BEA Liquid Data for WebLogic™

XQuery Developer's Guide

Version 8.5
Document Date: June 2005
Revised: August 2005

Copyright

Copyright © 2005 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks

BEA, BEA JRockit, BEA Liquid Data for WebLogic, BEA WebLogic Server, Built on BEA, Jolt, JoltBeans, SteelThread, Top End, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA AquaLogic, BEA AquaLogic Data Services Platform, BEA AquaLogic Enterprise Security, BEA AquaLogic Service Bus, BEA AquaLogic Service Registry, BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Manager, BEA MessageQ, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Enterprise Security, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic Java Adapter for Mainframe, BEA WebLogic JDriver, BEA WebLogic JRockit, BEA WebLogic Log Central, BEA WebLogic Personalization Server, BEA WebLogic Platform, BEA WebLogic Portal, BEA WebLogic Server Process Edition, BEA WebLogic WorkGroup Edition, BEA WebLogic Workshop, and Liquid Computing are trademarks of BEA Systems, Inc. BEA Mission Critical Support is a service mark of BEA Systems, Inc. All other company and product names may be the subject of intellectual property rights reserved by third parties.

All other trademarks are the property of their respective companies.

August 5, 2005 12:32 pm

Contents

1. Introducing the BEA XQuery Engine

2. BEA XQuery Implementation

A. XML Schema for Annotations

CHAPTER 1

Introducing the BEA XQuery Engine

This chapter briefly introduces the XQuery language and describes the version of the XQuery specification implemented in BEA Liquid Data for WebLogic. Links to more information about XQuery are also provided.

The following topics are covered:

- XML and XQuery
- XQuery Use in Liquid Data
- Supported XQuery Specifications
- Learning More About the XQuery Language

XML and XQuery

XML is an increasingly popular markup language that can be used to label content in a variety of data sources including structured and semi-structured documents, relational databases, and object repositories. XQuery is a query language that uses the structure of XML to express queries against data, including data physically stored in XML or transformed into XML using additional software. XQuery is therefore a language for querying XML-based information.

The relationship between XQuery and XML-based information is similar to the relationship between SQL and relational databases. Developers who are familiar with SQL will find XQuery to be conceptually a natural next step.

The W3C Query Working Group used a formal approach by defining a data model as the basis for XQuery. XQuery uses a type system and supports query optimization. It is statically typed, which supports compile-time type checking.

However, unlike SQL, which always returns two-dimensional result sets (rows and columns), XQuery results can conform to a complex XML schema. An XML schema can represent a hierarchy of nested elements that represent very detailed and complicated business data and information.

XQuery Use in Liquid Data

Liquid Data models the contents of various types of data sources as XML schemas. Once you have configured Liquid Data access to the data sources you want to use, such as relational databases, Web Services, application views, data views, and so on, you can issue queries written in XQuery to Liquid Data. Liquid Data evaluates the query, fetches the data from the underlying data sources, and returns the query results.

For information on developing XQueries in Liquid Data see the *Data Services Developer's Guide*.

Supported XQuery Specifications

Table 1-1 lists the XQuery and XML specifications with which the BEA implementation complies.

Table 1-1 Supported XQuery and XML Standards

Topic	Specification
XQuery 1.0 and XPath 2.0 Data Model	The XQuery and XPath data model implementation is based on the following specification: `http://www.w3.org/TR/2004/WD-xpath-datamodel-20040723/`

Table 1-1 Supported XQuery and XML Standards

XQuery 1.0 Specification	The BEA XQuery engine implements XQuery 1.0 based on the following specification: `http://www.w3.org/TR/2004/WD-xquery-20040723/`
XQuery 1.0 and XPath 2.0 Functions and Operators	The BEA XQuery engine implements functions and operators based on the following specification: `http://www.w3.org/TR/2004/WD-xpath-functions-20040723/` For information about BEA extensions implemented in Liquid Data, see "BEA XQuery Language Implementation" on page 2-28.

Learning More About the XQuery Language

You can learn more about XQuery and related technologies at the following locations:

- **XQuery**
  - http://www.w3.org/XML/Query
- **XML Schema**
  - http://www.w3.org/XML/Schema

CHAPTER 2

BEA XQuery Implementation

The World Wide Web (W3C) specification for XQuery defines a set of language features and functions. The BEA XQuery engine fully supports language features with one exception (modules) and also supports a robust subset of functions and adds a number of implementation-specific functions and language keywords.

This chapter describes the function and language implementation and extensions in the BEA XQuery engine.

The chapter includes the following topics:

BEA XQuery Function Implementation

Liquid Data supports a number of functions that are enhancements to the XQuery specification, which you can recognize by their extended function prefix `fn-bea:`. For example, the full XQuery notation for an extended function is: fn-bea:*function_name*.

This section describes the BEA XQuery function extensions, and contains the following topics:

Function Overview

Table 2-1 provides an overview of the BEA XQuery function extensions.

Table 2-1 BEA XQuery Function Extensions

Category	Function	Description
Access Control Functions	fn-bea:is-access-allowed	Checks whether a user associated with the current request context can access the specified resource.
	fn-bea:is-user-in-group	Checks whether the current user is in the specified group.
	fn-bea:is-user-in-role	Checks whether the current user is in the specified role.
	fn-bea:userid	Returns the identifier of the user making the request for the protected resource.
	fn-bea:rename	Renames a sequence of elements.
Duration, Date, and Time Functions	fn-bea:date-from-dateTime	Returns the date part of a dateTime value.
	fn-bea:date-from-string-with-format	Returns a new `date` value from a string source value according to the specified pattern.
	fn-bea:date-to-string-with-format	Returns a date string with the specified pattern.
	fn-bea:dateTime-from-string-with-format	Returns a new `dateTime` value from a string source value according to the specified pattern.
	fn-bea:dateTime-to-string-with-format	Returns a date and time string with the specified pattern.
	fn-bea:time-from-dateTime	Returns the time part of a dateTime value.
	fn-bea:time-from-string-with-format	Returns a new time value from a string source value according to the specified pattern.
	fn-bea:time-to-string-with-format	Returns a time string with the specified pattern.

Table 2-1 BEA XQuery Function Extensions (Continued)

Execution Control Functions	fn-bea:async	Evaluates an XQuery expression asynchronously, depositing the result of the evaluation into a buffer.
	fn-bea:fence	Enables you to define optimization boundaries, dividing queries into islands within which optimizations should occur.
	fn-bea:if-then-else	Accepts the value of a Boolean parameter to select one of two other input parameters.
	fn-bea:timeout	Returns either the full result of the primary expression, or the full result of the alternate expression in cases when the primary XQuery expression times out.
Numeric Functions	fn-bea:decimal-round	Returns a decimal value rounded to the specified precision or whole number.
	fn-bea:decimal-truncate	Returns a decimal value truncated to the specified precision or whole number.
Other Functions	fn-bea:get-property	Enables you to write data services that can change behavior based on external influence.
	fn-bea:inlinedXML	Parses textual XML and returns an instance of the XQuery 1.0 Data Model.
	fn-bea:format-number	Converts a double to a string using the specified format pattern.
QName Functions	fn-bea:QName-from-string	Creates an `xs:Qname` and uses the value of specified argument as its local name without a namespace.
Sequence Functions	fn-bea:interleave	Interleaves items specified in the arguments.

Table 2-1 BEA XQuery Function Extensions (Continued)

String Functions	fn-bea:match	Returns a list of integers (either an empty list with 0 integers or a list with 2 integers) specifying which characters in the string input matches the input regular expression.
	fn-bea:sql-like	Searches a string using a pattern, specified using the syntax of the SQL LIKE clause. The function optionally enables you to escape wildcards in the pattern.
	fn-bea:trim	Removes the leading and trailing white space.
	fn-bea:trim-left	Removes the leading white space.
	fn-bea:trim-right	Removes the trailing white space.

Access Control Functions

Liquid Data uses the role-base security policies of the underlying WebLogic platform to control access to data resources. A security policy is a condition that must be met for a secured resource to be accessed. If the outcome of condition evaluation is false — given the policy, requested resource, and user context — access to the resource is blocked and associated data is not returned.

Once the security policies have been configured using the Liquid Data Administration Console, you can use the security function extensions described in this section to determine:

- Whether a user associated with the current request context can access a specified resource.
- Whether the current user is in a specified role.
- Whether the current user is in a specified group.

This section describes the following Liquid Data access control function extensions to the BEA implementation of XQuery:

- fn-bea:is-access-allowed
- fn-bea:is-user-in-group
- fn-bea:is-user-in-role
- fn-bea:userid

fn-bea:is-access-allowed

The `fn-bea:is-access-allowed` function checks whether a user associated with the current request context can access the specified resource, which is denoted by a resource name and a data service identifier.

The function has the following signature:

```
fn-bea:is-access-allowed($resource as xs:string, $data_service as
xs:string) as xs:boolean
```

where `$resource` is the name of the resource, and `$data_service` is the resource identifier.

This function makes a call to the WebLogic security framework to check access for the specified resource. An example is shown below.

```
if (fn-bea:is-access-allowed("ssn", "ld:DataServices/CustomerProfile.ds"))
        then fn:true()
```

fn-bea:is-user-in-group

The `fn-bea:is-user-in-group` function checks whether the current user is in the specified group. This function analyzes the WebLogic authenticated subject for appropriate group membership.

This function has the following signature:

```
fn-bea:is-user-in-group($group as xs:string) as xs:boolean
```

where `$group` is the group to test against the current user.

Note: This operation is not automatically authenticated.

fn-bea:is-user-in-role

The `fn-bea:is-user-in-role` function checks whether the current user is in the specified glabal role.This function obtains a list of roles from the WebLogic security framework.

The function has the following signature:

```
fn-bea:is-user-in-role($role as xs:string) as xs:boolean
```

where `$role` is the role to test against the current user.

Note: This operation is not automatically authenticated.

fn-bea:userid

The `fn-bea:userid()` function returns the identifier of the user making the request for the protected resource.

The function has the following signature:

```
fn-bea:userid() as xs:string
```

Duration, Date, and Time Functions

This section describes the following duration, date, and time function extensions to the BEA implementation of XQuery:

- fn-bea:date-from-dateTime
- fn-bea:date-from-string-with-format
- fn-bea:date-to-string-with-format
- fn-bea:dateTime-from-string-with-format
- fn-bea:dateTime-to-string-with-format
- fn-bea:time-from-dateTime
- fn-bea:time-from-string-with-format
- fn-bea:time-to-string-with-format

fn-bea:date-from-dateTime

The `fn-bea:date-from-dateTime` function converts a `dateTime` to a `date`, and returns the date part of the `dateTime` value.

The function has the following signature:

```
fn-bea:date-from-dateTime($dateTime as xs:dateTime?) as xs:date?
```

where `$dateTime` is the date and time.

Examples:

- `fn-bea:date-from-dateTime(fn:dateTime("2005-07-15T21:09:44"))` returns a date value corresponding to July 15th, 2005 in the current time zone.
- `fn-bea:date-from-dateTime(())` returns an empty sequence.

fn-bea:date-from-string-with-format

The `fn-bea:date-from-string-with-format` function returns a new `date` value from a string source value according to the specified pattern.

The function has the following signature:

```
fn-bea:date-from-string-with-format($format as xs:string?, $dateString
as xs:string?) as xs:date?
```

where `$format` is the pattern and `$dateString` is the date. For more information about specifying patterns, see "Date and Time Patterns" on page 2-12.

Examples:

- `fn-bea:date-from-string-with-format("yyyy-MM-dd G", "2005-06-22 AD")` returns the specified date in the current time zone.
- `fn-bea:date-from-string-with-format("yyyy-MM-dd", "2002-July-22")` generates an error because the date string does not match the specified format.
- `fn-bea:date-from-string-with-format("yyyy-MMM-dd", "2005-JUL-22")` returns the specified date in the current time zone.

fn-bea:date-to-string-with-format

The `fn-bea:date-to-string-with-format` function returns a date string with the specified pattern.

The function has the following signature:

```
fn-bea:date-to-string-with-format($format as xs:string?, $date as
xs:date?) as xs:string?
```

where `$format` is the pattern and `$date` is the date. For more information about specifying patterns, see "Date and Time Patterns" on page 2-12.

Examples:

- `fn-bea:date-to-string-with-format("yy-dd-mm", xf:date("2005-07-15"))` returns the string "05-15-07".
- `fn-bea:date-to-string-with-format("yyyy-mm-dd", xf:date("2005-07-15"))` returns the string "2005-07-15".

fn-bea:dateTime-from-string-with-format

The `fn-bea:dateTime-from-string-with-format` function returns a new `dateTime` value from a string source value according to the specified pattern.

The function has the following signature:

```
fn-bea:dateTime-from-string-with-format($format as xs:string?,
$dateTimeString as xs:string?) as xs:dateTime?
```

where `$format` is the pattern and `$dateTimeString` is the date and time. For more information about specifying patterns, see "Date and Time Patterns" on page 2-12.

Examples:

- `fn-bea:dateTime-from-string-with-format("yyyy-MM-dd G", "2005-06-22 AD")` returns the specified date, 12:00:00AM in the current time zone.
- `fn-bea:dateTime-from-string-with-format("yyyy-MM-dd 'at' hh:mm", "2005-06-22 at 11:04")` returns the specified date, 11:04:00AM in the current time zone.
- `fn-bea:dateTime-from-string-with-format("yyyy-MM-dd", "2005-July-22")` generates an error because the date string does not match the specified format.
- `fn-bea:dateTime-from-string-with-format("yyyy-MMM-dd", "2005-JUL-22")` returns 12:00:00AM in the current time zone.

fn-bea:dateTime-to-string-with-format

The `fn-bea:dateTime-to-string-with-format` function returns a date and time string with the specified pattern.

The function has the following signature:

```
fn-bea:dateTime-to-string-with-format($format as xs:string?, $dateTime
as xs:dateTime?) as xs:string?
```

where `$format` is the pattern and `$dateTime` is the date and time. For more information about specifying patterns, see "Date and Time Patterns" on page 2-12.

Examples:

- `fn-bea:dateTime-to-string-with-format("dd MMM yyyy hh:mm a G", xf:dateTime("2005-01-07T22:09:44"))` returns the string "07 JAN 2005 10:09 PM AD".
- `fn-bea:dateTime-to-string-with-format("MM-dd-yyyy", xf:dateTime("2005-01-07T22:09:44"))` returns the string "01-07-2005".

fn-bea:time-from-dateTime

The `fn-bea:time-from-dateTime` function returns the time from a `dateTime` value.

The function has the following signature:

```
fn-bea:time-from-dateTime($dateTime as xs:dateTime?) as xs:time?
```

where `$dateTime` is the date and time.

Examples:

- `fn-bea:time-from-dateTime(fn:dateTime("2005-07-15T21:09:44"))` returns a time value corresponding to 9:09:44PM in the current time zone.
- `fn-bea:time-from-dateTime(())` returns an empty sequence.

fn-bea:time-from-string-with-format

The `fn-bea:time-from-string-with-format` function returns a new time value from a string source value according to the specified pattern.

The function has the following signature:

```
fn-bea:time-from-string-with-format($format as xs:string?, $timeString
as xs:string?) as xs:time?
```

where `$format` is the pattern and `$timeString` is the time. For more information about specifying patterns, see "Date and Time Patterns" on page 2-12.

Examples:

- `fn-bea:time-from-string-with-format("HH.mm.ss", "21.45.22")` returns the time 9:45:22PM in the current time zone.
- `fn-bea:time-from-string-with-format("hh:mm:ss a", "8:07:22 PM")` returns the time 8:07:22PM in the current time zone.

fn-bea:time-to-string-with-format

The `fn-bea:time-to-string-with-format` function returns a time string with the specified pattern.

The function has the following signature:

```
fn-bea:time-to-string-with-format($format as xs:string?, $time as
xs:time?) as xs:string?
```

where `$format` is the pattern and `$time` is the time. For more information about specifying patterns, see "Date and Time Patterns" on page 2-12.

Examples:

- `fn-bea:time-to-string-with-format("hh:mm a", xf:time("22:09:44"))` returns the string "10:09 PM".
- `fn-bea:time-to-string-with-format("HH:mm a", xf:time("22:09:44"))` returns the string "22:09 PM".

Date and Time Patterns

You can construct date and time patterns using standard Java class symbols. Table 2-2 outlines the pattern symbols you can use.

Table 2-2 Date and Time Patterns

This Symbol	Represents This Data	Produces This Result
G	Era	AD
y	Year	1996
M	Month of year	July, 07
d	Day of the month	19
h	Hour of the day (1–12)	10
H	Hour of the day (0–23)	22
m	Minute of the hour	30
s	Second of the minute	55
S	Millisecond	978

Table 2-2 Date and Time Patterns (Continued)

E	Day of the week	Tuesday
D	Day of the year	27
w	Week in the year	27
W	Week in the month	2
a	am/pm marker	AM, PM
k	Hour of the day (1–24)	24
K	Hour of the day (0–11)	0
z	Time zone	Pacific Standard Time Pacific Daylight Time

Repeat each symbol to match the maximum number of characters required to represent the actual value. For example, to represent 4 July 2002, the pattern is *d MMMM yyyy*. To represent 12:43 PM, the pattern is *hh:mm a*.

Execution Control Functions

This section describes the following Liquid Data execution control function extensions to the BEA implementation of XQuery:

- fn-bea:async
- fn-bea:fence
- fn-bea:if-then-else
- fn-bea:timeout

fn-bea:async

The `fn-bea:async` function evaluates an XQuery expression asynchronously, using a buffer to control data flow between threads of execution.

The function has the following signature:

```
fn-bea:async($expression as item()*, $cap as xs:integer) as item()*
```

where `$expression` is the XQuery expression to evaluate asynchronously and `$cap` is the size of the buffer.

The `fn-bea:async` function enables asynchronous execution of Web services to reduce problems caused by the latency of these services. When used in this manner, a very small buffer size such as 1 or 2 is sufficient, as the time to produce the first token can be long while the production of subsequent tokens should be quicker.

Example:

In the following example, `CUSTOMER` is a database table while the `getCreditScore` functions are Web services offered by two credit rating agencies.

```
for $cust in db:CUSTOMER()
where $cust/ID eq $param
return
    let $score1 := fn-bea:async(exper:getCreditScore($cust/SSN), 2),
        $score2 := fn-bea:async(equi:getCreditScore($cust/SSN), 2)
    return
        if (fn:abs($score1 - $score2) < $threshold)
        then fn:avg(($score1, $score2))
        else fn:max(($score1, $score2))
```

fn-bea:fence

The `fn-bea:fence` function enables you to define optimization boundaries, dividing queries into islands within which optimizations should occur while preventing optimizations across boundaries. You might consider using the `fn-bea:fence` function when building a query incrementally.

The function has the following signature:

```
fn-bea:fence($expression as item()*) as item()*
```

where `$expression` is the input expression.

The `fn-bea:fence` function is a pass-through function that does not change the input stream, but indicates to the optimizer that global rewritings should not occur across itself. Specifically, the `fn-bea:fence` function stops the following rewritings: view unfolding, loop unrolling, constant folding, and Boolean optimizations.

fn-bea:if-then-else

The `fn-bea:if-then-else` function examines the value of the first parameter. If the condition is true, Liquid Data returns the value of the second parameter (then). If the condition is false, Liquid Data returns the value of the third parameter (else). If the returned condition is not a Boolean value, Liquid Data generates an error.

The function has the following signature:

```
fn-bea:if-then-else($condition as xs:boolean?, $ifValue as
xdt:anyAtomicType, $elseValue as xdt:anyAtomicType)as
xdt:anyAtomicType
```

where `$condition` is the condition to test, `$ifValue` is the value to return when the condition evaluates to true, and `$elseValue` is the value to return when the condition evaluates to false.

Examples:

- `fn-bea:if-then-else (xf:true(), 3, "10")` returns the value 3.
- `fn-bea:if-then-else (xf:false(), 3, "10")` returns the string value 10.
- `fn-bea:if-then-else ("true", 3, "10")` generates a compile-time error because the condition is a string value and not a Boolean value.

fn-bea:timeout

The `fn-bea:timeout` function returns either the full result of the primary expression, or the full result of the alternate expression in cases when the primary XQuery expression times out.

The function has the following signature:

```
fn-bea:timeout($expression as item()*, $millisec as xs:integer, $alt
as item()*) as item()*
```

where `$expression` is the primary XQuery expression to evaluate, `$millisec` is the time out value in milliseconds, and `$alt` is an alternative XQuery expression to evaluate after a time out has occurred.

You can use the `fn-bea:timeout` function in the following ways:

- Around a region of an XQuery result which is optional, such as when you want the rest of the answer in any case.
- To select an available data source from among a set of possibly (very) heterogeneous sources that can provide the information of interest.

Note that the `fn-bea:timeout` function immediately returns the alternative expression in cases when accessing the data source causes an error. Also, an instance of `fn-bea:timeout` that has failed over to the alternate expression once will not re-evaluate the original expression during the same query evaluation.

Example:

$param is a external parameter

```
for $cust in db:CUSTOMER()
where $cust/ID eq $param
return
   fn-bea:timeout(exper:getCreditScore($cust/SSN), 200,
       fn-bea:timeout(equi:getCreditScore($cust/SSN), 200,
           fn:error()
       )
   )
```

Numeric Functions

This section describes the following numeric function extensions to the BEA implementation of XQuery:

- fn-bea:format-number
- fn-bea:decimal-round
- fn-bea:decimal-truncate

fn-bea:format-number

The `fn-bea:format-number` function converts a double to a string using the specified format pattern.

The function has the following signature:

```
fn-bea:format-number($number as xs:double, $pattern as xs:string) as
xs:string
```

where `$number` represents the double number to be converted to a string, and `$pattern` represents the pattern string. The format of this pattern is specified by the JDK 1.4.2 `DecimalFormat` class. (For information on DecimalFormat and other JDK 1.4.2 Java classes see: http://java.sun.com/j2se/1.4.2.)

fn-bea:decimal-round

The `fn-bea:decimal-round` function returns a decimal value rounded to the specified precision (scale) or to the nearest whole number.

The function has the following signatures:

```
fn-bea:decimal-round($value as xs:decimal?, $scale as xs:integer?) as
xs:decimal?

fn-bea:decimal-round($value as xs:decimal?) as xs:decimal?
```

where `$value` is the decimal value to round and `$scale` is the precision with which to round the decimal input. A scale value of 1 rounds the input to tenths, a scale value of 2 rounds it to hundreths, and so on.

Examples:

- `fn-bea:decimal-round(127.444, 2)` returns 127.44.
- `fn-bea:decimal-round(0.1234567, 6)` returns 0.123457.

fn-bea:decimal-truncate

The `fn-bea:decimal-truncate` function returns a decimal value truncated to the specified precision (scale) or to the nearest whole number.

The function has the following signatures:

```
fn-bea:decimal-truncate($value as xs:decimal?, $scale as xs:integer?)
as xs:decimal?

fn-bea:decimal-truncate($value as xs:decimal?) as xs:decimal?
```

where `$value` is the decimal value to truncate and `$scale` is the precision with which to truncate the decimal input. A scale value of 1 truncates the input to tenths, a scale value of 2 truncates it to hundreths, and so on.

Examples:

- fn-bea:decimal-truncate(192.454, 2) returns 192.45.
- fn-bea:decimal-truncate(192.454) returns 192.
- fn-bea:decimal-truncate(0.1234567, 6) returns 0.123456.

Other Functions

This section describes the following function extensions to the BEA implementation of XQuery:

fn-bea:get-property

The `fn-bea:get-property` function enables you to write data services that can change behavior based on external influence. This is an implicit way to parameterize functions.

The function first checks whether the property has been defined using the Liquid Data Administration Console. If so, it returns this value as a string. In cases when the property is not defined, the function returns the default value.

The function has the following signature:

```
fn-bea:get-property($propertyName as xs:string, $defaultValue as
xs:string) as xs:string
```

where `$propertyName` is the name of the property, and `$defaultValue` is the default value returned by the function.

fn-bea:inlinedXML

The `fn-bea:inlinedXML` function parses textual XML and returns an instance of the XQuery 1.0 Data Model.

The function has the following signature:

```
fn-bea:inlinedXML($text as xs:string) as node()*
```

where `$text` is the textual XML to parse.

Examples:

- `fn-bea:inlinedXML("<e>text</e>")` returns element “e”.
- `fn-bea:inlinedXML("<?xml version="1.0"><e>text</e>")` returns a document with root element “e”.

fn-bea:rename

The `fn-bea:rename` function renames an element or a sequence of elements.

The function has the following signature:

```
fn-bea:rename($oldelements as element()*, $newname as element()) as
element()*)
```

where `$oldelements` is the sequence of elements to rename, and `$newname` is an element from which the new name and type are extracted.

For each element in the original sequence, the fn-bea:rename function returns a new element with the following:

- The same name and type as `$newname`
- The same content as the old element

Example:

```
for $c in CUSTOMER()
return
<CUSTOMER>
   { fn-bea:rename($c/FIRST_NAME, <FNAME/>) }
   { fn-bea:rename($c/LAST_NAME, <LNAME/>) }
</CUSTOMER>
```

In the above, if `CUSTOMER()` returns:

```
<CUST><FIRST_NAME>John</FIRST_NAME><LAST_NAME>Jones</LAST_NAME></CUST>
```

The output value would be:

```
<CUSTOMER><FNAME>John</FNAME><LNAME>Jones</LNAME></CUSTOMER>
```

QName Functions

This section describes the following QName function extensions to the BEA implementation of XQuery:

fn-bea:QName-from-string

The `fn-bea:QName-from-string` function creates an `xs:QName` and uses the value of `$param` as its local name without a namespace.

The function has the following signature:

```
fn-bea:QName-from-string($name as xs:string) as xs:QName
```

where `$name` is the local name.

Sequence Functions

This section describes the following sequence function extensions to the BEA implementation of XQuery:

- fn-bea:interleave

fn-bea:interleave

The `fn-bea:interleave` function interleaves the specified arguments.

The function has the following signature:

```
fn-bea:interleave($item1 as item()*, $item2 as xdt:anyAtomicType) as
item()*
```

where `$item1` and `$item2` are the items to interleave.

For example, `fn-bea:interleave((<a/>, <b/>, </c>), " ")` returns the following sequence:

```
(<a/>, " ", <b/>, " ", </c>)
```

String Functions

This section describes the following string function extensions to the BEA implementation of XQuery:

- fn-bea:match
- fn-bea:sql-like
- fn-bea:trim
- fn-bea:trim-left
- fn-bea:trim-right

fn-bea:match

The `fn-bea:match` function returns a list of integers (either an empty list with 0 integers or a list with 2 integers) specifying which characters in the string input matches the input regular expression.

When the function returns a match, the first integer represents the index of (the position of) the first character of the matching substring and the second integer represents the number of matching characters starting at the first match.

The function has the following signature:

```
fn-bea:match($source as xs:string?, $regularExp as xs:string?) as
xs:int*
```

where `$source` is the input string and `$regularExp` uses the standard regular expression language.

Table 2-3 presents regular expression syntax examples.

Table 2-3 Regular Expression Syntax Examples

Category	Syntax Example	Description
Characters	unicode	Matches the specified unicode character.
	\	Used to escape metacharacters such as *, +, and ?.
	\\	Matches a single backslash (\ ) character.
	\0nnn	Matches the specified octal character.
	\0xhh	Matches the specified 8-bit hexidecimal character.
	\\uxhhh	Matches the specified 16-bit hexidecimal character.
	\t	Matches an ASCII tab character.
Characters	\n	Matches an ASCII new line character.
	\r	Matches an ASCII return character.
	\f	Matches an ASCII form feed character.
Simple Character Classes	[bc]	Matches the characters b or c.
	[a-f]	Matches any character between a and f.
	[^bc]	Matches any character except b and c.
Predefined Character Classes	.	Matches any character except the new line character.
	\w	Matches a word character: an alphanumeric character or the underscore (_) character.
	\W	Matches a non-word character.
	\s	Matches a white space character.
	\S	Matches a non-white space character.
	\d	Matches a digit.
	\D	Matches a non-digit.

Table 2-3 Regular Expression Syntax Examples (Continued)

Greedy Closures (Match as many characters as possible)	A*	Matches expression A zero or more times.
	A+	Matches expression A one or more times.
	A?	Matches expression A zero or one times.
	A(n)	Matches expression A exactly n times.
	A(n,)	Matches expression A at least n times.
	A(n, m)	Matches expression A between n and m times.
Reluctant Closures (Match as few characters as possible, and stops when a match is found)	A*?	Matches expression A zero or more times.
	A+?	Matches expression A one or more times.
	A??	Matches expression A zero or one times.
Logical Operators	AB	Matches expression A followed by expression B.
	A|B	Matches expression A or expression B.
	(A)	Used for grouping expressions.

Examples:

- `fn-bea:match("abcde", "bcd")` evaluates to the sequence (2,3).
- `fn-bea:match("abcde", ())` evaluates to the empty sequence ().
- `fn-bea:match((), "bcd")` evaluates to the empty sequence ().
- `fn-bea:match("abc", 4)` generates an error at compile time because the second parameter is not a string.
- `fn-bea:match("abcccdee", "[bc]")` evaluates to the sequence (2,1).

fn-bea:sql-like

The `fn-bea:sql-like` function tests whether a string contains the specified pattern. Typically, you can use this function as a condition for a query, similar to the SQL LIKE operator used in a predicate of SQL queries. The function returns TRUE if the pattern is matched in the source expression, otherwise the function returns FALSE.

The function has the following signatures:

```
fn-bea:sql-like($source as xs:string?, $pattern as xs:string?, $escape
as xs:string?) as xs:boolean?

fn-bea:sql-like($source as xs:string?, $pattern as xs:string?) as
xs:boolean?
```

where `$source` is the string to search, `$pattern` is the pattern specified using the syntax of the `SQL LIKE` clause, and `$escape` is the character to use to escape a wildcard character in the pattern.

You can use the following wildcard characters to specify the pattern:

- **Percent character (%).** Represents a string of zero or more characters.
- **Underscore character (_).** Represents any single character.

You can include the % or _ characters in the pattern by specifying an escape character and preceding the % or _ characters in the pattern with this escape character. The function then reads the character literally, instead of interpreting it as a special pattern-matching character.

Examples:

- `fn-bea:sql-like($RTL_CUSTOMER.ADDRESS_1/FIRST_NAME,"H%","\")` returns `TRUE` for all `FIRST_NAME` elements in `$RTL_CUSTOMER.ADDRESS` that start with the character `H`.
- `fn-bea:sql-like($RTL_CUSTOMER.ADDRESS_1/FIRST_NAME,"_a%","\")` returns `TRUE` for all `FIRST_NAME` elements in `$RTL_CUSTOMER.ADDRESS` that start with any character and have a second character of the letter `a`.
- `fn-bea:sql-like($RTL_CUSTOMER.ADDRESS_1/FIRST_NAME,"H\%%","\")` returns `TRUE` for all `FIRST_NAME` elements in `$RTL_CUSTOMER.ADDRESS` that start with the characters `H%`.

fn-bea:trim

The `fn-bea:trim` function removes the leading and trailing white space.

The function has the following signature:

```
fn-bea:trim($source as xs:string?) as xs:string?
```

where `$source` is the string to trim. In cases when `$source` is an empty sequence, the function returns an empty sequence. Liquid Data generates an error when the parameter is not a string.

Examples:

- `fn-bea:trim("abc")` returns the string value "abc".
- `fn-bea:trim("  abc  ")` returns the string value "abc".
- `fn-bea:trim(())` returns the empty sequence.
- `fn-bea:trim(5)` generates a compile-time error because the parameter is not a string.

fn-bea:trim-left

The `fn-bea:trim-left` function removes the leading white space.

The function has the following signature:

```
fn-bea:trim-left($input as xs:string?) as xs:string?
```

where `$input` is the string to trim.

Examples:

- `fn-bea:trim-left("    abc    ")` removes leading spaces and returns the string `"abc    "`.
- `fn-bea:trim-left(())` outputs an error. The input is the empty sequence (similar to a SQL null) which is a sequence containing zero items.

fn-bea:trim-right

The `fn-bea:trim-right` function removes the trailing white space.

The function has the following signature:

```
fn-bea:trim-right($input as xs:string?) as xs:string?
```

where `$input` is the string to trim.

Examples:

- `fn-bea:trim-right("    abc    ")` removes trailing spaces and returns the string `"    abc"`.
- `fn-bea:trim-right(())` outputs an error. The input is the empty sequence (similar to a SQL null) which is a sequence containing zero items.

Unsupported XQuery Functions

The following functions from the XQuery 1.0 specification are not supported in current BEA XQuery engine implementation:

- fn:base-uri
- fn:normalize-unicode
- fn:id
- fn:idref
- fn:collection

Unsupported XQuery Language Features

- Modules

BEA XQuery Language Implementation

This section describes the following BEA XQuery language implementation:

- Generalized FLWGOR (group by)
- Optional Indicator in Direct Element and Attribute Constructors

Generalized FLWGOR (group by)

BEA offers a group by clause extension to standard FLWOR expressions. The following EBNF shows the syntax of the general FLWGDOR:

```
flwgdorExpression := (forClause | letClause) (forClause
       | letClause
       | whereClause
       | groupbyClause
       | orderbyClause)* returnClause

groupbyClause := "group" [variable "as" variable] "by" (expression
    ["as" variable]) ("," (expression ["as" variable]))*
```

The remaining clauses referenced in the EBNF fragment follow the standard definition, as presented in the XQuery specification.

As an example, consider the case of grouping books by year, without loosing books that do not have a year attribute. Using standard XQuery, you would need to perform a self-join with the result of the `fn:distinct-values` function, concatenating the result of the self-join with the result for books without a year attribute.

The following illustrates the XQuery expression to accomplish this:

```
let $books := document("bib.xml")/bib/book return (
   for $year in fn:distinct-values($books/@year)
   return
       <g>
           <year>{ $year }</year>
           <titles>{ $books[@year eq $year]/title }</titles>
       </g>,
       <g>
           <year/>
           <titles>{ $books[fn:empty(@year)]/title }
       </g>
)
```

Using the BEA `group by` extension, you could write the same query as follows:

```
for $book in document("bib.xml")/bib/book
group $book as $partition by $book/@year as $year
return
   <g>
      <year>{ $year }</year>
      <titles>{ $partition/title }</titles>
   </g>
```

Table 2-4 Bindings Before Group By Clause is Applied

$book
<book year="1994" ISBN="147...">...</book>
<book year="1994" ISBN="198..."> ...</book>
<book year="2000" ISBN="123..."> ...</book>

Table 2-5 Bindings After Group By Clause is Applied

$year	$partition
1994	(<book year="1994" ISBN="147...">...</book>, <book year="1994" ISBN="198..."> ...</book>)
2000	<book year="2000" ISBN="123..."> ...</book>

The FLWGOR expression conceptually builds a sequence of binding tuples, where the size of the tuple is the number of variables in scope at that point in the FLWGOR. In the example, the tuple at the `group by` clause consists of a single variable binding `$book` which binds to each book in the `bib.xml` document, one book at a time (Table 1).

The `group by` creates a new sequence of binding tuples with each output tuple containing variables defined in the `group by` clause. After the `group by`, all variables there were previously in-scope go out of scope.

In the example, the output tuple from the `group by` clause is of size two with the variable bindings being for `$year` and `$partition` (Table 2).

The number of output tuples is equal to the number of unique group by value bindings. In the above example, this is the number of unique `book/@year` values: 2. The variable introduced in the `group` clause (`$partition` in the example above) binds to the sequence of all matching input values.

Optional Indicator in Direct Element and Attribute Constructors

This extension enables external consumers of XML generated by XQuery to have certain empty elements and attributes omitted. You can specify this using optional indicators, instead of employing computed constructors, conditional statements, and custom functions.

For example, consider the following query:

```
<a><b>{()}</b><c foo="{()}"/></a>,
```

The extension enables the following to be returned:

```
<a><c/></a>
```

instead of:

```
<a><b/><c foo=""/></a>
```

The extension uses the optional indicator '?' with direct element and attribute constructors. This mean that in the following you could change the production `DirElemConstructor` to the following:

```
[94]    DirElemConstructor    ::=    "<" QName "?"? DirAttributeList
("/>" | (">" DirElemContent* "</" QName S? ">")) /* ws: explicit */
```

Likewise, you could change the `DirAttributeList` to the following:

```
[95]    DirAttributeList    ::=    (S (QName "?"? S? "=" S?
DirAttributeValue)?)*
```

When ? is present, elements with no children and attributes with the value "" are omitted. The query in the example could then be written as:

```
<a><b?>{()}</b><c foo?="{()}"/></a>
```

which produces the following result:

```
<a><c/></a>
```

In another example, consider the case of constructing a new customer element with different tags. One requirement is that you do not want a phone element in the resulting customer when the phone number does not exist in the original customer. Using standard XQuery, you would have to write:

```
for $cust in CUSTOMER()
return
    <customer>
        <id>{ fn:data($cust/C_ID) }</id>
        {
            if (fn:exists($cust/PHONE))
            then <phone>{ fn:data($cust/PHONE) }</phone>
            else ()
        }
        ...
    </customer>
```

Using the optional element constructor, you could instead write the following:

```
for $cust in CUSTOMER()
return
   <customer>
       <id>{ fn:data($cust/C_ID) }</id>
       <phone?>{ fn:data($cust/PHONE) }</phone>
       ...
   </customer>
```

Similarly, when you want the resulting customer element to use attributes instead of elements, you would need to employ computed attribute constructors using standard XQuery, as illustrated by the following:

```
for $cust in CUSTOMER()
return
   <customer
       id="{ fn:data($cust/C_ID) }"
       {
           if (fn:exists($cust/PHONE))
           then attribute { "phone" } { fn:data($cust/PHONE) }
           else ()
       }
       ...
   />
```

Using the optional attribute constructor, the query becomes:

```
for $cust in CUSTOMER()
return
   <customer
      id="{ fn:data($cust/C_ID) }"
      phone?="{ fn:data($cust/PHONE) }"
      ...
   />
```

Implementation Specific Details

This version of the XQuery engine varies from the July, 2004 preliminary specification in the following regards:

- Modules are not supported.
- xs:integer is represented by 64-bit values.

CHAPTER 3

Understanding XML Namespaces

XML namespaces are a mechanism that ensures that there are no name conflicts (or ambiguity) when combining XML documents or referencing an XML element. Liquid Data fully supports XML namespaces and includes namespaces in the queries generated in WebLogic Workshop.

This section includes the following topics:

- Introducing XML Namespaces
- Using XML Namespaces in Liquid Data Queries and Schemas

Introducing XML Namespaces

Namespaces provide a mechanism to uniquely distinguish names used in XML documents. XML namespaces appear in queries as a namespace string followed by a colon. The W3C uses specific namespace prefixes to identity W3C XQuery data types and functions. In addition, BEA has defined the `fn-bea:` namespace to uniquely identify BEA-supplied functions and data types.

Table 3-1 lists the predefined XQuery namespaces used in Liquid Data queries.

Table 3-1 Predefined Namespaces in XQuery

Namespace Prefix	Description	Examples

`fn`	The prefix for XQuery functions.	`fn:data()`
		`fn:sum()`
		`fn:substring()`

`fn-bea:`	The prefix for Liquid Data-specific extensions to the standard set of XQuery functions.	`fn-bea:rename()` `fn-bea:is-access-allowed()`
`xs`	The prefix for XML schema types.	`xs:string`

For example, the `xs:integer` data type uses the XML namespace `xs`. Actually, `xs` is an alias (called a *prefix*) for the namespace URI.

XML namespaces ensure that names do not collide when combining data from heterogeneous XML documents. As an example, consider a document related to automobile manufacturers that contains the element `<tires>`. A similar document related to bicycle tire manufacturers could also contain a `<tires>` element. Combining these documents would be problematic under most circumstances. XML namespaces easily avoid these types of name collisions by referring to the elements as `<automobile:tires>` and `<bicycle:tires>`.

Exploring XML Schema Namespaces

XML schema namespaces—including the *target namespace*—are declared in the schema tag. The following is an example using a schema created during metadata import:

```
<xsd:schema
targetNamespace="http://temp.openuri.org/SampleApp/CustOrder.xsd"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:bea="http://www.bea.com/public/schemas"
elementFormDefault="unqualified" attributeFormDefault="unqualified">
   ...
```

The second line declares the target namespace using the `targetNamespace` attribute. It this case, the target namespace is bound to the namespace declared on the fourth line, meaning that all element and attribute names declared in this document belong to:

```
http://www.bea.com/public/schemas
```

The third line of the schema contains the *default namespace*, which is the namespace of all the elements that do not have an explicit prefix in the schema.

For example, if you see the following element in a schema document:

```
<element name="appliance" type="string"/>
```

the element `element` belongs to the default namespace, as do unprefixed types such as `string`.

The fifth line of the schema contains a namespace declaration (`bea`) which is simply an association of a URI with a prefix. There can be any number of these declarations in a schema.

References to types declared in this schema document must be prefixed, as illustrated by the following example:

```
<complexType name="AddressType">
         <sequence>
         <element name="street_address" type="string"/>

         ...

         </sequence>
</complexType>

<element name="address" type="bea:AddressType"/>
```

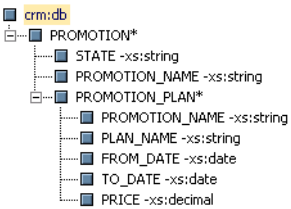
It is recommended that you create schemas with `elementFormDefault="unqualified"` and `attributeFormDefault="unqualified"`. This enables you to rename a namespace by renaming a single complex element, instead of having to explicitly map every element.

Using XML Namespaces in Liquid Data Queries and Schemas

Liquid Data automatically generates the namespace declarations when generating a query. Liquid Data employs a simple scheme using labels ns0, ns1, ns2, and so forth. Although it is easy to change assigned namespace names, care must be taken to make sure that all uses of that particular namespace are changed.

When a return type is created, by default it is `qualified`, meaning that the namespace of complex elements appear in the schema.

Figure 3-2 Example of a schema with unqualified attributes and elements



If you want simple elements or attributes to appear as qualified, you need to use an editor outside WebLogic Workshop to modify the generated schema for either or both `attributeFormDefault` and `elementFormDefault` to be set to *qualified*.

CHAPTER 4

Best Practices Using XQuery

This chapter offers a series of best practices for creating data services using XQuery. The chapter introduces a data service design model, and describes a conceptual model for layering data services to maximize management, maintainability, and reusability.

This chapter includes the following topics:

Introducing Data Service Design

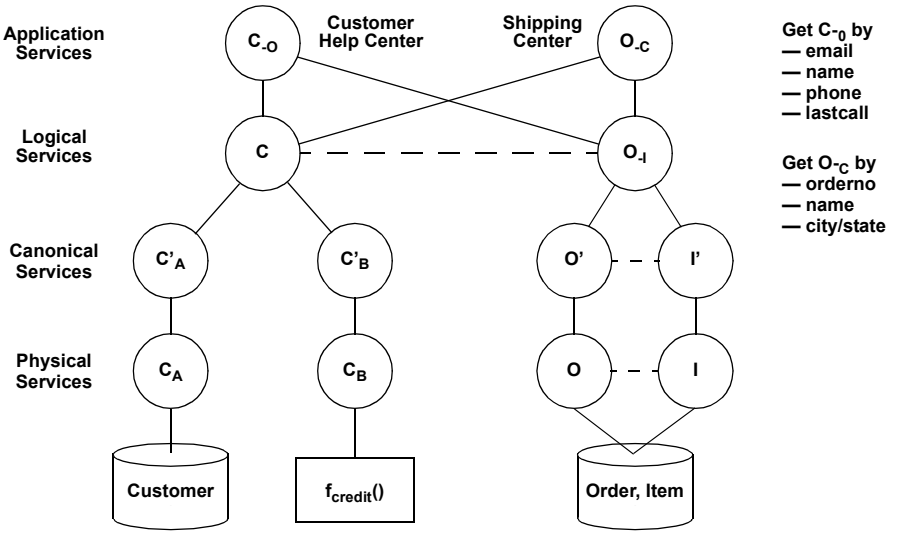
When designing data services, you should strive to maximize the ability to maintain, manage, and reuse queries. One approach is to adopt a layered design model that partitions services into the following levels:

- **Application Services.** Data services at the Application Services level are defined by client application requirements. Functions defined in this layer can additionally be used to constraint queries and to aggregate data, among other tasks.
- **Logical Services.** The Logical Services contain functions that perform general purpose logical operations and transformations on data accessed through Canonical and Physical Services.
- **Canonical Services.** Data services defined at the Canonical Services level normalize data obtained from the Physical Services level.

- **Physical Services.** The Physical Services are defined by the system based on introspection of physical data sources. The system creates data service functions that retrieve all rows in a table, offering the greatest flexibility for data service functions defined in higher layers. The system also defines relationships between data services, as required.

Figure 4-1 illustrates the data service design model.

Figure 4-1 Data Service Design Model



Using this design model, you can design and develop data services in the following manner:

1. Develop the Physical Services based on introspection of physical data sources.
2. Define the Application Services based on precise client application requirements.
3. Design the Canonical Services to normalize and create relationships between data accessed using the Physical Services.
4. Design the Logical Services to manipulate and transform data accessed through the Canonical and Physical Services, providing general purpose reusable services to the Application Services layer.
5. Work through the layers from the top down, determining optimal functions for each level and factoring our reusable queries.

Understanding Data Service Design Principles

This section describes best practices for designing and developing services at each layer of the data service design model. Table 4-2 describes the data service design principles.

Table 4-2 Data Service Design Principles

Level	Design Principle	Description
Application Services	Base design on client needs	Design data services and queries at the Application Services level specifically tuned to client needs, using functions defined at the Logical and Canonical Service levels.
	Nest or relate information, as required by the application	Use the XML practice of nesting related information in a single XML structure. Alternatively, use navigation functions to relate associated information, as required by the application.
	Introduce constraints at the highest level	Liquid Data propagates constraints down function levels when generating queries. By keeping constraints, such as function parameters, at the highest level, you encourage reuse of lower level functions and permit the system to efficiently optimize the final generated query.
	Aggregate data at the highest level	Aggregate data in functions at the highest level possible, preferably at the Application Services level.
Logical Services	Create common functions to serve multiple applications	Design functions that provide common services required by applications. Base function design at the Logical Services level on requirements already established at the Application Services level, based on client needs.
	Refactor to reduce the number of functions	Refactor the functions, as necessary, to reduce the overall number of functions to as few as possible. This reduces complexity, simplifies documentation, and eases future maintenance.
Canonical Services	Use function defined in the Physical Services level	Create (public) read functions can then all be expressed in terms of the main “get all instances” function.

Table 4-2 Data Service Design Principles (Continued)

Canonical Services	Create navigation functions to represent relationships	Use separate data services with relationships (implemented through navigation functions) rather than nesting data. For example, create navigation functions to relate customers and orders or customers and addresses instead of nesting this information. This keeps data services and their queries small, making them more manageable, maintainable, and reusable.
	Define keys to improve performance	Defining keys enables the system to use this information when optimizing queries.
	Establish relationships between unique identifiers and primary keys	Establish relationships between unique identifiers or primary keys that refer to the same data (such as Customer ID or SSN) but vary across multiple data sources. You can use either of the following methods: • Create navigation functions to create relationships between the data. • Create a new table in the database to relate the unique identifiers and primary keys.
Physical Services	Employ functions that get all records	Using private functions that get all records at the Physical Services level provides the system with the most flexibility to optimize data access based on constraints specified in higher level functions.
	Do not perform data type transformations	The system is unable to generate optimizations based on constraints specified at higher levels when data type transformations are performed at the Physical Services level.
	Do not aggregate	Use aggregates at the highest level possible to enable the system to optimize data access.

Applying Data Service Implementation Guidelines

Table 4-3 describes implementation guidelines to apply when designing and developing data services.

Table 4-3 Data Service Implementation Guidelines

Level	Design Principle	Description

Table 4-3 Data Service Implementation Guidelines (Continued)

Application Services	Use the group clause to aggregate	When performing a simple aggregate operation (such as count, min, max, and so forth) over data stored in a relational source, use a group clause as illustrated by the following:
		`for $x in f1:CUSTOMER()`
		`group $x as $g by 1`
		`return count($g)`
		instead of:
		`count( f1:CUSTOMER() )`
		in order to enable pushdown of the aggregation operation to the underlying relational data source.
		Note that the two formulations are semantically equivalent except for the case where the sequence returned by f1:CUSTOMER() is the empty sequence. Of course performance will be better for the pushed down statement.

	Use element(foo) instead of schema-element(foo)	Define function arguments and return types in data services as element(foo) instead of schema-element(foo). Using schema-element instead of element causes the Liquid Data to perform validation, potentially blocking certain optimizations.
	Use `xs:string` to cast data	Use `xs:string` when casting data instead of `fn:string()`. The two approaches are not equivalent when handling empty input, and the use of `xs:string` enables optimizations to be pushed to the database.
	Be aware of Oracle treating empty strings as NULL, and how this affects XQuery semantics	The Oracle RDBMS treats empty strings as NULL, without providing a method of distinguishing between the two. This can affect the semantics of certain XQuery functions and operations. For example, the `fn:lower-case()` function is pushed down to the database as LOWER, though the two have different semantics when handling an empty string, as summarized by the following: • `fn:lower-case()` returns an empty string • LOWER in Oracle returns NULL When using Oracle, consider using the `fn-bea:fence()` function and performing additional computation if precise XQuery semantics are required.

Table 4-3 Data Service Implementation Guidelines (Continued)

Application Services	Return plural for functions that contain FLWOR expressions	When a function body contains a FLWOR expression, or references to functions that contains FLWOR, the function should return plural.
		For example, consider the following XQuery expression:
		`For $c in CUSTOMER()`
		`Return`
		`  <CUSTOMER>`
		`    <LAST_NAME>$c/LAST_NAME</LAST_NAME>`
		`    <FIRST_NAME>$c/FIRST_NAME`
		`        </FIRST_NAME>`
		`    <ADDRESS>{`
		`    For $a in ADDRESS()`
		`    Where $a/CUSTOMER_ID =`
		`        $c/CUSTOMER_ID`
		`    Return`
		`        $a`
		`    }</ADDRESS>`
		`  </CUSTOMER>`
		Defining a one-to-one relationship between a CUSTOMER and an ADDRESS, as in the following, can block optimizations.
		`<element name=CUSTOMER>`
		`   <element name=LAST_NAME/>`
		`   <element name=FIRST_NAME/>`
		`   <element name=ADDRESS/>`
		`</element>`
		This is because Liquid Data determines that there can be multiple addresses for one CUSTOMER. This leads the system to insert a `TypeMatch` operation to ensure that there is exactly one ADDRESS. The TypeMatch operation blocks optimizations, thus producing a less efficient query plan.
		The Query Plan Viewer shows `TypeMatch` operations in red and should be avoided. Instead, the schema definition for ADDRESS should indicate that there could be zero or more ADDRESSes.
		`<element name=CUSTOMER>`
		`   <element name=LAST_NAME/>`
		`   <element name=FIRST_NAME/>`
		`   <element name=ADDRESS minOccurs="0"`
		`      maxOccurs="unbounded"/>`
		`</element>`

Table 4-3 Data Service Implementation Guidelines (Continued)

Application Services	Avoid cross product situations	Avoid cross product (Cartesian Product) situations when including conditions. For example, the following XQuery sample results in poor performance due to a cross product situation:
		`define fn ($p string)`
		`for $c in CUSTOMER()`
		`for $o in ORDER()`
		`where $c/id eq $p`
		`and $o/id eq $p`
		Instead, use the following form to specify the same query:
		`define fn ($p string)`
		`for $c in CUSTOMER()`
		`for $o in ORDER()`
		`where $c/id eq $o/id`
		`and $c/id eq $p`

CHAPTER 5

Understanding Liquid Data Annotations

This chapter describes the syntax and semantics of Liquid Data annotations in data service and XQuery function library (XFL) documents. Data service and XQuery function library documents define collections of XQuery functions. Annotations are XML fragments comprising the character content of XQuery pragmas.

There are two types of annotations:

- **Global annotations.** These pertain to the entire data service or XFL document. Global annotations are also referred to as XDS or XFL annotations respectively.
- **Local annotations.** These pertain to a particular function. Local annotations are also referred to as function annotations.

This chapter includes the following topics:

See Appendix A, "XML Schema for Annotations," for a listing of the XML Schema for annotations.

XDS Annotations

There is a single XDS annotation per data service document, which appears before all function annotations. The identifier for the pragma carrying the XDS annotation is `xds`. The qualified name of the top level element of the XML fragment corresponding to an XDS annotation has the local name `xds` and the namespace URI `urn:annotations.ld.bea.com`.

Each data service is associated with a unique target type. The prime type of the return type of every read function must match its target type. The target type of a data service is an element type whose qualified name is specified by the `targetType` attribute of the `xds` element. It is defined in a schema file associated with that data service.

The contents of the top-level `xds` element is a sequence of the following blocks of properties:

- General Properties
- Data Access Properties
- Target Type Properties
- Key Properties
- Relationship Properties
- Update Properties
- Security Properties

The following excerpt provides an example of an XDS annotation. In this case, the target type `t:CUSTOMER` associates the data service with a `t:CUSTOMER` type in a schema file.

```
(::pragma xds <x:xds xmlns:x="urn:annotations.ld.bea.com"
targetType="t:CUSTOMER" xmlns:t="ld:oracleDS/CUSTOMER">

<author>Joe Public</author>
<relationalDB name="OracleDS"/>

<field type="xs:string" xpath="FIRST_NAME">
   <extension nativeFractionalDigits="0" nativeSize="64"
      nativeTypeCode="12" nativeType="VARCHAR2"
      nativeXpath="FIRST_NAME"/>
      <properties nullable="false"/>
</field>
```

```
<field type="xs:string" xpath="LAST_NAME">
   <extension nativeFractionalDigits="0" nativeSize="64"
      nativeTypeCode="12" nativeType="VARCHAR2"
      nativeXpath="LAST_NAME"/>
   <properties nullable="false"/>
</field>

<field type="xs:string" xpath="CUSTOMER_ID">
   <extension nativeFractionalDigits="0" nativeSize="64"
      nativeTypeCode="12" nativeType="VARCHAR2"
      nativeXpath="CUSTOMER_ID"/>
   <properties nullable="false"/>
</field>

<field type="xs:dateTime" xpath="CUSTOMER_SINCE">
   <extension nativeFractionalDigits="0" nativeSize="7"
      nativeTypeCode="93" nativeType="DATE"
      nativeXpath="CUSTOMER_SINCE"/>
   <properties nullable="false"/>
</field>

<field type="xs:string" xpath="EMAIL_ADDRESS">
   <extension nativeFractionalDigits="0" nativeSize="32"
      nativeTypeCode="12" nativeType="VARCHAR2"
      nativeXpath="EMAIL_ADDRESS"/>
   <properties nullable="false"/>
</field>

<key name="CUSTOMER_PK11015727676593">
   <field xpath="CUSTOMER_ID">
      <extension nativeXpath="CUSTOMER_ID"/>
   </field>
</key>

<relationshipTarget roleName="CUSTOMER_ORDER" roleNumber="2"
   XDS="ld:oracleDS/CUSTOMER_ORDER.xds" minOccurs="0"
   maxOccurs="unbounded" opposite="CUSTOMER"/>
</x:xds>::)
```

General Properties

There are two types of general XDS properties:

- Standard Document Properties
- User-Defined Properties

Standard Document Properties

You can specify a set of standard document properties consisting of optional XML elements containing information pertaining to the author, creation date, or version of the document. You can also use the optional element `documentation` to specify related documentation. The names and types of the elements in the standard document properties block, as well as examples of their use, are shown in Table 5-1.

Table 5-1 Standard Document Properties

Element Name	Element Type	Optional	Example Instance
author	xs:string	Yes	`<author>J. Public</author>`
creationDate	xs:date	Yes	`<creationDate>2004-05-31</creationDate>`
version	xs:decimal	Yes	`<version>2.2<version>`
documentation	xs:string	Yes	`<documentation> Models an online Customer </documentation>`

User-Defined Properties

In addition to the standard properties, you can specify custom properties pertaining to the entire data service document using a sequence of zero (0) or more `property` elements. Each `property` element must be named using its `name` attribute and may contain any string content. For example:

```
<property name="data-refresh-rate">week</property>
```

Data Access Properties

Each data service document defines one or more XQuery functions that act as either data providers or *data transducers*. A data provider, or data source, is a function that is declared as *external*; its invocation causes data from an external source to be brought into the system. A data transducer, or data view, is defined in XQuery and it typically performs transformations on data derived from data sources or other data views.

The block of data access properties allows each data service to define whether its read functions include data sources or not. When data sources are included, the data access annotation describes the type of the external source being accessed by the external functions (there may be a single external source per data service) and its connection properties. When data sources are not included, the data service is designated as a user-defined view, and no connection information is required.

A data service may also define another form of XQuery functions known as *private* functions. The following types of data source data services are supported:

- Relational
- Web service
- Java function
- Delimited content
- XML content

The following sections describe the data access annotation for the data service types, as well as for data services that are designated as user-defined views. You can specify only one of the annotations in each data service. If no annotation is provided, the data service is considered a user-defined view.

Relational Data Service Annotations

The data access annotation for a relational data service consists of the empty element `relationalDB` with a single required attribute, `name`, whose value should be set to the JNDI name by which the external relational source has been registered with the application server. For example:

```
<relationalDB name="OracleDS"/>
```

In addition, the `relationalDB` element can contain the following optional parts:

- An optional element, named `properties`, that exposes the values of specific settings of the Relational Database Management System (RDBMS) represented by the relational source.
- An optional attribute, named `sourceBindingProviderClassName`, that specifies the transformation used to determine the relational source that should be used at system runtime in the place of the statically defined source.

Native Relational Properties

The `properties` element is an empty element with required attributes, as outlined in Table 5-2.

Table 5-2 Required Attributes for the properties Element

Attribute	Description
catalogSeparator	Specifies the string used by the RDBMS as a separator between a catalog and a table name.
identifierQuote	Specifies the string used by the RDBMS to quote SQL identifiers.
nullSortOrder	A string specifying how null values are sorted by the RDBMS, from among the following values: high, low, or unknown.
supportsCatalogsInDataManipulation	A Boolean specifying whether the RDBMS supports catalog names in Data Manipulation Language (DML) SQL statements.
supportsLikeEscapeClause	A Boolean specifying whether the RDBMS supports LIKE escape clauses.
supportsSchemasInDataManipulation	A Boolean specifying whether the RDBMS supports schema names in DML SQL statements.

Source Binding Provider

The value of the optional `sourceBindingProviderClassName` attribute should be bound to the fully-qualified name of a user-defined Java class implementing the `com.bea.ld.bindings.SourceBindingProvider` interface, defined by the following:

```
package com.bea.ld.bindings;
public interface SourceBindingProvider
{
    public String getBinding(String genericLocator, boolean isUpdate);
}
```

The user-defined implementation should provide the transformation that, given the statically configured relational source name (parameter `genericLocator`) and a Boolean flag indicating whether the relational source is accessed in query or update mode (parameter `isUpdate`), determines the name of the relational source name used by the system at runtime.

Note that you can use this transformation mechanism to perform credential mapping. In this case, a single set of query or update operations to be performed in the name of two distinct users U1 and U2 against the same statically-configured relational source R0, is executed against two distinct relational sources R1 and R2 respectively (where all sources R0, R1, R2 represent the same RDBMS and the security policies applied to the connection credentials used for R1 and R2 correspond to the security policies applied to the application credentials of user U1 and U2 respectively).

Note: You should set the source binding provider name uniformly across all relational data services sharing the same relational source JNDI name. Although this restriction is not enforced, its violation could result in unpredictable behavior at runtime.

Web Service Data Service Annotations

The data access annotation for a data service based on a Web service consists of the empty element `webService` with two required attributes, described in Table 5-3.

Table 5-3 Required Attributes for the webService Element

Attribute	Description
wsdl	A valid `http:` or `ld:` URI pointing to the location of the WSDL file containing the definition of the external Web service source.
targetNamespace	A valid URI that is identical to the `targetNamespace` URI of the WSDL.

For example:

```
<webService targetNamespace="urn:GoogleSearch"
       wsdl="ld:google/GoogleSearch.wsdl"/>
```

Java Function Data Service Annotations

The data access annotation for a Java function data service consists of the empty element `javaFunction` with a single required attribute named `class`, whose value should be set to the fully qualified name of the Java class serving as the external source. For example:

```
<javaFunction class="com.example.Test"/>
```

Delimited Content Data Service Annotations

The data access annotation for a delimited content data service is the empty element `delimitedFile`, accepting the optional attributes described in Table 5-4.

Table 5-4 Optional Attributes for the delimitedFile Element

Attribute	Description
file	A valid URI pointing to the location of the delimited file.
schema	A valid URI pointing to the location of the XML schema file defining the type (structure) of the delimited contents. If absent, the schema is derived based on the contents.
inferredSchema	Specifies whether the schema was inferred or provided by the user. The default value is false.

Table 5-4 Optional Attributes for the delimitedFile Element (Continued)

Attribute	Description
delimiter	The string used as the delimiter. If absent, the `fixedLength` attribute should be present.
fixedLength	The fixed length of the tokens contained in fixed length content. If absent, the delimiter attribute should be present.
hasHeader	A Boolean flag indicating whether the first line of the content should be interpreted as a header. The default value is false.

For example:

```
<delimitedFile schema="ld:df/schemas/ALL_TYPES.xsd" hasHeader="true"
      delimiter="," file="ld:df/ALL_TYPES.csv"/>
```

XML Content Data Service Annotations

The data access annotation for an XML content data service is the empty element `xmlFile` accepting the attributes described in Table 5-5.

Table 5-5 Attributes for the xmlFile Element

Attribute	Description
file	(Optional) A valid URI pointing to the location of the XML file.
schema	A valid URI pointing to the location of the XML schema file defining the type (structure) of the XML contents.

For example:

```
<xmlFile schema="ld:xml/somewhere/CUSTOMER.xsd"
      file="ld:xml/CUSTOMER_NESTED.xml"/>
```

User Defined View XDS Annotations

The data access annotation for a user-defined view data service is also known as a *logical* data service. It consists of the single empty element `userDefinedView`. For example:

```
<userDefinedView/>
```

Target Type Properties

The optional block of target type properties enables you to annotate simple valued fields in the target type of the data service with native type information pertaining to the following:

- The type of the corresponding field in the underlying external source (applicable only to data source data services)
- Information about the field's properties with respect to its update behavior. Each annotated field is represented by the element `field` with two required attributes, described in Table 5-6.

Table 5-6 Required Attributes for the field Element

Attribute	Description
xpath	An XPath value pointing to the field
type	The qualified name of the field's simple XML schema or XQuery type.

The following excerpt provides an example of a `field` element definition:

```
<field type="xs:string" xpath="FIRST_NAME">
   <extension nativeSize="64" nativeTypeCode="12" nativeType="VARCHAR2"
      nativeXpath="FIRST_NAME"/>
   <properties nullable="false"/>
</field>
```

Native Type Properties

Each `field` element can contain an optional `extension` element that accepts the optional attributes described in Table 5-7.

Table 5-7 Optional Attributes for the extension Element

Attribute	Description
nativeXpath	A native XPath value pointing to the corresponding native field in the external source.
nativeType	The native name of the native type of the corresponding native field, as it is known to the external source.

Table 5-7 Optional Attributes for the extension Element (Continued)

Attribute	Description
nativeTypeCode	The native type code of the native type of the corresponding native field, as it is known to the external source. In the case of relational sources, this is the type code as reported by JDBC.
nativeSize	The native size of the native type of the corresponding native field, as it is known to the external source. In the case of relational sources, this is the size as reported by JDBC.
nativeFractionalDigits	The native scale of the native type of the corresponding native field, as it is known to the external source. In the case of relational sources, this is the scale as reported by JDBC.

Update-related Type Properties

Each `field` element can also contain an optional `properties` element that accepts the optional attributes described in Table 5-8.

Table 5-8 properties element Optional Attributes

Attribute	Description
immutable	A Boolean value specifying whether the field is immutable (read-only) or not. The default value is false.
nullable	A Boolean value specifying whether the field accepts null values or not. The default value is false.

Key Properties

The optional block of key properties enables you to specify a set of identity constraints (keys) on the data service target type. Each key is represented by the element `key` that accepts an optional attribute, named `name`, whose value should serve as an identifier for the key.

Each `key` element contains a sequence of one or more `field` elements that collectively specify the simple-valued target type fields that the key comprises. Keys may be simple (having one field) or compound (having multiple fields). Each `field` element is identified by the value of its required `xpath` attribute (behaving similarly to the `xpath` attribute described in "Target Type Properties" on page 5-10).

Furthermore, each `field` element may optionally contain an extension element carrying a `nativeXpath` attribute that behaves similarly to the `nativeXpath` attribute described in "Native Properties" on page 5-24.

The following excerpt provides an example of a `key` element definition:

```
<key name="CUSTOMER_PK11015727676593">
   <field xpath="CUSTOMER_ID">
      <extension nativeXpath="CUSTOMER_ID"/>
   </field>
</key>
```

Relationship Properties

The optional block of relationship properties enables you to specify a set of relationship targets. A relationship target of a data service is a data service with which first service maintains a unidirectional or bidirectional relationship. Unidirectional relationships are realized through one or more *navigate* functions in the first data service that returns one or more instances of objects of the second service target type. Bidirectional relationships require that reciprocal functions are present in the second data service as well.

A relationship target is represented by the element `relationshipTarget` that accepts the attributes described in Table 5-9.

Table 5-9 Attributes for the relationshipTarget Element

Attribute	Description
roleName	A string that uniquely identifies the relationship target inside the data service.
roleNumber	(Optional) Either 1 or 2 (default is 1). The roleNumber specifies the index of the relationship target within the relationship.
XDS	The Liquid Data URI of the data service serving as the relationship target.
minOccurs	(Optional) The minimum cardinality of relationship target instances participating in this relationship. Possible values are all non-negative integers and the empty string. The default value is the empty string.
maxOccurs	(Optional) The maximum cardinality of relationship target instances participating in this relationship. Possible values are all positive integers, the string `unbounded`, and the empty string. The default is the empty string.
opposite	(Optional) String attribute that indicates the reciprocal relationship target in the case of bidirectional relationships. The value of this attribute is the identifier used to identify this data service as a relationship target in the data service identified by the value of the XDS attribute.

Additionally, the `relationshipTarget` element can itself contain the element `relationship` which in turn contains the nested element `description` that contains a human readable description about the relationship.

The following excerpt provides an example of a `relationshipTarget` element definition:

```
<relationshipTarget roleName="CUSTOMER_ORDER" roleNumber="2"
     XDS="ld:oracleDS/CUSTOMER_ORDER.xds" minOccurs="0"
     maxOccurs="unbounded" opposite="CUSTOMER"/>
```

Update Properties

The optional block of update properties enables you to specify a set of properties that establish certain policies about updating a data service's underlying sources. In particular, you can specify the following policies:

- The data service function that should be analyzed in order to build the plan for update decomposition.
- The external Java function to use as an update exit.
- The fields to use for optimistic locking purposes.
- Whether the data service is updateable or not.

Function for Update Decomposition

You can expose data obtained through data service read functions as SDO objects that can later be updated. In order for the changes to be persisted in the original data sources, the data service should specify which read function are to be used to perform data lineage analysis. The result of this analysis is a plan that allows the update to be decomposed into subplans that can be applied on each of the underlying sources. This feature is primarily used by logical data services.

The function for update decomposition is represented by the element `functionForDecomposition` that accepts the required attributes described in Table 5-10.

Table 5-10 Required Attributes for the functionForDecomposition Element

Attribute	Description
name	The qualified name of the read function to be used for update decomposition.
parity	The number of parameters of the read function specified in the `name` attribute.

When the `functionForDecomposition` element is not present, the first read function in the data service document is designated as the function for the update decomposition.

The following excerpt provides an example of a `functionForDecomposition` element definition:

```
<functionForDecomposition xmlns:f="ld:view/myView"
     name="f:firstNameFilter" arity="0"/>
```

Java Update Exit

A data source data service that is not automatically updateable (all non-relational XDS), or a data view XDS may specify an external mechanism to use for update. Supported external mechanisms include Java classes that implement a particular interface specified in the SDO update specification.

The Java class to use as update exit is represented by the empty element `javaUpdateExit` that accepts the attributes described in Table 5-11.

Table 5-11 Attributes for the javaUpdateExit Element

Attribute	Description
className	The fully qualified name of the Java class.
classFile	(Optional) The LD URI to the Java file for the class.

The following excerpt provides an example of a `functionForDecomposition` element definition:

```
<javaUpdateExit className="com.example.Exit"/>
```

Optimistic Locking Fields

SDO update assumes optimistic locking transactional semantics. The data service being updated can specify the fields that should be checked for updates during the interim using the empty element `optimisticLockingFields` that accepts one of the following as its content:

- An empty element, named `updated`, to specify only updated fields.
- An empty element, named `projected`, to specify all projected fields.
- One or more elements, named `field`, that accept a required string-valued attribute named `name` to specify user-specified fields.

The following excerpt provides an example of a `functionForDecomposition` element definition:

```
<optimisticLockingFields>
      <updated/>
</optimisticLockingFields>
```

Read-Only Data Service

You can designate a data service as read-only, in which case no updates will be allowed against the results obtained from the read functions of the service. You can use the empty element `readOnly` to designate a data service as read-only. For example:

```
<readOnly/>
```

Security Properties

You can use a data service to define one or more user-defined, logical protected resources. The element `secureResources`, containing one or more string-valued elements named `secureResource`, can be used for this purpose.

For example:

```
<secureResources>
     <secureResource>MyResource</secureResource/>
     <secureResource>MyOtherResource</secureResource/>
</secureResources>
```

You can link a logical resource defined using this syntax to a user-provided security policy using the Liquid Data Administration Console. Query content can inquire about a user's ability to access a logical resource using the built-in function `isAccessAllowed()`.

XFL Annotations

There is a single XFL annotation per XFL document, which appears before any function annotation in the document. The identifier for the pragma carrying the XFL annotation is `xfl`. The qualified name of the top level element of the XML fragment corresponding to an XFL annotation has the local name `xfl` and the namespace URI `urn:annotations.ld.bea.com`.

The contents of the top-level `xfl` element is a sequence of the following blocks of properties.

- General Properties
- Data Access Properties

The following sections provide detailed descriptions of each block of properties, while the following excerpt provides an example of a XFL annotation, which may serve as a reference.

```
(::pragma xfl <x:xfl xmlns:x="urn:annotations.ld.bea.com">
<creationDate>2005-03-09T17:48:58</creationDate>
<webService targetNamespace="urn:GoogleSearch"
      wsdl="ld:google/GoogleSearch.wsdl"/>
</x:xfl>::)
```

General Properties

The general properties applicable to an XFL document are identical to the general properties for a data service document, as described in "General Properties" on page 5-4.

Data Access Properties

Each XFL document defines one or more XQuery functions that serve as library functions that can be used either inside data service documents to define read navigate or private functions, or inside other XFL documents to specify other library functions.

Since XFL documents do not have a target type, the return types of the library functions found inside these document may differ from each other. In particular, a function inside an XFL document may return a value having a simple type (or any other type). XFL functions can be external data source functions or user-defined.

The following types of XFL documents are supported:

- Relational (logical)
- Web service (logical)
- Java function (logical)
- User-defined view (logical)

You can specify only one of the annotations in each XFL. If no annotation is provided, the XFL is considered a user-defined view.

The data access properties for Relational, Web service, Java function, and user-defined view XFL documents are the same as the corresponding properties for data service documents, as described above.

Function Annotations

There is a single function annotation per data service or XFL function, which appears before the function declaration in the document. The identifier for the pragma carrying the function annotation is `function`. The qualified name of the top level element of the XML fragment corresponding to an XDS or XFL annotation has the local name `function` and the namespace URI `urn:annotations.ld.bea.com`.

Each data service function is classified using one of the following categories:

- Read function
- Navigate function
- Private function

The classification of an data service function is determined by the value of a required attribute `kind` in the `function` element, which accepts the values `read`, `navigate`, or `private` to denote the corresponding categories. Each XFL function is considered to be a library function.

The prime type of the return type of a read function must match the target type of the data service. In addition, the `function` element for a navigate function must carry a string-valued attribute `returns` whose value must match the role name of a relationship target defined in the data service. Moreover, the prime type of the return type of a navigate function must match the target type of the data service serving as the relationship target.

Finally, the namespace URIs of the qualified names of all the functions in a data service or XFL must specify the location of the data service or XFL document in the LD repository. For example:

```
ld:{directory path to data service folder}/{data service file name
without extension}
```

or

```
lib:{directory path to XFL folder}/{XFL file name without extension}
```

The `function` element accepts the additional optional attributes described in Table 5-12.

Table 5-12 Optional Attributes for the function Element

Attribute	Description
nativeName	Applicable to data source functions, `nativeName` is the name of the function as it is known to the external source. In the case of relational sources, for example, it corresponds to the table name.
nativeLevel1Container	Applicable to data source functions that represent external sources employing hierarchical containment schemes; `nativeLevel1Container` is the name of the top-level native container, as it is known to the external source. In the case of relational sources, for example, it corresponds to the catalog name, whereas, in the case of Web service sources, it corresponds to the service name.
nativeLevel2Container	Applicable to data source functions that represent external sources employing hierarchical containment schemes; `nativeLevel2Container` is the name of the second-level native container, as it is known to the external source. In the case of relational sources, for example, it corresponds to the schema name. In the case of Web service sources, it corresponds to the port name.
nativeLevel3Container	Applicable to data source functions that represent external sources employing hierarchical containment schemes; `nativeLevel3Container` is the name of the top-level native container, as it is known to the external source. In the case of relational sources, for example, it corresponds to the stored procedure package name.
style	Applicable to data source functions, `style` is a native qualifier by which the function is known to the external source (e.g. table, view, storedProcedure, or sqlQuery for relational sources; rpc or document for Web services).
roleName	Applicable to navigate functions, `roleName` should match the value of the `roleName` attribute of the relationshipTarget implemented by the function.

The content of the top-level `function` element is a sequence of the following blocks of properties:

- General Properties
- UI Properties
- Cache Properties
- Signature Properties
- Native Properties

The following excerpt provides an example of a function annotation:

```
(::pragma function <f:function xmlns:f="urn:annotations.ld.bea.com"
kind="datasource" nativeName="CUSTOMER" nativeLevel2Container="RTL"
style="table">
<cacheable>
     <useCache TTL="600"/>
</cacheable>
</f:function>::)
```

General Properties

All standard document properties and user-defined properties defined in "Standard Document Properties" on page 5-4 and "User-Defined Properties" on page 5-4 are applicable to function annotations.

UI Properties

A set of user interface properties may be introduced by the XQuery Editor to persist location information about the graphical components representing the expression in the function body. UI properties are represented by the element `uiProperties` which accepts a sequence of one or more elements, named `component`, as its content. Each `component` element accepts the attributes described in Table 5-13

Table 5-13 Attributes for the component Element

Attribute	Description
identifier	An identifier for the UI component.
minimized	A Boolean flag indicating whether the UI component has been minimized or not.

Table 5-13 Attributes for the component Element (Continued)

Attribute	Description
x	The x-coordinate for the UI component.
y	The y-coordinate for the UI component.
w	The width of the UI component.
h	The height of the UI component.
viewPosX	The x-coordinate of the scrollbar position of the component.
viewPosY	The y-coordinate of the scrollbar position of the component.

In addition, each `component` element may optionally contain one or more `treeInfo` elements containing information about the tree representation of the types pertaining to the component. In the absence of the above property, the query editor uses the default layout.

Cache Properties

You can use the optional block of cache properties to specify whether a function can be cached or not. You should specify a function whose results for the same set of arguments are intrinsically highly volatile as non-cached. On the other hand, you should specify a function whose results for the same set of arguments are either fixed or remain unchanged for a period of time as cacheable.

This property of a function is represented by the empty element `nonCacheable`. In the absence of the `nonCacheable` element, a function is considered to be potentially cacheable. The following excerpt provides an example:

```
<nonCacheable/>
```

Signature Properties

You can use the optional block of signature properties to annotate the parameters of a data service or XFL function with additional information to that provided by the function signature. These properties are applicable to data source (data service or XFL) functions.

The signature properties block is represented by the element `params` which accepts a sequence of one or more elements, named `param`, as its content. Each `param` element is an empty element that accepts the optional attributes described in Table 5-14.

Table 5-14 param element Optional Attributes

Attribute	Description
name	The name of the parameter, as it is known to the external source.
nativeType	The native type of the parameter, as it is known to the external source.
nativeTypeCode	The native type code of the parameter, as it is known to the external source.
xqueryType	The qualified name of the XML Schema or XQuery type used for the parameter.
kind	One of the following values: unknown, in, inout, out, return or result (applicable to stored procedures).

The following excerpt provides an example of a `params` element definition:

```
<params>
   <param nativeType="java.lang.String"/>
   <param nativeType="java.lang.int"/>
</params>
```

Native Properties

You can use native properties to further annotate a data source function based on the type of the external source that it represents. There are two types of native properties pertaining to relational and Web service sources respectively:

- SQL query properties
- SOAP handler properties

SQL Query Properties

The `function` annotation element of a function that represents a user-defined SQL query has its style attribute set to `sqlQuery` and accepts a nested element, named `sql`. The `sql` element accepts string content that corresponds to the statement of the (possibly parameterized) SQL query that the function represents.

If required, the statement can be escaped inside a CDATA section to account for reserved XML characters (e.g. <, >, &). The `sql` element also accepts the optional attribute `isSubquery` whose boolean value indicates whether the SQL statement may be used as a nested SQL sub-query. If the attribute is absent, its value defaults to `true`.

The following excerpt provides an example of a `sqlQuery` element definition:

```
<sql isSubquery="true">
   SELECT t.FIRST_NAME FROM RTLALL.dbo.CUSTOMER t</sql>
```

SOAP Handler Properties

The `function` annotation element of a function that represents a Web service call accepts a nested element, named `interceptorConfiguration`. The `interceptorConfiguration` element accepts two required attributes, as described in Table 5-15.

Table 5-15 Required Attributes for the interceptorConfiguration Element

Attribute	Description
fileName	The location of the file containing the configuration of the SOAP handler chains that are applicable to the Web service.
aliasName	The alias name by which the SOAP handler chain has been configured.

APPENDIX A

XML Schema for Annotations

This chapter lists the XML Schema for annotations. For more information about the syntax and semantics of Liquid Data annotations in data service and XQuery function library (XFL) documents, see Chapter 5, "Understanding Liquid Data Annotations."

Listing A-1 XML Schema for Annotations

```
<?xml version="1.0"?>
<xs:schema targetNamespace="urn:annotations.ld.bea.com"
xmlns:tns="urn:annotations.ld.bea.com"
xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="unqualified"
attributeFormDefault="unqualified">
  <!--====================-->
  <!--  XDS annotation  -->
  <!--====================-->
  <xs:element name="xds">
    <xs:complexType>
      <xs:sequence>
        <!-- document properties -->
        <xs:element name="author" type="xs:string" minOccurs="0"/>
        <xs:element name="comment" type="xs:string" minOccurs="0"/>
        <xs:element name="creationDate" type="xs:dateTime" minOccurs="0"/>
        <xs:element name="documentation" type="xs:string" minOccurs="0"/>
        <xs:element name="version" type="xs:decimal" minOccurs="0"/>
        <!-- user defined properties -->
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element name="property">
            <xs:complexType>
              <xs:simpleContent>
```

```
                <xs:extension base="xs:string">
                  <xs:attribute name="name" type="xs:string"/>
                </xs:extension>
              </xs:simpleContent>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <!-- security -->
        <xs:element name="security" minOccurs="0">
          <xs:complexType>
            <xs:attribute name="rolesAllowed" type="xs:string"/>
            <xs:attribute name="runAs" type="xs:string"/>
            <xs:attribute name="runAsPrincipal" type="xs:string"/>
          </xs:complexType>
        </xs:element>
        <xs:choice>
          <!-- choice 1: java functions -->
          <xs:element name="javaFunction">
            <xs:complexType>
              <xs:attribute name="class" type="xs:string" use="required"/>
            </xs:complexType>
          </xs:element>
          <!-- choice 2: web services -->
          <xs:element name="webService">
            <xs:complexType>
              <xs:attribute name="wsdl" type="xs:anyURI" use="required"/>
              <xs:attribute name="targetNamespace" type="xs:anyURI"
use="required"/>
            </xs:complexType>
          </xs:element>
          <!-- choice 3: relational sources -->
          <xs:element name="relationalDB">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="properties" minOccurs="0">
                  <xs:complexType>
                    <xs:attribute name="catalogSeparator" type="xs:string"
use="required"/>
                    <xs:attribute name="identifierQuote" type="xs:string"
use="required"/>
                <xs:attribute name="nullSortOrder" type="tns:nullSortOrderType"
use="required"/>
                    <xs:attribute name="supportsCatalogsInDataManipulation"
type="xs:boolean" use="required"/>
                <xs:attribute name="supportsLikeEscapeClause" type="xs:boolean"
use="required"/>
                    <xs:attribute name="supportsSchemasInDataManipulation"
type="xs:boolean" use="required"/>
                  </xs:complexType>
```

```
              </xs:element>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="dbType" type="xs:string"/>
            <xs:attribute name="dbVersion" type="xs:string"/>
            <xs:attribute name="driver" type="xs:string"/>
            <xs:attribute name="uri" type="xs:string"/>
            <xs:attribute name="username" type="xs:string"/>
            <xs:attribute name="password" type="xs:string"/>
            <xs:attribute name="SID" type="xs:string"/>
            <xs:attribute name="sourceBindingProviderClassName"
type="xs:string"/>
          </xs:complexType>
        </xs:element>
        <!-- choice 4: delimited files -->
        <xs:element name="delimitedFile">
          <xs:complexType>
            <xs:attribute name="file" type="xs:anyURI"/>
            <xs:attribute name="schema" type="xs:anyURI"/>
            <xs:attribute name="inferredSchema" type="xs:boolean"
default="false"/>
            <xs:attribute name="delimiter" type="xs:string"/>
            <xs:attribute name="fixedLength" type="xs:positiveInteger"/>
           <xs:attribute name="hasHeader" type="xs:boolean" default="false"/>
          </xs:complexType>
        </xs:element>
        <!-- choice 5: XML files -->
        <xs:element name="xmlFile">
          <xs:complexType>
            <xs:attribute name="file" type="xs:anyURI"/>
            <xs:attribute name="schema" type="xs:anyURI" use="required"/>
          </xs:complexType>
        </xs:element>
        <!-- choice 6: user defined view -->
        <xs:element name="userDefinedView" minOccurs="0"/>
        <!-- choice 7: nothing, defaults to userDefinedView -->
        <xs:sequence/>
      </xs:choice>
      <!-- field annotations -->
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element name="field">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="extension" minOccurs="0">
                <xs:complexType>
                  <xs:sequence minOccurs="0">
                    <xs:element name="autoNumber">
                      <xs:complexType>
                        <xs:attribute name="type" type="tns:autoNumberType"
```

```
use="required"/>
                      <xs:attribute name="sequenceObjectName" type="xs:string"/>
                         </xs:complexType>
                       </xs:element>
                     </xs:sequence>
                     <xs:attribute name="nativeXpath" type="xs:string"/>
                     <xs:attribute name="nativeType" type="xs:string"/>
                     <xs:attribute name="nativeTypeCode" type="xs:int"/>
                     <xs:attribute name="nativeSize" type="xs:int"/>
                     <xs:attribute name="nativeFractionalDigits"
type="tns:scaleType"/>
                     <!-- relational: autoNumber -->
                     <!-- relational: native column names and types -->
                   </xs:complexType>
                 </xs:element>
                 <xs:element name="properties">
                   <xs:complexType>
                     <xs:attribute name="immutable" type="xs:boolean"
default="false"/>
                     <xs:attribute name="nullable" type="xs:boolean"
default="false"/>
                     <xs:attribute name="transient" type="xs:boolean"
default="false"/>
                   </xs:complexType>
                 </xs:element>
               </xs:sequence>
               <xs:attribute name="xpath" type="xs:string" use="required"/>
               <xs:attribute name="type" type="xs:string" use="required"/>
             </xs:complexType>
           </xs:element>
         </xs:sequence>
         <!-- keys -->
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
           <xs:element name="key">
             <xs:complexType>
               <xs:sequence>
                 <xs:element name="field" maxOccurs="unbounded">
                   <xs:complexType>
                     <xs:sequence>
                       <xs:element name="extension" minOccurs="0">
                         <xs:complexType>
                           <xs:attribute name="nativeXpath" type="xs:string"
use="required"/>
                         </xs:complexType>
                       </xs:element>
                     </xs:sequence>
                    <xs:attribute name="xpath" type="xs:string" use="required"/>
                   </xs:complexType>
                 </xs:element>
```

```
            </xs:sequence>
            <xs:attribute name="name" type="xs:string"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <!-- relationships -->
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element name="relationshipTarget">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="relationship" minOccurs="0">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="description" type="xs:string"
minOccurs="0"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="roleName" type="xs:string" use="required"/>
            <xs:attribute name="roleNumber" type="tns:roleType" default="1"/>
            <xs:attribute name="XDS" type="xs:string" use="required"/>
            <xs:attribute name="minOccurs" type="tns:allNNI" default="1"/>
            <xs:attribute name="maxOccurs" type="tns:allNNI" default="1"/>
            <xs:attribute name="opposite" type="xs:string"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <!-- SDO elements -->
      <xs:element name="functionForDecomposition" minOccurs="0">
        <xs:complexType>
          <xs:attribute name="name" type="xs:QName" use="required"/>
          <xs:attribute name="arity" type="xs:int" use="required"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="javaUpdateExit" minOccurs="0">
        <xs:complexType>
          <xs:attribute name="className" type="xs:string" use="required"/>
          <xs:attribute name="classFile" type="xs:string"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="optimisticLockingFields" minOccurs="0">
        <xs:complexType>
          <xs:choice>
            <xs:element name="updated">
              <xs:complexType/>
            </xs:element>
            <xs:element name="projected">
              <xs:complexType/>
```

```
              </xs:element>
              <xs:element name="field" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:attribute name="name" type="xs:string" use="required"/>
                </xs:complexType>
              </xs:element>
            </xs:choice>
          </xs:complexType>
        </xs:element>
        <!-- security -->
        <xs:element name="secureResources" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="secureResource" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="readOnly" minOccurs="0">
          <xs:complexType/>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="targetType" type="xs:QName" use="required"/>
    </xs:complexType>
  </xs:element>
  <!--====================-->
  <!--  XFL annotation  -->
  <!--====================-->
  <xs:element name="xfl">
    <xs:complexType>
      <xs:sequence>
        <!-- document properties -->
        <xs:element name="author" type="xs:string" minOccurs="0"/>
        <xs:element name="comment" type="xs:string" minOccurs="0"/>
        <xs:element name="creationDate" type="xs:dateTime" minOccurs="0"/>
        <xs:element name="documentation" type="xs:string" minOccurs="0"/>
        <xs:element name="version" type="xs:decimal" minOccurs="0"/>
        <!-- user defined properties -->
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element name="property">
            <xs:complexType>
              <xs:simpleContent>
                <xs:extension base="xs:string">
                  <xs:attribute name="name" type="xs:string"/>
                </xs:extension>
              </xs:simpleContent>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
```

```
        <xs:choice>
          <!-- choice 1: java functions -->
          <xs:element name="javaFunction">
            <xs:complexType>
              <xs:attribute name="class" type="xs:string" use="required"/>
            </xs:complexType>
          </xs:element>
          <!-- choice 2: web services -->
          <xs:element name="webService">
            <xs:complexType>
              <xs:attribute name="wsdl" type="xs:anyURI" use="required"/>
              <xs:attribute name="targetNamespace" type="xs:anyURI"
use="required"/>
            </xs:complexType>
          </xs:element>
          <!-- choice 3: relational sources -->
          <xs:element name="relationalDB">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="properties" minOccurs="0">
                  <xs:complexType>
                    <xs:attribute name="catalogSeparator" type="xs:string"
use="required"/>
                    <xs:attribute name="identifierQuote" type="xs:string"
use="required"/>
                 <xs:attribute name="nullSortOrder" type="tns:nullSortOrderType"
use="required"/>
                    <xs:attribute name="supportsCatalogsInDataManipulation"
type="xs:boolean" use="required"/>
                 <xs:attribute name="supportsLikeEscapeClause" type="xs:boolean"
use="required"/>
                    <xs:attribute name="supportsSchemasInDataManipulation"
type="xs:boolean" use="required"/>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
              <xs:attribute name="name" type="xs:string" use="required"/>
              <xs:attribute name="dbType" type="xs:string"/>
              <xs:attribute name="dbVersion" type="xs:string"/>
              <xs:attribute name="driver" type="xs:string"/>
              <xs:attribute name="uri" type="xs:string"/>
              <xs:attribute name="username" type="xs:string"/>
              <xs:attribute name="password" type="xs:string"/>
              <xs:attribute name="SID" type="xs:string"/>
              <xs:attribute name="sourceBindingProviderClassName"
type="xs:string"/>
            </xs:complexType>
          </xs:element>
          <!-- choice 6: user defined view -->
```

```
          <xs:element name="userDefinedView" minOccurs="0"/>
          <!-- choice 7: nothing, defaults to userDefinedView -->
          <xs:sequence/>
        </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!--=========================-->
  <!--  function annotation  -->
  <!--=========================-->
  <xs:element name="function">
    <xs:complexType>
      <xs:sequence>
        <!-- standard properties -->
        <xs:element name="author" type="xs:string" minOccurs="0"/>
        <xs:element name="comment" type="xs:string" minOccurs="0"/>
        <xs:element name="version" type="xs:decimal" minOccurs="0"/>
        <xs:element name="documentation" type="xs:string" minOccurs="0"/>
        <!-- user defined properties -->
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element name="property">
            <xs:complexType>
              <xs:simpleContent>
                <xs:extension base="xs:string">
                  <xs:attribute name="name" type="xs:string"/>
                </xs:extension>
              </xs:simpleContent>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <!-- UI properties -->
        <xs:element name="uiProperties" minOccurs="0">
          <xs:complexType>
            <xs:sequence maxOccurs="unbounded">
              <xs:element name="component">
                <xs:complexType>
                  <xs:sequence>
                 <xs:element name="treeInfo" minOccurs="0" maxOccurs="unbounded">
                       <xs:complexType>
                         <xs:sequence>
                         <xs:element name="collapsedNodes" minOccurs="0">
                             <xs:complexType>
                               <xs:sequence>
                             <xs:element name="collapsedNode" type="xs:string"
minOccurs="0" maxOccurs="unbounded"/>
                               </xs:sequence>
                             </xs:complexType>
                           </xs:element>
                         </xs:sequence>
```

```
                        <xs:attribute name="id" type="xs:string"/>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="identifier" type="xs:string"/>
                  <xs:attribute name="minimized" type="xs:boolean"
default="false"/>
                  <xs:attribute name="x" type="xs:int"/>
                  <xs:attribute name="y" type="xs:int"/>
                  <xs:attribute name="w" type="xs:int"/>
                  <xs:attribute name="h" type="xs:int"/>
                  <xs:attribute name="viewPosX" type="xs:int"/>
                  <xs:attribute name="viewPosY" type="xs:int"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <!-- sql statement -->
        <xs:element name="sql" minOccurs="0">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
               <xs:attribute name="isSubquery" type="xs:boolean" default="true"/>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <!-- cache -->
        <xs:element name="nonCacheable" minOccurs="0">
          <xs:complexType/>
        </xs:element>
        <!-- signature: used by java functions and stored procedures -->
        <xs:element name="params" minOccurs="0">
          <xs:complexType>
            <xs:sequence maxOccurs="unbounded">
              <xs:element name="param">
                <xs:complexType>
                  <xs:attribute name="name" type="xs:string"/>
                  <xs:attribute name="nativeType" type="xs:string"/>
                  <xs:attribute name="nativeTypeCode" type="xs:int"/>
                  <xs:attribute name="xqueryType" type="xs:QName"/>
                  <xs:attribute name="kind" type="tns:paramKindType"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
       <!-- interceptor configuration: used by webservice SOAP interceptors -->
```

```
        <xs:element name="interceptorConfiguration" minOccurs="0">
          <xs:complexType>
            <xs:attribute name="aliasName" type="xs:string" use="required"/>
            <xs:attribute name="fileName" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="kind" type="tns:functionKindType"/>
      <xs:attribute name="roleName" type="xs:string"/>
      <xs:attribute name="nativeName" type="xs:string"/>
      <xs:attribute name="nativeLevel1Container" type="xs:string"/>
      <xs:attribute name="nativeLevel2Container" type="xs:string"/>
      <xs:attribute name="nativeLevel3Container" type="xs:string"/>
      <xs:attribute name="style" type="tns:functionStyleType"/>
    </xs:complexType>
  </xs:element>
  <!--==================-->
  <!--  common types  -->
  <!--==================-->
  <xs:simpleType name="functionKindType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="read"/>
      <xs:enumeration value="navigate"/>
      <xs:enumeration value="private"/>
      <xs:enumeration value="library"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="functionStyleType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="table"/>
      <xs:enumeration value="view"/>
      <xs:enumeration value="storedProcedure"/>
      <xs:enumeration value="sqlQuery"/>
      <xs:enumeration value="document"/>
      <xs:enumeration value="rpc"/>
    </xs:restriction>
  </xs:simpleType>
  <!-- used by stored procedures -->
  <xs:simpleType name="paramKindType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="unknown"/>
      <xs:enumeration value="in"/>
      <xs:enumeration value="inout"/>
      <xs:enumeration value="out"/>
      <xs:enumeration value="return"/>
      <xs:enumeration value="result"/>
    </xs:restriction>
  </xs:simpleType>
  <!-- used by maxOccurs in relationship -->
```

```
  <xs:simpleType name="allNNI">
    <xs:union memberTypes="xs:nonNegativeInteger">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="unbounded"/>
          <xs:enumeration value=""/>
        </xs:restriction>
      </xs:simpleType>
    </xs:union>
  </xs:simpleType>
  <!-- used by relationships -->
  <xs:simpleType name="roleType">
    <xs:restriction base="xs:nonNegativeInteger">
      <xs:enumeration value="1"/>
      <xs:enumeration value="2"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="autoNumberType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="identity"/>
      <xs:enumeration value="sequence"/>
      <xs:enumeration value="userComputed"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="nullSortOrderType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="high"/>
      <xs:enumeration value="low"/>
      <xs:enumeration value="unknown"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="scaleType">
    <xs:union memberTypes="xs:int">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="null"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:union>
  </xs:simpleType>
</xs:schema>
```