BEA AquaLogic Data Services Platform™

XQuery Developer's Guide

Note: Product documentation may be revised post-release and made available from the following BEA e-docs site:

http://e-docs.bea.com/aldsp/docs21/index.html

Version: 2.1
Document Date: June 2005
Revised: March 2006

Copyright

Restricted Rights Legend

Trademarks or Service Marks

March 16, 2006 3:40 pm

Contents

Introducing the Data Services Platform XQuery Engine

BEA's XQuery Implementation

XQuery Engine and SQL

Understanding XML Namespaces

Best Practices Using XQuery

Understanding Data Services Platform Annotations

Annotations Reference

XQuery-SQL Mapping Reference

CHAPTER 1

Introducing the Data Services Platform XQuery Engine

This chapter briefly introduces the BEA AquaLogic Data Services Platform XQuery language and describes the version of the XQuery specification implemented in Data Services Platform (DSP). Links to more information about XQuery are also provided.

The following topics are covered:

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

Note: Data Services Platform was initially named Liquid Data. Some artifacts of the original name remain in the product, installation path, and components.

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 Data Services Platform

Data Services Platform models the contents of various types of data sources as XML schemas. Once you have configured Data Services Platform 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 Data Services Platform. Data Services Platform evaluates the query, fetches the data from the underlying data sources, and returns the query results.

For more information on developing data service XQueries 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 Data Services Platform, see "BEA XQuery Language Implementation" on page 2-26.

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's XQuery Implementation

The World Wide Web Consortium (W3C) defines a set of language features and functions for XQuery. The BEA AquaLogic Data Services Platform 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 XQuery engine.

The chapter includes the following topics:

BEA XQuery Function Implementation

Data Services Platform (DSP) supports the W3C Working Draft "XQuery 1.0 and XPath 2.0 Functions and Operators" dated 23 July 2004 (http://www.w3.org/TR/2004/WD-xpath-functions-20040723/). In addition, DSP 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

Data Services Platform (DSP) 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 Data Services Platform 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 DSP 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 global 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 2-11.

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 2-11.

Examples:

- `fn-bea:date-to-string-with-format("by-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 2-11.

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 2-11.

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 2-11.

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 2-11.

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 DSP 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, DSP returns the value of the second parameter (then). If the condition is false, DSP returns the value of the third parameter (else). If the returned condition is not a Boolean value, DSP 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
- fn-bea:inlinedXML
- fn-bea:rename

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 DSP 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("<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 two integers specifying the characters in the string input that match the input regular expression (or an empty list, if none found). 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. DSP 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

Implementation-Specific Functions and Operators

In addition to the support for XQuery functions and operators and the BEA extensions described previously, the W3C Working Draft "XQuery 1.0 and XPath 2.0 Functions and Operators" dated 23 July 2004 (http://www.w3.org/TR/2004/WD-xpath-functions-20040723/) allows implementors to use their discretion in implementing various aspects of the specification, as listed in Table 2-4.

Table 2-4 Implementation-Defined Values

Section	Description	DSP XQuery Engine
6.2—Operators on Numeric Values [Overflow and Underflow during Arithmetic Operations]	Choice between raising an error and other options for overflow or underflow of numeric operations.	Arithmetic overflow and underflow follows behavior of the underlying Application Server's JVM (Java Virtual Machine).
6.2—Operators on Numeric Values [xs:decimal value digit precision]	Number of digits of precision for xs:decimal results	18 digits.
7.4.6—fn:normalize-unicode	In addition to supporting required normalization form "NFC", conforming implementations may also support implementation-defined semantics.	Not supported.
7.5—Functions Based on Substring Matching	Ability to decompose strings into collation units.	No collations supporting this feature are available.
10.1.1—Limits and Precision	Limits and precision for Durations, Dates and Times larger then those specified in XML Schema Part 2: Data Types	Fractional seconds are supported for more than 3 digits of accuracy: seven digits for serialized data (binXML package), 18 digits during computations.
15.5.4—Functions and Operators on Sequences [fn:doc]	Processing or document URI, usage of DTD or Schema for validation, handling of non-XML media types and construction of data model instances from non-XML resources and error handling for document processing.	fn:doc() function does not validate. DSP uses predefined external functions for access to external XML and non-XML data sources.

BEA XQuery Language Implementation

This section describes the BEA XQuery language implementation, and contains the following topics:

- XQuery Language Support (and Unsupported Features)
- Extensions to the XQuery Language in the DSP XQuery Engine
- Implementation-Defined Values for XQuery Language Processing

XQuery Language Support (and Unsupported Features)

The Data Services Platform (Version: 2.0.1) conforms to the W3C Working Draft "XQuery 1.0: An XML Query Language" dated 23 July 2004 (http://www.w3.org/TR/2004/WD-xquery-20040723/), with these exceptions:

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

Extensions to the XQuery Language in the DSP XQuery Engine

Beyond compliance with the specification, BEA AquaLogic Data Services Platform's XQuery language implementation (the DSP XQuery engine) extends the XQuery language via the following:

- 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 2-5 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 2-6 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-Defined Values for XQuery Language Processing

In addition, for some aspects of language processing, the W3C working draft document leaves the details to the implementor's discretion, but requires each implementor to specify and document the implementation details. All such "implementation defined[1]" language features of the XQuery language as implemented in BEA AquaLogic Data Services Platform Version: 2.0.1 are listed in Table 2-7.

Table 2-7 Implementation-Defined Values

Section	Description	DSP XQuery Engine
2.1.2—Dynamic Context	Implicit timezone (value of type xdt:dayTimeDuration) that will be used when a date, time, or dateTime value that does not have a timezone is used in a comparison (or any other operation).	Timezone of the JVM of the underlying application server.

1. "Possibly differing between implementations, but specified and documented by the implementor for each particular implementation."

2.5.1—Kinds of Errors—Static Error	Mechanism for reporting static errors (errors that must be detected during the analysis phase, such as syntax errors).	Parser and compiler APIs throw Java exceptions
2.5.1—Kinds of Errors—Warnings	In addition to static, dynamic, and type errors, an XQuery implementation can (optionally) raise warnings during the analsyis or evaluation phases, in response to specific conditions.	Provides a WarningListener API, but has no special warnings defined for the core XQuery language implementation
2.6.3—Full Axis Feature	Set of optional axes when Full Axis Feature is not supported	None.
2.6.6.1—Must-Understand Extensions—XQuery Flagger	Mechanism by which the XQuery Flagger (which flags queries containing 'must understand' extensions) is enabled, if at all—by default, it is disabled.	XQuery Flagger is not supported.
2.6.7.1—Static Typing Extensions—XQuery Static Flagger	Mechanism by which the XQuery Static Flagger is provided, if at all.	XQuery Static Flagger is not supported.
3.1.1—Literals	Choice of XML 1.0 or XML 1.1 for character references (the XML-style references for Unicode characters, such as — for an em-dash).	XML 1.0
3.7.1.2—Namespace Declaration Attributes	Support for XML Names 1.1	No
3.8.3—Order By and Return Clauses	Ordering specification (orderspec) can be implemented as *empty least* or *empty greatest* (for evaluating greater-than relationship between two orderspec values in an order by clause of an XQuery).	Empty least.

4.10—Module Import	String literals following the `at` keyword are optional location hints in module import statements that can be interpreted (or disregarded) by the implementor.	Not applicable—Since the DSP XQuery engine does not support modules, there is no implementation.
4.13—Function Declaration	Protocol by which parameters are passed to an external function and the result of the function is returned to the invoking query.	Set of Java APIs provided.
A.2—Lexical structure	Lexical rules can follow XML 1.0 and XML Names, or XML 1.1 and XML Names 1.1.	XML 1.0 and XML Names

CHAPTER 3

XQuery Engine and SQL

This chapter provides an overview of how Data Services Platform works with relational data, especially focusing on many of the translations that must occur between XQuery and SQL: What happens when a relational data source is imported into DSP? How are SQL data types mapped to XQuery data types, and vice versa? What happens at runtime, after you have deployed a data-service-enabled application—how are the various types of queries handled, and what kind of performance can you expect?

Although the graphical-user interface tools component of BEA AquaLogic Data Services Platform available in WebLogic Workshop in many ways obviates the need for developers to get mired in many of these details, SQL developers and application-performance tuning experts should understand how DSP works with relational data so that they can:

- Create well-designed canonical data services that are potentially re-usable throughout an organization;
- Test and tune alternative query approaches;
- Validate execution paths for queries and identify opportunities to improve overall performance.

To facilitate developer's efforts with these tasks, this chapter includes these topics:

Note: For simplicity's sake, this chapter refers to the XQuery engine throughout when in fact some of the specific functionality is handled by other, ancillary sub-systems (for example, the Data Source API or other system components depicted in the "Data Services Platform Components Architecture" figure in the *Concepts Guide*).

Introduction

At the core of BEA AquaLogic Data Services Platform (DSP) is the data processing engine, often referred to as simply the XQuery engine—the robust, enterprise-class implementation of the XQuery language based on the standards listed in "Supported XQuery Specifications" on page 1-2, with additional enhancements as detailed in "BEA's XQuery Implementation" on page 2-1.

In addition to compliance with XQuery and XML recommendations, DSP XQuery engine also complies with the ANSI/ISO standard that bridges the SQL and XML worlds (the "SQL/XML (ISO-ANSI Working Draft) XML-Related Specifications" WD 9075-14 (SQL/XML), August, 2002). As a Java application (J2EE server application), Data Services Platform uses JDBC to generate SQL queries and submit them to the appropriate RDBMSs that comprise a data service, which means DSP must accommodate differences in both SQL and JDBC, as follows:

- **SQL Language.** The SQL standard has evolved over time, and vendor implementations (in their respective RDBMS products) may be at any number of stages of compliance with the standard (SQL-89, SQL-92, SQL:1999, and SQL:2003, for example). Furthermore, vendors implement various extensions to SQL in their respective RDBMS products. In short, DSP's support for SQL is not a "one-size-fits-all" exercise: achieving optimal integration with relational data sources requires DSP to generate vendor-specific SQL code at times.
- **JDBC API.** Drivers are provided by RDBMS vendors as well as third-parties; various drivers for each RDBMS can have different levels of JDBC compatibility.

Given these factors, BEA AquaLogic Data Services Platform (DSP) provides two different levels of SQL support for relational database management systems (RDBMS): base support and core support, as defined in the next section.

Base and Core RDBMS Support

DSP provides two different levels of support for relational data sources:

- **Base support**. Data Services Platform generates standard SQL code that is minimally required to be supported by any SQL RDBMS. Some examples of base platforms would include Oracle 7, Informix, IDMS, MySQL, and Teradata.

- **Core support**. Data Services Platform supports the native SQL dialect of specific versions of several leading commercial RDBMSs using the RDBMS-specific-JDBC of the vendor's JDBC driver or BEA's JDBC driver (see Table 3-1).

Table 3-1 Core Data Services Platform RDBMS Support

RDBMS and Versions	Vendor Driver	BEA WebLogic Driver
IBM DB2/NT 8	IBM DB2 JDBC thin driver, version 8.01	BEA (DataDirect) JDBC driver for DB2, version 3.4.
Microsoft SQL Server 2000	Microsoft SQLServer JDBC driver, version 2.2	BEA (DataDirect) JDBC driver for SQLServer, version 3.4
Oracle 8.1.x, 9.x, 10.x	Oracle JDBC Thin driver, version 10.1	BEA (DataDirect) JDBC driver for Oracle, version 3.4
Pointbase 4.4 (and higher)	Pointbase JDBC driver, version 4.4	N/A
Sybase Adaptive Server Enterprise 12.5.2 (and higher)	Sybase jConnect driver, version 5.5	BEA (DataDirect) JDBC driver for Sybase, version 3.4

How it Works—XQuery Engine's Support for SQL

BEA AquaLogic Data Services Platform supports SQL (relational) data sources throughout the life-cycle of a data services project, from metadata import, through query plan optimization, through runtime execution of queries and delivery of data to an end-user (or other) application. Specifically, the XQuery engine provides:

- **Metadata Mapping.** Importing metadata from relational data sources is the first step in creating a data service.
- **Data Type Mapping**. Upon import of metadata, DSP maps data types from the RDBMS data source into XQuery atomic data types, disregarding length and other constraints. If the data source tables or views include unsupported data types—an array, for example—the column is ignored (the GUI tool alerts the person performing the import if this issue arises, and enables the person to map the data type of the source table or view to a specific XQuery data type).
- **Query Optimization.** The XQuery processing engine is fast and efficient, and uses several optimizing strategies, including:

  - **SQL pushdown**. As much as possible, processing is shifted from the XQuery engine to the native RDBMS so that smallest practical result set is actually processed by the XQuery engine.
  - **Lazy evaluation**. Queries are executed against the physical data sources only as far as necessary to obtain results.
  - **Connection-sharing**. Multiple active queries can run over a single connection (assuming the data source RDBMS allows; see Table 3-2, "Runtime Connection Management," on page 3-5).

Metadata and Data Type Mappings Get Stored in Annotated Files

For each of the tables and views whose metadata is imported into DSP (using Import Source Metadata feature of the GUI), two files are generated:

- **Data service (.ds) file** that defines the main access function (an external XQuery function with annotations that specify the RDBMS catalog or schema name and other properties) to access to the table or view data and return a sequence of elements corresponding to the rows of the underlying table. The .ds file includes numerous annotations to handle metadata about the data service, including:
  - Database configuration information, including RDBMS brand name and version information (Oracle 9.x.x and Sybase 12.x.x., for example).
  - Table structure information, including column names (field names), SQL data types and corresponding XQuery data types, primary key, and foreign key information.
  - Relationship functions that provide access to related tables or views.
  - Relationship annotations.
  - JNDI lookup information. The <relationalDB> annotation in the data service file provides the JNDI name that will be used at runtime to obtain a connection to the data source and execute queries.
- **XML Schema definition (.xsd) file** that includes information about all the columns of the table (or view) and the data types for those columns, as mapped into the XQuery data types.

Runtime Connection Management—Connection Sharing

At runtime, the XQuery engine:

- Obtains a connection to the RDBMS.
- Prepares SQL statements, setting up parameters if necessary.

- Executes the SQL statements and releases the connection.
- Handles errors and exceptions.
- Translates the result of the query to the XML model used by XQuery engine.

Database connections (connection pools) are registered in the JNDI (Java naming and directory interface) tree of the WebLogic Server (an administrator with privileges on the server can configure connection pool, data source, and JNDI name by which connection pools are accessible).

When sub-plan execution completes, connections are typically not released back to the WebLogic Server. The XQuery engine holds the connection for the duration of the entire XQuery—not just the duration of the SQL—enabling subsequent queries to the same relational data source to be executed using an already obtained connection (which also improves performance). Whether the XQuery engine can share connections or not depends on the underlying data source and JDBC driver (see Table 3-2).

If the data source RDBMS or JDBC driver does not support connection sharing, and if the DSP has opened multiple connections to the same data source, the XQuery engine keeps the initial connection to a data source open during XQuery execution but releases any subsequent connections to the same data source once the SQL result is received in its entirety by the XQuery engine. The initial connection will be re-used subsequent SQL queries when the connection becomes available.

Table 3-2 Runtime Connection Management

RDBMS	Support
Base RDBMS	No connection sharing.
IBM DB2/NT 8 Microsoft SQL Server 2000 Oracle 8.1.x, 9.x, 10.x Sybase Adaptive Server Enterprise 12.5.2 (and higher)	Single shared connection for each JNDI data source; each connection supports multiple active SQL queries.
Pointbase 4.4 (and higher)	No connection sharing. Each access requires dedicated connection.

XQuery-SQL Data Type Mappings

XQuery-SQL data type mappings are specific to the RDBMS version and the JDBC driver, as discussed in "Base and Core RDBMS Support" on page 3-2. The specific data type mappings for each core RDBMS

and the general mappings for any base RDBMS are detailed in the "XQuery-SQL Mapping Reference." However, XQuery and SQL differ in some respects that may affect XQuery-to-SQL translation; these differences apply to all RDBMSs:

- Date and Time Data Type Differences: Timezones and Time Precision
- Scope Differences for Expressions and Data Types

Date and Time Data Type Differences: Timezones and Time Precision

The XQuery language defines richer data types than SQL for handling date and time information (temporal data). These data types provide more information (timezone data, for instance) or greater degree of precision (unlimited number of fractional seconds as part of a time or date, for example). The three built-in XQuery data types for data and time information are:

- xs:dateTime
- xs:date
- xs:time

Minimally, every RDBMS has a single datatype that conveys both date and time data. This datatype maps to XQuery's xs:dateTime data type. Some RDBMSs offer additional SQL data types for storing date and time data separately (see Table 3-3)

(Of all the RDBMSs supported by DSP, only Oracle 9.x (and higher) offers data types with timezone data (TIMESTAMP WITH TIMEZONE, TIMESTAMP WITH LOCAL TIMEZONE).

Table 3-3 Temporal Data Type Mappings

	xs:date	xs:dateTime	xs:time
Base RDBMS		Reported by JDBC driver for the specific RDBMS.	
IBM DB2/NT 8	DATE	TIMESTAMP	TIME
Microsoft SQL Server 2000		DATETIME[1], SMALLDATETIME[2]	
Oracle 8.1.x		DATE[3]	

Table 3-3 Temporal Data Type Mappings

Oracle 9.x, 10.x		DATE, TIMESTAMP, TIMESTAMP WITH LOCAL TIMEZONE, TIMESTAMPWITH TIMEZONE	
Pointbase 4.4 (and higher)	DATE	TIMESTAMP	TIME
Sybase Adaptive Server Enterprise 12.5.2 (and higher)	DATE	SMALLDATETIME,[2] DATETIME[1]	TIME

1. Supports fractional seconds up to 3 digits (miliseconds).
2. Accuracy of 1 minute.
3. Provides both date and time data, but supports neither fractional seconds nor timezone data (fractional-second data is truncated).

DSP XQuery engine maps all SQL date and time data types to XQuery data types (for example, during metadata import of a new data source) without loss of data or precision.

However, the converse is not true: depending on the specific RDBMS (and JDBC driver) for a specific data source, the XQuery engine may need to perform additional processing to minimize data loss and to handle the timezone information when mapping XQuery temporal data types to SQL.

How DSP Handles Timezone Information

When a query is being pushed down to an RDBMS that does not support timezone data, the DSP XQuery engine converts date and time data into the local time of the underlying application server and removes the timezone information. The conversion occurs each time a date or time value that includes timezone data is sent to the data source, as follows:

- During compile time, when SQL is generated for constant date or time expressions.
- During query run time, when executing parameterized SQL with parameters bound to date/time values.
- During SDO update, when a date or time value must be stored in the RDBMS.

How DSP Handles Fractional Seconds

The XQuery language supports unlimited precision for fractional seconds, while the DSP XQuery engine supports up to 7 digits only (for fractional seconds). However, depending on the specific

RDBMS, fractional second support may be far less than 7 digits — or may not be supported at all (Oracle 8.1.x, for example). In translating from XQuery to SQL, DSP truncates fractional seconds to the precision supported by that RDBMS.

For example, since Microsoft's DATETIME data type supports up to 3 digits (milliseconds) for fractional time precision, when DSP sends a datetime value to Microsoft SQL Server 2000, the value is first converted into the local time zone and then any fractional seconds are converted to the 3-digit-milliseconds allowed.

If fractional-second-precision is required (but the data source does not support it appropriately), use the fn-bea:fence() function to disable pushdown of date and time data types and operations, so that the XQuery engine processes the time- and date-related queries. (See "Preventing SQL Pushdown" on page 3-32 for more information.)

See "XQuery-SQL Mapping Reference" for more information about time and date data types for core and base RDBMS.

Scope Differences for Expressions and Data Types

The XQuery language is less restrictive than the SQL language in terms of the scope of expressions and data types. For example, for most all RDBMSs, an SQL query that returns a boolean can only be used inside a WHERE clause. XQuery does not have such restrictions, and as a result, in some cases, valid XQuery expressions cannot be *pushed down*. Expressions and data types that cannot be pushed include:

- expressions returning boolean type can only be used in the WHERE clause (all RDBMSs)
- some data types, such as CLOB, can be returned in the project list but cannot be grouped on or sorted on (depending on the RDBMS's SQL dialect; see "XQuery-SQL Mapping Reference" for details).
- aggregate functions inside an ordering expression, such as in ORDER BY clauses, are not pushed down for any base RDBMS or Pointbase or (but is supported by all other RDBMSs. See "XQuery-SQL Mapping Reference" for more information.

SQL Pushdown: Performance Optimization

Data Services Platform achieves optimal performance for queries by performing *SQL pushdown*—an optimization technique that offloads processing from the XQuery engine by sending native SQL queries to the data source so that minimal result sets necessary to answer the query get processed by the XQuery engine.

SQL pushdown reduces the amount of data transported and processed by DSP XQuery processing engine. This technique dramatically improves overall performance, especially when joining tables.

For example, a JOIN operation on two tables can be done by the underlying RDBMS, returning only the final result, rather than delivering all the data to the XQuery engine for processing the JOIN condition. Sorting criteria are also handled by the data source, eliminating the need to re-sort the data inside the XQuery engine.

For all core RDBMSs, the XQuery engine identifies the XQuery constructs and operations that can be translated into equivalent SQL operations. These include:

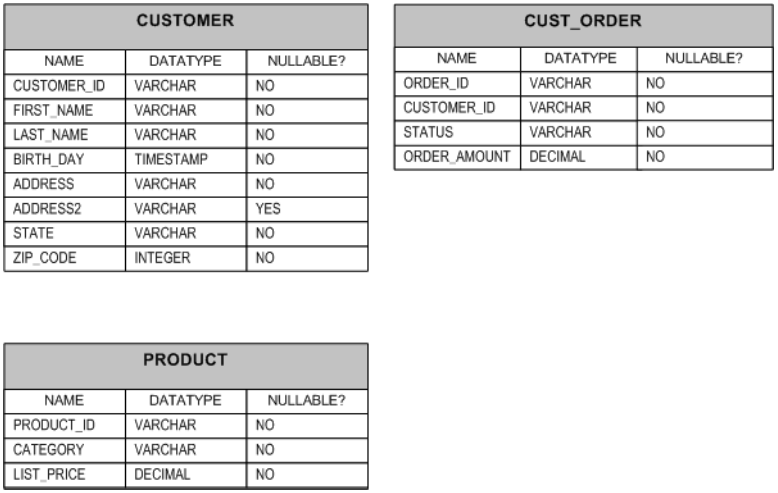
- Basic language constructs, including constants, variables, path expressions, functions and operators, and cast operations.
- Common query patterns, such as selections and projections (where clauses), joins (inner, outer, semi-join, anti-semi-join), ordering clauses, groupings and aggregations.

Not all queries can (or should) get pushed down. The XQuery engine does not pushdown:

- **Cross-joins**. Any join without a condition (any join that results in a Cartesian product)
- **Expressions tagged with the fn-bea:fence() function.**

The remainder of this section covers SQL pushdown in more detail, providing syntax samples based on the table structures shown in Figure 3-4. (For ease of reading, namespace references are not shown in the example queries.) In some cases, the query may not get pushed down as SQL, but the fragments of the query—names of columns, for example—may get pushed to the project list.

Figure 3-4 Table Structures for SQL Pushdown Examples



CUSTOMER

NAME	DATATYPE	NULLABLE?
CUSTOMER_ID	VARCHAR	NO
FIRST_NAME	VARCHAR	NO
LAST_NAME	VARCHAR	NO
BIRTH_DAY	TIMESTAMP	NO
ADDRESS	VARCHAR	NO
ADDRESS2	VARCHAR	YES
STATE	VARCHAR	NO
ZIP_CODE	INTEGER	NO

CUST_ORDER

NAME	DATATYPE	NULLABLE?
ORDER_ID	VARCHAR	NO
CUSTOMER_ID	VARCHAR	NO
STATUS	VARCHAR	NO
ORDER_AMOUNT	DECIMAL	NO

PRODUCT

NAME	DATATYPE	NULLABLE?
PRODUCT_ID	VARCHAR	NO
CATEGORY	VARCHAR	NO
LIST_PRICE	DECIMAL	NO

Function and Operator Pushdown

XQuery functions and operators are translated into SQL only when:

- all arguments can be pushed down directly (or as parameters)
- at least one of the argument expressions uses a value from the relational data source
- the XQuery function or operator has an equivalent SQL expression with equivalent semantics
- data type of the result is supported

Table 3-5 Function Pushdown Example

XQuery Statement	SQL Translation (Oracle Syntax)
for $c in CUSTOMER() return **lower-case**($c/LAST_NAME)	SELECT **LOWER**(t1."LAST_NAME") AS c1 FROM "CUSTOMER" t1

If some arguments to a function or operator are not directly pushable, but can be replaced with parameters, the XQuery engine will replace the arguments with parameters and pushdown the SQL. For example, since the XQuery's string-join() function has no explicit SQL equivalent, it is replaced with a parameter (see Table 3-6).

Table 3-6 External Variable Pushdown

XQuery Statement	SQL Statement
declare variable $p as xs:string external; ... for $c in CUSTOMER() where **starts-with($c/LAST_NAME, string-join(("a", "b"), $p))** return $c/FIRST_NAME	SELECT t1."FIRST_NAME" AS c1 FROM "CUSTOMER" t1 WHERE **t1."LAST_NAME" LIKE ?**

Aggregate Functions

DSP translates XQuery 1.0 and XPath 2.0 aggregate functions into corresponding SQL aggregate functions (Table 3-7).

Table 3-7 Aggregate Functions

XQuery Aggregate Function	SQL Aggregate Function
fn:avg()	AVG()
fn:count()	COUNT()
fn:max()	MAX()
fn:min()	MIN()
fn:sum()	SUM()
fn:count(fn:distinct-values()	COUNT(DISTINCT …)

Note that the distinct-values() XQuery aggregate function in conjunction with the fn:count() function is further translated into an SQL COUNT(DISTINCT...) operation, as shown in Table 3-7. See "Grouping and Aggregation" on page 3-21 for some examples of how aggregate functions in conjunction with other expressions affect the outcome of SQL pushdown.

Parameters in Generated SQL Statements

The DSP XQuery engine generates parameters from variables, functions, operators, and cast operations as needed for use by the SQL engine. If all arguments to a function are parameters, the entire function gets pushed as a parameter.

The functions that can be pushed down depend on the database. See the "XQuery-SQL Mapping Reference" on page B-1 for details.

Cast Operation Pushdown

As with functions and operators, support for cast operation pushdown is RDBMS-specific, although cast pushdown is available only for core (not base) RDBMSs. The XQuery engine can pushdown cast operations if the data source RDBMS:

- has equivalent SQL data types for both source and target of the cast XQuery data types (see the "XQuery Engine and SQL" appendix for details).
- has a semantically equivalent SQL operation to convert from source data type to target data type.

Table 3-8 shows an example of how a cast in XQuery would get pushed down to a Microsoft SQL Server 2000 data source.

Table 3-8 Cast Operation Pushdown

XQuery Statement	SQL Statement (Microsoft SQL Server 2000 Syntax)
for $c in CUSTOMER() where xs:string($c/ZIP_CODE) eq "95131" return $c/CUSTOMER_ID	SELECT t1."CUSTOMER_ID" AS c1 FROM "CUSTOMER" t1 WHERE CAST(t1."ZIP_CODE" AS VARCHAR) = '95131'

Path Expressions Pushdown

The XQuery engine maps table columns to XML elements that are children of the corresponding row elements. Simple XQuery path expressions are recognized by the XQuery engine as column accessors. For example, $c/ZIP_CODE and $c/LAST_NAME (see Table 3-9) provide access to ZIP_CODE and LAST_NAME columns.

Constant Pushdown

The DSP XQuery engine translates XQuery constants into SQL constants only if the data source has an equivalent SQL data type. Table 3-9 shows an example of a constant used in a FLWOR expression and how that constant gets translated in the SQL statement.

Table 3-9 SQL Pushdown for Constants

XQuery Statement	SQL Statement
for $c in CUSTOMER() where $c/ZIP_CODE eq **95131** return $c/LAST_NAME	SELECT t1."LAST_NAME" AS c1 FROM "CUSTOMER" t1 WHERE t1."ZIP_CODE" = **95131**

Variable Pushdown

Both external and internal variables in XQuery expressions can be translated into SQL parameters (in generated SQL statements) when the variable's data type is supported by the XQuery engine and:

- is atomic (static data type).
- can be translated into equivalent SQL type.

Table 3-10 Variable Pushdown

XQuery Statement	SQL Statement
declare variable **$extVar** as xs:string external; for $c in CUSTOMER() where $c/CUSTOMER_ID eq **$extVar** return $c/LAST_NAME	SELECT t1."LAST_NAME" as c1 FROM "CUSTOMER" t1 WHERE t1."CUSTOMER_ID" = **?**

Common Query Patterns

For each relational data source, the precise set of expressions pushed down depends on the capabilities of the underlying RDBMS; for details, see "XQuery Engine and SQL" on page 3-1.

Simple Projection Queries

Each of the example XQueries shown in Table 3-11 returns elements containing values of LAST_NAME columns from a CUSTOMER table. In all cases, the SQL statement generated by the DSP XQuery engine is the same (see Table 3-11).

Table 3-11 Projection Query

XQuery Statements	SQL Statement
for $c in CUSTOMER() return $c/LAST_NAME	SELECT t1."LAST_NAME" AS c1 FROM "CUSTOMER" t1
CUSTOMER()/LAST_NAME	
for $c in CUSTOMER() return data($c/LAST_NAME)	
data(CUSTOMER()/LAST_NAME)	

The difference between the first two queries and the last two queries is that the fn:data() function is used in the query to limit the results to values only. Without the fn:data() function, the result is a list of <LAST_NAME> elements containing corresponding column values. If a column value is NULL, the element is skipped. With the fn:data() function, the result is the actual values.

Where Clause Pushdown

An XQuery where clause is usually translated into an SQL WHERE clause. An XQuery where clause gets pushed down as SQL when:

- the where expression uses at least one value from a relational source.
- the where expression is pushable (using parameters if needed). See "SQL Pushdown: Performance Optimization" on page 3-8 for more information.

Table 3-12 Where Clause Pushdown

XQuery Statements	SQL Statements

for $c in CUSTOMER() **where $c/CUSTOMER_ID eq "CUSTOMER01"** return $c/LAST_NAME	SELECT t1."LAST_NAME" AS c1 FROM "CUSTOMER" t1 **WHERE t1."CUSTOMER_ID" = 'CUSTOMER01'**
for $c in CUSTOMER() **where year-from-dateTime($c/BIRTH_DAY)** **eq** **year-from-date(current-date())** return $c/LAST_NAME	**(DB2 syntax)** **SELECT t1."LAST_NAME" AS c1** FROM "CUSTOMER" t1 **WHERE** **YEAR(t1."BIRTH_DAY") = ?**

However, note that if the WHERE clause follows a group by clause, the WHERE clause is translated into a HAVING clause. See "Group-By with a Nested Where Clause Translates to SQL HAVING Clause" on page 3-23).

Order By Clause Pushdown

An XQuery order by expression comprises:

- ordering expression
- direction property for each ordering expression; that is, ascending or descending
- empty ordering property for each ordering expression; that is, empty least or empty greatest

The XQuery engine can pushdown SQL for ordering expressions, including properties, only when the ordering expression:

- is pushable and uses data from the database.
- is of the kind supported by the underlying data source (some RDBMSs can only support order by columns, not arbitrary expressions; some RDBMSs support non-column expressions in order by clause only if they do not contain aggregate functions.
- when an empty expression can result in empty sequence, the RDBMS must support the same NULL order as the empty order specified by the XQuery. (Some RDBMSs have fixed NULL order, some allow NULL order to be specified—see "XQuery Engine and SQL" for details).

Table 3-13 Order By Pushdown

XQuery Statement	SQL Statement
for $c in CUSTOMER() **order by $c/CUSTOMER_ID descending** return $c/CUSTOMER_ID	SELECT t1."CUSTOMER_ID" AS c1 FROM "CUSTOMER" t1 **ORDER BY t1."CUSTOMER_ID" DESC**

Table 3-14 shows an example of the SQL pushdown that occurs when ordering by a NULLable column (ADDRESS2) in the XQuery clause and the RDBMS supports dynamic setting of NULL order.

Table 3-14 Order By Query, Setting NULL Order Dyanmically

XQuery Statement	SQL Statement (Oracle Syntax)
for $c in CUSTOMER() **order by $c/ADDRESS2 ascending** **empty greatest** return $c/CUSTOMER_ID, $c/ADDRESS2	SELECT t1."CUSTOMER_ID" AS c1, t1."ADDRESS2" AS c2 FROM "CUSTOMER" t1 **ORDER BY t1."ADDRESS2" ASC NULLS LAST**

If the data source RDBMS does not support the required empty (NULL) order, the order by will not be pushed down.

As another optimization, the DSP XQuery engine can insert *order by* clauses into generated SQL statements—even when the original XQuery statement does not include them—to offload expensive sorting operations to the RDBMS. They are automatically inserted by the XQuery optimizer prior to execution. You cansee these as well in the Query Plan View.

Inner Join Pushdown

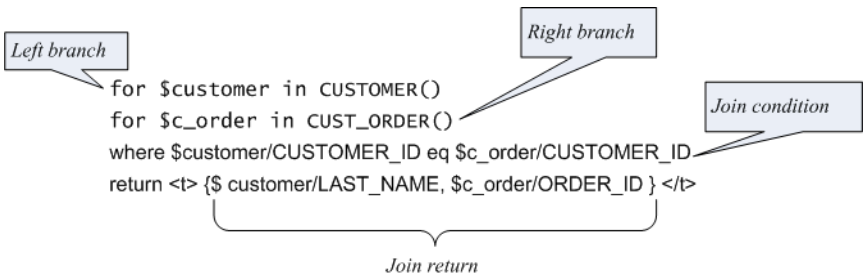
Joining data from multiple sources is a very common data integration task. In SQL terms, an inner join relates each row in one table (or view) to one or more corresponding rows in another table or view. In XQuery, an inner join is expressed as a FLWR expression comprising several *for* clauses that iterate over the data sources, *where* clauses that specify the join predicates, and a *return* clause returning data values.

If two relational sources are located in the same database, the inner join can sometimes be pushed down as a single SQL statement using either SQL-92 or SQL-89 syntax, depending on the RDBMS of the data source.

An inner join can be pushed down when:

- the condition itself is pushable.
- both join branches belong to the same RDBMS and can be addressed from a single SQL statement (both branches are in the same JNDI data source).
- join condition exists and uses values from both branches (cross joins are not pushed down).

Figure 3-15 XQuery Inner Join Pattern



Although the example in Figure 3-15 shows a simple inner join between two branches, the XQuery engine also supports *n*-way joins, with each branch comprising a different for statement.

Table 3-16 Rendering of XQuery Inner-Join as SQL-92 and SQL-89 Syntax

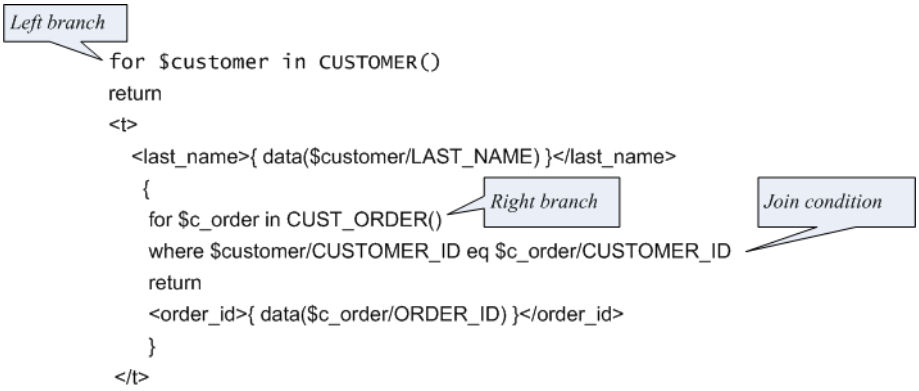
SQL-92 Syntax	SQL-89 Syntax
SELECT t1."LAST_NAME" AS c1, t2."ORDER_ID" AS c2 **FROM** "CUSTOMER" t1 **JOIN** "CUST_ORDER" t2 **ON** t1."CUSTOMER_ID" = t2."CUSTOMER_ID"	SELECT t1."LAST_NAME" AS c1, t2."ORDER_ID" AS c2 **FROM** "CUSTOMER" t1, "CUST_ORDER" t2 **WHERE** t1."CUSTOMER_ID" = t2."CUSTOMER_ID"

Outer Join Pushdown

The XQuery engine interprets nested FLWR expressions (see Figure 3-17) as an outer join and can generate SQL for a data source when:

- both join branches belong to the same database and are addressable from a single SQL statement (both branches must come from the same JNDI datasource), and
- join condition is present and uses values from both branches, and
- join condition is pushable, and
- the underlying RDBMS supports outer join syntax using either SQL-92 or proprietary syntax in its SQL language

Figure 3-17 Outer Join Pattern



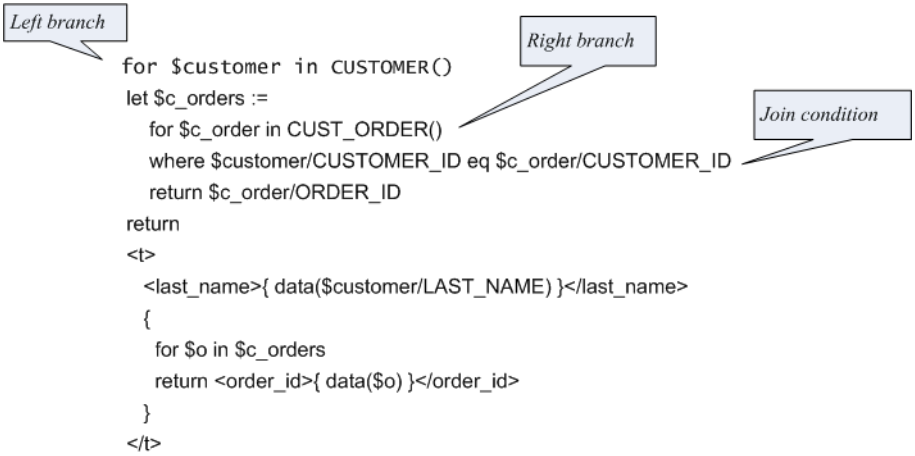
The SQL code generated by the XQuery engine depends on the SQL dialect supported by the source database (see "XQuery-SQL Mapping Reference" for details). Table 3-18 shows example SQL-92 and proprietary syntax for the query shown in Figure 3-17.

Table 3-18 SQL-92 and Proprietary Outer Join Syntax Comparison

SQL-92 Syntax	Oracle 8 Syntax
SELECT t1."LAST_NAME" AS c1, t2."ORDER_ID" AS c2 **FROM** "CUSTOMER" t1 **OUTER JOIN** "CUST_ORDER" t2 **ON** t1."CUSTOMER_ID" = t2."CUSTOMER_ID"	SELECT t1."LAST_NAME" AS c1, t2."ORDER_ID" AS c2 **FROM** "CUSTOMER" t1, "CUST_ORDER" t2 **WHERE** t1."CUSTOMER_ID" = t2."CUSTOMER_ID" **(+)**

Variations of the outer-join pattern are obtained from the original query by using equivalent XQuery expressions. Figure 3-19 is an example of a query equivalent to that shown in Figure 3-17 that will also result in a SQL statement with an outer join.

Figure 3-19 Outer Join Pattern



Semi-Joins and Anti-Semi-Joins

A semi-join returns data from a single branch of the join condition, when the join condition is satisfied. An anti-semi-join returns data from a single branch when the join condition is false. Although the XQuery language does not have specific constructs for semi-joins and anti-semi-joins, the XQuery engine translates several specific FLWR patterns into SQL semi-join or anti-semi-join patterns, assuming that:

- both sides (outer and inner) belong to the same database and are addressable from a single SQL statement (both branches must come from the same JNDI datasource).
- the join condition exists.
- the join condition is pushable.
- the RDBMS supports the EXISTS function and subqueries (see "XQuery-SQL Mapping Reference" on page B-1 for details).

The XQuery interprets a FLWR query containing an inner existential quantified expression as a semi-join, translating the expression into an SQL query with the EXISTS check in the WHERE clause.

Universal quantified expressions are also supported, but their SQL generation is slightly more complicated. The XQuery engine translates FLWRs with exist() or empty() predicates in the where clause into semi-joins. Table 3-20 shows several examples of such patterns.

Table 3-20 Various XQuery Patterns that Can Generate Semi-Join and Anti-Semi-Join SQL

	XQuery Statement	SQL Statement
FLWR with existential ("some") quantifer [semi-join]	for $customer in CUSTOMER() where **some** $c_order in CUST_ORDER() **satisfies** ($customer/CUSTOMER_ID eq $c_order/ORDER_ID) and ($c_order/STATUS eq "OPEN") return $customer/CUSTOMER_ID	SELECT t1."CUSTOMER_ID" AS c1 FROM "CUSTOMER" t1 WHERE **EXISTS**(SELECT 1 FROM "CUST_ORDER" t2 WHERE t1."CUSTOMER_ID" = t2."CUSTOMER_ID" AND t2."STATUS" = 'OPEN')
FLWR with negation of existential quantifier [anti-semi join]	for $customer in CUSTOMER() where **not**(**some** $c_order in CUST_ORDER() **satisfies** ($customer/CUSTOMER_ID eq $c_order/ORDER_ID) and ($c_order/STATUS eq "OPEN")) return $customer/CUSTOMER_ID	SELECT t1."CUSTOMER_ID" AS c1 FROM "CUSTOMER" t1 WHERE **NOT EXISTS**(SELECT 1 FROM "CUST_ORDER" t2 WHERE t1."CUSTOMER_ID" = t2."CUSTOMER_ID" AND t2."STATUS" = 'OPEN')
FLWR with universal ("every") quantified expression	for $customer in CUSTOMER() where **every** $c_order in CUST_ORDER() **satisfies** ($customer/CUSTOMER_ID eq $c_order/ORDER_ID) and ($c_order/STATUS eq "OPEN") return $customer/CUSTOMER_ID	SELECT t1."CUSTOMER_ID" AS c1 FROM "CUSTOMER" t1 WHERE **NOT EXISTS**(SELECT 1 FROM "CUST_ORDER" t2 WHERE **NOT**(t1."CUSTOMER_ID" = t2."CUSTOMER_ID" AND t2."STATUS" = 'OPEN'))

Table 3-20 Various XQuery Patterns that Can Generate Semi-Join and Anti-Semi-Join SQL

FLWR with exists() predicate	or $customer in CUSTOMER() **where exists**(for $c_order in CUST_ORDER() where ($customer/CUSTOMER_ID eq $c_order/ORDER_ID) and ($c_order/STATUS eq "OPEN") return $c_order) return $customer/CUSTOMER_ID	SELECT t1."CUSTOMER_ID" AS c1 FROM "CUSTOMER" t1 WHERE **EXISTS**(SELECT 1 FROM "CUST_ORDER" t2 WHERE t1."CUSTOMER_ID" = t2."CUSTOMER_ID" AND t2."STATUS" = 'OPEN')
FLWR with empty() predicate	for $customer in CUSTOMER() **where empty**(for $c_order in CUST_ORDER() where ($customer/CUSTOMER_ID eq $c_order/ORDER_ID) and ($c_order/STATUS eq "OPEN") return $c_order) return $customer/CUSTOMER_ID	SELECT t1."CUSTOMER_ID" AS c1 FROM "CUSTOMER" t1 WHERE **NOT(EXISTS**(SELECT 1 FROM "CUST_ORDER" t2 WHERE t1."CUSTOMER_ID" = t2."CUSTOMER_ID" AND t2."STATUS" = 'OPEN'))

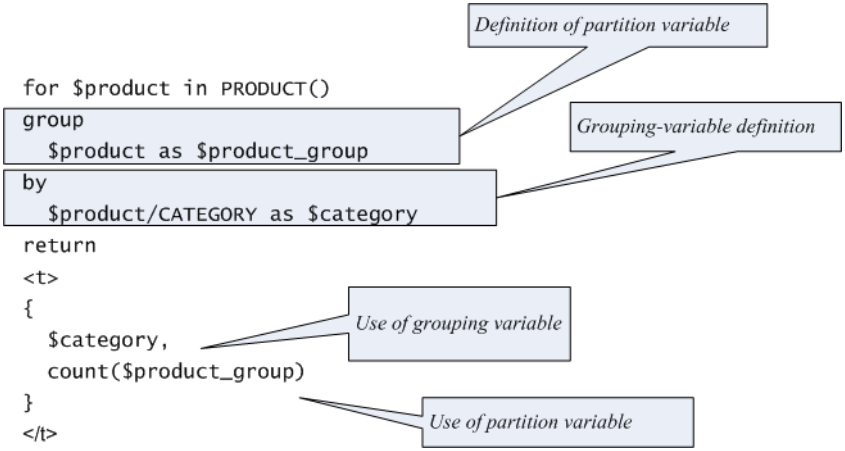
Grouping and Aggregation

The DSP XQuery engine supports several patterns for group by pushdown and aggregate function pushdown.

Group By Pushdown

The Group By clause is a BEA extension to the XQuery language (see "Generalized FLWGOR (group by)" on page 2-26 for more information). The XQuery engine implicitly adds a group by expression to some patterns to enable more efficient pushdown and query execution.

Figure 3-21 XQuery Containing a Group By



The XQuery engine translates group-by clauses into equivalent SQL GROUP BY clauses if:

- the expressions defining grouping variables are pushable
- the partition variable is used by an aggregate function only

Since the query shown in Figure 3-21 meets these requirements, the following SQL statement is generated:

```
SELECT t1."CATEGORY" AS c1, COUNT(*) AS c2

FROM "PRODUCT" t1

GROUP BY t1."CATEGORY"
```

The group-by pushdown is closely related to the Distinct-by Pushdown: When a group-by clause does not include a partition variable, the XQuery engine generates SQL that includes the DISTINCT keyword, as described in the next section.

Distinct-by Pushdown

An XQuery containing a Group By clause (without a partition definition), can be generated into SQL query that uses SQL's DISTINCT keyword to eliminate duplicates in the result. For example, the XQuery statement in Table 3-22 uses a group-by clause but has no partition defined, and the SQL statement created by DSP refines the result by using the DISTINCT keyword.

Table 3-22 Distinct By Pushdown

XQuery Statement	SQL Statement
for $product in PRODUCT() **group by $product/CATEGORY_ID as $category** return $category	SELECT **DISTINCT** t1."CATEGORY_ID" AS c1 FROM "PRODUCT" t1

Trivial Aggregate Pattern

An aggregate function operating on a single column from a data source is one of the simplest aggregate patterns that the XQuery engine supports, although it does so in a slightly non-intuitive way. It uses a constant as a single grouping expression (...GROUP ...BY n). The XQuery engine can pushdown the SQL if the RDBMS supports either a GROUP BY operation on a constant or supports sub-queries in the sub-clause (see Table 3-23).

Table 3-23 Aggregate Pushdown

XQuery Statement	SQL Statement[1]	SQL Statement[2]
for $product in PRODUCT() **group** $product/LIST_PRICE as $price_group **by 1** return min($price_group)	SELECT MIN(t1."LIST_PRICE") AS c1 FROM "PRODUCT" t1 **GROUP BY 1**	SELECT MIN(t2.c2) AS c3 FROM (SELECT **1 AS c1,** t1."LIST_PRICE" AS c2 FROM "PRODUCT" t1) t2 **GROUP BY t2.c1**

1. RDBMS supports GROUP BY constant
2. RDBMS does not support GROUP BY, but does support sub-queries in the FROM clause

Group-By with a Nested Where Clause Translates to SQL HAVING Clause

If a relational data source supports nested WHERE clauses, the XQuery engine can translate a where clause after a group-by clause into a SQL HAVING clause provided that the where clause meets other requirements for XQuery-SQL translation.

Table 3-24 Nested WHERE Clauses

XQuery Statement	SQL Statement

for $product in PRODUCT()
group $product/LIST_PRICE as $price_group
by $product/CATEGORY as $category
where max($price_group) gt 1000
return
<t>
{
$category,
min($price_group)
}
</t>
	SELECT t1."CATEGORY" AS c1,
	MIN(t1."LIST_PRICE") AS c2
	FROM "PRODUCT" t1
	GROUP BY t1."CATEGORY"
	HAVING MAX(t1."LIST_PRICE") > 1000

Outer Join with Aggregate Pattern

Another common pattern supported by the DSP XQuery engine is outer join with aggregation of the right branch, which is expressed in XQuery as nested FLWR expressions with aggregate functions in the inner level (Table 3-25).

Table 3-25 Outer Join with Aggregate

XQuery Statement	SQL Statement

for $customer in CUSTOMER()
return
<customer>
<name>{ data($customer/LAST_NAME) }</name>
<order-amount>
{
sum(
for $c_order in CUST_ORDER()
where $customer/CUSTOMER_ID eq $c_order/CUSTOMER_ID
return $c_order/ORDER_AMOUNT
)
}
</order-amount>
</customer>
	SELECT t1."LAST_NAME" AS c1,
	SUM(t2."ORDER_AMOUNT") AS c2
	FROM "CUSTOMER" t1
	LEFT OUTER JOIN "CUST_ORDER" t2
	ON (t2."CUSTOMER_ID" = t1."CUSTOMER_ID")
	GROUP BY t1."CUSTOMER_ID"

With this type of query, in order to fully push as much of the query as possible to the data source RDBMS, the XQuery engine evaluates the outer join first and then performs the group-by on the left branch's primary key column, to compute the aggregate. The XQuery engine can perform this optimization only if the left branch of the query has a key column. As shown in Table 3-25, the CUSTOMER does, so the optimization will be performed.

The net effect is that only the XML creation is performed in the XQuery engine.

If-Then-Else Pattern

The CASE expression, introduced in SQL:1992, provides a way to use if-then-else logic in SQL statements without having to invoke procedures. The CASE expression correlates a list of values and alternatives.

An XQuery if-then-else pattern can be translated into an SQL CASE expression if:

- the underlying data source (RDBMS) supports CASE expressions.

- the XQuery data type result is not an xs:boolean.
- the data types associated with the then and else expresssions are the same (quantifiers are disregarded).

The then and else expressions can contain (or fully consist of) parameters. If the if-then-else expression does not depend on the data source, the entire expression is pushed as a parameter.

Table 3-26 If-Then-Else Pushdown

XQuery Statement	SQL Statement
for $i in CUST_ORDER() return if ($i/STATUS eq "SHIPPED") then data($i/STATUS) else data($i/CUSTOMER_ID)	SELECT CASE WHEN (t1."STATUS" = 'SHIPPED') THEN t1."STATUS" ELSE t1."CUSTOMER_ID" END AS c1 FROM "CUST_ORDER" t1

Subsequence Pushdown

In the typical RDBMS application, it is quite common to paginate the results—output just 20 customer records per page, for example, for printing or other purposes. XQuery meets this need with its subsequence() function. XQuery provides two different subsequence functions, shown in Table 3-27.

Table 3-27 Two- and three-argument Variants of XQuery Subsequence Function

Two-argument variant	Three-argument variant
fn:subsequence($sourceSeq as item()*, $startingLoc as xs:double) as item()*	fn:subsequence($sourceSeq as item()*, $startingLoc as xs:double, $length as xs:double) as item()*

The two-argument variant returns the remaining items of an input sequence, starting from the $startingLoc. The three-argument variant returns $length items of the input sequence starting from the $startingLoc. Table 3-28 shows several different examples of the subsequence function in the context of specific queries.

Table 3-28 Examples of XQuery Expressions using Subsequence Function

Query statement	XQuery Expression
Return the 10 most expensive products only.	let $s := for $i in PRODUCT() order by $i/LIST_PRICE descending return $i for $p in subsequence($s, 1, 10) return <product> <name> { data($p/PRODUCT_NAME) } </name> <price> { data($p/LIST_PRICE) } </price> </product>

Return all service cases opened against each of the 10 most expensive products (outer join).	let $s :=
	  for $i in PRODUCT()
	  order by $i/LIST_PRICE descending
	  return $i
	for $p in subsequence($s, 1, 10)
	return <product>
	<name> { data($p/PRODUCT_NAME) } </name>
	{
	  for $sc in SERVICE_CASE()
	  where $p/PRODUCT_ID eq $sc/PRODUCT_ID and
	  $sc/STATUS = 'Open'
	  return <case>{ data($sc/CASE_ID) }</case>
	}
	</product>

Return the total number of service cases opened against each of the 10 most expensive products (aggregation).	let $s :=
	  for $i in PRODUCT()
	  order by $i/LIST_PRICE descending
	  return $i
	for $p in subsequence($s, 1, 10)
	return
	<product>
	<name> { data($p/PRODUCT_NAME) } </name>
	{
	  let $scs :=
	  for $sc in SERVICE_CASE()
	  where $p/PRODUCT_ID eq $sc/PRODUCT_ID and $sc/STATUS = 'Open'
	  return $sc
	  return <case_count>{ count($scs) }</case_count>
	}
	</product>

An XQuery subsequence pattern can be translated into an SQL subsequence expression if:

- the fn:subsequence() operates on a FLWR expression that returns items from the RDBMS
- the return expression in the inner FLWR must always return a single item (it can be a row element or column element)
- the underlying data source (RDBMS) supports subsequence

DSP can pushdown the subsequence pattern to the underlying RDBMS, thereby enhancing performance, as long as the underlying RDBMS supports it.

- IBM DB2/8 supports both variants of the subsequence function. However, if the $startingLoc or $length are typed as xs:double, pushdown does not occur.
- Oracle 8i, Oracle 9i, and Oracle Database 10g support both versions of the subsequence function, without restriction.
- Microsoft SQL Server 2000 supports the three-argument version only, and requires that $startingLoc must be 1 (a constant) and $length must be an xs:integer constant.

Subsequence pushdown is not supported for Pointbase, Sybase, or any base RDBMS (see "XQuery-SQL Mapping Reference" on page B-1 for other core and base RDBMS information.)

Table 3-29 Subsequence Pushdown

XQuery Statement	SQL Statement (Oracle)
let $s := for $i in t2:PRODUCT() order by $i/LIST_PRICE descending return $i for $p in subsequence($s, 1, 10) return <product> <name> { data($p/PRODUCT_NAME) } </name> <price> { data($p/LIST_PRICE) } </price> </product> };	SELECT t3.c1, t3.c2 FROM(SELECT ROWNUM as c3, t2.c1, t2.c2 FROM(SELECT t1."LIST_PRICE" as c1, t1."PRODUCT_NAME" as c2 FROM "RTLALL"."PRODUCT" t1 ORDER BY t1."LIST_PRICE" DESC)t2)t3 WHERE(t3.c3 <11)

Direct SQL Data Services and Pushdown

Data Services Platform lets you create data services not only from relational tables and views, but also from SQL queries. These direct SQL data services, as they are called, can also be composed by the DSP XQuery engine, and pushed down as native SQL to the target RDBMS, if:

- the RDBMS supports sub-queries in the FROM clause.
- for outer join pushdown, key information must be specified in the Direct SQL data service configuration (see "XQuery-SQL Mapping Reference" on page B-1).

If the RDBMS does not support sub-queries (the FROM clause), the pushdown will not occur.

For example, a user-defined SQL query, “recent_order” is configured as a relational source:

```
SELECT * from RECENT_ORDER
```

The XQuery that gets created in the data service and the resulting generated SQL that gets pushed down by the XQuery engine are shown in Table 3-30.

Table 3-30 Direct SQL Data Service Example

XQuery Statement	SQL Statement
declare variable $external_variable as xs:string external; for $recent_order in **RECENT_ORDER**() where $recent_order/ORDER_ID eq $external_variable return $recent_order/ORDER_AMOUNT	SELECT t1."ORDER_AMOUNT" AS c1 FROM (**SELECT * FROM RECENT_ORDER**) t1 WHERE t1."ORDER_ID" = ?

SQL pushdown on top of direct SQL is not limited to simple select-project queries. Any operation for which pushdown is supported for table and view sources is also supported for data services created for direct SQL queries. For example, Table 3-31 shows a join query and its generated result.

Table 3-31 Direct SQL Data Service with Join Condition

XQuery Statement	SQL Statement
for $customer in CUSTOMER() for $recent_order in **RECENT_ORDER()** where $customer/CUSTOMER_ID eq $recent_order/CUSTOMER_ID return <t>{ $customer/CUSTOMER_ID, $recent_order/ORDER_ID }</t>	SELECT t1."CUSTOMER_ID" AS c1, t2."ORDER_ID" AS c2 FROM "CUSTOMER" t1 JOIN (**SELECT * FROM RECENT_ORDER**) t2 ON t1."CUSTOMER_ID" = t2."CUSTOMER_ID"

Distributed Query Pushdown

Data Services Platform uses SQL pushdown to off-load query processing to the underlying data source RDBMS whenever possible. However, as mentioned in "How it Works—XQuery Engine's Support for SQL" on page 3-3, SQL pushdown is not always possible, nor beneficial. For example, when two data sources are running on two different systems, or when a query combines relational data with non-relational data, SQL pushdown may not provide any performance benefit.

In cases such as these, DSP uses special techniques to batch-process the outside portion of a query (the left branch) and send a cluster (or chunk) of data to the right branch as parameters (see Table 3-32). The XQuery engine chooses this optimization technique (a "clustered parameter passing join," also known as PPK) for a distributed query when:

- join pattern is recognized by the compiler, and
- the join cannot be pushed down in its entirety for any reason, and
- join condition is pushable to either branch when all expressions operating on another branch are treated as parameters in the generated SQL.

Table 3-32 Distributed Query Pushdown—PPK Join Example

XQuery Statement	SQL Statement
for $customer in CUSTOMER() for $order in ORDER() where $customer/CUSTOMER_ID eq $recent_order/CUSTOMER_ID return <t>{ $customer/CUSTOMER_ID, $order/ORDER_ID }</t>	SELECT t1."CUSTOMER_ID" AS c1, t1."ORDER_ID" as c2 from "ORDER" t1 WHERE t1."CUSTOMER_ID" = ? OR t1."CUSTOMER_ID" = ? ... OR t1."CUSTOMER_ID" =?

Unless all these conditions are met, the XQuery engine cannot use this optimization technique but will instead use the single parameter join instead (PP1 join).

Preventing SQL Pushdown

Developers can exercise control over SQL pushdown by using the fn-bea:fence() function (a BEA extension to XQuery functions and operations) to demarcate sections of XQuery code that the XQuery engine should ignore when it is evaluating query fragments for SQL pushdown.

For the example shown in Table 3-33, even though the upper-case function could be pushed down to the RDBMS, its pushdown is blocked by the fence() function and the upper-case function will be executed by the XQuery engine. Only the fragment comprising the lower-case function is included in the query plan as SQL pushdown. The result of the SQL will be returned to the XQuery engine, which will use the XQuery upper-case function on the result.

Use the fence() function whenever you want SQL to be sent as is, to the RDBMS. For example, if you are accessing an Oracle 8.5.x RDBMS that uses hints and Oracle's rule-based optimizer, you should send the hinted SQL queries to the data source by wrapping them in the fence() function.

Table 3-33 Using the fn-bea:fence() Function

XQuery Statement	SQL Statement
for $c in CUSTOMER() return upper-case(**fn-bea:fence**(lower-case($c/LAST_NAME)))	SELECT LOWER(t1."LAST_NAME") AS c1 FROM "CUSTOMER" t1

To circumvent SQL pushdown for specific clauses, extract those clauses into separate FLWOR expressions with the fence() function at the top of the clause, as shown here:

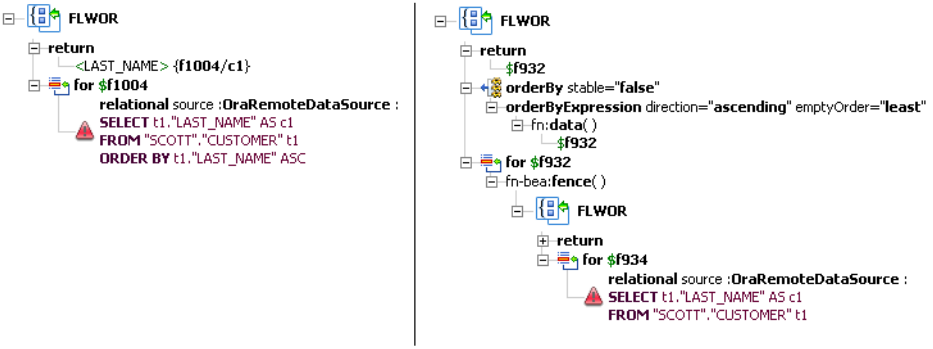
```
for $x in
    fn-bea:fence
    (
       for $c in CUSTOMER()
       return $c/LAST_NAME
    )
```

```
order by $x

return $x
```

As you develop data services that use relational data sources, use the Query Plan View of WebLogic Workshop to see the results of using the fence() function (Figure 3-34). In this example, the order by clause will be executed by the XQuery engine rather than pushed down as SQL.

Figure 3-34 Example of an XQuery Plan without (l) and with (r) the fn-bea:fence() Function



Note that the red triangles displayed in the SQL portions of Figure 3-34 are alerts calling attention to the fact that a where clause is missing from the XQuery statement.

CHAPTER 4

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. BEA AquaLogic Data Services Platform (DSP) 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 Data Services Platform 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 4-1 lists the predefined XQuery namespaces used in Data Services Platform queries.

Table 4-1 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 DSP-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"/>
```

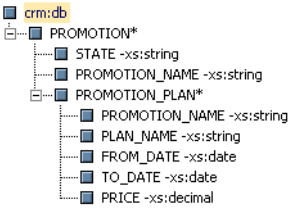
It is recommended that you create schemas with `elementFormDefault="unqualified"` and `attributeFormDefault="unqualified"`. This enables you to rename a namespace by renaming a single complex element, instead of having to explicitly map every element.

Using XML Namespaces in Data Services Platform Queries and Schemas

Data Services Platform (DSP) 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 4-2 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 5

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

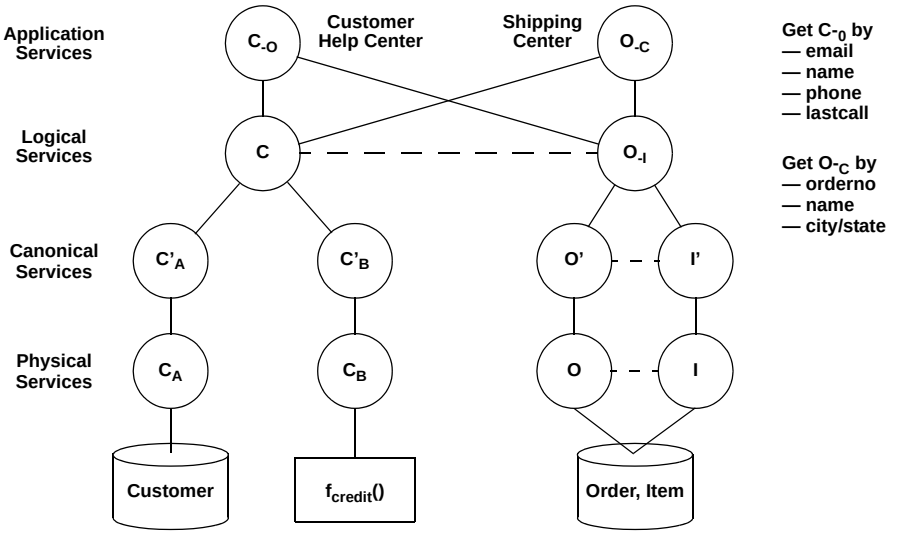
When designing data services, you should strive to maximize the ability to maintain, manage, and reuse queries. One approach is to adopt a layered design model that partitions services into the following levels:

- **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 5-1 illustrates the data service design model.

Figure 5-1 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 5-2 describes the data service design principles.

Table 5-2 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	DSP 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 5-2 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 5-3 describes implementation guidelines to apply when designing and developing data services.

Table 5-3 Data Service Implementation Guidelines

Level	Design Principle	Description

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 DSP 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 cast operations to be executed by 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 5-3 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 DSP 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 5-3 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 6

Understanding Data Services Platform Annotations

This chapter describes the syntax and semantics of BEA AquaLogic Data Services Platform (DSP) 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, "Annotations Reference," 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 named "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 6-1.

Table 6-1 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 several attributes. All attributes are required unless otherwise specified in Table 6-2.

Table 6-2 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. Required.
`identifierQuote`	Specifies the string used by the RDBMS to quote SQL identifiers. Required.
`catalogQuote`	Specifies the string used by the RDBMS to quote database catalog identifiers. Optional.
`schemaQuote`	Specifies the string used by the RDBMS to quote database schema identifiers. Optional.
`tableQuote`	Specifies the string used by the RDBMS to quote table identifiers. Optional.
`columnQuote`	Specifies the string used by the RDBMS to quote column identifiers. Optional.

Table 6-2 Attributes for the properties Element (Continued)

Attribute	Description
`nullSortOrder`	A string specifying how null values are sorted by the RDBMS, from among the following values: high, low, or unknown. Required.
`supportsCatalogsInDataManipulation`	A Boolean specifying whether the RDBMS supports catalog names in Data Manipulation Language (DML) SQL statements. Required.
`supportsLikeEscapeClause`	A Boolean specifying whether the RDBMS supports LIKE escape clauses. Required.
`supportsSchemasInDataManipulation`	A Boolean specifying whether the RDBMS supports schema names in DML SQL statements. Required.

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 6-3.

Table 6-3 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 6-4.

Table 6-4 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 6-4 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 6-5.

Table 6-5 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 named "field" with two required attributes, described in Table 6-6.

Table 6-6 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 6-7.

Table 6-7 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 6-7 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 6-8.

Table 6-8 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 6-11).

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 6-25.

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 6-9.

Table 6-9 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 Data Services Platform 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 6-10.

Table 6-10 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 6-11.

Table 6-11 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 DSP Console. Query content can inquire about a user's ability to access a logical resource using the built-in function isAccessAllowed().

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
- Procedure (side-effecting 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, private, or hasSideEffects 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.

A private function may be used only be the data service in which it has been defined.

A function designated as a procedure has in the general case side-effects. In other words, the invocation of the function entails modifications of the state of the affected data sources. Therefore, a procedure may only be directly invoked by Data Services Platform mediator clients. In particular, procedures may not be referenced by other DSP functions or ad hoc queries.

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 6-12.

Table 6-12 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
- Behavioral 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="read" nativeName="CUSTOMER" nativeLevel2Container="RTL"
style="table">
<nonCacheable/>
</f:function>::)
```

General Properties

All standard document properties and user-defined properties defined in "Standard Document Properties" on page 6-4 and "User-Defined Properties" on page 6-4 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 6-13

Table 6-13 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 6-13 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/>
```

Behavioral Properties

The optional block of behavioral properties allows you to provide information related to known associations between a function's input and its output, or across two or more functions. In particular, the user may specify the following:

- Inverse Functions
- Equivalent Transforms

Inverse Functions

Given an XQuery function *f*, the optional block of inverse functions may be used in order to denote a function *g*, defined over the range of *f*, that, when composed with *f* (i.e. *g*(*f*)), renders one of the parameters of *f*. If *f* has multiple parameters, an inverse function may be defined for each one of its parameters.

The inverse functions block is represented by an optional element, named inverseFunctions, which accepts as its content a sequence of empty elements, named inverseFunction. Each inverseFunction element accepts the following attributes:

- "**parameterIndex.** Optional attribute denoting the index of the parameter for which the inverse function is defined. The index of the first parameter is assumed to be 1. It may be omitted if the function being annotated has a single parameter.
- "**name.** Required attribute denoting the fully-qualified name of the inverse function.

Note: Both the annotated and the inverse function must be either built-in or external XQuery functions.

The following excerpt provides an example of an inverseFunctions element definition:

```
<inverseFunctions>
   <inverseFunction index="2" name="p:MyInverse" xmlns:p="urn:test"/>
</inverseFunctions>
```

Equivalent Transforms

Given an XQuery function *f*, the optional block of equivalent transforms may be used in order to denote a pair of functions *C* and *C'* with identical signatures and equivalent semantics, that accept *f* as one of their parameters. In simple terms, the equivalence is perceived to mean that each occurrence of *C(…,f,…)* may be safely substituted with *C'(…,f,…)*.

The equivalent transforms block is represented by an optional element, named equivalentTransforms, which accepts as its content a sequence of empty elements, named pair. Each pair element accepts the following required attributes:

- "**source.** Denotes the fully qualified name of the source transform (i.e.: *C*).
- "**target.** Denotes the fully qualified name of the target transform (i.e.: *C'*).
- "**arity.** Denotes the (common) arity of the source and target transforms.

Note: The source transform may be either a built-in or external function. Both source and target transforms must not be defined as invertible functions.

The following excerpt provides an example of an equivalentTransforms element definition:

```
<equivalentTransforms>
  <pair source="p:sourceFunction_1" target="p:targetFunction_1"
    arity="1" xmlns:p="urn:test1"/>
  <pair source="q:sourceFunction_2" target="q:targetFunction_2"
    arity="3" xmlns:q="urn:test2"/>
</equivalentTransforms>
```

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 6-14.

Table 6-14 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 6-15.

Table 6-15 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.

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 6-4.

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.

APPENDIX A

Annotations Reference

XML Schema for Annotations

This appendix contains the entire XML Schema definition file (XSD) that BEA AquaLogic Data Services Platform (DSP) uses for annotations. This file constitutes the complete grammer of the pragma annotations contained in data service source files.

For information about the syntax and semantics of Data Services Platform annotations in data service and XQuery function library (XFL) documents, see Chapter 6, "Understanding Data Services Platform 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>
        <!-- data access properties -->
        <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="catalogQuote" type="xs:string"/>
                    <xs:attribute name="schemaQuote" type="xs:string"/>
                    <xs:attribute name="tableQuote" type="xs:string"/>
                    <xs:attribute name="columnQuote" type="xs:string"/>
                <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>
        <!-- data access properties -->
        <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="catalogQuote" type="xs:string"/>
                    <xs:attribute name="schemaQuote" type="xs:string"/>
                    <xs:attribute name="tableQuote" type="xs:string"/>
              <xs:attribute name="columnQuote" type="xs:string"/>
                <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>
          <!-- optimization -->
          <xs:element name="outputIsOrderedBy" minOccurs="0">
            <xs:complexType>
              <xs:sequence>
                <!-- absent for parameters whose order in the function signature
                     coincides with their order in the order by list -->
                <xs:element name="parameter" minOccurs="0" maxOccurs="unbounded">
                  <xs:complexType>
                    <!-- 1, 2, ... -->
```

```
                <xs:attribute name="index" type="xs:int" use="required"/>
                <!-- overrides default -->
                <xs:attribute name="mode" type="tns:orderingModeType"/>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        <xs:attribute name="mode" type="tns:orderingModeType" use="required"/>
         </xs:complexType>
       </xs:element>

       <xs:element name="inverseFunctions" minOccurs="0">
         <xs:complexType>
           <xs:sequence>
             <xs:element name="inverseFunction" minOccurs="1"
maxOccurs="unbounded">
               <xs:complexType>
                 <!-- 1, 2, ... -->
                 <xs:attribute name="parameterIndex" type="xs:int"/>
                 <xs:attribute name="name" type="xs:QName" use="required"/>
               </xs:complexType>
             </xs:element>
           </xs:sequence>
         </xs:complexType>
       </xs:element>
       <xs:element name="equivalentTransforms" minOccurs="0">
         <xs:complexType>
           <xs:sequence>
             <xs:element name="pair" minOccurs="1" maxOccurs="unbounded">
               <xs:complexType>
                 <xs:attribute name="source" type="xs:QName" use="required"/>
                 <xs:attribute name="target" type="xs:QName" use="required"/>
                 <xs:attribute name="arity"  type="xs:int"   use="required"/>
               </xs:complexType>
             </xs:element>
           </xs:sequence>
         </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:enumeration value="hasSideEffects"/>
  </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:simpleType name="orderingModeType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="ascending"/>
```

```
      <xs:enumeration value="descending"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

APPENDIX B

XQuery-SQL Mapping Reference

This appendix provides the details of BEA AquaLogic Data Services Platform (DSP) core support and base support for relational data, and includes these topics:

- Core RDBMS Support:
  - IBM DB2/NT 8
  - Microsoft SQL Server 2000
  - Oracle 8.1.x
  - Oracle 9.x, 10.x
  - Pointbase 4.4 (and higher)
  - Sybase 12.5.2 (and higher)
- Base (Generic) RDBMS Support

Each section that follows includes information about:

- Database Capabilities Information
- Native RDBMS Data Type Support and XQuery Mappings
- Function and Operator Pushdown
- Cast Operation Pushdown
- Other SQL Generation Capabilities (including join pushdown support and SQL syntax for joins)

IBM DB2/NT 8

The tables in this section identify all data type and other mappings that the XQuery engine generates or supports for IBM DB2/NT 8.

Data Type Mapping

The following table lists supported data type mappings.

Table B-1 Data Type Mappings

DB2 Data Type	XQuery Type
BIGINT	xs:long
BLOB	xs:hexBinary
CHAR	xs:string
CHAR() FOR BIT DATA	xs:hexBinary
CLOB[1]	xs:string
DATE	xs:date
DOUBLE	xs:double
DECIMAL(p,s)[2] (NUMERIC)	xs:decimal (if s > 0), xs:integer (if s = 0)
INTEGER	xs:int
LONG VARCHAR[1]	xs:string
LONG VARCHAR FOR BIT DATA	xs:hexBinary
REAL	xs:float
SMALLINT	xs:short
TIME[3]	xs:time[4]
TIMESTAMP[5]	xs:dateTime[4]

VARCHAR	xs:string[4]
VARCHAR() FOR BIT DATA	xs:hexBinary

1. Pushed down in project list only.
2. Where *p* is precision (total number of digits, both to the right and left of decimal point) and *s* is scale (total number of digits to the right of decimal point).
3. Accurate to 1 second.
4. Values converted to local time zone (timezone information removed) due to TIME and TIMESTAMP limitations. See "Date and Time Data Type Differences: Timezones and Time Precision" on page 3-6 for more information.
5. Precision limited to milliseconds.

Function and Operator Pushdown

The following table lists functions and operators that are pushed down to IBM DB2/NT8 RDBMSs. See "fn-bea:sql-like" on page 2-22 for details about two-argument and three-argument versions of the fn-bea:sql-like() function.

Table B-2 Functions and Operators

Group	Functions and operators
Logical operators	and, or, not
Numeric arithmetic	+, -, *, div, idiv[1]
	mod[2]
Numeric comparisons[1]	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge
Numeric functions	abs, ceiling, floor, round
String comparisons[3]	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge
String functions	concat, upper-case, lower-case, substring(2,3)[4], string-length, contains[5], starts-with[5], ends-with[5], fn-bea:sql-like(2,3) fn-bea:trim[6], fn-bea:trim-left[6], fn-bea:trim-right[6]
Datetime comparisons	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge on xs:dateTime, xs:date, xs:time

Datetime functions	year-from-dateTime, year-from-date, month-from-dateTime, month-from-date, day-from-dateTime, day-from-date, hours-from-dateTime, hours-from-time, minutes-from-dateTime, minutes-from-time, seconds-from-dateTime, seconds-from-time, fn-bea:date-from-dateTime, fn-bea:time-from-dateTime
Aggregate	min, max, sum, avg, count, count(distinct-values)
Other	empty, exists, subsequence[7]

1. All numeric types.
2. xs:integer (and subtypes) only.
3. Arguments must have SQL data type CHAR or VARCHAR.
4. If second and third arguments are types xs:double or xs:float, they cannot be parameters.
5. Second argument must be a constant or a parameter.
6. Argument must be SQL data type CHAR or VARCHAR.
7. Both two- and three-argument variants supported, with the restriction that no pushdown occurs when $startingLoc or $length are typed as xs:double.

Cast Operation Pushdown

The following table lists supported cast operations.

Table B-3 Cast Operations

Source XQuery Type	Target XQuery Type
numeric	xs:double
numeric	xs:float
numeric	xs:int
numeric	xs:integer
numeric	xs:short
xs:decimal (and subtypes)	xs:string
xs:integer (and subtypes)	xs:decimal
xs:string	xs:double

xs:string	xs:float
xs:string	xs:int
xs:string	xs:integer
xs:string	xs:short
xs:dateTime	xs:time

Other SQL Generation Capabilities

The following table lists common query patterns that can be pushed down. See "Common Query Patterns" for details.

Table B-4 Other SQL Generation Capabilities

Feature	Description
If-then-else	yes
Inner joins	yes, SQL-92 syntax
Outer joins	yes, SQL-92 syntax
Semi joins, Anti semi joins	yes
Order by	yes
Order by: Empty (NULL) order supported	Fixed (always sorts NULLs high). Order-bys with "empty least" modifier (the XQuery default) are not pushed down.
Order by: Aggregate function in ordering expression	yes
Group by	yes
Distinct pattern	yes
Trivial aggregate pattern	yes (using GROUP BY constant)
Direct SQL composition	yes

Microsoft SQL Server 2000

The tables in this section identify all data type and other mappings that the XQuery engine generates or supports for Microsoft SQL Server 2000.

Data Type Mapping

The following table lists supported data type mappings for Microsoft SQL Server 2000.

Table B-5 Data Type Mapping

SQL Data Type	XQuery Type
BIGINT	xs:long
BINARY	xs:hexBinary
BIT	xs:boolean
CHAR	xs:string
DATETIME[1]	xs:dateTime[2]
DECIMAL(p,s)[3] (NUMERIC)	xs:decimal (if s > 0), xs:integer (if s = 0)
FLOAT	xs:double
IMAGE	xs:hexBinary
INTEGER	xs:int
MONEY	xs:decimal
NCHAR	xs:string
NTEXT[4]	xs:string
NVARCHAR	xs:string
REAL	xs:float
SMALLDATETIME[5]	xs:dateTime
SMALLINT	xs:short
SMALLMONEY	xs:decimal

Table B-5 Data Type Mapping

SQL_VARIANT	xs:string
TEXT[4]	xs:string
TIMESTAMP	xs:hexBinary
TINYINT	xs:short
VARBINARY	xs:hexBinary
VARCHAR	xs:string
UNIQUIDENTIFIER	xs:string

1. Fractional-second-precision up to 3 digits (milliseconds). No timezone.
2. Values converted to local time zone (timezone information removed) and fractional seconds truncated to milliseconds due to DATETIME limitations. See "Date and Time Data Type Differences: Timezones and Time Precision" on page 3-6 for more information.
3. Where *p* is precision (total number of digits, both to the right and left of decimal point) and *s* is scale (total number of digits to the right of decimal point).
4. Pushed down in project list only.
5. Accuracy of 1 minute.

Additionally, the following XQuery data types can be passed as parameters or returned by pushed functions:

- xs:date (see Table B-6 for functions and operators that use xs:date). When xs:date is sent to the database, it is converted to local time zone. See "Date and Time Data Type Differences: Timezones and Time Precision" on page 3-6 for more information.
- xdt:dayTimeDuration (see "Datetime Arithmetic" functions in Table B-6 for details).
- xdt:yearMonthDuration (see "Datetime Arithmetic" functions in Table B-6 for details).

Function and Operator Pushdown

The following table lists functions and operators that are pushed down to Microsoft SQL Server 2000. See "fn-bea:sql-like" on page 2-22 for details about two-argument and three-argument versions of the fn-bea:sql-like() function.

Table B-6 Function and Operator Pushdown

Group	Functions and Operators
Logical operators	and, or, not
Numeric arithmetic	+, -, *, div, idiv[1]
	mod[2]
Numeric comparisons[1]	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge
Numeric functions	abs, ceiling, floor, round
String comparisons[3]	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge
String functions	concat, upper-case, lower-case, substring(2,3)[4], string-length, contains[5], starts-with[5], ends-with[5], fn-bea:sql-like(2,3)[4], fn-bea:trim, fn-bea:trim-left, fn-bea:trim-right
Datetime comparisons	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge on xs:dateTime, xs:date, xdt:yearMonthDuration, xdt:dayTimeDuration
Datetime functions	year-from-dateTime, year-from-date, years-from-duration, month-from-dateTime, month-from-date, months-from-duration, day-from-dateTime, day-from-date, days-from-duration, hours-from-dateTime, hours-from-duration, minutes-from-dateTime, minutes-from-duration, seconds-from-dateTime, seconds-from-duration, fn-bea:date-from-dateTime

Datetime arithmetic	op:add-yearMonthDurations, op:add-dayTimeDurations, op:subtract-yearMonthDurations, op:subtract-dayTimeDurations, op:multiply-yearMonthDuration, op:multiply-dayTimeDuration, op:divide-yearMonthDuration, op:divide-dayTimeDuration, subtract-dateTimes-yielding-yearMonthDuration, subtract-dateTimes-yielding-dayTimeDuration, op:add-yearMonthDuration-to-dateTime, op:add-dayTimeDuration-to-dateTime, op:subtract-yearMonthDuration-from-dateTime, op:subtract-dayTimeDuration-from-dateTime, subtract-dates-yielding-yearMonthDuration, subtract-dates-yielding-dayTimeDuration, op:add-yearMonthDuration-to-date, op:add-dayTimeDuration-to-date, op:subtract-yearMonthDuration-from-date, op:subtract-dayTimeDuration-from-date
Aggregate	min, max, sum, avg, count, count(distinct-values)
Other	empty, exists, subsequence[6]

1. For all numeric types
2. For xs:integer and its subtypes only.
3. Arguments must be of SQL data type CHAR, NCHAR, VARCHAR, or NVARCHAR.
4. Both the 2-argument and 3-argument versions of function supported.
5. Second argument must be SQL data type CHAR, NCHAR, VARCHAR, or NVARCHAR.
6. Only the three-argument variant of fn:subsequence is supported, with the additionl requirement that the $startingLoc must be 1 (constant) and $length must be xs:integer type.

Cast Operation Pushdown

The following table lists supported cast operations.

Table B-7 Cast Operations

Source XQuery Data Type	Target XQuery Data Type
numeric	xs:string

numeric	xs:double
numeric	xs:float
numeric	xs:integer
numeric	xs:long
numeric	xs:int
numeric	xs:short
xs:integer (and subtypes)	xs:decimal
xs:string	xs:double[1]
xs:string	xs:float
xs:string	xs:integer
xs:string	xs:long
xs:string	xs:int
xs:string	xs:short
xs:dateTime	xs:date
xs:dateTime	xs:string

1. Source SQL type must be CHAR, NCHAR, VARCHAR, or NVARCHAR.

Other SQL Generation Capabilities

The following table lists common query patterns that can be pushed down. See "Common Query Patterns" for details.

Table B-8 Other SQL Generation Capabilities

Feature	Description
If-then-else	yes
Inner joins	yes, SQL-92 syntax

Outer joins	yes, SQL-92 syntax
Semi joins, Anti semi joins	yes
Order by	yes
Order by: Empty order (NULL order)	fixed (always sorts NULLs low). Order-bys with "empty greatest" modifier are not pushed down.
Order by: Aggregate function in ordering expression	yes
Group by	yes
Distinct pattern	yes
Trivial aggregate pattern	yes (using subquery)
Direct SQL composition	yes

Oracle 8.1.x

The tables in this section identify all data type and other mappings that the XQuery engine generates or supports for Oracle 8.1.x (Oracle 8*i*).

Data Type Mapping

Table B-9 Data Type Mapping

Oracle 8 Data Type	XQuery Type
BFILE	not supported
BLOB	xs:hexBinary
CHAR	xs:string
CLOB[1]	xs:string
DATE[2]	xs:dateTime
FLOAT	xs:double
LONG[1]	xs:string
LONG RAW	xs:hexBinary
NCHAR	xs:string
NCLOB[1]	xs:string
NUMBER	xs:double
NUMBER(p,s)[3]	xs:decimal (if s > 0), xs:integer (if s <=0)
NVARCHAR2	xs:string
RAW	xs:hexBinary
ROWID	xs:string
UROWID	xs:string

1. Pushed down in project list only.
2. Does not support fractional seconds.
3. Where *p* is precision (total number of digits, both to the right and left of decimal point) and *s* is scale (total number of digits to the right of decimal point).

Additionally, the following XQuery data types can be passed as parameters or returned by pushed functions:

- xs:date (see Table B-10 for functions and operators that use xs:date)
- xdt:yearMonthDuration (see "Datetime Arithmetic" in Table B-10 for details)
- xs:integer subtypes (see "Numeric ..." functions and operators in Table B-10 for details)

Function and Operator Pushdown

The following table lists functions and operators that are pushed down. See "fn-bea:sql-like" on page 2-22 for details about two-argument and three-argument versions of the fn-bea:sql-like() function.

Table B-10 Function and Operator Pushdown

Group	Functions and operators
Logical operators	and, or, not
Numeric arithmetic[1]	+, -, *, div, idiv, mod
Numeric comparisons[1]	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge
Numeric functions	abs, ceiling, floor, round
String comparisons[2]	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge
String functions	concat, upper-case[3], lower-case[3], substring(2,3)[3], string-length[4], contains[5], starts-with[5], ends-with[5], fn-bea:sql-like(2,3), fn-bea:trim, fn-bea:trim-left, fn-bea:trim-right
Datetime comparisons	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge on xs:dateTime, xs:date, xdt:yearMonthDuration
Datetime functions	year-from-dateTime, year-from-date, years-from-duration, month-from-dateTime, month-from-date, months-from-duration, day-from-dateTime, day-from-date, days-from-duration, hours-from-dateTime, minutes-from-dateTime, seconds-from-dateTime, fn-bea:date-from-dateTime

Table B-10 Function and Operator Pushdown

Datetime arithmetic	op:add-yearMonthDurations, op:subtract-yearMonthDurations, op:multiply-yearMonthDuration, op:divide-yearMonthDuration, subtract-dateTimes-yielding-yearMonthDuration, op:add-yearMonthDuration-to-dateTime, op:subtract-yearMonthDuration-from-dateTime, subtract-dates-yielding-yearMonthDuration, op:add-yearMonthDuration-to-date, op:subtract-yearMonthDuration-from-date
Aggregate	min, max, sum, avg, count, count(distinct-values)
Other	empty, exists, subsequence[6]

1. For all numeric types.
2. Arguments must be of SQL data type CHAR, NCHAR, NVARCHAR2, or VARCHAR2.
3. Empty input (NULL) handling deviates from XQuery semantics—returns empty sequence (instead of empty string).
4. Argument must be data type CHAR, NCHAR, NVARCHAR2, or VARCHAR2.
5. Second argument must be data type CHAR, NCHAR, NVARCHAR2, or VARCHAR2.
6. Both two- and three-argument variants of fn:subsequence() are supported without restriction.

Cast Operation Pushdown

The followingtable lists supported cast operations.

Table B-11 Cast Operation Pushdown

Source XQuery Type	Target XQuery Type
numeric	xs:string
numeric	xs:decimal
numeric	xs:integer
numeric	xs:float
numeric	xs:double

xs:string	xs:decimal[1]
xs:string	xs:integer[1]
xs:string	xs:float[1]
xs:string	xs:double[1]
xs:dateTime	xs:date
xs:date	xs:dateTime

1. Source data type must be CHAR, NCHAR, NVARCHAR2, or VARCHAR2.

Other SQL Generation Capabilities

The following table lists common query patterns that can be pushed down. See "Common Query Patterns" for details.

Table B-12 Other SQL Generation Capabilities

Feature	Description
If-then-else	yes
Inner joins	yes, SQL-89 syntax
Outer joins	yes, Oracle proprietary syntax
Semi joins, Anti semi joins	yes
Order by	yes
Order by: Empty order (NULL order)	dynamic, no restriction on order by pushdown
Order by: Aggregate function in ordering expression	yes
Group by	yes
Distinct pattern	yes

Trivial aggregate pattern	yes (using GROUP BY constant)
Direct SQL composition	yes

Oracle 9.x, 10.x

The tables in this section identify all data type and other mappings that the XQuery engine generates or supports for Oracle 9.x (Oracle 9*i*) and Oracle 10.x (Oracle 10*g*). Note that Oracle treats empty strings as NULLs, which deviates from XQuery semantics and may lead to unexpected results for expressions that are pushed down.

Data Type Mapping

Table B-13 Data Type Mapping

Oracle 9 Data Type	XQuery Type
BFILE	not supported
BLOB	xs:hexBinary
CHAR	xs:string
CLOB[1]	xs:string
DATE	xs:dateTime[2]
FLOAT	xs:double
INTERVAL DAY TO SECOND	xdt:dayTimeDuration
INTERVAL YEAR TO MONTH	xdt:yearMonthDuration
LONG[1]	xs:string
LONG RAW	xs:hexBinary
NCHAR	xs:string
NCLOB[1]	xs:string
NUMBER	xs:double
NUMBER(p,s)	xs:decimal (if s > 0), xs:integer (if s <=0)
NVARCHAR2	xs:string
RAW	xs:hexBinary
ROWID	xs:string
TIMESTAMP	xs:dateTime[3]
TIMESTAMP WITH LOCAL TIMEZONE	xs:dateTime
TIMESTAMP WITH TIMEZONE	xs:dateTime

Table B-13 Data Type Mapping

VARCHAR2	xs:string
UROWID	xs:string

1. Pushed down in project list only.
2. When SDO stores xs:dateTime value in Oracle DATE type, it is converted to local time zone and fractional seconds are truncated due to DATE limitations. See "Date and Time Data Type Differences: Timezones and Time Precision" on page 3-6 for more information.
3. XQuery engine maps XQuery xs:dateTime to either TIMESTAMP or TIMESTAMP WITH TIMEZONE data type, depending on presence of timezone information. Storing xs:dateTime using SDO may result in loss of precision for fractional seconds, depending on the SQL type definition.

Additionally, these XQuery data types can be passed as parameters or returned by pushed functions:

- xs:date (see Table B-14 for functions and operators that use xs:date)
- xs:integer subtypes (see "Numeric ..." functions and operators in Table B-14 for details)

Function and Operator Pushdown

The following table lists functions and operators that are pushed down to Oracle 9.x and 10.x. See "fn-bea:sql-like" on page 2-22 for details about two-argument and three-argument versions of the fn-bea:sql-like() function.

Table B-14 Function and Operator Pushdown

Group	Functions and Operators
Logical operators	and, or, not
Numeric arithmetic[1]	+, -, *, div, idiv, mod
Numeric comparisons[1]	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge
Numeric functions	abs, ceiling, floor, round
String comparisons[2]	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge

String functions	concat, upper-case[3], lower-case[3], substring(2,3)[3], string-length[4], contains[5], starts-with[5], ends-with[5], fn-bea:sql-like(2,3), fn-bea:trim, fn-bea:trim-left, fn-bea:trim-right
Datetime comparisons	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge on xs:dateTime, xs:date, xdt:yearMonthDuration, xdt:dayTimeDuration
Datetime functions	year-from-dateTime, year-from-date, years-from-duration, month-from-dateTime, month-from-date, months-from-duration, day-from-dateTime, day-from-date, days-from-duration, hours-from-dateTime, hours-from-duration, minutes-from-dateTime, minutes-from-duration, seconds-from-dateTime, seconds-from-duration, fn-bea:date-from-dateTime
Datetime arithmetic	op:add-yearMonthDurations, op:add-dayTimeDurations, op:subtract-yearMonthDurations, op:subtract-dayTimeDurations, op:multiply-yearMonthDuration, op:multiply-dayTimeDuration, op:divide-yearMonthDuration, op:divide-dayTimeDuration, subtract-dateTimes-yielding-yearMonthDuration, subtract-dateTimes-yielding-dayTimeDuration, op:add-yearMonthDuration-to-dateTime, op:add-dayTimeDuration-to-dateTime, op:subtract-yearMonthDuration-from-dateTime, op:subtract-dayTimeDuration-from-dateTime, subtract-dates-yielding-yearMonthDuration, subtract-dates-yielding-dayTimeDuration, op:add-yearMonthDuration-to-date, op:add-dayTimeDuration-to-date, op:subtract-yearMonthDuration-from-date, op:subtract-dayTimeDuration-from-date
Aggregate	min, max, sum, avg, count, count(distinct-values)
Other	empty, exists, subsequence[6]

1. For all numeric types
2. Arguments must be of SQL type (N)CHAR or (N)VARCHAR2
3. Empty input (NULL) handling deviates from XQuery semantics—returns empty sequence (instead of empty string).
4. Argument must be CHAR, CLOB, NCHAR, NVARCHAR2, or VARCHAR2 data type.

5. Second argument must be CHAR, NCHAR, NVARCHAR2, or VARCHAR2 data type.
6. Both two- and three-argument variants of fn:subsequence() are supported without restriction.

Cast Operation Pushdown

The following table lists cast operations that can be pushed down.

Table B-15 Cast Operation

Source XQuery Type	Target XQuery Type
numeric	xs:string
numeric	xs:decimal
numeric	xs:integer
numeric	xs:float
numeric	xs:double
xs:string	xs:decimal[1]
xs:string	xs:integer
xs:string	xs:float
xs:string	xs:double
xs:dateTime	xs:date
xs:date	xs:dateTime[2]

1. Source SQL type must be CHAR, NCHAR, VARCHAR2, or NVARCHAR2.
2. Source SQL type must be DATE or TIMESTAMP to achieve this mapping.

Other SQL Generation Capabilities

The following table lists common query patterns that can be pushed down. See "Common Query Patterns" for details.

Table B-16 Other SQL Generation Capabilities

Feature	Description
If-then-else	yes
Inner joins	yes, SQL-92 syntax
Outer joins	yes, SQL-92 syntax
Semi joins, Anti semi joins	yes
Order by	yes
Order by: Empty order (NULL order)	dynamic, no restriction on order by pushdown
Order by: Aggregate function in ordering expression	yes
Group by	yes
Distinct pattern	yes
Trivial aggregate pattern pushdown	yes (using GROUP BY constant)
Direct SQL composition	yes

Pointbase 4.4 (and higher)

The tables in this section identify all data type and other mappings that the XQuery engine generates or supports for Pointbase.

Data Type Mapping

Table B-17 Data Type Mapping

Pointbase Data Type	XQuery Type
BIGINT	xs:long
BLOB	xs:hexBinary
BOOLEAN	xs:boolean
CHAR (CHARACTER)	xs:string
CLOB	xs:string
DATE	xs:date
DECIMAL(p,s)[1] (NUMERIC)	xs:decimal (if s > 0), xs:integer (if s == 0)
DOUBLE PRECISION	xs:double
FLOAT	xs:double
INTEGER (INT)	xs:int
SMALLINT	xs:short
REAL	xs:float
TIME	xs:time
TIMESTAMP	xs:dateTime
VARCHAR	xs:string

1. Where *p* is precision (total number of digits, both to the right and left of decimal point) and *s* is scale (total number of digits to the right of decimal point).

Function and Operator Pushdown

The following table lists functions and operators that are pushed down to Pointbase. See "fn-bea:sql-like" on page 2-22 for details about two-argument and three-argument versions of the fn-bea:sql-like() function.

Table B-18 Function and Operator Pushdown

Group	Functions and operators
Logical operators	and, or, not
Numeric arithmetic[1]	+, -, *, div, idiv
Numeric comparisons[1]	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge
String comparisons[2]	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge
String functions	concat,upper-case, lower-case, substring(2,3), string-length, contains[3], starts-with[3], ends-with[3], fn-bea:sql-like(2,3) fn-bea:trim, fn-bea:trim-left, fn-bea:trim-right
Datetime comparisons	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge on xs:dateTime, xs:date, xs:time
Datetime functions	year-from-dateTime, year-from-date, month-from-dateTime, month-from-date, day-from-dateTime, day-from-date, hours-from-dateTime, hours-from-time, minutes-from-dateTime, minutes-from-time, seconds-from-dateTime, seconds-from-time, fn-bea:date-from-dateTime
Aggregate	min, max, sum, avg, count, count(distinct-values)
Other	empty, exists

1. All numeric types
2. CHAR or VARCHAR SQL data types only for arguments
3. Second argument must be constant or parameter.

Cast Operation Pushdown

The following table lists supported cast operations.

Table B-19 Cast Operation Pushdown

Source XQuery Type	Target XQuery Type
numeric	xs:decimal

numeric	xs:double
numeric	xs:float
numeric	xs:int
numeric	xs:short
numeric	xs:string
xs:integer and its subtypes	xs:integer
xs:integer and its subtypes	xs:long
xs:string	xs:decimal[1]
xs:string	xs:double[1]
xs:string	xs:float[1]
xs:string	xs:integer[1]
xs:string	xs:long[1]
xs:string	xs:int[1]
xs:string	xs:short[1]
xs:dateTime	xs:date

1. Source SQL data type must be CHAR or VARCHAR

Other SQL Generation Capabilities

The following table lists common query patterns that can be pushed down. See "Common Query Patterns" for details.

Table B-20 Other SQL Generation Capabilities

Feature	Description
If-then-else	no
Inner joins	yes, SQL-92 syntax

Feature	Description
Outer joins	yes (partially), SQL-92 syntax. Only simple outer joins are pushed, the ones that require subquery don't (e.g. when right branch has a where clause)
Semi joins, Anti semi joins	yes
Order by	yes
Order by: Empty order (NULL order)	fixed (always sorts NULLs low). Order-bys with "empty greatest" modifier are not pushed down.
Order by: Aggregate function in ordering expression	no
Group by	yes (Group by function expression is not supported, only group by column is pushed
Distinct pattern	yes
Trivial aggregate pattern pushdown	no
Direct SQL composition	no

Sybase 12.5.2 (and higher)

The tables in this section identify all data type and other mappings that the XQuery engine generates or supports for Sybase 12.5.2 (and higher).

As you read through the tables in this section, be aware that Sybase deviates from XQuery semantics (which ignores empty strings) and treats empty strings as a single-space string.

Data Type Mapping

This table defines all data type mappings supported.

Table B-21 Data Type Mapping

Sybase Data Type	XQuery Type
BINARY	xs:hexBinary
BIT	xs:boolean

Table B-21 Data Type Mapping

CHAR	xs:string
DATE	xs:date
DATETIME[1]	xs:dateTime[2]
DECIMAL(p,s)[3] (NUMERIC)	xs:decimal (if s > 0), xs:integer (if s == 0)
DOUBLE PRECISION	xs:double
FLOAT	xs:double
IMAGE	xs:hexBinary
INT (INTEGER)	xs:int
MONEY	xs:decimal
NCHAR	xs:string
NVARCHAR	xs:string
REAL	xs:float
SMALLDATETIME[4]	xs:dateTime
SMALLINT	xs:short
SMALLMONEY	xs:decimal
SYSNAME	xs:string
TEXT[5]	xs:string
TIME	xs:time
TINYINT	xs:short
VARBINARY	xs:hexBinary
VARCHAR	xs:string

1. Supports fractional seconds up to 3 digits (milliseconds) precision; no timezone information.
2. Values converted to local time zone (timezone information removed) and fractional seconds truncated to milliseconds due to DATETIME limitations. See "Date and Time Data Type Differences: Timezones and Time Precision" on page 3-6 for more information.

3. Where *p* is precision (total number of digits, both to the right and left of decimal point) and *s* is scale (total number of digits to the right of decimal point).
4. Accurate to 1 minute.
5. Expressions returning text are pushed down in the project list only.

Additionally, the following data types can be passed as parameters or returned by pushed functions:

- xdt:dayTimeDuration
- xdt:yearMonthDuration

See "Datetime arithmetic" in Table for details.

Function and Operator Pushdown

The following table lists functions and operators that are pushed down to base RDBMSs. See "fn-bea:sql-like" on page 2-22 for details about two-argument and three-argument versions of the fn-bea:sql-like() function.

Table B-22 Function and Operator Pushdown

Group	Functions and operators
Logical operators	and, or, not
Numeric arithmetic	+, -, *, div [1]
	idiv[2]
	mod[3]
Numeric comparisons[1]	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge
Numeric functions	abs, ceiling, floor, round
String comparisons[4]	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge
String functions	concat[5], upper-case, lower-case, substring(2,3), string-length, contains[6], starts-with[6], ends-with[6], fn-bea:sql-like(2,3), fn-bea:trim, fn-bea:trim-left, fn-bea:trim-right

Datetime comparisons	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge on xs:dateTime, xs:date, xs:time, xdt:yearMonthDuration, xdt:dayTimeDuration
Datetime functions	year-from-dateTime, year-from-date, years-from-duration, month-from-dateTime, month-from-date, months-from-duration, day-from-dateTime, day-from-date, days-from-duration, hours-from-dateTime, hours-from-time, hours-from-duration, minutes-from-dateTime, minutes-from-time, minutes-from-duration, seconds-from-dateTime, seconds-from-time, seconds-from-duration, fn-bea:date-from-dateTime, fn-bea:time-from-dateTime
Datetime arithmetic	op:add-yearMonthDurations, op:subtract-yearMonthDurations, op:multiply-yearMonthDuration, op:divide-yearMonthDuration, op:add-dayTimeDurations, op:subtract-dayTimeDurations, op:multiply-dayTimeDuration, op:divide-dayTimeDuration, op:add-yearMonthDuration-to-dateTime, op:add-yearMonthDuration-to-date, op:subtract-yearMonthDuration-from-dateTime, op:subtract-yearMonthDuration-from-date, op:add-dayTimeDuration-to-dateTime, op:add-dayTimeDuration-to-date, op:subtract-dayTimeDuration-from-dateTime, op:subtract-dayTimeDuration-from-date, fn:subtract-dateTimes-yielding-yearMonthDuration, fn:subtract-dates-yielding-yearMonthDuration, fn:subtract-dateTimes-yielding-dayTimeDuration, fn:subtract-dates-yielding-dayTimeDuration
Aggregate	min, max, sum, avg, count, count(distinct-values)
Other	empty, exists

1. All numeric types (+, -, *, div operators are pushed down for all numeric types).
2. xs:decimal (and subtypes) only
3. xs:integer (and subtypes) only
4. Arguments must be SQL data type CHAR, NCHAR, NVARCHAR, or VARCHAR.
5. Each argument must be SQL data type CHAR, NCHAR, NVARCHAR, or VARCHAR.
6. Second argument must be constant or SQL parameter.

Cast Operation Pushdown

The following table lists supported cast operations.

Table B-23 Cast Operation Pushdown

Source XQuery Type	Target XQuery Type
numeric	xs:double
numeric	xs:float
numeric	xs:int
numeric	xs:short
numeric	xs:string
xs:decimal (and subtypes)	xs:integer
xs:integer (and subtypes)	xs:decimal
xs:string	xs:double[1]
xs:string	xs:float
xs:string	xs:int
xs:string	xs:integer
xs:string	xs:short
xs:dateTime	xs:date
xs:dateTime	xs:time

1. Source SQL type must be (N)CHAR or (N)VARCHAR

Other SQL Generation Capabilities

The following table lists common query patterns that can be pushed down. See "Common Query Patterns" for details.

Table B-24 Other SQL Generation Capabilities

Feature	Description
If-then-else	yes
Inner joins	yes, SQL-92 syntax
Outer joins	yes, SQL-92 syntax
Semi joins, Anti semi joins	yes
Order by	yes
Order by: Empty order (NULL order)	fixed (always sorts NULLs low). Order-bys with "empty greatest" modifier are not pushed down.
Order by: Aggregate function in ordering expression	yes
Group by	yes
Distinct pattern	yes
Trivial aggregate pattern	yes (using subquery)
Direct SQL composition	yes

Base (Generic) RDBMS Support

Each JDBC drivers provide information about inherent properties and capabilities of the RDBMS with which it is associated. During the metadata import process, DSP queries a configured data source's JDBC driver for basic properties and capabilities information. Much of the information obtained is stored in the metadata section of the data service definition file (.ds). See "Understanding Data Services Platform Annotations" on page 6-1 for more information.

Database Capabilities Information

These database capabilities are obtained from the JDBC driver and stored as properties in the .ds (data service) definition file.

Table B-25 Database Properties for Capabilities

Property	Description	Possible Values
supportsSchemasInDataManipulation	Boolean that identifies whether SQL statements can include schema names	true, false
supportsCatalogsInDataManipulation	Boolean that identifies whether database catalogs can be addressed by SQL	true, false
supportsLikeEscapeClause	Boolean that identifies if the database supports ESCAPE clause in LIKE expression	true, false
nullSortOrder	Order in which NULLs are sorted	low, high, unknown
identifierQuote	String used as delimiter to denote (offset) identifier labels	String value (can be empty)
catalogSeparator	String used as delimiter (separator) between catalog (or schema) and table name	String value

The Data Services Platform XQuery engine typically quotes the names (identifiers) of object names to properly handle any special characters. The identifierQuote property (see Table) is obtained from the JDBC driver. However, different RDBMSs may use different identifiers for different database object names:

- catalogs
- schemas
- tables
- columns

If necessary, you can manually override the identifier quote property for each type of identifier (see Table).

Typically, the identifierQuote property obtained from the JDBC driver is used. However, if the specific quote property is available and the RDBMS uses it, you can modify the annotation settings in the .ds file (see “Relational Data Service Annotations” on page 6-6 for more information about these properties). The XQuery engine (metadata importer sub-system) uses the specific quote property (see Table) if it is available, otherwise, it uses the “identifierQuote” property provided by the JDBC driver.

The only exception to this rule is for Sybase versions below Sybase 12.5.2, which is treated as a base platform. Sybase does not use quotes for catalogs even though JDBC drivers return double quote ("") for “identifierQuote” property. The XQuery engine accommodates this mismatch by automatically setting “catalogQuote” property to the empty string.

Table B-26 Optional Quote Properties for Database Objects

Property	Description	Possible Values
catalogQuote	Special character used as quote to denote name of catalog	string
schemaQuote	Special character used as quote to denote name of schema	string
tableQuote	Special character used as quote to denote name of table	string
columnQuote	Special character used as quote to denote name of column	string

Data Type Mapping

When mapping SQL to XQuery datatypes, DSP XQuery engine first checks the JDBC typecode. If the typecode has a corresponding XQuery type, DSP uses the matching native type name. If no matching typecode or type name is available, the column is ignored.

Table B-27 Data Type Mapping (JDBC<–>XQuery Equivalents)

JDBC Data Type	Typecode	XQuery Data Type
BIGINT	-5	xs:long
BINARY	-2	xs:string
BIT	-7	xs:boolean
BLOB	2004	xs:hexBinary
BOOLEAN	16	xs:boolean
CHAR	1	xs:string
CLOB[1]	2005	xs:string

JDBC Data Type	Typecode	XQuery Data Type
DATE	91	xs:date[2]
DECIMAL (p,s)[3]	3	xs:decimal (if s > 0), xs:integer (if s =0)
DOUBLE	8	xs:double
FLOAT	6	xs:double
INTEGER	4	xs:int
LONGVARBINARY	-4	xs:hexBinary
LONGVARCHAR[1]	-1	xs:string
NUMERIC (p,s)[3]	2	xs:decimal (if s > 0), xs:integer (if s =0)
REAL	7	xs:float
SMALLINT	5	xs:short
TIME[4]	92	xs:time[4]
TIMESTAMP[4]	93	xs:dateTime[2]
TINYINT	-6	xs:short
VARBINARY	-3	xs:hexBinary
VARCHAR	12	xs:string
OTHER	1111	DSP uses native data type name to map to an appropriate XQuery data type.
Other vendor-specific JDBC type codes		

1. Pushed down in project list only.
2. Values converted to local time zone (timezone information removed) due to DATE limitations. See "Date and Time Data Type Differences: Timezones and Time Precision" on page 3-6 for more information.
3. Where *p* is precision (total number of digits, both to the right and left of decimal point) and *s* is scale (total number of digits to the right of decimal point).
4. Precision of underlying RDBMS determines the precision of TIME data type and how much truncation, if any, will occur in translating xs:time to TIME.

Function and Operator Pushdown

The following table lists functions and operators that are pushed down to base RDBMSs. See "fn-bea:sql-like" on page 2-22 for details about two-argument and three-argument versions of the fn-bea:sql-like() function.

Table B-28 Functions and Operators

Group	Functions and Operators
Logical operators	and, or, not
Numeric arithmetic	+, -, *[1]
	div[2]
Numeric comparisons[1]	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge
String comparisons[3]	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge
String functions	contains[4], starts-with[4], ends-with[4], fn-bea:sql-like(2), fn-bea:sql-like(3),[4] upper-case, lower-case
Datetime comparisons	=, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge on xs:dateTime, xs:date, xs:time
Other	empty, exists

1. All numeric types
2. Support for xs:decimal, xs:float, and xs:double data types only.
3. Arguments must be CHAR or VARCHAR SQL data types.
4. First argument must be SQL data type CHAR or VARCHAR; second argument must be a constant or parameter; and RDBMS must support LIKE (with ESCAPE) clause.

Cast Operation Pushdown

For base RDBMS, cast operations are not pushed down.

Other SQL Generation Capabilities

The following table shows other SQL Pushdown capabilities, as discussed in "Common Query Patterns" on page 3-13.

Table B-29 SQL Generation Capabilities

Query	Supported
If-Then-Else	no
Inner joins	yes (SQL-89 syntax)
Outer joins	no
Semi-joins, Anti-semi-joins	no
Order by	yes
Order by: Empty (NULL) order supported	Database-dependent
Order by: Aggregate function in ordering expression	no
Group by	yes (by column only)
Distinct pattern	yes
Trivial aggregate pattern	no
Direct SQL composition	no