XQuark Bridge 1.1

Mapping Reference Guide

XQUARK BRIDGE 1.1

MAPPING REFERENCE GUIDE

Document version 1.0

FOREWORD

This document refers to following W3C standards :

[XML Schema Part 1] *XML Schema Part 1: Structures.* W3C Recommendation 2 May 2001. See http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/

[XML Schema Part 2] XML Schema Part 2: Datatypes. W3C Recommendation 2 May 2001. See http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

This section describes the status of this document from W3C at the time of its publication. Other documents may supersede this document. The latest status of this document series is maintained at the W3C.

Table of contents

Part

Abstract

The XQuark Mapping Language is a language that allows the definition of a mapping between an XML Schema and a relational database schema. This document is the language reference manual.

Part

2

Introduction

Overview

XQuark Bridge provides a powerful and flexible XML data import mechanism, called schema-based mapping, to import data contained in XML documents into existing, user-defined relational tables.

The schema-based mapping requires XML documents to be conformant with one or several user-defined XML Schemas [XML Schema Part 1][XML Schema Part 2]. The mapping language provides mechanisms to:

- Associate XML Schema type definitions, elements and attributes declarations to relational tables and columns. When importing an XML document conformant with the mapped XML Schemas, element and attribute content will be stored in relational structures according to this association.
- Explicitly store in relational columns the implicit parent-child relations which appear in XML documents
- Check the presence of tuples (rows) before insertion, and update them at user option
- Transform XML data into internal reference code at insertion time
- Execute user-defined Java methods to produce column values (e.g. internal primary keys).

In addition, XQuark Bridge will check at declaration time the correctness of the mapping specification, e.g. the compatibility between XML types declared in the XML Schema specification and SQL types declared in the database, and more generally the feasibility of the mapping.

By imposing a model to documents and detecting mapping incompatibilities at declaration time, this approach ensures that schema-valid documents will be stored without errors in the relational database.

Mapping directives are defined in an XML document that uses its own XML vocabulary. The mapping directives reference XML Schema declarations, as well as relational database information. The mapping language is described in The mapping language.

Notations and conventions

This section introduces the typography used to present technical information in this manual.

The XQuark mapping language use a specific XML vocabulary to represent mapping directives as element information items.

In the XML representation, bold-face attribute names indicate a required attribute information item, and the rest are optional. Where an attribute information item has an enumerated type definition, the values are shown separated by vertical bars; if there is a default value, it is shown following a colon.

The allowed content of the information item is shown as a grammar fragment, using the Kleene operators `?` (0 or 1 occurrence), `*` (0 or more occurences) and + (1 or more occurrences).

```
<mapping
  schemaLocation = anyURI
  version = string : 1.0>
Content: (element | map)*
</mapping>
```

The XML Schema that formally defines the mapping language is provided at the end of this manual, in Appendix A - XML Schema for the mapping language. This schema is associated to the namespace identified by the following URI: `http://www.xquark.org/Bridge/1.0/Mapping`.

Java interfaces are introduced in this manual as a set of method signatures, using the standard Java notation:

```
Object getValue(StorageContext)
```

References

[XML Schema Part 1] *XML Schema Part 1: Structures*. W3C Recommendation 2 May 2001. See http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/

[XML Schema Part 2] *XML Schema Part 2: Datatypes*. W3C Recommendation 2 May 2001. See http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

Part

3

Concepts

Data model

The data model used to define the mapping is the standard XML data model, where:

- The XML document has a single document element (the root of the document tree).
- Each element can have a number of mandatory or optional mono-valued attributes (i.e. each attribute can have 0 or 1 occurrence), whose order is not significant. Attributes can be typed, according to a predefined XML Schema simple type or a user-defined type derived from a predefined type.
- Each element can contain (i) other elements only, (ii) character data only, (iii) nothing, or (iv) a mixture of elements and character data. The order of sub-elements and character data is generally significant.
- A sub-element can have 0 (optional), 1 (mono-valued) or several (multi-valued) occurrences within its parent element.

When an element contains only character data, this data can be typed, according to a predefined XML Schema simple type or a user-defined type derived from a predefined type. When the character data is not typed, or when it occurs in a mixed-typed element, the data is considered as a string.

XML types

XML types can be of two kinds:

- Simple types are those used to associate a data type to XML character data, which otherwise is considered as a string. Simple types are relevant only to the character contents of an XML document, and can be associated only to attributes and elements with no children (also referred to as character-data elements or simple-type elements). XML Schema [XML Schema Part 2] defines 19 primitive types and 25 predefined simple types derived from those primitive types. XML Schema also provides means to derive new simple types from existing types, by restricting their value space.
- Complex types are those used to specify the structure of an element containing attributes, character data and/or other elements. Those

complex types are structured user-defined types built by aggregating elements of simple or previously defined complex types. XML Schema [XML Schema Part 1] also provides means to create new complex types by derivation from previously defined ones. As such, a complex type is similar to a class definition in object-oriented languages, and, to some extent, a SQL table definition (with the difference that SQL tables can usually not aggregate nor derive from previously defined tables).

In XML Schema, type definitions are used to specify element and attribute declarations, which are the association of a name (the element or attribute name) and a type. For instance, the complete structure of an XML document can be specified by a single complex type associated to the document element declaration.

Both type definitions and element or attribute declarations can be involved in mapping specifications.

Mapping XML concepts to relational concepts

Specifying an XML-to-SQL mapping for an XML document is equivalent to specifying a mapping between the top-level document element declaration and existing SQL structures. In the case where only sub-trees of the document must be mapped to existing SQL structures, the considered element declarations will be those corresponding to the sub-trees to be mapped.

When several element declarations refer to the same complex type, it is usually easier to define the mapping for the shared complex type, and reference this mapping in the element declaration mappings.

An XML document contains several types of information that are relevant to the XML-to-SQL mapping:

- Contents: the content of an XML document is composed of all character data and attribute values. This content can thus be found in (i) attributes, (ii) character-data elements and (iii) mixed-type elements. In cases (i) and (ii), the contents will generally be typed by associating a simple type to the attribute or element declaration. Such contents will typically be stored as column values in the relational tables. In case (iii), as the character data can occur within a mixed-type element in multiple places, it is generally not possible to store this information in specific table columns.
- Structure: each complex-type element will typically be mapped onto one or several existing SQL tables and be stored as tuples in those tables. Such a mapping also specifies a scope that will allow the element children to be mapped onto the table columns, as described in Mapping simple-type elements and attributes to SQL columns.

- Aggregation links: the structure of the XML document specifies an implicit aggregation relation between a given element and all the elements and attributes it contains. This relation can be either 1-1, when the child element occurs once in the parent element, or 1-N, when the child element occurs multiple times. In the case of child attributes, the relation is always 1-1, as attributes cannot be multi-valued. The way aggregation relations are represented in the relational model depends on the type and number of occurrences of each child:
  - *Implicit relational aggregation: Mono-valued character-data element children and attribute children[1] are usually mapped on columns in the table associated to their parent element. In this case, the child element content or the attribute value is stored as the column value.*
  - *Foreign key in the relational table associated to the child: In the case of multi-valued element children, the children are necessarily mapped on separate relational tables (even in the case of simple-type child elements). In this case, the 1-N aggregation relations will typically be stored as foreign keys in those tables. Note that this approach can also be used in the case of mono-valued child elements and attributes.*
  - *Foreign key in the relational table associated to the parent: In the case of a mono-valued child element or attribute mapped on a separate relational table, it is possible to store to represent the aggregation as a foreign key stored in the parent table, instead of in the child table. This is only possible because the parent-child relation is 1-1.*
  - *Omitted aggregation: When a mono-valued child element is itself of a complex type, its own children can directly be associated to their grandparent element, according to the rules described here, thus effectively flattening the XML structure by omitting pure structural information. This approach can be applied recursively, as long as the intermediate elements are mono-valued in their parent.*
- Association links: XML defines several ways to specify association relations between elements of the same or different documents. The most usual way, inherited from the DTDs, is to attach a unique ID to an element using an ID-typed attribute, and refer to this element in other elements of the same document using IDREF- or IDREFS-typed attributes. A similar, more complex mechanism is defined in the XML Schema specification, which allows the definition of keys and keyrefs in the document schema specification. Finally, the XLink and XPointer specifications provide a mechanism for establishing links between parts of different documents. In all cases, the association relations are explicit and use a primary key – foreign key approach easily mapped onto SQL structures. Therefore, for storage purposes, keys and keyrefs can be considered as any other attributes.

The following table summarizes the main mapping principles:

XML Concept	Mapping type	SQL Concept	Comments

[1] Attributes are always mono-valued and associated to a simple type

XML Concept	Mapping type	SQL Concept	Comments
Complex type Complex-type element	Table mapping	Table	Stores elements as tuples and defines a scope for storing attributes and child elements into table columns
Complex type of mixed content Mixed-type element	Table mapping	Table	Stores elements as tuples and defines a scope for storing attributes and child elements into table columns. Character data occurring within the element will not be stored.
Attribute Simple-type element	Column mapping	Column	Stores the attribute value or character data contents into the column.
Attribute Simple-type element	Table mapping	Table	Stores the attribute value or character data contents into a new tuple in the table. This type of mapping is especially useful in case of multi-valued simple-type elements.
1-N parent-child aggregation	Column mapping	Foreign key	Stores a reference to the table holding parent information into a column of the table holding child information
1-1 parent-child aggregation	Column mapping	Foreign key	Stores a reference to the table holding child information into a column of the table holding parent information
1-1 parent-child aggregation (in the case of a complex-type child)	Column mappings	Columns	Stores the contents of the child attributes and elements in columns of the parent table. This approach can be generalized to any level in the hierarchy.

Table 1– Mapping principles

The above table does not specify conditions for a given mapping to be valid. Those conditions are discussed in the next sections.

4

Rules for mapping simple XML types

Standard conversions

Simple types can be used to specify the type of an attribute or of an element containing only character data. The XML Schema datatypes specification [XML Schema Part 2] defines 19 atomic primitive datatypes, and 25 derived datatypes (22 atomic and 3 list types[2]) that can be used as simple types. Attributes and elements having a simple type will normally be mapped onto a column in an SQL table. The XML-to-SQL mapping is able to verify that the datatypes used in a XML Schema are compatible with the SQL types used in the tables and columns in which schema-compliant documents will be stored. For portability reasons, the considered SQL types will be those defined in JDBC.

The following table shows the correspondence between the 44 XML Schema predefined datatypes and the JDBC types.

XML datatype	JDBC type	Comments
string	VARCHAR	The size constraints defined in the XML and relational models must be compatible.
normalizedString	VARCHAR	See *string*.
token	VARCHAR	See *string*.
Name	VARCHAR	See *string*.
NCName	VARCHAR	See *string*.
ID	VARCHAR	See *string*.
IDREF	VARCHAR	See *string*.
ENTITY	VARCHAR	See *string*.
NMTOKEN	VARCHAR	See *string*.
IDREFS	VARCHAR	*list* type
ENTITIES	VARCHAR	*list* type
NMTOKENS	VARCHAR	*list* type
anyUri	VARCHAR	See *string*.

[2] List types are obtained from base types through *list* derivation. In practice, they correspond to a whitespace-separated list of values compliant with the list base type.

XML datatype	JDBC type	Comments
QName	VARCHAR	QNames are stored as {namespaceURI}localName, i.e. the prefix is replaced by its namespace value and curly brackets are added as separators. The size constraints defined in the XML and relational models must be compatible.
NOTATION	VARCHAR	NOTATIONs are handled as QNames.
boolean	BIT	As BIT is not widely implemented by databases, booleans are generally mapped on small integers.
float	REAL	
double	DOUBLE	
decimal	NUMERIC	The precision and scale constraints defined in the XML and relational models must be compatible. *decimal* values can have arbitrary precision in XML. The mapping might induce precision losses when the XML value has a precision higher than the maximum precision supported in the database.
integer	NUMERIC with scale=0	See *decimal*.
long	BIGINT	
int	INTEGER	
short	SMALLINT	
byte	TINYINT	
nonNegativeInteger	NUMERIC with scale=0	See *decimal*.
unsignedLong	NUMERIC with scale=0	See *decimal*.
unsignedInt	BIGINT	
unsignedShort	INTEGER	
unsignedByte	SMALLINT	
positiveInteger	NUMERIC with scale=0	See *decimal*.
nonPositiveInteger	NUMERIC with scale=0	See *decimal*.
negativeInteger	NUMERIC with scale=0	See *decimal*.
base64Binary	VARBINARY	The size constraints defined in the XML and relational models must be compatible.

XML datatype	JDBC type	Comments
hexBinary	VARBINARY	The size constraints defined in the XML and relational models must be compatible.
duration	VARCHAR	*duration* values conceptually represent a sextuple (year, month, day, hour, minute, second), which cannot be mapped into another simpler representation (such as a number or a date) without loss of information. As JDBC does not define a standard type for holding such information, *duration* values are kept as strings.
dateTime	TIMESTAMP	
time	TIME	
date	DATE	
gYearMonth	DATE	*gYearMonth* values represent an incomplete date, namely a specific month in a specific year. In such a case, the complete date is obtained by using as default values for missing fields the corresponding fields in the date January 1st, 1970 at 00:00:00.
gYear	DATE	See gYearMonth.
gMonthDay	DATE	See gYearMonth.
gDay	DATE	See gYearMonth.
gMonth	DATE	See gYearMonth.

Table 2– XML Schema to table correspondence

For some types, the above table does not provide enough information: all VARCHAR and VARBINARY types should be associated with a maximum length. In fact, in an existing SQL database, all fields having those types will have a maximum length. Similarly, NUMERIC types can have precision and scale attributes, although default values are used for those attributes when they are not specified. Therefore, when designing an XML Schema to be mapped onto an existing database, the designer should derive new simple types enforcing the size or precision constraints of the database. Those constraints can be set in the derived types by using the *length* and *maxLength* facets[3] for size constraints, and the *precision* and *scale* facets for numeric types.

In the case of VARCHAR and VARBINARY, the insertion of a document in the database might fail, due to overflow, if the size constraints are not enforced in the XML Schema. XQuark Bridge will warn the user at mapping loading time when unconstrained XML types are mapped onto columns with limited size. It will then be up to the designer to constrain the XML

[3] A facet is a constraining property of a simple XML type. See the XML Schema specifications for details.

type or take the chance of a runtime error when storing a document. Note that it is a mapping error if size constraints are present in the XML Schema and not compatible with size constraints in the database.

Similarly, in the case of numerical types, XQuark Bridge will warn the user when unconstrained XML types are mapped onto columns with limited precision and scale, which can result in runtime error when storing a document. The mapping tool will also issue a warning when precision losses can result from a given mapping, and produce an error when data losses can occur (e.g. when mapping a 64-bit integer onto a 16-bit column).

Any data that can be stored in a VARCHAR column can also be stored in a CHAR column. The same size constraints apply. When the data is shorter than the size of the column, it will be padded with whitespace at the end.

Mapping XML data on large columns (LONGVARCHAR, LONGVARBINARY, CLOB, BLOB) is not supported in this release.

Implicit conversions

The previous section has shown the correspondence between the XML types and the most constraining corresponding SQL types. Obviously, it is possible to map a given XML type on a more general SQL type than the one shown in the above table. This type widening (only applicable to numerical types) is called an implicit conversion. The legal implicit type conversions for numerical types are the usual ones:

- Anything that can be stored in a BIT can be widened to a TINYINT, a TINYINT can be widened to a SMALLINT, a SMALLINT to an INTEGER, an INTEGER to a BIGINT, and a BIGINT to a NUMERIC (if its precision is large enough).
- Anything that can be stored in a BIT, TINYINT, SMALLINT, INTEGER, BIGINT can be stored in a REAL or DOUBLE.
- Anything that can be stored in a REAL can be stored in a DOUBLE.
- Conversions from NUMERIC to REAL and DOUBLE and from REAL and DOUBLE to NUMERIC are possible, but may induce precision losses when the NUMERIC precision and scale are not compatible with the implicit precision and scale of REAL and DOUBLE.

Another case of implicit conversion occurs when an XML type is mapped onto a CHAR or VARCHAR column: in this case, the string value as found in the XML document will be stored in the column[4]. Finally, XQuark Bridge also supports less usual conversions, such as conversions from XML string

[4] When the XML type implicitly or explicitly defines size constraints, they will be checked against the column size.

to VARBINARY and from gYear, gMonth and gDay to numerical column types.

The following table summarizes the legal conversions:

	CHAR	VARCHAR	LONGVARCHAR	BIT	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	NUMERIC	DECIMAL	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	CLOB	BLOB
string	2	2													2						
normalizedString	2	2													2						
token	2	2													2						
Name	2	2													2						
NCName	2	2													2						
ID	2	2													2						
IDREF	2	2													2						
ENTITY	2	2													2						
NMTOKEN	2	2													2						
IDREFS	2	2													2						
ENTITIES	2	2													2						
NMTOKENS	2	2													2						
anyUri	2	2																			
QName	2	2																			
NOTATION	2	2																			
boolean	4	4		1	1	1	1	1	1	1	1	1	1								
float	4	4			5	5	5	5	1	1	1	3	3								
double	4	4			5	5	5	5	5	1	1	3	3								
decimal	2	2			5	5	5	5	5	5	5	2	2								
integer	2	2			2	2	2	2	2	2	2	2	2								
long	4	4						1	1	1	1	1	1								
int	4	4					1	1	1	1	1	1	1								
short	4	4				1	1	1	1	1	1	1	1								
byte	4	4			1	1	1	1	1	1	1	1	1								
nonNegativeInteger	2	2			2	2	2	2	2	2	2	2	2								
unsignedLong	4	4							1	1	1	1	1								
unsignedInt	4	4						1	1	1	1	1	1								
unsignedShort	4	4					1	1	1	1	1	1	1								
unsignedByte	4	4				1	1	1	1	1	1	1	1								
positiveInteger	2	2			2	2	2	2	2	2	2	2	2								
nonPositiveInteger	2	2			2	2	2	2	2	2	2	2	2								
negativeInteger	2	2			2	2	2	2	2	2	2	2	2								
base64Binary	2	2													2						
hexBinary	2	2													2						
duration	1	1																			
dateTime	4	4																	1		
time	4	4																1			
date	4	4															1		1		
gYearMonth	4	4															1		1		
gYear	4	4				1	1	1	1	1	1	1	1				1		1		
gMonthDay	4	4															1		1		
gDay	4	4			1	1	1	1	1	1	1	1	1				1		1		
gMonth	4	4			1	1	1	1	1	1	1	1	1				1		1		

Table 3 – Legal conversions

The meaning of the codes used in the table is given below:

1. The validity of the conversion can be verified at mapping definition time.
2. When no additional XML facets are associated to the XML type, it is not possible to verify the legality of the conversion at mapping definition time. The conversion might fail at run-time, due to overflow.
3. Precision loss can happen during the conversion.
4. The conversion is safe, but the data type is lost during the conversion.
5. When no additional XML facets are associated to the XML type, it is not possible to verify the legality of the conversion at mapping definition time. Both precision loss and data overflow might occur during the conversion.

`list` *and* `union` *types*

The XML Schema specification allows `list` and `union` simple types to be derived from atomic simple types. List types are always stored as strings (VARCHAR or CHAR), using as value the whitespace-separated list of values. Note that because the *length* and *maxLength* facets apply to the number of items in the list rather than the number of characters in the data, it is usually not possible to verify the size constraints at mapping definition time in the case of a *list* type.

Union types can be converted to the most restricting common SQL type, as defined in the implicit conversions section above. In any cases, union types can always be stored as strings (VARCHAR or CHAR), as all types can be implicitly converted to strings.

5

Rules for mapping complex XML types

Mapping complex types and complex-type elements to SQL tables

Complex types are used to define the structure of XML documents. Complex-type elements contain structured information that is used to fill columns in the existing SQL tables. Complex types and complex-type elements are mapped onto SQL tables in two steps:

1. A complex type or complex-type element declaration is associated to an SQL table (through a table mapping) to define the scope in which child elements or attributes will be mapped onto the table columns. Each complex-type element will become a tuple in its associated table.
2. Child elements or attributes are associated either to a column of the table associated to their parent or to one of their ancestors (through a *column mapping*), or to another SQL table (as in step 1), in which case step 2 is recursively applied, thus resulting in nested table mappings. Nested table mappings correspond to parent-child aggregations in the XML document, and may involve the specification of foreign relations between the tables. The rules that control the validity of a column mapping depend on the type of the element or attribute, its number of occurrence (optional, mono-valued or multi-valued) and the column information. These rules are detailed below.

The following figure represents the mapping of complex XML elements into relational tuples. Each complex-type element is stored as a tuple in the relational tables.

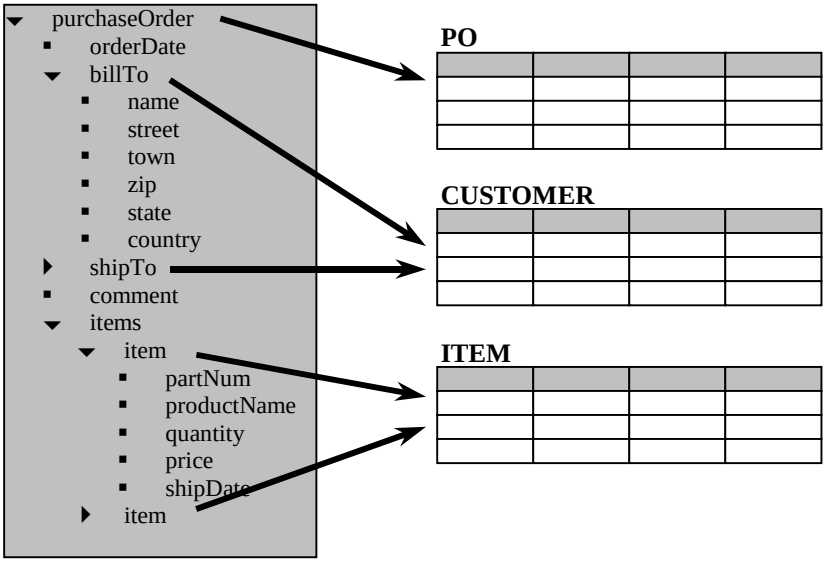

Figure 1 –Mapping example of complex XML types

This example is based on a small XML document describing a purchase order. This purchase order contains two customer references (a billing address and a shipping address) and a number of purchased items. In the figure, the XML document is represented as a tree, with expanded complex elements identified by a downward triangle, non-expanded complex elements by a rightward triangle, and leaf elements by a small square.

Mapping simple-type elements and attributes to SQL columns

Simple-type elements and attributes are usually mapped on table columns. A column mapping specifies that the *value* of the simple-type element or attribute is to be stored into the designated table column. As mentioned previously, the column is generally part of the table associated to the element or attribute parent element, but may also be part of the table associated to an ancestor of the element or attribute.

In order to be valid, such a column mapping must verify the following conditions:

- A simple-type element mapped onto a column cannot be multi-valued. If the column mapping refers to a table associated to an ancestor of the element or attribute to be mapped, none of the intermediate elements can be multi-valued either. This ensures that only a single value will ever be stored in the column.
- If a simple-type element or attribute is optional with no default value, then the associated column must be nullable. Similarly, if the column mapping refers to a table associated to an ancestor of the element or attribute to be mapped, and if any intermediate element is optional, then the associated column must be nullable. This ensures that absent

optional values will not result in errors when the document is inserted in the database.

- The simple-element or attribute XML type must be convertible to the SQL type of the column, as defined in the previous section.

The following figure represents some column mappings for the previous example.

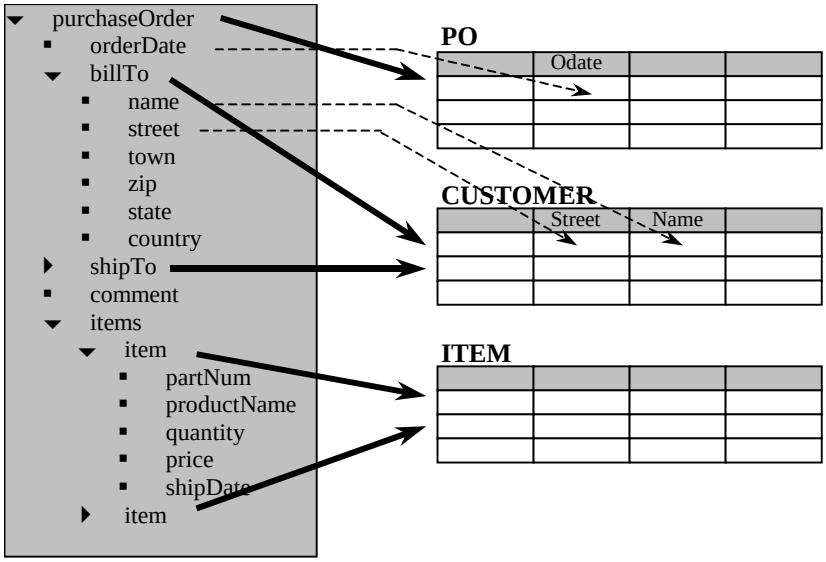

Figure 2 – Column mapping example

Mapping simple-type elements and attributes to SQL tables

In some cases, it may be necessary to map a simple-type element or an attribute to a separate table. This may occur for instance when a multi-valued simple-type element appears in a parent element. As for complex-type elements, each mapped element or attribute will become a tuple in its associated table.

However, as simple-type elements and attributes have no structured contents, column mappings cannot be specified in the usual way. Instead, *generators* (described in Generators) are used to specify the contents of the table columns.

Storing parent-child aggregation in the child table

When parent and child elements are stored in separate tables, it is often useful to represent the parent-child aggregation that is implicit in the XML document as an explicit association between the parent table and the child

table. This allows the SQL view of the document to reflect as closely as possible the XML structure.

The implicit parent-child aggregation can be either 1-1 (case of a mono-valued child) or 1-N (case of a multi-valued child). In the latter case, the corresponding relation must be stored in a column of the child table, while in the former case, it can be stored in a column of either the child or the parent table. This section describes storage in the child table, while the next one deals with storage in the parent table.

As the aggregation is implicit, the document does not generally contain any data that represents the link between the parent and the child. Therefore, it is usually necessary to generate a value to be stored as a foreign key in the child table column. In many cases, this value will be obtained by copying the corresponding primary key previously stored in the parent table. This special kind of column mapping is described in details in Column references.

Although the above discussion uses the concepts of primary and foreign keys, it is not required that the columns in the parent and child tables be linked through a formal foreign relation (i.e. a relation declared in the relational model). The only requirement is that the value obtained from the parent table can be legally stored in the child table column.

The following figure shows a possible way of storing in the relational table the 1-N aggregation between the purchase order and each ordered item in the case of the purchase order document:

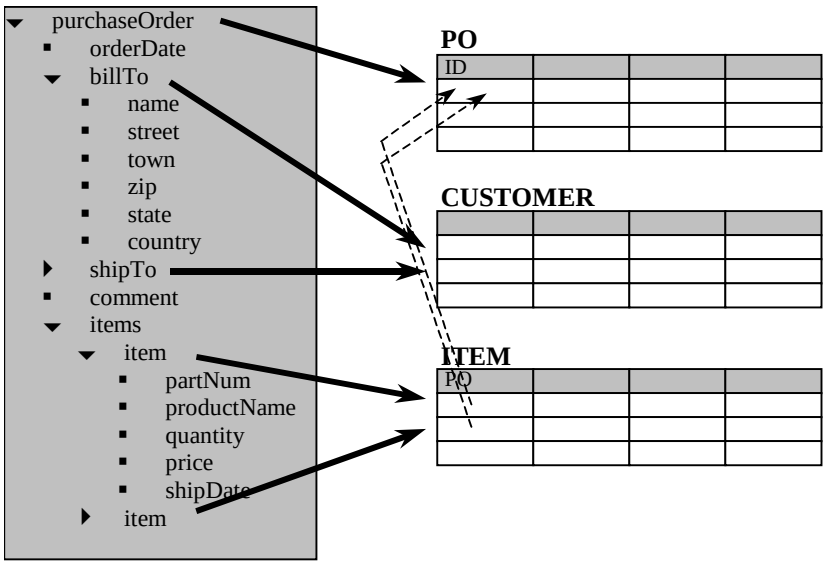

Figure 3 – Aggregation example

In this example, each tuple corresponding to an ordered item stores in its `PO` column a reference to the `ID` column of the `PO` tuple corresponding to its purchase order parent.

Storing 1-1 parent-child aggregation in the parent table

When the parent-child aggregation cardinality is 1-1, it is possible to store the relation as a foreign key in the parent table. This means that a complex-type child element will be mapped at the same time to a child table to generate the new tuple representing the element, and to a column in the parent table to store the reference to the newly created tuple. This reference is obtained by copying the corresponding primary key stored in the child table. The name of the column holding this key is a parameter of the dual table and column mapping.

Here also, there is no need for the parent and child tables to be linked by a formal foreign relation. The validity conditions for this mapping are as follows:

- The complete path between the parent element and the child element must consist of mono-valued elements.
- If one of the elements in this path is optional, then the parent table column must be nullable.
- The value obtained from the child table column must be legally storable in the parent table column.

The following figure illustrates the use of this mapping to store a reference to both billing and shipping address in a purchase order:

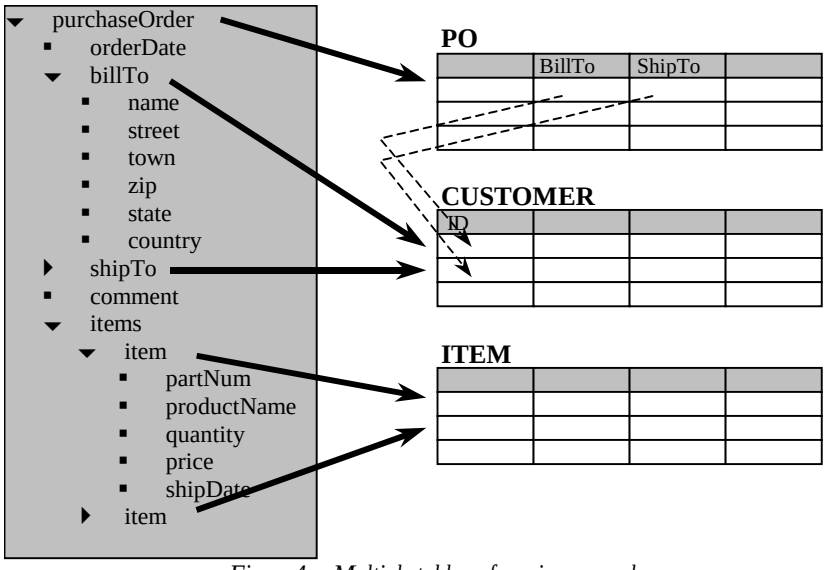

Figure 4 – Multiple tables referencing example

In this example, the tuple corresponding to the purchase order stores in its `BillTo` and `ShipTo` columns a reference to the `ID` column of the `CUSTOMER` tuples corresponding respectively to its `billTo` and `shipTo` child elements.

Controlling the insertion of element data

Unlike XML data, relational data is often normalized, i.e. the same data is generally not inserted twice in the same table. When mapping XML data onto relational tables, it is therefore sometimes necessary to control normalization at insertion time. In the above example, it might be necessary, for instance, to check that an address is not already present in the `CUSTOMER` table before inserting the contents of the `billTo` and `shipTo` elements. The mapping language provides a way to specify that a tuple must not be created if the content of the XML element to be inserted matches a subset of columns in the table. The list of columns to be used for the match can be specified in the language.

It is also possible to specify that a matched tuple will be updated, i.e. that the columns not used for the match will receive their corresponding values from the XML element..

Part 6

Generators

Rationale for generators

As explained previously, mapping XML documents onto relational tables may require the insertion in the table columns of information which is either non-relevant or implicit in the XML documents, and which therefore may not be found in those documents. Examples of such information include:

- Internal identifiers (e.g. primary keys) used in the relational tables
- Foreign keys used to represent the implicit aggregation between parent and child elements in a XML document

The mechanism that allows the user to specify a different source of information to obtain values to be stored into columns is called a *generator*. There are three types of generators, which are described in more details below:

- User-defined generators
- System variables
- Column references

User-defined generators

User-defined generators are Java classes that can be attached to a table column to generate a value. Each class must implement the `UserGenerator` interface. This interface consists of two methods,

```
Object getValue(StorageContext)

String getXMLType()
```

that return respectively a Java `Object` containing the value to be stored in the table column, and a string describing the XML type of the generated value. The object should be convertible to the column SQL type as defined in the JDBC specification.

The `StorageContext` interface provides access to some internal XQuark Bridge system variables, which can be used by the generator class to produce its value. Those variables include those described in the next section, as well

as the JDBC connection used by the storage algorithm. Access method details can be found in the `StorageContext` API documentation. Both interfaces are defined in the `org.xquark.xml.mapping` package.

A typical use of user-defined generators is the generation of unique primary keys for tuples constructed from XML documents. This might involve for instance the use of a SEQUENCE or any similar SQL mechanism to generate unique values.

Classes implementing the user generators must be available in the CLASSPATH at mapping loading time.

Pre-defined generators

Some standard user generators are provided with XQuark Bridge, for instance to generate keys for the rows created. Have a look to the API documentation (Javadoc) of the `org.xquark.mapping` package to check available generators.

System variables

System variables represent a special kind of generator that returns the current value of some internal variables. Those variables are generally associated to the current XML element being stored (i.e., the element to which the table currently being filled is associated).

Five system variables can be accessed:

- $NodeValue: this variable contains a string that represents the XML value of the current element or attribute. This variable is set to null for elements which do not contain only character data.
- $NodeRank: this variable contains an integer that represents the rank of the current element or attribute within its parent element.
- $LocalName: this variable contains a string that represents the local name of the current element or attribute.
- $NamespaceURI: this variable contains a string that represents the namespace URI of the current element or attribute.
- $QName: this variable contains a string that represents the qualified name of the current element or attribute.

System variables are particularly useful to fill table columns in the case where a simple-type element or attribute is mapped onto a table (e.g. when the element is multi-valued). In such cases, access to the element value and rank is often necessary.

Column references

Column reference generators are used to copy the contents of a previously filled column into a target column. The SQL types of the source and target columns must be compatible.

Column reference generators are useful to create foreign keys representing the parent-child aggregations in XML documents.

An important restriction in the use of column references is the requirement that:

- The source table must be in the current scope, i.e. the source table must be associated to an ancestor element of the element associated to the target table.

The source column must be filled before it can be copied into the target column.

Part

7

The mapping language

Overview

The previous sections have described the main concepts and rules that govern the schema-based mapping. This section introduces a specific XML vocabulary, which is used to specify and represent a mapping associated to a given XML Schema.

All the elements used in the XML representation of mappings are in a specific namespace, the URI of which is `http://www.xquark.org/Bridge/1.0/Mapping`.

The specification of the mapping language is also provided as an XML schema in Appendix A - XML Schema for the mapping language. This schema can also be used as a source of information for checking number of allowed occurrences, possible values and default values of both elements and attributes.

The `mapping` *element*

A mapping is represented in XML as a mapping document, using a `mapping` element as the document root element.

```
<mapping
  schemaLocation = anyURI
  version = string : 1.0>
Content: (element | map)*
</mapping>
```

The `mapping` element contains two attributes:

- *schemaLocation*: this required attribute declares the XML Schema of documents on which the mapping is defined. As in the XML Schema specifications, this attribute is made up of pairs of (namespace URI, schema URL) separated by blanks, allowing to specify multiple locations for a given target namespace, or locations for several target namespaces.
- *version:* this attribute specifies the version of the mapping language to be used. In the current version, the value of this attribute is fixed and equal to “1.0”.

As the top-level element in the mapping document, the `mapping` element will also contain in most cases namespace declarations. Besides the mapping namespace itself, the namespaces of all schemas to be mapped on the relational structures must be declared, as qualified elements from those namespaces will be used in the table and column mapping declarations.

The content of the mapping element consists of a set of `map` and `element` elements, which altogether represent the mapping specification.

The `map` *element*

The `map` element is used to define a table mapping by associating an XML type to a relational table. Map elements can appear at the top-level, in which case they must be explicitly associated to an XML complex type, or within the scope of an `element` or `attribute` element, in which case they are implicitly associated to the element or attribute type.

```
<map
  table = NCName
  type = QName
  name = NCName
  action = (insert | check | select | update) :
insert

  batchSize = short>
Content: (generator* ,element*, attribute*)
</map>
```

The `map` element can have four attributes:

- *table*: this required attribute contains the name of the table in which the content of the associated type or element will be stored.
- *type:* this attribute is required for top-level map declarations, and refers to the XML type that is mapped onto the table. In most cases, its value will be a qualified name, whose prefix will be the prefix associated to the namespace in which the type has been defined. This attribute should not be used when the mapping declaration appears in the scope of an `element` or `attribute` element.
- *name:* this attribute is also used in the case of a top-level declaration, and should be a unique name which can be used in further declarations to refer to this particular table mapping.
- *action:* this optional attribute specifies the action that is performed when an element or attribute using this table mapping is encountered in an XML document. The default action is to *insert* a new tuple in the table, based on element or attribute content. However, three other actions are available:

  - check*: specifies that the new tuple must be inserted only if it is not already present in the table. The existence of the tuple is checked based on a user-defined subset of the mapped element content.*
  - select*: specifies that an existing tuple must be retrieved from the table, based on a user-defined subset of the mapped element content. Once the tuple is loaded, other tuples may refer to its content.*
  - update*: specifies that an existing tuple must be updated with the mapped element or attribute content. The existing tuple is retrieved based on a user-defined subset of the mapped element content.*

- *batchSize:* this attribute allows to control the JDBC batch size used for this table (default is 20).

> This last parameter is an important feature for performance optimization (with auto-commit control that is performed through the XML/DBC API).

The content of the `map` element features:

- generators and column mappings, which specify how each column of the table must be filled,
- nested table mappings associated to child elements and more generally descendants of the mapped element.

Within a `map` element, all generators will be declared first, followed by mappings for elements and finally attributes.

The `element` *element*

The `element` element is used to specify a table or column mapping for each mapped element from the source XML document. This element can appear at the top level, within another `element` element or within a table mapping (`map` element).

```
<element
  name = QName
  column = NCName
  ref = NCName
  map = NCName
  inSelect (true | false) : true>
Content: (map* ,element*, attribute*)
</element>
```

The above syntax corresponds in reality to three distinct forms:

- *A scope declaration*: in this first form, only the *name* and *map* attributes are allowed. The element defines a scope for further mapping elements,

including in this order `map` elements, other `element` elements and `attribute` elements.

- *A column mapping*: in this second form, only the *name*, *column* and *inSelect* attributes can be present and the element content is empty. This form can only occur if the parent element is a `map` element.
- *A dual table-column mapping*: in this third form, all attributes can be present. In addition, the element can contain a `map` element, which specifies the table involved in the dual table-column mapping.

The `element` element has the following attributes:

- *name*: this required attribute represents the qualified name of the element. If the `element` element occurs at the top level, it must refer to an existing global element declaration in a declared namespace. If it occurs within another `element` element, it must refer to an existing local element declaration within that parent element.
- *column:* this attribute is required in the case of a column mapping, absent otherwise. It contains the name of the column into which the element value must be stored. The column name must be prefixed by a table name if the target column is defined in a containing table different from the immediate parent table (in the case of nested table mappings). If the table prefix is not present, it defaults to the table name in the innermost table mapping.
- *ref*: this optional attribute is used only in the case of a dual table and column mapping (when a 1-1 parent-child aggregation must be stored in the parent table). It contains the column name (prefixed with the child table name) which contains the value to be used as foreign key in the parent table).
- *map:* this optional attribute can be used when the element is associated to a table mapping (including the case of a dual table and column mapping). It refers to the name of a previously declared table mapping (using the `map` element). The element type must be compatible with the type associated to the referenced mapping.
- *inSelect:* this optional attribute can be used in the case of a column mapping to specify that the column participates in the request which is performed before inserting a new tuple when the action associated to the enclosing table mapping is *check*, *select* or *update*.

When the element defines a scope for further mapping declarations, the following child elements may appear in its content, in this order:

- `map` elements are used to associate table mappings to the element. Note that several table mappings may be associated to a single element, which means that an element can be stored as several tuples in different tables.
- `element` elements are used to create a new scope for a child of the current element.

- `attribute` elements are used to create a new scope for an attribute of the current element.

The `attribute` *element*

The `attribute` element is used to specify a table or column mapping for each mapped attribute from the source XML document. This element can only appear within an `element` element or within a table mapping (`map` element).

```
<attribute
  name = QName
  column = NCName
  inSelect (true | false) : true>
Content: (map*)
</attribute >
```

The above syntax corresponds in reality to two distinct forms:

- *A scope declaration*: in this first form, only the name attribute is allowed. The element defines a scope for further table mapping declarations (`map` elements).
- *A column mapping*: in this second form, all attributes can be present and the element content is empty. This form can only occur if the parent element is a `map` element.

The `attribute` element has the following attributes:

- *name*: this required attribute represents the qualified name of the attribute. It must refer to an existing local element declaration within the parent element.
- *column:* this attribute is required in the case of a column mapping, absent otherwise. It contains the name of the column into which the attribute value must be stored. The column name must be prefixed by a table name if the target column is defined in a containing table different from the immediate parent table (in the case of nested table mappings). If the table prefix is not present, it defaults to the table name in the innermost table mapping.
- *inSelect:* this optional attribute can be used in the case of a column mapping to specify that the column participates in the request which is performed before inserting a new tuple when the action associated to the enclosing table mapping is *check*, *select* or *update*.

When the `attribute` element defines a scope for further mapping declarations, `map` elements can appear within its content to associate table mappings to the attribute.

The `generator` *element*

The `generator` element is used to associate a generator to a column of a table. This element can only appear within a table mapping (`map` element).

```
<generator
  column = NCName
  ref = NCName
  class = NCName
  variable = NCName
  inKey = (true | false) : false
  inSelect (true | false) : true/>
```

This element can be used to define the three types of generators introduced in a previous section. The type of generator is specified by using one specific attribute among three mutually exclusive attributes *ref*, *class* and *variable*.

The `generator` element has the following attributes:

- *column*: this required attribute represents the name of the column into which the generated value must be stored. The column name must be prefixed by a table name if the target column is defined in a containing table different from the immediate parent table (in the case of nested table mappings). If the table prefix is not present, it defaults to the table name in the innermost table mapping.
- *ref*: this optional attribute is used to define a column reference generator. It contains the name of the source column from which the value must be copied. The source column name must be prefixed by the name of its containing table, which must itself be in scope (i.e. be used in an enclosing table mapping).
- *class*: this optional attribute is used to define a user generator. It contains the name of a user-defined Java class that implements the `UserGenerator` interface in the package `org.xquark.xml.mapping`.
- *variable*: this optional attribute is used to define a system variable generator. It contains the name of one of the system variables defined in the previous section (e.g. `$NodeValue` or `$LocalName`).

inSelect: this optional attribute can be used to specify that the column participates in the request which is performed before inserting a new tuple when the action associated to the enclosing table mapping is *check*, *select* or *update*.

Index

Table of figures

Appendix A - XML Schema for the mapping language

```
<?xml version="1.0" encoding="iso-8859-1"?>

<xsd:schema targetNamespace="http://www.xquark.org/Bridge/1.0/Mapping"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.xquark.org/Bridge/1.0/Mapping"
  elementFormDefault="qualified">

  <xsd:annotation>
    <xsd:documentation>
       XML Schema for mapping language. See:
       "XQuark Bridge - Mapping Reference Guide"
    </xsd:documentation>
  </xsd:annotation>

  <xsd:element name="mapping">
    <xsd:annotation>
      <xsd:documentation>
         A mapping is represented in XML as a mapping document,
         using a "mapping" element as the document root element
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexType>
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="map" type="topLevelMap"/>
        <xsd:element name="element" type="scopeElement"/>
      </xsd:choice>
      <xsd:attribute name="schemaLocation" type="xsd:string" use="required"/>
      <xsd:attribute name="version" type="versionNumber" fixed="1.0"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="topLevelMap">
    <xsd:annotation>
      <xsd:documentation>
         the "map" element is used to define a table mapping by
         associating an XML type to a relational table.
         top-level "map" elements must be explicitly associated to
         an XML complex type.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:choice maxOccurs="unbounded">
      <xsd:element ref="generator"/>
      <xsd:element name="element" type="mappableElement"/>
      <xsd:element name="attribute" type="mappableAttribute"/>
    </xsd:choice>
    <xsd:attribute name="table" type="tableName" use="required"/>
    <xsd:attribute name="type" type="xsd:QName" use="required"/>
    <xsd:attribute name="name" type="xsd:NCName" use="required"/>
    <xsd:attribute name="action" type="actionType" default="insert"/>
    <xsd:attribute name="batchSize" type="xsd:short"/>
```

```
</xsd:complexType>

<xsd:complexType name="localMap">
  <xsd:annotation>
    <xsd:documentation>
       the "map" element is used to define a table mapping by
       associating an XML type to a relational table.
       local "map" elements appear within the scope of an
       "element" or "attribute" element, in which case they are implicitly
       associated to the element or attribute type.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:choice maxOccurs="unbounded">
    <xsd:element ref="generator"/>
    <xsd:element name="element" type="mappableElement"/>
    <xsd:element name="attribute" type="mappableAttribute"/>
  </xsd:choice>
  <xsd:attribute name="table" type="tableName" use="required"/>
  <xsd:attribute name="action" type="actionType" default="insert"/>
  <xsd:attribute name="batchSize" type="xsd:short"/>
</xsd:complexType>

<xsd:complexType name="valueMap">
  <xsd:annotation>
    <xsd:documentation>
       the "map" element is used to define a table mapping by
       associating an XML type to a relational table.
       value "map" elements appear within the scope of an
       "attribute" element, in which case they are implicitly
       associated to the element or attribute type.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element ref="generator" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="table" type="tableName" use="required"/>
  <xsd:attribute name="action" type="actionType" default="insert"/>
</xsd:complexType>

<xsd:complexType name="scopeElement">
  <xsd:annotation>
    <xsd:documentation>
       The "element" element appearing outside a table mapping
     (column mappings are not allowed)
    </xsd:documentation>
  </xsd:annotation>
  <xsd:choice maxOccurs="unbounded">
    <xsd:element name="map" type="localMap"/>
    <xsd:element name="element" type="scopeElement"/>
    <xsd:element name="attribute" type="scopeAttribute"/>
  </xsd:choice>
  <xsd:attribute name="name" type="xsd:QName" use="required"/>
  <xsd:attribute name="map" type="xsd:NCName"/>
</xsd:complexType>

<xsd:complexType name="mappableElement">
  <xsd:annotation>
    <xsd:documentation>
```

```
         The "element" element appearing inside a table mapping
      </xsd:documentation>
    </xsd:annotation>
      <xsd:sequence>
            <xsd:element ref="generator" minOccurs="0" maxOccurs="1"/>
            <xsd:choice minOccurs="0" maxOccurs="unbounded">
              <xsd:element name="map" type="localMap"/>
              <xsd:element name="element" type="mappableElement"/>
              <xsd:element name="attribute" type="mappableAttribute"/>
      </xsd:choice>
      </xsd:sequence>
    <xsd:attribute name="name" type="xsd:QName" use="required"/>
    <xsd:attribute name="column" type="columnName"/>
    <xsd:attribute name="ref" type="columnName"/>
    <xsd:attribute name="map" type="xsd:NCName"/>
    <xsd:attribute name="inSelect" type="xsd:boolean" default="true"/>
  </xsd:complexType>

  <xsd:complexType name="scopeAttribute">
    <xsd:annotation>
      <xsd:documentation>
         The "attribute" element appearing outside a table mapping
       (column mappings are not allowed)
      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="map" type="valueMap" minOccurs="0"
maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:QName" use="required" />
  </xsd:complexType>

  <xsd:complexType name="mappableAttribute">
    <xsd:annotation>
      <xsd:documentation>
         The "attribute" element appearing inside a table mapping
      </xsd:documentation>
    </xsd:annotation>
      <xsd:sequence>
      <xsd:element ref="generator" minOccurs="0" maxOccurs="1"/>
        <xsd:element name="map" type="valueMap" minOccurs="0"
maxOccurs="unbounded"/>
      </xsd:sequence>
    <xsd:attribute name="name" type="xsd:QName" use="required" />
    <xsd:attribute name="column" type="columnName"/>
    <xsd:attribute name="inSelect" type="xsd:boolean" default="true"/>
  </xsd:complexType>

  <xsd:element name="generator">
    <xsd:annotation>
      <xsd:documentation>
         the "generator" element is used to associate a generator to
         a column of a table. this element can only appear within a
         table mapping (map element).
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexType>
      <xsd:attribute name="column" type="columnName" use="required"/>
```

```
      <xsd:attribute name="ref" type="columnName"/>
      <xsd:attribute name="variable" type="systemVariable"/>
      <xsd:attribute name="method" type="xsd:NCName"/>     <!-- deprecated -->
      <xsd:attribute name="class" type="xsd:NCName"/>
      <xsd:attribute name="inSelect" type="xsd:boolean" default="false"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:simpleType name="versionNumber">
    <xsd:restriction  base="xsd:decimal">
      <xsd:fractionDigits value="1"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="actionType">
    <xsd:restriction  base="xsd:string">
      <xsd:enumeration value="insert"/>
      <xsd:enumeration value="check"/>
      <xsd:enumeration value="select"/>
      <xsd:enumeration value="update"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="tableName">
    <xsd:restriction  base="xsd:NCName">
      <xsd:pattern value="\i[\c-[\.]]*(\.\i[\c-[\.]]+)?"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="columnName">
    <xsd:restriction  base="xsd:NCName">
      <xsd:pattern value="\i[\c-[\.]]*(\.\i[\c-[\.]]*){0,2}"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="systemVariable">
    <xsd:restriction  base="xsd:string">
      <xsd:enumeration value="$QName"/>
      <xsd:enumeration value="$NamespaceURI"/>
      <xsd:enumeration value="$LocalName"/>
      <xsd:enumeration value="$NodeRank"/>
      <xsd:enumeration value="$NodeValue"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```