SGG 4653
Advance Database System

Semi-structured Data and XML

Semi-structured Data

- Data that may be irregular or incomplete and have a structure that may change rapidly or unpredictably.
- Semi-structured data is data that has some structure, but the structure may not be rigid, regular, or complete.
- Generally, data does not conform to fixed schema (sometimes use terms *schema-less* or *self-describing*).

Semi-structured Data

- Information normally associated with schema is contained within data itself.
- Some forms of Semi-structured data have no separate schema, in others it exists but only places loose constraints on data.
- Unfortunately, relational, object-oriented, and object-relational DBMS do not handle data of this nature particularly well.

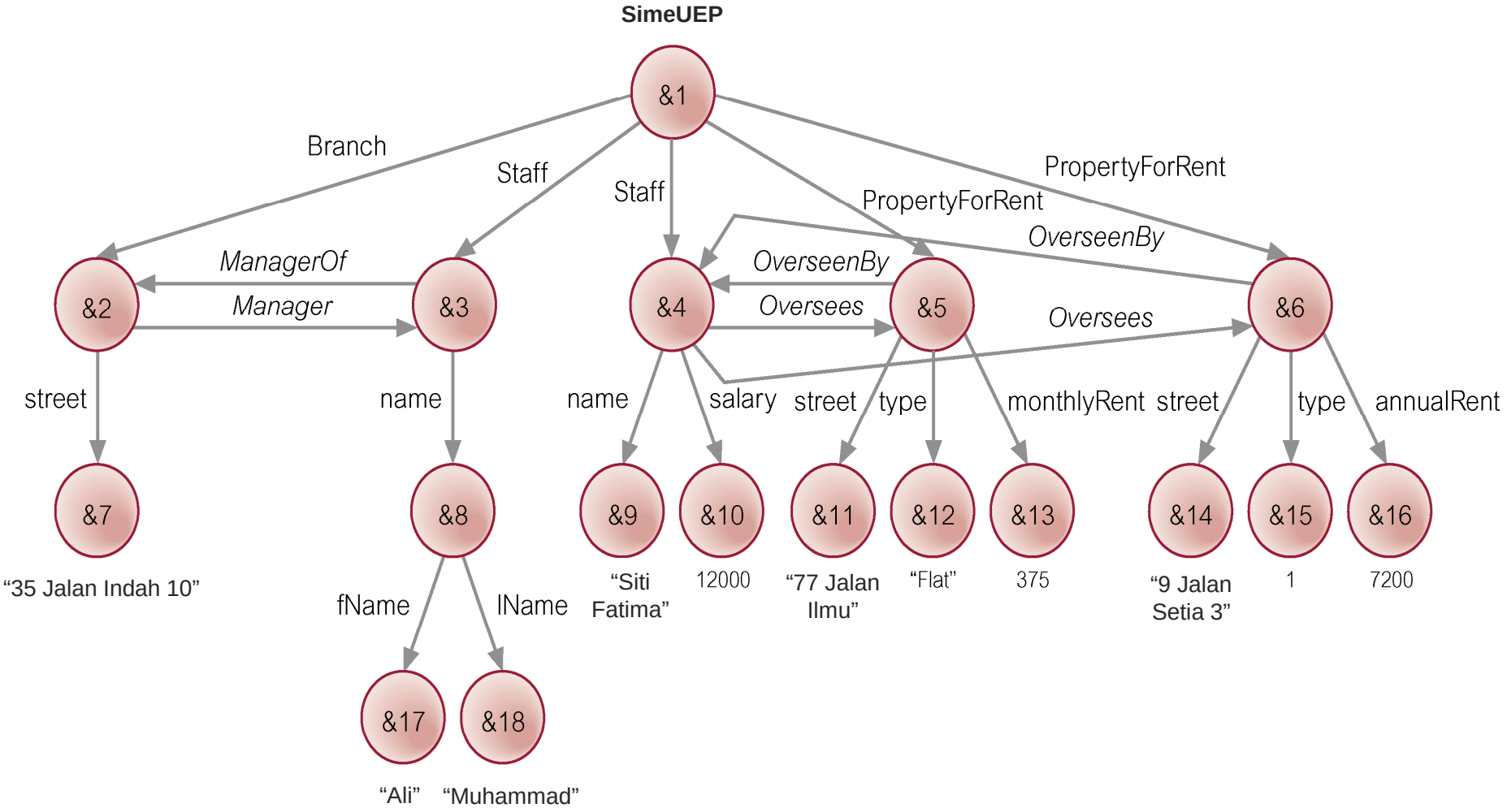
Semi-structured Data

- Has gained importance recently for various reasons:
  - may be desirable to treat Web sources like a database, but cannot constrain these sources with a schema;
  - may be desirable to have a flexible format for data exchange between disparate databases;
  - emergence of XML as standard for data representation and exchange on the Web, and similarity between XML documents and Semi-structured data
    - Based on query languages that traverse a tree-labeled representation

Example

- Note, data is not regular:
  - for Ali Muhammad, hold first and last names, but for Siti Fatima store single name and also store a salary;
  - for property at 2 Manor Rd, store a monthly rent whereas for property at 18 Dale Rd, store an annual rent;
  - for property at 77 Jalan Ilmu, store property type (flat) as a string, whereas for property at 9 Jalan Setia 3, store type (house) as an integer value.

Graphical representation of example



Object Exchange Model (OEM)

- Sample model for semi-structured data.
- Data in OEM is schema-less and self-describing, and can be thought of as labeled directed graph where nodes are *objects*, consisting of:
  - unique object identifier (for example, &7),
  - descriptive textual label (street),
  - type (string),
  - a value (“35 Jalan Indah 10”).
- Objects are decomposed into atomic and complex:
  - *atomic object* contains value for base type (e.g., integer or string) and in diagram has no outgoing edges.
  - All other objects are *complex objects* whose types are a set of object identifiers.

Object Exchange Model (OEM)

- A *label* indicates what the object represents and is used to identify the object and to convey the meaning of the object, and so should be as informative as possible.
- Labels can change dynamically.
- A *name* is a special label that serves as an alias for a single object and acts as an entry point into the database (for example, SimeUEP is a name that denotes object &1).

Object Exchange Model (OEM)

- An OEM object can be considered as a quadruple (label, oid, type, value).
- For example:

{Staff, &4, set, {&9, &10}}

{name, &9, string, “Siti Fatima”}

{salary, &10, decimal, 12000}

Approaches to develop DBMS for semi-structured data

- Built on top of RDBMS
- Built on top of OODBMS
- Example DBMS for handling Semi-structured data:
  - Lore / Lorel

Lore and Lorel

- Lore (Lightweight Object REpository), is a multi-user DBMS, supporting crash recovery, materialized views, bulk loading of files in some standard format (XML is supported), and a declarative update language.
- Has an external data manager that enables data from external sources to be fetched dynamically and combined with local data.

Lorel

- Lorel (the Lore language) is an extension to Object Query Language (OQL). Lorel was intended to handle:
  - queries that return meaningful results even when some data is absent;
  - queries that operate uniformly over single-valued and set-valued data;
  - queries that operate uniformly over data with different types;
  - queries that return heterogeneous objects;
  - queries where the object structure is not fully known.

Lorel

- Supports declarative path expressions for traversing graph structures and automatic coercion (force) for handling heterogeneous and type less data.
- A *path expression* is essentially a sequence of edge labels ($L_1.L_2...L_n$), which for given graph yields set of nodes. For example:
  - SimeUEP.PropertyForRent yields set of nodes {&5, &6};
  - SimeUEP.PropertyForRent.street yields set of nodes containing strings {“77 Jalan Ilmu”, “9 Jalan Setia 3”}.

Lore and Lorel

- Also supports general path expression that provides for arbitrary paths:
  - '|' indicates selection;
  - '?' indicates zero or one occurrences;
  - '+' indicates one or more occurrences;
  - '*' indicates zero or more occurrences.
- For example:
  - SimeUEP.(Branch | PropertyForRent).street
  - would match path beginning with SimeUEP, followed by either a Branch edge or a PropertyForRent edge, followed by a street edge.

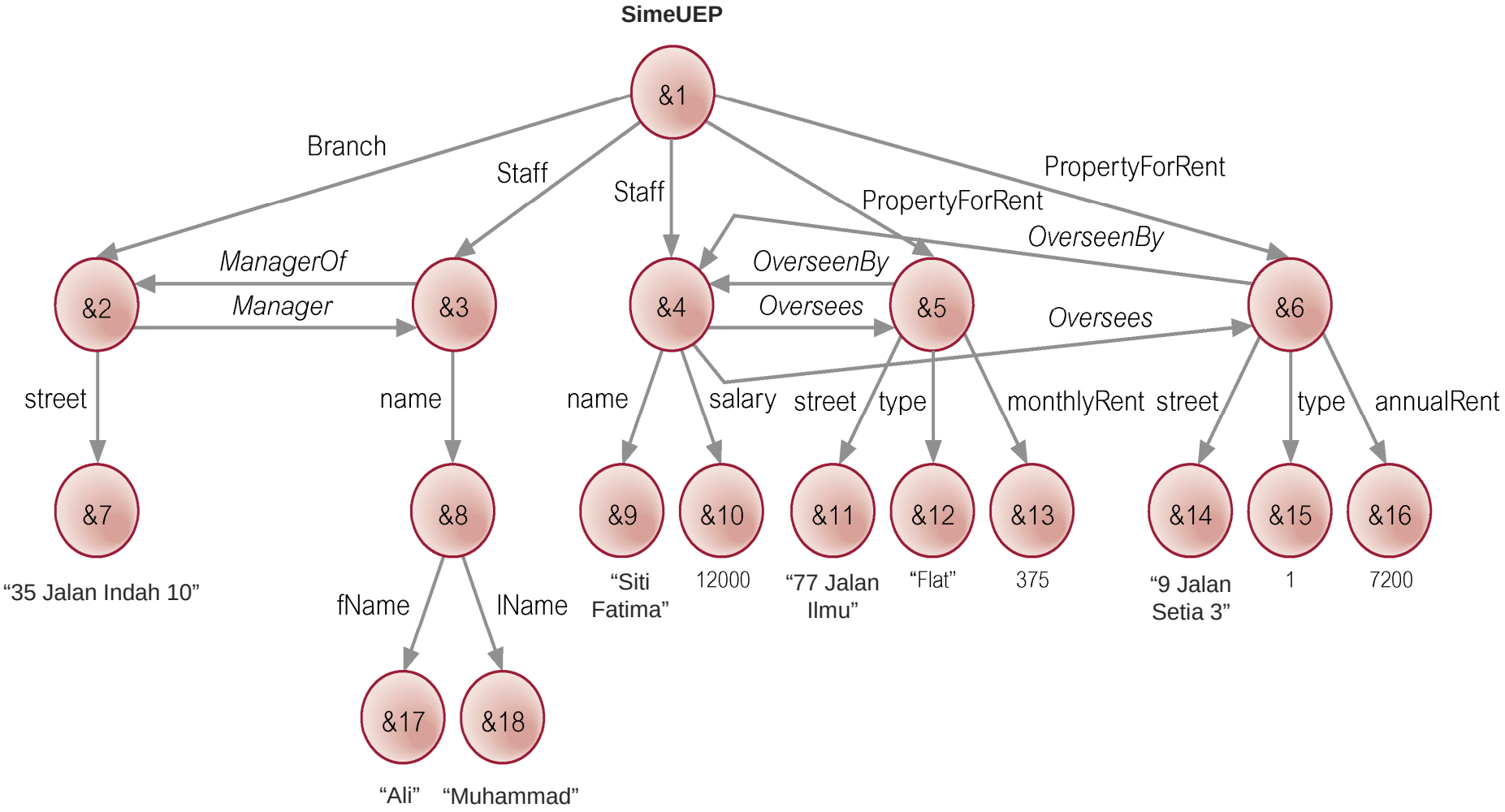
Example Lorel Queries

Find properties overseen by Siti Fatima.

```
SELECT s.Oversees
FROM SimeUEP.Staff s
WHERE s.name = “Siti Fatima”
```

- Data in FROM clause contains objects &3 and &4. Applying WHERE restricts this set to object &4. Then apply SELECT clause.

Graphical representation of example



Example Lorel Queries

Answer:

PropertyForRent &5
street &11 “77 Jalan Ilmu”
type &12 “Flat”
monthlyRent &13 375
OverseenBy &4

PropertyForRent &6
street &14 “9 Jalan Setia 3”
type &15 1
annualRent &16 7200
OverseenBy &4

Example Lorel Queries

Find all properties with annual rent.

```
SELECT SimeUEP.PropertyForRent
FROM SimeUEP.PropertyForRent.annualRent
```

Answer:

```
PropertyForRent &6
        street &14 “9 Jalan Setia 3”
        type &15 1
        annualRent &16 7200
        OverseenBy &4
```

Example Lorel Queries

Find all staff who oversee two or more properties.

```
SELECT SimeUEP.Staff.Name
 FROM SimeUEP.Staff SATISFIES
    2 <= COUNT(SELECT SimeUEP.Staff
                        WHERE SimeUEP.Staff.Oversees)
```

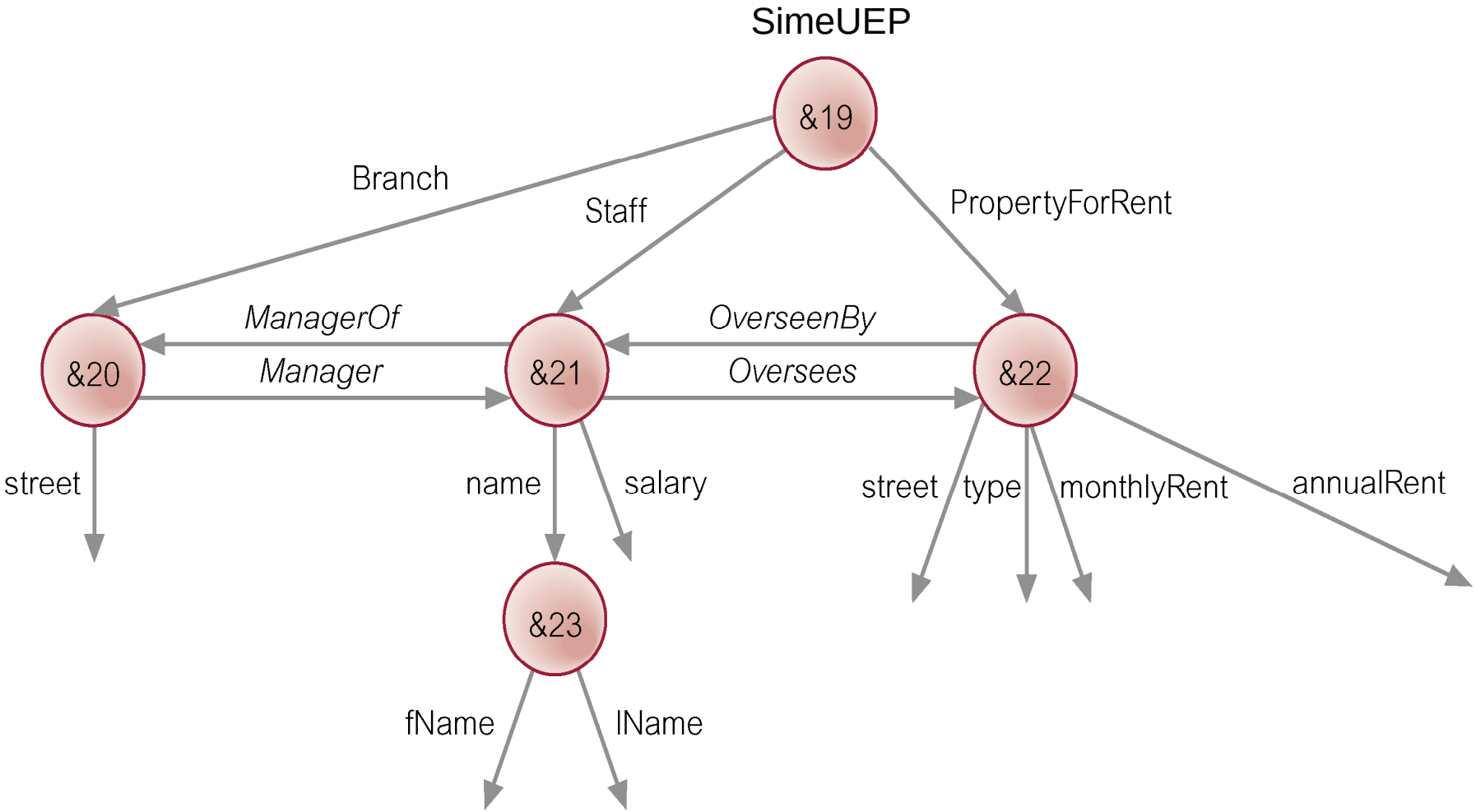
Answer:

name &9 “Siti Fatima”

DataGuide

- A dynamically generated and maintained structural summary of database, which serves as a dynamic schema.
- Has three properties:
  - *conciseness:* every label path in the database appears exactly once in the DataGuide;
  - *accuracy:* every label path in DataGuide exists in original database;
  - *convenience:* a DataGuide is an OEM (or XML) object, so can be stored and accessed using same techniques as for source database.

DataGuide


SimeUEP
&19
Branch
Staff
PropertyForRent
ManagerOf
Manager
OverseenBy
Oversees
&20
&21
&22
street
name
salary
street
type
monthlyRent
annualRent
&23
fName
lName

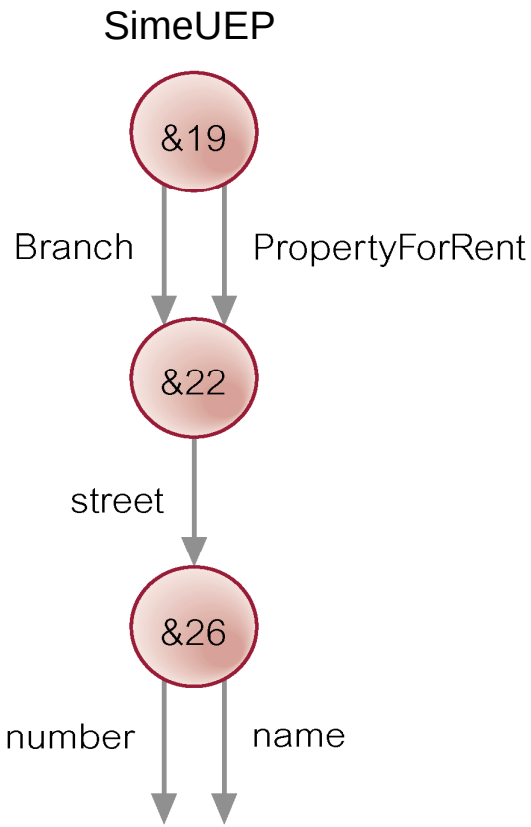
DataGuide

- Can determine whether a given label path of length *n* exists in source database by considering at most *n* objects in the DataGuide.
- For example, to verify whether path Staff.Oversees.annualRent exists, need only examine outgoing edges of objects &19, &21, and &22 in our DataGuide.
- Further, only objects that can follow Branch are the two outgoing edges of object &20.

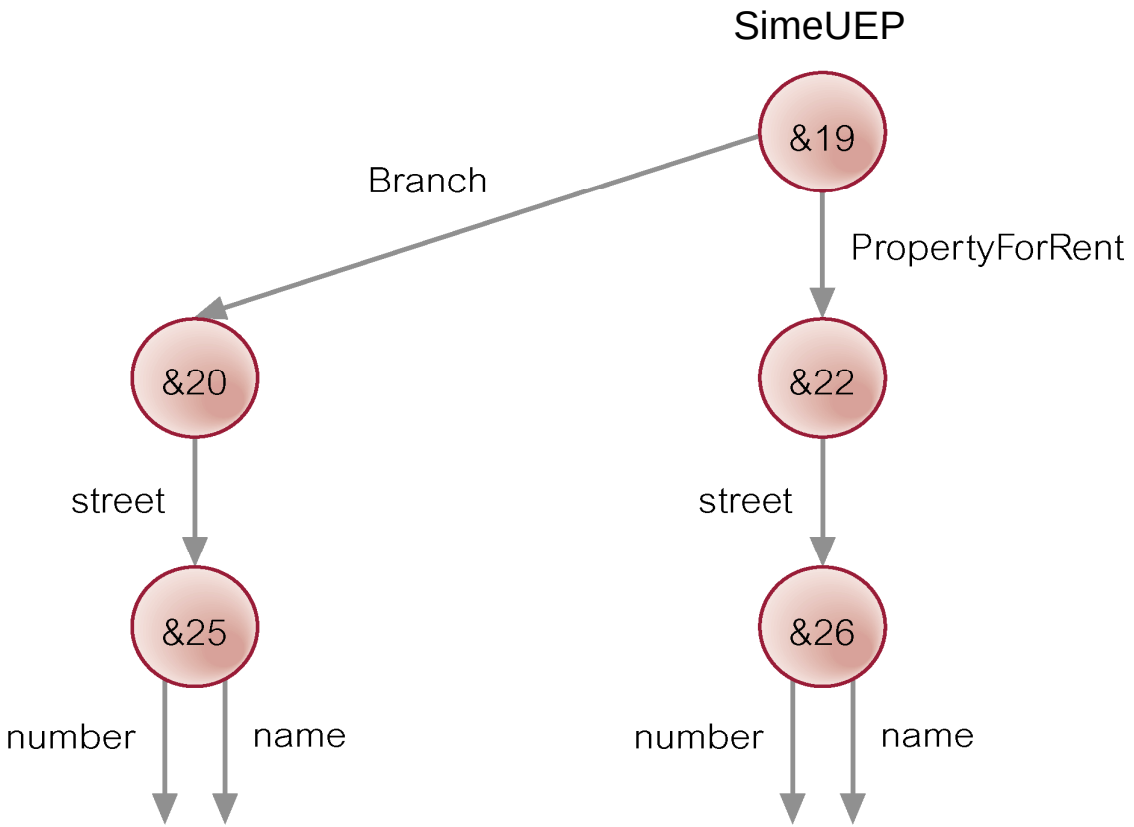
DataGuides

- DataGuides can be classified as strong or weak:
  - strong is where each set of label paths that share same target set in the DataGuide is exactly the set of label paths that share same target set in source database.

DataGuides



(a) weak DataGuide



(b) strong DataGuide

XML (eXtensible Markup Language)

- A meta-language (a language for describing other languages) that enables designers to create their own customized tags to provide functionality not available with HTML.
- Most documents on Web currently stored and transmitted in HTML.
- One strength of HTML is its simplicity. Simplicity may also be one of its weaknesses, with users wanting tags to simplify some tasks and make HTML documents more attractive and dynamic.

XML

- To satisfy this demand, vendors introduced some browser-specific HTML tags, making it difficult to develop sophisticated, widely viewable Web documents.
- W3C has produced XML, which could preserve general application independence that makes HTML portable and powerful.

XML

- XML is a restricted version of Standard Generalized Markup Language (SGML), designed especially for Web documents.
- SGML allows document to be logically separated into two: one that defines the structure of the document (DTD), other containing the text itself.
- By giving documents a separately defined structure, and by giving authors ability to define custom structures, SGML provides extremely powerful document management system.
- However, SGML has not been widely adopted due to its inherent complexity.

XML

- XML attempts to provide a similar function to SGML, but is less complex and, at same time, network-aware.
- XML retains key SGML advantages of extensibility, structure, and validation.
- Since XML is a restricted form of SGML, any fully compliant SGML system will be able to read XML documents (although the opposite is not true).
- XML is not intended as a replacement for SGML or HTML.

Advantages of XML

- Simplicity
- Open standard and platform/vendor-independent
- Extensibility
- Reusable
- Separation of content and presentation
- Improved load balancing

Advantages of XML

- Support for integration of data from multiple sources.
- Ability to describe data from a wide variety of applications.
- More advanced search engines.
- New opportunities.

XML - Elements

- Elements, or tags, are most common form of markup.
- First element must be a root element, which can contain other (sub)elements.
- XML document must have one root element (<STAFFLIST>. Element begins with start-tag (<STAFF>) and ends with end-tag (</STAFF>).
- XML elements are case sensitive.
- An element can be empty, in which case it can be abbreviated to <EMPTYELEMENT/>.
- Elements must be properly nested.

XML - Attributes

- Attributes are name-value pairs that contain descriptive information about an element.
- Attribute is placed inside start-tag after corresponding element name with the attribute value enclosed in quotes.

  <STAFF branchNo = “B005”>
- Could also have represented branch as sub-element of STAFF.
- A given attribute may only occur once within a tag, while sub-elements with same tag may be repeated.

XML – Other Sections

- *XML declaration*: optional at start of XML document.
- *Entity references*: serve various purposes, such as shortcuts to often repeated text or to distinguish reserved characters from content.
  - *Begin with &, end with ;*
  - *Eg. < to represent* <
- *Comments*: enclosed in <!– and --> tags.
- *CDATA sections*: instructs XML processor to ignore markup characters and pass enclosed text directly to application.
- *Processing instructions*: can also be used to provide information to application.

XML – Ordering

- Semi-structured data model described earlier assumes collections are unordered.
- In XML, elements are ordered:

```
<NAME>
    <FNAME>Nadhirah</FNAME>
    <LNAME>Khalim</LNAME>
</NAME>
```

Different from

```
<NAME>
    <LNAME>Khalim</LNAME>
    <FNAME>Nadhirah</FNAME>
</NAME>
```

XML – Ordering

- In contrast, in XML attributes are unordered.

<NAME LNAME= “Khalim FNAME=“Nadhirah”/>

same as

<NAME FNAME=“Nadhirah” LNAME= “Khalim />

Document Type Definitions (DTDs)

- Defines the valid syntax of an XML document.
- Lists element names that can occur in document, which elements can appear in combination with which other ones, how elements can be nested, what attributes are available for each element type, and so on.
- Term *vocabulary* sometimes used to refer to the elements used in a particular application.
- Grammar specified using Extended Backus–Naur Form (EBNF), not XML.
- Although optional, DTD is recommended for document conformity.

Document Type Definitions (DTDs)

```
<!ELEMENT STAFFLIST (STAFF)*>
<!ELEMENT STAFF (NAME, POSITION, DOB?, SALARY)>
<!ELEMENT NAME (FNAME, LNAME)>
<!ELEMENT FNAME (#PCDATA)>
<!ELEMENT LNAME (#PCDATA)>
<!ELEMENT POSITION (#PCDATA)>
<!ELEMENT DOB (#PCDATA)>
<!ELEMENT SALARY (#PCDATA)>
<!ATTLIST STAFF branchNo CDATA #IMPLIED>
```

DTDs – Element Type Declarations

- Identify the rules for elements that can occur in the XML document.
- Eg: <!ELEMENT STAFFLIST (STAFF)*> indicates STAFFLIST consists of zero of more STAFF elements
- Options for repetition are:
  - * indicates zero or more occurrences for an element;
  - + indicates one or more occurrences for an element;
  - ? indicates either zero occurrences or exactly one occurrence for an element.
- Name with no qualifying punctuation must occur exactly once.
- Commas between element names indicate they must occur in succession; if commas omitted, elements can occur in any order.

  <!ELEMENT STAFF (NAME, POSITION, DOB?, SALARY)> means what?
- Base elements declared using special symbol #PCDATA indicate parsable character data
- Element can contain both other elements and #PCDATA

DTDs – Attribute List Declarations

- Identify which elements may have attributes, what attributes they may have, what values attributes may hold, plus optional defaults.
- Eg: <!ATTLIST STAFF branchNo CDATA #IMPLIED> states that branchNo value is a string and is optional tith no default. (If #REQUIRED means must always be provided)
- Some types:
- CDATA: character data, containing any text.
- ID: used to identify individual elements in document (ID is an element name).
- IDREF/IDREFS: must correspond to value of ID attribute(s) for some element in document.
- List of names: values that attribute can hold (enumerated type).

DTDs – Element Identity, IDs, IDREFs

- ID allows unique key to be associated with an element.
- IDREF allows an element to refer to another element with the designated key, and attribute type IDREFS allows an element to refer to multiple elements.

DTDs – Element Identity, IDs, IDREFs

- To loosely model relationship Branch *Has* Staff:
  - <!ATTLIST STAFF staffNo ID #REQUIRED>
  - <!ATTLIST BRANCH staff IDREFS #IMPLIED>

```
<STAFF staffNo = “SL11”>
    <NAME>
      <FNAME>Ali</FNAME><LNAME>Selamat</LNAME>
    </NAME>
</STAFF>
<STAFF staffNo = “SL13”>
    <NAME>
      <FNAME>Dira</FNAME><LNAME>Khalim</LNAME>
    </NAME>
</STAFF>
<BRANCH  staff = “SL11 SL13”>
    <BRANCHNO>B007</BRANCH>
</BRANCH>
```

DTDs – Document Validity

- Two levels of document processing: well-formed and valid.
- Non-validating processor ensures an XML document is well-formed before passing information on to application.
- XML document that conforms to structural and notational rules of XML is considered well-formed; e.g.:
  - document must start with <?xml version “1.0”>;
  - all elements must be within one root element;
  - elements must be nested in a tree structure without any overlap;
  - All non empty elements must have start tag and end tag.

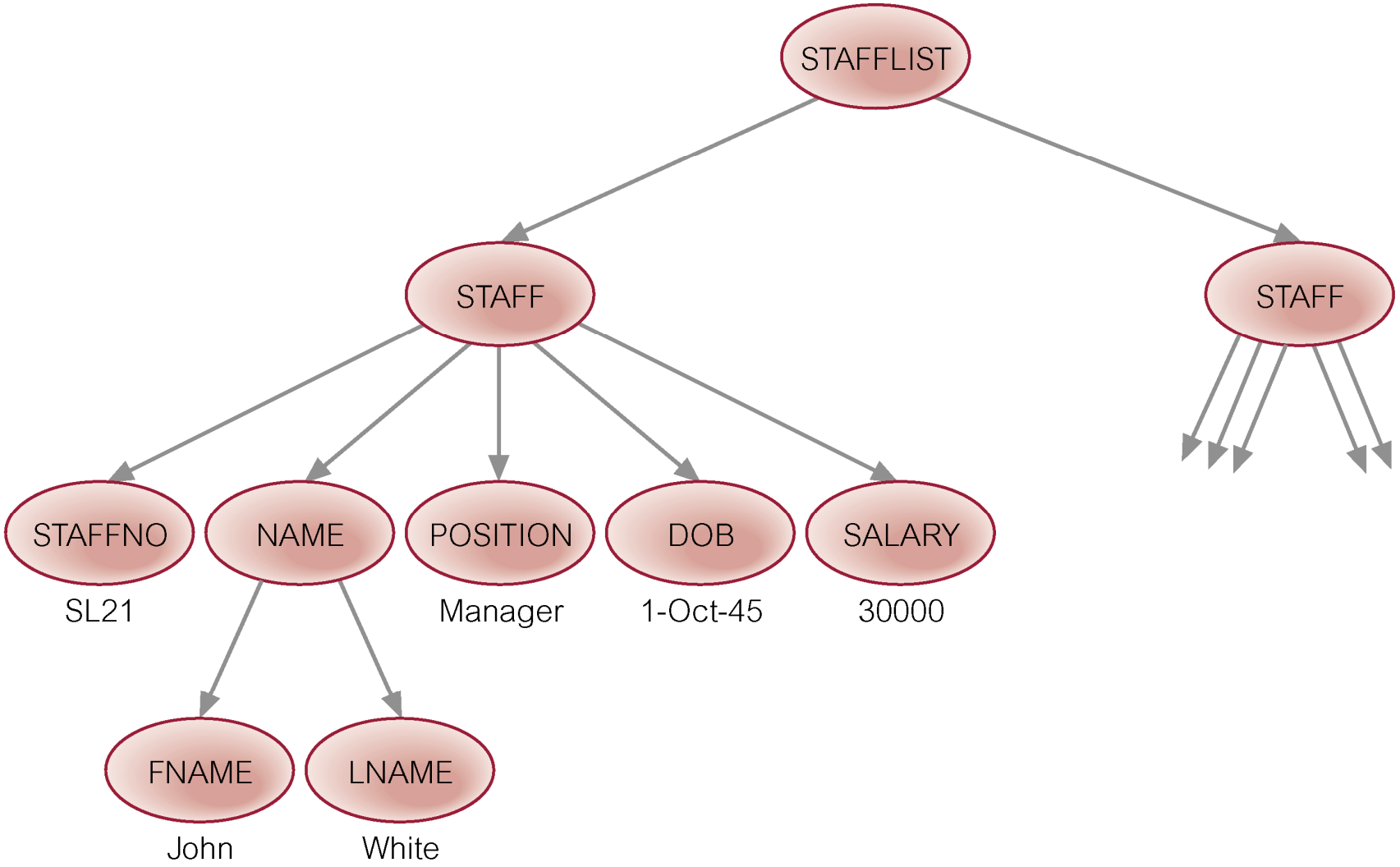
DTDs – Document Validity

- Validating processor will not only check that an XML document is well-formed but that it also conforms to a DTD, in which case XML document is considered valid.
- W3C proposed more expressive alternative to DTD: XML Schema

DOM and SAX

- XML APIs generally fall into two categories: tree-based and event-based.
- DOM (Document Object Model) is tree-based API that provides object-oriented view of data.
- API was created by W3C and describes a set of platform- and language-neutral interfaces that can represent any well-formed XML/HTML document.
- Builds in-memory representation of document and provides classes and methods to allow an application to navigate and process the tree.
- Eg: defines Node interface – has methods to access node's components such as parentNode() & childNode(), add/delete/reorder elements

Representation of Document as Tree-Structure



SAX (Simple API for XML)

- An event-based, serial-access API that uses callbacks to report parsing events to application.
- For example, there are events for start and end elements. Application handles these events through customized event handlers.
- Unlike tree-based APIs, event-based APIs do not built an in-memory tree representation of the XML document.
- API product of collaboration on XML-DEV mailing list, rather than product of W3C.

Namespaces

- Allows element names and relationships in XML documents to be qualified to avoid name collisions for elements that have same name but defined in different vocabularies.
- Allows tags from multiple namespaces to be mixed - essential if data comes from multiple sources.
- For uniqueness, elements and attributes given globally unique names using URI reference.

Namespaces

```
<STAFFLIST xmlns="http://www.simeuep.my/branch5/"
    xmlns:hq = "http://www. simeuep.my/HQ/">
    <STAFF branchNo = "B005">
        <STAFFNO>SL21</STAFFNO>
            ...
        <hq:SALARY>20000</hq:SALARY>
    </STAFF>
</STAFFLIST>
```

XSL (eXtensible Stylesheet Language)

- In HTML, default styling is built into browsers as tag set for HTML is predefined and fixed.
- Cascading Stylesheet Specification (CSS) provides alternative rendering for tags. Can also be used to render XML in a browser but cannot make structural alterations to a document.
- XSL created to define how XML data is rendered and to define how one XML document can be transformed into another document.

XSLT (XSL Transformations)

- A subset of XSL, XSLT is a language in both markup and programming sense, providing a mechanism to transform XML structure into either another XML structure, HTML, or any number of other text-based formats (such as SQL).
- XSLT's main ability is to change the underlying structures rather than simply the media representations of those structures, as with CSS.

XSLT

- XSLT is important because it provides a mechanism for dynamically changing the view of a document and for filtering data.
- Also robust enough to encode business rules and it can generate graphics (not just documents) from data.
- Can even handle communicating with servers (scripting modules can be integrated into XSLT) and can generate the appropriate messages within body of XSLT itself.

XPath

- Declarative query language for XML that provides simple syntax for addressing parts of an XML document.
- Designed for use with XSLT (for pattern matching) and XPointer (for addressing).
- With XPath, collections of elements can be retrieved by specifying a directory-like path, with zero or more conditions placed on the path.
- Uses a compact, string-based syntax, rather than a structural XML-element based syntax, allowing XPath expressions to be used both in XML attributes and in URIs.

XPointer

- Provides access to values of attributes or content of elements anywhere within an XML document.
- Basically an XPath expression occurring within a URI.
- Among other things, with XPointer can link to sections of text, select particular elements or attributes, and navigate through elements.
- Can also select data contained within more than one set of nodes, which cannot do with XPath.

Xpointer(/child::STAFF[attribute::branchNo="B005"]to/child:
:STAFF[attribute::branchNo=B003"])

XLink

- Allows elements to be inserted into XML documents to create and describe links between resources.
- Uses XML syntax to create structures that can describe links similar to simple unidirectional hyperlinks of HTML as well as more sophisticated links.
- Two types of XLink: *simple* and *extended*.
- Simple link connects a source to a destination resource; an extended link connects any number of resources.

XHTML (eXtensible HTML) 1.0

- Reformulation of HTML 4.01 in XML 1.0 and is intended to be next generation of HTML.

- Basically a stricter and cleaner version of HTML; e.g.:
  - tags and attributes must be in lowercase;
  - all XHTML elements must be have an end-tag;
  - attribute values must be quoted and minimization is not allowed;
  - ID attribute replaces the name attribute;
  - documents must conform to XML rules.

Simple Object Access Protocol (SOAP)

- An XML-based messaging protocol that defines a set of rules for structuring messages.
- Protocol can be used for simple one-way messaging but also useful for performing Remote Procedure Call (RPC) style request-response dialogues.
- Not tied to any particular operating system or programming language nor any particular transport protocol, although HTTP is popular.
- Important advantage of SOAP is that most firewalls allow HTTP to pass right through, facilitating point-to-point SOAP data exchanges.

Simple Object Access Protocol (SOAP)

- SOAP message is an XML document containing:
  - A required Envelope element that identifies the XML document as a SOAP message.
  - An optional Header element that contains application specific information such as authentication or payment information.
  - A required Body Header element that contains call and response information.
  - An optional Fault element that provides information about errors that occurred while processing message.

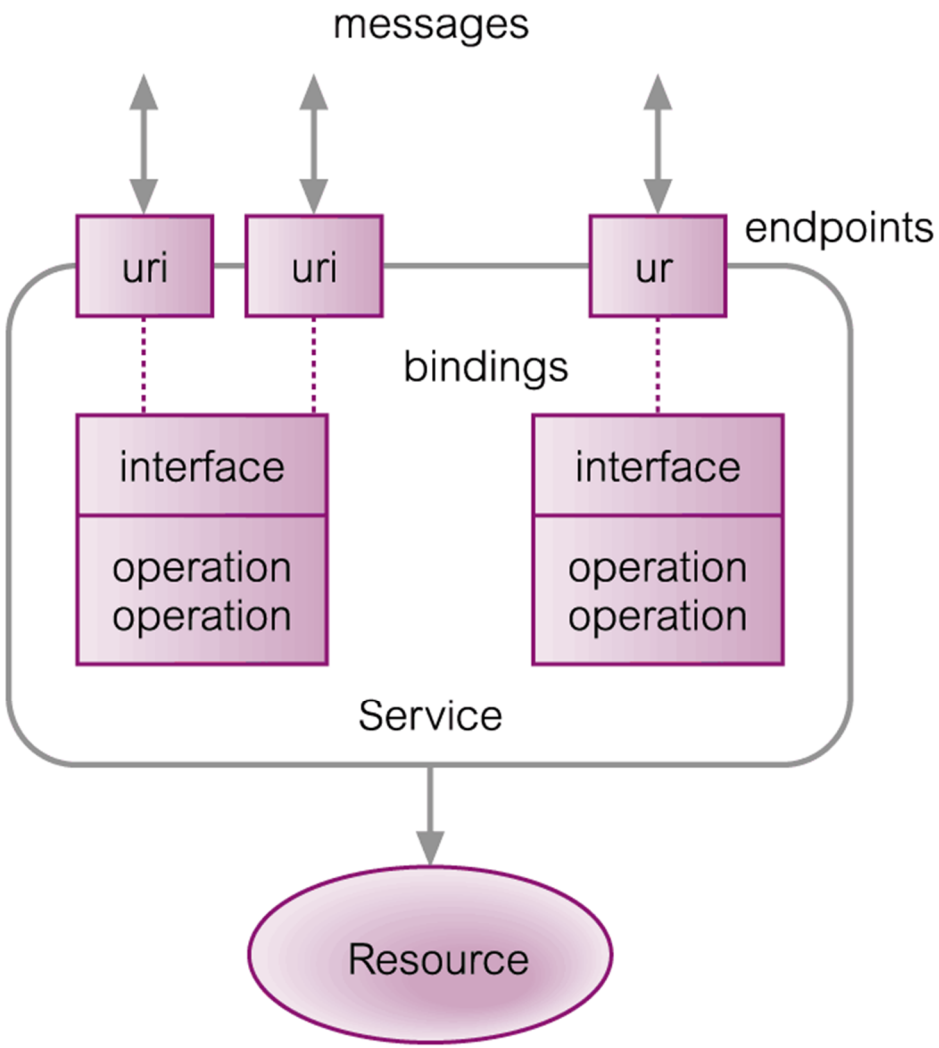
Web Services Description Language (WSDL)

- XML-based protocol for defining a Web service.
- Specifies location of a service, operations service exposes, SOAP messages involved, and communications protocol used to talk to service.
- Notation that a WSDL file uses to describe message formats is typically based on XML Schema.
- Published WSDL descriptions can be used to obtain information about available Web services.

Web Services Description Language (WSDL)

- WSDL 2.0 describes a Web service in two parts: an *abstract* part and a *concrete* part.
- At abstract level, WSDL describes a Web service in terms of the messages it sends and receives; messages are described independent of a specific wire format using a type system, typically XML Schema.
- At concrete level, a *binding* specifies transport and wire format details for one or more interfaces. An *endpoint* associates a network address with a binding and a *service* groups endpoints that implement a common interface.

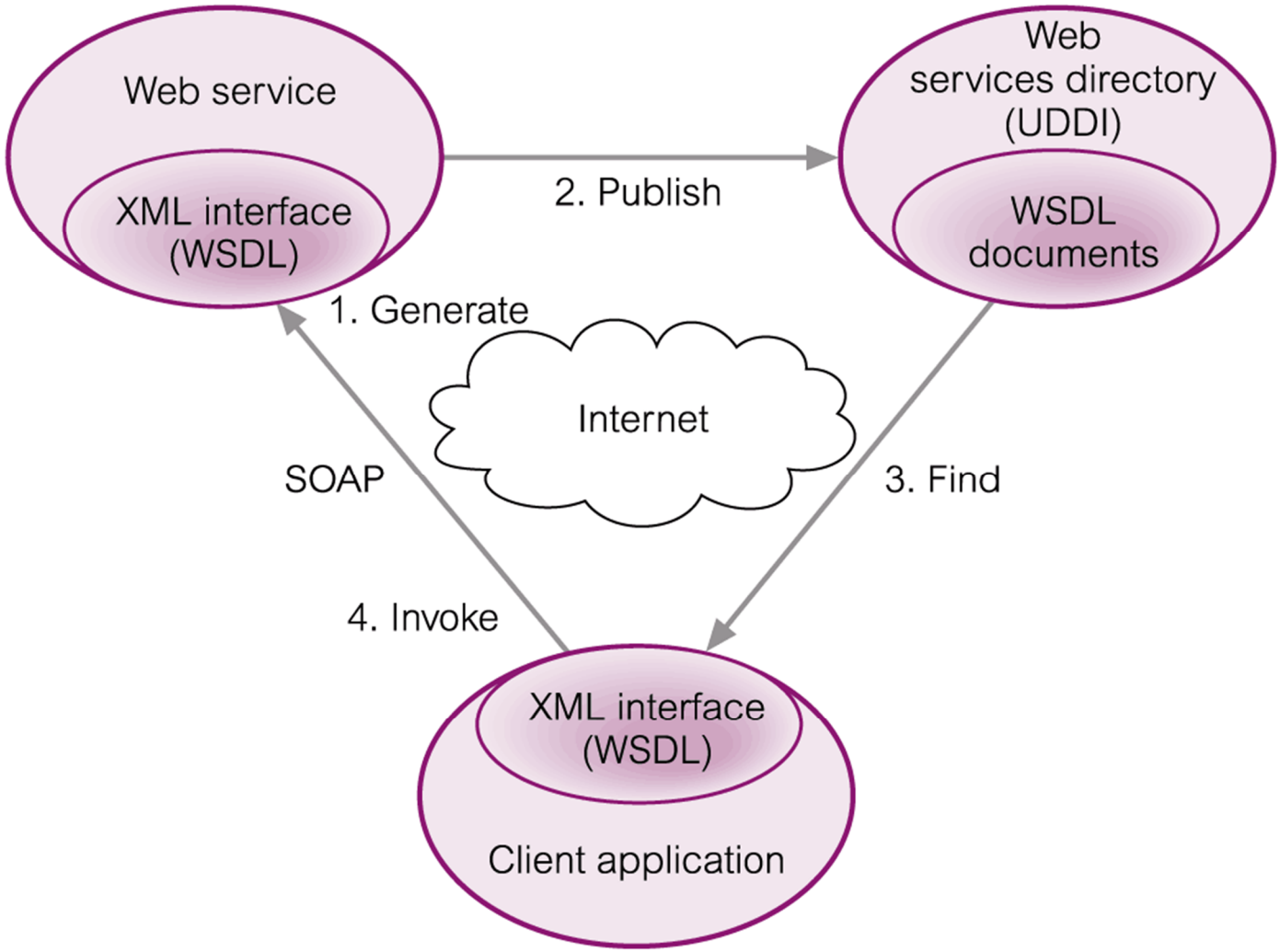
WSDL Concepts



Universal Discovery, Description and Integration (UDDI)

- Defines SOAP-based Web service for locating WSDL-formatted protocol descriptions of Web services.
- Essentially describes online electronic registry that serves as electronic Yellow Pages, providing information structure where various businesses register themselves and services they offer through their WSDL definitions.
- Based on industry standards including HTTP, XML, XML Schema, SOAP, and WSDL.
- Two types of UDDI registries: *public* and *private*.

WSDL and UDDI



XML Schema

- DTDs have number of limitations:
  - it is written in a different (non-XML) syntax;
  - it has no support for namespaces;
  - it only offers extremely limited data typing.
- XML Schema is more comprehensive method of defining content model of an XML document.
- Additional expressiveness will allow Web applications to exchange XML data more robustly without relying on *ad hoc* validation tools.

XML Schema

- XML schema is the definition (both in terms of its organization and its data types) of a specific XML structure.
- XML Schema language specifies how each type of element in schema is defined and the element's data type.
- Schema is an XML document, and so can be edited and processed by same tools that read the XML it describes.

XML Schema – Simple Types

- Elements that do not contain other elements or attributes are of type simpleType.

```
<xsd:element name="STAFFNO" type = "xsd:string"/>
<xsd:element name="DOB" type = "xsd:date"/>
<xsd:element name="SALARY" type = "xsd:decimal"/>
```

- Attributes must be defined last:

```
<xsd:attribute name="branchNo" type = "xsd:string"/>
```

XML Schema – Complex Types

- Elements that contain other elements are of type complexType.
- List of children of complex type are described by sequence element.

```
<xsd:element name = “STAFFLIST”>
    <xsd:complexType>
        <xsd:sequence>
            <!-- children defined here -->
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Cardinality

- Cardinality of an element can be represented using attributes minOccurs and maxOccurs.
- To represent an optional element, set minOccurs to 0; to indicate there is no maximum number of occurrences, set maxOccurs to “unbounded”.

```
<xsd:element name=“DOB” type=“xsd:date”
    minOccurs = “0”/>
<xsd:element name=“NOK” type=“xsd:string”
    minOccurs = “0” maxOccurs = “3”/>
```

References

- Can use references to elements and attribute definitions.

```
<xsd:element name="STAFFNO" type="xsd:string"/>
....
<xsd:element ref = "STAFFNO"/>
```

- If there are many references to STAFFNO, use of references will place definition in one place and improve the maintainability of the schema.

Defining New Types

- Can also define new data types to create elements and attributes.

```
<xsd:simpleType name = “STAFFNOTYPE”>
        <xsd:restriction base = “xsd:string”>
                <xsd:maxLength value = “5”/>
        </xsd:restriction>
 </xsd:simpleType>
```

- New type has been defined as a restriction of string (to have maximum length of 5 characters).

Groups

- Can define both groups of elements and groups of attributes. Group is not a data type but acts as a *container* holding a set of elements or attributes.

```
<xsd:group name = “StaffType”>
   <xsd:sequence>
      <xsd:element name=“StaffNo” type=“StaffNoType”/>
     <xsd:element name=“Position” type=“PositionType”/>
     <xsd:element name=“DOB” type =“xsd:date”/>
     <xsd:element name=“Salary” type=“xsd:decimal”/>
   </xsd:sequence>
</xsd:group>
```

Constraints

- XML Schema provides XPath-based features for specifying uniqueness constraints and corresponding reference constraints that will hold within a certain scope.

```
<xsd:unique name = “NAMEDOBUNIQUE”>
    <xsd:selector xpath = “STAFF”/>
    <xsd:field xpath = “NAME/LNAME”/>
    <xsd:field xpath = “DOB”/>
</xsd:unique>
```

Key Constraints

- Similar to uniqueness constraint except the value has to be non-null. Also allows the key to be referenced.

```
<xsd:key name = “STAFFNOISKEY”>
    <xsd:selector xpath = “STAFF”/>
    <xsd:field xpath = “STAFFNO”/>
</xsd:key>
```

Resource Description Framework (RDF)

- Even XML Schema does not provide the support for semantic interoperability required.
- For example, when two applications exchange information using XML, both agree on use and intended meaning of the document structure.
- Must first build a model of the domain of interest, to clarify what kind of data is to be sent from first application to second.
- However, as XML Schema just describes a grammar, there are many different ways to encode a specific domain model into an XML Schema, thereby losing the direct connection from the domain model to the Schema.

Resource Description Framework (RDF)

- Problem compounded if third application wishes to exchange information with other two.
- Not sufficient to map one XML Schema to another, since the task is not to map one grammar to another grammar, but to map objects and relations from one domain of interest to another.
- Three steps required:
  - reengineer original domain models from XML Schema;
  - define mappings between the objects in the domain models;
  - define translation mechanisms for the XML documents, for example using XSLT.

Resource Description Framework (RDF)

- RDF is infrastructure that enables encoding, exchange, and reuse of structured meta-data.
- This infrastructure enables meta-data interoperability through design of mechanisms that support common conventions of semantics, syntax, and structure.
- RDF does not stipulate semantics for each domain of interest, but instead provides ability for these domains to define meta-data elements as required.
- RDF uses XML as a common syntax for exchange and processing of meta-data.

RDF Data Model

- Basic RDF data model consists of three objects:

  Resource: anything that can have a URI; e.g., a Web page, a number of Web pages, or a part of a Web page, such as an XML element.

  Property: a specific attribute used to describe a resource; e.g., attribute Author may be used to describe who produced a particular XML document.

  Statement: consists of combination of a resource, a property, and a value.

RDF Data Model

- Components known as “subject”, “predicate”, and “object” of an RDF statement.
- Example statement:

  “Author of http://www.simeuep.my/staff_list.xml is Ali Muhammad”

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:s="http://www. simeuep.my/schema/">
      <rdf:Description about="http://www. simeuep.my/staff_list.xml">
       <s:Author> Ali Muhammad </s:Author>
     </rdf:Description>
    </rdf:RDF>
```

RDF Schema

- Specifies information about classes in a schema including properties (attributes) and relationships between resources (classes).
- RDF Schema mechanism provides a basic *type system* for use in RDF models, analogous to XML Schema.
- Defines resources and properties such as rdfs:Class and rdfs:subClassOf that are used in specifying application-specific schemas.
- Also provides a facility for specifying a small number of constraints such as cardinality.

XML Query Languages

- Data extraction, transformation, and integration are well-understood database issues that rely on a query language.
- SQL and OQL do not apply directly to XML because of the irregularity of XML data.
- However, XML data similar to Semi-structured data. There are many Semi-structured query languages that can query XML documents, including XML-QL, Unstructured data Query Language (UnQL), and XML Query Language (XQL).
- All have notion of a path expression for navigating nested structure of XML.

Example XML-QL

Find surnames of staff who earn more than RM30,000.

```
WHERE <STAFF>
    <SALARY> $S </SALARY>
    <NAME><FNAME> $F </FNAME> <LNAME> $L
    </LNAME></NAME>
    </STAFF> IN "http://www.simeuep.my/staff.xml"
    $S > 20000
CONSTRUCT <LNAME> $L </LNAME>
```

XML Query Working Group

- W3C formed an XML Query Working Group in 1999 to produce a data model for XML documents, set of query operators on this model, and query language based on query operators.

- Queries operate on single documents or fixed collections of documents, and can select entire documents or subtrees of documents that match conditions based on document content/structure.

- Queries can also construct new documents based on what has been selected.

XML Query Working Group

- Ultimately, collections of XML documents will be accessed like databases.

- Working Group has produced four documents:
  - XML Query (XQuery) Requirements;
  - XML XQuery 1.0 and XPath 2.0 Data Model;
  - XML XQuery 1.0 and XPath 2.0 Formal Semantics;
  - XQuery 1.0 – A Query Language for XML;
  - XML XQuery 1.0 and XPath 2.0 Functions and Operators;
  - XSLT 2.0 and XPath 1.0 Serialization.

XML Query Requirements

- Specifies goals, usage scenarios, and requirements for XQuery Data Model and query language. For example:
  - language must be declarative and must be defined independently of any protocols with which it is used;
  - queries should be possible whether or not a schema exists;
  - language must support both universal and existential quantifiers on collections and it must support aggregation, sorting, nulls, and be able to traverse inter- and intra-document references.

XQuery

- XQuery derived from XML query language called Quilt, which has borrowed features from XPath, XML-QL, SQL, OQL, Lorel, XQL.
- Like OQL, XQuery is a functional language in which a query is represented as an expression.
- XQuery supports several kinds of expression, which can be nested (supporting notion of a subquery).

XQuery – Path Expressions

- Uses syntax of XPath.
- In XQuery, result of a path expression is ordered list of nodes, including their descendant nodes, ordered according to their position in original hierarchy, top-down, left-to-right order.
- Result of path expression may contain duplicate values.
- Each step in path expression represents movement through document in particular direction, and each step can eliminate nodes by applying one or more predicates.

XQuery – Path Expressions

- Result of each step is list of nodes that serves as starting point for next step.
- Path expression can begin with an expression that identifies a specific node, such as function doc(string), which returns root node of named document.
- Query can also contain path expression beginning with "/" or "//", which represents an implicit root node determined by the environment in which query is executed.

XQuery – Path Expressions

- Find staff number of first member of staff in our XML document.

  doc("staff_list.xml")/STAFFLIST/STAFF[1]//STAFFNO

- Four steps:
  - first opens staff_list.xml and returns its document node;
  - second uses /STAFFLIST to select STAFFLIST element at top;
  - third locates first STAFF element that is child of root element;
  - fourth finds STAFFNO elements occurring anywhere within this STAFF element.

XQuery – Path Expressions

- Knowing structure of document, could also express this as:

  doc("staff_list.xml")//STAFF[1]/STAFFNO

  doc("staff_list.xml")/STAFFLIST/STAFF[1]/STAFFNO

XQuery – Path Expressions

Find staff numbers of first two members of staff.

```
doc("staff_list.xml")/STAFFLIST/STAFF[1 TO 2]/STAFFNO
```

XQuery – Path Expressions

Find surnames of staff at branch B005.

```
doc("staff_list.xml")/STAFFLIST/
        STAFF[@branchNo ="B005"]//LNAME
```

- Five steps:
  - first two as before;
  - third uses /STAFF to select STAFF elements within STAFFLIST element;
  - fourth consists of predicate that restricts STAFF elements to those with branchNo attribute = B005;
  - fifth selects LNAME element(s) occurring anywhere within these elements.

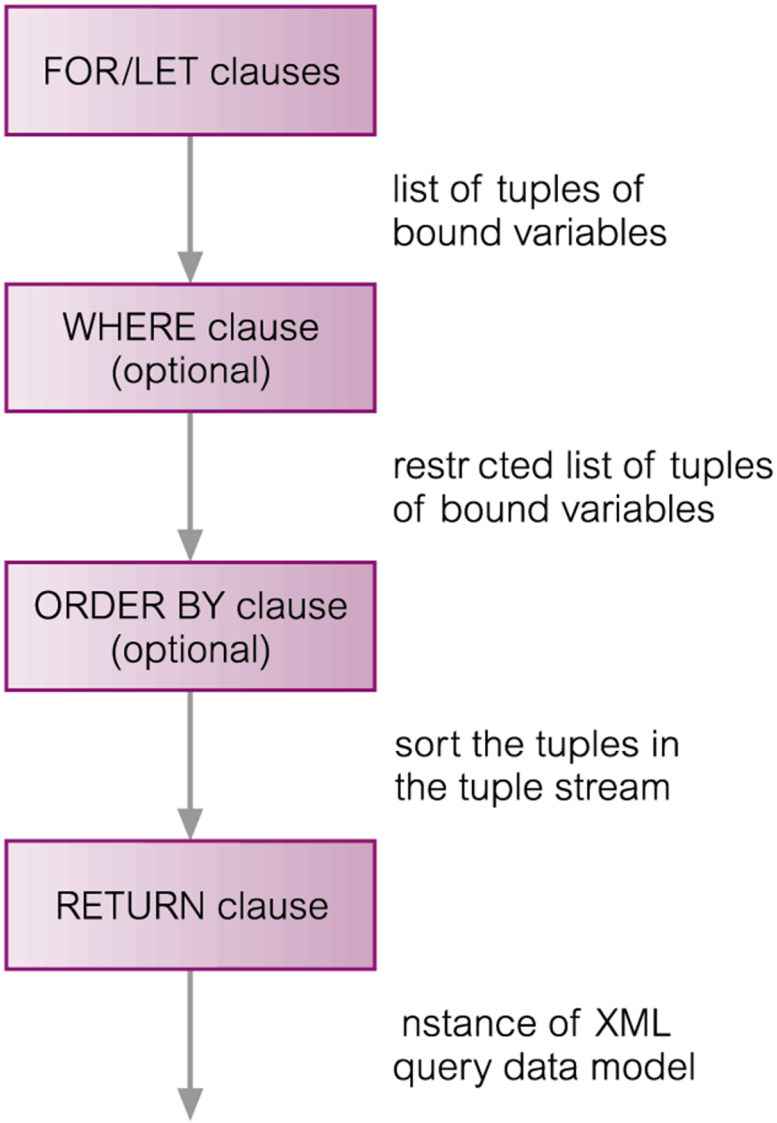
XQuery – FLWOR Expressions

- FLWOR ("flower") expression is constructed from FOR, LET, WHERE, ORDER BY, RETURN clauses.
- FLWOR expression starts with one or more FOR or LET clauses in any order, followed by optional WHERE clause, optional ORDER BY clause, and required RETURN clause.
- FOR and LET clauses serve to bind values to one or more variables using expressions (e.g., path expressions).
- FOR used for iteration, associating each specified variable with expression that returns list of nodes.
- FOR clause can be thought of as iterating over nodes returned by its respective expression.

XQuery – FLWOR Expressions

- LET clause also binds one or more variables to one or more expressions but without iteration, resulting in single binding for each variable.
- Optional WHERE clause specifies one or more conditions to restrict tuples generated by FOR and LET.
- RETURN clause evaluated once for each tuple in tuple stream and results concatenated to form result.
- ORDER BY clause, if specified, determines order of the tuple stream which, in turn, determines order in which RETURN clause is evaluated using variable bindings in the respective tuples.

XQuery – FLWOR Expressions



XQuery – FLWOR Expressions

List staff with salary = RM30,000.

```
LET $SAL := 30000
RETURN doc("staff_list.xml")//STAFF[SALARY = $SAL]
```

- Note, predicate seems to compare an element (SALARY) with a value (15000). In fact, '=' operator extracts typed value of element resulting in a decimal value in this case, which is then compared with 15000.

XQuery – FLWOR Expressions

- '=' operator is a *general comparison operator*. XQuery also defines *value comparison operators* ('eq', 'ne', 'lt', 'le', 'gt', 'ge'), which are used to compare two atomic values.

- If either operand is a node, atomization is used to convert it to an atomic value.

- If we try to compare an atomic value to an expression that returns multiple nodes, then a general comparison operator returns true if *any* value satisfies predicate; however, value comparison operator would raise an error.

XQuery – FLWOR Expressions

List staff at branch B005 with salary > RM15,000.

```
FOR $S IN doc("staff_list.xml")//STAFF
 WHERE $S/SALARY > 15000 AND
                   $S/@branchNo = "B005"
 RETURN $S/STAFFNO
```

XQuery – FLWOR Expressions

List all staff in descending order of staff number.

```
FOR $S IN doc("staff_list.xml")//STAFF
ORDER BY $S/STAFFNO DESCENDING"
RETURN $S/STAFFNO
```

XQuery – FLWOR Expressions

List each branch office and average salary at branch.

```
FOR $B IN
    distinct-values(doc("staff_list.xml")//@branchNo))
LET $avgSalary :=  avg(doc("staff_list.xml")//
                    STAFF[@branchNo = $B]/SALARY)
 RETURN
      <BRANCH>
          <BRANCHNO>{ $B/text() }</BRANCHNO>,
          <AVGSALARY>$avgSalary</AVGSALARY>
      </BRANCH>
```

XQuery – FLWOR Expressions

List branches that have more than 20 staff.

```
<LARGEBRANCHES>
  FOR $B IN
      distinct-values(doc("staff_list.xml")//@branchNo)
      LET $S := doc("staff_list.xml")//STAFF/[@branchNo = $B]
    WHERE count($S) > 20
     RETURN
          <BRANCHNO>{ $B/text() }</BRANCHNO>
</LARGEBRANCHES>
```

XQuery – FLWOR Expressions

List branches with at least one member of staff with salary > RM15,000.

```
<BRANCHESWITHLARGESALARIES>
  FOR $B IN
     distinct-values(doc("staff_list.xml")//@branchNo)
     LET $S := doc("staff_list.xml")//STAFF/[@branchNo = $B]
     WHERE SOME $sal IN $S/SALARY
        SATISFIES ($sal > 15000)
    RETURN
        <BRANCHNO>{ $B/text() }</BRANCHNO>
</ BRANCHESWITHLARGESALARIES >
```

XQuery – Joining Two Documents

List staff along with details of their next of kin.

```
FOR  $S IN doc("staff_list.xml")//STAFF,
          $NOK IN doc("nok.xml")//NOK
WHERE $S/STAFFNO = $NOK/STAFFNO
RETURN
     <STAFFNO>{ $S, $NOK/NAME }</STAFFNO>
```

XQuery – Joining Two Documents

List all staff along with details of their next of kin.

```
FOR  $S IN doc("staff_list.xml")//STAFF
RETURN
   <STAFFNOK>
        { $S }
        FOR $NOK IN doc("nok.xml")//NOK
        WHERE $S/STAFFNO = $NOK/STAFFNO
        RETURN $NOK/NAME
   </STAFFNOK>
```

XQuery – Joining Two Documents

List each branch office and staff who work there.

```
<BRANCHLIST>
  FOR  $B IN
          distinct-values(doc("staff_list.xml")//@branchNo)
  ORDER BY $B
  RETURN
   <BRANCHNO> { $B/text() } {
          FOR $S IN doc("staff_list.xml")//STAFF
          WHERE $S/@branchNo = $B
          ORDER BY $S/STAFFNO
          RETURN $S/STAFFNO, $S/NAME, $S/POSITION, $S/SALARY }
   </BRANCHNO>
</BRANCHLIST>
```

XQuery – User-Defined Function

Function to return staff at a given branch.

```
DEFINE FUNCTION staffAtBranch($bNo) AS element()* {
        FOR $S IN doc("staff_list.xml")//STAFF
        WHERE $S/@branchNo = $bNo
        ORDER BY $S/STAFFNO
        RETURN $S/STAFFNO, $S/NAME,
                        $S/POSITION, $S/SALARY
}
staffAtBranch($B)
```

XML Information Set (Infoset)

- Abstract description of information available in well-formed XML document that meets certain XML namespace constraints.
- XML Infoset is attempt to define set of terms that other XML specifications can use to refer to the information items in a well-formed (although not necessarily valid) XML document.
- Does not attempt to define complete set of information, nor does it represent minimal information that an XML processor should return to an application.
- It also does not mandate a specific interface or class of interfaces (although Infoset presents information as tree).

XML Information Set (Infoset)

- XML document's information set consists of two or more information items.
- An information item is an abstract representation of a component of an XML document such as an element, attribute, or processing instruction.
- Each information item has a set of associated properties. e.g., document information item properties include:
  - [document element];
  - [children];
  - [notations]; [unparsed entities];
  - [base URI], [character encoding scheme], [version], and [standalone].

XQuery 1.0 and XPath 2.0 Data Model

- Defines the information contained in the input to an XSLT or XQuery Processor.
- Also defines all permissable values of expressions in XSLT, XQuery, and XPath.
- Data Model is based on XML Infoset, with following new features:
  - support for XML Schema types;
  - representation of collections of documents and of simple and complex values.

XQuery 1.0 and XPath 2.0 Data Model

- Decided to make XPath subset of XQuery.
- XPath spec shows how to represent information in XML Infoset as a tree structure containing seven kinds of nodes (document, element, attribute, text, comment, namespace, or processing instruction), with XPath operators defined in terms of these seven nodes.
- To retain these operators while using richer type system provided by XML Schema, XQuery extended XPath data model with additional information contained in PSVI.

XQuery 1.0 and XPath 2.0 Data Model

- Data Model is node-labeled, tree-constructor, with notion of node identity to simplify representation of reference values (such as IDREF, XPointer, and URI values).
- An instance of data model represents one or more complete documents or document parts, each represented by its own tree of nodes.
- Every value is ordered sequence of zero or more *items*, where an item can be an *atomic value* or a *node*.
- An atomic value has a *type*, either one of atomic types defined in XML Schema or restriction of one of these types.
- When a node is added to a sequence its identity remains same. Thus, a node may occur in more than one sequence and a sequence may contain duplicate items.

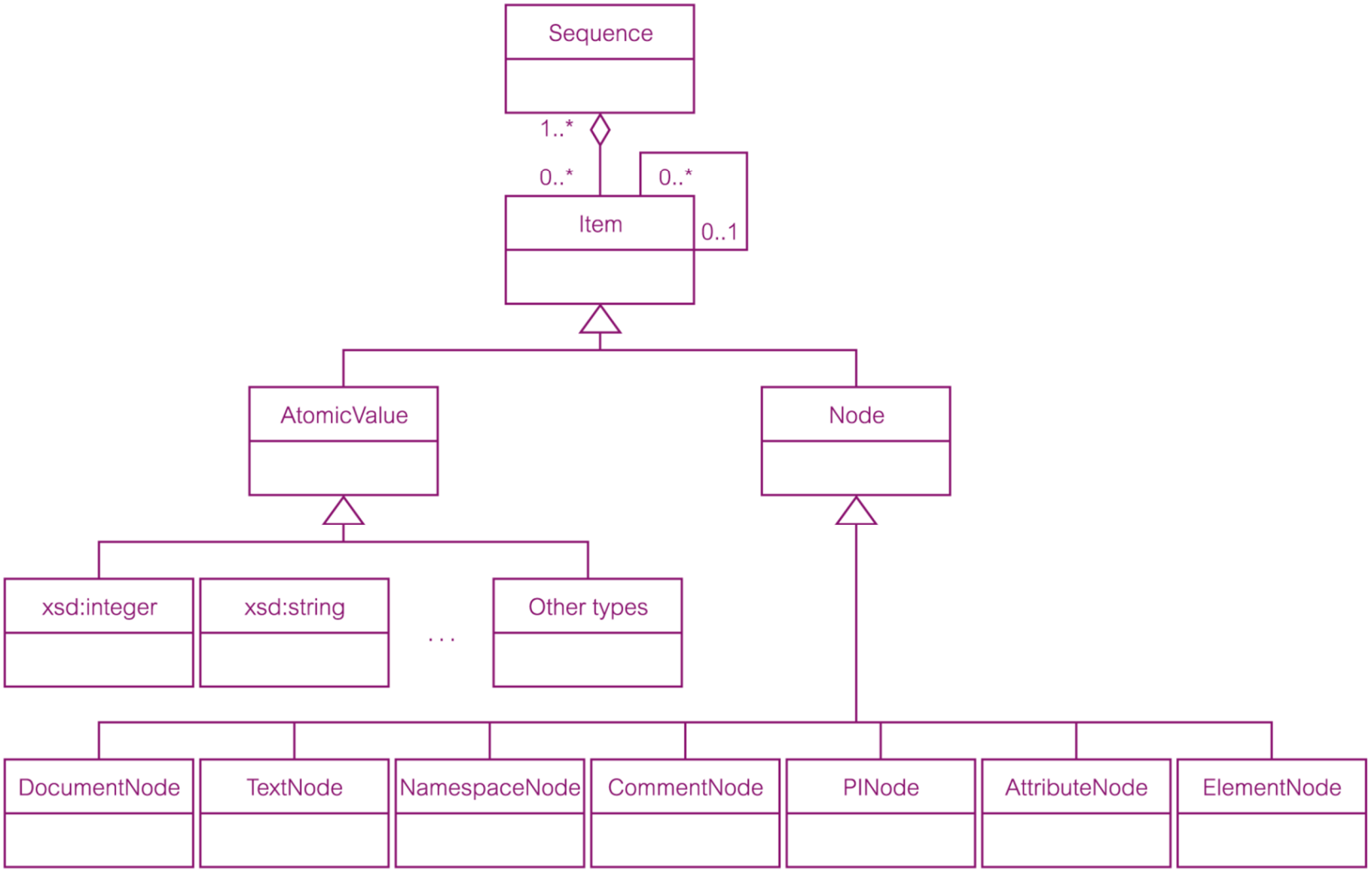
XQuery 1.0 and XPath 2.0 Data Model

- Root node representing XML document is a document node and each element in document is represented by an element node.
- Attributes represented by attribute nodes and content by text nodes and nested element nodes.
- Primitive data in document is represented by text nodes, forming the leaves of the node tree.
- Element node may be connected to attribute nodes and text nodes/nested element nodes.
- Every node belongs to exactly one tree, and every tree has exactly one root node.
- Tree whose root node is document node is referred to as a document and a tree whose root node is some other kind of node is referred to as a fragment.

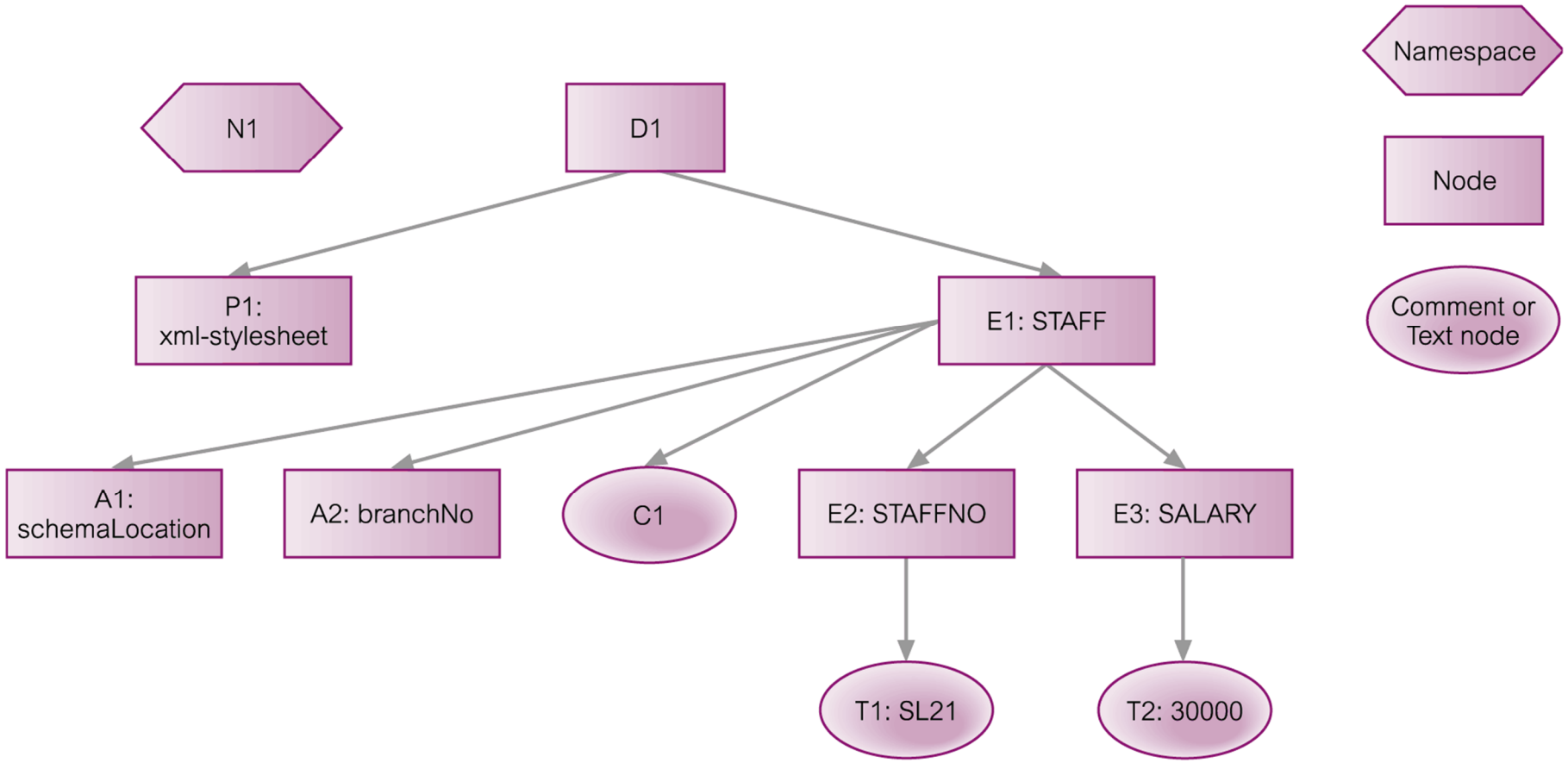
XQuery 1.0 and XPath 2.0 Data Model

- Information about nodes obtained via accessor functions that can operate on any node.
- Accessor functions are analogous to an information item's named properties.
- These functions are illustrative and intended to serve as concise description of information that must be exposed by Data Model.
- Data Model also specifies a number of constructor functions whose purpose is to illustrate how nodes are constructed.

ER Diagram Representing Main Components



XML Query Data Model



Instance of XML Query Data Model

XQuery Formal Semantics

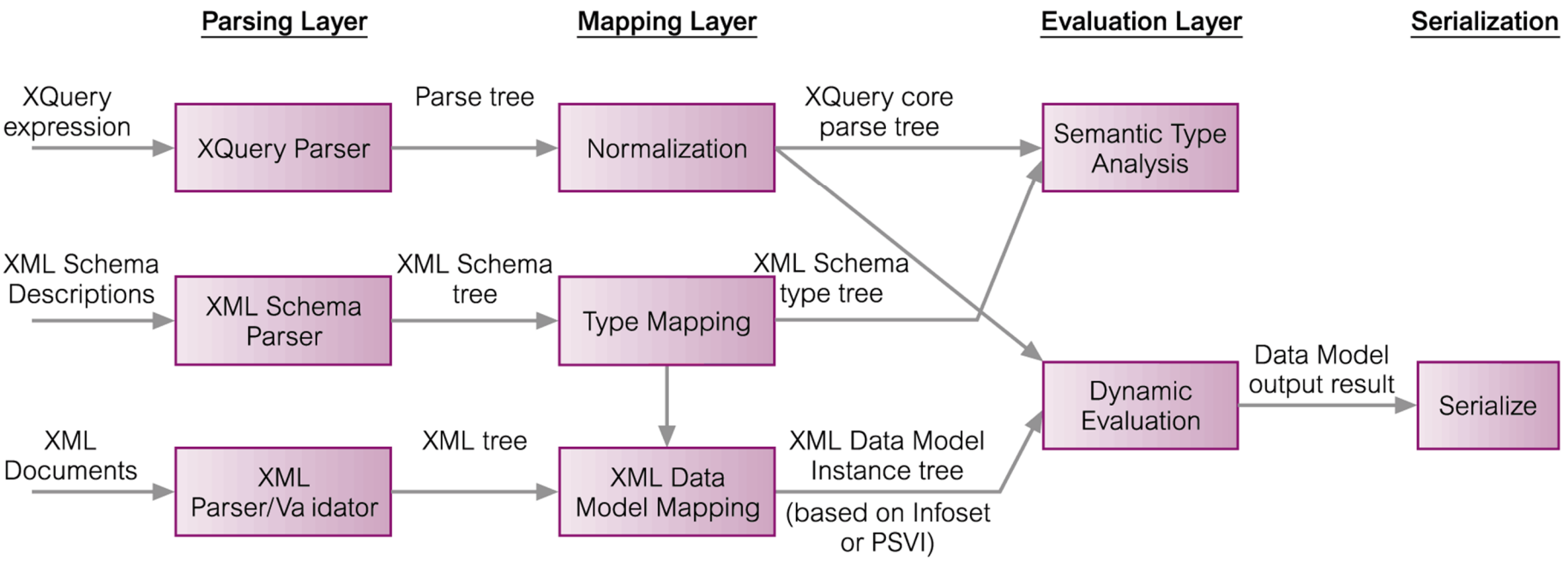
- ‘goal is to complement XPath/XQuery spec, by defining meaning of expressions with mathematical rigor. A rigorous formal semantics clarifies intended meaning of the English specification, ensures that no corner cases are left out, and provides reference for implementation’.

- Provides implementers with a processing model and a complete description of the language’s static and dynamic semantics.

XQuery Formal Semantics – Main Phases

- *Parsing*, ensures input expression is instance of language defined by the grammar rules and then builds an internal parse tree.
- *Normalization*, converts expression into an XQuery Core expression.
- *Static type analysis* (optional), checks whether each (core) expression is type safe and, if so, determines its static type. If expression is not type-safe, type error is *raised*; otherwise, parse tree built with each subexpression *annotated* with its static type.
- *Dynamic evaluation*, computes value of the expression from parse tree. May result in a dynamic error, either a type error (if static type analysis has done) or a non-type error.

XQuery Formal Semantics – Main Phases



XQuery Formal Semantics – Normalization

- Takes full XQuery expression and transforms it into an equivalent expression in the core XQuery.
- Written as follows:

$$[\text{Expr}]_{\text{Expr}}$$

==

CoreExpr

- States that Expr is normalized to CoreExpr (Expr subscript indicates an expression; other values possible; e.g. Axis).

XQuery Formal Semantics – Normalization

- FLWOR expression covered by two sets of rules; first splits expression at clause level then applies further normalization to each clause:

$$[(\textit{ForClause} \mid \textit{LetClause} \mid \textit{WhereClause} \mid \textit{OrderByClause})\ \textit{FLWORExpr}]_{\text{Expr}}$$

==

$$[(\textit{ForClause} \mid \textit{LetClause} \mid \textit{WhereClause} \mid \textit{OrderByClause})]_{\text{FLWOR}}\ ([\textit{FLWORExpr}]_{\text{Expr}})$$

$$[(\textit{ForClause} \mid \textit{LetClause} \mid \textit{WhereClause} \mid \textit{OrderByClause})\ \text{RETURN}\ \textit{Expr}]_{\text{Expr}}$$

==

$$[(\textit{ForClause} \mid \textit{LetClause} \mid \textit{WhereClause} \mid \textit{OrderByClause})]_{\text{FLWOR}}\ ([\textit{Expr}]_{\text{Expr}})$$

XQuery Formal Semantics – Normalization

- Second set applies to FOR and LET clauses and transforms each into series of nested clauses, each of which binds one variable. For example, for the FOR clause we have:

$[\text{FOR } \text{varRef}_1\ \text{TypeDec}_1?\ \text{PositionalVar}_1?\ \text{IN } \text{Expr}_1, \ldots,$
$\text{varRef}_n\ \text{TypeDec}_n?\ \text{PositionalVar}_n?\ \text{IN } \text{Expr}_n]_{\text{FLWOR}}(\text{Expr})$
$==$
$\text{FOR } \text{varRef}_1\ \text{TypeDec}_1?\ \text{PositionalVar}_1?\ \text{IN } [\text{Expr}_1]_{\text{Expr}}\ \text{RETURN } \ldots$
$\text{FOR } \text{varRef}_n\ \text{TypeDec}_n?\ \text{PositionalVar}_n?\ \text{IN } [\text{Expr}_n]_{\text{Expr}}\ \text{RETURN Expr}$

XQuery Formal Semantics – Normalization

- WHERE clause normalized to IF expression that returns an empty sequence if condition is false and normalizes result:

$$[\text{WHERE } \text{Expr}_1]_{\text{FLWOR}}(\text{Expr})$$

==

$$\text{IF } ([\text{Expr}_1]_{\text{Expr}}) \text{ THEN Expr ELSE ( )}$$

Normalization - Example

```
FOR $i IN $I, $j IN $J
    LET $k := $i + $j
    WHERE $k > 2
    RETURN ($i, $j)

FOR $i IN $I RETURN
    FOR $j in $J RETURN
      LET $k := $i + $j RETURN
                IF ($k > 2) THEN RETURN ($i, $j)
                ELSE ( )
```

XML and Databases

- Need to handle XML that:
  - may be strongly typed governed by XML Schema;
  - may be strongly typed governed by another schema language, such as a DTD;
  - may be governed by multiple schemas or one schema may be subject to frequent change;
  - may be schema-less;
  - may contain marked-up text with logical units of text (such as sentences) that span multiple elements;
  - has structure, ordering, and whitespace that may be significant;
  - may be subject to update as well as queries based on context and relevancy.

XML and Databases

- Four general approaches to storing an XML document in RDB:
  - store the XML as the value of some attribute within a tuple;
  - store the XML in a *shredded* form across a number of attributes and relations;
  - store the XML in a schema independent form;
  - store the XML in a parsed form; i.e., convert the XML to internal format, such as an Infoset and store this representation.

Storing XML in an Attribute

- In past the XML would have been stored in an attribute whose data type was CLOB.
- More recently, some systems have a new native XML data type (e.g. XML or XMLType).
- Raw XML stored in serialized form, which makes it efficient to insert documents into database and retrieve them in their original form.
- Relatively easy to apply full-text indexing to documents for contextual and relevance retrieval. However, question about performance of general queries and indexing, which may require parsing on-the-fly.
- Also, updates usually require entire XML document to be replaced with a new document.

Storing XML in Shredded Form

- XML decomposed (shredded) into its constituent elements and data distributed over number of attributes in one or more relations.
- Storing shredded documents may make it easier to index values of some elements, provided these elements are placed into their own attributes.
- Also possible to add some additional data relating to hierarchical nature of the XML, making it possible to recompose original structure and ordering, and to allow the XML to be updated.
- With this approach also have to create an appropriate database structure.

Schema-Independent Representation

- Could use DOM to represent structure of XML data.
- Since XML is a tree structure, each node may have only one parent. The rootID attribute allows a query on a particular node to be linked back to its document node.
- While this is schema independent, recursive nature of structure can cause performance problems when searching for specific paths.
- To overcome this, create denormalized index containing combinations of path expressions and a link to node and parent node.

XML and SQL

- SQL:2003 has extensions to enable publication of XML (commonly referred to as SQL/XML):
  - new native XML data type, XML, which allows XML documents to be treated as relational values in columns of tables, attributes in user-defined types, variables, and parameters to functions;
  - set of operators for the type;
  - implicit set of mappings from relational data to XML.

- Standard does not define any rules for the inverse process; i.e., shredding XML data into an SQL form, with some minor exceptions.

Creating Table using XML Type

```
CREATE TABLE XMLStaff (
        docNo CHAR(4), docDate DATE, staffData XML,
        PRIMARY KEY docNo);

INSERT INTO XMLStaff VALUES ('D001',    DATE'2011-04-01',
        XML('<STAFF branchNo = "B005">
                <STAFFNO>SL21</STAFFNO>
                <POSITION>Manager</POSITION>
                <DOB>1979-07-07</DOB>
                <SALARY>30000</SALARY> </STAFF>') );
```

SQL/XML Operators

- XMLELEMENT, to generate an XML value with a single element as a child of its root item. Element can have attributes specified via XMLATTRIBUTES subclause.
- XMLFOREST, to generate an XML value with a list of elements as children of a root item.
- XMLCONCAT, to concatenate a list of XML values.
- XMLPARSE, to perform a non-validating parse of a character string to produce an XML value.
- XMLROOT, to create an XML value by modifying the properties of the root item of another XML value.
- XMLCOMMENT, to generate an XML comment.
- XMLPI, to generate an XML processing instruction.

SQL/XML Functions

- XMLSERIALIZE, to generate a character or binary string from an XML value;
- XMLAGG, an aggregate function, to generate a forest of elements from a collection of elements.

Using XML Operators

List all staff with salary > RM20,000, as an XML element containing name and branch number as an attribute.

```
SELECT staffNo, XMLELEMENT (NAME “STAFF”,
                    fName || ‘ ’ || lName,
                    XMLATTRIBUTES (branchNo AS
      “branchNumber”) ) AS “staffXMLCol”
FROM Staff
WHERE salary > 20000;
```

Using XML Operators

For each branch, list names of all staff with each one represented as an XML element.

```
SELECT XMLELEMENT (NAME “BRANCH”,
    XMLATTRIBUTES (branchNo AS “branchNumber”),
    XMLAGG (
      XMLELEMENT (NAME “STAFF”,
                  fName || ‘ ’ || lName)
      ORDER BY fName || ‘ ’ || lName
                        )
      ) AS “branchXMLCol”
FROM Staff
GROUP BY branchNo;
```

SQL/XML Mapping Functions

- SQL/XML also defines mapping from tables to XML documents.
- Mapping may take as its source an individual table, all tables in a schema, or all tables in a catalog.
- Standard does not specify syntax for the mapping; instead it is provided for use by applications and as a reference for other standards.
- Mapping produces two XML documents: one that contains mapped table data and other that contains an XML Schema describing the first.

Mapping SQL Identifiers to XML Names

- Number of issues had to be addressed to map SQL identifiers to XML Names:
  - range of characters that can be used within an SQL identifier larger than range for an XML Name;
  - SQL delimited identifiers (identifiers within double-quotes), permit arbitrary characters to be used at any point in identifier;
  - XML Names that begin with 'XML' are reserved;
  - XML namespaces use ':' to separate namespace prefix from local component.

- Resolved using *escape notation* that changes unacceptable characters in XML Names into sequence of allowable characters based on Unicode values ("_xHHHH_").

Mapping SQL Data Types to XML Schema

- SQL/XML maps each SQL data type to closest match in XML Schema, in some cases using facets to restrict acceptable XML values to achieve closest match.

- For example:
  - SMALLINT mapped to a restriction of xsd:integer with minInclusive and maxInclusive facets set.
  - CHAR mapped to restriction of xsd:string with facet length set.
  - DECIMAL mapped to xsd:decimal with precision and scale set.

Mapping Tables to XML Documents

- Create root element named after table with <row> element for each row.
- Each row contains a sequence of column elements, each named after corresponding column.
- Each column element contains a data value.
- Names of table and column elements are generated using fully escaped mapping from SQL identifiers to XML Names.
- Must also specify how nulls are to be mapped, using ‘absent’ (column with null would be omitted) or ‘nil’.

Generating an XML Schema

- Generated by creating globally-named XML Schema data types for every type required to describe tables(s) being mapped.
- Naming convention uses suffix containing length or precision/scale to name of the base type (e.g. CHAR(10) would be CHAR_10).
- Next, named XML Schema type is created for types of the rows in table (name used is ‘RowType’ concatenated with catalog, schema, and table name).
- Named XML Schema type is created for type of the table itself (name used is ‘TableType’ concatenated with catalog, schema, and table name).
- Finally, an element is created for table based on this new table type.

Native XML Databases

- Defines (logical) data model for an XML document (as opposed to data in that document) and stores/retrieves documents according to that model.
- At a minimum, model must include elements, attributes, PCDATA, and document order.
- XML document must be unit of (logical) storage although not restricted by any underlying physical storage model (so traditional DBMSs not ruled out nor proprietary storage formats such as indexed, compressed files).

Native XML Databases

- Two types:
  - *text-based*, which stores XML as text, e.g. as a file in file system or as a CLOB in an RDBMS;
  - *model-based*, which stores XML in some internal tree representation, e.g., an Infoset, PSVI, or representation, possibly with tags tokenized.