

UNIVERSITÀ DELLA CALABRIA



Dipartimento di ELETTRONICA,
INFORMATICA E SISTEMISTICA

UNIVERSITÀ DELLA CALABRIA

Dipartimento di Elettronica,
Informatica e Sistemistica

Dottorato di Ricerca in
Ingegneria dei Sistemi e Informatica
XIX ciclo

Tesi di Dottorato

Knowledge Management and
Extraction in XML Data

Giovanni Costa



UNIVERSITÀ DELLA CALABRIA

Dottorato di Ricerca in
Ingegneria dei Sistemi e Informatica

XIX ciclo

Tesi di Dottorato

Knowledge Management and
Extraction in XML Data

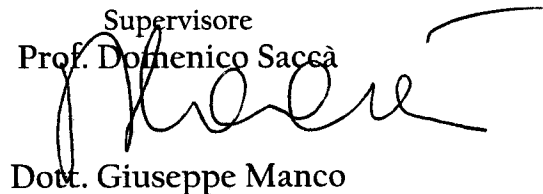
Giovanni Costa



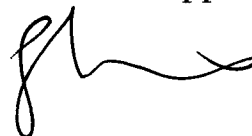
Coordinatore
Prof. Domenico Talia



Supervisore
Prof. Domenico Saccà



Dott. Giuseppe Manco



DIPARTIMENTO DI ELETTRONICA, INFORMATICA E SISTEMISTICA
Settore Scientifico Disciplinare: ING-INF/05

To Sira

Acknowledgments

This thesis is the result of an intense research activity performed at the Department of Electronics, Computer Science and Systems (DEIS) of the University of Calabria with the support of a number of persons to which I want to express my thankfulness.

I am very grateful to my advisors, whose help has been fundamental for my academic growth during Ph.D. years: Dr. Giuseppe Manco, for his precious guidance and supervision and Prof. Domenico Saccà for his encouragements and valuable advices.

Furthermore, I would like to thank my friends and colleagues of DEIS and ICAR-CNR institute with which I have shared enjoyments and disappointments during the doctoral course.

Foremost, I owe a special thank to my parents and to my brother for the continuous support and appreciation during these years.

Contents

1	Introduction	1
1.1	Background and Motivations	1
1.2	Main Contributions of the Thesis	3
1.3	Thesis Organization	5
2	An overview of XML	7
2.1	Introduction	7
2.2	XML Documents and XML Files	8
2.3	Elements, Tags, and Character Data	8
2.3.1	Tag Syntax	8
2.3.2	XML Trees	9
2.3.3	Mixed Content	10
2.4	Attributes	12
2.5	XML Names	13
2.6	EntityReferences	14
2.7	CDATA Sections	15
2.8	Comments	15
2.9	Processing Instructions	15
2.10	The XML Declaration	17
2.10.1	encoding	17
2.10.2	standalone	18
2.11	Checking Documents for Well-Formedness	18
2.12	DTD	19
2.12.1	Validation	20
2.13	XML Schema	21
2.13.1	Overview	21
3	XML compression techniques	23
3.1	Introduction	23
3.2	XML-conscious compressors	24
3.2.1	XMill	24

3.2.2	XMLZip	26
3.2.3	XMLPPM	27
3.2.4	Millau	28
3.2.5	XML-Xpress	29
3.3	Query-aware XML compressors	29
3.3.1	XGRIND	29
3.3.2	XPRESS	31
4	XQueC: XQuery processor and Compressor	33
4.1	Introduction	33
4.1.1	The XQueC system	35
4.1.2	A comparative overview of the main XML compression systems	36
4.2	Compressing XML documents in a queryable format	37
4.2.1	Compression principles	37
4.2.2	Compressed storage structures	39
4.2.3	Memory issues	41
4.3	Compression choices	42
4.3.1	Search space: possible compression configurations	43
4.3.2	Cost function: appreciating the quality of a compression configuration	44
4.3.3	Devising a suitable search strategy	45
4.4	Evaluating XML queries over compressed data	46
4.5	Implementation and experimental evaluation	49
4.5.1	Compression Factors	49
4.5.2	Query Execution Times	51
5	Data Mining: Proximity measures and cluster analysis	53
5.1	Data Mining	53
5.1.1	Introduction	53
5.1.2	Data mining as a process of knowledge discovery	55
5.2	Measures of similarity and dissimilarity	57
5.2.1	Basics	58
5.2.2	Similarity and Dissimilarity between Simple Attributes	59
5.2.3	Dissimilarities between Data Objects	60
5.2.4	Similarities between Data Objects	62
5.2.5	Examples of Proximity Measures	62
5.2.6	Selecting the right proximity Measure	67
5.3	Cluster analysis	68
5.3.1	Introduction	68
5.3.2	Different Types of Clusterings	69
5.3.3	Different Types of Clusters	71
5.3.4	Clustering Techniques	72
5.3.5	Which Clustering Algorithm	79

6	Clustering XML documents: state of the art	83
6.1	Introduction	83
6.2	Similarity measures for XML documents.....	83
6.3	Clustering approaches	85
6.4	Tree-based approach	86
6.4.1	Document representation	86
6.4.2	Tree similarity measures.....	87
6.4.3	XML specific approaches	89
6.5	Vector based approaches	93
6.5.1	Document Representation	93
6.5.2	Vector-based similarity measures	94
6.5.3	XML specific approaches	95
6.6	Other approaches.....	98
6.6.1	Time series based approach.....	98
6.6.2	Link-based similarity	99
7	<i>XREP</i>: clustering XML documents by structure	101
7.1	Introduction	101
7.1.1	A comparative overview of clustering scheme.....	101
7.2	Problem Statement	103
7.3	Mining Representatives from XML Trees	108
7.3.1	XML Tree Matching	110
7.3.2	Building a Merge Tree	112
7.3.3	Properties of Merging Process.....	112
7.3.4	Turning a merge tree into a cluster representative	114
7.4	Evaluation.....	117
8	Conclusions and further work	121
8.1	Summary.....	121
8.2	Further research.....	122
	References	125

List of Figures

2.1	A tree diagram for Example 2.2	11
3.1	Architecture of XGrind Compressor	30
3.2	The architecture of XPRESS	32
4.1	Architecture of the XQueC prototype (left); simplified summary of the XMark XML documents (right)	37
4.2	An example of encoding in ALM	39
4.3	Storage structures in the XQueC repository	41
4.4	Accesses to containers in case of XMark's Q14 with <i>descendant-or-self</i> axis in XPress/XGrind versus XQueC	42
4.5	Query execution plan for XMark's Q9	47
4.6	Average compression factor for Shakespeare, WashingtonCourse and Baseball data sets	50
4.7	Average compression factor (%) for XMark synthetic data sets	51
4.8	Comparative execution times between us and Optimized Galax	52
5.1	Data mining - searching for knowledge (interesting patterns) in data	54
5.2	Data mining as a process of knowledge discovery	56
5.3	Geometric illustration of the cosine measure	65
5.4	Cluster analysis of points in a 2D-space	69
5.5	Distances for a single-link and a complete-link clustering algorithm	75
6.1	Structural granularities in an XML document	84
6.2	External quality measures	86
6.3	XML documents containing recipes	87
6.4	Tree representation of XML documents containing recipes	87
6.5	Mapping	89

6.6	Tree edit distance algorithms (* marked operations are restricted to leaves)	90
6.7	Contained in relationship	91
6.8	(a) The structure of a document, (b) its s-graph, (c) its structural summary	91
6.9	Two simple s-graphs	92
6.10	3-dimensional Boolean matrix	96
6.11	(a) A tree document, (b) its full binary tree, and (c) the binary branch vector	98
7.1	The <i>XRep</i> algorithm for clustering XML documents.	104
7.2	Tree XML document trees, t_1 (a), t_2 (b) and t_3 (c), which make problematic the choice between deleting node d or relabelling node f	105
7.3	Tree XML document trees, t_1 (a), t_2 (b) and t_3 (c), that, despite considerable structural differences, are at a same tree edit distance.	105
7.4	Two XML document trees, t_1 (a) and t_2 (b), with considerable structural differences and, notwithstanding, characterized by low degrees of cosine and Jaccard dissimilarity.	106
7.5	(a) Strong and (b) multiple matching, and (c) their trees.	108
7.6	The algorithm for the computation of an XML cluster representative.	109
7.7	Data structures for the construction of an optimal matching tree.	111
7.8	Lower-bound (optimal matching tree) (a), upper-bound (merge tree) (b), and optimal representative tree (c) relative to the trees of Fig.7.7(a).	112
7.9	The process for building a merge tree	116

Introduction

1.1 Background and Motivations

Since World Wide Web Consortium (W3C) has released its specification on February 1998, XML (eXtensible Markup Language) has kindled great interest either in scientific community either in computer industry until to became the de facto standard format for electronic data structuring, storage and exchange. It has been enthusiastically adopted in a number of application fields, including information retrieval and multimedia systems, databases, e-business applications, geographical information systems and many others. Many expectations are surrounding XML, thus yielding a substantial growth of data available.

XML is a general-purpose markup language for creating special-purpose markup languages, capable of describing many different kinds of data. In other words, XML is a way of describing data. An XML file can contain the data too, as in a database. XML provides a text-based means to describe and apply a tree-based structure to information. At its base level, all information manifests as text, interspersed with markups that indicate a structuring into a hierarchy of character data, container-like elements, and attributes of those elements. It is a simplified subset of Standard Generalized Markup Language (SGML). Its primary purpose is to facilitate the sharing of data across different systems, particularly systems connected via the Internet. Languages based on XML (example are Geography Markup Language (GML), RDF/XML, RSS, Atom, MathML, XHTML, SVG, XUL, EAD, Klip and MusicXML) are defined in a formal way, allowing programs to modify and validate documents in these languages without prior knowledge of their particular form.

The ubiquity of text file authoring software (word processors) facilitates rapid XML document authoring and maintenance, whereas prior to the advent of XML, very few data description languages that were general-purpose, Internet protocol-friendly, and very easy to learn and author. In fact, most data interchange formats were proprietary, special-purpose, “binary” formats (based foremost on bit sequences rather than characters) and they could not

be easily shared by different software applications or across different computing platforms, much less authored and maintained in common text editors.

XML is well-suited for data transfer thanks to the following features:

- it is both human- and machine-readable format;
- it has support for Unicode, allowing almost any information in any written human language to be communicated;
- the ability to represent the most general computer science data structures: records, lists and trees;
- the self-documenting format that describes structure and field names as well as specific values;
- the strict syntax and parsing requirements that allow the necessary parsing algorithms to remain simple, efficient, and consistent.

XML is also heavily used as a format for document storage and processing, both online and offline, and offers several benefits:

- its robust, logically-verifiable format is based on international standards;
- the hierarchical structure is suitable for most (but not all) types of documents;
- it manifests as plain text files, unencumbered by licenses or restrictions;
- it is platform-independent, thus relatively immune to changes in technology;
- its predecessor, SGML, has been in use since 1986, so there is extensive experience and software available.

Close to the numerous merits, the XML standard has some limits. Parsers should be designed to recurse arbitrarily nested data structures and must perform additional checks to detect improperly formatted or differently ordered syntax or data. This causes a significant overhead for most basic uses of XML, particularly where resources may be scarce - for example in embedded systems. Furthermore, additional security considerations arise when XML input is fed from untrustworthy sources and resource exhaustion or stack overflows are possible. Some consider the syntax to contain a number of obscure, unnecessary features born of its legacy of SGML compatibility.

The basic parsing requirements do not support a very wide array of data types so interpretation sometimes involves additional work in order to process the desired data from a document. There is no provision in XML, for example, for mandating that “3.14159” is a floating-point number rather than a seven-character string (although some XML schema languages add this functionality). But the most problematic issue with XML is that it is text-based and verbose by its design (the XML standard explicitly states that terseness in XML markup is of minimal importance). Its syntax is wordy and redundant. This can hurt human readability and application efficiency, and yields higher storage costs. It can also make XML difficult to apply in cases where bandwidth is limited. This is particularly true for multimedia applications running on cell phones and PDAs which want to use XML to describe images

and video. As a consequence, data represented in the XML format, generally require more space than they would need if they were represented in a owner format. This happens, mainly, because the scheme (represented by the tags) must be repeated for every record unlike what happens, for example, in the relational table in which the scheme is unique. Also being, therefore, XML the standard format for the representation and exchange of data, its use behaves a waste of resources (space of memorization and band of transmission to quote the most important). As a result, the amount of information that has to be transmitted, processed and stored is often substantially larger in comparison to other data formats. This can be a serious problem in many occasions, since the data has to be transmitted quickly and stored compactly. Therefore, it is natural to investigate the use of syntactic and semantic models for the compression of such data.

In addition, with the continuous and heterogeneous growth in XML data sources, the ability to manage collections of XML documents and discover knowledge from them for decision support becomes increasingly important. Mining of XML documents significantly differs from structured data mining and text mining. XML allows the representation of semi-structured and hierarchal data containing not only the values of individual items but also the relationships between data items. Element tags and their nesting therein dictate the structure of an XML document. Due to the inherent flexibility of XML, in both structure and semantics, discovering knowledge from XML data is faced with new challenges as well as benefits. Mining of structure provides new insights and means into the process of knowledge discovery. The results of mining XML data have several interesting applications related to the management of Web data. For example, structural analysis of Web sites can benefit from the identification of similar documents, conforming to a particular schema, which can serve as the input for wrappers working on structurally similar Web pages. Also, query processing in semistructured data can take advantage from the re-organization of documents on the basis of their structure. Grouping semistructured documents according to their structural homogeneity can help in devising indexing techniques for such documents, thus improving the construction of query plans.

1.2 Main Contributions of the Thesis

This thesis is focused on management and extraction of knowledge from XML documents and proposes a number of novel contributions aiming at easing query optimization in the fields of XML compression and clustering of XML documents.

As pointed out above, XML documents have an inherent textual nature due to repeated tags and to PCDATA content. Therefore, they lead themselves naturally to compression. Once the compressed documents are produced, how-

ever, one would like to still query them under a compressed form as much as possible.

Compression of XML documents has been largely dealt with in literature. Different techniques have been proposed in order to improve the efficacy and the efficiency of the approaches. In this thesis, we propose an adequate compression of the values found in an XML document, coupled with a compact storage model for all parts of the document. We present the XQueC system that increases the compression benefits by adapting its compression strategy to the data and query workload, based on a suitable cost model. By doing data fragmentation and compression, XQueC indirectly targets the problem of main-memory XQuery evaluation, which has attracted the attention of the community[22, 96].

XQueC addresses the problem of fitting into memory a narrowed version of the tree of tags (which is however a small percentage of the overall document) in a two-fold way. Former, in order to diminish its footprint, it applies powerful compression to the XML documents. The compression algorithms that we use allow to evaluate most predicates directly on the compressed values. Thus, decompression is often necessary only at the end of the query evaluation. Later, the XQueC storage model includes lightweight access support structures for the data itself, providing thus efficient primitives for query evaluation.

The advantages of processing queries in the compressed domain are several: first, in a traditional query setting, access to small chunks of data may lead to less disk I/Os and reduce the query processing time; second, the memory and computation efforts in processing compressed data can be dramatically lower than those for uncompressed ones, thus even low-battery mobile devices can afford them; third, the possibility of obtaining compressed query results allows to spare network bandwidth when sending these results to a remote location, in the spirit of [32].

On the other hand, the problem of comparing semistructured documents has been recently investigated from different perspectives [146, 30, 35, 17]. Recent studies have also proposed techniques for clustering XML documents. XML document clustering is realized through algorithms that rely on the similarity between two documents computed exploiting a distance metric. The algorithms should guarantee that documents in the same cluster have a high similarity degree (low distance), whereas documents in different clusters have a low similarity degree (high distance).

In this thesis we propose a novel methodology for clustering XML documents by structure, which is based on the notion of XML *cluster representative*. A cluster representative is a prototype XML document subsuming the most relevant structural features of the documents within a cluster. The notion of cluster prototype is crucial in most significant application domains, such as wrapper induction, similarity search, and query optimization. The core of our approach is the observation that a suitable cluster prototype is represented by the result of a proper overlapping among all the documents contained within the cluster. Actually, the resulting tree has the main advan-

tage of retaining the specifics of the enclosed documents, while guaranteeing a compact representation for them. This eventually makes the proposed notion of cluster representative extremely profitable in the envisaged applications: in particular, as a summary for the cluster, a representative highlights common subparts in the enclosed documents, and can avoid expensive similarity searches against individual documents within the cluster. The proposed notion of cluster representative is computed by resorting to the notions of matching and merging among XML trees. Given a set of XML documents, we aim at finding the best overlap between them. To this purpose, we detect an initial tree structure resulting from the optimal matching between the trees, and enrich the matching tree by including the remaining uncommon substructures. Finally, we propose a hierarchical clustering algorithm, which exploits the devised notion of representative in order to group homogeneous XML documents.

1.3 Thesis Organization

The remainder of the thesis is structured as follows.

Chapter 2 presents an overview of XML. In particular, we explain what XML is and why XML has kindled this strong interest in the scientific and industry world. The syntax by which XML documents are structured as hierarchies of information is defined with the help of different examples. The concepts of *well-formed* and *valid* document are discussed and, finally, a brief presentation of *DTD* and *XML Schema* is given.

Chapter 3 is centred on compression of XML documents. We investigate the main motivations that lead to compression of XML data and analyze the different types of approaches presented in literature. Particular attention is focused on features that differentiate generic compressors from the so called *conscious* XML compressors: they make use of semantic information in order to obtain better compression factors. Also, we pay attention on the design of specific systems that are intended to work in conjunction with a query processor.

In chapter 4, the XQueC system is proposed. XQueC addresses the problem of compressing XML data in such a way as to allow efficient XQuery evaluation in the compressed domain. It is a XQuery processor on compressed data able to achieve a good trade-off among data compression factors, queryability and XQuery expressibility. Moreover, fragmentation and storage model for the compressed XML documents used by XQueC are detailed. Special attention is reserved to the presentation of the order preserving text compression algorithm used by XQueC in order to query data in compressed domain. In the last part of the chapter, a number of experimental tests are exhibited, showing the efficacy of the system.

Chapter 5 deals with knowledge discovery in XML data. We present data mining tasks as a step of more general knowledge discovery process and in-

roduce some basic concepts about proximity measures that can be used to compare objects. The final part of the chapter is dedicated to the cluster analysis. We detail the principal clustering techniques used to group similar objects highlighting what are the main factors to be considered in their selection.

Chapter 6 is an overview of the main XML document clustering approaches available from literature. Great emphasis is placed on the research efforts for developing similarity measures for clustering XML documents relying on their content, structure, and links. Approaches are presented depending on the adopted representation of documents (vector-based, tree-based, seldom graph and alternative representations).

Chapter 7 presents our methodology for clustering XML documents by structure, which is based on the notion of *XML cluster representative*. The proposed notion of cluster representative relies on the notions of XML tree *matching* and *merging*. The last part of the chapter presents experimental evaluation performed on both synthetic and real data that states the effectiveness of our approach.

Finally, chapter 8 draws some conclusions and highlights some still open issues that are worth considering in further investigation.

An overview of XML

2.1 Introduction

XML, the Extensible Markup Language, is a W3C-endorsed standard for document markup. It defines a generic syntax used to mark up data with simple, human-readable tags. It provides a standard format for computer documents. This format is flexible enough to be customized for domains as diverse as web sites, electronic data interchange, vector graphics, genealogy, real-estate listings, object serialization, remote procedure calls, voice-mail systems, and more.

By leaving the names, allowable hierarchy, and meanings of the elements and attributes open and definable by a customizable schema, XML provides a syntactic foundation for the creation of custom, XML-based markup languages. The general syntax of such languages is rigid—documents must adhere to the general rules of XML, assuring that all XML-aware software can at least read (parse) and understand the relative arrangement of information within them. The schema merely supplements the syntax rules with a set of constraints. Schemas typically restrict element and attribute names and their allowable containment hierarchies, such as only allowing an element named “birthday” to contain an element named “month” and an element named “day”, each of which has to contain only character data. The constraints in a schema may also include data type assignments that affect how information is processed; for example, the “month” element’s character data may be defined as being a month according to a particular schema language’s conventions, perhaps meaning that it must not only be formatted a certain way, but also must not be processed as if it were some other type of data.

In this way, XML contrasts with HTML, which has an inflexible, single-purpose vocabulary of elements and attributes that, in general, cannot be repurposed. With XML, it is much easier to write software that accesses the document’s information, since the data structures are expressed in a formal, relatively simple way.

XML makes no prohibitions on how it is used. Although XML is fundamentally text-based, software quickly emerged to abstract it into other, richer formats, largely through the use of datatype-oriented schemas and object-oriented programming paradigms (in which the document is manipulated as an object). Such software might treat XML as serialized text only when it needs to transmit data over a network, and some software doesn't even do that much. Such uses have led to "binary XML", the relaxed restrictions of XML 1.1, and other proposals that run counter to XML's original spirit and thus garner an amount of criticism.

2.2 XML Documents and XML Files

An XML document contains text, never binary data. It can be opened with any program that knows how to read a text file. The example 2.1 is close to the simplest XML document imaginable. Nonetheless, it is a well-formed XML document. XML parsers can read it and understand it (at least as far as a computer program can be said to understand anything).

Example 2.1. A very simple yet complete XML document

```
<person>
  Alan Turing
</person>
```

2.3 Elements, Tags, and Character Data

The document of example 2.1 is composed of a single element named person. The element is delimited by the start-tag `<person>` and the end-tag `</person>`. Everything between the start-tag and the end-tag of the element (exclusive) is called the element's content. The content of this element is the text string:

```
Alan Turing
```

The whitespace is part of the content, though many applications will choose to ignore it. `<person>` and `</person>` are markup. The string "Alan Turing" and its surrounding whitespace are character data. The tag is the most common form of markup in an XML document, but there are other kinds.

2.3.1 Tag Syntax

XML tags look superficially like HTML tags. Start-tags begin with `<` and end-tags begin with `</`. Both of these are followed by the name of the element and are closed by `>`. However, unlike HTML tags, it is allowed to make up new XML tags. To describe a person, use `<person>` and `</person>` tags. To

describe a calendar, use `<calendar>` and `</calendar>` tags. The names of the tags generally reflect the type of content inside the element, not how that content will be formatted.

Empty elements

There's also a special syntax for empty elements, i.e., elements that have no content. Such an element can be represented by a single empty-element tag that begins with `<` but ends with `/>`. For instance, in XHTML, an XMLized reformulation of standard HTML, the line-break and horizontal-rule elements are written as `
` and `<hr/>` instead of `
` and `<hr>`. These are exactly equivalent to `
</br>` and `<hr></hr>`, however. Which form to use for empty elements is completely up to authors. However, in XML and XHTML (unlike HTML) the use of only the start-tag (for instance `
` or `<hr>`) without using the matching the end-tag is not permitted. That would be a well-formedness error.

Case sensitivity

XML, unlike HTML, is case sensitive. `<Person>` is not the same as `<PERSON>` is not the same as `<person>`. If an element is opened by a `<person>` tag, it can't be closed by a `</PERSON>` tag. Authors are free to use upper or lowercase or both as they choose. They just have to be consistent within any one element.

2.3.2 XML Trees

Example 2.2 shows a more complicated XML document. The example is a person element that contains more information suitably marked up to show its meaning.

Example 2.2. A more complex XML document describing a person

```
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <profession>computer scientist</profession>
  <profession>mathematician</profession>
  <profession>cryptographer</profession>
</person>
```

Parents and children

XML document of example 2.2 is still composed of one person element. However, now this element doesn't merely contain undifferentiated character data. It contains four child elements: a name element and three profession elements. The name element contains two child elements of its own, `first_name` and `last_name`.

The person element is called the parent of the name element and the three profession elements. The name element is the parent of the `first_name` and `last_name` elements. The name element and the three profession elements are sometimes called each other's siblings. The `first_name` and `last_name` elements are also siblings.

As in human society, any one parent may have multiple children. However, unlike human society, XML gives each child exactly one parent, not two or more. Each element (with the exception of root element) has exactly one parent element. That is, it is completely enclosed by another element. If an element's start-tag is inside some element, then its end-tag must also be inside that element. Overlapping tags, as in

```
<strong>
  <em>
    this common example from HTML
  </strong>
</em>
```

are prohibited in XML. Since the *em* element begins inside the *strong* element, it must also finish inside the *strong* element.

The root element

Every XML document has one element that does not have a parent. This is the first element in the document and the element that contains all other elements. In example 2.1 and 2.2, the person element filled this role. It is called the root element of the document. It is also sometimes called the document element. Every well-formed XML document has exactly one root element. Since elements may not overlap, and since all elements except the root have exactly one parent, XML documents form a data structure programmers call a tree. Figure 2.1 diagrams this relationship for example 2.2. Each gray box represents an element. Each black box represents character data. Each arrow represents a containment relationship.

2.3.3 Mixed Content

In example 2.2, the contents of the `first_name`, `last_name`, and profession elements were character data, that is, text that does not contain any tags. The contents of the person and name elements were child elements and some

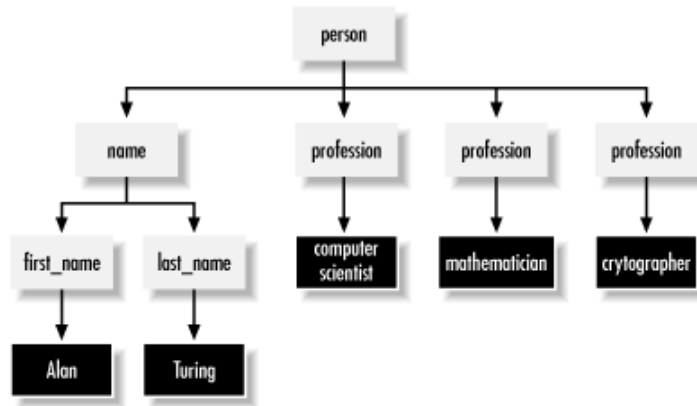


Fig. 2.1. A tree diagram for Example 2.2

whitespace that most applications will ignore. This dichotomy between elements that contain only character data and elements that contain only child elements (and possibly a little whitespace) is common in documents that are data oriented. However, XML can also be used for more free-form, narrative documents such as business reports, magazine articles, student essays, short stories, web pages, and so forth, as shown by example 2.3.

Example 2.3. A narrative-organized XML document

```

<biography>
  <name><first_name>Alan</first_name> <last_name>Turing
  </last_name> </name> was one of the first people to
  truly deserve the name <emphasize>computer scientist
  </emphasize>. Although his contributions to the field
  are too numerous to list, his best-known are the eponymous
  <emphasize> Turing Test</emphasize> and <emphasize>Turing
  Machine</emphasize>.

  <definition>The <term>Turing Test</term> is to this day the
  standard test for determining whether a computer is truly
  intelligent. This test has yet to be passed. </definition>

  <definition>The <term>Turing Machine</term> is an abstract
  finite state automaton with infinite memory that can be
  proven equivalent to any any other finite state automaton
  with arbitrarily large memory.
  Thus what is true for a Turing machine is true for all
  equivalent machines no matter how implemented.
  </definition>

```



```

<name><last_name>Turing</last_name></name> was also an
accomplished <profession>mathematician</profession> and
<profession>cryptographer</profession>. His assistance
was crucial in helping the Allies decode the German Enigma
machine. He committed suicide on <date><month>June</month>
<day>7</day>, <year>1954</year></date> after being
convicted of homosexuality and forced to take female
hormone injections.
</biography>

```

The root element of this document is biography. The biography contains name, definition, profession, and emphasize child elements. It also contains a lot of raw character data. Some of these elements such as last_name and profession only contain character data. Others such as name contain only child elements. Still others such as definition contain both character data and child elements. These elements are said to contain mixed content. Mixed content is common in XML documents containing articles, essays, stories, books, novels, reports, web pages, and anything else that's organized as a written narrative. Mixed content is less common and harder to work with in computer-generated and processed XML documents used for purposes such as database exchange, object serialization, persistent file formats, and so on. One of the strengths of XML is the ease with which it can be adapted to the very different requirements of human-authored and computer-generated documents.

2.4 Attributes

XML elements can have attributes. An attribute is a name-value pair attached to the element's start-tag. Names are separated from values by an equals sign and optional whitespace. Values are enclosed in single or double quotation marks. For example, this person element has a born attribute with the value 1912-06-23 and a died attribute with the value 1954-06-07:

```

<person born="1912-06-23" died="1954-06-07">
  Alan Turing
</person>

```

This next element is exactly the same as far as an XML parser is concerned. It simply uses single quotes instead of double quotes, puts some extra whitespace around the equals signs, and reorders the attributes.

```

<person died = '1954-06-07'  born = '1912-06-23'>
  Alan Turing
</person>

```

The whitespace around the equals signs is purely a matter of personal aesthetics. The single quotes may be useful in cases where the attribute value itself contains a double quote. Attribute order is not significant.

Example 2.4 shows how attributes might be used to encode much of the same information given in the data-oriented document of example 2.2.

Example 2.4. An XML document that describes a person using attributes

```
<person>
  <name first="Alan" last="Turing"/>
  <profession value="computer scientist"/>
  <profession value="mathematician"/>
  <profession value="cryptographer"/>
</person>
```

This raises the question of when and whether one should use child elements or attributes to hold information. This is a subject of heated debate. Some computer scientists maintain that attributes are for metadata about the element while elements are for the information itself. Others point out that it's not always so obvious what's data and what's metadata. Indeed, the answer may depend on where the information is put to use.

What's undisputed is that each element may have no more than one attribute with a given name. That's unlikely to be a problem for a birth date or a death date; it would be an issue for a profession, name, address, or anything else of which an element might plausibly have more than one. Furthermore, attributes are quite limited in structure. The value of the attribute is simply a text string. The division of a date into a year, month, and day with hyphens in the previous example is at the limits of the substructure that can reasonably be encoded in an attribute. Consequently, an element-based structure is a lot more flexible and extensible. Nonetheless, attributes are certainly more convenient in some applications.

2.5 XML Names

The XML specification can be quite legalistic and picky at times. Nonetheless, it tries to be efficient where possible. One way it does that is by reusing the same rules for different items where possible. For example, the rules for XML element names are also the rules for XML attribute names, as well as for the names of several less common constructs. Generally, these are referred to simply as XML names.

Element and other XML names may contain essentially any alphanumeric character. This includes the standard English letters A through Z and a through z as well as the digits 0 through 9. XML names may also include non-English letters, numbers, and ideograms. XML names may not contain other punctuation characters such as quotation marks, apostrophes, dollar

signs, carets, percent symbols, and semicolons. The colon is allowed, but its use is reserved for namespaces. XML names may not contain whitespace of any kind, whether a space, a carriage return, a line feed, a non-breaking space, and so forth. Finally, all names beginning with the string XML (in any combination of case) are reserved for standardization in W3C XML-related specifications. XML names may only start with letters, ideograms, and the underscore character. They may not start with a number, hyphen, or period. There is no limit to the length of an element or other XML name

2.6 EntityReferences

The character data inside an element may not contain a raw unescaped opening angle bracket (<). This character is always interpreted as beginning a tag. If it need to use this character in text, it is possible to escape it using the < entity reference. When a parser reads the document, it will replace the < entity reference with the actual < character. However, it will not confuse < with the start of a tag. For example:

```
<SCRIPT LANGUAGE="JavaScript">
  if (location.host.toLowerCase().indexOf("cafeconleche")<0)
  {
    location.href="http://www.cafeconleche.org/";
  }
</SCRIPT>
```

The character data inside an element may not contain a raw unescaped ampersand (&) either. This is always interpreted as beginning an entity or character reference. However, the ampersand may be escaped using the & entity reference like this:

Entity references such as & and < are considered to be markup. When an application parses an XML document, it replaces this particular markup with the actual characters to which the entity reference refers.

XML predefines exactly five entity references. These are:

- < the less-than sign; a.k.a. the opening angle bracket (<)
- & the ampersand (&)
- > the greater-than sign; a.k.a. the closing angle bracket (>)
- " the straight, double quotation marks (")
- ' the apostrophe; a.k.a. the straight single quote (')

In addition to the five predefined entity references, it is possible to define others in the document type definition.

2.7 CDATA Sections

When an XML document includes samples of XML or HTML source code, the `<` and `&` characters in those samples must be encoded as `<` and `&`. The more sections of literal code a document includes and the longer they are, the more tedious this encoding becomes. Instead it can enclose each sample of literal code in a CDATA section. A CDATA section is set off by a `<![CDATA[` and `]]>`. Everything between the `<![CDATA[` and the `]]>` is treated as raw character data. Less-than signs don't begin. Ampersands don't start entity references. Everything is simply character data, not markup.

The only thing that can not appear in a CDATA section is the CDATA section end delimiter `]]>`.

CDATA sections exist for the convenience of human authors, not for programs. Parsers are not required to tell you whether a particular block of text came from a CDATA section, from normal character data, or from character data that contained entity references such as `<` and `&`. By the time you get access to the data, these differences will have been washed away.

2.8 Comments

XML documents can be commented so that coauthors can leave notes for each other and themselves, documenting why they've done what they've done or items that remain to be done. XML comments are syntactically similar to HTML comments. Just as in HTML, they begin with `<!--` and end with the first occurrence of `-->`. For example:

```
<!-- I need to verify and update these links. -->
```

The double hyphen `--` should not appear anywhere inside the comment until the closing `-->`. In particular, a three hyphen close like `---<` is specifically forbidden.

Comments may appear anywhere in the character data of a document. They may also appear before or after the root element. (Comments are not elements, so this does not violate the tree structure or the one-root element rules for XML.) However, comments may not appear inside a tag or inside another comment.

Applications that read and process XML documents may or may not pass along information included in comments. They are certainly free to drop them out if they choose. Comments are strictly for making the raw source code of an XML document more legible to human readers.

2.9 Processing Instructions

In HTML, comments are sometimes abused to support nonstandard extensions. For instance, the contents of the `script` element are sometimes enclosed

in a comment to protect it from display by a nonscript-aware browser. The Apache web server parses comments in *.shtml* files to recognize server side includes. Unfortunately, these documents may not survive being passed through various HTML editors and processors with their comments and associated semantics intact. Worse yet, it's possible for an innocent comment to be misconstrued as input to the application.

XML provides the processing instruction as an alternative means of passing information to particular applications that may read the document. A processing instruction begins with `<?` and ends with `?>`. Immediately following the `<?` is an XML name called the target, possibly the name of the application for which this processing instruction is intended or possibly just an identifier for this particular processing instruction. The rest of the processing instruction contains text in a format appropriate for the applications for which the instruction is intended.

For example, in HTML a robots META tag is used to tell search-engine and other robots whether and how they should index a page. The following processing instruction has been proposed as an equivalent for XML documents:

```
<?robots index="yes" follow="no"?>
```

The target of this processing instruction is robots. The syntax of this particular processing instruction is two pseudoattributes, one named `index` and one named `follow`, whose values are either `yes` or `no`. The semantics of this particular processing instruction are that if the `index` attribute has the value `yes`, then search-engine robots should index this page. If `index` has the value `no`, then it won't be. Similarly, if `follow` has the value `yes`, then links from this document will be followed.

Other processing instructions may have totally different syntaxes and semantics. For instance, processing instructions can contain an effectively unlimited amount of text. PHP includes large programs in processing instructions. For example:

```
<?php
mysql_connect("database.unc.edu", "clerk", "password");
$result = mysql("HR", "SELECT LastName, FirstName FROM
  Employees ORDER BY LastName, FirstName");
$i = 0;
while ($i < mysql_numrows ($result)) {
  $fields = mysql_fetch_row($result);
  echo "<person>$fields[1] $fields[0] </person>\r\n";
  $i++;
}
mysql_close( );
?>
```

Processing instructions are markup, but they're not elements. Consequently, like comments, processing instructions may appear anywhere in an XML document outside of a tag, including before or after the root element. The most common processing instruction, *xml-stylesheet*, is used to attach stylesheets to documents. It always appears before the root element, as the following example demonstrates.

```
<?xml-stylesheet href="person.css" type="text/css"?>
<person>
  Alan Turing
</person>
```

In this example, the *xml-stylesheet* processing instruction tells browsers to apply the CSS stylesheet *person.css* to this document before showing it to the reader. The processing instruction names xml, XML, XmlL, etc., in any combination of case, are forbidden to avoid confusion with the XML declaration.

2.10 The XML Declaration

XML documents should (but do not have to) begin with an XML declaration. The XML declaration looks like a processing instruction with the name xml and version, standalone, and encoding attributes. Technically, it's not a processing instruction though, just the XML declaration; nothing more, nothing less. The following example demonstrates.

```
<?xml version="1.0" encoding="ASCII" standalone="yes"?>
<person>
  Alan Turing
</person>
```

XML documents do not have to have an XML declaration. However, if an XML document does have an XML declaration, then that declaration must be the first thing in the document. It must not be preceded by any comments, whitespace, processing instructions, and so forth. The reason is that an XML parser uses the first five characters (<?xml) to make some reasonable guesses about the encoding, such as whether the document uses a single byte or multi-byte character set. The only thing that may precede the XML declaration is an invisible Unicode byte-order mark.

2.10.1 encoding

By default XML documents are assumed to be encoded in the UTF-8 variable-length encoding of the Unicode character set. This is a strict superset of ASCII, so pure ASCII text files are also UTF-8 documents. However, most XML processors, especially those written in Java, can handle a much broader

range of character sets. To use a specific character set it is necessary to tell the parser which character encoding the document uses. Preferably this is done through meta information, stored in the file system or provided by the server. However, not all systems provide character-set metadata so XML also allows documents to specify their own character set with an encoding declaration inside the XML declaration. The following example shows how to indicate that a document was written in the ISO-8859-1 (Latin-1) character set that includes letters like `ñ` and `ü` needed for many non-English Western European languages.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<person>
  Erwin Schrdinger
</person>
```

The encoding attribute is optional in an XML declaration. If it is omitted and no metadata is available, then the Unicode character set is assumed. The parser may use the first several bytes of the file to try to guess which encoding of Unicode is in use. If metadata is available and it conflicts with the encoding declaration, then the encoding specified by the metadata wins. For example, if an HTTP header says a document is encoded in ASCII but the encoding declaration says it's encoded in UTF-8, then the parser will pick ASCII.

2.10.2 standalone

If the standalone attribute has the value `no`, then an application may be required to read an external DTD (that is a DTD in a file other than the one it's reading now) to determine the proper values for parts of the document. For instance, a DTD may provide default values for attributes that a parser is required to report even though they are not actually present in the document.

Documents that do not have DTDs, can have the value `yes` for the standalone attribute. Documents that do have DTDs can also have the value `yes` for the standalone attribute if the DTD does not in any way change the content of the document or if the DTD is purely internal.

The standalone attribute is optional in an XML declaration. If it is omitted, then the value `no` is assumed.

2.11 Checking Documents for Well-Formedness

Every XML document, without exception, must be well-formed. This means it must adhere to a number of rules, including the following:

- Every start-tag must have a matching end-tag.
- Elements may nest, but may not overlap.
- There must be exactly one root element.

- Attribute values must be quoted.
- An element may not have two attributes with the same name.
- Comments and processing instructions may not appear inside tags.
- No unescaped < or & signs may occur in the character data of an element or attribute.

This is not an exhaustive list. There are many, many ways a document can be malformed. Whether the error is small or large, likely or unlikely, an XML parser reading a document is required to report it. It may or may not report multiple well-formedness errors it detects in the document. However, the parser is not allowed to try to fix the document and make a best-faith effort of providing what it thinks the author really meant. It can't fill in missing quotes around attribute values, insert an omitted end-tag, or ignore the comment that's inside a start-tag. The parser is required to return an error. The objective here is to avoid the bug-for-bug compatibility wars that plagued early web browsers and continue to this day. Consequently, before you publish an XML document, whether that document is a web page, input to a database, or something else, you'll want to check it for well-formedness.

2.12 DTD

While XML is extremely flexible, not all the programs that read particular XML documents are so flexible. Many programs can work with only some XML applications but not others. And within a particular XML application, it's often important to ensure that a given document indeed adheres to the rules of that XML application. For instance, in XHTML, *li* elements should only be children of *ul* or *ol* elements. Browsers may not know what to do with them, or may act inconsistently, if *li* elements appear in the middle of a *blockquote* or *p* element.

The solution to this dilemma is a document type definition (DTD). DTDs are written in a formal syntax that explains precisely which elements and entities may appear where in the document and what the elements' contents and attributes are. A DTD can make statements such as "A *ul* element only contains *li* elements" or "Every *employee* element must have a `social_security_number` attribute". Different XML applications can use different DTDs to specify what they do and do not allow.

A validating parser compares a document to its DTD and lists any places where the document differs from the constraints specified in the DTD. The program can then decide what it wants to do about any violations. Some programs may reject the document. Others may try to fix the document or reject just the invalid element. Validation is an optional step in processing XML. A validity error is not necessarily a fatal error like a well-formedness error, though some applications may choose to treat it as one.

2.12.1 Validation

A valid document includes a document type declaration that identifies the DTD the document satisfies. The DTD lists all the elements, attributes, and entities the document uses and the contexts in which it uses them. The DTD may list items the document does not use as well. Validity operates on the principle that everything not permitted is forbidden. Everything in the document must match a declaration in the DTD. If a document has a document type declaration and the document satisfies the DTD that the document type declaration indicates, then the document is said to be valid. If it does not, it is said to be invalid.

There are many things the DTD does not say. In particular, it does not say the following:

- What the root element of the document is
- How many of instances of each kind of element appear in the document
- What the character data inside the elements looks like
- The semantic meaning of an element; for instance, whether it contains a date or a person's name

DTDs allow you to place some constraints on the form an XML document takes, but there can be quite a bit of flexibility within those limits. A DTD never says anything about the length, structure, meaning, allowed values, or other aspects of the text content of an element.

Validity is optional. A parser reading an XML document may or may not check for validity. If it does check for validity, the program receiving data from the parser may or may not care about validity errors. In some cases, such as feeding records into a database, a validity error may be quite serious, indicating that a required field is missing, for example. In other cases, rendering a web page perhaps, a validity error may not be so important, and it can work around it. Well-formedness is required of all XML documents; validity is not.

A Simple DTD Example

The following is a DTD for the example 2.2.

```
<!ELEMENT person      (name, profession*)>
<!ELEMENT name        (first_name, last_name)>
<!ELEMENT first_name  (#PCDATA)>
<!ELEMENT last_name   (#PCDATA)>
<!ELEMENT profession  (#PCDATA)>
```

This DTD described a person. The person have a name and three professions. The name have a first name and a last name. The particular person described in that example is Alan Turing. However, that's not relevant for DTDs. A DTD only describes the general type, not the specific instance. A

DTD for person documents say that a person element contains one name child element and zero or more profession child elements. It further say that each name element contains a first_name child element and a last_name child element. Finally it state that the first_name, last_name, and profession elements all contain text.

This DTD would probably be stored in a separate file from the documents it describes. This allows it to be easily referenced from multiple XML documents. However, it can be included inside the XML document if that's convenient, using the document type declaration.

Each line of this DTD is an element declaration. The first line declares the person element; the second line declares the name element; the third line declares the first_name element; and so on. However, the line breaks are not relevant except for legibility. Although it's customary to put only one declaration on each line, it's not required. Long declarations can even span multiple lines.

2.13 XML Schema

Although Document Type Definitions can enforce basic structural rules on documents, many applications need a more powerful and expressive validation method. The W3C developed the XML Schema Recommendation, released on May 2, 2001 after a long incubation period, to address these needs. Schemas can describe complex restrictions on elements and attributes. Multiple schemas can be combined to validate documents that use multiple XML vocabularies.

2.13.1 Overview

A schema is a formal description of what comprises a valid document. An XML schema is an XML document containing a formal description of what comprises a valid XML document. A W3C XML Schema Language schema is an XML schema written in the particular syntax recommended by the W3C.

An XML document described by a schema is called an instance document. If a document satisfies all the constraints specified by the schema, it is considered to be schema-valid. The schema document is associated with an instance document through one of the following methods:

An `xsi:schemaLocation` attribute on an element contains a list of namespaces used within that element and the URLs of the schemas with which to validate elements in those namespaces.

An `xsi:noNamespaceSchemaLocation` attribute contains a URL for the schema used to validate elements that are not in any namespace.

The validating parser may attempt to locate the schema using the namespace of the element itself in one of these ways: directly by looking for a schema at that namespace, indirectly by looking for a RDDL document at

that namespace, or implicitly by knowing in advance which schema is right for that namespace.

A validating parser may be instructed to validate a given document against an explicitly provided schema, ignoring any hints that might be provided within the document itself.

Schemas Versus DTDs

DTDs provide the capability to do basic validation of the following items in XML documents:

- Element nesting
- Element occurrence constraints
- Permitted attributes
- Attribute types and default values

However, DTDs do not provide fine control over the format and data types of element and attribute values. Other than the various special attribute types (ID, IDREF, ENTITY, NMTOKEN, and so forth), once an element or attribute has been declared to contain character data, no limits may be placed on the length, type, or format of that content. For narrative documents (such as web pages, book chapters, newsletters, etc.), this level of control is probably good enough.

But as XML makes inroads into more data-intensive applications (such as web services using SOAP), more precise control over the text content of elements and attributes becomes important. The W3C XML Schema standard includes the following features:

- Simple and complex data types
- Type derivation and inheritance
- Element occurrence constraints
- Namespace-aware element and attribute declarations

The most important of these features is the addition of simple data types for parsed character data and attribute values. Unlike DTDs, schemas can enforce specific rules about the contents of elements and attributes. In addition to a wide range of built-in simple types (such as string, integer, decimal, and dateTime), the schema language provides a framework for declaring new data types, deriving new types from old types, and reusing types from other schemas.

Besides simple data types, schemas add the ability to place more explicit restrictions on the number and sequence of child elements that can appear in a given location. This is even true when elements are mixed with character data, unlike the mixed content model (`#PCDATA`) supported by DTDs.

XML compression techniques

3.1 Introduction

In general, compressing data means to reduce the necessary space to represent the information that they contains. A reduction of the dimensions of the documents introduces different advantages:

- in data exchange, it increases the transferable amount of information in the unit of time across a channel of communication
- in data storage, it reduce the amount of occupied space;
- in data elaboration, it improves the performance thanks to the minor amount of data that must be treated

The main issue exposed by XML format is its intrinsic verbosity. The problem can be addressed if data compression is used to reduce the space requirements of XML. Because XML is text-based, the simplest and most common approach is to use the existing text compressors and to compress XML documents as ordinary text files. However, although it is possible to reduce the amount of data significantly in this way, the compressed XML documents often remain larger than equivalent text or binary formats. It is obvious that this solution is only suboptimal, since two documents carrying the same message should have the same entropy and therefore it should be possible to compress them to about the same size. The main reason is that general-purpose compressors often fail to discover and utilize the redundancy contained in the structure of XML. Another problem with these compressors is that they introduce another pass into XML processing, since decompression is necessary before the data can be processed. Recently, a number of XML-conscious compressors have emerged that improve on the traditional text compressors. Because they are designed to take advantage of the structure of XML during the compression, they often achieve considerably better results. Very often, these tools rely on the functionality of the existing text compressors, and only adapt them to XML.

3.2 XML-conscious compressors

The category of XML-conscious compressors contains those compressors that exploit structure information in order to obtain better compression factors. But they don't permit the execution of a query in compressed domain. Indeed, with this type of systems, it is necessary to decompress partially or, more frequently totally, the document in order to evaluate a query.

3.2.1 XMill

XMill [91] is an XML compressor based on Gzip, which can compress about twice as good, and at about the same speed. It allows to combine existing compressors in order to compress heterogeneous XML data. Further, it is extensible with user-defined compressors for complex data types, such as DNA sequences, etc.

Example 3.1. A sample XML document

```
<book>
  <title lang="en">Views</title>
  <author>Miller</author>
  <author>Tai</author>
</book>
```

XMill parses XML data with a SAX parser, and transforms it by splitting the data into three types of containers: one container for the element and attribute symbols, one for the document tree structure, and several containers for storing the character data. By default, each element or attribute is assigned one data container. XMill employs a path processor that is driven by so called container expressions. The container expressions are based on the XPath language [154], and allow experienced users to group the data within a certain set of elements into one container to improve compression efficiency. In the output file, the individual containers are compressed using Gzip.

XMill applies three principles to compress XML data:

Separate structure from data. The structure, represented by XML tags and attributes, and the data are compressed separately. XMill uses numeric tokens to represent the XML structure. Start-tags are dictionary encoded, i.e. assigned an integer value, while all end-tags are replaced by the token /. Data values are replaced with their container number. When complete document is processed, the token table, the structure container and the data containers are compressed using Gzip. The tokens are represented as integers with 1, 2, or 4 bytes; tags and attributes are positive integers, / is 0, and container numbers are negative integers. To illustrate the tokenization of the structure, consider the sample XML document in example 3.1. After the document is processed, the structure will be tokenized as:

T1 T2 T3 C3 / C4 / T4 C5 / T4 C5 / /

and the following dictionary is created: book = T1, title = T2, @lang = T3, author = T4. Data values are assigned containers C3, C4, and C5 depending on their parent tag.

Group data items with related meaning. The data items are grouped into containers, which are compressed separately. XMill groups the data items based on the element type, but this can be overridden through the container expressions. By grouping similar data items, the compression can improve substantially.

The container expression describe the mappings from paths to containers. Consider the following regular expressions derived from XPath:

$e ::= \text{label} \mid * \mid \# \mid e1/e2 \mid e1//e2 \mid (e1|e2) \mid (e)^+$

Except for $(e)^+$ and $\#$, all are XPath constructs: label is either tag or an @attribute, $*$ denotes any tag or attribute, $e1/e2$ is concatenation, $e1//e2$ is concatenation with any path in between, and $(e1|e2)$ is alternation. To these constructs, $(e)^+$ has been added, which is the strict Kleene-Closure. The construct $\#$ stands for any tag or attribute (much like $*$), but each match of $\#$ will determine a new container.

The container expression has the form $c ::= /e//e$, where $/e$ matches e starting from the root of the XML tree while $//e$ matches e at arbitrary depth of the tree. $//^*$ is abbreviated by $//$.

Apply semantic compressors to containers. Because the data items can be of different types (text, numbers, dates, etc.), XMill allows the users to apply different specialized semantic compressors to different containers. At first, the items in the container are processed by the semantic compressor, and then they are passed to Gzip.

There are 8 different semantic compressors in XMill. These can be used to encode integers, enumerations and texts, or the sequences or the repetitions of them more efficiently. For example, positive integers are binary encoded as follows: numbers less than 128 use one byte; those less than 16384 use two bytes, otherwise they use four bytes. The most significant one or two bits describe the length of the sequence.

Semantic compressors are specified on the command line using the syntax $c => s$ where c is a container expression and s is a semantic compressor.

It is possible to write his own semantic compressor (for example, for encoding the DNA sequences) and link it into XMill. The list of semantic compressors can be extended by the users. Under the default setting, XMill compresses 40%-60% better than Gzip. With the user assistance (grouping related data, applying semantic compressors), it is possible to further improve the compression by about 10%.

The main disadvantage of XMill is that it scatters parts of the documents, making incremental processing impossible.

3.2.2 XMLZip

Java-based XMLZip is a creation of XML Solutions [153]. It operates in a rather interesting way. The compression is driven by the level parameter l . Based on this parameter, XMLZip processes the document using the DOM interface, and breaks the structural tree into multiple components: a root component containing the elements up to the level l , and one component for each of the remaining subtrees starting at level l . The root component is modified by adding references to the subtrees, and the individual components are then compressed using the Java Zip/DeDeflate library (which uses a variant of the LZSS method).

Example 3.2. A sample XML document

```
<root>
  <child id="1">
    ...
  </child>
  <child id="2">
    ...
  </child>
</root>
```

Consider the XML document in example 3.2. Suppose that $l = 2$. XMLZip splits the original document into three components, as displayed in example 3.3. In the root component, `<xmlzip>` tags are inserted to reference the detached subtrees. After that, the individual components are compressed.

Example 3.3. Decomposition of the XML document with $l = 2$

```
<root>                <child id="1">        <child id="2">
  <xmlzip id="1"/>      ...                ...
  <xmlzip id="2"/>    </child>            </child>
</root>
```

The compression efficiency depends on the value of l . In most occasions, increasing l causes the performance to deteriorate, since the redundancies across the separated subtrees cannot be used in the compression.

In a comparison to other XML compressors, XMLZip yields considerably worse results. It is often outperformed even by the ordinary text compressors, such as Gzip. However, the main benefit of XMLZip is that it allows limited random access to the compressed XML documents without storing the whole document uncompressed or in the memory. Only the portion of the XML tree that needs to be accessed is uncompressed. It is possible to implement a DOM-like API to control the amount of memory required, or to speed up the access for queries, for example.

XMLZip can only be run on entire XML documents, and therefore the compression is off-line.

3.2.3 XMLPPM

In [33], the PPM compression has been adapted to compress XML. The compressor—called XMLPPM—uses so called multiplexed hierarchical modeling (MHM) for modeling the structure of XML. In MHM, several PPM models are multiplexed together, and switches among them are performed based on the syntactic context supplied by the parser.

XMLPPM uses four PPM models: one for element and attribute names, one for element structure, one for attributes, and one for character data. Each model maintains its own state but all share the access to one underlying arithmetic coder. Element start tags, end tags, and attribute names are dictionary encoded using numeric tokens. Whenever a new symbol is encountered, the encoder sends the symbol name and the decoder enters it to the corresponding dictionary. Some tokens are reserved, and are used to encode the events such as the start of the character data, the element end tag, etc.

To demonstrate the operation of XMLPPM, we encode the following XML fragment:

```
<elt att="abcd">XYZ</elt>
```

Suppose that the tag `elt` has been seen before, and is represented by the token 10, but the attribute `att` has not, and the next available token for attribute names is 0D. The state of the individual MHM models after processing the XML fragment is shown in Table 3.1.

	<elt	att=	"abcd"	>	XYZ	</elt>
Elt:	10				FE	FF
Att:		10 0D	asdf00	10 FF		
Char:					10 XYZ00	
Sym:			att00			

Table 3.1. MHM models after processing the example XML fragment

Notice the token `10` that has been “injected” into the attribute and character data models. This token is not encoded; instead, it is used to indicate the cross-class sequential dependencies within the XML document. A common case for these dependencies is a strong correlation between the enclosing element tag and the enclosed data. Using the “injection” mechanism, these correlation can be exploited by MHM.

The authors of XMLPPM have also implemented a variant of MHM that uses PPM* instead of PPM, which they call MHM*. Compared to MHM, MHM* performs slightly better on average, but is considerably slower. On structured documents, MHM performs much worse (about 20%-40%) than MHM*; on the other hand, MHM* is worse on textual documents.

XMLPPM (using either MHM or MHM*) compresses extremely well, outperforming most of the concurrent compressors. The compression is on-line, and therefore the XML data can be processed incrementally.

3.2.4 Millau

Millau [56] is an on-line XML compression method that is suitable for compression and streaming of small XML documents (smaller than 5 kilobytes). Millau can make use of the associated schema (if available) in the compression of the structure.

The encoding is based on the Wireless Binary XML format (WBXML) proposed by the Wireless Application Protocol Forum which performs a lossless reduction of the size of XML documents. This method uses a table of tokens to encode the XML tags and the attribute names. Some tokens are reserved, and are used to indicate events such as the character data, the end of element, etc. The meaning of a particular token is dependent on the context in which it is used. There are two basic types of tokens: global tokens and application tokens. Global tokens are assigned to fixed set of codes in all contexts and are unambiguous in all situations. Global codes are used to encode inline data (such as strings, entities, etc.) and to encode a variety of miscellaneous control functions. Application tokens have a context-dependent meaning and are split into two overlapping code spaces: the tag code space and the attribute code space. The tag code space represents specific tag names and the attribute code space comprises of attribute-start token and attribute-value token. Since the set of tags and attributes is known in advance in the WAP protocol, the table is fixed and does not have to be contained in the encoded data. The output data is a stream of tokens and uncompressed character data. In this stream, the structure of the original XML document is preserved.

Millau improves on the WBXML scheme, making it possible to compress the character data: the structure and the character data are separated into two streams, and the character data stream is compressed using conventional text compressors. In the structure stream, special tokens are inserted to indicate the occurrences of compressed data.

Since the set of elements and attributes is not known in advance in the case of ordinary XML documents, Millau sets out a strategy for building the token table. If the DTD exists, Millau constructs the table based upon it; otherwise, the document is pre-parsed and the tokens are assigned to the encountered elements and attributes. The table of tokens is contained in the encoded data.

An XML parser for processing the Millau streams is implemented using both DOM and SAX. Because binary tokens are processed –instead of strings in the case of uncompressed XML documents–, the parser usually operates very fast.

Although Millau is outperformed by the traditional text compression algorithms on large XML files, it achieves better compression for file sizes between

0-5 kilobytes, which is the typical file size for e-Business transactions, such as orders, bill payments, etc.

3.2.5 XML-Xpress

Commercially available XML-Xpress [78] is a schema-aware compressor that can work either with the DTD or XML Schema [152]. When the schema is known to XML-Xpress, XML tags can be encoded very efficiently. For example, if an element is defined in the schema as having only one of two sub-elements (A|B), only a binary decision needs to be included in the encoded file to determine which of A or B is present. The compression can be further improved by using XML Schema instead of DTD, because the information about the data types of the element data is utilized in the compression. In [78], compression ratios exceeding 30 : 1 are reported.

The main disadvantage of XML-Xpress is that it is primarily a schema-specific compressor, and thus the above average compression ratios are dependent on the presence of a known schema. In the absence of such a schema, XML-Express uses a general-purpose compressors, and the outstanding compression performance is lost.

3.3 Query-aware XML compressors

Those systems designed in order to work in conjunction with a query processor belong to the category of query-aware XML compressors. This approach permits to execute some type of queries directly on the compressed domain. However, for type of queries not supported, it is necessary the decompression of XML documents.

3.3.1 XGRIND

The XGrind project [140] pioneered the field of query processing on compressed XML documents. XGrind directly supports queries in the compressed domain. It compresses at the granularity of individual *element/attribute* values using a simple context-free compression scheme based on Huffman coding [76]. This means that *exact-match* and *prefix-match* user queries can be entirely executed directly on the compressed document, with decompression restricted to only the final results provided to the user.

A novel and especially useful feature of XGrind is that it retains the structure of the original XML document in the compressed format also. In fact, the compressed XML document can be viewed as the original XML document with its tags and element/attribute values replaced by their corresponding encodings. The advantage of doing so is that the variety of efficient techniques available for parsing/querying XML documents can also be used to process

the compressed document. Second, indexes, such as those proposed in [100], can now be built on the compressed document in similar manner to those built on regular XML documents. Third, updates to the XML document can be directly executed on the compressed version. Finally, a compressed document can be checked for validity against the compressed version of its DTD, without having to resort to any decompression.

Another feature of XGrind is that, for XML documents adhering to a DTD, it attempts to utilize the information in the DTD to enhance the compression ratio. For example, attribute values that are of enumerated-type are recognized from the DTD and are encoded differently from other attribute values.

XGrind uses different techniques for compressing *metadata*, *enumerated-type attribute values*, and (general) *element/attribute* values, respectively.

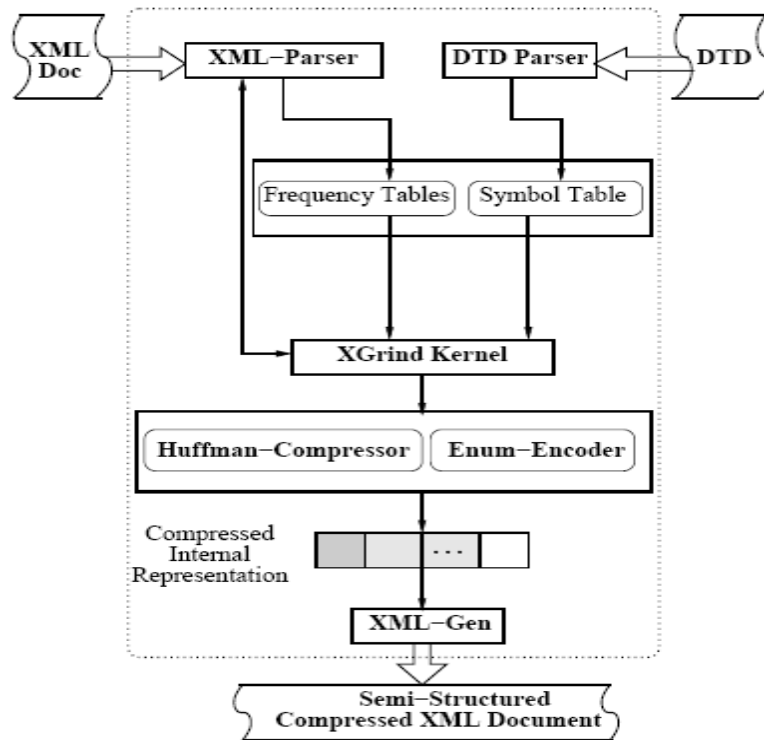


Fig. 3.1. Architecture of XGrind Compressor

The architecture of the XGrind compressor, along with the information flows, is shown in Figure 3.1. The **XGrind Kernel** is the heart of the compressor. It starts off by invoking the **DTD Parser**, which parses the DTD

of the XML document, initializes *frequency tables* for each element or non-enumerated attribute, and populates a *symbol table* for attributes having enumerated-type values. The kernel then invokes the **XML Parser**, which scans the XML document and populates the set of frequency tables containing statistics (in the form of frequencies of character occurrences) for each element and non-enumerated attribute. The XML Parser is invoked a second time by the kernel to construct a tokenized form (tag, attribute, or data value) of the XML document. These tokens are supplied in streaming fashion to the kernel which calls for each token, based on its type, one of the following encoders:

- **Enum-Encoder:** is used for metadata and enumerated type data items. Each start-tag of an element is encoded by a ‘T’ followed by a unique element-ID. All end-tags are encoded by ‘/’s. Attribute names are encoded by the character ‘A’ followed by a unique attribute-ID. Enumerated-type attribute values, on the other hand, are encoded using the symbol table information.
- **Huffman-Compressor** is used for non-enumerated data items. This module implements the non-adaptive Huffman coding compression scheme. It encodes each *element/attribute* value with the help of its associated Huffman tree, which is constructed from its corresponding frequency table. The last byte of the encoded sequence is padded to be *byte-aligned*, and this encoded sequence is then “escaped” so that the compressed XML document can be parsed without ambiguity.

The compressed output of the above encoders, along with the various frequency and symbol tables, is called the *Compressed Internal Representation* (CIR) of the compressor and is fed to **XML-Gen**, which converts the CIR into a semi-structured compressed XML document. This conversion is done on the fly during the second pass while the document is being compressed.

3.3.2 XPRESS

In contrast to the other XML compressors, XPRESS gets rid of this overhead by using an encoding method, called *reverse arithmetic encoding*, and minimizes the overhead of partial decompression by utilizing diverse encoding methods.

XPRESS has the following combination of characteristics to compress and retrieve XML data efficiently:

- **Reverse Arithmetic Encoding:** XPRESS adopts the reverse arithmetic encoding method that encodes a label path as a distinct interval in $[0.0, 1.0)$. Using the containment relationships among the intervals, path expressions are evaluated on compressed XML data.
- **Automatic Type Inference:** XPRESS devises an efficient type inference engine that does not require the human interference in order to apply effective encoding methods to various kinds of data values of XML elements.

- **Apply Diverse Encoding Methods to Different Types:** According to the inferred type information, proper encoding methods is applied to data values.
- **Semi-adaptive Approach:** The compression scheme used by XPress is categorized as semi-adaptive approach which uses a preliminary scan of the input file to gather statistics. Since the semi-adaptive approach does not change the statistics during the compression phase, the encoding rules for data are independent to the locations of data. This property allow the system to query compressed XML data directly.
- **Homomorphic Compression:** Like XGrind, XPRESS is a homomorphic compressor which preserves the structure of the original XML data in compressed XML data.

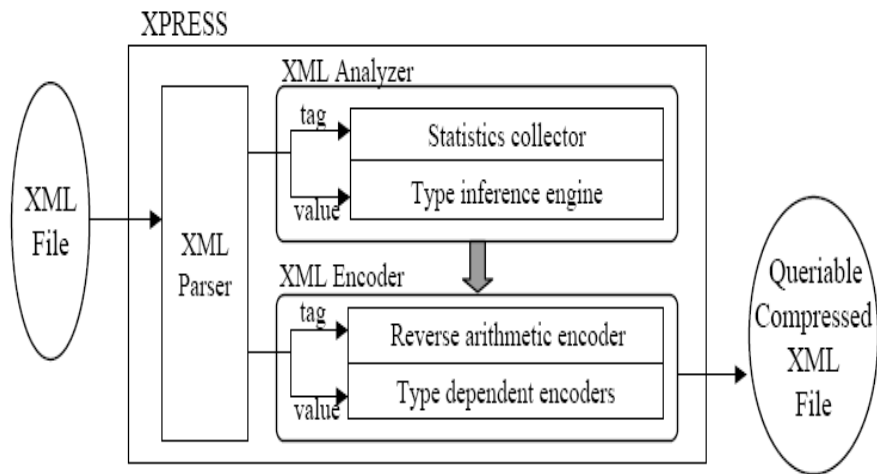


Fig. 3.2. The architecture of XPRESS

XQueC: XQuery processor and Compressor

4.1 Introduction

XML documents have an inherent textual nature due to redundant tags and to the PCDATA content. Therefore, they lead themselves naturally to compression. Once the compressed documents are produced, however, one would like to be able to still query them under a compressed form as much as possible (“lazy decompression”). The advantages of processing queries in the compressed domain are several: first, in a traditional query setting, access to small chunks of data may lead to less disk I/Os and reduce the query processing time; second, the memory and computation efforts in processing compressed data can be dramatically lower than those for uncompressed ones, thus even low-battery mobile devices can afford them; third, the possibility of obtaining compressed query results allows to spare network bandwidth when sending these results to a remote location.

As pointed out in chapter 3, previous systems have been proposed, such as e.g. XGrind [140] and XPRESS [106], which allow the evaluation of simple path expressions in the compressed domain. However, these systems are based on a naive top-down query evaluation mechanism, which is not enough to execute queries efficiently. Most of all, they are not able to execute a large set of common XML queries (such as joins, inequality predicates, aggregates, nested queries etc.), without spending prohibitive times in decompressing intermediate results.

We address the problem of compressing XML data in such a way as to allow efficient XQuery evaluation in the compressed domain. It achieves a good trade-off among data compression factors, queryability and XQuery expressibility. To that purpose, we have carefully chosen a fragmentation and storage model for the compressed XML documents, providing selective access paths to the XML data, and thus further reducing the memory needed in order to process a query.

The basis of our fragmentation strategy is borrowed from the XMill [91] project. XMill is a very efficient compressor for XML data, however, it

was not designed to allow querying the documents under their compressed form. XMill made the important observation that data nodes (leaves of the XML tree) found on the same path in an XML document (for example `/site/people/person/address/city` in the XMark [128] documents) often exhibit similar content. Therefore, it makes sense to group all such values into a single *container* and choose the compression strategy *once per container*. Subsequently, XMill treated a container like a single “chunk of data” and compressed it as such, which disables access to any individual data node, unless the whole container is decompressed. Separately, XMill compressed and stored the structure tree of the XML document.

While in XMill a container may in fact contain leaf nodes found under several paths, leaving to the user or the application the task of defining these containers, in XQueC the fragmentation is always dictated by the paths, i.e., we use one container per root-to-leaf path expression. But most importantly, unlike XMill, we compress *each individual value in a container* using a compressor that is aware of the commonality between *all values in the container*. Thus, each compressed value is individually accessible, enabling expressive and efficient query processing.

We base our work on the principle that XML compression (for saving disk space) and sophisticated query processing techniques (like complex physical operators, indexes, query optimization etc.) can be used together when properly combined. This principle has been stated and forcefully validated in the domain of relational query processing [147], [32]. Thus, it is not less important in the realm of XML.

We focus on the right compression of the *values* found in an XML document, coupled with a compact storage model for all parts of the document. Compressing the *structure* of an XML documents has two facets. First, XML tags and attribute names are extremely repetitive, and practically all systems (indeed, even those not claiming to do “compression”) encode such tags by means of much more compact tag numbers. Second, an existing work [22] has addressed the summarization of the tree structure itself, connecting among them parent and child nodes. While structure compression is interesting, its advantages are not very visible when considering the XML document as a whole. Indeed, for a rich corpus of XML datasets, both real and synthetic, our measures have shown that values make up 70% to 80% of the document structure. Projects like XGrind [140] and XPRESS [106] have already proposed schemes for value compression that would enable querying, but they suffer from limited query evaluation techniques (see also Section 4.1.2). Moreover, these systems use one kind of compression algorithm at a time. However, it is desirable to tailor the algorithm to the data and to the queries. This has been done in our system by means of a suitable cost model.

By doing data fragmentation and compression, XQueC indirectly targets the problem of main-memory XQuery evaluation, which has recently attracted the attention of the community [96], [22]. In [96], the authors show that some current XQuery prototypes do not work for more than 33MB documents. Fur-

thermore, some such in-memory prototypes are shown to exhibit prohibitive query execution times even for simple lookup queries.

XQueC addresses this problem in a two-fold way. First, in order to diminish its footprint, it applies powerful compression to the XML documents. The compression algorithms that we use allow to evaluate most predicates directly on the compressed values. Thus, decompression is often necessary only at the end of the query evaluation (see Section 4.4). Second, the XQueC storage model includes lightweight access support structures for the data itself, providing thus efficient primitives for query evaluation.

4.1.1 The XQueC system

The system we propose compresses XML data and queries it as much as possible under its compressed form, covering all real-life, complex classes of queries. The XQueC system adheres to the following principles:

1. As in XMill, data is collected into containers, and the document structure stored separately. In XQueC, there is a container for each different $\langle \text{type}, pe \rangle$, where pe is a distinguished root-to-leaf path expression and type is a distinguished elementary type. The set of containers is then partitioned again to allow for better sharing of compression structures, as explained in Section 4.2.2.
2. In contrast with previous compression-aware XML querying systems, whose storage was plainly based on files, XQueC is the first to use a complete and robust storage model for compressed XML data, including a set of access support structures. Such storage is fundamental to guarantee a fast query evaluation mechanism.
3. XQueC seamlessly extends a simple algebra for evaluating XML queries to include compression and decompression. This algebra is exploited by a cost-based optimizer, which may choose query evaluation strategies, that freely mix regular operator and compression-aware ones.
4. XQueC is the first system to exploit the *query workload* to (i) partition the containers into sets according to the source model¹ and to (ii) properly assign the most suitable compression algorithm to each set. We have devised an appropriate cost model, which helps making the right choices.
5. XQueC is the first compressed XML querying system to use the order-preserving² textual compression. Among several alternatives, we have chosen to use the ALM [7] compression algorithm, which provides good compression ratios and still allows fast decompression, which is crucial

¹ The source model is the model used for the encoding, for instance the Huffman encoding tree for Huffman compression [76] and the dictionary for ALM compression [7], outlined later.

² Note that a compression algorithm $comp$ preserves order if for any x_1, x_2 , $comp(x_1) < comp(x_2)$ iff $x_1 < x_2$.

for an algorithm to be used in a database setting [58]. This feature enables XQueC to evaluate, in the compressed domain, the class of queries involving inequality comparisons, which are not featured by the other compression-aware systems.

In the following sections, we will use XMark [128] documents for describing XQueC. A simplified structural outline of these documents is depicted in Figure 4.1 (at right). Each document describes an auction site, with people and open auctions (dashed lines represent IDREFs pointing to IDs and plain lines connect the other XML items). We describe XQueC following its architecture, depicted in Figure 4.1 (at left). It contains the following modules:

1. The *loader and compressor* converts XML documents in a compressed, yet queryable format. A cost analysis leverages the variety of compression algorithms and the query workload predicates to decide the partition of the containers.
2. The *compressed repository* stores the compressed documents and provides: (i) compressed data access methods, and (ii) a set of compression-specific utilities that enable, e.g., the comparison of two compressed values.
3. The *query processor* evaluates XQuery queries over compressed documents. Its complete set of physical operators (regular ones and compression-aware ones) allows for efficient evaluation over the compressed repository.

4.1.2 A comparative overview of the main XML compression systems

XML data compression was first addressed by XMill [91], following the principles outlined in the previous chapter. After coalescing all values of a given container into a single data chunk, XMill compresses separately each container with its most suited algorithm, and then again with `gzip` to shrink it as much as possible. However, an XMill-compressed document is opaque to a query processor: thus, one must fully decompress a whole chunk of data before being able to query it.

The XGrind system [140] aims at query-enabling XML compression. XGrind does not separate data from structure: an XGrind-compressed XML document is still an XML document, whose tags have been dictionary-encoded, and whose data nodes have been compressed using the Huffman [76] algorithm and left at their place in the document. XGrind's query processor can be considered an extended SAX parser, which can handle *exact-match* and *prefix-match queries* on compressed values and *partial-match* and *range queries* on decompressed values. However, several operations are not supported by XGrind, for example, non-equality selections in the compressed domain. Therefore, XGrind cannot perform any join, aggregation, nested queries, or `construct` operations. Such operations occur in many XML query scenarios, as illustrated by XML benchmarks (e.g., all but the first two of the 20 queries in XMark [128]).

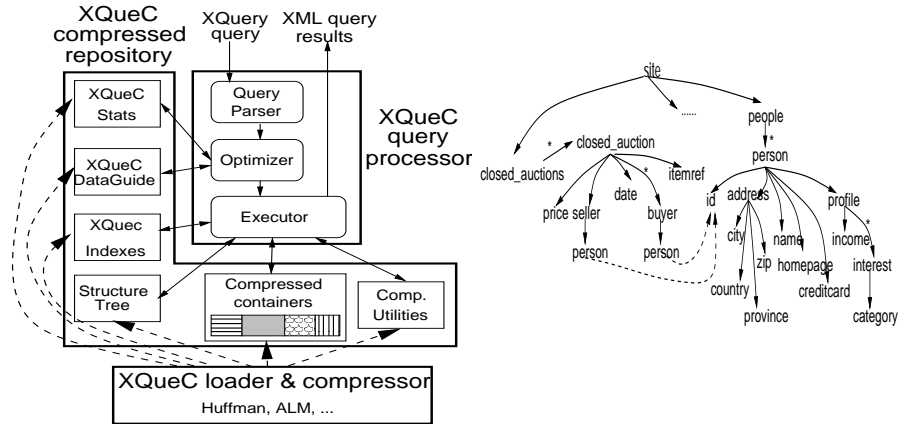


Fig. 4.1. Architecture of the XQueC prototype (left); simplified summary of the XMark XML documents (right).

Also, XGrind uses a fixed naive top-down navigation strategy, which is clearly insufficient to provide for interesting alternative evaluation strategies, as it was done in existing works on querying compressed relational data (e.g., [31], [147]). These works considered evaluating arbitrary SQL queries on compressed data, by comparing (in the traditional framework of cost-based optimization) many query evaluation alternatives, including compression / decompression at several possible points.

XPRESS [106] uses the novel *reverse arithmetic encoding* method, mapping entire path expressions to intervals. Also, XPRESS uses a simple mechanism to infer the type (and therefore the compression method suited) of each elementary data item. XPRESS's compression method, like XGrind's, is homomorphic, meaning it preserves the document structure.

To summarize, these compression and querying systems do not come anywhere near to efficiently executing complex XQuery queries. Indeed, even the evaluation of XPath queries is slowed down by the use of the fixed top-down query evaluation strategy.

4.2 Compressing XML documents in a queryable format

In this section, we present the principles behind our approach for storing compressed XML documents, and the resulting storage model.

4.2.1 Compression principles

In general, we make the observation that within XML text, strings represent a large percentage of the document, while instead numbers are not crucial. Thus, compression of strings, when effective, can truly reduce the occupancy of XML documents. Nevertheless, not all the compression algorithms can seamlessly

afford string comparisons in the compressed domain. In our system, we include both order-preserving and order-agnostic compression algorithms, and the final choice is entrusted to a suitable cost model.

Our approach for compressing XML was guided by the following principles:

Order-agnostic compression.

As an order-agnostic algorithm, we chose Huffman, which is universally known as a simple algorithm which achieves the best possible redundancy among the resulting codes. The process of encoding and decoding is also faster than universal compression techniques. Finally, it has a set of fixed codewords, thus strings compressed with Huffman can be compared in the compressed domain within equality predicates. However, inequality predicates need to be decompressed. That is why in XQueC we may exploit order-preserving compression as well as not order-preserving one.

Order-preserving compression.

Whereas everybody knows the potentiality of Huffman, the choice of an order-preserving algorithm is not immediate. We had initially three choices for encoding strings in an order-preserving manner: the Arithmetic [149], Hu-Tucker [74] and ALM [7] algorithms. We knew that dictionary-based encoding has demonstrated its effectiveness w.r.t. other non-dictionary approaches [109] while ALM has outperformed Hu-Tucker (as described in [6]). The former being both dictionary-based and efficient, happened to be a good choice in our system. ALM has been used in relational databases for blank-padding (i.e. in Oracle) and for indexes compression. Due to its very nature (dictionary-based), ALM decompresses faster than Huffman, since it outputs bigger portions of a string at a time, when decompressing. Moreover, ALM solved the problem of order-preserving dictionary compression, raised by encodings such as Zilch encoding, string prefix compression and composite key compression by improving each of these. Indeed, ALM eliminates the prefix property exhibited by those former encodings by allowing in the dictionary more than one symbol for the same prefix. The fundamental mechanics behind the algorithm tells to consider the original set of source substrings, to split it into disjunct partitioning intervals set and to associate an interval prefix to each partitioning interval. For example, in Figure 4.2, it is shown the mapping from the original source (made of the strings **there**, **their**, **these**) into some partitioning intervals and associated prefixes, which clearly do not scramble the original order among the source strings. We have implemented our own version of the algorithm, and we have obtained encouraging results w.r.t. previous compression-aware XML processors (see Section 4.5).

Workload-based choices of compression.

Among the possible predicates writable in an XQuery query, we distinguish among the inequality, equality and wildcard. The ALM algorithm [7] is able

token	code	interval
.....	
the	c	[theaa, therd]
there	d	[there, there]
the	e	[therf, thezz]
ir	b	[ir, ir]
.....	
se	v	[se, se]

their	cb
there	d
these	ev

Fig. 4.2. An example of encoding in ALM.

to afford inequality and equality predicates in the compressed domain, but not wildcards, whereas Huffman [76] supports prefix-wildcards and equality but not inequality. Thus, the choice of the algorithm can be aided by a proper query workload, whenever this turns to be available. In case, instead, the workload has not been provided, XQueC uses ALM for strings and decompresses the compared values in case of wildcard operations.

Structures for algebraic evaluation.

Containers in XQueC act as free-of-charge indexes and resemble B+trees on values. Moreover, a light-weight structure summary allows for accessing the structure tree and the data containers in the query evaluation process. Data fragmentation allows for better exploiting all the possible evaluation plans, i.e. bottom-up, top-down, hybrid or index-based. As shown below, several queries of the XMark benchmark take advantage of the XQueC appropriate structures and of the consequent flexibility in parsing and querying these compressed structures.

4.2.2 Compressed storage structures

The XQuec loader/compressor parses and splits an XML document into the data structures depicted in Figure 4.1.

Node name dictionary. We use a dictionary to encode the element and attribute names present in an XML document. Thus, if there are N_t distinct names, we assign to each of them a bit string of length $\log_2(N_t)$. For example, the XMark documents use 92 distinct names, which we encode on 7 bits

Structure tree. We assign to each non-value XML node (element or attribute) an unique integer ID. The structure tree is stored as a sequence of *node records*, where each record contains: its own ID, the corresponding tag code; the IDs of its children; and (redundantly) the ID of its parent. For better query performance, as an access support structure, we construct and store a B+ search tree on top of the sequence of node records. Finally, each node record points to all its attribute and text children in their respective containers.

Value containers. All data values found under the same root-to-leaf path expression in the document are stored together into homogeneous containers. A container is a sequence of *container records*, each one consisting of a compressed value and a pointer to parent of this value in the structure tree. Records are not placed in the document order, but more reasonably in a lexicographic order, to enable fast binary search. Note that container generation as done in XQueC is reminiscent of vertical partitioning of relational databases [117]. This kind of partitioning seamlessly guarantees random access to the document content at different points, i.e. the containers access points. In order to better search the space of query evaluation strategies, this choice has been demonstrated to be effective in practice (see Section 4.5). Moreover, containers, even if separated, may share the same source model or, they can be compressed with different algorithms if not involved in the same queries. This is decided by a cost analysis which exploits the query workload and the similarities among containers, as described in Section 4.3.

Structure summary. The loader also constructs, as a redundant access support structure, a structural summary representing all possible paths in the document. For tree-structured XML documents, it will always have less nodes than the document (typically by several orders of magnitude). A structural summary of the auction documents can be derived from Figure 4.1, by (i) omitting the dashed edges, which brings it to a tree form, and (ii) storing in each non-leaf node in Figure 4.3, accessible in this tree by a path p , the list of nodes reachable in the document instance by the same path. Finally, the leaf nodes of our structure summary point to the corresponding value containers. Note that the structure summary is small enough not to worsen the compression ratios of our documents, when compressed. Indeed, in our experiments on the corpus of XML documents described in Section 4.5, the structure summary amounts to about 19% of the original document size.

Other indexes and statistics. When loading a document, other indexes and/or statistics can be created, either on the value containers, or on the structure tree. Our loader prototype currently gathers simple fan-out and cardinality statistics (e.g. number of `person` elements).

To measure the occupancy of our structures, we have used a set of documents produced by means of the *xmlgen* generator of the XMark project and

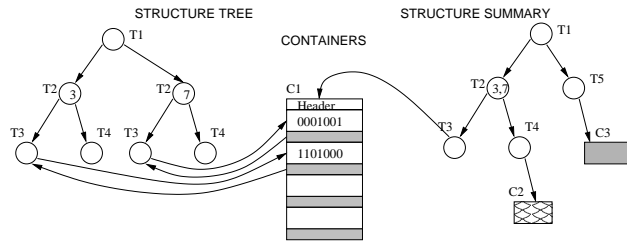


Fig. 4.3. Storage structures in the XQueC repository

ranged from 115KB to 46MB. They have been reduced by an average factor of 60% after compression (these figures include all the above access structures).

Our proposed storage structure is the simplest and most compact one that fulfills the principles listed at the beginning of Section 4.2; there are many ways to store XML in general [3]. If we omit our access support structures (backward edges, B+ index, and the structure summary), we shrink the database by a factor of 3 to 4, albeit at the price of deteriorated query performance.

There are many ways to store XML in general [3]. Any storage mechanism for XML can be seamlessly adopted in XQueC, as long as it allows the presence of containers and the facilities to access container items.

4.2.3 Memory issues

Data fragmentation in XQueC guarantees a wide variety of query evaluation strategies, and not solely top-down evaluation as in homomorphic compressors [140], [106]. Instead of identifying at compile-time the parts of the documents necessary for query evaluation, as given by an XQuery projection operator [96], in XQueC the path expressions are hard-coded into the containers and projection is already prepared in advance when compressing the document and without any additional effort for the loader. Consider as examples the following query Q14 of XMark:

```
FOR $i IN document("auction.xml")/site//item
WHERE CONTAINS($i/description,"gold")
RETURN $i/name/text()
```

This query would require prohibitive parsing times in XGrind and XPRESS, which basically have to load into main-memory all the document and parse

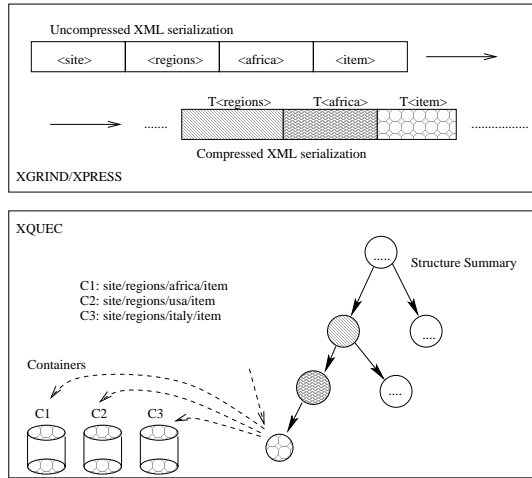


Fig. 4.4. Accesses to containers in case of XMark’s Q14 with *descendant-or-self* axis in XPress/XGrind versus XQueC.

it entirely in order to find the sought items. For this query, as shown in Figure 4.4, all the XML stream has to be parsed to find the elements `<item>`.

In XQueC, the compressor has already shredded the data and accessibility to these data from the structure summary allows to save the parsing and loading times. Thus, in XQueC the structure summary is parsed (not all the structure tree), then the involved containers are directly accessed (or alternatively their selected single items) and loaded into main-memory. Precisely, as shown in Figure 4.4, once the structure summary leads to the containers C_1 , C_2 and C_3 , only these (or part of them) need to be fetched in memory. Finally, note that in Galax, extended with the projection operator [96], the execution times for queries involving the *descendant-or-self* axis (such as XMark Q14) are significantly increased, since additional complex computation is demanded to the loader for those queries.

4.3 Compression choices

XQueC exploits the query workload to choose the way containers are compressed. As already highlighted, the containers are filled up with textual data, which represents a big share of the whole documents. Thus, achieving a good trade-off between compression ratio and query execution times, must necessarily imply the capability to make a good choice for textual container compression.

First, a container may be compressed with any compression algorithm, but obviously one would like to apply a compression algorithm with nice

properties. For instance, the decompression time for a given algorithm strongly influences the times of queries over data compressed with that algorithm. Also, the compression ratio achieved by a given algorithm on a given container depends both on the algorithm and on the nature of container data.

Second, a container can be compressed separately or can share the same source model with other containers. The latter choice would be very convenient whenever for example two containers exhibit hidden similarities, which can be exploited during compression. Moreover, the occupancy of the source model matters as well as the occupancy of containers and this makes the choice challenging.

In order to illustrate the impact of compression choices, we give a possible intuition.

Intuition.

Consider two binary-encoded containers, ct_1 and ct_2 . ct_1 contains only strings composed of letters **a** and **b**, whereas ct_2 contains only strings composed of letters **c** and **d**. Suppose, as one extreme case, that two separate source models are built for the two containers; in this case, containers are encoded with 1 bit per letter. As the other extreme case, a common source model is used for both containers, thus requiring 2 bits per letter for the encoding, and raising the containers occupancy. This scenario may get even more complicated when one thinks that different kinds of encodings are possible for each container. This very simple example shows that compressing many containers with the same source model leads to losses in the compression ratio.

In the sequel, we show how our system addresses these problems, by proposing a suitable cost model, a greedy algorithm that let the user make the right choice, and some experimental results. The cost model of XQueC is based on the set of non-numerical (textual) containers, the set of available compression algorithms \mathcal{A} , and the query workload \mathcal{W} . As it is typical for optimization problems, we will characterize the search space, define the cost function, and finally propose a simple search strategy.

4.3.1 Search space: possible compression configurations

Let \mathcal{C} be the set of containers built from a set of documents \mathcal{D} . A *compression configuration* s for \mathcal{D} is denoted by a tuple $\langle P, alg \rangle$ where P is a partition of \mathcal{C} 's elements, and the function $alg : P \rightarrow \mathcal{A}$ associates a compression algorithm with each set p in the partition P . The configuration s dictates thus that all values of the containers in p will be compressed using $alg(p)$, and a single common source model. Moreover, let \mathcal{P} be the set of possible partitions of \mathcal{C} . The cardinality of \mathcal{P} is the *Bell number* $B_{|\mathcal{C}|}$, which is exponential with $|\mathcal{C}|$. For each possible partition $P_i \in \mathcal{P}$, there are $|\mathcal{A}|^{|P_i|}$ ways of assigning a compression algorithm to each set in P_i . Therefore, the size of the search space is: $\sum_{i=1}^{B_{|\mathcal{C}|}} |\mathcal{A}|^{|P_i|}$, which is exponential in $|\mathcal{A}|$.

4.3.2 Cost function: appreciating the quality of a compression configuration

Intuitively, the cost function for a configuration s reflects the time needed to apply the necessary data decompressions in order to evaluate the predicates involved in the queries of \mathcal{W} . Reasonably, it also accounts for the compression ratios of the employed compression algorithms, and it includes the cost of storing the source model structures. The cost of a configuration s is an integer value computed as a weighted sum of storage and decompression costs.

Characterization of compression algorithms.

Each algorithm $a \in \mathcal{A}$ is denoted by a tuple $\langle d_c, c_s(F), c_a(F), eq, ineq, wild \rangle$. The *decompression cost* d_c is an estimate of the cost of decompressing a container record by using a , the *storage cost* $c_s(F)$ is a function estimating the cost of storing a container record compressed with a , and the *storage cost of the source model structures* $c_a(F)$ is a function estimating the cost of storing the source model structures for a container record. F is a symmetric *similarity matrix* whose generic element $F[i, j]$ is a real number ranging between 0 and 1, capturing the normalized similarity between a container ct_i and a container ct_j . F is built on the basis of data statistics, such as the number of overlapping values, the character distribution within the container entries, and possibly other type information, whenever available (e.g. the XSchema types, using results presented in [19]). Finally, the *algorithmic properties* eq , $ineq$ and $wild$ are boolean values indicating whether the algorithm supports in the compressed domain: (i) equality predicates without prefix-matching (eq), (ii) inequality predicates without prefix-matching ($ineq$) and (iii) equality predicates with prefix-matching ($wild$). For instance, Huffman will have $eq = true$, $ineq = false$ and $wild = true$, while ALM will have $eq = true$, $ineq = true$ and $wild = false$. We denote each parameter of algorithm a with an array notation, e.g., $a[eq]$.

Storage costs.

The containers and source model storage costs are simply computed as $\sum_{p \in P} (alg(p)[\hat{c}(F_p)] * \sum_{c \in p} |c|)$ where $\hat{c} = c_s$ for the case of container storage and $\hat{c} = c_a$ for source model storage³. Obviously, c_s and c_a need not to be evaluated on the overall F but solely on F_p , that is the projection of F over the containers of the partition p .

Decompression cost.

In order to evaluate the decompression cost associated with a given compression configuration s , we define three square matrices, E , I and D , having

³ We are not considering here the containers that are not involved in any query in \mathcal{W} . Those do not incur a cost so they can be disregarded in the cost model.

size $(|\mathcal{C}| + 1) \times (|\mathcal{C}| + 1)$. These matrices reflect the comparisons (equality, inequality and prefix-matching equality comparisons, respectively) made in \mathcal{W} among container values or between container values and constants. More formally, the generic element $E_{i,j}$, with $i \neq |\mathcal{C}| + 1$ and $j \neq |\mathcal{C}| + 1$, is the number of equality predicates in \mathcal{W} between ct_i and ct_j not involving prefix-matching, whereas with $i = |\mathcal{C}| + 1$ or $j = |\mathcal{C}| + 1$, it is the number of equality predicates in \mathcal{W} between ct_i and a constant (if $j = |\mathcal{C}| + 1$), or between ct_j and a constant (if $i = |\mathcal{C}| + 1$), not involving prefix-matching. Matrices I and D have the same structure but refer to inequality and prefix-matching comparisons, respectively. Obviously, E , I and D are symmetric.

Considering the generic element of the three matrices, say $M[i, j]$, the associated decompression cost is obviously zero if ct_i and ct_j share the same source model and the algorithm they are compressed with supports the corresponding predicate in the compressed domain. A decompression cost occurs in three cases: (i) ct_i and ct_j are compressed using different algorithms; (ii) ct_i and ct_j are compressed using the same algorithm but different source models; (iii) ct_i and ct_j share the same source model but the algorithm does not support the needed comparison (equality in the case of E , inequality for I and prefix-matching for D) in the compressed domain. For instance, for a generic element $I[i, j]$, in the case of $i \neq j$, $i \neq |\mathcal{C}| + 1$ and $j \neq |\mathcal{C}| + 1$, the cost would be:

- zero, if $ct_i \in p$, $ct_j \in p$, $alg(p)[ineq] = true$;
- $|ct_i| * alg(p')[d_c] + |ct_j| * alg(p'')[d_c]$, if $ct_i \in p'$, $ct_j \in p''$, $p' \neq p''$ (cases (i) and (ii));
- $(|ct_i| + |ct_j|) * alg(p)[d_c]$, if $ct_i \in p$, $ct_j \in p$, $alg(p)[ineq] = false$ (case (iii)).

The decompression cost is calculated by summing up the costs associated with each element of the matrices E , I , and D . However, note that (i) for the cases of E and D , we consider $alg(p)[eq]$ and $alg(p)[wild]$, respectively, and that (ii) the term referring to the cardinality of the containers to be decompressed is adjusted in the cases of self-comparisons (i.e. $i = j$) and comparisons with constants ($i = |\mathcal{C}| + 1$ or $j = |\mathcal{C}| + 1$).

4.3.3 Devising a suitable search strategy

XQueC currently uses a greedy strategy for moving into the search space. The search starts with an initial configuration $s_0 = \langle P_0, alg_0 \rangle$, where P_0 is a partition of \mathcal{C} having sets of exactly one container, and alg_0 blindly assigns to each set a generic compression algorithm (e.g. **bzip**) and a separate source model. Next, s_0 is gradually improved by a sequence of *configuration moves*.

Let $Pred$ be the set of value comparison predicates appearing in \mathcal{W} . A move from $s_k = \langle P_k, alg_k \rangle$ to $s_{k+1} = \langle P_{k+1}, alg_{k+1} \rangle$ is done by first randomly extracting a predicate $pred$ from $Pred$. Let ct_i and ct_j be the containers involved in $pred$ (for instance $pred$ makes an equality comparison,

such as $ct_i = ct_j$, or an inequality one, such as $ct_i > ct_j$). Let p' and p'' the sets in P_k to which ct_i and ct_j belong, respectively. If $p' = p''$, we build a new configuration s' where $alg_{k+1}(p')$ is such that the evaluation of $pred$ is enabled on compressed values, and alg_{k+1} has the greatest number of algorithmic properties holding *true*. Then, we evaluate the costs of s_k and s' , and let s_{k+1} be the one with the minimum cost. In the case of $p' \neq p''$, we build two new configurations s' and s'' . s' is obtained by dropping ct_i and ct_j from p' and p'' , respectively, and adding the set $\{ct_i, ct_j\}$ to P_{k+1} . s'' is obtained by replacing p' and p'' with their union. For both s' and s'' , alg_{k+1} associates to the new sets in P_{k+1} an algorithm enabling the evaluation of $pred$ in the compressed domain and having the greatest number of algorithmic properties holding *true*. Finally, we evaluate the costs of s_k , s' and s'' , and let s_{k+1} be the one with the minimum cost.

Example 4.1. To give a flavor of the savings gained with partitioning the set of containers, consider an initial configuration, which has five containers on an XMark document, all of them sized about 6MB, which we initially (naively) choose to compress with ALM only; let us call this configuration *NaiveConf*. The workload is made of XQuery queries with inequality predicates over the path expressions leading to the above containers. The first three containers are filled with Shakespeare's sentences, the fourth is filled with person names and the fifth with dates. Using the workload we considered, we obtain the best partitioning, which has three partitions, one with the first three containers and a distinct partition for the fourth and fifth, let us call it *GoodConf*. The compression factor shifts from 56,14% for the *NaiveConf* to 67,14%, 71,75% and 65,15% respectively for the three partitions of *GoodConf*. While in such a case the source models sizes do not vary significantly, the decompression cost in *Good Conf* is clearly advantageous w.r.t. *NaiveConf*, leading to gain 21,42% for shakespearean text, 28,57% for person names and to loose only 6% for dates.

Note that, for each predicate in $Pred$, the strategy explores a fixed subset of possible configuration moves, so its complexity is linear in $|Pred|$. Of course, due to this partial exploration, the search yields a locally optimal solution. Moreover, containers not involved in W are not considered by the cost model, and a reasonable choice could be to compress them using order-unaware algorithms offering good compression ratios, e.g. `bzip2` [23]. Finally, note also that the choice of a suitable compression configuration is orthogonal with respect to the choosing of an optimal XML storage model [19]; we can combine both for an automatic storage-and-compression design.

4.4 Evaluating XML queries over compressed data

The XQueC query processor consists of a parser, an optimizer, and a query evaluation engine. The set of physical operators used by the query evaluation engine can be divided in three classes:

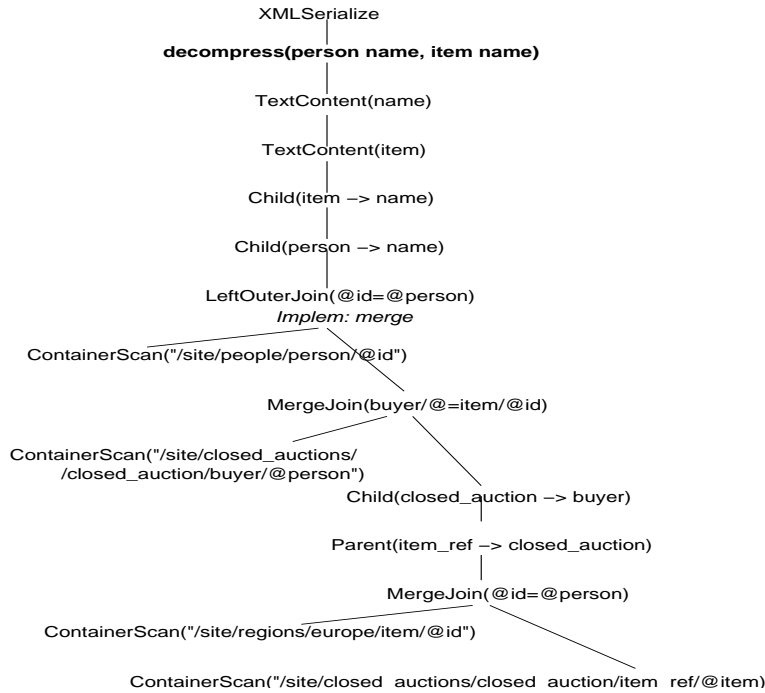


Fig. 4.5. Query execution plan for XMark's Q9.

- *data access operators*, retrieving information from the compressed storage structures;
- *regular data combination operators* (joins, outer joins, selections etc.);
- *compression and decompression operators*.

Among our data access operators, there are *ContScan* and *ContAccess*, which allow, respectively, to scan all (elementID, compressed value) pairs from a container, and to access only some of them, according to an interval search criteria. *StructureSummaryAccess* provide direct access to the identifiers of all elements reachable through a given path. *Parent* and *Child* allow to fetch the parent, respectively, the children (all children, or just those with a specific tag) for a given set of elements. Finally, *TextContent* pairs element IDs with all their immediate text children, retrieved from their respective containers. *TextContent* is implemented as a hash join pairing the element IDs with the content obtained from a *ContScan*.

Due to our choice of a storage model (Section 4.2.2), the *StructureSummaryAccess* operator provides the identifiers of the required elements *in the correct document order*. Furthermore, the *Parent* and *Child* operator preserve the order of the elements with respect to which they are applied. Also, if the *Child* operator returns more than one child for a given node, these chil-

dren are returned in correct order. This order-preserving behavior allow us to perform many path computations through comparatively inexpensive 1-pass merge joins; furthermore, many simple queries can be answered without requiring a sort to re-constitute document order.

While these operators respect document order, *ContScan* and *ContAccess* respect data order, provides fast access to elements (and values) according to a given value search criteria. Also, as soon as predicates on container values are given in the query, it is often profitable to start query evaluation by scanning (and perhaps merge-joining) a few containers.

As an example of QEP, consider query Q9 from XMark:

```

FOR $p IN document("auction.xml")/site/people/person
LET $a :=
  FOR $t IN document("auction.xml")/site/
    closed_auctions/closed_auction
  LET $n :=
    FOR $t2 IN document("auction.xml")/site/
      regions/europe/item
    WHERE $t/itemref/@item = $t2/@id
    RETURN $t2
  WHERE $p/@id = $t/buyer/@person
  RETURN <item> $n/name/text() </item>
RETURN <person name=$p/name/text()> $a </person>

```

Figure 4.5 shows a possible XQueC execution plan for Q9 (actually, this is the plan that we used for the measures in the experimental section). Based on this example, we make several remarks. First, notice that we only decompress the necessary pieces of information (person name and item name), only at the very end of the query execution (the decompress operators shown in bold fonts). All the way down in the QEP, we were able to compute the three-ways join between persons, buyers, and items, using directly the compressed attributes *person/@id*, *buyer/@person*, and *item_ref/@item*. Second, due to the order of data obtained from *ContainerScans*, we are able to make extensive use of MergeJoins, without the need for sorting. Third, this plan mixes *Parent* and *Child* operators, alternating judiciously between top-down and bottom-up strategy, in order to minimize the number of tuples manipulated at any particular moment. This feature is made possible by the usage of a full set of algebraic evaluation choices, which XQueC does, but is not available to the XGrind or XPress query processors.

Finally, note that for instance in query Q9 also an *XMLSerialize* operator is employed in order to correctly construct the new XML which the query outputs. To this purpose, we recall that XML construction plays a minor role within the XML algebraic evaluation, and, being not crucial, it can be disregarded in the whole query execution time [131]. This has been confirmed by our experiments.

XQueC’s query optimizer follows a top-down, transformation based approach, in the spirit of Volcano [51]. The search for a query plan attempts first to “transform” path expressions, performing navigation in the document, into combinations of operators like *ContainerScan*, *Parent*, *Child*, *StructureSummaryAccess* etc. Many combinations are possible, but our optimizer systematically favors query sub-plans whose lowest operators are the most selective and efficient ones (*ContainerAccess* if a selection is specified on the container values, or *StructureSummaryAccess* for direct access to a set of elements on a given path). In doing so, the optimizer analyzes the relationships between the path expressions present in the query and may re-formulate them, i.e., is not bound by the particular syntax used by the query; several equivalent reformulation rules [77] are applied to abstract away from the user-specified syntax, and make the query easier to optimize.

The second step attempts to combine the products of the previous steps through appropriate joins or left outer joins (modelling the existential semantics of some expressions in XQuery). Note that due to the presence of our structural summary, we are able to decide when a relationship between two elements found on the paths t_1 and t_2 is guaranteed to have some t_2 ’s for every t_1 , or when this is not the case. This enables the transformation of some left outer joins (as required by the semantics of XQuery) into direct joins.

4.5 Implementation and experimental evaluation

XQueC is being implemented entirely in Java, using as back-end an embedded database, Berkeley DB [15]. We have performed some interesting regression tests, that show that XQueC is a competitor of both query-aware compressors, and of early XQuery prototypes.

In the following, we want to illustrate both XQueC good compression ratios and query execution times. To this purpose, we have done two kinds of experiments:

Compression Factors. We have considered experiments on both synthetic data (XMark documents) and on real-life data sets (in particular, we considered the ones chosen in [106] for the purpose of cross-comparison with it).

Query Execution Times. We show how our system performs on some XML benchmark queries [128] and cross-compare them with the query execution times of optimized Galax [96], an open-source XQuery prototype.

4.5.1 Compression Factors

We have compared the obtained compression factors (defined as $1 - (cs/os)$), where cs and os are the sizes of the compressed and original documents, respectively) with the corresponding factors of XMill, XGrind and XPRESS.

Document	Size(MB)	Containers	Distinct tags	Tree nodes
Shakespeare	15.0	39	22	65621
Baseball	16.8	41	46	27181
Washington-course	12.1	12	18	99729
xmark11	11.3	432	77	76726

Table 4.1. Data Sets used in the experiments (XMark11 is used in QETs measures.)

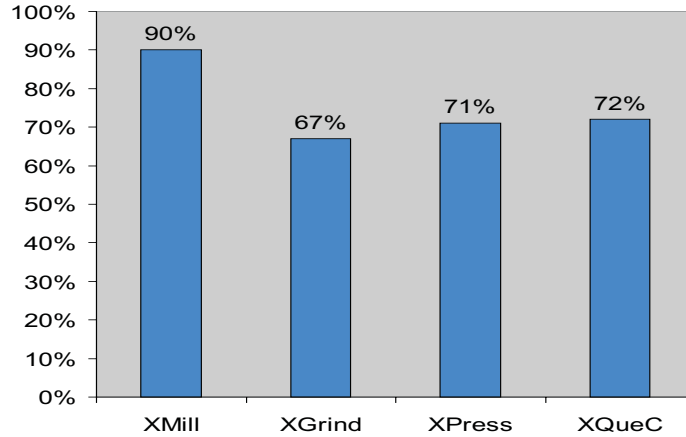


Fig. 4.6. Average compression factor for Shakespeare, WashingtonCourse and Baseball data sets.

Figure 4.6 shows the average compression factor obtained for a corpus of documents composed of *Shakespeare.xml*, *Washington-Course.xml* and *Baseball.xml*, whose main characteristics are shown in Table 4.1. Note that, on average, XQueC closely tracks XPRESS. It is interesting to notice that some limitations affect some of the XML compressors that we tested - for example, the documents decompressed by XPRESS have lost all their white spaces. Thus, the XQueC compression factor could be smoothed if blanks were not considered.

Moreover, we have also tested the compression factors on different-sized XMark synthetic data sets (we considered documents ranging from 1MB to 25MB), generated by means of *xmlgen* [128]. As Figure 4.7 shows, we have obtained again good compression factors w.r.t XPRESS and XMill.

Note also that XGrind does not appear in these experiments. Indeed, due to repetitive crashes, we were not able to upload in the XGrind system (the version available through the site <http://sourceforge.net>) any XMark document except for one sized 100KB, whose compression factor however is very low and not representative of the system (precisely equal to 17,36%).

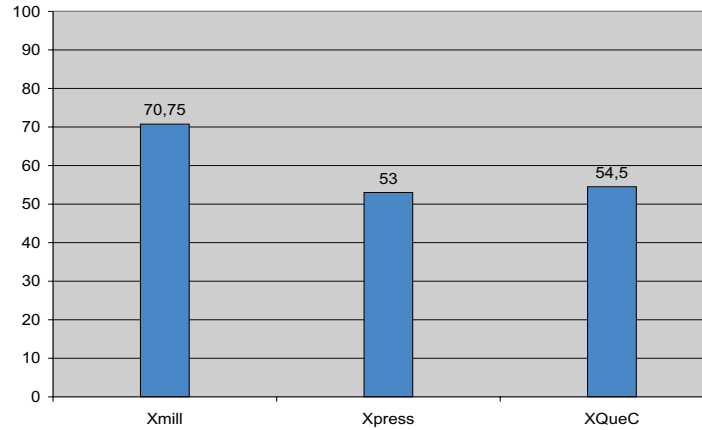


Fig. 4.7. Average compression factor (%) for XMark synthetic data sets.

4.5.2 Query Execution Times

We have tested our system against the optimized version of Galax by running XMark queries and other queries. Due to space limits, we select here a set of significant XMark queries. Indeed, XMark queries left out stress language features, on which compression will likely have no significant impact whatsoever, e.g., support for functions, deep nesting etc. The reasons why we chose Galax is that it is open-source and has an optimizer. Note that XQueC does not have an optimizer by the time being, thus the results we gathered are only due to the compression ratios and to our proper data structures.

Figure 4.8 shows the executions of XQueC queries on the document XMark11, sized 11.3MB. For the sake of better readability, in Figure 4.8, we have omitted Q9, and Q8. These queries measured in our system 2,133 sec. and 2,142 sec. respectively, whereas in Galax Q9 could not be measured on our machine and Q8 took 126,33 sec. Note also that on Q2, Q3, Q16, the QET is a little worse than the Galax one, because in the current implementation we use simple unique IDs, given that our data model imposes a large number of parent-child joins. However, even with this limitation, we are still reasonably close to Galax. Most importantly, note that the previous XQueC QETs are to be intended as the times taken to both execute the queries in the compressed and decompress the obtained results. Thus, those measures show that there is no performance penalty in XQueC w.r.t. Galax due to compression. Thus, with comparable times w.r.t. an XQuery engine over uncompressed data, XQueC exhibits the advantage of compression.

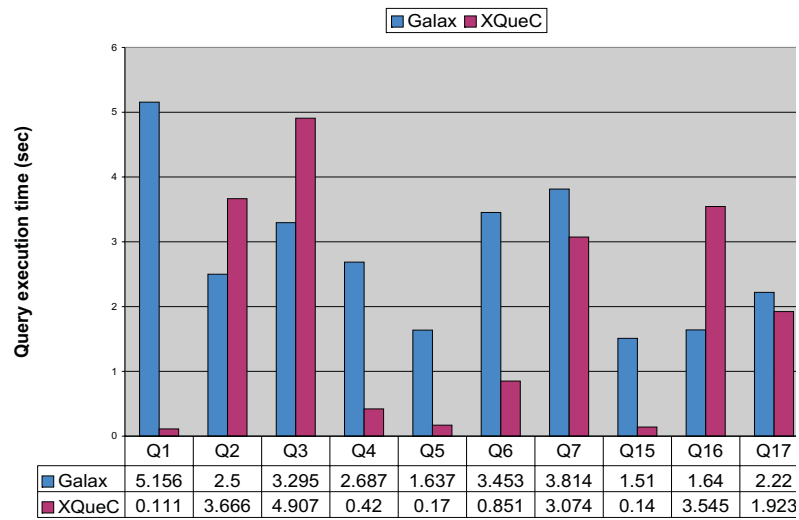


Fig. 4.8. Comparative execution times between us and Optimized Galax .

Data Mining: Proximity measures and cluster analysis

5.1 Data Mining

Progress in digital data acquisition and storage technology has resulted in the growth of huge databases. This has occurred in all areas of human endeavor, from the mundane (such as supermarket transaction data, credit card usage records, telephone call details, and government statistics) to the more exotic (such as images of astronomical bodies, molecular databases, and medical records). Little wonder, then, that interest has grown in the possibility of tapping these data, of extracting from them information that might be of value to the owner of the database. The discipline concerned with this task has become known as *data mining*.

5.1.1 Introduction

Defining a scientific discipline is always a controversial task; researchers often disagree about the precise range and limits of their field of study. Bearing this in mind, and without trying to cover all possible approaches and all different views about data mining as a discipline, let us start with one possible, sufficiently broad definition of data mining:

Data mining is the analysis of (often large) observational data sets to find unsuspected relationships and to summarize the data in novel ways that are both understandable and useful to the data owner.

The relationships and summaries derived through a data mining exercise are often referred to as *models* or *patterns*. Examples include linear equations, rules, clusters, graphs, tree structures, and recurrent patterns in time series.

The definition above refers to “observational data”, as opposed to “experimental data”. Data mining typically deals with data that have already been collected for some purpose other than the data mining analysis (for example, they may have been collected in order to maintain an up-to-date record of all

the transactions in a bank). This means that the objectives of the data mining exercise play no role in the data collection strategy. This is one way in which data mining differs from much of statistics, in which data are often collected by using efficient strategies to answer specific questions. For this reason, data mining is often referred to as “secondary” data analysis.

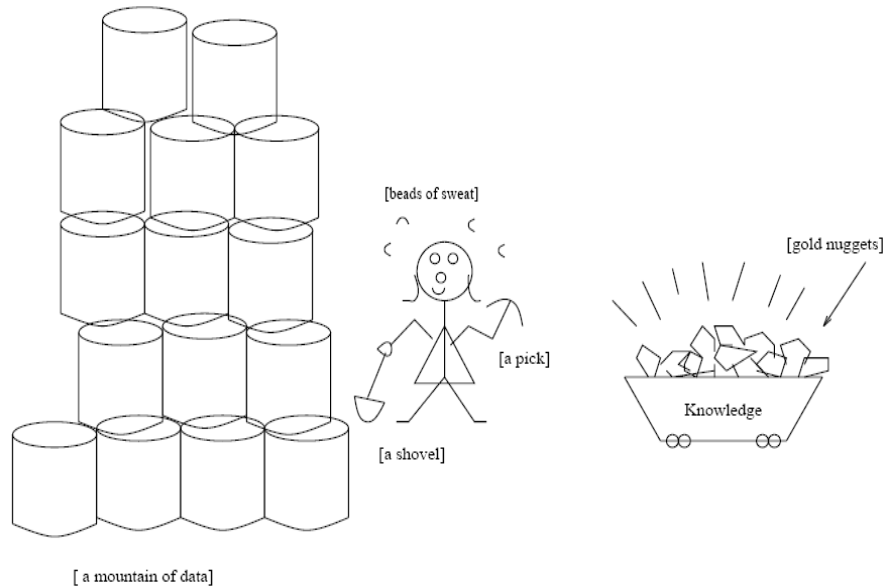


Fig. 5.1. Data mining - searching for knowledge (interesting patterns) in data

The definition also mentions that the data sets examined in data mining are often large. If only small data sets were involved, we would merely be discussing classical exploratory data analysis as practiced by statisticians. When we are faced with large bodies of data, new problems arise. Some of these relate to housekeeping issues of how to store or access the data, but others relate to more fundamental issues, such as how to determine the representativeness of the data, how to analyze the data in a reasonable period of time, and how to decide whether an apparent relationship is merely a chance occurrence not reflecting any underlying reality. Often the available data comprise only a sample from the complete population (or, perhaps, from a hypothetical super-population); the aim may be to *generalize* from the sample to the population. For example, we might wish to predict how future customers are likely to behave or to determine the properties of protein structures that we have not yet seen. Such generalizations may not be achievable through standard statistical approaches because often the data are not (classical statistical) “random samples”, but rather “convenience” or “opportunity” samples. Sometimes we

may want to summarize or *compress* a very large data set in such a way that the result is more comprehensible, without any notion of generalization. This issue would arise, for example, if we had complete census data for a particular country or a database recording millions of individual retail transactions.

The relationships and structures found within a set of data must, of course, be novel. There is little point in regurgitating well-established relationships (unless, the exercise is aimed at “hypothesis“ confirmation, in which one was seeking to determine whether established pattern also exists in a new data set) or necessary relationships (that, for example, all pregnant patients are female). Clearly, novelty must be measured relative to the user’s prior knowledge. Unfortunately few data mining algorithms take into account a user’s prior knowledge. For this reason we will not say very much about novelty in this text. It remains an open research problem.

While novelty is an important property of the relationships we seek, it is not sufficient to qualify a relationship as being worth finding. In particular, the relationships must also be understandable. For instance simple relationships are more readily understood than complicated ones, and may well be preferred, all else being equal.

5.1.2 Data mining as a process of knowledge discovery

It is important to realize that the problem of discovering or estimating dependencies from data or discovering totally new data is only one part of the general experimental procedure used by scientists, engineers, and others who apply standard steps to draw conclusions from the data.

Data mining is often set in the broader context of *knowledge discovery in databases*, or KDD. This term originated in the artificial intelligence (AI) research field. Knowledge discovery as a process is depicted in Figure 5.2, and consists of an iterative sequence of the following steps:

- **data cleaning** (to remove noise or irrelevant data)
- **data integration** (where multiple data sources may be combined)
- **data selection** (where data relevant to the analysis task are retrieved from the database)
- **data transformation** (where data are transformed or consolidated into forms appropriate for mining by performing summary or aggregation operations, for instance)
- **data mining** (an essential process where intelligent methods are applied in order to extract data patterns)
- **pattern evaluation** (to identify the truly interesting patterns representing knowledge based on some interestingness measures)
- **knowledge presentation** (where visualization and knowledge representation techniques are used to present the mined knowledge to the user)

The data mining step may interact with the user or a knowledge base. The interesting patterns are presented to the user, and may be stored as

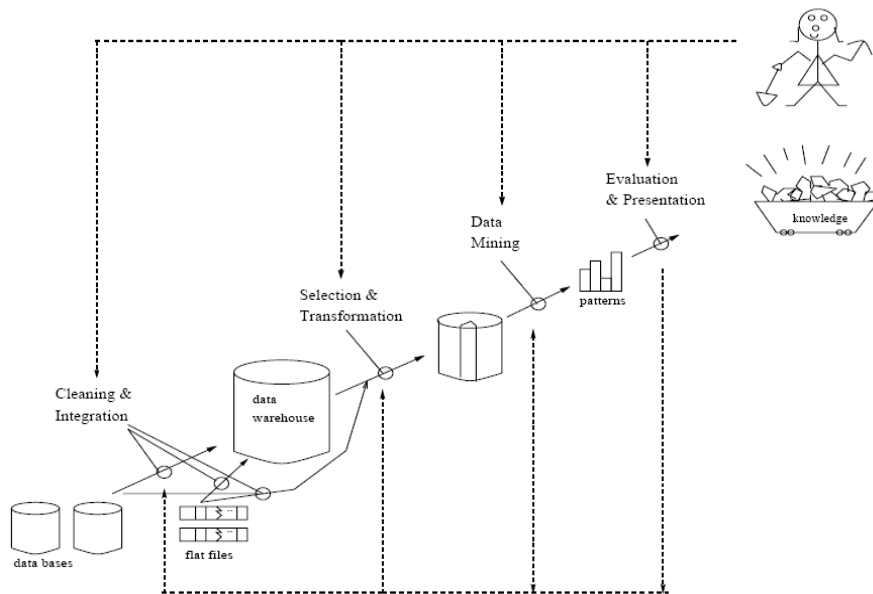


Fig. 5.2. Data mining as a process of knowledge discovery

new knowledge in the knowledge base. Note that according to this view, data mining is only one step in the entire process, albeit an essential one since it uncovers hidden patterns for evaluation. We agree that data mining is a knowledge discovery process. However, in industry, in media, and in the database research milieu, the term “data mining” is becoming more popular than the longer term of “knowledge discovery in databases”.

Based on this view, the architecture of a typical data mining system may have the following major components (Figure 5.2):

1. **Database, data warehouse, or other information repository.** This is one or a set of databases, data warehouses, spread sheets, or other kinds of information repositories. Data cleaning and data integration techniques may be performed on the data.
2. **Database or data warehouse server.** The database or data warehouse server is responsible for fetching the relevant data, based on the user’s data mining request.
3. **Knowledge base.** This is the domain knowledge that is used to guide the search, or evaluate the interestingness of resulting patterns. Such knowledge can include concept hierarchies, used to organize attributes or attribute values into different levels of abstraction. Knowledge such as user beliefs, which can be used to assess a pattern’s interestingness based on its unexpectedness, may also be included. Other examples of domain knowl-

edge are additional interestingness constraints or thresholds, and meta-data (e.g., describing data from multiple heterogeneous sources).

4. **Data mining engine.** This is essential to the data mining system and ideally consists of a set of functional modules for tasks such as characterization, association analysis, classification, evolution and deviation analysis.
5. **Pattern evaluation module.** This component typically employs interestingness measures and interacts with the data mining modules so as to focus the search towards interesting patterns. It may access interestingness thresholds stored in the knowledge base. Alternatively, the pattern evaluation module may be integrated with the mining module, depending on the implementation of the data mining method used. For efficient data mining, it is highly recommended to push the evaluation of pattern interestingness as deep as possible into the mining process so as to confine the search to only the interesting patterns.
6. **Graphical user interface.** This module communicates between users and the data mining system, allowing the user to interact with the system by specifying a data mining query or task, providing information to help focus the search, and performing exploratory data mining based on the intermediate data mining results. In addition, this component allows the user to browse database and data warehouse schemas or data structures, evaluate mined patterns, and visualize the patterns in different forms.

Data mining involves an integration of techniques from multiple disciplines such as database technology, statistics, machine learning, high performance computing, pattern recognition, neural networks, data visualization, information retrieval, image and signal processing, and spatial data analysis. By performing data mining, interesting knowledge, regularities, or high-level information can be extracted and viewed or browsed from different angles. The discovered knowledge can be applied to decision making, process control, information management, query processing, and so on.

5.2 Measures of similarity and dissimilarity

Similarity and dissimilarity are important because they are used by a number of data mining techniques, such as clustering, nearest neighbor classification, and anomaly detection. In many cases, the initial data set is not needed once these similarities or dissimilarities have been computed. Such approaches can be viewed as transforming the data to a similarity (dissimilarity) space and then performing the analysis.

We begin with a discussion of the basics high-level definitions of similarity and dissimilarity, and a discussion of how they are related. For convenience, the term *proximity* is used to refer to either similarity or dissimilarity. Since the proximity between two objects is a function of the proximity between the

corresponding attributes of the two objects, we first describe how to measure the proximity between objects having only one simple attribute, and then consider proximity measures for objects with multiple attributes. This includes measures such as correlation and Euclidean distance, which are useful for dense data such as time series or two-dimensional points, as well as the Jaccard and cosine similarity measures, which are useful for sparse data like documents.

5.2.1 Basics

Definitions

Informally, the *similarity* between two objects is a numerical measure of the degree to which the two objects are alike. Consequently, similarities are higher for pairs of objects that are more alike. Similarities are usually non-negative and are often between 0 (no similarity) and 1 (complete similarity). The *dissimilarity* between two objects is a numerical measure of the degree to which the two objects are different. Dissimilarities are lower for more similar pairs of objects. Frequently, the term *distance* is used as a synonym for dissimilarity, although distance is often used to refer to a special class of dissimilarities. Dissimilarities sometimes fall in the interval $[0, 1]$, but it is also common for them to range from 0 to ∞ .

Transformations

Transformations are often applied to convert a similarity to a dissimilarity, or vice versa, or to transform a proximity measure to fall within a particular range, such as $[0, 1]$. For instance, we may have similarities that range from 1 to 10, but the particular algorithm or software package that we want to use may be designed to only work with dissimilarities, or it may only work with similarities in the interval $[0, 1]$.

Frequently, proximity measures, especially similarities, are defined or transformed to have values in the interval $[0, 1]$. Informally, the motivation for this is to use a scale in which a proximity value indicates the fraction of similarity (or dissimilarity) between two objects. Such a transformation is often relatively straightforward. For example, if the similarities between objects range from 1 (not at all similar) to 10 (completely similar), we can make them fall within the range $[0, 1]$ by using the transformation $s' = (s - 1)/9$, where s and s' are the original and new similarity values, respectively. In the more general case, the transformation of similarities to the interval $[0, 1]$ is given by the expression $s' = (s - \min_s)/(max_s - \min_s)$, where max_s and \min_s are the maximum and minimum similarity values, respectively. Likewise, dissimilarity measures with a finite range can be mapped to the interval $[0, 1]$ by using the formula $d' = (d - \min_d)/(max_d - \min_d)$.

There can be various complications in mapping proximity measures to the interval $[0, 1]$, however. If, for example, the proximity measure originally takes values in the interval $[0, \infty]$, then a non-linear transformation is needed and values will not have the same relationship to one another on the new scale. Consider the transformation $d' = d/(1 + d)$ for a dissimilarity measure that ranges from 0 to ∞ . The dissimilarities 0, 0.5, 2, 10, 100, and 1000 will be transformed into the new dissimilarities 0, 0.33, 0.67, 0.90, 0.99, and 0.999, respectively. Larger values on the original dissimilarity scale are compressed into the range of values near 1, but whether or not this is desirable depends on the application. Another complication is that the meaning of the proximity measure may be changed. For example, correlation, which is discussed later, is a measure of similarity that takes values in the interval $[-1, 1]$. Mapping these values to the interval $[0, 1]$ by taking the absolute value loses information about the sign, which can be important in some applications.

Transforming similarities to dissimilarities and vice versa is also relatively straightforward, although we again face the issues of preserving meaning and changing a linear scale into a non-linear scale. If the similarity (or dissimilarity) falls in the interval $[0, 1]$, then the dissimilarity can be defined as $d = 1 - s$ ($s = 1 - d$). Another simple approach is to define similarity as the negative of the dissimilarity (or vice versa). To illustrate, the dissimilarities 0, 1, 10, and 100 can be transformed into the similarities 0, -1 , -10 , and -100 , respectively.

The similarities resulting from the negation transformation are not restricted to the range $[0, 1]$, but if that is desired, then transformations such as $s = \frac{1}{d+1}$, $s = e^{-d}$, or $s = 1 - \frac{d - \min_d}{\max_d - \min_d}$ can be used. For the transformation $s = \frac{1}{d+1}$, the dissimilarities 0, 1, 10, 100 are transformed into 1, 0.5, 0.09, 0.01, respectively. For $s = e^{-d}$, they become 1.00, 0.37, 0.00, 0.00, respectively, while for $s = 1 - \frac{d - \min_d}{\max_d - \min_d}$ they become 1.00, 0.99, 0.00, 0.00, respectively.

In general, any monotonic decreasing function can be used to convert dissimilarities to similarities, or vice versa. Of course, other factors also must be considered when transforming similarities to dissimilarities, or vice versa, or when transforming the values of a proximity measure to a new scale. We have mentioned issues related to preserving meaning, distortion of scale, and requirements of data analysis tools, but this list is certainly not exhaustive.

5.2.2 Similarity and Dissimilarity between Simple Attributes

The proximity of objects with a number of attributes is typically defined by combining the proximities of individual attributes, and thus, we first discuss proximity between objects having a single attribute. Consider objects described by one nominal attribute. What would it mean for two such objects to be similar? Since nominal attributes only convey information about distinctness of objects, all we can say is that two objects either have the same value or they do not. Hence, in this case similarity is traditionally defined 1

if attribute values match, and 0 otherwise. A dissimilarity would be defined in the opposite way: 0 if the attribute values match, and 1 if they do not.

For objects with a single ordinal attribute, the situation is more complicated because information about order should be taken into account. Consider an attribute that measures the quality of a product, e.g., a candy bar, on scale $\{poor, fair, OK, good, wonderful\}$. It would seem reasonable that a product, P_1 , which is rated *wonderful*, would be closer to a product P_2 , which is rated *good*, than it would be to a product P_3 , which is rated *OK*. To make this observation quantitative, the values of the ordinal attribute are often mapped to successive integers, beginning at 0 or 1, e.g., $\{poor=0, fair=1, OK=2, good=3, wonderful=4\}$. Then, $d(P_1, P_2) = 3 - 2 = 1$ or, if we want the dissimilarity to fall between 0 and 1, $d(P_1, P_2) = \frac{3-2}{4} = 0.25$. A similarity for ordinal attributes can then be defined as $s = 1 - d$.

This definition of similarity (dissimilarity) for an ordinal attribute can seem a bit strange since this assumes equal intervals, and this is not so. Otherwise, we would have an interval or ratio attribute. Is the difference between the values fair and good really the same as that between the values OK and wonderful? Probably not, but in practice, our options are limited, and in the absence of more information, this is the standard approach for defining proximity between ordinal attributes.

For interval or ratio attributes, the natural measure of dissimilarity between two objects is the absolute difference of their values. For example, we might compare our current weight and our weight a year ago by saying “I am ten pounds heavier”. In cases such as these, the dissimilarities typically range from 0 to ∞ , rather than from 0 to 1. The similarity of interval or ratio attributes is typically expressed by transforming a similarity into a dissimilarity, as previously described.

5.2.3 Dissimilarities between Data Objects

We discuss various kinds of dissimilarities. We begin with a discussion of distances, which are dissimilarities with certain properties, and then provide examples of more general kinds of dissimilarities.

Distances

We first present some examples, and then offer a more formal description of distances in terms of the properties common to all distances. The *Euclidean distance*, d , between two points, \mathbf{x} and \mathbf{y} , in one-, two-, three-, or higher dimensional space, is given by the following familiar formula:

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{k=1}^n (x_k - y_k)^2} \quad (5.1)$$

where n is the number of dimensions and x_k and y_k are, respectively, the k -th attributes (components) of \mathbf{x} and \mathbf{y} .

The Euclidean distance measure given in Equation 5.1 is generalized by the *Minkowski* distance metric shown in Equation 5.2,

$$d(\mathbf{x}, \mathbf{y}) = \left(\sum_{k=1}^n |x_k - y_k|^r \right)^{1/r} \quad (5.2)$$

where r is a parameter. The following are the three most common examples of Minkowski distances:

- $r = 1$. City block (Manhattan, taxicab, L_1 norm) distance. A common example is the Hamming distance, which is the number of bits that are different between two objects that have only binary attributes, i.e., between two binary vectors.
- $r = 2$. Euclidean distance (L_2 norm).
- $r = \infty$. Supremum (L_{max} or L_∞ norm) distance. This is the maximum difference between any attribute of the objects. More formally, the L_∞ distance is defined by Equation 5.3

$$d(\mathbf{x}, \mathbf{y}) = \lim_{r \rightarrow \infty} \left(\sum_{k=1}^n |x_k - y_k|^r \right)^{1/r} \quad (5.3)$$

The r parameter should not be confused with the number of dimensions (attributes) n . The Euclidean, Manhattan, and supremum distances are defined for all values of $n : 1, 2, 3, \dots$, and specify different ways of combining the differences in each dimension (attribute) into an overall distance.

Distances, such as the Euclidean distance, have some well-known properties. If $d(\mathbf{x}, \mathbf{y})$ is the distance between two points, \mathbf{x} and \mathbf{y} , then the following properties hold:

1. **Positivity**
 - $d(\mathbf{x}, \mathbf{y}) \geq 0$ for all \mathbf{x} and \mathbf{y} ,
 - $d(\mathbf{x}, \mathbf{y}) = 0$ only if $\mathbf{x} = \mathbf{y}$.
2. **Symmetry**
 - $d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x})$ for all \mathbf{x} and \mathbf{y} .
3. **Triangle Inequality**
 - $d(\mathbf{x}, \mathbf{z}) \leq d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z})$ for all \mathbf{x}, \mathbf{y} and \mathbf{z} .

Measures that satisfy all three properties are known as *metrics*. Some people only use the term distance for dissimilarity measures that satisfy these properties, but that practice is often violated. The three properties described here are useful, as well as mathematically pleasing. Also, if the triangle inequality holds, then this property can be used to increase the efficiency of techniques (including clustering) that depend on distances possessing this property. Nonetheless, many dissimilarities do not satisfy one or more of the metric properties. We give an example of such measures.

Example 5.1. Non-metric Dissimilarities: Set Differences

This example is based on the notion of the difference of two sets, as defined in set theory. Given two sets A and B , $A - B$ is the set of elements of A that are not in B . For example, if $A = \{1, 2, 3, 4\}$ and $B = \{2, 3, 4\}$, then $A - B = \{1\}$ and $B - A = \emptyset$. We can define the distance d between two sets A and B as $d(A, B) = \text{size}(A - B)$, where size is a function returning the number of elements in a set. This distance measure, which is an integer value greater than or equal to 0, does not satisfy the second part of the positivity property, the symmetry property, or the triangle inequality. However, these properties can be made to hold if the dissimilarity measure is modified as follows: $d(A, B) = \text{size}(A - B) + \text{size}(B - A)$.

5.2.4 Similarities between Data Objects

For similarities, the triangle inequality (or the analogous property) typically does not hold, but symmetry and positivity typically do. To be explicit, if $s(\mathbf{x}, \mathbf{y})$ is the similarity between points \mathbf{x} and \mathbf{y} , then the typical properties of similarities are the following:

1. $s(\mathbf{x}, \mathbf{y}) = 1$ only if $\mathbf{x} = \mathbf{y}$. ($0 \leq s \leq 1$)
2. $s(\mathbf{x}, \mathbf{y}) = s(\mathbf{y}, \mathbf{x})$ for all \mathbf{x} and \mathbf{y} . (Symmetry)

There is no general analog of the triangle inequality for similarity measures. It is sometimes possible, however, to show that a similarity measure can easily be converted to a metric distance. The cosine and Jaccard similarity measures, which are discussed shortly, are two examples. Also, for specific similarity measures, it is possible to derive mathematical bounds on the similarity between two objects that are similar in spirit to the triangle inequality.

Example 5.2. A Non-symmetric Similarity Measure

Consider an experiment in which people are asked to classify a small set of characters as they flash on a screen. The *confusion matrix* for this experiment records how often each character is classified as itself, and how often each is classified as another character. For instance, suppose that 0 appeared 200 times and was classified as a 0 160 times, but as an o 40 times. Likewise, suppose that o appeared 200 times and was classified as an o 170 times, but as 0 only 30 times. If we take these counts as a measure of the similarity between two characters, then we have a similarity measure, but one that is not symmetric. In such situations, the similarity measure is often made symmetric by setting $s'(\mathbf{x}, \mathbf{y}) = s'(\mathbf{y}, \mathbf{x}) = (s(\mathbf{x}, \mathbf{y}) + s(\mathbf{y}, \mathbf{x}))/2$, where s' indicates the new similarity measure.

5.2.5 Examples of Proximity Measures

We provide specific examples of some similarity and dissimilarity measures.

Similarity Measures for Binary Data

Similarity measures between objects that contain only binary attributes are called similarity coefficients, and typically have values between 0 and 1. A value of 1 indicates that the two objects are completely similar, while a value of 0 indicates that the objects are not at all similar. There are many rationales for why one coefficient is better than another in specific instances.

Let \mathbf{x} and \mathbf{y} be two objects that consist of n binary attributes. The comparison of two such objects, i.e., two binary vectors, leads to the following four quantities (frequencies):

f_{00} = the number of attributes where \mathbf{x} is 0 and \mathbf{y} is 0

f_{01} = the number of attributes where \mathbf{x} is 0 and \mathbf{y} is 1

f_{10} = the number of attributes where \mathbf{x} is 1 and \mathbf{y} is 0

f_{11} = the number of attributes where \mathbf{x} is 1 and \mathbf{y} is 1

Simple Matching Coefficient

One commonly used similarity coefficient is the simple matching coefficient (*SMC*), which is defined as:

$$SMC = \frac{\text{number of matching attribute values}}{\text{number of attribute}} = \frac{f_{11} + f_{00}}{f_{01} + f_{10} + f_{11} + f_{00}}$$

This measure counts both presences and absences equally. Consequently, the *SMC* could be used to find students who had answered questions similarly on a test that consisted only of true/false questions

Jaccard Coefficient

Suppose that \mathbf{x} and \mathbf{y} are data objects that represent two rows (two transactions) of a transaction matrix. If each asymmetric binary attribute corresponds to an item in a store, then a 1 indicates that the item was purchased, while a 0 indicates that the product was not purchased. Since the number of products not purchased by any customer far outnumber the number of products that were purchased, a similarity measure such as *SMC* would say that all transactions are very similar. As a result, the *Jaccard* coefficient is frequently used to handle objects consisting of asymmetric binary attributes. The Jaccard coefficient, which is often symbolized by J , is given by the following equation:

$$J = \frac{\text{number of matching presences}}{\text{number of attribute not involved in 00 matches}} = \frac{f_{11}}{f_{01} + f_{10} + f_{11}}$$

Example 5.3. The *SMC* and Jaccard Similarity Coefficients

To illustrate the difference between these two similarity measures, we calculate SMC and J for the following two binary vectors.

$$\mathbf{x} = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0)$$

$$\mathbf{y} = (0, 0, 0, 0, 0, 0, 1, 0, 0, 1)$$

$f_{00} = 7$ the number of attributes where \mathbf{x} is 0 and \mathbf{y} is 0

$f_{01} = 2$ the number of attributes where \mathbf{x} is 0 and \mathbf{y} is 1

$f_{10} = 1$ the number of attributes where \mathbf{x} is 1 and \mathbf{y} is 0

$f_{11} = 0$ the number of attributes where \mathbf{x} is 1 and \mathbf{y} is 1

$$SMC = \frac{f_{11} + f_{00}}{f_{01} + f_{10} + f_{11} + f_{00}} = \frac{0 + 7}{2 + 1 + 0 + 7} = 0.7$$

$$J = \frac{f_{11}}{f_{01} + f_{10} + f_{11}} = \frac{0}{2 + 1 + 0} = 0$$

Cosine Similarity

Documents are often represented as vectors, where each attribute represents the frequency with which a particular term (word) occurs in the document. It is more complicated than this, of course, since certain common words are ignored and various processing techniques are used to account for different forms of the same word, differing document lengths, and different word frequencies.

Even though documents have thousands or tens of thousands of attributes (terms), each document is sparse since it has relatively few non-zero attributes. (The normalizations used for documents do not create a non-zero entry where there was a zero entry; i.e., they preserve sparsity.) Thus, as with transaction data, similarity should not depend on the number of shared 0 values since any two documents are likely to “not contain” many of the same words, and therefore, if 0-0 matches are counted, most documents will be highly similar to most other documents. Therefore, a similarity measure for documents needs to ignore 0-0 matches like the Jaccard measure, but also must be able to handle non-binary vectors. The cosine similarity, defined next, is one of the most common measure of document similarity. If \mathbf{x} and \mathbf{y} are two document vectors, then

$$\cos(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} \quad (5.4)$$

where \cdot indicates the vector dot product, $\mathbf{x} \cdot \mathbf{y} = \sum_{k=1}^n x_k y_k$, and $\|\mathbf{x}\|$ is the length of vector \mathbf{x} , $\|\mathbf{x}\| = \sqrt{\sum_{k=1}^n x_k^2} = \sqrt{\mathbf{x} \cdot \mathbf{x}}$

Example 5.4. Cosine Similarity of Two Document Vectors

This example calculates the cosine similarity for the following two data objects, which might represent document vectors:

$$\mathbf{x} = (3, 2, 0, 5, 0, 0, 0, 2, 0, 0)$$

$$\mathbf{y} = (1, 0, 0, 0, 0, 0, 0, 1, 0, 2)$$

$$\begin{aligned} \mathbf{x} \cdot \mathbf{y} &= 3 * 1 + 2 * 0 + 0 * 0 + 5 * 0 + 0 * 0 + 0 * 0 + 0 * 0 + 2 * 1 + 0 * 0 + 0 * 2 = 5 \\ \|\mathbf{x}\| &= \sqrt{3 * 3 + 2 * 2 + 0 * 0 + 5 * 5 + 0 * 0 + 0 * 0 + 0 * 0 + 2 * 2 + 0 * 0 + 0 * 0} = 6.48 \\ \|\mathbf{y}\| &= \sqrt{1 * 1 + 0 * 0 + 0 * 0 + 0 * 0 + 0 * 0 + 0 * 0 + 0 * 0 + 1 * 1 + 0 * 0 + 2 * 2} = 2.24 \\ \cos(\mathbf{x}, \mathbf{y}) &= 0.31 \end{aligned}$$

As indicated by Figure 5.3, cosine similarity really is a measure of the (cosine of the) angle between \mathbf{x} and \mathbf{y} . Thus, if the cosine similarity is 1, the angle between \mathbf{x} and \mathbf{y} is 0° , and \mathbf{x} and \mathbf{y} are the same except for magnitude (length). If the cosine similarity is 0, then the angle between \mathbf{x} and \mathbf{y} is 90° , and they do not share any terms (words).

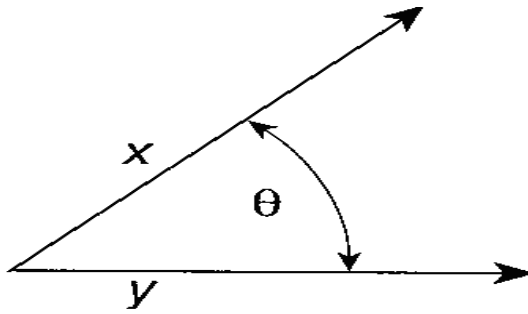


Fig. 5.3. Geometric illustration of the cosine measure

Equation 5.4 can be written as:

$$\cos(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x}}{\|\mathbf{x}\|} \cdot \frac{\mathbf{y}}{\|\mathbf{y}\|} = \mathbf{x}' \cdot \mathbf{y}' \quad (5.5)$$

where $\mathbf{x}' = \frac{\mathbf{x}}{\|\mathbf{x}\|}$ and $\mathbf{y}' = \frac{\mathbf{y}}{\|\mathbf{y}\|}$. Dividing \mathbf{x} and \mathbf{y} by their lengths normalizes them to have a length of 1. This means that cosine similarity does not take the magnitude of the two data objects into account when computing similarity. (Euclidean distance might be a better choice when magnitude is important). For vectors with a length of 1, the cosine measure can be calculated by taking a simple dot product. Consequently, when many cosine similarities between objects are being computed, normalizing the objects to have unit length can reduce the time required.

Extended Jaccard Coefficient (Tanimoto Coefficient)

The extended Jaccard coefficient can be used for document data and that reduces to the Jaccard coefficient in the case of binary attributes. The extended Jaccard coefficient is also known as the Tanimoto coefficient. (However, there is another coefficient that is also known as the Tanimoto coefficient.) This coefficient, which we shall represent as EJ , is defined by the following equation:

$$EJ(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{(\|\mathbf{x}\|)^2 + (\|\mathbf{y}\|)^2 - \mathbf{x} \cdot \mathbf{y}} \quad (5.6)$$

Correlation

The correlation between two data objects that have binary or continuous variables is a measure of the linear relationship between the attributes of the objects. (The calculation of correlation between attributes, which is more common, can be defined similarly.) More precisely, *Pearson's correlation* coefficient between two data objects, \mathbf{x} and \mathbf{y} , is defined by the following equation:

$$\text{corr}(\mathbf{x}, \mathbf{y}) = \frac{\text{covariance}(\mathbf{x}, \mathbf{y})}{\text{standard_deviation}(\mathbf{x}) * \text{standard_deviation}(\mathbf{y})} = \frac{s_{xy}}{s_x s_y} \quad (5.7)$$

where we are using the following standard statistical notation and definitions:

$$\text{covariance}(\mathbf{x}, \mathbf{y}) = s_{xy} = \frac{1}{n-1} \sum_{k=1}^n (x_k - \bar{x})(y_k - \bar{y}) \quad (5.8)$$

$$\text{standard_deviation}(\mathbf{x}) = s_x = \sqrt{\frac{1}{n-1} \sum_{k=1}^n (x_k - \bar{x})^2}$$

$$\text{standard_deviation}(\mathbf{y}) = s_y = \sqrt{\frac{1}{n-1} \sum_{k=1}^n (y_k - \bar{y})^2}$$

$$\bar{x} = \frac{1}{n} \sum_{k=1}^n (x_k) \text{ is the mean of } \mathbf{x}$$

$$\bar{y} = \frac{1}{n} \sum_{k=1}^n (y_k) \text{ is the mean of } \mathbf{y}$$

Example 5.5. Perfect Correlation

Correlation is always in the range -1 to 1 . A correlation of 1 (-1) means that \mathbf{x} and \mathbf{y} have a perfect positive (negative) linear relationship; that is, $x_k = ay_k + b$, where a and b are constants. The following two sets of values

for \mathbf{x} and \mathbf{y} indicate cases where the correlation is -1 and $+1$, respectively. In the first case, the means of \mathbf{x} and \mathbf{y} were chosen to be 0, for simplicity.

$$\mathbf{x} = (-3, 6, 0, 3, -6)$$

$$\mathbf{y} = (1, -2, 0, -1, 2)$$

$$\mathbf{x} = (3, 6, 0, 3, 6)$$

$$\mathbf{y} = (1, 2, 0, 1, 2)$$

5.2.6 Selecting the right proximity Measure

The following are a few general observations that may be helpful. First, the type of proximity measure should fit the type of data. For many types of dense, continuous data, metric distance measures such as Euclidean distance are often used. Proximity between continuous attributes is most often expressed in terms of differences, and distance measures provide a well-defined way of combining these differences into an overall proximity measure. Although attributes can have different scales and be of differing importance, these issues can often be dealt with as described earlier.

For sparse data, which often consists of asymmetric attributes, we typically employ similarity measures that ignore 0-0 matches. Conceptually, this reflects the fact that, for a pair of complex objects, similarity depends on the number of characteristics they both share, rather than the number of characteristics they both lack. More specifically, for sparse, asymmetric data, most objects have only a few of the characteristics described by the attributes, and thus, are highly similar in terms of the characteristics they do not have. The cosine, Jaccard, and extended Jaccard measures are appropriate for such data.

There are other characteristics of data vectors that may need to be considered. Suppose, for example, that we are interested in comparing time series. If the magnitude of the time series is important (for example, each time series represent total sales of the same organization for a different year), then we could use Euclidean distance. If the time series represent different quantities (for example, blood pressure and oxygen consumption), then we usually want to determine if the time series have the same shape, not the same magnitude. Correlation, which uses a built-in normalization that accounts for differences in magnitude and level, would be more appropriate.

In some cases, transformation or normalization of the data is important for obtaining a proper similarity measure since such transformations are not always present in proximity measures. For instance, time series may have trends or periodic patterns that significantly impact similarity. Also, a proper computation of similarity may require that time lags be taken into account. Finally, two time series may only be similar over specific periods of time. For example, there is a strong relationship between temperature and the use of natural gas, but only during the heating season.

Practical consideration can also be important. Sometimes, a one or more proximity measures are already in use in a particular field, and thus, others will have answered the question of which proximity measures should be used. Other times, the software package or clustering algorithm being used may drastically limit the choices. If efficiency is a concern, then we may want to choose a proximity measure that has a property, such as the triangle inequality, that can be used to reduce the number of proximity calculations.

However, if common practice or practical restrictions do not dictate a choice, then the proper choice of a proximity measure can be a time-consuming task that requires careful consideration of both domain knowledge and the purpose for which the measure is being used. A number of different similarity measures may need to be evaluated to see which ones produce results that make the most sense.

5.3 Cluster analysis

Cluster analysis groups data objects based only on information found in the data that describes the objects and their relationships. The goal is that the objects within a group be similar (or related) to one another and different from (or unrelated to) the objects in other groups. The greater the similarity (or homogeneity) within a group and the greater the difference between groups, the better or more distinct the clustering.

5.3.1 Introduction

Clustering refers to the grouping of records, observations, or cases into classes of similar objects. A *cluster* is a collection of records that are similar to one another and dissimilar to records in other clusters. Clustering algorithms seek to segment the entire data set into relatively homogeneous subgroups or clusters, where the similarity of the records within the cluster is maximized, and the similarity to records outside this cluster is minimized.

In clustering, some details are disregarded in exchange for data simplification. Clustering can be viewed as a data modeling technique that provides for concise summaries of the data. From a machine learning perspective, clusters correspond to hidden patterns, the search for clusters is unsupervised learning, and the resulting system represents a data concept. Therefore, clustering is unsupervised learning of a hidden data concept. Clustering is related to many disciplines and plays an important role in a broad range of applications. The applications of clustering usually deal with large datasets and data with many attributes.

Clustering is often performed as a preliminary step in a data mining process, with the resulting clusters being used as further inputs into a different technique downstream, such as neural networks. Due to the enormous size of

many present-day databases, it is often helpful to apply clustering analysis first, to reduce the search space for the downstream algorithms.

Clustering is a very difficult problem because data can reveal clusters with different shapes and sizes in an n -dimensional data space. To compound the problem further, the number of clusters in the data often depends on the resolution (fine vs. coarse) with which we view the data. The next example illustrates these problems through the process of clustering points in the Euclidean 2D space. Figure 5.4a shows a set of points (samples in a two-dimensional space) scattered on a 2D plane. Let us analyze the problem of dividing the points into a number of groups. The number of groups N is not given beforehand. Figure 5.4b shows the natural clusters G_1 , G_2 , and G_3 bordered by broken curves. Since the number of clusters is not given, we have another partition of four clusters in Figure 5.4c that is as natural as the groups in Figure 5.4b. This kind of arbitrariness for the number of clusters is a major problem in clustering.

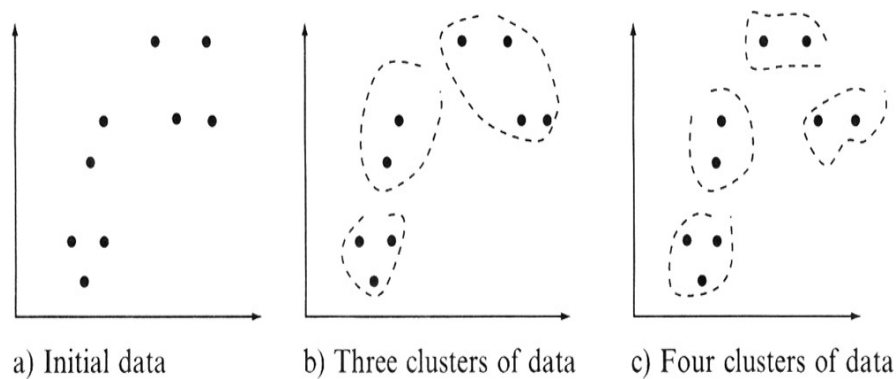


Fig. 5.4. Cluster analysis of points in a 2D-space

Note that the above clusters can be recognized by sight. For a set of points in a higher-dimensional Euclidean space, we cannot recognize clusters visually.

5.3.2 Different Types of Clusterings

An entire collection of clusters is commonly referred to as a clustering, and we can distinguish various types of clusterings: hierarchical (nested) versus partitional (unnested), exclusive versus overlapping versus fuzzy, and complete versus partial.

Hierarchical versus Partitional

The most commonly discussed distinction among different types of clusterings is whether the set of clusters is nested or unnested, or in more traditional terminology, hierarchical or partitional. A *partitional clustering* is simply a division of the set of data objects into non-overlapping subsets (clusters) such that each data object is in exactly one subset.

If we permit clusters to have subclusters, then we obtain a *hierarchical clustering*, which is a set of nested clusters that are organized as a tree. Each node (cluster) in the tree (except for the leaf nodes) is the union of its children (subclusters), and the root of the tree is the cluster containing all the objects. Often, but not always, the leaves of the tree are singleton clusters of individual data objects. Finally, note that a hierarchical clustering can be viewed as a sequence of partitional clusterings and a partitional clustering can be obtained by taking any member of that sequence; i.e., by cutting the hierarchical tree at a particular level.

Exclusive versus Overlapping versus Fuzzy

Clusterings that assign each object to a single cluster are *exclusive*. There are many situations in which a point could reasonably be placed in more than one cluster, and these situations are better addressed by non-exclusive clustering. In the most general sense, an overlapping or non-exclusive clustering is used to reflect the fact that an object can simultaneously belong to more than one group (class). For instance, a person at a university can be both an enrolled student and an employee of the university. A non-exclusive clustering is also often used when, for example, an object is “between” two or more clusters and could reasonably be assigned to any of these clusters.

In a *fuzzy clustering*, every object belongs to every cluster with a membership weight that is between 0 (absolutely does not belong) and 1 (absolutely belongs). In other words, clusters are treated as fuzzy sets. (Mathematically, a fuzzy set is one in which an object belongs to any set with a weight that is between 0 and 1). In fuzzy clustering, we often impose the additional constraint that the sum of the weights for each object must equal 1. Similarly, probabilistic clustering techniques compute the probability with which each point belongs to each cluster, and these probabilities must also sum to 1. Because the membership weights or probabilities for any object sum to 1, a fuzzy or probabilistic clustering does not address true multiclass situations, such as the case of a student employee, where an object belongs to multiple classes. Instead, these approaches are most appropriate for avoiding the arbitrariness of assigning an object to only one cluster when it may be close to several. In practice, a fuzzy or probabilistic clustering is often converted to an exclusive clustering by assigning each object to the cluster in which its membership weight or probability is highest.

Complete versus Partial

A complete clustering assigns every object to a cluster, whereas a partial clustering does not. The motivation for a partial clustering is that some objects in a data set may not belong to well-defined groups. Many times objects in the data set may represent noise, outliers, or “uninteresting background”. For example, some newspaper stories may share a common theme, such as global warming, while other stories are more generic or one-of-a-kind. Thus, to find the important topics in last month’s stories, we may want to search only for clusters of documents that are tightly related by a common theme. In other cases, a complete clustering of the objects is desired. For example, an application that uses clustering to organize documents for browsing needs to guarantee that all documents can be browsed.

5.3.3 Different Types of Clusters

Clustering aims to find useful groups of objects (clusters), where usefulness is defined by the goals of the data analysis. Not surprisingly, there are several different notions of a cluster that prove useful in practice.

Well-Separated

A cluster is a set of objects in which each object is closer (or more similar) to every other object in the cluster than to any object not in the cluster. Sometimes a threshold is used to specify that all the objects in a cluster must be sufficiently close (or similar) to one another. This idealistic definition of a cluster is satisfied only when the data contains natural clusters that are quite far from each other. Well-separated clusters do not need to be globular, but can have any shape.

Prototype-Based

A cluster is a set of objects in which each object is closer (more similar) to the prototype that defines the cluster than to the prototype of any other cluster. For data with continuous attributes, the prototype of a cluster is often a **centroid**, i.e., the average (mean) of all the points in the cluster. When a centroid is not meaningful, such as when the data has categorical attributes, the prototype is often a *medoid*, i.e., the most representative point of a cluster. For many types of data, the prototype can be regarded as the most central point, and in such instances, we commonly refer to *prototype based* clusters as *center-based* clusters. Not surprisingly, such clusters tend to be globular.

Graph-Based

If the data is represented as a graph, where the nodes are objects and the links represent connections among objects, then a cluster can be defined as a *connected component*; i.e., a group of objects that are connected to one another, but that have no connection to objects outside the group. An important example of graph-based clusters are *contiguity-based* clusters, where two objects are connected only if they are within a specified distance of each other. This implies that each object in a contiguity-based cluster is closer to some other object in the cluster than to any point in a different cluster. This definition of a cluster is useful when clusters are irregular or intertwined, but can have trouble when noise is present since a small bridge of points can merge two distinct clusters. Other types of graph-based clusters are also possible. One such approach defines a cluster as a *clique*; i.e., a set of nodes in a graph that are completely connected to each other. Specifically, if we add connections between objects in the order of their distance from one another, a cluster is formed when a set of objects forms a clique. Like prototype-based clusters, such clusters tend to be globular.

Density-Based

A cluster is a dense region of objects that is surrounded by a region of low density. A density based definition of a cluster is often employed when the clusters are irregular or intertwined, and when noise and outliers are present. By contrast, a contiguity based definition of a cluster would not work well for the data when the noise would tend to form bridges between clusters.

5.3.4 Clustering Techniques

A large number of clustering algorithms exists in literature. The selection of clustering algorithm depends both on the type of data available and on the particular purpose and application. In general, the most important clustering methods can be classified into the following categories [72]: *Partitional*, *Hierarchical*, *Density-Based*, *Model-Based* and *Grid-Based*. In the next subsections such categories and the most important algorithms belonging to them are described.

Partitional Clustering

Given a database of n objects, a partitional clustering algorithm constructs k partitions of the data, where each partition represents a cluster and $k \leq n$. In other words, it classifies the data in k groups, which together satisfy two requirements: each group must contain at least one object and each object must belong to exactly one group. Given the number k of partition to construct, a partitioning method creates an initial partitioning. It then use an

iterative relocation technique which attempts to improve the partitioning by moving objects from one group to another. The general criterion of a good partitioning is that objects in the same cluster are “close” or related to each other, whereas objects of different clusters are “far apart” or very different. There are various types of criteria for evaluating the quality of partitions. The family of partitioning clustering algorithms includes the first ones that appeared in the Data Mining Community. The most commonly used are *K-means*, *PAM*, *CLARA* and *CLARANS*.

The *K-means*[81, 85] algorithm, taking the number of clusters k as input parameter, partitions a set of n objects into k clusters, so that the resulting intra-cluster similarity is high whereas the inter-cluster similarity is low. The *K-means* algorithm proceeds as follows. First, it randomly selects k of the objects in the dataset, each one representing a cluster mean or center (called also representative). The representative of a cluster C is an object containing, for each attribute, the mean value of the data objects in C . Each of the remaining objects is assigned to the cluster to which it is the most similar, based on the distance between the object and the cluster representative. Then, it computes the new representative for each cluster. This process iterates until the criterion function converges, that is cluster assignments are stable. The goal of the algorithm is to minimize the cost function

$$E = \sum_{j=1}^k \sum_{x_i \in C_j} d(x_i, \mu_{C_j})$$

where μ_{C_j} is the representative of the cluster C_j . The complexity of the algorithm is $\mathcal{O}(Ikn)$, where I is the number of iterations.

The *PAM (Partitioning Around Medoids)* [85] algorithm is an extension to *k-means*, intended to handle outliers efficiently. Instead of cluster centers, it represents each cluster by its *medoid*. A *medoid* is the most centrally located object inside a cluster. As a consequence, *medoids* are less influenced by extreme values; the mean of a number of objects would have to follow these values while a medoid would not. After an initial random selection of k medoids, the algorithm repeatedly tries to make a better choice of medoids. All the possible pairs of objects are analyzed, where one object in each pair is considered a medoid and the other is not. The quality of the resulting clustering is computed for each such combination. An object x is replaced with the object causing the greatest reduction in square-error. The set of the best objects for each cluster in the current iteration form the medoids for the next iteration. The algorithm iterates as long as the quality of the result is improved. Quality is also measured using the squared-error between the objects in a cluster and its medoid. The computational complexity of *PAM* is $\mathcal{O}(Ik(n-k)^2)$, where I is the number of iterations, and it is very high for large n and k values. For such a reason, a typical *k-medoids* partition algorithm like *PAM* works effectively for small data sets, but it does not scale well for large data sets.

CLARA (*Clustering LARge Applications*) algorithm [85] is a *sampling-based* method and it was born as an extension of *PAM*. This approach works on several samples of size s , of the n tuples in the database, applying *PAM* on each one of them. The output depends on the s samples and is the “best” result given by the application of *PAM* on these samples. The complexity of each iteration now becomes $\mathcal{O}(ks^2 + k(n - k))$, where s is the size of the sample, k is the number of clusters and n is the total number of objects. It is clear that a good clustering based on samples will not necessarily represent a good clustering of the whole data set. In [85] is shown that *CLARA* works well with 5 samples of $40 + k$ size.

CLARANS (*Clustering LARge ApplicatioNS*) [112] combines the sampling technique with *PAM*. However, unlike *CLARA*, *CLARANS* does not confine itself to any sample at any given time. While *CLARA* has a fixed sample at each stage of the search, *CLARANS* draws a sample with some randomness in each step of the search. The clustering process can be presented as searching a graph where every node is a potential solution, i.e. a set of k medoids. The clustering obtained after replacing a single medoid is called the *neighbor* of the current clustering. The number of neighbors to be randomly tried is restricted by a user specified parameter. If a better neighbor is found (i.e. having a lower square-error), *CLARANS* moves to the neighbor’s node and the process starts again; otherwise, the current clustering produces a local optimum. *CLARANS* has been experimentally shown to be more effective than both *PAM* and *CLARA*. However, its computation complexity is $\mathcal{O}(n^2)$.

Hierarchical Clustering

Hierarchical clustering builds a cluster hierarchy or, in other words, a tree of clusters, also known as a *dendrogram*. Every cluster node contains child clusters; sibling clusters partition the points covered by their common parent. Such an approach allows exploring data on different levels of granularity

Hierarchical clustering methods are categorized into *agglomerative* and *divisive*, depending on whether the hierarchical decomposition is formed in a bottom-up or top-down fashion. A brief description of such two types of hierarchical clustering algorithms follows:

- *Agglomerative algorithms* start with each object being a separate cluster itself, and successively merge groups according to a distance measure. The clustering may stop when all objects are in a single group or until a certain termination condition is satisfied (i.e., a desired number of clusters is obtained). These methods generally follow a greedy-like bottom-up merging.
- *Divisive algorithms* follow the opposite strategy. They start with all objects in one cluster and successively split the cluster into smaller and smaller pieces, until each object forms a cluster or until a certain termination condition is satisfied. This is similar to the approach followed by divide-and-conquer algorithms.

Most agglomerative hierarchical clustering algorithms are variants of the *single-link* or *complete-link* algorithms. These two basic algorithms differ only in the way they characterize the similarity between a pair of clusters. In the single-link method, the distance between two clusters is the minimum of the distances between all pairs of samples drawn from the two clusters (one element from the first cluster, the other from the second). In the complete-link algorithm, the distance between two clusters is the maximum of all distances between all pairs drawn from the two clusters. A graphical illustration of these two distance measures is given in Figure 5.5.

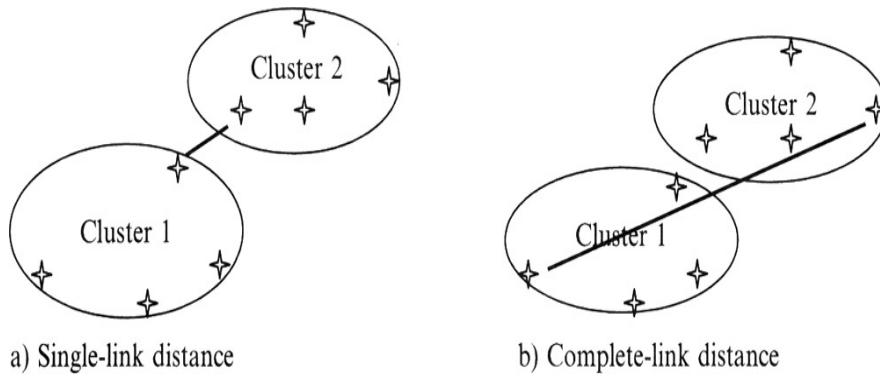


Fig. 5.5. Distances for a single-link and a complete-link clustering algorithm

Sometimes, partitional and hierarchical methods can be integrated. This would mean that a result given by a hierarchical method can be improved via a partitional step, which refines the result via iterative relocation of points. A brief description of the most important hierarchical algorithm proposed in literature follows.

AGNES (*AGglomerative NESTing*) and *DIANA* (*DIVisive ANALysis*) are two pioneering hierarchical clustering algorithm, agglomerative and divisive respectively. *AGNES* initially places each object into a cluster of its own; the clusters are then merged step-by-step according to some criterion. For example, clusters C_1 and C_2 may be merged if an object in C_1 and an object in C_2 form the minimum euclidean distance between any two objects from different clusters. The cluster merging process repeats until all of the objects are eventually merged to form one cluster or if a desired number of clusters (input parameter) is obtained. *DIANA* follows an opposite strategy. Initially, all the objects are used to form one initial cluster. The cluster is split according to some principle, such as the maximum euclidean distance between the closest neighboring objects in the cluster. The cluster splitting process repeats until, eventually, each new cluster contains only a single object or a certain termination condition is satisfied (i.e., a desired number of clusters is obtained

or the distance between the two closest clusters is above a certain threshold distance). Such standard hierarchical approaches suffer from high computational complexity, namely $\mathcal{O}(n^2)$, and some approaches have been proposed to improve this performance.

BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) [164] is an integrated hierarchical clustering method. It is based on the idea that we do not need to keep whole tuples or whole clusters in main memory, but instead, their sufficient statistics. For each cluster, BIRCH stores only the triple (n, LS, SS) , where n is the number of data objects in the cluster, LS is the linear sum of the attribute values of the objects in the cluster and SS is the sum of squares of the attribute values of the objects in the cluster. These triples are called *Cluster Features (CF)* and they are kept in a tree, called *CF-tree*. In [164] it is proved how standard statistical quantities, such as distance measures, can be derived from the *CFs*. The algorithm works as follows. First, it scans input dataset to build an initial in-memory *CF-tree*, which can be viewed as a multilevel compression of the data, that tries to preserve the inherent clustering structure of the data. Then, it applies a clustering algorithm to cluster the leaf nodes of the *CF-tree*. Experiments have shown the linear scalability of the algorithm with respect to the number of objects, and good quality of clustering results.

CURE (Clustering Using REpresentative)[67] is a hierarchical clustering algorithm which overcomes the problem of favoring clusters with spherical shape and similar size. Moreover, it is more robust with respect to outliers. *CURE* employs a novel hierarchical clustering algorithm that adopts a middle ground between centroid-based and representative object-based approaches. Instead of using a single centroid or object to represent a cluster, a fixed number of representative points in space are chosen. The representative points of a cluster are generated by first selecting well-scattered objects for the cluster and then “shrinking” or moving them toward the cluster center by a specified fraction, or *shrinking factor*. At each step of the algorithm, the two clusters with the closest pair of representative points (where each point in the pair is from a different cluster) are merged. *CURE* produces high quality clusters in the existence of outliers, allowing clusters of complex shapes and different sizes. Given n objects, the complexity of *CURE* is $\mathcal{O}(n)$.

CURE does not handle categorical attributes. *ROCK* [68] is an alternative agglomerative hierarchical clustering algorithm that is suited for categorical data. It computes distances between records using the Jaccard coefficients and, using a threshold, for each record it determines who are its neighbors.

CHAMELEON [83] is a clustering algorithm which explores dynamic models in hierarchical clustering. In its clustering process, two clusters are merged if the inter-connectivity and proximity between two clusters are highly related to the internal inter-connectivity and proximity of objects within the clusters. The merge process, based on the dynamic model, facilitates the discovery of natural and homogeneous clusters, and applies to all types of data as long as a similarity function is specified. The algorithm first uses a graph parti-

tioning algorithm to cluster the data objects into a large number of relatively small subclusters. Then, it uses an agglomerative hierarchical clustering algorithm to find the best clusters by repeatedly combining these clusters. To determine the pairs of most similar subclusters, it takes into account both the inter-connectivity as well as the closeness of the clusters, especially the internal-characteristics of the clusters themselves.

Density-Based Clustering

Density-based methods have been developed to discover clusters with arbitrary shapes. To achieve such a goal, clusters can be thought as regions of high density, separated by regions of no or low density. Density here is considered as the number of data objects in the neighborhood. Density-based algorithms typically regard clusters as dense regions of objects in the data space which are separated by regions of low density, representing noise or outliers. A brief description of the most important density-based algorithms proposed in literature follows.

DBSCAN (Density-Based Spatial Clustering of Applications with Noise)[43] is of the most popular density-based clustering algorithm. It finds, for each object, the neighborhood that contains a minimum number of objects. Finding all points whose neighborhood falls into the above class, a cluster is defined as the set of all points transitively connected by their neighborhoods. *DBSCAN* finds arbitrary-shaped clusters and it is not sensitive to the input order. On the other hand, it requires the user to specify the radius of the neighborhood and the minimum number of objects it should have; this is a drawback, because an optimal setting of its parameters is difficult to determine. Algorithm complexity is $\mathcal{O}(n^2)$, but it can be improved to $\mathcal{O}(n \log n)$ if an index structure to help finding neighbors of a data point is employed. Finally, if Euclidean distance is used to measure proximity of objects, its performance degrades for high dimensional data.

OPTICS (Ordering Points To Identify the Clustering Structure) [5] is an extension to *DBSCAN* that relaxes the strict requirements of input parameters. *OPTICS* computes an increasing cluster ordering to automatically and interactively cluster the data. The ordering represents the density based clustering structure of the data and contains information that is equivalent to density-based clustering obtained by a range of parameter settings [72]. *OPTICS* considers a minimum radius that makes a neighborhood legitimate for the algorithm, i.e., having the minimum number of objects, and extends it to a maximum value. *DBSCAN* and *OPTICS* are similar in structure and have the same computational complexity. Moreover, in [87] an incremental version of *OPTICS* has been proposed.

DENCLUE (DENsity-based CLUstEring) [73] is a clustering method based on a set of density distribution functions. The method is built on the following ideas. First, the influence of each data point can be formally modelled using a mathematical function, called an *influence function*, which describes the

impact of a data point within its neighborhood. Second, the overall density of the data space can be modelled analytically as the sum of the influence function of all data points. Third, clusters can be determined mathematically by identifying *density attractors*, where *density attractors* are local maxima of the overall density function.

Model-Based Clustering

Model-Based clustering algorithms attempt to optimize the fit between the given data and some mathematical model. Such methods are often based on the assumption that the data are generated by a mixture of underlying probability distributions. In this section we cite the *Expectation-Maximization (EM)* algorithm [21], which is generally considered as a model-based algorithm or just an extension to the *K-means* algorithm. Indeed, *EM* assigns each object to a dedicated cluster according to the probability of membership for that object. The probability distribution function is the multivariate Gaussian and its main goal is the iterative discovery of good values for its parameters; to achieve such a goal, the algorithm tries to maximize the logarithm of the likelihood of the data, given how well the probabilistic model fits it. The algorithm can handle various shapes of clusters, while at the same time it can be very expensive since hundreds of iterations may be required for the iterative refinement of parameters. In [20] is proposed a scalable solution to EM, based on the observation that data can be compressed, maintained in main memory or discarded.

Grid-Based Clustering

The *Grid-Based Clustering* approach uses a multiresolution grid data structure. It quantizes the space into a finite number of cells forming a grid structure on which all of the operations for clustering are performed. The main advantage of the approach is its fast processing time which is typically independent of the number of data objects, yet dependent on only the number of cells in each dimension in the quantized space. A brief description of the most important *Grid-Based* algorithms follows.

STING (STatistical INformation Grid) [145] breaks the spatial data space into a finite number of cells using a rectangular hierarchical structure. It then processes the data set and computes the mean, variance, minimum, maximum and type of distribution of the objects within each cell. As we go higher in the structure, statistics are being summarized from lower levels (similar to the summarization done with *CFs* in a *CF-tree* in *BIRCH*). New objects are easily inserted in the grid and spatial queries can be answered visiting appropriate cells at each level of the hierarchy. A spatial query is defined as one that retrieves information of spatial data and their interrelationships. *STING* is highly scalable, since it requires one pass over the data, but uses a *multi-resolution* method that highly depends on the granularity of the lowest

level. *Multi-resolution* is the ability to decompose the data set into different levels of detail. Finally, when merging grid cells to form clusters, children are not properly merged (because they only correspond to dedicated parents) and the shapes of clusters have vertical and horizontal boundaries, conforming to the boundaries of the cells.

WaveCluster [132] employs a multi-resolution approach as *STING*, but it follows a different strategy. It uses *Wavelets* to find arbitrary shaped clusters at different levels of resolution. A *wavelet transform* is a signal processing method that decomposes a signal into different frequency bands. Hence, applying this transform into clustering helps in detecting clusters of data objects at different levels of detail. The algorithm handles outliers well, and is highly scalable, $\mathcal{O}(n)$, but not suitable for high dimensional data sets. In [72] is reported that *WaveCluster* performs better than *BIRCH*, *CLARANS* and *DBSCAN*.

5.3.5 Which Clustering Algorithm

A variety of factors need to be considered when deciding which type of clustering technique to use. Our goal is to succinctly summarize these factors in a way that sheds some light on which clustering algorithm might be appropriate for a particular clustering task.

Type of Clustering

One important factor in making sure that the type of clustering matches the intended use is the type of clustering produced by the algorithm. For some applications, such as creating a biological taxonomy, a hierarchy is preferred. In the case of clustering for summarization, a partitional clustering is typical. In yet other applications, both may prove useful.

Most clustering applications require a clustering of all (or almost all) of the objects. For instance, if clustering is used to organize a set of documents for browsing, then we would like most documents to belong to a group. However, if we wanted to find the strongest themes in a set of documents, then we might prefer to have a clustering scheme that produces only very cohesive clusters, even if many documents were left unclustered.

Finally, most applications of clustering assume that each object is assigned to one cluster (or one cluster on a level for hierarchical schemes). As we have seen, however, probabilistic and fuzzy schemes provide weights that indicate the degree or probability of membership in various clusters. Other techniques, such as *DBSCAN* and density-based clustering, have the notion of core points, which strongly belong to one cluster. Such concepts may be useful in certain applications.

Type of Cluster

Another key aspect is whether the type of cluster matches the intended application. There are three commonly encountered types of clusters: prototype-, graph-, and density-based. Prototype-based clustering schemes, as well as some graph-based clustering schemes (complete link, centroid, and Ward's) tend to produce globular clusters in which each object is sufficiently close to the cluster's prototype or to the other objects in the cluster. If, for example, we want to summarize the data to reduce its size and we want to do so with the minimum amount of error, then one of these types of techniques would be most appropriate. In contrast, density-based clustering techniques, as well as some graph-based clustering techniques, such as single link, tend to produce clusters that are not globular and thus contain many objects that are not very similar to one another. If clustering is used to segment a geographical area into contiguous regions based on the type of land cover, then one of these techniques is more suitable than a prototype-based scheme such as K-means.

Characteristics of the Data Sets and Attributes

As previously discussed, the type of data set and attributes can dictate the type of algorithm to use. For instance, the K-means algorithm can only be used on data for which an appropriate proximity measure is available that allows meaningful computation of a cluster centroid. For other clustering techniques, such as many agglomerative hierarchical approaches, the underlying nature of the data sets and attributes is less important as long as a proximity matrix can be created.

Noise and Outliers

Noise and outliers are particularly important aspects of the data. We have tried to indicate the effect of noise and outliers on the various clustering algorithms that we have discussed. In practice, however, it may be difficult to evaluate the amount of noise in the data set or the number of outliers. More than that, what is noise or an outlier to one person may be interesting to another person. For example, if we are using clustering to segment an area into regions of different population density, we do not want to use a density-based clustering technique, such as DBSCAN, that assumes that regions or points with density lower than a global threshold are noise or outliers. As another example, hierarchical clustering schemes, such as CURE, often discard clusters of points that are growing slowly since such groups tend to represent outliers. However, in some applications we may be most interested in relatively small clusters; e.g., in market segmentation, such groups might represent the most profitable customers.

Number of Data Objects

We have considered how clustering is affected by the number of data objects in considerable detail in previous sections. We reiterate, however, that this factor often plays an important role in determining the type of clustering algorithm to be used. Suppose that we want to create a hierarchical clustering of a set of data, we are not interested in a complete hierarchy that extends all the way to individual objects, but only to the point at which we have split the data into a few hundred clusters. If the data is very large, we cannot directly apply an agglomerative hierarchical clustering technique. We could, however, use a divisive clustering technique, but this would only work if the data set is not too large. In this situation, a technique such as BIRCH, which does not require that all data be in main memory, becomes more useful.

Number of Attributes

We have also discussed the impact of dimensionality at some length. Again, the key point is to realize that an algorithm that works well in low or moderate dimensions may not work well in high dimensions. As in many other cases in which a clustering algorithm is inappropriately applied, the clustering algorithm may run and produce clusters, but the clusters may not represent the true structure of the data.

Cluster Description

One aspect of clustering techniques that is often overlooked is how the resulting clusters are described. Prototype clusters are succinctly described by a small set of cluster prototypes. In the case of mixture models, the clusters are described in terms of small sets of parameters, such as the mean vector and the covariance matrix. This is also a very compact and understandable representation. For graph- and density-based clustering approaches, however, clusters are typically described as sets of cluster members. Nonetheless, in CURE, clusters can be described by a (relatively) small set of representative points.

Algorithmic Considerations

There are also important aspects of algorithms that need to be considered. Is the algorithm non-deterministic or order-dependent? Does the algorithm automatically determine the number of clusters? Is there a technique for determining the values of various parameters? Many clustering algorithms try to solve the clustering problem by trying to optimize an objective function. Is the objective a good match for the application objective? If not, then even if the algorithm does a good job of finding a clustering that is optimal or close to optimal with respect to the objective function, the result is not meaningful.

Also, most objective functions give preference to larger clusters at the expense of smaller clusters.

Clustering XML documents: state of the art

6.1 Introduction

XML document clustering is realized through algorithms that rely on the similarity between two documents computed exploiting a distance metric. The algorithms should guarantee that documents in the same cluster have a high similarity degree (low distance), whereas documents in different clusters have a low similarity degree (high distance). As far as clustering of XML data is concerned, the document content, the document structure as well as links among documents can be exploited for identifying similarities among documents. Several measures have been proposed for computing the structural and content similarity among XML documents whereas few XML specific approaches exist for computing link similarity (even if the approaches developed for Web data can be easily applied). Purpose of the chapter is to present and compare the research efforts for developing similarity measures for clustering XML documents relying on their content, structure, and links. Approaches are presented relying on the adopted representation of documents. Vector-based as well as tree-based representations are the most commonly adopted, though more seldom graph and alternative representations have been adopted as well.

6.2 Similarity measures for XML documents

As mentioned in section 5.2, in the definition of a similarity measure we have to point out the objects on which the measure is evaluated, and the relationships existing among such objects. In the XML case, documents are hierarchical in nature and can be viewed as compositions of simpler constituents, including elements, attributes, links, and plain text. The hierarchy of composition is quite rich: attributes and texts are contained in elements, and elements themselves are organized in higher-order structures such as paths and subtrees. We will refer to each level in the compositional structure of an XML document as a *granularity* level. The following levels occur in the literature:

- the whole XML document
- subtrees (i.e., portions of documents)
- paths
- elements
- links
- attributes
- textual content (of attributes and data content elements)

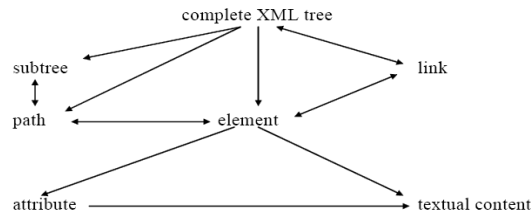


Fig. 6.1. Structural granularities in an XML document

The relationships between the granularity levels are depicted in Figure 6.1 through arrows. An arrow from a granularity level A to a granularity level B means that a similarity measure at level A can be formulated in terms of objects at granularity B. Similarity measures for XML are usually defined according to these natural relations of composition. For instance, a measure for complete XML documents can be defined by evaluating the similarity of paths, which in turn requires some criterion to compare the elements contained in the path.

In addition to composition, other relationships among elements/documents that can be exploited for measuring structural similarity include:

- father-children relationship, that is the relationship between each element and its direct subelements/attributes;
- ancestor-descendant relationship, that is the relationship between each element and its direct and indirect subelements/attributes;
- order relationship among siblings;
- link relationship among documents/elements

In measuring similarity at textual granularity, common IR approaches can be applied on text. Words that are deemed irrelevant are eliminated (e.g. stop list) as well as punctuation. Words that share a common stem are replaced by the stem word. A list of terms is then substituted to the actual text.

The approaches developed in the literature takes some of these objects and relationships into account for the specification of their measures. Approaches can be classified relying on the representation of documents. Some approaches represent documents through labeled trees (eventually extended as graphs to

consider links) and mainly define the similarity measure as an extension of the tree edit distance. Others represent the features of XML documents through a vector model and define the similarity measure as an extension of the distance between two vectors. The tree representation of documents allows pointing out the hierarchical relationships existing among elements/attributes.

6.3 Clustering approaches

Different algorithms have been proposed for clustering XML documents that are extensions of the classical hierarchical and partitioning clustering approaches. We remind that agglomerative algorithms find the clusters by initially assigning each document to its own cluster and then repeatedly merging pairs of clusters until a certain stopping criterion is met. The end result can be graphically represented as a tree called a *dendrogram*. The dendrogram shows the clusters that have been merged together, and the distance between these merged clusters (the horizontal length of the branches is proportional to the distance between the merged clusters). By contrast, partitioning algorithms find clusters by partitioning the set of documents into either a predetermined or an automatically derived number of clusters. The collection is initially partitioned into clusters whose quality is repeatedly optimized, until a stable solution based on a criterion function is found. Hierarchical clustering in general produces clusters of better quality but its main drawback is the quadratic time complexity. For large documents, the linear time complexity of partitioning techniques has made them more popular especially in IR systems where the clustering is employed for efficiency reasons.

Clusters quality is evaluated by internal and external quality measures. The external quality measures use an (external) manual classification of the documents, whereas the internal quality measures are evaluated by calculating average inter and intra-clustering similarity. Standard external quality measures are the *entropy* (which measures how the manually tagged classes are distributed within each cluster), the *purity* (which measures how much a cluster is specialized in a class by dividing its largest class by its size), and the *F-measure* which combines the *precision* and *recall* rates as an overall performance measure. Figure 6.2 reports the formulas of external quality measures (Zhao & Karypis (2004) [166]) relying on the recall and precision formulas. Specifically, we report the measure for a single cluster and for the entire set of clusters determined. A specific external quality measure specifically tailored for XML documents has been proposed by Nierman & Jagadish in [114]. They introduce the notion of *misclustering* for the evaluation of the obtained cluster of XML documents. Given a dendrogram, the misclustering degree is equal to the minimal number of documents in the dendrogram that would have to be moved, so that the documents from the same schema are grouped together.

The Unweighted Pair-Group Method (or UPGMA) is an example of internal quality measure. The distance between clusters C and C' , given $|C|$ the

Quality Measure	Formula	
Recall and precision	$R(i, j) = \frac{n_{ij}}{n_i}$	$P(i, j) = \frac{n_{ij}}{n_j}$
Entropy	$E(j) = -\frac{1}{\log q} \sum_{i=1}^q P(i, j) \log P(i, j)$	$Entropy = \sum_{j=1}^k \frac{n_j}{n} E(j)$
Purity	$Q(j) = \max_{i=1}^q P(i, j)$	$Purity = \sum_{j=1}^k \frac{n_j}{n} Q(j)$
F-measure	$F(i, j) = \frac{2R(i, j)P(i, j)}{R(i, j) + P(i, j)}$	$F = \sum_{j=1}^k \frac{n_j}{n} F(i, j)$

Fig. 6.2. External quality measures

number of objects in C , is computed as follows:

$$Sim(C, C') = \frac{\sum_{o \in C} \sum_{o' \in C'} Sim(o, o')}{|C||C'|}$$

6.4 Tree-based approach

In this section we deal with approaches for measuring the similarity between XML documents that rely on a tree representation of the documents. We first discuss the document representation as trees, the basics of measures for evaluating tree similarity, and then approaches specifically tailored to XML.

6.4.1 Document representation

XML documents can be represented as labelled trees. In trees representing documents, internal nodes are labelled by element/attribute names and leaves are labelled by textual content. In the tree representation, attributes are not distinguished from elements, both are mapped to the tag name set; thus, attributes are handled as elements. Attribute nodes appear as children of the element they refer to and, for what concerns the order, they are sorted by attribute name, and appear before all sub-elements “siblings”. XML document elements may actually refer to, that is, contain links to, other elements. Including these links in the model gives rise to a graph rather than a tree. Even if such links can contain important semantic information that can be exploited in evaluating similarity, most approaches disregard them and simply model documents as trees. The tree representation of the document in Figure 6.3b is reported in Figure 6.4.

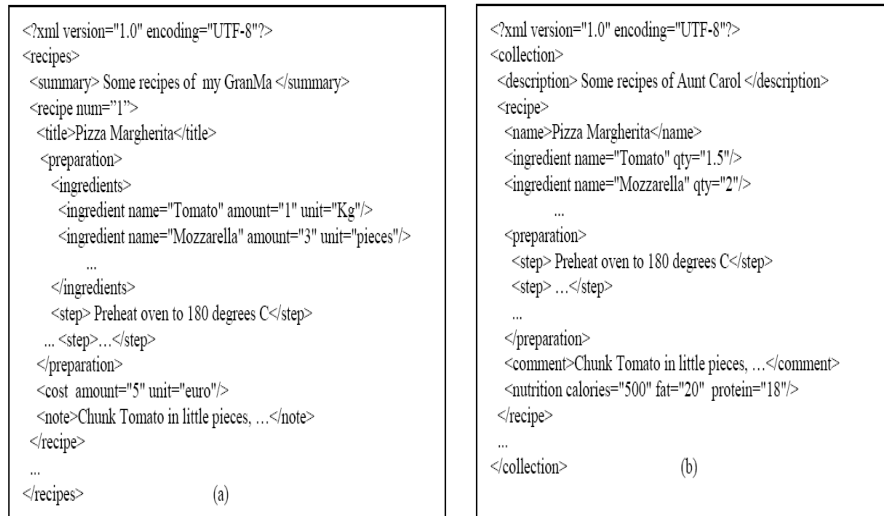


Fig. 6.3. XML documents containing recipes

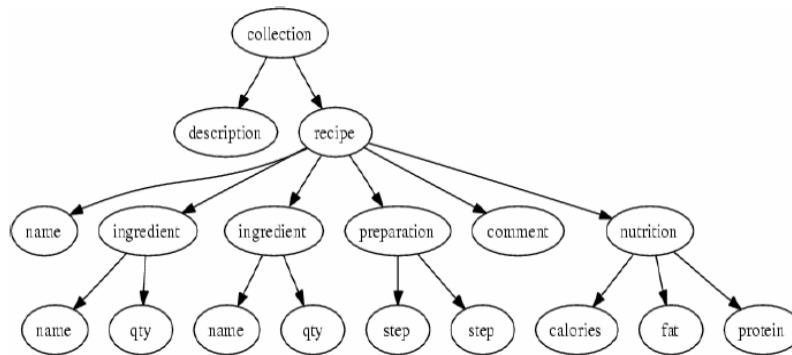


Fig. 6.4. Tree representation of XML documents containing recipes

6.4.2 Tree similarity measures

The problem of computing the distance between two trees, also known as *tree editing problem*, is the generalization of the problem of computing the distance between two strings (Wagner & Fischer, 1974 [143]) to labelled trees. The editing operations available in the tree editing problem are changing (i.e., relabelling), deleting, and inserting a node. To each of these operations a cost is assigned, that can depend on the labels of the involved nodes. The problem is to find a sequence of such operations transforming a tree T_1 into a tree T_2 with minimum cost. The distance between T_1 and T_2 is then defined to be the

cost of such a sequence.

The best known and reference approach to compute edit distance for ordered trees is Zhang and Shasha, 1989 [163]. They consider three kinds of operations for ordered labelled trees. *Relabelling* a node n means changing the label on n . *Deleting* a node n means making the children of n become the children of the parent of n and then removing n . *Inserting* n as the child of m will make n the parent of a consecutive subsequence of the current children of m . Let Σ be the node label set and let λ be a unique symbol not in Σ , denoting the null symbol. An edit operation is represented as $a \rightarrow b$, where a is either λ or the label of a node in T_1 and b is either λ or the label of a node in T_2 . An operation of the form $\lambda \rightarrow b$ is an insertion, an operation of the form $a \rightarrow \lambda$ is a deletion. Finally, an operation of the form $a \rightarrow b$, with $a, b \neq \lambda$ is a relabelling. Each edit operation $a \rightarrow b$ is assigned a cost, that is, a nonnegative real number $\gamma(a \rightarrow b)$ by a cost function γ . Function γ is a distance metric, that is:

$$\begin{aligned} \gamma(a \rightarrow b) &\geq 0; \quad \gamma(a \rightarrow a) = 0, \quad \gamma(a \rightarrow b) = \gamma(b \rightarrow a); \\ \gamma(a \rightarrow c) &\leq \gamma(a \rightarrow b) + \gamma(b \rightarrow c); \end{aligned}$$

Function γ is extended to a sequence of edit operation $S = s_1, \dots, s_k$ s.t. $\gamma(S) = \sum_{i=1}^k \gamma(S_i)$. The edit distance between the two trees T_1 and T_2 is defined as the minimum cost edit operation sequence that transforms T_1 to T_2 , that is:

$$D(T_1, T_2) = \min_S \{ \gamma(S) \mid S \text{ is an edit operation sequence taking } T_1 \text{ to } T_2 \}$$

The edit operations give rise to a mapping which is a graphical specification of which edit operations apply to each node in the two trees. Figure 6.5 is an example of mapping showing a way to transform T_1 to T_2 . It corresponds to the edit sequence $name \rightarrow \lambda$; $calories \rightarrow fat$; $\lambda \rightarrow preparation$. The figure also shows a left-to-right postorder of nodes, which is commonly used to identify nodes in a tree.

For a tree T , let $t[i]$ represent the i th node of T . A mapping (or matching) from T_1 to T_2 is a triple (M, T_1, T_2) where M is a set of pairs of integers (i, j) such that:

- $1 \leq i \leq |T_1|, 1 \leq j \leq |T_2|$;
- for any pair (i_1, j_1) and (i_2, j_2) in M :
 - $i_1 = i_2$ iff $j_1 = j_2$ (one-to-one)
 - $t_1[i_1]$ is to the left of $t_1[i_2]$ iff $t_2[j_1]$ is to the left of $t_2[j_2]$ (sibling order preserved)
 - $t_1[i_1]$ is an ancestor of $t_1[i_2]$ iff $t_2[j_1]$ is an ancestor of $t_2[j_2]$ (ancestor order preserved)

The mapping graphically depicted in Figure 6.5 consist of the pairs: $\{(7, 7), (4, 3), (1, 1), (2, 2), (6, 6), (5, 5)\}$. Let M be a mapping from T_1 to T_2 , the cost of M is defined as:

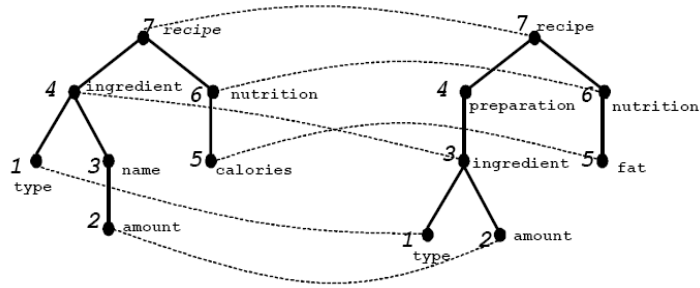


Fig. 6.5. Mapping

$$\begin{aligned} \gamma(M) = & \sum_{(i,j) \in M} \gamma(t_1[i] \rightarrow t_2[j]) + \sum_{\{i | \exists j \text{ s.t. } (i,j) \in M\}} \gamma(t_1[j] \rightarrow \lambda) \\ & + \sum_{\{i | \exists j \text{ s.t. } (i,j) \in M\}} \gamma(\lambda \rightarrow t_2[j]) \end{aligned}$$

There is a straightforward relationship between a mapping and a sequence of edit operations. Specifically, nodes in T_1 not appearing in M correspond to deletions; nodes in T_2 not appearing in M correspond to insertions; nodes that participate to M correspond to relabellings if the two labels are different, to null edits otherwise.

Different approaches (Selkow, 1977 [130], Chawathe et al., 1996 [30], Chawathe, 1999 [29]) to determine tree edit distance have been proposed as well. They rely on similar tree edit operations with minor variations. Figure 6.6 (Dalamagas et al., 2005 [40]) summarizes the main differences among the approaches. The corresponding algorithms are all based on similar dynamic programming techniques. The Chawathe [29] algorithm is based on the same edit operations (i.e., insertion and deletion at leaf nodes and relabelling at any nodes) considered by Selkow [130] but it significantly improves the complexity by reducing the number of recurrences needed, through the use of edit graphs.

6.4.3 XML specific approaches

The basic ideas discussed above for measuring the distance among two trees have been specialized to the XML context by the following approaches.

Nierman et al.

They introduce an approach to measure the structural similarity specifically tailored for XML documents with the aim of clustering together documents presumably generated from the same DTD. Since the focus is strictly on structural similarity, the actual values of document elements and attributes are not

	Edit operations	Complexity
Selkow (1977)	insert node*, delete node*, relabel node	$4^{\min(N,M)}$ M,N numbers of nodes of the trees
Zhang & Shasha (1989)	insert node, delete node, relabel node	$O(M \cdot N \cdot b \cdot d)$ M,N numbers of nodes of the trees, b, d depths of the trees
Chawathe et al. (1996)	insert node*, delete node*, relabel node, move subtree	$O(N \cdot D)$ N numbers of nodes of both trees, D number of misaligned nodes
Chawathe (1999)	insert node*, delete node*, relabel node	$O(M \cdot N)$ M,N dimension of the matrix that represents the edit graph

Fig. 6.6. Tree edit distance algorithms (* marked operations are restricted to leaves)

represented in their tree representations (i.e., leaf nodes of the general representation are omitted from the tree). They suggest to measure the distance between two ordered labelled trees relying on a notion of tree edit distance. However, two XML documents produced from the same DTD may have very different sizes due to optional and repeatable elements. Any edit distance that permits changes to only one node at a time will necessarily find a large distance between such a pair of documents, and consequently will not recognize that these documents should be clustered together as being derived by the same DTD.

Thus, they develop an edit distance metric that is more indicative of this notion of structural similarity. Specifically, in addition to insert, delete, and re-label operations of Zhang & Shasha, 1989 [163], they also introduce the insert subtree and delete subtree editing operations, allowing the cutting and pasting of whole sections of a document. Specifically, operation $insertTreeT(A, i)$ adds A as a child of T at position $i + 1$ and operation $deleteTreeT(T_i)$ deletes T_i as the i -th child of T . They impose however the restriction that the use of the insertTree and deleteTree operations is limited to when the subtree that is being inserted (or deleted) is shared between the source and the destination tree. Without this restriction, one could delete the entire source tree in one step and insert the entire destination tree in a second step, thus making completely useless insert and delete operations. The subtree A being inserted/deleted is thus required to be contained in the source/destination tree T , that is, all its nodes must occur in T , with the same parent/child relationships and the same sibling order; additional siblings may occur in T (to handle the presence of optional elements), as graphically shown in Figure 6.7.

A second restriction imposes that a tree that has been inserted via the $insertTree$ operation cannot subsequently have additional nodes inserted, and, analogously, a tree that have been deleted via the $deleteTree$ operation cannot previously had had nodes deleted. This restriction provides an efficient means for computing the costs of inserting and deleting the subtrees found

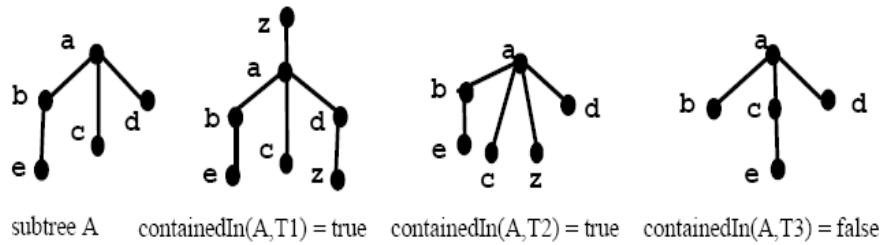


Fig. 6.7. Contained in relationship

in the destination and source trees, respectively. The resulting algorithm is a simple bottom up algorithm obtained as an extension of the Zhang and Shashas basic algorithm, with the difference that any subtree T_i has a graft cost which is the minimum among the cost of a single *insertTree* (if allowable) and of any sequence of insert and (allowable) *insertTree* operations, and similarly any subtree has a prune cost.

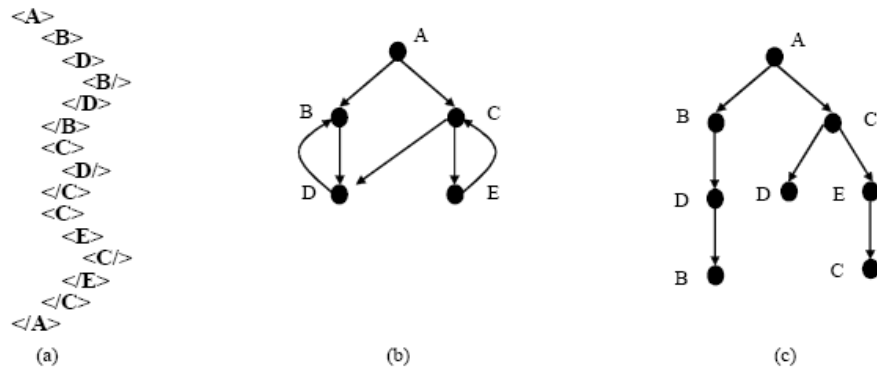


Fig. 6.8. (a) The structure of a document, (b) its s-graph, (c) its structural summary

Lian et al.

They propose a similarity measure for XML documents which, though based on a tree representation of documents, is not based on the tree edit distance. Given a document D they introduce the concept of structure graph (or *s-graph*) of D , $sg(D) = (N, E)$, as a direct graph such that N is the set of all elements and attributes in document D and $(a, b) \in E$ if and only if a is in the parent-child relationship with b . The notion of structure graph is very similar to that of *dataguide* introduced by Goldman et Widom [57] for semi-structured data. Figure 6.8b shows the s-graph of the document in Figure 6.8a. The similarity between two documents D_1 and D_2 is then defined as

$$Sim(D_1, D_2) = \frac{|sg(D_1) \cap sg(D_2)|}{\max\{|sg(D_1)|, |sg(D_2)|\}}$$

Where, $|sg(D_i)|$ is the cardinality of edges in $sg(D_i)$, $i = 1, 2$ and $|sg(D_1) \cap sg(D_2)|$ is the set of common edges between $sg(D_1)$ and $sg(D_2)$. Relying on this metric, if the number of common parent-child relationships between D_1 and D_2 is large, the similarity between the *s-graphs* will be high, and vice-versa. Since the definition of *s-graph* can be easily applied to sets of documents, the comparison of a document with respect to a cluster can be easily accomplished by means of their corresponding *s-graphs*. However a main problem with this approach relies on the loosegrained similarity which occurs. Indeed, two documents can share the same *s-graph*, and still have significant structural differences. Thus, the approach fails in dealing with application domains, such as wrapper generation, requiring finer structural dissimilarities. Moreover, the similarity between the two *s-graphs* in Figure 6.8 is zero according to their definition. Thus, the measure fails to consider similar documents that do not share common edges even if they have some elements with the same labels.

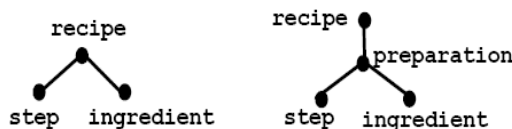


Fig. 6.9. Two simple s-graphs

Dalamagas et al.

They present an approach for measuring the similarity between XML documents modelled as rooted ordered labelled trees. The motivating idea is the same of Nierman and Jagadish [114], that is, that XML documents tend to have many repeated elements and thus they can be large and deeply nested and, even if generated from the same DTD, can have quite different size and structure. Starting from this idea, the approach of Dalamagas et al. [40] is based on extracting structural summaries from documents by nesting and repetition reductions. Nesting reduction consists in eliminating non-leaf nodes whose labels are the same of the ones of their ancestors. By contrast, repetition reduction consists in eliminating, in a pre-order tree traversal, nodes whose paths (starting from the root down to the node itself) have already been traversed. Figure 6.8c shows the structural summary of the document structure in Figure 6.8a. The similarity between two XML documents is then the tree edit distance computed through an extension of the basic Chawathe [29] algorithm. They claim, indeed, that using insertions and deletions only at leaves fits better in the XML context.

6.5 Vector based approaches

In this section we deal with approaches for measuring the similarity that rely on a vector representation of documents. We first discuss the possible document representations as vectors, the different measures that can be exploited for evaluating vector similarity, and then present some approaches specifically tailored to XML.

6.5.1 Document Representation

Vector-based techniques represent objects as vectors in an abstract n -dimensional feature space. Let $O = (o_1, \dots, o_m)$ be a collection of m objects; in our context, these can be whole XML documents, but also paths, individual elements, text, or any other component of a document as reported in Figure 6.4. Each object is described in terms of a set of *features* $F = (F_1, \dots, F_n)$, where each feature $F_i, i \in [1, n]$, has an associated *domain* D_i which defines its allowed values. For instance, the level of an element is a feature whose domain is the positive integers (0 for the root, 1 for first-level elements, and so on).

Feature domains can be either quantitative (continuous or discrete) or qualitative (nominal or ordinal). An object $o \in O$ is described as a tuple $(F_1(o), \dots, F_n(o))$, where each $F_i(o) \in D_i$.

Consider for instance the two documents in Figure 6.1; we can represent them taking the *elements* as the objects to be compared. The simplest possible feature is just the *label* of the document element, whose domain is a string according to the standard XML rules; in this case the roots of both documents are just described as the tuples $(\textit{recipes})$ and $(\textit{collections})$, respectively. Of course, other features are usually considered, possibly of different structural granularities. A typical example is the *path* to the root; for example, consider the leftmost *ingredient* element in each document. Both can be represented using the label and the path as features:

$$\begin{aligned} F_{\textit{ingredient}1} &= (\textit{ingredient}', /recipes/preparation/ingredients') \\ F_{\textit{ingredient}2} &= (\textit{ingredient}', /collection/recipe') \end{aligned}$$

Some authors suggest restricting the length of the paths to avoid a combinatorial explosion. For example, Theobald et al. [139] use paths of length 2.

Another important feature of elements is the *k-neighbourhood*, that is, the set of elements within distance k of the element. For example, consider the 1-neighbourhood (that is, parent and children) of the ingredient elements:

$$\begin{aligned} F_{\textit{ingredient}1} &= (\textit{ingredient}', \{\textit{ingredients}', \textit{name}', \textit{amount}', \textit{unit}'\}) \\ F_{\textit{ingredient}2} &= (\textit{ingredient}', \{\textit{recipe}', \textit{name}', \textit{qty}'\}) \end{aligned}$$

Many variations are possible; for example, one of the components of the Cupid system by Madhavan et al. [95] uses as features the label, the *vicinity* (parent and immediate siblings), and the textual contents of leaf elements.

6.5.2 Vector-based similarity measures

Once the features have been selected, the next step is to define functions to compare them. Given a domain D_i a *comparison criterion* for values in D_i is defined as a function $C_i : D_i \times D_i \rightarrow G_i$, where G_i is a totally ordered set, typically the real numbers. The following property must hold: $C_i(f_i, f_i) = \max_{y \in G_i} y$, that is, when comparing a value with itself the comparison function yields the maximum possible result. The simplest example of a comparison criterion is strict equality:

$$C_i(f_i, f_j) = \begin{cases} 1 & \text{if } f_i = f_j \\ 0 & \text{otherwise} \end{cases}$$

A *similarity function* $S : (D_1, \dots, D_n) \times (D_1, \dots, D_n) \rightarrow L$, where L is a totally ordered set, can now be defined, that compares two objects represented as feature vectors and returns a value that corresponds to their similarity. An example of a similarity function is the *weighted sum*, which associates a weight w_i ($w_i \in [0, 1]$, $\sum_{i=1}^n w_i = 1$) with each feature:

$$S(o, o') = \frac{1}{n} \sum_{i=1}^n w_i C_i(F_i(o), F_i(o'))$$

If feature vectors are real vectors, *metric distances* induced by norms are typically used. The best-known examples are the L_1 (Manhattan) and L_2 (Euclidean) distances. Other measures have been proposed based on the geometric and probabilistic models. The most popular geometric approach to distance is the vector space model used in Information Retrieval [127]. Originally it was intended to be used to compare the similarity among the textual content of two documents, but for the XML case it has been adapted for structural features as well.

The similarity in vector space models is determined by using associative coefficients based on the inner product of the document vectors, where feature overlap indicates similarity. The inner product is usually normalized, since, in practice, not all features are equally relevant when assessing similarity. Intuitively, a feature is more relevant to a document if it appears more frequently in it than in the rest of documents. This is captured by *tfidf weighting*. Let $tf_{i,j}$ be the number of occurrences of feature i in document j , df_i the number of documents containing i , and N the total number of documents. The *tfidf* weight of feature i in document j is:

$$w_{i,j} = tf_{i,j} \log \frac{N}{df_i}$$

The most popular similarity measure is the cosine coefficient, which corresponds to the angle between the vectors. Other measures are the Dice and Jaccard coefficients

$$\begin{aligned}\cos(\mathbf{u}, \mathbf{v}) &= \frac{\mathbf{u}\mathbf{v}}{|\mathbf{u}||\mathbf{v}|} \\ Dice(\mathbf{u}, \mathbf{v}) &= \frac{2\mathbf{u}\mathbf{v}}{|\mathbf{u}|^2 + |\mathbf{v}|^2} \\ Jac(\mathbf{u}, \mathbf{v}) &= \frac{\mathbf{u}\mathbf{v}}{|\mathbf{u}|^2 + |\mathbf{v}|^2 - \mathbf{u}\mathbf{v}}\end{aligned}$$

Another vector-based approach considers the objects as probability mass distributions. This requires some appropriate restrictions on the values of the feature vectors (f_1, \dots, f_n) ; namely, all values must be nonnegative reals, and $\sum_{i=1}^n f_i = 1$. Intuitively, the value of f_i is the probability that the feature F_i is assigned to the object. In principle, correlation statistics can be used to measure the similarity between distributions. The most popular are Pearsons and Spearmans correlation coefficients and Kendalls τ [133]. In addition, some information-theoretic distances have been widely applied in the probabilistic framework, especially the relative entropy, also called the Kullback-Leibler divergence.

$$KL(p_k||q_k) = \sum_k p_k \log_2 \frac{p_k}{q_k}$$

where p_k and q_k are the probability functions of two discrete distributions. Another measure of similarity is the *mutual information*.

$$I(X, Y) = \sum_{x \in X} \sum_{y \in Y} P(x, y) \log_2 \frac{P(x, y)}{P(x)P(y)}$$

where $P(x, y)$ is the joint probability density function of x and y (i.e., $P(x, y) = Pr[X = x, Y = y]$), and $P(x)$ and $P(y)$ are the probability density functions of x and y alone. An important use of information-theoretical measures is to restrict the features and objects to be included in similarity computations, by considering only the most informative. For example, Theobald et al. [139] use the Kullback-Leibler divergence to cut down the number of elements to be compared in an XML classification system.

6.5.3 XML specific approaches

Standard vector-based approaches previously presented can easily be applied to XML documents whenever clustering is performed on a single granularity (e.g. clustering based on contents, on elements, or on paths). Specifically tailored approaches have been developed for XML documents that take more than one granularity along with their relationships into account. In these cases, given C the number of granularities, documents are represented through a C -dimensional matrix M in an Euclidean space based on one of two models, *Boolean* and *weighted*. With the Boolean model, $M(g_1, \dots, g_C) = 1$ if the feature corresponding to the matrix intersection among granularities g_1, \dots, g_C

exists, $M(g_1, \dots, g_C) = 0$ otherwise. With the weighted model, $M(g_1, \dots, g_C)$ is the frequency of the feature corresponding to the matrix intersection among granularities. Figure 6.10 reports a 3-dimensional Boolean matrix on granu-

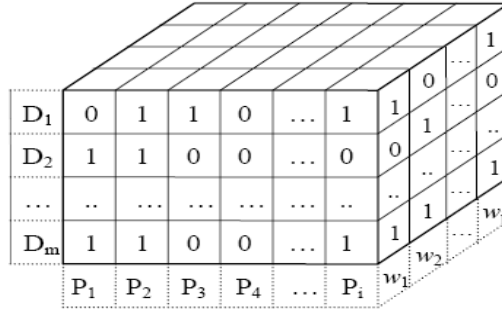


Fig. 6.10. 3-dimensional Boolean matrix

larities (document, path, term) stating the presence (or absence) of a term w_j in the element reached by a path P_j in a document D_m . As suggested by Liu et al. [92], once the documents are represented in the Euclidean space, standard approaches can be applied for measuring their similarity and create clusters. The big issue that should be faced is that the matrix can be sparse. Therefore, approaches for reducing the matrix dimension should be investigated along with the possibility to obtain approximate results.

Yoon et al.

According to our classification, they propose a Boolean model with granularities (document, path, term) in which the path is a root-to-leaf path. A document is defined as a set of (p, v) pairs, where p denotes a root-to-leaf path (named *ePath*) and v denotes a word or a content for an ePath. A collection of XML documents is represented through a 3-dimensional matrix, named *BitCube*, $BC(d, p, v)$, where D denotes a document, p denotes an ePath, v denotes word or content for p , and $BC(D, p, v) = 1$ or 0 depending on the presence or absence of v in the ePath p in D . The distance between two documents is defined through the Hamming Distance as

$$Sim(D_1, D_2) = |XOR(BC(D_1), BC(D_2))|$$

where XOR is a bit-wise exclusive OR operator applied on the representations of the two documents in the BitCube.

Yang J. et al.

According to our classification, they exploit a weighted model with granularities (document, element, term). They employ the Structured Link Vector

Model (SLVM) to represent XML documents. In the model of SLVM, each document, D_x in a document collection C , is represented as a matrix $d_x \in R^{n \times m}$, such that, $d_x = \langle d_{x(1)}, \dots, d_{x(n)} \rangle^T$ and $d_{x(i)} = \langle d_{x(i,1)}, \dots, d_{x(i,m)} \rangle$, where m is the number of elements, $d_{x(i,1)} \in R^m$ is a feature vector related to the term w_i for all subelements, $d_{x(i,j)}$ is a feature related to the term w_i and specific to the element e_j , given as $d_{x(i,j)} = TF(w_i, doc_x.e_j) \cdot IDF(w_i)$ and $TF(w_i, doc_x.e_j)$ is the frequency of the term w_i in the element e_j of the document D_x , $IDF(w_i)$ is the inverse document frequency of w_i based on C (each $d_{x(i,j)}$ is then normalized by $\sum_i d_{x(i,j)}$). The similarity measure between two documents D_x and D_y is then simply defined as

$$Sim(D_x, D_y) = \cos(d_x, d_y) = d_x \cdot d_y = \sum_{i=1}^n d_{x(i)} d_{y(i)}$$

Where, \cdot indicates the vector dot product, and d_x, d_y are the normalized document feature vectors of D_x and D_y . A more sophisticated similarity measure is also presented by introducing a kernel matrix

$$Sim(D_x, D_y) = \sum_{i=1}^n d_{x(i)}^T \cdot M_e \cdot d_{y(i)}$$

where M_e is a $m \times m$ kernel matrix which captures the similarity between pairs of elements as well as the contribution of a pair to the overall similarity. An entry in M_e being small means that the two elements should be semantically unrelated and some words appearing in the two elements should not contribute to the overall similarity and vice versa. An interactive estimation procedure has been proposed for learning a kernel matrix which captures both the element similarity and the element relative importance.

Yang R. et al.

They propose an approach for determining a degree of similarity between a pair of documents that it is easier to compute with respect to tree edit distance and forms a lower bound for the tree edit distance. Their approach thus allows filtering out very dissimilar documents and computes the tree edit distance only with a restricted number of documents. Starting from a tree representation of XML documents (as the one in Figure 6.11a), they represent them as standard full binary trees (Figure 6.11b). A full binary tree is a binary tree in which each node has exactly zero or two children (the first child represents the parent-child relationship, whereas the second child represents the sibling relationship). Whenever one of the children is missing, it is substituted with ϵ . The *binary branch* of the full binary tree (i.e., all nodes with their direct children) are then represented in a binary branch vector, $BRV(D) = (b_1, \dots, b_\Gamma)$, in which b_i represents the number of occurrences of the i^{th} binary branch in the tree, Γ is the size of the binary branch space of the dataset. The binary branch vector for the document in Figure 6.11a is

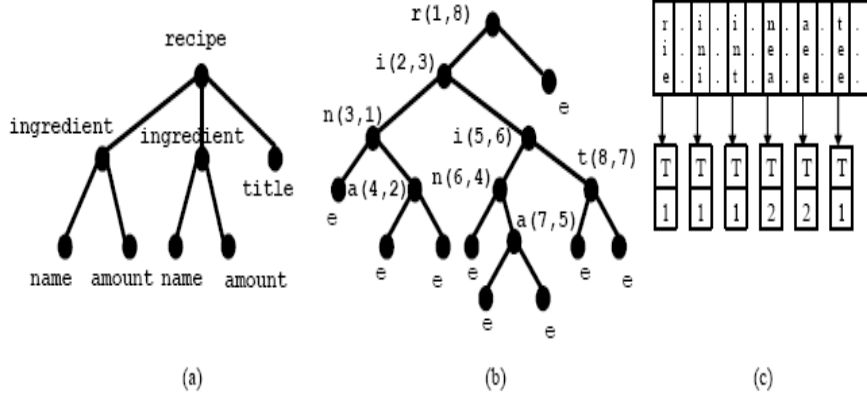


Fig. 6.11. (a)A tree document, (b)its full binary tree, and (c) the binary branch vector

shown in Figure 6.11c. The binary branch distance between XML documents D_1 and D_2 , such that $BRV(D_1) = (b_1, \dots, b_r)$, and $BRV(D_2) = (b'_1, \dots, b'_r)$, is computed through the Manhattan distance:

$$BDist(D_1, D_2) = ||BRV(D_1) - BRV(D_2)||_1 = \sum_{i=1}^r |b_i - b'_i|$$

In this approach the authors consider three granularities (element, element, element) that are bound by the parent-child and the sibling relationships. Then, thanks to the transformation of the document tree structure in a full binary tree structure, they are able to use a 1-dimensional vector for the representation of a document.

6.6 Other approaches

We now present some approaches for evaluating similarity that do exploit neither the vector-based nor the tree-based representation of documents.

6.6.1 Time series based approach

Flesca et al. [48] represent the structure of an XML document as a time series in which each occurrence of a tag corresponds to a given impulse. Thus, they take into account the order in which tags appear in the documents. They interpret an XML document as a discrete-time signal in which numeric values summarize some relevant features of the elements enclosed within the document. If, for instance, one simply indent all tags in a given document according

to their nesting level, the sequence of indentation marks, as they appear within the document rotated by 90 degrees, can be looked at as a time series, whose shape roughly describe the document structure. These time-series data are then analysed through their Discrete Fourier Transform (DFT), leading to abstract from structural details which should not affect the similarity estimation (such as different number of occurrences of an element or small shift in its position). More precisely, during a preorder visit of the XML document tree, as soon as a node is visited an impulse is emitted containing the information relevant to the tag. Thus: **(1)** each element is encoded as a real value; **(2)** the substructures in the documents are encoded using different signal shapes; **(3)** context information can be used to encode both basic elements and substructures, so that the analysis can be tuned to handle in a different way mismatches occurring at different hierarchical levels. Once having represented each document as a signal, document shapes are analysed through DFT. Some useful properties of this transform, namely, the concentration of the energy into few frequency coefficients, its invariance of the amplitude under shifts, allow to reveal much about the distribution and relevance of signal frequencies without the need of resorting to edit distance based algorithms, and, thus, more efficiently. As the encoding guarantees that each relevant subsequence is associated with a group of frequency components, the comparison of their magnitudes allows the detection of similarities and differences between documents. With variable-length sequences, however, the computation of the DFT should be forced on M fixed frequencies, where M is at least as large as the document sizes, otherwise the frequency coefficients may not correspond. To avoid increasing the complexity of the overall approach, the missing coefficients are interpolated starting from the available ones. The distance between documents D_1 and D_2 is then defined as:

$$Dist(D_1, D_2) = \left(\sum_{k=1}^{M/2} (|[\widehat{DFT}(enc(D_1))](k) - [\widehat{DFT}(enc(D_2))]|)^2 \right)^{1/2}$$

where enc is the document encoding function, \widehat{DFT} denotes the interpolation of DFT to the frequencies appearing in both D_1 and D_2 , and M is the total number of points appearing in the interpolation. Comparing two documents using this technique costs $O(n \log n)$, where $n = \max(|D_1|, |D_2|)$ is the maximum number of tags in the documents. The authors claim their approach is practically effective as those based on tree edit distance.

6.6.2 Link-based similarity

Similarity among documents can be measured relying on links. Links can be specified at element granularity through ID/IDREF(S) attributes, or at document granularity through Xlink specifications. To the best of our knowledge

no link-based similarity measures have been specified tailored for XML documents at element granularity. At this granularity a measure should consider the structure and content of the linked elements in order to be effective.

The problem of computing link-based similarity at document granularity has been investigated both for clustering together similar XML documents (Catania & Maddalena, [27]) and for XML document visualization as a graph partitioning problem (Guillaume et al. [69]). An XML document can be connected to other documents by means of both internal or external Xlink link specifications. A weight can be associated with the link depending on a variety of factors (e.g. the type of link, the frequency it is used, its semantics). The similarity between two documents can be expressed in terms of the weight of the minimum path between two nodes. Given a connection graph $G = (V, E)$ where each v_i in V represents an XML document D_i , and each (v_i, v_j, w) is a direct w -weighted edge in E , Catania & Maddalena (2002) specify the similarity between documents D_i and D_j as

$$Sim(D_1, D_2) = \begin{cases} 1 - \frac{1}{2^{cost(minPath(v_1, v_2)) + cost(minPath(v_2, v_1))}} & \text{if } existPath(v_1, v_j) = true \\ & i, j \in [1, 2] \\ 0 & \text{otherwise} \end{cases}$$

where: $minPath(v_1, v_2)$ is the minimal path from v_1 to v_2 , $cost(minPath(v_1, v_2))$ is the sum of the weights on the edge in the minimal path, $existPath(v_1, v_2) = true$ if a path exists from v_1 to v_2 . A key feature of their approach is assigning a different weight to edges depending on the possible type (and, therefore, semantics) an Xlink link can have (simple/extended, on load/on demand).

***XREP*: clustering XML documents by structure**

7.1 Introduction

The increasing relevance of the Web as a means for sharing information has made traditional approaches to information handling ineffective. Indeed, they are mainly devoted to the management of highly structured information, like relational databases, whereas Web data are semistructured and encoded using different formats. In particular, XML is touted as the driving-force for exchanging data on the Web, since it benefits from several advantages with respect to other data models. Examples are the flexibility for designing ad hoc markup languages for the representation and exchange of semistructured data within any application context, and the support of suitable document type definitions (DTDs) and XML Schema that permit to specify both the structure and the content of the documents.

As the heterogeneity of XML sources increases, the need for organizing XML documents according to their structural features has become challenging. In such a context, we address the problem of inferring structural similarities among XML documents, with the adoption of clustering techniques. This problem has several interesting applications related to the management of Web data. For example, structural analysis of Web sites can benefit from the identification of similar documents, conforming to a particular schema, which can serve as the input for wrappers working on structurally similar Web pages. Also, query processing in semistructured data can substantially benefit from the re-organization of documents on the basis of their structure. Grouping semistructured documents according to their structural homogeneity can help in devising indexing techniques for such documents, thus improving the construction of query plans.

7.1.1 A comparative overview of clustering scheme

The problem of comparing semistructured documents has been recently investigated from different perspectives [35, 146, 30, 17, 48]. For example, in the

context of change detection [35, 146, 30], or with the purpose of characterizing a document with respect to a given DTD [17]. Apart from their effectiveness in the application domains considered here, most of these methods are based on the concept of edit distance [163] and use graph-matching algorithms to calculate a minimum-cost edit script that contains the updates necessary to transform a document into another. From a computational point of view, these techniques are rather expensive, i.e. at least $O(N^2)$, where N is the number of elements within any two XML documents.

A rather different approach has been recently proposed in [48]. Here, the structure of an XML document is represented as a time series, in which each occurrence of a tag corresponds to an impulse and the degree of similarity among documents is computed by analyzing the frequencies of the corresponding Fourier transform. The overall cost of this method is $O(N \log N)$, where N here denotes the maximum number of tags in the documents to be compared.

Recent studies have also proposed techniques for clustering XML documents. [42] proposes a partitioning method that clusters documents, represented in a vector-space model, according to both textual contents and structural relations among tags. The approach in [114] proposes to measure structural similarity by means of an XML-aware edit distance, and applies a standard hierarchical clustering algorithm to evaluate how closely cluster documents correspond to their respective DTDs.

In our opinion, a main drawback of the above approaches is the lack of a notion of *cluster prototype*, i.e. a summarization of the relevant features of the documents belonging to a cluster. The notion of cluster prototype is crucial in most significant application domains, such as wrapper induction, similarity search, and query optimization. Indeed, in the context of wrapper induction, the efficiency and effectiveness of the extraction techniques strongly rely on the capability of rapidly detecting homogeneous subparts of the documents under consideration. Similarity search can substantially benefit from narrowing the search space. In particular, this can be achieved by discarding clusters whose prototypes exhibit features which do not comply with the target properties specified by a user-supplied query.

To the best of our knowledge, the only approach devising a notion of cluster prototype is [90]. Indeed, the authors propose to compare documents according to a structure graph, *s-graph*, summarizing the relations between elements within documents. Since the notion of s-graph can be easily generalized to sets of documents, the comparison of a document with respect to a cluster can be easily accomplished by means of their corresponding s-graphs. However, a main problem with the above approach relies on the loose-grained similarity which occurs. Indeed, two documents can share the same prototype s-graph, and still have significant structural differences, such as in the hierarchical relationship between elements. It is clear that the approach fails in dealing with application domains, such as wrapper generation, requiring finer structural dissimilarities.

In this thesis we propose a novel methodology for clustering XML documents by structure, which is based on the notion of *XML cluster representative*. A cluster representative is a prototype XML document subsuming the most relevant structural features of the documents within a cluster. The intuition at the core of our approach is that a suitable cluster prototype can be obtained as the outcome of a proper overlapping among all the documents within a given cluster. Actually, the resulting tree has the main advantage of retaining the specifics of the enclosed documents, while guaranteeing a compact representation. This eventually makes the proposed notion of cluster representative extremely profitable in the envisaged applications: in particular, as a summary for the cluster, a representative highlights common subparts in the enclosed documents, and can avoid expensive comparisons with individual documents in the cluster.

The proposed notion of cluster representative relies on the notions of XML tree *matching* and *merging*. Specifically, given a set of XML documents, our approach initially builds an *optimal matching tree*, i.e. an XML tree that is built from the structural resemblances that characterize the original documents. Then, in order to capture all such peculiarities within a cluster, a further tree, called a *merge tree*, is built to include those document substructures that are not recurring across the cluster documents. Both trees are exploited for suitably computing a cluster representative as will be later detailed. Finally, a hierarchical clustering algorithm exploits the devised notion of representative to partition XML documents into structurally homogeneous groups. Experimental evaluation performed on both synthetic and real data states the effectiveness of our approach in identifying document partitions characterized by a high degree of homogeneity.

7.2 Problem Statement

Clustering is the task of organizing a collection of objects (whose classification is unknown) into meaningful or useful groups, namely *clusters*, based on the interesting relationships discovered in the data. The goal is grouping highly-similar objects into individual partitions, with the requirement that objects within distinct clusters are dissimilar from one another.

Several clustering algorithms [81] can be suitably adapted for clustering semistructured data. We concentrate on hierarchical approaches, which are widely known as providing clusters with a better quality, and can be exploited to generate cluster hierarchies. Fig.7.1 shows *XRep*, an adaptation of the agglomerative hierarchical algorithm to our problem. Initially each XML tree (derived by parsing the corresponding XML document) is placed in its own cluster, and a matrix containing the pair-wise tree distance is computed. Next, the algorithm walks into an iterative step in which the least dissimilar clusters are merged. As a consequence, the distance matrix is updated to reflect this merge operation. The overall process is stopped when an optimal partition

<p>Input: A set $\mathcal{S} = \{t_1, \dots, t_n\}$ of XML document trees;</p> <p>Output: A cluster partition $\mathcal{P} = \{C_1, \dots, C_k\}$ of \mathcal{S}.</p> <p>Method:</p> <p> let $\mathcal{P} := \{C_1, \dots, C_n\}$, where initially $C_i = \{t_i\}$;</p> <p> set $r_i := t_i$ as the representative for C_i;</p> <p> compute a tree-distance matrix M_d, where $M_d(i, j) = d(t_i, t_j)$;</p> <p> repeat</p> <p> choose clusters C_i and C_j such that $d(\text{rep}(C_i), \text{rep}(C_j))$ is minimized;</p> <p> compute the representative $r = \text{rep}(r_i, r_j)$ for cluster $C = C_i \cup C_j$;</p> <p> set $\mathcal{P} := \mathcal{P} - \{C_i, C_j\} \cup \{C\}$, and update M_d;</p> <p> until \mathcal{P} has maximal quality;</p>
--

Fig. 7.1. The XRep algorithm for clustering XML documents.

(i.e. a partition whose intra-distance within clusters is minimized and inter-distance between clusters is maximized) is reached. The technique proposed in this chapter follows the approach devised in [64], and addresses the problem of clustering XML documents in a parametric way. More precisely, the general scheme of the XRep algorithm is parametric to the notions of *distance measure* and *cluster representative*.

Distance measure

The identification of a proper notion of distance for measuring degrees of similarity between pairs of XML documents affects the effectiveness of the overall clustering approach. A number of proposals are available from the current literature. Some of these are reviewed next.

The notion of edit distance between two trees relies on the identification of a sequence of node modifications with minimum cost, required to convert one given tree into another [163]. Given a pair of trees t_i and t_j , three edit operations are possible to convert the former into the latter [80], namely the insertion of a node $w \in V_{t_j}$ in t_i , the deletion of leaf nodes from V_{t_i} and the relabelling of some node of t_i . Each edit operation has an associated cost γ . Let $S = \langle s_1, \dots, s_k \rangle$ denote a sequence of node modifications. A *S-derivation* of t_j from t_i is a sequence of intermediate trees $\langle u_0, \dots, u_k \rangle$, where $u_0 = t_i$, $u_k = t_j$, and u_{i-1} is modified into u_i via the corresponding edit operation $s_i \in S$. The cost [70] of sequence S is $\gamma(S) = \sum_{i=1}^k \gamma(s_i)$. If \mathcal{S} indicates the set of all possible *S-derivations* of t_j from t_i , the edit distance between such trees can be formalized as follows

$$d_E(t_i, t_j) = \min_{S \in \mathcal{S}} \{\gamma(S)\}$$

A major criticism to the notion of edit distance is that its computational complexity is $\mathcal{O}(m \times n)$, where m and n are the respective sizes of the trees under comparison. Some refinements have been proposed in the literature, such as the one in [114], where specific edit operations, namely the insertion and deletion of whole subtrees, are allowed to the purpose of overcoming the original limitation of a node change per time. Notwithstanding, the overall

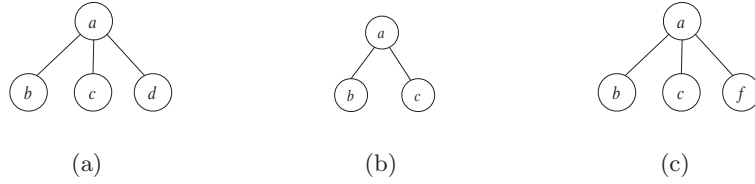


Fig. 7.2. Tree XML document trees, t_1 (a), t_2 (b) and t_3 (c), which make problematic the choice between deleting node d or relabelling node f .

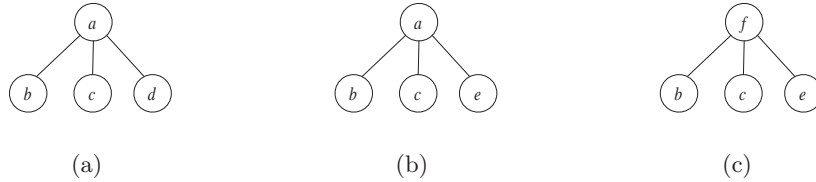


Fig. 7.3. Tree XML document trees, t_1 (a), t_2 (b) and t_3 (c), that, despite considerable structural differences, are at a same tree edit distance.

complexity still exhibits a quadratic dependance on the combined size of the two trees. This makes edit distance inadequate for practical applications, involving huge collections of (large) XML documents. Moreover, this notion of distance suffers from two further limitations [90]. First, relative differences in operation costs may affect the resulting clustering performance.

Example 7.1. To better exemplify this latter point, consider the toy database of three XML document trees in fig. 7.2. If node (resp. subtree) deletion costs less than node (resp. subtree) relabelling, then $d_E(t_1, t_2) < d_E(t_1, t_3)$ and t_3 is separated from t_1 and t_2 . Otherwise, $d_E(t_1, t_2) > d_E(t_1, t_3)$ and t_2 originates a singleton cluster, thus being isolated from t_1 and t_3 . \square

Also, it may not be effective in distinguishing among XML documents with actually different tree structures.

Example 7.2. Fig. 7.3 elucidates such a challenging issue. Trees t_1, t_2, t_3 are neatly distinct. However, a single node relabelling is required to transform any such a tree into another. This vanishes the relevant structural differences among the trees. Indeed, t_1 and t_2 more structurally homogeneous, since they only differ in a leaf node, whereas t_3 shares no path with t_1 and t_2 (having a different root). \square

An alternative approach for capturing similarity between XML documents consists in modelling them as transactions of binary attributes, where each such an attribute indicates the presence of a specific element tag in a given document. Viewed in this respect, the notions of cosine or Jaccard similarity

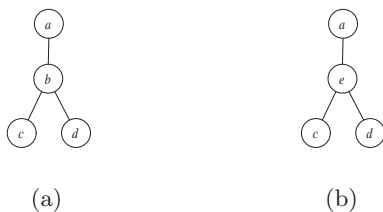


Fig. 7.4. Two XML document trees, t_1 (a) and t_2 (b), with considerable structural differences and, notwithstanding, characterized by low degrees of cosine and Jaccard dissimilarity.

can be straightforwardly adapted to be exploited in the context of XML trees. Let $\text{tag}(t)$ denote the set of tag names in a generic XML tree t and assume that \mathbf{t}_i and \mathbf{t}_j are two vector representations of the corresponding trees t_i and t_j over the tag space $\text{tag}(t_i) \cup \text{tag}(t_j)$. Cosine distance $d_C(t_i, t_j)$ between t_i and t_j is defined as

$$d_C(t_i, t_j) = 1 - \frac{\mathbf{t}_i \cdot \mathbf{t}_j}{\|\mathbf{t}_i\| \|\mathbf{t}_j\|}$$

Jaccard distance $d_J^{(1)}(t_i, t_j)$ directly follows from taking into account all those tags common to both trees t_i and t_j

$$d_J^{(1)}(t_i, t_j) = 1 - \frac{|\text{tag}(t_i) \cap \text{tag}(t_j)|}{|\text{tag}(t_i) \cup \text{tag}(t_j)|}$$

However, a primary limitation of both cosine and Jaccard distances lies in their difficulty at actually distinguishing between XML documents with almost the same set of tags, even if structurally different.

Example 7.3. Consider the XML trees t_1 and t_2 in fig. 7.4. It holds that $d_C(t_1, t_2) = 0.25$ and $d_J^{(1)} = 0.4$. In practice, these dissimilarity values are too low for two trees with no common path. \square

Basically, cosine and Jaccard dissimilarities fail at ignoring structural differences between two XML trees under comparison. A more effective definition of dissimilarity for any two XML document trees can be given by combining the exploitation of node labels with information on their location within both XML trees. To this purpose, tree paths can be taken as basic elements for a structural comparison: the higher the number of common paths between two XML trees, the lower the resulting degree of dissimilarity. The notion of Jaccard dissimilarity naturally lends itself to being employed for such a comparison, provided that it is adapted to take tree paths into account

$$d_J^{(2)}(t_i, t_j) = 1 - \frac{|path(t_i) \cap path(t_j)|}{|path(t_i) \cup path(t_j)|}$$

where notation $path(t)$ denotes the set of paths in a tree t . $d_J^{(2)}$ can be effectively used to partition a large collection of XML documents on the basis of their structural characteristics, without incurring into any of the aforementioned issues.

Example 7.4. Let us come back to the example database of fig. 7.2. In such a case, it holds that $d_J^{(2)}(t_1, t_2) = \frac{1}{3}$ and $d_J^{(2)}(t_1, t_3) = \frac{1}{2}$. Such a result poses no ambiguities to the partitioning process, since it is clear that t_1 and t_2 would be clustered together. Also, $d_J^{(2)}$ yields an expected clustering from the trees in fig. 7.3. Here, $d_J^{(2)}(t_3, t_i) = 1$ for $i = 1, 2$, whereas $d_J^{(2)}(t_1, t_2) = \frac{1}{2}$, which determines the assignment of t_1 and t_2 to a same cluster. Finally, $d_J^{(2)}(t_1, t_2) = 0$ in the case of fig. 7.4, which prevents the two trees from being clustered together. \square

Cluster representative

Intuitively, a representative of a cluster of XML documents is a document which effectively synthesizes the most relevant structural features of the documents in the cluster. The notion of representative can be formalized as follows.

Definition 7.1 *Given a set \mathcal{U} , equipped with a distance function $d : \mathcal{U} \times \mathcal{U} \mapsto \mathbb{R}$, and a set $\mathcal{S} = \{t_1, \dots, t_n\} \subseteq \mathcal{U}$ of XML document trees, the representative of \mathcal{S} (denoted by $rep(\mathcal{S})$) is the tree t^* that minimizes the sum of the distances:*

$$t^* = rep(\mathcal{S}) \in \mathcal{U} \iff t^* = argmin_{t \in \mathcal{U}} f(t)$$

where $f(t) = \sum_{i=1}^n d(t_i, t)$. \square

The computation of the representative of a set turns out to be a hard problem if the above distance measures are adopted. Therefore we exploit a suitable heuristic for addressing the above minimization problem. Viewed in this respect, our goal is to find a *lower-bound-tree* and an *upper-bound-tree* for the optimal representative. The lower-bound-tree (resp. upper-bound-tree) is a tree on which any node deletion (resp. node insertion) leads to a worsening in function f . Thus, a representative can be heuristically computed by traversing the search space delimited by the above trees. Two alternative greedy strategies can be devised: either a growing approach, which iteratively adds nodes to the lower-bound, or a pruning approach, which iteratively removes nodes from the upper-bound. In the following, we will denote the lower-bound-tree and the upper-bound-tree as *optimal matching tree* and *merge tree*, respectively. Notice that the optimal matching tree represents a stopping condition for the pruning approach, whereas the merge tree is always a sub-optimal solution since it contains the optimal representative. Dually, the merge tree

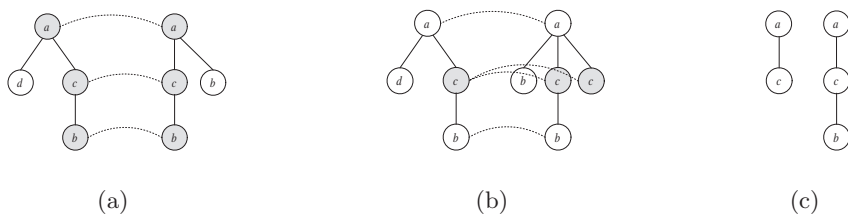


Fig. 7.5. (a) Strong and (b) multiple matching, and (c) their trees.

defines a stopping condition for the growing approach, whereas the optimal matching tree is a sub-optimal solution since it is contained in the optimal representative.

We develop a pruning approach in which the computation of an XML cluster representative consists of the following three main stages: the construction of an optimal matching tree, the computation of a merge tree and the pruning of the merge tree. Fig.7.6 sketches an algorithm which has been developed according to the above three stages.

7.3 Mining Representatives from XML Trees

We give some definitions which are at the basis of our approach. A tree t is a tuple $t = (r_t, V_t, E_t, \lambda_t)$ where $V_t \subseteq \mathbb{N}$ is the set of nodes, $E_t \subseteq V_t \times V_t$ is the set of edges, r_t is the root node of t , and $\lambda_t : V_t \mapsto \Sigma$ is a node labelling function where Σ is an alphabet of node labels. In particular, we say that an *XML tree* is a tree where Σ is an alphabet of *element tags*. Moreover, let $depth_t(v)$ denote the depth level of node v in t , with $depth_t(r_t) = 0$, and let $path_t(v) = \langle v_{i_1} = r_t, v_{i_2}, \dots, v_{i_p} = v \rangle$ denote the list of p nodes that lead up to the node v from the root r_t .

Definition 7.2 (strong matching) *Given two trees t_1 and t_2 , and two nodes $v \in V_{t_1}$, $w \in V_{t_2}$, a strong matching $match(v, w)$ between v and w exists if $\lambda_{t_1}(v_i) = \lambda_{t_2}(w_i)$ and $depth_{t_1}(v_i) = depth_{t_2}(w_i)$, for each pair of nodes (v_i, w_i) such that $v_i \in path_{t_1}(v)$ and $w_i \in path_{t_2}(w)$. \square*

The above definition states that any two nodes, v and w , have a strong matching if v and w together with their respective ancestors share both the same label (i.e. tag name) and depth level. Fig.7.5(a) displays an example of strong matching among the colored nodes.

The detection of matching nodes between two trees allows the construction of a new tree, called a *matching tree*, which resembles the *intersection* of the original trees.

```

Input:
  An XML tree  $r_1 = \langle r_{r_1}, V_{r_1}, E_{r_1}, \lambda_{r_1} \rangle$  as representative of cluster  $C_1$ , and
  an XML tree  $r_2 = \langle r_{r_2}, V_{r_2}, E_{r_2}, \lambda_{r_2} \rangle$  as representative of cluster  $C_2$ .
Output:
  An XML tree  $rep$  as representative of cluster  $C = C_1 \cup C_2$ .
Method:
  compute the matching matrix  $M_m$ , with size  $(|V_{r_1}| \times |V_{r_2}|)$ ;
  compute the marking vectors  $V_{m_1}, V_{m_2}$ , where  $V_{m_1}.size = |V_{r_1}|$  and  $V_{m_2}.size = |V_{r_2}|$ ;
  set  $m_1 := |\{v_i \in V_{r_1} | V_{m_1}[i] \neq -1\}|$ , and  $m_2 := |\{v_i \in V_{r_2} | V_{m_2}[i] \neq -1\}|$ ;
  if  $(m_1 > m_2)$ 
     $match := buildMatch(r_1, r_2, V_{m_1}, V_{m_2});$     $merge := buildMerge(r_1, r_2, V_{m_1}, V_{m_2});$ 
  else
     $match := buildMatch(r_2, r_1, V_{m_2}, V_{m_1});$     $merge := buildMerge(r_2, r_1, V_{m_2}, V_{m_1});$ 
   $rep := prune(C_1 \cup C_2, merge, match);$ 
  return  $rep$ ;

Function  $buildMatch(t_1, t_2, V_{m_1}, V_{m_2}) : t;$ 
   $t := t_1;$ 
  for each  $v_i \in V_{t_1}, V_{m_1}[i] = -1$  do
     $remove(t, v_i);$  /* removes the subtree rooted at  $v_i$  from  $t$  */
  let  $I_j = \{v_{i_1}, \dots, v_{i_h} \in V_{t_1} | V_{m_1}[i_p] = j, p \in [1..h]\};$ 
  for each  $I_j$  do
     $removeDuplicates(t, I_j);$  /* removes duplicated paths from  $t$  */
  return  $t$ ;

Function  $buildMerge(t_1, t_2, V_{m_1}, V_{m_2}) : t;$ 
   $t := t_1;$ 
  for each  $v_i \in V_{t_1}$  do
    let  $J = \{w_{j_1}, \dots, w_{j_h} \in V_{t_2} | V_{m_2}[j_p] = i, p \in [1..h]\};$ 
    let  $v \in V_{t_1}$  such that  $(v, v_i) \in E_{t_1};$ 
     $insert(t, v, v_i, |J| - 1);$  /* inserts node  $v_i$  as a child of  $v$  into  $t$ ,  $|J| - 1$  times */
  for each  $w_i \in V_{t_2}, V_{m_2}[i] = -1$  do
    let  $w_j \in V_{t_2}$  such that  $(w_j, w_i) \in E_{t_2}$ , and  $v_h \in V_{t_1}$  such that  $V_{m_2}[j] = h;$ 
     $insert(t, v_h, w_i);$  /* inserts node  $w_i$  as a child of  $v_h$  into  $t$  */
  return  $t$ ;

Function  $prune(C, t, t') : r;$ 
  set  $r := t;$ 
  do
    let  $\mathcal{L} \subseteq V_r$  be the set of leaf nodes in  $r$ ;
    compute  $d_0 := \sum_{t \in C} d(t, r);$ 
    for each  $v_l \in \mathcal{L}$  do
      compute  $r^{(l)} := removeLeaf(r, v_l);$ 
       $l^* = \arg \min_{v_l} \{\sum_{t \in C} d(t, r^{(l)})\};$ 
    set  $d^* := \sum_{t \in C} d(t, r^{(l^*)});$ 
    if  $(d^* < d_0)$ ;
       $r := r^{(l^*)};$ 
  while  $d^* < d_0$  and  $V_r \subseteq V_{t'}$ ;
  return  $r$ ;

```

Fig. 7.6. The algorithm for the computation of an XML cluster representative.

Definition 7.3 (matching tree) Given two trees t_1 and t_2 , a tree $t = \langle r_m, V_m, E_m, \lambda_m \rangle$ is a matching tree, denoted by $t = match(t_1, t_2)$, if the following conditions hold:

1. there exist two mappings $f_1 : t \mapsto t_1$ and $f_2 : t \mapsto t_2$ associating nodes and edges in t with a subtree in t_1 and t_2 ;
2. for each $u \in V_m$, there exists a strong matching between $v = f_1(u)$ and $w = f_2(u)$ (i.e. $match(v, w)$ holds); moreover, $\lambda_m(u) = \lambda_{t_1}(v) = \lambda_{t_2}(w)$;
3. $f_1(r_m) = r_{t_1}$, and $f_2(r_m) = r_{t_2}$; moreover, for each $e = (u, v) \in E_m$, $f_1(e) = (f_1(u), f_1(v))$ and $f_2(e) = (f_2(u), f_2(v))$. \square

Notice that, in general, multiple matchings may occur when a node in a tree has a matching with more than one node in a different tree. More formally, given two trees t_1 and t_2 , a node $v \in V_{t_1}$ has a *multiple matching* if

$\exists w', w'' \in V_{t_2}$ such that both $match(v, w')$ and $match(v, w'')$ hold. An example of multiple matching between nodes in two trees is shown in Fig.7.5(b). Multiple matchings trigger ambiguities in defining matching trees: Fig.7.5(c) represents two alternative matching trees for the documents in Fig.7.5(b).

7.3.1 XML Tree Matching

In order to capture as many structural affinities as possible, we are interested in finding matching trees with maximal size. Formally, a matching tree $t_m = match(t_1, t_2)$ is an *optimal matching tree* for two XML trees t_1, t_2 if there does not exist another matching tree $t'_m = match(t_1, t_2) \neq t_m$ such that $|V_{t_m}| \geq |V_{t'_m}|$. We describe a dynamic-programming technique for building an optimal matching tree from two XML trees. The technique consists of three steps: *i*) detection of matching nodes, *ii*) selection of best matchings, and *iii*) optimal matching tree construction.

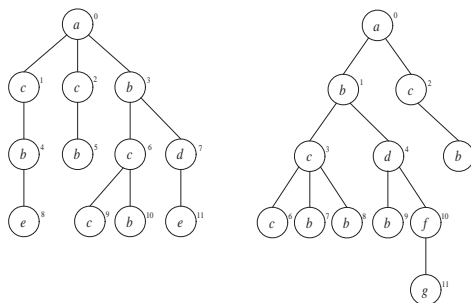
Matching detection. Given two trees t_1 and t_2 , the detection of matching nodes is performed building a $(|V_{t_1}| \times |V_{t_2}|)$ matching matrix M_m . In this matrix, the generic (i, j) -th element corresponds to nodes $v_i \in V_{t_1}$ and $w_j \in V_{t_2}$, and contains a weight $\omega_m(v_i, w_j)$ to be associated with the matching between v_i and w_j . Initially, the weight is 1 if $match(v_i, w_j)$ holds, and 0 otherwise. In order to ease the construction of the matching matrix, nodes are enumerated by level, thus guaranteeing a particular block structure for M_m . Indeed, for each level k , a sub-matrix $M_m(k)$ collects the matchings among the nodes in t_1 and t_2 with depth equal to k .

Fig.7.7(a) displays two example XML trees with numbered nodes. The corresponding matching matrix is shown in Fig.7.7(b).

Selection of best matchings. The problem of multiple matchings can be addressed by discarding those matchings which are less relevant according to the weighting function ω_m . The weight $\omega_m(v, w)$, associated to two matching nodes $v \in V_{t_1}$ and $w \in V_{t_2}$, is computed by taking into account the matches between the children nodes of both v and w . Formally, $\omega_m(v, w) = 1 + \sum_{i,j} \omega_m(v_i, w_j)$, where nodes v_i, w_j are such that $(v, v_i) \in E_{t_1}$ and $(w, w_j) \in E_{t_2}$.

Fig.7.7(c) shows the weights associated with each possible node pair.

Multiple matchings relative to any node of t_1 (resp. t_2) can be detected by checking multiple entries with non-zero values within the corresponding row (resp. column) of M_m . We now describe the process for detecting multiple matchings. In the following we focus on the identification of nodes within t_1 that have multiple matchings with those in t_2 : the dual situation (i.e. identification of nodes in t_2 having multiple matching with nodes in t_1) has a similar treatment.



(a) Examples XML trees t_1 and t_2

	0	1	2	3	4	5	6	7	8	9	10	11
0	1											
1		0	1									
2		0	1									
3		1	0									
4				0	0	1						
5				0	0	1						
6				1	0	0						
7				0	1	0						
8							0	0	0	0	0	
9							1	0	0	0	0	
10							0	1	1	0	0	
11							0	0	0	0	0	

(b) Matching matrix

	0	1	2	3	4	5	6	7	8	9	10	11
0	11											
1		0	2									
2		0	2									
3		6	0									
4				0	0	1						
5				0	0	1						
6				4	0	0						
7				0	1	0						
8							0	0	0	0	0	
9							1	0	0	0	0	
10							0	1	1	0	0	
11							0	0	0	0	0	

0	3	1	6	7	4	9	10	10	-1	-1	-1
---	---	---	---	---	---	---	----	----	----	----	----

(c) Matching selection

Fig. 7.7. Data structures for the construction of an optimal matching tree.

Let $v_i \in V_{t_1}$ denote the node corresponding to the i -th row in M_m , and let $J_{v_i} = \{j_1, \dots, j_h\}$ be the set of column indexes, corresponding to the nodes w_{j_1}, \dots, w_{j_h} of t_2 , such that $M_m(i, j_k) > 0$ (i.e. such that $\omega_m(v_i, w_{j_k}) > 0$), $k = [1..h]$. Thus, v_i exhibits multiple matchings if $|J_{v_i}| > 1$. For each node $v_i \in V_{t_1}$, the *best matching* node corresponds to the column index $j_{v_i}^* = \arg \max_{j_1, \dots, j_h} \{M_m(i, j_1), \dots, M_m(i, j_h)\}$. If the maximum in $\{M_m(i, j_1), \dots, M_m(i, j_h)\}$ is not unique we choose $j_{v_i}^*$ to be the minimum index. The overall best matchings for nodes of t_1 can be easily tracked by using a *marking vector* $V_{m_1} = \{j_{v_1}^*, \dots, j_{v_n}^*\}$, whose generic i -th entry indicates the node of t_2 with which $v_i \in V_{t_1}$ has the best matching. We set $V_{m_1}[i] = -1$ if the node $v_i \in V_{t_1}$ has no matching. Fig.7.7(c) shows the marking vectors V_{m_1} and V_{m_2} associated with t_1 and t_2 , respectively.

Optimal matching tree construction. An optimal matching tree is effectively built by exploiting the above marking vectors: it suffices that all nodes with no matching are discarded. Fig.7.8(a) shows the optimal matching tree computed

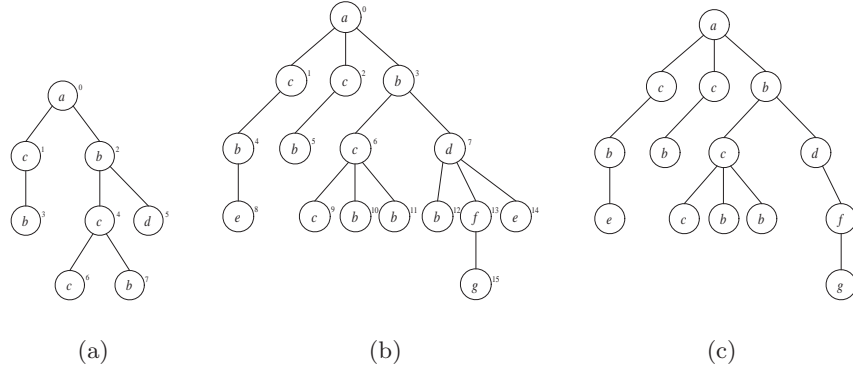


Fig. 7.8. Lower-bound (optimal matching tree) (a), upper-bound (merge tree) (b), and optimal representative tree (c) relative to the trees of Fig.7.7(a).

for t_1 and t_2 of Fig.7.7(a). As we can see in the figure, the optimal matching tree is obtained from t_1 by removing nodes 2, 5, 8, 11.

7.3.2 Building a Merge Tree

The optimal matching tree of two documents represents an optimal intersection between the documents. The notion of *merge tree* resembles an optimized *union* of the original trees. Notice that, firstly an optimal matching tree has to be detected, in order to avoid redundant nodes to be added. Indeed, a trivial merge tree could be simply built as the union of the trees under investigation. Function `buildMerge` in Fig.7.6 details the construction of a merge tree, which takes into account nodes discarded while building the optimal matching tree. To this purpose, given two trees t_1 and t_2 , we first consider nodes in t_1 having duplicate nodes, and insert such duplicates into the merge tree. Next, nodes in t_2 which do not match with any node in t_1 are added.

Fig.7.8(b) shows the merge tree associated to the trees of Fig.7.7(a). Nodes 8, 11 from t_1 and 9, 10, 11 from t_2 have no matching, whereas nodes 2, 5 from t_1 and 8 from t_2 exhibit multiple matchings.

7.3.3 Properties of Merging Process

Since the notion of cluster representative has to be generalized to a set of XML documents, it is interesting to evaluate whether the proposed *merge* function is both *commutative* and *associative*, i.e., whether $merge(t_1, t_2) = merge(t_2, t_1)$ and $merge(t_1, merge(t_2, t_3)) = merge(merge(t_1, t_2), t_3)$. The former property can be easily verified, whereas the latter does not hold. Notwithstanding, the set of cluster representatives, that can be computed by taking different orders of the data, represents an equivalence class on document similarity as stated by the following:

Theorem 7.1 Let t_1 , t_2 , and t_3 be XML trees. Moreover, assume that r' , r'' and r''' are defined as:

- $r' = \text{merge}(t_1, \text{merge}(t_2, t_3))$,
- $r'' = \text{merge}(t_2, \text{merge}(t_1, t_3))$,
- $r''' = \text{merge}(t_3, \text{merge}(t_1, t_2))$.

Then, it holds that

$$d_J(t_i, r') = d_J(t_i, r'') = d_J(t_i, r''') \quad (7.1)$$

$$d_E(t_i, r') = d_E(t_i, r'') = d_E(t_i, r''') \quad (7.2)$$

for each $i \in \{1, 2, 3\}$.

Proof. The following two lemmas prove the statement on d_J and d_E □

Lemma 7.1 $d_J(t_i, r') = d_J(t_i, r'') = d_J(t_i, r''')$ for each $i \in \{1, 2, 3\}$.

Proof. The statement is shown with respect to $d_J^{(2)}$, since the proof for $d_J^{(1)}$ is analogous. Let $\text{path}(t)$ denote the set of paths within a generic tree t . The definition of merge tree implies that:

$$\begin{aligned} \text{path}(\text{merge}(t_2, t_3)) &= \text{path}(t_2) \cup \text{path}(t_3), \\ \text{path}(\text{merge}(t_1, t_3)) &= \text{path}(t_1) \cup \text{path}(t_3), \\ \text{path}(\text{merge}(t_1, t_2)) &= \text{path}(t_1) \cup \text{path}(t_2) \end{aligned}$$

Thus, it follows that:

$$\text{path}(r') = \text{path}(t_1) \cup \text{path}(\text{merge}(t_2, t_3)) = \text{path}(t_1) \cup (\text{path}(t_2) \cup \text{path}(t_3))$$

Since the union operator is associative, it holds that:

$$\text{path}(r') = \text{path}(t_1) \cup \text{path}(t_2) \cup \text{path}(t_3) = \text{path}(r'') = \text{path}(r''') \quad \square$$

Lemma 7.2 $d_E(t_i, r') = d_E(t_i, r'') = d_E(t_i, r''')$ for each $i \in \{1, 2, 3\}$.

Proof. Assuming a unit cost for the removal of a node, $d_E(t_i, r') = |r'| - |t_i|$ ¹, $i = 1, 2, 3$. It must be shown that $|r'| = |r''| = |r'''|$ to prove that a merge tree is associative with respect to the edit distance. Let $\text{match}^*(t_1, t_2)$ be an optimal matching tree between trees t_1 and t_2 . Since $|\text{merge}(t_2, t_3)| = |t_2| + |t_3| - |\text{match}^*(t_2, t_3)|$, it holds that:

$$\begin{aligned} |r'| &= |\text{merge}(t_1, \text{merge}(t_2, t_3))| \\ &= |t_1| + |\text{merge}(t_2, t_3)| - |\text{match}^*(t_1, \text{merge}(t_2, t_3))| \\ &= |t_1| + |t_2| + |t_3| - |\text{match}^*(t_2, t_3)| - |\text{match}^*(t_1, t_2)| - |\text{match}^*(t_1, t_3)| + \\ &\quad |\text{match}^*(\text{match}^*(t_1, t_2), \text{match}^*(t_1, t_3))| \\ &= |t_1| + |t_2| + |t_3| - |\text{match}^*(t_2, t_3)| - |\text{match}^*(t_1, t_2)| - |\text{match}^*(t_1, t_3)| + \\ &\quad |\text{match}^*(t_1, t_2, t_3)| \end{aligned}$$

¹ Here, notation $|t|$ denotes the overall number of nodes within tree t .

The same result can be obtained for both r'' and r''' . Hence, it holds that $|r'| = |r''| = |r'''|$. \square

7.3.4 Turning a merge tree into a cluster representative

An effective cluster representative can be obtained by removing nodes from a merge tree in such a way to minimize the distance between the refined merge tree and the original XML trees in the cluster. Procedure **prune**, shown in Fig.7.6, iteratively tries to remove leaf nodes until the distance between the refined merge tree and the original trees in the cluster cannot be further decreased. It is worth noticing that, on the basis of the definition of procedure **prune**, the representative of a cluster is always bounded by the optimal matching tree built from the documents in that cluster. The correctness of the pruning procedure is established by the following result.

Theorem 7.2 *Let t_1, t_2 be two XML trees. Moreover, let $t_M = \text{merge}(t_1, t_2)$, $t_m = \text{match}(t_1, t_2)$ and $t^* = \text{rep}(\{t_1, t_2\})$. Then, $t_m \subseteq t^* \subseteq t_M$.*

Proof. The following lemmas prove this assertion. \square

Lemma 7.3 *Given a set $\mathcal{S} = \{t_1, t_2\}$ of XML trees, $\text{rep}(\mathcal{S}) \subseteq \text{merge}(t_1, t_2)$*

Proof. Let $t_M = \text{merge}(t_1, t_2)$ and $t^* = \text{rep}(\mathcal{S})$. Assume that $t^* \not\subseteq t_M$. Hence, t^* can be partitioned into two trees, $t_1^* = (r_{t^*}, V_1, E_1, \lambda_{t^*})$ and $t_2^* = (r_{t^*}, V_2, E_2, \lambda_{t^*})$, such that $\text{path}(t^*) = \text{path}(t_1^*) \cup \text{path}(t_2^*)$, and $\text{path}(t_1^*) \cap \text{path}(t_2^*) = \emptyset$. Moreover,

- $\text{path}(t_1^*) \subseteq \text{path}(t_M)$
- $\text{path}(t_2^*) \cap \text{path}(t_M) = \emptyset$
- $|\text{path}(t_2^*)| = w > 0$

Assume that $|\text{path}(t_1^*) \cup \text{path}(t_i)| = k_i$ and $|\text{path}(t_1^*) \cap \text{path}(t_i)| = h_i$ ($i = 1, 2$). Whenever $h_i > 0$, it holds that

$$d_J(t_i, t_1^*) = \frac{k_i - h_i}{k_i} < \frac{k_i + w - h_i}{k_i + w} = d_J(t_i, t^*)$$

Notice that $|\text{path}(t_1^*)| > 0$, since otherwise $\sum_{i=1}^2 d_J(t_i, t^*) = 2$, whereas for any t_j , $\sum_{i=1}^2 d_J(t_i, t^*) < 2$. Since $t_1^* \subseteq t_M$, there is at least a tree t_j ($j = 1, 2$) such that $h_j > 0$. As a consequence,

$$\sum_{i=1}^2 d_J(t_i, t_1^*) < \sum_{i=1}^2 d_J(t_i, t^*)$$

that is a contradiction, by the definition of t^* . \square

The above lemma states that the only tree paths that are to be taken into account to form a representative are those contained in the union of the paths within the original trees t_1 and t_2 . However, a cluster representative does not necessarily contain all of the paths in the above union.

Lemma 7.4 *Given a set $\mathcal{S} = \{t_1, t_2\}$ of XML trees, $\text{match}(t_1, t_2) \subseteq \text{rep}(\mathcal{S})$*

Proof. Let $t_m = \text{match}(t_1, t_2)$ and $t^* = \text{rep}(\mathcal{S})$. Again, the assertion is shown by contradiction. Assume that $t_m \not\subseteq t^*$. Hence, t^* can be decomposed into two trees, t_1^* and t_2^* , such that

- $\text{path}(t^*) = \text{path}(t_1^*) \cup \text{path}(t_2^*)$
- $\text{path}(t_1^*) \subset \text{path}(t_m)$
- $\text{path}(t_2^*) \cap \text{path}(t_m) = \emptyset$

A tree t' can be built from t_m and t_2^* as follows. Root $r_{t'}$ is chosen so that $\lambda_{t'}(r_{t'}) = \lambda_{t_m}(r_{t_m}) = \lambda_{t_2^*}(r_{t_2^*}) = r$. Paths in $\text{path}(t_m)$ are rooted at r . Then a path $p \in \text{path}(t_2^*)$ is added to t' by checking if any subpath of p already belongs to $\text{path}(t')$. Precisely, let p' be a prefix of p and p'' its corresponding suffix. If $p' \in \text{path}(t_2^*)$, p'' is appended to the last node in p' . Otherwise, p is rooted at r . This scheme, exemplified in fig. 7.9, guarantees that $\text{path}(t') = \text{path}(t_m) \cup \text{path}(t_2^*)$.

Now, for each tree $t_i \in \mathcal{S}$ ($i = 1, 2$), it holds that $d_J(t_i, t') < d_J(t_i, t^*)$. Formally,

$$\begin{aligned}
d_J(t_i, t') &= \\
&= \frac{|\text{path}(t_i) \cup \text{path}(t_m) \cup \text{path}(t_2^*)| - |\text{path}(t_i) \cap (\text{path}(t_m) \cup \text{path}(t_2^*))|}{|\text{path}(t_i) \cup \text{path}(t_m) \cup \text{path}(t_2^*)|} = \\
&= \frac{|\text{path}(t_i) \cup \text{path}(t_2^*)| - |(\text{path}(t_i) \cap \text{path}(t_m)) \cup (\text{path}(t_i) \cap \text{path}(t_2^*))|}{|\text{path}(t_i) \cup \text{path}(t_2^*)|} = \\
&= \frac{|\text{path}(t_i) \cup \text{path}(t_2^*)| - (|\text{path}(t_m)| + |\text{path}(t_i) \cap \text{path}(t_2^*)|)}{|\text{path}(t_i) \cup \text{path}(t_2^*)|} < \\
&< \frac{|\text{path}(t_i) \cup \text{path}(t_2^*)| - (|\text{path}(t_1^*)| + |\text{path}(t_i) \cap \text{path}(t_2^*)|)}{|\text{path}(t_i) \cup \text{path}(t_2^*)|} = \\
&= \frac{|\text{path}(t_i) \cup \text{path}(t_2^*)| - |(\text{path}(t_i) \cap \text{path}(t_1^*)) \cup (\text{path}(t_i) \cap \text{path}(t_2^*))|}{|\text{path}(t_i) \cup \text{path}(t_2^*)|} = \\
&= \frac{|\text{path}(t_i) \cup \text{path}(t_1^*) \cup \text{path}(t_2^*)| - |\text{path}(t_i) \cap (\text{path}(t_1^*) \cup \text{path}(t_2^*))|}{|\text{path}(t_i) \cup \text{path}(t_1^*) \cup \text{path}(t_2^*)|} = \\
&= d_J(t_i, t^*)
\end{aligned}$$

As a consequence, $\sum_{i=1}^2 d_J(t_i, t') < \sum_{i=1}^2 d_J(t_i, t^*)$, which leads to a contradiction. \square

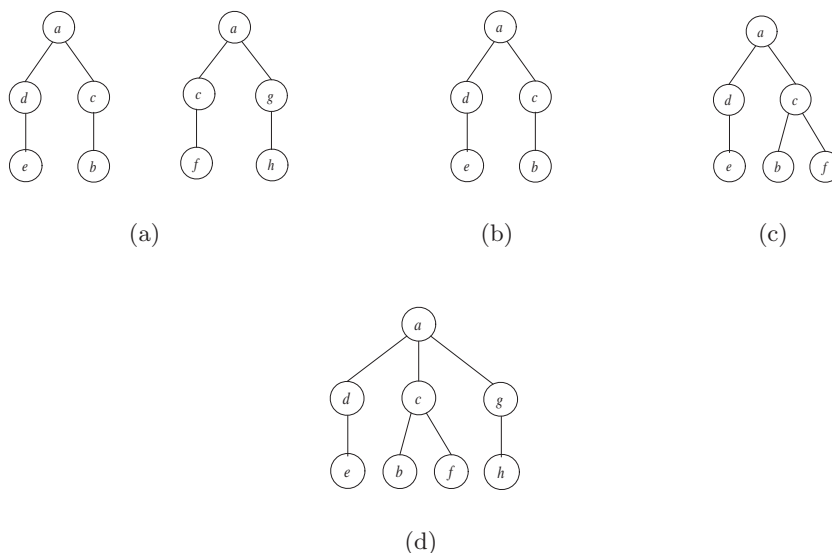


Fig. 7.9. The construction scheme employed in lemma 7.4. Trees t_m and t_2^* correspond to the ones respectively sited on the left and on the right sides of fig. 7.9(a). In fig. 7.9(b), t' is initially formed with paths in t_m . In fig.7.9(c), t' is augmented with path $\langle a, c, f \rangle$ from t_2^* . However, a prefix of this path, namely $\langle a, c \rangle$, already figures in t' . Hence, suffix $\langle f \rangle$ is appended to node c in t' . Finally, in fig.7.9(d), t' is further augmented with path $\langle a, g, h \rangle$, that is entirely appended to root a .

This result states that a cluster representative always contains the intersection of the paths within the original trees t_1 and t_2 . Still, a representative has a finer characterization. Indeed, it is never empty, though the optimal matching tree can be empty.

Lemma 7.5 *Given a set $\mathcal{S} = \{t_1, t_2\}$ of XML trees, $\text{rep}(\mathcal{S})$ is not empty.*

Proof. The assertion can be trivially shown by contradiction. Assume that $t^* = \text{rep}(\mathcal{S})$ is empty. In such a case, $\sum_{i=1}^2 d_J(x_i, t^*) = 2$, whereas for any $t_j \in \mathcal{S}$ ($j = 1, 2$) it holds that $\sum_{i=1}^2 d_J(x_i, t_j) < 2$. \square

Example 7.5. Let us consider again the trees t_1 and t_2 of Fig.7.7(a) and their associated merge tree $\text{merge}(t_1, t_2)$ in Fig.7.8(b). Suppose that t_1 and t_2 belong to the same cluster C . In order to compute a representative tree for C , the pruning procedure is initially applied to the set of leaves $\mathcal{L} = \{5, 8, \dots, 12, 14, 15\}$. If we choose to adopt the $d_J^{(2)}$ distance, the procedure computes an initial intra-cluster distance $d_0^C = 5/8$. This distance is reduced to $4/7$ as leaf node 14 is removed. Yet, d_0^C can be further decreased

by removing node 12. Since at this point no further node contributes to the minimization of d_0^C , the pruning process ends. Fig.7.8(c) shows the cluster representative resulting from pruning the merge tree in Fig.7.8(b), with the adoption of the $d_j^{(2)}$ distance. \square

7.4 Evaluation

We evaluated the effectiveness of *XRep* by performing experiments on both synthetic and real data. In the former case, we mainly aimed at assessing the effectiveness of our clustering scheme with respect to some prior knowledge about the structural similarities among the XML documents taken into account. Specifically, we exploited a synthetic data set that comprises seven distinct classes of XML documents, where each such class is a structurally homogeneous group of documents randomly generated from a previously chosen DTD. Tests were performed in order to investigate the ability of *XRep* in catching such groups.

To the purpose of assembling a valuable data set, we developed an automatic generator of synthetic XML documents, that allows the control of the degree of structural resemblance among the document classes under investigation. The generation process works as follows. Given a seed DTD DTD_0 , a similarity threshold τ , and a number k of classes, the generator randomly yields a set S_τ^k of k different DTDs, hereinafter called class DTDs, that individually retain at most τ percent of the element definitions within DTD_0 . The k class DTDs are eventually leveraged to generate as many collections of conforming XML documents, on the basis of suitable statistical models ruling the occurrences of the document elements [48].

The seed DTD was manually developed and exhibits a quite complex structure. For the sake of brevity, we only focus on its major features. DTD_0 contains 30 distinct element declarations that adopt neither attributes nor recursion. Non empty elements contain at most 4 children. Yet, the occurrences of such elements are suitably defined by exploiting all kinds of operators, namely $+$, $*$, $?$, and $|$. Finally, the tree-based representation of any XML document conforming to DTD_0 has a depth that is equal to 6.

Each test on synthetic data was performed on a distinct set of seven class DTDs, sampled from DTD_0 , at increasing values of the similarity threshold τ : we chose τ to be respectively equal to 0.3, 0.5, and 0.8.

Real XML documents were extracted from six different collections available on Internet:

- Astronomy, 217 documents extracted from an XML-based metadata repository, that describes an archive of publications owned by the *Astronomical Data Center* at NASA/GSFC.
- Forum, 264 documents concerning messages sent by users of a Web forum.

- News, 64 documents concerning press news from all over the world, daily collected by *PR Web*, a company providing free online press release distribution.
- Sigmod, 51 documents concerning issues of SIGMOD Record. Such documents were obtained from the XML version of the ACM SIGMOD Web site produced within the *Araneus* project [39].
- Wrapper, 53 documents representing wrapper programs for Web sites, obtained by means of the *Lixto* system [13].
- Xyleme_Sample, a collection of 1000 documents chosen from the *Xyleme*'s repository, which is populated by a Web crawler using an efficient native XML storage system [103].

The distribution of tags within these documents is quite heterogeneous, due to the complexity of the DTDs associated with the classes, and to the semantic differences among the documents. In particular, wrapper programs may have substantially different forms, as a natural consequence of the structural differences existing among the various Web sites they have been built on: thus, the skewed nature of the documents in *Wrapper* should be taken into account. Also, documents sampled from *Xyleme* exhibit a more evident heterogeneity, since they have been crawled from very different Web sources.

Clustering results were evaluated by exploiting the standard *precision* and *recall* measures [11]. However, in the case of *Xyleme_Sample*, we had no knowledge of an a-priori classification. As a consequence, we resorted to an internal quality criterium that takes into account the compactness of the discovered clusters. More precisely, given a cluster partition $\mathcal{P} = \{C_1, \dots, C_n\}$, where $C_i = \{x_1^i, \dots, x_{n_i}^i\}$, we defined an *intra-cluster distance* measure as: $\mathcal{IC}(\mathcal{P}) = \frac{1}{n} \sum_{C_i \in \mathcal{P}} \frac{1}{n_i} \sum_{x \in C_i} d(x, rep(C_i))$.

Table 7.1 summarizes the quality values obtained testing *XRep* on both synthetic and real data. All the experiments have been carried out by adopting the Jaccard distance $d_J^{(2)}$ introduced in Section 7.2. Tests on synthetic data evaluated the performance of *XRep* on three collections of 1400 documents (200 documents for each class DTD). Experimental evidence highlights the overall accuracy of *XRep* in distinguishing among classes of XML documents characterized by different average sizes due to different choices for the threshold τ . As we can see, *XRep* exhibits an excellent behavior for $\tau = \{0.3, 0.5\}$, while the acceptable performance reported on row 3 (i.e. $\tau = 0.8$) is due to the intrinsic difficulty in catching minimal differences in the structure of the involved XML documents. Indeed, two clearly distinct class DTDs, namely DTD_i and DTD_j , may share a number of element definitions inducing similar paths within the conforming XML documents. If such definitions assign multiple occurrences to the elements of the common paths, the initial class separation between DTD_i and DTD_j may be potentially vanished by a strong degree of document similarity due to a large number of common paths in the corresponding XML trees.

type	docs	avg size	classes	clusters	τ	precision	recall	F-measure	\mathcal{IC}
synth	1400	0.13KB	7	7	0.3	0.979	0.978	0.978	0.219
synth	1400	0.81KB	7	7	0.5	0.802	0.909	0.852	0.304
synth	1400	3.19KB	7	7	0.8	0.689	0.773	0.728	0.369
real	649	5.74KB	5	5	-	1	1	1	0.208
real	500	8.56KB	-	7	-	-	-	-	0.376
real	1000	9.42KB	-	9	-	-	-	-	0.43

Table 7.1. Quality results

Tests on real data considered separately the first five collections (649 XML documents with an average size that is equal to 5.74KB), and the `Xyleme_Sample` collection. In the first case, `XRep` showed amazingly optimal accuracy in identifying even latent differences among the involved real documents. As far as `Xyleme_Sample` is concerned, we conducted two experiments (rows 5 and 6 in Table 7.1), where in the first one we considered only one and a half of the dataset. However, as we expected, in both cases intra-cluster distance provides fairly good values: this is mainly due to the high heterogeneity which characterizes documents in `Xyleme_Sample`.

Conclusions and further work

8.1 Summary

XML (Extensible Markup Language) is a standard meta language used to describe a class of data objects, called XML documents and to specify how they are to be processed by computer programs. In a very short space of time, XML has become a hugely popular format for marking up all kinds of data, from web content to data used by applications. However, a significant disadvantage of XML technology is document size, which is a consequence of verbosity arising from markup information. It is commonly observed that non-standardized text formats for describing equivalent data are significantly shorter. Theoretically, therefore, one should be able to compress XML documents down to the same size as the compressed versions of their non-standard counterparts.

From an information-theoretic standpoint, portions of the document that have to do with its layout should not add to its entropy. These considerations lead naturally to investigate the use of suitable models for the compression of such data. Present day XML documents are ubiquitous and the need for compression is pressing.

A desirable feature of a compression scheme is the ability to be able to query the compressed document without decompressing the whole document. To this purpose, in this thesis, we have presented our approach XQueC, a compression-aware XQuery processor. XQueC works on compressed XML documents, which can be a huge advantage when query results must be shipped around a network. The architecture of the system has been designed in such way to hold separated structure from content. Use of the Structure tree to represent the structure of document and employment of containers for the content, have allowed improvement on query evaluation. Indeed, thanks to this approach, during the phase of evaluation of a query, the system will exclusively access the necessary data to get the result without having to load in memory container that has not involved in the query. Evaluation of queries in compressed domain is possible thanks to exploitation of order-preserving

compression algorithm ALM that has been chosen for its features that make it comparable with general-purpose compression algorithms. The experimental tests executed on the system have furnished very encouraging results: XQueC exhibits a good trade-off between compression factors over different XML data sets and query evaluation times on XMark queries..

The rapid growth of XML documents available has posed the scientific community in face of new interesting challenges. In order to analyze huge amount of information formatted with XML, decomposing the XML documents and storing them in relational tables is a popular practice. However, query processing becomes expensive since, in many cases, an excessive number of joins is required to recover information from the fragmented data. If a collection consists of documents with different structures, mining clusters in the documents could alleviate the fragmentation problem. Furthermore, many Web applications that process XML data, such as grouping similar XML documents and searching for XML documents that match a sample XML document require techniques for clustering efficiently XML documents. It has well established in such fields as database management and information retrieval that the more semantic (e.g. metadata) about data are understood by a system, the more precise the queries can become.

To this purpose, we have presented a novel methodology for clustering XML documents, focusing on the notion of *XML cluster representative* which is capable of capturing the significant structural specifics within a collection of XML documents. By exploiting the tree nature of XML documents, we provided suitable strategies for tree matching, merging, and pruning. Tree matching allows the identification of structural similarities to build an initial substructure that is common to all the XML document trees in a cluster, whereas the phase of tree merging leads to an XML tree that even contains uncommon document substructures. Moreover, we devised a suitable pruning strategy for minimizing the distance between the documents in a cluster and the document built as the cluster representative. The clustering framework was validated both on synthetic and real data, revealing high effectiveness.

8.2 Further research

We conclude by mentioning some future developments worth further research. The XQueC system proposed in chapter 4 can be improved in several ways: by moving to three-valued IDs for XML elements, in the spirit of [134, 135, 118] and by incorporating further storage techniques that lead to additionally reduce the occupancy of structures. Moreover, the implementation of our XQuery optimizer for querying compressed XML data is ongoing. Furthermore, we are testing the suitability of our system w.r.t. the full-text queries [151], which are being defined for the XQuery language at W3C. Another important extension we have devised is needed for uploading in our system very large documents. In order to do this, we plan to access the con-

tainers during the parsing phase directly on secondary storage rather than in memory, as the current implementation does. Conceivably, the OS-based containers on disk are temporary and can be removed once the upload in BerkeleyDB is completed. Other more general future research directions include, among the others, the capability of issuing updates over compressed XML data and of windowing compressed XML data streams.

Also, forthcoming research on our clustering scheme proposed in chapter 7 is possible. Further notions of cluster representative can be investigated, e.g. by relaxing the requirement that a prototype corresponds to a single XML document. Indeed, there are many cases in which a collection of XML documents is better summarized by a forest of subtrees, where each subtree represents a given peculiarity shared by some documents in the collection. A typical case raises, for instance, when the collection has an empty matching tree, and still exhibits significant homogeneities. Moreover, we believe that the proposed clustering scheme can be profitably applied to query optimization.

To this purpose, the aim is to devise an effective mechanism for searching and retrieving data from large databases of XML documents by XPath queries. In such context, representatives play a key role: indeed, they can be exploited to match queries against clusters, rather than against individual XML documents, to the purpose of progressively narrowing the search space during the identification of results to an XPath query q . In practice, this can be accomplished by discarding those clusters, at any level in the hierarchy produced by the hierarchical clustering algorithm, whose representatives are uncomplying with q . Indeed, each XML document can be represented as a tree. Trees are also suited for modelling queries in the Core XPath fragment [61, 63], i.e. those queries that essentially allows to define navigational paths for traversing an XML tree, to the purpose of extracting a set of nodes reachable (from one or multiple starting nodes) through these paths. So, an answer to a XPath query q on an XML document t can be found by “matching” q against t . If the query q is on a collection of XML documents, the answers can be conceptually, found by performing a sequential scan over all documents in the collection. However, this is not computationally feasible, since not only it requires accessing the whole database, whose size may be potentially huge, but, also, it requires checking whether q is a subtree of each individual document in dataset. The process of query handling would take considerable advantage of an index structure, that organizes documents of the collection into subsets of trees, capable of providing answers to the same XPath queries. Intuitively, this would guarantee that the set of answers to a query q can be obtained from inspecting a narrowed search space, corresponding to the documents of the database within those subsets, that are actually relevant for q .

Lastly, it would be interesting to extend the XML clustering process to first order logic which would give a much more expressive language for data and patterns. In particular, using this powerful representation that can handle relations and thus, structure, it is possible to represent general relationships embedded in data, transforming XML documents into a set of logic rules and

facts. Hence, the set can be partitioned using the *XRep* clustering technique, exploiting an ad hoc distance measure calculated on first order predicates.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000.
2. R. Agrawal, C. Faloutsos, and A. Swami. Efficient Similarity Search in Sequence Databases. In *Proceedings of the International Conference of Foundations of Data Organization (FODO)*, pages 69–84, 1993.
3. S. Amer-Yahia and M. Fernandez. Overview of existing XML storage techniques. Technical report, AT&T Labs Research, 2002.
4. P. Andritsos, P. Tsaparas, R. J. Miller, and K. C. Sevcik. LIMBO: Scalable Clustering of Categorical Data. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 123–146, 2004.
5. M. Ankerst, M. M. Breunig, H. P. Kriegel, and J. Sander. OPTICS: Ordering Points To Identify the Clustering Structure. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, pages 49–60, 1999.
6. G. Antoshenkov, D. Lomet, and J. Murray. Order Preserving String Compression. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 655–663, 1996.
7. Gennady Antoshenkov. Dictionary-Based Order-Preserving String Compression. *VLDB Journal*, 6(1):26–39, 1997.
8. A. Arion, A. Bonifati, G. Costa, S. D’Aguanno, I. Manolescu, and A. Pugliese. XQuec: Pushing Queries to Compressed XML Data. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1065–1068, 2003.
9. A. Arion, A. Bonifati, G. Costa, S. D’Aguanno, I. Manolescu, and A. Pugliese. Efficient Query Evaluation over Compressed XML Data. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 200–218, 2004.
10. T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto, and S. Arikawa. Efficient Substructure Discovery from Large Semi-Structured Data. In *Proceedings of the SIAM International Conference on Data Mining (SDM)*, 2002.
11. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press Books. Addison Wesley, 1999.
12. D. Barbará, Y. Li, and J. Couto. COOLCAT: an entropy-based algorithm for categorical clustering. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 582–589, 2002.

13. R. Baumgartner, S. Flesca, and G. Gottlob. Visual Web Information Extraction with Lixto. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 119–128, 2001.
14. S. Bergamaschi, S. Castano, and M. Vincini. Semantic Integration of Semistructured and Structured Data Sources. *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, 28(1):54–59, 1999.
15. Berkeley DB Data Store. <http://www.sleepycat.com/products/data.shtml/>.
16. E. Bertino, G. Guerrini, and M. Mesiti. Measuring the structural similarity among XML documents and DTDs. Technical report, DISI-TR-02-02, University of Genova, 2002. Available at <http://www.disi.unige.it/person/MesitiM>.
17. E. Bertino, G. Guerrini, and M. Mesiti. A matching algorithm for measuring the structural similarity between an XML document and a DTD and its applications. *Information Systems*, 29(1), 2004.
18. C. L. Blake and C. J. Merz. UCI repository of machine learning databases, 1998.
19. P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML Schema to Relations: A Cost-based Approach to XML Storage. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2002.
20. P. Bradley, U. Fayyad, and C. Reina. Scaling Em (Expectation-Maximization) Clustering to Large Databases. Technical report, Microsoft Research, MSR-TR-98-35, 1998.
21. P. S. Bradley, U. M. Fayyad, and C. Reina. Scaling Clustering Algorithms to Large Databases. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 9–15, 1998.
22. P. Buneman, M. Grohe, and C. Koch. Path Queries on Compressed XML. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2003.
23. The bzip2 and libbzip2 official home page. <http://www.bzip.org/>.
24. I.V. Cadez, S. Gaffney, and P. Smyth. A General Probabilistic Framework for Clustering Individuals and Objects. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, 2000.
25. M. Cannataro, G. Carelli, A. Pugliese, and D. Sacca. Semantic Lossy Compression of XML Data. In *Proceedings of the International Workshop on Knowledge Representation meets Databases (KRDB)*, 2001.
26. M. Cannataro, C. Comito, and A. Pugliese. Squeezex: Synthesis and Compression of XML Data. In *Proceedings of the International Conference on Information Technology: Coding and Computing*, pages 326–331, 2002.
27. B. Catania and A. Maddalena. A Clustering Approach for XML Linked Documents. In *Proceedings of the International Workshop on Database and Expert Systems Applications (DEXA)*, pages 121–128, 2002.
28. S. Chakrabarti. *Mining the Web: Discovering Knowledge from Hypertext Data*. Morgan-Kaufmann, 2002.
29. S. S. Chawathe. Comparing Hierarchical Data in External Memory. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 90–101, 1999.
30. S. S. Chawathe, A. Rajaraman, H. Molina, and J. Widom. Change Detection in Hierarchically Structured Information. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, pages 493–504, 1996.

31. Z. Chen, J. Gehrke, and F. Korn. Query Optimization In Compressed Database Systems. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, 2000.
32. Z. Chen and P. Seshadri. An algebraic compression framework for query results. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2000.
33. James Cheney. Compressing XML with Multiplexed Hierarchical PPM Models. In *Proceeding of International Conference on Data Compression*, pages 163–172, 2001.
34. S. Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2002.
35. G. Cobena, S. Abiteboul, and A. Marian. Detecting Changes in XML Document. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 41–52, 2002.
36. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.
37. G. Costa, G. Manco, R. Ortale, and A. Tagarelli. A Tree-Based Approach to Clustering XML Documents by Structure. In *Proceedings of the European Conference on Principles of Knowledge Discovery in Databases (PKDD)*, pages 137–148, 2004.
38. G. Costa, G. Manco, R. Ortale, and A. Tagarelli. Clustering of XML Documents by Structure based on Tree Matching and Merging. In *Proceedings of the Italian Symposium on Advanced Database Systems (SEBD)*, pages 314–325, 2004.
39. V. Crescenzi, G. Mecca, and P. Merialdo. RoadRunner: Towards Automatic Data Extraction from Large Web Sites. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 109–118, 2001.
40. T. Dalamagas, T. Cheng, K. J. Winkel, and T. Sellis. A methodology for clustering XML documents by structure. *Information Systems*, 31(3):187–228, 2005.
41. D. Dhyani, W. K. Ng, and S. S. Bhowmick. A survey of Web metrics. *ACM Computing Surveys*, 34(4):469–503, 2002.
42. A. Doucet and H. A. Myka. Naive clustering of a large XML document Collection. In *Proceedings of the INEX Workshop*, 2002.
43. M. Ester, H. P. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 226–231, 1996.
44. D. Fasulo. An analysis of Recent Work on Clustering Algorithms. Technical report, University of Washington, 1999. Available at <http://www.cs.washington.edu/homes/dfasulo>.
45. T. Fiebig and G. Moerkotte. Algebraic XML Construction and its Optimization in Natix. *World Wide Web*, 4(3):167–187, 2001.
46. D. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2:139–172, 1987.
47. S. Flesca, G. Manco, and E. Masciari. Web wrapper induction: a brief survey. *AI Communications*, 17(2):57–61, 2004.

48. S. Flesca, G. Manco, E. Masciari, L. Pontieri, and A. Pugliese. Detecting structural similarities between XML documents. In *Proceedings of the International Workshop on The Web and Databases (WebDB)*, pages 55–60, 2002.
49. J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Simeon. StatiX: Making XML count. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, 2002.
50. M. Frigo and S. G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, 1998.
51. W. McKenna G. Graefe. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 1993.
52. G. Gan and J. Wu. Subspace clustering for high dimensional categorical data. *SIGKDD Explorations*, 6(2):87–94, 2004.
53. V. Ganti, J. Gehrke, and R. Ramakrishnan. CACTUS: Clustering Categorical Data Using Summaries. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 73–83, 1999.
54. H. Garcia-Molina et al. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.
55. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.
56. M. Girardot and N. Sundaresan. Millau: an encoding format for efficient representation and exchange of XML over the Web. In *Proceedings of the international World Wide Web conference on Computer networks*, pages 747–765, 2000.
57. R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 436–445, 1997.
58. J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing Relations and Indexes. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 370–379, 1998.
59. T. Gonzalez. Clustering to Minimize the Maximum Intercluster Distance. *Theoretical Computer Science*, 38:293–306, 1985.
60. A. Gordon. *Classification*. Chapman & Hall/CRC Press, 1999.
61. G. Gottlob and C. Koch. Monadic Queries over Tree-Structured Data. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*, pages 189–202, 2002.
62. G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2002.
63. G. Gottlob, C. Koch, and R. Pichler. The complexity of XPATH query evaluation. In *Proceedings of the ACM Symposium on Principle of Databases Systems*, pages 179–190, 2003.
64. C. Gozzi, F. Giannotti, and G. Manco. Clustering Transactional Data. In *Proceedings of the European Conference on Principles of Knowledge Discovery in Databases (PKDD)*, pages 175–187, 2002.
65. Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

66. G. Guerrini, M. Mesiti, and I. Sanz. *An Overview of Similarity Measures for Clustering XML Documents*. Idea Group Publishing, 2006.
67. S. Guha, R. Rastogi, and K. Shim. CURE: an efficient clustering algorithm for large databases. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, pages 73–84, 1998.
68. S. Guha, R. Rastogi, and K. Shim. ROCK: A Robust Clustering Algorithm for Categorical Attributes. *Information Systems*, 25(5):345–366, 2000.
69. D. Guillaume and F. Murtagh. Clustering of XML documents. *Computer Physics Communications*, 127:215–227, 2000.
70. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
71. E. Han and G. Karypis. Centroid-Based Document Classification: Analysis and Experimental Results. In *Proceedings of the European Conference on Principles of Knowledge Discovery in Databases (PKDD)*, pages 424–431, 2000.
72. J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufman, Los Altos, CA, 2001.
73. A. Hinneburg and D. A. Keim. An Efficient Approach to Clustering in Large Multimedia Databases with Noise. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 58–65, 1998.
74. T. C. Hu and A. C. Tucker. Optimal Computer Search Trees And Variable-Length Alphabetical Codes. *SIAM J. APPL. MATH*, 21(4):514–532, 1971.
75. Z. Huang. Extensions to the k-Means Algorithm for Clustering Large Data Sets with Categorical Values. *Data Mining and Knowledge Discovery*, 2(3):283–304, 1999.
76. D. A. Huffman. A Method for Construction of Minimum-Redundancy Codes. In *Proceedings of the IRE*, 1952.
77. D. Kossmann I. Manolescu, D. Florescu. Answering XML Queries over Heterogeneous Data Sources. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2001.
78. XML-Xpress. http://www.ictcompress.com/products_xmlxpress.html.
79. Yannis E. Ioannidis. Query optimization. *Handbook for Computer Science*, pages 1038–1057, 1997.
80. Carsten Isert. The Editing Distance Between Trees. Technical report, Ferienakademie, for course 2: B`ume: Algorithmik Und Kombinatorik, Italy, 1999.
81. A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.
82. C. C. Kanne and G. Moerkotte. Efficient Storage of XML Data. In *Proceedings of the International Conference on Data Engineering (ICDE)*, page 198, 2000.
83. G. Karypis, E. H. Han, and V. Kumar. Chameleon: Hierarchical Clustering Using Dynamic Modeling. *IEEE Computer*, 32(8):68–75, 1999.
84. H. Kashima and T. Koyanagi. Kernels for Semi-Structured Data. In *Proceedings of the International Conference on Machine Learning (ICML'02)*, pages 291–298, 2002.
85. L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley and Sons, 1990.
86. E. Keogh, S. Lonardi, and C. A. Ratanamahatana. Towards parameter-free data mining. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 206–215, 2004.

87. K. Krger and P. Gotlibovich. Incremental OPTICS: Efficient Computation of Updates in a Hierarchical Cluster Ordering. In *Proceeding of International Conference on Data Warehousing and Knowledge Discovery (DaWaK)*, 2003.
88. V. Kumar. An Introduction to Cluster Analysis for Data Mining. Technical report, University of Minnesota, 2000. Available at <http://www.cs.umn.edu/classes/Spring-2000/csci5980-dm/>.
89. D. A. Lelewer and D. S. Hirschberg. Data Compression. *ACM Computing Surveys*, 19(3), 1987.
90. W. Lian, D. Cheung, N. Mamoulis, and S. M. Yiu. An Efficient and Scalable Algorithm for Clustering XML Documents by Structure. *IEEE Trans. on Knowledge and Data Engineering*, 16(1), 2004.
91. H. Liefke and D. Suciu. XMill: An Efficient Compressor for XML Data. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, 2000.
92. H. Liu and L. Yu. Toward Integrating Feature Selection Algorithms for Classification and Clustering. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):491–502, 2005.
93. M. Liu, T. W. Ling, and T. Guan. Integration of Semistructured Data with Partial and Inconsistent Information. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)*, pages 44–52, 1999.
94. J. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. Le Cam and J. Neyman, editors, *Proceedings of the Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, 1967.
95. J. Madhavan, P. A. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 49–58, 2001.
96. A. Marian and J. Simeon. Projecting XML Documents. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2003.
97. J. McHugh and J. Widom. Optimizing branching path expressions. Technical report, Stanford University, 1999.
98. J. McHugh and J. Widom. Query Optimization for Semistructured Data. Technical report, Stanford University Database Group., 1999.
99. J. McHugh and J. Widom. Query Optimization for XML. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 315–326, 1999.
100. J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing semistructured data. Technical report, Stanford University, Computer Science Department., 1998.
101. G. Mecca, P. Merialdo, and P. Atzeni. Araneus in the Era of XML. *IEEE Data Engineering Bulletin*, 22(3):19–26, 1999.
102. P. Michaud. Clustering Techniques. *Future Generation Computer Systems*, 13, 1997.
103. L. Mignet, D. Barbosa, and P. Veltri. The XML Web: a First Study. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 500–510, 2003.
104. T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 277–295, 1999.
105. J. K. Min, J. Ahn, and C. Chung. Efficient Extraction of Schemas for XML Documents. *Information Processing Letters*, 85:7–12, 2003.

106. J. K. Min, M. Park, and C. Chung. XPRESS: A Queriable Compression for XML Data. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, pages 122–133, 2003.
107. B. Mobasher, R. Cooley, and J. Srivastava. Automatic Personalization Based on Web Usage Mining. *Communications of ACM*, 43(8), 2000.
108. R. Cooley B. Mobasher and J. Srivastava. Grouping Web Page References into Transactions for Mining World Wide Web Browsing Patterns. In *Proceedings of the IEEE Knowledge and Data Engineering Exchange Workshop (KDEX)*, 1997.
109. A. Moffat and J. Zobel. Coding for Compression in Full-Text Retrieval Systems. In *Proceedings of the International Conference on Data Compression (DCC)*, pages 72–81, 1992.
110. I. Muslea, S. Minton, and C. A. Knoblock. Hierarchical Wrapper Induction for Semistructured Information Sources. *Autonomous Agents and Multi-Agent Systems*, 4(1/2):93–114, 2001.
111. R. Nayak, R. Witt, and A. Tonev. Data Mining and XML Documents. In *Proceedings of the International Conference on Internet Computing*, pages 660–666, 2002.
112. R. T. Ng and J. Han. Efficient and Effective Clustering Methods for Spatial Data Mining. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 144–155. Morgan Kaufmann Publishers, 1994.
113. W. K. Ng and C. V. Ravishankar. Block-Oriented Compression Techniques for Large Statistical Databases. *Knowledge and Data Engineering*, 9(2):314–328, 1997.
114. A. Nierman and H. V. Jagadish. Evaluating Structural Similarity in XML Documents. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, 2002.
115. A.V. Oppenheim and R.W. Shafer. *Discrete-Time Signal Processing*. Prentice Hall, 1999.
116. M. M. Ozdal and C. Aykanat. Hypergraph Models and Algorithms for Data-Pattern-Based Clustering. *Data Mining and Knowledge Discovery*, 9:29–57, 2004.
117. M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 1999.
118. S. Pappas, S. Al-Khalifa, A. Chapman, H. V. Jagadish, V. S. Lakshmanan, A. Nierman, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER:A Native System for Querying XML. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, page 672, 2003.
119. M. Pelillo, K. Siddiqi, and S. W. Zucker. Matching Hierarchical Structures Using Association Graphs. In *Proceedings of the European Conference on Computer Vision*, 1998.
120. D. Pelleg and A. Moore. X-Means: Extending K-Means with Efficient Estimation of the Number of Clusters. In *Proceedings of the International Conference on Machine Learning*, pages 727–734, 2000.
121. M. Poess and D. Potapov. Data Compression in Oracle. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2003.
122. N. Polyzotis and M. Garofalakis. Statistical synopses for graph-structured XML databases. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, pages 358–369, 2002.

123. N. Polyzotis and M. Garofalakis. Structure and Value Synopses for XML Data Graphs. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 466–477, 2002.
124. H. Ralambondrainy. A Conceptual Version of the K-Means Algorithm. *Pattern Recognition Letters*, 16:1147–1157, 1995.
125. G. Ray, J. Haritsa, and S. Seshadri. Database compression: a performance enhancement tool. In *COMAD*, 1995.
126. M. Roth and S. Van Horn. Database Compression. *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, 22(3):31–39, 1993.
127. G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.
128. A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2002.
129. H. Schoning and J. Wasch. Tamino - An Internet Database System. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 383–387, 2000.
130. S. M. Selkow. The Tree-to-Tree Editing Problem. *Information Processing Letters*, 6(6):184–186, 1977.
131. J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2000.
132. G. Sheikholeslami, S. Chatterjee, and A. Zhang. Wave Cluster: A Multi-Resolution Clustering Approach for Very Large Spatial Databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 428–439, 24–27 1998.
133. D. Sheskin. *Handbook of parametric and nonparametric statistical procedures*. Boca Raton, Chapman & Hall/CRC, 2003.
134. D. Srivastava, S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, and Y. Wu. Accelerating XPath location steps. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, pages 109–120, 2002.
135. D. Srivastava, S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2002.
136. A. Strehl, J. Ghosh, and R. Mooney. Impact of Similarity Measures on Web-Page Clustering. In *Proceedings AAAI Workshop on Artificial Intelligence for Web Search*, pages 58–64, 2000.
137. D. Suci. Semistructured Data and XML. In *Proceedings of the International Conference on Foundations of Data Organization*, 1998.
138. P. N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison-Wesley, 2006.
139. M. Theobald, R. Schenkel, and G. Weikum. Exploiting Structure, Annotation, and Ontological Knowledge for Automatic Classification of XML Data. In *WebDB, International Workshop on Web and Databases, San Diego, California, June 12-13, 2003*, pages 1–6, 2003.
140. P. Tolani and J. Haritsa. XGrind: A Query-friendly XML Compressor. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2002.

141. J. D. Ullman. Information Integration using Logical Views. In *Proceedings of the International Conference on Database Theory*, volume 1186 of *Lecture Notes in Computer Science*, pages 19–40, 1997.
142. D. Papadias V. Delis. Querying Multimedia Documents By Spatiotemporal structure. In *Proceedings of the International Conference on Flexible Query Answering (FQAS)*, pages 126–137, 1998.
143. R. A. Wagner and M. J. Fischer. The String-to-String Correction Problem. *J. ACM*, 21(1):168–173, 1974.
144. K. Wang, C. Xu, and B. Liu. Clustering transactions using large items. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 483–490, 1999.
145. W. Wang, J. Yang, and R. R. Muntz. STING: A Statistical Information Grid Approach to Spatial Data Mining. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 186–195, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.
146. Y. Wang, D. J. DeWitt, and J. Cai. X-Diff: A Fast Change Detection Algorithm for XML Documents. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 519–530, 2003.
147. T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, 29(3):55–67, 2000.
148. J. Widom. Data Management for XML: Research Directions. *IEEE Data Engineering Bulletin*, 22(3):44–52, 1999.
149. I. H. Witten. Arithmetic Coding For Data Compression. *Communications of ACM*, 1987.
150. WAP Binary XML Content Format. <http://www.w3.org/TR/wbxml>.
151. XML Query Use Cases. <http://www.w3.org/TR/xmlquery-use-cases/>.
152. XML Schema. <http://www.w3.org/XML/Schema>.
153. XMLZip. <http://www.xmls.com>.
154. XML Path Language (XPath). <http://www.w3.org/TR/xpath>.
155. XQuery Language. <http://www.w3.org/XML/Query>.
156. Lucie Xyleme. Xyleme: A Dynamic Warehouse for XML Data of the Web. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)*, pages 3–7, 2001.
157. X. Yan and J. Han. gSpan: Graph-Based Substructure Pattern Mining. In *Proceedings IEEE International Conference on Data Mining (ICDM'02)*, pages 721–724, 2002.
158. Y. Yang, X. Guan, and J. You. CLOPE: a fast and effective clustering algorithm for transactional data. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 682–687, 2002.
159. B. Yi, H. V. Jagadish, and C. Faloutsos. Efficient Retrieval of Similar Time Sequences Under Time Warping. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 201–208, 1998.
160. M. J. Zaki and C. C. Aggarwal. XRules: an effective structural classifier for XML data. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 316–325, 2003.
161. M. J. Zaki and M. Peters. CLICK: Mining Subspace Clusters in Categorical Data via K-partite Maximal Cliques. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2005.

162. O. Zamir and O. Etzioni. Web Document Clustering: A Feasibility Demonstration. In *Proceedings of the International Conference on Research and Development in Information Retrieval (SIGIR)*, Event Detection and Clustering, pages 46–54, 1998.
163. K. Zhang and D. Shasha. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM Journal of Computing*, 18(6):1245–1262, 1989.
164. T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An Efficient Data Clustering Method for Very Large Databases. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, pages 103–114, 1996.
165. Z. Zhang, R. Li, S. Cao, and Y. Zhu. Similarity Metric for XML Documents. In *Workshop on Knowledge Experience and Management*, 2003.
166. Y. Zhao and G. Karypis. Empirical and Theoretical Comparisons of Selected Criterion Functions for Document Clustering. *Machine Learning*, 55(3):311–331, 2004.