The semantics of VXQL is in general independent of the particular technique adopted to measure the semantic dissimilarity between two XML elements, i.e., any technique which is able to assess whether two XML elements refer to the same real-world object can be used. In this section we define the dissimilarity function currently employed.

In general, since different information sources adopt different representations of the same information, it may be very difficult to establish whether XML elements coming from different sources refer to the same object. For this purpose, a naive strategy is to measure the structure or content dissimilarity between (whole) elements. This approach is unsuitable since two elements can be dissimilar from one another but still refer to the same object. For instance, elements e_1 and e_2 in Fig. 4.7 are much different from each other, even if e_1 and e_2 definitely refer to the same book, as they share the ISBN's.

A more suitable approach consists in evaluating the dissimilarity degree between element's keys, if key constraints are imposed. In this way, elements e_1 and e_2 in Fig. 4.10, which have the same key (same `ISBN` subelement), and thus refer to the same book, are recognized to be similar. Nevertheless, since element's keys are defined locally on each data source, it can happen that elements referring to the same object are characterized by keys having a different structure. For instance, a `person` element can be identified by its `name` and `surname` subelements in a source, and by its `fullname` subelement in another source. In such cases, techniques as those proposed in [114, 54] can be employed to assess key dissimilarity. Specifically, in [114], the computation of the edit distance between two trees is introduced, defined as the minimum number of operations (node insertion, deletion, renaming) required to transform one tree into the other. [54] presents a framework for approximate XML joins based on tree-edit distance where upper and lower bounds are given for the distance, and reference sets are used that reduce the number of distances to compute in a join. It should be noted that, in general, evaluating the dissimilarity between XML elements by looking at their keys makes sense only if the elements refer to objects of the same type. Thus, this approach can be used when no disjunction appears in the output nodes of the queries.

Moreover, when comparing element's keys, the possibility of two elements referring to the same object but being identified by completely different keys must be taken into account. For instance, elements e_2 and e_3 in Fig. 4.10 are identified by different keys, but both describe the same book. In these cases, any dissimilarity function that checks key dissimilarity, even by applying the edit distances defined in [114, 54], fails in recognizing elements referring to the same object.

A more effective approach is that of testing whether the information represented by the key of one element is contained in the second, or vice-versa. The XML dissimilarity measure we adopt in VXQL is based on this idea. In the absence of key constraints defined on the XML elements, the distance measures proposed in [114, 54] can still be applied to estimate dissimilarity degrees between whole elements.

Several alternative definitions of XML keys have been proposed in the literature, such as XSchema key constraints and XML functional dependencies. Here we simply assume that the key of an XML element e is an XML element $\mathcal{K}(e)$ obtained by possibly removing some of the subelements of e or some of the attributes of e itself or of one of its subelements. For instance, the keys of the elements in Fig. 4.7 are reported in Fig. 4.10.

```
<book>
    <ISBN>47</ISBN>
</book>
```

$\mathcal{K}(e_1)$

```
<book>
    <ISBN>47</ISBN>
</book>
```

$\mathcal{K}(e_2)$

```
<book>
    <title>T<title>
    <authors>
        <author>A1</author>
        <author>A2</author>
    </authors>
</book>
```

$\mathcal{K}(e_3)$

Fig. 4.10: Keys of the elements of Fig. 4.7

We exploit VXQL expressions to measure the dissimilarity degree of the objects described by two XML elements. Specifically, given two XML elements e', e'', our approach exploits VXQL expressions to check whether the information contained in $\mathcal{K}(e')$ is "represented" in e'' and vice-versa. Indeed, if either the information of $\mathcal{K}(e')$ is represented in e'', or the information of $\mathcal{K}(e'')$ is represented in e', it is reasonable to say that e' and e'' refer to the same object. For instance, consider elements e_2 and e_3 in Fig. 4.10. It can be noted that the keys identifying the two elements are completely different, but the information contained in $\mathcal{K}(e_3)$ is fully contained in e_2. In this case, we conclude that e_2 and e_3 refer to the same book.

In order to check whether the information contained in $\mathcal{K}(e')$ is represented in e'' we first translate $\mathcal{K}(e')$ into a *key testing VXQL expression* and then execute it on e'' (named *target* element). Obviously, the VXQL expression associated with $\mathcal{K}(e')$ must represent every relevant information represented in $\mathcal{K}(e')$, while allowing that this information is somehow "rearranged" inside

$$xp^{\mathcal{K}}_e = \begin{cases} \texttt{en\{AR[1],NR[}r\texttt{],ND[1+}r\texttt{+}d\texttt{]\}} & \text{if } e = \texttt{<en>}e_1 \ldots e_n\texttt{<en>} \\ \quad [xp^{\mathcal{K}}_{e_1}] \ldots [xp^{\mathcal{K}}_{e_n}] & \text{where } e_1 \ldots e_n \text{ are XML elements} \\ \texttt{en\{AR[1],NR[}r\texttt{]\}} & \text{if } e = \texttt{<en/>} \\ \texttt{en\{AR[1],NR[}r\texttt{],ND[1+}r\texttt{+}d\texttt{]\}} & \text{if } e = \texttt{<en>}txt\texttt{<en>} \\ \quad \texttt{[text()='txt']\{SD[1],EQ[1]\}} & \text{where } txt \text{ is a string (\#PCDATA)} \end{cases}$$

Fig. 4.11: Key testing VXQL expression

e''. Hence, the relative order of subelements in the key is disregarded in the VXQL expression, since it is unlikely that key information is represented by using the relative order of subelements (this feature is indeed not supported by XML key constraint languages). Moreover, some flexibility is guaranteed in the execution of the expression. Specifically, lower weights are associated with transformations which alter only the "structure" of the key, retaining the semantics of its information. The formal definition of the VXQL expression associated with the key of an XML element e (denoted as $xp^{\mathcal{K}}_e$) is reported in Fig. 4.11.

Observe that $xp^{\mathcal{K}}_e$ does not permit the deletion of steps corresponding to textual filters or leaf elements, since they are considered to be the most important information in the recognition of dissimilarity between the key and the target element. $xp^{\mathcal{K}}_e$ instead assumes a unitary cost for axis relaxation, *-node insertion and relaxation of equality predicates, since these transformations only modify the structure of the key. Moreover, $xp^{\mathcal{K}}_e$ assigns an higher cost ($r \geq 1$) to node renaming, since this transformation permits the modification of the semantics of element names in the key. Finally, (internal) node deletion essentially corresponds to applying both node renaming and axis relaxation. Hence, the cost assigned to node deletion in $xp^{\mathcal{K}}_e$ is greater than the sum of the costs assigned to node renaming and axis relaxation ($1 + r + d$, with $d \geq 0$).

Example 4.26. The key testing VXQL expression associated with element e_3 in Fig. 4.10 ($xp^{\mathcal{K}}_{e_3}$) is the following (where we assume $r = d = 1$):

```
book{NR[1],ND[3]}
  [title{AR[1],NR[1],ND[3]}[text()='T']{SD[1],EQ[1]}]
  [  authors{AR[1],NR[1],ND[3]}
       [author{AR[1],NR[1],ND[3]}[text()='A1']{SD[1],EQ[1]}]
       [author{AR[1],NR[1],ND[3]}[text()='A2']{SD[1],EQ[1]}]   ]
```

Given two XML elements e' and e'', and the minimum-cost relaxed version rxp of $xp^{\mathcal{K}}_{e'}$ such that e'' satisfies rxp, we take the cost of rxp as a measure of semantic dissimilarity between e' and e''. For instance, consider elements e_2 and e_3 of Fig. 4.10; since $xp^{\mathcal{K}}_{e_3}$ is satisfiable on e_2 applying no transformation at all, then e_2 and e_3 refer to the same book.

Before introducing the formal definition of the dissimilarity function based on key testing VXQL expressions, denoted as $\delta^{\mathcal{K}}$, we introduce some preliminary definitions. Given a VXQL expression xp and an XML element e, the function *cost* applied to xp and e returns the minimum cost of obtaining a relaxed expression rxp from xp such that e satisfies rxp, i.e. $cost(xp, e) = min_{rxp \in \mathcal{T}_D(xp,\infty,\infty)} cost(xp, rxp)$, where D is the document having e as its root element. As it will be clearer in the following, cost thresholds are not needed in this computation. If a relaxed expression capturing e does not exist, i.e. $\mathcal{T}_D(xp, \infty, \infty) = \emptyset$, we consider $cost(xp, e) = \infty$. In the following, given a VXQL expression xp and an XML element e, we denote as $cost(xp, e)$ the cost of applying xp to e.

Given two XML elements e' an e'', the dissimilarity function $\delta^{\mathcal{K}}$ uses function *cost* to check whether the information contained in $\mathcal{K}(e')$ is present in e'' and vice-versa, and thus if the two elements refer to the same object. The formal definition of $\delta^{\mathcal{K}}$ is the following:

$$\delta^{\mathcal{K}}(e', e'') = min\left(\frac{cost(xp^{\mathcal{K}}_{e'}, e'')}{\eta_\infty(xp^{\mathcal{K}}_{e'})}, \frac{cost(xp^{\mathcal{K}}_{e''}, e')}{\eta_\infty(xp^{\mathcal{K}}_{e''})}\right)$$

Observe that function $\delta^{\mathcal{K}}$ states that e' and e'' refer to the same object if either the information of $\mathcal{K}(e')$ is represented in e'' or the information of $\mathcal{K}(e'')$ is represented in e'. For instance, function $\delta^{\mathcal{K}}$, applied to elements e_1, e_2 and e_3 of Fig. 4.10, returns the following dissimilarity values: $\delta^{\mathcal{K}}(e_1, e_2) = 0$, $\delta^{\mathcal{K}}(e_2, e_3) = 0$ and $\delta^{\mathcal{K}}(e_1, e_3) = \infty$.

4.4.3 Complexity of VXQL Query Evaluation

The complexity of the evaluation of a VXQL query is stated by the following proposition.

Proposition 4.27. ***(Local evaluation complexity)*** *Let $q = \langle xp, \tau_g, \tau_l, \kappa_g, \kappa_l \rangle$ be a VXQL query, and D an XML document. $q(D)$ can be computed in time $O(|xp|^2 \cdot |D|^2)$, where $|D|$ is the number of elements in D and $|xp|$ is the number of steps in xp.*

Proof. It is straightforward to see that evaluating $q(D)$ requires at most $|xp|$ (recursive) invocations of function *evStep*, one for each weighted step appearing in xp. Since the generic invocation of *evStep* on a set of bindings $\mathcal{B}$ and a weighted step $ws = s[f]$ computes $\odot(\phi(\sigma(\mathcal{B}, s), f))$, the complexity of evaluating *evStep*$(\mathcal{B}, ws)$ is given by the sum of the complexities of computing:

1. $\mathcal{B}' = \sigma(\mathcal{B}, s) = \bigcup_{b \in \mathcal{B}}(\sigma^*(b, s))$;
2. $\mathcal{B}'' = \phi(\mathcal{B}', f)$;
3. $\odot(\mathcal{B}'')$.

Since for each invocation of *evStep* $\mathcal{B}$ either derives from the execution of another invocation of *evStep* or it is the initial set of bindings, the cardinality of $\mathcal{B}$ is bounded by $m \cdot |D|$, where m is the number of marked transformations in xp (thus m is $O(|xp|)$). Moreover, it is easy to see that every invocation of function *ext* can be accomplished in time $O(|D|)$, since it suffices to visit the XML document (tree) starting from a context element. Hence, σ^* can be computed in time $O(|D|)$ and computing $\mathcal{B}'$ takes time $O(m \cdot |D|^2)$; the cardinality of $\mathcal{B}'$ is $O(m \cdot |D|^2)$ as well. Moreover, as ϕ filters out the bindings which do not satisfy the filter f, the cardinality of $\mathcal{B}''$ is $O(m \cdot |D|^2)$, too. Operator $\odot$ filters out the unneeded bindings from $\mathcal{B}''$ in at most $O(|\mathcal{B}''|) = O(m \cdot |D|^2) = O(|xp| \cdot |D|^2)$ steps.

Computing $\mathcal{B}''$ either requires to invoke *evExp* (if f consists of an XPath expression) or to compute $\phi(\phi(\mathcal{B}', f_1), f_2))$ (if $f = f_1\ AND\ f_2$) or $\odot(\phi(\mathcal{B}', f_1) \cup \phi(\mathcal{B}', f_2))$ (if $f = f_1\ OR\ f_2$). Therefore, as $|\phi(\mathcal{B}', f_1) \cup \phi(\mathcal{B}', f_2)| \leq 2 \cdot |\mathcal{B}'|$, $\mathcal{B}''$ can be computed in time $O(|\mathcal{B}'|) = O(m \cdot |D|^2) = O(|xp| \cdot |D|^2)$.

Finally, since Steps (1), (2) and (3) can be computed in time $O(|xp| \cdot |D|^2)$, $q(D)$ can be computed in time $O(|xp|^2 \cdot |D|^2)$.

Given a set of XML data sources $\mathcal{D}$, we denote as $\Delta_{\mathcal{D}}$ the maximum size of the XML databases exported by the sources, i.e., $\Delta_{\mathcal{D}} = max_{D \in\ \mathcal{D}}(|D|)$. The following proposition states the overall complexity of Algorithm 4.25.

Proposition 4.28. ***(VXQL evaluation complexity)*** *Let $\mathcal{D}$ be a set of XML data sources, and $q = \langle xp, \tau_g, \tau_l, \kappa_g, \kappa_l \rangle$ a VXQL query. Algorithm 4.25 invoked on q and $\mathcal{D}$ returns $q(\mathcal{D})$ in time $O\left(\Delta_{\mathcal{D}}^{3 \cdot |\mathcal{D}|} + |\mathcal{D}| \cdot |xp|^2 \cdot \Delta_{\mathcal{D}}^2\right)$.*

Proof. Step 1 requires to evaluate q on each source, thus its complexity is $O(|\mathcal{D}| \cdot |xp|^2 \cdot \Delta_{\mathcal{D}}^2)$ (Proposition 4.27).

Since for each $D \in \mathcal{D}$ the cardinality of $q(D)$ is $O(\Delta_{\mathcal{D}})$ and no vague element can contain two XML elements coming from the same source, during the whole execution of Algorithm 4.25 the cardinality of V is $O(\Delta_{\mathcal{D}}^{|\mathcal{D}|})$. Since computing the fixpoint of the fusion operator applied on V requires at most $\Delta_{\mathcal{D}}^{|\mathcal{D}|}$ joins, and each join can be performed in quadratic time w.r.t. the cardinality of V, Step 2 is accomplishable in time $O(\Delta_{\mathcal{D}}^{3 \cdot |\mathcal{D}|})$. Steps 3 and 4 perform a selection on set V, and are thus feasible in time $O(\Delta_{\mathcal{D}}^{2 \cdot |\mathcal{D}|})$. Therefore the complexity of Algorithm 4.25 is $O\left(\Delta_{\mathcal{D}}^{3 \cdot |\mathcal{D}|} + |\mathcal{D}| \cdot |xp|^2 \cdot \Delta_{\mathcal{D}}^2\right)$.

Proposition 4.28 states that the complexity of Algorithm 4.25 may be exponential w.r.t. the number of XML data sources. Hence, running Algorithm 4.25 may be impractical in the presence of many sources. The main source of complexity is the cardinality of set V, i.e., the number of vague elements in $\bowtie^v \left(\cup_{D \in \mathcal{D}}\, q(D)\right)$. However, since the number of different objects represented in the set of XML data sources is at most $|\mathcal{D}| \cdot \Delta_{\mathcal{D}}$, the number of vague elements in $\bowtie^v_{D \in \mathcal{D}}\, q(D)$ should be at most $|\mathcal{D}| \cdot \Delta_{\mathcal{D}}$ if the vague join operator correctly recognizes different elements describing the same object.

Definition 4.29. ***(Robust dissimilarity function)*** *An XML element dissimilarity function δ is said to be* robust *iff, given three XML elements e_1, e_2,* *and e_3 such that e_1 and e_2 refer to the same object, and this object is different from the one described by e_3, it holds that $\delta(e_1, e_2) < \delta(e_1, e_3)$.*

If the dissimilarity function is robust, it is feasible to prune intermediate results during the evaluation of the vague join. The *pruned vague join* operator $\bowtie^*$ is defined by the following algorithm.

Algorithm 4.30 $\bowtie^*$
Input*: A set of vague elements V.*
Output*: The set of vague elements $\bowtie^* (V)$.*

```
1) $V' \leftarrow V$;
2) while $\exists v', v'' \in V'$ such that $\delta(v', v'') \le \lambda$
  2.a) select $v_1, v_2 \in V'$ such that $\delta(v_1, v_2) = min_{v',v'' \in V'}(\delta(v', v''))$
  2.b) $V' \leftarrow V' - \{v_1, v_2\} \cup \{v_1 \uplus v_2\}$;
3) Return $V'$.
```

Observe that the pruned vague join operator, at step $2.a$, performs a greedy selection of a pair of vague elements in the set. However, as shown in Proposition 4.31, if δ is robust, the result of the operator contains all the vague elements which refer to the same object and are contained in $\rho(\bowtie^v (V))$.

Proposition 4.31. *Let q be a VXQL query, and $\mathcal{D}$ a set of XML data sources. If δ is a robust dissimilarity function, then $\bowtie^* (\cup_{D \in \mathcal{D}} q(D))$ contains all the vague elements $v \in \rho(\bowtie^v (\cup_{D \in \mathcal{D}} q(D)))$ which refer to the same object.*

Proof. Let V be $\cup_{D \in \mathcal{D}} q(D)$. We prove that $\bowtie^* (V)$ contains all the vague elements $v \in \rho(\bowtie^v (V))$ which refer to the same object reasoning by contradiction. Assume that there is a vague element $v \in \rho(\bowtie^v (V))$ which contains only elements referring to the same object o and $v \notin \bowtie^* (V)$. Since $v \notin \bowtie^* (V)$, it must be the case that there is a subset v' of v such that there is a vague element $v'' \in V$ with $v'' \nsubseteq v$, $\delta(v', v'') \le \lambda$ and there is no vague element $v^* \in V$ with $v^* \subseteq v - v'$ such that $\delta(v', v^*) < \delta(v', v'')$. Observe that both v^* and v'' consist of a single element since they belong to V. We show the contradiction reasoning by cases.

- v'' refers to o. In this case, since no XML element in v comes from the same source of v'', then $v \uplus v''$ belongs to V, thus contradicting that $v \in \rho(\bowtie^v (V))$.
- v'' does not refer to o. In this case, since δ is robust, for each vague element $\hat{v} \in v - v'$ consisting of a single XML element, it holds that $\delta(v', \hat{v}) < \delta(v', v'')$. The latter is contradicted by $\hat{v} \in V$.

Example 4.32. The application of the pruned vague join operator to the set of vague elements $V = \{\langle\{e_1\}, 0\rangle, \langle\{e_2\}, 0\rangle, \langle\{e_3\}, 0\rangle\}$, where e_1, e_2, e_3 are the XML elements of Fig. 4.10, provides $\bowtie^* (V) = \{\langle\{e_1, e_2, e_3\}, 0\rangle\}$ independently of the order of selection of pairs in the set.

In general, it holds that function δ is robust if it behaves correctly w.r.t. the join threshold, i.e., it returns a dissimilarity value under the join threshold if and only if the compared elements refer to the same object.

By applying $\bowtie^*$ in Step 2 of Algorithm 4.25 instead of $\bowtie^v$, we obtain a polynomial-time algorithm (*VXQL query evaluation with boosted pruning*) for evaluating $q(\mathcal{D})$.

Proposition 4.33. *Let $q = \langle xp, \tau_g, \tau_l, \kappa_g, \kappa_l \rangle$ be a VXQL query, and $\mathcal{D}$ a set of XML data sources. VXQL query evaluation with boosted pruning, invoked on q and $\mathcal{D}$, works in time $O\left(|\mathcal{D}| \cdot |xp|^2 \cdot \Delta_{\mathcal{D}}^2 + (|\mathcal{D}| \cdot \Delta_{\mathcal{D}}|)^3\right)$, where $|xp|$ is the number of steps in xp.*

Proof. As shown in the proof of Proposition 4.28, the first step of Algorithm 4.25 can be done in time $O(|\mathcal{D}| \cdot |xp|^2 \cdot \Delta_{\mathcal{D}}^2)$. Moreover, by applying boosted pruning, the size of V is $O(\mathcal{D} \cdot \Delta_{\mathcal{D}})$. Therefore, Steps 3 and 4 can be done in time $O\left((|\mathcal{D}| \cdot \Delta_{\mathcal{D}}|)^2\right)$. Finally, $\bowtie^*$ (V) can be computed in time $O\left((|\mathcal{D}| \cdot \Delta_{\mathcal{D}}|)^3\right)$, as the cycle in Algorithm 4.30 is executed at most $|\mathcal{D}| \cdot \Delta_{\mathcal{D}}|$ times and every step of the cycle can be done in time $O\left((|\mathcal{D}| \cdot \Delta_{\mathcal{D}}|)^2\right)$.

Observe that the bound stated by Proposition 4.33 is actually very conservative, as the answer of a query usually consists in only a fraction of the XML elements in each source. Thus, VXQL query evaluation with boosted pruning can be profitably exploited to evaluate a VXQL query even in the case that the number of sources is high.

4.5 An Application Scenario

The techniques proposed in this paper have been implemented in a peer-to-peer (P2P) application scenario [38, 39]. In particular, we considered a *hybrid* P2P system [17, 85, 89], where some distinguished peers (*super-peers*) act as resource information indices, that maintain meta-information about the resources made available by the different peers, and are possibly organized in P2P networks themselves.

The system implements the architecture shown in Fig. 4.12. In particular, the left-hand side of the figure depicts the modules implemented by peers, and its right-hand side depicts the modules implemented by super-peers. Each peer is connected to a unique super-peer.

Besides the underlying database management subsystem, the architecture of peers comprises four main modules: the *P2P network sublayer*, the *Synopsis builder*, the *Querying API/User interface*, and the *Query engine*. The P2P network sublayer manages the interactions with the underlying network. The synopsis builder computes concise representation of the stored XML data (whose structure will be detailed in the following), and sends them to the super-peer of reference, through the P2P network sublayer. The querying API/user interface module manages the interactions with users. It provides an

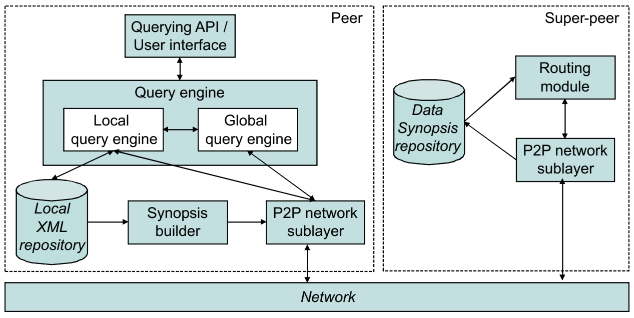


Fig. 4.12: System architecture

API for submitting queries in their textual form and collecting results. A user interface allows the user to (i) specify queries in both graphical and textual form; (ii) obtain a graphical representation of the results as they are received (as it will be clearer in the following, the systems aims at firstly contacting the peers that are likely to provide results); (iii) decide, on the basis of his/her degree of satisfaction, when to stop the process. The query engine implements the query evaluation algorithm and the logic for combining partial answers coming from different sources. These functionalities are managed separately by two submodules:

- The *Local query engine* applies the vague query evaluation process over the local XML database, producing partial answers. Such answers may bring along the information which is subsequently used to evaluate the degree of dissimilarity among different XML elements. The results of the local query evaluation process are returned to the global query engine if the query was submitted to the local peer, otherwise they are sent back through the P2P network sublayer. The local query engine also connects to an external ontology (not shown in the figure) that provides the semantic distance function between two element names.
- The *Global query engine* is employed when a query is issued locally. It forwards the query to the super-peer of reference and collects answers through the P2P network sublayer, then completes the global query evaluation process by joining the partial results obtained and returning them to the user through the querying API.

The architecture of super-peers comprises three main modules: the *Synopsis repository*, the *P2P network sublayer*, and the *Routing module*. The P2P network sublayer receives data synopses from peers and stores them into the repository. Moreover, it receives vague queries from peers and passes them to the routing module. The routing module works in co-operation with the other super-peers. It gathers data synopses from its local repository and from the repositories of other super-peers, then it applies a routing strategy that, by exploiting the information in the synopses, is capable of (*i*) reducing the number of query issued on non-relevant peers, i.e., peers whose local schema ensures that the local query evaluation would not provide results; (*ii*) giving priority to peers that will possibly provide more results. The strategy is described in the next section.

4.5.1 Routing Strategy

Our proposed routing strategy uses the *XSketch* data synopses proposed in [88]. The XSketch synopsis associated with an XML document is a graph whose nodes represent sets of elements in the document that have the same name. Each node in the synopsis is annotated with the cardinality and the shared element name of the corresponding set. An edge between two nodes n_1, n_2 represents a parent-child relationship between an element in n_1 and an element in n_2. Moreover, the edge from n_1 to n_2 is labeled with `F` iff every element in n_1 has at least one child in n_2; the edge is instead labeled with `B` iff for every element in n_2, its parent is in n_1. For instance, in the document represented by the synopsis in Fig. 4.13,[4] (*i*) there are 2 `book` and 4 `title` elements; (*ii*) each `book` and `paper` has a `title`; (*iii*) each `book` has an `isbn`, and `isbn`s are children of `book`s only; (*iv*) the 4 `authors` elements are children of both `book`s and `paper`s, but all of the 11 `author` elements are children of `authors` elements.

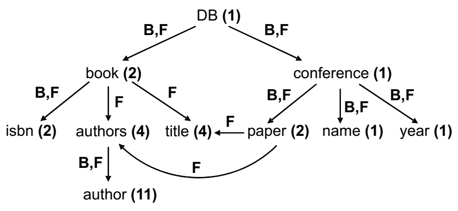


Fig. 4.13: An example XSketch synopsis

An XSketch synopsis can be exploited to estimate the selectivity of an XPath expression, that is the number of XML elements that are selected by the expression. In general, the selectivity estimation of an XPath expression xp using a synopsis $\mathcal{S}$ is performed by first computing the whole set of embeddings of xp in $\mathcal{S}$, then summing up the selectivity associated with each embedding. In particular, our algorithm uses the algorithm proposed in [88] to compute the selectivity of an XPath expression xp w.r.t. a node n of the synopsis, denoted as $sel(xp, n)$.

Selectivity estimation is used by the query routing module to compute an overall *score* given to a synopsis with respect to a VXQL query. This score

[4] For the sake of readability, textual nodes are not represented in the figure.

is then employed to drive routing decisions, i.e., more priority is given to the peers whose synopses exhibit higher scores. For each node in the synopsis, the selectivity of the transformed versions of the query w.r.t. the node is computed. Since a transformed query may not represent all the original query conditions, we weigh the selectivity associated with a node in the synopsis with the "relative" cost of the transformed query which selects the node. Given a VXQL query $q = \langle xp, \tau_g, \tau_l, \kappa_g, \kappa_l \rangle$ and a relaxed expression rxp obtained from xp, the relative transformation cost of q is given by the cost for transforming xp into rxp ($cost(xp, rxp)$) divided by the maximum transformation cost of xp ($\eta_{\kappa_l}(xp)$). The score given to a synopsis $\mathcal{S}$ w.r.t. a VXQL expression xp is defined as follows:

$$score(q, \mathcal{S}) = \sum_{n \in \mathcal{S}} max_{rxp \in \mathcal{T}_{\mathcal{S}}(xp, \tau_l, \kappa_l, n)} \left(sel(rxp, n) * \left(1 - \frac{cost(xp, rxp)}{\eta_{\kappa_l}(xp)}\right)\right)$$

Note that the formula correctly rules out non-output nodes as no transformed query exists for them under the cost thresholds.

4.5.2 Experimental Evaluation

In this section we describe the experimental evaluation we performed to assess the effectiveness of our proposed techniques in the previously-described P2P scenario. Observe that, with respect to the preliminary experimental results reported in [38, 39], new sources have been considered and some of them have been used to replace data sources that in [38, 39] provided synthetical data. The resulting set of sources is more heterogeneous as the schemas of the new sources are different. The actual setting on which the experiments have been carried on is described by the following parameters:

- the system was composed of a network of 104 Pentium IV machines, with RAMs ranging from 512MB to 2GB;
- the peers provided clinical and diagnostical data;
- the peers in the system adopted 12 different schemas and differently-structured keys comprising social security, fiscal, and personal data;
- 8 different (uniform-cost) queries, with different degrees of selectivity, were issued against the system; the queries are reported in Table 4.1;
- 12 of the 104 peers acted as super-peers and were part of a fully-connected network;
- the data had an overall size of 120MB;
- three different global cost thresholds were employed, corresponding to the 50% (*high*), 30% (*medium*), and 10% (*low*) of the maximum cost of the transformed versions of the queries;
- the local cost threshold was set equal to the global one increased by a 25%;
- cost constants r and d (see Section 4.4.2) were set to 1.

Query ID	Meaning
Q1	Patients who suffered from a specific disease
Q2	Patients who suffered from a specific disease in a certain year
Q3	Patients who suffered from two specific diseases
Q4	Patients with a specific treatment
Q5	Patients with a specific treatment in a certain year
Q6	Patients with two specific treatments
Q7	Patients with a specific surgery undergone
Q8	Patients with two specific surgeries undergone

Table 4.1: Queries used in the experiments

- the timeout was set to 2 minutes.

Fig. 4.14 shows the number of correct answers returned. Specifically, for each of the 8 queries considered, the diagram reports the number of actual objects satisfying the query, the number of correct answers retrieved through vague evaluation when varying the cost threshold and the number of correct answers retrieved through exact evaluation. The number of objects satisfying the query has been computed by manually translating queries to the schemas used by the sources. A vague element is assumed to be an incorrect answer if either it contains an element describing an object that is not an answer to the query, or if it contains two elements describing different objects. The figure compares the number of correct answers with the baseline of exact evaluation. The results show that in all cases relaxed queries allow the retrieval of more answers than exact queries ($45,75\%$ more on average).

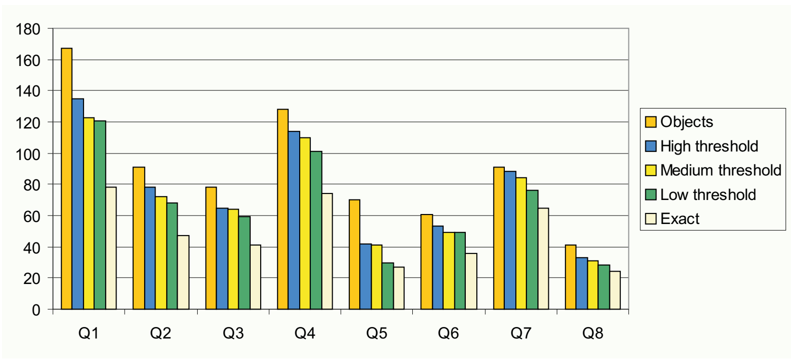


Fig. 4.14: Correct answers returned

Table 4.2 reports the average *precision* obtained, defined as the ratio between the number of correct answers and the total number of answers, and *recall*, defined as the ratio between the number of correct answers and the number of objects satisfying the query. Note that, if more than one vague element in query result refer to the same object, these vague elements are not considered separately when computing the recall. The table also reports a value that indicates the increase in the number of correct answers obtained through vague evaluation. This value, called *gain*, is defined as $ans/exAns-1$ where ans is the number of correct answers to the query, and $exAns$ is the number of correct answers to the exact version of the query.

	High threshold	Medium threshold	Low threshold	Exact
Precision	95,75%	97,29%	98,95%	99,24%
Recall	82,94%	78,54%	72,90%	53,65%
Gain	55,10%	46,43%	35,71%	–

Table 4.2: Average precision, recall, and gain

The experiments show that our approach is able to retrieve and properly combine data from heterogenous sources, providing high precision (97.31% on average) and recall ($78,13\%$ on average).

We also evaluated our proposed scoring function by looking at how the number of partial answers returned by peers is related to the score given to their synopses. Fig. 4.15 reports the percentage of partial answers retrieved as the evaluation proceeds; the X-axis reports the percentage of peers already contacted (we recall that peers are contacted in decreasing score order). We averaged the values over the 8 queries with medium threshold. The results obtained show that the routing policy gives proper priority to the peers that are more likely to contribute to the query results. Specifically, in the case depicted in the figure, almost 80% of the total number of answers are returned to the user after having accessed just 65% of the contributing peers.

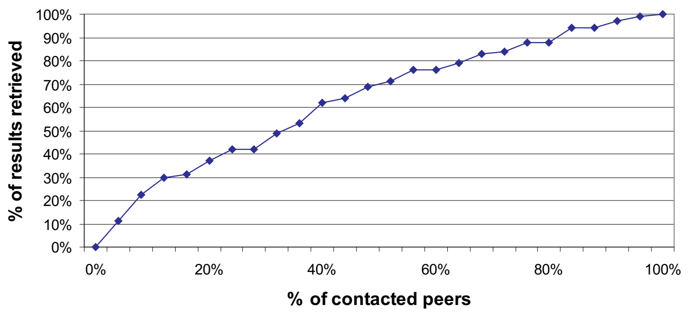


Fig. 4.15: Effect of the routing policy

5

Conclusions

Given the increasing use of networks by people, organizations and companies and, consequently, the substantial amount of resources available on the networks, the problem of sharing and retrieving data from several sources has become an important issue.

As regards companies, they are stimulated to develop applications for retrieving information spread across the networks in order to extend their own knowledge and manage data as complete as possible. In order to effectively automate the process of retrieval of information by applications, data to be shared have to be available in a "machine readable" format, such as XML [105]. Therefore, first of all it is necessary to build proper mechanisms for generating XML data, since the majority of data available in the networks are embedded in HTML pages. Systems that extract data from Web pages and convert them in a more structured format are called *wrappers*. The first step in an architecture for sharing data among sources is the design of proper wrappers that provide data in a unique format for all sources, such as XML. The second step is the retrieval of the XML data spread across the sources. As regards this aspect, it is necessary to take in account that sources store data according to different schemas, since in the general case each source has total autonomy in deciding the best organization for its own data. The classical approach is based on a global view summarizing all the data stored in the sources, allowing the user to pose queries on the global view and hiding the differences among the schemas of the various sources. This solutions requires to build and maintain mappings between source schemas and global view, thus it does not face well the network dynamism and the volatility of sources in the network.

In this thesis, we proposed proper techniques for the extraction of XML data from Web pages and for the retrieval of XML data spread across several sources without using a global view.

As the first issue, we posed the theoretical basis for extensively using the schema of the information to be extracted in both the design and evaluation of a wrapper. The main advantages of this approach range from the capability of easily guiding and controlling the extraction and integration of required data portions from HTML documents, to the specification of structured yet simple extraction rules.

We provided a clean declarative semantics for schema-based wrappers, and we introduced the notion of extraction model as a mapping between the structure and the semantics of data to be extracted. We addressed the issue of wrapper evaluation by developing an algorithm that works in polynomial time with respect to the size of a source document. This algorithm computes the preferred extraction model, which is further used to build the output extracted XML document.

The schema-based wrapping approach has been implemented into SCRAP, a visual support based wrapping system. We also presented an inductive learning method to speed up the specification of schema-based wrappers and improve their robustness with respect to structural changes occurring in source HTML documents. Empirical evidence argues that the SCRAP system is able to make wrapper generation and maintenance rapid and simple.

An the second issue, the proposed approach enables the retrieval of meaningful answers from different sources with a limited knowledge about their local schemas, by exploiting vague querying and approximate join techniques. It essentially consists in first applying transformations to the original query, then using transformed queries to retrieve partial answers and finally combining them using information about retrieved objects.

We proposed a new technique to combine partial results coming from different XML data sources, which uses approximate queries to check whether two XML elements coming from different sources refer to the same object. We provided two algorithms for computing query answers on multiple heterogeneous XML data sources: a complete algorithm working in polynomial time w.r.t. the size of the data provided by each source, and in exponential time w.r.t. the number of sources; an incomplete algorithm working in polynomial time w.r.t. both the size of the data and the number of sources. The completeness of the latter algorithm has been proved in restricted cases.

Furthermore, we characterized the complexity of the problem of answering queries on multiple heterogeneous XML data sources and we presented an experimental validation in a medical application scenario. Specifically, the proposed algorithms have been implemented in a P2P context, and queries have been done against sources containing clinical and diagnostical data.

References

1. K. Aberer, P. Cudr-Mauroux, M. Hauswirth. "The Chatty web: emergent semantics through gossiping". *Int. World Wide Web Conf.* (*WWW*), 2003.
2. S. Abiteboul. "Querying Semistructured Data". *Int. Conf. on Database Theory* (*ICDT*), 1997.
3. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan-Kaufman, 2000.
4. B. Adelberg. "NoDoSE: A Tool for Semi-Automatically Extracting Semistructured Data from Text Documents". *ACM SIGMOD Conf. on Management of Data* (*SIGMOD*), 1998.
5. S. AlKhalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. "Structural joins: Efficient matching of XML query patterns". *Int. Conf. on Data Engineering* (*ICDE*), 2002.
6. Amazon - Online store.
   http://www.amazon.com.
7. S. Amer-Yahia, S. Cho, D. Srivastava. "Tree pattern relaxation". *Int. Conf. on Extending Database Technology* (*EDBT*), 2002.
8. S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava and D. Toman. "Structure and Content Scoring for XML". *Int. Conf. on Very Large Databases* (*VLDB*), 2005.
9. ANSA - Italian news agency.
   http://www.ansa.it.
10. A. Arasu and H. Garcia-Molina. "Extracting Structured Data from Web Pages". *ACM SIGMOD Conf. on Management of Data* (*SIGMOD*), 2003.
11. C. K. Baru, A. Gupta, B. Ludscher, R. Marciano, Y. Papakonstantinou, P. Velikhov and V. Chu. "XML-based information mediation with mix". *ACM SIGMOD Conf. on Management of Data* (*SIGMOD*), 1999.
12. R. Baumgartner, S. Flesca and G. Gottlob. "Declarative Information Extraction, Web Crawling, and Recursive Wrapping with Lixto". *Int. Conf. on Logic Programming and Nonmonotonic Reasoning* (*LPNMR*), 2001.
13. R. Baumgartner, S. Flesca, and G. Gottlob. "Visual Web Information Extraction with Lixto". *Int. Conf. on Very Large Databases* (*VLDB*), 2001.
14. S. Bergamaschi, P. R. Fillottrani and G. Gelati. "The SEWASIE multi-agent system". *Int. Work. on Agents and Peer-to-Peer Computing* (*AP2PC*), 2004.

15. D. Beneventano, S. Bergamaschi. “The MOMIS methodology for integrating heterogeneous data sources”. *IFIP Congress Topical Sessions*, 2004.
16. J. Biskup and D.W. Embley. “Extracting Information from Heterogeneous Information Sources Using Ontologically Specified Target Views”. *Information Systems*, 2003.
17. BitTorrent.
http://www.bittorrent.com.
18. A. Bonifati, E. Q. Chang, T. Ho, L. V. S. Lakshmanan and R. Pottinger. “HePToX: Marrying XML and heterogeneity in your P2P databases”. *Int. Conf. on Very Large Databases* (*VLDB*), 2005.
19. A. Bonifati, U. Matrangolo, A. Cuzzocrea and M. Jain. “XPath lookup queries in P2P networks”. *ACM Int. Work. on Web Information and Data Management* (WIDM), 2004.
20. A. Bonifati, G. Mecca, A. Pappalardo, S. Raunich and G. Summa. “Schema Mapping Verification: The Spicy Way”. *Int. Conf. on Extending Database Technology* (*EDBT*), 2008.
21. A. Brüggemann-Klein and D. Wood. “One-Unambiguous Regular Languages”. *Information and Computation*, 1998.
22. P. Buneman. “Semistructured Data”. *ACM Symp. on Principles of Database Systems* (*PODS*), 1997.
23. M. E. Califf and R. J. Mooney. “Relational Learning of Pattern-Match Rules for Information Extraction”. *Conf. of the American Association for Artificial Intelligence* (*AAAI*), 1999.
24. S. D. Camillo, C. A. Heuser and R. S. Mello. “Querying heterogeneous XML sources through a conceptual schema”. *Int. Conf. on Conceptual Modeling* (*ER*), 2003.
25. S. Ceri, P. Fraternali, and S. Paraboschi. “XML: Current Developments and Future Challenges for the Database Community”. *Int. Conf. on Extending Database Technology* (*EDBT*), 2000.
26. C. X. Chen, G. A. Mihaila, S. Padmanabhan and I. Rouvellou. “Query translation scheme for heterogeneous XML data sources”. *ACM Int. Work. on Web Information and Data Management* (WIDM), 2005.
27. B. Chidlovskii. “Automatic repairing of Web Wrappers”. *ACM Int. Work. on Web Information and Data Management* (WIDM), 2001.
28. T. T. Chinenyanga, N. Kushmerick. “An expressive and efficient language for XML information retrieval”. *J. of the American Society for Information Science and Technology* (*JASIST*), 2002.
29. C. Comito, S. Patarin and D. Talia. “PARIS: A peer-to-peer architecture for large-scale semantic data integration”. *Int. Work. on Databases, Information Systems and Peer-to-Peer Computing* (*DBISP2P*), 2005.
30. V. Crescenzi, G. Mecca, and P. Merialdo. “RoadRunner: Towards Automatic Data Extraction from Large Web Sites”. *Int. Conf. on Very Large Databases* (*VLDB*), 2001.
31. E. Damiani and L. Tanca. “Blind queries to XML data”. *Int. Conf. on Database and Expert Systems Applications* (*DEXA*), 2000.
32. A. Deutsch, M. Fernandez, D. Florescu, A. Y. Levy, D. Maier, and D. Suciu. “Querying XML Data”. *IEEE Data Engineering Bull.*, v. 22, n. 3, 1999.
33. H. Do and E. Rahm. “COMA - A system for flexible combination of schema matching approaches”. *Int. Conf. on Very Large Databases* (*VLDB*), 2002.

34. A. Doan, P. Domingos, and A. Halevy. “Reconciling schemas of disparate data sources: A machine-learning approach”. *ACM SIGMOD Conf. on Management of Data (SIGMOD)*, 2001.
35. R. B. Doorenbos, O. Etzioni and D. S. Weld. “A Scalable Comparison-Shopping Agent for the World Wide Web”. *Int. Conf. on Autonomous Agents (AGENTS)*, 1997.
36. D. W. Embley, D. M. Campbell, Y. S. Jiang, S. W. Liddle, D. W. Lonsdale, Y. -K. Ng, and R. D. Smith. “Conceptual-Model-Based Data Extraction from Multiple-Record Web Pages”. *Data and Knowledge Engineering*, 1999.
37. D. W. Embley, C. Tao, and S. W. Liddl. “Automatically Extracting Ontologically Specified Data from HTML Tables of Unknown Structure”. *Int. Conf. on Conceptual Modeling (ER)*, 2002.
38. B. Fazzinga, S. Flesca and A. Pugliese. “Vague Queries on Peer-to-Peer XML Databases”. *Int. Conf. on Database and Expert Systems Applications (DEXA)*, 2007.
39. B. Fazzinga, S. Flesca and A. Pugliese. “Vague XML Querying in Peer-to-Peer Networks”. *Sistemi Evoluti per Basi di Dati (SEBD)*, 2006.
40. B. Fazzinga, S. Flesca and A. Tagarelli. “Learning Robust Web Wrappers”. *Int. Conf. on Database and Expert Systems Applications (DEXA)*, 2005.
41. S. Flesca and S. Greco. “Partially Ordered Regular Languages for Graph Queries”. *Int. Colloquium on Automata, Languages and Programming (ICALP)*, 1999.
42. S. Flesca and A. Tagarelli. “Schema-Based Web Wrapping”. *Int. Conf. on Conceptual Modeling (ER)*, 2004.
43. E. Franconi, G. M. Kuper, A. Lopatenko, I. Zaihrayeu. “Queries and Updates in the coDB Peer to Peer Database System”. *Int. Conf. on Very Large Databases (VLDB)*, 2004.
44. D. Freitag. “Information Extraction from HTML: Application of a General Machine Learning Approach”. *Conf. of the American Association for Artificial Intelligence (AAAI)*, 1998.
45. D. Freitag. “Machine Learning for Information Extraction in Informal Domains”. *Machine Learning*, 2000.
46. D. Freitag and N. Kushmerick. “Boosted Wrapper Induction”. *Conf. of the American Association for Artificial Intelligence (AAAI)*, 2000.
47. N. Fuhr and K. Grojohann. “XIRQL: An XML query language based on information retrieval concepts”. *ACM Trans. on Information Systems (TODS)*, 2004.
48. A. Fuxman, P. G. Kolaitis, R. J. Miller and W. Chiew Tan. “Peer data exchange”. *ACM Symp. on Principles of Database Systems (PODS)*, 2005.
49. H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, V. Vassalos and J. Widom. “The TSIMMIS Approach to Mediation: Data Models and Languages”. *J. of Intelligent Information Systems*, 1997.
50. M. R. Garey and David S. Johnson. “Computers and Intractability: A Guide to the Theory of NP-Completeness”. *W H Freeman & Co*, 1979.
51. G. Gottlob and C. Koch. “Monadic Datalog and the Expressive Power of Languages for Web Information Extraction”. *ACM Symp. on Principles of Database Systems (PODS)*, 2002.
52. T. J. Green, G. Karvounarakis, N. E. Taylor, O. Biton, Z. G. Ives, V. Tannen. “ORCHESTRA: facilitating collaborative data sharing”. *ACM SIGMOD Conf. on Management of Data (SIGMOD)*, 2007.

53. J-R. Gruser, L. Raschid, M. E. Vidal, and L. Bright. "Wrapper Generation for Web Accessible Data Sources". *Int. Conf. on Cooperative Information Systems* (*CoopIS*), 1998.
54. S. Guha, H. V. Jagadish, Nick Koudas, Divesh Srivastava and Ting Yu. "Integrating XML data sources using approximate joins". *ACM Trans. on Database Systems* (*TODS*), 2006.
55. L. M. Haas, D. Kossmann, E. L. Wimmers, J. Yang. "Optimizing Queries Across Diverse Data Sources". *Int. Conf. on Very Large Databases* (*VLDB*), 1997.
56. A. Y. Halevy, A. Rajaraman, J. J. Ordille. "Querying Heterogeneous Information Sources Using Source Descriptions". *Int. Conf. on Very Large Databases* (*VLDB*), 1996.
57. J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. "Extracting Semistructured Information from the Web". *ACM SIGMOD Workshop on Management of Semistructured Data*, 1997.
58. W. Han, D. Buttler and C. Pu. "Wrapping Web Data into XML". *ACM SIGMOD Record*, 2001.
59. C.-H. Hsu and M.-T. Dung. "Generating Finite-State Transducers for Semistructured Data Extraction from the Web". *Information Systems*, 1998.
60. G. Huck, P. Fankhauser, K. Aberer, and E. Neuhold. "Jedi: Extracting and Synthesizing Information from the Web". *Int. Conf. on Cooperative Information Systems* (*CoopIS*), 1998.
61. IMDB - Internet Movie Database.
http://www.imdb.com.
62. Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, D. S. Weld. "An Adaptive Query Execution System for Data Integration". *ACM SIGMOD Conf. on Management of Data* (*SIGMOD*), 1999.
63. Y. Kanza and Y. Sagiv. "Flexible Queries Over Semistructured Data". *ACM Symp. on Principles of Database Systems* (*PODS*), 2001.
64. Y. Kanza, W. Nutt and Y. Sagiv. "Queries with Incomplete Answers over Semistructured Data". *ACM Symp. on Principles of Database Systems* (*PODS*), 1999.
65. D. Kim, H. Jung, and G. Geunbae Lee. "Unsupervised Learning of mDTD Extraction Patterns for Web Text Mining". *Information Processing and Management*, 2003.
66. G. Kokkinidis and V. Christophides. "Semantic query routing and processing in P2P database systems: The ICS-FORTH SQPeer middleware". *EDBT Workshops*, 2004.
67. N. Kushmerick. "Wrapper Verification". *World Wide Web Journal*, 2000.
68. N. Kushmerick, D. S. Weld, and R. Doorenbos. "Wrapper Induction for Information Extraction". *Int. Joint Conf. on Artificial Intelligence* (*IJCAI*), 1997.
69. A. H. F. Laender, B. A. Ribeiro-Neto, and A. S. da Silva. "DEByE - Data Extraction By Example". *Data and Knowledge Engineering*, 2002.
70. A. H. F. Laender, B. A. Ribeiro-Neto, A. S. da Silva, and J. S. Teixeira. "A Brief Survey of Web Data Extraction Tools". *ACM SIGMOD Record*, 2002.
71. K. Lerman, S. N. Minton, and C. A. Knoblock. "Wrapper Maintenance: A Machine Learning Approach". *J. of Artificial Intelligence Research*, 2003.
72. A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. "Answering queries using views". *ACM Symp. on Principles of Database Systems* (*PODS*), 1995.

73. L. Liu, C. Pu, and W. Han. “XWRAP: An XML-Enabled Wrapper Construction System for Web Information Sources”. *Int. Conf. on Data Engineering* (*ICDE*), 2000.
74. J. Madhavan, P.A. Bernstein and E. Rahm. “Generic schema matching with Cupid”. *Int. Conf. on Very Large Databases* (*VLDB*), 2001.
75. F. Mandreoli, R. Martoglia and P. Tiberio. “Approximate query answering for a heterogeneous XML document base”. *Int. Conf. on Web Information Systems Engineering* (*WISE*), 2004.
76. I. Manolescu, D. Florescu and D. Kossmann. “Answering XML queries on heterogeneous data sources”. *Int. Conf. on Very Large Databases* (*VLDB*), 2001.
77. S. Melnik, H. Garcia-Molina, E. Rahm. “Similarity Flooding: A Versatile Graph Matching Algorithm and ist Application to Schema Matching”. *Int. Conf. on Data Engineering* (*ICDE*), 2002.
78. X. Meng, D. Hu, and C. Li. “Schema-Guided Wrapper Maintenance for Web Data Extraction”. *ACM Int. Work. on Web Information and Data Management* (WIDM), 2003.
79. X. Meng, H. Lu, H. Wang, and M. Gu. “Data Extraction from the Web Based on Pre-Defined Schema”. *J. of Computer Science and Technology*, 2002.
80. X. Meng, H. Lu, H. Wang and M. Gu. “SG-WRAP: A Schema-Guided Wrapper Generator”. *Int. Conf. on Data Engineering* (*ICDE*), 2002.
81. G. Miklau, D. Suciu. “Containment and equivalence for a fragment of XPath”. *J. ACM*, 2004.
82. S. Muggleton and C. Feng. “Efficient Induction of Logic Programs”. *Int. Conf. on Algorithmic Learning Theory* (*ALT*), 1990.
83. S. Muggleton and L. De Raedt. “Inductive Logic Programming: Theory and methods”. *J. of Logic Programming*, 1994.
84. I. Muslea, S. Minton, and C. Knoblock. “Hierarchical Wrapper Induction for Semistructured Information Sources”. *Autonomous Agents and Multi-Agent Systems*, 2001.
85. Napster.
http://www.napster.com.
86. W. S. Ng, B. C. Ooi, K. Tan, A. Zhou. “PeerDB: A P2P-based System for Distributed Data Sharing”. *Int. Conf. on Data Engineering* (*ICDE*), 2003.
87. E. Pitoura, S. Abiteboul, D. Pfoser, G. Samaras and M. Vazirgiannis. “DBGlobe: A service-oriented P2P system for global computing”. *ACM SIGMOD Record*, 2003.
88. N. Polyzotis and M. N. Garofalakis. “Xsketch synopses for xml data graphs”. *ACM Trans. on Database Systems* (*TODS*), 2006.
89. C. Qu, W. Nejdl. “Interacting the Edutella/JXTA Peer-to-Peer Network with Web Services”. *Symp. on Applications and the Internet* (*SAINT*), 2004.
90. R. Quinlan. “Learning Logical Definitions from Relations”. *Machine Learning*, 1990.
91. J. Raposo, A. Pan, M. Alvarez, and J. Hidalgo. “Automatically Generating Labeled Examples for Web Wrapper Maintenance”. *IEEE/WIC/ACM Int. Conf. on Web Intelligence* (*WI*), 2005.
92. P. Rodriguez-Gianolli and J. Mylopoulos. “A semantic approach to XML-based data integration”. *Int. Conf. on Conceptual Modeling* (*ER*), 2001.
93. The SAX Project. Simple API for XML Parsing.
http://www.saxproject.org/.

94. G. Saake, K.U. Sattler and S. Conrad. "Rule-based schema matching for ontology-based mediators". *J. Applied Logic*, 2005.
95. A. Sahuguet and F. Azavant. "Building Intelligent Web Applications Using Lightweight Wrappers". *Data and Knowledge Engineering*, 2001.
96. C. Sartiani, P. Manghi, G. Ghelli and G. Conforti. XPeer: A self-organizing XML P2P database system. *EDBT Workshops*, 2004.
97. T. Schlieder. "Schema-driven evaluation of approximate tree-pattern queries". *Int. Conf. on Extending Database Technology* (*EDBT*), 2002.
98. S. Soderland. "Learning Information Extraction Rules for Semistructured and Free Text". *Machine Learning*, 1999.
99. D. Suciu. "Semistructured Data and XML". *Int. Conf. on Foundations of Data Organization* (*FODO*), 1998.
100. I. Tatarinov and A. Y. Halevy. "Efficient query reformulation in peer-data management systems". *ACM SIGMOD Conf. on Management of Data* (*SIGMOD*), 2004.
101. A. Theobald, G. Weikum. "Adding Relevance to XML". *Int. Work. on the Web and Databases* (*WebDB*), 2000.
102. R. Vdovjak and G. Houben. "RDF-based architecture for semantic integration of heterogeneous information sources". *Work. on Information Integration on the Web* (*WIIW*), 2001.
103. V. Vianu. "A Web Odissey: from Codd to XML". *ACM Symp. on Principles of Database Systems* (*PODS*), 2001.
104. WordNet.
http://wordnet.princeton.edu/.
105. The World Wide Web Consortium. Extensible Markup Language (XML).
http://www.w3.org/XML.
106. The World Wide Web Consortium. Document Object Model.
http://www.w3.org/DOM/.
107. The World Wide Web Consortium. Hyper Text Markup Language (HTML.
http://www.w3.org/html.
108. The World Wide Web.
http://www.w3.org/
109. The World Wide Web Consortium. XML Schema.
http://www.w3.org/XML/Schema.
110. The World Wide Web Consortium. XML Path Language.
http://www.w3.org/TR/xpath.
111. The World Wide Web Consortium. XML Query.
http://www.w3.org/XML/Query.
112. The World Wide Web Consortium. XML Query Use Cases.
http://www.w3.org/TR/xquery-use-cases/.
113. C. Yu and L. Popa. "Constraint-based XML query rewriting for data integration". *ACM SIGMOD Conf. on Management of Data* (*SIGMOD*), 2004.
114. K. Zhang, R. Stgatman, and D. Shasha. Simple fast algorithm for the editing distance between trees and related problems. *SIAM J. on Computing*, 1989.