Title: XMLCast

Author: Fred Zemke, for the H2 ad hoc subcommittee on SQL/XML
Source: U.S.A.

Status: Change proposal
Date: March 14, 2004

Abstract

This paper is the second in a series of three, which collectively will provide the ability to query XML values in SQL. This paper specifies a cast function to convert between SQL predefined types and XML types. This is necessary to input values to XMLQuery, and also to interpret the result.

changes in r1: change ANYTYPE to ANY; discuss when copying is required when casting for XML to XML; other editorial improvements

References

[Foundation:2003]	Jim Melton (ed), "ISO International Standard (IS) Database Language SQL - Part 2: SQL/Foundation", ISO/IEC 9075-2:2003
[SQL/XML:2003]	Jim Melton (ed), "ISO International Standard (IS) Database Language SQL - Part 14: SQL/XML", ISO/IEC 9075-14:2003
[Foundation WD]	Jim Melton (ed), "Working Draft (WD) Database Language SQL - Part 2: SQL/Foundation", ISO/IEC JTC1/SC32 WG3:HBA-003 = ANSI INCITS H2-2003-305
[SQL/XML WD]	Jim Melton (ed.), "Working Draft (WD) XML-Related Specifications (SQL/XML)", ISO/IEC JTC1/SC32 WG3:SIA-010 = ANSI INCITS H2-2003-427
[SQL:2003 TC]	Stephen Cannan (ed.), "Draft Technical Corrigendum", ISO/IEC JTC1/SC32 WG3:HBA-011
[XQuery DM]	"XQuery 1.0 and XPath 2.0 Data Model", W3C working draft, 12 November 2003, available at http://www.w3.org/TR/xpath-datamodel/
[H2-2004-019]	Fred Zemke, "Moving to the XQuery data model", ANSI INCITS H2-2004-019

1. XMLCast

This is the second of three papers in a series. This paper assumes [H2-2004-019] as a prerequisite.

To convert values of XML type to and from other SQL types, we propose XMLCast. The reason for not using the existing CAST pseudofunction is because some implementations have already treated casting XML to/from a character string as equiavlent to XMLParse/XMLSerialize respectively.

The proposed syntax is the same as CAST except for the keyword:

```
<XML cast specification> ::=
     XMLCAST <left paren> <XML cast operand> AS
     <XML cast target> <right paren>

<XML cast operand> ::=
       <value expression>
     | NULL
     | EMPTY

<XML cast target> ::=
       <domain name>
     | <data type>
```

Either the source's data type or the target's data type, or both, must be XML. As usual, a domain is treated as just an alias for a built-in type. Distinct types are effectively treated the same as their source type; there are no user-defined casts to or from XML types. Collection types, row types, structured types and reference types are not supported as either the source or the target. Such casts may be proposed in the future.

1.1 Casting EMPTY to XML

Casting the keyword EMPTY to XML is handled as follows:

XMLCast (EMPTY AS XML(UNTYPED CONTENT)) results in a document node with no children.

XMLCast (EMPTY AS XML(ANY CONTENT)) also results in a document node with no children.

XMLCast (EMPTY AS XML(SEQUENCE)) results in an empty XQuery sequence.

XMLCast (EMPTY AS XML(UNTYPED DOCUMENT)) or to XML(ANY DOCUMENT) is forbidden by a Syntax Rule.

1.2 Casting from XML to XML

Casting from one XML type to another is permitted in all twenty five cases (a five-by-five matrix). When casting to an UNTYPED type, any XML Schema type information is lost. Aside from this, the result is identical to the argument, if the argument meets the constraints of the result type, otherwise an exception is raised. For example, casting an XML(SEQUENCE) value to XML(UNTYPED DOCUMENT) is permitted, and succeeds if and only if the input value is a document node with a single child element.

The main issue is when a copy is required, which loses node identity. Basically, if the source is typed (XML(ANY DOCUMENT), XML(ANY CONTENT) or XML(SEQUENCE)) and the target is untyped (XML(UNTYPED DOCUMENT) or XML(UNTYPED CONTENT)) then a copy is required. The complete semantics are summarized in the following table:

	Target type				
Source type:	UNTYPED DOC.	ANY DOC.	UNTYPED CONTENT	ANY CONTENT	SEQUENCE
UNTYPED DOC.	no copy no exceptions	no copy no exceptions	no copy no exceptions	no copy no exceptions	no copy no exceptions
ANY DOC.	copy no exceptions	no copy no exceptions	copy no exceptions	no copy no exceptions	no copy no exceptions
UNTYPED CONTENT	no copy possible exc.	no copy possible exc	no copy no exceptions	no copy no exceptions	no copy no exceptions
ANY CONTENT	copy possible exc	no copy possible exc.	copy no exceptions	no copy no exceptions	copy no exceptions
SEQUENCE	copy possible exc	no copy possible exc.	copy possible exc.	no copy possible exc.	no copy no exceptions

1.3 Casting from SQL to XML

The null value of any input type, or the keyword NULL, results in the null value in the target XML type.

For casting other SQL types to XML, Subclauses 9.15, “Mapping SQL data types to XML Schema data types”, and 9.16, “Mapping values of SQL data types to values of XML Schema data types”, are the starting point. These subclauses provide mappings for the built-in types in Foundation (character string, BLOB, numeric, boolean, datetime, and interval), but not the built-in types in other parts.

For all these types, the result of Subclause 9.16 is a text node. When casting to XML(CONTENT), this text node is simply placed in a document node.

As for casting to XML(SEQUENCE), this will be used for input to XMLQuery, where we believe it will usually be more convenient to input XQuery atomic values rather than text nodes. For example,

```
     XMLQuery ('//book/author[$N]' PASSING BY VALUE
                                   T.Library, X.AuthNum AS N
                RETURNING SEQUENCE)
```

In this example, assume that T.Library is a value of XML(CONTENT). Anticipating a little, T.Library is being used to initialize the context item of XQuery, indicated by the fact that no variable name is given for it. Also assume that X.AuthNum is an integer, which will be used to initialize the XQuery variable $N. The intent is that the XPath expression will pick out the N-th <author> of each <book> in the library. For this to work, $N must be an integer atomic value within XQuery.

As stated above, the input arguments of XMLQuery are implicitly cast to XML(SEQUENCE). To support examples like the preceding, it is necessary that SQL built-in types (other than XML) should be converted to XQuery atomic values. As it happens, Subclause 9.15 already specifies an XML Schema simple type for each of the predefined types of SQL/Foundation, so this is the atomic type that we cast to when casting to XML(SEQUENCE).

Finally, when casting to XML(UNTYPED DOCUMENT) or XML(ANY DOCUMENT), we propose that the source type must be some XML type. The result of casting a non-XML type to XML(UNTYPED CONTENT) or XML(ANY CONTENT) is a text node, and casting to XML(SEQUENCE) results in an atomic value. Neither of these is a value of type XML(ANY DOCUMENT), so it is not possible to specify such casts.

The conversions from non-XML to XML types rely on Subclauses 9.15 and 9.16, which each have three parameters: the SQL type, the encoding for binary strings (base64 or hex) and *NULLS*, an indication of how nulls are to be handled. When invoked from <XML cast specification>, the binary string encoding is determined by the innermost <XML binary encoding> containing the <XML cast specification>. As for *NULLS*, it is only needed for row types, which we have excluded.

1.4 Casting from XML to SQL

Casting from XML to SQL is more complicated. The first simplification is that we only support casts to XML and Foundation predefined types.

The second simplification is that casting from XML to XML has already been covered in the preceding section.

This leaves casts from XML to the Foundation predefined types. The algorithm is schematized in following diagram:

```
                         source XQuery value
                                    |
     1. Replace document nodes      |
                              V
                       XQuery sequence
                              |
2. XQuery atomization         |
                              V
               XQuery sequence of atomic values
                              |
3. XQuery cast expression     |
                              V
                     XQuery atomic value
                              |
4. value space identification |
                              V
             value of Foundation predefined type
                              |
5. SQL cast expression        |
                              V
                value of target SQL data type
```

(For simplicity, the paths that result in a null value have been omitted. One such path is if the input XML value is null. Another will be mentioned shortly.)

The first step is to remove any document nodes, replacing them by the sequence of their children. This prepares us for the next step, an XQuery technique called atomization, also known as the fn:data function. Here are some examples of atomization:

1. if <x xsi:nil='true'/> is a validated XML element for which the nilled property is true, then fn:data (<x xsi:nil='true'/>) is the empty sequence, and the result of the cast is the null value (not shown in the diagram).

1. fn:data (<x>203</x>) = 203

2. fn:data (<x id=’203’>hello</x>) = 'hello'

3. Starting from the element <x id='203'>hello</x>, if one performs an XPath to extract the attribute node, obtaining (using an XQuery attribute constructor) attribute id { 203 }, then fn:data (attribute id { 203 }) = 203

4. Thus fn:data ((<x id='203'>hello</x>)/@id) = 203

5. fn:data of a text node is an xs:string value equivalent to the text node.

6. fn:data of an atomic value is the atomic value

7. fn:data (<?pi target?>) = 'target'

8. fn:data (<-- this is a comment -->) = ' this is a comment '

The result of atomization is an XQuery sequence of atomic values. If this sequence is empty, the result of the cast is the null value. If this sequence contains more than one value, an exception is raised.

Step 3 is to convert the atomized XML value to an intermediate XQuery atomic type. The intermediate atomic type is determined by the existing rules of Subclause 9.15, "Mapping SQL data types to XML Schema data types", applied to the target SQL type. The following table shows the important cases:

SQL target data type	XQuery intermediate atomic type
character types	xs:string
BLOB	xs:hexBinary or xs:base64Binary
INTEGER, etc. (exact numeric with scale 0)	xs:integer
exact numeric with nonzero scale	xs:decimal
approximate numeric	xs:float or xd:double
DATE	xs:date
TIME	xs:time
TIMESTAMP	xs:dateTime
year-month interval	xdt:yearMonthDuration
day-time interval	xdt:dayTimeDuratino
BOOLEAN	xs:boolean

Step 4 is called "value space identification". This basically consists of equating every value of an XQuery atomic type with a value in a corresponding SQL/Foundation data type category. For example, a value of xs:string is a character string with character set Unicode. A value of type xs:integer or xs:decimal is an exact numeric value. A value of type xs:double or xs:float is an approximate numeric value. And so forth.

There are a few XQuery atomic values that cannot make this hop into the SQL world, namely:

1. In xs:float and xs:double, there are special values INF, -INF and NaN that are not SQL values.
2. XML Schema types xs:date and xs:dateTime support negative dates, but SQL does not.

Step 5 is to convert the value (which is now a value of a category of SQL/Foundation predefined type) to the target type. This final cast does two things for us:

1. In the case of character string types, it converts from Unicode to the character data type.
2. For all types, it imposes any constraints of the target SQL type.

An issue we faced in designing the cast rules was how strict to be. For example, SQL permits conversion of 1.2 to an integer, but XQuery cast expression does not (there are separate round and truncate functions that one may call). As another example, SQL has rules for handling timezones if the source has a timezone but the target does not, or vice versa. XQuery does not provide such rules in its cast expression (again, there are separate functions to provide timezone adjustment). We decided to abide by the stricter XQuery rules, for two reasons:

1. We believe that XML data should be interpreted by XML rules
2. It is more conservative to be strict initially. We can relax the rules later if desired.

Another issue we considered was whether to support textual data that is formatted according to SQL rules instead of XML Schema lexical conventions. We examined all the SQL/Foundation predefined types, comparing their lexical rules with XML Schema. We found that, for the most part, they use the same lexical conventions. For example, for character string, exact and approximate numerics, dates and times, the XML Schema literals are a superset of the SQL literals. (XML approximate numerics also have INF, -INF and NaN; XML dates support time zone, but SQL does not; XML supports Z as an abbreviation for the timezone 00:00). The three exceptions are:

SQL boolean literals are case-insensitive, whereas XML Schema boolean literals are in lower case.

SQL interval literals are completely different from XML Schema duration literals, which begin with 'P', '+P' or '-P' .

SQL timestamp literals separate the date and time portion with a space; in XML Schema literals, they are separated by the letter ‘D’.

Using these clues, it would be possible to detect XML text values that were encoded using SQL literals rather than XML Schema literals. However, we decided not to do this, following the principle that XML data follows XML rules. This could be liberalized in a later version.

1.5 Roundtrip conversions (“return” in the Queen’s English)

While not absolutely essential, it is interesting to ask what happens if one casts from SQL to XML and then back to the original SQL type again. The answer is that, for all the Foundation predefined types, you get the original value back again, modulo a possible rounding or truncation on the conversion to XML. This is true whether the intermediate XML type is XML(UNTYPED CONTENT), XML(ANY CONTENT) or XML(SEQUENCE).

If you cast from SQL to XML(SEQUENCE), then you get an XQuery atomic value in the corresponding XQuery atomic type. This step may involve a rounding or truncation. Coming back to SQL, steps 1 through 3 are basically no-ops. The value space identification in step 4 brings you back to the same SQL value that you started with (modulo the rounding or truncation already noted), and the final cast is assured of success because the value must already clear any constraints of the target type (= the original type).

If you cast an SQL type to XML(UNTYPED CONTENT) or XML(ANY CONTENT), then you have a document node containing a text node. This text node is just the XML Schema literal for the XQuery atomic value that you would have had, if you had cast to XML(SEQUENCE). Coming back to SQL, Step 1 removes the document node, Step 2 converts the text node to a value of type xs:string, Step 3 converts it to the XQuery atomic type you would have had in the XML(SEQUENCE) case, and then the remaining steps are the same as for XML(SEQUENCE).

Starting with an XML value, casting to an SQL type and then back to XML, you are unlikely to get your original XML value back. The starting point might be (a document node containing) an element or attribute, or even a processing instruction or comment; the result of the round trip must be a document node containing a text node in the XML(UNTYPED CONTENT) and XML(ANY CONTENT) cases, or an XQuery atomic value in the XML(SEQUENCE) case.

2. Conformance features

We decided that no Conformance Rules are necessary for XMLCast. Instead, to cast to a particular XML type, the user will need the conformance feature that supports the target type. Or to cast from a particular XML type, the user will need the conformance feature that supports the source type, since without the feature, the value cannot arise.

3. Future possibilities

We have not defined casts between XML and any of the following types: collection types, row types, structured types and reference types. These remain as future possibilities.

4. Proposal conventions

This proposal uses the following conventions:

1. SMALLCAPS	denote numbered editorial instructions;
~~strikeout~~	denotes existing text to be deleted;
bold red	denotes new text to be inserted;
plain	denotes existing text to be retained;
[Note:...]	brackets enclose italicized notes to the proposal reader;
$\boxed{}$	boxes surround “editing tags,” which are part of the document (not instructions to the editor) and may be deleted, inserted, modified or retained, depending on the typeface within the box.

5. Proposal for [SQL/XML WD]

5.1 New Subclause 4.2a, Data conversions

1. ADD THE FOLLOWING SUBCLAUSE PRIOR TO EXISTING SUBCLAUSE 4.3, "DATA ANALYSIS OPERATIONS (INVOLVING TABLES)":

4.2a Data conversions

This Subclause modifies Subclause 4.11, "Data conversions" in ISO/IEC 9075-2.

| Insert after 2nd paragraph | Data conversions between the predefined data types defined in ISO/IEC 9075-2 and the XML types can be specified by an <XML cast specification>.

5.2 New Subclause 4.9.11, Mapping XQuery atomic values to SQL values

1. ADD THE FOLLOWING SUBCLAUSE AT THE END OF SUBCLAUSE 4.9, "OVERVIEW OF MAPPINGS"

4.9.11 Mapping XQuery atomic values to SQL values

As defined in [XQuery DM], an XQuery atomic type is either an XML Schema primitive type, or derived from an XML Schema primitive type by restriction (and not by union or list). Each XQuery atomic type has a value space, consisting of all values of that XQuery atomic type.

Let *AV* be an XQuery atomic value. Let *AT* be the XQuery atomic type of *AV*. Let *PT* be given by:

Case:

— if *AT* is an XML Schema primitive type, then *AT*

— if *AT* is xdt:yearMonthDuration, or derived from xdt:yearMonthDuration, then xdt:yearMonthDuration

— if *AT* is xdt:dayTimeDuration, or derived from xdt:dayTimeDuration, then xdt:yearMonthDuration

— Otherwise, the XML Schema primitive type that *AT* is derived from.

This specification (notably, in Subclause 6.n, "<XML cast specification>") regards *AV* as being a value belonging to some category of SQL built-in type, as follows:

Case:

— If *PT* is xs:string, then *AV* is regarded as being a character string whose character repertoire is Unicode.

— If *PT* is xs:hexBinary or xs:base64Binary, then *AV* is regarded as being a binary string.

— If *PT* is xs:decimal, then *AV* is regarded as being an exact numeric value.

— If *PT* is xs:float or xs:double, then *AV* is regarded as being an approximate numeric value.

— If *PT* is xs:time and the XQuery datetime timezone component of *AV* is an empty XQuery sequence, then the XQuery datetime normalized value of *AV* is regarded as being a value of type TIME WITHOUT TIME ZONE.

— If *PT* is xs:time and the XQuery datetime timezone component of *AV* is not an empty XQuery sequence, then *AV* is regarded as being a value of type TIME WITH TIME ZONE, in which the XQuery datetime timezone component of *AV* is the timezone component, and the XQuery datetime normalized value is the UTC component.

— If *PT* is xs:dateTime, the XQuery datetime normalized value of *AV* is positive, and the XQuery datetime timezone component of *AV* is an empty XQuery sequence, then the XQuery datetime normalized value of *AV* is regarded as being a value of type TIMESTAMP WITHOUT TIME ZONE.

— If *PT* is xs:dateTime, the XQuery datetime normalized value of *AV* is positive, and the XQuery datetime timezone component of *AV* is not an empty XQuery sequence, then *AV* is regarded as being a value of type TIMESTAMP WITH TIME ZONE, in which the XQuery datetime timezone component of *AV* is the timezone component, and the XQuery datetime normalized value is the UTC component.

— If *PT* is xs:date, the XQuery datetime normalized value of *AV* is positive, and the XQuery datetime timezone component of *AV* is an empty XQuery sequence, then the XQuery datetime normalized value of *AV* is regarded as being a value of type DATE.

— If *PT* is xdt:yearMonthDuration, then *AV* is regarded as being a year-month interval.

— If *PT* is xdt:dayTimeDuration, then *AV* is regarded as being a day-time interval.

— If *PT* is xs:boolean, then *AV* is regarded as being a value of type BOOLEAN.

5.3 Changes to 5.1 <token> and <separator>

1. ADD THE FOLLOWING <RESERVED WORD>:

```
<reserved word> ::=
     . . .
     XMLCAST
```

5.4 New Subclause 6.2a <value expression primary>

1. ADD THE FOLLOWING SUBCLAUSE TO CLAUSE 6, "SCALAR EXPRESSIONS":

6.2a <value expression primary>

This Subclause modifies Subclause 6.3, "<value expression primary>", in ISO/IEC 9075-2.

Function

Specify a value that is syntactically self-delimited.

Format

```
<nonparenthesized value expression primary> ::=
       !! All alternatives from ISO/IEC 9075-2
     | <XML cast specification>
```

Syntax Rules

1) | Replace SR 1) | The declared type of a <value expression primary> is the declared type of the simply contained <value expression>, <unsigned value specification>, <column reference>, <set function specification>, <window function>, <scalar subquery>, <case expression>, <cast specification>, <XML cast specification>, <field reference>, <subtype treatment>, <method invocation>, <static method invocation>, <new specification>, <attribute or method reference>, <reference resolution>, <collection value constructor>, <array element reference>, <multiset element reference>, or <next value expression>, or the effective returns type of the simply contained <routine invocation>, respectively.

Access Rules

No additional Access Rules

General Rules

1) | Replace GR 1) | The value of a <value expression primary> is the value of the simply contained <value expression>, <unsigned value specification>,

<column reference>, <set function specification>, <window function>, <scalar subquery>, <case expression>, <cast specification>, <XML cast specification>, <field reference>, <subtype treatment>, <method invocation>, <static method invocation>, <new specification>, <attribute or method reference>, <reference resolution>, <collection value constructor>, <array element reference>, <multiset element reference>, <routine invocation>, or <next value expression>.

Conformance Rules

No additional Conformance Rules

5.5 Changes to 6.3, <cast specification>

1. ADD THE FOLLOWING SYNTAX RULE PRIOR TO SR 1) " | REPLACE SR 6) | ...

   n) | Replace SR 5) | If the <cast operand> specifies an <empty specification>, then *TD* shall be a collection type, XML(UNTYPED CONTENT), XML(ANY CONTENT) or XML(SEQUENCE).

2. DOES DL (DATALINK) BELONG IN THE LEGEND OF THE TABLE? IF IT DOES, THEN THE LETTER IS N FOR NO WHEN CASTING FROM XML TO DL OR DL TO XML.

3. EDIT THE GENERAL RULES AS FOLLOWS:

   0.1) | Replace GR 2)b) | If the <cast operand> specifies an <empty specification>, then

   Case:

   a) If *TD* is XML(UNTYPED CONTENT) or XML(ANY CONTENT), then *TV* is an XQuery document node whose children and unparsed-entity properties are empty, and whose base-uri and document-uri properties are implementation-defined. No further General Rules of this Subclause are applied.

   b) If *TD* is XML(SEQUENCE), then *TV* is an empty XQuery sequence, and no further General Rules of this Subclause are applied.

   c) Otherwise, *TV* is an empty collection of declared type *TD*, and no further General Rules of this Subclause are applied.

   1) | Insert after GR 2) | If *SD* is **an** XML **type**, then

   a) Case:

   i) If *TD* is XML(UNTYPED DOCUMENT) or XML(ANY DOCUMENT), and *SV* is not a value of type XML(ANY

DOCUMENT) , then an exception is raised: *data exception — not an XML document.*

ii) If *TD* is XML(UNTYPED CONTENT) or XML(ANY CONTENT), and *SV* is not a value of type XML(ANY CONTENT), then an exception is raised: *data exception — not an XQuery document node.*

b) Case:

i) If *TD* is XML(UNTYPED DOCUMENT) or XML(UNTYPED CONTENT), then let *TV* be the result of Subclause 10.n, "Unvalidating an XQuery document node".

ii) Otherwise, *TV* is *SV.*

4. DELETE THE ONLY CONFORMANCE RULE:

~~1) Insert this CR Without Feature X010, "XML type", conforming SQL language shall not contain a <cast operand> whose declared type is XML.~~

[NOTE to the proposal reader: I believe this CR is unnecessary. the truth is that without the Feature, the user cannot *specify a <cast operand> whose declared type is an XML type. Deleting this CR is preferable to creating one for each type modifier.]*

5.6 New Subclause 6.3a <XML cast specification>

1. INSERT THE FOLLOWING SUBCLAUSE AFTER 6.3 <CAST SPECIFICATION>:

6.3a <XML cast specification>

Function

Specify a data conversion whose source or target type is an XML type.

Format

```
<XML cast specification> ::=
     XMLCAST <left paren> < XML cast operand> AS
     <XML cast target> <right paren>

<XML cast operand> ::=
       <value expression>
     | <implicitly typed value specification>

<XML cast target>  ::=
       <domain name>
```

```
      | <data type>
```

Syntax Rules

1) Case:

a) If the <XML cast specification> is contained within the scope of an <XML binary encoding>, then let *XBE* be the <XML binary encoding> with innermost scope that contains the <XML cast specification>.

b) Otherwise, let *XBE* be an implementation-defined <XML binary encoding>.

2) Case:

a) If *XBE* specifies BASE64, then let *ENC* be an indication that binary strings are to be encoded in base64.

b) Otherwise, let *ENC* be an indication that binary strings are to be encoded in hex.

3) Case:

a) If <domain name> is specified, then let *TD* be the <data type> of the specified domain.

b) Otherwise, let *TD* be the specified <data type>.

4) At least one of the following shall be true:

a) *TD* is an XML type.

b) The <XML cast operand> is a <value expression> whose declared type is an XML type.

5) *TD* shall not be a collection type, a row type, a structured type or a reference type.

6) The declared type of the result of the <XML cast specification> is *TD.*

7) If *TD* is XML(UNTYPED DOCUMENT) or XML(ANY DOCUMENT), then the <XML cast operand> shall be either NULL or a <value expression> whose declared type is an XML type.

8) If the <XML cast operand> is a <value expression> *VE*, then let *SD* be the declared type of the <value expression>.

a) *SD* shall not be a collection type, a row type, a structured type or a reference type.

b) If *SD* is an XML type, and *TD* is an XML type, then the <XML cast specification> is equivalent to

```
CAST ( VE AS TD )
```

9) If <XML cast operand> is an <implicitly typed value specification> *ITVS*, then the <XML cast specification> is equivalent to

```
CAST ( ITVS AS TD )
```

10) If *TD* is character string type, then *TD* shall not specify <collate clause>. The declared type collation of the <XML cast specification> is the character set collation of the character set of *TD* and its collation derivation is *implicit*.

11) If <domain name> is specified, then let *D* be the domain identified by the <domain name>. The schema identified by the explicit or implicit qualifier of the <domain name> shall include the descriptor of *D*.

Access Rules

1) If *TD* is a distinct type, then let *STD* be the source type of *TD*, and let *AVE* be an arbitrary <value expression> of declared type *STD*. The Access Rules of Subclause 6.12, "<cast specification>" in ISO/IEC 9075-2 are applied to

```
CAST ( VE AS TD )
```

2) If *SD* is a distinct type, then let *SSD* be the source type of *SD*. The Access Rules of Subclause 6.12, "<cast specification>" in ISO/IEC 9075-2 are applied to

```
CAST ( AVE AS SSD )
```

3) If <XML cast target> is a <domain name>, then

Case:

a) If the <XML cast specification> is contained, without an intervening <SQL routine spec> that specifies SQL SECURITY INVOKER, then the applicable privileges of the <authorization identifier> that owns the containing SQL schema shall include USAGE on the domain identified by the <domain name>.

b) Otherwise, the current privileges shall include USAGE on the domain identified by the <domain name>.

General Rules

1) Let *V* be the value of the <value expression> simply contained in the <XML cast specification>.

2) If *V* is the null value, then the result of the <XML cast specification> is the null value, and no further General Rules of this Subclause are executed.

3) If *TD* is an XML type, then

a) Let *XMLV* be a <character string literal> whose value is the result of Subclause 9.16, "Mapping values of SQL data types to values of XML Schema datatypes" applied to *V* as the value of an SQL data type, *ENC* as the *ENCODING* and "absent" as the *NULLS*.

b) Let *TEMPV* be result of

```
XMLPARSE (CONTENT XMLV PRESERVE WHITESPACE)
```

c) If *TD* is XML(UNTYPED CONTENT) or XML(ANY CONTENT), then let *TV* be *TEMPV*.

d) If *TD* is XML(SEQUENCE), then:

i) Let *XMLT* be the result of applying the General Rules of Subclause 9.15 "Mapping SQL data types to XML Schema data types" to *SD*. Let *XST* be the XML Schema type that is represented by *XMLT*.

ii) Case:

1) If *SD* is a year-month interval type, then let *XSBT* be the XQuery simple type xdt:yearMonthDuration

2) If *SD* is a day-time interval type, then let *XSBT* be the XQuery simple type xdt:dayTimeDuration

3) If *XST* is an XML Schema built-in type, then let *XSBT* be *XST*.

4) If *XST* an XML Schema atomic type, then let *XSBT* be the XML Schema built-in type that *XST* is derived from.

iii) Let *XSBTN* be an XML 1.1 QName for *XSBT*.

iv) Let *XSC* be an XQuery static context created according to the Syntax Rules of Subclause 10.n, "Creation of an XQuery execution context", with the BNF non-terminal argument omitted. Let *XDC* be an XQuery dynamic context created according to the General Rules of Subclause 10.n, "Creation of an XQuery execution context".

v) Let *XSC* and *XDC* be augmented with an XQuery variable whose XML 1.1 QName is TEMP, whose XQuery formal type notation is "document { text ? }", and whose value is *TEMPV*.

vi) Let *TV* be the result of the XQuery evaluation with XML 1.1 lexical rules, using *XSC* and *XDC* as the XQuery expression context, of the XQuery expression

```
$TEMP cast as XSBTN
```

If this XQuery evaluation raises an XQuery error, then an exception condition is raised: *XQuery error.*

e) *TV* is the result of the <XML cast specification>.

4) If *SD* is an XML type and *TD* is not an XML type, then:

a) Case:

i) If the <XML cast target> is a <domain name>, then let *SQLT* be the <data type> of the identified domain.

ii) If the <XML cast target> identifies a distinct type, then let *SQLT* be the source type of the distinct type.

iii) Otherwise, let *SQLT* be <data type>.

b) Let *XV* be the result of applying the General Rules of Subclause 10.n, "Removing XQuery document nodes from an XQuery sequence", with *V* as the *SEQUENCE*.

c) Let *AV* be the result of applying the XQuery function fn:data to *XV*.

d) If *AV* is the empty sequence, then the result is the null value and no further General Rules of this Subclause are executed.

e) Let *XT* be the XML Schema type obtained by applying the General Rules of Subclause 9.15, "Mapping SQL data types to XML Schema data types" with *SQLT* as the SQL data type, *ENC* as the *ENCODING*, and "absent" as *NULLS.*

f) Let *XMLT* be a XML 1.1 QName for *XT.*

NOTE nnn: *XMLT* may be in one of the built-in namespaces denoted by the prefixes xs: or xdt:, or it may be in an implementation-dependent namespace, not necessarily available to the user.

g) Let *XSC* be an XQuery static context created according to the Syntax Rules of Subclause 10.n, "Creation of an XQuery execution context", with the BNF non-terminal argument omitted.

h) Let *XDC* be an XQuery dynamic context created according to the General Rules of Subclause 10.n, "Creation of an XQuery execution context".

i) Let *XSC* and *XDC* be augmented with an XQuery variable whose QName is TEMP, whose XQuery formal type notation is "xdt:untypedAtomic", and whose value is *AV.*

j) Let *BV* be the result of the XQuery evaluation with XML 1.1 lexical rules, using *XSC* and *XDC* as the XQuery expression context, of the XQuery expression

```
$TEMP cast as XMLT
```

If this XQuery evaluation raises an XQuery error, then an exception condition is raised: *XQuery error*

k) *BV* is an XQuery atomic value.

NOTE: the following rules assume that the value space of XQuery atomic types is the same as the value space of corresponding SQL types. That is, it is possible to treat *BV* as a value of an SQL type. For example, if *BV* is of type xs:integer, then *BV* is a (mathematical) integer, and as such, it may also be the value of an exact numeric type with scale 0. See Subclause 4.9.11, "Mapping XQuery atomic values to SQL values".

Case:

i) If *XT* is xs:string or derived from xs:string, then let A be an SQL variable of character type whose character repertoire is Unicode and whose value is *BV.*

ii) If *XT* is xs:hexBinary or xs:base64Binary,or derived from xs:hexBinary or xs:base64Binary, then let A be an SQL variable of binary string type whose value is *BV.*

iii) If *XT* is xs:decimal or derived from xs:decimal, then let A be an SQL variable of exact numeric type whose value is *BV.*

iv) If *XT* is xs:float or xs:double, or derived from xs:float or xs:decimal, then:

1) If *AV* is INF, -INF or NaN, then an exception condition is raised: *data exception — numeric value out of range*

2) Let A be an SQL variable of approximate numeric type whose value is *BV.*

v) If *XT* is xs:date or derived from xs:date, then:

1) If the XQuery datetime normalized value component of *AV* is not positive, or if the XQuery datetime timezone component of *AV* is not the XQuery empty sequence, then an exception is raised: *data exception — invalid datetime format.*

2) Let A be an SQL variable of declared type *SQLT* whose value is *BV*.

vi) If *XT* is xs:dateTime or derived from xs:dateTime, then:

1) If the XQuery datetime normalized value component of *AV* is not positive, then an exception is raised: *data exception — invalid datetime format.*

2) If *SQLT* is TIMESTAMP WITHOUT TIME ZONE then:

A) If the XQuery datetime timezone component of *AV* is not the XQuery empty sequence, then an exception is raised: *data exception — invalid datetime format.*

B) Let A be an SQL variable of declared type TIMESTAMP WITHOUT TIME ZONE whose value is *BV*.

3) If *SQLT* is TIMESTAMP WITH TIME ZONE then:

A) If the XQuery datetime timezone component of *AV* is the XQuery empty sequence, then an exception is raised: *data exception — invalid datetime format.*

B) Let A be an SQL variable of declared type TIMESTAMP WITH TIME ZONE whose value is *BV*.

vii) If *XT* is xs:time or derived from xs:time, then

Case:

1) *SQLT* is TIME WITHOUT TIME ZONE, then:

A) If the XQuery datetime timezone component of *AV* is not the XQuery empty sequence, then an exception is raised: *data exception — invalid datetime format.*

B) Let A be an SQL variable of declared type TIME WITHOUT TIME ZONE whose value is the XQuery datetime normalized value of *BV*.

2) If *SQLT* is TIME WITH TIME ZONE, then:

A) If the XQuery datetime timezone component of *AV* is the XQuery empty sequence, then an exception is raised: *data exception — invalid datetime format.*

B) Let A be an SQL variable of declared type TIME WITH TIME ZONE whose value is *BV* (more precisely, whose UTC component is the XQuery datetime normalized value of *BV* and whose timezone component is the XQuery datetime timezone component of *BV*).

viii) If *XT* is xdt:yearMonthDuration or derived from xdt:yearMonthDuration, then let A be an SQL variable of year-month type whose value is *BV.*

ix) If *XT* is xdt:dayTimeDuration or derived from xdt:dayTimeDuration, then let A be an SQL variable of day-time type whose value is *BV.*

x) If *XT* is xs:boolean or derived from xs:boolean, then let A be an SQL variable of type BOOLEAN whose value is *BV.*

l) The result of the <XML cast specification> is the result of

```
CAST (A AS SQLT)
```

NOTE: this may raise an exception if the value of A does not conform to constraints of *SQLT.* It may also perform rounding or truncation, etc., in order to produce a value of type *SQLT.*

Conformance Rules

None

5.7 New Subclause 12.n, <user-defined cast definition>

1. ADD THE FOLLOWING SUBCLAUSE TO CLAUSE 12, "SCHEMA DEFINITION AND MANIPULATION"

12.n <user-defined cast definition>

This Subclause modifes Subclause 11.53, "<user-defined cast definition>", of ISO/IEC 9075-2.

Function

Define a user-defined cast

Syntax Rules

1) Neither *SDT* nor *TDT* shall be a distinct type whose source type is XML.

Access Rules

None

General Rules

None

Conformance Rules

None

5.8 Changes to 13.1, Calls to an <externally-invoked procedure>

1. Harmonize the Ada header with the changes to Subclause 23.1, SQLSTATE.

5.9 Changes to 23.1, SQLSTATE

1. Add the following exceptions to Table 12, "SQLSTATE class and subclass values":

Category	Condition	Class	Subcondition	Subclass
X	***data exception***	**22**	***not an XQuery document node***	*editor's choice*
			invalid operand for XML concatenation	*editor's choice*
			not an XQuery document node	*editor's choice*
			invalid XQuery context item	*editor's choice*
			XQuery sequence serialization error	*editor's choice*
W	***warning***	**01**	***XQuery document node was stripped***	*editor's choice*
X	***XQuery error***	*editor's choice*	***(no subclass)***	**000**

5.10 Changes to Annex B, Implementation-defined elements

1. Add the following items:

n.1) Subclause 6.3, "<cast specification>"

a) The result of CAST (EMPTY AS XML(UNTYPED CONTENT)) or CAST (EMPTY AS XML(ANY CONTENT))is an XQuery document node with an implementation-defined base-uri and document-uri properties.

n.2) Subclause 6.3a, "<XML cast specification>"

a) The default encoding of binary strings (either base64 or hex) is implementation-defined.

5.11 Changes to Annex D, Incompatibilities

1. ADD THE FOLLOWING NEW <RESERVED WORD>S:

XMLCAST

6. Checklist

Concepts	
Access Rules	
Conformance Rules	
Lists of SQL-statements by category	
Table of identifiers used by diagnostics statements	
Collation derivation for character strings	
Closing Possible Problems	
Any new Possible Problems clearly identified	
Reserved and non-reserved keywords	
SQLSTATE tables and Ada package	
Information and Definition Schemas, including short-name views	
Implementation-defined and –dependent Annexes	
Incompatibilities Annex	
Embedded SQL and host language implications	
Dynamic SQL issues: including descriptor areas	
CLI issues	

- End of paper -