# Title: Mapping SQL types to XML types - an overview

Author: Fred Zemke, Ashok Malhotra, Jim Melton
Source: U.S.A.
Status: Change proposal
Date: February 9, 2001

## Abstract

This paper is an introduction to the candidate base document for SQL/XML. It discusses the three kinds of mappings found there:

- character set mappings
- identifier mappings
- data type mappings

## References

[Foundation CD] Jim Melton (ed), " Committee Draft (CD) Database Language SQL - Part 2: SQL/Foundation", ISO/IEC JTC1/SC32 WG3:PER-004 = ANSI NCITS H2-2000-556

[XML Schema: Datatypes] World Wide Web Consortium, "XML Schema Part 2: Datatypes", available at http://www.w3.org/TR/xmlschema-2/

[XML 1.0 Rec (2nd ed)] World Wide Web Consortium, "Extensible Markup Language (XML) 1.0." Available at: http://www.w3.org/TR/2000/WD-xml-2e-20000814

[XML Namespaces] World Wide Web Consortium, "Namespaces in XML", available at http://www.w3c.org/TR/REC-xml-names

[HEL-026r2] Jim Melton, "Subproject: 'XML-related specifications (SQL/XML)'", ISO/IEC JTC1/SC32 WG3:HEL-026r2 = ANSI NCITS H2-2000-3331r1

[SQL/XML candidate] Fred Zemke, et.al., "SQL/XML candidate base document", ISO/IEC JTC1/SC32 WG3:E3A-003 = H2-2001-009r1

# 1. Introduction

This paper provides an overview of the accompanying candidate base document for SQL/XML. The candidate base document treats an important, though admittedly still incomplete, portion of the problem of mapping SQL to XML. The mappings that are defined in the candidate base document are:

- character set mappings,
- identifier mappings, and
- predefined data type mappings.

These three mappings should be seen as infrastructure that will be required by other components of SQL/XML. As such they are relevant to the first three bullet items in section 5, “Program of work” in the subproject proposal [HEL-026r2]. The three bullet items in question are quoted below, with an explanation of the proposal’s applicability to each:

- Specifications for the representation of SQL data (specifically rows and tables of rows, as well as views and query results) in XML form, and vice versa.

  Applicability: in order to represent SQL data in XML, it will be necessary to map the SQL data values to XML data values. Before such data values mappings can be defined, one must first define the mappings of the SQL data types.

- Specifications associated with mapping SQL schemata to and from XML schemata. This may include performing the mapping between existing arbitrary XML and SQL schemata.

  Applicability: in order to map SQL schemata to XML schemata, it will be necessary to map both SQL identifiers to XML Names and SQL data types to XML Schema data types.

- Specifications for the representation of SQL Schemas in XML.

  Applicability: it is expected that the data type mappings and possibly also the identifier mapping will be relevant to this item.

See section 4, “Example”, of this paper, for detailed examples of how the mappings in the candidate base document might be applied to the problem of creating an XML Schema definition corresponding to an SQL <table definition>. The reader is cautioned, however, that these examples are at this stage still hypothetical, because the candidate base document does not propose mappings for <table definition>s yet.

The candidate base document represents a consensus arrived at by industry representatives from at least eleven different companies, who have met in an informal open forum known as SQLX, publicized within the U.S national body, and convened for the purpose of preparing contributions to SQL/XML.

# 2. Preliminaries

## 2.1 Lexical conventions

Since SQL/XML is a standard bridging two worlds, SQL and XML, it is important to be clear which world each object of discussion inhabits. Since SQL/XML will be a part of the SQL standard, there is a presumption that each object lies in the SQL world unless otherwise noted. Consequently plain text (as opposed to bold) is reserved for SQL objects, while **`bold monospace`** is reserved for literal XML text, and ***`italic bold monospace`*** is used for variables denoting XML text. However, text on a separate line and clearly marked in an accompanying paragraph as being XML is merely set in monosapce font (i.e., without the bold), for example

```
<xsd:whatever/>
```

## 2.2 Namespaces

[XML Namespaces] provides a namespace capability so that users can avoid naming collisions. XML Schema uses two namespaces, denoted in the XML documents as **`xsd:`** and **`xsi:`**. The choice of letters to represent these namespaces is actually arbitrary, so that a user could pick different letters as long as they are defined to refer to the correct URLs. Since the letters chosen to represent these namespaces are arbitrary, the SQL/XML candidate base document defines two variables, ***`xsd`*****`:`** and ***`xsi`*****`:`**, for them, with the italics convention indicating that these are variables and the actual letters used may differ..

The candidate base document also needs to create XML definitions which will reside in a namespace. An XML namespace is a URI. It may be desirable to permit references to a specific file associated with that URI. Our editor accepted an action item to approach ISO on this subject. He asked ISO for their permission to use a domain belonging to them, along with permission to store a file on a directory associated with such a domain, and to use the URI of that file for the XML namespace. ISO has indicated considerable interest in providing the permission, the domain information, and the directory and space for a file. Formal permission has not yet been granted, but is anticipated before the FCD ballot terminates.

While the location of this namespace is not yet known, the candidate base document uses the variable ***`sqlxml`*****`:`** to represent the user-defned letters to denote this namespace.

# 3. Mappings

## 3.1 Character set mappings

XML must be written in Unicode, though the choice of encoding is not dictated.

SQL, on the other hand, does not mandate any particular character set. An implementation may support several character sets, and even if there is only one, it may be different from Unicode.

Consequently it is necessary to assume an implementation-defined mapping of each SQL character set to Unicode.

An important property of a character set mapping is whether fixed-length character strings are mapped to fixed-length character strings; such character set mappings are termed *homomorphic*. (Formally, a character set mapping is homomorphic if for each positive integer *N*, there exists a positive integer *M* such that all strings of length *N* are mapped to strings of length *M*.)

For example, each character of ASCII maps to a single character of Unicode; therefore the length of a character string remains the same when mapped from ASCII to Unicode. Conversion ratios than one-to-one are also conceivable, though the authors are not aware of any. Although we are not aware of any non-homomorphic character set mappings, the candidate base document has been written to allow for the possibility.

It is important to note that the XML entities (escape sequences) such as **`&`** and **`<`** are not obstacles to homomorphism, since each entity represents a single character. The length of an XML character string is determined by counting characters after replacing the entities by their equivalents.

## 3.2 Identifier mappings

It is expected that SQL/XML will eventually include the ability to port or represent SQL metadata in XML. For example, the results of an SQL query might be serialized in XML using either elements or attributes to represent columns of the result. In that case it is natural to expect that the XML element or attribute tags will correspond to the column names in the SQL query.

In the simplest cases, the ones we love to put up in chalk talks, the choice of element or attribute tags will be obvious. For example,

```
SELECT name
FROM emp
```

is actually equivalent to

```
SELECT NAME
FROM EMP
```

so that **`NAME`** is the natural element or attribute tag in XML for the result column.

The general case is not so simple. SQL places no restrictions on the characters that may appear in a <delimited identifier>, whereas XML has some restrictions on XML Names. Consequently, an SQL <identifier>, when mapped to Unicode using the implementation-defined mapping from SQL_TEXT to Unicode, may be invalid as an XML Name. This means that it will be necessary to have a mapping to convert an SQL <identifier> into an XML Name.

In addition, two different scenarios, with slightly different requirements, were recognized:

1. In one scenario, the SQL metadata was created independently of XML, so that there was no (original) intention to use the SQL metadata to generate XML. For example, a table created before the advent of XML might have a column named “Salary: Previous Posi-

tion". In this example, the user did not intend for "Salary" to be a namespace identifier in XML. Consequently it would be wrong to pass the <colon> through to the XML Name. Besides colons, the other problem is names beginning with the letters "xml" in any combination of upper or lower case. All such names are reserved by XML, and it would be dangerous to pass them through as the initial sequence in the XML Name.

2. In the other scenario, the user is crafting the SQL metadata with a conscious intent to create XML. In that case, the user will want the ability to generate namespace prefixes and names beginning with the letters "xml".

To deal with this problem, the candidate base document proposes two variants of the identifier mapping. For the first scenario, the escape variant is called "fully escaped", whereas for the other scenario the escape variant is called "partially escaped".

The precise algorithm, including conditional rules to account for the escape variants, is found in the candidate base document, Subclause 5.1, "Mapping SQL <identifier>s to XML Names". The basic idea in both variants is that any character in the SQL <identifier> that is not valid (or desired, for the fully escaped variant) in an XML Name is mapped to an escape sequence consisting of an underscore, a lower case 'x', four or eight uppercase hex digits, and a final underscore. The hex digits are taken from the Unicode U+*HHHH* representation of the character if it has a UCS-2 representation, and from the U+*HHHHHHHH* representation in UCS-4 otherwise. For example, '@' is not a valid XML Name character. The UCS-2 representation of '@' is U+0040. Consequently the escape sequence used to represent '@' is **_x0040_** .

The fully escaped variant must consider a couple additional cases. First, a <colon> must be escaped to **_x003A_** . (Incidentally, an initial <colon> is escaped even in the partially escaped variant, since initial <colon> does not delimit a valid namespace tag.) Second, if the SQL <identifier> begins with 'xml' (in any combination of upper and lower case), then the XML Name is prefixed with **_xFFFF_** . Note that U+FFFF is the official Unicode "not a character", so that this prefix cannot be generated as the escape of any Unicode character.

An important aspect of the proposal is that a single mapping from XML Names to SQL <identifier>s can be used to regenerate the original SQL <identifier>, no matter which variant is used to generate the XML Name. The technique is simply to scan the XML Name from left to right, replacing substrings of the form **_x*HHHH*_** or **_x*HHHHHHHH*_** by the character with that escape sequence. (An initial **_xFFFF_** is simply discarded.)

A little care must be taken to insure that the mapping from SQL <identifier>s to XML Names is invertible. Consider for example the SQL <identifier> "_x005F@". The at-sign is not a valid XML Name character, so it would seem that the mapping would be **_x005F_x0040_** . Now if you use the reverse mapping, scanning from left to right, you encounter the escape sequence **_x005F_** , which would be mapped back to <underscore>, and then the resulting SQL <identifier> would be "_x0040_", which is not the one we started with.

To avoid this problem, it was decided that whenever the source SQL <identifier> contains "_x", then the <underscore> would be escaped as **_x005F_** . Thus the example "_x005F@" is actually mapped to the XML Name **_x005F_x005F_x0040_** . The reverse mapping takes the first **_x005F_** as an escape sequence, which is converted back to <underscore>. The remainder

of the XML Name is **x005F_x0040_** . Since this does not begin with an <underscore>, the next escape sequence is **_x0040_** , which is mapped back to "@". Altogether, the reverse mapping gives "_x005F@", i.e., the original SQL <identifier>.

Examples in which the escape variant does not matter:

| SQL <identifier> | XML Name: | Remarks |
|---|---|---|
| emp | **EMP** | SQL identifiers default to upper-case |
| "emp" | **emp** | Delimited identifiers suffer no case conversion |
| "Emp" | **Emp** | |
| "work@home" | **work_x0040_home** | @ = U+0040 |
| "work_x0040_home" | **work_x005F_x0040_home** | The first underscore is followed by 'x', so it is escaped as **_x005F_** . The subsequent **x0040_** is not an escape sequence, it is simply copied from the source. |
| "work_home" | **work_home** | The underscore is not followed by 'x' so it is not escaped. |
| "@@" | **_x0040__x0040_** | Note the two successive underscores when two adjacent characters must be escaped. |
| ":1990" | **_x003A_1990** | Initial colon is always escaped because it would not be a valid first character of an XML name. See the next table for internal colons. |

Examples in which the escape variant matters:

**Table 1:**

| SQL <identifier> | XML Name | |
|---|---|---|
| | partially escaped | fully escaped |
| "po:customer" | **po:customer** | **po_x003A_customer** |
| xml | **XML** | **_xFFFF_XML** |
| "xml" | **xml** | **_xFFFF_xml** |

As already noted, the mapping from SQL <identifier>s to XML is invertible. It is expected that a later change proposal for the base document will propose the inverse mapping from XML Names to SQL <identifier>s.

Note that the reverse mapping is only a "one-sided inverse" (in mathematical parlance). That is, while the sequence

SQL <identifier> -> XML Name -> SQL <identifier>

restores the original SQL <identifier>, the sequence

XML Name -> SQL <identifier> -> XML Name

is not guaranteed to restore the original XML Name. For example, both _ and **_x005F_** are mapped back to _ by this sequence.

## 3.3 Data type mappings

The SQLX group considered two styles of mapping data types from SQL to XML, which might be called the SQL-centric and the XML-centric styles.

1. In the SQL-centric style, the ***sqlxml:*** namespace defines named types which correspond as closely as possible to the SQL predefined types. The user of the SQL-centric mapping style uses these types, rather than translating directly into the XML Schema type system. When two SQL types map to the same XML Schema type, there are separate named XML Schema types corresponding to each of them. Thus the original SQL type of, e.g., a column of SQL data is apparent from the name of the XML Schema type that the column is mapped to.
2. In the XML-centric style, the ***sqlxml:*** namespace provides minimal support for the mappings, and in particular does not try to provide a predefined XML Schema type corresponding to each SQL predefined type. Instead, an optional set of annotations is available to record the original SQL type of columns, etc. Since the annotations are optional, the original SQL type may not be apparent.

After both styles had been elaborated, it was found that the SQL-centric style did not have much added value over the XML-centric style. The reason is that XML does not have "type templates", that is, the ability to define, e.g., a type ***sqlxml:*****CHAR** with a required parameter for the character string length. Instead the length can only be specified using the optional ***xsd:*****length** facet that is already present in the XML built-in type ***xsd:*****string**. Consequently it was decided to only propose the XML-centric style. In the future, if XML adds type templates, it may be appropriate to revisit this decision.

Only the mappings for SQL predefined types have been defined so far; work is continuing to define mappings for the user-defined and constructed types. It is also expected that there will be mappings of the XML Schema types into SQL.

### 3.3.1 Annotations

As explained, the philosophy behind the proposed data type mappings is to map each SQL predefined type to its closest analog in XML Schema, and to use the optional XML Schema facets to capture as much as possible the semantics of the SQL type.

Mapping to the closest analog of the SQL type is not always sufficient to enable the reconstruction of the SQL type. For example, both VARCHAR and CLOB are mapped to ***xsd*:string** with the ***xsd*:maxLength** facet to indicate the maximum length of the string. In addition, some aspects of the SQL type system have no analog in XML at all, such as character set name and collation. For some purposes it is expected that reconstructing the SQL type will be desired; consequently the mappings have been augmented with optional annotations that may be used to fully document the original SQL type.

For uniformity, every aspect of the SQL type system has an annotation, although at times this is redundant with information already present in the non-annotational part of the type mapping.

The annotations use the single XML Schema element **<*sqlxml*:sqltype>**, which has the attributes shown in the following table:

| **<sqltype>** attribute | Use |
|---|---|
| **name** | name of the SQL type |
| **length** | length of the SQL type |
| **maxLength** | maximum length of the SQL type |
| **characterSetName** | character set name |
| **collation** | collation name |
| **precision** | precision |
| **leadingPrecision** | precision of the leading field of an interval type |
| **userPrecision** | user-specified precision of DECIMAL and FLOAT types (may be less than the actual precision) |
| **scale** | scale or fractional seconds precision |
| **maxExponent** | maximum binary exponent of approximate numeric types |
| **minExponent** | minimum binary exponent of approximate numeric types |

The XML Schema for the annotations may be found in [SQL/XML candidate] Clause 6, “The ***sqlxml:*** namespace”.

Obviously not all annotations are relevant to all types. The definitions in Clause 6 do not define all the restrictions that might be defined on the attributes; instead the General Rules in Subclause

5.2, “Mapping SQL data types to XML Schema data types”, are relied on to insure that only the appropriate annotations are generated for any particular SQL type.

Whether to generate the defined annotations, in whole or in part, is entirely implementation-dependent, since the contexts in which these mappings might be used are not known, and hence it is not known whether implementations will be able to always generate them.

### 3.3.2 Character string types

The SQL character string types are mapped to the XML Schema type ***xsd*:string** . The facet ***xsd*:length** is used if it can be determined that every string in the SQL type will map to the same length in the XML Schema type, otherwise the ***xsd*:maxLength** facet is used. This means that ***xsd*:length** is used if the source type is CHAR and the character string mapping is homomorphic, as defined in section 3.1, “Character string mappings”.

Other facets are not used.

The relevant annotations are shown in the following table:

| **<sqltype>** attribute | Value |
|---|---|
| **name** | **CHAR** , **VARCHAR** or **CLOB** |
| **length** | length in characters (CHAR) |
| **maxLength** | maximum length (VARCHAR and CLOB) |
| **characterSetName** | character set name |
| **collation** | collation name |

Finally, here are some examples, with all annotations supplied:

| **SQL type** | **XML Schema type** |
|---|---|
| `CHAR (10)`<br>`  CHARACTER SET`<br>`  LATIN1`<br>`  COLLATION`<br>`  DEUTSCH` | `<xsd:simpleType>`<br>`  <xsd:restriction base=”xsd:string”>`<br>`  <xsd:length value=”10”/>`<br>`  <xsd:annotation>`<br>`  <sqlxml:sqltype name=”CHAR”`<br>`  length=”10”`<br>`  characterSetName=”LATIN1”`<br>`  collation=”DEUTSCH”/>`<br>`  </xsd:annotation>`<br>`  <xsd:restriction>`<br>`</xsd:simpleType>` |

| SQL type | XML Schema type |
|---|---|
| VARCHAR (10)<br><br>*on an implementation that defaults to the UTF8 character set with NORSK collation* | <xsd:simpleType><br><xsd:restriction base=”xsd:string”><br><xsd:maxLength value=”10”/><br><xsd:annotation><br><sqlxml:sqltype name=”VARCHAR”<br>maxLength=”10”<br>characterSetName=”UTF8”<br>collation=”NORSK”/><br></xsd:annotation><br></xsd:restriction><br></xsd:simpleType> |
| NCLOB (1 M)<br><br>*on an implementation in which national character types default to UCS2 with collation SVENSK* | <xsd:simpleType><br><xsd:restriction base=”xsd:string”><br><xsd:maxLength value=”1048576”/><br><xsd:annotation><br><sqlxml:sqltype name=”CLOB”<br>maxLength=”1048576”<br>characterSetName=”UCS2”<br>collation=”SVENSK”/><br></xsd:annotation><br></xsd:restriction><br></xsd:simpleType> |

### 3.3.3 BINARY LARGE OBJECT

The XML Schema type that corresponds to the SQL data type BINARY LARGE OBJECT or BLOB is ***xsd*:binary**. To use this XML Schema data type you must specify the **encoding** facet, which must be either **hex** or **base64**. Since [XML Schema:Datatypes] does not permit the direct use of the binary type, the ***sqlxml*:** namespace defines two data types, **binaryhex** and **binarybase64** in Clause 6 as follows:

```
<xsd:simpleType name=binaryhex>
  <xsd:restriction base=”binary”>
    <xsd:encoding value=”hex”/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name=binarybase64>
  <xsd:restriction base=”binary”>
    <xsd:encoding value=”base64”/>
  </xsd:restriction>
</xsd:simpleType>
```

The ***xsd*:maxLength** facet is used to specify the maximum length of the BLOB in octets. Other facets are not used.

The relevant annotations are shown in the following table:

| **<sqltype>** attribute | Value |
|---|---|
| **name** | **BLOB** |
| **maxLength** | maximum length |

The SQLX group is aware of an issue with BLOB and CLOB types, which is the cost in both space and transmission time when a LOB value is included 'in-line' in an XML document. It is possible that other alternative solutions for representing LOBs will be developed. Currently, however, the candidate base document only supports the definition of LOBs as in-line values.

Finally, here is an example showing the two possible mappings:.

| **SQL type** | **XML Schema type** |
|---|---|
| BLOB (1000)<br><br>*if the implementation chooses hex encoding* | <xsd:simpleType><br>  <xsd:restriction base="sqlxml:binaryhex"><br>  <xsd:maxLength value="2000"/><br>  <xsd:annotation><br>  <sqlxml:sqltype name="BLOB"<br>  maxLength="1000"/><br>  </xsd:annotation><br>  </xsd:restriction><br></xsd:simpleType> |
| BLOB (1000)<br><br>*if the implementation chooses base64 encoding* | <xsd:simpleType><br>  <xsd:restriction<br>  base="sqlxml:binarybase64"><br>  <xsd:maxLength value="1334"/><br>  <xsd:annotation><br>  <sqlxml:sqltype name="BLOB"<br>  maxLength="1000"/><br>  </xsd:annotation><br>  </xsd:restriction><br></xsd:simpleType> |

### 3.3.4 Bit string types

The [XML Schema: Datatypes] equivalent of BIT and BIT VARYING is ***xsd*:binary**. The implementation may use either value of **encoding** facet (**hex** or **base64**) by choosing either ***sqlxml*:binaryhex** or ***sqlxml*:binarybase64**, defined in the previous subsection. The ***xsd*:length** facet is used when mapping BIT to specify the length of the representation in

characters using the chosen encoding. For example, BIT(5) in **hex** encoding requires 2 characters (one to encode 4 bits and another character for the fifth bit) and 1 character in the **base64** encoding. Similarly, the ***xsd*:maxLength** facet is used when mapping BIT VARYING to specify the maximum length in characters using the chosen encoding. Other facets are not used.

The relevant annotations are shown in the following table:

| **<sqltype>** attribute | Value |
|---|---|
| **name** | name of the SQL type |
| **length** | length (for BIT) |
| **maxLength** | maximum length (for BIT VARYING) |

Finally, here are examples, with full annoations:

| **SQL type** | **XML Schema type** |
|---|---|
| BIT (23) | *either*<br><br><xsd:simpleType><br><xsd:restriction base=”sqlxml:binaryhex”><br><xsd:length value=”6”/><br><xsd:annotation><br><sqlxml:sqltype<br>name=”BIT”<br>length=”23”/><br></xsd:annotation><br></xsd:restriction><br></xsd:simpleType><br><br>*or*<br><br><xsd:simpleType><br><xsd:restriction<br>base=”sqlxml:binarybase64”><br><xsd:length value=”4”/><br><xsd:annotation><br><sqlxml:sqltype<br>name=”BIT”<br>length=”23”/><br></xsd:annotation><br></xsd:restriction><br></xsd:simpleType> |

| SQL type | XML Schema type |
|---|---|
| `BIT VARYING (23)`<br><br>*if the implementation chooses hex encoding* | `<xsd:simpleType>`<br>`  <xsd:restriction base=”sqlxml:binaryhex”>`<br>`  <xsd:maxLength value=”6”/>`<br>`  <xsd:annotation>`<br>`  <sqlxml:sqltype name=”BIT VARYING”`<br>`  length=”23”/>`<br>`  </xsd:annotation>`<br>`  </xsd:restriction>`<br>`</xsd:simpleType>` |
| `BIT VARYING (23)`<br><br>*if the implementation chooses base64 encoding* | `<xsd:simpleType>`<br>`  <xsd:restriction`<br>`  base=”sqlxml:binarybase64”>`<br>`  <xsd:length value=”4”/>`<br>`  <xsd:annotation>`<br>`  <sqlxml:sqltype name=”BIT VARYING”`<br>`  length=”23”/>`<br>`  </xsd:annotation>`<br>`  </xsd:restriction>`<br>`</xsd:simpleType>` |

### 3.3.5 Exact numeric types - NUMERIC and DECIMAL

The [XML Schema: Datatypes] equivalent of NUMERIC and DECIMAL is ***xsd*:decimal**. The facets ***xsd*:precision** and ***xsd*:scale** are used to define the characteristics of the SQL data type.

Note that according to [Foundation CD] 6.1 <data type> SR 23), an implementation may “round up” the precision that the user specifies with DECIMAL to a greater value. The ***xsd*:precision** facet represent the actual precision; the user-specified precision, if the RDBMS even retains it, may be reported in the **userPrecision** annotation.

The relevant annotations are shown in the following table:

| **<sqltype>** attribute | Value |
|---|---|
| **name** | **NUMERIC** or **DECIMAL** |
| **precision** | precision |
| **userPrecision** | user-specified precision of DECIMAL type |
| **scale** | scale or fractional seconds precision |

Here are examples with the complete complement of annotations:.

| SQL type | XML Schema type |
|---|---|
| `NUMERIC (7, 2)` | `<xsd:simpleType>`<br>`  <xsd:restriction base=”xsd:decimal”>`<br>`  <xsd:precision value=”7”/>`<br>`  <xsd:scale value=”2”/>`<br>`  <xsd:annotation>`<br>`  <sqlxml:sqltype name=”NUMERIC”`<br>`  precision=”7”`<br>`  scale=”2”/>`<br>`  </xsd:annotation>` *]*<br>`  </xsd:restriction>`<br>`</xsd:simpleType>` |
| `DECIMAL (8, 2)`<br><br>*on an implementation that assigns this an actual precision of 9.* | `<xsd:simpleType>`<br>`  <xsd:restriction base=”xsd:decimal”>`<br>`  <xsd:precision value=”9”/>`<br>`  <xsd:scale value=”2”/>`<br>`  <xsd:annotation>`<br>`  <sqlxml:sqltype name=”DECIMAL”`<br>`  userPrecision=”8”`<br>`  scale=”2”/>`<br>`  </xsd:annotation>`<br>`  </xsd:restriction>`<br>`</xsd:simpleType>` |

### 3.3.6 Exact numeric types - INTEGER and SMALLINT

The [XML Schema: Datatypes] equivalent of INTEGER and SMALLINT is ***xsd*:integer**. The ***xsd*:maxInclusive** and ***xsd*:minInclusive** are used to define the representational range of the INTEGER and SMALLINT data types for a particular implementation. No other facets are used.

The relevant annotation is shown in the following table:

| **<sqltype>** attribute | Value |
|---|---|
| **name** | **INTEGER** or **SMALLINT** |

Here are examples showing the full complement of annotations:.

| SQL type | XML Schema type |
|---|---|
| `INTEGER`<br><br>*on an implementation using 4-byte twos-complement binary integers* | `<xsd:simpleType>`<br>`  <xsd:restriction base=”xsd:integer”>`<br>`  <xsd:maxInclusive value=”2157483647”/>`<br>`  <xsd:minInclusive value=”-2157483648”/>`<br>`  <xsd:annotation>`<br>`  <sqlxml:sqltype name=”INTEGER”/>`<br>`  </xsd:annotation>`<br>`  </xsd:restriction>`<br>`</xsd:simpleType>` |
| `SMALLINT`<br><br>*on an implementation using 2-byte twos-complement binary integers* | `<xsd:simpleType>`<br>`  <xsd:restriction`<br>`  base=”xsd:integer”>`<br>`  <xsd:maxInclusive value=”32767”/>`<br>`  <xsd:minInclusive value=”-32768”/>`<br>`  <xsd:annotation>`<br>`  <sqlxml:sqltype name=”SMALLINT”/>`<br>`  </xsd:annotation>`<br>`  </xsd:restriction>`<br>`</xsd:simpleType>` |

### 3.3.7 Approximate numeric types

[XML Schema: Datatypes] supports IEEE float and double data types which have fixed precision of 24 bits and 53 bits, respectively, and binary exponents in the ranges [-149, 104] and [-1075, 970], respectively . In [Foundation CD], the binary precision and exponent range of approximate numeric types are not nearly so tightly specified. The SQLX group concluded that the best way to map the approximate numeric types is on the basis of the actual binary precision and exponent range of the SQL type. Essentially, if the binary precision of the SQL type is less than or equal to 24 bits, and the exponent range is contained in [-149, 104], then every value of the SQL type can be mapped to a value of the XML Schema type ***xsd*:float**, otherwise the larger type ***xsd*:double** is used. No facets are used.

The relevant annotations are shown in the following table:

| **<sqltype>** attribute | Value |
|---|---|
| **name** | **REAL**, **DOUBLE PRECISION** or **FLOAT** |
| **precision** | precision |
| **userPrecision** | user-specified precision of FLOAT types (may be less than the actual precision) |

| **<sqltype>** attribute | Value |
|---|---|
| **maxExponent** | maximum binary exponent of approximate numeric types |
| **minExponent** | minimum binary exponent of approximate numeric types |

Here are some examples showing the full complement of annotations:.

| **SQL type** | **XML Schema type** |
|---|---|
| FLOAT (20)<br><br>*on an implementation that uses a 4-byte floating point with one byte for binary exponent (ranging from -128 to 127)* | <xsd:simpleType><br><xsd:restriction base=”xsd:double”/><br><xsd:annotation><br><sqlxml:sqltype name=”FLOAT”<br>precision=”24”<br>minExponent=”-128”<br>maxExponent=”127”<br>userPrecision=”20”/><br></xsd:annotation><br></xsd:restriction><br></xsd:simpleType> |
| REAL<br><br>*on an implementation that uses IEEE float type for this type* | <xsd:simpleType><br><xsd:restriction base=”xsd:float”><br><xsd:annotation><br><sqlxml:sqltype name=”REAL”<br>precision=”24”<br>minExponent=”-149”<br>maxExponent=”104”/><br></xsd:annotation><br></xsd:restriction><br></xsd:simpleType> |
| DOUBLE PRECISION<br><br>*on an implementation that uses IEEE double for this type* | <xsd:simpleType><br><xsd:restriction base=”xsd:double”><br><xsd:annotation><br><sqlxml:sqltype<br>name=”DOUBLE PRECISION”<br>precision=”53”<br>minExponent=”-1075”<br>maxExponent=”970”/><br></xsd:annotation> *]*<br></xsd:restriction><br></xsd:simpleType> |

### 3.3.8 BOOLEAN

The [XML Schema: Datatypes] equivalent of BOOLEAN is ***xsd*:boolean**. No facets are used.

The relevant annotations are shown in the following table:

| **<sqltype>** attribute | Value |
|---|---|
| **name** | **BOOLEAN** |

Here is an example. Note that in the minimal case, no facets and no annotations are used, so it is not necessary to use **<*xsd*:simpleType>** ... **</*xsd*:simpleType>**.

| **SQL type** | **XML Schema type** |
|---|---|
| BOOLEAN | `<... type="xsd:boolean"/>`<br><br>*or*<br><br>`<xsd:simpleType>`<br>`  <xsd:restriction base="xsd:boolean"/>`<br>`  <xsd:annotation>`<br>`  <sqlxml:sqltype name="BOOLEAN"/>`<br>`  </xsd:annotation>`<br>`  </xsd:retriction>`<br>`</xsd:simpleType>` |

### 3.3.9 DATE

The [XML Schema: Datatypes] equivalent of DATE is ***xsd*:date**. Since XML permits a time zone but SQL does not, the ***xsd*:pattern** facet is used to specify a pattern that prohibits a timezone. No other facets are used.

The relevant annotations are shown in the following table:

| **<sqltype>** attribute | Value |
|---|---|
| **name** | **DATE** |

Here is an example showing the optional annotation:

| SQL type | XML Schema type |
|---|---|
| DATE | <xsd:simpleType><br>  <xsd:restriction base=”xsd:date”><br>  <xsd:pattern value=<br>  ”\p{Nd}{4}-\p{Nd}{2}-\p{Nd}{2}”/><br>  <xsd:annotation><br>  <sqlxml:sqltype name=”DATE”/><br>  </xsd:annotation><br>  </xsd:retriction><br></xsd:simpleType> |

### 3.3.10 TIME with or without time zone

The [XML Schema: Datatypes] equivalent of TIME and TIME WITH TIME ZONE is ***xsd*:time**. The ***xsd*:pattern** facet is used to specify the seconds precision and to either forbid or require the time zone, as necesary. No other facets are used.

The relevant annotations are shown in the following table:

| **<sqltype>** attribute | Value |
|---|---|
| **name** | **TIME** or **TIME WITH TIME ZONE** |
| **scale** | fractional seconds precision |

Here are some examples:.

| SQL type | XML Schema type |
|---|---|
| TIME (*2*) | <xsd:simpleType><br>  <xsd:restriction base=”xsd:time”><br>  <xsd:pattern value=<br> ”\p{Nd}{2}:\p{Nd}{2}:\p{Nd}{2}.\p{Nd}{*2*}”/><br>  <xsd:annotation><br>  <sqlxml:sqltype name=”TIME”<br>  scale=”*s*”/><br>  </xsd:annotation><br>  </xsd:restriction><br></xsd:simpleType> |

| SQL type | XML Schema type |
|---|---|
| TIME WITH TIME ZONE (0)<br><br><xsd:simpleType><br><xsd:restriction base=”xsd:time”><br><xsd:pattern value=<br>”\p{Nd}{2}:\p{Nd}{2}:\p{Nd}{2}(+\|-)\p{Nd}{2}:\p{Nd}{2}”/><br><xsd:annotation><br><sqlxml:sqltype name=”TIME WITH TIME ZONE”<br>scale=”0”/><br></xsd:annotation><br></xsd:restriction><br></xsd:simpleType> | |

### 3.3.11 TIMESTAMP with and without time zone

The [XML Schema: Datatypes] equivalent of TIMESTAMP and TIMESTAMP WITH TIME ZONE is ***xsd*:timeInstant**. The ***xsd*:pattern** facet is used to specify the seconds precision and either forbid or require the time zone, as required.

The relevant annotations are shown in the following table:

| **<sqltype>** attribute | Value |
|---|---|
| **name** | **TIMESTAMP** or **TIMESTAMP WITH TIME ZONE** |
| **scale** | fractional seconds precision |

The patterns for the timestamp types is too long to display on a single line, so no examples are given.

### 3.3.12 Interval types

The [XML Schema: Datatypes] equivalent of INTERVAL is **timeDuration**. The **xsd:pattern** facet is used to require the particular fields in the interval.

The relevant annotations are shown in the following table:

| **<sqltype>** attribute | Value |
|---|---|
| **name** | **INTERVAL YEAR**<br>**INTERVAL YEAR TO MONTH**<br>**INTERVAL MONTH**<br>**INTERVAL DAY**<br>**INTERVAL DAY TO HOUR**<br>**INTERVAL DAY TO MINUTE**<br>**INTERVAL DAY TO SECOND**<br>**INTERVAL HOUR**<br>**INTERVAL HOUR TO MINUTE**<br>**INTERVAL HOUR TO SECOND**<br>**INTERVAL MINUTE**<br>**INTERVAL MINUTE TO SECOND**<br>**INTERVAL SECOND** |
| **leadingPrecision** | precision of the leading field of the interval type |
| **scale** | fractional seconds precision if the seconds field is included |

Here are some representative examples:

| **SQL type** | **XML Schema type** |
|---|---|
| INTERVAL YEAR (4) | <xsd:simpleType><br>  <xsd:restriction base=”xsd:timeDuration”><br>  <xsd:pattern value=”-?P\p{Nd}{1,4}Y”/><br>  <xsd:annotation><br>  <sqlxml:sqltype<br>  name=”INTERVAL YEAR”<br>  leadingPrecision=”4”/><br>  </xsd:annotation> *]*<br>  </xsd:restriction><br></xsd:simpleType> |

| SQL type | XML Schema type |
|---|---|
| INTERVAL<br>YEAR (4) TO MONTH | <xsd:simpleType><br>  <xsd:restriction base=”xsd:timeDuration”><br>  <xsd:pattern value=<br>  ”-?P\p{Nd}{1,4}Y\p{Nd}{2}M”/><br>  <xsd:annotation><br>  <sqlxml:sqltype<br>  name=”INTERVAL YEAR TO MONTH”<br>  leadingPrecision=”4”/><br>  </xsd:annotation><br>  </xsd:restriction><br></xsd:simpleType> |

```
INTERVAL DAY (6) TO SECOND (2)

<xsd:simpleType>
  <xsd:restriction base=”xsd:timeDuration”>
    <xsd:pattern value=
     ”-?P\p{Nd}{1,p}DT\p{Nd}{2}H\p{Nd}{2}M\p{Nd}{2}.\p{Nd}{s}S”
    />
    <xsd:annotation>
      <sqlxml:sqltype name=”INTERVAL DAY TO SECOND”
        leadingPrecision=”6”
        scale=”2”/>
    </xsd:annotation>
  </xsd:restriction>
</xsd:simpleType>
```

# 4. Example

As mentioned, it is expected that eventually SQL/XML will contain mappings for tables as well. Two of the flavors commonly mentioned for such mappings are to map columns as either elements or attributes. Given the table definition

```
CREATE TABLE Hoopla
( HooplaName CHAR(10),
  HooplaCode NUMERIC(4),
  HooplaDate DATE )
```

Using elements to map columns might look like this (all annotations have been omitted for brevity):

```
<xsd:complexType name=”HOOPLA”>
  <xsd:sequence>
```

```
      <xsd:element name="HOOPLANAME">
        <xsd:simpleType>
          <xsd:restriction base="string">
            <xsd:length value="10"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="HOOPLACODE">
        <xsd:simpleType>
          <xsd:restriction base="decimal">
            <xsd:precision value="4"/>
            <xsd:scale value="0"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="HOOPLADATE">
        <xsd:simpleType>
          <xsd:restriction base="date">
            <xsd:pattern
             value="\p{Nd}{4}-\p{Nd}{2}-\p{Nd}{2}"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
```

Since every column of the table maps to a simple type, it would also be possible to use attributes for the columns. In that case the XML correspondence for the table definition might be

```
  <xsd:complexType name="HOOPLA">
    <xsd:sequence>
      <xsd:attribute name="HOOPLANAME">
        <xsd:simpleType>
          <xsd:restriction base="string">
            <xsd:length value="10"/>
          </xsd:restriction>
        </xsd:simpleType>
      <xsd:attribute/>
      <xsd:attribute name="HOOPLACODE">
        <xsd:simpleType>
          <xsd:restriction base="decimal">
            <xsd:precision value="4"/>
            <xsd:scale value="0"/>
          </xsd:restriction>
        </xsd:simpleType>
      <xsd:attribute/>
```

```
      <xsd:element name="HOOPLADATE">
        <xsd:simpleType>
          <xsd:restriction base="date">
            <xsd:pattern
             value="\p{Nd}{4}-\p{Nd}{2}-\p{Nd}{2}"/>
          </xsd:restriction>
        </xsd:simpleType>
      <xsd:attribute/>
    </xsd:sequence>
  </xsd:complexType>
```

# 5. Possible Problems

The candidate base document notes a few Possible Problems. There are many other problems that the authors are well aware of, but did not see fit to make into Possible Problems, because these problems are self-evident. For example, the numerous passages that are "to be supplied" will be obvious ballot comments if they are still present when the document is progressed to CD status, and no PP is necessary.

***- End of paper -***