Unit 5

Database Related Standards

By

Bhupendra Singh Saud

Database related standards

SQL standards

The language SQL is standardized by international standard bodies such as ISO and ANSI. By using standard SQL it should be easier to move applications between different database systems without the need to rewrite a substantial amount of code. Using standard SQL does not give any warranty though as all vendors does not implement all features in the standard.

SQL provides statements for a variety of tasks, including:

- Querying data
- Inserting, updating, and deleting rows in a table
- Creating, replacing, altering, and dropping objects
- Controlling access to the database and its objects
- Guaranteeing database consistency and integrity

Mimer Information Technology's policy is to develop Mimer SQL as far as possible in accordance with the established standard. This enables users to switch to and from Mimer SQL easily. The standard for SQL is ISO/IEC 9075:1999, referred to here as SQL-99.

The SQL standard has gone through a number of revisions

Year	**Name**	**Alias**	Comments
1986	SQL-86	SQL-87	First published by ANSI. Ratified by ISO in 1987.
1989	SQL-89		Minor revision.
1992	SQL-92	SQL2	Major revision.
1999	SQL:1999	SQL3	Added object-oriented features, introduced OLAP,…
2003	SQL:2003		Introduced XML-related features,…

SQL 1999

- Intended as a major enhancement.
- Characterized as "object-oriented SQL".
- In addition to the object oriented extensions, there are some other new features like; Triggers, Stored procedures and user-defined functions, Recursive queries, OLAP, SQL procedural constructs, Expressions in ORDER BY Savepoints, Update through unions and joins.
- The new features are divided into five category: new data types, new predicates, enhanced semantics, additional security, and active database.
- SQL:1999 has four new data types:
    a. Large Object (LOB) type
        i. CHARACTER LARGE OBJECT (CLOB)
        ii. BINARY LARGE OBJECT (BLOB)
    b. Boolean type
    c. Two new *composite* types: ARRAY (storing collections of values in a column) and ROW (storing structured values in single columns of the database)
    d. Distinct types
-

SQL: 1999 is much more than merely SQL-92 plus object technology. It involves additional features that we consider to fall into SQL's relational heritage, as well as a total restructuring of the standards documents themselves with an eye towards more effective standards progression in the future. The features of SQL: 1999 can be partitioned into its "relational features" and its "Object-oriented features". Although we call this category of features "relational", we'll quickly recognize that it's more appropriately categorized as "features that relate to SQL's traditional role and data model" somewhat less pithy phrase. The features here are not strictly limited to the relational model, but are also unrelated to object orientation. These features are often divided into

about five groups: new data types, new predicates, enhanced semantics, additional security, and active database.

New Data Types SQL: 1999 has four new data types (although one of them has some identifiable variants). The first of these types is the LARGE OBJECT, or LOB, type. This is the type with variants: CHARACTER LARGE OBJECT (CLOB) and BINARY LARGE OBJECT (BLOB). CLOBs behave a lot like character strings, but have restrictions that disallow their use in PRIMARY KEYs or UNIQUE predicates, in FOREIGN KEYs, and in comparisons other than purely equality or inequality tests.

BLOBs have similar restrictions. (By implication, LOBs cannot be used in GROUP BY or ORDER BY clauses, either.) Applications would typically not transfer entire LOB values to and from the database (after initial storage, that is), but would manipulate LOB values through a special client-side type called a LOB locator. In SQL:1999, a locator is a unique binary value that acts as a sort of surrogate for a value held within the database; locators can be used in operations (such as SUBSTRING) without the overhead of transferring an entire LOB value across the client-server interface.

Another new data type is the BOOLEAN type, which allows SQL to directly record truth values true, false, and unknown. Complex combinations of predicates can also be expressed in ways that are somewhat more user-friendly than the usual sort of restructuring might make them

```
WHERE COL1 > COL2 AND
    COL3 = COL4 OR
UNIQUE (COL6) IS NOT FALSE
```

SQL: 1999 also has two new composite types: ARRAY and ROW. The ARRAY type allows one to store collections of values directly in a column of a database table. For examples WEEKDAYS VARCHAR (10) ARRAY[7]

New Predicates

SQL: 1999 has three new predicates, one of which we'll consider along with the object-oriented features. The other two are the SIMILAR predicate and the DISTINCT predicate. Since the first version of the SQL standard, character string searching has been limited to very simple comparisons (like =, >, or <>) and the rather rudimentary pattern matching capabilities of the LIKE predicate:

```
WHERE NAME LIKE '%SMIT_'
```

SQL 2003

- Makes revisions to all parts of SQL: 1999.
- Adds a brand new part: SQL/XML (XML-Related Specifications).
- New features are categorized as:

- New data types,
- Enhancements to SQL-invoked routines,
- Extensions to CREATE TABLE statement,
- A new MERGE statement,
- A new schema object - the sequence generator,
- Two new sorts of columns – identity columns and generated columns.

- Retains all data types that existed in SQL: 1999 with the exception of the BIT and BIT VARYING data types.
- Introduces three new data types:
  - BIGINT
  - MULTISET
  - XML
- SQL-invoked function that returns a "table".
  - Table functions give increased functionality by allowing sets of tuples from any external data sources to be invoked (as if they were a table).
  - Table function execution can be parallelized giving improved speed and scalability.
- In addition to the three statements for updating the database, (INSERT, UPDATE, and DELETE) SQL: 2003 adds MERGE.
- Combining INSERT and UPDATE into MERGE.
- Transferring a set of rows from a "transaction table" to a "master table" maintained by the database.
- Mechanism for generating unique values automatically.
- User can define minimum value, a maximum value, a start value, an increment, and a cycle option for the sequence generator they are creating.

```
CREATE SEQUENCE PARTSEQ AS INTEGER
START WITH 1
INCREMENT BY 1
MINVALUE 1
MAXVALUE 10000
NO CYCLE
```

ODMG 3.0 (Object data management group)

ODMG 3.0 was developed by the Object Data Management Group (ODMG). The ODMG is a consortium of vendors and interested parties that work on specifications for object database and object-relational mapping products.

ODMG 3.0 is a portability specification. It is designed to allow for portable applications that could run on more than one product. ODMG 3.0 uses the Java, C++, and Smalltalk languages as much as possible, to allow for the transparent integration of object programming languages.

The ODMG object model is the data model upon which the object definition language (ODL) and object query language (OQL) are based. In fact, this object model provides the data types, type constructors, and other concepts that can be utilized in the ODL to specify object database schemas. Hence, it is meant to provide a standard data model for object-oriented databases, just as the SQL report describes a standard data model for relational databases. It also provides a standard terminology in a field where the same terms were sometimes used to describe different concepts.

Major components of ODMG 3.0 are described below;

Objects and Literals

Objects and literals are the basic building blocks of the object model. The main difference between the two is that an object has both an object identifier and a state (or current value), whereas a literal has only a value but no object identifier. In either case, the value can have a complex structure. The object state can change over time by modifying the object value. A literal is basically a constant value, possibly having a complex structure that does not change.

An object is described by four characteristics: (1) identifier, (2) name, (3) lifetime, and (4) structure. The object identifier is a unique system-wide identifier (or Object_Id . Every object must have an object identifier. In addition to the Object_Id, some objects may optionally be given a unique name within a particular database—this name can be used to refer to the object in a program, and the system should be able to locate the object given that name. Obviously, not all individual objects will have unique names. Typically, a few objects, mainly those that hold collections of objects of a particular object type—such as extents—will have a name. These names are used as entry points to the database; that is, by locating these objects by their unique name, the user can then locate other objects that are referenced from these objects. Other important objects in the application may also have unique names. All such names within a particular database must be unique.

The lifetime of an object specifies whether it is a persistent object (that is, a database object) or transient object (that is, an object in an executing program that disappears after the program terminates). Finally, the structure of an object specifies how the object is constructed by using the type constructors. The structure specifies whether an object is atomic or a collection object.. The

term atomic object is different than the way we defined the atom constructor. It is quite different from an atomic literal.

In the object model, a literal is a value that does not have an object identifier. However, the value may have a simple or complex structure. There are three types of literals: (1) atomic, (2) collection, and (3) structured. Atomic literals correspond to the values of basic data types and are predefined.

The basic data types of the object model include long, short, and unsigned integer numbers (these are specified by the keywords Long, Short, Unsigned Long, Unsigned Short in ODL), regular and double precision floating point numbers (Float, Double), boolean values (Boolean), single characters (Char), character strings (String), and enumeration types (Enum), among others. Structured literals correspond roughly to values that are constructed using the tuple constructor. They include Date, Interval, Time, and Timestamp as built-in structures, as well as any additional user-defined type structures as needed by each application.

User-defined structures are created using the Struct keyword in ODL, as in the C and C++ programming languages. Collection literals specify a value that is a collection of objects or values but the collection itself does not have an Object_Id. The collections in the object model are Set<t>, Bag<t>, List<t>, and Array<t>, where t is the type of objects or values in the collection. Another collection type is Dictionary <k,v>, which is a collection of associations <k,v> where each k is a key (a unique search value) associated with a value v; this can be used to create an index on a collection of values.

Object Definition Language

The concepts of ODMG 2.0 object model can be utilized to create an object database schema using the object definition language ODL. The ODL is designed to support the semantic constructs of the ODMG 2.0 object model and is independent of any particular programming language. Its main use is to create object specifications—that is, classes and interfaces. Hence, ODL is not a full programming language. A user can specify a database schema in ODL independently of any programming language, then use the specific language bindings to specify how ODL constructs can be mapped to constructs in specific programming languages, such as C++, SMALLTALK, and JAVA.

Figure of EER shows one possible set of ODL class definitions for the UNIVERSITY database. In general, there may be several possible mappings from an object schema diagram (or EER schema diagram) into ODL classes.

Figure EER shows the straightforward way of mapping part of the UNIVERSITY database. Entity types are mapped into ODL classes, and inheritance is done using EXTENDS. However, there is no direct way to map categories (union types) or to do multiple inheritance, the classes Person, Faculty, Student, and GradStudent have the extents persons, faculty, students, and

grad_students, respectively. Both Faculty and Student EXTENDS Person, and GradStudent EXTENDS Student. Hence, the collection of students (and the collection of faculty) will be constrained to be a subset of the collection of persons at any point in time. Similarly, the collection of grad_students will be a subset of students. At the same time, individual Student and Faculty objects will inherit the properties (attributes and relationships) and operations of Person, and individual GradStudent objects will inherit those of Student.

The classes Department, Course, Section, and CurrSection are straightforward mappings of the corresponding entity types. However, the class Grade requires some explanation. The Grade class corresponds to the M:N relationship between Student and Section. The reason it was made into a separate class (rather than as a pair of inverse relationships) is because it includes the relationship attribute grade. Hence, the M:N relationship is mapped to the class Grade, and a pair of 1:N relationships, one between Student and Grade and the other between Section and Grade. These two relationships are represented by the following relationship properties: completed_sections of Student; section and student of Grade; and students of Section Finally, the class Degree is used to represent the composite, multivalued attribute degrees of GradStudent.

Object Query Language

The object query language (OQL) is the query language proposed for the ODMG object model. It is designed to work closely with the programming languages for which an ODMG binding is defined, such as C++, SMALLTALK, and JAVA. Hence, an OQL query embedded into one of these programming languages can return objects that match the type system of that language. In addition, the implementations of class operations in an ODMG schema can have their code written in these programming languages. The OQL syntax for queries is similar to the syntax of the relational standard query language SQL, with additional features for ODMG concepts, such as object identity, complex objects, operations, inheritance, polymorphism, and relationships.

Simple OQL Queries, Database Entry Points, and Iterator Variables

The basic OQL syntax is a select . . . from . . . where. . Structure, as for SQL. For example, the query to retrieve the names of all departments in the college of ‘Engineering’ can be written as follows:

```
Q1: SELECT d.dname
        FROM d IN Departmant
        WHERE d.college = ”Engineering”;
```

In general, an **entry point** to the database is needed for each query, which can be any *named persistent object.* For many queries, the entry point is the name of the **extent** of a class. Recall that the extent name is considered to be the name of a persistent object whose type is a collection (in most cases, a set) of objects from the class. the named object departments is of type set<Department>; persons is of type set<Person>; faculty is of type set<Faculty>; and so on.

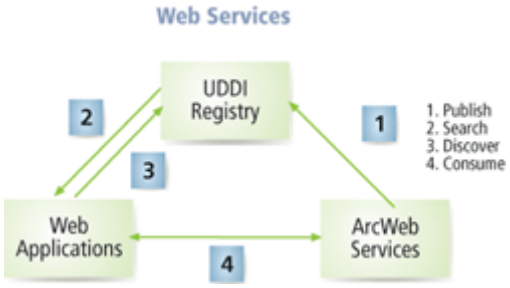
The use of an extent name—departments in Q0—as an entry point refers to a persistent collection of objects. Whenever a collection is referenced in an OQL query, we should define an **iterator variable.** In Q0—that ranges over each object in the collection. In many cases, as in Q0, the query will select certain objects from the collection, based on the conditions specified in the where-clause. In Q0, only persistent objects d in the collection of departments that satisfy the condition d.college = 'Engineering' are selected for the query result. For each selected object d, the value of d.dname is retrieved in the query result. Hence, the *type of the result* for Q0 is bag<string>, because the type of each dname value is string (even though the actual result is a set because dname is a key attribute). In general, the result of a query would be of type *bag* for **select** . . . **from** . . . and of type *set* for **select distinct** . . . **from** . . ., as in SQL (adding the keyword distinct eliminates duplicates).

Query Results and Path Expressions

The result of a query can in general be of any type that can be expressed in the ODMG object model. A query does not have to follow the **select . . . from . . . where** . . . structure; in the simplest case, any persistent name on its own is a query, whose result is a reference to that persistent object.

Web Services—A Standards-Based Framework for Integration

Web services are software components that can be accessed over the Web through standards-based protocols such as HTTP or SMTP for use in other applications. They provide a fundamentally new framework and set of standards for a computing environment that can include servers, workstations, desktop clients, and lightweight "pervasive" clients such as phones and PDAs. Web services are not limited to the Internet; they supply a powerful architecture for all types of distributed computing.



Web services standards are the glue that allows computers and devices to interact. UDDI allows clients to discover Web services.

Web services standards are the glue that allows computers and devices to interact, forming a greater computing whole that can be accessed from any device on the network.

In Web services, computing nodes have three roles—client, service, and broker.

- A client is any computer that accesses functions from one or more other computing nodes on the network. Typical clients include desktop computers, Web browsers, Java applets, and mobile devices. A client process makes a request for a computing service and receives results for that request.

- A service is a computing process that receives and responds to requests and returns a set of results.
- A broker is essentially a service metadata portal for registering and discovering services. Any network client can search the portal for an appropriate service.

Because Web services can support the integration of information and services that are maintained on a distributed network, they are appealing to local governments and other organizations that have departments that independently collect and manage spatial data but must integrate these datasets.

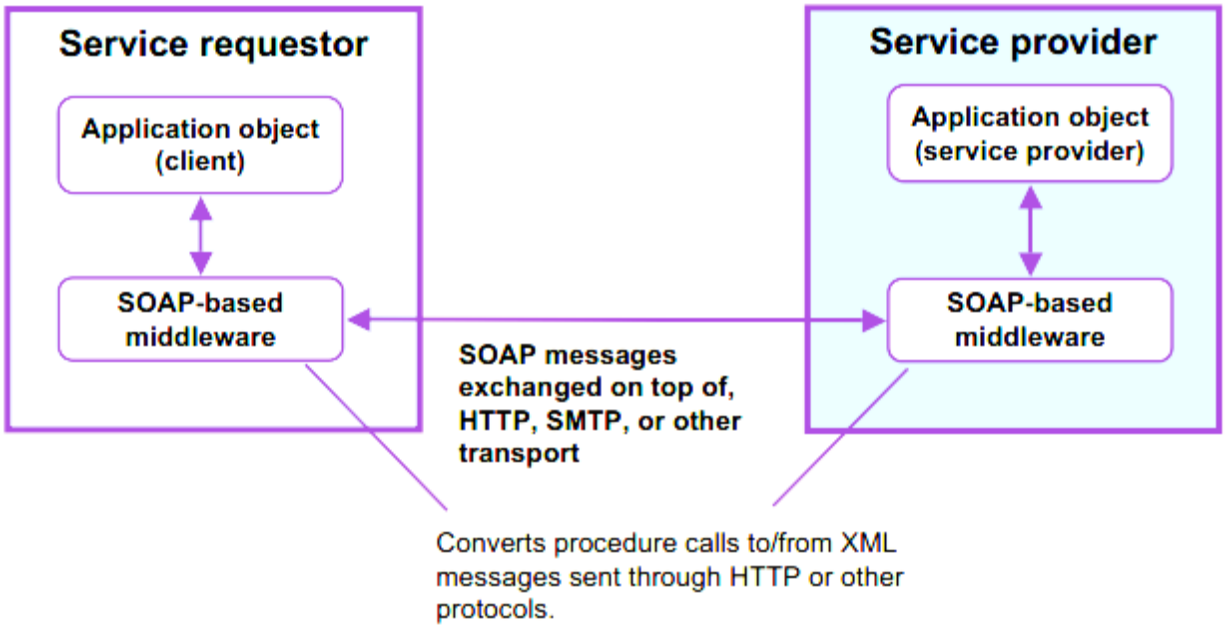
The use of a connecting technology (Web services) coupled with an integrating technology (GIS) can efficiently support this requirement. Various layers of information can be dynamically queried and integrated but will still be maintained independently in a distributed computing environment. Esri's Web services technology, ArcWeb Services, is built on top of ArcIMS. ArcWeb Services leverage core business logic in ArcGIS and support Internet-based distributed computing.

A series of protocols—eXtensible Markup Language (XML); Simple Object Access Protocol (SOAP); Web Service Description Language (WSDL); and Universal Description, Discovery, and Integration (UDDI)—provides the key standards for Web services and supports sophisticated communications between various nodes on a network. These protocols enable smarter communication and collaborative processing among nodes built within any Web services-compliant architecture.

Simple Object Access Protocol (SOAP)

SOAP (originally **Simple Object Access Protocol**) is a protocol specification for exchanging structured information in the implementation of web services in computer networks. Its purpose is to induce extensibility, neutrality and independence. It uses XML Information Set for its message format, and relies on application layer protocols, most often Hypertext Transfer Protocol (HTTP) or Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission.

SOAP allows processes running on disparate operating systems (such as Windows and Linux) to communicate using Extensible Markup Language (XML). Since Web protocols like HTTP are installed and running on all operating systems, SOAP allows clients to invoke web services and receive responses independent of language and platforms.



SOAP covers the following four main areas:

- **A message format** for one-way communication describing how a message can be packed into an XML document.
- **A description** of how a SOAP message should be transported using HTTP (for Web-based interaction) or SMTP (for e-mail-based interaction).
- **A set of rules** that must be followed when processing a SOAP message and a simple classification of the entities involved in processing a SOAP message.
- **A set of conventions** on how to turn an RPC call into a SOAP message and back.

What is SOAP?

- SOAP stands for Simple Object Access Protocol
- SOAP is a communication protocol
- SOAP is for communication between applications
- SOAP is a format for sending messages
- SOAP communicates via Internet
- SOAP is platform independent
- SOAP is language independent
- SOAP is based on XML
- SOAP is simple and extensible
- SOAP allows you to get around firewalls
- SOAP is a W3C recommendation

Why SOAP?

- It is important for application development to allow Internet communication between programs. Today's applications communicate using Remote Procedure Calls (RPC) between objects like DCOM and CORBA, but HTTP was not designed for this. RPC represents a compatibility and security problem; firewalls and proxy servers will normally block this kind of traffic.
- A better way to communicate between applications is over HTTP, because HTTP is supported by all Internet browsers and servers. SOAP was created to accomplish this.
- SOAP provides a way to communicate between applications running on different operating systems, with different technologies and programming languages.

SOAP Building Blocks

A SOAP message is an ordinary XML document containing the following elements:

- An Envelope element that identifies the XML document as a SOAP message
- A Header element that contains header information
- A Body element that contains call and response information
- A Fault element containing errors and status information

Syntax Rules

Here are some important syntax rules:

- A SOAP message MUST be encoded using XML
- A SOAP message MUST use the SOAP Envelope namespace
- A SOAP message MUST use the SOAP Encoding namespace
- A SOAP message must NOT contain a DTD reference
- A SOAP message must NOT contain XML Processing Instructions

The SOAP Envelope Element

The required SOAP Envelope element is the root element of a SOAP message. This element defines the XML document as a SOAP message.

Example

```
<? xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  ...
Message information goes here
```

...
</soap: Envelope>

XML (Extensible Markup Language)

What is XML?

XML stands for Extensible Markup Language. A markup language specifies the structure and content of a document because it is extensible, XML can be used to create a wide variety of document types. XML is a subset of a Standard Generalized Markup Language (SGML) which was introduced in the 1980s. SGML is very complex and can be costly. Hypertext Markup Language (HTML) is a more easily used markup language. XML can be seen as sitting between SGML and HTML – easier to learn than SGML, but more robust than HTML.

Main features of XML:

- XML is a markup language much like HTML
- XML was designed to carry data, not to display data
- XML tags are not predefined. You must define your own tags
- XML is designed to be self-descriptive
- XML files are text files, which can be managed by any text editor.
- XML is very simple, because it has less than 10 syntax rules.
- XML is extensible, because it only specifies the structural rules of tags. No specification on tags them self.

Example of document structure

```
<bank>
 <account>
 <account-number> A-101     </account-number>
 <branch-name>      Downtown </branch-name>
 <balance>            500        </balance>
 </account>
 <Depositor>
 <account-number> A-101    </account-number>
 <customer-name> Johnson </customer-name>
 </depositor>
</bank>
```

Comparison of XML with Relational Data

- **Inefficient:** tags, which in effect represent schema information, are repeated
- **Better than relational tuples as a data-exchange format**
  - ✓ Unlike relational tuples, XML data is self-documenting due to presence of tags

- ✓ Non-rigid format: tags can be added
- ✓ Allows nested structures
- ✓ Wide acceptance, not only in database systems, but also in browsers, tools, and applications

- **The data is *self-describing:*** e.g. the meaning of the data is included: identifiers surround every bit of data, indicating what it means
- **Far more *flexible* method of representing transmitted information**
- **Open, standard technologies for moving, processing and validating the data:** e.g. the XML parser can automatically parse, validate, and feed the information to an application, instead of every application having to include this functionality

XML Declaration

The XML document can optionally have an XML declaration. It is written as follows:

<? xml version="1.0" encoding="UTF-8"?>

Where version is the XML version and encoding specifies the character encoding used in the document.

Syntax Rules for XML Declaration

- The XML declaration is case sensitive and must begin with "<? xml>" where "xml" is written in lower-case.
- If the document contains XML declaration, then it strictly needs to be the first statement of the XML document.
- The XML declaration strictly needs be the first statement in the XML document.
- An HTTP protocol can override the value of encoding that you put in the XML declaration.

What is an XML Element?

An XML element is everything from (including) the element's start tag to (including) the element's end tag.

<price>29.99</price>

An element can contain:

- text
- attributes
- other elements
- or a mix of the above

Example:

```
<bookstore>
  <book category="children">
    <title>Harry Potter</title>
```

```
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
 </book>
 <book category="web">
  <title>Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price>39.95</price>
 </book>
</bookstore>
```

In the example above:
<title>, <author>, <year>, and <price> have text content because they contain text (like 29.99).
<bookstore> and <book> have element contents, because they contain elements.
<book> has an attribute (category="children").

Empty XML Elements
An element with no content is said to be empty.
In XML, you can indicate an empty element like this:
<element></element>

You can also use a so called self-closing tag:
<element />

Tags and Elements

An XML file is structured by several XML-elements, also called XML-nodes or XML-tags. The names of XML-elements are enclosed in triangular brackets < > as shown below:

```
<element>
```

Syntax Rules for Tags and Elements
Element Syntax: Each XML-element needs to be closed either with start or with end elements as shown below:
<element>....</element>
or in simple-cases, just this way:
<element/>

XML Attributes

An attribute specifies a single property for the element, using a name/value pair. An XML- element can have one or more attributes. For example:

```
<a href="http://www.tutorialspoint.com/">Tutorialspoint!</a>
```

Here href is the attribute name and http://www.tutorialspoint.com/ is attribute value.

Syntax Rules for XML Attributes

- Attribute names in XML (unlike HTML) are case sensitive. That is, HREF and href are considered two different XML attributes.
- Same attribute cannot have two values in a syntax. The following example shows incorrect syntax because the attribute b is specified twice:

```
<a b="x" c="y" b="z">....</a>
```

- Attribute names are defined without quotation marks, whereas attribute values must always appear in quotation marks. Following example demonstrates incorrect xml syntax:

```
<a b=x>....</a>
```

In the above syntax, the attribute value is not defined in quotation marks.

CDATA

The term CDATA means, Character Data. CDATA is defined as blocks of text that are not parsed by the parser, but are otherwise recognized as markup.

The predefined entities such as <, >, and & require typing and are generally difficult to read in the markup. In such cases, CDATA section can be used. By using CDATA section, you are commanding the parser that the particular section of the document contains no markup and should be treated as regular text.

Syntax

Following is the syntax for CDATA section:

```
<![CDATA[
    characters with markup
]]>
```

The above syntax is composed of three sections:

- **CDATA Start section:** CDATA begins with the nine-character delimiter <![CDATA[
- **CDATA End section:** CDATA section ends with]]> delimiter.
- **CData section:** Characters between these two enclosures are interpreted as characters, and not as markup. This section may contain markup characters (<, >, and &), but they are ignored by the XML processor.

Example

The following markup code shows an example of CDATA. Here, each character written inside the CDATA section is ignored by the parser.

```
<script>
<![CDATA[
        <message> Welcome to NSC </message>
]] >
</script >
```

In the above syntax, everything between <message> and </message> is treated as character data and not as markup

CDATA Rules

The given rules are required to be followed for XML CDATA:

- CDATA cannot contain the string "]]>" anywhere in the XML document.
- Nesting is not allowed in CDATA section

XML Encoding

Encoding is the process of converting Unicode characters into their equivalent binary representation. When the XML processor reads an XML document, it encodes the document depending on the type of encoding. Hence, we need to specify the type of encoding in the XML declaration.

Encoding Types

There are mainly two types of encoding:

- UTF-8
- UTF-16

UTF stands for UCS Transformation Format, and UCS itself means Universal Character Set. The number 8 or 16 refers to the number of bits used to represent a character. They are either 8 (one byte) or 16 (two bytes). For the documents without encoding information, UTF-8 is set by default.

Syntax

Encoding type is included in the prolog section of the XML document. The syntax for UTF-8 encoding is as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
```

The syntax for UTF-16 encoding is as follows:

```
<?xml version="1.0" encoding="UTF-16" standalone="no" ?>
```

Example

Following example shows the declaration of encoding:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<contact-info>
        <name>College</name>
        <company>NSC</company>
        <phone>01-456543</phone>
</contact-info>
```

In the above example encoding="UTF-8", specifies that 8-bits are used to represent the characters. To represent 16-bit characters, UTF-16 encoding can be used. The XML files encoded with UTF-8 tend to be smaller in size than those encoded with UTF-
16 format

Nesting of element

Nesting of data is useful in data transfer

Example: elements representing customer-id, customer name, and address nested within an order element

Nesting is not supported, or discouraged, in relational databases

With multiple orders, customer name and address are stored redundantly

normalization replaces nested structures in each order by foreign key into table storing customer name and address information

Nesting is supported in object-relational databases

But nesting is appropriate when transferring data

External application does not have direct access to data referenced by a foreign key

Example of Nested Elements

```
<bank-1>
    <customer>
            <customer-name> Hayes </customer-name>
            <customer-street> Main </customer-street>
            <customer-city>    Harrison </customer-city>
            <account>
           <account-number> A-102 </account-number>
           <branch-name>     Perryridge </branch-name>
           <balance>            400 </balance>
            </account>
```

```
        ……
        ……..
          </customer>
    </bank-1>
```

XML Comment

XML comments are similar to HTML comments. The comments are added as notes or lines for understanding the purpose of an XML code.

Comments can be used to include related links, information, and terms. They are visible only in the source code; not in the XML code. Comments may appear anywhere in XML code.

Syntax

XML comment has the following syntax:

```
<!-------Your comment----->
```

A comment starts with <!-- and ends with -->. You can add textual notes as comments between the characters. You must not nest one comment inside the other.

Example: Following example demonstrates the use of comments in XML document:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!---Students grades are uploaded by months---->
<class_list>
    <student>
        <name>Tanmay</name>
        <grade>A</grade>
    </student>
</class_list>
```

Any text between <!-- and --> characters is considered as a comment.

XML Document

An XML document is a basic unit of XML information composed of elements and other markup in an orderly package. An XML document can contain a wide variety of data. For example, database of numbers, numbers representing molecular structure or a mathematical equation.

XML Document Example

A simple document is shown in the following example:

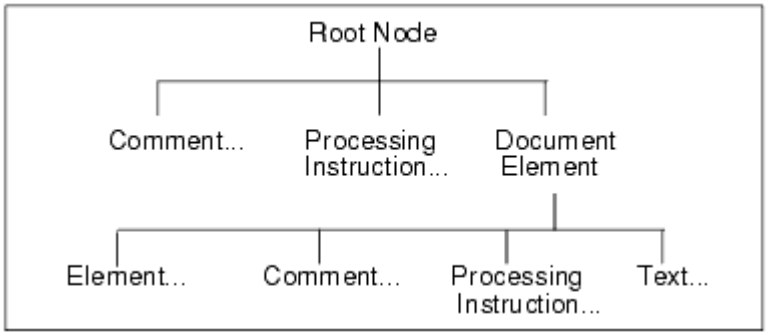
```
<?xml version="1.0"?>
<contact-info>
    <name>NSC</name>
    <company>Education_company</company>
    <phone>01-449839</phone>
```

</contact-info>

The following image depicts the parts of XML document.

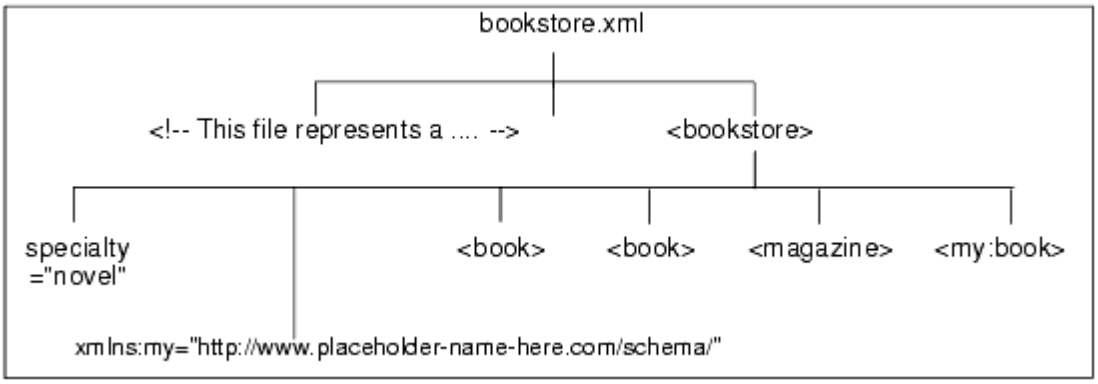
XML Document Structure:

The XPath processor operates on a tree representation of XML data that looks like the following figure:



The root node has no actual text associated with it. You can think of the file name as the root node. A document can include zero or more comments and zero or more processing instructions.

A document element is required, and there can be only one. The document element contains all elements in the document. For example:



In the preceding figure, bookstore.xml is the name of a file that contains XML data. There is a comment near the beginning of the document that starts with "This file represents a ..." The document element is bookstore. The immediate children of bookstore include an attribute, a namespace declaration (not supported by Stylus Studio), three book elements (one is in the my namespace), and a magazine element. The book and magazine elements contain elements and attributes

An XML Document

A complete well-formed XML document may look like:

<?xml version="1.0"?>

```
<bank>
        <account>
        <account-number> A-101     </account-number>
        <branch-name>      Downtown </branch-name>
        <balance>            500        </balance>
        </account>
        <Depositor>
        <account-number> A-101    </account-number>
        <customer-name> Johnson </customer-name>
        </depositor>
    </bank>
```

The Bank element, above, is the root element.

XML Document Type Declaration (DTDs)

The XML Document Type Declaration, commonly known as DTD, is a way to describe XML language precisely. DTDs check vocabulary and validity of the structure of XML documents against grammatical rules of appropriate XML language.

An XML DTD can be either specified inside the document, or it can be kept in a separate document and then liked separately.

Syntax

Basic syntax of a DTD is as follows:

```
<! DOCTYPE element DTD identifier
[
   declaration1
   declaration2
   ........
]>
```

In the above syntax,

- The DTD starts with <! DOCTYPE delimiter.
- An element tells the parser to parse the document from the specified root element.
- DTD identifier is an identifier for the document type definition, which may be the path to a file on the system or URL to a file on the internet. If the DTD is pointing to external path, it is called External Subset.
- The square brackets [ ] enclose an optional list of entity declarations called Internal Subset.

Internal DTD

A DTD is referred to as an internal DTD if elements are declared within the XML files. To refer it as internal DTD, standalone attribute in XML declaration must be set to yes. This means, the declaration works independent of external source.

Syntax

The syntax of internal DTD is as shown:

```
<! DOCTYPE root-element [element-declarations]>
```

Where root-element is the name of root element and element-declarations is where you declare the elements.

Example

Following is a simple example of internal DTD:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
    <!DOCTYPE address [
        <!ELEMENT address (name,company,phone)>
        <!ELEMENT name (#PCDATA)>
        <!ELEMENT company (#PCDATA)>
        <!ELEMENT phone (#PCDATA)>
    ]>
    <address>
        <name>Tanmay Patil</name>
        <company>NDC</company>
        <phone>0146575655</phone>
    </address>
```

Let us go through the above code:

Start Declaration- Begin the XML declaration with following statement

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
```

DTD- Immediately after the XML header, the document type declaration follows, commonly referred to as the DOCTYPE:

```
<!DOCTYPE address [
```

The DOCTYPE declaration has an exclamation mark (!) at the start of the element name. The DOCTYPE informs the parser that a DTD is associated with this XML document.

DTD Body- The DOCTYPE declaration is followed by body of the DTD, where you declare elements, attributes, entities, and notations:

```
<!ELEMENT address (name,company,phone)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT company (#PCDATA)>
```

<!ELEMENT phone_no (#PCDATA)>

Several elements are declared here that make up the vocabulary of the <name> document. <!ELEMENT name (#PCDATA)> defines the element *name* to be of type "#PCDATA". Here #PCDATA means parse-able text data.

End Declaration - Finally, the declaration section of the DTD is closed using a closing bracket and a closing angle bracket (]>). This effectively ends the definition, and thereafter, the XML document follows immediately.

Rules

- The document type declaration must appear at the start of the document (preceded only by the XML header) — it is not permitted anywhere else within the document.
- Similar to the DOCTYPE declaration, the element declarations must start with an exclamation mark.
- The Name in the document type declaration must match the element type of the root element.

External DTD

In external DTD elements are declared outside the XML file. They are accessed by specifying the system attributes which may be either the legal .dtd file or a valid URL. To refer it as external DTD, standalone attribute in the XML declaration must be set as no. This means, declaration includes information from the external source.

Syntax

Following is the syntax for external DTD:

```
<!DOCTYPE root-element SYSTEM "file-name">
```

where file-name is the file with .dtd extension.

Example

The following example shows external DTD usage:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE address SYSTEM "address.dtd">
<address>
    <name>Tanmay Patil</name>
    <company>NSC</company>
    <phone>0167655565</phone>
</address>
```

The content of the DTD file address.dtd are as shown:

```
<!ELEMENT address (name,company,phone)>
<!ELEMENT name (#PCDATA)>
```

<!ELEMENT company (#PCDATA)>
<!ELEMENT phone (#PCDATA)>

XML Schema

XML Schema is commonly known as XML Schema Definition (XSD). It is used to describe and validate the structure and the content of XML data. XML schema defines the elements, attributes and data types. Schema element supports Namespaces. It is similar to a database schema that describes the data in a database.

Syntax

You need to declare a schema in your XML document as follows:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

Example

The following example shows how to use schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="contact">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="name" type="xs:string" />
         <xs:element name="company" type="xs:string" />
         <xs:element name="phone" type="xs:int" />
      </xs:sequence>
   </xs:complexType>
</xs:element>
</xs:schema>
```

The basic idea behind XML Schemas is that they describe the legitimate format that an XML document can take.

Elements

As we saw in the XML - Elements chapter, elements are the building blocks of XML document. An element can be defined within an XSD as follows:

```
<xs:element name="x" type="y"/>
```

Definition Types

You can define XML schema elements in following ways:

Simple Type - Simple type element is used only in the context of the text. Some of predefined simple types are: xs:integer, xs:boolean, xs:string, xs:date. For example:

```
<xs:element name="phone_number" type="xs:int" />
```

Complex Type - A complex type is a container for other element definitions. This allows you to specify which child elements an element can contain and to provide some structure within your XML documents. For example:

```
<xs:element name="Address">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string" />
        <xs:element name="company" type="xs:string" />
      <xs:element name="phone" type="xs:int" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

In the above example, *Address* element consists of child elements. This is a container for other <xs:element> definitions, that allows to build a simple hierarchy of elements in the XML document.

Global Types - With global type, you can define a single type in your document, which can be used by all other references. For example, suppose you want to generalize the *person* and *company* for different addresses of the company. In such case, you can define a general type as below:

```
<xs:element name="AddressType">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string" />
        <xs:element name="company" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Now let us use this type in our example as below:

```
<xs:element name="Address1">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="address" type="AddressType" />
        <xs:element name="phone1" type="xs:int" />
```

```
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="Address2">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="address" type="AddressType" />
            <xs:element name="phone2" type="xs:int" />
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Instead of having to define the name and the company twice (once for *Address1* and once for *Address2*), we now have a single definition. This makes maintenance simpler, i.e., if you decide to add "Postcode" elements to the address, you need to add them at just one place.

X-Query

XQuery is a standardized language for combining documents, databases, Web pages and almost anything else. It is very widely implemented. It is powerful and easy to learn. XQuery is replacing proprietary middleware languages and Web Application development languages. XQuery is replacing complex Java or C++ programs with a few lines of code. XQuery is simpler to work with and easier to maintain than many other alternatives.

Characteristics

- XQuery is the language for querying XML data
- XQuery for XML is like SQL for databases
- XQuery is built on XPath expressions
- XQuery is supported by all major databases

Benefits of XQuery

- Using XQuery, both hierarchical and tabular data can be retrieved.
- XQuery can be used to query tree and graphical structures.
- XQuery can be directly used to query webpages.
- XQuery can be directly used to build webpages.
- XQuery can be used to transform xml documents.
- XQuery is ideal for XML-based databases and object-based databases. Object databases are much more flexible and powerful than purely tabular databases.

Example

Following is a sample XML document containing the records of a bookstore of various books.

books.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book category="JAVA">
    <title lang="en">Learn Java in 24 Hours</title>
    <author>Robert</author>
    <year>2005</year>
    <price>30.00</price>
  </book>

 <book category="DOTNET">
    <title lang="en">Learn .Net in 24 hours</title>
    <author>Peter</author>
    <year>2011</year>
    <price>70.50</price>
  </book>

  <book category="XML">
    <title lang="en">Learn XQuery in 24 hours</title>
    <author>Robert</author>
    <author>Peter</author>
    <year>2013</year>
    <price>50.00</price>
  </book>

  <book category="XML">
    <title lang="en">Learn XPath in 24 hours</title>
    <author>Jay Ban</author>
    <year>2010</year>
    <price>16.50</price>
  </book>
</books>
```

Following is a sample XQuery document containing the query expression to be executed on the above XML document. The purpose is to get the title elements of those XML nodes where the price is greater than 30.

books.xqy

```
for $x in doc("books.xml")/books/book
where $x/price>30
return $x/title
```

Result

```
<title lang="en">Learn .Net in 24 hours</title>
<title lang="en">Learn XQuery in 24 hours</title>
```

XSLT

XSLT (Extensible Style sheet Language Transformations) is a language for transforming XML documents into other XML documents, or other formats such as HTML for web pages, plain text or XSL Formatting Objects, which may subsequently be converted to other formats, such as PDF, PostScript and PNG.

X-path

XPath is a major element in the XSLT standard. **XPath** can be used to navigate through elements and attributes in an XML document. **XPath** is a syntax for defining parts of an XML document. **XPath** uses **path** expressions to navigate in XML documents. **XPath** contains a library of standard functions.

- An XPath expression returns a collection of element nodes that satisfy certain patterns specified in the expression.
- The names in the XPath expression are node names in the XML document tree that are either tag (element) names or attribute names, possibly with additional **qualifier conditions** to further restrict the nodes that satisfy the pattern.
- There are two main **separators** when specifying a path: **single slash (/) and double slash (//).**
- A single slash before a tag specifies that the tag must appear as a direct child of the previous (parent) tag, whereas a double slash specifies that the tag can appear as a descendant of the previous tag *at any level.*

Some examples are given below:

```
1. /company
2. /company/department
3. //employee [employeeSalary gt 70000]/employeeName
4. /company/employee [employeeSalary gt 70000]/employeeName
5. /company/project/projectWorker [hours ge 20.0]
```

XML Applications

- **Storing and exchanging data with complex structures**
  - E.g. Open Document Format (ODF) format standard for storing Open Office and Office Open XML (OOXML) format standard for storing Microsoft Office documents
  - Numerous other standards for a variety of applications
    - ChemML, MathML
- **Standard for data exchange for Web services**
  - Remote method invocation over HTTP protocol
  - More in next slide
- **Data mediation**
  - Common data representation format to bridge different systems