# BEA AquaLogic® Data Services Platform

## XQuery and XQSE 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/docs30/index.html

Version: **3.0**
Document Date: June 2005
Revised: January 2008

# Contents

CHAPTER 1

# Introducing the AquaLogic 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 AquaLogic Data Services Platform (ALDSP). Links to more information about XQuery are also provided.

The following topics are covered:

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

## XML and XQuery

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

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

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

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

## XQuery Use in AquaLogic Data Services Platform

AquaLogic Data Services Platform models the contents of various types of data sources as XML schemas. After you have configured AquaLogic 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 AquaLogic Data Services Platform. AquaLogic 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/` |
| 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 AquaLogic Data Services Platform, see "BEA XQuery Language Implementation" on page 2-34. |

## 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

# 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 these 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
- BEA XQuery Language Implementation

## BEA XQuery Function Implementation

AquaLogic Data Services Platform 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, AquaLogic Data Services Platform 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. |
| **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: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. |
| | `fn-bea:timeout-with-label` | Same as fn-bea:timeout but with label to support auditing. |
| | `fn-bea:fail-over`<br>`fn-bea:fail-over-with-label` | Returns either the full result of the primary expression, or the full result of the alternate expression in cases when the primary XQuery expression fails.<br>For fn-bea:fail-over-with-label the audit record also contains the label, specified as an argument |
| | `fn-bea:fail-over-retry`<br>`fn-bea:fail-over-retry-with-label` | Returns either the full result of the primary expression, or the full result of the alternate expression in cases when the primary XQuery expression fails.<br>The functions re-evaluate the primary expression for each subsequent evaluation even if the evaluation of the expression raises an error.<br>For fn-bea:fail-over-retry-with-label the audit record also contains the label, specified as an argument. |

**Table 2-1 BEA XQuery Function Extensions (Continued)**

| | | |
|---|---|---|
| **Numeric Functions** | `fn-bea:format-number` | Converts a double to a string using the specified format pattern. |
| | `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:rename` | Renames a sequence of elements. |
| **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. |
| | `fn-bea:pad-left` | Adds a specified number of characters to the left of a specified string. Optionally, the character string used in padding can also be specified. |
| | `fn-bea:pad-right` | Adds a specified number of characters to the right of a specified string. Optionally the character string used in padding can also be specified. |
| **Extended XQuery Data Model (XXDM) Functions** | `fn-bea:current-value` | Returns an XQuery Data Model (XDM) instance representing the current value of the specified argument. |
| | `fn-bea:old-value` | Returns an XDM instance representing the value of the specified argument prior to modification. |

## Access Control Functions

AquaLogic Data Services Platform (ALDSP) 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 AquaLogic 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 AquaLogic Data Services Platform 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 `$resource` is the name of the resource, and `$dataservice` 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("CustomerProfile/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(xs: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 2-13.

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 2-13.

Examples:

- `fn-bea:date-to-string-with-format("yy-dd-mm", xs:date("2005-07-15"))` returns the string "05-15-07".
- `fn-bea:date-to-string-with-format("yyyy-mm-dd", xs: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 2-13.

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 2-13.

Examples:

- `fn-bea:dateTime-to-string-with-format(“dd MMM yyyy hh:mm a G”, xs: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”, xs: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(xs: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 2-13.

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 2-13.

Examples:

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

**Table 2-2 Date and Time Patterns (Continued)**

| | | |
|---|---|---|
| **S** | Millisecond | 978 |
| **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 | PST<br>PDT |

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 AquaLogic Data Services Platform execution control function extensions to the BEA implementation of XQuery:

- fn-bea:async
- fn-bea:fail-over, fn-bea:fail-over-with-label, fn-bea:fail-over-retry, and fn-bea:fail-over-retry-with-label
- fn-bea:fence
- fn-bea:timeout and fn-bea:timeout-with-label

## 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()*) as item()*
```

where `$expression` is the XQuery expression to evaluate asynchronously.

The `fn-bea:async` function enables asynchronous execution of Web services to reduce problems caused by the latency of these services.

**Note:** Asynchronous web services do not propagate the transaction context to other threads, regardless of the transaction settings. Asynchronous operations are likewise unable to start new transactions.

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:timeout and fn-bea:timeout-with-label

The timeout functions return either of the following:

- The full result of the primary expression.
- The full result of an *alternate expression,* in cases where the primary XQuery expression times out or fails. One or two alternate expressions can be returned, as described below.

Timeout functions are designed to be highly configurable. In the case of an error condition, the function can return either a single `$alt` expression or it can return more detailed information as `$timeout` and `$failure`.

The difference between the two functions `fn-bea-timeout( )` and `fn-bea-timeout-with-label( )` is that the latter returns `$label` along with other auditing information when an error condition is encountered.

### fn-bea-timeout Signature

The `fn-bea:timeout( )` function has the following signature:

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

where `$seq` is the primary XQuery expression to evaluate, `$millisec` is the timeout value in milliseconds, `$timeout` is returned if the evaluation of `$seq` takes more than `$millis` milliseconds to execute. `$failure` is returned if the evaluation of `$seq` raises an error.

Alternatively, you can replace the `$timeout` and `$failure` parameters with a single `$alt` parameter. The result of `$alt` will then be returned if a timeout or other error occurs.

### fn-bea-timeout-with-label Signature

The `fn-bea:timeout-with-label( )` function has the following signature:

```
fn-bea:timeout-with-label($seq as item()*,
                          $millisec as xs:integer,
                          $timeout as item()*
                          $failure as item(),
                          $label as xs:string) as item()*
```

where `$label` represents information provided to the audit record.

### Operational Details

Both functions return the result of evaluating `$seq` if the evaluation of `$seq`:

1. Does not raise an error and
2. Does not take more than the value of `$millis` (in milliseconds) to execute.

If an error does occur or the millisecond limit is exceeded, the alternate expression is returned along with the audit record.

The audit record contains:

- Name of the function call
- Source location of the function call (if available)
- Timeout value that was exceeded, if the execution of `$seq` timed out, or
- The error that was raised by execution of `$seq`
- Label, if the version of timeout that returns `$label` is invoked.

If the evaluation of `$millis` or `$alt` raises an error, the error is reported in the usual way. That is, neither of the functions attempts to handle the returned error.

If — for a specific instance of one of these functions in a query — the evaluation of `$seq` raises an error or “times out”, all subsequent evaluations of this instance during the same query evaluation will return `$timeout` and `$failure` (or `$alt`). No attempt to re-evaluate `$seq` is made in such a case.

You can use the timeout functions 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.
- To handle slow or unavailable resources uniformly.

Note that the timeout functions immediately return the alternative expression in cases when accessing the data source causes an error.

Here is an example where `$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()
      )
   )
```

## fn-bea:fail-over, fn-bea:fail-over-with-label, fn-bea:fail-over-retry, and fn-bea:fail-over-retry-with-label

The `fn:bea:fail-over` and `fn:bea:fail-over-with-label` functions return the result of evaluating $seq if the evaluation of $seq does not raise an exception. If it does raise an exception, $alt is returned. Both functions are polymorphic and their static return type is the union of the static types of `$seq` and `$alt`.

The functions have the following signatures:

```
fn-bea:fail-over($seq as item()*,
                 $alt as item()*) as item()*


fn-bea:fail-over-with-label($seq as item()*,
                            $alt as item()*,
                            $label as xs:string) as item()*
```

If `$alt` is returned the audit record contains:

- The name of the function call
- The source locations of the function call (if available)
- The exception that was raised by the execution of `$seq`

  For `fn-bea:fail-over-with-label` the audit record also contains `$label`.

If the evaluation of `$seq` raises an exception, all subsequent evaluations of this instance during the same query evaluation will return `$alt`. No attempt to re-evaluate `$seq` is made. If the evaluation of $alt raises an exception, it is simply reported. No attempt is made to handle the error.

The `fn:bea:fail-over-retry` and `fn:bea:fail-over-retry-with-label` functions return the result of evaluating $seq if the evaluation of $seq does not raise an exception. If it does raise an exception, $alt is returned.

In contrast to the `fn:bea:fail-over` and `fn:bea:fail-over-with-label` functions, however, the `fn:bea:fail-over-retry` and `fn:bea:fail-over-retry-with-label` functions re-evaluate $seq for each subsequent evaluation even if the evaluation of $seq raises an error.

The `fn:bea:fail-over-retry` and `fn:bea:fail-over-retry-with-label` functions have the following signatures:

```
fn-bea:fail-over-retry($seq as item()*,
                       $alt as item()*) as item()*

fn-bea:fail-over-retry-with-label($seq as item()*,
                                  $alt as item()*,
                                  $label as xs:string) as item()*
```

### Usage Suggestions

The `fn-bea:fail-over( )` functions can be used in two ways:

- A fail-over function can be placed around an "optional" XQuery result. Then, if expected result is not returned, at least the remainder of the query results will be returned. In such a case, the XMLtype (schema) needs to be constructed in such a way that the results remain valid when some portion of the information is not returned.
- Nested invocations can be used to select an available data source from among a set of (possibly) heterogeneous and (possibly) unavailable data sources. Each invocation can access the appropriate available source and restructure its answer set appropriately for the surrounding context. Best practices in query construction would likely involve the use of functions to restructure the content.

## 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 1.5.0 `DecimalFormat` class. (For information on DecimalFormat and other JDK 1.5.0 Java classes see: http://java.sun.com/j2se/1.5.0.)

## 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 hundredths, 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 hundredths, 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 AquaLogic Data Services Console. If so, it returns this value as a string. In cases when the property is not defined, the function returns the default value.

The function has the following signature:

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

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

### fn-bea:inlinedXML

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

The function has the following signature:

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

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

Examples:

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

## fn-bea:rename

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

The function has the following signature:

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

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

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

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

Example:

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

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

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

The output value would be:

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

## QName Functions

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

### fn-bea:QName-from-string

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

The function has the following signature:

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

where `$name` is the local name.

## Sequence Functions

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

- fn-bea:interleave

### fn-bea:interleave

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

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

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

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

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

## String Functions

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

- fn-bea:match
- fn-bea:sql-like
- fn-bea:trim
- fn-bea:trim-left
- fn-bea:trim-right
- fn-bea:pad-left
- fn-bea:pad-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. 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 is the regular expression.

Regular expression use standard `java.util.regex.Pattern` class patterns. Currently the following link to regular expression constructs is valid:

`http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html`

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

The $escape character has to be exactly one character in length and cannot be either the percent ("%") or underscore ("_") 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. AquaLogic Data Services Platform 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(())` returns the empty sequence.

## fn-bea:trim-right

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

This 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(())` returns the empty sequence.

## fn-bea:pad-left

The `fn-bea:pad-left` functions add padding characters to the left of a string to create a fixed-length string. There are two variations of the function:

- The other uses the default character, which is a space (ASCII 32).
- One allows pad characters to be specified.

If the input string exceeds the requested length, only a substring as long as the length is returned.

### Pad Left Function Using Default Character (ASCII 32)

The function has the following signature:

```
fn-bea:pad-left($str as xs:string?, $size as xs:integer?) as xs:string?
```

where string (`$str`) is returned with a specified number (`$size`) of characters (ASCII 32) prepended to the left of the string. The result is a string of length $size. It consists of $str prepended with $size - fn:length($str) space characters.

Examples:

- `fn-bea:pad-left("abcd", 6)` prepends spaces to the string up to the maximum 6 specified. The returned string is: `"  abcd"`.
- `fn-bea:pad-left("abcd", 2)` returns only `"ab"` because characters are only prepended to the complete string. In addition, only the first two characters are returned since that is the setting of `$size`.

Additional notes:

- If either argument is an empty sequence, an empty sequence is returned.
- If `$size` is negative, a runtime exception occurs.

### Pad Left Function with Specified Pad String

This function has the following signature:

```
fn-bea:pad-left($str as xs:string?, $size as xs:integer?, $pad as
xs:string?) as xs:string?
```

where string (`$str`) is returned with an arbitrary number (`$size`) of prepended characters with the pad string (`$pad`) replicated as many times as necessary.

Examples:

- `fn-bea:pad-left("abcd", 6, "01")` prepends a pad string to the string up to the maximum 6 specified. The returned string is: `"01abcd"`.
- `fn-bea:pad-left("abcd", 2, "01")` returns only `"ab"` because characters are only prepended to a complete string. In addition, only the first two characters are returned since that is the setting of `$size`.
- `fn-bea:pad-left("abc", 6, "01")` returns `"010abc"`. Note that the prepended string is returned completely once and then partially up to the length ($size) specified.

Additional notes:

- If either argument is an empty sequence, an empty sequence is returned.
- If `$size` is negative, a runtime exception occurs.

## fn-bea:pad-right

The `fn-bea:pad-right` functions add padding characters to the right of a string to create a fixed-length string. There are two variations of the function:

- The other uses the default character, which is a space (ASCII 32).
- One allows pad characters to be specified.

If the input string exceeds the requested length, only a substring as long as the length is returned.

### Pad Right Function Using Default Character (ASCII 32)

The function has the following signature:

```
fn-bea:pad-right($str as xs:string?, $size as xs:integer?) as xs:string?
```

where string (`$str`) is returned with a specified number (`$size`) of characters (ASCII 32) appended to the string. The result is a string of length $size. It consists of $str appended with $size - fn:length($str) space characters.

Examples:

- `fn-bea:pad-right("abcd", 6)` appends spaces to the string up to the maximum 6 specified. The returned string is: `"abcd "`.
- `fn-bea:pad-right("abcd", 2)` returns only `"ab"` because characters are only appended to a complete string. In addition, only the first two characters are returned since that is the setting of `$size`.

Additional notes:

- If either argument is an empty sequence, an empty sequence is returned.
- If `$size` is negative, a runtime exception occurs.

### Pad Right Function with Specified Pad String

This function has the following signature:

```
fn-bea:pad-right($str as xs:string?, $size as xs:integer?, $pad as
xs:string?) as xs:string?
```

where string (`$str`) is returned with an arbitrary number (`$size`) of appended characters with the pad string (`$pad`) replicated as many times as necessary.

Examples:

- `fn-bea:pad-right("abcd", 6, "01")` prepends a pad string to the string up to the maximum 6 specified. The returned string is: `"abcd01"`.
- `fn-bea:pad-right("abcd", 2, "01")` returns only `"ab"` because characters are only appended to a complete string. In addition, only the first two characters are returned since that is the setting of `$size`.
- `fn-bea:pad-right("abc", 6, "01")` returns only `"abc010"`. Note that the appended string is returned completely once and then partially up to the length ($size) specified.

Additional notes:

- If either argument is an empty sequence, an empty sequence is returned.
- If `$size` is negative, a runtime exception occurs.

## Extended XQuery Data Model (XXDM) Functions

AquaLogic Data Services Platform includes functions to support the Extended XQuery Data Model (XXDM). The XXDM represents instances of the XQuery Data Model (XDM) along with information about changes to the instances.

This section describes functions that you can use to convert XXDM instances to XDM instances.

**Note:** ALDSP 3.2 offers several new XQuery functions for manipulating and applying changes to XML element instances. See "Introducing Mutators for Updates" in the ALDSP 3.2 New Features Supplement

### fn-bea:current-value

The `fn-bea:current-value` function returns an XDM instance representing the current value of the specified argument (discarding information about applied changes).

The function has the following signature:

```
fn-bea:current-value($changed as changed-element()) as element()?
```

where `$changed` is the XXDM instance.

### fn-bea:old-value

The `fn-bea:old-value` function returns an XDM instance representing the value of the specified argument prior to modification.

The function has the following signature:

```
fn-bea:old-value($changed as changed-element()) as element()?
```

where `$changed` is the XXDM instance.

Both the `fn-bea:current-value` and `fn-bea:old-value` functions are polymorphic.

Example:

The following function returns the salary difference for a customer before and after modification.

```
declare function salaryDifference($cus as changed-element
(cus:customer)) as xs:decimal {
   fn:data(fn-bea:get-current-value($cus)/salary - fn:data(fn-
   bea:get-old-value($cus)/salary)
}
```

The function does this by accessing the current and old versions of the customer element, extracting the salaries, and subtracting to determine the difference.

## 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:collection`
- `fn:doc`
- `fn:id`
- `fn:idref`
- `fn:normalize-unicode`

## Implementation-Specific Functions and Operators

This section describes BEA-specific implementation details related to functions and operators.

**Table 2-3 Implementation-Defined Values**

| Section | Description | AquaLogic Data Services Platform 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. AquaLogic Data Services Platform 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 AquaLogic Data Services Platform XQuery Engine
- Implementation-Defined Values for XQuery Language Processing

### XQuery Language Support (and Unsupported Features)

The AquaLogic Data Services Platform 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 AquaLogic Data Services Platform XQuery Engine

Beyond compliance with the specification, BEA AquaLogic Data Services Platform's XQuery language implementation (the AquaLogic Data Services Platform 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 problem of grouping books by year, without losing 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 an XQuery expression that can be used 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 function, 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>
```

The following tables (Table 2-4 and Table 2-5) show book bindings before and after the group by clause is applied.

**Table 2-4 Bindings Before Group By Clause is Applied**

| $book |
|---|
| `<book year=”1994” ISBN=”147...”>...</book>` |
| `<book year=”1994” ISBN=”198...”>...</book>` |
| `<book year=”2000” ISBN=”123...”>...</book>` |

**Table 2-5 Bindings After Group By Clause is Applied**

| $year | $partition |
|---|---|
| 1994 | `(<book year=”1994” ISBN=”147...”>...</book>,`<br>` <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 (see Table 2-4).

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` (see Table 2-5).

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 means 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

This section describes the BEA-specific implementation details related to XQuery language processing.

**Table 2-6 Implementation-Defined Values**

| Section | Description | AquaLogic Data Services Platform 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. |

**Table 2-6 Implementation-Defined Values**

| | | |
|---|---|---|
| 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 analysis 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; the XQuery flagger | Mechanism by which the XQuery flagger (which flags queries containing 'must understand' extensions) is enabled, if at all. By default the flagger is disabled. | XQuery flagger is not supported. |
| 2.6.7.1 Static Typing Extensions; the 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 &#0151; 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. |

**Table 2-6 Implementation-Defined Values**

| | | |
|---|---|---|
| 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 implementer. | Not applicable—Since the AquaLogic Data Services Platform 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 |

# XQuery Engine and SQL

This chapter provides an overview of how AquaLogic Data Services Platform works with relational data, and describes what happens when a relational data source is imported into AquaLogic Data Services Platform.

The chapter also explains how SQL data types are mapped to XQuery data types and describes what happens during runtime after deploying a data-service-enabled application. The chapter further explains how queries are handled and describes the kind of performance you can expect.

This chapter covers the following topics:

Note that while the graphical-user interface tools handle many of the details, SQL developers and application-performance tuning experts should understand how AquaLogic Data Services Platform 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

**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 "AquaLogic Data Services Platform Components Architecture" figure in the *Concepts Guide*).

## Introduction

At the core of BEA AquaLogic Data Services Platform 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 1-3, with additional enhancements as detailed in "BEA's XQuery Implementation" on page 2-1.

In addition to compliance with XQuery and XML recommendations, AquaLogic Data Services Platform 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), AquaLogic Data Services Platform uses JDBC to generate SQL queries and submit them to the appropriate RDBMSs that comprise a data service, which means AquaLogic Data Services Platform 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, AquaLogic Data Services Platform's support for SQL is not a "one-size-fits-all" exercise: achieving optimal integration with relational data sources requires AquaLogic Data Services Platform 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 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

AquaLogic Data Services Platform provides two different levels of support for relational data sources:

- **Base support**. AquaLogic 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, and MySQL.
- **Core support**. AquaLogic 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 AquaLogic Data Services Platform RDBMS Support**

| RDBMS and Versions | Vendor Driver | BEA WebLogic Driver |
|---|---|---|
| IBM DB2/NT 8 (and higher) | IBM DB2 JDBC thin driver, version 8.01 | BEA (DataDirect) JDBC driver for DB2, version 3.6. |
| Microsoft SQL Server 2000 (and higher) | Microsoft SQLServer JDBC driver, version 2.2 | BEA (DataDirect) JDBC driver for SQLServer, version 3.6 |
| Oracle 8.1.x, 9.x, 10.x | Oracle JDBC Thin driver, version 10.1 | BEA (DataDirect) JDBC driver for Oracle, version 3.6 |
| PointBase 5.1 (and higher) | PointBase JDBC driver, version 5.1 | N/A |
| Sybase Adaptive Server Enterprise 12.5.2 (and higher) | Sybase jConnect driver, version 5.5 | BEA (DataDirect) JDBC driver for Sybase, version 3.6 |
| Teradata V2R5 (and higher) | Teradata JDBC driver | N/A |

## How the XQuery Engine Supports SQL Data Sources

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, AquaLogic Data Services Platform 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 3-6).

### Metadata and Data Type Mappings Get Stored in Annotated Files

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

- **Entity 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:
  - Relational provider identifier.
  - 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 AquaLogic Data Services Platform 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 (and higher)<br>Microsoft SQL Server 2000 (and higher)<br>Oracle 8.1.x, 9.x, 10.x<br>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 5.1<br>Teradata V2R5 | 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 3-3. 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 AquaLogic Data Services Platform, 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] | |
| Oracle 9.x, 10.x | | `DATE, TIMESTAMP, TIMESTAMP WITH LOCAL TIMEZONE, TIMESTAMPWITH TIMEZONE` | |
| PointBase 5.1 | `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).

AquaLogic Data Services Platform 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 AquaLogic Data Services Platform Handles Timezone Information

When a query is being pushed down to an RDBMS that does not support timezone data, the AquaLogic Data Services Platform 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 update, when a date or time value must be stored in the RDBMS.

### How AquaLogic Data Services Platform Handles Fractional Seconds

The XQuery language supports unlimited precision for fractional seconds, while the AquaLogic Data Services Platform 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 there may be no fractional second support at all (Oracle 8.1.x, for example). In translating from XQuery to SQL, AquaLogic Data Services Platform 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 AquaLogic Data Services Platform 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 3-33 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' 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

AquaLogic Data Services Platform achieves optimal performance for queries by performing *SQL pushdown*. Pushdown is 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 AquaLogic Data Services Platform 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:

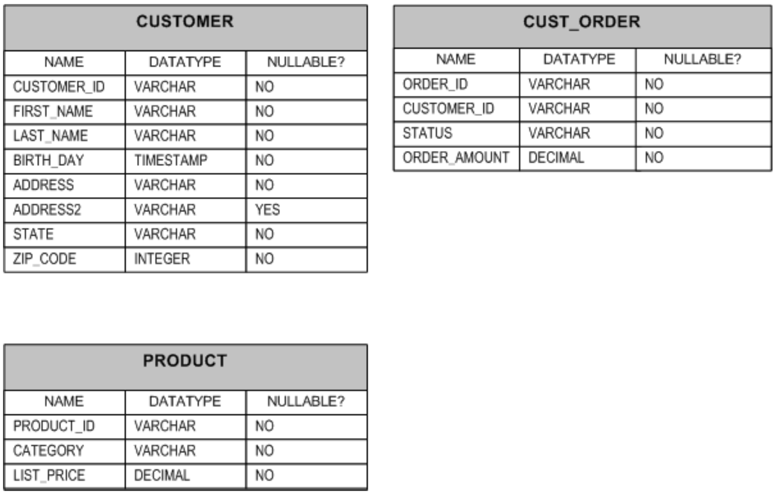
- Basic language constructs, including constants, variables, path expressions, functions and operators, and cast operations.
- Common query patterns, such as selections and projections (where clauses), joins (inner, outer, semi-join, anti-semi-join), ordering clauses, groupings and aggregations.

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 shows an XQuery statement and its corresponding "pushdown" or SQL translation. (Oracle syntax is used.)

**Table 3-5 Function Pushdown Example**

| XQuery Statement | SQL Translation (Oracle Syntax) |
|---|---|
| `for $c in CUSTOMER()`<br>`return lower-case($c/LAST_NAME)` | `SELECT LOWER(t1."LAST_NAME") AS c1`<br>`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;`<br>`...`<br>`for $c in CUSTOMER()`<br>`where starts-with($c/LAST_NAME, string-join( ("a", "b"), $p ))`<br>`return $c/FIRST_NAME` | `SELECT t1."FIRST_NAME" AS c1`<br>`FROM "CUSTOMER" t1`<br>`WHERE t1."LAST_NAME" LIKE ?` |

### Aggregate Functions

AquaLogic Data Services Platform 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 3-8. See "Grouping and Aggregation" on page 3-23 for some examples of how aggregate functions in conjunction with other expressions affect the outcome of SQL pushdown.

## Parameters in Generated SQL Statements

The AquaLogic Data Services Platform 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 7-1 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 $c in CUSTOMER()`<br>`where xs:string($c/ZIP_CODE) eq "95131"`<br>`return $c/CUSTOMER_ID` | `SELECT t1."CUSTOMER_ID" AS c1`<br>`FROM "CUSTOMER" t1`<br>`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 3-10) provide access to ZIP_CODE and LAST_NAME columns.

## Constant Pushdown

The AquaLogic Data Services Platform 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()`<br>`where $c/ZIP_CODE eq 95131`<br>`return $c/LAST_NAME` | `SELECT t1."LAST_NAME" AS c1`<br>`FROM "CUSTOMER" t1`<br>`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 datatype is supported by the XQuery engine and:

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

Table 3-10 shows an example of variable pushdown.

**Table 3-10 Variable Pushdown**

| XQuery Statement | SQL Statement |
|---|---|
| `declare variable $extVar`<br>`as xs:string external;`<br><br>`for $c in CUSTOMER()`<br>`where $c/CUSTOMER_ID eq $extVar`<br>`return $c/LAST_NAME` | `SELECT t1."LAST_NAME" as c1`<br>`FROM "CUSTOMER" t1`<br>`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 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 3-9 for more information.

Table 3-12 shows an example of a where clause pushdown.

**Table 3-12 Where Clause Pushdown**

| XQuery Statements | SQL Statements |
|---|---|
| `for $c in CUSTOMER()`<br>`where $c/CUSTOMER_ID eq “CUSTOMER01”`<br>`return $c/LAST_NAME` | `SELECT t1.”LAST_NAME” AS c1`<br>`FROM “CUSTOMER” t1`<br>`WHERE t1.”CUSTOMER_ID” = ‘CUSTOMER01’` |
| `for $c in CUSTOMER()`<br>`where year-from-dateTime($c/BIRTH_DAY)`<br>`eq`<br>`  year-from-date(current-date())`<br>`return`<br>`  $c/LAST_NAME` | `(DB2 syntax)`<br>`SELECT t1.”LAST_NAME” AS c1`<br>`FROM “CUSTOMER” t1`<br>`WHERE`<br>`  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 3-25).

## 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 shows an example of an order by clause pushdown.

**Table 3-13 Order By Pushdown**

| XQuery Statement | SQL Statement |
|---|---|
| `for $c in CUSTOMER()`<br>`  order by $c/CUSTOMER_ID descending`<br>`return $c/CUSTOMER_ID` | `SELECT t1."CUSTOMER_ID" AS c1`<br>`FROM "CUSTOMER" t1`<br>`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 when the RDBMS supports dynamic setting of NULL order.

**Table 3-14 Order By Query When Setting NULL Order Dynamically**

| XQuery Statement | SQL Statement (Oracle Syntax) |
|---|---|
| `for $c in CUSTOMER()`<br>`order by $c/ADDRESS2 ascending`<br>`      empty greatest`<br>`return $c/CUSTOMER_ID, $c/ADDRESS2` | `SELECT t1."CUSTOMER_ID" AS c1,`<br>`       t1."ADDRESS2"    AS c2`<br>`FROM "CUSTOMER" t1`<br>`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 AquaLogic Data Services Platform 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 can see these as well in the Query Plan View.

## Inner Join Pushdown

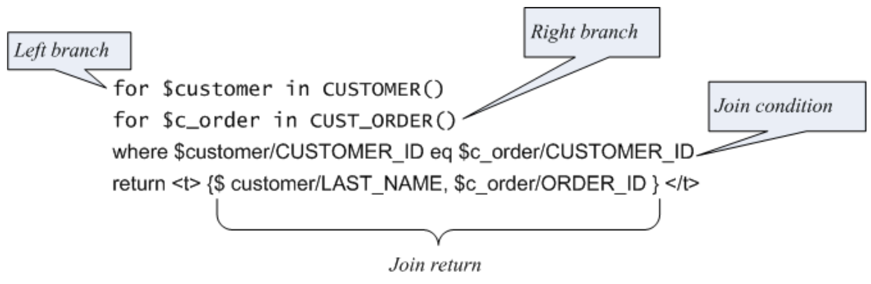
Joining data from multiple sources is a very common data integration task. In SQL terms, an inner join relates each row in one table (or view) to one or more corresponding rows in another table or view. In XQuery, an inner join is expressed as a FLWR expression comprising several *for* clauses that iterate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



```
for $customer in CUSTOMER()
for $c_order in CUST_ORDER()
where $customer/CUSTOMER_ID eq $c_order/CUSTOMER_ID
return <t> {$ customer/LAST_NAME, $c_order/ORDER_ID } </t>
```

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. See also **Table 3-16**.

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

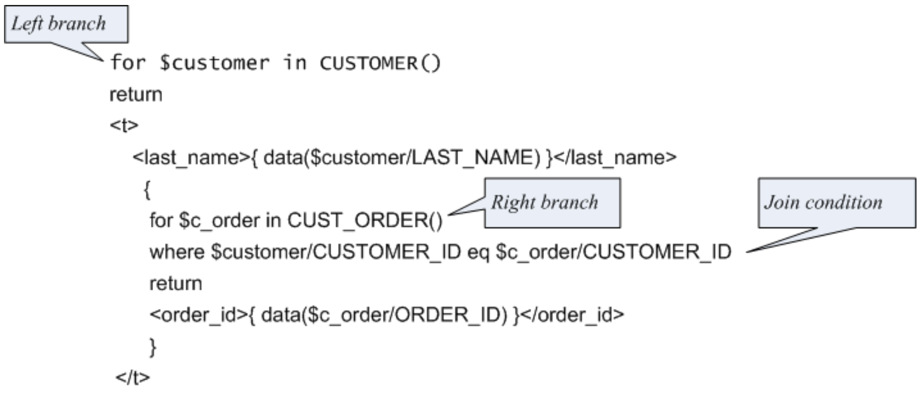
| SQL-92 Syntax | SQL-89 Syntax |
|---|---|
| `SELECT t1."LAST_NAME" AS c1, t2."ORDER_ID" AS c2`<br><br>`FROM "CUSTOMER" t1 JOIN "CUST_ORDER" t2`<br>`ON t1."CUSTOMER_ID" = t2."CUSTOMER_ID"` | `SELECT t1."LAST_NAME" AS c1, t2."ORDER_ID" AS c2`<br><br>`FROM "CUSTOMER" t1, "CUST_ORDER" t2`<br>`WHERE t1."CUSTOMER_ID"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



```
for $customer in CUSTOMER()
return
<t>
   <last_name>{ data($customer/LAST_NAME) }</last_name>
    {
     for $c_order in CUST_ORDER()
     where $customer/CUSTOMER_ID eq $c_order/CUSTOMER_ID
     return
     <order_id>{ data($c_order/ORDER_ID) }</order_id>
     }
 </t>
```

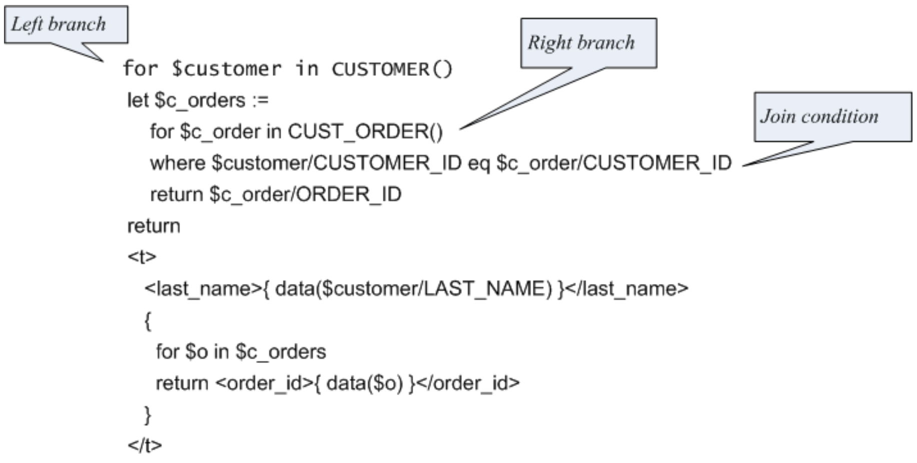
The SQL code generated by the XQuery engine depends on the SQL dialect supported by the source database (see "XQuery-SQL Mapping Reference" for details). Table 3-18 shows example SQL-92 and proprietary syntax for the query shown in Figure 3-17.

**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`<br>`FROM "CUSTOMER" t1 OUTER JOIN "CUST_ORDER" t2`<br>`ON t1."CUSTOMER_ID" = t2."CUSTOMER_ID"` | `SELECT t1."LAST_NAME" AS c1, t2."ORDER_ID" AS c2`<br>`FROM "CUSTOMER" t1, "CUST_ORDER" t2`<br>`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**



```
for $customer in CUSTOMER()
let $c_orders :=
   for $c_order in CUST_ORDER()
   where $customer/CUSTOMER_ID eq $c_order/CUSTOMER_ID
   return $c_order/ORDER_ID
return
<t>
  <last_name>{ data($customer/LAST_NAME) }</last_name>
  {
    for $o in $c_orders
    return <order_id>{ data($o) }</order_id>
  }
</t>
```

## 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 7-1 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") quantifier [semi-join] | for $customer in CUSTOMER()<br>where<br>**some** $c_order in CUST_ORDER()<br>**satisfies** ($customer/CUSTOMER_ID eq $c_order/ORDER_ID)<br>and<br>($c_order/STATUS eq "OPEN")<br>return<br>$customer/CUSTOMER_ID | SELECT t1."CUSTOMER_ID" AS c1<br>FROM "CUSTOMER" t1<br>WHERE **EXISTS**(<br>SELECT 1<br>FROM "CUST_ORDER" t2<br>WHERE t1."CUSTOMER_ID" = t2."CUSTOMER_ID" AND t2."STATUS" = 'OPEN'<br>) |
| FLWR with negation of existential quantifier [anti-semi join] | for $customer in CUSTOMER()<br>where **not**(<br>**some** $c_order in CUST_ORDER()<br>**satisfies** ($customer/CUSTOMER_ID eq $c_order/ORDER_ID)<br>and<br>($c_order/STATUS eq "OPEN")<br>)<br>return<br>$customer/CUSTOMER_ID | SELECT t1."CUSTOMER_ID" AS c1<br>FROM "CUSTOMER" t1<br>WHERE **NOT EXISTS**(<br>SELECT 1<br>FROM "CUST_ORDER" t2<br>WHERE t1."CUSTOMER_ID" = t2."CUSTOMER_ID" AND t2."STATUS" = 'OPEN'<br>) |

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

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

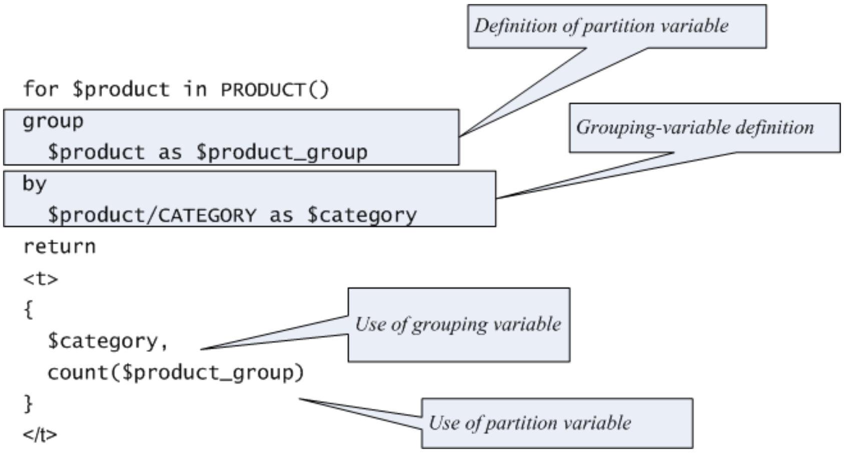
## Grouping and Aggregation

The 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 2-34 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**



```
for $product in PRODUCT()
group
  $product as $product_group
by
  $product/CATEGORY as $category
return
<t>
{
  $category,
  count($product_group)
}
</t>
```

Definition of partition variable
Grouping-variable definition
Use of grouping variable
Use of partition variable

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 AquaLogic Data Services Platform 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` |

## Simple 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 (see Table 3-24), 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()`<br>`group $product/LIST_PRICE as $price_group`<br>`by $product/CATEGORY as $category`<br>`where max($price_group) gt 1000`<br>`return`<br>`<t>`<br>`{`<br>`  $category,`<br>`   min($price_group)`<br>`}`<br>`</t>` | `SELECT t1."CATEGORY" AS c1, MIN(t1."LIST_PRICE") AS c2`<br>`FROM "PRODUCT" t1`<br>`GROUP BY t1."CATEGORY"`<br>`HAVING MAX(t1."LIST_PRICE") > 1000` |

## Outer Join with Aggregate Pattern

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

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 3-26, 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 expressions 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.

An example can be seen in Table 3-26.

**Table 3-26 If-Then-Else Pushdown**

| XQuery Statement | SQL Statement |
|---|---|
| `for $i in CUST_ORDER()`<br>`return`<br>`     if ($i/STATUS eq "SHIPPED")`<br>`     then data($i/STATUS)`<br>`     else data($i/CUSTOMER_ID)` | `SELECT`<br>`     CASE WHEN (t1."STATUS" = 'SHIPPED')`<br>`     THEN t1."STATUS"`<br>`     ELSE t1."CUSTOMER_ID" END AS c1`<br>`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 and in Table 3-28.

Table 3-27 Two- and Three-Argument Variants of XQuery Subsequence Function

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

Table 3-28 Subsequence Pushdown

| XQuery Statement | SQL Statement (DB2) |
|---|---|
| `let $s :=`<br>`   for $i in t2:PRODUCT()`<br>`    order by $i/LIST_PRICE descending`<br>`    return $i`<br>`for $p in subsequence($s, 1, 10)`<br>`return <product>`<br>`    <name>`<br>`      { data($p/PRODUCT_NAME) } </name>`<br>`    <price>`<br>`      { data($p/LIST_PRICE) }`<br>`    </price>`<br>`</product>`<br>`};` | `SELECT t3.c1, t3.c2 FROM(`<br>`  SELECT ROW_NUMBER() OVER()`<br>`      as c3, t2.c1, t2.c2`<br>`    FROM(`<br>`      SELECT t1."LIST_PRICE" as c1,`<br>`        t1."PRODUCT_NAME" as c2`<br>`        FROM "RTLALL"."PRODUCT" t1`<br>`        ORDER BY t1."LIST_PRICE" DESC`<br>`      )t2`<br>`    )t3`<br>`WHERE(t3.c3 <11)` |

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 3-29 shows several different examples of the subsequence function in the context of specific queries.

**Table 3-29 Examples of XQuery Expressions using Subsequence Function**

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

**Table 3-29 Examples of XQuery Expressions using Subsequence Function (Continued)**

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

AquaLogic Data Services Platform 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.
- 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.
- Teradata V2R5 supports both versions of the subsequence function, without restriction.

**Note:** Subsequence pushdown is not supported for PointBase, Sybase, or any base RDBMS (see "XQuery-SQL Mapping Reference" on page 7-1 for other core and base RDBMS information.)

## Direct SQL Data Services and Pushdown

AquaLogic 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 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 7-1).

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`<br>`$external_variable as xs:string external;`<br>`for $recent_order in RECENT_ORDER()`<br>`where $recent_order/ORDER_ID eq $external_variable`<br>`return $recent_order/ORDER_AMOUNT` | `SELECT t1."ORDER_AMOUNT" AS c1`<br>`FROM (`<br>`  SELECT * FROM RECENT_ORDER`<br>`) t1`<br>`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()`<br>`for $recent_order in RECENT_ORDER()`<br>`where $customer/CUSTOMER_ID eq $recent_order/CUSTOMER_ID`<br>`return`<br>`<t>{ $customer/CUSTOMER_ID, $recent_order/ORDER_ID }</t>` | `SELECT t1."CUSTOMER_ID" AS c1, t2."ORDER_ID" AS c2`<br>`FROM "CUSTOMER" t1`<br>`JOIN (`<br>`  SELECT * FROM RECENT_ORDER`<br>`) t2`<br>`ON t1."CUSTOMER_ID" = t2."CUSTOMER_ID"` |

## Distributed Query Pushdown

AquaLogic Data Services Platform uses SQL pushdown to off-load query processing to the underlying data source RDBMS whenever possible. However, as mentioned in "How the XQuery Engine Supports SQL Data Sources" on page 3-4, 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, AquaLogic Data Services Platform 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 — a PPK Join Example

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

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

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

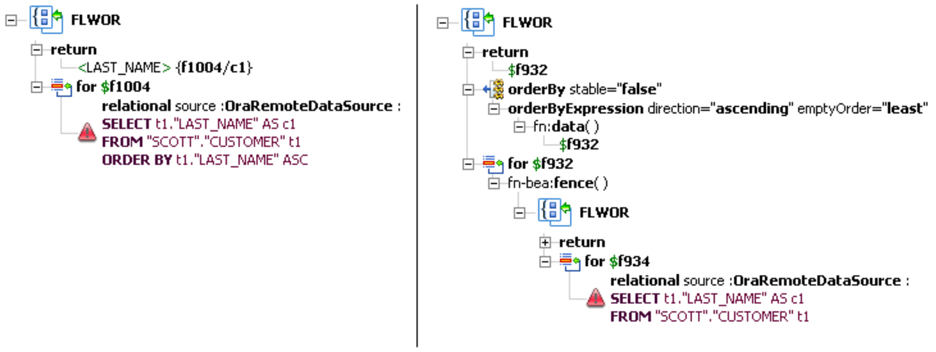
Use the fence() function whenever you want SQL to be sent as is, to the RDBMS. For example, if you are accessing an Oracle 8.5.x RDBMS that uses hints and Oracle's rule-based optimizer, you should send the hinted SQL queries to the data source by wrapping them in the fence() function.

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 AquaLogic Data Services Platform Query Plan View 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**



```
FLWOR
  return
    <LAST_NAME> {f1004/c1}
  for $f1004
    relational source :OraRemoteDataSource :
    SELECT t1."LAST_NAME" AS c1
    FROM "SCOTT"."CUSTOMER" t1
    ORDER BY t1."LAST_NAME" ASC
```

```
FLWOR
  return
    $f932
  orderBy stable="false"
    orderByExpression direction="ascending" emptyOrder="least"
      fn:data( )
        $f932
  for $f932
    fn-bea:fence( )
      FLWOR
        return
        for $f934
          relational source :OraRemoteDataSource :
          SELECT t1."LAST_NAME" AS c1
          FROM "SCOTT"."CUSTOMER" t1
```

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.

# 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 fully supports XML namespaces and includes namespaces in the queries generated in AquaLogic Data Services Studio.

This section includes the following topics:

- Introducing XML Namespaces
- Using XML Namespaces in AquaLogic 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 prefix to uniquely identify BEA-supplied functions.

Table 4-1 lists the predefined XQuery namespaces used in AquaLogic Data Services Platform queries.

**Table 4-1 Predefined Namespaces in XQuery**

| Namespace Prefix | Description | Examples |
|---|---|---|
| `fn` | The prefix for XQuery functions. | `fn:data()`<br>`fn:sum()`<br>`fn:substring()` |
| `fn-bea` | The prefix for AquaLogic Data Services Platform-specific extensions to the standard set of XQuery functions. | `fn-bea:rename()`<br>`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:

```
http://temp.openuri.org/SampleApp/CustOrder.xsd
```

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 fourth line of the schema binds the `xsd` prefix to the standard XML Schema URI. The fifth line of the schema contains a namespace declaration which associates a URI with the `bea` 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"/>
```

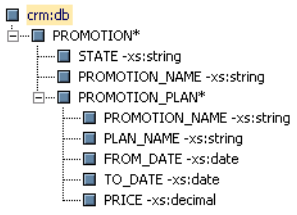
It is recommended that you create schemas with `elementFormDefault="unqualified"` and `attributeFormDefault="unqualified"`. This enables you to move elements between namespaces by renaming a single complex element, instead of having to explicitly map every element.

## Using XML Namespaces in AquaLogic Data Services Platform Queries and Schemas

AquaLogic Data Services Platform (ALDSP) automatically generates the namespace declarations when generating a query. ALDSP 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 the outermost complex element appears in the schema.

**Figure 4-2 Schema with Unqualified Attributes and Elements**



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

# 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
- Understanding Data Service Design Principles
- Applying Data Service Implementation Guidelines

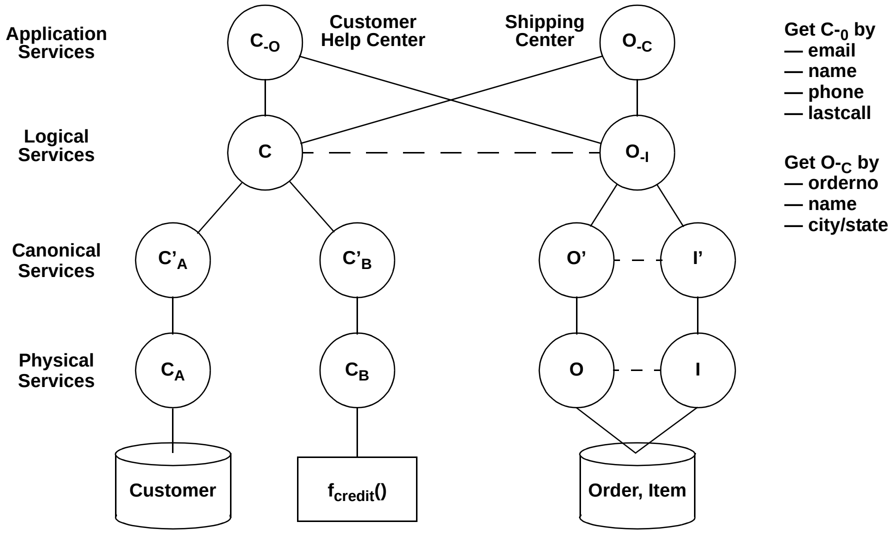
## Introducing Data Service Design

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 out 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 | AquaLogic Data Services Platform 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. |

**Table 5-2 Data Service Design Principles (Continued)**

| | | |
|---|---|---|
| **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. |
| **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.<br>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:<br>• Create navigation functions to create relationships between the data.<br>• Create a new table in the database to relate the unique identifiers and primary keys. |

**Table 5-2 Data Service Design Principles (Continued)**

| | | |
|---|---|---|
| **Physical Services** | Employ functions that get all records | Using protected 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:<br>`for $x in f1:CUSTOMER()`<br>`group $x as $g by 1`<br>`return count($g)`<br>instead of:<br>`count( f1:CUSTOMER() )`<br>in order to enable pushdown of the aggregation operation to the underlying relational data source.<br>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. |

**Table 5-3 Data Service Implementation Guidelines (Continued)**

| | | |
|---|---|---|
| **Application Services** | 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 AquaLogic Data Services Platform 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.<br>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:<br>• `fn:lower-case()` returns an empty string<br>• LOWER in Oracle returns NULL<br>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.<br><br>For example, consider the following XQuery expression:<br><pre>For $c in CUSTOMER()<br>Return<br>  &lt;CUSTOMER&gt;<br>    &lt;LAST_NAME&gt;$c/LAST_NAME&lt;/LAST_NAME&gt;<br>    &lt;FIRST_NAME&gt;$c/FIRST_NAME<br>        &lt;/FIRST_NAME&gt;<br>    &lt;ADDRESS&gt;{<br>       For $a in ADDRESS()<br>       Where $a/CUSTOMER_ID =<br>         $c/CUSTOMER_ID<br>       Return<br>         $a<br>    }&lt;/ADDRESS&gt;<br>  &lt;/CUSTOMER&gt;</pre>Defining a one-to-one relationship between a CUSTOMER and an ADDRESS, as in the following, can block optimizations.<br><pre>&lt;element name=CUSTOMER&gt;<br>   &lt;element name=LAST_NAME/&gt;<br>   &lt;element name=FIRST_NAME/&gt;<br>   &lt;element name=ADDRESS/&gt;<br>&lt;/element&gt;</pre> |

**Table 5-3 Data Service Implementation Guidelines (Continued)**

| | | |
|---|---|---|
| **Application Services** | Return plural for functions that contain FLWOR expressions *(continued)* | This is because AquaLogic Data Services Platform 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.<br><br>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.<br><br>`<element name=CUSTOMER>`<br>`   <element name=LAST_NAME/>`<br>`   <element name=FIRST_NAME/>`<br>`   <element name=ADDRESS minOccurs=”0”`<br>`      maxOccurs=”unbounded”/>`<br>`</element>` |
| | 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:<br><br>`define fn ($p string)`<br>`for $c in CUSTOMER()`<br>`for $o in ORDER()`<br>`where $c/id eq $p`<br>`and $o/id eq $p`<br><br>Instead, use the following form to specify the same query:<br><br>`define fn ($p string)`<br>`for $c in CUSTOMER()`<br>`for $o in ORDER()`<br>`where $c/id eq $o/id`<br>`and $c/id eq $p` |

# BEA XQuery Scripting Extension (XQSE)

This chapter describes the BEA XQuery Scripting Extension (XQSE) that enables you to add procedural constructs to XQuery-based data services. The chapter describes the language extensions and includes the statement grammar along with one or more examples.

This chapter includes the following topics:

## Introducing the XQuery Scripting Extension

ALDSP data services are based on the XQuery language, which enables you to use the structure of XML to express queries against data. The XQuery Scripting Extension builds on this base by adding procedural constructs, including basic statements, control flow, and user-defined procedures to XQuery.

XQSE is therefore a superset of XQuery, extending it with additional features that enable you to create richer and more complex data services while working within the context of XML and XQuery. You can think of XQSE as an extension to XQuery in the same way as Oracle PL/SQL is an extension of SQL.

## Prolog and Query Body

XQSE extends XQuery by adding procedural capabilities to the declarative query capabilities of XQuery. In XQuery, a query consists of a prolog followed by an XQuery expression. The prolog of a query sets up the environment for the expression by defining namespaces, external variables, and functions, among other information.

In XQSE, a prolog can also contain definitions of procedures and XQSE functions which are, respectively, side-effecting and non-side-effecting callable units of execution written in XQSE.

The following shows the grammar of the XQSE prolog and query body:

```
Prolog ::= ((DefaultNamespaceDecl | Setter | NamespaceDecl | Import)
   Separator)* ((VarDecl | FunctionDecl | ProcedureDecl |
   XQSEFunctionDecl | OptionDecl) Separator)*

QueryBody ::= Expr | Block
```

In XQSE, the body of a top-level query can be either an XQuery expression or an XQSE block. A block is a sequence of statements that are executed sequentially.

## Procedure Declaration

XQSE enables you to declare a procedure in the prolog of an ALDSP data service. An XQSE procedure is similar to an XQuery function, but unlike a function, a procedure can have one or more side effects. Another difference is that XQuery functions are declarative; the body of an XQuery function is an XQuery expression.

The body of an XQSE procedure, in contrast, consists of a block, which is a list of statements executed in sequential order when the procedure is called. Alternatively, as in XQuery, you can declare a procedure as external, in which case it does not have a body and is implemented by ALDSP by importing procedures from external data sources such as relational stored procedures or Web services.

The following shows the grammar of the XQSE procedure declaration:

```
ProcedureDecl ::= "declare" "procedure" QName "(" ParamList? ")"
   ("as" SequenceType)? (Block | ("external") )
```

A procedure may, but is not required to, return a value. Individual XQSE statements do not have return values by themselves, so a procedure returns a value only when an explicit return statement is included in the body of procedure. If the return type of a procedure is not specified, its return value is of type `item()*` by default.

**Note:** Since returning a value is optional, a return statement is not required in the body of a procedure. In the absence of a return statement, the return value of a procedure is the empty sequence and its return type is `empty()`.

You can use recursion in procedures. This is another difference between XQuery functions and XQSE procedures in ALDSP.

Example:

```
(: Procedure to delete an employee given just their employee ID :)
(: Calls the default delete function on the data service after retrieving
the employee object using the ID :)
declare procedure tns:deleteByEmployeeID($id as xs:string?) as empty() {
   declare $emp as element(empl:Employee)? := tns:getByEmployeeID($id);
   tns:delete($emp);
};
```

## XQSE Function Declaration

XQSE extends the XQuery function declaration syntax to enable you to declare XQSE-based functions in addition to procedures. An XQSE function is essentially a read-only procedure written in XQSE with no side effects.

As with a procedure, the body of an XQSE function consists of a block, which is a list of statements. The following shows the grammar of the XQSE function declaration:

```
XQSEFunctionDecl ::= "declare" xqse "function" QName "(" ParamList? ")"
   ("as" SequenceType)? (Block | ("external") )
```

Since an XQSE function does not have any side-effects, you can call it from within an XQuery expression anywhere that a normal XQuery function can be called.

Since XQuery functions are declarative and therefore amenable to compile-time query optimization, you should write data service functions using XQuery instead of XQSE where possible. However, it is sometimes necessary (or at least conceptually more convenient) to express certain read-only computations procedurally. XQSE functions are appropriate in these cases.

For example, you could use an XQSE function to perform calculations that would otherwise require the use of tail recursion in XQuery. This is necessary since ALDSP does not permit the use of recursion in XQuery functions.

Example:

```
(: Procedure to compute the level of an employee in the org tree :)
declare xqse function tns:distanceFromTop($id as xs:string?) as
xs:integer? {
   declare $mgrCnt as xs:integer := 0;
   declare $curEmp as element(empl:Employee)? :=
      tns:getByEmployeeID($id);
   declare $mgrId as xs:string? := fn:data($curEmp/ManagerID);
   if (fn:empty($curEmp)) then return value ();
   while (fn:not(fn:empty($mgrId))) {
      set $mgrCnt := $mgrCnt + 1;
      set $curEmp := tns:getByEmployeeID($mgrId);
      set $mgrId  := fn:data($curEmp/ManagerID);
   };
   return value ($mgrCnt);
};
```

## Value Statement and Procedure Call

XQSE offers the value statement and procedure call statement to distinguish between function and procedure calls.

**Note:** You can call a function wherever an expression can be used, but procedures can only be called in certain parts of XQSE because they include side effects.

The following shows the grammar of the XQSE value statement and procedure call statement:

```
ValueStatement := ExprSingle | ProcedureCall
ProcedureCall ::= FunctionCall
Statement := SimpleStatement | BlockStatement
SimpleStatement ::= SetStatement | IfStatement | ReturnStatement |
   ProcedureCall
```

## Block

An XQSE block contains a list of statements. You can use a block to declare mutable variables (using declare clauses) and manipulate those variables in subsequent statements, which are executed in sequential order.

The following shows the grammar of the XQSE `block` statement:

```
Block ::= {" ((BlockDecl ";")* StatementSequence "}"
BlockDecl ::= "declare" "$" VarName TypeDeclaration?
   (":=" ValueStatement)?
StatementSequence := ((SimpleStatement ";") | (BlockStatement (";")?))*
BlockStatement := WhileStatement | IterateStatement | TryStatement |
   Block
```

While XQuery expressions have values, statements do not (with the exception of the `return` statement, described in "Return Statement" on page 6-7.). Therefore, a block does not have a return value since a block is itself a compound statement.

Every variable in a block must be declared before it can be used. If you declare a variable without explicitly specifying a type, the variable will have a default type of `item()*`.

You also need to initialize declared variables using a `set` statement or as part of the `BlockDecl` before they can be used. Referencing uninitialized variables (not appearing on the left hand side of an assignment statement) results in an error.

**Note:** Variables in a block are mutable. Unlike `let` and `for` variables that appear in XQuery expressions, which are immutable bindings of names to values, variables in an XQSE block are assignable (similar to variables in Java and C++, among other languages).

You can define nested blocks, in which case regular scoping rules apply. For example, a variable with a specific fully-qualified name declared in an inner block will shadow (redefine and hide) variables in a containing outer block that has the identical fully-qualified name.

Example:

```
(: Procedure to compute the level of an employee in the org tree :)
declare xqse function tns:distanceFromTop($id as xs:string?)
   as xs:integer? {
   declare $mgrCnt as xs:integer := 0;
   declare $curEmp as element(empl:Employee)? :=
      tns:getByEmployeeID($id);
   declare $mgrId as xs:string? := fn:data($curEmp/ManagerID);
   if (fn:empty($curEmp)) then return value ();
   while (fn:not(fn:empty($mgrId))) {
      set $mgrCnt := $mgrCnt + 1;
      set $curEmp := tns:getByEmployeeID($mgrId);
      set $mgrId  := fn:data($curEmp/ManagerID);
   }
   return value ($mgrCnt);
};
```

## Set Statement

The XQSE `set` statement sets the variable `VarName` to the value specified by `ValueStatement`. The following shows the grammar of the XQSE `set` statement:

```
SetStatement ::= "set" "$" VarName ":=" ValueStatement
```

Before using the `set` statement, you must first declare the variable `VarName` using a `declare` statement. Only variables declared in this way are mutable and can therefore be changed using the `set` statement.

**Note:** The `set` statement has copy semantics. Consider the following instance:

```
set $z := ($x, $y)
```

If `$x` and `$y` are mutable variables and `$x` and `$y` are later changed, $z retains the originally copied values of `$x` and `$y`.

Example:

```
(: Procedure to compute the level of an employee in the org tree :)
declare xqse function tns:distanceFromTop($id as xs:string?)
      as xs:integer? {
   declare $mgrCnt as xs:integer := 0;
   declare $curEmp as element(empl:Employee)? :=
      tns:getByEmployeeID($id);
   declare $mgrId as xs:string? := fn:data($curEmp/ManagerID);
   if (fn:empty($curEmp)) then return value ();
   while (fn:not(fn:empty($mgrId))) {
      set $mgrCnt := $mgrCnt + 1;
      set $curEmp := tns:getByEmployeeID($mgrId);
      set $mgrId  := fn:data($curEmp/ManagerID);
   };
   return value ($mgrCnt);
};
```

## While Statement

The XQSE `while` statement loops and performs the actions in the block while the effective boolean value of the condition evaluates to true. The following shows the grammar of the XQSE `while` statement:

```
WhileStatement ::= "while" "(" Expr ")" Block
```

The while statement reevaluates the condition expression before each loop. Typically, the condition depends upon a mutable variable that is manipulated in the block. The loop therefore terminates when code in the block causes the effective boolean value of the condition to cease being true.

Example:

```
(: Procedure to compute the level of an employee in the org tree :)
declare xqse function tns:distanceFromTop($id as xs:string?)
      as xs:integer? {
   declare $mgrCnt as xs:integer := 0;
   declare $curEmp as element(empl:Employee)? :=
      tns:getByEmployeeID($id);
   declare $mgrId as xs:string? := fn:data($curEmp/ManagerID);
   if (fn:empty($curEmp)) then return value ();
   while (fn:not(fn:empty($mgrId))) {
      set $mgrCnt := $mgrCnt + 1;
      set $curEmp := tns:getByEmployeeID($mgrId);
      set $mgrId  := fn:data($curEmp/ManagerID);
   };
   return value ($mgrCnt);
};
```

## Return Statement

The XQSE `return` statement computes the expression represented by `ValueStatement` and returns the resulting value while exiting from the current procedure.

The following shows the grammar of the XQSE `return` statement:

```
ReturnStatement ::= "return" "value" ValueStatement
```

In the special case where a block containing a return statement is the body of the main query, the return statement returns the value to the invoking environment.

Example:

```
(: Procedure to compute the level of an employee in the org tree :)
declare xqse function tns:distanceFromTop($id as xs:string?)
      as xs:integer? {
   declare $mgrCnt as xs:integer := 0;
   declare $curEmp as element(empl:Employee)? :=
      tns:getByEmployeeID($id);
   declare $mgrId as xs:string? := fn:data($curEmp/ManagerID);
   if (fn:empty($curEmp)) then return value ();
   while (fn:not(fn:empty($mgrId))) {
      set $mgrCnt := $mgrCnt + 1;
      set $curEmp := tns:getByEmployeeID($mgrId);
      set $mgrId  := fn:data($curEmp/ManagerID);
   };
   return value ($mgrCnt);
};
```

## Iterate Statement

XQSE offers an `iterate` statement that is equivalent to the XQuery `for` clause and enables you to perform data-driven looping over a block of XQSE statements. This enables you to iterate through the result of an XQuery expression, for example.

The following shows the grammar of the XQSE `iterate` statement:

```
IterateStatement ::= "iterate" "$" VarName PositionalVar? "over"
   ValueStatement Block
```

The iteration variable `VarName` is bound to each item in the sequence produced by evaluating `ValueStatement`, which can be either an XQuery expression or a procedure call. Optionally, the `PositionalVar` variable represents the index of the current item in the sequence.

**Note:** The `VarName` and `PositionalVar` variables are both mutable, though it is not advisable that you exploit this capability.

Examples:

```
(: Procedure to allow only updates that don't violate the salary change
business rules :)
declare procedure tns:updateChecked($changedEmps as
      changed-element(empl:Employee)*) {
   iterate $sourceEmp over $changedEmps {
      if (tns:invalidSalaryChange($sourceEmp)) then
         fn:error(xs:QName("INVALID_SALARY_CHANGE"), ":
            Salary change exceeds the limit.");
   };
   tns:update($changedEmps);
};


(: Procedure to perform "lite ETL", copying and transforming data from
one source to another :)
declare procedure tns:copyAllToEMP2() as xs:integer {
   declare $backupCnt as xs:integer := 0;
   declare $emp2 as element(emp2:EMP2)?;
   iterate $emp1 over ens1:getAll() {
      set $emp2 := tns:transformToEMP2($emp1);
      emp2:createEMP2($emp2);
      set $backupCnt := $backupCnt + 1;
   }
   return value ($backupCnt);
};
```

## Try Statement

The `try` statement enables you to perform procedural error handling in XQSE, such as those raised by the XQuery `fn:error` function. The `try` statement works in much the same way as traditional try/catch statements in languages such as Java or C++.

The following shows the grammar of the XQSE `try` statement:

```
TryStatement ::= "try" Block CatchClauseStatement+

CatchClauseStatement ::= catch "(" NameTest ("into" VarNameExpr (("," 
VarNameExpr)? "," VarNameExpr)?)? ")" Block
```

XQSE enables you to catch all XQuery errors. XQuery errors have an associated `QName`, enabling you to use the XQuery `NameTest` to restrict the errors handled by a specific `catch` clause. In addition, the following variables in the `catch` clause work similarly to the arguments of the XQuery `fn:error` function:

- `$error—xs:QName`
- `$description—xs:string`
- `$error-object—item()*`

Similar to exceptions in other languages, you can re-throw errors in XQSE using the `fn:error` function. When doing so, you need to ensure that all components of the error, including the name, description, and error-object, are properly passed to the new `fn:error` call.

Example:

```
(: Procedure to create a replicated employee and return an appropriately 
specific error message if it fails :)
declare procedure tns:create($newEmps as element(empl:Employee)*)
      as element(empl:ReplicatedEmployee_KEY)* {
   iterate $newEmp over $newEmps {
      declare $newEmp2 as element(emp2:EMP2)? :=
         bns:transformToEMP2($newEmp);
      try { tns:createEmployee($newEmp); }
      catch (* into $err, $msg) {
         fn:error(xs:QName("PRIMARY_CREATE_FAILURE"),
         fn:concat("Create failed on primary copy due to: ", $err,
            $msg));
      };
```

```
            try { emp2:createEMP2($newEmp2); }
            catch (* into $err, $msg) {
               fn:error(xs:QName("SECONDARY_CREATE_FAILURE"),
               fn:concat("Create failed on backup copy due to: ", $err,
                  $msg));
            };
         }
      };
```

## If Statement

XQSE offers an `if` statement that is equivalent to the XQuery `IfExpr` expression. The XQSE `if` statement differs from the XQuery `IfExpr` expression in the following ways:

- The XQSE `if` statement is a control flow statement, and does not return a value.
- The else clause is optional in an XQSE `if` statement.

The following shows the grammar of the XQSE `if` statement:

```
IfStatement ::= "if" "(" Expr ")" "then" Statement ("else" Statement)?
```

Example:

```
(: Procedure to compute the level of an employee in the org tree :)
declare xqse function tns:distanceFromTop($id as xs:string?)
      as xs:integer? {
   declare $mgrCnt as xs:integer := 0;
   declare $curEmp as element(empl:Employee)? :=
      tns:getByEmployeeID($id);
   declare $mgrId as xs:string? := fn:data($curEmp/ManagerID);
   if (fn:empty($curEmp)) then return value ();
   while (fn:not(fn:empty($mgrId))) {
      set $mgrCnt := $mgrCnt + 1;
      set $curEmp := tns:getByEmployeeID($mgrId);
      set $mgrId  := fn:data($curEmp/ManagerID);
   };
   return value ($mgrCnt);
};
```

## Changed Element

The XQSE language extends the XQuery data model with information about elements that have been updated and resubmitted to ALDSP. This enables XQSE to support SDO client updates along with associated server-side update logic.

An XML node that contains changes has the XQSE type `changed-element`. The following shows the grammar of the XQSE `changed-element` type:

```
ItemType := AtomicType | KindTestType | <"item" "(" ")"> |
   ChangedElementType
```

```
ChangedElementType := "changed-element" "(" ElementNameOrWildcard ")"
```

XQSE provides two built-in functions, `fn-bea:old-value` and `fn-bea:current-value`, to access the pre-update and post-update contents of the changed XML node respectively.

**Note:** You can only pass instances of `changed-element` into XQSE as arguments to procedures and functions. Instances of `changed-element` cannot be created or incrementally modified within XQSE.

You can also use instances of `changed-element` in variable declarations and assignments.

Example:

```
(: function to determine whether or not a given salary change is legal
according to business rules :)
declare function tns:invalidSalaryChange($emp as
      changed-element(empl:Employee)) as xs:boolean {
   let $newSalary := fn:data(fn-bea:current-value($emp)/Salary)
   let $oldSalary := fn:data(fn-bea:old-value($emp)/Salary)
   return (100.0 * fn:abs($newSalary - $oldSalary) div $oldSalary)
      gt 10.0
};
```

## XQSE Grammar Summary

The following summarizes the XQSE grammar:

```
Prolog ::= ((DefaultNamespaceDecl | Setter | NamespaceDecl | Import)
Separator)*

((VarDecl | FunctionDecl | ProcedureDecl | XQSEFunctionDecl| OptionDecl)
Separator)*

XQSEFunctionDecl ::= "declare" xqse "function" QName "(" ParamList? ")"
("as" SequenceType)? (Block | ("external") )

ProcedureDecl ::= "declare" "procedure" QName "(" ParamList? ")" ("as"
SequenceType)? (Block | ("external") )

QueryBody ::= Expr | Block

Statement := SimpleStatement | BlockStatement

SimpleStatement ::= SetStatement | IfStatement | ReturnStatement |
ProcedureCall

BlockStatement := WhileStatement | IterateStatement | TryStatement |
Block

ValueStatement := ExprSingle | ProcedureCall

ReturnStatement ::= "return" "value" ValueStatement

Block ::= {" ((BlockDecl ";")* StatementSequence "}"

StatementSequence := ((SimpleStatement ";") | (BlockStatement (";")?))*

BlockDecl ::= "declare" "$" VarName TypeDeclaration? (":="
ValueStatement)?

SetStatement ::= "set" "$" VarName ":=" ValueStatement

WhileStatement ::= "while" "(" Expr ")" Block

IterateStatement ::= "iterate" "$" VarName PositionalVar? "over"
ValueStatement Block

ProcedureCall ::= FunctionCall

TryStatement ::= "try" Block CatchClauseStatement+

CatchClauseStatement ::= catch "(" NameTest ("into" VarNameExpr ((","
VarNameExpr)? "," VarNameExpr)?)? ")" Block

IfStatement ::= "if" "(" Expr ")" "then" Statement ("else" Statement)?

ItemType := AtomicType | KindTestType | <"item" "(" ")"> |
ChangedElementType

ChangedElementType := "changed-element" "(" ElementNameOrWildcard ")"
```

# XQuery-SQL Mapping Reference

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

- Core RDBMS Support:
  - IBM DB2/NT 8 (and higher)
  - Microsoft SQL Server 2000 (and higher)
  - Oracle 8.1.x
  - Oracle 9.x, 10.x
  - Sybase 12.5.2 (and higher)
  - PointBase 5.1
  - Teradata V2R5 (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 (and higher)

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

Table G-1 lists supported data type mappings for IBM DB2/NT 8.

**Table G-1 IBM DB2 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

Table G-2 lists functions and operators that are pushed down to IBM DB2/NT8 RDBMSs. See "fn-bea:sql-like" on page 2-26 for details about two-argument and three-argument versions of the fn-bea:sql-like() function.

**Table G-2 IBM DB2 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], fn-bea:repeat[6], fn-bea:pad-left[6], fn-bea:pad-right[6] |
| Datetime comparisons | =, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge on xs:dateTime, xs:date, xs:time |

**Table G-2 IBM DB2 Functions and Operators (Continued)**

| | |
|---|---|
| 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.

## Cast Operation Pushdown

Table G-3 lists supported cast operations.

**Table G-3 IBM DB2 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 |

**Table G-3 IBM DB2 Cast Operations (Continued)**

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

## Other SQL Generation Capabilities

Table G-4 lists common query patterns that can be pushed down. See also "Common Query Patterns".

**Table G-4 IBM DB2 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 (and higher)

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

Table G-5 lists supported data type mappings for Microsoft SQL Server 2000.

**Table G-5 SQL Server 2000 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 G-5 SQL Server 2000 Data Type Mapping (Continued)**

| | |
|---|---|
| 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 G-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 G-6 for details).
- xdt:yearMonthDuration (see "Datetime Arithmetic" functions in Table G-6 for details).

## Function and Operator Pushdown

Table G-6 lists functions and operators that are pushed down to Microsoft SQL Server 2000. (See "fn-bea:sql-like" on page 2-26 for details about two-argument and three-argument versions of the fn-bea:sql-like() function.)

**Table G-6 SQL Server 2000 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, fn-bea:repeat, fn-bea:pad-left, fn-bea:pad-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 |

**Table G-6 SQL Server 2000 Function and Operator Pushdown (Continued)**

| | |
|---|---|
| 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

Table G-7 lists supported cast operations.

**Table G-7 SQL Server 2000 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

Table G-8 lists common query patterns that can be pushed down. (See “Common Query Patterns” for details.)

**Table G-8 SQL Server 2000 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 G-9 lists supported data type mappings for Oracle 8.1.x (Oracle 8*i*).

**Table G-9 Oracle 8.1.x 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 G-10 for functions and operators that use xs:date)
- xdt:yearMonthDuration (see "Datetime Arithmetic" in Table G-10 for details)
- xs:integer subtypes (see "Numeric ..." functions and operators in Table G-10 for details)

## Function and Operator Pushdown

Table G-10 lists functions and operators that are pushed down. See "fn-bea:sql-like" on page 2-26 for details about two-argument and three-argument versions of the fn-bea:sql-like() function.

**Table G-10 Oracle 8.1.x 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, fn-bea:repeat, fn-bea:pad-left, fn-bea:pad-right |
| Datetime comparisons | =, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge on xs:dateTime, xs:date, xdt:yearMonthDuration |

**Table G-10 Oracle 8.1.x Function and Operator Pushdown (Continued)**

| | |
|---|---|
| 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 |
| 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

Table G-11 lists supported cast operations.

**Table G-11 Oracle 8.1.x Cast Operation Pushdown**

| Source XQuery Type | Target XQuery Type |
|---|---|
| numeric | xs:string |
| numeric | xs:decimal |

**Table G-11 Oracle 8.1.x Cast Operation Pushdown (Continued)**

| | |
|---|---|
| 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

Table G-12 lists common query patterns that can be pushed down. See "Common Query Patterns" for details.

**Table G-12 Oracle 8.1.x 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 |

**Table G-12 Oracle 8.1.x Other SQL Generation Capabilities (Continued)**

| | |
|---|---|
| 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 G-13 lists supported data type mappings for Oracle 9.x and 10.x.

**Table G-13 Oracle 9.x, 10.x 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 |

**Table G-13 Oracle 9.x, 10.x Data Type Mapping (Continued)**

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

## Function and Operator Pushdown

Table G-14 lists functions and operators that are pushed down to Oracle 9.x and 10.x. See "fn-bea:sql-like" on page 2-26 for details about two-argument and three-argument versions of the fn-bea:sql-like() function.

**Table G-14 Oracle 9.x, 10.x 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, fn-bea:repeat, fn-bea:pad-left, fn-bea:pad-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 |

**Table G-14 Oracle 9.x, 10.x Function and Operator Pushdown (Continued)**

| | |
|---|---|
| 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

Table G-15 lists cast operations that can be pushed down.

**Table G-15 Oracle 9.x, 10.x 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

Table G-16 lists common query patterns that can be pushed down. (See "Common Query Patterns" for details.)

**Table G-16 Oracle 9.x, 10.x 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 |

## 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

Table G-17 lists supported data type mappings for Sybase 12.5.2.

**Table G-17 Sybase 12.5.2 Data Type Mapping**

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

**Table G-17 Sybase 12.5.2 Data Type Mapping (Continued)**

| | |
|---|---|
| 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

Table G-18 lists functions and operators that are pushed down to base RDBMSs. (See "fn-bea:sql-like" on page 2-26 for details about two-argument and three-argument versions of the fn-bea:sql-like( ) function.)

**Table G-18 Sybase 12.5.2 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, fn-bea:repeat, fn-bea:pad-left, fn-bea:pad-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 |

**Table G-18 Sybase 12.5.2 Function and Operator Pushdown (Continued)**

| | |
|---|---|
| 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 Table G-19 lists supported cast operations.

**Table G-19 Sybase 12.5.2 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

Table G-20 lists common query patterns that can be pushed down. See "Common Query Patterns" for details.

**Table G-20 Sybase 12.5.2 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 |

## PointBase 5.1

The tables in this section identify all data type and other mappings that the XQuery engine generates or supports for PointBase 5.1.

### Data Type Mapping

Table G-21 lists supported data type mappings for PointBase 5.1.

**Table G-21 PointBase 5.1 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

Table G-22 lists functions and operators that are pushed down to PointBase. See “fn-bea:sql-like” on page 2-26 for details about two-argument and three-argument versions of the fn-bea:sql-like() function.

**Table G-22 PointBase 5.1 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

Table G-19 lists supported cast operations.

**Table G-23 PointBase 5.1 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

Table G-24 lists common query patterns that can be pushed down. (See "Common Query Patterns" for details.)

**Table G-24 PointBase 5.1 Other SQL Generation Capabilities**

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

## Teradata V2R5 (and higher)

The tables in this section identify all data type and other mappings that the XQuery engine generates or supports for Teradata V2R5 (and higher).

### Data Type Mapping

Table G-25 lists supported data type mappings for Teradata V2R5.

**Table G-25 Teradata V2R5 Data Type Mapping**

| Teradata Data Type | XQuery Type |
|---|---|
| BYTE | xs:hexBinary |
| BYTEINT | xs:short |
| CHAR | xs:string |
| DATE | xs:date |
| DECIMAL(p,s) (NUMERIC) | xs:decimal (if s > 0),<br>xs:integer (if s == 0) |
| FLOAT (REAL, DOUBLE PRECISION) | xs:double |
| INTEGER | xs:int |
| LONG VARCHAR | xs:string |
| SMALLINT | xs:short |
| TIME | xs:time |
| TIMESTAMP | xs:dateTime |
| VARBYTE | xs:hexBinary |
| VARCHAR | xs:string |

## Function and Operator Pushdown

Table G-26 lists functions and operators that are pushed down to Teradata. See "fn-bea:sql-like" on page 2-26 for details about two-argument and three-argument versions of the fn-bea:sql-like() function.

**Table G-26 Teradata V2R5 Function and Operator Pushdown**

| 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 | concat[3], upper-case[4], lower-case[4], contains[5], starts-with[5], ends-with[5], fn-bea:sql-like(2,3)[5] |
| Datetime comparisons | =, !=, <, <=, >, >=, eq, ne, lt, le, gt, ge on xs:dateTime, xs:date, xs:time |
| Aggregate | min, max, sum, avg, count, count(distinct-values) |
| Other | empty, exists |

1. All numeric types
2. Only xs:decimal, xs:float, and xs:double
3. CHAR or VARCHAR SQL data types only for all arguments
4. CHAR or VARCHAR SQL data type only for first argument
5. First argument must be CHAR or VARCHAR SQL data type, second argument must be a constant or parameter

## Cast Operation Pushdown

Cast operations are not pushed down.

## Other SQL Generation Capabilities

Table G-27 lists common query patterns that can be pushed down. (See "Common Query Patterns" for details.)

**Table G-27 Teradata V2R5 Other SQL Generation Capabilities**

| Feature | Description |
|---|---|
| Inner joins | yes, SQL-92 syntax |
| Outer joins | yes |
| 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 |
| Distinct pattern | yes |
| Trivial aggregate pattern pushdown | yes (using GROUP BY constant) |
| If-then-else pushdown | yes |
| Subsequence pushdown | yes |
| SQL Exit query composition (pushdown on top of SQL Exit) | yes |
| Runtime connection management | no connection sharing |

## 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, AquaLogic Data Services Platform 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 Service Annotations" in the *Data Services Developer's Guide* for more information.

## Database Capabilities Information

The database capabilities listed in Table G-28 are obtained from the operative JDBC driver and stored as properties in the `.ds` (data service) definition file.

**Table G-28 Database Properties Derived from the JDBC Driver**

| 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 AquaLogic 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 "Understanding Data Service Annotations" in the *Data Services Developer's Guide* for more information about these properties). The XQuery engine (metadata importer sub-system) uses the specific quote property (see Table G-29) 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 G-29 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, the XQuery engine first checks the JDBC typecode. If the typecode has a corresponding XQuery type, AquaLogic Data Services Platform uses the matching native type name. If no matching typecode or type name is available, the column is ignored. Table G-30 shows this mapping.

**Table G-30 Base Platform 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 |
| 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] |

**Table G-30 Base Platform Data Type Mapping (JDBC<–>XQuery Equivalents) (Continued)**

| JDBC Data Type | Typecode | XQuery Data Type |
|---|---|---|
| TIMESTAMP[4] | 93 | xs:dateTime[2] |
| TINYINT | -6 | xs:short |
| VARBINARY | -3 | xs:hexBinary |
| VARCHAR | 12 | xs:string |
| OTHER | 1111 | AquaLogic Data Services Platform 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.

Table G-31 lists functions and operators that are pushed down to base RDBMSs. See "fn-bea:sql-like" on page 2-26 for details about two-argument and three-argument versions of the fn-bea:sql-like() function.

**Table G-31 Base Platform 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 |

**Table G-31 Base Platform Functions and Operators (Continued)**

| Group | Functions and Operators |
|---|---|
| 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

Table G-32 displays other SQL Pushdown capabilities, as discussed in "Common Query Patterns" on page 3-15.

**Table G-32 Base Platform 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 |

Table G-32 Base Platform SQL Generation Capabilities (Continued)

| Query | Supported |
|---|---|
| Group by | yes (by column only) |
| Distinct pattern | yes |
| Trivial aggregate pattern | no |
| Direct SQL composition | no |