BEA Liquid Data for WebLogic™

XQuery Reference Guide

Version 8.1
Document Date: December 2003
Revised: December 2003

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

About This Document

1. XQuery and XML Specification Implementation

2. Understanding XQuery in Liquid Data

3. Functions Reference

4. Supported Data Types

Index

About This Document

This document provides reference material for the XQuery language implemented in BEA Liquid Data for WebLogic. It describes the emerging XQuery standard from the World Wide Web Consortium (W3C) and includes reference material for the Liquid Data XQuery implementation.

This document covers the following topics:

- Chapter 1, "XQuery and XML Specification Implementation," indroduces the W3C XQuery standard and lists the version supported by Liquid Data.
- Chapter 2, "Understanding XQuery in Liquid Data," describes a query written in the XQuery language.
- Chapter 3, "Functions Reference," provides information about complete reference of the World Wide Web (W3C) functions supported in Liquid Data.
- Chapter 4, "Supported Data Types," is a reference list of data types supported in Liquid Data.

What You Need to Know

Users creating queries with Data View Builder should have a high-level understanding of XML, XML schemas, and declarative database query languages. Users creating ad hoc queries to run in a Liquid Data environment should have the additional skill of being proficient in the W3C standard XQuery syntax.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA home page, click on Product Documentation or go directly to the "e-docs" Product Documentation page at e-docs.bea.com.

How to Print the Document

You can print a copy of this document from a Web browser, one file at a time, by using the File—>Print option on your Web browser.

A PDF version of this document is also available on the Liquid Data documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF using Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDF files, open the Liquid Data documentation Home page, click PDF files and select the document you want to print.

If you do not have the Adobe Acrobat Reader, you can obtain a free version from the Adobe Web site at www.adobe.com.

Related Information

For more information about XQuery and XML Query languages, see the World Wide Web Consortium (W3C) Web site at http://www.w3.org/.

Contact Us!

Your feedback on the BEA Liquid Data documentation is important to us. Send us e-mail at **docsupport@bea.com** if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the Liquid Data documentation.

In your e-mail message, please indicate that you are using the documentation for the BEA Liquid Data for WebLogic 1.0 release.

If you have any questions about this version of Liquid Data, or if you have problems installing and running Liquid Data, contact BEA Customer Support through BEA WebSupport at **www.bea.com**. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address

- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `()`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*

Convention	Item
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Example*s: LPT1 SIGNON OR
{ }	Indicates a set of choices in a syntax line. The braces themselves should never be typed.
[]	Indicates optional items in a syntax line. The brackets themselves should never be typed. *Example*: `buildobjclient [-v] [-o name] [-f file-list]... [-l file-list]...`
|	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.
. . .	Indicates one of the following in a command line: • That an argument can be repeated several times in a command line • That the statement omits additional optional arguments • That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. *Example*: `buildobjclient [-v] [-o name] [-f file-list]... [-l file-list]...`
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

For the conventions on reading the XQuery syntax diagrams, see "Reading the XQuery Syntax Diagrams" on page 2-30.

CHAPTER 1

XQuery and XML Specification Implementation

This chapter describes the version of the XQuery specification implemented in BEA Liquid Data for WebLogic. It also briefly describes the XQuery specification and provides links to more information about XQuery.

The following topics are covered:

- Supported XQuery and XML Schema Versions In Liquid Data
- W3C XML and XQuery
- XQuery Use in Liquid Data and the Data View Builder
- Learning More About the XQuery Language

Supported XQuery and XML Schema Versions In Liquid Data

This section lists the XQuery and XML specifications with which Liquid Data complies.

XQuery

The Liquid Data XQuery implemenation is based on the following XQuery specification:

```
http://www.w3.org/TR/2001/WD-xquery-20011220
```

XQuery Functions and Operators

Liquid Data implements functions and operators based on the following specification:

```
http://www.w3.org/TR/2002/WD-xquery-operators-20020430
```

For a list and the syntax of the functions and operators implemented in Liquid Data, see "Functions Reference" on page 3-1.

XML Schema

XML schemas are used in Liquid Data to describe the hierarchical structure of the various data sets with which you are working. For XML Schema specifications and information, see the following URL:

```
http://www.w3.org/XML/Schema#dev
```

W3C XML and XQuery

XML is evolving from a W3C specification for a markup language to an entire family of specifications and technologies. The W3C has chartered working groups focused on creating, among other things, specifications for schemas and a query language. The evolving query language is *XQuery*, which gives XML developers a structured solution for accessing and querying XML data. The W3C Query Working Group used a formal approach by defining a data model as the basis for XQuery. XQuery uses a simple type system and supports query optimization. It is statically typed, which supports compile-time type checking.

Whereas SQL is a well-known query language for querying relational databases, XQuery is a query language for querying XML-based information. Developers who are familiar with SQL will find XQuery to be a natural next step.

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

For information about the syntax of XQuery in Liquid Data, see "Understanding XQuery in Liquid Data" on page 2-1.

XQuery Use in Liquid Data and the Data View Builder

Liquid Data models various types of data sources as XML schemas. You can combine elements and attributes of the schemas in a query written in the XQuery language. The Liquid Data Server then executes the query and returns the results.

Once you have configured Liquid Data access to the data sources you want to use (relational databases, Web Services, application views, data views, and so on), you can query the data by issuing queries written in XQuery to Liquid Data, and the Liquid Data Server will fetch the data from the underlying data sources and return the query results.

The Data View Builder provided with Liquid Data is a tool that generates queries in the XQuery language. You can combine data from multiple sources by dragging-and-dropping the XML schemas and a full complement of functions to generate queries in XQuery. The Data View Builder also allows you to test, save, and deploy queries. For details on the Data View Builder, see *Building Queries and Data Views*.

Learning More About the XQuery Language

You can learn more about XML schemas on the W3C Web site at http://www.w3.org/XML/Schema and http://www.w3.org/2001/12/xmlbp/xml-schema-wg-charter.html.

You can learn more about the standard on the W3C Web site at http://www.w3.org/TR/xquery/.

For a comprehensive list of relevant XQuery references, see "XQuery Links" in the Liquid Data *Product Overview*.

For the syntax of XQuery in Liquid Data, see "Understanding XQuery in Liquid Data" on page 2-1.

CHAPTER 2

Understanding XQuery in Liquid Data

This chapter describes the syntax for queries written in the Liquid Data implementation of the XQuery language. The Liquid Data XQuery syntax is based on the syntax described in the December 2001 draft specification "XQuery 1.0: An XML Query Language" from the W3C:

`http://www.w3.org/TR/2001/WD-xquery-20011220/#nt-bnf`

For more information on the versions of the XQuery and XML Schema specifications supported in Liquid Data, see "XQuery and XML Specification Implementation" on page 1-1.

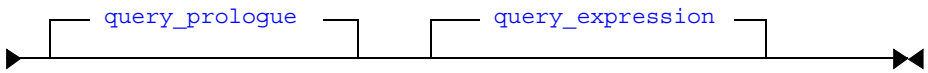
The following topics are covered:

XQuery Syntax in Liquid Data

This section describes the syntax of an XQuery in Liquid Data. The syntax is described in blocks, and each block is defined in its own subsection. The syntax is shown as *railroad diagrams*. For details on how to read the railroad diagrams, see "Reading the XQuery Syntax Diagrams" on page 2-30.

Figure 2-1 shows the basic syntax for an XQuery.

Figure 2-1 Basic XQuery Syntax Diagram

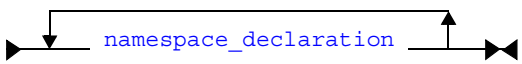


An XQuery query expression is typically a combination of XML markup and query logic. The XQuery language allows you to mix the query logic with literal XML markup in much the same way as you can combine HTML and Java on a Java Server Page (JSP).

query_prologue

The query prologue includes zero or more namespace declarations and has the syntax shown in Figure 2-2.

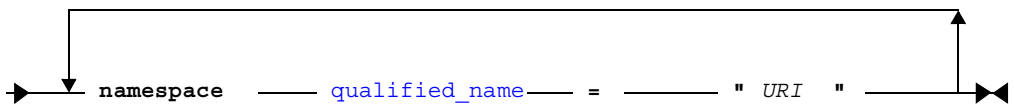
Figure 2-2 Query Prologue Syntax Diagram



namespace_declaration

A namespace declaration defines XML namespaces used in the query. Figure 2-3 shows the syntax.

Figure 2-3 Namespace Declaration Syntax Diagram



where *URI* is a valid URI string.

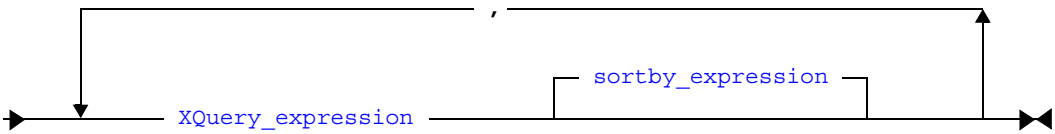
The following example shows a valid namespace declaration:

```
namespace myspace = "http://mycorp.com/name"
```

query_expression

Query expressions in XQuery specify the results of a query by iterating over data, applying functions to expressions, specifying XML markup, and specifying any logic needed to get the desired result. Figure 2-4 shows the query expression syntax in Liquid Data.

Figure 2-4 Query Expression Syntax Diagram



Usage Notes

An XQuery expression can be one of many types of expressions. For details and syntax of the different types of XQuery expressions, see "XQuery Expressions" on page 2-7.

One of the main building blocks of an XQuery is the FLWR expression, as described in "FLWR Expression" on page 2-9.

Use curly braces ({}) to surround a query expression if the containing query includes XML markup (see "XML Markup Expression" on page 2-7) directly before or after the query expression.

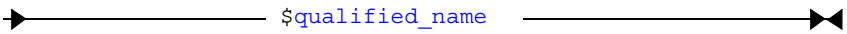
The optional `sortby` expression (see "sortby_expression" on page 2-4) is used to sort the data from the XQuery expression.

If you separate query expressions with a comma (`,`), the results are concatenated together. Depending on the structure of your query, this can form the basis for a union-all operation.

variable_definition

XQuery variable definitions begin with the dollar sign (`$`) character and have the syntax shown in Figure 2-5.

Figure 2-5 Variable Definition Syntax Diagram



For examples of variables, see "Variables" on page 2-17.

qualified_name

A qualified_name is a qualified name string in XML. The string must begin with a letter and can have any alpha-numeric character following the letter. It can also use the underscore (_), hyphen (-), and period (.) characters. For more details on qualified name strings (QName) in XML, see:

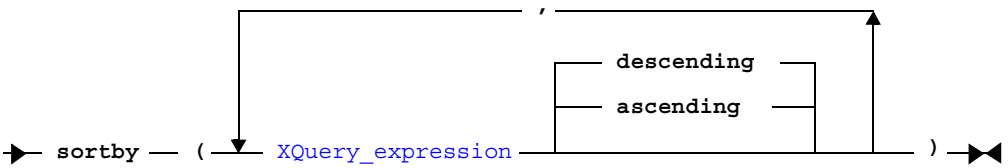
`http://www.w3.org/TR/REC-xml-names/#NT-QName`

Note: For improved query readability, do not qualify variable names with namespace prefixes. While it is technically permissible to use a namespace prefix in a variable name, it is more readable to omit the namespace prefix in the qualified name. Also, because the variable is local to the query (or even to a portion of the query), the namespace is not needed. For example, while a variable named `$pre:order` is syntactically legal, it is clearer and easier to read a variable named `$order`.

sortby_expression

Sorts the data in the expression preceding the `sortby` clause by the specified XQuery expression, either from smallest to largest (`ascending`) or from largest to smallest (`descending`). The `sortby` expression has the syntax shown in Figure 2-6.

Figure 2-6 Sort By Expression Syntax Diagram



Usage Notes

The XQuery expression that is the argument of the `sortby` clause must evaluate to a unit value (simple type).

If neither `ascending` nor `descending` is specified, Liquid Data defaults to `ascending` sort order.

To sort by multiple values, specify multiple comma-separated arguments. Multiple sort expressions will first sort by the first expression, then sort by the second expression (within groups that match on the first expression), and so on (see the second example below).

The following query sorts the results in descending order:

```
<root>
       {
       for $x in (3, 1, 2)
       return
       <result>
              <number>{ $x }</number>
       </result>
       sortby(number descending)
       }
</root>
```

This query produces the following results:

```
<root>
  <result>
    <number>3</number>
  </result>
  <result>
    <number>2</number>
  </result>
  <result>
    <number>1</number>
  </result>
</root>
```

The following example query uses a `sortby` expression with multiple arguments:

```
<root>
       {
       for $x in (3, 1, 2)
       for $y in ("b", "c", "a")
       return
       <NumbersAndLetters>
              <number>{ $x }</number>
              <letter>{ $y } </letter>
       </NumbersAndLetters>
       sortby (./number descending, ./letter)
       }
</root>
```

This query produces the following results:

```
<root>
  <NumbersAndLetters>
    <number>3</number>
    <letter>a</letter>
  </NumbersAndLetters>
  <NumbersAndLetters>
    <number>3</number>
    <letter>b</letter>
  </NumbersAndLetters>
  <NumbersAndLetters>
    <number>3</number>
    <letter>c</letter>
  </NumbersAndLetters>
  <NumbersAndLetters>
    <number>2</number>
    <letter>a</letter>
  </NumbersAndLetters>
  <NumbersAndLetters>
    <number>2</number>
    <letter>b</letter>
  </NumbersAndLetters>
  <NumbersAndLetters>
    <number>2</number>
    <letter>c</letter>
  </NumbersAndLetters>
  <NumbersAndLetters>
    <number>1</number>
    <letter>a</letter>
  </NumbersAndLetters>
  <NumbersAndLetters>
    <number>1</number>
    <letter>b</letter>
  </NumbersAndLetters>
  <NumbersAndLetters>
    <number>1</number>
    <letter>c</letter>
  </NumbersAndLetters>
</root>
```

XQuery Expressions

Like other query languages, XQuery uses expressions as the building blocks of the query. Expressions can contain any number of expressions and can be arbitrarily complex.

An XQuery expression can have any combination of functions, operators, and other valid XQuery expressions. For a description of the functions available in Liquid Data, see "Functions Reference" on page 3-1. For details about the FLWR expression, see "FLWR Expression" on page 2-9.

This section describes the following building blocks of XQuery expressions:

- XML Markup Expression
- FLWR Expression
- PATH Expressions
- Conditional Expressions (if-then-else)
- Built-In Functions
- Constants
- Variables
- Operators
- Quantified Expressions
- Query Parameters

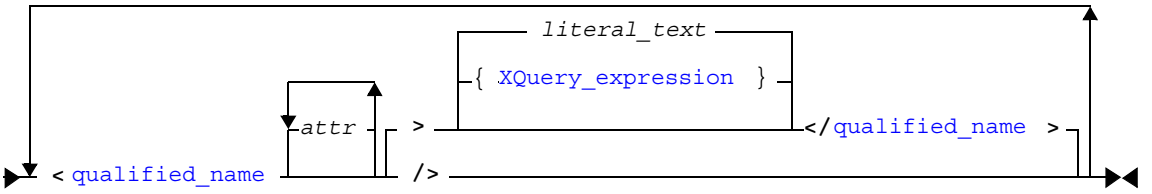
XML Markup Expression

The output of an XQuery is typically an XML document. You can place XML markup in the XQuery to create the opening and closing XML tags in the result document. You can add XML markup anywhere you can add an expression. Some common places for the markup to appear in a query are at the beginning of the query body, in the `return` clause of a FLWR expression, and at the end of the query body.

The Liquid Data Server requires that the results a query returns are well-formed XML. In Liquid Data, queries that return XML markup that is not well-formed (for example, results that do not have a single root node) will fail with a runtime exception.

The general syntax of XML markup in an XQuery is shown in Figure 2-7.

Figure 2-7 XML Markup Expression Syntax Diagram



where:

literal_text	Any text that can appear in the data portion of an XML tag.
attr	A valid XML attribute name/value pair. Can also be an expression that evaluates to a valid XML attribute name/value pair.

Usage Notes

The XQuery specification refers to XML Markup Expression as *element and attribute constructors*.

Curly braces ({}) are used in an XQuery to separate XQuery expressions with XML markup. If there is an XQuery expression that comes before some XML markup in a query, enclose the XQuery expression with a curly braces. If there is a closing curly brace (}) before the XML markup, there must be an opening curly brace ({) earlier in the query.

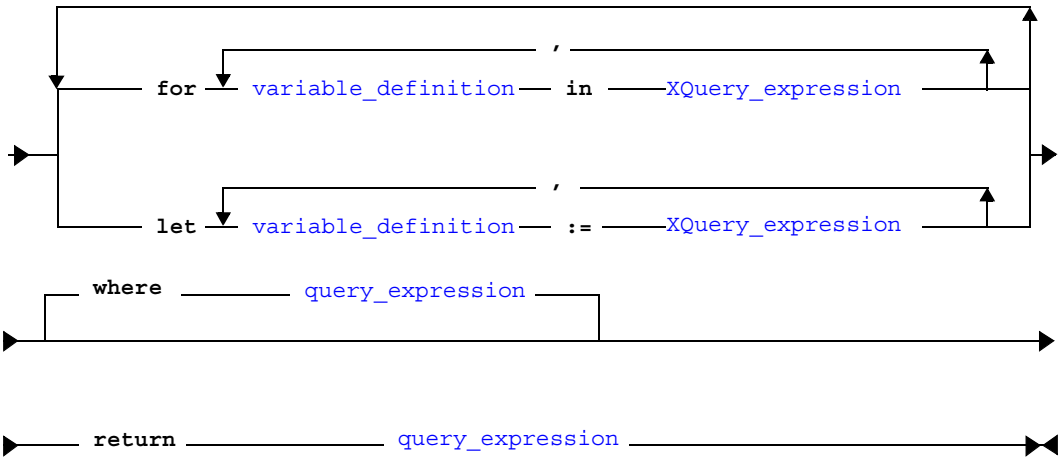
The following query includes XML markup at the beginning and end of the query and a query expression between the XML markup in the return clause:

```
<Founders>
{
for $x in ("Bill", "Ed", "Alfred")
return
   <founder>{$x}</founder>
}
</Founders>
```

FLWR Expression

The For, Let, Where, Return (FLWR) expression is an important building block of an XQuery query expression. A FLWR expression is a loop (when there is a `for` clause) that iterates over XML data and returns the desired results. By mixing query expressions and XML markup, the output of the FLWR expressions write out an XML document. You can create FLWR expressions within other FLWR expressions, nesting FLWR expressions to as many levels as needed. Figure 2-8 shows the basic syntax of a FLWR expression in Liquid Data.

Figure 2-8 FLWR Expression Syntax Diagram



Usage Notes

If you have a `for` or a `let` clause (or both, or any combination of `for` or `let` clauses), you must have a single `return` clause.

The XQuery Expressions in the `for` and `let` clause evaluates to a *sequence* of values. The FLWR expression then binds the sequence to the variable. A *sequence* is a set of zero or more values. For example, the sequence that contains the numbers 1, 2, and 3 can be expressed as `(1, 2, 3)`. The sequence of values can be expressed as any XQuery expression, including PATH Expressions and expressions containing literal or derived values (from other expressions, for example).

XQuery is a declarative query language, so when you specify loops or other programming constructs in an expression, you are specifying a logical expression of the data, not necessarily the physical plan for the query to be executed. The Liquid Data server will determine the most efficient execution plan for the query that produces the results declared in the query.

Nested FLWR expressions can represents joins between data sources. For details on specifying joins, see "Specifying Joins and Unions in XQuery" on page 2-21.

for Clause

The `for` clause binds a series of values to a variable. The variable(s) defined in the `for` clause represent an item in a sequence, and the loop is evaluated for each item in the sequence. When a variable is referenced later in a FLWR expression (in the `where` or `return` clauses, for example), it evaluates to the item in the sequence corresponding to the iteration of the `for` loop.

For example, consider the following `for` clause:

```
for $i in (1, 2, 3)
```

This `for` clause creates a loop whose body will be evaluated three times, first for the value 1, next for the value 2, and finally for the value 3.

let Clause

The `let` clause binds a whole sequence to a variable. The variable is then available for use in the FLWR expression. When reading the let clause, you can read the assign string (`:=`) as the phrase "be bound to." For example, consider the following `let` clause:

```
let $x := (1, 2, 3)
```

You can read this as "let the variable named x be bound to the sequence containing the items 1, 2, and 3."

where Clause

The `where` clause places a condition on the `for` and/or `let` clause that precedes the `where` clause. A `where` clause can be any query expression, including another FLWR expression. The `where` clause typically filters the number of matches for the FLWR loop. The filter specified by the `where` clause can specify a join between two sources. For example, consider the following query:

```
<results>
{
for $x in (1, 2, 3), $y in (2, 3, 4)
where $x eq $y
return
     <matches>{$x}</matches>
}
</results>
```

The `where` clause in this query filters (or joins) the results that match two sequences specified in the `for` clause. In this case, the numbers 2 and 3 match, and the query returns the following results:

```
<results>
     <matches>2</matches>
     <matches>3</matches>
</results>
```

return Clause

The `return` clause is evaluated for each successful (non-filtered) variable binding of the `for` loop. A `return` clause often includes XML markup combined with expressions that manipulate data. The combination of the XML markup and XQuery expressions writes out a portion of the XML result document that the query returns. For the syntax of XML markup in an XQuery expression, see "XML Markup Expression" on page 2-7.

PATH Expressions

Use PATH expressions to specify a node or a sequence of nodes in an XML tree. A PATH expression has the general syntax described in Figure 2-9 and Figure 2-11.

Note: This is only a partial syntax to highlight common PATH expression use cases in Liquid Data. For more detailed syntax of PATH expressions, see the PATH section of the XQuery specification:

http://www.w3.org/TR/2001/WD-xquery-20011220/#id-path-expressions

Figure 2-9 Partial PATH Expression Syntax Diagram

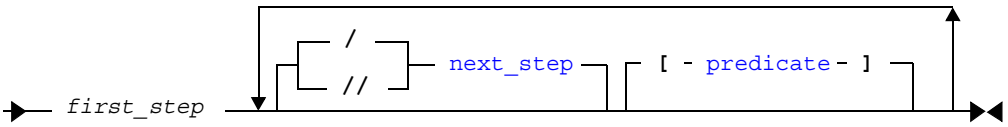


Figure 2-10 First Step of PATH Expression Syntax Diagram

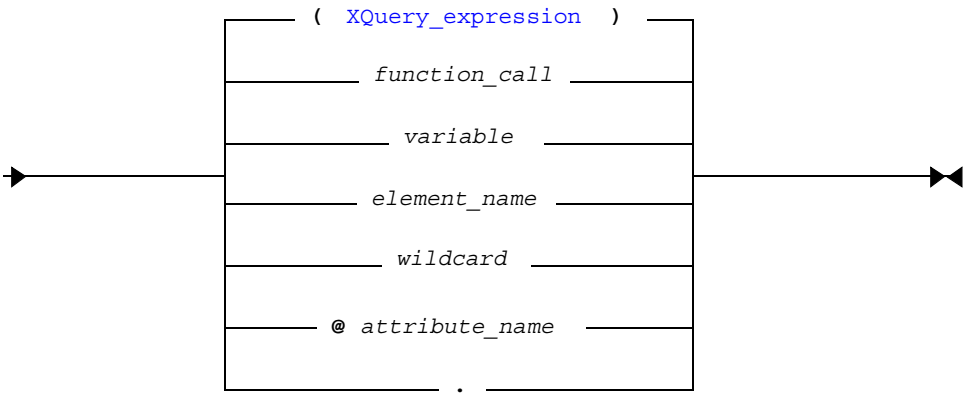
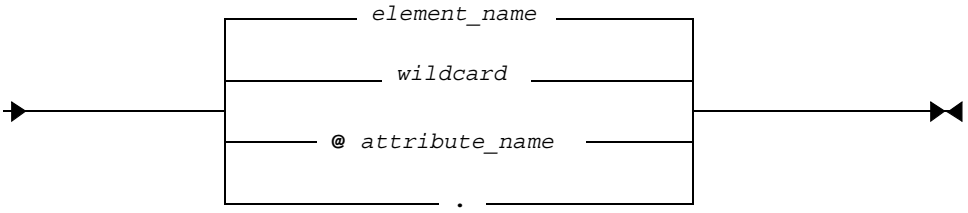


Figure 2-11 Next Step of PATH Expression Syntax Diagram



where:

`first_step`	An expression that specifies the first step of a PATH expression. See Figure 2-10 for the syntax of the first step.
`next_step`	An expression that specifies a step of a PATH expression. See Figure 2-11 for the syntax of a step.
`function_call`	An XQuery function call which specifies the step. For example, you can call the `xf:document` function to access many Liquid Data data sources.
`variable`	An variable accessible to the query. For details on variables, see "Variables" on page 2-17.
`element_name`	The qualified name of a node specifying a step. Precede an element name by a colon (`:`) if the element name is the same as an XQuery keyword (for example, `:for`).
`attribute_name`	The name of an attribute for the step (must be proceeded by the `@` character).
`wildcard`	A wildcard (`*`) specifies everything in the context node. Wildcards can also be used in conjunction with namespace prefixes; the wildcard can represent either the prefix or the node names. For example, `foo:*` represents all nodes with the prefix `foo`, and `*:foo` represents all prefixes with a node named `foo`.
`/`	The next node in the XML tree (go down one level from this node).
`//`	All descendants of this node in the XML tree (go down as many levels as there are from this node).
`dot (.)`	Specifies to use the current node as the step.
`predicate`	An XQuery Expression (see "XQuery Expressions" on page 2-7) that returns a boolean value. If the boolean evaluates to FALSE for a given value of the sequence of values produced by the preceding PATH expression, then that value is filtered from the result. If the boolean evaluates to TRUE for a given value of the sequence of values, then that value is allowed in the result. PATH expressions in the predicate are evaluated relative to the step in the tree in which the predicate occurs.

Usage Notes

You can qualify steps in a PATH expression with a predicate. Predicates in PATH expressions are surrounded by square brackets (`[]`).

XQuery expressions in the first step other than the ones specified in Figure 2-10 must be surrounded by parenthesis (`()`).

Liquid Data also supports the XPath non-abbreviated step syntax (for example, `child::`, `descendant::`, and so on).

Figure 2-12 shows some example PATH expressions and describes their meanings.

Table 2-12 PATH Expression Examples and Their Meanings

PATH Expression	Evaluates to...
`document("RTL-CUSTOMER")/db/CUSTOMER/FIRST_NAME`	The `FIRST_NAME` child elements of `CUSTOMER` from the `RTL-CUSTOMER` relational database. The document function is used to fetch the schema for the relational database, and then the steps take you to the `FIRST_NAME` elements.
`./number`	From the current context, find the children named `number`. See the second example in "sortby_expression" on page 2-4 for an example of this PATH expression.
`document("RTL-CUSTOMER")/db/CUSTOMER/FIRST_NAME[. eq "Homer"]`	The `FIRST_NAME` elements containing the data "Homer".

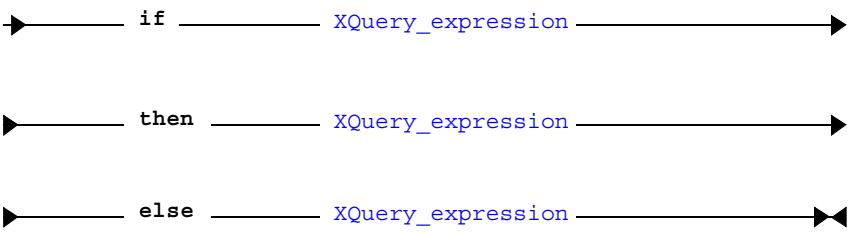
This section does not cover all of the expressions you can create with PATH expressions; there are many more complex expressions you can create with PATH expressions. For more detailed syntax and examples of PATH expressions, see the PATH section of the XQuery specification:

`http://www.w3.org/TR/2001/WD-xquery-20011220/#id-path-expressions`

Conditional Expressions (if-then-else)

You can create conditional expressions using the `if then else` construct. The conditional expression syntax is described in Figure 2-13.

Figure 2-13 Conditional Expression Syntax Diagram



Usage Notes

You can provide `elseif` logic by nesting another conditional expression as the argument of the `else` clause. For example:

```
if $a + $b lt 20
then "less than 20"
else
    if $a + $b gt 50
    then "greater than 50"
    else "between 20 and 50"
```

The `else` clause is required, but you can provide if-then logic (with no `else`) by specifying the empty set for the argument of the `else` clause. For example, the following query:

```
<a>
{
for $x in (1, 2), $y in (3, 4)
return
if $x + $y lt 5
then <b>{"less than 5"}</b>
else <b>{ () }</b>
}
</a>
```

Returns the following results:

```
<a>
  <b>less than 5</b>
  <b/>
  <b/>
  <b/>
</a>
```

You can also use the XQuery function `xfext:if-then-else` to provide conditional logic, as described in "xfext:if-then-else" on page 3-79.

Built-In Functions

Liquid Data has many functions available for use in queries. You can include functions in any XQuery expression. The functions take zero or more inputs and return a single output. For details (including syntax and examples) on the functions available in Liquid Data, see "Functions Reference" on page 3-1.

Constants

You can specify constant literal values in an XQuery. String constants are surrounded by double-quotation marks (`"`); numeric constants are not.

String Constants

You use string constants to specify strings of characters in an XQuery expression. String constants are surrounded by either single-quotation marks (`'`) or double-quotation marks (`"`). The output of a string constant has the `xs:string` data type. Table 2-14 lists some examples of string constants.

Table 2-14 String Constants Expressions and What They Evaluate To

Numeric Constant	Evaluates to...
`"Hello there."`	Hello there.
`"123.45"`	123.45

Numeric Constants

You can specify numeric constants by specifying a number (with or without a decimal point). You can also specify a number using exponent notation. Table 2-15 lists some examples of numeric constants.

Table 2-15 Numeric Constants Expressions and What They Evaluate To

Numeric Constant	Evaluates to...
`123`	The number 123
`123.45`	The decimal number 123.45
`12345 e -2`	The double value equal to 123.45

Variables

The dollar sign character (`$`) specifies a variable in XQuery. The string that immediately follows the dollar sign is the variable name. You often specify variables and bind values to them in the `for` or `let` clause of a FLWR expression, and then use the variables in an expression in the `return` clause. For the syntax of a variable definition, see "variable_definition" on page 2-3.

Table 2-16 shows some examples of variable definitions and uses.

Table 2-16 XQuery Variable Expressions and their English Translations

XQuery Fragment	English Translation
`$x`	Value if a variable named "x".
`let $x := "hello"`	Let the value of the variable named "x" be bound to the string "hello".
`let $y := ("hello", "goodbye")`	Let the value of the variable named "y" be bound to the sequence ("hello", "goodbye").

You declare variable names and bind them to values in the `for` or `let` clauses of a FLWR expression. When you declare a variable that evaluates to a sequence in the `for` clause, the value of the variable when referenced in the `where` or `return` clause is bound to the item in the sequence corresponding to the iteration of the loop; the variable value is not the sequence to which the variable was declared in the `for` clause.

For example, consider the following query that binds the variable `$x` in the `for` clause:

```
<Beatles>
{
for $x in ("JOHN","PAUL", "GEORGE", "RINGO")
return
   <beatle>{$x}</beatle>
}
</Beatles>
```

This query evaluates the variable `$x` once for each iteration of the `for` loop, and in each instance the value is an item in the sequence. It returns the following result:

```
<Beatles>
    <beatle>JOHN</beatle>
    <beatle>PAUL</beatle>
    <beatle>GEORGE</beatle>
    <beatle>RINGO</beatle>
</Beatles>
```

Now consider the following query that binds the variable `$x` in the `let` clause:

```
<Beatles>
{
let $x := ("JOHN","PAUL", "GEORGE", "RINGO")
return
   <beatle>{$x}</beatle>
}
</Beatles>
```

This query evaluates the variable `$x` only once, and the value is the sequence to which the variable is bound in the `let` clause. It returns the following result:

```
<Beatles>
    <beatle>JOHNPAULGEORGERINGO</beatle>
</Beatles>
```

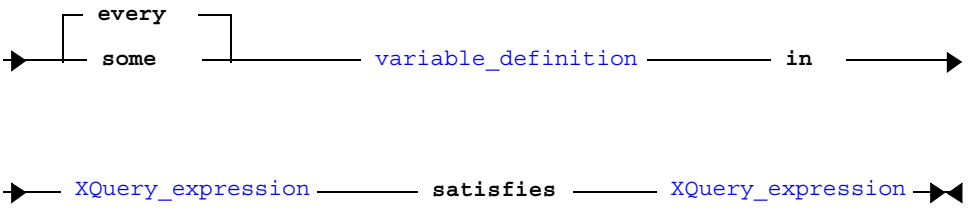
Operators

Operators allow you to construct an expression that compares or combines expressions. Use operators to construct mathematical expressions, logic tests, and tests comparing values (for example, greater than, less than, and so on). For details (including syntax and examples) on the operators available in Liquid Data, see "Comparison Operators" on page 3-30, "Logical Operators" on page 3-67, and "Numeric Operators" on page 3-70.

Quantified Expressions

A quantified expression returns a boolean based on the comparison of two XQuery expressions. When using the construct with the keyword `every`, it evaluates to `true` if every instance of the `satisfies` expression is `true`. When using the keyword `some`, it evaluates to `TRUE` if any instance of the `satisfies` expression is `true`. Figure 2-17 shows the syntax of the quantified expression.

Figure 2-17 Quantified Expression Syntax Diagram



Example

The following query:

```
<results>
  <a>{every $y in (2, 3, 4) satisfies ($y eq 2)}</a>
</results>
```

returns the following results:

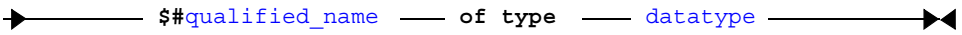
```
<results>
  <a>false</a>
</results>
```

The reason for the `false` result data value in this query is because not every instance of the `satisfies` expression evaluates to `true`; only the first instance (when the variable `$y` is bound to the value `2`) evaluates to `TRUE`. Therefore, the quantified expression returns `false`. If you substitute the keyword `some` for `every` in this query, the quantified expression will return `true`.

Query Parameters

Query parameters are variables whose value is supplied at query runtime. A Liquid Data query parameter begins with the string `$#` and has the syntax described in Figure 2-18.

Figure 2-18 Query Parameter Syntax Diagram



The specified `datatype` must be a valid Liquid Data data type, as described in "Data Types" on page 3-3.

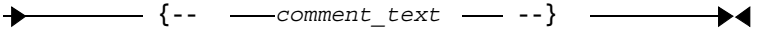
XQuery Comments and Join Hints

You can add comments anywhere in an XQuery. Comments are a useful means of documenting what the query is trying to accomplish. Also, the Liquid Data Server interprets certain specific comments as query compilation join hints when compiling and executing the query.

Comments

Any text, except for the hints (described below in Join Hints), between the opening and closing comment tags is considered a comment and is ignored by the Liquid Data query processor. Figure 2-19 shows the syntax for comments.

Figure 2-19 Query Comment Syntax Diagram



where:

comment_text	Any text used for a comment. The text is ignored at query compile-time and runtime.

Note: You cannot have comments within comments.

Join Hints

The Liquid Data query processor interprets join hints in a query to force certain optimizations on a query. Join hints are useful when you know something about the underlying data where one method of processing might perform better than another.

Table 2-20 describes the hints available in Liquid Data.

Table 2-20 Join Hints in Liquid Data

Join Hint Text	Description
`{--! ppright !--}`	Right parameter passing join
`{--! ppleft !--}`	Left parameter passing join
`{--! merge !--}`	Merge join
`{--! index !--}`	Index join

To use these join hints, place them to the right of the operator in the join condition. For more details on optimizing queries and examples of using join hints, see "Analyzing and Optimizing Queries" in *Building Queries and Data Views*.

Specifying Joins and Unions in XQuery

There is often a need to create result documents which combine data sourced from different places, whether those places are different tables in a relational database, different files containing XML data, or any other kinds of systems containing data, including combinations of relational, XML, Web Services, and other data. You can specify joins and unions in XQuery to combine data.

The main building block for specifying a join in XQuery is a set of nested FLWR expressions. Since FLWR expressions are loops, and since XML data is structured hierarchically, you can recursively loop over XML documents, taking values from an outer loop and using them in an inner loop.

This section describes some design patterns for specifying joins in XQuery. The following topics are included:

- Using Multiple For Statements to Create a Result
- Working From a Hierarchical Result Document Backwards: a Technique
- Specifying Aggregates and Groups (Group By)
- Specifying a Union-All Query

Using Multiple For Statements to Create a Result

To specify a join between two sources in an XQuery, use a nested FLWR expression. You iterate over one of the join documents in the outer loop, then iterate over the other in the inner loop, specifying the join condition in the `where` clause.

The following example query iterates over the `CUSTOMER` element of the `PB-WL` document, then joins the `PB-WL` document with the `PB_BB` document where the `CUSTOMER_ID` values are equal (where the same `CUSTOMER_ID` value appears in both documents). Then it prints out the `FIRST_NAME` element for each `CUSTOMER_ID` that is in both documents.

```
<RESULTS>
     {
     for $x in document("PB-WL")/db/CUSTOMER
          for $z in document("PB-BB")/db/CUSTOMER
          where ($z/CUSTOMER_ID  eq $x/CUSTOMER_ID)
          return
          <CUSTOMER>
               <FIRST_NAME>{ xf:data($x/FIRST_NAME) }</FIRST_NAME>
          </CUSTOMER>
     }
</RESULTS>
```

The basic pattern for this join is:

```
for_clause
    for_clause
    where_clause (with join conditions)
    return_clause (with XML markup and data)
```

This technique is similar to a join in SQL because it binds values in the `for` clauses to data source values, which is analogous to how SQL selects values from tables in the `FROM` clause. In the `where` clause of the XQuery FLWR expression, you can specify join conditions by specifying a condition where the value in the outer loop compares to the value in the inner loop. The query then returns results that satisfy these join conditions.

Note: If you want the XML markup to appear in the result document even if there are no matches found in the query (similar to an outer join in SQL), you can set up the query so there are return values associated with each part of the `for` clause. For an example of a query that uses this pattern, see the query in the following Hierarchical Result Document example.

Working From a Hierarchical Result Document Backwards: a Technique

One technique for building a query is to look at the shape of the result document and begin to build that result document from the innermost elements working towards the outermost elements. This technique works especially well for schemas that have complex, repeatable elements that are nested within other complex elements.

Starting with the innermost repeatable elements, place a `return` clause followed by the XML markup for that part of the result. Continue this process until you have all of the XML markup for the query. Next, add the rest of the FLWR expression to correspond to each `return` clause. Finally, complete the `return` clause to add the data needed for each element.

For example, consider a query that needs to display results in the following shape:

```
<customers>
   <Customer>*
      <name>
      <orders>*
           <orderID>
           <lineItems>*
                 <quantity>
                 <price>
                 <product>
```

In this case, the inner-most repeatable element is `lineItems`. You can build the following return clause:

```
            return
            <lineItems>
                 <quantity>{ }</quantity>
                 <price>{ }</price>
                 <product>{ }</product>
            </lineItems>
```

Continuing this to the outside of the result schema yields the following:

```
return
<Customer>
   <name>{ }</name>
   return
   {
   <orders>
       <orderID>{ }</orderID>
       return
       {
       <lineItems>
            <quantity>{ }</quantity>
            <price>{ }</price>
            <product>{ }</product>
       </lineItems>
       }
   </orders>
   }
<Customer>
```

Next, fill in the outermost XML markup and some syntax details (like the `for` statements and the outer XML markup) to yield the following:

```
<customers>
{
     for
     return
     <Customer>
        <name>{ }</name>
        {
        for
        return
        <orders>
            <orderID>{ }</orderID>
            {
            for
            return
            <lineItems>
                 <quantity>{ }</quantity>
                 <price>{ }</price>
                 <product>{ }</product>
            </lineItems>
            {
        </orders>
        {
     <Customer>
}
</customers>
```

You now have the basic structure of the query. To make the query executable, add the variable bindings for the `for` clauses, any join conditions or other filters in the `where` clause, and the query expressions to fetch the data. The following query is executable against the sample database installed with Liquid Data:

```
<customers>
{
    for $cust in document("PB-BB")/db/CUSTOMER
    return
    <Customer>
        <name>{ xf:data($cust/FIRST_NAME) }</name>
       {
        for $orders in document("PB-BB")/db/CUSTOMER_ORDER
        where ($cust/CUSTOMER_ID  eq
            $orders/CUSTOMER_ID)
        return
        <orders>
            <orderID>{ xf:data($orders/ORDER_ID) }</orderID>
            {
                for $lineItems in
                    document("PB-BB")/db/CUSTOMER_ORDER_LINE_ITEM
                where ($orders/ORDER_ID  eq  $lineItems/ORDER_ID)
                return
                <lineItems>
                   <quantity>{ xf:data($lineItems/QUANTITY) }</quantity>
                   <price>{ xf:data($lineItems/PRICE) }</price>
                   <product>{ xf:data($lineItems/PRODUCT_NAME) }</product>
                </lineItems>
                }
            </orders>
            }
    </Customer>
}
</customers>
```

Note that this example uses the `xf:data` function to extract just the data portion from the node specified in the PATH expressions for the order ID, quantity, price, and product. If you omit the `xf:data` functions, the results will include the XML tags for those pieces of data in addition to the data between the tags. For more details on the `xf:data` function, see "xf:data" on page 3-9.

Specifying Aggregates and Groups (Group By)

Queries that use aggregate functions (for example, `xf:avg`, `xf:sum`, `xf:min`, `xf:max`, `xf:count`) often compute results based on the group at which the aggregation function is calculated. For example, if you want to find the sum of products calculated for each product, the aggregate group is *product*. To create queries with aggregate functions that apply at a particular group (analogous to the `GROUP BY` clause in SQL), you must use the `xf:distinct-values` function for each group of the aggregate.

For example, if you want to find the sum of sales for each product (that is, the total sales for product *a*, the total sales for product *b*, and so on), then the query must first find the distinct products; then, for each distinct product, iterate through the data to calculate the sum of orders of that product.

The following query demonstrates an aggregate (sum) grouped by product:

```
<results> {

       {-- For each distinct product --}
for $eachProduct in xf:distinct-values
          (document("PB-BB")/db/CUSTOMER_ORDER_LINE_ITEM/PRODUCT_NAME)

{-- Compute the sales for each product and bind the values to a sequence --}

let $listOfLineItemSales :=
   for $eachLineItem in document("PB-BB")/db/CUSTOMER_ORDER_LINE_ITEM
        {-- This condition restricts the line items to the product --}
   where ($eachLineItem/PRODUCT_NAME  eq  $eachProduct)
   return    {-- the Sales of the product per line item --}
       $eachLineItem/QUANTITY * $eachLineItem/PRICE

return    {-- Once for each product, print out the product name and add up
           the sum of all items in the list of line item salessales  --}
     <SalesbyProduct>
        <product_name>{ $eachProduct }</product_name>
        <sumOfSalesForProduct>{ xf:sum($listOfLineItemSales)
                              }</sumOfSalesForProduct>
     </SalesbyProduct>
       }
</results>
```

If the data included two distinct products, *Product A* and *Product B*, then the results of this query are as follows:

```
<results>
     <SalesByProduct>
          <product_name>Product A</product_name>
          <sumOfSalesForProduct>526.34</sumOfSalesForProduct>
     </SalesByProduct>
     <SalesByProduct>
          <product_name>Product B</product_name>
          <sumOfSalesForProduct>226.48</sumOfSalesForProduct>
     </SalesByProduct>
</results>
```

Specifying a Union-All Query

Use a comma character (`,`) to concatenate two query expressions into a single result document. For example, consider the following simple union-all query:

```
<RESULTS>
     {
     for $x in (1,2,3)
     return
     <number>{ $x }</number>
{-- insert a comma to specify a Union-All of the 2 query expressions --}
,
    for $y in (4,5,6)
    return
     <number>{ $y }</number>
     }
</RESULTS>
```

This query returns the following results:

```
<RESULTS>
    <number>1</number>
    <number>2</number>
    <number>3</number>
    <number>4</number>
    <number>5</number>
    <number>6</number>
</RESULTS>
```

The concatenated expressions are evaluated in the order they appear in the query. To break this down further, the following list describes the order in which each part of this query is evaluated:

1. The opening tag `<RESULTS>` is produced.
2. The first `for` loop is evaluated for the value 1, returning `<number>1</number>`.
3. This first `for` loop evaluates for the value 2, then again for the value 3.
4. Since the first `for` loop has completed all the values in the sequence `$x`, the query continues to the second `for` loop. The first iteration of the second for loop returns `<number>4</number>`.
5. The second for loop evaluates for the value 5, then again for the value 6.
6. The closing tag `</RESULTS>` is produced.

Reading the XQuery Syntax Diagrams

The XQuery syntax in this document is described using *railroad diagrams*. Railroad diagrams describe the language syntax in blocks. You read the diagrams from left-to-right and top-to-bottom, taking a path defined by the lines and arrows in the diagram. This section describes how to read the railroad diagrams used to define the XQuery syntax in this reference, and includes the following subsections.

Text Conventions

The following shows the conventions for text in the syntax diagrams:

- **Bold** indicates literal text.
- *Italics* indicate a variable. The contents of the variable are described below the diagram.
- Hyperlinks indicate a syntax block which is defined in the section that is the target of the hyperlink.

Follow the Lines and Arrows Coming Into the Diagram

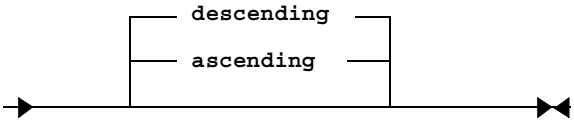
The syntax diagrams begin with an arrow and end with two facing arrows, as shown in Figure 2-21. From the left side of the diagram, follow the arrow to navigate through the diagram. Figure 2-21 represents a syntax block with no content. To read the diagram, start from the top line, continue on the next line when you reach the next arrow, and end at the left and right facing arrows.

Figure 2-21 Syntax Diagram Begin and end Arrows

Blocks with No Arrows Indicate Optional Content

Syntax blocks that extend above a line with no arrows indicate optional content. Follow the lines through the syntax block in one of the possible ways to form the syntax. Figure 2-22 shows a syntax block that can include either the literal text `descending`, the literal text `ascending`, or no text at all.

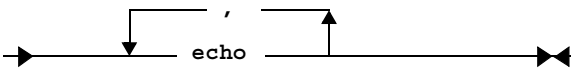
Figure 2-22 Optional Non-Repeatable Content Blocks



Blocks with Arrows (Loops) Indicate Repeatable Options

Syntax blocks that extend above a line with arrows indicate optional and repeatable content that can loop. Follow the lines through the syntax block in one of the possible ways to form the syntax. Figure 2-23 shows a syntax block that can include the literal text `hello` and, optionally, can repeat by placing a comma between instances of the literal text `hello`.

Figure 2-23 Optional Repeatable Content Blocks



Therefore, the following are legal according to this syntax diagram:

- `echo`
- `echo, echo`
- `echo, echo, echo, echo, echo`

The literal text `echo echo` is invalid according to this diagram (it is missing the comma).

CHAPTER 3

Functions Reference

The World Wide Web (W3C) specification for XQuery supports a discrete set of functions. BEA Liquid Data for WebLogic supports a subset of those functions as *built-in functions*. The Liquid Data built-in functions are accessible in the Data View Builder from Builder Toolbar—>Toolbox tab—>Functions panel.

For more information on the functions described here, see also:

- W3C XQuery 1.0 and XPath 2.0 Functions and Operators specification.
- Appendix D, the "Function and Operator Quick Reference" in the *XQuery 1.0 and XPath 2.0 Functions and Operators* specification
- XML Schema Part 2: Datatypes

This section provides a complete reference of the W3C functions Liquid Data supports, as well as any extended functions Liquid Data supports. This functions reference is organized by category as follows:

- About Liquid Data XQuery Functions
  - Naming Conventions
  - Occurrence Indicators
  - Data Types
  - Date and Time Patterns
- Accessor and Node Functions
- Aggregate Functions

About Liquid Data XQuery Functions

You can browse the Liquid Data XQuery functions in the Data View Builder. The functions are located in the Design tab —> Toolbox tab —> XQuery Functions. You can also make your own custom functions. This section describes the conventions used in the Liquid Data XQuery functions and describes the XQuery data types.

Naming Conventions

The xf: prefix is a W3C XML naming convention, also known as a *namespace.* Liquid Data supports extended functions that are enhancements to the XQuery specification, which you can recognize by their extended function prefix xfext:. For example, the full XQuery notation for an extended function is xfext:*function_name*. Extended functions accept standard input types, but they are limited to single values.

Liquid Data also supports extensions to XQuery data types that are designated with xsext:*datatype* notation. When you encounter the xsext: prefix, it means that the data type may have Liquid Data-imposed restrictions that are necessary to interface successfully with the Liquid Data Server.

The xfext: prefix identifies an extended function. The prefix identifies the type of function to you but the Data View Builder does not recognize or process the prefix.

Occurrence Indicators

An occurrence indicator indicates the number of items in a sequence. This notation usually appears on a parent node in a schema. Use these identifiers to determine the repeatability of a node.

- A question mark (?) indicates zero items or one single item.
- An asterisk (*) indicates zero or more items.
- A plus sign (+) indicates one or more items.

These occurrence indicators also communicate information about the data type when they appear in a function signature. For example:

- xs:*integer** represents a list of zero or more integers.
- *string*+ represents a list of one or more strings.
- *decimal*? represents zero or one decimal values. Therefore, the decimal value is optional.

Data Types

Every data element or variable has a data type. Function parameters have data type requirements and the function result is returned as a data type. The following table describes other data types that conform to the XQuery specification. Current compliance with the W3C XQuery specification extends to *XQuery 1.0 and XPath 2.0 Functions and Operators* specification dated 30 April 2002. Another helpful reference is *XML Schema Part 2: Datatypes*.

Table 3-1 Data Types

Data Type Name	Description
xs:anyType	Represents the most generic data type. All data types including anyAttribute, anyElement, anySimpleType, anyValue, as well as sequences, items, nodes, strings, decimals.
xsext:anyValue	A subset of xs:anyType including dateTime, boolean, string, numeric values, or any single value. Does not include anyAttribute, anyElement, item, node, sequence, or anySimpleType.
xs:boolean	A subset of xsext:anyValue. A value that supports the mathematical concept of binary-valued logic: true or false.

Table 3-1 Data Types (Continued)

Data Type Name	Description
xs:byte	A subset of xs:short. A sequence of decimal digits (0–9) with a range of 127 to -128. If the sign is omitted, plus (+) is assumed. **Examples:** -1, 0, 126, +100
xs:date	A subset of xsext:anyValue. Represents the leftmost component of dateTime *YYYY-MM-DD* where: • *YYYY* is the year • *MM* is the month • *DD* is the day May be preceded by a leading minus (-) sign to indicate a negative number. If the sign is omitted, plus (+) is assumed. May be immediately followed by a Z to indicate Coordinated Universal Time (UTC) or, to indicate the time zone (the difference between the local time and Coordinated Universal Time), immediately followed by a sign, + or -, followed by the difference from UTC represented as *hh:mm*. **Example:** To specify 1:20 pm on May the 31st, 1999, write: 1999-05-31.

Table 3-1 Data Types (Continued)

Data Type Name	Description
xs:dateTime	A subset of xsext:anyValue. Represents the format *YYYY-MM-DDThh:mm:ss* where: • *YYYY* is the year • *MM* is the month • *DD* is the day • T is the date/time separator • *hh* is the hour • *mm* is the minute • *ss* is the second May be preceded by a leading minus (-) sign to indicate a negative number. If the sign is omitted, plus (+) is assumed. Additional digits can be used to increase the precision of fractional seconds if desired (ss.ss...) with any number of digits after the decimal point is supported. May be immediately followed by a Z to indicate Coordinated Universal Time (UTC) or, to indicate the time zone (the difference between the local time and Coordinated Universal Time), immediately followed by a sign, + or -, followed by the difference from UTC represented as hh:mm. **Example:** To specify 1:20 pm on May the 31st, 1999 EST, which is five hours behind Coordinated Universal Time (UTC), write: 1999-05-31T13:20:00-05:00.
xs:decimal	A subset of xsext:anyValue. Includes all integer types, such as xs:integer, xs:long, xs:short, xs:int, or xs:byte. Represents a finite-length sequence of decimal digits (0–9) separated by an optional period as a decimal indicator. An optional leading sign is allowed. If the sign is omitted, plus (+) is assumed. Leading and trailing zeroes are optional. If the fractional part is zero, the period and following zeroes can be omitted. **Examples:** -1.23, 12678967.543233, +100000.00, 210
xs:double	A subset of xsext:anyValue. There are no subordinate data types; however, xs:float and xs:decimal, and all derived types, can be promoted to xs:double in certain cases, such as function calls. Represents a double precision 64-bit floating point value. Supports the special values positive and negative zero, positive and negative infinity and not-a-number (0, -0, INF, -INF and NaN).

Table 3-1 Data Types (Continued)

Data Type Name	Description
xs:float	A subset of xsext:anyValue. There are no subordinate data types; however, xs:decimal, and all derived types, can be promoted to xs:float in certain cases, such as function calls. Represents a Single-precision 32-bit floating point value. Supports the special values positive and negative zero, positive and negative infinity and not-a-number (0, -0, INF, -INF and NaN).
xsext:item	A subset of xs:anyType. Includes xsext:anyValue and xsext:node. Excludes any sequence. Represents a list element, individual value, or attribute.
xs:int	A subset of xs:long. Represents a finite-length sequence of decimal digits (0–9). An optional leading sign is allowed. If the sign is omitted, plus (+) is assumed. **Examples:** -1, 0, 12678967543233, +100000
xs:integer	A subset of xs:decimal. Represents a finite-length sequence of decimal digits (0–9). An optional leading sign is allowed. If the sign is omitted, plus (+) is assumed. **Examples:** -1, 0, 12678967543233, +100000
xs:long	A subset of xs:decimal. A sequence of decimal digits (0–9) with a range of 9223372036854775807 to -9223372036854775808. If the sign is omitted, plus (+) is assumed. **Examples:** -1, 0, 12678967543233, +100000
xsext:node	A subset of xsext:anyValue. A component in a tree structure that represents a data element.
xs:short	A subset of xs:int. A sequence of decimal digits (0–9) with a range of 32767 to -32768. If the sign is omitted, plus (+) is assumed. **Examples:** -1, 0, 12678, +10000

Table 3-1 Data Types (Continued)

Data Type Name	Description
xs:string	A subset of xsext:anyValue. A sequence that contains alphabetic, numeric, or special characters.
xs:time	A subset of xsext:anyValue. Represents the rightmost segment of the dateTime format where: • *hh* is the hour • *mm* is the minute • *ss* is the second May contain an optional following time zone indicator. **Examples:** • To indicate 1:20 pm EST, which is five hours behind Coordinated Universal Time (UTC), write: 13:20:00-05:00. • Midnight is 00:00:00.

Date and Time Patterns

You can construct date and time patterns using standard Java class symbols. The following table shows the pattern symbols you can use.

Table 3-2 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 3-2 Date and Time Patterns (Continued)

This Symbol	Represents This Data	Produces This Result
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	Pacific Standard Time Pacific Daylight Time

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

Accessor and Node Functions

Accessor and node functions operate on different types of nodes and node values. They accept single node input and return a value based on the node type. These function are not available in the XQuery functions section of the Data View Builder, but the Data View Builder will, in some circumstances, generate queries that use these functions. The functions available are:

- xf:data
- xf:document (format 1)
- xf:document (format 2)
- xf:local-name

xf:data

Returns the typed-value of each input node. This function is not available in the XQuery functions section of the Data View Builder.

Data Types

- Input data type: xsext:node?
- Returned data type: xsext:anyValue?

Notes

The `xf:data` function is available to Liquid Data, but you cannot explicitly map a node in the Data View Builder, so you therefore cannot construct a query in the Data View Builder that uses the xf:data function. In some cases, however, the Data View Builder will implicitly generate queries that use the xf:data function. The typical case when the Data View Builder generates the xf:data function is when it does not know the name of the elements at query generation time, and it uses the xf:data function in a variable expression containing wildcard characters.

If the source value is not a node, the function returns an error.

XQuery Specification Compliance

- Liquid Data does not use a list of nodes; it uses only an optional node.
- Liquid Data does not generate an error when you specify a document node. It returns an empty list.

Examples

- `xf:data(<a>{3}</a>)` returns the numeric value 3.
- `xf:data(<a/>)` returns an empty list ().
- `xf:data((<a>{3}</a>, <a>{7}</a>))` generates a compile-time error because the parameter is a list of nodes.
- `xf:data(<date location="SD">2002-07-12</date>)` returns the string value "2002-07-12.".
- `xf:data(3)` generates a compile-time error because 3 is not a node.

xf:document (format 1)

Returns the specified document. This function is not available in the XQuery functions section of the Data View Builder.

Data Types

- Input data type: `xs:string`
- Returned data type: node

Notes

The input of this version of the `xf:document` function is the logical name of a Liquid Data data source.

Use the `xf:document` function to specify an XML document. Because Liquid Data models data sources as XML documents, the XML document specified can represent a relational database, an XML file, or other data sources registered in the Liquid Data Administration Console. The `xf:document` function is available to Liquid Data, but you cannot explicitly map a node in the Data View Builder. In many cases, however, the Data View Builder implicitly generates queries that use the `xf:document` function.

Example

```
xf:document("My_Relational_DS")
```

xf:document (format 2)

Returns the specified document for the given dynamic data source. This function is not available in the XQuery functions section of the Data View Builder.

Data Types

- Input data type: `xs:string`
- Input data type: `xs:string`
- Returned data type: node

Notes

This version of the `xf:document` function is used with dynamic XML and delimited file data sources (a dynamic data source is a data source in which the data file is specified at query runtime). For the first input, specify the logical name of a Liquid Data data source. For the second input, specify a URL or file (absolute path or relative to the Liquid Data Repository for the type of data source).

Use the `xf:document` function to specify an XML document. Because Liquid Data models data sources as XML document, the XML document specified can represent a relational database, an XML file, or other data sources registered in the Liquid Data Administration Console. The `xf:document` function is available to Liquid Data, but you cannot explicitly map a node in the Data View Builder. In many cases, however, the Data View Builder implicitly generates queries that use the `xf:document` function.

Example

```
xf:document("My_XML_DS", "c:\myFolder\file.xml")
```

xf:local-name

Returns a string value that corresponds to the local name of the specified node. This function is not available in the XQuery functions section of the Data View Builder.

Data Types

- Input data type: *xsext:node*
- Returned data type: *xs:string?*

Notes

The xf:local-name function is available to Liquid Data, but you cannot explicitly map a node in the Data View Builder, so you therefore cannot construct a query in the Data View Builder that uses the xf:local-name function. In some cases, however, the Data View Builder will implicitly generate queries that use the xf:local-name function. The typical case when the Data View Builder generates the xf:local-name function is when it does not know the name of the elements at query generation time, and it uses the xf:local-name function in a variable expression containing wildcard characters.

XQuery Specification Compliance

- Liquid Data does not support the format that accepts no input parameters.
- Liquid Data supports an optional string as the returned value instead of a required string.

Examples

- `xf:local-name(<db:homes/>)` returns the string value "homes."
- `xf:local-name(73)` generates a compile-time error because the parameter is a number and not a node.

Aggregate Functions

Aggregate functions process a sequence as argument and return a single value computed from values in the sequence. Except for the Count function, if the sequence contains nodes, the function extracts the value from the node and uses it in the computation. The following aggregate functions are available:

- xf:avg
- xf:count
- xf:max
- xf:min
- xf:sum

xf:avg

Returns the average of a sequence of numbers.

Data Types

- Input data type: *xs:double**
- Returned data type: xs:*double?*

Notes

If the source value contains nodes, the value of each node is extracted using the xf:data function. If an empty list occurs, it is discarded.

If the source value contains only numbers, the Avg function returns the average of the numbers, which is the sum of the source sequence divided by the count of the source sequence.

If the source value is an empty list, the function returns an empty list.

If the source value contains non-numeric data, the function returns an error.

XQuery Specification Compliance

Liquid Data requires a list of double precision values instead of a list of items.

Examples

- `xf:avg((4, 10))` returns the double precision floating point value 7.0.
- `xf:avg((4, (), 10))` also returns the double precision floating point value 7.0.
- `xf:avg((4, "10"))` generates a compile-time error because the input sequence contains a string.

---

xf:count

Returns the number of items in the sequence in an unsigned integer.

Data Types

- Input data type: *xs:item**
- Returned data type: *xs:integer*

Notes

If the source value is an empty list, the function returns an empty list.

XQuery Specification Compliance

Liquid Data returns an integer value (xs:integer) instead of an unsigned int (xs:unsignedInt) value.

Examples

- `xf:count((3, "10"))` returns the integer value 2.
- `xf:count(())` returns the integer value 0.
- `xf:count((3, "10", (), ))` returns the value 3 (the empty list is ignored).

---

xf:max

Returns the maximum value from a sequence. If there are two or more items with the same value, the specific item whose value is returned is implementation-dependent.

Data Types

- Input data type: *xsext:item**
- Returned data type: *xsext:item?*

Notes

If the source value contains nodes, the value of each node is extracted using the xf:data function. If an empty list occurs, it is discarded.

All values in the list must be instances of one of the following types:

- *numeric*
- xs:*string*
- xs:*date*
- xs:*time*
- xs:*dateTime*

For example, if the list contains items with typed values that represent both decimal values and dates, an error will occur.

The values in the sequence must have a total order:

- DateTime values must all contain a time zone or omit a time zone.
- Duration values must contain only years and months or contain only days, hours, minutes and seconds.

Both of these conditions must be true; otherwise, the function returns an error.

XQuery Specification Compliance

- Liquid Data does not support a format with a collation literal.
- Liquid Data has no restrictions on date and time input values.

- Liquid Data supports a correct return type of *xs:item?* instead of *xs:anySimpleType?*, which is incorrect.
- Liquid Data supports only numeric, xs:*string*, xs:*date*, xs:*time*, and xs:*dateTime* data types.

Examples

- `xf:max((3, 10))` returns the value `10`.
- `xf:max((<a>{4}</a>, 3, (), <b >{2}</b>))` returns `<a>{4}</a>`.

---

xf:min

Returns the minimum value from a sequence of numbers. If there are two or more items with the same value, the specific item whose value is returned is implementation-dependent.

Data Types

- Input data type: *xsext:item**
- Returned data type: *xsext:item?*

Notes

If the source value contains nodes, the value of each node is extracted using the Data function. If an empty list occurs, it is discarded.

After extracting the values from nodes, the sequence must contain only values of a single type.

The values in the sequence must have a total order:

- DateTime values must all contain a time zone or omit a timezone
- Duration values must contain only years and months or contain only days, hours, minutes and seconds

Both of these conditions must be true; otherwise, the function returns an error.

XQuery Specification Compliance

- Liquid Data does not support a format with a collation literal.
- Liquid Data has no restrictions on date and time input values.

- Liquid Data supports a correct return type of *xs:item?* instead of *xs:anySimpleType?*, which is incorrect.
- Liquid Data supports only numeric, xs:string, xs:date, xs:time, and xs:dateTime data types.

Examples

- `xf:min((3, 10))` returns the value 3.
- `xf:min((<a>{4}</a>, 3, (), <b>{2}</b>))` returns `<b>{2}</b>`.
- `xf:min((3, 4, "2"))` generates an error because the sequence contains both numeric and string values.
- `xf:min(())` returns an empty list ().

---

xf:sum

Returns the sum of a sequence of numbers.

Data Types

- Input data type: *xsext:anyValue**
- Returned data type: *xsext:anyValue?*

Notes

If the source value contains nodes, the value of each node is extracted using the Data function. If an empty list occurs, it is discarded.

If the source value contains only numbers, the Sum function returns the sum of the numbers.

If the source value contains non-numeric data, the function returns an error.

If the input sequence is empty, the function returns an empty list.

XQuery Specification Compliance

- Liquid Data adheres to the prior XQuery specification (December, 2001) by returning an empty list if the input sequence is empty.

- Liquid Data output depends on the input type. If the input type is xs:*decimal*, the returned value is xs:*decimal*; if the input type is xs:*decimal* and xs:*float*, the returned value is xs:*float*; if the input type is xs:*double*, the returned value is xs:*double*.

Examples

- `xf:sum((3, 8, (), 1))` returns the value `12`.
- `xf:sum(())` returns an empty list ().
- `xf:sum((<a>{4}</a>, 3))` returns a value of `7`.
- `xf:sum(("7", 3))` generates a compile-time error because the sequence that is passed in to the function is not homogenous.

Boolean Functions

Boolean functions return true (1) or false(0) values. The following boolean functions are available:

- xf:false
- xf:not
- xf:true

xf:false

Returns the boolean value false.

Data Types

- Input data type: No input data required.
- Returned data type: *xs:boolean*

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:false()` returns `false`.
- `xf:false(34)` generates a compile-time error because the function does not accept any parameters.

xf:not

Returns `true` if the value of the source value is `false` and `false` if the value of the source value is `true`.

Data Types

- Input data type: *xs:boolean?*
- Returned data type: *xs:boolean?*

XQuery Specification Compliance

- Liquid Data accepts an optional boolean value instead of a sequence as input.
- Liquid Data returns a true value if the input is an empty list.
- Liquid Data returns an optional boolean value instead of one boolean value.

Examples

- `xf:not(xf:false())` returns the boolean value `true`.
- `xf:not(xf:true())` returns the boolean value `false`.
- `xf:not(32)` generates a compile-time error because the input value is not boolean.
- `xf:not(())` returns the boolean value `true`.

---

xf:true

Returns the boolean value `true`.

Data Types

- Input data type: No input data required.
- Returned data type: *xs:boolean*

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:true()` returns `true`.
- `xf:true("34")` generates a compile-time error because the function does not accept any parameters.

Cast Functions

Cast functions process a source value as the argument and type cast the output to a different datatype. Type casting will typically fail if applied to more than one element. An empty list is allowed, but the result of the type casting will consist of an empty list. Type casting functions are more likely to generate exceptions at run time if the parameter cannot be converted to the corresponding type.

The following table describes Liquid Data data types that conform to the XQuery specification that you can use in type casting functions. For more information about data types, see the *XQuery 1.0 and XPath 2.0 Functions and Operators* specification. The following cast functions are available:

- cast as xs:boolean
- cast as xs:byte
- cast as xs:date
- cast as xs:dateTime
- cast as xs:decimal
- cast as xs:double
- cast as xs:float
- cast as xs:int
- cast as xs:integer
- cast as xs:long
- cast as xs:short
- cast as xs:string
- cast as xs:time

cast as xs:boolean

Converts the input to a boolean value (`true` or `false`).

If the input parameter is empty, the function returns an empty list. Otherwise, Liquid Data generates an error.

Data Types

- Input data type: *xs:anyValue*
- Returned data type: *xs:boolean*

Notes

This function uses the xf:boolean-from-string function.

XQuery Specification Compliance

Conforms to the current specification; however, Liquid Data does not accept the values “1” and “0” to represent true and false, as described in the *W3C XML Schema* document.

Examples

- `cast as xs:boolean ("true")` returns the boolean value `true`.
- `cast as xs:boolean ("FalSE")` returns the boolean value `false`.
- `cast as xs:boolean (0)` generates a runtime error because the value cannot be cast to a boolean value.
- `cast as xs:boolean (1)` generates a runtime error because the value cannot be cast to a boolean value.
- `cast as xs:boolean (())` returns an empty list ().

cast as xs:byte

Converts the input to a byte value.

Data Types

- Input data type: *xs:anyValue*
- Returned data type: *xs:byte*

Notes

This function uses the `xf:byte` function.

This function will complete sucessfully only if the value cast is a numeric value greater than -128 or less than 128; all other values will fail.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `cast as xs:byte(22)` returns the byte value of `22`.
- `cast as xs:byte(22.9334)` returns the byte value `22`.

---

cast as xs:date

Converts the input to a date value.

Data Types

- Input data type: *xs:anyValue*
- Returned data type: *xs:date*

Notes

This function uses the xf:date function.

The string must contain a date in one of these formats:

- *YYY-MM-DD*

- *YYYY-MM-DDZ*
- *YYYY-MM-DD-hh:mm*

where *YYYY* represents the year, *MM* represents the month (as a number), *DD* represents the day, *hh* and *mm* represents the number of hours and minutes that the timezone differs from GMT (UTC). Z indicates that the date is in the GMT timezone.

If the string cannot be parsed into a date value, Liquid Data generates an error.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `cast as xs:date ("2002-07-23")` returns the date `2002-07-23`.
- `cast as xs:date ("2002-07")` generates a runtime error because the value cannot be converted to a date.

---

cast as xs:dateTime

Converts the input to a dateTime value.

Data Types

- Input data type: *xs:anyValue*
- Returned data type: *xs:dateTime*

Notes

This function uses the xf:date function.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `cast as xs:dateTime ("2002-07-23T23:04:44")` returns the dateTime value July 23rd, 2002 at 11:04:44 PM in the local timezone.

- `cast as xs:dateTime ("2002-07-23T23:04:44-08:00")` returns the dateTime value July 23rd, 2002 at 11:04:44 PM in the a timezone that is offset by -8 hours from GMT (UTC).
- `cast as xs:date ("2002-07-23")` generates a runtime error because no time value is specified.

---

cast as xs:decimal

Converts the input to a decimal value.

Data Types

- Input data type: *xs:anyValue*
- Returned data type: *xs:decimal*

Notes

This function uses the xf:decimal function.

XQuery Specification Compliance

- Liquid Data does not support not-a-number (NaN) or the negative and positive infinity values -INF and INF.
- Liquid Data attempts to support any input value, instead of just string literals, and convert it at run time.
- Liquid Data supports "e" and "E" to construct floating point integer values.

Examples

- `cast as xs:decimal ("213")` returns the decimal value `213`.
- `cast as xs:decimal ("-100")` returns the decimal value `-100`.
- `cast as xs:decimal (0)` returns the decimal value `0`.

---

cast as xs:double

Converts the input to a double precision value.

Data Types

- Input data type: *xs:anyValue*
- Returned data type: *xs:double*

Notes

This function uses the xf:double function.

XQuery Specification Compliance

- Liquid Data does not support not-a-number (NaN) or the negative and positive infinity values -INF and INF.
- Liquid Data attempts to support any input value, instead of just string literals, and convert it at run time.

Examples

- `cast as xs:double ("21")` returns the double precision value `21.0`.
- `cast as xs:double ("-3e3")` returns the double precision value `-3000.0`.
- `cast as xs:double (0)` returns the double precision value `0.0`.
- `cast as xs:double ("abc)` generates a runtime error because the string cannot be converted to a double precision value.

---

cast as xs:float

Converts the input to a floating point value.

Data Types

- Input data type: *xs:anyValue*
- Returned data type: *xs:float*

Notes

This function uses the xf:float function.

XQuery Specification Compliance

- Liquid Data does not support not-a-number (NaN), -0, or the negative and positive infinity values -INF and INF.
- Liquid Data attempts to support any input value, instead of just string literals, and convert it at run time.

Examples

- `cast as xs:float ("21")` returns the floating point value `21.0`.
- `cast as xs:float ("-3e3")` returns the floating point value `-3000.0`.
- `cast as xs:float (0)` returns the floating point value `0.0`.
- `cast as xs:float ("abc)` generates a runtime error because the string cannot be converted to a floating point value.

---

cast as xs:int

Converts the input to an int value.

Data Types

- Input data type: *xs:anyValue*
- Returned data type: *xs:int*

Notes

This function uses the xf:int function.

XQuery Specification Compliance

Conforms to the current specification.

---

cast as xs:integer

Converts the input to an integer value.

Data Types

- Input data type: *xs:anyValue*
- Returned data type: *xs:integer*

Notes

This function uses the xf:integer function.

XQuery Specification Compliance

Conforms to the current specification.

---

cast as xs:long

Converts the input to a long value.

Data Types

- Input data type: *xs:anyValue*
- Returned data type: *xs:long*

Notes

This function uses the xf:long function.

XQuery Specification Compliance

Conforms to the current specification.

---

cast as xs:short

Converts the input to a short value.

Data Types

- Input data type: *xs:anyValue*
- Returned data type: *xs:short*

Notes

This function uses the xf:short function.

XQuery Specification Compliance

Conforms to the current specification.

---

cast as xs:string

Converts the input to a string value.

Data Types

- Input data type: *xs:anyValue*
- Returned data type: *xs:string*

Notes

This function uses the xf:string function.

XQuery Specification Compliance

- Liquid Data treats `xf:string` as both a constructor and an accessor.
- Liquid Data supports only the string format that requires one node of any type as the input.
- Liquid Data accepts `xsext:anyType` input instead of a list of items.
- Liquid Data returns an optional string.
- Liquid Data does not recognize entities.

Examples

- `cast as xs:string ("abc")` returns the string value `abc`.
- `cast as xs:string (21)` returns the string value `21`.
- `cast as xs:string (xf:true())` returns the string value `true`.
- `cast as xs:string (xf:false())` returns the string value `false`.

---

cast as xs:time

Converts the input to a time value.

Data Types

- Input data type: *xs:anyValue*
- Returned data type: *xs:time*

Notes

This function uses the `xf:time` function.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `cast as xs:time ("09:35:20")` returns the time value 9:35:20 AM in the current timezone.
- `cast as xs:time (<a>09:35:20</a>)` returns the time value 9:35:20 AM in the current timezone.
- `cast as xs:time ("9:35:20")` generates a runtime error because the time format is incorrect (hour specified with 1 digit instead of 2) and therefore the string cannot be converted to a time value.
- `cast as xs:time ("21:35:20-08:00")` returns the time value 9:35:20 PM in the a timezone that is offset by -8 hours from GMT (UTC).

Comparison Operators

XQuery has operators that are specific to comparisons operations. The following operators are available:

- eq
- ge
- gt
- le
- lt
- ne

eq

Returns true if Parameter1 is exactly equal to Parameter2.

Data Types

- Parameter1 data type: *xsext:anyValue?*
- Parameter2 data type: *xsext:anyValue?*
- Returned data type: *xs:boolean?*

Notes

This is a comparison operator that you can use as a function to compare operands.

If either operand is a node, Liquid Data extracts its typed value first, then performs a type check to ensure that the type of one operand is promotable to the other type; otherwise Liquid Data generates an error.

If either operand is an empty list, the function returns an empty list.

XQuery Specification Compliance

- Liquid Data does not cast xs:anySimpleType to any other supported type.
- Liquid Data does not support these data types: xs:yearMonthDuration, xs:dayTimeDuration, gregorian, xs:hexBinary, xs:base64Binary, xs:anyURI, xs:QName, or xs:NOTATION values.

Examples

- 45 eq 45.0 returns the boolean value true.
- 170 eq 34 returns the boolean value false.
- 3 eq "3" generates an error because the decimal value 3 cannot be promoted to the string value "3."
- 1 eq xf:true() generates an error because the decimal value 1 cannot be promoted to the boolean value true.
- "abc" eq "abc" returns the boolean value true.
- (1, ()) eq 1 evaluates to the boolean value true because there is exactly one value in the leftmost list and that value is equal to the rightmost value.
- (1, 2) eq 1 generates a compile-time error because the operator does not evaluate lists.

---

ge

Returns true if Parameter1 is greater than or equal to Parameter2.

Data Types

- Parameter1 data type: *xsext:anyValue?*
- Parameter2 data type: *xsext:anyValue?*
- Returned data type: *xs:boolean?*

Notes

This is a comparison operator that you can use as a function to compare operands.

If either operand is a node, Liquid Data extracts its typed value first, then performs a type check to ensure that the type of one operand is promotable to the other type; otherwise Liquid Data generates an error.

If either operand is an empty list, the function returns an empty list.

XQuery Specification Compliance

- Liquid Data does not cast xs:anySimpleType to any other supported type.

- Liquid Data does not support these data types: xs:yearMonthDuration, xs:dayTimeDuration, gregorian, xs:hexBinary, xs:base64Binary, xs:anyURI, xs:QName, or xs:NOTATION values.

Examples

See the examples for "eq" on page 3-30.

---

gt

Returns true if Parameter1 is greater than Parameter2.

Data Types

- Parameter1 data type: *xsext:anyValue?*
- Parameter2 data type: *xsext:anyValue?*
- Returned data type: *xs:boolean?*

Notes

This is a comparison operator that you can use as a function to compare operands.

If either operand is a node, Liquid Data extracts its typed value first, then performs a type check to ensure that the type of one operand is promotable to the other type; otherwise Liquid Data generates an error.

If either operand is an empty list, the function returns an empty list.

XQuery Specification Compliance

Liquid Data does not cast xs:anySimpleType to any other supported type.

Liquid Data does not support these data types: xs:yearMonthDuration, xs:dayTimeDuration, gregorian, xs:hexBinary, xs:base64Binary, xs:anyURI, xs:QName, or xs:NOTATION values.

Examples

See the examples for the "eq" operator (previous entry in this table).

---

le

Returns true if Parameter1 is less than or equal to Parameter2.

Data Types

- Parameter1 data type: *xsext:anyValue?*
- Parameter2 data type: *xsext:anyValue?*
- Returned data type: *xs:boolean?*

Notes

This is a comparison operator that you can use as a function to compare operands.

If either operand is a node, Liquid Data extracts its typed value first, then performs a type check to ensure that the type of one operand is promotable to the other type; otherwise Liquid Data generates an error.

If either operand is an empty list, the function returns an empty list.

XQuery Specification Compliance

Liquid Data does not cast xs:anySimpleType to any other supported type.

Liquid Data does not support these data types: xs:yearMonthDuration, xs:dayTimeDuration, gregorian, xs:hexBinary, xs:base64Binary, xs:anyURI, xs:QName, or xs:NOTATION values.

Examples

See the examples for for "eq" on page 3-30.

---

lt

Returns true if Parameter1 is less than or equal to Parameter2.

Data Types

- Parameter1 data type: *xsext:anyValue?*
- Parameter2 data type: *xsext:anyValue?*
- Returned data type: *xs:boolean?*

Notes

This is a comparison operator that you can use as a function to compare operands.

If either operand is a node, Liquid Data extracts its typed value first, then performs a type check to ensure that the type of one operand is promotable to the other type; otherwise Liquid Data generates an error.

If either operand is an empty list, the function returns an empty list.

XQuery Specification Compliance

Liquid Data does not cast xs:anySimpleType to any other supported type.

Liquid Data does not support these data types: xs:yearMonthDuration, xs:dayTimeDuration, gregorian, xs:hexBinary, xs:base64Binary, xs:anyURI, xs:QName, or xs:NOTATION values.

Examples

See the examples for for "eq" on page 3-30.

---

ne

The result is false if both values are false and true if at least one of the values is true. Parameter2 is not evaluated if Parameter1 evaluates to true.

Data Types

- Parameter1 data type: *xsext:boolean?*
- Parameter2 data type: *xsext:boolean?*
- Returned data type: *xs:boolean?*

Notes

This is a boolean operator that you can use as a function to return a true or false result. It is not a standard XQuery operator, but necessary to complete certain comparative expressions in Liquid Data.

The arguments and return type are all boolean.

If either operand is a node, Liquid Data extracts its typed value first, then performs a type check to ensure that the type of one operand is promotable to the other type; otherwise Liquid Data generates an error.

If either operand is an empty list, the function returns an empty list.

XQuery Specification Compliance

Liquid Data does not support these data types: xs:yearMonthDuration, xs:dayTimeDuration, gregorian, xs:hexBinary, xs:base64Binary, xs:anyURI, xs:QName, or xs:NOTATION values.

Examples

See the examples for "eq" on page 3-30.

Constructor Functions

Constructor functions process a source value as the argument. Every data element or variable has a data type. The data type determines the value that any function parameter can contain and the operations that can be performed on it. The Liquid Data supports the following type casting functions. The following constructor functions are available:

- xf:boolean-from-string
- xf:byte
- xf:decimal
- xf:double
- xf:float
- xf:int
- xf:integer
- xf:long
- xf:short
- xf:string

xf:boolean-from-string

Returns a boolean value of true or false from the string source value.

Data Types

- Input data type: *xs:string?*
- Returned data type: *xs:boolean?*

Notes

If the input parameter is empty, the function returns an empty list. Otherwise, Liquid Data generates an error.

XQuery Specification Compliance

- Conforms to the current specification; however, Liquid Data does not accept the values "1" and "0" to represent true and false, as described in the *W3C XML Schema* document.

Examples

- `xf:boolean-from-string("true")` returns the boolean value `true`.
- `xf:boolean-from-string("FaLSe")` returns the boolean value `false`.
- `xf:boolean-from-string("43")` generates a runtime error because the input value cannot be parsed into a boolean value.
- `xf:boolean-from-string(43)` generates a compile-time error because the input value is not a string.

---

xf:byte

Constructs a byte integer value from the string source value.

Data Types

- Input data type: *xsext:anyValue?*
- Returned data type: *xs:byte?*

Notes

An error occurs if the source value is greater than 127 or less than -128.

Liquid Data truncates the input if it is a non-integer number.

If the number falls outside of the range of byte values, the number wraps.

If the number is an integer that falls within the range, the value is unchanged.

If the input is a string, Liquid Data tries to parse it into a byte value.

If the input is the boolean value true, the function returns 1. If it is false, it returns 0.

XQuery Specification Compliance

- Liquid Data does not support not-a-number (NaN) or -0.
- Liquid Data attempts to support any input value and convert it at run time.

Examples

- `xf:byte('127')` returns the byte value one hundred twenty seven.
- `xf:byte(38)` returns the byte value 38.
- `xf:byte("-4")` returns the byte value -4.
- `xf:byte(128)` returns the byte value -128 because the number wraps.
- `xf:byte(-129)` returns the byte value 127 because the number wraps.
- `xf:byte(xf:true())` returns the byte value 1.
- `xf:byte(xf:false())` returns the byte value 0.
- `xf:byte("true")` generates a runtime error because the string literal cannot be converted to a byte value.
- `xf:byte('128')` returns an error because one hundred twenty eight is invalid for a byte integer expression.

xf:decimal

Constructs a decimal value from the source value.

Data Types

- Input data type: *xsext:anyValue?*
- Returned data type: *xs:decimal?*

XQuery Specification Compliance

- Liquid Data does not support not-a-number (NaN), -0, or the negative and positive infinity values -INF and INF.
- Liquid Data attempts to support any input value, instead of just string literals, and convert it at run time.
- Liquid Data supports "e" and "E" to construct floating point integer values.

Examples

- `xf:decimal("3")` returns the decimal value 3.
- `xf:decimal(99.1)` returns the decimal value 99.1 (the same value that is input to the function).
- `xf:decimal(xf:true())` returns the decimal value 1.
- `xf:decimal(xf:false())` returns the decimal value 0.
- `xf:decimal("true")` generates a runtime error because the string literal cannot be converted to a decimal value.

---

xf:double

Constructs a double precision value from the source value.

Data Types

- Input data type: *xsext:anyValue?*
- Returned data type: *xs:double?*

XQuery Specification Compliance

- Liquid Data does not support not-a-number (NaN), -0, or the negative and positive infinity values -INF and INF.
- Liquid Data attempts to support any input value, instead of just string literals, and convert it at run time.

Examples

- `xf:double("3")` returns the double precision floating point value 3.0.
- `xf:double(5.1)` returns the double precision floating point value 5.1.
- `xf:double(xf:true())` returns the double precision floating point value 1.0.
- `xf:double(xf:false())` returns the double precision floating point value 0.0.
- `xf:double("true")` generates a runtime error because the string literal cannot be converted to a double precision floating point value.
- `xf:double("12345678901234567890")` evaluates to the double precision floating point value 1.2345678901234567E19.

---

xf:float

Constructs a floating point value from the source value.

Data Types

- Input data type: *xsext:anyValue?*
- Returned data type: *xs:float?*

XQuery Specification Compliance

- Liquid Data does not support not-a-number (NaN), -0, or the negative and positive infinity values -INF and INF.
- Liquid Data attempts to support any input value, instead of just string literals, and convert it at run time.

Examples

- `xf:float(1)` returns the floating-point value 1.0.
- `xf:float("1")` returns the floating-point value 1.0.
- `xf:float(xf:true())` returns the floating point value 1.0.
- `xf:float(xf:false())` returns the floating-point value 0.0.
- `xf:float("true")` generates a runtime error because the string literal cannot be converted to a floating-point value.
- `xf:float("12345678901234567890")` returns the floating-point value 1.2345679E19.

---

xf:int

Constructs an integer value from the source value. The largest integer value is limited to a 32-bit expression.

Data Types

- Input data type: *xsext:anyValue?*
- Returned data type: *xs:integer?*

Notes

An error occurs if the source value is greater than 2,147,483,647 or less than -2,147,483,648. To the Liquid Data Server, the xf:int function is exactly the same as the xf:integer function.

XQuery Specification Compliance

- Liquid Data does not support not-a-number (NaN) or -0.
- Liquid Data attempts to support any input value, instead of just string literals, and convert it at run time.

Examples

- `xf:int(4056)` returns the int value 4056.
- `xf:int("-35")` returns the int value -35.

- `xf:int(xf:true())` returns the int value 1.
- `xf:int(xf:false())` returns the int value 0.
- `xf:int("true")` generates a runtime error because the string literal cannot be converted to an int value.

---

xf:integer

Constructs an integer value from the source value. The largest integer value is limited to a 32-bit expression.

Data Types

- Input data type: *xsext:anyValue?*
- Returned data type: *xs:integer?*

Notes

An error occurs if the source value is greater than 2,147,483,647 or less than -2,147,483,648. To the Liquid Data Server, the xf:integer function is exactly the same as the xf:int function.

XQuery Specification Compliance

- Liquid Data does not support not-a-number (NaN), -0, or the negative and positive infinity values -INF and INF.
- Liquid Data attempts to support any input value, instead of just string literals, and convert it at run time.

Examples

- `xf:integer(4056)` returns the int value 4056.
- `xf:integer("-35")` returns the int value -35.
- `xf:integer(xf:true())` returns the int value 1.
- `xf:integer(xf:false())` returns the int value 0.
- `xf:integer("true")` generates a runtime error because the string literal cannot be converted to an int value.

---

xf:long

Constructs a four-byte integer value from the source value. Use a long integer data type when the value exceeds the limitations imposed by other integer data types.

Data Types

- Input data type: *xsext:anyValue?*
- Returned data type: *xs:long?*

Notes

An error occurs if the source value is greater than 9,223,372,036,854,775,807 or less than -9,223,372,036,854,775,808.

XQuery Specification Compliance

- Liquid Data does not support not-a-number (NaN) or -0.
- Liquid Data attempts to support any input value, instead of just string literals, and convert it at run time.

Examples

- `xf:long(1)` returns the long integer value 1.
- `xf:long("-91")` returns the long integer value -91.
- `xf:long(xf:true())` returns the long integer value 1.
- `xf:long(xf:false())` returns the long integer value 0.
- `xf:long("true")` generates a runtime error because the string literal cannot be converted to a long integer value.

xf:short

Constructs a two-byte integer value from the source value. The largest short integer value is limited to a 16-bit expression.

Data Types

- Input data type: *xsext:anyValue?*
- Returned data type: *xs:short?*

Notes

An error occurs if the source value is greater than 32,767 or less than -32,768.

XQuery Specification Compliance

- Liquid Data does not support not-a-number (NaN) or -0.
- Liquid Data attempts to support any input value, instead of just string literals, and convert it at run time.

Examples

- `xf:short(1)` returns the short integer value 1.
- `xf:short("-91")` returns the short integer value -91.
- `xf:short(xf:true())` returns the short integer value 1.
- `xf:short(xf:false())` returns the short integer value 0.
- `xf:short("true")` generates an error because the string literal cannot be converted to a short integer value.

xf:string

Constructs a string value from the source value. The source value can be a sequence, a node of any kind, or a simple value.

Data Types

- Input data type: *xs:anyType*
- Returned data type: *xs:string?*

Notes

Liquid Data accepts any simple value, but supports no other accessor types, such as a sequence or other type of node.

XQuery Specification Compliance

- Liquid Data treats xf:string as both a constructor and an accessor.
- Liquid Data supports only the string format that requires one node of any type as the input.
- Liquid Data accepts xsext:anyType input instead of a list of items.
- Liquid Data returns an optional string.
- Liquid Data does not recognize entities.

Examples

- `xf:string(1)` returns the string value `1`.
- `xf:string("-91")` returns the string value `-91`.
- `xf:string(xf:true())` returns the string value `true`.
- `xf:string(xf:false())` returns the string value `false`.
- `xf:string("abc", "def")` generates a compile-time error because the function does not accept two parameters.
- `xf:string(("abc", "def"))` generates a compile-time error because the function does not accept a sequence as parameter.
- `xf:string(<a/>)` returns an empty string value "".
- `xf:string(<a>abc</a>)` returns the string value `abc`.

Date and Time Functions

Date and Time functions extract all or part of a dateTime expression and use it in a query. The following date and time functions are available:

- xf:add-days
- xf:current-dateTime
- xf:date
- xf:dateTime
- xf:get-day-from-date
- xf:get-day-from-dateTime
- xf:get-hours-from-dateTime
- xf:get-hours-from-time
- xf:get-minutes-from-dateTime
- xf:get-minutes-from-time
- xf:get-month-from-date
- xf:get-month-from-dateTime
- xf:get-seconds-from-dateTime
- xf:get-seconds-from-time
- xf:get-year-from-date
- xf:get-year-from-dateTime
- xf:time
- xfext:date-from-dateTime
- xfext:date-from-string-with-format
- xfext:date-to-string-with-format
- xfext:dateTime-from-string-with-format
- xfext:dateTime-to-string-with-format
- xfext:time-from-dateTime
- xfext:time-from-string-with-format
- xfext:time-to-string-with-format

xf:add-days

Adds the number of days specified by Parameter2 to the date specified by Parameter1. The value of Parameter2 may be negative.

Data Types

- Parameter1 data type: xs:*date?*
- Parameter2 data type: xs:*decimal?*
- Returned data type: xs:*date?*

Notes

If Parameter1 has a timezone, it remains unchanged. The returned value is always normalized into a correct Gregorian calendar date. If either parameter is an empty list, the function returns an empty list.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:add-days(xf:date("2002-07-15"), -3)` returns a date value corresponding to July 12, 2002.
- `xf:add-days(xf:date("2002-07-15"), 0)` returns a date value corresponding to July 15, 2002.
- `xf:add-days(xf:date("2002-07-15"), 2)` returns a date value corresponding to July 17, 2002.
- `xf:add-days("2002-07-15", 2)` generates a compile-time error because the first parameter is a string and not a date value.

xf:current-dateTime

Returns the current date and time.

Data Types

No parameters required.

Returned data type: xs:*dateTime*

Notes

The function returns the current date and time in the current timezone.

If the function is called multiple times during the execution of a query, it returns the same value each time.

XQuery Specification Compliance

Liquid Data returns the time zone where the Liquid Data Server is running.

Example

`xf:current-dateTime()` can return a dateTime value such as 2002-07-25T01:00:38.812-08:00, which represents July 25th, 2002 at 1:00:38 and 812 thousandths of a second in a time zone that is offset by -8 hours from GMT (UTC).

---

xf:date

Takes a string (rather than dateTime) and a parameter and returns a date from a source value, which must contain a date in one of these formats:

- *YYYY-MM-DD*
- *YYYY-MM-DDZ*
- *YYYY-MM-DD+hh:mm*
- *YYYY-MM-DD-hh:mm*

where:

- *YYYY* represents the year

- *MM* represents the month (as a number)
- *DD* represents the day
- Plus (+) or minus (-) is a positive or negative time zone offset
- *hh* represents the hours
- *mm* represents the number minutes that the time zone differs from GMT (UTC)
- Z indicates that the time is in the GMT time zone

Data Types

- Input data type: *xs:string?*
- Returned data type: *xs:date?*

Notes

The representation for date is the leftmost representation for dateTime: *YYYY-MM-DD+hh:mm* with an optional following time zone indicator (Z).

Liquid Data supports this year range: 0000–9999.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:date("2002-07-15")` returns a date value corresponding to July 15th, 2002 in the current time zone.
- `xf:date("2002-07-15-08:00")` returns a date value corresponding to July 15th, 2002 in a timezone that is offset by -8 hours from GMT (UTC).
- `xf:date("2002-7-15")` generates a runtime error because the month is not specified with two digits.
- `xf:date("2002-07-15Z")` returns a date value corresponding to July 15th, 2002 in the GMT time zone.
- `xf:date("2002-02-31")` generates a runtime error because the string (02-31) does not represent a valid date.

xf:dateTime

Returns a dateTime value from a source value, which must contain a date and time in one of these formats:

- *YYYY-MM-DDThh:mm:ss*
- *YYYY-MM-DDThh:mm:ssZ*
- *YYYY-MM-DDThh:mm:ss+hh:mm*
- *YYYY-MM-DDThh:mm:ss-hh:mm*

where the following is true:

- *YYYY* represents the year
- *MM* represents the month (as a number)
- *DD* represents the day
- T is the date and time separator
- *hh* represents the number of hours
- *mm* represents the number of minutes
- *ss* represents the number of seconds
- Plus (+) or minus (-) is a positive or negative time zone offset
- *hh* represents the hours
- *mm* represents the number minutes that the time zone differs from GMT (UTC)
- Z indicates that the time is in the GMT time zone

Data Types

- Input data type: *xs:string?*
- Returned data type: *xs:dateTime?*

Notes

Returns a date and time in *YYYY-MM-DD*T+*hh:mm:ss* format.

This expression can be preceded by an optional leading minus (-) sign to indicate a negative number. If the sign is omitted, positive (+) is assumed.

Use additional digits to increase the precision of fractional seconds if desired. The format ss.ss... with any number of digits after the decimal point is supported. Fractional seconds are optional.

Liquid Data supports this year range: 0000–9999.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:dateTime("2002-07-15T21:09:44")` returns a date value corresponding to July 15th, 2002 at 9:09PM and 44 seconds in the current time zone.
- `xf:dateTime("2002-07-15T21:09:44.566")` returns a date value corresponding to July 15th, 2002 at 9:09PM and 44.566 seconds in the current time zone
- `xf:dateTime("2002-07-15T21:09:44-08:00")` returns a date value corresponding to July 15th, 2002 at 9:09PM and 44 seconds, in a time zone that is offset by -8 hours from GMT (UTC).
- `xf:dateTime("2002-7-15T21:09:44")` generates a runtime error because the month is not specified using two digits
- `xf:dateTime("2002-07-15T21:09:44Z")` returns a date value corresponding to July 15th, 2002 at 9:09PM and 44 seconds, in the GMT timezone

---

xf:get-day-from-date

Returns an integer value representing the day identified in *date*.

Data Types

- Input data type: *xs:date?*
- Returned data type: xs:*integer?*

Notes

The day value ranges from 1 to 31.

If the source value is an empty list, the function returns an empty list.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:get-day-from-date(xf:date("2002-07-15"))` returns the integer value `15`.
- `xf:get-hours-from-dateTime(())` returns an empty list ().

---

xf:get-day-from-dateTime

Returns an integer value representing the day identified in *dateTime*.

Data Types

- Input data type: *xs:dateTime?*
- Returned data type: xs:*integer?*

Notes

The day value ranges from 1 to 31.

If the source value is an empty list, the function returns an empty list.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:get-day-from-dateTime(xf:dateTime("2004-01-07T21:09:44"))` returns the integer value `7`.
- `xf:get-hours-from-dateTime(())` returns an empty list ().

---

xf:get-hours-from-dateTime

Returns an integer value representing the hour identified in *dateTime*.

Data Types

- Input data type: *xs:dateTime?*
- Returned data type: xs:*integer?*

Notes

The hour value ranges from 0 to 23.

If the source value is an empty list, the function returns an empty list.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:get-hours-from-dateTime(xf:dateTime("2002-07-15T21:09:44"))` returns the integer value 21.
- `xf:get-hours-from-dateTime(())` returns an empty list ().

---

xf:get-hours-from-time

Returns an integer representing the hour identified in *time*.

Data Types

- Input data type: *xs:time?*
- Returned data type: *xs:integer?*

Notes

The hour value ranges from 0 to 23, inclusive.

If the source value is an empty list, the function returns an empty list.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:get-hours-from-time(xf:time("21:09:44"))` returns the integer value 21.
- `xf:get-hours-from-time(())` returns an empty list ().

---

xf:get-minutes-from-dateTime

Returns an integer value representing the minutes identified in *dateTime*.

Data Types

- Input data type: *xs:dateTime?*
- Returned data type: *xs:integer?*

Notes

Returns an integer value representing the minute identified in the source value. The minute value ranges from 0 to 59, inclusive.

If the source value is an empty list, the function returns the empty list.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:get-minutes-from-dateTime(xf:dateTime("2002-07-15T21:09:44"))` returns the integer value 9.
- `xf:get-minutes-from-dateTime(())` returns an empty list ().

---

xf:get-minutes-from-time

Returns an integer value representing the minutes identified in *time*.

Data Types

- Input data type: *xs:time?*
- Returned data type: *xs:integer?*

Notes

The minute value ranges from 0 to 59.

If the source value is an empty list, the function returns an empty list.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:get-minutes-from-time(xf:time("21:09:44"))` returns the integer value 9.
- `xf:get-minutes-from-time(())` returns an empty list ().

---

xf:get-month-from-date

Returns an integer value representing the month identified in *date*.

Data Types

- Input data type: *xs:date?*
- Returned data type: *xs:integer?*

Notes

Returns an integer value representing the month identified in the source value. The month value ranges from 1 to 12, inclusive.

If the source value is an empty list, the function returns the empty list.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:get-month-from-date(xf:date("2004-01-07"))` returns the integer value `1`.
- `xf:get-month-from-date(())` returns an empty list ().

---

xf:get-month-from-dateTime

Returns an integer value representing the month identified in *dateTime*.

Data Types

- Input data type: *xs:dateTime?*
- Returned data type: *xs:integer?*

Notes

The month value ranges from 1 to 12.

If the source value is an empty list, the function returns an empty list.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:get-month-from-dateTime(xf:dateTime("2004-01-07T21:09:44"))` returns the integer value `1`.
- `xf:get-month-from-dateTime(())` returns an empty list ().

---

xf:get-seconds-from-dateTime

Returns an integer value representing the seconds identified in *dateTime*.

Data Types

- Input data type: *xs:dateTime?*
- Returned data type: *xs:integer?*

Notes

The seconds value ranges from 0 to 60.999. The precision (number of digits) of fractional seconds depends on the relevant facet of the argument.

The value can be greater than 60 seconds to accommodate occasional leap seconds used to keep human time synchronized with the rotation of the planet.

If the source value is an empty list, the function returns an empty list.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:get-seconds-from-dateTime(xf:dateTime("2002-07-15T21:09:44"))` returns the integer value 44.
- `xf:get-seconds-from-dateTime(())` returns an empty list ().

xf:get-seconds-from-time

Returns an integer value representing the seconds identified in *time*.

Data Types:

- Input data type: *xs:time?*
- Returned data type: *xs:integer?*

Notes

The seconds value ranges from 0 to 60.999. The precision (number of digits) of fractional seconds depends on the relevant facet of the argument.

The value can be greater than 60 seconds to accommodate occasional leap seconds used to keep human time synchronized with the rotation of the planet.

If the source value is an empty list, the function returns an empty list.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:get-seconds-from-time(xf:time("21:09:44"))` returns the integer value 44.
- `xf:get-seconds-from-time(())` returns an empty list ().

---

xf:get-year-from-date

Returns an integer value representing the year identified in *date*.

Data Types

- Input data type: *xs:date?*
- Returned data type: *xs:integer?*

Notes

The year value ranges from 1000 to 999999.

If the source value is an empty list, the function returns an empty list.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:get-year-from-date(xf:date("2004-01-07"))` returns the integer value `2004`.
- `xf:get-year-from-date(())` returns an empty list ().

---

xf:get-year-from-dateTime

Returns an integer value representing the year identified in *dateTime*.

Data Types:

- Input data type: *xs:dateTime?*
- Returned data type: *xs:integer?*

Notes

The year value ranges from 1000 to 999999.

If the source value is an empty list, the function returns an empty list.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:get-year-from-dateTime(xf:dateTime("2002-07-15T21:09:44"))` returns the integer value `2004`.
- `xf:get-year-from-dateTime(())` returns an empty list ().

---

xf:time

Returns a tine from a source value, which must contain the time in one of these formats:

- *hh:mm:ss*
- *hh:mm:ssZ*
- *hh:mm:ss+hh:mm*
- *hh:mm:ss-hh:mm*

where the following is true:

- *hh* represents the number of hours
- *mm* represents the number of minutes
- *ss* represents the number of seconds
- Plus (+) or minus (-) is a positive or negative time zone offset
- *hh* represents the number of hours that the time zone differs from GMT (UTC)
- *mm* represents the number of minutes that the time zone differs from GMT (UTC)
- Z indicates that the time is in the GMT time zone

Data Types

- Input data type: *xs:string?*
- Returned data type: *xs:time?*

Notes

Liquid Data generates an error if it cannot parse the string successfully.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:time("22:04:22")` returns a time value corresponding to 10:04PM and 22 seconds in the current time zone.

- `xf:time("22:04:22.343")` returns a time value corresponding to 10:04PM and 22.343 seconds, in the current time zone.
- `xf:time("22:04:22-08:00")` returns a time value corresponding to 10:04PM and 22 seconds in a time zone that is offset by -8 hours from GMT (UTC).
- `xf:time("22:4:22")` generates a runtime error because the minutes are not specified with two digits.
- `xf:time("22:04:22Z")` returns a time value corresponding to 10:04PM and 22 seconds in the GMT time zone.

---

xfext:date-from-dateTime

Can be used to convert a dataTime to a date. Returns the leftmost date portion of a dateTime value.

Data Types

- Input data type: *xs:dateTime?*
- Returned data type: *xs:date?*

Notes

This is an extended function. It has an xfext: prefix identifier (namespace), which is the extension to the standard XQuery function namespace (xf:). For more information about extended functions, see "Naming Conventions" on page 3-2. For more information about valid formats for dateTime, see "xf:dateTime" on page 3-49.

XQuery Specification Compliance

Liquid Data supports date-from-dateTime as an extended function.

Examples

- `xfext:date-from-dateTime(xf:dateTime("2002-07-15T21:09:44"))` returns a date value corresponding to July 15th, 2002 in the current time zone.
- `xfext:date-from-dateTime(())` returns an empty list ().

---

xfext:date-from-string-with-format

Returns the right-most date portion of a dateTime value according to the pattern specified by Parameter1. For more information, see "Date and Time Patterns" on page 3-7.

Data Types

- Parameter1 data type: *xs:string?*
- Parameter2 data type: *xs:string?*
- Returned data type: *xs:date?*

Notes

This is an extended function. It has an xfext: prefix identifier (namespace), which is the extension to the standard XQuery function namespace (xf:). For more information about extended functions, see "Naming Conventions" on page 3-2.

XQuery Specification Compliance

Liquid Data supports `date-from-string-with-format` as an extended function.

Examples

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

xfext:date-to-string-with-format

Returns the date as a string formatted according to the pattern specified by Parameter1. For more information on the date patterns, see "Date and Time Patterns" on page 3-7.

Data Types

- Parameter1 data type: *xs:string?*
- Parameter2 data type: *xs:date?*
- Returned data type: *xs:string?*

Notes

This is an extended function. It has an xfext: prefix identifier (namespace), which is the extension to the standard XQuery function namespace (xf:). For more information about extended functions, see "Naming Conventions" on page 3-2.

XQuery Specification Compliance

Liquid Data supports `date-to-string-with-format` as an extended function.

Examples

- `xfext:date-to-string-with-format("yy-dd-mm", xf:date("2004-07-15"))` returns the string `04-15-07`.
- `xfext:date-to-string-with-format("yyyy-mm-dd", xf:date("2004-07-15"))` returns the string `2004-07-15`.

xfext:dateTime-from-string-with-format

Returns a new dateTime value from a string source value according to the pattern specified by Parameter1.

Data Types

- Parameter1 data type: *xs:string?*
- Parameter2 data type: *xs:string?*
- Returned data type: *xs:dateTime?*

Notes

This is an extended function. It has an xfext: prefix identifier (namespace), which is the extension to the standard XQuery function namespace (xf:).

For more information about extended functions, see "Naming Conventions" on page 3-2, and see "Date and Time Patterns" on page 3-7.

XQuery Specification Compliance

Liquid Data supports `dateTime-from-string-with-format` as an extended function.

Examples

- `xfext:dateTime-from-string-with-format("yyyy-MM-dd G", "2002-06-22 AD")` returns the specified date, 12:00:00AM in the current time zone.
- `xfext:dateTime-from-string-with-format("yyyy-MM-dd 'at' hh:mm", "2002-06-22 at 11:04")` returns the specified date, 11:04:00AM in the current time zone.
- `xfext:dateTime-from-string-with-format("yyyy-MM-dd", "2002-July-22")` generates an error because the date string does not match the specified format.
- `xfext:dateTime-from-string-with-format("yyyy-MMM-dd", "2002-July-22")` returns 12:00:00AM in the current time zone.

xfext:dateTime-to-string-with-format

Returns the *dateTime* as a string formatted according to the pattern specified by Parameter1. For more information on the date patterns, see "Date and Time Patterns" on page 3-7.

Data Types

- Parameter1 data type: *xs:string?*
- Parameter2 data type: *xs:dateTime?*
- Returned data type: *xs:string?*

Notes

This is an extended function. It has an xfext: prefix identifier (namespace), which is the extension to the standard XQuery function namespace (xf:). For more information about extended functions, see "Naming Conventions" on page 3-2.

XQuery Specification Compliance

Liquid Data supports `dateTime-to-string-with-format` as an extended function.

Examples

- `xfext:dateTime-to-string-with-format("dd MMMM yyyy hh:mm a G", xf:dateTime("2004-01-07T22:09:44"))` returns the string `07 January 2004 10:09 PM AD`.
- `xfext:dateTime-to-string-with-format("MM-dd-yyyy", xf:dateTime("2004-01-07T22:09:44"))` returns the string `01-07-2004`.

xfext:time-from-dateTime

Returns the time from *dateTime*.

Data Types

- Input data type: *xs:dateTime?*
- Returned data type: *xs:time?*

Notes

This is an extended function. It has an xfext: prefix identifier (namespace), which is the extension to the standard XQuery function namespace (xf:). For more information about extended functions, see "Naming Conventions" on page 3-2. For more information about valid formats for dateTime, see "xf:dateTime" on page 3-49.

XQuery Specification Compliance

Liquid Data supports `time-from-dateTime` as an extended function.

Examples

- `xfext:time-from-dateTime(xf:dateTime("2002-07-15T21:09:44"))` returns a date value corresponding to 9:09:44PM in the current time zone.
- `xfext:time-from-dateTime(())` returns an empty list ().

xfext:time-from-string-with-format

Returns a new time value from a string source value according to the pattern specified by Parameter1.

Data Types

- Parameter1 data type: *xs:string?*
- Parameter2 data type: *xs:string?*
- Returned data type: *xs:time?*

Notes

This is an extended function. It has an xfext: prefix identifier (namespace), which is the extension to the standard XQuery function namespace (xf:).

For more information about extended functions, see "Naming Conventions" on page 3-2, and see "Date and Time Patterns" on page 3-7.

XQuery Specification Compliance

Liquid Data supports `time-from-string-with-format` as an extended function.

Examples

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

xfext:time-to-string-with-format

Returns the *time* as a string formatted according to the pattern specified by Parameter1. For more information on the date patterns, see "Date and Time Patterns" on page 3-7.

Data Types

- Parameter1 data type: *xs:string?*
- Parameter2 data type: *xs:time?*
- Returned data type: *xs:string?*

Notes

This is an extended function. It has an xfext: prefix identifier (namespace), which is the extension to the standard XQuery function namespace (xf:). For more information about extended functions, see "Naming Conventions" on page 3-2.

XQuery Specification Compliance

Liquid Data supports `time-to-string-with-format` as an extended function.

Examples

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

Logical Operators

XQuery has operators that are specific to logical operations. The following logical operators are available:

- and
- or

and

The result is `true` if both values are `true`, and `false` if one of the values is `false`.

Data Types

- Parameter1 data type: *xs:boolean?*
- Parameter2 data type: *xs:boolean?*
- Returned data type: *xs:boolean?*

Notes

This is a boolean operator that you can use as a function to return a `true` or `false` result.

The arguments and return type are all boolean.

The following table shows how Liquid Data determines the result. The leftmost column contains the possible values of the first parameter; the top row contains the possible values of the second parameter.

	true	false	( )
true	true	false	false
false	false	false	false

XQuery Specification Compliance

- Liquid Data does not support error values.
- Liquid Data does not support a list of nodes as an input parameter to a boolean operator.

Examples

- `xf:true() and xf:true()` returns the boolean value `true`.
- `xf:true() and xf:false()` returns the boolean value `false`.
- `xf:false() and xf:false()` returns the boolean value `false`.
- `xf:true() and (<a/>, <b/>)` generates a compile-time error because lists are not supported as input parameters to boolean operators.
- `xf:false() and "false"` generates a compile-time error because the second parameter is not a boolean value.

or

The result is false if both values are false and true if at least one of the values is true. Parameter2 is not evaluated if Parameter1 is true.

Data Types

- Parameter1 data type: *xs:boolean?*
- Parameter2 data type: *xs:boolean?*
- Returned data type: *xs:boolean?*

Notes

This is a boolean operator that you can use as a function to return a `true` or `false` result.

The arguments and return type are all boolean.

The following table shows how Liquid Data determines the result. The leftmost column contains the possible values of the first parameter; the top row contains the possible values of the second parameter

	true	false	( )
true	true	true	true
false	true	false	false
( )	true	false	false

XQuery Specification Compliance

- Liquid Data does not support error values.
- Liquid Data does not support a list of nodes as an input parameter to a boolean operator.

Examples

- `xf:true() or xf:true()` returns the boolean value true.
- `xf:true() or xf:false()` returns the boolean value true.
- `xf:false() or xf:false()` returns the boolean value false.
- `xf:true() or (<a/>, <b/>)` generates a compile-time error because lists are not supported as parameters to boolean operators.

- `xf:false() or "false"` generates a compile-time error because the second parameter is not a boolean value.

Numeric Operators

XQuery has operators that are specific to numeric operations. The following numeric operators are available:

- * (multiply)
- + (add)
- - (subtract)
- div
- mod

* (multiply)

Returns the arithmetic product of the operands: ($operand1*$operand2).

Data Types

- Parameter1 data type: *xs:anyValue?*
- Parameter2 data type: *xs:anyValue?*
- Returned data type: *xs:anyValue?*

Notes

This is a numeric operator that you can use as if it were a function to compute numeric results.

The operator accepts two numeric values as parameters, computes their product, and returns the result.

Liquid Data applies the following rules:

- If both parameters are promotable to xs:decimal, the operator returns their product as a decimal value.
- If both parameters are promotable to xs:float, the operator returns their product as a floating point value.

- If both parameters are promotable to xs:double, the operator returns their product as a double precision value.
- Otherwise, an error occurs because one of the parameters is not a number.

XQuery Specification Compliance

- Liquid Data supports only numeric multiplication (op:numeric-multiply) and no other backup functions. It does not support values, such as xs:yearMonthDuration and xs:dayTimeDuration.
- Liquid Data does not support not-a-number (NaN) or the negative and positive infinity values -INF and INF.

Examples

- `12 * 3` returns the decimal value 36.
- `xf:integer("1") * 3.1` returns the decimal value 3.1.
- `"abc" * "cde"` generates a compile-time error because the operator can be used only with numbers.

+ (add)

Returns the arithmetic sum of the operands: ($operand1+$operand2).

Data Types

- Parameter1 data type: *xs:anyValue?*
- Parameter2 data type: *xs:anyValue?*
- Returned data type: *xs:anyValue?*

Notes

This is a numeric operator that you can use as if it were a function to compute numeric results.

The operator accepts two numeric values as parameters, computes their sum, and returns the result.

Liquid Data applies the following rules:

- If both parameters are promotable to xs:decimal, the operator returns their sum as a decimal value.

- If both parameters are promotable to xs:float, the operator returns their sum as a floating point value.
- If both parameters are promotable to xs:double, the operator returns their sum as a double precision value.
- Otherwise, an error occurs because one of the parameters is not a number.

XQuery Specification Compliance

- Liquid Data supports only numeric multiplication (op:numeric-add) and no other backup functions. It does not support values, such as xs:yearMonthDuration and xs:dayTimeDuration.
- Liquid Data does not support not-a-number (NaN) or the negative and positive infinity values -INF and INF.

Examples

- `20 + 1` returns the decimal value 21.
- `xf:integer("1") + 3.1` returns the decimal value 4.1.
- `"abc" + "cde"` generates a compile-time error because the operator can only be used with numbers.

- (subtract)

Returns the arithmetic difference of the operands: ($operand1-$operand2).

Data Types

- Parameter1 data type: *xs:anyValue?*
- Parameter2 data type: *xs:anyValue?*
- Returned data type: *xs:anyValue?*

Notes

This is a numeric operator that you can use as if it were a function to compute numeric results.

Liquid Data applies the following rules:

- If both parameters are promotable to xs:decimal, the operator returns their difference as a decimal value.
- If both parameters are promotable to xs:float, the operator returns their difference as a floating point value.
- If both parameters are promotable to xs:double, the operator returns their difference as a double precision value.
- Otherwise, an error occurs because one of the parameters is not a number.

XQuery Specification Compliance

- Liquid Data supports only numeric multiplication (op:numeric-subtract) and no other backup functions. It does not support values, such as xs:yearMonthDuration and xs:dayTimeDuration.
- Liquid Data does not support not-a-number (NaN) or the negative and positive infinity values -INF and INF.

Examples

- `20 - 1` returns the decimal value 19.
- `xf:integer("1") - 3.1` returns the decimal value -2.1.
- `"abc" - "cde"` generates a compile-time error because the operator can only be used with numbers.

---

div

Returns the arithmetic quotient of the operands ($operand1/$operand2).

Data Types

- Parameter1 data type: *xs:anyValue?*
- Parameter2 data type: *xs:anyValue?*
- Returned data type: *xs:anyValue?*

Notes

This is a numeric operator that you can use as if it were a function to compute numeric results.

Liquid Data applies the following rules:

- If both parameters are promotable to xs:decimal, the operator returns their quotient as a decimal value.
- If both parameters are promotable to xs:float, the operator returns their quotient as a floating point value.
- If both parameters are promotable to xs:double, the operator returns their quotient as a double precision value.
- Otherwise, an error occurs because one of the parameters is not a number.

XQuery Specification Compliance

- Liquid Data supports only numeric multiplication (op:numeric-divide) and no other backup functions. It does not support values, such as xs:yearMonthDuration and xs:dayTimeDuration.
- Liquid Data does not support not-a-number (NaN) or the negative and positive infinity values -INF and INF.

Examples

- `2 div 5` returns the decimal value 0.
- `3 div 5` returns the decimal value 1.
- `4 div "abc"` generates a compile-time error because the operator can only be used with numbers.

---

mod

Returns the remainder after dividing the first operand by the second operand: ($operand1 mod $operand2).

Data Types

- Parameter1 data type: *xs:anyValue?*
- Parameter2 data type: *xs:anyValue?*
- Returned data type: *xs:anyValue?*

Notes

This is a numeric operator that you can use as if it were a function to compute numeric results.

Liquid Data applies the following rules:

- If both parameters are promotable to xs:decimal, the operator returns the remainder as a decimal value.
- If both parameters are promotable to xs:float, the operator returns the remainder as a floating point value.
- If both parameters are promotable to xs:double, the operator returns the remainder as a double precision value.
- Otherwise, an error occurs because one of the parameters is not a number.

XQuery Specification Compliance

Liquid Data does not support not-a-number (NaN) or the negative and positive infinity values -INF and INF.

Examples

- `2 mod 5` returns the decimal value 2.
- `3 mod 5` returns the decimal value -2.
- `4 mod "abc"` generates a compile-time error because the operator can only be used with numbers.

Numeric Functions

Numeric functions operate on numeric data types. The following numeric functions are available:

- xf:ceiling
- xf:floor
- xf:round
- xfext:decimal-round
- xfext:decimal-truncate

xf:ceiling

Returns the smallest (closest to negative infinity) integer that is not smaller than the source value.

Data Types

- Input data type: *xs:double*?
- Returned data type: *xs:integer?*

Notes

If the argument is an empty list, the function returns an empty list.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:ceiling(38.3)` returns the integer value 39.
- `xf:ceiling(38)` returns the integer value 38.
- `xf:ceiling(-3.3)` returns the integer value -3.
- `xf:ceiling("38.3")` generates a compile-time error because the parameter is a string and not a numeric value.

---

xf:floor

Returns the largest (closest to positive infinity) integer that is not greater than the source value.

Data Types

- Input data type: *xs:double?*
- Returned data type: xs:*integer?*

Notes

If the argument is an empty list, the function returns an empty list.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:floor(38.3)` returns the integer value 38.
- `xf:floor(38)` returns the integer value 38.
- `xf:floor(-3.3)` returns the integer value -4.
- `xf:floor("38.3")` generates a compile-time error because the parameter is a string and not a numeric value.

---

xf:round

Returns the integer that is closest to the source value.

Data Types

- Input data type: *xs:double?*
- Returned data type: *xs:integer?*

Notes

Round(x) produces the same result as the Floor function(x+0.5). If there are two such numbers, returns the one that is closest to +INF.

If the argument is +INF, returns +INF.

If the argument is -INF, returns -INF.

If the argument is +0, returns +0.

If the source value is an empty list, the function returns an empty list.

XQuery Specification Compliance

Liquid Data does not support not-a-number (NaN) or -0.

Examples

- `xf:round(3)` returns the integer value 3.

- `xf:round(3.3)` returns the integer value 3.
- `xf:round(3.5)` returns the integer value 4.
- `xf:round(3.7)` returns the integer value 4.
- `xf:round(-3.3)` returns the integer value -3.
- `xf:round(-3.5)` returns the integer value -3.
- `xf:round(-3.7)` returns the integer value -4.
- `xf:round(-0)` returns the integer value 0.
- `xf:round("3.3")` generates an error because the parameter is a string and not a numeric value.

---

xfext:decimal-round

Returns a decimal value rounded to the specified precision (scale).

Data Types

- dec - Input data type: *xs:decimal?*
- scale - Input data type: *xs:integer?*
- Returned data type: *xs:decimal?*

Notes

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

XQuery Specification Compliance

This is an extended function and is not part of the XQuery specification.

Examples

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

---

xfext:decimal-truncate

Returns a decimal value truncated to the specified precision (scale).

Data Types

- dec - Input data type: *xs:decimal?*
- scale - Input data type: *xs:integer?*
- Returned data type: *xs:decimal?*

Notes

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

XQuery Specification Compliance

This is an extended function and is not part of the XQuery specification.

Examples

- `xfext:decimal-truncate(127.444, 2)` returns 127.44.
- `xfext:decimal-truncate(0.1234567, 6)` returns 0.123456.

Other Functions

The other functions folder is where the if-then-else function is in the Data View Builder.

xfext:if-then-else

The xfext:if-then-else function accepts the value of a boolean parameter to select one of two other input parameters.

Data Types

- Parameter1 data type: *xs:boolean?*
- Parameter2 data type: *xs:anyValue?*
- Parameter3 data type: *xs:anyValue?*
- Returned data type: *xs:anyValue*

Notes

The If-then-else function is an extended function. For more information about extended functions, see "Naming Conventions" on page 3-2.

Liquid Data examines the value of the first parameter. If the condition is true, Liquid Data returns the value of the second parameter (then). If the condition is false, Liquid Data returns the value of the third parameter (else). If the returned condition is not a boolean value, Liquid Data generates an error.

XQuery Specification Compliance

This is an extended function. Liquid Data converts it to an XQuery if-then-else expression.

Examples

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

Sequence Functions

A sequence is an ordered collection of zero or more items. An item may be a node or a simple typed value. Therefore, a sequence can be an ordered collection of nodes, a collection of simple typed values, or any mix of nodes and simple typed values. Sequences may not contain other sequences but may contain duplicate items. There is no difference between a single item, such as a node or a simple typed value, and a sequence containing that single item.

xf:distinct-values

If the source value contains only nodes, the function removes duplicates and returns a subset of unique values.

Data Types

- Input data type: *xsext:item**
- Returned data type: *xsext:anyValue**

Notes

The Liquid Data `xf:distinct-values` function varies from the standard XQuery function by removing duplicates from the result.

If the source value is an empty list, the function returns an empty list.

XQuery Specification Compliance

- Liquid Data does not support the distinct-values format that accepts collations.
- Liquid Data uses the eq operator instead of xf:deep-equal to identify duplicates.
- Liquid Data does not support duration values.

Examples

- `xf:distinct-values(("a", "b", "c", "b"))` returns the string `abc`.
- `xf:distinct-values((<x>a</x>, <x>b</x>, <x>c</x>, <x>b</x>))` returns the string sequence (<x>a</x>, <x>b</x>, <x>c</x>).
- `xf:distinct-values(("a", <x>b</x>, <x>c</x>, "b"))` generates a compile-time error because the list contains mixed nodes and values.

xf:empty

Returns true if the specified list of items is empty; otherwise, returns false.

Data Types

- Input data type: *xsext:item**
- Returned data type: *xs:boolean?*

XQuery Specification Compliance

Liquid Data supports an optional boolean returned value.

Examples

- `xf:empty((1, 2, 3))` returns the boolean value `false`.
- `xf:empty(1)` returns the boolean value `false`.
- `xf:empty(())` returns the boolean value `true`.

---

xf:subsequence (format 1)

Returns the contiguous sequence of items described by Parameter 1 beginning at the position indicated by the Parameter 2 and continuing until the end of the sequence.

Data Types

- Parameter1 data type: *xsext:item**
- Parameter2 data type: *xs:integer*
- Returned data type: *xsext:item**

Notes

The first item of a sequence is located at position 1, not position 0.

If you omit the length parameter, the function returns all items up to the end of the source sequence.

If the starting location is greater than the number of items in the sequence, the function returns an empty list.

If the item list is empty, Liquid Data returns an empty list.

XQuery Specification Compliance

- Liquid Data supports xs:integer instead of xs:decimal as the starting location and length parameters.
- If the starting location is greater than the length of the input sequence, Liquid Data returns an empty list instead of generating an error.

Examples

- `xf:subsequence(("a", "b", "c", "d", "e"), 2)` returns the string value `bcde`.
- `xf:subsequence("abcde", 2)` returns the string value `bcde`.
- `xf:subsequence("abcde", 6)` returns the empty string "".
- `xf:subsequence("abcde", 2, 3)` returns the string value `bcd`.
- `xf:subsequence("abcde", 2, 10)` returns the string value `bcde`.
- `xf:subsequence("abcde", ())` returns an empty list ().

---

xf:subsequence (format 2)

Returns the contiguous sequence of items described by Parameter 1 beginning at the position indicated by the Parameter 2 and continuing for the number of items indicated by the value of Parameter 3.

Data Types

- Parameter1 data type: *xsext:item**
- Parameter2 data type: *xs:integer*
- Parameter3 data type: *xs:integer*
- Returned data type: *xsext:item**

Notes

The value of Parameter 2 can be greater than the number of items in the value of Parameter 1, in which case the subsequence includes items to the end of Parameter 3.

If the sum of the starting location and the length parameter is greater than the length of the source sequence, the function returns all items up to the end of the sequence.

The first item of a sequence is located at position 1, not position 0.

If the starting location is greater than the number of items in the sequence, the function returns an empty list.

If the item list is an empty list, Liquid Data returns an empty list.

Liquid Data is able to process either format of xf:subsequence. Adding a third parameter automatically invokes Format 2.

XQuery Specification Compliance

- cimal as the starting location and length parameters.
- If the starting location is greater than the length of the input sequence, Liquid Data returns an empty list instead of generating an error.

Examples

- `xf:subsequence(("a", "b", "c", "d", "e"), 2)` returns the string value `bcde`.
- `xf:subsequence("abcde", 2)` returns the string value `bcde`.
- `xf:subsequence("abcde", 6)` returns the empty string "".
- `xf:subsequence("abcde", 2, 3)` returns the string value `bcd`.
- `xf:subsequence("abcde", 2, 10)` returns the string value `bcde`.
- `xf:subsequence("abcde", ())` returns an empty list ().

String Functions

Strings from a character set may need to be sorted differently for different applications. You must consider the sort order when you invoke string comparisons. Some string functions will require understanding of the default sort order and any other special collation. The string functions are case sensitive. For more information, see the *Character Model for the World Wide Web 1.0*. The following string functions are available:

- xf:compare
- xf:concat
- xf:contains
- xf:ends-with
- xf:lower-case
- xf:starts-with
- xf:string-length
- xf:substring (format1)
- xf:substring (format 2)
- xf:substring-after
- xf:substring-before
- xf:upper-case
- xfext:match
- xfext:trim
- xfext:sql-like

xf:compare

Returns -1, 0, or 1, depending on whether the value of Parameter1 is less than (-1), equal to (0), or greater than (1)the value of Parameter2.

Data Types

- Parameter1 data type: *xs:string?*
- Parameter2 data type: *xs:string?*
- Returned data type: *xs:integer?*

Notes

If either argument is an empty list, the result is an empty list.

Liquid Data generates an error if either parameter is not a string.

XQuery Specification Compliance

Liquid Data does not support the xf:compare format that accepts collations.

Examples

- `xf:compare("a", "b")` returns the integer value -1.
- `xf:compare("a", "a")` returns the integer value 0.
- `xf:compare("b", "a")` returns the integer value 1.
- `xf:compare("a", 3)` generates a compile-time error because the second parameter is not a string.
- `xf:compare("a", ())` returns an empty list ().
- `xf:compare((), "a")` returns an empty list ().

xf:concat

Returns a string that concatenates Parameter1 with Parameter2.

Data Types

- Parameter1 data type: *xs:string?*
- Parameter2 data type: *xs:string?*
- Returned data type: *xs:string?*

Notes

The result string may not reflect Unicode or other W3C normalization.

Returns an empty string if the function has no arguments. If any argument is an empty list, it is treated as an empty string.

Liquid Data generates an error if either parameter is not a string.

XQuery Specification Compliance

Liquid Data does not support a variable number of parameters to be concatenated. Choose only two strings to concatenate with each operation.

Examples

- `xf:concat("a", "b")` returns the string value "ab."
- `xf:concat("a", xf:concat("b", "c"))` returns the string value "abc."
- `xf:concat("abc", ())` returns the string value "abc."
- `xf:concat((), "abc")` returns the string value "abc."
- `xf:concat((), ())` returns an empty list ().
- `xf:concat("a", 4)` generates a compile-time error because the second parameter is not a string.

xf:contains

Returns a boolean value of true or false indicating whether Parameter1 contains a string that is equal to Parameter2 at the beginning, at the end, or anywhere within Parameter1.

Data Types

- Parameter1 data type: *xs:string?*
- Parameter2 data type: *xs:string?*
- Returned data type: *xs:boolean?*

Notes

If the value of Parameter2 is a zero-length string, the function returns true. If the value of Parameter1 is a zero-length string and the value of Parameter2 is not a zero-length string, the function returns false.

If the value of Parameter1 or Parameter2 is an empty list, the function returns an empty list.

Liquid Data generates an error if either parameter is not a string.

XQuery Specification Compliance

Liquid Data does not support the xf:contains format that accepts collations.

Examples

- `xf:contains("abc", "a")` returns the boolean value true.
- `xf:contains("abc", "b")` returns the boolean value true.
- `xf:contains("abc", "c")` returns the boolean value true.
- `xf:contains("abc", "d")` returns the boolean value false.
- `xf:contains("abc", "")` returns the boolean value true.
- `xf:contains("abc", ())` returns an empty list ().
- `xf:contains((), "abc")` returns an empty list ().
- `xf:contains("abc", 4)` generates a compile-time error because the second parameter is not a string.

xf:ends-with

Returns a boolean value or true or false indicating whether Parameter1 ends with a string that is equal to Parameter2.

Data Types

- Parameter1 data type: *xs:string?*
- Parameter2 data type: *xs:string?*
- Returned data type: *xs:boolean?*

Notes

If Parameter2 is a zero-length string, then the function returns true. If Parameter1 is a zero-length string and Parameter2 is not a zero-length string, the function returns false.

If Parameter1 or Parameter2 is an empty list, the function returns an empty list.

Liquid Data generates an error if either parameter is not a string.

XQuery Specification Compliance

Liquid Data does not support the xf:ends-with format that accepts collations.

Examples

- `xf:ends-with("abc", "a")` returns the boolean value false.
- `xf:ends-with("abc", "b")` returns the boolean value false.
- `xf:ends-with("abc", "c")` returns the boolean value true.
- `xf:ends-with("abc", "d")` returns the boolean value false.
- `xf:ends-with("abc", "")` returns the boolean value true.
- `xf:ends-with("abc", ())` returns an empty list ().
- `xf:ends-with((), "abc")` returns an empty list ().
- `xf:ends-with("abc", 4)` generates a compile-time error because the second parameter is not a string.

xf:lower-case

Returns the value of the input string after translating every uppercase letter to its corresponding lower-case value.

Data Types

- Input data type: *xs:string?*
- Returned data type: *xs:string?*

Notes

Every uppercase letter that does not have a lower-case corresponding value and every character that is not an uppercase letter appears in the output in its original form.

If the source value is an empty list, the function returns an empty list.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:lower-case("ABc!D")` returns the string value `abc!d`.
- `xf:lower-case("")` returns the empty string "".
- `xf:lower-case(())` returns the empty list ().
- `xf:lower-case(4)` generates a compile-time error because the parameter is not a string.

---

xf:starts-with

Returns a boolean value or true or false indicating whether Parameter1 starts with a string that is equal to Parameter2.

Data Types

- Parameter1 data type: *xs:string?*
- Parameter2 data type: *xs:string?*

- Returned data type: *xs:boolean?*

Notes

If Parameter2 is a zero-length string, then the function returns true. If Parameter1 is a zero-length string and tParameter2 is not a zero-length string, the function returns false.

If Parameter1 or Parameter2 is an empty list, the function returns an empty list.

Liquid Data generates an error if either parameter is not a string.

XQuery Specification Compliance

Liquid Data does not support the xf:ends-with format that accepts collations.

Examples

- `xf:starts-with("abc", "a")` returns the boolean value true.
- `xf:starts-with("abc", "b")` returns the boolean value false.
- `xf:starts-with("abc", "c")` returns the boolean value false.
- `xf:starts-with("abc", "d")` returns the boolean value false.
- `xf:starts-with("abc", "")` returns the boolean value true.
- `xf:starts-with("abc", ())` returns the empty list ().
- `xf:starts-with((), "abc")` returns the empty list ().
- `xf:starts-with("abc", 4)` generates a compile-time error because the second parameter is not a string.

---

xf:string-length

Returns an integer equal to the number of characters in the input source string.

Data Types

- Input data type: *xs:string?*
- Returned data type: *xs:integer?*

Notes

If the source value is an empty list, the function returns an empty list.

Liquid Data generates an error if either parameter is not a string.

XQuery Specification Compliance

- Liquid Data treats xf:string as both a constructor and an accessor.
- Liquid Data supports only the string format that requires one node of any type as the input.
- Liquid Data accepts xsext:anyType input instead of a list of items.
- Liquid Data returns an optional string.
- Liquid Data does not recognize entities.

Examples

- `xf:string-length("abc")` returns the integer value 3.
- `xf:string-length("")` returns the integer value 0.
- `xf:string-length(())` returns the empty list ().
- `xf:string-length(4)` generates a compile-time error because the parameter is not a string.

---

xf:substring (format1)

Returns that part of the Parameter1 source string from the starting location specified by Parameter2.

Data Types

- Parameter1 data type: *xs:string?*
- Parameter2 data type: *xs:integer?*
- Returned data type: *xs:string?*

Notes

If the starting location is a negative value, or greater than the length of source string, an error occurs.

The first character of a string is located at position 1 (not position 0).

If Parameter1 or Parameter2 is an empty list, the function returns an empty list.

If you omit Parameter3, the function returns characters up to the end of the source string.

Liquid Data generates an error if Parameter1 is not a string or if the starting location is less than 1.

XQuery Specification Compliance

- Liquid Data supports xs:integer instead of xs:decimal as the starting location and length parameters.
- If the starting location is greater than the length of the input sequence, Liquid Data returns an empty list instead of generating an error.

---

xf:substring (format 2)

Returns that part of the Parameter1 source string from the starting location specified by Parameter2 and continuing for the number of characters equal to the length specified by Parameter3.

Data Types

- Parameter1 data type: *xs:string?*
- Parameter2 data type: *xs:integer?*
- Parameter3 data type: *xs:integer?*
- Returned data type: *xs:string?*

Notes

If the starting location is a negative value, or greater than the length of the source string, an error occurs.

The first character of a string is located at position 1 (not position 0).

If you omit length, the substring identifies characters to the end of the source string.

If length exceeds the number of characters in the source string, the function identifies only characters until the end of the source string.

If Parameter1, Parameter2, or Parameter3 is an empty list, the function returns an empty list.

Liquid Data generates an error if Parameter1 is not a string or if the starting location is less than 1.

Liquid Data is able to process either format of xf:substring. Adding a third parameter automatically invokes Format 2.

XQuery Specification Compliance

- Liquid Data supports xs:integer instead of xs:decimal as the starting location and length parameters.
- If the starting location is greater than the length of the input sequence, Liquid Data returns an empty list instead of generating an error.

---

xf:substring-after

Returns that part of the Parameter1 source string that follows the first occurrence of those characters specified in Parameter2.

Data Types

- Parameter1 data type: *xs:string?*
- Parameter2 data type: *xs:string?*
- Returned data type: *xs:string?*

Notes

If Parameter2 is a zero-length string, the function returns the value of Parameter1. If Parameter1 is a zero-length string and Parameter2 is a zero-length string, the function returns a zero-length string.

If Parameter1 does not contain a string that is equal to Parameter2, the function returns a zero-length string.

If Parameter1 or Parameter2 is an empty list, the function returns an empty list.

XQuery Specification Compliance

Liquid Data does not support the xf:substring-after format that accepts collations.

Examples

- `xf:substring-after("abcde", "d")` returns the string value "e."
- `xf:substring-after("abcde", "")` returns the string value "abcde."

- `xf:substring-after("abcde", "x")` returns the empty string "".
- `xf:substring-after("abcde", ())` returns the empty list ().
- `xf:substring-after((), "x")` returns the empty list ().
- `xf:substring-after("abc34de", 3)` generates a compile-time error because the second parameter is not a string.

---

xf:substring-before

Returns that part of the Parameter1 source string that precedes the first occurrence of those characters specified in Parameter2.

Data Types

- Parameter1 data type: *xs:string?*
- Parameter2 data type: *xs:string?*
- Returned data type: *xs:string?*

Notes

If Parameter2 is a zero-length string, the function returns the value of Parameter1. If Parameter1 is a zero-length string and Parameter2 is a zero-length string, the function returns a zero-length string.

If Parameter1 does not contain a string that is equal to Parameter2, the function returns a zero-length string.

If Parameter1 or Parameter2 is an empty list, the function returns an empty list.

XQuery Specification Compliance

Liquid Data does not support the xf:substring-before format that accepts collations.

Examples

- `xf:substring-before("abcde", "d")` returns the string value `abc`.
- `xf:substring-before("abcde", "")` returns the string value `abcde`.
- `xf:substring-after("abcde", "x")` returns the empty string "".

- `xf:substring-before("abcde", ())` returns an empty list ().
- `xf:substring-before((), "x")` returns an empty list ().
- `xf:substring-before("abc34de", 3)` generates a compile-time error because the second parameter is not a string.

---

xf:upper-case

Returns the value of the input string after translating every lower-case letter to its uppercase correspondent.

Data Types

- Input Parameter data type = *xs:string?*
- Returned data type: *xs:string?*

Notes

Every lower-case letter that does not have an uppercase corresponding value and every character that is not a lower-case letter appears in the output in its original form.

If the source value is an empty list, the function returns an empty list.

Liquid Data generates an error if the parameter is not a string.

XQuery Specification Compliance

Conforms to the current specification.

Examples

- `xf:upper-case("ABc!D")` returns the string value `ABC!D`.
- `xf:upper-case("")` returns the empty string "".
- `xf:upper-case(())` returns the empty list ().
- `xf:upper-case(4)` generates a compile-time error because the parameter is not a string.

---

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

Data Types

- source - Input data type: *xs:string?*
- regularExpression - Input data type: *xs:string?*
- Returned data type: *xs:int?*

Notes

The index of the first character of the input `source` is 1, the index of the second character is 2, and so on.

The `regularExpression` input uses a standard regular expression language. The regular expression language uses the following components:

Table 3-3 Regular expression syntax examples for the xfext:match function

Syntax Example	Description
	Characters
unicode	Matches the specified unicode character.
\	Used to escape metacharacters such as *, +, and ?.
\\	Matches a single backslash (\ ) character.
\0*nnn*	Matches the specified octal character.
\0xhh	Matches the specified 8-bit hexidecimal character.
\\uxhhh	Matches the specified 16-bit hexidecimal character.
\t	Matches an ASCII tab character.
\n	Matches an ASCII new line character.
\r	Matches an ASCII return character.
\f	Matches an ASCII form feed character.
	Simple Character Classes
[bc]	Matches the characters b or c.
[a-f]	Matches any character between a and f.
[^bc]	Matches any character except b and c.

Table 3-3 Regular expression syntax examples for the xfext:match function (Continued)

Syntax Example	Description
Predefined Character Classes	
.	Matches any character except the new line character.
\w	Matches a word character: an alphanumeric character or the underscore (_) character.
\W	Matches a non-word character.
\s	Matches a white space character.
\S	Matches a non-white space character.
\d	Matches a digit.
\D	Matches a non-digit.
Greedy Closures—match as many characters as possible	
A*	Matches expression A zero or more times.
A+	Matches expression A one or more times.
A?	Matches expression A zero or one times.
A(n)	Matches expression A exactly n times.
A(n,)	Matches expression A at least n times.
A(n, m)	Matches expression A between n and m times.
Reluctant Closures—match as few characters as possible (stops when a match is found)	
A*?	Matches expression A zero or more times.
A+?	Matches expression A one or more times.
A??	Matches expression A zero or one times.

Table 3-3 Regular expression syntax examples for the xfext:match function (Continued)

Syntax Example	Description
	Logical Operators
`AB`	Matches expression `A` followed by expression `B`.
`A|B`	Matches expression `A` or expression `B`.
`(A)`	Used for grouping expressions.

XQuery Specification Compliance

This is an extended function and is not part of the XQuery specification.

Examples

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

xfext:trim

Returns the value of the input string with leading and trailing white space removed from the string.

Data Types

- Input data type: *xs:string?*
- Returned data type: *xs:string?*

Notes

If the input string is an empty list, the function returns an empty list.

Liquid Data generates an error if the parameter is not a string.

XQuery Specification Compliance

The `xfext:trim` function is an extended function. For more information about extended functions, see "Naming Conventions" on page 3-2.

Examples

- `xfext:trim("abc")` returns the string value "abc"
- `xfext:trim(" abc ")` returns the string value "abc"
- `xfext:trim(())` returns the empty list ()
- `xfext:trim(5)` generates a compile-time error because the parameter is not a string

---

xfext:sql-like

Tests whether a string contains the specified pattern. Typically, this function is used as a condition for a query, similar to the SQL `LIKE` operator used in a predicate of SQL queries. Returns `TRUE` if the pattern is matched in the source expression, otherwise returns `FALSE`.

Data Types

- Parameter1 `source` Input data type: *xs:string?*
- Parameter2 `pattern` Input data type: *xs:string?*
- Parameter3 `escape` Input data type: *xs:string?*
- Returned data type: *xs:boolean?*

Notes

The percentage character (`%`) is a wildcard character representing a string of zero or more characters. The underscore character (_) is a is a wildcard character representing any single character.

Use the `xfext:sql-like` function to specify query conditions that satisfy a search pattern. For example, you can use the `xfext:sql-like` function to constrain a query that returns first names to return only first names that begin with the letter H.

The escape input parameter specifies an escape character. The escape character is needed to specify one of the wildcard characters (_ and `%`) in the search pattern.

XQuery Specification Compliance

The `xfext:sql-like` function is an extended function. For more information about extended functions, see "Naming Conventions" on page 3-2.

Examples

- `xfext: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`.
- `xfext: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`.
- `xfext: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%`.

Treat Functions

The `treat` functions process a source value as the argument and treat that source value as if it is the datatype in the treat function. These functions are used when mapping optional values (which do not have to have data associated with them) to mandatory values (which do have to have data associated with them). From the Query menu Automatic Treat As checkbox, you can set up the Data View Builder to automatically add `treat` functions when they are needed, or you can add them manually. Without the `treat` functions, some queries that attempt to map optional fields (for example, nullable relational database columns) to mandatory fields might fail.

Unlike the `cast` functions, the `treat` functions do not change the type of the input value; instead they ensure that an expression has the intended type when it is evaluated for query execution.

A typical use case is when you need to map elements from a nullable relational database column that you know do not contain any null values.

Another use case is where you need to map non-nullable (mandatory) elements to a function that produces optional (nullable) output. For example, if you map an `xf:string` type to a custom function that outputs an `xf:string?` type, and then map that output to an `xf:string` type, there will be a type mismatch which will cause the query to fail during compilation. The type mismatch is because the output type of the function is `xf:string?`, which mismatches `xf:string`. You can correct this by placing a `treat as xs:string` function betweenthe custom function and the output.

The following table describes Liquid Data data types that conform to the XQuery specification that you can use in `treat as` functions. For more information about data types, see the *XQuery 1.0 and XPath 2.0 Functions and Operators* specification. The following `treat as` functions are available:

- treat as xs:boolean
- treat as xs:byte
- treat as xs:date
- treat as xs:dateTime
- treat as xs:decimal
- treat as xs:double
- treat as xs:float
- treat as xs:int
- treat as xs:integer
- treat as xs:long
- treat as xs:short
- treat as xs:string
- treat as xs:time

treat as xs:boolean

Treats the input value as if it is a boolean value (true or false). Use to map optional boolean elements to mandatory boolean elements.

Data Types

- Input data type: *xs:anyValue?*
- Returned data type: *xs:anyValue?*

Notes

See "Treat Functions" on page 3-102.

XQuery Specification Compliance

Conforms to the current specification.

treat as xs:byte

Treats the input value as if it is a byte value. Use to map optional boolean elements to mandatory boolean elements.

Data Types

- Input data type: *xs:anyValue?*
- Returned data type: *xs:anyValue?*

Notes

See "Treat Functions" on page 3-102.

XQuery Specification Compliance

Conforms to the current specification.

---

treat as xs:date

Treats the input value as if it is a date value. Use to map optional boolean elements to mandatory boolean elements.

Data Types

- Input data type: *xs:anyValue?*
- Returned data type: *xs:anyValue?*

Notes

See "Treat Functions" on page 3-102.

XQuery Specification Compliance

Conforms to the current specification.

---

treat as xs:dateTime

Treats the input value as if it is a dateTime value. Use to map optional boolean elements to mandatory boolean elements.

Data Types

- Input data type: *xs:anyValue?*
- Returned data type: *xs:anyValue?*

Notes

See "Treat Functions" on page 3-102.

XQuery Specification Compliance

Conforms to the current specification.

---

treat as xs:decimal

Treats the input value as if it is a decimal value. Use to map optional boolean elements to mandatory boolean elements.

Data Types

- Input data type: *xs:anyValue?*
- Returned data type: *xs:anyValue?*

Notes

See "Treat Functions" on page 3-102.

XQuery Specification Compliance

Conforms to the current specification.

---

treat as xs:double

Treats the input value as if it is a double value. Use to map optional boolean elements to mandatory boolean elements.

Data Types

- Input data type: *xs:anyValue?*
- Returned data type: *xs:anyValue?*

Notes

See "Treat Functions" on page 3-102.

XQuery Specification Compliance

Conforms to the current specification.

---

treat as xs:float

Treats the input value as if it is a float value. Use to map optional boolean elements to mandatory boolean elements.

Data Types

- Input data type: *xs:anyValue?*
- Returned data type: *xs:anyValue?*

Notes

See "Treat Functions" on page 3-102.

XQuery Specification Compliance

Conforms to the current specification.

---

treat as xs:int

Treats the input value as if it is a int value. Use to map optional boolean elements to mandatory boolean elements.

Data Types

- Input data type: *xs:anyValue*
- Returned data type: *xs:int*

Notes

See "Treat Functions" on page 3-102.

XQuery Specification Compliance

Conforms to the current specification.

---

treat as xs:integer

Treats the input value as if it is a integer value. Use to map optional boolean elements to mandatory boolean elements.

Data Types

- Input data type: *xs:anyValue?*
- Returned data type: *xs:anyValue?*

Notes

See "Treat Functions" on page 3-102.

XQuery Specification Compliance

Conforms to the current specification.

---

treat as xs:long

Treats the input value as if it is a long value. Use to map optional boolean elements to mandatory boolean elements.

Data Types

- Input data type: *xs:anyValue?*
- Returned data type: *xs:anyValue?*

Notes

See "Treat Functions" on page 3-102.

XQuery Specification Compliance

Conforms to the current specification.

---

treat as xs:short

Treats the input value as if it is a short value. Use to map optional boolean elements to mandatory boolean elements.

Data Types

- Input data type: *xs:anyValue?*
- Returned data type: *xs:anyValue?*

Notes

See "Treat Functions" on page 3-102.

XQuery Specification Compliance

Conforms to the current specification.

---

treat as xs:string

Treats the input value as if it is a string value. Use to map optional boolean elements to mandatory boolean elements.

Data Types

- Input data type: *xs:anyValue?*
- Returned data type: *xs:anyValue?*

Notes

See "Treat Functions" on page 3-102.

XQuery Specification Compliance

Conforms to the current specification.

---

treat as xs:time

Treats the input value as if it is a time value. Use to map optional boolean elements to mandatory boolean elements.

Data Types

- Input data type: *xs:anyValue?*
- Returned data type: *xs:anyValue?*

Notes

See "Treat Functions" on page 3-102.

XQuery Specification Compliance

Conforms to the current specification.

CHAPTER 4

Supported Data Types

This section provides information about the data types supported in BEA Liquid Data for WebLogic. The following topics are included:

- JDBC Types in Liquid Data
  - java.sql.Types Data Types
  - JDBC Data Type Names
- Database-Specific Data Type Names
  - Oracle Data Type Names
  - Microsoft SQL Server Data Type Names
  - DB2 Data Type Names
  - Sybase Data Type Names
  - Informix Data Type Names

JDBC Types in Liquid Data

In relational databases, data types are described using two methods. The conventional way is to use a JDBC number. For example, an integer is 4, varchar is 12, a date is 91, an so on. These numbers are represented by constants in the `java.sql.Types` class, such as `Types.INTEGER = 4` and `Types.VARCHAR = 12`. This numbering system describes all the JDBC standardized types. However, there are many vendor-specific types, and most of them use the default JDBC number 1111, meaning "other." For this method, there is a name instead of a number associated with each type.

The Liquid Data query generation engine first looks at the JDBC number for a match. If no match occurs, then it uses the name. For example, if the number is 1111, then the query generation engine looks for a name. If there is no match found for either one, the query generation engine treats the column as a string.

Depending on the type of database you access, you need to map external database fields with a compatible data type when you invoke Liquid Data functions. You will notice that some external data types are not supported by Liquid Data. You may need to transform these data types to a supported type before you access that data in a query. The following tables can help you make these decisions.

java.sql.Types Data Types

The following table maps the `java.sql.Types` data type to the appropriate data type that you should use with Liquid Data.

Table 4-1 java.sql.Types and Liquid Data Equivalents

java.sql.Types Data Type	Liquid Data **Data Type**
`Types.ARRAY`	*not supported*
`Types.BIGINT`	`xs:long`
`Types.BINARY`	`xs:string`
`Types.BIT`	`xs:boolean`
`Types.BLOB`	*not supported*
`Types.CHAR`	`xs:string`
`Types.CLOB`	*not supported*
`Types.DATE`	`xs:date`

Table 4-1 java.sql.Types and Liquid Data Equivalents (Continued)

java.sql.Types Data Type	Liquid Data **Data Type**
Types.DECIMAL	xs:decimal
Types.DOUBLE	xs:double
Types.FLOAT	xs:double
Types.INTEGER	xs:integer
Types.JAVA_OBJECT	*not supported*
Types.LONGVARBINARY	xs:string
Types.LONGVARCHAR	xs:string
Types.NUMERIC	xs:decimal
Types.REAL	xs:float
Types.REF	xs:string
Types.SMALLINT	xs:short
Types.STRUCT	*not supported*
Types.TIME	xs:time
Types.TIMESTAMP	xs:dateTime
Types.TINYINT	xs:byte
Types.VARBINARY	xs:string
Types.VARCHAR	xs:string

JDBC Data Type Names

The following table maps the native JDBC Data Type name to Liquid Data data types.

Table 4-2 JDBC Data Types and Liquid Data Equivalents

JDBC Name	Liquid Data **Data Type**
ARRAY	*not supported*
BIGINT	xs:long
BINARY	xs:string
BIT	xs:boolean
BLOB	*not supported*
CHAR	xs:string
CLOB	*not supported*
DATE	xs:date
DEC	xs:decimal
DECIMAL	xs:decimal
DOUBLE	xs:double
FLOAT	xs:double
INTEGER	xs:integer
JAVA_OBJECT	*not supported*
LONGVARBINARY	xs:string
LONGVARCHAR	xs:string
NUM	xs:decimal
NUMERIC	xs:decimal
REAL	xs:float
REF	xs:string

Table 4-2 JDBC Data Types and Liquid Data Equivalents (Continued)

JDBC Name	Liquid Data **Data Type**
SMALLINT	xs:short
STRUCT	*not supported*
TIME	xs:time
TIMESTAMP	xs:dateTime
TINYINT	xs:byte
VARBINARY	xs:string
VARCHAR	xs:string

Database-Specific Data Type Names

This section includes tables showing the database-specific data type names and the corresponding Liquid Data data types. This section includes the following:

- Oracle Data Type Names
- Microsoft SQL Server Data Type Names
- DB2 Data Type Names
- Sybase Data Type Names

Oracle Data Type Names

The following table maps Oracle names to Liquid Data data types.

Table 4-3 Oracle Data Types and Liquid Data Equivalents

Oracle Name	Liquid Data **Data Type**
FLOAT	xs:float
BFILE	*not supported*
LONG	*not supported*
LONG RAW	*not supported*
NCHAR	xs:string
NCLOB	*not supported*
NUMBER	xs:decimal
NVARCHAR2	xs:string
RAW	xs:string
ROWID	xs:string
UROWID	*not supported*
VARCHAR2	xs:string

Microsoft SQL Server Data Type Names

The following table maps Microsoft SQL Server names to Liquid Data data types.

Table 4-4 Microsoft SQL Server Data Types and Liquid Data Equivalents

SQL Name	Liquid Data **Data Type**
DATETIME	xs:dateTime
IMAGE	*not supported*
INT	xs:integer
MONEY	xs:float
NTEXT	xs:string
NVARCHAR	xs:string
SMALLDATETIME	xs:dateTime
SMALLMONEY	xs:float
SQL_VARIANT	xs:string
UNIQUEIDENTIFIER	xs:string

DB2 Data Type Names

The following table maps DB2 data types to Liquid Data data types.

Table 4-5 IBM DB2 Data Types and Liquid Data Equivalents

DB2 Name	Liquid Data **Data Type**
`CHARACTER`	`xs:string`
`CHARACTER` (for bit data)	`xs:string`
`DATALINK`	`xs:string`
`LONG VARCHAR`	`xs:string`
`LONG VARCHAR` (for bit data)	`xs:string`
`VARCHAR` (for bit data)	`xs:string`

Sybase Data Type Names

The following table maps Sybase data types to Liquid Data data types.

Table 4-6 Sybase Data Types and Liquid Data Equivalents

Sybase Name	Liquid Data **Data Type**
`SYSNAME`	`xs:string`
`TEXT`	`xs:string`

Informix Data Type Names

The following table maps Informix data types to Liquid Data data types.

Table 4-7 Informix Data Types and Liquid Data Equivalents

Informix Name	Liquid Data **Data Type**
BLOB	*not supported*
BYTE	*not supported*
BOOLEAN	xs:boolean
CHAR(n)	xs:string
CHARACTER(n)	xs:string
CLOB	*not supported*
DATE	xs:date
DATETIME	xs:dateTime
DEC/DECIMAL	xs:decimal
DOUBLE PRECISION/FLOAT	xs:double
INT/INTEGER	xs:integer
INT8	xs:long
INTERVAL	*not supported*
LVARCHAR	xs:string
MONEY	xs:float
NCHAR	xs:string
NUMERIC	xs:decimal
REAL	xs:float
SMALLFLOAT	xs:float
SMALLINT	xs:short
TEXT	xs:string

Index

Symbols

A

B

C

D

E

F

W

X