Chapter 11

XML

Most research on data integration has, as described in the previous chapters of this book, centered on the relational model. In many ways, the relational model (and the Datalog query language) are the simplest and cleanest formalisms for data and query representation, so many of the fundamental issues were considered first in that setting.

However, as such techniques are adapted to meet real-world needs, they typically get adapted to incorporate XML (or its close cousin JSON[1]). For instance, IBM's Rational Data Architect or Microsoft's BizTalk Mapper both use XML-centric mappings.

The reason for this is straightforward. XML has become the default format for data export from both database and document sources; and many additional tools have been developed to export to XML from legacy sources (e.g., COBOL files, IMS hierarchical databases). Prior to XML's adoption, data integration systems needed custom wrappers that did "screen scraping" (custom HTML parsing and content extraction) to extract content from XML, and that translated to the proprietary wire formats of different legacy tools. Today, we can expect most sources to have an XML interface (URIs as the request mechanism and XML as the returned data format), and thus the data integrator can focus on the semantic mappings rather than the low-level format issues.

We note that XML brings standardization not only in the actual data format, but also in terms of an entire ecosystem of interfaces, standards, and tools: DTD and XML Schema for specifying schemas, DOM and SAX for language-neutral parser interfaces, WSDL/SOAP or REST for invoking Web services, text editors and browsers with

[1]JSON, the JavaScript Object Notation, can be thought of as a "simplified XML" although it also closely resembles previous *semistructured data* formats like the Object Exchange Model.

built-in support for XML creation, display, and validation. Any tool that produces and consumes XML automatically benefits from having these other components.

This book does not attempt to cover all of the details of XML, but rather to provide the core essentials. In this chapter, we focus first on the XML data model in Section 11.1 and the schema formalisms in Section 11.2. Section 11.3 presents several models for querying XML, culminating in the XQuery standard that is implemented in many XML database systems. We then discuss the subsets of XQuery typically used for XML schema mappings (Section 11.5) and then discuss XML query processing for data integration (Section 11.4).

11.1 Data Model

Like HTML (HyperText Markup Language), XML (eXtensible Markup Language) is essentially a specialized derivative of an old standard called SGML (Structured Generalized Markup Language). As with these other markup standards, XML encodes document meta-information using *tags* (in angle brackets) and *attributes* (attribute-value pairs associated with specific tags).

XML distinguishes itself from its predecessors in that (if correctly structured, or *well-formed*) it is *always parsable* by an XML parser — regardless of whether the XML parser has any information that enables it to interpret the XML tags. To ensure this, XML has strict rules about how the document is structured. We briefly describe the essential components of an XML document.

Processing instructions to aid the parser. The first line of an XML file tells the XML parser information about the character set used for encoding the remainder of the document; this is critical since it determines how many bytes encode each character in the file. Character sets are specified using a *processing-instruction*, such as `<?xml version="1.0" encoding="ISO-8859-1"?>`, which we see at the top of the example XML fragment in Figure 11.1 (an excerpt from the research paper bibliography Web site DBLP, at `dblp.uni-trier.de`). Other processing instructions may specify constraints on the content of the XML document, and we will discuss them later.

Tags, elements, and attributes. The main content of the XML document consists of tags, attributes, and data. XML tags are indicated using angle-brackets, and must come in pairs: for each open-tag `<tag>`, there must be a matching close-tag `</tag>`. An open-tag / close-tag pair and its contents are said to be an *XML element*. An element may have one or more *attributes*, each with a unique name and a value specified within the open-tag: `<tag attrib1="value1" attrib2="value2">`.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<dblp>
  <mastersthesis mdate="2002-01-03" key="ms/Brown92">
   <author>Kurt P. Brown</author>
   <title>PRPL: A Database Workload Specification Language</title>
   <year>1992</year>
   <school>Univ. of Wisconsin-Madison</school>
  </mastersthesis>
  <article mdate="2002-01-03" key="tr/dec/SRC1997-018">
   <editor>Paul R. McJones</editor>
   <title>The 1995 SQL Reunion</title>
   <journal>Digital System Research Center Report</journal>
   <volume>SRC1997-018</volume>
   <year>1997</year>
   <ee>db/labs/dec/SRC1997-018.html</ee>
   <ee>http://www.mcjones.org/System_R/SQL_Reunion_95/</ee>
 </article>
  ...
</dblp>
```

Figure 11.1: Sample XML data from the DBLP Web site

Of course, an element may contain nested elements, nested text, and a variety of other types of content we will describe shortly. It is important to note that every XML document must contain a single *root element*, meaning that the element content can be thought of as a tree and not a forest.

Example 11.1: Figure 11.1 shows a detailed fragment of XML from DBLP, as mentioned on the previous page. The first line is a processing instruction specifying the character set encoding. Next comes the single *root element*, `dblp`. Within the DBLP element we see two sub-elements, one describing a `mastersthesis` and the other an `article`. Additional elements are elided.

Both the MS thesis and article elements contain two attributes, `mdate` and `key`. Additionally, they contain sub-elements such as `author` or `editor`. Note that `ee` appears twice within the `article`. Within each of the sub-elements at this level is *text content*, each contiguous fragment of which is represented in the XML data model by a text node.

Namespaces and qualified names. Sometimes an XML document consists of content merged from multiple sources. In such situations, we may have the same tag names in several sources, and may wish to differentiate among them. In such a situation, we can give each of the source documents a *namespace*: this is a globally unique name, specified in the form of a Uniform Resource Indicator (URI). (The URI is simply a unique name specified in the form of a qualified path, and does not necessarily represent the address of any particular content. The more familiar Uniform Resource Locator, or URL, is a special case of a URI where there is a data item whose content can be retrieved according to the path in the URI.) Within an XML document, we can assign a much shorter name, the *namespace prefix*, to each of the namespace URIs. Then, within the XML document, we can "qualify" individual tag names with this prefix, followed by a colon, e.g., `<ns:tag>`. The *default namespace* is the one for all tags without qualified names.

Document order. XML was designed to serve several different roles simultaneously: extensible document format generalizing and replacing HTML; general-purpose markup language; structured data export format. It distinguishes itself from most database-related standards in that it is *order-preserving* and generally *order-sensitive*. More specifically, the order between XML elements is considered to be meaningful, and is preserved and queriable through XML query languages: this enables, e.g., ordering among paragraphs in a document to be maintained or tested. Perhaps surprisingly, XML *attributes* (which are treated as properties of elements) are *not* order-sensitive, although XML tools will typically preserve the original order.

We typically represent the logical or data model structure of an XML document as a tree, where each XML node is represented by a node in the tree, and parent-child relationships are encoded as edges. There are seven node types, briefly alluded to above:

- **Document root**: this node represents the entire XML document, and generally has as its children at least one processing instruction (representing the XML encoding information) and a single root element.
- **Processing instruction**: these nodes instruct the parser on character encodings, parse structure, etc.
- **Comment**: as with HTML comments, these are human-readable notes.
- **Element**: most data structures are encoded as XML elements, which include open and close tags plus content. A content-free element may be represented as a single *empty tag* of the form `<tag/>`, which is considered equivalent to an open-tag / close-tag sequence.

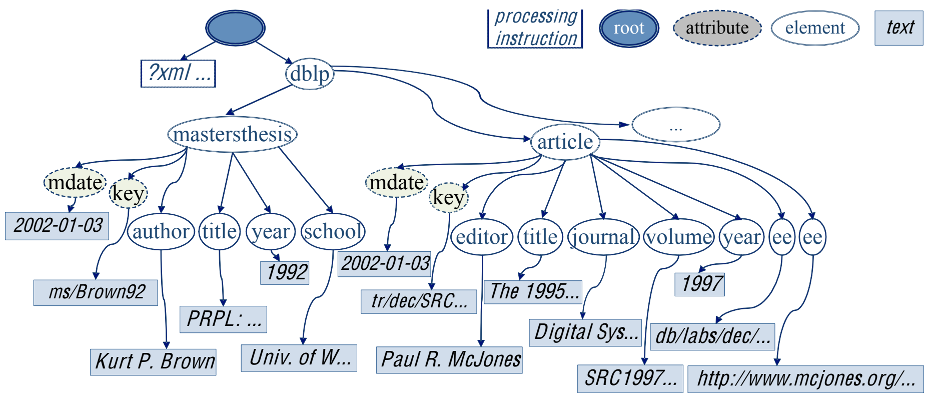

Figure 11.2: Example data model representation for the XML fragment of Figure 11.1. Note that the different node types are represented here using different shapes.

- **Attribute**: an attribute is a name-value pair associated with an element (and embedded in its open tag). Attributes are not order-sensitive, and no single tag may have more than one attribute with the same name.
- **Text**: a text node represents contiguous data content within an element.
- **Namespace**: a namespace node qualifies the name of an element within a particular URI. This creates a *qualified name*.

Each node in the document has a unique identity, as well as a relative ordering and (if a schema is associated with the document) a datatype. A depth-first, left-to-right traversal of the tree representation corresponds to the node ordering within the associated XML document.

Example 11.2: Figure 11.2 shows an XML data model representation for the document of Figure 11.1. Here we see five of the seven node types (comment and namespace are not present in this document).

11.2 XML Structural and Schema Definitions

In general, XML can be thought of as a semi-structured hierarchical data format, whose leaves are primarily string (text) nodes and attribute node values. To be most useful, we must add a *schema* describing the semantics and types of the attributes and elements — this enables us to encode non-string datatypes as well as inter- and intra-document links (e.g., foreign keys, URLs).

```
<!ELEMENT dblp((mastersthesis | article)*)>
<!ELEMENT mastersthesis(author,title,year,school,committeemember*)>
<!ATTLIST mastersthesis(mdate   CDATA  #REQUIRED
                        key     ID     #REQUIRED
                        advisor CDATA  #IMPLIED>
<!ELEMENT author(#PCDATA)>
<!ELEMENT title(#PCDATA)>
<!ELEMENT year(#PCDATA)>
<!ELEMENT school(#PCDATA)>
<!ELEMENT committeemember(#PCDATA)>
 ...
```

Figure 11.3: Fragment of an example DTD for the XML of Figure 11.1. Note that each definition here is expressed using a processing instruction.

11.2.1 Document Type Definitions (DTDs)

When the XML standard was introduced, the focus was on document markup, and hence the original "schema" specification was more focused on the legal structure of the markup than on specific datatypes and data semantics. The *document type definition* or DTD is expressed using processing instructions, and tells the XML parser about the structure of a given element.

In DTD, we specify for each element which sub-elements and/or text nodes are allowed within an element, using EBNF notation to indicate alternation or nesting. A sub-element is represented by its name, and a text node is designated as `#PCDATA` ("parsed character" data).

Example 11.3: Figure 11.3 shows a fragment of a DTD for our running example in this section. We focus in this portion of the example on the element definitions (`ELEMENT` processing instructions). The DBLP element may have a sequence of `mastersthesis` and `article` sub-elements, interleaved in any order. In turn the `mastersthesis` has mandatory `author`, `title`, `year`, and `school` sub-elements, followed by zero or more `committeemember`s. Each of these sub-elements contains text content.

Attributes are specified in a series of rows within the `ATTLIST` processing instruction: each attribute specification includes a name, a special type, and an annotation indicating whether the attribute is optional (`#IMPLIED`) or mandatory (`#REQUIRED`). The types are as follows: text content is called `CDATA` ("character data"), which is somewhat more restricted than the `PCDATA` allowed in elements; `ID` designates that the attribute contains an *identifier* that is globally unique within the document; `IDREF` or `IDREFS` specifies that the attribute contains a *reference* or space-separated references, respectively, to `ID`-typed attributes within the document. `ID`s and `IDREF`s can be thought of as special cases of keys and foreign keys, or anchors and links.

Example 11.4: Figure 11.3 defines three attributes related to the `mastersthesis`. The `mdate` is mandatory and of text-type. The `key` is also mandatory, but a unique identifier. Finally, the advisor string is optional.

Oddly enough, the DTD does not specify what element is the root within a document. The root element is instead specified within the processing instruction *in the source XML* that references the DTD. The DTD can be directly embedded within the document using a syntax like:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE dblp [
  <!ELEMENT dblp((mastersthesis | article)*)>
  <!ELEMENT mastersthesis(author,title,year,school,committeemember*)>
  ...
 ]>
<dblp>
 ...
```

but more commonly, the DTD is in a separate file that must be referenced using the `SYSTEM` keyword:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE dblp SYSTEM "dblp.dtd">
<dblp>
 ...
```

In both cases, the first parameter after `DOCTYPE` is the name of the root element, and the XML parser will parse the DTD before continuing beyond that point.

Pros and cons of DTD. DTD, as the first XML schema format, is very commonly used in practice. It is relatively simple to understand and concise (if syntactically awkward), it is supported by virtually every XML parser in existence, and it is sufficient for most document structural specifications. Unfortunately, it has many limitations for data interchange. One cannot directly map the database concepts of key and foreign key to `ID` and `IDREFS` (there is no support for compound keys, and the value of a key must be *globally unique* within a document, even to the exclusion of `ID`-typed attributes with other names). The concept of null values does not map to any aspect of DTD-based XML, meaning that relational database export is awkward. Primitive datatypes such as integers and dates are not possible to specify.

All of these limitations, as well as a variety of other desiderata, led to the development of a newer specification in the early 2000s, called XML Schema.

11.2.2 XML Schema (XSD)

XML Schema (commonly abbreviated to its standard 3-letter file extension, XSD) is an extremely comprehensive standard designed to provide a superset of DTD's capabilities, to address the limitations mentioned in the previous section, and to itself be an XML-encoded standard.

Since an XML Schema is itself specified in XML, we will always be dealing with (at least) two namespaces: the namespace for XML schema itself (used for the built-in XSD definitions and datatypes), and the namespace being defined by the schema. Typically, we will use the default namespace for the tags defined in the schema, and will use a prefix (commonly `xs:` or `xsd:`) associated with the URI `www.w3.org/2001/XMLSchema` to refer to the XML Schema tags.

Beyond the use of XML tags, XML Schema differs significantly from DTD in two respects. First, the notion of an *element type* has been separated from the *element name*. Now we define an element type (either a `complexType` representing a structured element, or a `simpleType` representing a scalar or text node) and later associate it with one or more element names. Second, the use of EBNF notation has been completely eliminated, and instead we group sequences of content using `sequence` or `choice` elements, and specify a number of repetitions using `minOccurs` and `maxOccurs` attributes.

Example 11.5: Figure 11.4 shows an XML Schema fragment for our running example. We see first that the schema definition has a root tag `schema` within the XML Schema namespace (abbreviated here as `xsd`).

The fragment shows the definition of a complex element type for a thesis. Associated with this element type are three attributes (`mdate`, `key`, and optional `advisor`). Observe that each of the attributes has a particular type (one of the built-in XML Schema *simple types*): date, string, and string, respectively. Within the `ThesisType` is a sequence of subelements: an `author` string, a `title` string, a `year` integer, a school `string`, and a sequence of zero or more `committeemember`s of a complex type

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 ...
 <xsd:complexType name="ThesisType">
  <xsd:attribute name="mdate" type="xsd:date"/>
  <xsd:attribute name="key" type="xsd:string"/>
  <xsd:attribute name="advisor" type="xsd:string" minOccurs="0"/>
  <xsd:sequence>
   <xsd:element name="author" type="xsd:string"/>
   <xsd:element name="title" type="xsd:string"/>
   <xsd:element name="year" type="xsd:integer"/>
   <xsd:element name="school" type="xsd:string"/>
   <xsd:element name="committeemember" type="CommitteeType"
                minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:complexType name="CommitteeType">
  ...
 </xsd:complexType>
 ...
 <xsd:element name="mastersthesis" type="ThesisType"/>
 ...
</xsd:schema>
```

Figure 11.4: Example XML Schema fragment corresponding to the XML of Figure 11.1. This excerpt focuses on defining the structure of the `mastersthesis`.

called `CommitteeType`. The `ThesisType` can be used for more than one element; we finally associated it with the `mastersthesis` near the bottom of the figure.

XML Schema allows for the definition of both keys and foreign keys, which we shall discuss later in this chapter when we have introduced XPath. It also has many features that we do not discuss at all: it is possible to define simple types that restrict the built-in types (e.g., positive integers, dates from 2010-2020, strings starting with "S"), to make use of inheritance in defining types, and to create reusable structures. For more detail we suggest consulting the many resources available on the Web.

XML Schema and its associated data model are the "modern" schema format for XML, and used by most XML-related Web standards. Examples include SOAP, the Simple Object Access Protocol used to pass parameters across systems; WSDL, the Web Service Description Language; the datatype primitives of RDF Schema, the schema language for the Resource Description Framework used in the Semantic Web (see Section 12.3); and XQuery, the XML query language we describe later in this chapter.

11.3 Query Language

Given that XML represents documents and data in a standard way, it quickly became apparent that applications and developers would need standardized mechanisms for extracting and manipulating XML content.

Over the years, models emerged for parsing XML into objects (DOM) and messages (SAX), the XPath primitive, and ultimately the XQuery language. We discuss the parsing standards in Section 11.3.1, XPath in Section 11.3.2, and XQuery in Section 11.3.3. Another XML transformation language, XSLT (XML Stylesheet Language: Transformations), is used in many document and formatting scenarios, but is not well-suited to data manipulation and hence we do not discuss it in this book.

11.3.1 Precursors: DOM and SAX

Prior to the creation of XML query primitives, the XML community established a series of language-independent APIs for parsing XML: the goal was that any parser would implement these APIs in a consistent way, making it easy to port code to a variety of platforms and libraries.

The first standard, the Document Object Model or DOM, specifies a common object-oriented hierarchy for parsed HTML and XML. DOM actually emerged from the internal object models supported by HTML web browsers, but it was generalized to XML as well. DOM establishes a common interface, the DOM Node, to encompass all of the various XML node types; instances include, e.g., the Document Node, Element Node, and Text Node. A DOM parser typically builds an in-memory object tree representing a parsed document, and returns the document root node to the calling program. From here the application can traverse the entire document model: Every DOM node has methods for traversing to the node's parent and children (if applicable), testing the node type, reading the text value of the node, and so on. An interface also exists to retrieve nodes directly by their name, rather than through the document hierarchy. Later versions of DOM also support *updates* to the in-memory document model.

DOM is a fairly heavyweight representation of XML: each node in an XML document is instantiated as an object, which typically consumes significantly more space than the original source file. Moreover, early versions of DOM were not in any way *incremental*: no processing would happen until the entire document was parsed. To meet the needs of applications that only wanted to manipulate a small portion of an XML document — or to incrementally process the data — the SAX, the Simple API for XML, was created. SAX is not an object representation, but rather a standard parser API. As an XML parser reads through an input XML file, it *calls back* to user-supplied methods, notifying them when a new element is beginning, when an element is closing, when a text node is encountered, etc. The application can take the information provided in the callback, and perform whatever behaviors it deems appropriate. The application might instantiate an object for each callback, or it might simply update progress information or discard the callback information entirely.

Both SAX and DOM are designed for developers in object-oriented languages to manipulate XML; they are not declarative means for extracting content. Of course, such declarative standards have also been developed, and we discuss them next.

11.3.2 XPath: A primitive for XML querying

XPath was originally developed to be a simple XML query language, whose role is to extract subtrees from an individual XML document. Over time it has become more commonly used as a building block for other XML standards: e.g., it is used to specify keys and foreign keys in XML Schema, and it is used to specify collections of XML nodes to be assigned to variables in XQuery (described below).

XPath actually has two versions, the original XPath 1.0 and the later XPath 2.0. The original version was limited to expressions that did not directly specify a source XML document (i.e., one needed to use a tool to apply the XPath to a specific XML file). Version 2.0 was revised during the development of XQuery, in order to make it a fully integrated subset of that language: it adds a number of features, with the most notable being the ability to specify the source document for the expression, and a data model and type system matching that of XQuery. As of the writing of this book, many tools still use the original XPath 1.0 specification.

Path expressions. The main construct in XPath is the *path expression*, which represents a sequence of *steps* from a *context node*. The *context node* is by default the root of the source document to which the XPath is being applied. The result of evaluating the path expression is a *sequence of nodes* (and their subtree descendants), with duplicates removed, returned in the order they appear in the source document. (This sequence is often termed a "node set" for historic reasons.)

The context node. XPaths are specified using a Unix path-like syntax. As in Unix, the *current* context node is designated by "." If we start the XPath with a leading "/" then this starts at the document root. In XPath 2.0, we can also specify a particular source document to use as the context node: the function `doc("URL")` parses the XML document at *`URL`* and returns its document root as the context node.

From the context node, an XPath typically includes a sequence of *steps* and optional *predicates*. If we follow the usual interpretation of an XML document as a tree, than a step in an XPath encodes a *step type* describing how to traverse from the context node, and a *node restriction* specifying which nodes to return. The *step-type* is typically a delimiter like "`/`" or "`//`" and the *node-restriction* is typically a label of an element to match; we specify these more precisely in a moment.

By default, traversal is downwards in the tree, i.e., to descendants. We can start at the context node and traverse a *single level* to a child node (specified using the delimiter "`/`") or through *zero or more descendant sub-elements* (specified using "`//`"). We can restrict the matching node in a variety of ways:

- A step "···`/label`" or "···`//label`" will only return child or descendant elements, respectively, with the designated label. (Any intervening elements are unrestricted.)
- A step with a "`*`" will match any label: "···`/*`" or "···`//*`," will return the set of all child elements, or descendant elements, respectively.
- A step "···`/@label`" will return attribute nodes (including both label and value) that are children of the current element, which match the specified label; the step or "···`//@label`" will return attribute nodes of the current *or any descendant element*, if they have the specified label.
- A step "···`/@*`" or "···`//@*`" will return any attribute nodes associated with the current element, or the current and any descendant elements, respectively.
- A step "`/..`" represents a step up the tree to the *parent* of each node matched by the previous step in the XPath.
- A step with a *node-test* will restrict the type of node to a particular class: e.g., ···`/text()` returns child nodes that are text nodes; ···`/comment()` returns child nodes that are comments; ···`/processing-instruction()` returns child nodes that are processing instructions; ···`/node()` returns any type of node (not just elements, attributes, etc.).

Example 11.6: Given the data of Figure 11.1, the XPath `./dblp/article` would begin with the document root as the context node, traverse downward to the `dblp` root element and any `article` subelements, and return an XML node sequence containing the `article` elements (which would still maintain all attachments to its subtree). Thus, if we were to serialize the node sequence back to XML, the result would be:

```
  <article mdate="2002-01-03" key="tr/dec/SRC1997-018">
   <editor>Paul R. McJones</editor>
   <title>The 1995 SQL Reunion</title>
   <journal>Digital System Research Center Report</journal>
   <volume>SRC1997-018</volume>
   <year>1997</year>
   <ee>db/labs/dec/SRC1997-018.html</ee>
   <ee>http://www.mcjones.org/System_R/SQL_Reunion_95/</ee>
 </article>
  ...
```

(Note that if there were more than one `article`, the result of the XPath would be a *forest* of XML trees, rather than a single tree. Hence the output of an XPath is often not a legal XML document since it does not have a single root element.)

Example 11.7: Given the example data, the XPath `//year` would begin at the document root (due to the leading "`/`"), traverse any number of subelements downward, and return:

```
  <year>1992</year>
  <year>1997</year>
  ...
```

where the first `year` comes from the `mastersthesis` and the second from the `article`.

Example 11.8: Given the example data, the XPath `/dblp/*/editor` would begin at the document root, traverse downwards to match first the `mastersthesis` and look for an `editor`. No such match is found, so the next traversal will occur in the `article`. From the `article`, an `editor` can be found, and the result would be:

```
  <editor>Paul R. McJones</editor>
  ...
```

where, of course, further results might be returned if there were matches among the elided content in the document.

More complex steps: axes. While they are by far the most heavily used step specifiers, "`/`" and "`//`" are actually considered to be a special *abbreviated syntax* for a more general XML traversal model called *axes*. Axes allow an XPath author to not only specify a traversal to a descendant node, but also an ancestor, a predecessor, a successor, etc. The syntax for axes is somewhat cumbersome: instead of using "`/`" or "`//`" followed by a node restriction, we instead use "`/`" followed by an *axis specifier* followed by "`::`", then the node restriction. The axes include:

- `child`: traverses to a child node, identically to a plain "`/`".
- `descendant`: finds a descendant of the current node, identically to a plain "`//`".
- `descendant-or-self`: returns the current node or any descendant.
- `parent`: returns the parent of the current node, identically to "`..`".
- `ancestor`: finds any ancestor of the current node.
- `ancestor-or-self`: returns the current node or an ancestor.
- `preceding-sibling`: returns any sibling node that appeared earlier in the document order.
- `following-sibling`: returns any sibling node that appeared later in the document order.
- `preceding`: returns any node, at any level, appearing earlier in the document.
- `following`: returns any node, at any level, appearing later in the document.
- `attribute`: matches attributes, as with the "" prefix.
- `namespace`: matches namespace nodes.

Example 11.9: The XPath of Example 11.6 could be written as:

```
./child::dblp/child::article
```

and Example 11.7 as:

```
/descendant::year
```

An XPath to return *every* XML node in the document is:

```
/descendant-or-self::node()
```

Predicates. Often we want to further restrict the set of nodes to be returned, e.g., by specifying certain data values we seek. This is where XPath *predicates* come in. A predicate is a Boolean test that can be attached to a specific step in an XPath; the Boolean test is applied to each result that would ordinarily be returned by the

XPath. The Boolean test may be as simple as the existence of a path (this enables us to express queries that test for specific *tree patterns* as opposed simply to paths), it could be a test for a particular value of a text or attribute node, etc.

Predicates appear in square brackets, `[` and `]`. The expression within the brackets has its context node set to the node matched by the previous step in the XPath.

Example 11.10: If we take the XPath expression `/dblp/*[./year/text()="1992"]` and evaluate it against Figure 11.1, then first `mastersthesis` is matched (becoming the context node becomes the `mastersthesis`), and then the predicate expression is evaluated. Since it returns true, the `mastersthesis` node is returned. When the `article` node is matched (becoming the context node for the predicate), the predicate is again evaluated, but this time it fails to satisfy the conditions.

Predicates referring to position. Predicates can also be used to select only specific values from a node set, based on the index position of the nodes. If we use an integer value i as the position, e.g., `p[5]`, this selects the ith node in the node sequence. We can also explicitly request the index of a node with the `position()` function, and the index of the *last* node in the node sequence with the `last()` function.

Example 11.11: To select the last article in the XML file of Figure 11.1, we could use the XPath:

```
//article[position()=last()]
```

11.3.3 XQuery: Query capabilities for XML

The XPath language is not expressive enough to capture the kinds of data transformations that are employed in a typical database query or schema mapping: for instance, there is no support for cross-document joins, changing of labels, restructuring, etc. To address these needs, the XQuery language was developed. Intuitively, XQuery is the “SQL of XML”: the single standard language that is intended to be implemented by XML databases, document transformation engines, and so on.

XQuery currently consists of a core language, plus a series of extensions: the core language provides exact-answers semantics and supports querying and transformation. There exist extensions for full-text querying with ranked answers (XQuery Full Text) and for XQuery updates (the XQuery Update Facility). In this book we focus on the “core” XQuery.

```
for $docRoot in document("http://my.org/dblp.xml"),
  $rootElement in $docRoot/dblp,
    $rootChild in $rootElement/article
let $textContent := $rootChild//text()
where $rootChild/author/text() = "Bob"
return <BobResult>
         { $rootChild/editor }
         { $rootChild/title }
         { for $txt in $textContent
           return <text> { $txt } </text>
         }
       </BobResult>
```

Figure 11.5: Simple XQuery example

Basic XQuery Structure: “FLWOR” Expressions

SQL is well-known for its basic “`SELECT`...`FROM`...`WHERE`” pattern for describing query blocks (with an optional “`ORDER BY`” and “`GROUP BY`...`HAVING`”. XQuery similarly has a basic form for a query block, called a FLWOR (“flower”) expression. FLWOR is an acronym for “`for`...`let`...`where`...`order by`...`return`.” (Note that XQuery keywords must be in lowercase, in contrast to SQL, which is case-insensitive.) (In reality, the different XQuery clauses can be interleaved in a variety of orders, and several clauses are optional, but the FLWOR ordering is by far the prevalent one.)

Intuitively, the XQuery `for`/`let` clauses correspond to the `FROM` clause in SQL; the XQuery `where` corresponds to its namesake in SQL; and the `return` clause corresponds to the `SELECT` clause in SQL. However, there are a number of key differences in the semantics of the clauses: the relational model consists of tables and tuples, with completely different operators that apply to each. In XML, a document or any subtree is represented as a node with a set of subtrees. Hence XQuery distinguishes between nodes, scalar types, and collections, and operators are defined in a relatively clean way over these. XQuery also allows for the outputs of multiple (possibly correlated) queries to be nested, in order to form nested output.

`for`: iteration and binding over collections. The most common way of specifying an input operand for a query is to *bind* a variable to each node matching a pattern. The `for` clause allows us to define a variable that ranges over a node sequence returned by an XPath. The syntax is `for` *$var* `in` *XPath-expression*. To bind multiple variables, we can nest `for` clauses. For each possible valuation of the variables, the XQuery engine will evaluate the `where` condition(s) and optionally `return` content.

Example 11.12: Suppose we have a document called `dblp.xml` located on the `my.org` server. If we use the following sequence of nested `for` clauses:

```
for $docRoot in document("http://my.org/dblp.xml")
  for $rootElement in $docRoot/dblp
    for $rootChild in $rootElement/article
```

then the variable `$docRoot` will take on a single value, that of the document root node of the XML source file. The variable `$rootElement` will iterate over the child elements of the document root — namely, the (single) root element of the XML file. Next `$rootChild` will iterate over the child element nodes of the root element.

We can abbreviate nested `for` clauses using an alternate form, in which a comma is used to separate one expression from the next, and the second `for` clause is omitted.

Example 11.13: The above example can be rewritten as in Lines 1–3 of Figure 11.5.

`let`**: assignment of collections to variables.** Unlike SQL, XQuery has a notion of collection-valued variables, where the collection may be a sequence of nodes or of scalar values. The `let` clause allows us to assign the results of any collection-valued expression (e.g., an XPath) to a variable. The syntax is `let` *$var* `:=` *collection-expression*.

Example 11.14: Continuing our example from above, the first 4 lines of Figure 11.5 will assign a different value to `$textContent` for each value of `$rootChild`: namely, the set of *all* text nodes that appear in the element's subtree.

`where`**: evaluation of conditions against bindings.** For each possible valuation of the variables in the `for` and `let` clauses, the XQuery engine will attempt to evaluate the predicates in the `where` clause. If these are satisfied, then the `return` clause will be invoked to produce output. We can think of XQuery's execution model as being one in which "tuples of bindings" (one value for each variable from each source) are joined, selected, and projected; the `where` clause performs the selection and join operations.

Example 11.15: We can restrict our example from above to only consider `$rootChild` elements that have `editor` subelements with value "Bob," as seen in Lines 1–5 of Figure 11.5.

`return`: output of XML trees. The `return` clause gets invoked each time the `where` conditions are satisfied, and in each case it returns a fragment of output that is typically an XML tree. The content output by the `return` clause may be literal XML, the results of evaluating some expression (even XPath expression) based on bound variables, or the results of evaluating a nested XQuery.

If a `return` clause begins with an angle-bracket or a literal value, the XQuery engine assumes that it is to output whatever it reads as literal text. To instead force it to interpret the content as an expression, we use the *escape* characters { and } around the expression.

Example 11.16: The full query in Figure 11.5 completes our example, where we return an XML subtree each time "Bob" is matched. Within the subtree, we output the editor and title within the `$rootChild` subtree. Then we use a nested XQuery to iterate over all of the text nodes in `$textContent`, outputting each in an element labeled `text`.

Aggregation and Uniqueness

SQL allows the query author to specify duplicate removal using a special `DISTINCT` keyword in the `SELECT` clause or an aggregate function (e.g., `COUNT DISTINCT`). In XQuery, the notion of computing distinct values or items is handled as a collection-valued function, which essentially converts an ordered sequence into an ordered set. There are two functions in most XQuery implementations, `fn:distinct-values` and `fn:distinct-nodes`, which take node sequences, and remove items that match on value equality or node identity, respectively. These functions can be applied to the results of an XPath expression, to the output of an XQuery, etc.; `fn:distinct-values` can even be applied to a collection of scalar values.

As with computing unique values, aggregation in XQuery is accomplished in a way that is entirely different from SQL. In XQuery, there is no `GROUP BY` construct, and hence aggregate functions are not applied to attributes in grouped tables. Instead, an aggregate function simply takes a collection as a parameter, and returns a scalar result representing the result of the function: `fn:average`, `fn:max`, `fn:min`, `fn:count`, `fn:sum`.

Example 11.17: Suppose we want to count the number of theses or articles written by each author (assuming author names are canonicalized), for a document of the form shown in Figure 11.1. Rather than using a `GROUP BY` construct, we express this by computing the set of all authors, then finding the papers for each author.

```
  let $doc := doc("dblp.xml")
  let $authors := fn:distinct-values($doc//author/text())
  for $auth in $authors
  return <author>
           { $auth }
           { let $papersByAuth := $doc/*[author/text() = $auth]
             return <papers> <count> { fn:count(papersByAuth) } </count>
                             <titles> { $papersByAuth/title } </titles>
                    </papers> }
         </author>
```

Here, the inner query computes the set of all papers by the author, then returns the count as well as the complete list of titles.

Functions

XQuery does not expressely have the notion of a view, as in SQL. Rather, XQuery supports arbitrary *functions* that can return scalar values, nodes (as the roots of XML trees), or collections (of scalars or XML nodes/trees). Since a function can return an XML forest or tree, one can think of it as a view that might optionally be parameterized.

Note that XQuery allows for recursion as well as if/else and iteration within functions, hence the language is Turing-complete, unlike SQL. A function in XQuery is specified using the keywords `declare function`, followed by the function name, its parameters and their types, the keyword `as`, and the function return type. Note that the types for the parameters and return type are the XML Schema types. If there is no accompanying schema, a generic element can be specified using `element()`, and an untyped element with a particular name n can be specified using `element(`*n*`)`.

Example 11.18: The following function returns all authors in the `dblp.xml` document.

```
declare function paperAuthors() as element(author)* {
  return doc("dblp.xml")//author
}
```

Note that the return type here is specified to be zero or more elements (hence the asterisk) called `author`, for which we may (or might not) have an accompanying XML Schema type.

Example 11.19: The following function returns the number of coauthors who wrote an article with the author whose name is specified as an input parameter `$n`. It does this by finding the sets of authors for each article authored by `$n`, then doing duplicate removal, and finally counting the number of entries and then subtracting 1 to avoid counting `$n` himself or herself.

```
declare function paperAuthors($n as xs:string) as xs:integer {
  let $authors := doc("dblp.xml")/article[author/text()=$n]/author
  return fn:count(fn:distinct-values($authors)) - 1
}
```

Note that the return type here is specified to be zero or more elements (hence the asterisk) called `author`, for which we may (or might not) have an accompanying XML Schema type.

11.4 Query Processing for XML

Now that we are familiar with the form of XML and its query languages, we consider the problem of *processing* XPaths and XQueries over XML data. Inherently, XML is a tree-structured, *non-first-normal-form* data format. The bulk of work on large-scale XML query processing, especially in terms of matching XPaths over the input, has focused on settings in which the data is *stored on disk*, e.g., in a relational DBMS. Here the challenges are how to break up the XML trees and index them (if using a "native" XML storage system) or how to "shred" the XML into relational tuples.

For the latter case, the goal is to intuitively to take each XML node and create a corresponding tuple in a relational table. Each such tuple is encoded in a way that enables it to be joined with its parent (and possibly ancestors) and children (and possibly descendants). A common approach is to annotate each tuple T with an *interval code* describing the position of the first and last item in the tree rooted at the node corresponding to T: then we can compare the intervals of tuples T_1 and T_2 for containment to see if one tree is a subtree of the other.

In data integration, the time-consuming portion of evaluation is often *reading an XML source file* and extracting subtrees that match particular XPath expressions. Rather than *store* the XML before extracting relevant trees, our goal is to, in a *single pass*, read, parse, and extract XPath matches from each source document. This is often referred to as "streaming XPath evaluation."

Streaming XPath evaluation relies on two critical observations:

1. SAX-style XML parsing very efficiently matches against XML as it is read from an input stream (e.g., across a network), and uses a callback interface to trigger an *event handler* whenever specific content types, start tags, or end tags are encountered. This enables parsing as data is being pipelined into the query processor's system.

2. A large subset of XPath expressions can be mapped to regular expressions, where the alphabet consists of the set of possible edge labels plus a wildcard character. In general, we end up with a set of *nested* regular expressions for each XPath (since, e.g., predicates must be separately matched).

These observations have led to a variety of XML *streaming XPath* matchers, based on event handlers and modified finite state machines. A streaming XPath matcher typically returns sets of nodes, or, if it matches multiple XPaths simultaneously as for XQuery, *tuples of bindings to nodes* (i.e., tuples in which there exists one attribute per variable, whose value is a reference to a node, or a set of references to nodes). These tuples of bindings are typically processed by an extended version of a relational-style query processor, which has the standard set of operators (e.g., select, project, join) as well as a few new ones (in particular, evaluating an XPath against a tree-valued attribute in a tuple, adding XML tags, collecting sequences or sets of tuples into an XML tree). Finally, XML content is assembled at the end and output in the form of a tree, using a combination of tagging and grouping operators.

Example 11.20: We see an example of a typical XML query plan in Figure 11.6, corresponding to the query of Figure 11.5. Starting from the bottom, we see a streaming XPath operator that fetches `dblp.xml` and begins parsing it, looking for trees corresponding to the root element, its child, and any descendent text content. This operator returns tuples with values for `rootElement`, `rootChild`, and `textContent`, of which the first two are trees and the last is a forest. These tuples are then left-outerjoined with a nested expression: the nested expression matches an XPath against the forest in `textContent`, and returns a sequence of tuples with values for `txt`. These tuples are tagged within `text` elements, then grouped into a single forest. The results of the left outerjoin are then fed into further XPath matching, which extracts the `editor` and `title` from the `rootChild`. After further projection, the element `BobsResult` is created for each the tuple, and its children are set to be the `editor`, `title`, and results of the nested expression. The result is a sequence of `BobsResult` tuples that can be written to disk, a data stream, etc.

Let us now look at the specifics of how the operators are implemented.

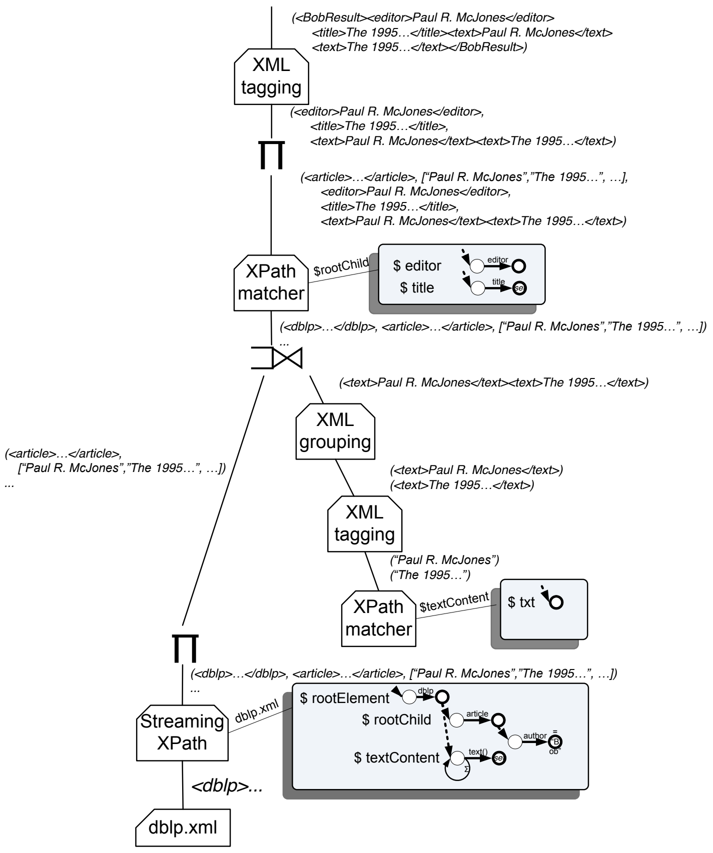
(<BobResult><editor>Paul R. McJones</editor> <title>The 1995…</title><text>Paul R. McJones</text> <text>The 1995…</text></BobResult>)
XML tagging
(<editor>Paul R. McJones</editor>, <title>The 1995…</title>, <text>Paul R. McJones</text><text>The 1995…</text>)
(<article>…</article>, ["Paul R. McJones","The 1995…", …], <editor>Paul R. McJones</editor>, <title>The 1995…</title>, <text>Paul R. McJones</text><text>The 1995…</text>)
XPath matcher
$rootChild
$ editor
$ title
editor
title
(<dblp>…</dblp>, <article>…</article>, ["Paul R. McJones","The 1995…", …])
(<text>Paul R. McJones</text><text>The 1995…</text>)
XML grouping
(<article>…</article>, ["Paul R. McJones","The 1995…", …])
(<text>Paul R. McJones</text>) (<text>The 1995…</text>)
XML tagging
("Paul R. McJones") ("The 1995…")
XPath matcher
$textContent
$ txt
(<dblp>…</dblp>, <article>…</article>, ["Paul R. McJones","The 1995…", …])
Streaming XPath
dblp.xml
$ rootElement
$ rootChild
$ textContent
dblp
article
author
text()
<dblp>...
dblp.xml

Figure 11.6: Sketch of a query plan for the query of Figure 11.5. Most of the engine remains a tuple processor, where tuples are no longer in first-normal-form, but rather *binding tuples* containing (references to) trees.

11.4.1 XML path matching

The XML Path matching operator appears in two guises, one based on streaming evaluation over incoming data, and the other based on traversal of the tree representation within an attribute of a binding tuple. Both can be implemented using the same mechanisms, as handlers to an event-driven XML tree parser.

A variety of design points have been considered for XML path matchers.

Type of automaton. In general, within a single XPath segment (series of steps separated by `/` or `//`), the presence of wildcards or the `//` step allows for multiple matches at each level. This introduces *nondeterminism* into the picture. Several different schemes have been considered: building a single nondeterministic finite state

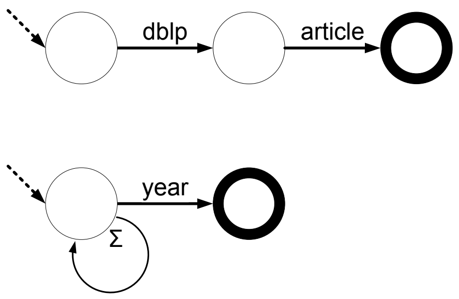

Figure 11.7: Nondeterministic finite automata for the XPaths `/dblp/article` and `//year`. Note that the Σ on the edge label in the automata represents the alphabet of all possible labels, i.e., a wildcard.

machine (NFA) for the XPath, and keeping track of multiple possible matches simultaneously; converting the expression to a deterministic finite state machine (DFA); or creating an NFA that is "lazily" expanded to a DFA as needed (where portions of the expansion are cached).

Example 11.21: We can see the NFAs for two of the example XPaths of Examples 11.6 and 11.7 in Figure 11.7. Depending on the specific style of XPath match, this NFA might be directly represented in memory and used to match against events from input; or it might be converted to a DFA (either eagerly or lazily) and used to match.

Different schemes for evaluating sub-paths. Many XPaths have "subsidiary" segments — for instance, nested paths within predicates. One can create a single automaton with different final states indicating which XPaths have been matched; or one can create a series of nested automata, where matching one automaton activates the others.

Predicate push-down and short-circuiting. In any case, predicates may include operations beyond simple path-existence tests, which can often be "pushed" into the path-matching stage. Here the XPath matcher must invoke subsidiary logic to, e.g., test for value equality of a text node. Any non-match must be discarded.

XPath versus XQuery. If the query processor is computing XPath queries, then the XPath matcher's goal is to return a node set for each query. Sometimes, however, the goal is to provide a set of *input binding tuples* for the `FOR` (or, in some cases, `LET`) clauses of an XQuery. Here, we want the Cartesian product of the possible node matches to a set of XPaths. The *x-scan* operator combines XPath matching with an

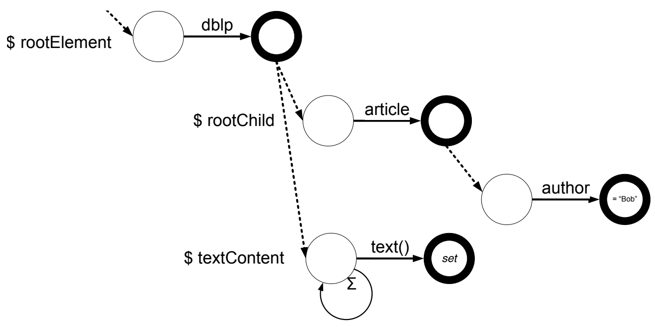

Figure 11.8: Hierarchy of nondeterministic finite automata for matching the set of XPaths in the XQuery of Figure 11.5. Note that upon reaching the final state of one machine, we may activate other (nested) XPaths, and we may need to test the match against a value or collect it within a set.

extension of the pipelined hash join, such that it can incrementally output tuples of bindings to be evaluated.

Example 11.22: In evaluating an XQuery within a streaming operator, the first stage simultaneously matches a hierarchy of XPaths corresponding to `for` and `let` expressions. See Figure 11.8, which corresponds to the leftmost leaf operator from Figure 11.6. For this streaming XPath matcher, reaching the final state in one NFA may trigger certain behaviors. One such behavior is that a match to the first NFA may activate other, nested NFAs, corresponding to nested XPaths. Moreover, upon matching a particular pattern we may need to test the match against a particular value (a predicate push-down, indicated in the test for `author = ‘‘Bob’’`), and we may add matches to a set, rather than adding one binding per match (as is needed for the `$rootElement//text() let` clause, versus the other `for` clauses).

The output of the streaming operator for this collection of NFAs will be a sequence of binding tuples for the variables `rootElement`, `rootChild`, and (collection-typed) `textContent`. Example 11.20 describes in more detail how the output from the streaming XPath evaluator is propagated through the other operators in the query plan.

11.4.2 XML output

The process of creating XML within a tuple is typically broken into two classes of operations.

Tagging takes an attribute (which is already tree-valued) and "wraps" it within an XML element or attribute. In certain cases well-structured properties may need to be tested (e.g., certain kinds of content are illegal within an attribute).

Grouping takes a set of tuples (with some common key attributes) and combines their results, typically within a single collection-valued attribute. It is very analogous to SQL `GROUP BY`, except that instead of computing a scalar aggregate value over an attribute, it instead collects all of the different attribute values into a single XML forest.

Of course, another key component of XQuery processing tends to be nesting within the `return` clause. This is typically accomplished using the **left outer join** operator, as in Example 11.20.

The focus of this chapter has been on implementing the SQL-like "core" of XQuery. Support for recursive functions and some of XQuery's more advanced features typically requires additional operators: for instance, some form of function invocation plus an `if/else` construct that helps determine when to terminate, the ability to query nodes for their XML Schema types, and support for calling library functions.

11.4.3 XML query optimization

In many ways, XQuery optimization is little different from conventional relational optimization: there is a process of enumeration and cost and cardinality estimation. The challenges tend to lie in the unique properties of XML. XQuery is much more expressive than SQL, and is in fact Turing-complete. In general XQuery optimizers are most effective on subsets of the overall language. Even for subsets, there are often many more potential rewrites than in the normal SQL case.

Moreover, the cardinalities (and hence costs) of XPath matching tend to be more difficult to determine: it becomes essential to estimate "fan-outs" of different portions of an XML document. New metadata structures have been developed to try to help estimate branching factors. Such techniques are beyond the scope of this textbook.

11.5 Schema Mapping for XML

Not surprisingly, XML introduces new complexities in terms of *schema mappings*. The two main ones are as follows.

- XML is *semi-structured*, meaning that different XML elements (even of the same type) may have variations in structure. Moreover, XML queries contain multiple path expressions with optional wildcards and recursion. Hence the conditions for query containment are somewhat different.

- XML contains nesting, and the specific nesting relationships may be *different* between a source instance and a target instance. Hence it becomes necessary to specify nesting relationships, and optionally merge operations on nodes, within XML mappings.

As with schema mappings for the relational database model, we find that schema mappings from XML are expressed in a restricted language. This language roughly corresponds to a nested version of GLAV constraints, though it is typically formulated in a logical constraint language called *nested tuple-generating dependencies* (nested tgds). Traditional tuple-generating dependencies, expressed over relations, are essentially equivalent to GLAV mappings and are used in *data exchange* (Section 17.3). Nested tgds are an extension to the tgd formalism that (naturally enough) allow for constraints among hierarchical data structures.

11.5.1 Nested tgds

We begin by introducing the tuple-generating dependency itself.

Definition 11.1: *(Tuple-generating dependency). A* tuple-generating dependency *(tgd) is an assertion about the relationship between a* source data instance *(in our case, a source database) and a* target data instance *(in our case, a central database or mediated schema). A tgd takes the form:*

$$\forall \bar{x}, \bar{y}(\phi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z} \psi(\bar{x}, \bar{z}))$$

where ϕ and ψ are atomic formulae over the source and target instances, respectively. If the condition ϕ holds over the left-hand-side (lhs) of the tgd, then condition ψ must hold over the right-hand-side (rhs).

Note that this formulation can be restated in GLAV notation as the constraint:

$$\mathcal{Q}^S(\bar{x}, \bar{y}) \subseteq \mathcal{Q}^T(\bar{y}, \bar{z})$$

where

$$\mathcal{Q}^S(\bar{x}, \bar{y}) \text{ :- } \phi(\bar{x}, \bar{y})$$

$$\mathcal{Q}^T(\bar{y}, \bar{t}) \text{ :- } \psi(\bar{y}, \bar{z})$$

As we have described it, the tgd is an assertion about relational instances. An interesting application of such constraints is discussed in Section 17.3. For this chapter, however, our interest is not in relational instances, but instances with hierarchical nesting, i.e., XML.

Definition 11.2: *(Nested tuple-generating dependency). A* nested tuple-generating dependency *(nested tgd) is an assertion about the relationship between a source data instance and a target data instance, of the form:*

$$\forall \bar{x}, \bar{y}, \bar{S}(\phi(\bar{x}, \bar{y}) \wedge \bar{\Phi}(\bar{S}) \rightarrow \exists \bar{z}, \bar{T}(\psi(\bar{x}, \bar{z}) \wedge \bar{\Psi}(\bar{T})))$$

where $\bar{x}, \bar{y}, \bar{z}$ are variables representing attributes, ϕ and ψ are atomic formulae over source and target instances, respectively; $\bar{S}$ and $\bar{T}$ are set-valued variables representing nested relations; and $\bar{\Phi}$ and $\bar{\Psi}$ are sets of atomic formulae, one for each of the respective variables of $\bar{S}$ and $\bar{T}$. Each set-valued variable in $\bar{T}$ must also have a grouping key*, to which a variable-specific* Skolem function *is applied to uniquely specify the desired output set — such that multiple matches to the rhs of the tgd must use the same set.*

We illustrate a nested tgd as a schema mapping with an example.

Example 11.23: Suppose we want to map between two sites., where the target contains `books` with nested `authors`, whereas the source contains `authors` with nested `publications`. We illustrate partial schemas for these sources below, using a format in which indentation illustrates nesting and a * suffix indicates "0 or more occurrences of...", as in a BNF grammar.

Target:	**Source:**
`pubs`	`authors`
`  book*`	`  author*`
`    title`	`    full-name`
`    author*`	`    publication*`
`      name`	`      title`
`    publisher*`	`      pub-type`
`      name`	

where we know that the input `pub-type` must be `book`, and the publisher has a name that is not present in the input document. We can capture this schema mapping using the following nested tgd as follows, where we omit universally quantifiers (implicit for all variables that appear on the lhs), where boldface indicates a set-typed (relation) variable, and where a subscripted with an atom name specifies the grouping key:

$$\textbf{authors}(\textbf{author}) \wedge \textbf{author}(f, \textbf{publication}) \wedge \textbf{publication}(t, \texttt{book}) \rightarrow$$
$$\exists p(\textbf{pubs}(\textbf{book}) \wedge \textbf{book}_t(t, \textbf{author'}, \textbf{publisher}) \wedge \textbf{author'}_{t,f}(f) \wedge \textbf{publisher}_t(p))$$

Observe that in this formulation, we treat an XML structure as a series of nested relations, reflecting the hierarchical structure. However, note that multiple combinations of variables on the input may satisfy the lhs of the assertion, for which we must find corresponding bindings on the output to satisfy the rhs. The use of the grouping keys specifies when the *same* node in the target must be used to satisfy the rhs. In the output, we create a single `pubs` root element, then nest within it a `book` element for each unique title; within the book we add an `author` (created for each title-author combination), and a single "unknown" `publisher` entry (created separately for each title).

The example above hints at two constraints on the parameters for the grouping keys. First, if a parent element (e.g., `book`) has a particular variable as a grouping key, then all descendant elements must also inherit this variable as a grouping key: this ensures a tree structure, as different parent elements will not share the same descendant. Second, as in `publisher`, existential variables are not included in the grouping keys.

A common use of nested tgds — like traditional tgds — is as constraints that are used to *infer* tuples in a target instance. In essence, we populate the target instance with any tuples that are needed to satisfy the constraint, using a procedure called the *chase*. This is called *data exchange* and is described in Section 17.3. However, we can also use nested tgds in query reformulation, as we describe next.

11.5.2 Query reformulation

In query reformulation, we are given a query over the target schema, and want to compose this query with the mappings in order to create a single query (in this case, XQuery) that directly returns answers to the user.

We sketch here a standard approach to reformulation, introduced in the Piazza system. For simplicity, let us assume that the query over the target schema (1) only contains a single level of nesting, and (2) only contains *child* axes in its XPaths. Intuitively, the rewriting algorithm performs the following tasks. Given a query Q, it begins by comparing the tree patterns of the mapping definition with the tree pattern of Q: the goal is to find a corresponding node in the mapping definition's tree pattern for every node in the Q's tree pattern. Then the algorithm must restructure Q's tree pattern along the same lines as the mapping restructures its input tree patterns (since Q must be rewritten to match against the *target* of the mapping rather than its source). Finally, the algorithm must ensure that the predicates of Q can be satisfied using the values output by the mapping. The two main steps performed by the algorithm are as follows.

Step 1: pattern matching. This step considers the tree patterns in the query, and finds corresponding patterns in the target schema. Intuitively, given a tree pattern, t in Q, its goal is to find a tree pattern t' on the target schema such that the mapping guarantees that an instance of that pattern could only be created by following t in the source. The algorithm first matches the tree patterns in the query to the expressions in the mapping and records the corresponding nodes. The algorithm then creates the tree pattern over the target schema as follows: starting with the recorded nodes in the mapping, it recursively marks all of their ancestor nodes in the output template. It then builds the new tree pattern over the target schema by traversing the mapping for all marked nodes.

Note that t' may enforce additional conditions to t, and that there may be several patterns in the target that match a pattern in the query, ultimately yielding several possible queries over the target that provide answers to Q. If no match is found, then the resulting rewriting will be empty (i.e., the target data does not enable answering the query on the source).

Step 2: Handling returned variables and predicates. In this step the algorithm ensures that all the variables required in the query can be returned, and that all the predicates in the query have been applied.

Query predicates can be handled in one of three ways. First, a query predicate (or one that subsumes it) might already be applied by the relevant portion of the mapping (or might be a known property of the data being mapped). In this case, the algorithm can consider the predicate to be satisfied. A second case is when the mapping does not impose the predicate, but returns all nodes necessary for testing the predicate. Here, the algorithm simply inserts the predicate into the rewritten query. The third possibility is that the predicate is not applied by the portion of the mapping used in the query rewriting, nor can the predicate be evaluated over the mapping's output — but a different portion of the mapping may impose the predicate. If this occurs, the algorithm can *add a new path* into the rewritten tree pattern, traversing into the sub-block. Now the rewritten query will only return a value if the sub-block (and hence the predicate) is satisfied.

11.6 Bibliographic Notes

XML, XPath, and XQuery are discussed in great detail in the official W3C recommendations, at www.w3.org. For a more readable (though slightly dated) introduction to XQuery, consult [70]. XML's relationship to semi-structured data is discussed in significant detail in some of the early works on adapting semi-structured langauges and databases to XML, such as those from the Lore project [203] and the XML-QL language derived from StruQL [140]. Such works tended to abstract away many of the details of XML's data model (e.g., ordering) but laid the foundation for studying XML as a data model rather than a file format.

The properties of XPath, and query containment for XPath have been extensively studied in the literature. A good survey appears in [403], with several notable papers on the topic being [336, 355], and with [404] showing how to validate that streaming XML conforms to a schema. Streaming XPath evaluation has also been the subject of a great deal of study, with early work being that of XFilter [21] (which supported Boolean XPath queries) and x-scan [254] (which supported evaluation of the `for` clause in an XQuery). Other notable algorithms include [215], which espoused the use of *lazy DFAs*, and a variety of papers making use of multiple concurrent streams in the form of "twig joins" [71, 102] or explore other types of automata beyond DFAs [373].

Of course, streaming XPath/XQuery evaluation is not appropriate for all settings. Commercial IBM DB2, which contains both relational and native XQuery engines, uses a very similar construct to x-scan called TurboXPath [262], to process hierarchical "native XML" data pages written to disk. Earlier efforts to incorporate XML into commercial database systems tended to use the "shredding" approach (normalizing the XML tree into multiple relations) espoused by [418]. A number of fully native XQuery engines have been studied in the research community, including MonetDB/XQuery [453] (based on the MonetDB column-store architecture), Natix [266], and TIMBER [20]. In a distributed setting, work has also been done on including XML with embedded function calls: this work is called Active XML [7, 8].

Schema mappings for XML have been proposed in a variety of settings. The one presented here, nested tgds, is based on the formalism used by IBM's Clio project [187], but is essentially the same as that used internally by the Piazza PDMS [226]. Both projects have a "friendlier" language syntax over the nested tgds; Clio uses an XQuery-like syntax with `for/where` syntax, and Piazza uses XML templates annotated with modified XQuery and XPath clauses. Clio does not do query reformulation, but rather data exchange (see Section 17.3); Piazza does XML-based query reformulation, and the discussion in this chapter is based on the Piazza work. Other reformulation work for XML includes the chase-based reasoning used in the MARS (Mixed and Redundant Storage) project [141].