



BEA Liquid Data for WebLogic™

Building Queries and Data Views

Copyright

Copyright © 2003 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

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

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

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

Trademarks or Service Marks

BEA, Jolt, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Manager, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic Personalization Server, BEA WebLogic Platform, BEA WebLogic Portal, BEA WebLogic Server, BEA WebLogic Workshop, BEA Liquid Data for WebLogic, and How Business Becomes E-Business are trademarks of BEA Systems, Inc.

All other trademarks are the property of their respective companies.

Building Queries and Data Views

Part Number	Date	Software Version
N/A	October 2002	1.0
N/A	April 2003	1.1

About This Document

Read this document to learn how to build and test queries in XQuery language that can retrieve real-time information from heterogeneous data sources using the BEA Liquid Data for WebLogic™ server.

This document describes how to use the Data View Builder to design and generate XML-based queries with the Builder drag-and-drop tools, functions, source and target schemas. The focus of this document is on how to use the Data View Builder to create queries in Liquid Data. Liquid Data accepts queries written in XQuery, which is an Extensible Markup Language (XML) Query language that adheres to the standards described by the World Wide Web Consortium (W3C). The XQuery standard, version 1.0, is the structured query language used by the Liquid Data server.

This document covers the following topics:

- [Chapter 1, “Overview and Key Concepts,”](#) introduces key concepts such as XQuery, ad hoc queries, and Builder-generated queries.
- [Chapter 2, “Starting the Builder and Touring the GUI,”](#) explains how to start the Data View Builder and provides graphical user interface (GUI) tour and reference.
- [Chapter 3, “Designing Queries,”](#) explains how to design a query using the Data View Builder to define source conditions; map source data to target schemas; use joins, unions, and functions; and how to apply explicit scope to a target schema for well-defined query results. Provides examples of building basic queries.
- [Chapter 4, “Optimizing Queries,”](#) describes some advanced concepts that can improve query performance and refine query output. It also has more information about using some Data View Builder features.
- [Chapter 5, “Testing Queries,”](#) describes how you run the query and view the results.

-
- [Chapter 9, “Query Cookbook,”](#) provides detailed examples about how to construct queries using some advanced techniques and functions.
 - [Chapter 6, “Using Data Views,”](#) has information and examples about saving and reusing data views as new query resources.
 - [Appendix A, “Functions Reference”](#) provides information about complete reference of the World Wide Web (W3C) functions supported in Liquid Data as *built-in functions*.
 - [Appendix B, “Supported Data Types,”](#) is a reference list of data types supported in Liquid Data.
 - [Appendix C, “Type Casting Reference,”](#) is a reference list of

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 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.
<i>italics</i>	Indicates emphasis or book titles.

Convention	Item
monospace text	<p>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.</p> <p><i>Examples:</i></p> <pre>#include <iostream.h> void main () the pointer psz chmod u+w * \tux\data\ap .doc tux.doc BITMAP float</pre>
monospace boldface text	<p>Identifies significant words in code.</p> <p><i>Example:</i></p> <pre>void commit ()</pre>
<i>monospace</i> <i>italic</i> text	<p>Identifies variables in code.</p> <p><i>Example:</i></p> <pre>String <i>expr</i></pre>
UPPERCASE TEXT	<p>Indicates device names, environment variables, and logical operators.</p> <p><i>Examples:</i></p> <pre>LPT1 SIGNON OR</pre>
{ }	<p>Indicates a set of choices in a syntax line. The braces themselves should never be typed.</p>
[]	<p>Indicates optional items in a syntax line. The brackets themselves should never be typed.</p> <p><i>Example:</i></p> <pre>buildobjclient [-v] [-o name] [-f <i>file-list</i>]... [-l <i>file-list</i>]...</pre>
	<p>Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.</p>

Convention	Item
-------------------	-------------

...	Indicates one of the following in a command line: <ul style="list-style-type: none">■ 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.
.	

Contents

About This Document

What You Need to Know	iv
e-docs Web Site	iv
How to Print the Document	v
Related Information	v
Contact Us!	v
Documentation Conventions	vi

1. Overview and Key Concepts

W3C XQuery, XML, and Liquid Data	1-2
XQuery Use in Liquid Data and Data View Builder	1-3
The Role of XML in Creating Global Business Solutions	1-3
Supported XML Schema Versions In Liquid Data	1-4
Learning More About the XQuery Language	1-4
Data View Builder Overview	1-4
Advantages of the Data View Builder	1-5
How the Data View Builder Works	1-5
Key Concepts of Query Building	1-6
Query Plans	1-6
Stored Queries	1-6
Ad Hoc Queries	1-7
Different Kinds of Data Sources	1-7
Relational Databases	1-8
XML Files	1-9
Web Services	1-9
Application Views	1-9
Data Views—Using the Result of a Query as a Data Source	1-10

Stored Procedures.....	1-10
Source and Target Schemas.....	1-10
Understanding Source Schemas.....	1-11
Understanding Target Schemas.....	1-11
Anatomy of a Query: Joins, Unions, Aggregates, and Functions	1-14
Joins.....	1-15
Unions	1-16
Aggregates.....	1-16
Functions	1-16
Query Parameters	1-17
Understanding XML Namespaces.....	1-17
XML Namespace Overview.....	1-17
Predefined Namespaces in XQuery.....	1-18
Other XML Namespace References.....	1-18
XML Namespaces in Liquid Data Queries	1-19
Namespace Declarations in XQuery Prolog.....	1-19
Namespaces in Target Schema Definitions.....	1-20
Data Sources that Require Namespace Declarations	1-20
Migrating Liquid Data 1.0 Queries	1-21
Next Steps.....	1-21

2. Starting the Builder and Touring the GUI

Starting the Data View Builder	2-2
Data View Builder GUI Tour	2-3
Design Tab.....	2-4
Overview Picture of Design Tab Components.....	2-4
1. Menu Bar for the Design Tab.....	2-6
2. Toolbar for the Design Tab	2-8
3. Builder Toolbar	2-9
4. Source Schemas.....	2-21
5. Target Schema.....	2-22
6. Conditions Tab	2-24
7. Mappings Tab.....	2-28
8. Sort By Tab	2-29
9. Status Bar	2-30

Optimize Tab.....	2-30
Overview Picture of Optimize Tab Components	2-30
1. Source Order Optimization	2-31
2. Join Pair Hints	2-32
Test Tab.....	2-32
Overview Picture of Test Tab Components	2-33
1. Menu Bar for the Test Tab	2-34
2. Toolbar for the Test Tab	2-34
3. Builder-Generated XQuery	2-35
4. Query Parameters: Submitted at Query Runtime	2-35
5. Query Results - Large Results	2-36
6. Run Query	2-36
7. Result of a Query	2-37
Working With Projects	2-38
To Make a Project Portable, Save Target Schema to Repository	2-38
Saving a Project is Not the Same as Saving a Query	2-38
Using Schemas Saved With Projects	2-39
Special Characters: Occurrence Indicators.....	2-39
Next Steps: Building and Testing Sample Queries	2-40

3. Designing Queries

Designing a Query.....	3-2
Building a Query	3-3
Opening the Source Schemas for the Data Sources You Want to Query ..	3-4
Adding a Target Schema	3-5
Editing a Target Schema	3-7
Mapping Source and Target Schemas	3-8
Mapping Node to Node.....	3-8
Example: Query Customers by State	3-9
Mapping Nodes to Functions	3-10
Supported Mapping Relationships	3-12
Removing Mappings	3-13
Setting Conditions	3-14
What are Functions?.....	3-14
Using Constants and Variables in Functions	3-15

Enabling and Disabling Conditions.....	3-16
Removing Conditions.....	3-16
Adding or Deleting Parameters in a Condition Statement.....	3-17
Showing or Hiding Data Types.....	3-17
Using Automatic Type Casting.....	3-17
Exceptions to Automatic Type Casting.....	3-18
Examples of Simple Queries.....	3-19
Example: Return Customers by Name.....	3-19
Build the Query.....	3-19
View the XQuery and Run the Query to Test it.....	3-22
Example: Query Customers by ID and Sort by State.....	3-25
Open the Data Sources and Add a Target Schema.....	3-26
Map Nodes from Source to Target Schema to Project Output.....	3-26
Join Two Sources.....	3-26
Specify the Order of the Result Using the Sort By Features.....	3-27
View the XQuery and Run the Query to Test it.....	3-28
Understanding Scope in Basic and Advanced Views.....	3-30
Where Does Scope Apply?.....	3-31
Basic View (Automatic Scope Settings).....	3-31
Advanced View (Setting the Scope Manually).....	3-31
When to Use Advanced View to Set Scope Manually.....	3-33
Task Flow Model for Advanced View Manual Scoping.....	3-34
Returning to Basic View.....	3-37
Saving Projects from Basic or Advanced View.....	3-38
Version Control.....	3-38
Scope Recursion Errors.....	3-38
Recommended Action.....	3-39
Understanding Query Design Patterns.....	3-39
Target Schema Design Guidelines and Query Examples.....	3-39
Design Guidelines.....	3-40
Examples of Effective Query Design.....	3-41
Source Replication.....	3-49
Why is source replication necessary?.....	3-50
When is source replication necessary?.....	3-50
When should you manually replicate sources?.....	3-50

Next Steps.....	3-51
-----------------	------

4. Optimizing Queries

Factors in Query Performance.....	4-1
Using the Features on the Optimize Tab.....	4-2
Source Order Optimization.....	4-3
Example: Source Order Optimization.....	4-4
Optimization Hints for Joins.....	4-5
Choosing the Best Hint.....	4-5
Using Parameter Passing Hints (ppleft or ppright).....	4-6
Using a Merge Hint.....	4-8

5. Testing Queries

Switching to the Test View.....	5-1
Using Query Parameters.....	5-2
Specifying Large Results for File Swapping.....	5-3
Running the Query.....	5-4
Viewing the Query Result.....	5-5
Saving a Query.....	5-6
Saving a Query to the Repository as a “Stored Query”.....	5-6
Naming Conventions for Stored Queries.....	5-7

6. Using Data Views

Enterprise and the Data View.....	6-1
Understanding Data Views.....	6-2
A Data View Use Case.....	6-3
Simple and Parameterized Data Views.....	6-4
Using Data Views as Data Sources.....	6-4
Creating a Data View.....	6-4
Creating and Saving the Query to the Liquid Data Repository.....	6-5
Configuring a Data View Data Source Description.....	6-5
Adding a Data View as a Data Source.....	6-6
Creating a Parameterized Data View.....	6-6
Data View Query Samples.....	6-12

7. Using Complex Parameter Types in Queries

Understanding Complex Parameter Types	7-2
A CPT Use Case	7-3
Understanding CPT Schema and Data	7-4
Sample CPT Schema	7-4
Sample XML Data Stream	7-5
Notes on Hand-Crafting CPT XQueries	7-6
Unique Namespace	7-7
XQuery of type element Declaration	7-7
Creating a Complex Parameter Type	7-8
1. Create a CPT Schema	7-8
2. Create Your Runtime Source	7-8
3. Define Your CPT in the Administration Console	7-8
4. Build Your Query	7-9
4. Run your query	7-9
Complex Parameter Type Query Samples	7-13

8. Defining Stored Procedures

Defining Stored Procedures to Liquid Data	8-2
To Define Stored Procedures to Liquid Data	8-2
Stored Procedure Description File Schema	8-4
Basic Structure	8-4
Type Definitions	8-4
Function Definitions	8-4
Schema Definition File for Stored Procedure Description File	8-5
Element and Attribute Reference for Stored Procedure Description File ..	8-6
Supported Datatypes	8-9
Rules for Specifying Stored Procedure Description Files	8-10
Rules for Element and Attribute Names	8-11
Rules for Procedure Names Containing a Semi-Colon	8-12
Rules and Examples of <type> Declarations to Use in the <function> return_type Attribute	8-12
Example 1: Type Definition with No Return Value	8-13
Example 2: Type Definition with Simple Return Value	8-14
Example 3: Type Definition for Complex Row Set Type	8-14

Example 4: Type Definition with Complex Return Value	8-15
Example 5: Type Definition with Simple Return Value and Two Row Sets	8-15
Rules for the mode Attribute output_only <argument> Element.....	8-16
Rules for Transforming the Function Signature When Hand Writing an XQuery.....	8-16
Namespace Declaration.....	8-16
Function Transformation.....	8-17
Sample Stored Procedure Description Files	8-19
DB2 Simple input_only, output_only, and input_output Example.....	8-20
Oracle Cursor Output Parameter Example.....	8-22
DB2 Multiple Result Set Example	8-23
Oracle Cursor as return_value.....	8-25
Stored Procedure Support by Database	8-26
Oracle	8-27
Microsoft SQL Server	8-28
Sybase.....	8-29
IBM DB2.....	8-30
Informix.....	8-31
Using Stored Procedures in Queries.....	8-32
Define Stored Procedures to Liquid Data	8-33
Example: Defining and Using a Customer Orders Stored Procedure	8-33
Business Scenario	8-33
View a Demo	8-33
Step 1: Create the Stored Procedure in the Database.....	8-34
Step 2: Create the Stored Procedure Description File.....	8-34
Step 3: Specify the Stored Procedure Description File in the Liquid Data Console	8-35
Step 4: Open the Data View Builder to See Your Stored Procedures	8-36
Step 5: Use the Stored Procedure in a Query.....	8-36
Step 6: Run the Query.....	8-37

9. Query Cookbook

Example 1: Simple Joins	9-2
The Problem	9-3

The Solution	9-3
View a Demo.....	9-4
Ex 1: Step 1. Verify the Target Schema is Saved in Repository.....	9-4
Ex 1: Step 2. Open Source and Target Schemas	9-5
Ex 1: Step 3. Map Nodes from Source to Target Schema to Project the Output.....	9-6
Ex 1: Step 4. Create a Query Parameter for a Customer ID to be Provided at Query Runtime	9-6
Ex 1: Step 5. Assign the Query Parameter to a Source Node	9-6
Ex 1: Step 6. Join the Wireless and Broadband Customer IDs	9-6
Ex 1: Step 7. Set Optimization Hints	9-7
Ex 1: Step 8. View the XQuery and Run the Query to Test it	9-7
Ex. 1: Step 9. Verify the Result.....	9-8
Example 2: Aggregates.....	9-8
The Problem	9-9
The Solution	9-9
View a Demo.....	9-10
Ex 2: Step 1. Locate and Configure the “AllOrders” Data View.....	9-10
Ex 2: Step 2. Restart the Data View Builder and Find the New Data View 9-13	9-13
Ex 2: Step 3. Verify the Target Schema is Saved in the Repository.	9-13
Ex 2: Step 4. Open the Data Sources and Target Schema.....	9-14
Ex 2: Step 5. Map Source Nodes to Target to Project the Output.....	9-14
Ex 2: Step 6. Create Two Query Parameters to be Provided at Query Runtime	9-15
Ex 2: Step 7. Assign the Query Parameters to Source Nodes	9-15
Ex 2: Step 8. Add the “count” Function.....	9-15
Ex 2: Step 9. Verify Mappings and Conditions	9-16
Ex 2: Step 10. View the XQuery and Run the Query to Test it	9-17
Ex 2: Step 11. Verify the Result.....	9-18
Example 3: Date and Time Duration	9-18
The Problem	9-18
The Solution	9-18
View a Demo.....	9-19
Ex 3: Step 1. Verify the Target Schema is Saved in Repository.....	9-19

Ex 3: Step 2. Open Source and Target Schemas	9-21
Ex 3: Step 3. Map Source to Target Nodes to Project the Output	9-21
Ex 3: Step 4. Create Joins	9-23
Ex 3: Step 5. Create Two Query Parameters for Customer ID and Date to be Provided at Query Runtime.....	9-23
Ex 3: Step 6. Set a Condition Using the Customer ID.....	9-24
Ex 3: Step 7. Set a Condition to <i>Determine if Order Ship Date is Earlier or Equal to a Date Submitted at Query Runtime</i>	9-24
Ex 3: Step 8. Set a Condition to Include Only “Open” Orders in the Result	9-25
Ex 3: Step 9. View the XQuery and Run the Query to Test it	9-25
Ex 3: Step 9. Verify the Result	9-27
Example 4: Union.....	9-28
The Problem	9-28
The Solution	9-28
View a Demo	9-29
Ex 4: Step 1. Verify the Target Schema is Saved in Repository	9-29
Ex 4: Step 2. Open Source and Target Schemas	9-30
Ex 4: Step 3. Clone the Orders Element of the Target Schema	9-31
Ex 4: Step 4. Create a Query Parameter for a Customer ID	9-31
Ex 4: Step 5. Assign a Query Parameters	9-31
Ex 4: Step 6. Define Source Relationships	9-31
Ex 4: Step 7. Project the Output to the Target Schema.....	9-32
Ex 4: Step 8. Add Optimization Hints	9-32
Ex 4: Step 9. View the XQuery and Run the Query to Test it	9-33
Ex 4: Step 10. Verify the Result	9-34
Example 5: Minus.....	9-35
The Problem	9-36
The Solution	9-36
View a Demo	9-37
Ex 5: Step 1. Verify the Target Schema is Saved in Repository	9-37
Ex 5: Step 2. Open Source and Target Schemas	9-38
Ex 5: Step 3. Find Broadband and Wireless Customers with the Same Customer ID.....	9-38
Ex 5: Step 4. Find the Count of the Wireless Customers.....	9-39

Ex 5: Step 5. Set a Condition that Specifies the Output of “count” is Zero 9-39	
Ex 5: Step 6. View the XQuery and Run the Query to Test it	9-40
Ex 5: Step 7. Verify the Result.....	9-40
Example 6: Complex Parameter Type (CPT).....	9-41
The Problem	9-41
The Solution	9-41
View a Demo.....	9-42
Ex 6: Step 1. Verify the Availability of Schemas and Sample Data Stream	9-42
Ex 6: Step 2. Open the Target Schema and CO-CPTSAMPLE CPT	9-45
Ex: 6: Step 3. Create an orderLimit Query Parameter	9-45
Ex 6: Step 4. Save the Project	9-46
Ex 6: Step 5. Test Access to the Complex Parameter Source	9-46
Ex 6: Step 6: Determine the Total Amount of New Orders	9-47
Ex 6: Step 7. Create the Necessary Joins and Mappings to the Target Schema.....	9-49
Ex 6: Step 8. Determine the Amount of Currently Open Orders	9-51
Ex 6: Step 9: Determine the Total Amount of All Open and New Orders 9-52	
Ex 6: Step 10: Test If Open Orders + New Orders Exceeds the Order Limit.....	9-52
Ex 6: Step 11: Determine If the Order is Accepted or Rejected	9-52
Ex 6: Step 12: View the XQuery.....	9-53
Ex 6: Step 13. Run the XQuery to Verify the Result	9-54

A. Functions Reference

About in Liquid Data XQuery Functions	A-3
Naming Conventions	A-3
Occurrence Indicators.....	A-3
Data Types.....	A-4
Date and Time Patterns	A-7
Accessor and Node Functions	A-9
xf:data	A-9
xf:local-name.....	A-10
Aggregate Functions.....	A-11

xf:avg.....	A-12
xf:count.....	A-13
xf:max.....	A-13
xf:min	A-15
xf:sum	A-16
Boolean Functions	A-17
xf:false	A-18
xf:not	A-18
xf:true	A-19
Cast Functions	A-20
cast as xs:boolean	A-21
cast as xs:byte.....	A-22
cast as xs:date	A-22
cast as xs:dateTime.....	A-23
cast as xs:decimal	A-24
cast as xs:double.....	A-25
cast as xs:float	A-26
cast as xs:int	A-27
cast as xs:integer.....	A-28
cast as xs:long.....	A-28
cast as xs:short.....	A-29
cast as xs:string.....	A-29
cast as xs:time.....	A-30
Comparison Operators.....	A-32
eq	A-32
ge	A-33
gt.....	A-34
le	A-35
lt.....	A-36
ne	A-37
Constructor Functions	A-38
xf:boolean-from-string	A-39
xf:byte.....	A-40
xf:decimal.....	A-41
xf:double.....	A-42

xf.float	A-43
xf.int	A-44
xf.integer.....	A-45
xf.long.....	A-46
xf.short.....	A-47
xf.string.....	A-48
Date and Time Functions.....	A-50
xf.add-days	A-51
xf.current-dateTime	A-52
xf.date	A-53
xfext:date-from-dateTime.....	A-54
xfext:date-from-string-with-format	A-55
xf.dateTime.....	A-56
xfext:dateTime-from-string-with-format.....	A-58
xf.get-hours-from-dateTime	A-59
xf.get-hours-from-time	A-60
xf.get-minutes-from-dateTime	A-61
xf.get-minutes-from-time	A-62
xf.get-seconds-from-dateTime	A-62
xf.get-seconds-from-time	A-63
xf.time.....	A-64
xfext:time-from-dateTime	A-66
xfext:time-from-string-with-format.....	A-67
Logical Operators	A-68
and	A-68
or.....	A-70
Numeric Operators	A-71
* (multiply).....	A-71
+ (add)	A-73
- (subtract)	A-75
div	A-76
mod.....	A-77
Numeric Functions	A-79
xf.ceiling.....	A-79
xf.floor.....	A-80

xf:round	A-81
xfext:decimal-round	A-82
xfext:decimal-truncate	A-83
Other Functions	A-84
xfext:if-then-else	A-84
Sequence Functions	A-85
xf:distinct-values	A-85
xf:empty	A-86
xf:subsequence (format 1)	A-87
xf:subsequence (format 2)	A-88
String Functions	A-90
xf:compare	A-90
xf:concat	A-92
xf:contains	A-93
xf:ends-with	A-94
xf:lower-case	A-96
xf:starts-with	A-97
xf:string-length	A-98
xf:substring (format 1)	A-99
xf:substring (format 2)	A-100
xf:substring-after	A-102
xf:substring-before	A-103
xf:upper-case	A-104
xfext:match	A-105
xfext:trim	A-108
Treat Functions	A-109
treat as xs:boolean	A-111
treat as xs:byte	A-111
treat as xs:date	A-112
treat as xs:dateTime	A-112
treat as xs:decimal	A-113
treat as xs:double	A-114
treat as xs:float	A-114
treat as xs:int	A-115
treat as xs:integer	A-116

treat as xs:long	A-116
treat as xs:short	A-117
treat as xs:string	A-118
treat as xs:time	A-118

B. Supported Data Types

Overview	B-1
JDBC Types	B-2
JDBC Names	B-3
Database-Specific Names	B-5
Oracle Names	B-5
Microsoft SQL Server Names	B-6
DB2 Names	B-7
Sybase Names	B-7
Informix Names	B-8

C. Type Casting Reference

Type Casting to a Numeric Target	C-2
Type Casting to a Non-Numeric Target	C-3
Type Casting Function Parameters	C-4

Index

1 Overview and Key Concepts

This section introduces key concepts you need to understand to plan, design, build and test queries using BEA Liquid Data for WebLogic™. The notion of a *stored query* versus an *ad hoc query* is introduced. Also covered is using a *hand-coded query* versus a *Builder-generated query*. Since we want to encourage most users to leverage the Data View Builder to generate queries for Liquid Data, many of the considerations and concepts introduced here assume use of the Builder, including a GUI overview for the Builder. However, key concepts that are relevant to all types of query-smiths are introduced here as well such as data sources, stored queries, data views, XQuery, the anatomy of a query (joins, unions, aggregates and so on), and the process of building and testing a query.

The following topics are covered.

- [W3C XQuery, XML, and Liquid Data](#)
 - [XQuery Use in Liquid Data and Data View Builder](#)
 - [The Role of XML in Creating Global Business Solutions](#)
 - [Supported XML Schema Versions In Liquid Data](#)
 - [Learning More About the XQuery Language](#)
- [Data View Builder Overview](#)
 - [Advantages of the Data View Builder](#)
 - [How the Data View Builder Works](#)
- [Key Concepts of Query Building](#)
 - [Query Plans](#)

- [Stored Queries](#)
- [Ad Hoc Queries](#)
- [Different Kinds of Data Sources](#)
- [Source and Target Schemas](#)
- [Anatomy of a Query: Joins, Unions, Aggregates, and Functions](#)
- [Understanding XML Namespaces](#)
- [Next Steps](#)

W3C XQuery, XML, and Liquid Data

XQuery is a World Wide Web consortium (W3C) standard XML-based Query language. 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. Liquid Data uses XQuery to query multiple types of data sources—the structure of which are represented as XML by the query engine.

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, a more approachable XML language for database developers, including the published specifications for schemas and a query language. The evolving language is XQuery, which gives XML developers a structured solution for accessing XML data. The W3C Query Working Group used a formal approach by defining a data model and formal query algebra 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. It includes familiar database operations such as projection, iteration, selection, and join.

XQuery Use in Liquid Data and Data View Builder

BEA Liquid Data uses a stable components of the W3C XQuery specification to take advantage of XML query power as the standards continue to evolve. By using XQuery, Liquid Data can model XML schemas for various types of data sources. These schemas are surfaced as design tools in the Data View Builder, which generates queries in XQuery in the background. The Liquid Data server can process ad hoc or Builder-generated queries in XQuery syntax and use them to query all different kinds of data sources (relational databases, Web services, application views, data views, and so on) and return results in XML.

Once you have configured Liquid Data access to the data sources you want to use (see the Liquid Data *Administration Guide*), you can query the data by sending queries written in XQuery to the data sources via Liquid Data.

The Role of XML in Creating Global Business Solutions

By supporting XML technology, creating specifications, fostering software development, the W3C hopes to use XML as a forum for information exchange, business development, and global communication.

XML is being used on the Internet is to create a simple way to exchange data among diverse clients. Proprietary data definitions and access methods inhibit data exchange. They lock you into using only those products and programs that can send, receive, and process your data.

You could compare the universality of XML to a global monetary exchange standard, or to an international spoken language that removes barriers to global commerce and communication. Data View Builder and the Liquid Data query generation engine adhere to these standards to facilitate cross-platform and cross-repository access to critical business information.

You can learn more about XML on the W3C Web site at <http://www.w3.org/XML/>.

Supported XML Schema Versions In Liquid Data

XML schemas are used in Liquid Data to describe the hierarchical structure of the various data sets with which you are working. Liquid Data recognizes XML Schema versions 2001, 2000/08, and 2000/10.

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>.

For an introduction on working with schemas in the Data View Builder see “[Source and Target Schemas](#)” on page 1-10.

Learning More About the XQuery Language

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 the topic [XQuery Links](#) in “Liquid Data Concepts” in the Liquid Data *Product Overview*.

Data View Builder Overview

The Data View Builder is a GUI-based tool for designing and generating XML-based queries (in W3C XQuery syntax). You can then run the queries against heterogeneous data sources to retrieve information. The Builder provides a pictorial, drag-and-drop *mapping* approach to query design and construction. Using the Data View Builder frees you from having to focus on the intricacies of query languages so that you can give full attention to information design, the conceptual synthesis of information coming from multiple sources, and the content and shape of the information you want in the query result or *target*. In this way, you can to directly access distributed, heterogeneous data sources as “integrated logical views.”

Advantages of the Data View Builder

The Data View Builder lets you create queries using an intuitive, drag-and-drop mapping strategy that frees you from having to grapple with the details of query languages. The XML schema representations and mappings of source and target data are packaged and saved as a project. Users can retrieve the full picture of the query complete with source schemas and target mappings thereby getting access to the query in the context of a design model. Queries can also be stored as *data views* that can be configured as data sources themselves in Liquid Data and re-used to create nested subqueries; “views on views” of information.

How the Data View Builder Works

In the Data View Builder, you drag and drop elements and attributes among XML schema representations of data sources to create source conditions (joins, unions, and so on). The default source condition is a join (that uses an equality function); you can also use the more complex functions provided in the Builder toolbar. You also map source to target schema elements and attributes to shape the structure of the query result. As you build up the query with drag-and-drop modeling, the Builder is constructing the query in the background in valid, well-formed XQuery syntax. An “Optimize” view is also available for adding optimization hints to a query to improve performance. When you are ready to run a query, you can switch to the Builder “Test” view, see the generated XQuery for the current query, run it and see the query result in XML.

The Data View Builder provides hierarchical tree *XML schema* representations of all data sources configured in Liquid Data, regardless of the data source type. Once a data source has been configured in Liquid Data using the WebLogic Server Administration Console (see the Liquid Data [Administration Guide](#)), it shows up in the Builder toolbar where you can access its XML schema representation. The structure of the data stored in relational databases, Web services, application views, data views, and XML files themselves are all represented as XML schemas in the Data View Builder. By creating this coherent picture of heterogeneous data sources as XML schemas, Data View Builder makes it easy for you to browse and map data elements and attributes among different types of data sources.

Key Concepts of Query Building

The following terms and concepts introduced here:

- [Query Plans](#)
- [Stored Queries](#)
- [Ad Hoc Queries](#)
- [Different Kinds of Data Sources](#)
- [Source and Target Schemas](#)
- [Anatomy of a Query: Joins, Unions, Aggregates, and Functions](#)

Query Plans

A *query plan* is a compiled query. Before a query is run, Liquid Data compiles the XQuery into an optimized query plan. At runtime, Liquid Data executes the query plan against physical data sources and returns the query results.

Stored Queries

A *stored query* is a query that has been saved to the Liquid Data repository in the `stored_queries` folder. Queries must be saved with a `.xq` extension to be recognized as stored queries in Liquid Data. There is a performance benefit to using a stored query because caching is available as follows:

- The query plan for a stored query is always cached in memory. (For an ad-hoc query, the query plan is not cached.)
- The *query result* for a stored query can be cached.

Caching of query results for stored queries is configurable through the Administration Console (see [Configuring the Query Results Cache](#) in the Liquid

Data Administration Guide). Using this feature, you can specify whether or not to cache query results for stored queries.

Note: Queries can be stored in subdirectories of the `stored_queries` folder and accessed similarly to a path expression. For example, if a query is saved in a repository directory under:

```
stored_queries/uCustomer/custQuery.xq
```

it could be executed from a jsp with:

```
<lds:query name="uCustomer.custQuery" >  
</lds:query>
```

Ad Hoc Queries

An *ad hoc query* is a query that has not been stored in the Liquid Data repository as a stored query but rather is passed to the Liquid Data server on the fly. Liquid Data does not cache the query plan or the result for an ad hoc query the way it can for a stored query, so an ad hoc query cannot leverage the performance benefit of caching.

Different Kinds of Data Sources

Information can reside in various kinds of *data sources* in an enterprise or across business entities. The most obvious of these is the relational database, which we typically think of as a data storage and retrieval resource. The reality is that the development of global business and distributed systems has generated information in many other types of data sources as well. Information resides not only in various kinds of databases, but also in packaged enterprise information system (EIS) applications such as PeopleSoft or Siebel, and in emerging net-based technologies like Web services and XML documents. Liquid Data and Data View Builder give you the ability to query and get views into data that resides in all these kinds of information sources.

Relational Databases

All types of businesses and other organizations use an RDBMS (relational database management system) to store information. *Relational* refers to the way the database organizes information. All information in a relational database appears in logical tables with rows and columns. Instead of a series of static records with one or more data fields that can be redundant from one file to another, information is directly accessible using *queries*. You can create logical table records that contain just the data you need by constructing a query.

Some databases track information, such as reservations or overnight package delivery information. Other databases store information for perpetual access, such as the IRS or the Library of Congress. Others change dynamically, depending on frequent updates, additions, and deletions, such as a newspaper subscriber database. Databases can reside on large mainframes, web servers, or powerful desktop systems.

Imagine how many times your employee number can appear in static records that describe your company 401K investment, employment, and health insurance records. In an isolated case, this represents three separate files with much of the same information repeated in each instance where there is a record of information. A comprehensive RDBMS would store your employee number, name, address, and other information once with pointers to other related pieces of information about you. Well designed queries could extract only information related to a specific task.

Note: When Data View Builder inspects the metadata for a relational database, if the schema contains any columns that start with numeric values, the Data View Builder adds an underscore character (`_`) to the beginning of the element name that represents the column. For example, if you have a column in the database named `123_COLUMN`, the element corresponding to that column in the Data View Builder is labeled `_123_COLUMN`.

Also, the following characters from any catalog, schema, table, or column names are replaced with an underscore character:

`: < > \ / $, tab, newline, and spaces`

For example, a table named `customer><$table` can be referenced as `customer___table` (three underscore characters replace the three special characters).

Additionally, if you are hand editing queries, the element or attribute names that refer to column names that begin with a numeric value must begin with an underscore character (`_`) when used in XPath expressions.

Tuples

Tuples are another way to refer to data in a database. In a relational database, a tuple is a complete set of information, or a logical record. For example, a personnel schema might contain records that has four fields: an employee number, a name field, an address field, and a phone number field. This tuple, or record, might occur many thousands of times in a very large company with many employees.

XML Files

Extensible Markup Language (XML) files are proving to be a convenient and portable format for storing many different kinds of information for document processing and information exchange. Liquid Data and Data View Builder supports use of XML files as data sources.

Web Services

A web service is a self-contained, platform-independent morsel of business logic, located somewhere on the Internet, that is accessible through standards-based Internet protocols like HTTP or SMTP. Web services facilitate application-to-application communication over the Internet or within and across enterprises. A familiar example of an externalized Web service is a weather portlet or stock quotes that you can integrate into your Web browser. You can use Web services to encapsulate information and operations. With the standards and wide-spread use of Web services for enterprise information exchange evolving, Web services are becoming important resources of global business information. Liquid Data and Data View Builder support the use of Web services as data sources.

Application Views

Enterprise Information Systems (EIS) and custom applications store information that you might need to aggregate for a complete view of data. You can query and retrieve subsets of relevant information from applications such as SAP, Siebel, PeopleSoft, Oracle Financial and so on and treat these views as *application view* data sources in your data integration solution.

Data Views—Using the Result of a Query as a Data Source

A data view is a special type of data source in which the result of a query is used as a data source. The query result will change as your data changes. In this way, you can build on the queries you design to create "views on data views" for an up-to-date picture of continually changing information.

Stored Procedures

For relational databases that support stored procedures, you can create stored procedures in the RDBMS and expose them to Liquid Data. Liquid Data treats a stored procedure as a function which requires one or more inputs to produce zero or more outputs. A stored procedure allows database programmers to combine business logic with database queries, and they provide a powerful way to produce information from relational databases. Also, stored procedures allow the database administrators to tune the queries run by the stored procedures, thus ensuring good performance and minimizing impact on database performance.

Source and Target Schemas

XML *schemas* are used in Liquid Data to describe the hierarchical structure of the various data sets with which you are working. The Data View Builder uses XML schema representations as follows:

- Source Schemas—XML schemas that describe the structure of the source data.
- Target Schema—An XML schema that describes the structure of the target data; that is, the structure of the *query result*.

To define schema to Liquid Data, you configure Liquid Data data sources. These data sources must have an associated schema. For relational databases, you can specify a schema or use the default schema determined automatically based on the metadata available through the database JDBC driver. For XML files, Views, Complex Parameter Types, Stored Procedures, or Web Services, you specify the schema when you define the data source to Liquid Data. Once the data sources are defined, you can use them in queries.

Note: For information on which versions of XML schema are supported, see [“Supported XML Schema Versions In Liquid Data” on page 1-4](#).

This section includes the following:

- [Understanding Source Schemas](#)
- [Understanding Target Schemas](#)

Understanding Source Schemas

A *source schema* is the XML schema representation of the structure of the data in a data source.

The Data View Builder provides graphical representation of source schemas in a tree structure format. Nodes contain elements, attributes, and sub-elements and attributes that you can expand or collapse. If you are building a query that queries more than one data source, you will use multiple source schemas (one for each data source).

Understanding Target Schemas

A *target schema* describes the structure of a query result that will be produced when the query runs. As with source schemas, the Data View Builder provides a graphical representation of target schemas in a tree structure format. Nodes represent elements, attributes, and sub-elements that you can expand or collapse. Only one target schema per query is allowed.

Target schemas have two main purposes:

- Target schemas provide a template for mapping data from source schemas in order to generate a query.
- Target schemas provide the schema definition when creating Liquid Data Web Service definitions and Data View definitions.

You can specify a target schema in the Data View Builder in the following ways:

- You can set the target schema to simply use an existing schema for the target schema. For example, you might have a target schema that a data architect built to use in a given class of queries.
- You can open an existing target schema and then use the Data View Builder to modify and save it.

1 Overview and Key Concepts

- You can use the Data View Builder to build a new target schema from scratch, using the tools available on the right-click menu in the target schema Data View Builder pane.

The way you use target schemas with the Data View Builder varies depending upon the situation. Sometimes you have a schema-driven scenario where you have an existing target schema and want to generate a query that maps data to it appropriately. Other times you have a query-driven scenario where you are trying to generate a particular query result set, and you can use the target schema as a means of shaping your query results.

A target schema can either describe just the portion of the hierarchy that you want to appear in the result set or it can be a superset of the data that actually is mapped in the query. For example, you could choose the same schema for both the source and target data structure, but then map only some of the source elements to the target schema. The query result will show only those data elements that are actually mapped to nodes in the target schema. When a target schema contains unmapped elements, as in this example, the unmapped elements must be specified as Optional in the target schema definition.

You can specify Repeatable and Optional attributes of each node for a target schema in the properties dialog of each node. The Data View Builder uses these target schema attributes to infer your intent during query construction, and these attributes therefore affect the queries that the Data View Builder generates. If a node is not specified as optional, then it must be mapped in the query in order for the query to conform to that target schema. If a node is repeatable, then it can be repeated in the result set of the generated query. For example, consider the following target schema:

Figure 1-1 Target schema with non-repeatable node



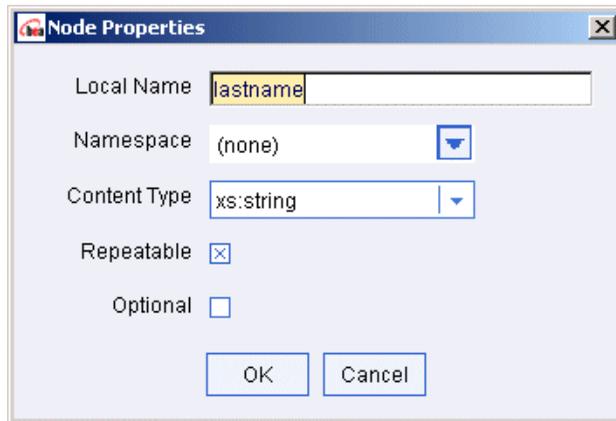
In this target schema, the `firstname` and `lastname` elements are defined as non-repeatable, and the `custrecord` node is defined as repeatable and required (the + next to the `custrecord` name indicates that the node is repeatable and that it must appear at least once—if it had an asterisk [*] instead of the +, that would indicate the

node is repeatable and optional, and could therefore occur zero or more times). If you map data to the `firstname` and `lastname` nodes, this target schema will generate results similar to the following sample result set:

```
<customers>
  <custrecord>
    <firstname>John</firstname>
    <lastname>Parker</lastname>
  </custrecord>
  <custrecord>
    <firstname>John</firstname>
    <lastname>Warfin</lastname>
  </custrecord>
  .....
  .....
</customers>
```

If you modify the target schema so the `firstname` and `lastname` elements are also repeatable, the result set schema for the generated query will be different. For example, in the schema shown in [Figure 1-1](#), you could open the properties dialog on the `firstname` and `lastname` elements and modify it to look as follows:

Figure 1-2 Properties dialog from target schema with repeatable elements



1 Overview and Key Concepts

With the changed target schema, the Data View Builder will now generate a query with results similar to the following sample result set:

```
<customers>
  <custrecord>
    <firstname>John</firstname>
    <firstname>John</firstname>
    .....
    .....
    <lastname>Parker</lastname>
    <lastname>Warfin</lastname>
    .....
    .....
  </custrecord>
</customers>
```

In this case, it is likely that the query designer would want the result set to display the first and last names together for the same customer, and would therefore desire the non-repeatable nodes for the `firstname` and `lastname` elements. To understand how these attributes affect the query results, experiment with different Node Property settings, run the queries, and compare the results.

When you use the Liquid Data Console to create a Data View definition or a Web Service definition, you must specify both a target schema and a stored query. The queries in these definitions must conform to the specified target schema; that is, if the target schema contains elements that are required (Optional box not checked), those elements must be mapped in the query. Make sure your Data View and Web Service queries conform to the specified target schema, as Data Views and Web Services whose queries and target schemas do not conform might produce unexpected behavior.

Anatomy of a Query: Joins, Unions, Aggregates, and Functions

A query can be thought of as a way of filtering through large amounts of data or information to extract only the specifics relevant to a particular instance. A query is made up of one or more types of conditions that accomplish the filtering task or “ask the question.” The most commonly used techniques for establishing the source conditions are:

- [Joins](#)
- [Unions](#)

- [Aggregates](#)
- [Functions](#)
- [Query Parameters](#)

Joins

A join operation merges data from two sources based on a common field. A query with a *join* operation combines information in two data source schemas when there is a match on a common field. The common field is a link between the two schemas.

Using this common field, you can gather other information that is unique to each source into a single target schema or *view* of the data.

For example, you could specify first and last names of all customers in two data sources Broadband and Wireless, but limit the output (query result described by target schema) to the subset of those customers with matching customer IDs in both source schemas.

There are two types of join operations based on equality of matching fields (or columns):

- **Inner Joins**—An inner join is the default join type. It combines data from two data sources only if values in the joined fields match. The matching fields must have compatible data types or contain similar data.
- **Outer Joins**—An outer join behaves like an inner join, but it includes data that does not match the join condition. You can create two different types of outer joins by specifying which unmatched data elements to include in the query results
 - **Left Outer Join**—In a left outer join, all selected nodes from the leftmost schema in the join clause appear. Unmatched data nodes in the rightmost schema in the join clause do not appear in the result.
 - **Right Outer Join**—In a right outer join, all rows in the rightmost schema in the join clause appear. Unmatched data nodes in the leftmost schema in the join clause do not appear in the result.

Unions

Union operations enable you to combine data from multiple sources into a single set of results described by the structure of the target schema. Even though the content of the source schemas can be the same, or different, you can use a union query to combine selected data nodes in the source schemas into a complete view of the data. For example, we could construct a query that reports all customer orders from multiple sources into a single result.

Aggregates

You can create queries in Liquid Data that aggregate query results, providing summary information on a set of data. The typical aggregate functions are average (`avg`), count, maximum (`max`), minimum (`min`), and `sum`. You can use aggregate operations to perform various business calculations such as counting the number of customers, calculating the total purchases of a single employee, calculating the average salary of workers, and so on.

Functions

Liquid Data provides a functions library with built-in functions can be used by any Liquid Data client, and also supports configuration and use of user-defined *custom functions* related to specific business needs.

All source conditions are established using some type of function. The default function for a simple join is the standard “equals” function (abbreviated `eq`). If you drag and drop one data source element onto another of the same name you have created a simple join using the equals function with two parameters (the two data source elements) which gets expressed as `value1 eq value2` in the Builder-generated XQuery.

You can also choose from the functions library to explicitly specify a function to use.

The W3C specification for XQuery supports a discrete list of functions—Liquid Data supports a subset of those functions. For more information on W3C standard functions, see the [XQuery 1.0 and XPath 2.0 Functions and Operators](#) specification.

Query Parameters

The parameters to a function can be elements in a source schema, or they can be *query parameters* that you define as generic placeholders for a variable value. You can specify the variable value at query run time. For example, a query parameter could be defined as *lastname*, which is a placeholder for a real last name, such as Smith, that you identify when the query runs. Each time you run the query you can change the value of *lastname*, which gives you a lot of flexibility without creating a separate query for every unique last name you might be interested in. By supplying a new value each time, you could run queries on customers named Smith, or Jones, or any other last name of interest without redesigning the query.

Note: Query parameters are case sensitive.

Understanding XML Namespaces

XML namespaces are a mechanism to ensure there are no name conflicts when combining XML documents. Liquid Data supports XML namespaces and includes namespaces in the XQuery queries generated in Data View Builder. This section includes the following topics:

- [XML Namespace Overview](#)
- [XML Namespaces in Liquid Data Queries](#)
- [Migrating Liquid Data 1.0 Queries](#)

XML Namespace Overview

XML Namespaces appear in queries as a string followed by a colon. For example, the `xs:integer` data type uses the XML namespace `xs`.

XML namespaces ensure that names do not collide when combining data from heterogeneous XML documents. For example, if there is an element named `<tires>` from an automobile tire manufacturer in one document and an element named `<tires>` from a bicycle tire manufacturer, it might not be appropriate to combine

1 Overview and Key Concepts

those elements, as they are very different things. XML namespaces can prevent such name collisions by referring to the elements as `<automobile:tires>` and `<bicycle:tires>`.

Predefined Namespaces in XQuery

The following table shows predefined namespaces used in XQuery:

Table 1-1 Predefined Namespaces in XQuery

Namespace Prefix	Description	Examples
<code>xf</code>	The prefix for XQuery functions.	<code>xf:data</code> <code>xf:sum</code> <code>xf:substring</code>
<code>xfext</code>	The prefix for Liquid Data-specific extensions to the standard set of XQuery functions.	<code>xfext:match</code> <code>xfext:trim</code>
<code>xs</code>	The prefix for XML schema types.	<code>xs:element</code> <code>xs:string</code>
<code>xsect</code>	The prefix for Liquid Data-specific extensions to the standard set of XML schema types.	<code>xsect:myownstringtype</code>

Other XML Namespace References

The following are some internet links to more information on XML namespaces:

- <http://www.w3.org/TR/REC-xml-names/>
- <http://www.rpbouret.com/xml/NamespacesFAQ.htm/>

XML Namespaces in Liquid Data Queries

If you are using the Data View Builder to generate queries, it automatically generates the correct namespace declarations when it generates the query. If you are hand-coding your queries, you must include the necessary namespace declaration(s). For a list of data sources that require namespace declarations, see [“Data Sources that Require Namespace Declarations,”](#) on page 1-20.

Namespace Declarations in XQuery Prolog

The beginning portion of an XQuery is known as the *prolog*. For Liquid Data queries, the namespace declarations appear in the XQuery prolog. There can be zero or more namespace declarations in a query prolog. Each namespace has the following form:

```
namespace <logical_name> = "<URI>"
```

where *<logical_name>* is a string used as a prefix in the query and *<URI>* is a uniform resource indicator.

Consider the following simple query:

```
namespace view = "urn:views"

<CustomerOrderID>
{
  for $view:MY_VIEW.order_2 in
    view:MY_VIEW()/results/result/broadband/order
  return
    <ORDER_ID>{ xf:data($view:MY_VIEW.order_2/ORDER_ID) }
    </ORDER_ID>
}
</CustomerOrderID>
```

The line in the prolog:

```
namespace view = "urn:views"
```

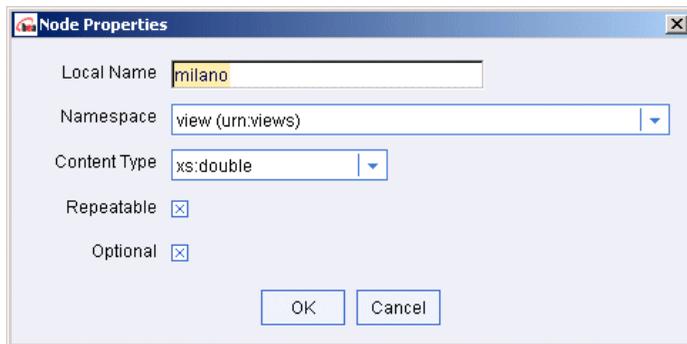
is the namespace declaration in this query. Each time the object (in this case, a view) is referenced in the query, the object name is prefixed with the logical name `view`.

1 Overview and Key Concepts

You must define all namespaces in the XQuery prolog to use them in a query (except for the predefined namespaces described in [“Predefined Namespaces in XQuery,”](#) on page 1-18). If you do not define namespaces in the XQuery prolog, the query will fail with a compilation error.

Namespaces in Target Schema Definitions

When you use the Data View Builder to create or modify target schemas, you can specify a namespace for an element or an attribute. If you specify a namespace, it is added to the XML markup in the query (and therefore to the query results). For example, if you add the `view` namespace to an element of the target schema named `milano` as follows:



The query results for this target schema definition are of a form similar to the following:

```
<view:milano xmlns:view="urn:views">100.0</view:milano>
```

Data Sources that Require Namespace Declarations

All data sources except relational databases and XML files require the namespace declaration in the XQuery prolog. Queries involving the following data sources therefore require namespace declarations in the XQuery prolog:

- Data Views
- Web Services
- Application Views
- Stored Procedures

- Complex Parameter Types

Migrating Liquid Data 1.0 Queries

Liquid Data 1.0 did not support XML namespaces, and any queries used in Liquid Data 1.0 must be migrated to work in Liquid Data 1.1. If you have queries that are generated in a Data View Builder project file, you can open the project file in Data View Builder 1.1. When you click the test tab, the Data View Builder automatically generates the new query with the proper namespace declarations in the query prolog.

If you have stored queries and data views, you must use the `queryMigrate` tool to migrate the queries so they work properly in Liquid Data 1.1. For information on the `queryMigrate` tool, see [Migrating Queries and Data Views](#) in the *Installation and Migration Guide*.

Next Steps

- If you have not already done so, we suggest working through the steps in [Getting Started](#), which takes you through the basic tasks of configuring some data sources and using the Data View Builder to design a query using the Order Query example from our Avitek Sample.
- To learn how to start the Data View Builder and understand the GUI tools and views, see [Chapter 2, “Starting the Builder and Touring the GUI.”](#)
- To learn more about planning and designing queries and using the Data View Builder to build them, see [Chapter 3, “Designing Queries.”](#)
- For information on query optimization and performance, see [Chapter 4, “Optimizing Queries.”](#)
- For information on defining stored procedures to Liquid Data, see [Chapter 8, “Defining Stored Procedures.”](#)
- For examples of building different types of queries using advanced functions and tools, see [Chapter 9, “Query Cookbook.”](#)

1 *Overview and Key Concepts*

- For details on creating queries by using custom functions, see [“Using Custom Functions”](#) in *Invoking Queries Programmatically*.

2 Starting the Builder and Touring the GUI

This section describes how to start the Data View Builder tool in BEA Liquid Data for WebLogic™. It provides a complete GUI reference and explanation of the tools, data sources, schemas, and task flow for designing, optimizing, and testing a Builder-generated query. The following topics are covered:

- [Starting the Data View Builder](#)
- [Data View Builder GUI Tour](#)
 - [Design Tab](#)
 - [Optimize Tab](#)
 - [Test Tab](#)
- [Working With Projects](#)
- [Special Characters: Occurrence Indicators](#)
- [Next Steps: Building and Testing Sample Queries](#)

Starting the Data View Builder

To start the Data View Builder, follow these basic steps.

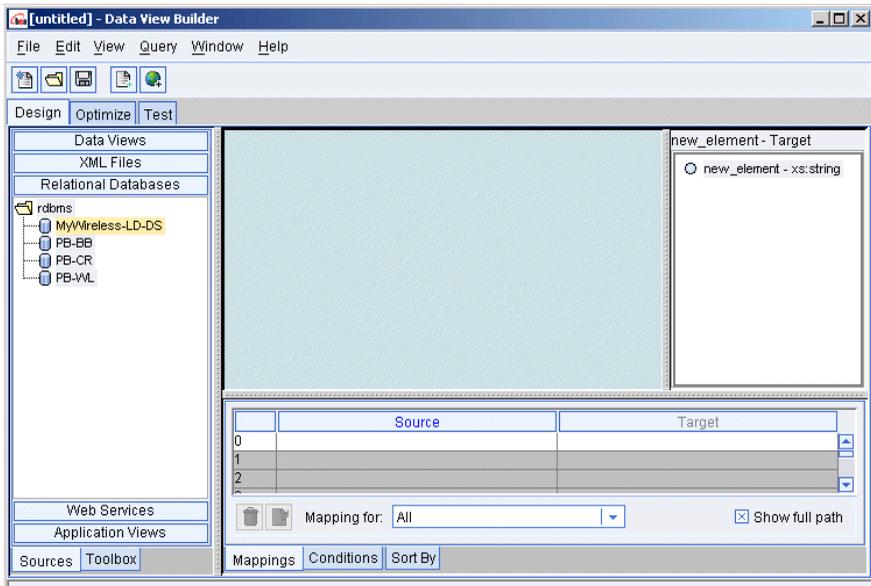
1. Start the Data View Builder.
 - On a Windows platform, choose the menu item:
Start—>Programs—>BEA WebLogic Platform 7.0—>BEA Liquid Data for WebLogic 1.0—>Launch Data View Builder

You can also start the Data View Builder by double-clicking on the file:
`BEA_Home\WL_HOME\liquiddata\DataViewBuilder\bin\DVBuilder.cmd`

A login window is displayed. This is for logging in to a Liquid Data server.
2. Connect to the Liquid Data server on which the data sources you want to use are located.
 - a. The username and password for the Data View Builder is specified in the WebLogic Server (WLS) Compatibility Security via the WLS Administration Console for the Liquid Data server to which you want to connect. For more information, see [Implementing Security](#) in the *Liquid Data Administration Guide*. If the server allows guest users, you do not need to enter a username and password—you can leave these fields blank.
 - b. Enter the URL for the Liquid Data server. For example, to connect to a server running on your own machine as a local host you would enter the following:
`t3://localhost:7001`
 - c. Click the Login button.

The Data View Builder work area and tools appear, as shown in [Figure 2-1](#).

Figure 2-1 Starting Data View Builder



Data View Builder GUI Tour

The Data View Builder consists of three main views or modes that you can get to by clicking on the associated tabs. Each tab represents a phase in the process of designing and testing a query. Generally, you will use the Design and Test tabs to design and run (test) the query, respectively. Some, but not all, queries will require the use of optimization hints and techniques on the Optimize tab.

- [Design Tab](#)
- [Optimize Tab](#)
- [Test Tab](#)

Design Tab

The Design tab is where you construct the query by working with source and target schemas to specify source conditions and source-to-target mappings.

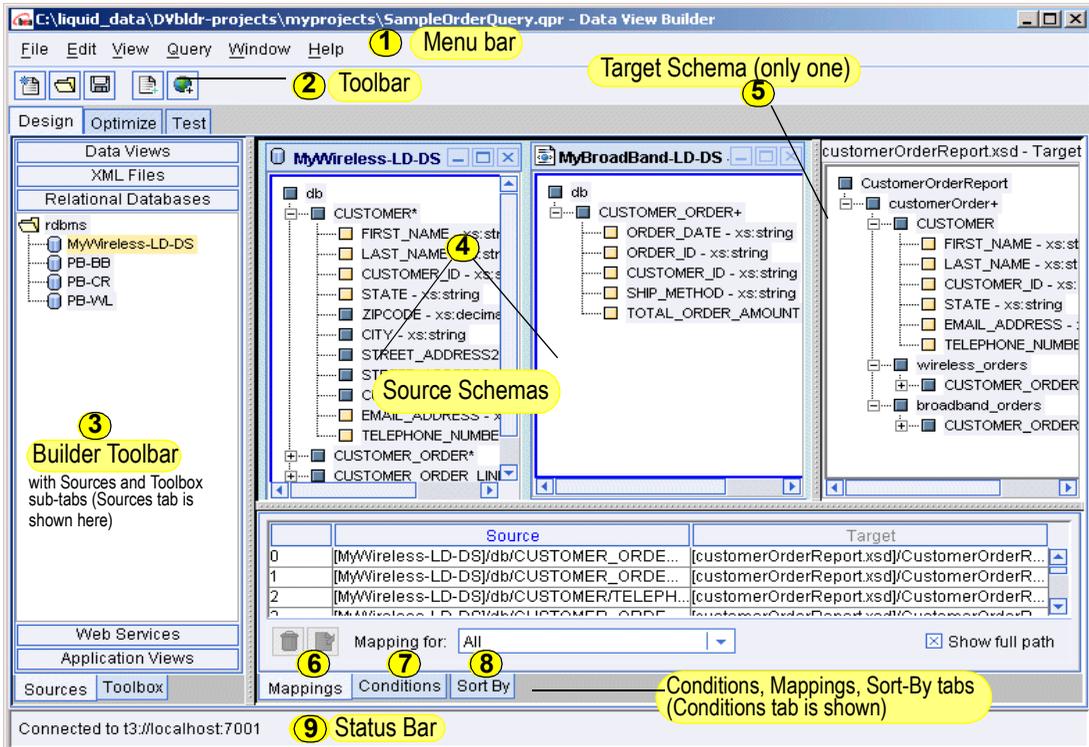
The following sections describe the features available on the Design tab.

- [Overview Picture of Design Tab Components](#)
- [1. Menu Bar for the Design Tab](#)
- [2. Toolbar for the Design Tab](#)
- [3. Builder Toolbar](#)
- [4. Source Schemas](#)
- [5. Target Schema](#)
- [6. Conditions Tab](#)
- [7. Mappings Tab](#)
- [8. Sort By Tab](#)
- [9. Status Bar](#)

Overview Picture of Design Tab Components

The following figure and accompanying sections describe the components on the Design tab. (Click the tab to access it.)

Figure 2-2 Design Tab



Note: Although not entirely specific to the Design tab, the menus, horizontal shortcut toolbar and status bar are also covered in detail in this section since this is the first place you encounter them. Although some menu options and toolbar shortcut buttons do stay the same regardless of which tab you are on, there are mode-specific menus and toolbar options for Design, Optimize, and Test tabs which are explained in those topics.

1. Menu Bar for the Design Tab

The menus provide File, Schema, View, and Window menus as detailed in [Table 2-1](#).

Table 2-1 Menu Bar for the Design Tab

Menu	Description of Menu Options
File Menu	<p>Provides Project-related actions (creating a new project, saving a project, and so on) along with an Exit option that closes the Data View Builder application. For more information on working with projects in the Data View Builder, see “Working With Projects” on page 2-38.</p> <ul style="list-style-type: none">■ New Project—Creates a new “blank slate” project. When you choose this option while you have an unsaved project in the workspace you are given the option to save your current work to a project. If you choose not to save, the query you had in work along with any associated source conditions and schema mappings will be lost.■ Open Project—Opens an existing project you specify.■ Close Project—Closes the current project. If you have not saved your work, you are given an opportunity to do so.■ Open Query—Opens an existing saved query. When you open a saved query, you only see the Test Tab; the Design and Optimize tabs are greyed out. You can then edit, run, and save the query.■ Save Project—Saves the current project. Data View Builder projects are saved with a <code>.qpr</code> filename extension.■ Save Project As—Saves the current project under another file name. Data View Builder projects are saved with a <code>.qpr</code> filename extension.■ Add Selected Schema—Adds/opens the source schema that is selected in the Builder Toolbar to the current project.■ Set Target Schema—Brings up a file browser for browsing to and choosing a target schema file from local system, network drive, or Liquid Data server repository. The file you select is added to the current project as the target schema.■ Set Selected Source Schema as Target Schema—Causes the source schema that is selected on the Builder Toolbar to be set as the target schema in the current project.■ Save Target Schema—Saves the current target schema to the Liquid Data Repository or to a folder location and filename you choose. If you choose Repository when saving a target schema, it saves a relative path to the file in the project file, making the target schema available to other Liquid Data users and ensuring that the target schema is found if the project is run on another server. If you save a target schema to a local file, the fully qualified path is saved in the project file, making the schema accessible only on the local machine.■ Save Query—For a description of this option, see Table 2-3, “Menu Bar for the Test Tab,” on page 2-34.■ Exit—Closes the Data View Builder application.

Table 2-1 Menu Bar for the Design Tab (Continued)

Menu	Description of Menu Options
Edit Menu	<p>Provides standard edit features. These are activated or deactivated depending on what is selected on the User Interface. For example, you can delete a node in a schema so when any schema node is selected “Delete” is active.</p> <ul style="list-style-type: none"> ■ Cut ■ Copy ■ Paste ■ Delete ■ Select All
View Menu	<p>As an alternative to using the tabs the View menu lets you navigate to the following UI views:</p> <ul style="list-style-type: none"> ■ Design—Same as clicking on Design tab. ■ Optimize—Same as clicking on Optimize tab. ■ Test—Same as clicking on Test tab. ■ Sources and Tools—Provides navigation to the tabs (Sources and Toolbox) and panels on the Builder Toolbar. Same as clicking on the associated tab and panel. For example, choosing View—>Sources and Tools—>Relational Databases is the same as clicking on the Sources Tab and then clicking Relational Databases. <p>To help with screen real estate and workspace, the View menu provides toggles to show or hide various windows, tools, and tabs in the Design view. You can show or hide the following:</p> <ul style="list-style-type: none"> ■ Toolbars—Includes submenu with options to show/hide horizontal shortcut Toolbar or Builder Toolbar. ■ Panels—Includes submenu with options to show/hide various windows and tabs. ■ Messages—Brings up a Messages dialog for you to keep notes on queries. ■ Data Types—Toggle to show/hide data types for all source and target nodes in the schema windows, as well as required function parameter types. Clear the Data Types check box to disable this feature. <p>On the menu, an “x” by an option indicates it is currently displayed. By default, all tools, windows and tabs are shown when you first open the Data View Builder.</p>

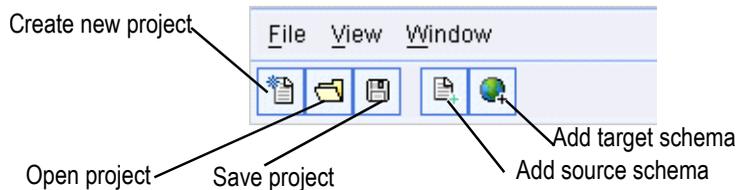
2 Starting the Builder and Touring the GUI

Table 2-1 Menu Bar for the Design Tab (Continued)

Menu	Description of Menu Options
Query Menu	<ul style="list-style-type: none">■ Run Query—Runs the query. (See “6. Run Query” on page 2-36)■ Stop Query Execution—Stops a running query. (See “Stopping a Running Query” on page 2-36.)■ Automatic Type Casting—Toggle to turn automatic type casting on/off. An “X” next to this option indicates that automatic type casting is <i>on</i>. For more information about using automatic type casting see “Using Automatic Type Casting” on page 3-17.■ Automatic Treat-as—Toggle to turn automatic treat-as on/off. An “X” next to this option indicates that automatic type casting is <i>on</i>. When automatic treat-as is on, <code>treat</code> functions are automatically placed in the query whenever there is a type mismatch. For details on the treat functions, see “Treat Functions,” on page A-109.■ Condition Targets—>Advanced View—Toggle to turn Advanced View for manual scoping on/off. For more information on using scoping in Advanced View see “Understanding Scope in Basic and Advanced Views” on page 3-30. <p>For a description of the other options in the Query menu (Run or Stop Query Execution) which are relevant only for running/testing a query, see Table 2-3, “Menu Bar for the Test Tab,” on page 2-34.</p>
Window Menu	<p>The Window menu provides various options for window management:</p> <p>As you open source schema windows they are listed in the Window menu so that you choose an open schema from the menu to navigate to it.</p>
Help Menu	<p>Provides online documentation for the Data View Builder.</p> <p>Note: For this release Liquid Data, the online help for the Data View Builder simply links into the main topics in online documentation for the Data View Builder.</p>

2. Toolbar for the Design Tab

The toolbar, located directly below the menus, provides shortcuts to a subset of commonly used actions also available from the menus.

Figure 2-3 Toolbar

3. Builder Toolbar

The Builder Toolbar includes two subtabs:

- Sources Tab—Provides access to the XML schema representations for data sources configured in the Liquid Data server. This is where you can get source schema windows for a data source.
- Toolbox Tab—Provides access to functions, constants, query parameters, and other components used in query design.

Sources Tab

The Sources tab on the Builder Toolbar contains the data sources configured on the Liquid Data Server to which you are connected. Note that a data source type only shows up as a button on the Builder Toolbar if it has been configured in the Server to which you are connecting.

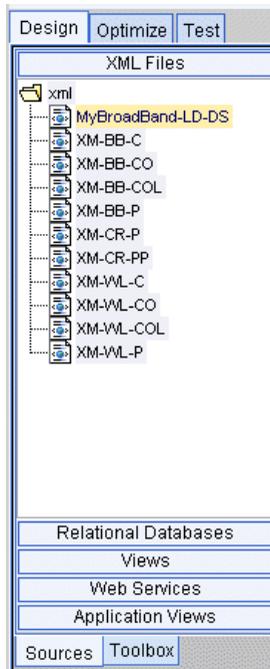
- Relational Databases
- XML Files
- Web Services
- Application Views (defined with the Application Integration)
- Data Views

Note: For a detailed introduction to these data sources, see [“Different Kinds of Data Sources”](#) on page 1-7 in Chapter 1, [“Overview and Key Concepts.”](#)

2 Starting the Builder and Touring the GUI

To open a schema for a data source, click on the data source type (for example Relational Databases) to get a list of configured data sources of that type. Then double-click on the particular data source you want to work with. The schema window for that source is displayed in a movable window on the Liquid Data desktop.

Figure 2-4 Builder Toolbar: Sources Tab



Toolbox Tab

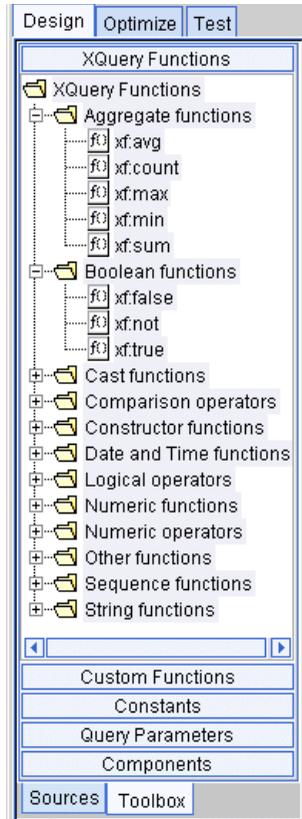
The Toolbox tab on the Builder Toolbar provides the following tools to use in query construction:

- [XQuery Functions](#) (information on [The Function Editor](#) is included here)
 - [Custom Functions](#) (included with Functions in a custom grouping you define)
- [Constants](#)
- [Query Parameters: Defining](#)

- Components

Note: Any custom functions configured in the Liquid Data Server through the WebLogic Server Administration Console will show up on the Builder Toolbar on the Functions panel along with the standard functions provided.

Figure 2-5 Builder Toolbar: Toolbox Tab



XQuery Functions

XQuery Functions are built-in code modules that return a value when they run. The XQuery Functions panel provides a library of standard W3C functions compliant with the W3C *XQuery 1.0 and XPath 2.0 Functions and Operators* specification. (See [Figure 2-5](#) for an example of the Functions panel on the Builder Toolbar Toolbox tab.)

2 Starting the Builder and Touring the GUI

In Data View Builder, the Functions are displayed in the Builder Toolbar on the Toolbox tab XQuery Functions panel by category names like Aggregate Functions, Boolean Functions, Cast Functions, and so on. To view all the functions in a category or group, expand the group node.

You can double click or drag and drop a function object to the Liquid Data desktop where it appears in a tree format showing the number and type of parameters required.

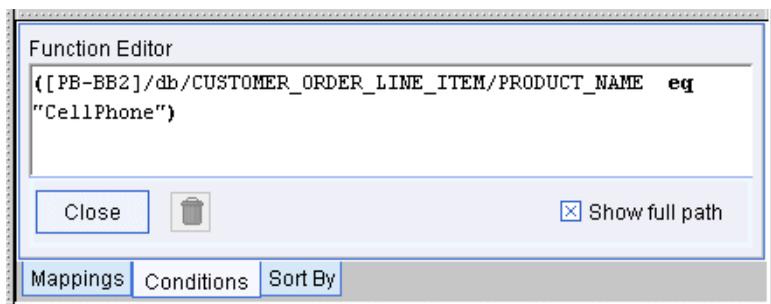
A copy of any mapped function saves automatically with the project when you close it. The saved function (with associated parameters) appears in the Components panel when you reopen the project. If you do not map the new function and you terminate the session, Data View Builder discards it and it does not appear in the Components panel.

Each function has a specification for required parameters and expected behavior. Some functions cannot be used in the work area, but must appear only on the desktop. For complete information about each function, its parameters, and expected behavior, see [Appendix A, “Functions Reference.”](#) For more detailed information, see the [W3C XQuery 1.0 and XPath 2.0 Functions and Operators](#) specification.

The Function Editor

The functions editor gives you a space to create functions using drag-and-drop and to view existing functions in your project.

Figure 2-6 Function Editor



There are two ways to open the Functions Editor:

- When you drag and drop a function into the work area the Function Editor is displayed with the chosen function and anticipates the equation with placeholder values (for example *anyValue1 eq anyValue2*).

- You can open the Functions Editor to view or modify an existing function by selecting a condition in a particular row and then clicking the edit button.

Figure 2-7 Click Edit Button to Get the Functions Editor



Click Close to close the Functions Editor.

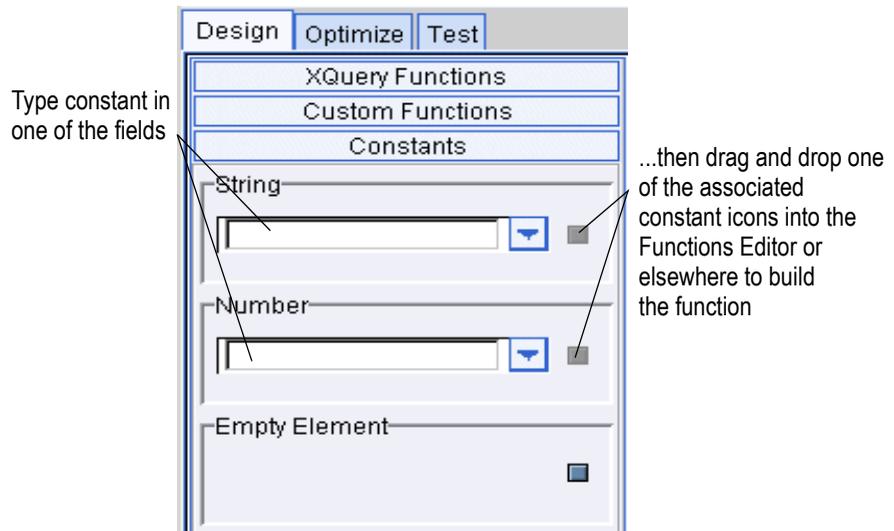
Custom Functions

If you have custom functions configured through the Administration Console, these will show up in the Data View Builder on the Toolbar Functions tree in a custom group. The name of the group is what you specify in the “presentation group” element in the custom functions library definition (.CFLD) file. If no grouping label is specified “Ungrouped”. For more information on this, see “Contents of a CFLD File” and “Structure of a CFLD File” in [“Using Custom Functions”](#) in *Invoking Queries Programmatically*.

Constants

You can use the Constants panel to create function parameters with constant values.

Figure 2-8 Builder Toolbar: Toolbox Tab: Constants



Choose the type of constant based on how you want the data to be considered in the query. Strings are alphanumeric values that typically contain alphabetic letters, special characters, and digits used in non-numeric comparisons. Names, zip codes, phone numbers, and street addresses are typical examples of string values.

Numbers can be integers (positive or negative), decimal values, or floating point expressions. The Empty element enables you to force an element to appear in the query. We expect mapped data elements to appear in the query result, but you may wish to see other data elements appear that are not mapped. If you drag and drop the Empty element onto a node, that node will appear in the query result.

To include a String, Number, or Empty element constant as a function parameter, follow steps similar to those shown in this example:

1. Drag an appropriate function to a row on the Condition tab or to the Liquid Data desktop. For example: choose the `startswith` function. You get the following placeholder in the Functions Editor:

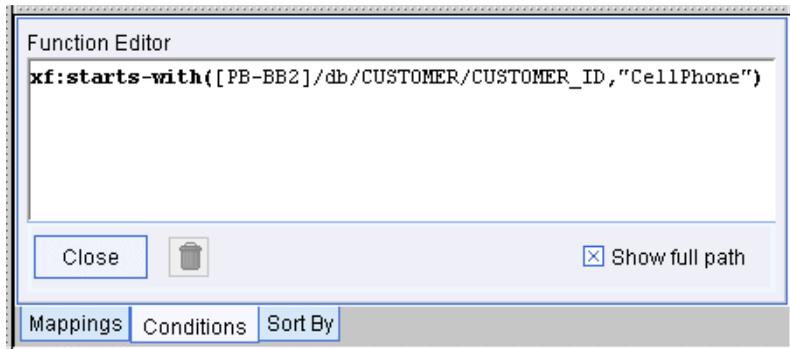
```
xf:starts-with (str1, str2)
```

2. Drag an appropriate source node onto the first string placeholder (*str1*). For example, choose `CustomerID` from a source schema.

3. Type a value in the String constant text box. For example, `CUS`. Drag the Constants icon onto the second string placeholder (`str2`).

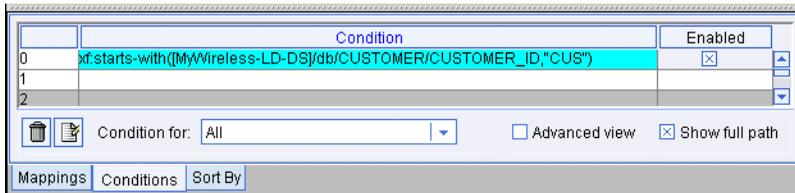
The condition appears in the Functions Editor as shown in the following figure.

Figure 2-9 Condition with starts-with Constant in Functions Editor



Close the Functions Editor by clicking the Close button. The new condition you created is also now displayed in the Source column on the Condition tab.

Figure 2-10 Condition with starts-with Constant in Row on Conditions Tab



Note: If you design a query with a constant, and then design another query using a query parameter that specifies exactly the same value, the generated XQuery translation is different even though the functionality in each query is exactly the same.

Query Parameters: Defining

Query parameters can be strings, integers, floating point numbers, boolean expressions, or date and time types. They are variables that you define with no static value. On the Test tab, you can supply a different value each time you run the query (see “4. Query Parameters: Submitted at Query Runtime” on page 2-35).

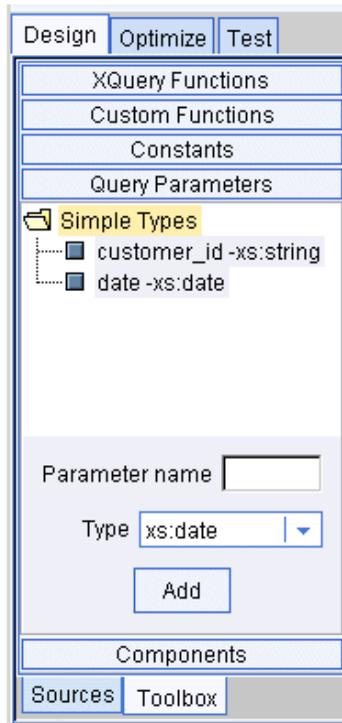
2 Starting the Builder and Touring the GUI

The Query Parameter section of the Toolbox provides a text box where you can enter a new parameter value to be stored.

- Type a value that is one of the five supported types of parameters and click the drop-down list to select the type of parameter.
- Click Add to save it to the Query Parameter resource tree.

The parameters you add are added to the tree in the Query Parameters panel.

Figure 2-11 Builder Toolbar: Toolbox Tab: Query Parameters



You can invoke these variables when you build conditions. As a convenience, you can:

- Right-click a parent node and click Expand to show all child nodes.
- Right-click a child node and click Delete or Rename.

[Table 2-2](#) describes supported simple query parameter data types.

To use a simple query parameter, drag and drop a parameter from the Query Parameter resource area to the appropriate item of source data. Then, when you run your query, a window will appear where you can enter your test parameter.

To see use of a simple query parameter, please see the Order Query demo, available from the Liquid Data documentation home page.

<http://e-docs.bea.com/liquiddata/docs10/interm/demopage.html>

Note: If you design a query with a constant, and then design another query using a query parameter, the generated XQuery translation is different even though the functionality in each query is exactly the same.

Table 2-2 Query Parameter Types

Parameter Type	Examples
Boolean (<code>xs:boolean</code>)	Boolean expressions test true or false. You can specify that the Boolean Query Parameter has an implicit definition of <code>True</code> or <code>False</code> , then use it as query resource.
Byte (<code>xs:byte</code>)	A positive or negative whole number. The maximum value is 127 and the minimum value is -128. For example: <ul style="list-style-type: none"> ■ -1 ■ 0 ■ 126 ■ +100
Date (<code>xs:date</code>)	Input must be in this format: <code>MMM dd, YYYY</code> For example: JUN 12, 2002
Date and Time (<code>xs:dateTime</code>)	Input must be in this format: <code>MMM dd, YYYY HH:MM:SS AM/PM</code> For example: MAY 12, 2002 12:12:11 AM

Table 2-2 Query Parameter Types

Parameter Type	Examples
Decimal (<code>xs:decimal</code>)	<p>A precise real number (negative or positive) that can contain a fractional part. If the fractional part is zero, the period and following zero(s) can be omitted. For example:</p> <ul style="list-style-type: none">■ -1.23■ 12678967.543233■ +100000.00■ 210.
Double (<code>xs:double</code>)	<p>A real number (negative or positive) that can contain fractional part. For example: 3.159</p> <p>Liquid Data does not support floating point formats expressed in fractions ($\frac{1}{2}$) or IEEE floating point notation (3E-5).</p>
Floating Point (<code>xs:float</code>)	<p>A real number (negative or positive) that can contain a fractional part. For example:</p> <ul style="list-style-type: none">■ 100.0■ -100.5 <p>Liquid Data does not support floating point formats expressed in fractions ($\frac{1}{2}$) or IEEE floating point notation (3E-5).</p>
Int (<code>xs:int</code>)	<p>A positive or negative whole number. The maximum value is 2147483647 and minimum value is -2147483648. For example:</p> <ul style="list-style-type: none">■ -1■ 0■ 126789675■ +100000
Integer (<code>xs:integer</code>)	<p>A positive or negative whole number. The maximum value is 2147483647 and minimum value is -2147483648. For example:</p> <ul style="list-style-type: none">■ 1■ -100■ +100

Table 2-2 Query Parameter Types

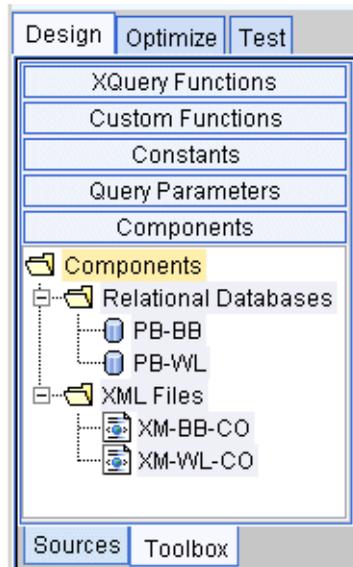
Parameter Type	Examples
Long (<code>xs:long</code>)	<p>A positive or negative whole number. The maximum value is 9223372036854775807 and minimum value is -9223372036854775808. For example:</p> <ul style="list-style-type: none">■ -1■ 0■ 12678967543233■ +100000
Short (<code>xs:short</code>)	<p>A positive or negative whole number. The maximum value is 32767 and minimum is -32768. For example:</p> <ul style="list-style-type: none">■ -1■ 0■ 126789■ +10000
String (<code>xs:string</code>)	<p>An alphanumeric expression such as:</p> <ul style="list-style-type: none">■ Smith■ Jones■ 12345 State St. <p>Note: An unspecified value for a query parameter of type String is considered an empty string.</p>
Time (<code>xs:time</code>)	<p>Input must be in this format: HH:MM:SS AM/PM</p> <p>For example:</p> <p>01:02:15 AM</p>

Components

The Components panel shows the structure of the current project in Design View. All elements of the query, except the target schema appear in this view of the project, including any data source schemas you are using or functions that you map with parameters.

If a particular component schema is unavailable when a project is re-opened, the schema will still be listed, but it will be flagged as unavailable (off-line) and a red mark will appear over the schema name.

Figure 2-12 Builder Toolbar: Toolbox Tab: Components



Any component that appears in this panel can be minimized on the Liquid Data desktop by double clicking the appropriate node. Click again and the component reappears on the desktop. The target schema does not appear in the Components panel because you cannot close it while working on the project.

When you save a project by name and reopen it, the project components appear in this window, but minimized on the desktop. You can move them to the desktop by double clicking a selected component. When you reopen a saved project, the output schema appears directly on the desktop instead of in the Components tree.

You can right-click any parent node and click Edit, Delete, or Rename to complete those tasks.

4. Source Schemas

Source schema windows show XML schema representations of the structure of the data in the selected data source. Used to create source conditions and mappings to a target schema. You can have multiple data source schemas open on the Liquid Data desktop as needed.

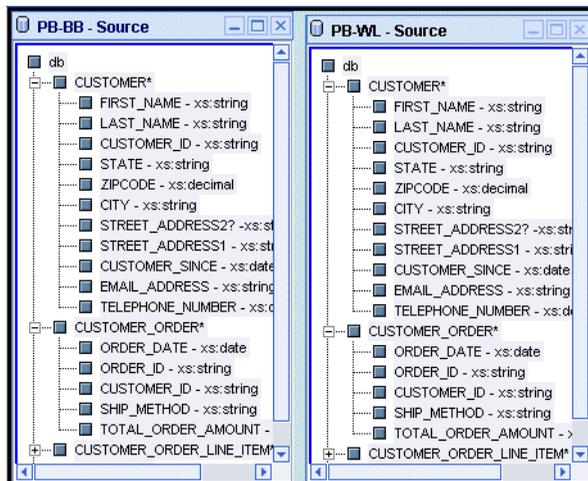
Note: For a detailed description of the special characters used to identify characteristics of schema nodes, see [“Special Characters: Occurrence Indicators”](#) on page 2-39.

To open a schema for a data source:

1. Click on the Sources tab on the Builder Toolbar (if the Sources tab is not already showing).
2. Click on the data source type (for example Relational Databases) to get a list of configured data sources of that type.
3. Double-click on the particular data source you want to work with.

The schema window for that source is displayed in a movable window on the Liquid Data desktop.

Figure 2-13 Source Schemas



5. Target Schema

The Target Schema window shows the XML schema representation for the structure of the *target* data (query result).

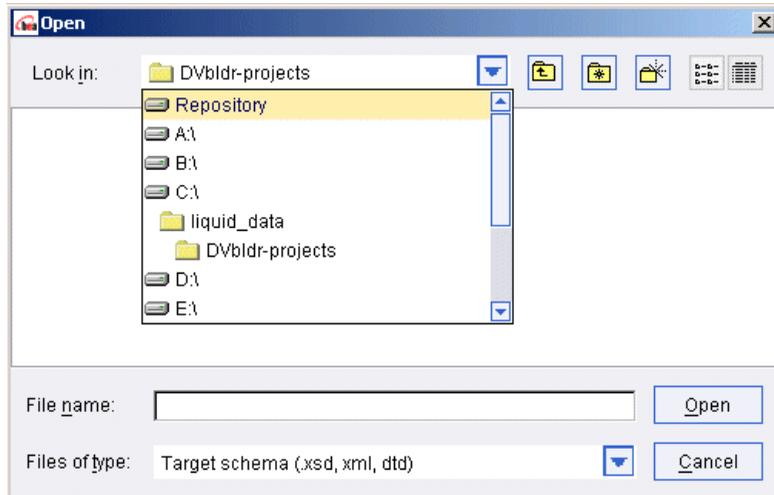
Only one target schema per project is allowed. If you have a target schema open and decide to choose another, the current target schema is closed and the new one replaces it.

Note: For a detailed description of the special characters used to identify characteristics of schema nodes, see [“Special Characters: Occurrence Indicators”](#) on page 2-39.

To open and set a target schema for a project:

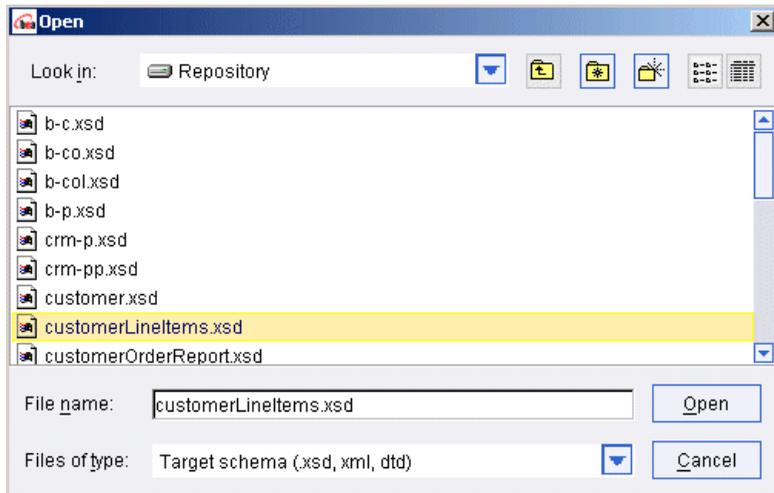
1. Choose the menu item File—>Set Target Schema.

This brings up a file browser.



If you choose Repository in the Open dialog, the Data View Builder displays any target schemas saved in the Liquid Data repository.

2. Navigate to the schema you want to use, select the file and click Open in the file browser.

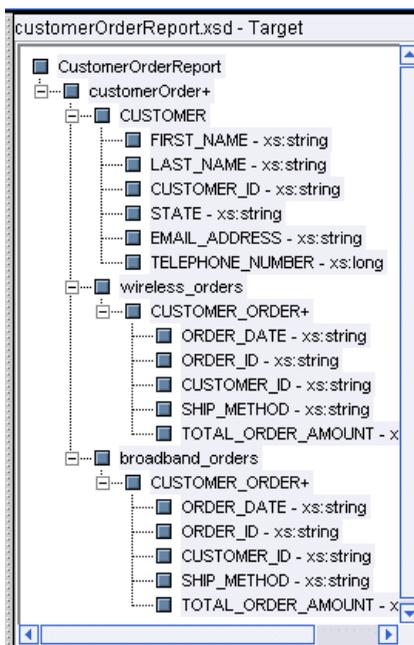


The target schema is displayed as a docked on the right side of the Design tab.

2 Starting the Builder and Touring the GUI

(You can also choose the menu item File—>Set Selected Source Schema as Target Schema to add a source schema selected on the Builder Toolbar as the target schema.)

Figure 2-14 Target Schema



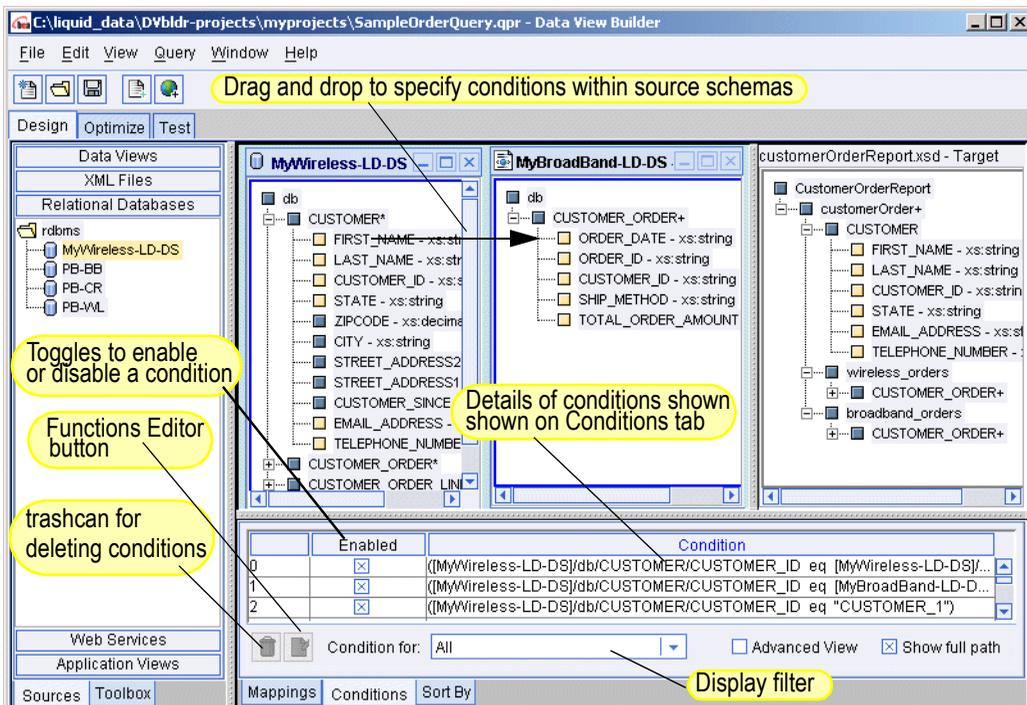
6. Conditions Tab

The Conditions tab shows:

- In Basic mode, conditions (filtering) defined for the source data (see [“Conditions” on page 2-25](#))
- In Advanced mode, conditions (filtering) defined to force Scope for the target data or query result (see [“Advanced View for Defining Explicit Scope for Conditions” on page 2-27](#))

The Conditions area functions both as a tracking and reflection tool, and as a workspace that you can manipulate directly. Whenever you do a drag-and-drop operation that causes an update to Conditions, the Conditions tab is automatically displayed.

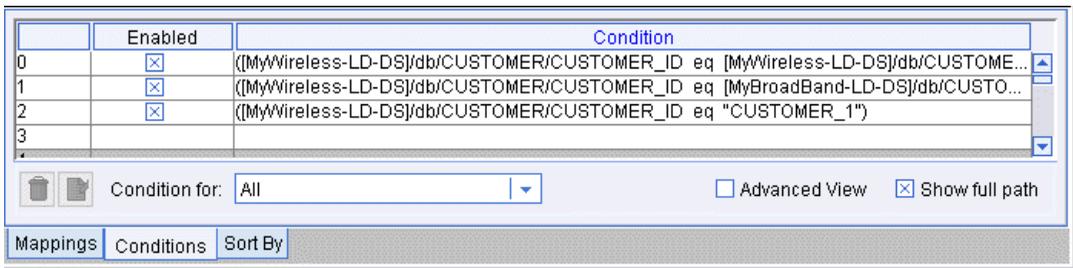
Figure 2-15 Conditions Tab on the Design tab



Conditions

The Conditions section shows conditions (filters) for source data. As you build up the query by creating drag-and-drop source-to-source node relationships among data source schemas, the implied condition statements are recorded and reflected as joins under the Conditions. Even if you don't drag and drop anything directly into the Conditions tab, you will see the appropriate conditions building up here as a result of your work with the source schemas. (When you drag and drop a source element onto another source element, the equals function is used by default to create a simple *join*.)

Figure 2-16 Conditions Tab in Basic View



You can also use the Conditions area as a workspace to explicitly drag-and-drop elements of a query statement into the rows under Conditions to build up the query. You can drag-and-drop elements and attributes from source schemas as well as functions, constants, and parameters from the Builder Toolbar “Toolbox” tab directly into the rows under Conditions to craft conditions statements.

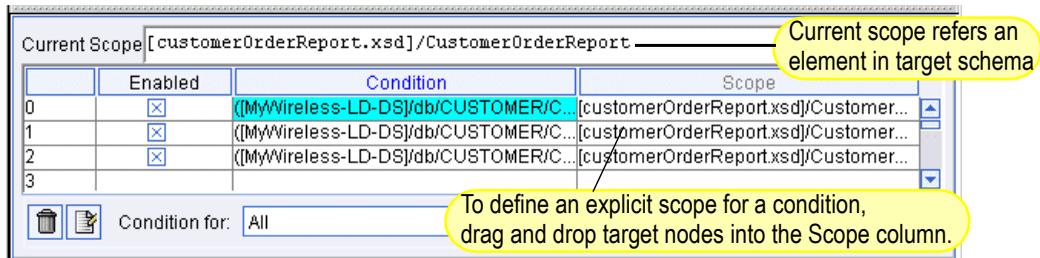
This tab includes the following features to facilitate working with conditions:

- **Function Editor**—To edit an existing condition, select it and click on the Function Editor. You can also drag and drop a function from the Functions panel on the Toolbox panel into an empty row on the Conditions tab. For more about working with the Function Editor, see [“XQuery Functions” on page 2-11](#) and [“The Function Editor” on page 2-12](#).
- **Trashcan for Deleting Conditions**—To remove a condition, select the row that contains the condition you want to remove and click the trashcan.
- **Enabled/Disabled Toggle**—For each condition you can use the Enabled/Disabled toggle to include the selected condition in the query or disable it. Select the row that contains the condition you want to enable or disable and then click the Enabled/Disabled toggle for that condition. When a condition is *disabled*, it will not be used to generate the XQuery. When a condition is *enabled* it will be included when the XQuery is generated.

Advanced View for Defining Explicit Scope for Conditions

When you click the Advanced View toggle, the Conditions tab displays a column for defining explicit *scope* for each condition.

Figure 2-17 Conditions Tab in Advanced View Showing Explicit Scope



The Scope area on the Conditions tab shows any explicit narrowing conditions (filters) you define for the target data to refine the query result. In basic mode (with Advanced toggle *off*) Data View Builder creates queries based on the scope implied by the source conditions you create and the structure of the target schema (*implicit scope*). In other words, by default the implicit scope is auto-generated by the Data View Builder. The auto-generated, implicit scope should be sufficient for most cases. However, there may be situations in which you want to control scoping explicitly. In these cases, you can switch to the Advanced view.

A scope setting affects the placement of a *where* clause in the XQuery generation. The Data View Builder best guess at implicit scope will satisfy most cases, and you will generally not have to specify scope. For cases where you need to explicitly define scope to force the *where* clause to the right place in the query or sometimes to force it to be there at all, you can do this directly by dragging the appropriate node in the target schema into a row under Scope.

For more information and examples about when and how to set scope, see [“Understanding Scope in Basic and Advanced Views”](#) on page 3-30 in Chapter 3, [“Designing Queries.”](#)

Returning to Basic View (Automated Scope)

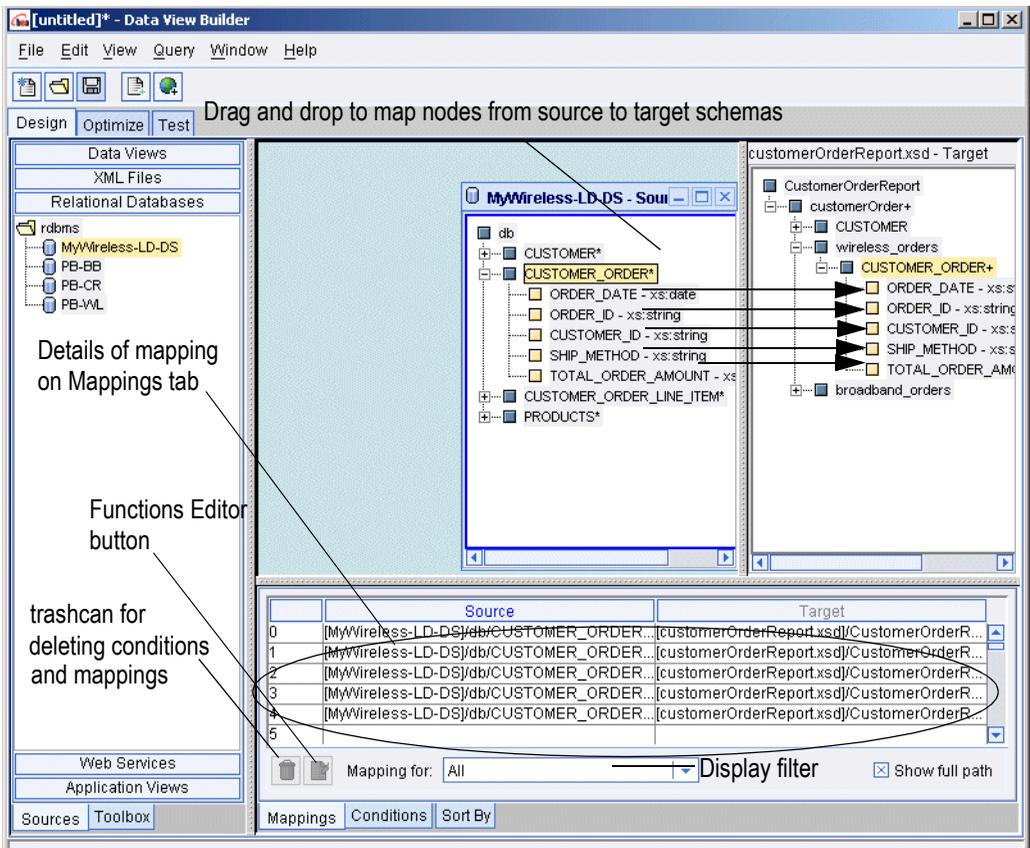
When you toggle Advanced View *off* (no X showing next to Advanced View), Data View Builder returns to automatic scoping mode and discards the changes you made in manual mode. The Current Scope text box and the Targets column disappear.

7. Mappings Tab

The Mappings tab shows source-to-target mappings that will define the structure of the query result. As you drag-and-drop source elements onto target elements among the schema windows, the Mappings tab records these relationships, which build up the shape the data will take in the query result. For example, dragging and dropping `FIRST_NAME` and `LAST_NAME` elements from `CUSTOMER` in a source schema to the associated `CUSTOMER` elements in the target schema specifies that in the query result customers will be identified with first and last names as defined.

Whenever you do a drag-and-drop operation that causes an update to Mappings, the Mappings tab is automatically displayed.

Figure 2-18 Mappings Tab



Deleting a Mapping

To delete a mapping, select the row on the Mappings tab that contains the source-to-target mapping you want to delete (selected mapping is highlighted) and click the trashcan.

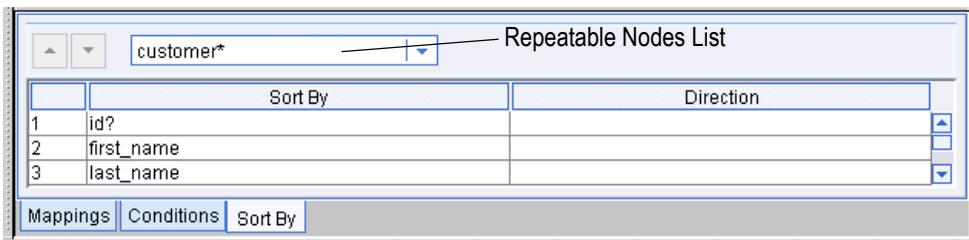
8. Sort By Tab

The Sort By tab specifies how the result should be ordered and a list of candidate nodes that you can order. [Figure 2-19](#) shows the order of a repeatable node segment of the target schema. The drop-down list shows all repeatable data nodes in the target schema marked with an asterisk (*) or a plus sign (+). A repeatable node is the parent of child nodes that can appear in the query result once for every instance of a match. A repeatable node is an ancestor to one or more nodes that will represent unique data returned by the query.

The blue arrows move rows up and down. These icons are enabled only when you select a data item that can move up or down. The drop-down list shows the repeatable nodes with subordinate nodes that can be sorted. When you select a repeatable node from the drop-down list, the associated child nodes appear in the Sort By list. Move these child nodes up or down to specify how the result should be sorted. For example, a CUSTOMER* element can be sorted first by LAST_NAME and then by FIRST_NAME by having the LAST_NAME row at the top and the FIRST_NAME row directly beneath it.

An item can be moved if it is assigned an ascending or descending attribute in the source schema. (The database administrator or data architect who creates the source schema specifies this.) Items with ascending or descending attributes can be moved up only if there is another item above, and they can be moved down only if the next item down also has an ascending or descending attribute.

Figure 2-19 Sort By Tab



9. Status Bar

The Status Bar is a horizontal bar at the bottom of the Data View Builder that provides status information about current actions and processes.

Figure 2-20 Status Bar



Optimize Tab

The Optimize tab is where you can optionally add more information such as “hints” to data sources to improve query performance.

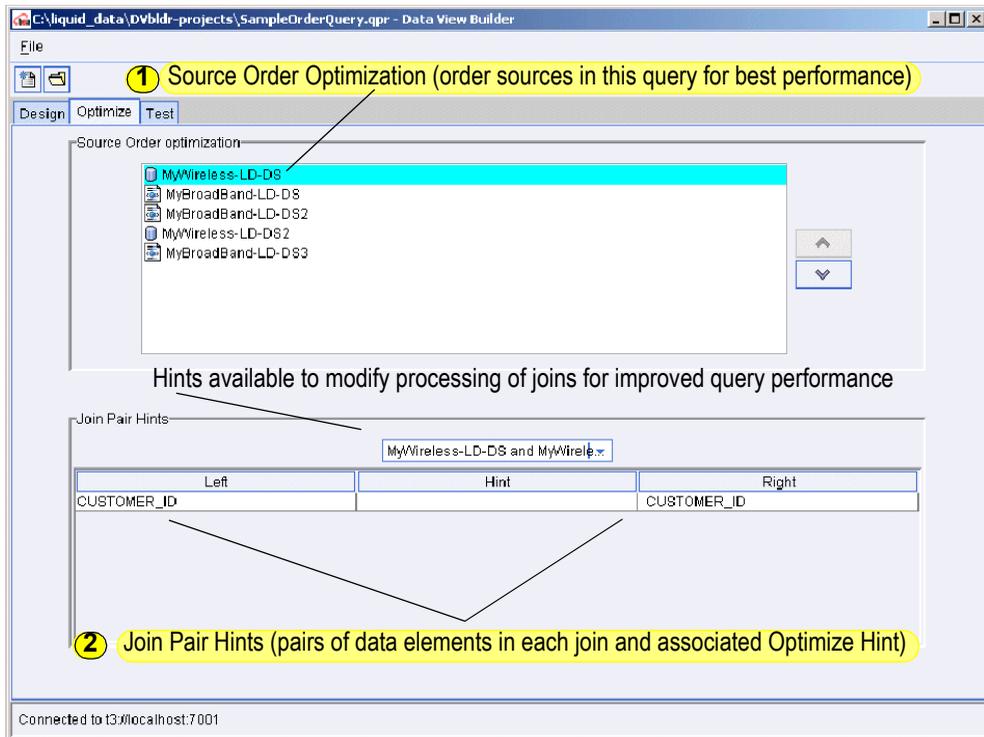
The following sections describe the features available on the Optimize tab.

- [Overview Picture of Optimize Tab Components](#)
- [1. Source Order Optimization](#)
- [2. Join Pair Hints](#)

Overview Picture of Optimize Tab Components

The following figure and accompanying sections describe the components on the Optimize tab. (Click the tab to access it.)

Figure 2-21 Optimize Tab



Note: The Optimize tab contains a subset of the menu options and toolbar buttons available on the Design tab. For a full description of these options, see “1. Menu Bar for the Design Tab” on page 2-6 and “2. Toolbar for the Design Tab” on page 2-8.

1. Source Order Optimization

You can re-order source schemas on the top frame on the Optimize tab to improve query performance. To move a schema up or down, select the schema and click the up or down arrow buttons to the right of the list of schemas.

When a query uses data from two sources, the Liquid Data Server brings the two data sources into memory and creates an intermediate result (*cross-product*) using the two sources. If you specify more than two sources, the Liquid Data Server creates a

cross-product of the first two sources, then continues to integrate each additional resource, one at a time, in the order that they appear in FOR clauses. The intermediate result grows with each integration, until all sources are accounted for.

The size of a source is the number of tuples, or records, used in the query from that source. The size of the intermediate result depends on the input size of the first source multiplied by the input size of the second source and so on. A query is generally more efficient when it minimizes the size of intermediate results. You can re-order source schemas in certain situations to improve performance.

For detailed information on how to optimize a query by ordering source schemas, see [Chapter 4, “Optimizing Queries.”](#)

2. Join Pair Hints

A query hint is a way to supply more information to the Liquid Data Server about the amount of data each source contains when processing a query. The Join Hints frame contains a drop-down list of data source pairs, and a table that shows all the joins for each pair. Only source pairs that have join conditions across them appear in the drop-down list. For each join condition in the table, you can provide a hint about how to join the data most efficiently.

For detailed information on how to optimize a query by using optimization *hints*, see [Chapter 4, “Optimizing Queries.”](#)

Test Tab

The Test tab is where you view the generated XQuery language interpretation of the query elements you developed on the Design and Optimize tabs, and run the query against your data sources to verify the result and evaluate performance.

From this view, you can provide different parameters to the query before you run it.

The following sections describe the features available on the Test tab.

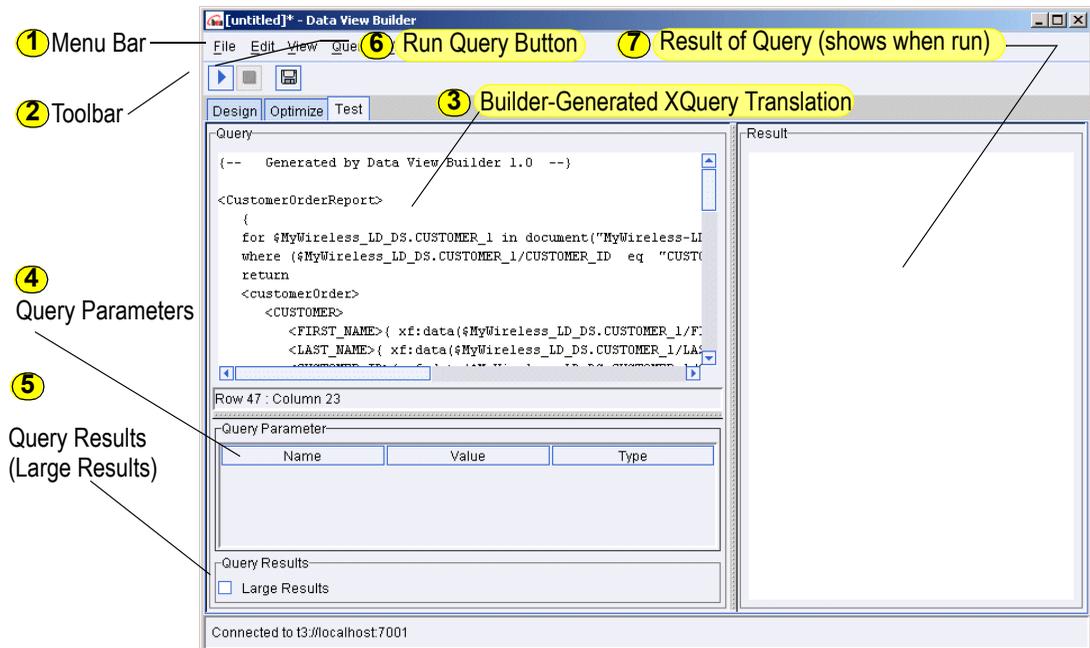
- [Overview Picture of Test Tab Components](#)
- [1. Menu Bar for the Test Tab](#)
- [2. Toolbar for the Test Tab](#)

- 3. Builder-Generated XQuery
- 4. Query Parameters: Submitted at Query Runtime
- 6. Run Query
- 7. Result of a Query

Overview Picture of Test Tab Components

The following figure and accompanying sections describe the components on the Test tab. (Click the tab to access it.)

Figure 2-22 Test Tab



1. Menu Bar for the Test Tab

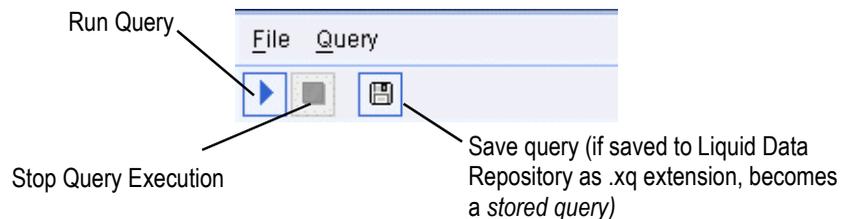
Table 2-3 Menu Bar for the Test Tab

Menu	Description of Menu Options
File Menu	<p>Provides most of the same options as shown on the Design tab menu bar with one additional menu option as follows:</p> <ul style="list-style-type: none">■ Save Query—Saves the current query to a file you specify. The file must be saved with a <code>.xq</code> extension. (If you do not add a <code>.xq</code> extension, Data View Builder will append it automatically.) If the query is saved into the <code>stored_queries</code> folder in the Liquid Data server Repository, it is considered a <i>stored query</i> in Liquid Data. For more details on saving a query, see “Saving a Query” on page 5-6 in Chapter 5, “Testing Queries.” <p>For a description of the other File menu items available from the Test tab (which are a subset of those on the File menu for the Design tab), see Table 2-1 in “Design Tab” on page 2-4</p>
Query Menu	<p>Provides the following options related to running a query:</p> <ul style="list-style-type: none">■ Run Query—Runs the query. (See “6. Run Query” on page 2-36)■ Stop Query Execution—Stops a running query. (See “Stopping a Running Query” on page 2-36.) <p>The Query menu options for Automatic Type Casting and Condition Targets—>Advanced View are more relevant to designing a query and, therefore, are described in “1. Menu Bar for the Design Tab” on page 2-6.</p>

2. Toolbar for the Test Tab

The toolbar, located directly below the menus, provides shortcuts to a subset of commonly used actions also available from the menus.

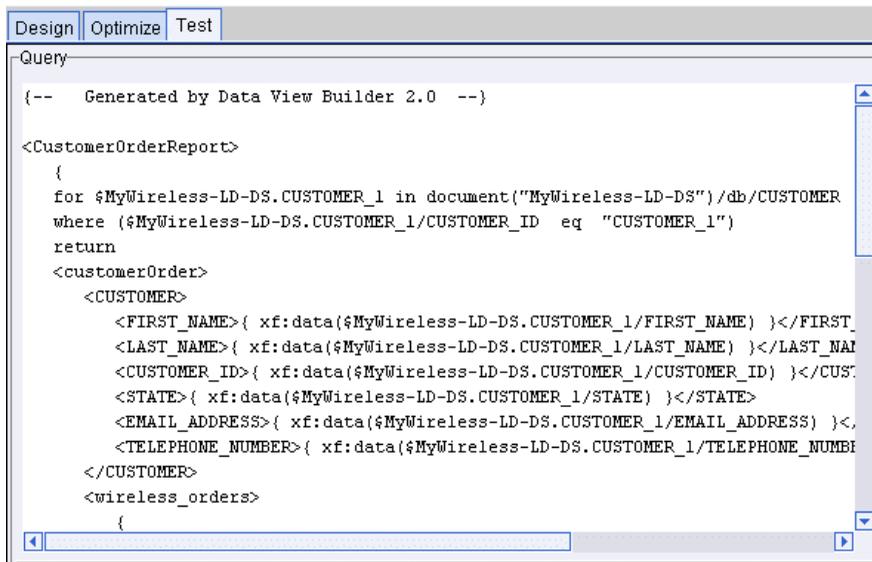
Figure 2-23 Toolbar on the Test Tab



3. Builder-Generated XQuery

The query you developed on the Design and Optimize tabs is shown in XQuery language in the “Query” window on the upper left panel on the Test tab.

Figure 2-24 Builder-Generated XQuery Shown in Query Window



```

-- Generated by Data View Builder 2.0 --

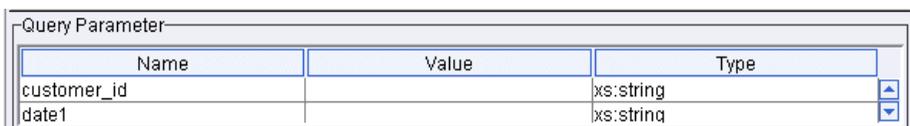
<CustomerOrderReport>
{
  for $MyWireless-LD-DS.CUSTOMER_1 in document("MyWireless-LD-DS")/db/CUSTOMER
  where ($MyWireless-LD-DS.CUSTOMER_1/CUSTOMER_ID eq "CUSTOMER_1")
  return
  <customerOrder>
    <CUSTOMER>
      <FIRST_NAME>{ xf:data($MyWireless-LD-DS.CUSTOMER_1/FIRST_NAME) }</FIRST_NAME>
      <LAST_NAME>{ xf:data($MyWireless-LD-DS.CUSTOMER_1/LAST_NAME) }</LAST_NAME>
      <CUSTOMER_ID>{ xf:data($MyWireless-LD-DS.CUSTOMER_1/CUSTOMER_ID) }</CUSTOMER_ID>
      <STATE>{ xf:data($MyWireless-LD-DS.CUSTOMER_1/STATE) }</STATE>
      <EMAIL_ADDRESS>{ xf:data($MyWireless-LD-DS.CUSTOMER_1/EMAIL_ADDRESS) }</EMAIL_ADDRESS>
      <TELEPHONE_NUMBER>{ xf:data($MyWireless-LD-DS.CUSTOMER_1/TELEPHONE_NUMBER) }</TELEPHONE_NUMBER>
    </CUSTOMER>
  <wireless_orders>
    {

```

4. Query Parameters: Submitted at Query Runtime

You can use the Query Parameters panel to add variable values to a query each time you run it. The list of variables depends on the number of variables you defined as Query Parameters on the Design tab (see “[Query Parameters: Defining](#)” on page 2-15) and which ones appear as one or more function parameters.

Figure 2-25 Query Parameters Settings on Test Tab



Name	Value	Type
customer_id		xs:string
date1		xs:string

5. Query Results - Large Results

If you anticipate a large set of data coming back in the query result, click Large Results (an X in the box indicates this feature is *on*). The default is *off* (no X).

When this option is on, Liquid Data uses swap files to temporarily store results on disk in order to prevent an out-of-memory error when the query is run.

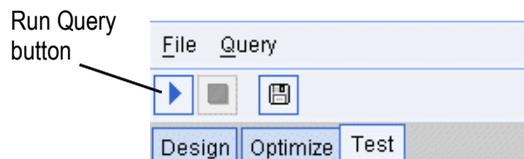
Figure 2-26 Specifying Large Results



6. Run Query

To run a query, click the Run Query button on the toolbar in the upper left of the Test tab. (You can also choose the Run Query option from the Query menu.)

Figure 2-27 Click the “Run Query” Button to Run the Query

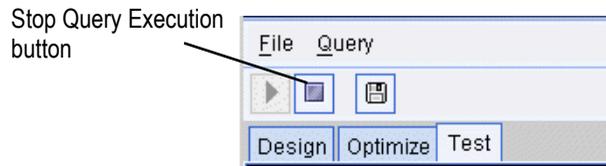


The query is run against your data sources and the result is displayed in the Results panel in XML format.

Stopping a Running Query

You can stop a running query before it has finished processing by clicking the Stop Query Execution button in the toolbar. (You can also choose the Stop Query option from the Query menu.)

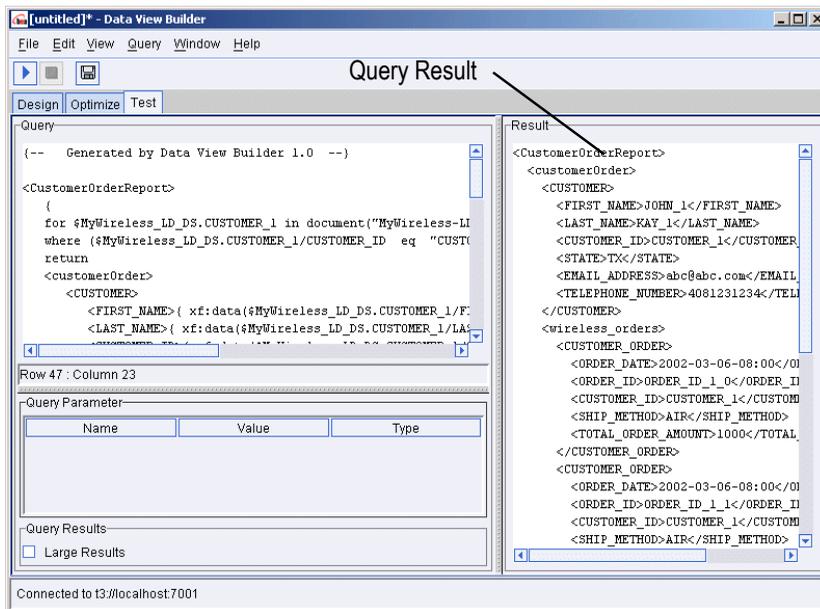
Figure 2-28 Click the “Stop Query Execution Button” to Stop a Running Query



7. Result of a Query

When you run a query, the result is displayed in the Results window on the Test tab in XML format.

Figure 2-29 Query Result is Shown on Test Tab When Query is Run



Working With Projects

It is a good practice to save the project file immediately once you have chosen and set up a target schema, and started creating conditions and mappings for a query. Save frequently or after you make a significant change to avoid losing your work. To save the project for the first time.

To save a project choose File—>Save Project or File—>Save Project As from the menus (or click the “Save the project” toolbar button). Data View Builder projects are saved with a `.qpr` filename extension. (For a complete description of options available for handling projects, see [Table 2-1](#) in “1. Menu Bar for the Design Tab” on page 2-6.

To Make a Project Portable, Save Target Schema to Repository

For the project to be portable so that other users can open the project and use it, the target schema must be saved to the Liquid Data server repository on the server where the project will be used.

Saving a Project is Not the Same as Saving a Query

Please keep in mind that “saving a *project*” is not the same as “saving a *query*”. Saving a project creates a Data View Builder `.qpr` file that includes the conditions and mappings for source and target schemas used in a particular query. You can re-open any project in the Data View Builder, modify the conditions and mappings on the XML schemas, and re-optimize or re-run the query from within the Builder tool.

However, saving a project does not make the query in that project available as a *stored query* in Liquid Data. To create a stored query, you need to use the *Save Query* option on the Test tab. For more information on saving a query, see “[Saving a Query](#)” on page 5-6 in [Chapter 5](#), “[Testing Queries](#).”

Using Schemas Saved With Projects

When you save a project, the schema definitions of all source and target schemas that you mapped in the project are saved. When you reopen the project, Data View Builder first looks for the schema definitions in the Liquid Data repository.

If a schema definition is unavailable, the schema definition saved in the project file is used. Data View Builder adds the schema to the list of available resources, but flags it as offline by putting a red mark over the schema name. A warning is also generated in the Administration Console log that queries using this schema will not run.

Offline resources are available only to the previously associated project.

Special Characters: Occurrence Indicators

The Data View Builder uses a set of special characters (occurrence indicators) to indicate the number of items in a sequence. Occurrence indicators are generally used to specify characteristics for elements or attributes in schemas, but are also found elsewhere in the Builder user interface (UI) where they are needed to specify occurrence characteristics. You can apply these characteristics to elements and attributes of schemas that you build or modify by accessing the right-mouse click pop-up menu on schema nodes.

Table 2-4 Occurrence Indicators in Data View Builder

Character	Description
Question mark (?)	Indicates zero items or one single item. The item is optional and does not have to be included or mapped.
Asterisk (*)	Indicates zero or more items. This item is optional and multiple occurrences of it are allowed.
Plus sign (+)	Indicates one or more items. This is a required element of which multiple occurrences are allowed.

Next Steps: Building and Testing Sample Queries

If you have not already done so, we suggest working through the steps in [Getting Started](#), which takes you through the basic tasks of configuring some data sources and using the Data View Builder to design a query using the Order Query example from our Avitek Sample. (For more information about the Avitek Sample and other samples, see the [Samples](#) introduction page.) Working through the Getting Started (or even reading through the steps related specifically to using the Data View Builder) is an easy, hands-on way to get familiar with working with schema representations of data sources and using the basic query-building tools, task flow, and workspaces in the Data View Builder.

If you have already worked through the Getting Started topic or if you are ready to get started on building some other basic queries, we suggest you skip to the following topics in this document:

- [“Examples of Simple Queries” on page 3-19](#) (at the end of [Chapter 3](#), [“Designing Queries”](#)) provides two example queries that provide practice in some basic techniques such as creating join conditions, using functions, setting scope of the target, using the sort-by feature to specify the order of the result, and running queries.
- [“Query Cookbook” on page 9-1](#) provides several examples of complex queries using more advanced features and functions such as creating unions, using date and time functions, using aggregate functions, using hints to optimize queries, and using data views in queries.

3 Designing Queries

This section explains how to design and build a BEA Liquid Data for WebLogic™ query using the Data View Builder, and provides example walk-throughs of how to build some simple queries. The following topics are included:

- [Designing a Query](#)
- [Building a Query](#)
 - [Opening the Source Schemas for the Data Sources You Want to Query](#)
 - [Adding a Target Schema](#)
 - [Mapping Source and Target Schemas](#)
 - [Setting Conditions](#)
 - [Showing or Hiding Data Types](#)
 - [Using Automatic Type Casting](#)
- [Examples of Simple Queries](#)
 - [Example: Return Customers by Name](#)
 - [Example: Query Customers by ID and Sort by State](#)
- [Understanding Scope in Basic and Advanced Views](#)
 - [Where Does Scope Apply?](#)
 - [Basic View \(Automatic Scope Settings\)](#)
 - [Advanced View \(Setting the Scope Manually\)](#)
 - [When to Use Advanced View to Set Scope Manually](#)
 - [Task Flow Model for Advanced View Manual Scoping](#)
 - [Returning to Basic View](#)

- [Saving Projects from Basic or Advanced View](#)
- [Scope Recursion Errors](#)
- [Understanding Query Design Patterns](#)
 - [Target Schema Design Guidelines and Query Examples](#)
 - [Source Replication](#)
- [Next Steps](#)

Designing a Query

The first step in constructing a query (or, more often, a set of queries) is a *design* step—drawing on the requirements identified to answer the following questions critical to the query design:

- What types of data sources do I need to query?
 - What is the structure of each data source; that is, what do the XML source schemas look like?
 - What do I want the query result (that is, the *output* of the query) to look like? In other words, how do I want to structure the output?
 - What should the target XML schema look like? (The target schema defines the structure of the query result.)
 - What target schema *design pattern* should I use?
- Note:** Proper design of the target schema is a key factor in building a successful query. In a nutshell, you need to ensure that cardinality is correct and check for target conformity. For complete guidelines and examples of recommended design patterns, see [“Target Schema Design Guidelines and Query Examples” on page 3-39](#).
- What source conditions do I need to define to get the information I need from the data sources? (Source conditions are joins, unions, aggregations and so on defined to filter the source data in a certain way.)

Once you have designed or “modeled” the query in this way based on what you want the query to do and defined an outline strategy for accomplishing the information filtering, you are ready to build a test version of the query. For other than very simple queries, you will probably revise, refine and test the query several times adding optimization if necessary.

Building a Query

Building a query involves specifying one or more source schemas that describe resource data, selecting a single target schema that describes the shape of the query result, creating source-to-target mappings to further define what the query result will look like, and defining source conditions or *filters* on the data sources. The query extracts the results based on the conditions and mappings that you define in the query. The results can change dramatically depending on how you do the following:

- Specify conditions (filters on the source data)
- Map or *project* source data from one or more sources to the target schema.

If you have taken the time to outline a design for the query first, considering all the factors mentioned in the previous section (“[Designing Queries](#)” on page 3-1), constructing it will be a matter of following your design as a blueprint for drag-and-drop query building. Then you can test, fine-tune, and modify as needed to produce variations on the results, or to optimize the query for better performance.

The following sections take you through the basic tasks involved in building a query:

- [Opening the Source Schemas for the Data Sources You Want to Query](#)
- [Adding a Target Schema](#)
- [Mapping Source and Target Schemas](#)
- [Setting Conditions](#)

Opening the Source Schemas for the Data Sources You Want to Query

A source schema is the XML schema representation for the structure of the data in a data source. You can use multiple data source schemas per query.

The Sources tab on the Builder Toolbar contains the data sources configured on the Liquid Data Server to which you are connected. Note that a data source type only shows up as a button on the Builder Toolbar if it has been configured in the Server to which you are connecting.

Note: Only data sources that have been configured for access by Liquid Data are available from the Builder Toolbar. For information on how to configure Liquid Data data sources, see the Liquid Data [Administration Guide](#).

For this example, open the schemas for the following two data sources which are already configured on the Liquid Data Samples server:

- PB-WL relational database
- XM-BB-C XML file

To do this, follow these steps:

1. Click the Design tab.
2. On the Builder Toolbar, click the Sources tab (on the bottom of the left vertical panel).
3. Open the data sources from the Builder toolbar as follows:
 - Click the Relational Databases and double-click on PB-WL data source to open the associated XML schema showing “Wireless” customers.
 - Click the XML Files button in the navigation panel and double-click on XM-BB-C data source to open the associated XML schema showing “BroadBand” customers.

The XML schemas for the each of the data sources are displayed.

Position the schema windows so you can view the data nodes in each schema. You can expand the data nodes by clicking the plus (+) sign. For example, in the PB-WL data source, CUSTOMER is a *parent node* with subordinate *child nodes*.

The child nodes are the ones you will use as function parameters and map to the target schema.

Adding a Target Schema

A target schema is the XML schema representation for the structure of the *target* data (query result). *Only one target schema per project is allowed.* If you have a target schema open and decide to choose another, the current target schema is closed and the new one replaces it.

You can use a target schema file that you have saved on your local system or on the network, or one that has been saved to the Liquid Data server repository.

Note: Only target schemas that are saved to the Liquid Data server repository will be available to other Liquid Data users for distributed, team-style development.

For this example, we will use a target schema called `amtByState`. If this schema is not available in the Samples server repository and you would like to follow along with our example, you can create it yourself and save it locally or to the server repository as a `.xsd` file.

To create and set the target schema do the following:

1. Use a text editor to copy the following XML into a plain text file and save it to the server Repository as `amtByState.xsd`.

The path to the schemas folder in the Liquid Data server repository is:

```
<WL_HOME>liquiddata/samples/config/ld_samples/repository/schemas/
```

Note: It is not necessary to save the target schema to the server Repository in order to use it in your local project—you can save it anywhere on your system. However, we recommend saving schemas to the Repository because it makes projects more “portable” and schema files accessible to all users who log onto this server.

Listing 3-1 XML Source for `amtByState.xsd` Target Schema File

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
  <xsd:element name="customers">
```

3 Designing Queries

```
<xsd:complexType>
  <xsd:sequence>
    <xsd:element name="STATE" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="state" type="xsd:string" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="CUSTOMER" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="FIRST_NAME" type="xsd:string"/>
          <xsd:element name="LAST_NAME" type="xsd:string"/>
          <xsd:element name="AVERAGE_ORDER" type="xsd:string"/>
          <xsd:element name="CUSTOMER_ID" type="xsd:string"/>
          <xsd:element name="STATE" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

2. In the Data View Builder, choose File—>New Project to clear your desktop and reset all default values.
3. Choose the menu option File—>Set Target Schema.

Navigate to the server Repository or to the location where you saved the `amtByState.xsd` schema. Choose `amtByState.xsd` and click Open.

`amtByState.xsd` appears as the target schema.

This target schema is displayed as a docked schema window on the right side of the workspace.

Note: Remember that only one target schema per project is allowed. The target schema docks on the right side of the desktop area. The target schema may have more data nodes than you need for your result, but it must contain the data nodes required for the query result. Unreferenced nodes are disregarded in the result.

Editing a Target Schema

You can make simple changes to a target schema by right-clicking a node. A shortcut menu shows the editing functions that are available.

Function	Rules
Copy	<p>You can copy both a source and a target node if:</p> <ul style="list-style-type: none"> ■ The node or its parent is not a clone. ■ The node or its parent has not been cloned.
Paste	<p>Appends the copied node and its children as a child of the selected node. If a copied node contains cloned nodes, Data View Builder pastes them as regular nodes. Only the hierarchical structure transfers.</p> <ul style="list-style-type: none"> ■ If a pasted node is a duplicate, Data View Builder renames the node as <code>_copy1</code>, <code>_copy2</code>, and so on. ■ Pasted nodes lose any mapping attributes; however, Data View Builder will display a warning and allow you to abandon the task. ■ The paste function works only on an element node. You cannot paste a child node to an attribute node. ■ This menu item is unavailable unless you have data on the clipboard.
Add a Child (Node or Attribute)	<p>Appends a new node or attribute as a child to the selected node. The name of the new element or node is <code>new_node</code> or <code>new_attribute</code>.</p> <p>The add function works only on an element node. You cannot add a child node to an attribute node.</p> <p>Note: Only string type data is supported.</p>
Delete	<p>Removes a selected node. If the node to be deleted is a mapped node, Data View Builder will display a warning and allow you to abandon the task.</p>
Rename	<p>Allows you to rename the selected node. An error message appears if the new name is a duplicate of a node at the same hierarchical level. As detailed in the XML specification, target schema element names must consist of ASCII characters and must not include double-byte characters or single-byte Katakana characters.</p>

Mapping Source and Target Schemas

Mapping is a visual relationship structure among the critical data elements in the query. When you combine these relationships with functions, you have a set of instructions for the query generation engine. If you think of the selected data nodes as the nouns (what we want to work on), the functions as the verbs (the action), then the mapping among the data elements creates a logical sentence that expresses the query.

Before you begin creating a query, it is important to expand the nodes in the source schemas to reveal the data elements that you want to use at their lowest level. To expand a node, right click on it and choose Expand.

Note: You can use automatic type casting to ensure that input parameters used in functions and mappings are appropriate to the function in which they are used. When Automatic Type Casting is in effect, Liquid Data verifies (and if necessary promotes) the data types of input parameters for all source-to-target mappings and functions. For more information about automatic type casting, see [“Using Automatic Type Casting” on page 3-17](#).

Mapping Node to Node

You can use drag-and-drop mappings from one element or attribute to another to create conditions on source data and source-to-target mappings that will define the shape of the query result.

Mapping Nodes to Create Conditions on Source Data

Choose one of the following methods to map a source schema node to another source schema node to create a Condition.

- Drag and drop a source schema element/attribute to another source schema element/attribute to define an `eq` (equality) source condition

Or

- For all functions other than `eq` (equality), drag and drop the function to the first empty row in the Condition column in the Work area first before you drag and drop elements/attributes as function parameters. Then drag a source schema element/attribute and drop it into the same row of the Condition column. Drag a second source schema element/attribute and drop it into the same row of the Condition column.

Mapping Nodes to Create Source-to-Target Mappings

Choose one of the following methods to map a source schema node to a target schema node.

- Drag and drop a source schema element/attribute to a target schema element/attribute.
- On the Mappings tab, drag-and-drop a source schema element/attribute into a row in the Sources column and drag-and-drop a target schema element/attribute into the same row in the Target column. For all functions other than eq (equality), drag-and-drop the function first before you drag and drop elements or attributes as function parameters.

Note: You cannot map the same node from more than one source schema to a single node in the target schema. For example, if you map STATE (under CUSTOMER) from the Broadband database to state? in the target schema, you cannot successfully map STATE from a second source schema to state? in the target schema. The last mapping completed is the only mapping from source to target that the query generation engine processes. If you need to create a relationship among all three STATE elements, map the element in one source schema to the element in the second source schema. Then map one of the source elements to the target element.

Example: Query Customers by State

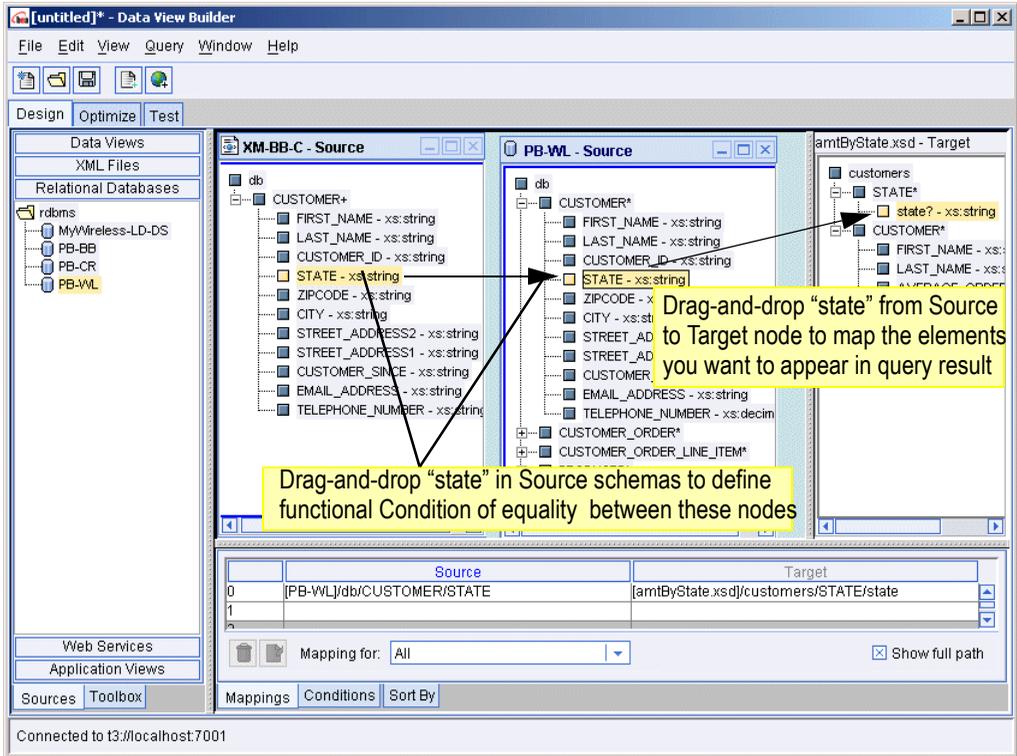
[Figure 3-1](#) shows a simple join of the source element STATE in the Broadband source schema (XM-BB-C) with a source element STATE in the Wireless source schema (PB-WL). This action joins the common elements in each schema and disregards those that do not occur in both schemas.

Next, to project a result we designate what the output of this relationship should look when the query runs. By mapping one of the sources to the target, we specify that we want to store the result in the target schema. Because we are collecting only information about states and defining only one element in the target schema, we are in effect asking that Liquid Data fill only that data element in the result when the query runs.

To do this, drag and drop the STATE element in PB-WL source schema onto the state? element (under STATE*) in the Target schema.

See [Figure 3-1](#) for an example of the mappings described in this example.

Figure 3-1 Mapping Element to Element



Mapping Nodes to Functions

When you drag and drop a source node onto another source node (either within the same source schema or among different source schemas) you are automatically creating an equality relationship between the two elements/attributes using the `eq` (equals) function. In other words, the `eq` function is mapped by default for all drag-and-drop relationships you create among source elements/attributes.

You can also create the same equality relationship the “long” way by dragging and dropping the `eq` function onto a row in the Conditions tab and then dragging and dropping source elements/attributes into the same row, or by opening the Functions Editor and dragging and dropping the function and elements directly into the Editor.

To use any of the available functions other than `eq` (equality) function, you must use this second method of dealing directly with the functions as described below.

To use a function:

1. Drag and drop the function from the Toolbox “Functions” panel to the first empty row under the “Conditions” on the Conditions tab.
2. Drag a source schema node and drop it into the same row of the Condition column. Drag a second source schema node and drop it into the same row of the Condition column.

To edit an existing functional relationship:

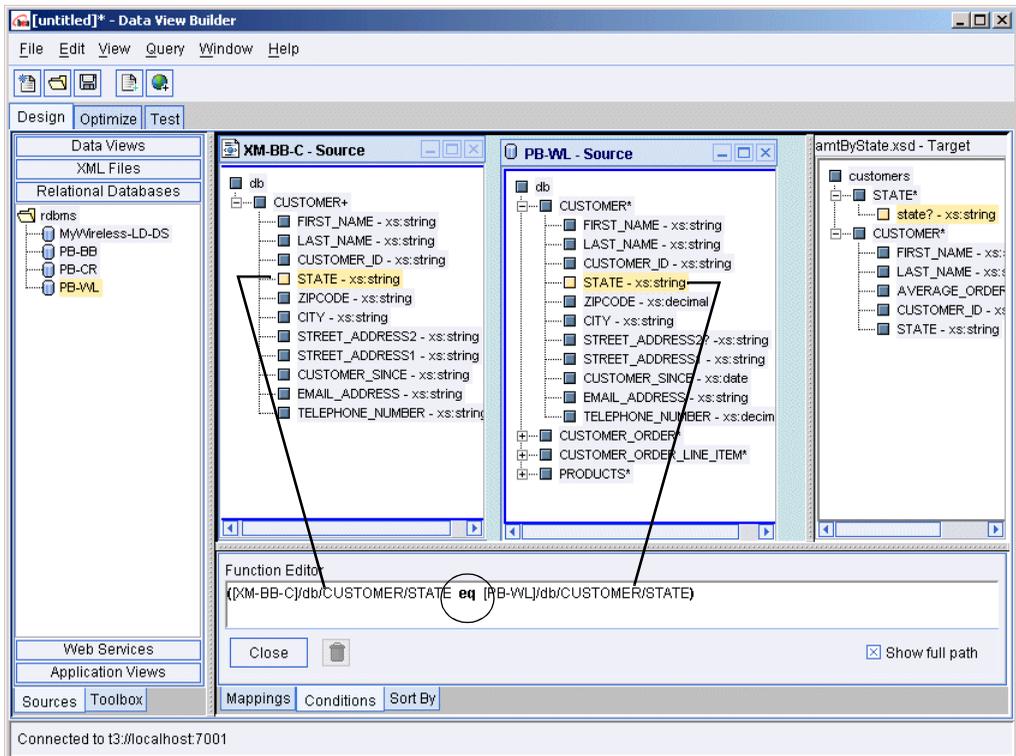
1. Open the Functions Editor by clicking the Edit button.



2. Edit the statement as needed. You can delete the current parameters or function, and drag and drop a new function and source elements/attributes into the Functions Editor.

[Figure 3-2](#) shows the functional relationship of equality ($=$) between two source elements that was created by default when you mapped the source elements in [“Example: Query Customers by State” on page 3-9](#). (Note that you could have created this same relationship directly in the Functions Editor the way you would create any other functional relationship between elements/attributes.)

Figure 3-2 Mapping Elements to Functions



To get the view shown in [Figure 3-2](#), click on the Conditions tab, select the row with the condition in it to activate the Edit button, and click the Edit button. This displays the condition in the Functions Editor.

For more information about functions, see [“What are Functions?”](#) on page 3-14.

For more information about using the Functions Editor and working with functions on the UI, see [“XQuery Functions”](#) on page 2-11 and [“The Function Editor”](#) on page 2-12 in [Chapter 2, “Starting the Builder and Touring the GUI.”](#)

Supported Mapping Relationships

Data View Builder and Liquid Data support any of the Mapping actions described in the following table.

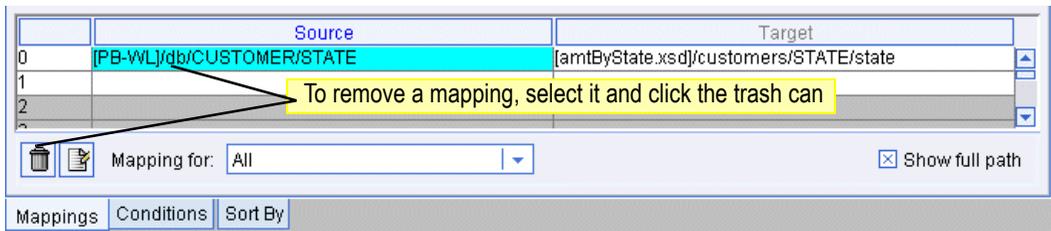
Table 3-1 Supported Mapping Relationships

Types of Mappings	Description
Source node to another source node	Creates an equality relationship between the two elements/attributes using the eq (equals) function. The eq function is used by default for all drag-and-drop mappings created among source elements/attributes to create a condition that will filter for matching items found.
Source node to a function	The data becomes an input parameter to a function. (You can also provide constants and variables as function parameters.) Each function has its own specification of parameters. The output from a function can be input to another function. For an example of this, see “Example 2: Aggregates” on page 9-8 in Chapter 9, “Query Cookbook” (specifically, the step “Ex 2: Step 8. Add the “count” Function” on page 9-15 within the Aggregates example).
Source node to a target node	By mapping a source to a target, you are <i>projecting</i> , or storing, the data onto the target schema. All query examples provided in this documentation show how to map source schema elements/attributes to target elements/attributes. For example, see “Example: Return Customers by Name” on page 3-19 and “Example: Query Customers by ID and Sort by State” on page 3-25 .
Function to target node	A function ($f1$) output can be another function's ($f2$) input. For an example of this, see “Example 2: Aggregates” on page 9-8 in Chapter 9, “Query Cookbook” (specifically, the step “Ex 2: Step 8. Add the “count” Function” on page 9-15 within the Aggregates example).

Removing Mappings

Mapped elements/attributes in a query are displayed on the Mappings tab. You can change your mind and remove a mapping by selecting the row or cell that contains it and then clicking the trashcan button. (See [Figure 3-3](#).)

Figure 3-3 Removing a Mapping



Setting Conditions

You can create *conditions* or filters on source data by doing any of the following:

- Drag-and-drop a source node onto another source node to build a conditional statement that defines the default *eq* (equality) functional relationship between the mapped elements/attributes. (See “Supported Mapping Relationships” on page 3-12.)
- Drag-and-drop source elements/attributes and functions directly into a row on the Conditions tab to build a conditional statement with any of the functions available from Design tab → Toolbox tab → Functions panel.

What are Functions?

Functions are used as the verbs or actions in condition statements that establish relationships between or operations on data source elements or attributes. (The data source elements/attributes become one type of *parameter* to the functions.) A function is a built-in executable process that manipulates the data to perform a task. You must pass one or more parameters, which can be source data, variables, or constant values, for the function to produce output. The function returns a result to you based on the conditional statements you build and how you specify where to store the result.

In the previous example (“Example: Query Customers by State” on page 3-9) we defined the default equality relationship between two source elements (by dragging and dropping the CUSTOMER “STATE” element from one source to another); then defined the result by dragging and dropping the CUSTOMER “STATE” element from one of the source schemas onto the analogous “STATE” element in the target schema.

If you need to find out something other than information based on equality, you will need to use a different function. For example, suppose you want to find out how many customer IDs in the Broadband database are *not equal to* those in the Wireless database. The default functional action is to look for equality. If you simply map one customer ID source element/attribute to the other, the query engine looks for those instances of matching data, or equality.

(For the relationship of *not equal to*, you need to go to Builder Toolbar—>Sources tab—>Functions panel, expand the `Operators` node, and choose the `ne` function.)

When any functional relationship is involved besides equality, you must choose from the list of functions available in the Builder Toolbar—>Sources tab—>Functions panel. At that point you are applying a filter of your choice. It is very important to *choose the function before you map the elements*. Most of the Data View Builder functions are standard XML query language functions supported by the W3C. For related information about using functions, see [Appendix A, “Functions Reference.”](#)

Note: You can use automatic type casting to ensure that input parameters used in functions and mappings are appropriate to the function in which they are used. When Automatic Type Casting is in effect, Liquid Data verifies (and if necessary promotes) the data types of input parameters for all source-to-target mappings and functions. For more information about automatic type casting, see [“Using Automatic Type Casting” on page 3-17.](#)

Using Constants and Variables in Functions

Instead of choosing an existing element/attribute as a parameter value, you can use one of these methods to specify that a constant value should be used instead of a data element from a source schema.

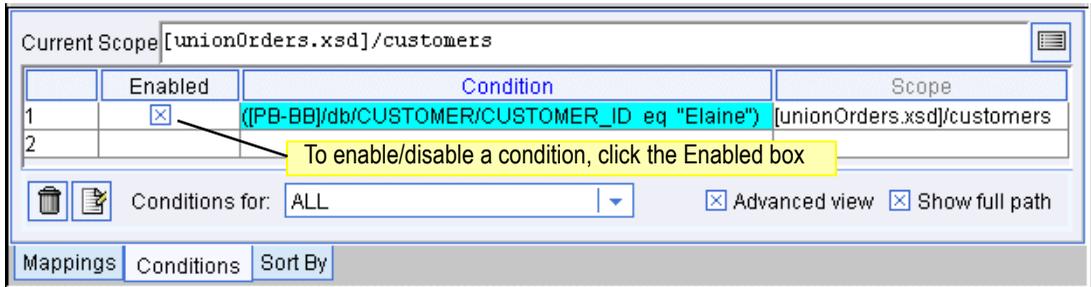
- Click the **Query Parameter** Navigation panel. If you wish to add a new variable, type the variable name in the available text box. Click **Add**. The new variable appears in the Query Parameter tree. Double click the new variable to use it as a parameter value. You can change the value each time you run the query. (For details on defining query parameters, see [“Query Parameters: Defining” on page 2-15](#), which includes a list of supported data types for query parameters in [Table 2-2, “Query Parameter Types,” on page 2-17.](#))
- Click the **Constant** Navigation panel. If the constant already exists in the Constant tree, double click the constant you want to use as a parameter in a function or drag and drop the constant to the appropriate row in the Condition column.(For details on defining constants, see [“Constants” on page 2-13.](#))

Enabling and Disabling Conditions

Conditions are displayed in the Design view on the Conditions tab. If you want to test your query without one or more of the conditions you have set, but still keep the condition configured for possible later use, you can disable a condition. Conversely, you can enable a disabled condition.

To enable or disable a condition, click the Enabled box to the left of the Condition. (See [Figure 3-4](#).) When the box is checked, the condition is used when the query is generated; when the box is blank, the condition is not used in the generated query.

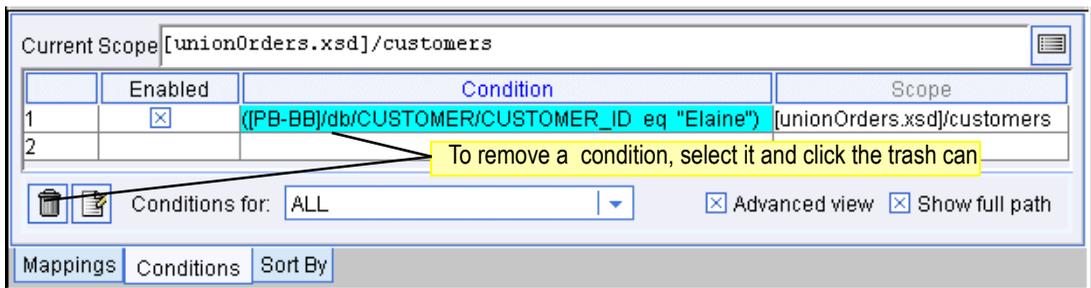
Figure 3-4 Enabling or Disabling a Condition



Removing Conditions

Conditions are displayed in the Design view on the Conditions tab. You can change your mind and remove a condition by selecting the row or cell that contains it and then clicking the trashcan button. (See [Figure 3-5](#).)

Figure 3-5 Removing a Condition



Adding or Deleting Parameters in a Condition Statement

To add or delete a parameter, select the row that contains the condition you want to edit and click the Edit button to bring up the Functions Editor.



In the Functions Editor, you can select the parameter you want to delete and click the trash can or use the options on the Edit menu to modify the condition statement.

You can drag and drop different functions into the Functions Editor from the Functions panel on the Builder Toolbar—>Toolbox tab.

Showing or Hiding Data Types

You can show or hide data types on all source and target elements/attributes in schema windows. Select View—>Data Types to display the data type of any source or target element/attribute, as well as required function parameter types. (An “X” next to the Data Types option on the View menu indicates that it is *on*.)

Using Automatic Type Casting

You can use automatic type casting to ensure that input parameters used in functions and mappings are appropriate to the function in which they are used.

Note: For a complete reference showing how Liquid Data transforms source element/attribute data types to data types of target elements/attributes, see [Appendix C, “Type Casting Reference.”](#)

Select Automatic Type Casting on the Query menu to ensure that Liquid Data will assign (cast) a new data type when the source node data type does not match the mapped target node data type, *and* the source node is eligible to be type cast to the target node data type. (An “X” next to the Automatic Type Casting option on the Query menu indicates that it is *on*.)

When function parameters have a numeric type mismatch, the Liquid Data server can promote the input source to the input type required by the function if the promotion adheres to the prescribed promotion hierarchy. The promotion hierarchy exists only for numeric values.

Type	Promoted Type
byte	short
short	int
int	long
long	integer
integer	decimal
decimal	float
float	double

If the type mismatch requires casting in the reverse order, the server does not attempt type casting. In this case, Liquid Data attempts to type cast but the results may be unpredictable. For example, if the required function input type is *xs:decimal*, then source data that is integer, long, int, short, or byte can easily be promoted to a data type with more precision or larger number of digits. The server will complete that task. If the input function type is *xs:double* or *xs:float* and the required input type is *xs:integer* or *xs:byte*, Liquid Data tries to type cast successfully, but there may be unpredictable rounding or truncating. All other type mismatches, such as *xs:date*, *xs:dateTime*, or *xs:string*, require a type cast to avoid a type mismatch error.

Clear the Automatic Type Casting check box to disable this feature.

Exceptions to Automatic Type Casting

Liquid Data does not type cast comparison operators (such as eq, le, ge, ne, gt, lt, or ne) or any functions that accept *xsect:anytype*.

Type casting does not apply to function parameters (as well as target schema elements/attributes) that require these data types:

- `xsect:item`

- `xsext:anyValue`
- `xsext:anyType`
- Any other data type that cannot be cast

Automatic type casting does not succeed in all cases. If the source data is not compatible with the data type of the target node, automatic type casting will not improve the query results. For example, mapping a date to a numeric type may not produce useful results if the data is not relevant. You may not see an error on a type mismatch until the Liquid Data Server tries to run the query.

Examples of Simple Queries

This section includes walk-through examples of how to build some simple queries using the Data View Builder tools and features just described:

- [Example: Return Customers by Name](#)
- [Example: Query Customers by ID and Sort by State](#)

To work through these examples, begin on the Data View Builder “Design” tab. If you have worked through the previous example using `amtByState` target schema, we suggest you close that project and open/save a new project for each of the examples described below.

Example: Return Customers by Name

In this example, you want to return the last and first names of all Wireless customers with a last name that begins with “K.”

Build the Query

The approach we will use is similar to the first example in this chapter; however, you are adding a condition that the last name begins with “K.” Build the condition with the `starts-with` function as follows.

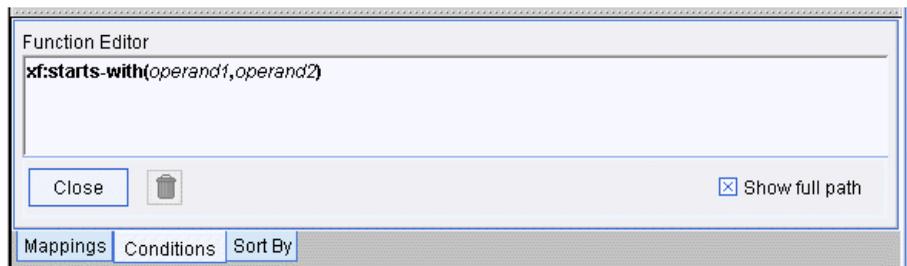
3 Designing Queries

1. Choose File—>New Project to clear your desktop and reset all default values.
2. On the Builder Toolbar—>Sources tab, click Relational Databases. Double-click on the PB-WL (Wireless) relational database to add it to the project.
3. Create `amtByState.xsd` target schema and add it to the server repository. (For a copy of the schema file and instructions on how to save it to the repository, see [“Adding a Target Schema” on page 3-5](#) and the schema shown in [Listing 3-1.](#))
4. Choose File—>Set Target Schema. Use the file browser to navigate to the Repository and select `amtByState.xsd` as the target schema.

This target schema is displayed as a docked schema window on the right side of the workspace. To expand all nodes in the target schema, select the top level node, right mouse click and choose Expand from the popup menu.

5. Map the source schema `CUSTOMER LAST_NAME` to the corresponding `LAST_NAME` element in the target schema.
6. On the Builder Toolbar—>Toolbox tab, click Functions. Under String functions, find the `starts-with` function. Drag and drop `starts-with` onto the first row in the Conditions Tab.

When you do this, the Functions Editor will automatically pop up and show you the condition statement with the `starts-with` function and variable placeholders.

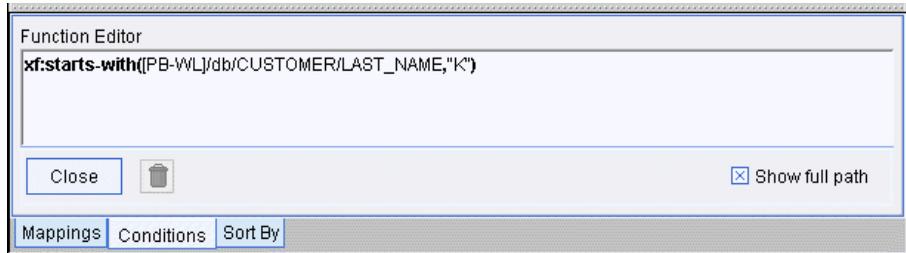


7. Drag and drop `CUSTOMER “LAST_NAME”` element from the Source schema onto the first parameter (`operand1`).

Note: This example shows what the function looks like with menu option View—>Data Types turned off. If you have this option *on* (it is on by default), data types for each parameter will also show.

- On the Builder Toolbar—>Toolbox tab, click Constants. Type “K” in the text box, then drag and drop the Constants icon to the right of the text field onto the second parameter (*operand2*). (For details on using the Constants panel, see “Constants” on page 2-13 in Chapter 2, “Starting the Builder and Touring the GUI.”)

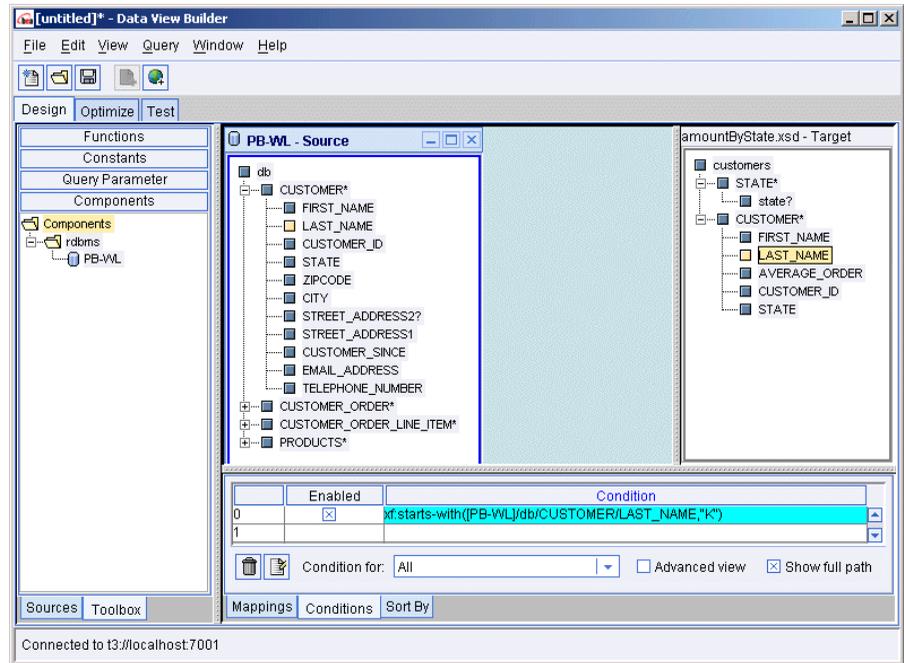
The condition statement should look similar to that shown in following figure.



- Close the Function Editor by clicking Close. (The condition statement is displayed on the first row of the Conditions tab in the Source column.)

Figure 3-6 shows the Design view of the query with conditions and source-to-target mappings completed.

Figure 3-6 Design View of Query Example: Return Customers By Name



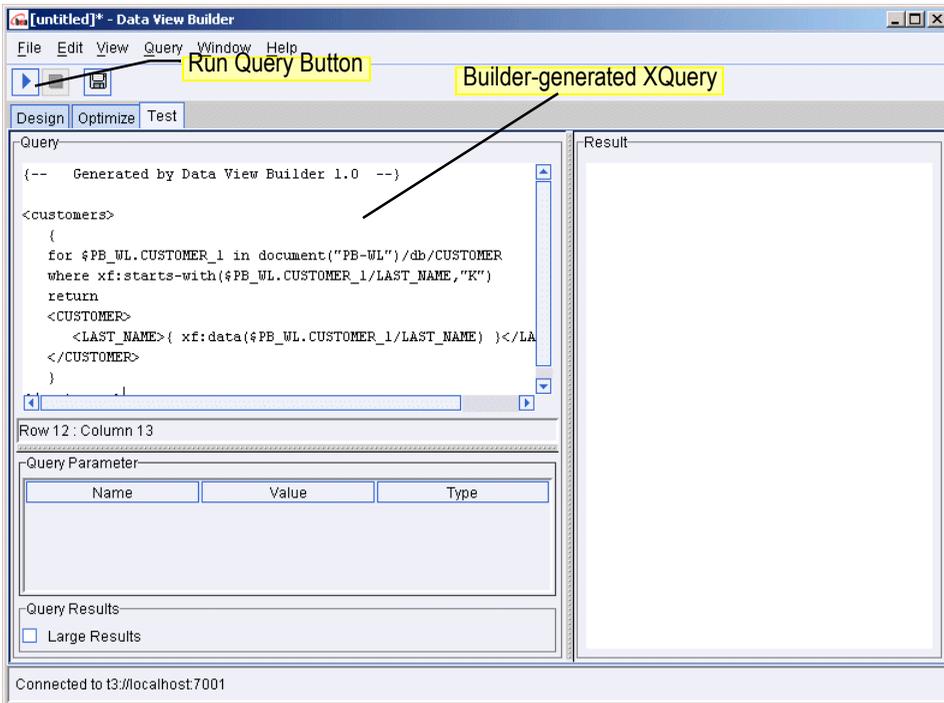
View the XQuery and Run the Query to Test it

Now that you have built the query, you can switch to the Test tab to view the generated XQuery and run the query to see the kind of result it returns.

1. Click on the Test tab.

The generated XQuery is displayed in the Query panel on the left side of the Test tab as shown in [Figure 3-7](#). The full XQuery is also provided in [Listing 3-2](#).

Figure 3-7 XQuery for Example: Return Customers By Name



Listing 3-2 XQuery for Example: Return Customers By Name

```

{-- Generated by Data View Builder 1.0 --}

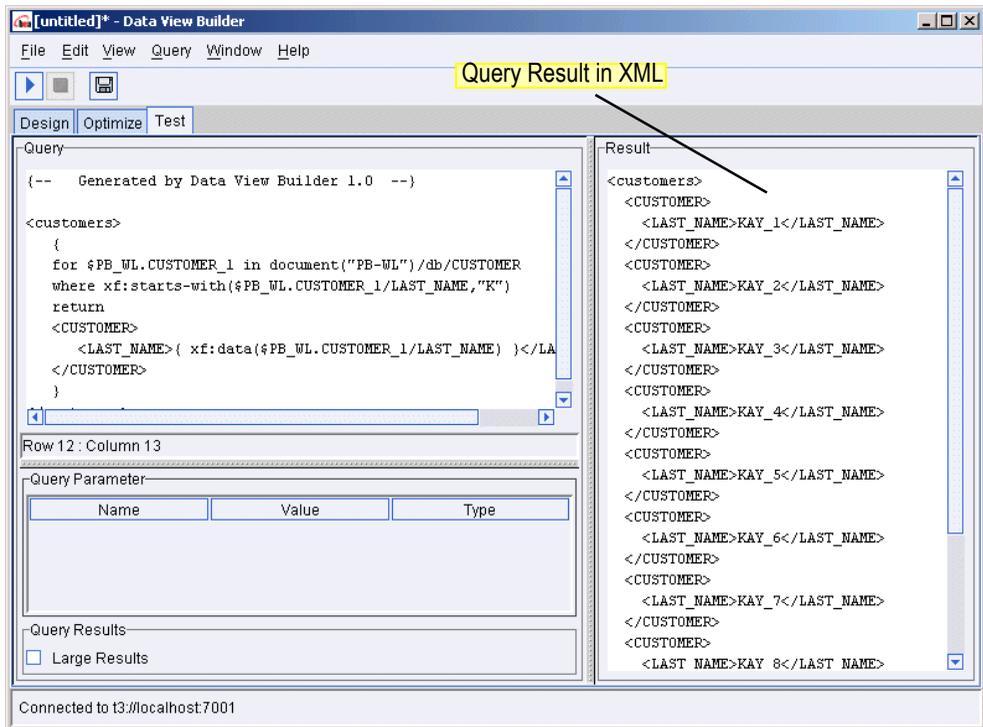
<customers>
{
  for $PB_WL.CUSTOMER_1 in document("PB-WL")/db/CUSTOMER
  where xf:starts-with($PB_WL.CUSTOMER_1/LAST_NAME,"K")
  return
  <CUSTOMER>
    <LAST_NAME>{ xf:data($PB_WL.CUSTOMER_1/LAST_NAME) }</LAST_NAME>
  </CUSTOMER>
}
</customers>

```

2. Click the “Run query” button to run the query against the data sources.

The query result is shown in the Result panel on the right side of the Test tab as shown in [Figure 3-8](#). The full XML query result is provided in [Listing 3-3](#).

Figure 3-8 Query Result for Example: Return Customers By Name



Listing 3-3 XML Query Result for Example: Return Customers By Name

```
<customers>
  <CUSTOMER>
    <LAST_NAME>KAY_1</LAST_NAME>
  </CUSTOMER>
  <CUSTOMER>
    <LAST_NAME>KAY_2</LAST_NAME>
  </CUSTOMER>
  <CUSTOMER>
    <LAST_NAME>KAY_3</LAST_NAME>
```

```
</CUSTOMER>
<CUSTOMER>
  <LAST_NAME>KAY_4</LAST_NAME>
</CUSTOMER>
<CUSTOMER>
  <LAST_NAME>KAY_5</LAST_NAME>
</CUSTOMER>
<CUSTOMER>
  <LAST_NAME>KAY_6</LAST_NAME>
</CUSTOMER>
<CUSTOMER>
  <LAST_NAME>KAY_7</LAST_NAME>
</CUSTOMER>
<CUSTOMER>
  <LAST_NAME>KAY_8</LAST_NAME>
</CUSTOMER>
<CUSTOMER>
  <LAST_NAME>KAY_9</LAST_NAME>
</CUSTOMER>
<CUSTOMER>
  <LAST_NAME>KAY_10</LAST_NAME>
</CUSTOMER>
</customers>
```

(For complete details on how to test and run a query, see [Chapter 5, “Testing Queries.”](#))

Example: Query Customers by ID and Sort by State

In this example, there are two pieces of information that we want to display in the result. We want to find Customer IDs for customers who exist in both databases and we want to know the state each found customer resides in.

This example shows how to do the following:

- Project output
- Specify the order of the result

Open the Data Sources and Add a Target Schema

1. Choose File—>New Project to clear your desktop and reset all default values.
2. On the Builder Toolbar—>Sources tab, click Relational Databases and open two data sources:
 - Double-click on the PB-WL relational database to open the schema for this data source.
 - Double-click on the PB-BB relational database to open the schema for this data source.
3. Choose File—>Set Target Schema. Use the file browser to navigate to the Repository and select `amtByState.xsd` as the target schema.

Note: If `amtByState.xsd` is not already saved in the Samples server Repository, you can create it yourself and save it to the Repository. For a copy of the schema file and instructions on how to save it to the Repository, see [“Adding a Target Schema” on page 3-5](#) and the schema shown in [Listing 3-1](#).

This target schema is displayed as a docked schema window on the right side of the workspace.

Map Nodes from Source to Target Schema to Project Output

To project Customer first and last names and state to Target, do the following:

1. Drag and drop Wireless (PB-WL) `FIRST_NAME` (under `CUSTOMER*`) onto `FIRST_NAME` in the Target schema.
2. Drag and drop Wireless (PB-WL) `LAST_NAME` (under `CUSTOMER*`) onto `LAST_NAME` in the Target schema.
3. Drag and drop Wireless (PB-WL) `STATE` (under `CUSTOMER*`) onto `STATE` (under `CUSTOMER*`) in the Target schema.

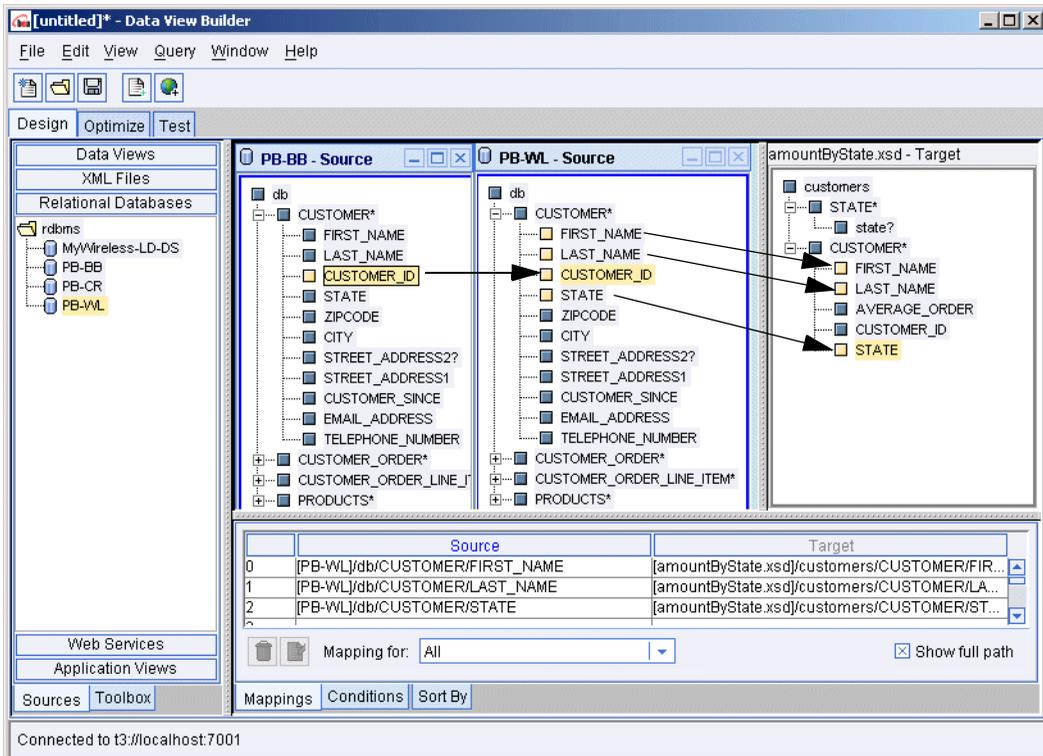
Join Two Sources

To create a *join* between Wireless (PB-WL) and Broadband (PB-BB) on customer IDs, do the following:

- Drag and drop Broadband (PB-BB) CUSTOMER_ID (under CUSTOMER*) onto the associated Wireless (PB-WL) CUSTOMER_ID element.

The following shows the mappings in the Data View Builder.

Figure 3-9 Example: Query Customers by ID and Sort Output by State



Specify the Order of the Result Using the Sort By Features

To order the output alphabetically by State do the following:

1. Click the Sort By tab.

This tab shows repeatable nodes in the target schema with subordinate fields that you can select for ordering.

2. From the drop-down menu choose CUSTOMER*, and then click into the Direction cell next to STATE and set STATE to Ascending.

This will cause the query to display the results in ascending order by state.

View the XQuery and Run the Query to Test it

Now that you have built the query, you can switch to the Test tab to view the generated XQuery and run the query to see the kind of result it returns.

1. Click on the Test tab.

The generated XQuery for this query is shown in the following code listing.

Listing 3-4 XQuery for Example: Query Customers by ID and Sort by State

```
{--      Generated by Data View Builder 1.0--}

<customers>
  {
    for $PB_WL.CUSTOMER_1 in document("PB-WL")/db/CUSTOMER
    let $CUSTOMER_2 :=
      for $PB_BB.CUSTOMER_3 in document("PB-BB")/db/CUSTOMER
      where ($PB_BB.CUSTOMER_3/CUSTOMER_ID eq
$PB_WL.CUSTOMER_1/CUSTOMER_ID)
      return
      xf:true()
    where xf:not(xf:empty($CUSTOMER_2))
    return
    <CUSTOMER>
      <FIRST_NAME>{ xf:data($PB_WL.CUSTOMER_1/FIRST_NAME) }</FIRST_NAME>
      <LAST_NAME>{ xf:data($PB_WL.CUSTOMER_1/LAST_NAME) }</LAST_NAME>
      <STATE>{ xf:data($PB_WL.CUSTOMER_1/STATE) }</STATE>
    </CUSTOMER>
    sortby(STATE ascending)
  }
</customers>
```

2. Click the “Run query” button to run the query against the data sources.

Querying these data sources as described in this example produces the XML query result shown in the following code listing.

Listing 3-5 XML Result for Example: Query Customers by ID and Sort by State

```
<customers>
  <CUSTOMER>
    <FIRST_NAME>JOHN_3</FIRST_NAME>
    <LAST_NAME>KAY_3</LAST_NAME>
    <STATE>AZ</STATE>
  </CUSTOMER>
  <CUSTOMER>
    <FIRST_NAME>JOHN_8</FIRST_NAME>
    <LAST_NAME>KAY_8</LAST_NAME>
    <STATE>AZ</STATE>
  </CUSTOMER>
  <CUSTOMER>
    <FIRST_NAME>JOHN_10</FIRST_NAME>
    <LAST_NAME>KAY_10</LAST_NAME>
    <STATE>CA</STATE>
  </CUSTOMER>
  <CUSTOMER>
    <FIRST_NAME>JOHN_5</FIRST_NAME>
    <LAST_NAME>KAY_5</LAST_NAME>
    <STATE>CA</STATE>
  </CUSTOMER>
  <CUSTOMER>
    <FIRST_NAME>JOHN_4</FIRST_NAME>
    <LAST_NAME>KAY_4</LAST_NAME>
    <STATE>NV</STATE>
  </CUSTOMER>
  <CUSTOMER>
    <FIRST_NAME>JOHN_9</FIRST_NAME>
    <LAST_NAME>KAY_9</LAST_NAME>
    <STATE>NV</STATE>
  </CUSTOMER>
  <CUSTOMER>
    <FIRST_NAME>JOHN_1</FIRST_NAME>
    <LAST_NAME>KAY_1</LAST_NAME>
    <STATE>TX</STATE>
  </CUSTOMER>
  <CUSTOMER>
    <FIRST_NAME>JOHN_6</FIRST_NAME>
    <LAST_NAME>KAY_6</LAST_NAME>
    <STATE>TX</STATE>
  </CUSTOMER>
  <CUSTOMER>
    <FIRST_NAME>JOHN_2</FIRST_NAME>
    <LAST_NAME>KAY_2</LAST_NAME>
    <STATE>WA</STATE>
  </CUSTOMER>
  <CUSTOMER>
    <FIRST_NAME>JOHN_7</FIRST_NAME>
    <LAST_NAME>KAY_7</LAST_NAME>
    <STATE>WA</STATE>
  </CUSTOMER>
</customers>
```

Understanding Scope in Basic and Advanced Views

Adding Scope to a condition is a way to specify the extent that the condition applies to the result. It helps you specify which part of a data view is the focal point for a particular condition in the query. A scope setting affects the placement of a “*where*” clause in the XQuery generation.

When you add a condition, the Data View Builder makes a best guess as to where the condition should appear in the query. The Data View Builder draws information from the structure of the target schema, the mappings from source schemas to the target schema, and the conditions.

In most cases, scope is implicit and the query generator can determine what the desired result should be. In other cases, it makes a very conservative assumption about the resulting scope of the condition. You can communicate your objectives more efficiently if you specify exactly what you want the query to return. By toggling to the Advanced view in the Data View Builder (“[Advanced View for Defining Explicit Scope for Conditions](#)” on page 2-27) and setting scope you explicitly indicate what part of the output, or query result, is affected by the condition.

The following sections are included here:

- [Where Does Scope Apply?](#)
- [Basic View \(Automatic Scope Settings\)](#)
- [Advanced View \(Setting the Scope Manually\)](#)
- [When to Use Advanced View to Set Scope Manually](#)
- [Task Flow Model for Advanced View Manual Scoping](#)
- [Returning to Basic View](#)

Where Does Scope Apply?

There are three candidate areas where Liquid Data sets scope. Scope candidates are:

- All repeatable elements in the target schema
- All repeatable input elements in functions
- The root of the target schema

Remember that a repeatable element always appears with an asterisk (*) or plus sign (+) occurrence indicator.

Basic View (Automatic Scope Settings)

The default setting in the Data View Builder is the basic view. In this view, when you add a condition to any query, the Data View Builder applies an automatic scope setting using internal rules that specify where the condition should appear in the query.

In most cases, the scope setting that Data View Builder chooses for each condition is the correct setting. When you have complex conditions, or a particular result in mind, you may want to switch to the Advanced mode where you can change scope settings as you wish.

Advanced View (Setting the Scope Manually)

Data View Builder enables you to override the automatic scope setting by using an Advanced view of the existing scope settings. When you switching to an Advanced view, Data View Builder displays the setting it selected for automatic scoping. You can change any or all of the individual scope settings or allow them to retain their original values.

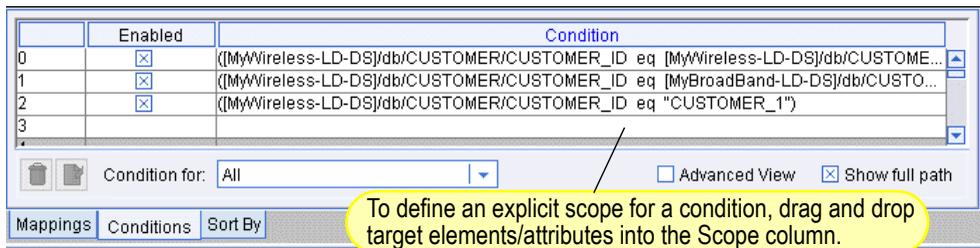
When you switch to the Advanced view, it is not necessary to change any of the explicit scope settings selected by Data View Builder. However, if you add new conditions when you are in Advanced view, or change existing conditions, you must set the new scope manually for each condition.

3 Designing Queries

To switch to the Advanced view, click Advanced view toggle so that an X is displayed next to Advanced view. The Conditions tab expands to show more information about the condition targets.

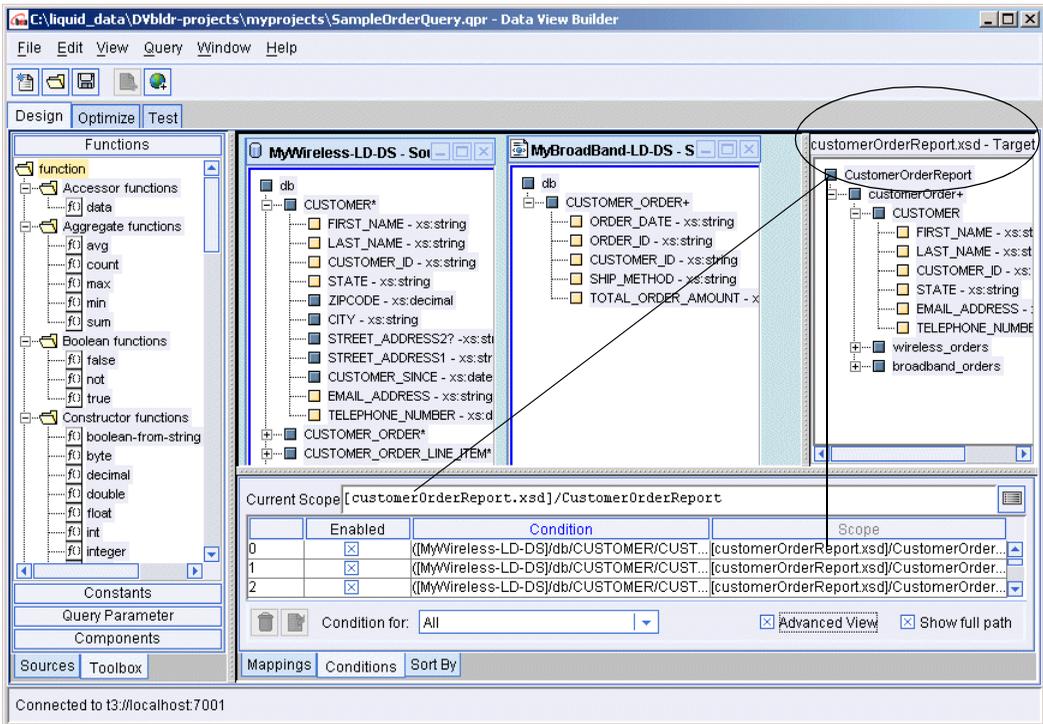
- The Current Scope initially shows the target schema root. Before you map schema sections and create conditions, you can drag a repeatable target schema or function input node to set the scope for a complete section of the target schema. Thereafter, the value in the Current Scope text box determines what will appear automatically in a Scope column cell for any new condition that you create. For more information about this technique, see [Task Flow Model for Advanced View Manual Scoping](#).
- The Enabled column contains a switch to include or exclude a condition when the query runs.
- The Condition column shows the source node, condition, and condition target node.
- The Scope column shows which node Liquid Data automatically selected in the target schema to focus the result. You can also drag a repeatable target schema node directly to a cell in this column to change the scope for that condition.
- The Reset button in the upper right recalculates all scope settings and returns them to the automatic settings selected by Liquid Data.

Figure 3-10 Advanced View Showing Explicit Scope on Conditions Tab



Condition and Target pairs appear row by row. If there are multiple scope settings for a condition, the condition reappears in separate rows to display each unique scope setting.

Figure 3-11 Advanced View



The Current Scope text box shows the default scope setting for every condition that you add. Remember that the Basic view settings will continue to appear until you change them. If you add a new condition in Advanced view, the default scope is the target schema root until you change that value.

When to Use Advanced View to Set Scope Manually

Data View Builder automatically scopes conditions wherever it is most logical, possibly in more than one place. However, occasionally it may not put the automatic scope setting where you think it should be. In these cases, you can switch to the Advanced view to overwrite the automatic scope setting.

The most common case occurs when a condition logically applies in two places, but you want it to appear in only one place. You can diagnose this by examining the XQuery translation for *where* clauses, or do a test run of the query to view the result. If you are not satisfied, switch to the Advanced view to determine where the condition appears. Remember that Data View Builder lists the same condition more than once if it has more than one scope setting. Change the scope setting that you do not want by following the directions in [“Task Flow Model for Advanced View Manual Scoping” on page 3-34](#).

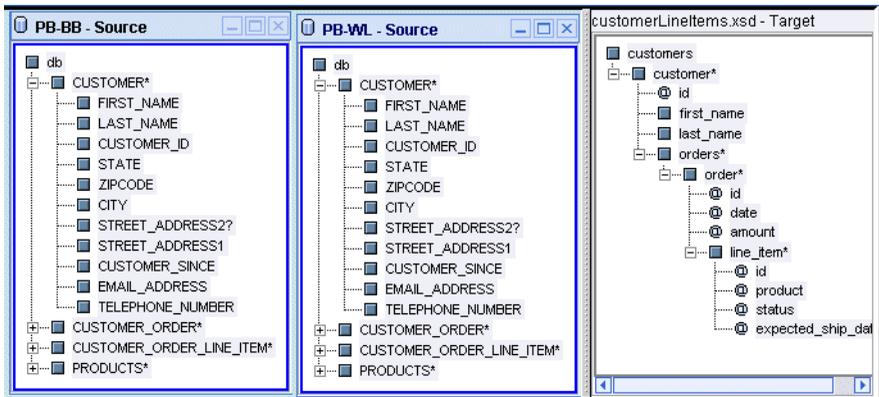
A less common case is when you want to create an assertion. For example, the Liquid Data Server should return a result only when a certain condition occurs. You can accomplish this if you switch to the Advanced view, create the condition, and set the scope for the condition to be the root of the target schema.

Note: It is a good idea to run the query using the automatic scope settings first to ensure that it is necessary to revise the scope setting.

Task Flow Model for Advanced View Manual Scoping

If you decide to override automatic scope settings, there is a workflow model that will help you design the query, create conditions, and determine the scope. By following this methodology, you will find it is easy to create a query where you control the scope. Consider the example shown in [Figure 3-12](#) of two source schemas: PB-BB and PB-WL, and the target schema `customerLineItems.xsd`.

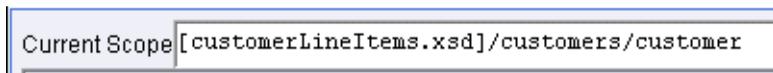
Figure 3-12 Schemas for Manual Scope Example



The target schema, `customerLineItems.xsd`, has a hierarchical structure. There are three distinct sections in the schema that represent repeatable data. `customer` and `order` each have an asterisk (*) as the occurrence indicator. `line_item` has a plus sign (+) as the occurrence indicator. This means that the child nodes without an asterisk or plus are non-repeating. For each customer, there is one occurrence of `first_name`, `last_name`, and `id`. Each customer may have zero or more orders. When an order exists, each order has one `id`, `date`, and `amount`. If an order exists, there must be at least one `line_item`. Work on sections that appear under a repeatable node.

This workflow model assumes that you can build your query in steps, focusing on each section in the target schema as you go. Follow these steps for each section in the target schema where you want a result to appear.

1. Choose a repeatable section of the target schema for our scope. A section is a repeatable node (parent) and its children. It is recommended that you work from the outside in. In this case, the outermost section is the `customer*` section. (For this example we want to collect the `first_name`, `last_name`, and `id` in the result.)
2. Set the highest repeatable node in this section as the default scope, which in this case is `customer*`. Drag that element from the target schema onto the Current Scope text box on the Conditions tab. (For this example we drag and drop `customerLineItems.xsd` onto the Current Scope text box.)



3. Map selected source elements/attributes to that repeatable section in the target schema.

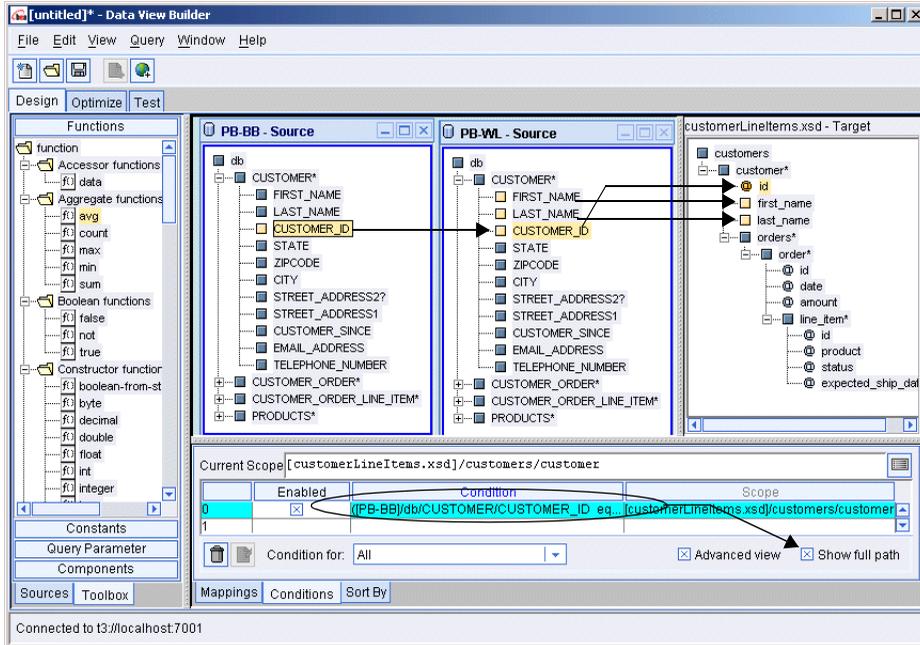
For this example, we do the following mappings:

- Map [PB-WL]/db/CUSTOMER*/FIRST_NAME to [customerLineItems.xsd]/customers/customer*/first_name.
 - Map [PB-WL]/db/CUSTOMER*/LAST_NAME to [customerLineItems.xsd]/customers/customer*/last_name.
 - Map [PB-WL]/db/CUSTOMER*/CUSTOMER_ID to [customerLineItems.xsd]/customers/customer*/id.
4. Set any conditions that connect and filter the mapped sources.

By setting the default scope before creating the condition, Data View Builder sets the condition scope to that value.

By mapping one section at a time and using the repetitive ancestor node as the default scope, your conditions will apply exactly where you need them to appear in the result.

For our example, we set as a condition a join between CUSTOMER_ID in the PB-BB schema and CUSTOMER_ID in the PB-WL schema as shown in the figure below.



- Repeat these steps for each section of the target schema where you want data to appear in the result. Work on one section at a time and work from the outside (more general) to the inside (most specific). Ensure that you set the default target, map, and define the conditions, before you move to the next section. The general rule is that any mapping with an associated condition requires a scope setting.

In a small number of cases, you may apply a condition on the argument (input) to a function that requires choosing the function as the default scope. This is not common but will occur when you choose a complex aggregate function.

Returning to Basic View

When you toggle Advanced view *off* (no X showing next to Advanced view), Data View Builder returns to automatic scoping mode and discards the changes you made in manual mode. The Current Scope text box and the Targets column disappear.

Saving Projects from Basic or Advanced View

If you save a project from Basic view, the project file discards scope information. When you reopen this project, Liquid Data once again applies automatic scope using its internal algorithms.

If you save a project from the Advanced view, all conditions retain current scope settings. When you reopen this project, all Advanced view settings appear.

Version Control

Liquid Data assigns a version attribute to the project file. If you open a project file created with an earlier version of Liquid Data, the project opens in the Advanced view if all conditions have explicit scope settings.

Scope Recursion Errors

It is possible to create a query where a condition depends on the values returned by a function, but the function input depends on the condition. For example:

- Select the `xf:count` function and map a source node to be the input of `xf:count`.
- Create a condition that uses the output of the `xf:count` function.
- In the Advanced view, set the condition target to the input of the `xf:count` function.

The `xf:count` function input must be filtered by applying the condition, but the condition input is the output of `xf:count`.

Data View Builder does not allow this to happen when automatic scoping is enabled. However, if you clear the Auto-Select Targets check box and set scope manually, it is possible for you to set the scope of a condition to a function input that creates a circular dependency. Data View Builder cancels the action and generates an error message:

```
Setting Scope/Target of condition {condition} to {scope node}
creates circular dependency
```

Recommended Action

Basic view should generally support most scenarios—we expect that only a few users and/or queries will require use of the Advanced view manual scoping feature. You can assume that Liquid Data can interpret the scope requirements correctly for most types of queries. If you do choose to set scope manually, examine the generated XQuery to ensure the condition targets meet your expectations. If the recursion error message appears, consider resetting all condition scope targets. Override the automatic settings one at a time, switch to Test view to examine the query, run it, and assess the results.

Understanding Query Design Patterns

Here we present some common query patterns generated by the Data View Builder and provide high-level guidelines for effective query design including target schema design and source replication.

- [Target Schema Design Guidelines and Query Examples](#)
- [Source Replication](#)

Target Schema Design Guidelines and Query Examples

This section provides several examples of queries built with the Data View Builder. We describe the conditions and mappings for a query and the resulting XQuery. The purpose of this is to illustrate how we follow certain guidelines to design the various types of example queries.

- [Design Guidelines](#)
- [Examples of Effective Query Design](#)

For a detailed description of target schemas, see [“Understanding Target Schemas,”](#) on page 1-11.

Design Guidelines

Use these guidelines when working with target schemas in your queries:

1. Make sure the target schema has proper cardinality. For example, if you intend to project customer orders in your result, the target schema should reflect the parent-child relationship between “customer” and “orders.”

All examples in [“Examples of Effective Query Design” on page 3-41](#) demonstrate this guideline.

2. Understand how target schema conformity works and use it efficiently. In an XML schema:

- A plus sign (+) next to a node indicates that the node is repeatable and *required*. (In other words, there must be 1 or more occurrences of this.)

Since this setting requires extra checking of the data, most queries that use it pay some performance penalty.

- An asterisk (*) next to a node indicates that the node is repeatable and *optional*. (In other words, there can be 0 or more occurrences of this.)

Always use this setting if appropriate to avoid unnecessary data checking and the associated performance hit. Use this especially if you know that the underlying data sources enforce referential integrity between parent-child items.

The following examples demonstrate this guideline:

- [“Example 3: Find all Broadband customers \(CUSTOMER is Repeatable and Optional\)” on page 3-46](#)
- [“Example 4: Find all Broadband customers \(CUSTOMER is Repeatable and Required\)” on page 3-46](#)
- [“Example 5: Find all Broadband customers and return their Wireless orders if the customer has Wireless orders \(ORDER is Required and Optional\)” on page 3-47](#)
- [“Example 6: Find the list of all Broadband customers that have at least one Wireless order and return their Wireless orders \(ORDER is Repeatable and Required\)” on page 3-48](#)

3. Project at least one element from each data source that is part of the query to the target schema. The following examples illustrates this guideline:

- “Example 1: Find all Broadband customers who are also Wireless customers” on page 3-42
 - “Example 2: Find all Broadband customers and their Wireless line items” on page 3-43
4. If your eventual goal is to create a data view from your query, your target schema should only contain required elements that are utilized in the query. For example, if the Customer table contains first_name, last_name, email, and phone elements and each of those elements is required in the target schema, then you need to map each element of your query before saving it.

Alternatively, you can modify your target schema to reflect only the elements your query is using or give your revised target schema a new name. One of the benefits of this approach is that when your data view is created, the only elements of your schema available for queries are those you specifically identify through the target schema.

In the Data View Builder you can save a target schema using the menu commands:

File -> Save Target Schema

See Adding a Target Schema for details.

Examples of Effective Query Design

For the following examples, assume we have two schema sets (databases) with the following entities (tables).

Broadband Schema with the following tables:

- Customer
- Order
- Line Item

Wireless Schema with the following tables:

- Customer
- Order
- Line Item

Example 1: Find all Broadband customers who are also Wireless customers

In this situation you do not project anything from the Wireless customer table.

The generated query will iterate over all customers in Broadband and check for the existence of a matching customer in Wireless. This query also ensures duplicate customers are not returned from Broadband in the event a Broadband customer matches more than one Wireless customer.

Instructions to create query using Data View Builder:

1. Map the source PB-BB.CUSTOMER FIRST_NAME and LAST_NAME to the target CUSTOMER FIRST_NAME and LAST_NAME respectively.
2. Create the condition PB_BB.CUSTOMER CUSTOMER_ID eq PB_WL.CUSTOMER.CUSTOMER_ID

The query looks like:

```
<db>
{
  for $PB_BB.CUSTOMER_1 in document("PB-BB")/db/CUSTOMER
  let $CUSTOMER_2 :=
    for $PB_WL.CUSTOMER_3 in document("PB-WL")/db/CUSTOMER
    where ($PB_BB.CUSTOMER_1/CUSTOMER_ID eq
           $PB_WL.CUSTOMER_3/CUSTOMER_ID)
    return
    xf:true()
  where xf:not(xf:empty($CUSTOMER_2))
  return
  <CUSTOMER>
    <FIRST_NAME>{ xf:data($PB_BB.CUSTOMER_1/FIRST_NAME) }</FIRST_NAME>
    <LAST_NAME>{ xf:data($PB_BB.CUSTOMER_1/LAST_NAME) }</LAST_NAME>
  </CUSTOMER>
}
</db>
```

If you do not care about duplicates or know there will not be duplicates, you can avoid `xf:entity(...)` checking by projecting an element from the Wireless customer table.

Instructions to create this alternative version of the query using Data View Builder:

1. Map the source PB-BB.CUSTOMER FIRST_NAME and LAST_NAME to the target CUSTOMER FIRST_NAME and LAST_NAME respectively.

2. Map the source PB-WL.CUSTOMER STATE to the target CUSTOMER STATE (Additional projection).
3. Create the condition PB_BB.CUSTOMER CUSTOMER_ID eq PB_WL.CUSTOMER.CUSTOMER_ID

The query now looks like:

```
<db>
{
  for $PB_WL.CUSTOMER_1 in document("PB-WL")/db/CUSTOMER
  for $PB_BB.CUSTOMER_2 in document("PB-BB")/db/CUSTOMER
  where ($PB_BB.CUSTOMER_2/CUSTOMER_ID eq $PB_WL.CUSTOMER_1/CUSTOMER_ID)
  return
  <CUSTOMER>
    <FIRST_NAME>{ xf:data($PB_BB.CUSTOMER_2/FIRST_NAME) }</FIRST_NAME>
    <LAST_NAME>{ xf:data($PB_BB.CUSTOMER_2/LAST_NAME) }</LAST_NAME>
    <STATE>{ xf:data($PB_WL.CUSTOMER_1/STATE) }</STATE>
  </CUSTOMER>
}
</db>
```

Example 2: Find all Broadband customers and their Wireless line items

This query basically asks for all Broadband customers and Wireless line items for which there exists a Wireless order that joins with both the Broadband customer and Wireless line item.

Now for this situation user does not project anything from Wireless order table.

The generated query will iterate over all customers and in Broadband, then for each line item it will check for the existence of a matching order in Wireless that also matches a customer in Broadband.

Instructions to create query using Data View Builder:

1. Map the source PB-BB.CUSTOMER FIRST_NAME and LAST_NAME to the target CUSTOMER FIRST_NAME and LAST_NAME respectively.
2. Map the source PB-WL.CUSTOMER_ORDER_LINE_ITEM PRODUCT_NAME and EXPECTED_SHIP_DATE to the target CUSTOMER_ORDER_LINE_ITEM PRODUCTION and EXPECTED_SHIP-DATE respectively.
3. Create the condition PB_BB.CUSTOMER CUSTOMER_ID eq PB_WL.CUSTOMER_ORDER.CUSTOMER_ID

3 Designing Queries

4. Create the condition `PB_WL.CUSTOMER_ORDER.ORDER_ID eq PB_WL.CUSTOMER_ORDER_LINE_ITEM.ORDER_ID`

The query looks like:

```
<ROWS>
{
  for $PB_BB.CUSTOMER_1 in document("PB-BB")/db/CUSTOMER
  return
  <CUSTOMER>
  <FIRST_NAME>{ xf:data($PB_BB.CUSTOMER_1/FIRST_NAME) }</FIRST_NAME>
  <LAST_NAME>{ xf:data($PB_BB.CUSTOMER_1/LAST_NAME) }</LAST_NAME>
  <CUSTOMER_ORDER>
  {
    for $PB_WL.CUSTOMER_ORDER_LINE_ITEM_2 in
      document("PB-WL")/db/CUSTOMER_ORDER_LINE_ITEM
    let $CUSTOMER_ORDER_LINE_ITEM_3 :=
      for $PB_WL.CUSTOMER_ORDER_4 in
        document("PB-WL")/db/CUSTOMER_ORDER
      where ($PB_BB.CUSTOMER_1/CUSTOMER_ID eq
              $PB_WL.CUSTOMER_ORDER_4/CUSTOMER_ID)
            and ($PB_WL.CUSTOMER_ORDER_4/ORDER_ID eq
                  $PB_WL.CUSTOMER_ORDER_LINE_ITEM_2/ORDER_ID)
      return
      xf:true()
    where xf:not(xf:empty($CUSTOMER_ORDER_LINE_ITEM_3))
    return
    <CUSTOMER_ORDER_LINE_ITEM>
    <PRODUCT_NAME>{
      xf:data($PB_WL.CUSTOMER_ORDER_LINE_ITEM_2/PRODUCT_NAME)
    }</PRODUCT_NAME>
    <EXPECTED_SHIP_DATE>{
      xf:data($PB_WL.CUSTOMER_ORDER_LINE_ITEM_2/EXPECTED_SHIP_DATE)
    }</EXPECTED_SHIP_DATE>
  }</CUSTOMER_ORDER_LINE_ITEM>
}
</CUSTOMER_ORDER>
</CUSTOMER>
}
</ROWS>
```

For performance reasons, we recommend that you project the intermediate data, especially if you do not care about duplicates or know there will not be duplicates. In the example above, you can project an element from the Wireless order table.

Instructions to create query using Data View Builder:

1. Map the source PB-BB.CUSTOMER FIRST_NAME and LAST_NAME to the target CUSTOMER FIRST_NAME and LAST_NAME respectively.
2. Map the source PB-WL.CUSTOMER_ORDER ORDER_ID to the target CUSTOMER_ORDER ORDER_ID. (Additional projection)
3. Map the source PB-WL.CUSTOMER_ORDER_LINE_ITEM PRODUCT_NAME and EXPECTED_SHIP_DATE to the target CUSTOMER_ORDER_LINE_ITEM PRODUCTION and EXPECTED_SHIP-DATE respectively.
4. Create the condition PB_BB.CUSTOMER CUSTOMER_ID eq PB_WL.CUSTOMER_ORDER.CUSTOMER_ID
5. Create the condition PB_WL.CUSTOMER_ORDER.ORDER_ID eq PB_WL.CUSTOMER_ORDER_LINE_ITEM.ORDER_ID

The query looks like:

```
<ROWS>
{
  for $PB_BB.CUSTOMER_1 in document("PB-BB")/db/CUSTOMER
  return
  <CUSTOMER>
    <FIRST_NAME>{ xf:data($PB_BB.CUSTOMER_1/FIRST_NAME) }</FIRST_NAME>
    <LAST_NAME>{ xf:data($PB_BB.CUSTOMER_1/LAST_NAME) }</LAST_NAME>
    {
      for $PB_WL.CUSTOMER_ORDER_2 in document("PB-WL")/db/CUSTOMER_ORDER
      where ($PB_BB.CUSTOMER_1/CUSTOMER_ID eq
              $PB_WL.CUSTOMER_ORDER_2/CUSTOMER_ID)
      return
      <CUSTOMER_ORDER>
        <ORDER_ID>{ xf:data($PB_WL.CUSTOMER_ORDER_2/ORDER_ID)
                    }</ORDER_ID>
        {
          for $PB_WL.CUSTOMER_ORDER_LINE_ITEM_3 in
            document("PB-WL")/db/CUSTOMER_ORDER_LINE_ITEM
          where ($PB_WL.CUSTOMER_ORDER_2/ORDER_ID eq
                  $PB_WL.CUSTOMER_ORDER_LINE_ITEM_3/ORDER_ID)
          return
          <CUSTOMER_ORDER_LINE_ITEM>
            <PRODUCT_NAME>{
              xf:data($PB_WL.CUSTOMER_ORDER_LINE_ITEM_3/PRODUCT_NAME)
            }</PRODUCT_NAME>
            <EXPECTED_SHIP_DATE>{
              xf:data($PB_WL.CUSTOMER_ORDER_LINE_ITEM_3/EXPECTED_SHIP_DATE) }
            </EXPECTED_SHIP_DATE>
        }
    }
}
```

3 Designing Queries

```
        </CUSTOMER_ORDER_LINE_ITEM>
      }
    </CUSTOMER_ORDER>
  }
</CUSTOMER>
}
</ROWS>
```

Example 3: Find all Broadband customers (CUSTOMER is Repeatable and Optional)

The target schema is `ROWS (CUSTOMER*)`. This query returns the root element and all Broadband customers. Since, `CUSTOMER` is optional, an empty `<ROWS/>` element could be returned as the result of the query since it would conform to the schema.

Instructions to create query using Data View Builder:

- Map the source `PB-BB.CUSTOMER FIRST_NAME` and `LAST_NAME` to the target `CUSTOMER FIRST_NAME` and `LAST_NAME` respectively.

The following query will be generated. Notice that this query will indeed return an empty root element `<ROWS/>` if there are not any Broadband customers.

The query looks like:

```
<ROWS>
{
  for $PB_BB.CUSTOMER_1 in document("PB-BB")/db/CUSTOMER
  return
  <CUSTOMER>
    <FIRST_NAME>{ xf:data($PB_BB.CUSTOMER_1/FIRST_NAME) }</FIRST_NAME>
    <LAST_NAME>{ xf:data($PB_BB.CUSTOMER_1/LAST_NAME) }</LAST_NAME>
  </CUSTOMER>
}
</ROWS>
```

Example 4: Find all Broadband customers (CUSTOMER is Repeatable and Required)

This time the target schema is `ROWS (CUSTOMER+)`. Again, this query returns the root element and all Broadband customers. But, since, `CUSTOMER` is required, in case there are no Broadband customer, an empty `<ROWS/>` element cannot be returned as the result of the query since such a result would not conform to the given target schema.

Instructions to create query using Data View Builder:

- Map the source `PB-BB.CUSTOMER FIRST_NAME` and `LAST_NAME` to the target `CUSTOMER FIRST_NAME` and `LAST_NAME` respectively.

Below is the query generated under this schema. This query will return the root element and all Broadband customers that exist. Observe that if there are not any Broadband customers then an empty result will be returned (not even the root element).

```
let $CUSTOMER_1 :=
  for $PB_BB.CUSTOMER_2 in document ("PB-BB")/db/CUSTOMER
  return
  <CUSTOMER>
  <FIRST_NAME>{ xf:data($PB_BB.CUSTOMER_2/FIRST_NAME)
  }</FIRST_NAME>
  <LAST_NAME>{ xf:data($PB_BB.CUSTOMER_2/LAST_NAME) }</LAST_NAME>
  </CUSTOMER>
where xf:not(xf:empty($CUSTOMER_1))
return
<ROWS>
  { $CUSTOMER_1 }
</ROWS>
```

The pattern of this query is discussed in more detail in [“Example 6: Find the list of all Broadband customers that have at least one Wireless order and return their Wireless orders \(ORDER is Repeatable and Required\)”](#) on page 3-48.

Example 5: Find all Broadband customers and return their Wireless orders if the customer has Wireless orders (ORDER is Required and Optional)

In this case, the target schema is `ROWS (CUSTOMER* (ORDER*))`. The target schema allows for customers with zero orders. This means that the query can (and should) return customers without orders. Practically, this makes the query is a left outer-join between customers and orders.

Instructions to create query using Data View Builder:

1. Map the source `PB-BB.CUSTOMER FIRST_NAME` and `LAST_NAME` to the target `CUSTOMER FIRST_NAME` and `LAST_NAME` respectively.
2. Map the source `PB-WL.CUSTOMER_ORDER ORDER_DATE` and `SHIP_METHOD` to the target `CUSTOMER_ORDER ORDER_DATE` and `SHIP_METHOD` respectively.
3. Create the condition `PB_BB.CUSTOMER CUSTOMER_ID eq PB_WL.CUSTOMER_ORDER.CUSTOMER_ID`

The query looks like:

3 Designing Queries

```
<ROWS>
{
  for $PB_BB.CUSTOMER_1 in document ("PB-BB")/db/CUSTOMER
  return
  <CUSTOMER>
    <FIRST_NAME>{ xf:data($PB_BB.CUSTOMER_1/FIRST_NAME) }</FIRST_NAME>
    <LAST_NAME>{ xf:data($PB_BB.CUSTOMER_1/LAST_NAME) }</LAST_NAME>
    {
      for $PB_WL.CUSTOMER_ORDER_2 in document ("PB-WL")/db/CUSTOMER_ORDER
      where ($PB_BB.CUSTOMER_1/CUSTOMER_ID eq
              $PB_WL.CUSTOMER_ORDER_2/CUSTOMER_ID)
      return
      <CUSTOMER_ORDER>
        <ORDER_DATE>{ xf:data($PB_WL.CUSTOMER_ORDER_2/ORDER_DATE) }
        </ORDER_DATE>
        <SHIP_METHOD>{ xf:data($PB_WL.CUSTOMER_ORDER_2/SHIP_METHOD) }
        </SHIP_METHOD>
      </CUSTOMER_ORDER>
    }
  </CUSTOMER>
}
```

Example 6: Find the list of all Broadband customers that have at least one Wireless order and return their Wireless orders (ORDER is Repeatable and Required)

In this case, the target schema is `ROWS (CUSTOMER* (ORDER+))`. Now, the target schema does not allow for customers with zero orders. This means that the query should not return customers without orders. Practically, this makes the query is a (natural) join between customers and orders.

Instructions to create query using Data View Builder:

1. Map the source `PB-BB.CUSTOMER FIRST_NAME` and `LAST_NAME` to the target `CUSTOMER FIRST_NAME` and `LAST_NAME` respectively.
2. Map the source `PB-WL.CUSTOMER_ORDER ORDER_DATE` and `SHIP_METHOD` to the target `CUSTOMER_ORDER ORDER_DATE` and `SHIP_METHOD` respectively.
3. Create the condition `PB_BB.CUSTOMER CUSTOMER_ID eq PB_WL.CUSTOMER_ORDER.CUSTOMER_ID`

The query looks like:

```
<ROWS>
{
```

```

for $PB_BB.CUSTOMER_1 in document("PB-BB")/db/CUSTOMER
let $CUSTOMER_ORDER_2 :=
    for $PB_WL.CUSTOMER_ORDER_3 in
        document("PB-WL")/db/CUSTOMER_ORDER
    where ($PB_BB.CUSTOMER_1/CUSTOMER_ID eq
            $PB_WL.CUSTOMER_ORDER_3/CUSTOMER_ID)
    return
        <CUSTOMER_ORDER>
        <ORDER_DATE>{ xf:data($PB_WL.CUSTOMER_ORDER_3/ORDER_DATE) }
        </ORDER_DATE>
        <SHIP_METHOD>{ xf:data($PB_WL.CUSTOMER_ORDER_3/SHIP_METHOD) }
        </SHIP_METHOD>
        </CUSTOMER_ORDER>
where xf:not(xf:empty($CUSTOMER_ORDER_2))
return
<CUSTOMER>
    <FIRST_NAME>{ xf:data($PB_BB.CUSTOMER_1/FIRST_NAME) }</FIRST_NAME>
    <LAST_NAME>{ xf:data($PB_BB.CUSTOMER_1/LAST_NAME) }</LAST_NAME>
    { $CUSTOMER_ORDER_2 }
</CUSTOMER>
}
</ROWS>

```

The general pattern for handling required repeatable elements in the target schema is as follows. The goal is to be able to check for existence of at least one element before we generate the parent. Generation of required repeatable elements is “promoted” the element to the nearest optional repeatable ancestor (or the root of the result if there is no such element). There the list of elements is computed inside a `let` clause. After that, and the result (list) of this `let` clause is checked whether it is empty or not, before producing the rest of the result.

In this case, the `ORDER` element is required so we need to check for existence of orders before we produce customer. This means that we need to generate the list of orders for each customer, and output the customer only if this list is not empty.

Source Replication

A source is said to be replicated if the source appears multiple times in a query. Specifically in XQuery, a source is replicated if `document`(“source name”) appears multiple times in the XQuery, usually appearing in two different for clauses. Similarly in SQL, a source is replicated if the source (table) appears twice in a `FROM` clause (or in two different `FROM` clauses). (See the next section for examples.)

Why is source replication necessary?

The simplest example of a necessary source replication would be a self-join in SQL. In the classic example of a self-join, the query wants to get all the pairs of employee names to manager names from a single employee table:

```
SELECT e.name, m.name
FROM employee e, employee m
WHERE e.manager_id = m.id
```

In XQuery, the query would look like:

```
<employee_managers>
{
for $e in document("employee")//employee
for $m in document("employee")//employee
where $e.manager_id eq $m.id
return
<employee_manager>
  <employee> {$e.name} </employee>
  <manager> {$m.name} </manager>
}
</employee_managers>
```

In both of these examples, given the sources, there is no way to write these queries without replicating the sources.

When is source replication necessary?

Source replication is necessary whenever you want to use a source for two different purposes that will require iterating over the source twice. Another way to state it is when two different tuples from a source will be required at the same time.

When should you manually replicate sources?

In ambiguous cases, both replicating and not replicating a source would lead to reasonable queries.

For example, at the beginning of this section, we presented a self-join to get employee-manager pairs. Without replicating the source, you might try the following:

1. Map name to the target (get the employee name)
2. Join manager_id with id (join to get the manager)

3. Map name to the target (get the manager name)

Of course, the Data View Builder would interpret this query as: “give me all employees who are their own manager.” This interpretation is no less valid than the desired one.

There is no way to go into Advanced mode to fix this query. You simply must replicate the source in this case.

Next Steps

- If you are ready to jump in and start designing more complex queries, refer to the advanced example queries in [Chapter 9, “Query Cookbook.”](#)
- For detailed information on how to run a query, see [Chapter 5, “Testing Queries.”](#)
- For information on how to optimize a query for better performance, see [Chapter 4, “Optimizing Queries.”](#)

4 Optimizing Queries

The topics covered here relate directly to tasks you can accomplish in the Data View Builder while building and testing BEA Liquid Data for WebLogic™ queries. See [Tuning Performance](#) in *Deploying Liquid Data* for a broader discussion of factors related to tuning and performance of Liquid Data including query design, data sources, and platform considerations.

The following sections are included here:

- [Factors in Query Performance](#)
- [Using the Features on the Optimize Tab](#)
- [Source Order Optimization](#)
- [Optimization Hints for Joins](#)
 - [Choosing the Best Hint](#)
 - [Using Parameter Passing Hints \(ppleft or ppright\)](#)
 - [Using a Merge Hint](#)

Factors in Query Performance

Queries can be designed and built to optimize performance. Query performance tuning and optimization can be accomplished through the following approaches:

- Making decisions about what type of query to use based on a consideration of data sources and the nature of the data you are querying.

- Planning and designing the type of query to use and how to implement it based on factors like expected query result size, memory requirements, and ability to leverage stored queries as appropriate.
- Adding standard optimization *hints* to queries

This section covers some key factors related to performance and memory that you should consider while designing and building queries with the Data View Builder. Examples and recommendations for some typical scenarios and use cases are provided.

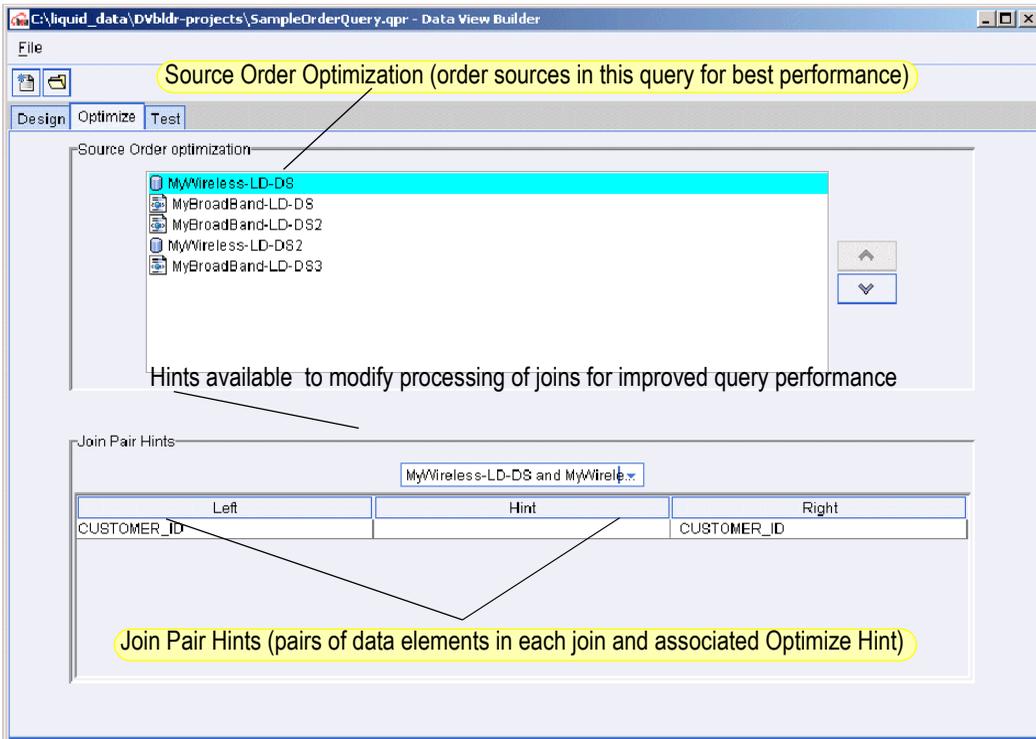
See [Tuning Performance](#) in *Deploying Liquid Data* for a broader discussion of factors related to tuning and performance of Liquid Data including query design, data sources, and platform considerations.

Using the Features on the Optimize Tab

To access tools to improve query performance, click on the Optimize tab. (See [Figure 4-1](#).) You can re-order data sources and add hints to a query from this tab provides as described in the following sections:

- [Source Order Optimization](#)
- [Optimization Hints for Joins](#)

Figure 4-1 Optimize Tab



Source Order Optimization

You can re-order source schemas on the top frame on the Optimize tab to improve query performance. To move a schema up or down, select the schema and click the up or down arrow buttons to the right of the list of schemas.

When a query uses data from two sources, the Liquid Data Server brings the two data sources into memory and creates an intermediate result (*cross-product*) using the two sources. If you specify more than two sources, the Liquid Data Server creates a cross-product of the first two sources, then continues to integrate each additional resource, one at a time, in the order that they appear in FOR clauses.

The size of a source is the number of tuples, or records, used in the query from that source. The size of the intermediate result depends on the input size of the first source multiplied by the input size of the second source and so on. A query is generally more efficient when it minimizes the size of intermediate results.

The order of the FOR clauses in the query matches the order of the data sources in the Source Order list. In general, you should order sources in ascending order by increasing size—that is, the smallest resource should appear first in the list and the largest resource should appear last.

Example: Source Order Optimization

Consider a query to find all managers and the departments they manage that contains a three-way join across three sources: Employees, Employees2 (Employees opened a second time), and Departments. This query joins the Employees schema ID field and the Employees2 schema MANAGER_ID to return all managers, and joins on the Employees schema DEPT_ID and Departments schema DEPARTMENT_NO to return the corresponding department information. The generated XQuery language looks like the following example.

```
for $EMP1 in document("Employees")/db/EMP
for $EMP2 in document("Employees")/db/EMP
for $DEPT in document("Department")/db/DEPT
where $EMP1/id eq $EMP2/manager_id and
      $EMP1/dept_id eq $DEPT/department_no
...

```

This creates a cross-product of Employees ID and Employees MANAGER_ID, then a cross-product with Departments DEPARTMENT_NO. If there are 100 employees, and five departments, the query would generate $(100 * 100) + (10,000 * 5)$ intermediate results.

A better plan would be to combine Employees with Departments first, then combine that result with Employees2. The effect is to generate $(100 * 5) + (500 * 100)$ intermediate results. The generated XQuery language looks like the following example.

```
for $EMP1 in document("Employees")/db/EMP
for $DEPT in document("Department")/db/DEPT
for $EMP2 in document("Employees")/db/EMP
where $EMP1/id eq $EMP2/manager_id and
      $EMP1/dept_id eq $DEPT/department_no
...

```

Optimization Hints for Joins

A query hint is a way to supply more information to the Liquid Data server about how to process the join.

The Optimize tab on the Data View Builder provides a drop-down menu for where you can select a hint for each join in the query that helps the Liquid Data Server choose the most appropriate join algorithm. (See the Optimize tab in [Figure 4-1](#).)

Query hints appear in the query as character strings enclosed within braces `{--! hint !--}`. They specify which join algorithm should be selected when the query runs. The Join Hints frame contains a drop-down list of data source pairs, and a table that shows all the joins for each pair. Only source pairs that have join conditions across them appear in the drop-down list. For each join condition in the table, you can provide a hint about how to join the data most efficiently.

After you run the query, you can always return to the Optimize view to change the source orders and the hints for each join operation.

Choosing the Best Hint

The Liquid Data Server has three hints to choose from when it processes a join request. By default no hints are specified. To add a hint, select the join to which you want the hint to apply and choose a hint from the drop-down Query Hints list. The available hints are shown below.

Table 4-1 Optimization Hints

Hint	Description	Syntax
No Hint (default)	Index	
Left	Parameter Pass to the Left (ppleft)	<code>{--! ppleft !--}</code>
Right	Parameter Pass to the Right (ppright)	<code>{--! ppright !--}</code>
Merge	Merge	<code>{--! merge !--}</code>

Apply these rules to determine the correct hint to choose.

Table 4-2 When to Use a Hint

Use this Hint	When
No Hint (default)	The size of the source identified on the right side of the hint is small enough to fit into memory. Where the left and right sources are generally equal in size. There is at least one non-relational source used in the join.
Merge	Both the sources in the join are relational databases. Both the sources are large and cannot fit into memory.
Parameter Passing (Left or Right)	One of the sources has fewer objects than the other. When you choose the direction for the Parameter Passing hint, always choose the database to the left or right with the larger number of items as the receiver. For example, if there are more items on the right side of the equality, choose Right. The direction indicated in the hint identifies the side in the equation that receives the parameter.

Notes:

- Using optimization hints can help you improve performance on *equijoin* conditions, which contain only one equality. The optimization features do not support complex join conditions, such as $(A \text{ eq } B) \text{ eq } (C \text{ eq } D)$. This type of conditional expression would be treated as $A \text{ eq } (B \text{ eq } C \text{ eq } D)$.
- Choosing the wrong direction could degrade performance instead of improving it.

Using Parameter Passing Hints (`ppleft` or `ppright`)

Choose a Parameter Passing hint when one of the sources has fewer objects than the other. In order to use the parameter passing hints (`ppleft` and `ppright`) effectively, you need to know which data sources contain the larger data sets.

When you choose the direction for the Parameter Passing hint, always choose the data source to the left or right with the larger number of items as the *receiver*. For example, if there are more items on the right side of the equality, then pass the parameter to the Right. The direction indicated in the hint identifies the side in the equation that receives the parameter. In other words, the hints are named for the receiver.

Consider the following example, which is described fully in “[Example 1: Simple Joins](#)” on page 9-2 in [Chapter 9](#), “[Query Cookbook](#).”

Listing 4-1 XQuery with ppwright Hints

```
{--      Generated by Data View Builder 1.0  --}

<customers>
  {
    for $PB-WL.CUSTOMER_1 in document("PB-WL")/db/CUSTOMER
    where ($#wireless_id of type xs:string eq $PB-WL.CUSTOMER_1/CUSTOMER_ID)
    return
      <customer id={$PB-WL.CUSTOMER_1/CUSTOMER_ID}>
        <first_name>{ xf:data($PB-WL.CUSTOMER_1/FIRST_NAME) }</first_name>
        <last_name>{ xf:data($PB-WL.CUSTOMER_1/LAST_NAME) }</last_name>
        <orders>
          {
            for $PB-BB.CUSTOMER_ORDER_3 in
document("PB-BB")/db/CUSTOMER_ORDER
              where
($PB-WL.CUSTOMER_1/CUSTOMER_ID eq {--! ppwright !--} $PB-BB.CUSTOMER_2/CUSTOMER_ID)
              return
                <order id={$PB-BB.CUSTOMER_ORDER_3/ORDER_ID}
date={$PB-BB.CUSTOMER_ORDER_3/ORDER_DATE}></order>
          }
        </orders>
      </customer>
    }
  </customers>
```

Let’s focus on the second join in the example; the join between PB-WL customer IDs and PB-BB customer IDs:

```
      where
($PB-WL.CUSTOMER_1/CUSTOMER_ID eq {--! ppwright !--} $PB-BB.CUSTOMER_2/CUSTOMER_ID)
```

In the example above, the *where* clause indicates that the PB-WL data source CUSTOMER table will output only one customer ID. We can assume the PB-BB data source has a larger amount of customer IDs. We can optimize the join by providing the hint shown above (*ppwright*), which tells the server to retrieve the PB-WL customer

4 Optimizing Queries

information first and then pass the CUSTOMER ID as a parameter to the *right* to look for matches in the PB-BB data source. The engine will thus require much less memory and respond faster than if no hint was provided. Then it might have iterated through multiple records, and for each one asked the database to select the one with a highly optimized query.

Using a Merge Hint

Choose a merge hint when both the sources in the join are relational databases and both the sources are large and cannot fit into memory.

The following example shows the XQuery for a merge hint.

Listing 4-2 XQuery with Merge Hint

```
<root>
  {
    for $Wireless.CUSTOMER_1 in document("Wireless")/db/CUSTOMER
    for $Broadband.CUSTOMER_2 in document("Broadband")/db/CUSTOMER
    where ($Wireless.CUSTOMER_1/CUSTOMER_ID eq {--!merge!--}
$Broadband.CUSTOMER_2/CUSTOMER_ID)
    return
    <row>
      <CUSTOMER_ID>{ xf:data($Broadband.CUSTOMER_2/CUSTOMER_ID)
} </CUSTOMER_ID>
    </row>
  }
</root>
```

A *merge join* requires a minimal amount of memory to operate; however, it requires that the input be sorted on join attributes. A query using a merge join might have slower response time than a query without a hint, but the memory footprint is typically much smaller with the merge join.

Note: A merge join in a character column might yield unexpected results because the collating sequence for each database may vary. See [Table 4-3](#) for an example of how incompatible ordering sequences for strings from two different vendors can affect query results.

Table 4-3 Collation Sequences for Some Data Types Vary by Database Vendor

Oracle	MS SQL
ORDER_ID_8009_4	ORDER_ID_8009_4
ORDER_ID_8010_0	ORDER_ID_801_0
ORDER_ID_8011_0	ORDER_ID_8010_0
ORDER_ID_8012_0	ORDER_ID_8011_0
ORDER_ID_801_0	ORDER_ID_8011_1
ORDER_ID_801_1	ORDER_ID_8012_0

To ensure predictable results you should use an index join when merging character (varchar, string, and so forth) columns from different databases.

5 Testing Queries

This topic describes how to test BEA Liquid Data for WebLogic™ queries. The following sections are included here:

- [Switching to the Test View](#)
- [Using Query Parameters](#)
- [Specifying Large Results for File Swapping](#)
- [Running the Query](#)
- [Viewing the Query Result](#)
- [Saving a Query](#)
 - [Saving a Query to the Repository as a “Stored Query”](#)
 - [Naming Conventions for Stored Queries](#)

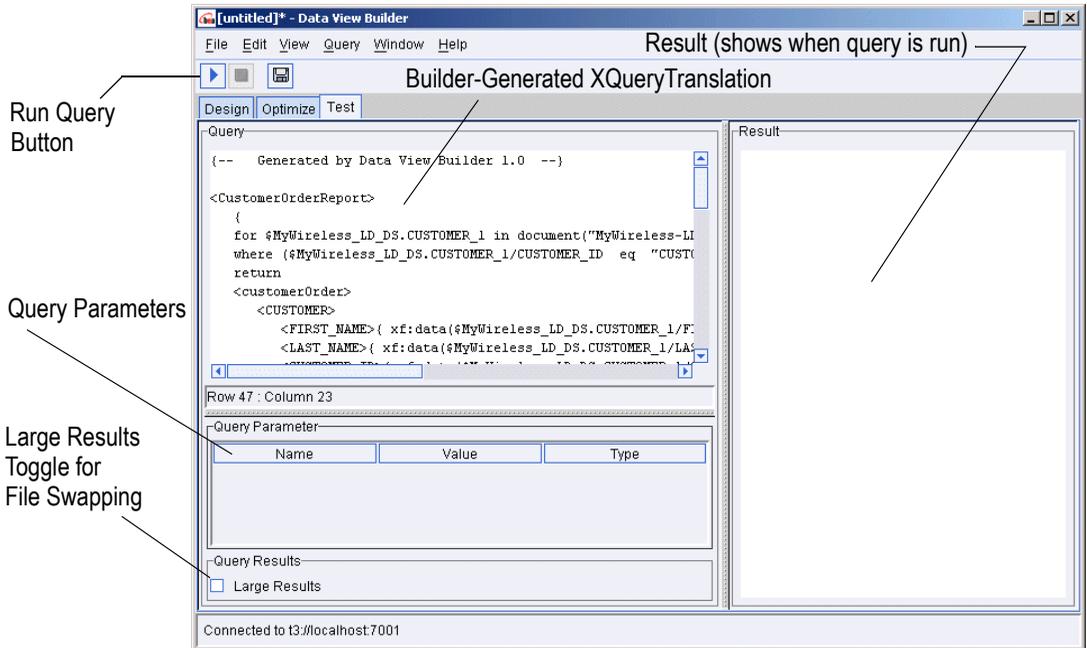
Switching to the Test View

The Data View Builder provides a “Test” view where you can view the generated XQuery language interpretation of the query elements you developed on the Design and Optimize tabs, and run the query against your data sources to verify the result.

From this view, you can provide different parameters to the query before you run it.

To switch to the Test View click the Test tab.

Figure 5-1 Test Tab



The query you developed on the Design and Optimize tabs is shown in XQuery language in the “Query” window on the upper left panel on the Test tab.

Using Query Parameters

You can use the Query Parameters panel to add variable values to a query each time you run it. The list of variables depends on the number of variables you defined as Query Parameters on the Design tab and which ones appear as one or more function parameters. (For details on defining query parameters, see [“Query Parameters: Defining” on page 2-15](#), which includes a list of supported data types for query parameters in [Table 2-2, “Query Parameter Types,” on page 2-17](#).)

Figure 5-2 Query Parameters Settings on Test Tab

Query Parameter		
Name	Value	Type
customer_id		xs:string
date1		xs:string

Double-click into a cell in the Values column to type a value.

Figure 5-3 Entering Values for Query Parameters

Query Parameter		
Name	Value	Type
customer_id	CUSTOMER_3	xs:string
date1	2002-08-01	xs:string

For some examples of using query parameters see the following example queries in the [Chapter 9, “Query Cookbook.”](#)

- “[Example 1: Simple Joins](#)” on page 9-2
- “[Example 2: Aggregates](#)” on page 9-8
- “[Example 3: Date and Time Duration](#)” on page 9-18

Specifying Large Results for File Swapping

Note: The “Large Results” option is a space/performance trade-off. You can expect a query to run more slowly when the this option is set to *on*. Before using this option, please increase heap size first. For other alternatives, see [Performance Tuning](#) in *Deploying Liquid Data*.

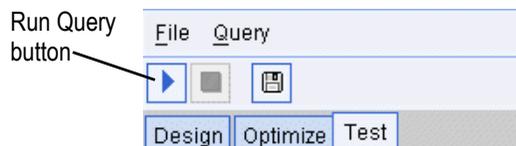
For a query that you know will produce a large result set, you can select “Large Results” in the Query Results panel on the Test tab. (An X next to Large Results indicates that the feature is *on*.) If the Large Results option is selected for a query, then Liquid Data uses swap files to temporarily store intermediate results on disk in order to prevent an out-of-memory error when the query is run.

You can explicitly specify a directory to use for file swapping on the Liquid Data Administration Console. For more information about this, see [Configuring Liquid Data Server Settings](#) in the *Liquid Data Administration Guide*.

Running the Query

When you are ready to run the query, click the Run Query button on the toolbar in the upper left.

Figure 5-4 Click the Run Query Button to Run the Query

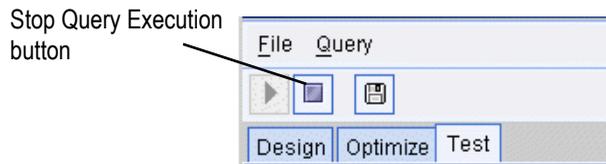


The query is run against your data sources and the result is displayed in the Results panel in XML format.

Stopping a Running Query

You can stop a running query before it has finished processing by clicking the Stop Query Execution button in the toolbar.

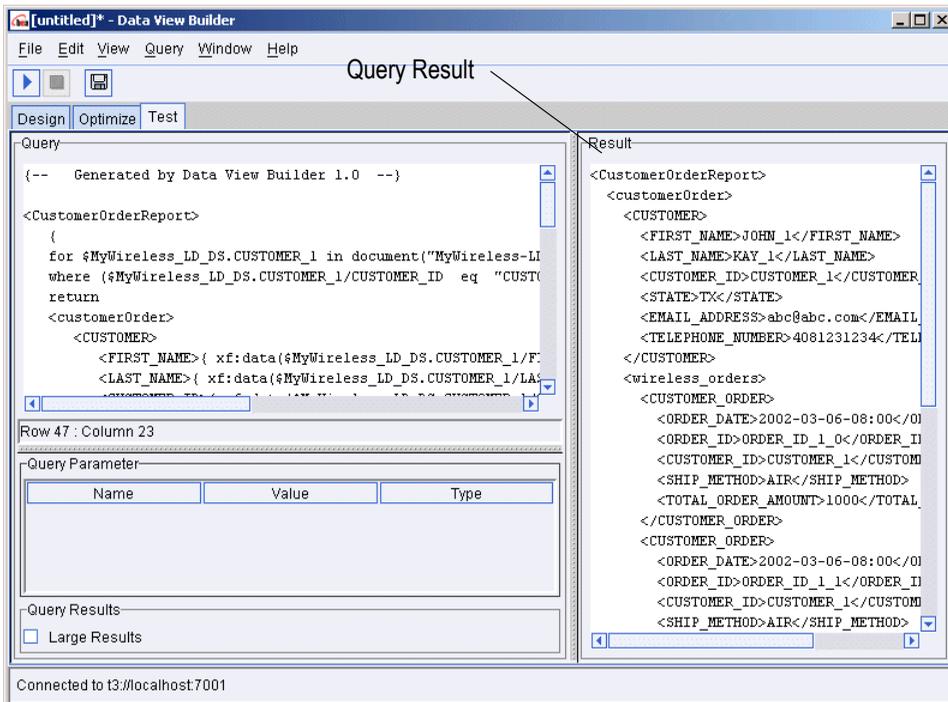
Figure 5-5 Click the “Stop Query Execution Button” to Stop a Running Query



Viewing the Query Result

When you run a query, the result is displayed in the Results window on the Test tab in XML format.

Figure 5-6 Query Result



Saving a Query

From the Test tab, you can save a query by choosing File—>Save Query from the menus or by clicking on the Save Query button on the toolbar.

You can save the query to a file on a local folder or other location on the network, or you can save the query to the Liquid Data server Repository in the `stored_queries` folder as described in the following section.

Note: Query files must be saved with a `.xq` extension. If you do not specify an extension, the Data View Builder automatically appends the `.xq` extension to the filename when the query file is saved.

Saving a Query to the Repository as a “Stored Query”

If the query is saved into the `stored_queries` folder in the Liquid Data server Repository, it becomes a *stored query* in Liquid Data. There is a performance benefit to using stored queries in Liquid Data in that the *query plan* is automatically cached in Liquid Data and you have the option to configure caching on the *query result* as well. For more information about using stored queries, see [“Stored Queries” on page 1-6 in Chapter 1, “Overview and Key Concepts.”](#)

To create a stored query do the following:

1. On the Test tab, choose File—>Save Query from the menus. (The File—>Save Query menu option is available only from the Test view.)
This brings up a file browser.
2. Use the file browser to navigate to the Repository.
3. The `stored_queries` folder is the only Liquid Data server repository directory available from the Data View Builder. This is the appropriate location in the repository in which to save a query.
4. Enter a name for the query in the File name field on the file browser and click Save. The query is saved to the `stored_queries` folder in the server repository with the appropriate `.xq` extension which identifies it as a stored query in Liquid Data.

You can reload a stored query using the Data View Build File : Open Query command. You may need to navigate to the directory containing your stored queries.

Once a stored query has been loaded, it can be run. See [Running the Query](#) for details.

Naming Conventions for Stored Queries

- **Stored queries need .xq extension**—Queries saved to the Liquid Data `stored_queries` folder in the Liquid Data server repository must have a `.xq` extension which identifies it as a stored query in Liquid Data. If you save the query via the Data View Builder, the `.xq` extension is automatically appended.
- **Names of queries to be generated as Web services must follow W3C XML tag naming conventions**—If you want to use Liquid Data to generate a Web service from a query, the query name must adhere to the same naming conventions as an XML tag since the query name is converted to an XML tag in the Web service generation process.
 - The most salient of these XML naming conventions is that the query name (which will be converted to an XML tag name) must be alphanumeric and must *begin* with an alphabetic character (letter)—not a number. No special characters (such as an underscore) are allowed in the name. For example, `myquery.xq` and `my12query.xq` are both query names that will work with Web services generation, whereas `12query.xq` will not work as a generated Web service.
 - For a complete description of naming conventions for schema tags see described in the *W3C XML Schema* document at <http://www.w3.org/XML/Schema>.

(For information on how to generate a Web service from a stored query, see [Generating and Publishing Web Services](#) in the Liquid Data *Administration Guide*.)

6 Using Data Views

Data views play a central role in the Liquid Data enterprise development model.

- [Enterprise and the Data View](#)
- [Understanding Data Views](#)
- [Creating a Data View](#)
- [Creating a Parameterized Data View](#)
- [Data View Query Samples](#)

Enterprise and the Data View

In Liquid Data, data views are central to solving the data integration problem one time (as opposed to once per query) and providing a basis for simpler application development work on top of that integrated view. In this model:

- A data architect with an intimate knowledge of the relationship of the available diverse data sources develops a set of data views based on the needs of various parts of the enterprise.

For example, a view of an employee developed for an enterprise might include employee salary and address information from one data source; information about their health insurance from another data source; information from their company assets (computer, phone, etc.) might be included from a third data source.

- Liquid Data is then used to create, refine, and validate each of the data views through queries built up through the Data View Builder.

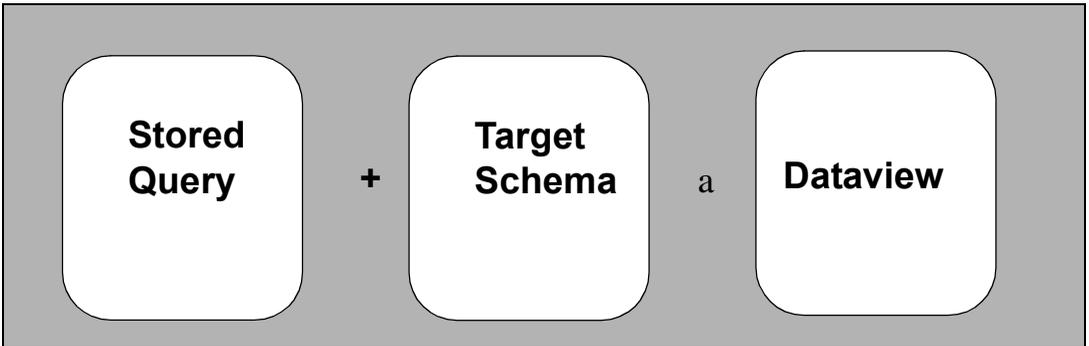
- Once validated, a reusable representation of each data view is developed through the Data View Builder and Administration Console as a new data view.
- Then the Liquid Data data view can be used throughout the enterprise as a *virtual data source* for queries. For example, a query for a new payroll division application might select salary information from this view.

In this model a data view provides an appropriate architectural view of corporate data that is available for specialized queries and sharable throughout the enterprise.

Understanding Data Views

In Liquid Data a stored query and a target schema comprise a data view.

Figure 6-1 Components of a Data View



To create a data view from a query:

- You first create a query and save it.
- Then you configure a data view data source description for the query in the WebLogic Server Administration Console.

To create a virtual data source in this way, you must first create a query and save it to the Liquid Data server repository, then configure a data view data source description for the query in the WebLogic Administration Console. It is recommended that you create the query and save it to the repository using the Data View Builder, but it is also possible to use hard-coded queries in generally the same way.

The following sections explain what a data view is and how to use a data view data source with the assumption that you are using the Data View builder to construct the query. Also included is a clarification of the relationship between a query and a data view.

Functionally, a data view extends the power of a stored query through its association with a target schema that describes the data. This combination allows a data view to be identified in the Data View Builder as a data source for additional queries.

The following sections describe in detail how to create Liquid Data data views and use such views as data sources. Also included is a discussion of the relationship between a query and a data view.

A Data View Use Case

eWorld Co, a company that through multiple mergers and acquisitions has 50,000 employees, also has multiple payroll systems. Using Liquid Data, information in each of these systems can be accessed. The company also has two relational databases from separate vendors for tracking incentive bonuses. Human Resources very frequently gets questions about when such bonus payments will show up in affected employee's paychecks.

- To enable HR to get answers to employees quickly and economically, an Information Technology data architect creates a query using Liquid Data that can access relevant information from the multiple payroll systems and the company's incentive bonus databases.
- Once satisfied that the query works, the IT architect creates a data view and makes it available to an HR data specialist. This specialist can then use Liquid Data to quickly get answers to inquiries from individual employees about their bonus payments.

The benefits of this approach are significant:

- A single integrated view can be created for use throughout an enterprise. Access to sensitive information is controlled and consistency is maintained.
- HR can quickly get the information it needs without having to either staff up with its own data architect or get in the queue for expensive and low-availability IT custom programming services.

- Since data views are typically created by information architects, more time can be spent designing and testing the generalized query.

Simple and Parameterized Data Views

The difference between a simple and a parameterized data view is that a parameterized data view has one or more input parameters. Specifically views that centrally contain functional sources such as an application view, web service, custom function, or stored procedure often require an input parameter.

Using Data Views as Data Sources

From the Data View Builder, you access a data view as you would any other data source. There is no limit to the number of data views that can be used in creating a new query, although currently there may be performance implications to nesting data views. A data view can reference on another data view.

Creating a Data View

The following sections explain the steps needed to turn a query into a data view data source:

- [Creating and Saving the Query to the Liquid Data Repository](#)
- [Configuring a Data View Data Source Description](#)
- [Adding a Data View as a Data Source](#)

Creating and Saving the Query to the Liquid Data Repository

Follow these steps to create and save a query to the Liquid Data Repository:

1. Construct the query in the Design view as described in [Chapter 3, “Designing Queries.”](#)
2. Test the query in the Test Query view as described in [Chapter 5, “Testing Queries.”](#)
3. Save the query to the Liquid Data repository as a stored query as described in [Saving a Query to the Repository as a “Stored Query”](#) in [Chapter 5, “Testing Queries.”](#)

Note: When you are creating a data view, it is important that the query and its target schema be in conformance. In the current release this means that all required elements in a target schema must be mapped if the query is to be turned into a view. See [“Source and Target Schemas”](#) and subsequent discussions for details.

Alternatively you can load queries and target schemas into the Liquid Data repository directly using the Administration Console. See [Uploading Files to the Server Repository](#) for details.

Configuring a Data View Data Source Description

In the WebLogic Server Administration console, configure a data view source description for the query as described in [Configuring Access to Data Views](#) in the *Liquid Data Administration Guide*. Then follow these steps:

1. In the Administration Console click the Repository tab.
2. Double-click on the Stored Queries folder.
3. Find the stored query you want to use and click the `Data View Data Source` link.

This links you into the Data View configuration tab, automatically copies the stored query to the `data_views` folder for you, and assigns an `xv` extension to the file name.

See “[Managing the Liquid Data Server Repository](#)” in the *Liquid Data Administration Guide* for additional details.

Adding a Data View as a Data Source

After you have created the data view, reconnect to the liquid Data server using the `File -> Connect` menu command. Your new data view should appear under Data Views when the Sources tab in Design mode is selected (see [Figure 6-5](#)).

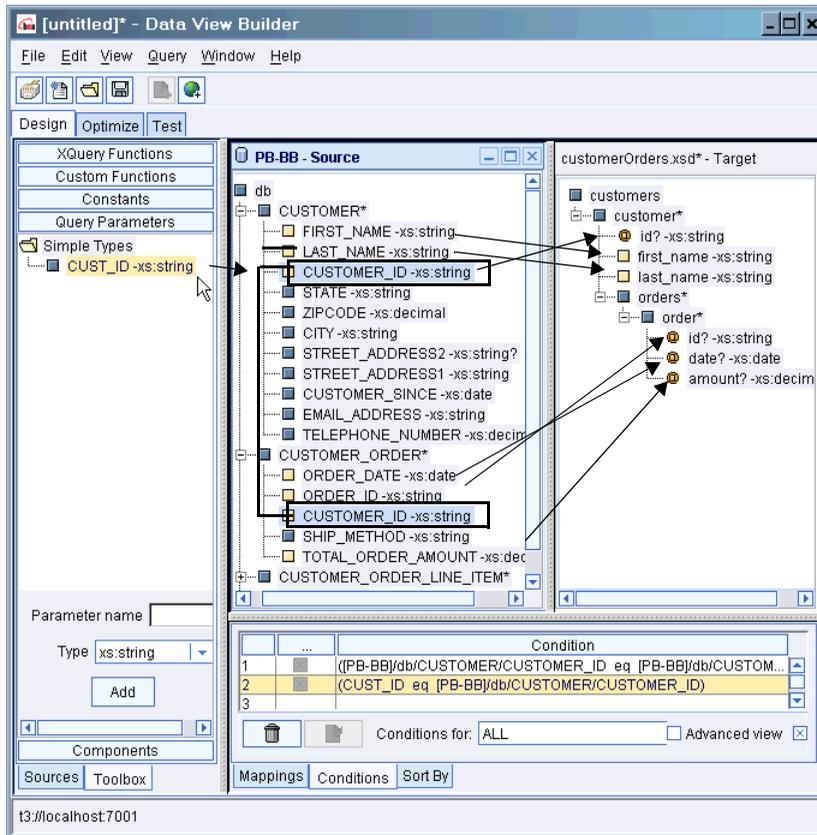
Creating a Parameterized Data View

You can use the following simple example to create a stored query and then turn it into a parameterized data view that retrieves customer order information based on a unique customer ID.

Note: To follow along with the creation of this example data view, you should have the Liquid Data sample server installed and running and be familiar with the sample. If not, please see the Liquid Data *Getting Started* guide.

1. Open the Data View Builder, drag the relational database source `pb-bb` onto the Liquid Data desktop. Set your target schema to `customerOrders.xsd`. Map elements to your target schema as shown in [Figure 6-2](#).

Figure 6-2 Creating a Parameterized Query in the Data View Builder



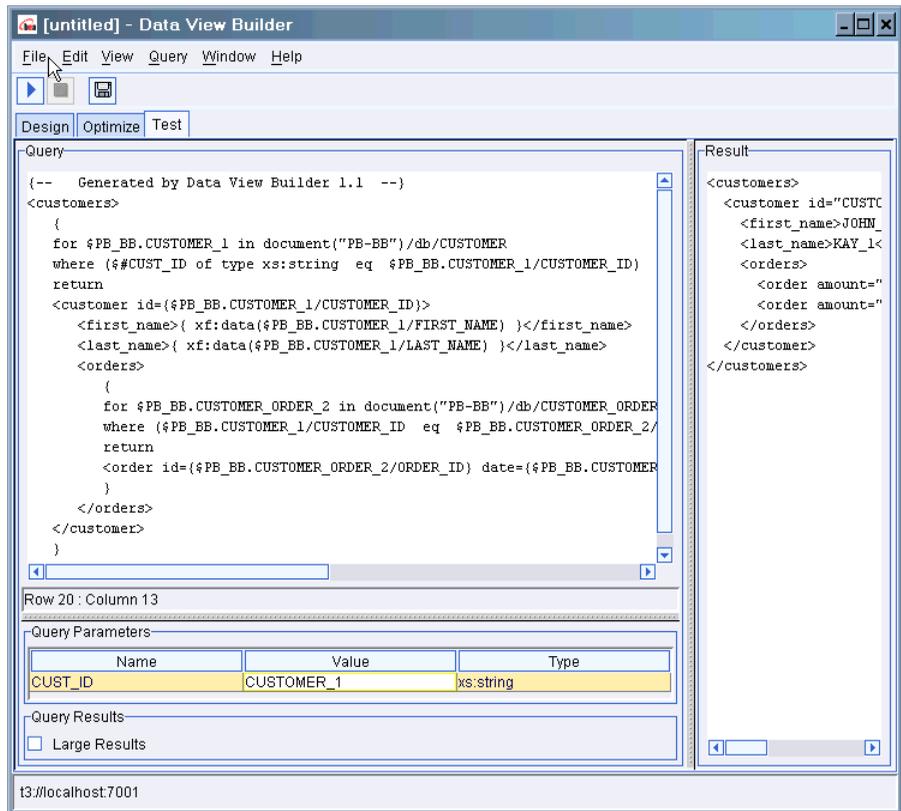
1. From the Liquid Data Toolbox tab, choose Query Parameter. Create a single query parameter, `CUST_ID` and using the pulldown Type menu. Assign it a type string of `xs:string`.
2. Drag `cust_id` to the `CUSTOMER_ID` field of the `CUSTOMER` table in the `PB_BB` data source (also shown in [Figure 6-2](#)).
3. Drag `CUSTOMER_ID` in the `Customer` table to `CUSTOMER_ID` in the `Customer Order` table to create a join.
4. In `Test` mode supply `CUSTOMER_1` as a value for `CUST_ID` and run the query.

Note: Values are case-sensitive.

The Data View Builder will display an XML report containing information on the orders made by this particular customer.

- Using the File -> Save Query menu command in the Data View Builder, save your query (shown in Figure 6-3) under the name `param_dv` to the Repository folder. It will automatically be placed in the `ld_repository/stored_query` folder and the extension `.xq` appended.

Figure 6-3 Saving the Query

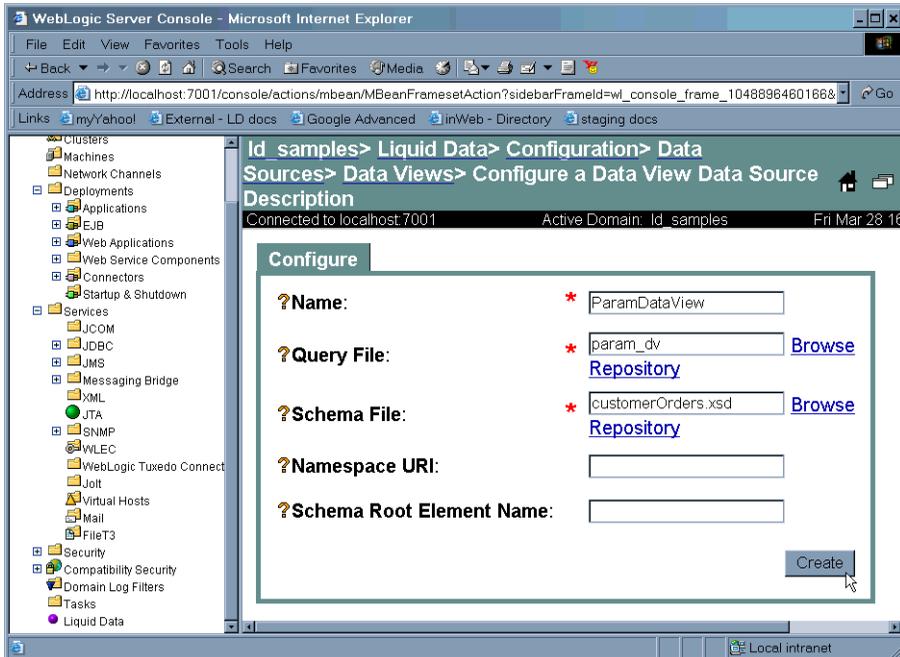


- Now you can use the Liquid Data Administration Console to create your data view from your newly saved query.

- a. Start the Administration Console.
- b. Click the Liquid Data node.
- c. Select the Repository tab.
- d. Go to stored_queries.
- e. Choose the query param-dv.xq
- f. Select the Data View Data Source option.
- g. Enter information as shown in Figure 6-4.
- h. Click Create.

Your new data view should appear in the Administration Console list of available data views.

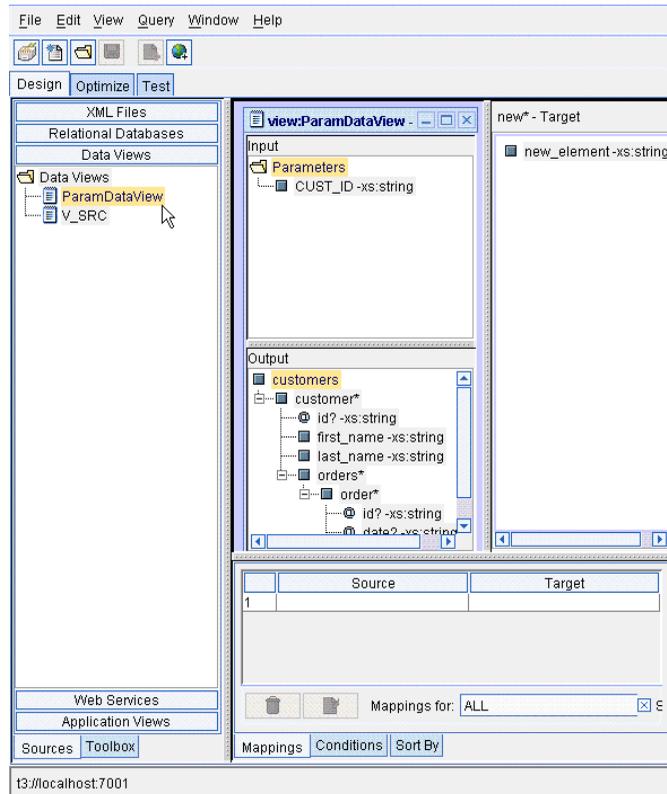
Figure 6-4 Creating the Data View in the Administration Console



See “Creating Data Views from Stored Queries” in the *Liquid Data Administration Guide* for information on how to generate data views from stored queries.

7. Return to the Data View Builder. Select **File -> Connect**. When you click on Data Views, your newly created data view should appear.

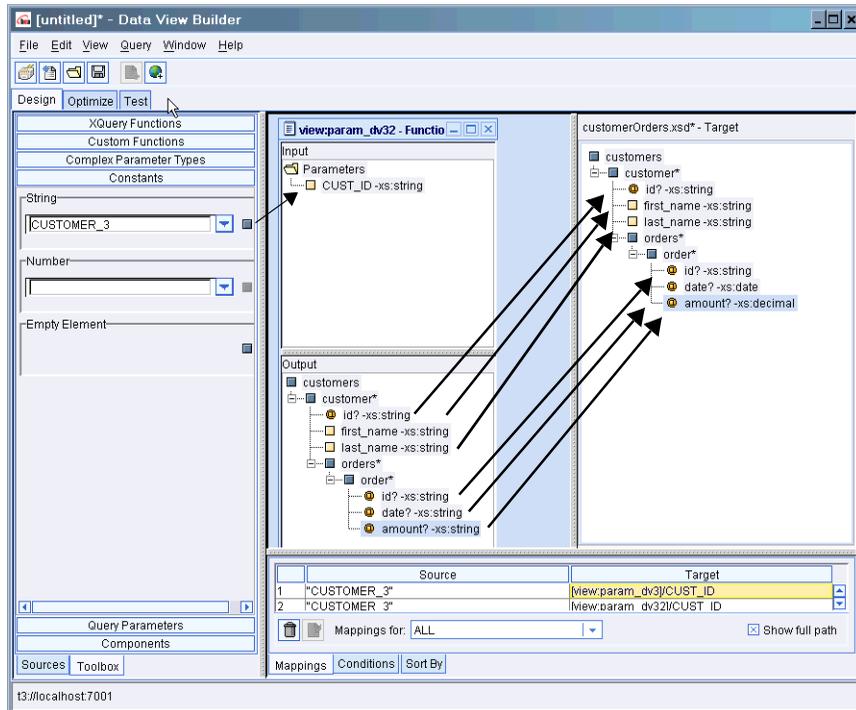
Figure 6-5 Data View on the Liquid Data Desktop



8. Drag the data view onto the Liquid Data desktop as you would any other data source (Figure 6-5).
9. Add a valid target schema. In this case, you can use the File menu **Set Selected Source as Target Schema** command. (The generated schema is shown in Figure 6-6.)

You may see a message asking if it is OK to close the existing target schema since that will remove all its mappings in the Data View Builder. Click Yes.

Figure 6-6 Setting Input and Associating Columns With Target Schema



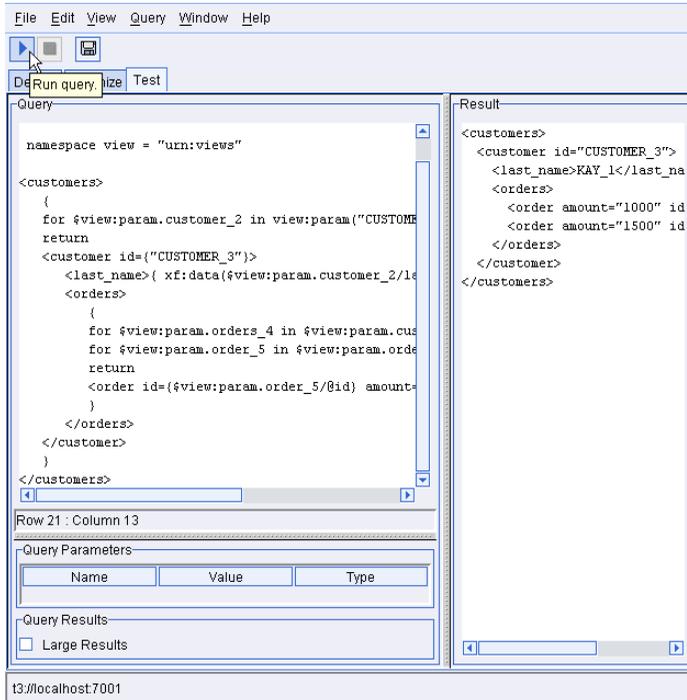
10. Using the Data View Builder toolbox create a string constant called `CUSTOMER_3` and drag it into the data view input `CUST_ID` (see [Figure 6-6](#)).

You could have provided an input parameter from a built-in XQuery function, custom function, an input from a web service, or another source.

11. Map all the elements in your target schema.

12. Test, then run your new query.

Figure 6-7 XQuery and Generated XML Report



Data View Query Samples

Two additional Data View Query samples are installed with the Liquid Data samples. These samples show how to create a data view, configure it as a data source, and then use that data source in other data views.

Instructions for running the samples are provided in readme files located at:

- `<LDHome>/samples/buildQuery/view/readme.htm`
- `<LDHome>/samples/buildQuery/parameter_view/readme.htm`

Also see the Liquid Data [Samples](#) page for more information on other available query samples.

7 Using Complex Parameter Types in Queries

You can use complex parameter types (CPTs) to make one or several streaming data sources available for Liquid Data queries. Such content is variously called *runtime source*, *data stream*, *real-time data*, or *in-flight XML*. As long as an XML schema can model the runtime source, you can use it with Liquid Data queries. Liquid Data complex parameter types enable the on-the-fly aspect of this query.

The following subjects are discussed in this chapter:

- [Understanding Complex Parameter Types](#)
- [Creating a Complex Parameter Type](#)
- [Complex Parameter Type Query Samples](#)

Understanding Complex Parameter Types

A complex parameter type is a user-defined variable that allows modelling of runtime data as a data source for a Liquid Data query. CPTs are defined through an XML schema. In other words, a CPT is a user-defined variable whose signature is expressed in via an XML schema.

In order to use CPTs, you need:

- An XML schema that models the type of the user-defined variable as a CPT.
- A complex parameter type definition that you configure through the Liquid Data Administration Console (see *Configuring Access to Complex Parameter Types* in the *Administration Guide*).
- A runtime source that provides XML data conforming to the XML schema mentioned above.

You can use the Data View Builder to develop and test the query that uses CPT. When testing a complex parameter types using the Data View Builder, you must assign an XML file to the CPT variable.

When you are satisfied with your ability to run your query using a runtime data source, the query can be invoked via a EJB API or via a JSP. For details on invoking queries programmatically and the Liquid Data API see:

- [Setting Complex Parameter Types](#) in *Invoking Liquid Data Queries Programmatically*.
- [Invoking Queries in EJB Clients](#) in *Invoking Liquid Data Queries Programmatically*.
- [Liquid Data 1.1 API Reference](#) (Javadoc).

There is also a Liquid Data EJB API sample in:

`BEA_HOME/weblogic700/liquidata/samples/ejbAPI`

A CPT Use Case

Complex parameter types allow you to access information that may not be available from traditional static data sources. For example, an enterprise frequently needs to bring together highly diverse pieces of information to complete a business activity or analysis.

Use Case

eWorldCo typically receives large electronically-transmitted orders for customized computer chips from companies and governments all over the world. Orders are transmitted using an agreed-upon XML schema provided by eWorld. When an order is received, a variety of commissions and bonuses become payable.

Some information about the transaction is readily available from existing data sources such as:

- products database
- sales discount schedule (from sales management software)
- a partner commission structure (from a spreadsheet maintained by the regional sales organization)

However, some data only becomes available when the order is received including:

- customer identification
- item
- quantity
- salesperson

When an order arrives, Liquid Data uses information from all these data sources, including the runtime order information. As the order is received as an in-flight XML document, a query is run and an XML report generated that calculates costs and commissions, taking into account both the order and cumulative discount and commission schedules for the buyers, middlemen, and sales people involved in the transaction.

Understanding CPT Schema and Data

This section describes elements of a sample CPT schema and its XML instance. It uses the DB-CPTCO sample installed with Liquid Data when describing a CPT Schema and XML Data Source. The sample is installed in the following directory:

```
BEA_HOME/weblogic700/liquidata/samples/buildQuery/db-cptco
```

The project file is `coCPTSample`.

Sample CPT Schema

The CPT schema shown in [Listing 7-1](#) (`coCptSample2.xsd`) is from the DB-CPTCO sample. It is located in the `LDrepository/schemas` directory.

This simple example defines a complex variable type containing four data elements: `customer_id`, `product_name`, `quantity`, and `price`. The complex type you design will vary depending on the signature of your runtime source.

Listing 7-1 DB-CPTCO Sample CPT Schema (`coCptSample2.xsd`)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="urn:schemas-bea-com:ld-cocpt"
  xmlns:cocpt="urn:schemas-bea-com:ld-cocpt"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="CustOrder">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="CUSTOMER_ORDER" minOccurs="0" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="CUSTOMER_ID" type="xsd:string"/>
              <xsd:element name=
                "NEW_ORDER_LINE_ITEM" type="cocpt:NEW_ORDER_LINE_ITEMType"
                minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="NEW_ORDER_LINE_ITEMType">
    <xsd:sequence>
```

```
<xsd:element name="PRODUCT_NAME" type="xsd:string"/>
<xsd:element name="QUANTITY" type="xsd:decimal"/>
<xsd:element name="PRICE" type="xsd:decimal"/>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Components of the sample schema shown in [Listing 7-1](#) include:

```
xmlns:cocpt="urn:schemas-bea-com:ld-cocpt"
```

Declares a namespace `cocpt` associated with the URI.

```
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

Declares a namespace `xsd` to the standard XML schema URI.

```
<xsd:element name="CustOrder">
```

Declares `CustOrder` as the schema root element.

It is the unique combination of namespace and schema root element that defines the portion of a schema used as a complex parameter type.

Note: One one instance of a CPT (that is, a unique combination of namespace and schema root element) can be available in the Data View Builder. If you try and duplicate a CPT under another alias name, a red mark will appear over the duplicate CPT name to indicate that it is unavailable.

Sample XML Data Stream

When you are first developing your query, you will likely want to create a sample XML data file to test your query with an EJB client such as the data in the Data View Builder. In the following listing from the DB-CPTCO sample XML data stream, note that the namespace must match that in the DB-CPTCO sample schema.

Listing 7-2 DB-CPTCO Sample XML Data Stream (coCptSample2.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<cocpt:CustOrder xmlns:cocpt="urn:schemas-bea-com:ld-cocpt"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:schemas-bea-com:ld-cocpt
  coCptSample2.xsd">
```

```
<CUSTOMER_ORDER>
  <CUSTOMER_ID>CUSTOMER_1</CUSTOMER_ID>
  <NEW_ORDER_LINE_ITEM>
    <PRODUCT_NAME>RBBC01</PRODUCT_NAME>
    <QUANTITY>1000</QUANTITY>
    <PRICE>20</PRICE>
  </NEW_ORDER_LINE_ITEM>
  <NEW_ORDER_LINE_ITEM>
    <PRODUCT_NAME>CS2610</PRODUCT_NAME>
    <QUANTITY>1000</QUANTITY>
    <PRICE>20</PRICE>
  </NEW_ORDER_LINE_ITEM>
</CUSTOMER_ORDER>
</cocpt:CustOrder>
```

The DB-CPTCO sample XML file is located in the *LDRepository/xml_files* directory.

Components in the sample XML data stream shown in [Listing 7-2](#) include:

```
cocpt:CustOrder xmlns:cocpt="urn:schemas-bea-com:ld-cocpt"
```

Defines `urn:schemas-bea-com:ld-cocpt` as the namespace aliased to `cocpt` and `CustOrder` as the complex data type:

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

Declares the standard XML schema instance URI.

```
xsi:schemaLocation="urn:schemas-bea-com:ld-cocpt coCptSample2.xsd"
```

Identifies the schema location to resolve the name space declaration. Note that Liquid Data automatically looks in the repository `schema` directory for the specified file. Otherwise a full path name to `coCptSample2.xsd` is needed.

Notes on Hand-Crafting CPT XQueries

There are two issues to remember when hand crafting an XQuery that accesses a Complex Parameter Type:

- [Unique Namespace](#)
- [XQuery of type element Declaration](#)

Unique Namespace

The namespace of your Complex Parameter Type must be unique. It is a good design pattern to have a namespace defined in your schema file and specified when you define your Complex Parameter Type to Liquid Data. If a namespace is specified in the Complex Parameter Type definition, all XQueries that access the Complex Parameter Type must specify the namespace. Regardless of whether you use namespaces, uniqueness is required.

XQuery of type element Declaration

When you use a Complex Parameter Type in a query, you need to specify a query parameter with the following declaration.

```
of type element [<namespace>:]<root element>
```

The namespace is optional, but the specified root element must be unique. For example, consider the following query:

```
namespace cocpt = "urn:schemas-bea-com:ld-cocpt"
```

```
<cocpt:CustOrder>
{
  for $CO_CPTSAMPLE.CUSTOMER_ORDER_2 in
    (#$QParamForCO-CPTSAMPLE of type element cocpt:CustOrder
     /CUSTOMER_ORDER
    )
  return
    <CUSTOMER_ORDER>
      <CUSTOMER_ID>
{xf:data($CO_CPTSAMPLE.CUSTOMER_ORDER_2/CUSTOMER_ID) }
      </CUSTOMER_ID>
    </CUSTOMER_ORDER>
}
</cocpt:CustOrder>
```

The alias `cocpt` is used in the namespace declaration of this query, and the bold section (which uses the `cocpt` alias) defines the XML input stream for the Complex Parameter Type.

Creating a Complex Parameter Type

This section describes the steps needed to create and run a complex parameter type.

[Step 1. Create a CPT Schema](#)

[Step 2. Create Your Runtime Source](#)

[Step 3. Define Your CPT in the Administration Console](#)

[Step 4. Build Your Query](#)

[Step 5. Run your query](#)

Step 1. Create a CPT Schema

Create a schema that models the runtime source. See the [“Sample CPT Schema” on page 7-4](#) for a small schema example.

Note: In some design situations you may first create a CPT schema and then develop a model for the runtime source. The important point is that there is a tightly coupled relationship between the schema and the runtime data that it models. Both must work together and, once working, the structure of the documents cannot be changed independently.

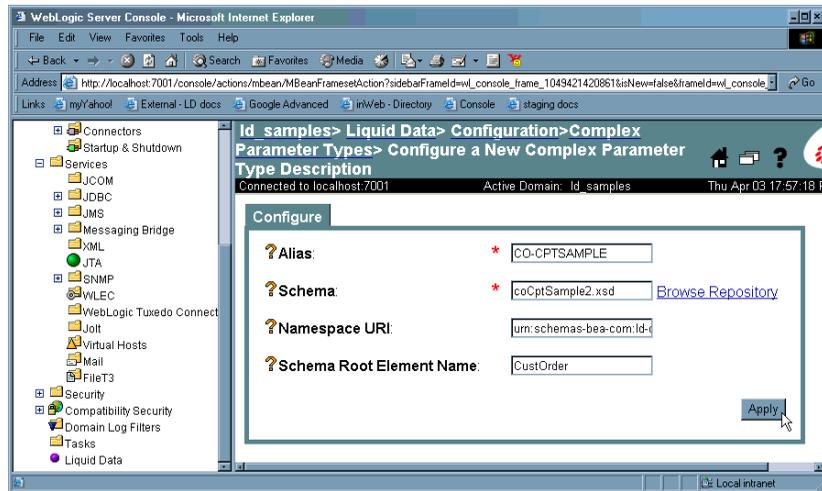
Step 2. Create Your Runtime Source

Create an instance of your runtime source. The runtime source needs to be in XML.

Step 3. Define Your CPT in the Administration Console

Using the Liquid Data Administration Console, define a complex parameter type. for a detailed procedure, see [Configuring Access to Complex Parameter Types in the Administration Guide](#).

Figure 7-1 Creating a Complex Parameter Type in the Administration Console



A valid CPT definition includes an alias identifier and a schema. It is also a good programming practice to provide both a namespace URI and a schema root element name to uniquely identify your CPT.

Step 4. Build Your Query

Create your query either using the Data View Builder or by hand. See “[Key Concepts of Query Building](#)” on page 1-6 for information on designing queries in Liquid Data.

Step 5. Run your query

Once you have created a CPT schema, have a data sample available, and have defined the CPT in the Administration Console, you are ready to use a complex parameter type in a query.

Note: Complex parameter types are not type aware and are always of the type `xs:string` in Liquid Data. You need to cast each element appropriately.

The Liquid Data DB-CPT sample `cptSample.qpr` uses a CPT to supply a promotion plan name for a given state. The sample is installed in the following directory:

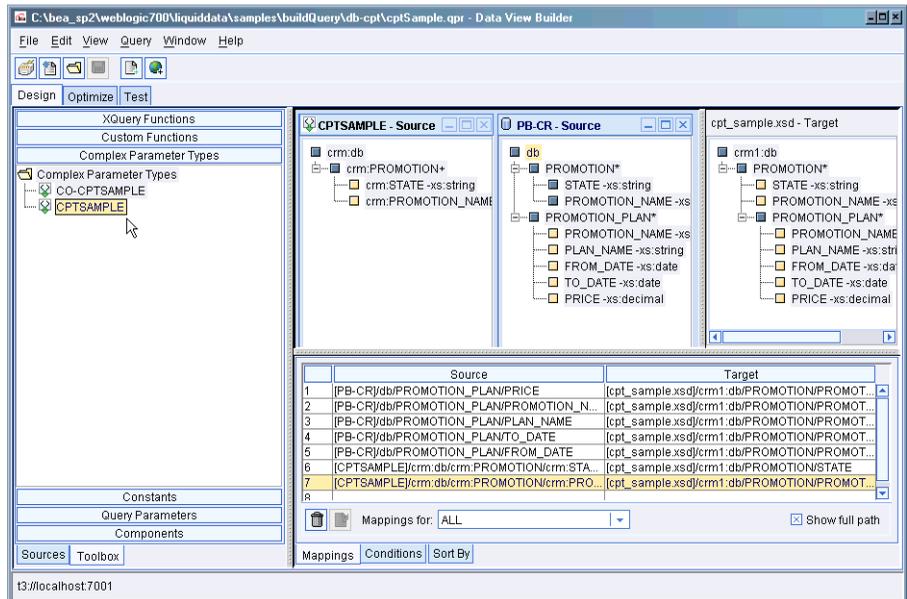
```
BEA_HOME/weblogic700/liquiddata/samples/buildQuery/db-cpt
```

7 Using Complex Parameter Types in Queries

When the query runs details of one or more matching promotion plans names are retrieved from a database.

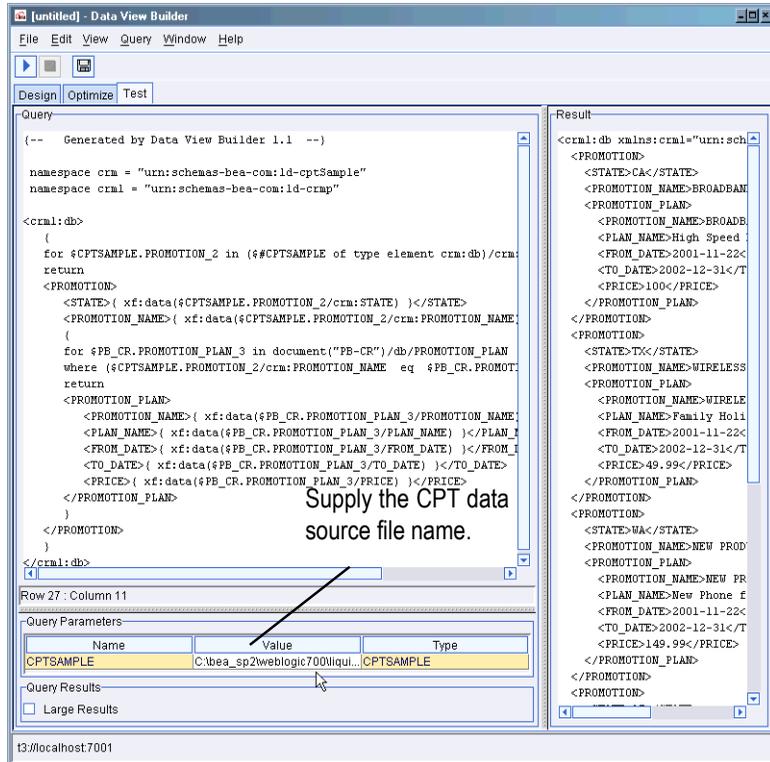
Figure 7-2 shows the DB-CPT project. Notice that there are two complex parameter types available for use. CPTSAMPLE is the complex parameter type used in the query.

Figure 7-2 DB-CPT Project (CPTSAMPLE.QPR) with Complex Parameter Types Displayed



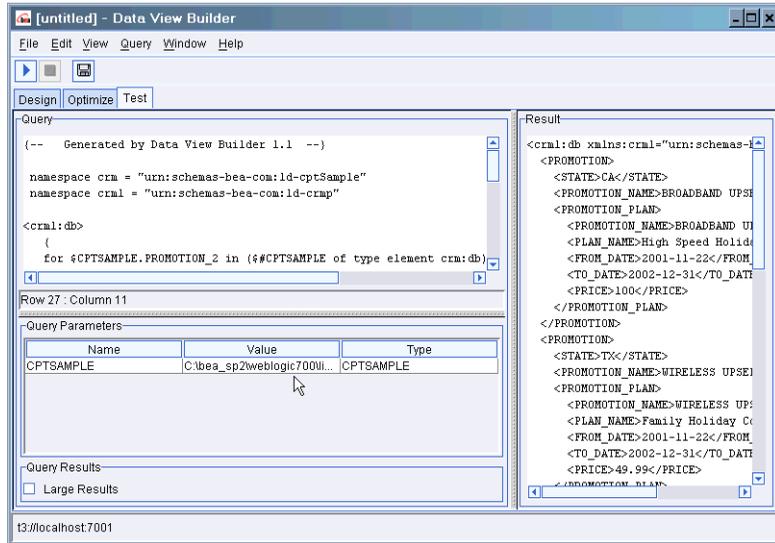
To test your query the name of an XML data file that is modeled on the CPT schema must be entered in the Data View Builder (see [Figure 7-3](#)). For Liquid Data sample XML click on the Value field to open the Liquid Data file browser to the `LDRepository/XML_files` directory.

Figure 7-3 DB-CPT Project in Test Mode



[Figure 7-4](#) shows the DB-CPT (`CPTSAMPLE.QPR`) project when the query is run.

Figure 7-4 DB-CPT Project (CPTSAMPLE.QPR) in Run Mode



When you are satisfied with your ability to run your query using runtime data source, the query can be invoked through an EJB API. For details on invoking queries programmatically and the Liquid Data API see:

- [Setting Complex Parameter Types](#) in *Invoking Liquid Data Queries Programmatically*.
- [Invoking Queries in EJB Clients](#) in *Invoking Liquid Data Queries Programmatically*.
- [Liquid Data 1.1 API Reference](#) (Javadoc).

There is also a Liquid Data EJB API sample in:

`BEA_HOME/weblogic700/liquiddata/samples/ejbAPI`

Complex Parameter Type Query Samples

For a step by step example of building a query with a CPT, see [“Example 6: Complex Parameter Type \(CPT\)”](#) on page 9-41.

There are also two CPT query samples installed with the Liquid Data samples. These samples show how to create a complex parameter type, configure it as a complex parameter type, and then run the query.

Instructions for running the samples are provided in readme files located at:

- `LDHome/samples/buildQuery/db-cpt/readme.htm`
- `LDHome/samples/buildQuery/db-cptco/readme.htm`

Also see the Liquid Data [Samples page](#) for information on other available query samples.

8 Defining Stored Procedures

If you have stored procedures defined in your databases, you can expose them to Liquid Data as a data source and use them in your Liquid Data queries.

- [Defining Stored Procedures to Liquid Data](#)
- [Stored Procedure Description File Schema](#)
- [Rules for Specifying Stored Procedure Description Files](#)
- [Sample Stored Procedure Description Files](#)
- [Stored Procedure Support by Database](#)
- [Using Stored Procedures in Queries](#)

For an example and a demo of defining a stored procedure and using it in a query, see “[Example: Defining and Using a Customer Orders Stored Procedure](#)” on page 8-33.

Defining Stored Procedures to Liquid Data

To use stored procedures in Liquid Data, you must create a Stored Procedure Description file. The Stored Procedure Description file is an XML schema file that defines the types and the functions for a set of stored procedures. For details on defining a Stored Procedure Description file, see [“Stored Procedure Description File Schema” on page 8-4](#) and [“Rules for Specifying Stored Procedure Description Files” on page 8-10](#). For database-specific information, see [“Stored Procedure Support by Database” on page 8-26](#).

To Define Stored Procedures to Liquid Data

Perform the following steps to define a stored procedure for use with Liquid Data.

1. Create your stored procedures in the underlying database, if they do not already exist. For details about Liquid Data support of stored procedures for your database, see [“Stored Procedure Support by Database” on page 8-26](#).
2. In the WebLogic Console, create a JDBC Connection Pool to access your database, if one does not already exist.
3. In the WebLogic Console, create a JDBC Data Source for the connection pool created in the previous step.
4. Create a Stored Procedure Description file for your stored procedures and save it to the `stored_procedures` directory of the Liquid Data repository. For details, see [“Stored Procedure Description File Schema” on page 8-4](#) and [“Rules for Specifying Stored Procedure Description Files” on page 8-10](#).
5. In the Liquid Data Administration Console (to access the Liquid Data Console, click the Liquid Data link at the bottom of the list on the WebLogic administration console), click the Data Sources tab.
6. Click the Relational Databases tab.
7. Click the Configure a New Relational Data Source Description Link (or open an existing Data Source to modify it).

8. If you are creating a new data source, enter values for Name, Data Source Name, and Schema fields in the Configure Relational Data Source Description screen. For more details on configuring relational data sources, see [Configuring Access to Relational Databases](#) in the *Administration Guide*.
9. In the Configure Relational Data Source Description screen, specify a Stored Procedure Description file by clicking the Browse Repository link next to the Stored Procedure Description File field.
10. In the Repository Browser, select the file you created containing your stored procedure definitions. After making your selection, click the Select button.

 **Stored Procedures Description File:** [Browse Repository](#)

11. In the Configure Relational Data Source Description screen, click the Apply button to save your Data Source definition.
12. Check the WebLogic Server log file for any errors, and correct them as necessary.

You can now access your stored procedures in the Data View Builder. If you are already connected to the server in the Data View Builder, you must re-connect by selecting File —> Connect from Data View Builder menu. The Stored Procedures tab appears in the Design view under the sources tab of the Data View Builder. You can now use your stored procedures as you do other building blocks (for example, data sources, XQuery functions, and so on) to build queries.

Stored Procedure Description File Schema

The Stored Procedure Description file is an XML file that defines stored procedures to Liquid Data. This section describes the schema of the Stored Procedure Description file, and contains the following sections:

- [Basic Structure](#)
- [Schema Definition File for Stored Procedure Description File](#)
- [Element and Attribute Reference for Stored Procedure Description File](#)
- [Supported Datatypes](#)

For sample Stored Procedure Description files, see “[Sample Stored Procedure Description Files](#)” on page 8-19.

Basic Structure

The Stored Procedure Description file has the following main sections:

- [Type Definitions](#)
- [Function Definitions](#)

Type Definitions

The type definitions section of the Stored Procedure Description file is defined in the `<types>` element. This element defines namespaces and complex types for the stored procedures defined in the Stored Procedure Description file.

Function Definitions

The function definitions section of the Stored Procedure Description file is defined in the `<functions>` element. Within the `<functions>` element are `<function>` elements, each of which defines the signature of a stored procedure. You can define one or more stored procedures in a single Stored Procedure Description file.

Schema Definition File for Stored Procedure Description File

The following is the schema definition file for the Stored Procedure Description file:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xsd:element name="definitions">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="types"/>
        <xsd:element ref="functions"/>
      </xsd:sequence>
      <xsd:attribute name="targetNamespace" use="optional"
type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="types">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="schema"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="schema" type="xsd:anyType"/>
  <xsd:element name="functions">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="function" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="function">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="argument" minOccurs="0"
maxOccurs="unbounded"/>
        <xsd:element ref="presentation" minOccurs="0"/>
        <xsd:element ref="description" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="name" use="required"
type="xsd:string"/>
      <xsd:attribute name="return_type" use="required"
type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```

</xsd:element>
<xsd:element name="argument">
  <xsd:complexType>
    <xsd:attribute name="type" use="required"
      type="xsd:string"/>
    <xsd:attribute name="label" use="required"
      type="xsd:string"/>
    <xsd:attribute name="mode" use="required" type="modeType"/>
  </xsd:complexType>
</xsd:element>
<xsd:simpleType name="modeType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="input_only"/>
    <xsd:enumeration value="output_only"/>
    <xsd:enumeration value="input_output"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:element name="presentation">
  <xsd:complexType>
    <xsd:attribute name="group" use="required"
      type="xsd:string"/>
  </xsd:complexType>
</xsd:element>
<xsd:element name="description" type="xsd:string"/>
</xsd:schema>

```

Element and Attribute Reference for Stored Procedure Description File

Table 8-1 lists and describes the elements and attributes of the Stored Procedure Description file.

Table 8-1 Stored Procedure Description file XML elements and descriptions

Element	Attribute	Description
<definitions>	targetNamespace	The namespace declared for the stored procedures.
<types>		Declares any primitive or complex data types used in the stored procedure.

Table 8-1 Stored Procedure Description file XML elements and descriptions

Element	Attribute	Description
<functions>		Contains the function definitions for all of the stored procedures represented in this file.
<function>		Function definition for a single stored procedure.
	name	Name of the stored procedure as defined in the database. If the stored procedure is part of a package, the name is the fully qualified name of the stored procedure (for example, <code>packagename.sp_name</code>). If you are using procedure groups in Sybase or Microsoft SQL Server, see “Rules for Procedure Names Containing a Semi-Colon” on page 8-12.
	return_type	Return type of the stored procedure. The type is defined in the <code><types></code> element of this file. Note that this type differs from the type which the stored procedure returns. If you are hand-crafting your own XQueries, you must perform a function signature transformation; for details, see “Rules for Transforming the Function Signature When Hand Writing an XQuery” on page 8-16.

Table 8-1 Stored Procedure Description file XML elements and descriptions

Element	Attribute	Description
<argument>		Contains the argument declarations for the inputs and/or outputs of the stored procedure.
	label	The name of the argument input or output. This name is used in queries and is displayed in clients such as the Data View Builder.
	type	Type of the argument. The type can be one of the types listed in “Supported Datatypes” on page 8-9, or it can be a complex type declared in the Stored Procedure Description file.
	mode	Lists whether the argument is part of the input, output, or both. Possible values are: <ul style="list-style-type: none"> ■ input_only ■ output_only ■ input_output
<presentation group>		Currently not supported.
<description>		Comment text describing the stored procedures used in the Stored Procedure Description file.

Note: An element within the `types` definition with a name specified with `name="return_value"` is reserved to specify the return value from a function or a procedure. For an example, see [“Example 2: Type Definition with Simple Return Value”](#) on page 8-14.

Supported Datatypes

The stored procedures you define in the Stored Procedure Description files must use the XML data types shown in [Table 8-2](#). You must map the database data types to one of the types in this table. For datatype support by database, see [“Stored Procedure Support by Database”](#) on page 8-26.

Except for user-defined complex data types, the types are all primitive data types.

Table 8-2 XML data types and their Java equivalents for Stored Procedure Description files

XML Data Type	Equivalent Java Data Type
xs:boolean	java.lang.boolean
xs:byte	java.lang.byte
xs:short	java.lang.short
xs:integer	java.lang.Integer
xs:int	java.lang.Integer
xs:long	java.lang.Long
xs:float	java.lang.float
xs:double	java.lang.double
xs:decimal	java.math.BigDecimal
xs:string	java.lang.String
xs:dateTime	java.util.Date
Complex Element Type	org.w3c.dom.Element

For JDBC and database-specific type mapping, see [Appendix B, “Supported Data Types.”](#)

Rules for Specifying Stored Procedure Description Files

This section describes rules for the `return_type` attribute of the `<function>` element and the `mode` attribute of the `<argument>` element. This section includes the following rules:

- [Rules for Element and Attribute Names](#)
- [Rules for Procedure Names Containing a Semi-Colon](#)
- [Rules and Examples of `<type>` Declarations to Use in the `<function>` `return_type` Attribute](#)
- [Rules for the `mode` Attribute `output_only` `<argument>` Element](#)
- [Rules for Transforming the Function Signature When Hand Writing an XQuery](#)

Rules for Element and Attribute Names

XML requires that element and attribute names begin with a non-numeric character. Therefore, when you specify the `name` attribute of the `<xs:element>` element in a Stored Procedure Description file, you must specify a name that does not start with a numeric character. For example, if you have a stored procedure that returns a cursor, and the cursor returns columns that start with a numeric character, you must map those column names to valid XML element names in your Stored Procedure Description file.

For the W3C definition of a valid name for an attribute or element, see:

<http://www.w3.org/TR/2000/WD-xml-2e-20000814#dt-name>

For example, consider a cursor named `MY_CURSOR` is declared with the following SQL statement:

```
open MY_CURSOR for
  select 1_column, 2_column, 3_column
  from MY_TABLE
return MY_CURSOR
```

When you define the cursor type in the `<types>` section of your Stored Procedure Description file, you must map the column names from the cursor output to start with a non-numeric character so the resulting XML generated is valid. You could use the following type definition for this cursor, which starts each numeric column name with an underscore character (`_`):

```
<types>
  <xs:element name="MY_CURSOR" minOccurs="0"
    maxOccurs="unbounded">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="_1_column" type="xs:integer"/>
        <xs:element name="_2_column" type="xs:string"/>
        <xs:element name="_3_column" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</types>
```

For some notes on relational database object names and how to specify them so they can be used in an XQuery, see [“Relational Databases” on page 1-8](#).

Rules for Procedure Names Containing a Semi-Colon

Sybase and Microsoft SQL Server databases provide the ability to group stored procedures by using a semi-colon character (;) to separate a procedure name with a number. For example, you can have two stored procedures with the following names:

```
MY_SP;1  
MY_SP;2
```

When you specify these procedures in the Stored Procedure Description file, use the database name (the name with the semi-colon character). When you use these procedure names in an XQuery, however, you must substitute an underscore character () for the semi-colon character. The Data View Builder automatically substitutes the underscore character for the semi-colon character in the XQuery it generates.

For example, consider the following definition for a stored procedure in a Stored Procedure Description file:

```
<function name="MY_SP;2" return_type="Results">  
  <argument label="COLUMN_123" mode="input_only"  
    type="xs:string"/>  
  <argument label="ANOTHER_COLUMN" mode="output_only"  
    type="xs:int"/>  
</function>
```

When you reference this function in an XQuery, it is referred to as follows:

```
MY_SP_2
```

Rules and Examples of `<type>` Declarations to Use in the `<function>` `return_type` Attribute

The `return_type` attribute of the `<function>` element specifies the complex type for the stored procedure. The complex type must be declared in the `<types>` section of the Stored Procedure Description file. For example, the following element opening tag shows a function named `myFunction` with a `return_type` of `myReturnType`:

```
<function name="myFunction" return_type="myReturnType" >
```

The return type `myReturnType` must be declared in the `<types>` section. The type must contain the actual return value of the stored procedure (if it has a return value) and the row set definitions (if applicable). The row set definitions must appear in the order in which the stored procedure returns them.

When a stored procedure returns a primitive type, you must declare the primitive type using the `return_value` keyword for the `name` attribute. For an example of this, see [“Example 2: Type Definition with Simple Return Value”](#) on page 8-14.

Example 1: Type Definition with No Return Value

The following is a type definition for a stored procedure that has no return value and returns no row sets.

```
<types>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="EmptyOutput">
      <xs:complexType>
        <xs:sequence>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </types>
```

Use a similar type definition in a stored procedure that does not have any return value. This `EmptyOutput` type definition is required for all stored procedures and functions that do not return anything.

Example 2: Type Definition with Simple Return Value

The following is a type definition for a stored procedure that has a simple return value (`xs:integer`) and returns no row sets.

```
<types>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="SimpleOutput">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="return_value"
            type="xs:integer" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>
</types>
```

Use a similar type definition with a stored procedure that returns a status code or a single value.

Example 3: Type Definition for Complex Row Set Type

The following is a type definition for a stored procedure that returns a row set, which is a complex type.

```
<types>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="customerTable">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="CUSTOMER" minOccurs="0"
            maxOccurs="unbounded" >
            <xs:complexType>
              <xs:sequence>
                <xs:element name="C_NAME"
                  type="xs:string"/>
                <xs:element name="C_ACCTBAL"
                  type="xs:decimal"/>
              </xs:sequence>
            </xs:complexType>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </types>
```

Use a similar type definition with a stored procedure that returns a result set (for example, in Sybase, Microsoft SQL Server, or DB2).

Example 4: Type Definition with Complex Return Value

The following is a type definition for a stored procedure that returns a complex type. Assume that the `customerTable` complex type (shown in the previous example) is defined in the same Stored Procedure Definition file.

```
<types>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="return_value" type="customerTable">
    </xs:element>
  </xs:schema>
</types>
```

Use a similar type definition with an Oracle stored procedure that returns a cursor with a complex type.

Example 5: Type Definition with Simple Return Value and Two Row Sets

The following is a type definition for a stored procedure that has a simple return value (`xs:integer`) and returns two row sets.

```
<types>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
.....
.....{-- Definitions for complex types customerTable and
      ordersTable go here --}
.....
    <xs:element name="OutputName">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="return_value"
            type="xs:integer" />
          <xs:element ref="customerTable" />
            {-- customerTable defined above--}
          <xs:element ref="ordersTable" />
            {-- ordersTable defined above --}
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>
</types>
```

Rules for the mode `Attribute output_only` `<argument>` Element

If you define a function that has an `<argument>` element that has the mode `output_only`, then you need only reference the type definition in the function definition. The following example references the `customerTable` type (defined in the `<types>` section of the Stored Procedure Description file, as described in “[Example 3: Type Definition for Complex Row Set Type](#)” on page 8-14). Assume that the `customerTable` type maps to a cursor returned from an Oracle stored procedure.

```
<function name="GetAllCustomersByState" return_type="EmptyOutput">
  <argument label="state" mode="input_only" type="xs:string"/>
  <argument label="CustomersOutput" mode="output_only"
    type="customerTable"/>
</function>
```

Rules for Transforming the Function Signature When Hand Writing an XQuery

There are two issues to remember when hand crafting an XQuery that accesses a stored procedure:

- [Namespace Declaration](#)
- [Function Transformation](#)

Namespace Declaration

All queries that access stored procedures must have a unique namespace with a URI of the of the following form:

```
urn:<Liquid_Data_Relational_data_source_name>
```

Declare the namespace in the query prolog. The namespace declaration has the following syntax:

```
namespace <alias>="<URI>"
```

For example:

```
namespace SY_WL_NS="urn:SY-WL"
```

You can then access the stored procedure using the namespace alias and the name of the stored procedure object. For example:

```
SY_WL_NS:wireless.dbo.RetAndOpParamTransformation("CUSTOMER_11")
```

Function Transformation

For stored procedures that return both a return value (for example, an integer return value) and have `output` or `input_output` parameters, the function signature in the Stored Procedure Description file is different from the signature that is used to write queries that access the stored procedure. If you look at the schema that displays in the the Stored Procedures palette of the Data View Builder, you will see the transformed signature.

The transformed signature combines the return value and any `output` or `input_output` parameters.

For example, consider an example using a Sybase stored procedure with the following signature:

```
create proc RetAndOpParamTransformation (@custidPattern
varchar(64), @custCount numeric(10) output )
as
    select @custCount = count(*) from CUSTOMER
    where
        CUSTOMER.CUSTOMER_ID Like '%' + @custidPattern + '%'
RETURN 1
```

The following is the Stored Procedure Description file for this stored procedure:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions>
  <types>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<!-- The stored procedure returns an integer, which is mapped as
the return_value, a reserved element name for stored procedure with
a return.
-->
      <xs:element name="RetAndOpParamTransformation">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="return_value"
              type="xs:integer"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </types>
</definitions>
```

```
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>
</types>

<!-- The stored procedure signature mapping. The element
RetAndOpParamTransformation wraps the return_value of the stored
procedure. This stored procedure has custidPattern as an input
parameter. custCount is defined as an output parameter of type
integer (because it returns an integer count).
-->
  <functions>
    <function name="wireless.dbo.RetAndOpParamTransformation"
      return_type="RetAndOpParamTransformation">
      <argument label="custidPattern" mode="input_only"
        type="xs:string"/>
      <argument label="custCount" mode="output_only"
        type="xs:integer"/>
      <presentation group="Sample to show transformation of
return_value and output prameter in a stored procedure"/>
    </function>
  </functions>
</definitions>
```

Because this stored procedure has a return value and an output parameter, the output of the function is transformed to the following schema:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="RetAndOpParamTransformation">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="return_value"
          type="xsd:integer"/>
        <xsd:element name="custCount" type="xsd:integer"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

where the `return_value` is the return from the stored procedure and `custCount` is the output parameter. If you view this stored procedure in the Data View Builder, you will see the transformed schema.

The following is a query against this stored procedure:

```
namespace SY-WL-NS = "urn:SY-WL"

let $SY_WL_SP_Return :=
SY-WL-NS:wireless.dbo.RetAndOpParamTransformation("CUSTOMER_11")
return

<RetAndOpParamTransformation>
  <return_value>{
xf:data($SY_WL_SP_Return/RetAndOpParamTransformation/return_value
) }
  </return_value>
  <custCount>{
xf:data($SY_WL_SP_Return/RetAndOpParamTransformation/custCount) }
  </custCount>
</RetAndOpParamTransformation>
```

In this query, the XPath expressions for `return_value` and for `custCount` have the same parent element.

Sample Stored Procedure Description Files

This section shows several sample Stored Procedure Description files. For simplicity and readability, each of the examples shown defines a single stored procedure and its supporting type; your Stored Procedure Description files can define multiple stored procedures and multiple types. This section includes the following examples:

- [DB2 Simple input_only, output_only, and input_output Example](#)
- [Oracle Cursor Output Parameter Example](#)
- [DB2 Multiple Result Set Example](#)
- [Oracle Cursor as return_value](#)

DB2 Simple `input_only`, `output_only`, and `input_output` Example

The following Stored Procedure Description file describes a DB2 stored procedure that returns simple data types with `input_only`, `output_only`, and `input_output` parameters.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions>
  <types>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name="EmptyOutput">
        <xs:complexType>
          <xs:sequence>
            </xs:sequence>
          </xs:complexType>
        </xs:schema>
      </types>
    <functions>
      <!-- given a customer id if valid, returns as output
           parameters all the customer details -->
      <function name="CALLINCALLOUT" return_type="EmptyOutput">
        <argument label="custid" mode="input_only" type="xs:string"/>
        <argument label="fname" mode="output_only" type="xs:string"/>
        <argument label="lname" mode="output_only" type="xs:string"/>
        <argument label="telephoneNumber" mode="output_only"
          type="xs:long"/>
        <argument label="customerSinceAsData" mode="output_only"
          type="xs:date"/>
        <argument label="customerSinceAsTimeStamp"
          mode="output_only" type="xs:dateTime"/>
        <presentation group="DB2 stored procedures"/>
      </function>
    </functions>
  </definitions>
```

The following is the stored procedure signature for this Stored Procedure Description file. This signature creates a procedure that has one simple input and returns five simple outputs.

```
CREATE PROCEDURE DB2ADMIN.CALLINCALLOUT (
  IN CUSTID varchar(64), OUT FNAME varchar(4000),
  OUT LNAME varchar(4000), OUT TELEPHONENUMBER bigint,
  OUT CUSTOMERSINCE date, OUT CUSTOMERSINCE1 timestamp )
EXTERNAL NAME
```

```
'DB2ADMIN'.SQL30205005750980:db2test.CallInCallOut.callInCallOut'  
SPECIFIC DB2ADMIN.CALLINCALLOUT  
RESULT SETS 0  
LANGUAGE JAVA  
PARAMETER STYLE JAVA  
NOT DETERMINISTIC  
FENCED NO  
DBINFO NULL  
CALL MODIFIES SQL DATA
```

Oracle Cursor Output Parameter Example

The following Stored Procedure Description file describes an Oracle stored procedure that returns an output parameter as a cursor.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions>
  <types>
    <xs:element name="OutCursor">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="CUSTOMER" minOccurs="0"
            maxOccurs="unbounded">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="C_CUSTKEY" type="xs:integer"/>
                <xs:element name="C_FNAME" type="xs:string"/>
                <xs:element name="C_LNAME" type="xs:string"/>
                <xs:element name="C_STATE" type="xs:string"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="Output">
      <xs:complexType>
        <xs:sequence/>
      </xs:complexType>
    </xs:element>
  </types>
  <functions>
    <function name="TEST_PACKAGE.GETCUSTOMER" return_type="Output">
      <argument label="CUSTID" mode="input_only" type="xs:string"/>
      <argument label="customer_OUT" mode="output_only"
        type="OutCursor"/>
      <presentation group="OR-TEST stored procedures"/>
    </function>
  </functions>
</definitions>
```

The following is the stored procedure signature for this Stored Procedure Description file. This signature creates a procedure that returns a cursor as an output parameter.

```
create or replace package test_package as
-- Stored procedure that returns a cursor as an output parameter
  procedure getCustomer
    ( CUSTOMERID IN VARCHAR, cust_cursor1 OUT ref_cursor );
end test_package ;
```

DB2 Multiple Result Set Example

The following Stored Procedure Description file describes a DB2 stored procedure that returns multiple result sets.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions>
  <types>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name="CustomerAndOrders">
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="resultSetCustomer"/>
            <xs:element ref="resultSetCustomerOrders"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="resultSetCustomer">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="customerRow" minOccurs="1"
              maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="C_CUSTKEY" type="xs:string"/>
                    <xs:element name="C_FNAME" type="xs:string"/>
                    <xs:element name="C_LNAME" type="xs:string"/>
                    <xs:element name="C_STATE" type="xs:string"/>
                    <xs:element name="C_SINCE" type="xs:date"/>
                    <xs:element name="C_TELEPHONENO" type="xs:long"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:element>
      <xs:element name="resultSetCustomerOrders">
        <xs:complexType>
```

```
<xs:sequence>
<xs:element name="orderRow" minOccurs="1"
            maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="C_CUSTKEY" type="xs:string"/>
      <xs:element name="CO_ORDERKEY" type="xs:string"/>
      <xs:element name="CO_ORDERDATE" type="xs:date"/>
      <xs:element name="CO_SHIPMETHOD" type="xs:date"/>
      <xs:element name="CO_TOTALORDERAMT" type="xs:decimal"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
</types>
<functions>
  <!-- result one returns a customer, result 2 has the
        orders for that customer -->
  <function name="CALLMULTIPLERESULTSET"
            return_type="CustomerAndOrders">
    <argument label="custid" mode="input_only" type="xs:string"/>
    <presentation group="DB2 stored procedures"/>
  </function>
</functions>
</definitions>
```

The following is the stored procedure signature for this Stored Procedure Description file. This signature creates a procedure that returns multiple result sets.

```
CREATE PROCEDURE DB2ADMIN.CALLMULTIPLERESULTSET (
  IN CUSTID varchar(64) )
EXTERNAL NAME
  'DB2ADMIN'.SQL30206110348560:db2test.CallMultipleResultSet.
callMultipleResultSet'
SPECIFIC DB2ADMIN.CALLMULTIPLERS
RESULT SETS 2
LANGUAGE JAVA
PARAMETER STYLE JAVA
NOT DETERMINISTIC
FENCED NO
DBINFO NULL
CALL MODIFIES SQL DATA
```

Oracle Cursor as return_value

The following Stored Procedure Description file describes an Oracle stored procedure that returns a cursor as a return_value.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions>
  <types>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name="Output_TEST_PACKAGE.GETCUSTOMERBYID">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="return_value">
              <xs:complexType>
                <xs:sequence>
                  <xs:element maxOccurs="unbounded" minOccurs="0"
                    name="customer">
                    <xs:complexType>
                      <xs:sequence>
                        <xs:element name="FIRST_NAME" type="xs:string"/>
                        <xs:element name="LAST_NAME" type="xs:string"/>
                        <xs:element name="CUSTOMER_ID" type="xs:string"/>
                        <xs:element name="STATE" type="xs:string"/>
                        <xs:element name="ZIPCODE" type="xs:string"/>
                        <xs:element name="CITY" type="xs:string"/>
                        <xs:element name="STREET_ADDR2"
                          type="xs:string"/>
                        <xs:element name="STREET_ADDR1"
                          type="xs:string"/>
                        <xs:element name="CUSTOMER_SINCE"
                          type="xs:dateTime"/>
                        <xs:element name="EMAIL_ADDRESS"
                          type="xs:string"/>
                        <xs:element name="TELEPHONE_NUMBER"
                          type="xs:string"/>
                      </xs:sequence>
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </types>
```

```
<functions>
  <function name="TEST_PACKAGE.GETCUSTOMERBYID"
    return_type="Output_TEST_PACKAGE.GETCUSTOMERBYID">
    <argument label="CUSTID" mode="input_only" type="xs:string"/>
  </function>
</functions>
</definitions>
```

The following is the stored procedure signature for this Stored Procedure Description file. This signature creates a procedure that returns a cursor.

```
create or replace package body test_package as
-- SP that returns a cursor
  FUNCTION getCustomerByID (custID varchar)
    RETURN CUST_CURSOR IS cur CUST_CURSOR;
  BEGIN
    open cur for
      select first_name, last_name, customer_id, state,
             zipCode,city, street_address2, street_address1,
             customer_since, email_address, telephone_number
      from wireless.customer
      where customer_id = custID;
    return cur;
  END;
end test_package ;
```

Stored Procedure Support by Database

This section lists stored procedure support by database vendor. Each vendor supports the datatypes supported in their respective databases. The following databases are supported:

- [Oracle](#)
- [Microsoft SQL Server](#)
- [Sybase](#)
- [IBM DB2](#)
- [Informix](#)

Oracle

[Table 8-3](#) describes the stored procedure support for Oracle databases. [Table 8-4](#) describes Oracle stored procedures return values.

Table 8-3 Oracle Stored Procedure parameter support

Parameter Mode	Data Types Supported	Notes and Restrictions
input_only	Only database data types that you can map to one of the Liquid Data primitive types defined in “Supported Datatypes” on page 8-9 .	<ul style="list-style-type: none"> ■ The PL/SQL %TYPE definitions must be translated to the XML schema types defined in “Supported Datatypes” on page 8-9.
output_only	<ul style="list-style-type: none"> ■ A Cursor ■ Only database data types that you can map to one of the Liquid Data primitive types defined in “Supported Datatypes” on page 8-9. 	
input_output	Only database data types that you can map to one of the Liquid Data primitive types defined in “Supported Datatypes” on page 8-9 .	

Table 8-4 Oracle Stored Procedure returned values support

Return Value	Types Supported
Primitive type	An primitive type such as an integer, a string, etc.
Return cursor	See Table 8-3 .

Microsoft SQL Server

Table 8-5 describes the stored procedure support for Microsoft SQL Server databases. Table 8-6 describes Microsoft SQL Server stored procedures return values.

Table 8-5 Microsoft SQL Server Stored Procedure parameter support

Parameter Mode	Data Types Supported	Notes and Restrictions
input_only	Only database data types that you can map to one of the Liquid Data primitive types defined in “Supported Datatypes” on page 8-9.	
output_only	Only database data types that you can map to one of the Liquid Data primitive types defined in “Supported Datatypes” on page 8-9.	<ul style="list-style-type: none"> You must map TINYINT values to xs:short in the Stored Procedure Description File.

Table 8-6 Microsoft SQL Server Stored Procedure returned values support

Return Value	Types Supported	Notes and Restrictions
Return Status code	An integer value.	
Row Set	Single or multiple result sets.	<ul style="list-style-type: none"> You must map TINYINT values to xs:short in the Stored Procedure Description File.

If you are using procedure groups, see “Rules for Procedure Names Containing a Semi-Colon” on page 8-12 for information on mapping the procedure names to the Stored Procedure Description file and using the names in an XQuery.

Microsoft SQL Server parameter names begin with the @ character, but the name must appear in the Stored Procedure Description file without the @ character. For example, a parameter named @myInputParameter must be mapped as myInputParameter.

Sybase

[Table 8-7](#) describes the stored procedure support for Sybase databases. [Table 8-8](#) describes Sybase stored procedures return values.

Table 8-7 Sybase Stored Procedure parameter support

Parameter Mode	Data Types Supported	Notes and Restrictions
input_only	Only database data types that you can map to one of the Liquid Data primitive types defined in “Supported Datatypes” on page 8-9 .	
output_only	Only database data types that you can map to one of the Liquid Data primitive types defined in “Supported Datatypes” on page 8-9 .	<ul style="list-style-type: none"> You must map TINYINT values to xs:short in the Stored Procedure Description File.

Table 8-8 Sybase Stored Procedure returned values support

Return Value	Types Supported	Notes and Restrictions
Return Status code	An integer value.	
Row Set	Single or multiple result sets.	<ul style="list-style-type: none"> You must map TINYINT values to xs:short in the Stored Procedure Description File.

If you are using procedure groups, see [“Rules for Procedure Names Containing a Semi-Colon” on page 8-12](#) for information on mapping the procedure names to the Stored Procedure Description file and using the names in an XQuery.

Sybase parameter names begin with the @ character, but the name must appear in the Stored Procedure Description file without the @ character. For example, a parameter named @myInputParameter must be mapped as myInputParameter.

IBM DB2

[Table 8-9](#) describes the stored procedure support for IBM DB2 databases. [Table 8-10](#) describes IBM DB2 stored procedures return values.

Table 8-9 IBM DB2 Stored Procedure parameter support

Parameter Mode	Data Types Supported
input_only	Only database data types that you can map to one of the Liquid Data primitive types defined in “Supported Datatypes” on page 8-9.
output_only	Only database data types that you can map to one of the Liquid Data primitive types defined in “Supported Datatypes” on page 8-9.
input_output	Only database data types that you can map to one of the Liquid Data primitive types defined in “Supported Datatypes” on page 8-9.

Table 8-10 IBM DB2 Stored Procedure returned values support

Return Value	Types Supported
Primitive type	An primitive type such as an integer, a string, etc.
Row Set	Single or multiple result sets.

Informix

[Table 8-11](#) describes the stored procedure support for Informix databases. [Table 8-12](#) describes Informix stored procedures return values.

Table 8-11 Informix Stored Procedure parameter support

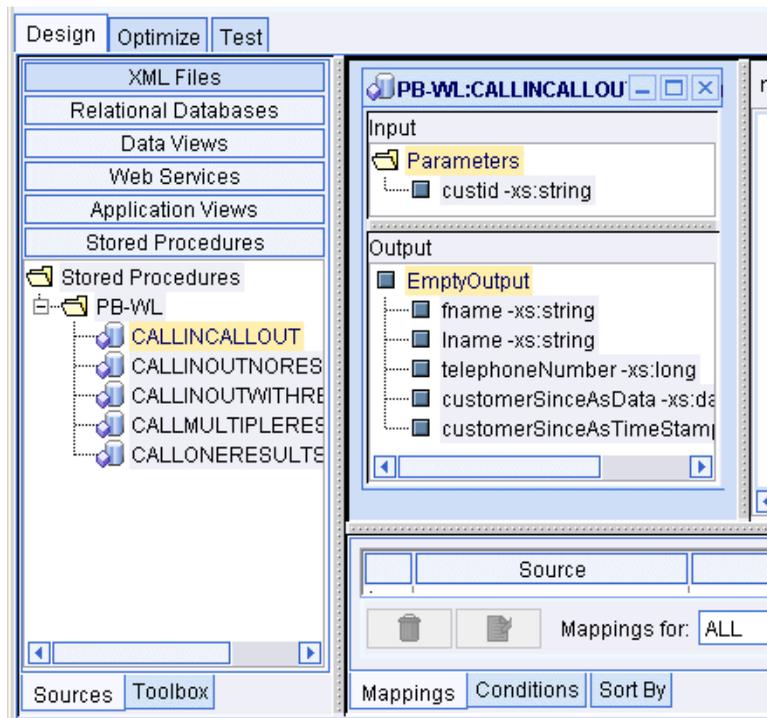
Parameter Mode	Data Types Supported
input_only	Only database data types that you can map to one of the Liquid Data primitive types defined in “Supported Datatypes” on page 8-9.

Table 8-12 Informix Stored Procedure returned values support

Return Value	Types Supported
Row Set	Single or multiple result sets.

Using Stored Procedures in Queries

You can use stored procedures to build queries in the Data View Builder just like you use other data sources. Drag and drop input elements into the inputs of the procedure and drag and drop output elements to combine with other sources or to map onto a target schema.



This section shows an example of defining a stored procedure and then using it in a query, and is divided into the following sections:

- [Define Stored Procedures to Liquid Data](#)
- [Example: Defining and Using a Customer Orders Stored Procedure](#)

Define Stored Procedures to Liquid Data

You must define the stored procedures to Liquid Data before you can use them in queries. For details, see [“To Define Stored Procedures to Liquid Data” on page 8-2](#). To use a stored procedure in the Data View Builder, select Stored Procedures from the Sources tab, navigate to your stored procedure, then drag and drop it into the design workspace. You can then connect data by dragging and dropping inputs and outputs.

Example: Defining and Using a Customer Orders Stored Procedure

This example details the steps to define a stored procedure to Liquid Data and then use it in a query. This example is similar to the example installed in the following directory:

```
BEA_HOME/liquiddata/samples/buildQuery/stored-procedure
```

The demo in this directory includes the Stored Procedure Description file and a Data View Builder project file.

Business Scenario

The stored procedure in this example answers the following business question: For all orders greater than or equal to \$500.00, find the number of orders and the total value of all of those orders for a given customer.

View a Demo

Stored Procedure Demo... If you are looking at this documentation online, you can click the “Demo” button to see a viewlet demo showing how to define a stored procedure and use it in a query. This demo previews the steps described in detail in the following sections.

Step 1: Create the Stored Procedure in the Database

You must have stored procedures defined in your database before you can access them through Liquid Data.

Every database has its own way of creating stored procedures. This sample uses a Pointbase database, and Pointbase uses Java stored procedures. The source code for the sample stored procedure is installed with Liquid Data in the following file:

```
BEA_HOME/liquiddata/samples/buildQuery/stored-procedure/pbsp.java
```

The signature for this stored procedure is created with the following SQL statements:

```
create procedure
    GetOrderInfo( IN P1 VARCHAR(20), IN P2 INTEGER,
                 OUT P3 INTEGER, OUT P4 INTEGER)
LANGUAGE JAVA
SPECIFIC GetOrderInfo
EXTERNAL NAME "com.bea.lidi.sample.pbsp::GetOrderInfo"
PARAMETER STYLE SQL;
```

Step 2: Create the Stored Procedure Description File

For details on the structure of the Stored Procedure Description file, see [“Stored Procedure Description File Schema” on page 8-4](#) and [“Rules for Specifying Stored Procedure Description Files” on page 8-10](#).

The Stored Procedure Description file for this example defines an empty complex type in the `<types>` section and defines a function that returns that complex type in the `<functions>` section. The function definition contains `<argument>` elements for each input and output argument. The `<argument>` elements specify the name (`label` attribute), parameter type (`mode` attribute), and data type (`type` attribute) for each input and output of the stored procedure.

The following is a code listing of the Stored Procedure Description file for this example.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions>
  <types>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name="Results">
        <xs:complexType>
          <xs:sequence>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:schema>
    </types>
  </definitions>
```

```

        </xs:complexType>
    </xs:element>
</xs:schema>
</types>
<functions>

    <function name="GetOrderInfo" return_type="Results">
        <argument label="customer_id" mode="input_only"
            type="xs:string"/>
        <argument label="order_amount" mode="input_only"
            type="xs:integer"/>
        <argument label="totalsum" mode="output_only"
            type="xs:integer"/>
        <argument label="totalorder" mode="output_only"
            type="xs:integer"/>
        <presentation group="Pointbase stored procedures"/>
    </function>

</functions>
</definitions>

```

Step 3: Specify the Stored Procedure Description File in the Liquid Data Console

Perform the following steps to specify the Stored Procedure Description File in the data source description:

1. In the Liquid Data Administration Console (to access the Liquid Data Console, click the Liquid Data link at the bottom of the list on the WebLogic administration console), click the Data Sources tab.
2. Click the Relational Databases tab.
3. Select an existing relational data source and edit it or create a new relational data source.

If you are creating a new data source, you must also configure a JDBC Connection Pool to access your database and a JDBC Data Source for the connection pool.

4. In the Configure Relational Data Source Description screen, enter values for Name, Data Source Name, and Schema fields, if they are not already entered. For more details on configuring relational data sources, see [Configuring Access to Relational Databases](#) in the *Administration Guide*.

5. In the Configure Relational Data Source Description screen, click the Browse Repository link next to the Stored Procedure Description File field.
6. In the Repository Browser, select the file you created containing your stored procedure definitions. After making your selection, click the Select button.
7. In the Configure Relational Data Source Description screen, click the Apply button to save your Data Source definition.

Step 4: Open the Data View Builder to See Your Stored Procedures

Start the Data View Builder and connect to the Liquid Data server. If you are already connected, run the File > Connect command to reconnect. If you configured the Stored procedure correctly, it appears in the Sources tab as one of the stored procedures.

Step 5: Use the Stored Procedure in a Query

Perform the following steps in the Data View Builder to create a query that uses the stored procedure.

1. Start a new Data View Builder project (File → New Project).
2. Open the source and target schemas.
 - Drag and drop Source → Stored Procedure → PB-WL → getOrderInfo into the design area.
 - Set the target schema to `getorderinfo.xsd` (in the repository).
3. Create a query parameter named `CUST_ID` of type `xs:string` for `customer_id`.
4. Drag the `CUST_ID` query parameter into the `customer_id` stored procedure input.
5. Create a numeric constant of 500 and drag it into the `order_amount` input parameter.
6. Drag the `totalsum` stored procedure output to the `totalsum` element of the target schema.
7. Drag the `totalorder` stored procedure output to the `totalorder` element of the target schema.

Step 6: Run the Query

Perform the following to run this query:

1. Click the test tab in the Data View Builder.
2. Enter a value for the `CUST_ID` query parameter. For example, enter `CUSTOMER_1`.
3. Click the Run button. The results will look similar to the following:

```
<Results>  
  <totalsum>7000</totalsum>  
  <totalorder>3</totalorder>  
</Results>
```


9 Query Cookbook

This section provides examples of more complex BEA Liquid Data for WebLogic™ queries using some of the advanced features and tools offered in the Data View Builder. At this point, we assume that you are familiar with the Data View Builder user interface (described in [Chapter 2, “Starting the Builder and Touring the GUI”](#)) and that you have an understanding of the basic concepts and tasks presented in [Getting Started, Chapter 1, “Overview and Key Concepts”](#), and [Chapter 3, “Designing Queries.”](#)

The following use cases and examples are provided here to give you a jump-start for constructing real-world queries to solve common problems. Each use case includes a viewlet demo of building the solution using Data View Builder. Watching a viewlet takes 3 to 5 minutes—we suggest sitting back and enjoying with popcorn and your favorite soda pop.

- [Example 1: Simple Joins \(View a Demo\)](#)
- [Example 2: Aggregates \(View a Demo\)](#)
- [Example 3: Date and Time Duration \(View a Demo\)](#)
- [Example 4: Union \(View a Demo\)](#)
- [Example 5: Minus \(View a Demo\)](#)
- [Example 6: Complex Parameter Type \(CPT\) \(View a Demo\)](#)

For an example of using a stored procedure in a query, see [“Example: Defining and Using a Customer Orders Stored Procedure”](#) on page 8-33.

Each use case has an example with a description of the problem and the steps to solve the problem. The examples use two databases:

- The Broadband database (PB-BB) contains “Broadband” subscribers and service orders
- The Wireless database (PB-WL) contains “Wireless” subscribers.

In cases where the target schemas do not already exist in the Samples repository, they are provided in this documentation along with the examples. You can cut-and-paste the schema content into an `.xsd` file to construct your own target schemas. (You can also copy from the PDF version of this document which may give you a copy that formats better your text editor.)

Note: To find out what data are contained in any data source, create a new “test” project, open the source schema you are interested in, and map key source nodes to any appropriate target schema. (For example, map customer first and last names and customer ID from source to target schemas.) Then click on Test tab and choose Query—>Run Query. The result will return all customers in the data source queried.

As you work through the examples, remember to save any projects that you want to keep before creating new ones.

Example directories

Examples used in this chapter assume that the Liquid Data 1.1 directory is located at:

```
BEAHOME/weblogic700/liquiddata/
```

where *BEAHOME* is the location of the WebLogic server and `liquiddata` is the location of the Liquid Data 1.1 Samples server installation.

The examples also assume that the path to the Samples server Liquid Data repository is:

```
BEAHOME/weblogic700/liquiddata/samples/config/ld_samples/ldrepository
```

The symbolic name `ld_repository` represents the Liquid Data repository and its contents.

Example 1: Simple Joins

A join merges data from two data sources based on a certain relation.

The Problem

For each Wireless Customer ID, determine whether the customer has any Broadband orders. Assume that the Customer ID matches across databases.

The Solution

First, you want to find matching Broadband customers (who are also included in the Wireless database), then return Broadband Order IDs for the matching customers. Because Customer IDs in the Wireless database align with those in Broadband, we can find matching Broadband customers with a simple join of Wireless Customer IDs with the Customer IDs in the Broadband order information.

To create the solution, follow these steps:

- [View a Demo](#)
- [Ex 1: Step 1. Verify the Target Schema is Saved in Repository](#)
- [Ex 1: Step 2. Open Source and Target Schemas](#)
- [Ex 1: Step 3. Map Nodes from Source to Target Schema to Project the Output](#)
- [Ex 1: Step 4. Create a Query Parameter for a Customer ID to be Provided at Query Runtime](#)
- [Ex 1: Step 5. Assign the Query Parameter to a Source Node](#)
- [Ex 1: Step 6. Join the Wireless and Broadband Customer IDs](#)
- [Ex 1: Step 7. Set Optimization Hints](#)
- [Ex 1: Step 8. View the XQuery and Run the Query to Test it](#)
- [Ex. 1: Step 9. Verify the Result](#)

View a Demo

Simple Joins Demo... If you are looking at this documentation online, you can click the “Demo” button to see a viewlet demo showing how to build the conditions and create the mappings described in this example. This demo previews the steps described in detail in the following sections. The demo assumes you already have the target schema in the server Repository.

Ex 1: Step 1. Verify the Target Schema is Saved in Repository

For this example, we will use a target schema called `customerOrders.xsd`. This schema is available in the Samples server repository. The path to the schemas folder in the Liquid Data server repository is:

```
ld_repository/schemas/
```

See “[Example directories](#)” on page 9-2 for information on how example directory names are used.

Just in case you want to verify that you have the right schema file, the following code listing shows the XML for this schema.

Listing 9-1 XML Source for customerOrders.xsd Target Schema File

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
  <xsd:element name = "customers">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref = "customer" minOccurs = "0" maxOccurs = "unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "customer">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref = "first_name"/>
        <xsd:element ref = "last_name"/>
        <xsd:element ref = "orders" minOccurs = "0" maxOccurs = "unbounded"/>
      </xsd:sequence>
      <xsd:attribute name = "id" use = "optional" type = "xsd:string"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "first_name" type = "xsd:string"/>
  <xsd:element name = "last_name" type = "xsd:string"/>
</xsd:schema>
```

```
<xsd:element name = "orders">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref = "order" minOccurs = "0" maxOccurs = "unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name = "order">
  <xsd:complexType>
    <xsd:attribute name = "id" use = "optional" type = "xsd:string"/>
    <xsd:attribute name = "date" use = "optional" type = "xsd:string"/>
    <xsd:attribute name = "amount" use = "optional" type = "xsd:string"/>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

Ex 1: Step 2. Open Source and Target Schemas

1. In the Data View Builder, choose File—>New Project to clear your desktop and reset all default values.
2. On the Builder Design—>Sources tab, click Relational Databases and open two data sources:
 - Double-click on the PB-WL (Wireless) relational database to open the schema for this data source.
 - Double-click on the PB-BB (Broadband) relational database to open the schema for this data source.
3. Choose the menu option File—>Set Target Schema.

Navigate to the server Repository or to the location where you saved the customerOrders.xsd schema. Choose customerOrders.xsd and click Open. customerOrders.xsd appears as the target schema.

This target schema is displayed as a docked schema window on the right side of the design area.
4. Click the plus (+) sign (or right-mouse click and choose Expand) to expand the nodes in each source schema and in the target schema.

Ex 1: Step 3. Map Nodes from Source to Target Schema to Project the Output

1. Drag and drop [PB-WL]db/CUSTOMER/CUSTOMER_ID from source schema onto the target schema [customerOrders.xsd]/customers/customer/id.
2. Drag and drop [PB-WL]db/CUSTOMER/FIRST_NAME from source schema onto the target schema [customerOrders.xsd]/customers/customer/first_name.
3. Drag and drop [PB-WL]db/CUSTOMER/LAST_NAME from source schema onto the target schema [customerOrders.xsd]/customers/customer/last_name.
4. Drag and drop [PB-BB]db/CUSTOMER_ORDER/ORDER_DATE onto the target schema [customerOrders.xsd]/customers/customer/orders/order/order_date.
5. Drag and drop [PB-BB]db/CUSTOMER_ORDER/ORDER_ID onto the target schema [customerOrders.xsd]/customers/customer/orders/order/id.

Ex 1: Step 4. Create a Query Parameter for a Customer ID to be Provided at Query Runtime

Create a Query Parameter *wireless_id* variable for a Wireless Customer ID that you will supply at query execution time:

1. On the Builder Design, click Toolbox and then click Query Parameter.
2. From the “Type” drop-down menu, choose `xs:string`.
3. In Parameter Name field, enter `wireless_id` and click Add.

The new parameter is displayed in the Query Parameters tree.

Ex 1: Step 5. Assign the Query Parameter to a Source Node

Drag and drop the *wireless_id* query parameter to [PB-WL]db/CUSTOMER/CUSTOMER_ID.

Ex 1: Step 6. Join the Wireless and Broadband Customer IDs

Drag and drop (join) [PB-WL]db/CUSTOMER/CUSTOMER_ID to [PB-BB]db/CUSTOMER_ORDER/CUSTOMER_ID.

Ex 1: Step 7. Set Optimization Hints

1. Click the Optimize tab.
2. Under Join Pair Hints, on the drop-down menu select PB-WL and PB-BB.
This represents the first join you created between Wireless and Broadband Customer IDs.
3. Click into the empty cell under Hints to get the drop-down menu and choose “Pass Parameter to Right” for the PB-WL and PB-BB join.

Note: For information on using optimization hints see [“Optimization Hints for Joins” on page 4-5](#).

Ex 1: Step 8. View the XQuery and Run the Query to Test it

1. Click on the Test tab.
The generated XQuery for this query is shown in the following code listing.

Listing 9-2 XQuery for Example 1: Simple Joins

```
{--      Generated by Data View Builder 1.0--}

<customers>
  {
    for $PB_WL.CUSTOMER_1 in document("PB-WL")/db/CUSTOMER
    where ($#wireless_id of type xs:string eq $PB_WL.CUSTOMER_1/CUSTOMER_ID)
    return
      <customer id={$PB_WL.CUSTOMER_1/CUSTOMER_ID}>
        <first_name>{ xf:data($PB_WL.CUSTOMER_1/FIRST_NAME) }</first_name>
        <last_name>{ xf:data($PB_WL.CUSTOMER_1/LAST_NAME) }</last_name>
        <orders>
          {
            for $PB_BB.CUSTOMER_ORDER_2 in
document("PB-BB")/db/CUSTOMER_ORDER
              where ($PB_WL.CUSTOMER_1/CUSTOMER_ID eq
$PB_BB.CUSTOMER_ORDER_2/CUSTOMER_ID)
              return
                <order id={$PB_BB.CUSTOMER_ORDER_2/ORDER_ID} date={cast as
xs:string($PB_BB.CUSTOMER_ORDER_2/ORDER_DATE)}></order>
          }
        </orders>
      </customer>
    }
  </customers>
```

2. Set the variable value to submit to the query when the query runs. To do this, you need to enter a value in the Query Parameter panel. Double-click into the cell under Value and enter CUSTOMER_3.

(Customer IDs CUSTOMER_1 through CUSTOMER_10 are available in the data source to try.)

3. Click the “Run query” button to run the query against the data sources.

Ex. 1: Step 9. Verify the Result

Running this query with the `wireless_id` parameter set to CUSTOMER_3 produces the following XML query result.

Listing 9-3 Result for Example 1: Simple Joins

```
<customers>
  <customer id="CUSTOMER_3">
    <first_name>JOHN_3</first_name>
    <last_name>KAY_3</last_name>
    <orders>
      <order date="2002-03-06-08:00" id="ORDER_ID_3_0"/>
      <order date="2002-03-06-08:00" id="ORDER_ID_3_1"/>
      <order date="2002-03-06-08:00" id="ORDER_ID_3_2"/>
      <order date="2002-03-06-08:00" id="ORDER_ID_3_3"/>
    </orders>
  </customer>
</customers>
```

Example 2: Aggregates

Aggregate functions produce a single value from a set of input values. An example of an aggregate function in Data View Builder is the count function, which takes a list of values and returns the number of values in the list.

The Problem

Find the number of orders placed in the Broadband database for a given customer who is also in the Wireless database.

The Solution

This query relies on a data view called “AllOrders” which retrieves customers who are in the Broadband database and also in the Wireless database. For each of these customers, the customer ID and orders are retrieved. Then, we use the Aggregate function “count” to determine how many orders are associated with a given customer. At query runtime, a customer ID is submitted as a query parameter and the result returns the number of orders associated with the given customer ID.

To create the solution, follow these steps:

- [View a Demo](#)
- [Ex 2: Step 1. Locate and Configure the “AllOrders” Data View](#)
- [Ex 2: Step 2. Restart the Data View Builder and Find the New Data View](#)
- [Ex 2: Step 3. Verify the Target Schema is Saved in the Repository](#)
- [Ex 2: Step 4. Open the Data Sources and Target Schema](#)
- [Ex 2: Step 5. Map Source Nodes to Target to Project the Output](#)
- [Ex 2: Step 6. Create Two Query Parameters to be Provided at Query Runtime](#)
- [Ex 2: Step 7. Assign the Query Parameters to Source Nodes](#)
- [Ex 2: Step 8. Add the “count” Function](#)
- [Ex 2: Step 9. Verify Mappings and Conditions](#)
- [Ex 2: Step 10. View the XQuery and Run the Query to Test it](#)
- [Ex 2: Step 11. Verify the Result](#)

View a Demo

Aggregates Demo... If you are looking at this documentation online, you can click the “Demo” button to see a viewlet demo showing how to build the conditions and create the mappings described in this example. This demo previews the steps described in detail in the following sections. The demo assumes you already have the target schema in the server Repository and have created and configured the data view data source required for this example.

Ex 2: Step 1. Locate and Configure the “AllOrders” Data View

For this example, we will use a data view data source called `AllOrders.xv`. This data view is available in the Samples server repository. The path to the `data_views` folder in the Liquid Data server repository is:

```
ld_repository/data_views/
```

See “[Example directories](#)” on page 9-2 for information on how example directory names are used.

Just in case you want to verify that you have the right data view file, the following code listing shows the XML for this data view.

Listing 9-4 XML Source for AllOrders.xv Data View File

```
<customers>
  {
    for $PB-BB.CUSTOMER_1 in document("PB-BB")/db/CUSTOMER
    for $PB-WL.CUSTOMER_2 in document("PB-WL")/db/CUSTOMER
    where ($PB-WL.CUSTOMER_2/CUSTOMER_ID eq {--! ppright !--}
$PB-BB.CUSTOMER_1/CUSTOMER_ID)

    return
    <customer id={$PB-WL.CUSTOMER_2/CUSTOMER_ID}>
      <first_name>{ xf:data($PB-WL.CUSTOMER_2/FIRST_NAME) }</first_name>
      <last_name>{ xf:data($PB-WL.CUSTOMER_2/LAST_NAME) }</last_name>
      <orders>
        {
          for $PB-BB.CUSTOMER_ORDER_4 in
document("PB-BB")/db/CUSTOMER_ORDER
          where ($PB-BB.CUSTOMER_1/CUSTOMER_ID eq {--! ppright !--}
$PB-BB.CUSTOMER_ORDER_4/CUSTOMER_ID)
          return
          <order id={$PB-BB.CUSTOMER_ORDER_4/ORDER_ID}
date={$PB-BB.CUSTOMER_ORDER_4/ORDER_DATE}></order>
```

```

    }
  </orders>
</customer>
}
</customers>

```

Use the WLS Administration Console to Configure Your Data View Data Source

1. Start and login to the WLS Administration Console for the Samples server you are using.

To start the WLS Administration Console for the Liquid Data Samples server running on your local machine, type the following URL in a Web browser address field:

```
http://localhost:7001/console
```

Login to the console by providing the following default username and password for the Samples server.

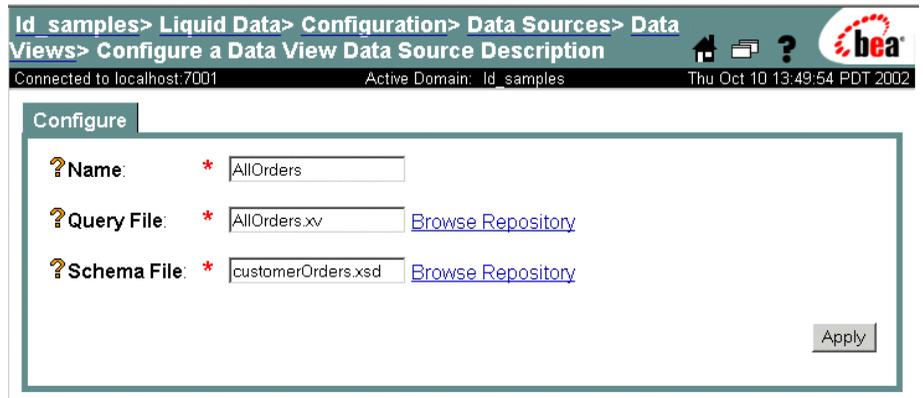
Table 9-1 User Name and Password for Samples WLS Administration Console

Field	Defaults
Username	system
Password	security

2. In the left pane, click the Liquid Data node.
3. In the right pane, click the Configuration tab.
4. Click the Data Sources tab.
5. Click the Data Views tab.
6. Start and login to the Administration Server. See [Start the WLS Administration Console](#) is the Getting Started guide for details.
7. In the left pane, click the Liquid Data node.
8. In the right pane, click the Configuration tab.
9. Click the Data Sources tab.

10. Click the Data Views tab.
11. Click the Configure a new Data View source description text link.
 The configuration tab for creating a new Data View Liquid Data source description is displayed.

Figure 9-1 Configuring Liquid Data Source Description for a Data View



12. Fill in the fields as indicated in the following table.

Table 9-2 Liquid Data Data View Configuration

Field	Description
Name	AllOrders
Query File	AllOrders.xv
Schema	customerOrders.xsd

13. Click Create.
 You can click on Data Views in the breadcrumbs path at the top of the console to see the data view you added displayed in the summary table.

Ex 2: Step 2. Restart the Data View Builder and Find the New Data View

1. Restart the Data View Builder.

If the Data View Builder was running while you configured the new data view, shut it down (menu option File—>Exit) and restart it in order to see the new data view you created show up in the Builder as a data source.

2. On the Design tab, on the Builder Toolbar, click the Sources tab, then click Data Views.

The `AllOrders.xv` data view should be displayed in the list of available data views.

Ex 2: Step 3. Verify the Target Schema is Saved in the Repository

For this example, we will use a target schema called `customerOrdersA.xsd`. This schema is available in the Samples server repository. The path to the schemas folder in the Liquid Data server repository is:

```
ld_repository/schemas/
```

Just in case you want to verify that you have the right schema file, the following code listing shows the XML for this schema.

Listing 9-5 XML Source for customerOrdersA.xsd Target Schema File

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
  <xsd:element name="customers">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="customer" minOccurs="0" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="first_name" type="xsd:string"/>
              <xsd:element name="last_name" type="xsd:string"/>
              <xsd:element name="orders" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="order" minOccurs="0" maxOccurs="unbounded">
                      <xsd:complexType>
                        <xsd:sequence>
                          </xsd:sequence>
                        </xsd:sequence>
                      <xsd:attribute name="id" type="xsd:string"/>
                    </xsd:element>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```
<xsd:attribute name="date" type="xsd:string"/>
<xsd:attribute name="amount" type="xsd:string"/>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name="amount" type="xsd:string"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:string"/>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

Ex 2: Step 4. Open the Data Sources and Target Schema

1. In the Data View Builder, choose File—>New Project to clear your desktop and reset all default values.
2. On the Builder Design—>Sources tab, click Data Views, and double-click on `AllOrders.xv` to open the schema for that data source.
3. Choose File—>Set Target Schema. Use the file browser to navigate to the Repository and select `CustomerOrdersA.xsd` as the target schema.

`CustomerOrdersA.xsd` appears as the target schema.

This target schema is displayed as a docked schema window on the right side of the design area.

Ex 2: Step 5. Map Source Nodes to Target to Project the Output

1. Drag and drop `[AllOrders]/customers/customer/first_name` from `AllOrders` source schema onto `[CustomerOrdersA.xsd]/customers/customer/first_name` in the target schema.
2. Drag and drop `[AllOrders]/customers/customer/last_name` from `AllOrders` source schema onto `[CustomerOrdersA.xsd]/customers/customer/last_name` in the target schema.

Ex 2: Step 6. Create Two Query Parameters to be Provided at Query Runtime

Create two Query Parameter variables: *first_name* and *last_name*, that you can use to insert variable customer information when the query runs. Create both variables as type `xs:string`. Do this as follows:

1. On the Builder Toolbar, click Toolbox and then click Query Parameter.
2. From the “Type” drop-down menu, choose `xs:string`.
3. In Parameter Name field, enter `first_name` and click Add.

The new parameter is displayed in the Query Parameters tree.

4. Repeat steps 2 and 3 to create the `last_name` variable.

You should now see both parameters displayed in the Query Parameters tree.

Ex 2: Step 7. Assign the Query Parameters to Source Nodes

Assign the *first_name* and *last_name* Query Parameter variables to customer first name and last name nodes in the AllOrders data view as follows:

1. Drag and drop the *first_name* variable onto `[allOrders]/customers/customer/first_name` in the AllOrders source schema.
2. Drag and drop the *last_name* variable onto `[allOrders]/customers/customer/last_name` in the AllOrders source schema.

Ex 2: Step 8. Add the “count” Function

Add the count function and specify the input and output as follows:

1. On the Builder Toolbar, click Toolbox and then click Functions.
2. Double-click on the `count` function (under Aggregate Functions)

The `count` function window is displayed, showing input parameter *srcval* and output as some *integer*.

Note: Create complex or aggregate functions only on the desktop by double-clicking as described in this step. Do not attempt to drag and drop them directly into the Conditions tab.

3. Drag and drop [AllOrders]/customer/orders/order/date from the AllOrders source schema onto [count-Function]input/Parameters/srcval.
4. Drag and drop [count-Function]Output/integer to [customerOrdersA.xsd]/customers/customer/amount in the target schema.

Note: Make sure to drag *integer* onto the customer amount—the last node in the fully expanded schema tree; not onto the optional orders amount?.

Ex 2: Step 9. Verify Mappings and Conditions

Your mappings should look like those shown in [Figure 9-2](#).

Figure 9-2 Mappings for Example2: Aggregates

	Source	Target
0	[AllOrders]/customers/customer/first_name	[customerOrdersA.xsd]/customers/customer/first_name
1	[AllOrders]/customers/customer/last_name	[customerOrdersA.xsd]/customers/customer/last_name
2	[AllOrders]/customers/customer/orders/order/@date	[count]/Parameters/srcval
3	[count]/integer	[customerOrdersA.xsd]/customers/customer/amount
4		

Mapping for: All Show full path

Mappings Conditions Sort By

Your Conditions should like those shown in [Figure 9-3](#).

Figure 9-3 Conditions for Example 2: Aggregates

	Condition	Scope
0	(first_name eq [AllOrders]/customers/customer/first_name)	
1	(last_name eq [AllOrders]/customers/customer/last_name)	
2		
3		
4		

Condition for: All Show full path

Mappings Conditions Sort By

Ex 2: Step 10. View the XQuery and Run the Query to Test it

1. Click on the Test tab.

The generated XQuery for this query is shown in the following code listing.

Listing 9-6 XQuery for Example 2: Aggregates

```
{--      Generated by Data View Builder 1.0 --}

<customers>
  {
    for $AllOrders.customer_1 in document("AllOrders")/customers/customer
    let $srcval_2 :=
      for $AllOrders.order_3 in $AllOrders.customer_1/orders/order
      where ($#first_name of type xs:string eq
$AllOrders.customer_1/first_name)
          and ($#last_name of type xs:string eq
$AllOrders.customer_1/last_name)
      return
        $AllOrders.order_3/@date
    let $count_4 := xf:count($srcval_2)
    where ($#first_name of type xs:string eq $AllOrders.customer_1/first_name)
          and ($#last_name of type xs:string eq $AllOrders.customer_1/last_name)
    return
      <customer>
        <first_name>{ xf:data($AllOrders.customer_1/first_name) }</first_name>
        <last_name>{ xf:data($AllOrders.customer_1/last_name) }</last_name>
        <amount>{ $count_4 }</amount>
      </customer>
    }
  }
</customers>
```

2. In the Query Parameter panel on the Test tab, set the variable values as follows:

- *last_name*
(For *last_name*, KAY_1 through KAY_10 are available in the data source.)
- *first_name*
(For *first_name*, JOHN_1 through JOHN_10 are available in the data source.)

3. Click the “Run query” button to run the query against the data sources.

Ex 2: Step 11. Verify the Result

Running this query with `last_name` set to “KAY_1” and `first_name` set to “JOHN_1” produces the following XML query result.

Listing 9-7 Result for Example 2: Aggregates

```
<customers>
  <customer>
    <first_name>JOHN_1</first_name>
    <last_name>KAY_1</last_name>
    <amount>2</amount>
  </customer>
</customers>
```

Example 3: Date and Time Duration

Data View Builder supports a set of functions that operate on date and time. For more information on date and time functions see “[Date and Time Functions](#)” on page A-50 in the “Functions Reference.”

The Problem

Determine if a Broadband customer has any open orders in the Broadband database before a specified date.

The Solution

For each Broadband order that matches the given Customer ID, you need to set these conditions:

- The order status is “OPEN”

- The ship date for a given *customer_id* is earlier than or equal to the date (*date1*) provided. (*customer_id* and *date1* are variables that you define as query parameters to be submitted at query runtime).

To create the solution, follow these steps:

- [View a Demo](#)
- [Ex 3: Step 1. Verify the Target Schema is Saved in Repository](#)
- [Ex 3: Step 2. Open Source and Target Schemas](#)
- [Ex 3: Step 3. Map Source to Target Nodes to Project the Output](#)
- [Ex 3: Step 4. Create Joins](#)
- [Ex 3: Step 5. Create Two Query Parameters for Customer ID and Date to be Provided at Query Runtime](#)
- [Ex 3: Step 6. Set a Condition Using the Customer ID](#)
- [Ex 3: Step 7. Set a Condition to Determine if Order Ship Date is Earlier or Equal to a Date Submitted at Query Runtime](#)
- [Ex 3: Step 8. Set a Condition to Include Only “Open” Orders in the Result](#)
- [Ex 3: Step 9. View the XQuery and Run the Query to Test it](#)
- [Ex 3: Step 9. Verify the Result](#)

View a Demo

Date and Time Duration Demo... If you are looking at this documentation online, you can click the “Demo” button to see a viewlet demo showing how to build the conditions and create the mappings described in this example. This demo previews the steps described in detail in the following sections. The demo assumes you already have the target schema in the server Repository.

Ex 3: Step 1. Verify the Target Schema is Saved in Repository

For this example, we will use a target schema called `customerLineItems.xsd`. This schema is available in the Samples server repository.

`ld_repository/schemas/`

See “[Example directories](#)” on page 9-2 for information on how example directory names are used.

Just in case you want to verify that you have the right schema file, the following code listing shows the XML for this schema.

Listing 9-8 XML Source for customerLineItems.xsd Target Schema File

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
  <xsd:element name = "customers">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref = "customer" minOccurs = "0" maxOccurs = "unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "customer">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref = "first_name"/>
        <xsd:element ref = "last_name"/>
        <xsd:element ref = "orders" minOccurs = "0" maxOccurs = "unbounded"/>
      </xsd:sequence>
      <xsd:attribute name = "id" use = "required" type = "xsd:string"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "first_name" type = "xsd:string"/>
  <xsd:element name = "last_name" type = "xsd:string"/>
  <xsd:element name = "orders">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref = "order" minOccurs = "0" maxOccurs = "unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "order">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref = "line_item" minOccurs = "0" maxOccurs = "unbounded"/>
      </xsd:sequence>
      <xsd:attribute name = "id" use = "required" type = "xsd:string"/>
      <xsd:attribute name = "date" use = "required" type = "xsd:string"/>
      <xsd:attribute name = "amount" use = "required" type = "xsd:string"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name = "line_item">
    <xsd:complexType>
      <xsd:attribute name = "id" use = "required" type = "xsd:string"/>
      <xsd:attribute name = "product" use = "required" type = "xsd:string"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```
<xsd:attribute name = "status" use = "required" type = "xsd:string"/>
<xsd:attribute name = "expected_ship_date" use = "required" type = "xsd:string"/>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

Ex 3: Step 2. Open Source and Target Schemas

1. In the Data View Builder, choose File—>New Project to clear your desktop and reset all default values.
2. On the Builder Design—>Sources tab, click Relational Databases and open one data source:
 - Double-click on the PB-BB (Broadband) relational database to open the schema for this data source.
3. Choose the menu option File—>Set Target Schema.

Navigate to the server Repository or to the location where you saved the customerLineItems.xsd schema. Choose customerLineItems.xsd and click Open.

customerLineItems.xsd appears as the target schema.

This target schema is displayed as a docked schema window on the right side of the design area.

4. Click the plus (+) sign (or right-mouse click and choose Expand) to expand the nodes in each source schema and in the target schema.

Ex 3: Step 3. Map Source to Target Nodes to Project the Output

Project the output values as follows.

1. Drag and drop [PB-BB]/db/CUSTOMER/FIRST_NAME from the source schema onto [customerLineItems.xsd]/customers/customer/first_name in the target schema.
2. Drag and drop [PB-BB]/db/CUSTOMER/LAST_NAME from the source schema onto [customerLineItems.xsd]/customers/customer/last_name in the target schema.

3. Drag and drop
 [PB-BB]/db/CUSTOMER/CUSTOMER_ORDER_LINE_ITEM/LINE_ID from the source schema onto
 [customerLineItems.xsd]/customers/customer/orders/order/line_item/id in the target schema (*id* is an *attribute* of *line_item*).
4. Drag and drop
 [PB-BB]/db/CUSTOMER/CUSTOMER_ORDER_LINE_ITEM/PRODUCT_NAME from the source schema onto
 [customerLineItems.xsd]/customers/customer/orders/order/line_item/product in the target schema (*product* is an *attribute* of *line_item*).
5. Drag and drop
 [PB-BB]/db/CUSTOMER/CUSTOMER_ORDER_LINE_ITEM/STATUS from the source schema
 [customerLineItems.xsd]/customers/customer/orders/order/line_item/status in the target schema (*status* is an *attribute* of *line_item*).
6. Drag and drop
 [PB-BB]/db/CUSTOMER/CUSTOMER_ORDER_LINE_ITEM/EXPECTED_SHIP_DATE from the source schema
 [customerLineItems.xsd]/customers/customer/orders/order/line_item/expected_ship_date in the target schema (*expected_ship_date* is an *attribute* of *line_item*).

At this point, the following mappings should be displayed on the Mappings tab. (Getting the mappings in the same order as shown is not as important as verifying that the relationships between source and target nodes are the same. The @ symbols indicate attributes.)

Source	Target
[PB-BB]/db/CUSTOMER/FIRST_NAME	[customerLineItems.xsd]/customers/customer/first_name
[PB-BB]/db/CUSTOMER/LAST_NAME	[customerLineItems.xsd]/customers/customer/last_name
[PB-BB]/db/CUSTOMER/CUSTOMER_ORDER_LINE_ITEM/LINE_ID	[customerLineItems.xsd]/customers/customer/orders/order/line_item/@id

Source	Target
[PB-BB]/db/CUSTOMER/CUSTOMER_ORDER_LINE_ITEM/PRODUCT_NAME	[customerLineItems.xsd]/customers/customer/orders/order_line_item/@product
[PB-BB]/db/CUSTOMER/CUSTOMER_ORDER_LINE_ITEM/STATUS	[customerLineItems.xsd]/customers/customer/orders/order_line_item/@status
[PB-BB]/db/CUSTOMER/CUSTOMER_ORDER_LINE_ITEM/EXPECTED_SHIP_DATE	[customerLineItems.xsd]/customers/customer/orders/order_line_item/@expected_ship_date

Ex 3: Step 4. Create Joins

Join customer with corresponding line-item data. This requires two joins, one to find the customer's Order IDs, and another that uses the Order IDs and finds the corresponding line-item information:

1. Drag and drop [PB-BB]/db/CUSTOMER/CUSTOMER_ID onto [PB-BB]/db/CUSTOMER_ORDER/CUSTOMER_ID.
2. Drag and drop [PB-BB]/db/CUSTOMER_ORDER/ORDER_ID onto [PB-BB]/db/CUSTOMER_ORDER_LINE_ITEM/ORDER_ID.

Ex 3: Step 5. Create Two Query Parameters for Customer ID and Date to be Provided at Query Runtime

Create two Query Parameter variables: *customer_id* and *date1*, that you can use to insert as variable values when the query runs. Create both variables as type `xs:string`. Do this as follows:

1. On the Builder Toolbar, click Toolbox and then click Query Parameter.
2. From the "Type" drop-down menu, choose `xs:string`.
3. In Parameter Name field, enter `customer_id` and click Add.
The new parameter is displayed in the Query Parameters tree.
4. Repeat steps 2 and 3 to create the `date1` variable.

You should now see both parameters displayed in the Query Parameters tree.

Ex 3: Step 6. Set a Condition Using the Customer ID

1. On the Builder Toolbar, click Toolbox and then click Functions.
2. Drag and drop the equals (eq) function (under Operators) onto the next empty row in the Conditions tab.

The Functions Editor pops up and displays a statement with placeholder variables for you to fill in.

3. On the Builder Toolbar, click on Query Parameter, then drag *customer_id* onto *anyValue1* onto the left side of the equation.
4. Drag [PB-BB]/db/CUSTOMER/CUSTOMER_ID onto the right side of the equation.

The function should look like this:

```
(customer_id eq [PB-BB]/db/CUSTOMER/CUSTOMER_ID)
```

5. Close the Functions Editor.

Ex 3: Step 7. Set a Condition to Determine if Order Ship Date is Earlier or Equal to a Date Submitted at Query Runtime

1. Click on Functions, and drag and drop the Operator function *le* (less than or equal) onto the next empty row on the Conditions tab.

The Functions Editor pops up and displays a statement with placeholder variables for you to fill in.

2. Drag and drop [PB-BB]/db/CUSTOMER_ORDER_LINE_ITEM/EXPECTED_SHIP_DATE onto *anyValue1* on the left side of the equation.
3. Click on Functions, and drag and drop the *date-from-string-with-format* function onto *anyValue2* on the right side of the equation.

At this point, the expression in the Functions Editor should look like this:

```
( [PB-BB]/db/CUSTOMER_ORDER_LINE_ITEM/EXPECTED_SHIP_DATE le  
xfext:date-from-string-with-format(pattern,srcval) )
```

4. Click Constants, enter the following in the String field:

```
yyyy-MM-dd
```

Now drag it (via the Constant icon next to the field) onto *pattern* (first placeholder parameter to the date function).

5. Click on Query Parameter, and drag and drop *date1* from the Query Parameters panel onto *srcval* (the second placeholder parameter to the date function).

The completed expression should look like this:

```
( [PB-BB] /db/CUSTOMER_ORDER_LINE_ITEM/EXPECTED_SHIP_DATE le
xfext:date-from-string-with-format("yyyy-MM-dd", date1)
```

6. Close the Functions Editor.

The condition you created is displayed on the Conditions tab in the Source column.

Ex 3: Step 8. Set a Condition to Include Only “Open” Orders in the Result

Set the second condition to an Open ORDER status.

1. Click on Functions, and drag and drop the Operator function `eq` (equal) onto the Conditions tab.

The Functions Editor pops up and displays a statement with placeholder variables for you to fill in.

2. For the left parameter (*anyValue1*), drag and drop `[PB-BB]/db/CUSTOMER_ORDER_LINE_ITEM/STATUS` on to *anyValue1*.
3. For the right parameter (*anyValue2*), create a constant String with a value of OPEN, and drop it (via the Constant icon next to the field) onto *anyValue2*.

The completed expression should look like this:

```
( [PB-BB] /db/CUSTOMER_ORDER_LINE_ITEM/STATUS eq "OPEN")
```

Close the Functions Editor.

Ex 3: Step 9. View the XQuery and Run the Query to Test it

1. Click on the Test tab.

The generated XQuery for this query is shown in the following code listing.

Listing 9-9 XQuery for Example 3: Date and Time Duration

```
{--      Generated by Data View Builder 1.0 --}
<customers>
  {
    for $PB-BB.CUSTOMER_1 in document("PB-BB")/db/CUSTOMER
    where ($#customer_id of type xs:string eq $PB-BB.CUSTOMER_1/CUSTOMER_ID)
    return
      <customer>
        <first_name>{ xf:data($PB-BB.CUSTOMER_1/FIRST_NAME) }</first_name>
        <last_name>{ xf:data($PB-BB.CUSTOMER_1/LAST_NAME) }</last_name>
        <orders>
          <order>
            {
              for $PB-BB.CUSTOMER_ORDER_2 in
document("PB-BB")/db/CUSTOMER_ORDER
              for $PB-BB.CUSTOMER_ORDER_LINE_ITEM_3 in
document("PB-BB")/db/CUSTOMER_ORDER_LINE_ITEM
              where ($PB-BB.CUSTOMER_ORDER_2/ORDER_ID eq
$PB-BB.CUSTOMER_ORDER_LINE_ITEM_3/ORDER_ID)
                and
($PB-BB.CUSTOMER_ORDER_LINE_ITEM_3/EXPECTED_SHIP_DATE le
xfext:date-from-string-with-format("yyyy-MM-dd", $date1 of type xs:string))
                and ($PB-BB.CUSTOMER_ORDER_LINE_ITEM_3/STATUS eq
"OPEN")
                and ($PB-BB.CUSTOMER_1/CUSTOMER_ID eq
$PB-BB.CUSTOMER_ORDER_2/CUSTOMER_ID)
              return
                <line_item
id={$PB-BB.CUSTOMER_ORDER_LINE_ITEM_3/LINE_ID}
product={$PB-BB.CUSTOMER_ORDER_LINE_ITEM_3/PRODUCT_NAME}
status={$PB-BB.CUSTOMER_ORDER_LINE_ITEM_3/STATUS}
expected_ship_date={$PB-BB.CUSTOMER_ORDER_LINE_ITEM_3/EXPECTED_SHIP_DATE}>
</line_item>
            }
          </order>
        </orders>
      </customer>
    }
  </customers>
```

2. In the Query Parameter panel on the Test tab, set the variable values for *customer_id* and *date1* to submit to the query when the query runs.

For example:

- *customer_id*: CUSTOMER_1 (CUSTOMER_1 through CUSTOMER_10 are available in the data source.)

- `date1`: 2002-08-01 (You can enter any date in the form yyyy-MM-dd.)
3. Click the “Run query” button to run the query against the data sources.

Ex 3: Step 9. Verify the Result

Running this query with `customer_id` set to “CUSTOMER_1” and `date1` set to “2002-08-01” produces the following XML query result.

Listing 9-10 Result for Example 3: Date and Time Duration

```
<customers>
  <customer>
    <first_name>JOHN_B_1</first_name>
    <last_name>KAY_1</last_name>
    <orders>
      <order>
        <line_item expected_ship_date="2002-03-06-08:00" id="LINE_ID_1" product="RBBC01"
status="OPEN"/>
        <line_item expected_ship_date="2002-03-06-08:00" id="LINE_ID_3" product="BN16"
status="OPEN"/>
        <line_item expected_ship_date="2002-03-06-08:00" id="LINE_ID_5" product="CS100"
status="OPEN"/>
        <line_item expected_ship_date="2002-03-06-08:00" id="LINE_ID_1" product="RBBC01"
status="OPEN"/>
        <line_item expected_ship_date="2002-03-06-08:00" id="LINE_ID_3" product="BN16"
status="OPEN"/>
        <line_item expected_ship_date="2002-03-06-08:00" id="LINE_ID_5" product="CS100"
status="OPEN"/>
      </order>
    </orders>
  </customer>
</customers>
```

Example 4: Union

A union query is equivalent to concatenating two or more subordinate queries, and pooling the query results into the same output. There are two important rules for a union query.

- Each subordinate query produces a result directed at a repeatable target schema node that is not shared (parent or child) with any other subordinate query target.
- You cannot specify any conditions across these subordinate queries.

The Problem

For any Broadband Customer ID, list any Broadband and Wireless orders. Assume the Customer IDs match across databases.

The Solution

This query requests a union of Broadband orders and Wireless orders. Remember that a union retrieves data from multiple sources, such as the Broadband and Wireless databases, but there are no conditions for the query. If you specify any condition, such as matching order dates, then you are creating a join query. In this example, you need a target schema that contains a repeatable list of Customer IDs, and within that list, a repeatable list of orders. Then you will clone the orders element, using one element for Broadband orders and the other element for Wireless orders.

To create the solution, follow these steps:

- [View a Demo](#)
- [Ex 4: Step 1. Verify the Target Schema is Saved in Repository](#)
- [Ex 4: Step 2. Open Source and Target Schemas](#)
- [Ex 4: Step 3. Clone the Orders Element of the Target Schema](#)
- [Ex 4: Step 4. Create a Query Parameter for a Customer ID](#)

- [Ex 4: Step 5. Assign a Query Parameters](#)
- [Ex 4: Step 6. Define Source Relationships](#)
- [Ex 4: Step 7. Project the Output to the Target Schema](#)
- [Ex 4: Step 8. Add Optimization Hints](#)
- [Ex 4: Step 9. View the XQuery and Run the Query to Test it](#)
- [Ex 4: Step 10. Verify the Result](#)

View a Demo

Union Demo... If you are looking at this documentation online, you can click the “Demo” button to see a viewlet demo showing how to build the conditions and create the mappings described in this example. This demo previews the steps described in detail in the following sections. The demo assumes you already have the target schema in the server Repository.

Ex 4: Step 1. Verify the Target Schema is Saved in Repository

For this example, we will use a target schema called `unionOrders.xsd`. This schema is available in the Samples server repository. The path to the schemas folder in the Liquid Data server repository is:

```
ld_repository/schemas/
```

See “[Example directories](#)” on page 9-2 for information on how example directory names are used.

Just in case you want to verify that you have the right schema file, the following code listing shows the XML for this schema.

Listing 9-11 XML Source for `unionOrders.xsd` Target Schema File

```
<?xml version = "1.0" encoding = "UTF-8"?>
<!--Generated by Data View Builder 1.1. Conforms to w3c
http://www.w3.org/2001/XMLSchema-->
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema" >
  <xsd:element name="customers">
    <xsd:complexType>
      <xsd:sequence>
```

```
<xsd:element name="customer" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="first_name" type="xsd:string"/>
      <xsd:element name="last_name" type="xsd:string"/>
      <xsd:element name="state" type="xsd:string"/>
      <xsd:element name="orders" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="order" minOccurs="0" maxOccurs="unbounded">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="date" type="xsd:string"/>
                  <xsd:element name="amount" type="xsd:decimal"/>
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

Ex 4: Step 2. Open Source and Target Schemas

1. In the Data View Builder, choose File—>New Project to clear your desktop and reset all default values.
2. On the Builder Toolbar Design—>Sources tab, click Relational Databases and open two data sources:
 - Double-click on the PB-WL (Wireless) relational database to open the schema for this data source.
 - Double-click on the PB-BB (Broadband) relational database to open the schema for this data source.
3. Choose the menu option File—>Set Target Schema.
4. Navigate to the server Repository. Choose `unionOrders.xsd` and click Open. `unionOrders.xsd` appears as the target schema.

This target schema is displayed as a docked schema window on the right side of the design area.

5. Click the plus (+) sign (or right-mouse click and choose Expand) to expand the nodes in each source schema and in the target schema.

Ex 4: Step 3. Clone the Orders Element of the Target Schema

1. In the Data View Builder, select the Orders element of the target schema and right-mouse click. The Orders element is a child of the Customers element and has a child called Order.
2. Choose Clone from the right-mouse menu.
The Cloned element labeled Orders(2) appears.

Ex 4: Step 4. Create a Query Parameter for a Customer ID

Create a Query Parameter variable, *customer_id*, that you can use to insert as a variable for a Broadband customer ID value when the query runs. To create this parameter, do the following:

1. On the Builder Toolbar, click Toolbox and then click Query Parameter.
2. From the “Type” drop-down menu, choose `xs:string`.
3. In Parameter Name field, enter `customer_id` and click Add.

The new parameter is displayed in the Query Parameters tree.

Ex 4: Step 5. Assign a Query Parameters

- Assign the query parameter *customer_id* to the Broadband customer ID as follows:

Drag and drop query parameter *customer_id* to the [PB-BB]/db/CUSTOMER/CUSTOMER_ID node.

Ex 4: Step 6. Define Source Relationships

1. Within PB-BB, join the Broadband Customer ID to the Order Customer ID.

Drag and drop [PB-BB]/db/CUSTOMER/CUSTOMER_ID onto Broadband [PB-BB]/db/CUSTOMER_ORDER/CUSTOMER_ID.

2. Join the Broadband customer ID from the Broadband Customer table with the Wireless customer ID from the Wireless Customer Order table as follows:

Drag and drop [PB-BB]/db/CUSTOMER/CUSTOMER_ID onto [PB-WL]/db/CUSTOMER_ORDER/CUSTOMER_ID.

Ex 4: Step 7. Project the Output to the Target Schema

1. Project the Broadband order information.
 - Drag and drop [PB-BB]/db/CUSTOMER_ORDER/TOTAL_ORDER_AMOUNT onto [UnionOrders.xsd]/customers/customer/orders(1)/order/amount.
 - Drag and drop [PB-BB]/db/CUSTOMER_ORDER/ORDER_DATE onto [UnionOrders.xsd]/customers/customer/orders(1)/order/date.
2. Project the Wireless (PB-WL) order information.
 - Drag and drop [PB-WL]/db/CUSTOMER_ORDER/TOTAL_ORDER_AMOUNT onto [UnionOrders.xsd]/customers/customer/orders(2)/order/amount.
 - Drag and drop [PB-WL]/db/CUSTOMER_ORDER/ORDER_DATE onto [UnionOrders.xsd]/customers/customer/orders(2)/order/date.
3. Project the Broadband user information.
 - Drag and drop [PB-BB]/db/CUSTOMER/FIRST_NAME onto [unionOrders.xsd]/customers/customer/first_name.
 - Drag and drop [PB-BB]/db/CUSTOMER/LAST_NAME onto [unionOrders.xsd]/customers/customer/last_name.
 - Drag and drop [PB-BB]/db/CUSTOMER/STATE onto [unionOrders.xsd]/customers/customer/state.

Ex 4: Step 8. Add Optimization Hints

Because you know that the Customer table is much smaller than the Customer Orders table, you can add optimization hints to improve query performance.

1. Click the Optimize tab in the Data View Builder.
2. On the Join Pair Hints panel, choose All from the drop-down list to display all of the join conditions.
3. For both of the join pairs, select Pass Parameter to Right. You pass the parameter to the right because the right table (the Customer Orders table) is much larger than the left table (the Customer table).

Ex 4: Step 9. View the XQuery and Run the Query to Test it

1. Click on the Test tab.

The generated XQuery for this query is shown in the following code listing.

Listing 9-12 XQuery for Example 4: Union

```
{--Generated by Data View Builder 1.1--}
<customers>
  {
    for $PB_BB.CUSTOMER_1 in document("PB-BB")/db/CUSTOMER
    where ($#customer_id of type xs:string eq $PB_BB.CUSTOMER_1/CUSTOMER_ID)
    return
      <customer>
        <first_name>{ xf:data($PB_BB.CUSTOMER_1/FIRST_NAME) }</first_name>
        <last_name>{ xf:data($PB_BB.CUSTOMER_1/LAST_NAME) }</last_name>
        <state>{ xf:data($PB_BB.CUSTOMER_1/STATE) }</state>
        <orders>
          {
            for $PB_BB.CUSTOMER_ORDER_2 in document("PB-BB")/db/CUSTOMER_ORDER
            where ($PB_BB.CUSTOMER_1/CUSTOMER_ID eq {--! ppright !--}
              $PB_BB.CUSTOMER_ORDER_2/CUSTOMER_ID)
            return
              <order>
                <date>{ cast as xs:string(xf:data($PB_BB.CUSTOMER_ORDER_2/ORDER_DATE))
                  }</date>
                <amount>{ xf:data($PB_BB.CUSTOMER_ORDER_2/TOTAL_ORDER_AMOUNT) }</amount>
              </order>
          }
        </orders>
      <orders>
        {
          for $PB_WL.CUSTOMER_ORDER_3 in document("PB-WL")/db/CUSTOMER_ORDER
          where ($PB_BB.CUSTOMER_1/CUSTOMER_ID eq {--! ppright !--}
            $PB_WL.CUSTOMER_ORDER_3/CUSTOMER_ID)
          return
            <order>
              <date>{ cast as xs:string(xf:data($PB_WL.CUSTOMER_ORDER_3/ORDER_DATE))
```

```
    }</date>
    <amount>{ xf:data($PB_WL.CUSTOMER_ORDER_3/TOTAL_ORDER_AMOUNT) }</amount>
  </order>
}
</orders>
</customer>
}
</customers>
```

2. In the Query Parameter panel, click into the cell under “Value” and enter a value for `customer_id`. (CUSTOMER_1 through CUSTOMER_10 are available to try.)
3. Click the “Run query” button to run the query against the data sources.

Ex 4: Step 10. Verify the Result

Querying these data sources as described in this example produces an XML query result similar to that shown in the following code listing where CUSTOMER_4 was used as the query parameter value for `customer_id`.

Listing 9-13 Result for Example 4: Union

```
<customers>
  <customer>
    <first_name>JOHN_B_4</first_name>
    <last_name>KAY_4</last_name>
    <state>NV</state>
    <orders>
      <order>
        <date>2002-03-06-08:00</date>
        <amount>1000</amount>
      </order>
      <order>
        <date>2002-03-06-08:00</date>
        <amount>1500</amount>
      </order>
      <order>
        <date>2002-03-06-08:00</date>
        <amount>2000</amount>
      </order>
      <order>
        <date>2002-03-06-08:00</date>
```

```
    <amount>2500</amount>
  </order>
  <order>
    <date>2002-03-06-08:00</date>
    <amount>3000</amount>
  </order>
</orders>
<orders>
  <order>
    <date>2002-03-06-08:00</date>
    <amount>1000</amount>
  </order>
  <order>
    <date>2002-03-06-08:00</date>
    <amount>2000</amount>
  </order>
  <order>
    <date>2002-03-06-08:00</date>
    <amount>4000</amount>
  </order>
  <order>
    <date>2002-03-06-08:00</date>
    <amount>5000</amount>
  </order>
  <order>
    <date>2002-03-06-08:00</date>
    <amount>10000</amount>
  </order>
</orders>
</customer>
</customers>
```

Example 5: Minus

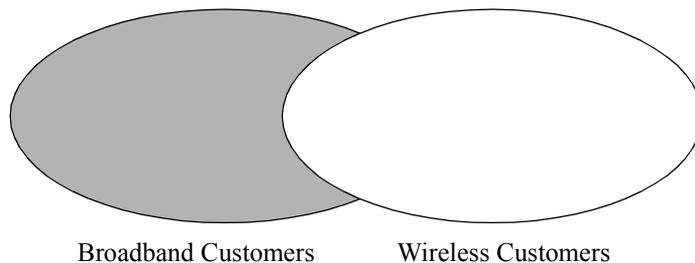
A minus relationship (A minus B) returns all instances of some named value that are in A but not in B. There is no explicit minus operation in the XQuery language or Data View Builder; however, a simple compare and count technique can be used. For example: for each instance of the named value in A, count all matching instances in B; if the count is zero, that means there are no matches, and the query therefore returns the instance from A.

The Problem

Find all customers that are Broadband customers, but not Wireless customers. Assume that Customer IDs match across databases.

The shaded area in [Figure 9-4](#) represents the Broadband customers who are not Wireless customers.

Figure 9-4 Broadband and Wireless Customers



The Solution

If a customer has only a Broadband account, then a join across the Broadband and Wireless databases on that Customer ID produces an empty result. We can take advantage of that fact by counting the number of instances produced by the join. If the number is zero, then the Customer ID represents a Broadband-only customer.

To create the solution, follow these steps:

- [View a Demo](#)
- [Ex 5: Step 1. Verify the Target Schema is Saved in Repository](#)
- [Ex 5: Step 2. Open Source and Target Schemas](#)
- [Ex 5: Step 3. Find Broadband and Wireless Customers with the Same Customer ID](#)
- [Ex 5: Step 4. Find the Count of the Wireless Customers](#)

- [Ex 5: Step 5. Set a Condition that Specifies the Output of “count” is Zero](#)
- [Ex 5: Step 6. View the XQuery and Run the Query to Test it](#)
- [Ex 5: Step 7. Verify the Result](#)

View a Demo

Minus Demo... If you are looking at this documentation online, you can click the “Demo” button to see a viewlet demo showing how to build the conditions and create the mappings described in this example. This demo previews the steps described in detail in the following sections. The demo assumes you already have the target schema in the server Repository.

Ex 5: Step 1. Verify the Target Schema is Saved in Repository

For this example, we will use a target schema called `minus.xsd`. This schema is available in the Samples server repository. The path to the schemas folder in the Liquid Data server repository is:

```
ld_repository/schemas/
```

See [“Example directories” on page 9-2](#) for information on how example directory names are used.

Just in case you want to verify that you have the right schema file, the following code listing shows the XML for this schema.

Listing 9-14 XML Source for minus.xsd Target Schema File

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema" >
  <xsd:element name="results">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="CUSTOMER" minOccurs="1" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="FIRST_NAME" type="xsd:string"/>
              <xsd:element name="LAST_NAME" type="xsd:string"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

Ex 5: Step 2. Open Source and Target Schemas

1. In the Data View Builder, choose File—>New Project to clear your desktop and reset all default values.
2. On the Builder Toolbar Design—>Sources tab, click Relational Databases and open two data sources:
 - Double-click on the PB-WL (Wireless) relational database to open the schema for this data source.
 - Double-click on the PB-BB (Broadband) relational database to open the schema for this data source.
3. Choose the menu option File—>Set Target Schema.

Navigate to the server Repository or to the location where you saved the `minus.xsd` schema. Choose `minus.xsd` and click Open.

`minus.xsd` appears as the target schema.

This target schema is displayed as a docked schema window on the right side of the design area.

4. Click the plus (+) sign (or right-mouse click and choose Expand) to expand the nodes in each source schema and in the target schema.

Ex 5: Step 3. Find Broadband and Wireless Customers with the Same Customer ID

- Drag and drop [PB-BB]/db/CUSTOMER/CUSTOMER_ID onto [PB-WL]/db/CUSTOMER/CUSTOMER_ID to join the Broadband CUSTOMER_ID and the Wireless CUSTOMER_ID.

Ex 5: Step 4. Find the Count of the Wireless Customers

1. On the Builder Toolbar Design—>Toolbox tab, click Functions and double-click on the `xf:count` function (under Aggregate functions) to open it.
2. Drag and drop the `[PB-WL]/db/CUSTOMER/CUSTOMER_ID` onto the input of the `xf:count` function.

Ex 5: Step 5. Set a Condition that Specifies the Output of “count” is Zero

1. Click on the Conditions tab.
2. Drag and drop the `eq` (equal) function (in the XQuery functions Comparison operators folder) onto the next empty row under Conditions on the Conditions tab.

The Functions Editor is displayed.

3. For the first parameter, drop `[count-Function:Output]/Parameters/integer` onto *anyValue1*.
4. For the second parameter, create a Number constant, set it to 0 and drop it on *anyValue2*.

Note: To create the Number constant, on Builder—>Toolbox tab, click Constants, enter 0 in the Number field, and drag the Constant icon next to that field onto *anyValue2* in the equation in the Functions Editor.

The equality condition should look like this:

```
([xf:count]/result eq 0)
```

Close the Functions Editor.

5. Project the Broadband customers to the target results.
 - Drag and drop `[PB-BB]/db/CUSTOMER/FIRST_NAME` onto `[minus.xsd]/results/CUSTOMER/FIRST_NAME`.
 - Drag and drop `[PB-BB]/db/CUSTOMER/LAST_NAME` onto `[minus.xsd]/results/CUSTOMER/LAST_NAME`.

Ex 5: Step 6. View the XQuery and Run the Query to Test it

1. Click on the Test tab.

The generated XQuery for this query is shown in the following code listing.

Listing 9-15 XQuery for Example 5: Minus

```
{--Generated by Data View Builder 1.1--}
<results>
  {
    for $PB_BB.CUSTOMER_1 in document("PB-BB")/db/CUSTOMER
    let $srcval_2 :=
      for $PB_WL.CUSTOMER_3 in document("PB-WL")/db/CUSTOMER
      where ($PB_BB.CUSTOMER_1/CUSTOMER_ID eq $PB_WL.CUSTOMER_3/CUSTOMER_ID)
      return
        xf:data($PB_WL.CUSTOMER_3/CUSTOMER_ID)
    let $xf:count_4 := xf:count($srcval_2)
    where ($xf:count_4 eq 0)
    return
      <CUSTOMER>
        <FIRST_NAME>{ xf:data($PB_BB.CUSTOMER_1/FIRST_NAME) }</FIRST_NAME>
        <LAST_NAME>{ xf:data($PB_BB.CUSTOMER_1/LAST_NAME) }</LAST_NAME>
      </CUSTOMER>
  }
</results>
```

2. Click the “Run query” button to run the query against the data sources.

Ex 5: Step 7. Verify the Result

When you run this query on the sample data sources as described here, the result will be one record because the sample Broadband data source has one customer record that is different from the Wireless customer records.

Listing 9-16 Result for Example 5: Minus

```
<results>
  <CUSTOMER>
    <FIRST_NAME>JOHN</FIRST_NAME>
    <LAST_NAME>PARKER</LAST_NAME>
  </CUSTOMER>
</results>
```

Example 6: Complex Parameter Type (CPT)

The Complex Parameter Type Cookbook example shows how to use Liquid Data to create an integrated view that connects two enterprise information systems: a database and an in-flight XML data source using a complex parameter type (CPT). A query that uses both data sources determines whether the customer has sufficient credit for the incoming order to be processed.

The Problem

The company receives dozens of electronically transmitted orders daily and needs to quickly respond to its field office if an order cannot be accepted because a customer has exceeded their credit limit. The credit limit and amount of outstanding orders is known to the system. The quantity and price of the items being ordered is supplied in real-time along with the order.

The Solution

The company develops a complex parameter type (CPT) that models the incoming purchase order as an XML schema and sets a simple `orderLimit` parameter that an operator can change whenever the query is run. The query also calculates the total amount outstanding of current orders and the total amount of the incoming order. The objective is to accept orders if the total amount of both outstanding and incoming orders is within the order limit. Otherwise, the order is rejected.

To recreate the solution, follow these steps:

- [View a Demo](#)
- [Ex 6: Step 1. Verify the Availability of Schemas and Sample Data Stream](#)
- [Ex 6: Step 2. Open the Target Schema and CO-CPTSAMPLE CPT](#)
- [Ex: 6: Step 3. Create an orderLimit Query Parameter](#)
- [Ex 6: Step 4. Save the Project](#)

- Ex 6: Step 5. Test Access to the Complex Parameter Source
- Ex 6: Step 6: Determine the Total Amount of New Orders
- Ex 6: Step 7. Create the Necessary Joins and Mappings to the Target Schema
- Ex 6: Step 8. Determine the Amount of Currently Open Orders
- Ex 6: Step 9: Determine the Total Amount of All Open and New Orders
- Ex 6: Step 10: Test If Open Orders + New Orders Exceeds the Order Limit
- Ex 6: Step 11: Determine If the Order is Accepted or Rejected
- Ex 6: Step 12: View the XQuery
- Ex 6: Step 13. Run the XQuery to Verify the Result

Note: The implementation details of the Complex Parameter Type demo, the DB-COCPT sample, and the CPT cookbook example vary slightly.

View a Demo

Complex Parameter Type (CPT) Demo. If you are looking at this documentation online, you can click the “Demo” button to see a viewlet demo showing how to build the conditions and create the mappings described in this example.

Ex 6: Step 1. Verify the Availability of Schemas and Sample Data Stream

In creating the DB-CPTCO sample query, we use the following files that are installed with Liquid Data samples. (See “[Example directories](#)” on page 9-2 for information on how example directory names are used.)

From the Liquid Data samples repository schema directory:

- **BroadBand database schema.** The path to this file is:

`ld_repository/schemas/broadbandp.sql`

- **Complex parameter type schema.** The path to this file is:

`ld_repository/schemas/coCptSample2.xsd`

- **Target schema.** The path to this file is:

`ld_repository/schemas/COCPTSampleTarget-Schema.xsd`

■ **CPT sample XML stream.** The path to this file is:

`ld_repository/xml_files/coCPTsample2.xml`

If you want to refer to the sample DB-CPTCO project, it is installed as the following file:

`<BEA_HOME>/liquiddata/samples/buildQuery/db-cptco/coCPTSample.qpr`

For reference purposes, code listings for several of the XML files used in this example appear below:

Listing 9-17 DB-CPTCO Sample CPT Schema (`coCptSample2.xsd`)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="urn:schemas-bea-com:ld-cocpt"
  xmlns:cocpt="urn:schemas-bea-com:ld-cocpt"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="CustOrder">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="CUSTOMER_ORDER" minOccurs="0" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="CUSTOMER_ID" type="xsd:string"/>
              <xsd:element name=
                "NEW_ORDER_LINE_ITEM" type="cocpt:NEW_ORDER_LINE_ITEMType"
                minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="NEW_ORDER_LINE_ITEMType">
    <xsd:sequence>
      <xsd:element name="PRODUCT_NAME" type="xsd:string"/>
      <xsd:element name="QUANTITY" type="xsd:decimal"/>
      <xsd:element name="PRICE" type="xsd:decimal"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Listing 9-18 DB-CPTCO Target Schema (COCPSTSampleTargetSchema.xsd)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="urn:schemas-bea-com:ld-cocpt"
xmlns:cocpt="urn:schemas-bea-com:ld-cocpt"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="CustOrder">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="CUSTOMER_ORDER" minOccurs="0" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="CUSTOMER_ID" type="xsd:string"/>
              <xsd:element name="NEW_ORDER_LINE_ITEM"
type="cocpt:NEW_ORDER_LINE_ITEMType" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="NEW_ORDER_LINE_ITEMType">
    <xsd:sequence>
      <xsd:element name="PRODUCT_NAME" type="xsd:string"/>
      <xsd:element name="QUANTITY" type="xsd:decimal"/>
      <xsd:element name="PRICE" type="xsd:decimal"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Listing 9-19 DB-CPTCO Sample XML Data Stream (coCptSample2.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<cocpt:CustOrder xmlns:cocpt="urn:schemas-bea-com:ld-cocpt"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-bea-com:ld-cocpt
coCptSample2.xsd">
  <CUSTOMER_ORDER>
    <CUSTOMER_ID>CUSTOMER_1</CUSTOMER_ID>
    <NEW_ORDER_LINE_ITEM>
      <PRODUCT_NAME>RBBC01</PRODUCT_NAME>
      <QUANTITY>1000</QUANTITY>
      <PRICE>20</PRICE>
    </NEW_ORDER_LINE_ITEM>
    <NEW_ORDER_LINE_ITEM>
      <PRODUCT_NAME>CS2610</PRODUCT_NAME>
      <QUANTITY>1000</QUANTITY>
      <PRICE>20</PRICE>
    </NEW_ORDER_LINE_ITEM>
  </CUSTOMER_ORDER>
</cocpt:CustOrder>
```

```
</CUSTOMER_ORDER>  
</cocpt:CustOrder>
```

You may also want to examine the CO-CPTSAMPLE definition in the WLS Administration Console for the Samples server you are using.

1. Login to the Administration Server. See [Start the WLS Administration Console](#) in the Getting Started guide for details.
2. In the left pane, click the Liquid Data node.
3. In the right pane, click the Complex Parameter Types tab and click on CO-CPTSAMPLE.

See the section “[Creating a Complex Parameter Type](#)” on page 7-8 for details.

Ex 6: Step 2. Open the Target Schema and CO-CPTSAMPLE CPT

1. In the Data View Builder, choose `File`→`New Project`.
2. On the Design tab, on the Builder Toolbar, click the Toolbox tab, then click Complex Parameter Type. The CO-CPTSAMPLE complex parameter type is listed.
3. Double-click on CO-CPTSAMPLE to open the CPT schema. Right click on the top element (`cocpt:CustOrder`) to expand.
4. Choose `File`→`Set Target Schema`. Browse to the Liquid Data repository schema directory.
5. Select the following file as the target schema:

```
ld_repository/schemas/COCPSTsampleTargetSchema.xsd
```

In the target schema window on the right side of the design area, right-click on the top element expand the target schema.

Ex: 6: Step 3. Create an orderLimit Query Parameter

Since credit limits vary from customer to customer, it is convenient to have an order limit query parameter that can be changed whenever a query is run.

1. In the Data View Builder select `Design`→`Toolbox`.

2. Click on the Query Parameter tab.
3. Enter `orderLimit` as a parameter name.
4. Select `xs:decimal` as the parameter type.
5. Click Add.
6. Drag and drop the `orderLimit` parameter icon onto the target schema `[COCPSTSampleTargetSchema.xsd]/cocpt:CustOrderStatus/CUSTOMER/CUSTOMER_ORDER/TOTAL_OPEN_ORDERLIMIT`.

Ex 6: Step 4. Save the Project

You can save a project at any time. To initially create a project, use `File→Save Project As`. Use the file browser to choose a location and project name (we use `myCoCPT`).

Ex 6: Step 5. Test Access to the Complex Parameter Source

Follow these steps to verify access to the CPT data source:

1. Drag and drop output from `[CO-CPTSAMPLE]/cocpt:CustOrder/CUSTOMER_ORDER/NEW_ORDER_LINE_ITEM/CUSTOMER_ID` onto the target schema `[COCPSTSampleTargetSchema.xsd]/cocpt:CustOrderStatus/CUSTOMER/CUSTOMER_ID`.
2. Click the Test tab.
3. In the lower-left pane of the Data View Builder (Test mode), click in the Values area under Query Parameters to the right of the CPT name (`CO-CPTSAMPLE`).
4. Navigate to the XML data file associated with the `CO-CPTSAMPLE` complex parameter type.

```
ld_repository/xml_files/coCptSample2.xml
```
5. Enter an `orderLimit` value such as `200000`.
6. Now we can click the Run button to execute a preliminary query. The following result shows that your CPT is successfully retrieving from the XML file data:

Listing 9-20 Interim Results (1) from CPT Example Query

```
<cocpt:CustOrderStatus xmlns:cocpt="urn:schemas-bea-com:ld-cocpt">
  <CUSTOMER>
    <CUSTOMER_ID>CUSTOMER_1</CUSTOMER_ID>
    <CUSTOMER_ORDER>
      <TOTAL_OPEN_ORDERLIMIT>200000</TOTAL_OPEN_ORDERLIMIT>
    </CUSTOMER_ORDER>
  </CUSTOMER>
</cocpt:CustOrderStatus>
```

7. Return to the Toolbar Design mode.

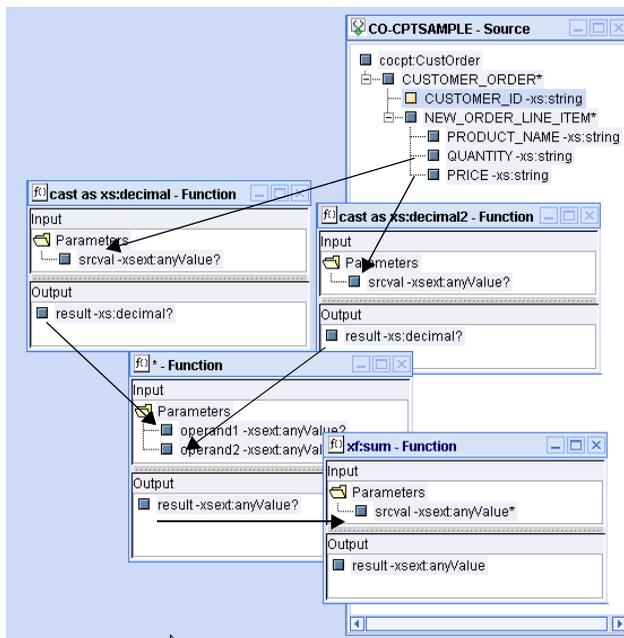
In the case of this data source, the customer identification is provided so there is no need to create a customer ID query parameter.

Ex 6: Step 6: Determine the Total Amount of New Orders

Since all runtime source items from a CPT are treated as character strings, any data items from the CO-CPTSAMPLE data source must first be cast appropriately. Then quantities and prices are multiplied together. The sum of the products of quantity and prices is the total amount of new orders.

1. Click `xQuery Functions`.
2. Drag and drop (or double-click) two `cast as xs:decimal` functions into the design area. These are labeled as `xs:decimal` and `xs:decimal2`.
3. Drag and drop a multiply (*) function into the design area.
4. Drag and drop a sum aggregate function into the design area.

Figure 9-5 Functions Used to Calculate Total New Orders in Data Stream



5. Drag and drop
[CO-CPTSAMPLE]/cocpt:CustOrder/CUSTOMER_ORDER/NEW_ORDER_LINE_ITEM/QUANTITY to become the input parameter to `xs:decimal`.
6. Drag and drop
[CO-CPTSAMPLE]/cocpt:CustOrder/CUSTOMER_ORDER/NEW_ORDER_LINE_ITEM/PRICE to become the input parameter to `xs:decimal2`.
7. Drag and drop the output result of `xs:decimal` to one side of the multiply equation and the output result of `xs:decimal2` to the other.
8. Drag the * - Function output result to the input parameter of `xf:sum`. This gives us the total order amount in the CPT data source.
9. Drag and drop output from `xf:sum` onto the target schema
[COCPTSampleTargetSchema.xsd]/cocpt:CustOrderStatus/CUSTOMER/CUSTOMER_ORDER/NEW_ORDER_TOTAL_AMOUNT.
10. Rerun your query. A new order total of 40,000 appears.

11. Return to Design mode.

The only query construction components we will reuse are `sum` and `CO-CPTSAMPLE` source. The others can be closed or minimized.

Ex 6: Step 7. Create the Necessary Joins and Mappings to the Target Schema

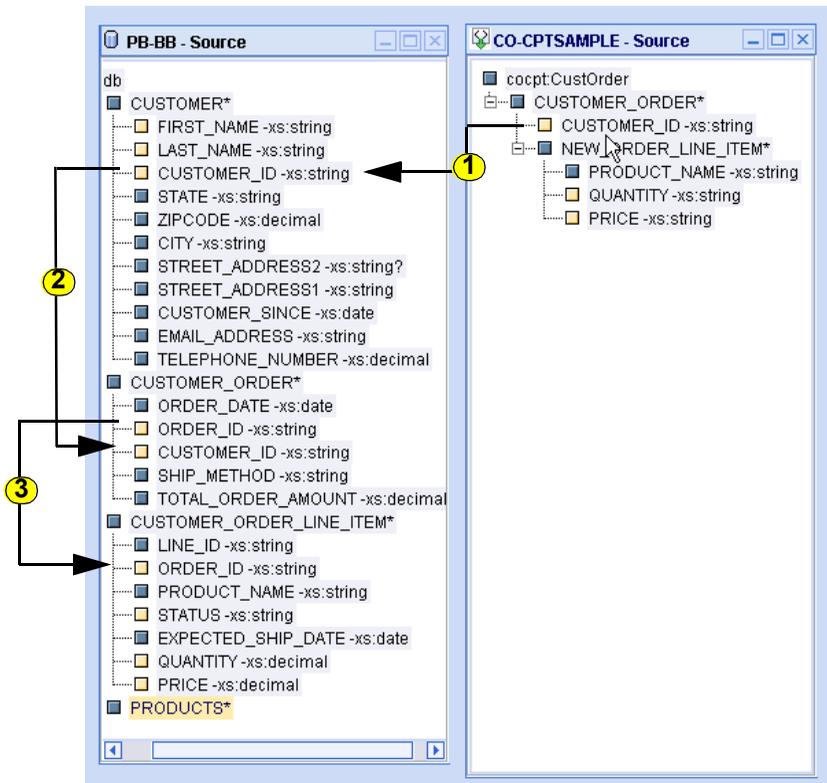
Move the PB-BB relational source schema onto the design area.

1. On the Builder Toolbar Design—>Sources tab, click Relational Databases, and double-click on `PB-BB` to open the schema for the broadband sample data source.
2. Expand the `PB-BB` schema.

Create the necessary joins to allow us to fetch the line items for a particular order for a particular customer.

1. Drag and drop `[CO-CPTSAMPLE]/cocpt:CustOrder/CUSTOMER_ORDER/CUSTOMER_ID` onto `[PB-BB]/db/CUSTOMER/CUSTOMER_ID`.
2. Drag and drop `[PB-BB]/db/CUSTOMER/CUSTOMER_ID` onto the `[PB-BB]/db/CUSTOMER_ORDER/CUSTOMER_ID`.
3. Drag and drop `[PB-BB]/db/CUSTOMER_ORDER/ORDER_ID` onto `[PB-BB]/db/CUSTOMER_ORDER_LINE_ITEM/ORDER_ID`.

Figure 9-6 Setting Joins Between CPT and Relational Data Source



Next, project `FIRST_NAME` and `LAST_NAME` elements onto the target schema.

1. Drag and drop `[PB-BB]/db/CUSTOMER/FIRST_NAME` onto the target schema `[cocptsampltarget-schema]/cocptCustOrderStatus/CUSTOMER/FIRST_NAME`.
2. Drag and drop `[PB-BB]/db/CUSTOMER/LAST_NAME` onto the target schema `[cocptsampltarget-schema]/cocptCustOrderStatus/CUSTOMER/LAST_NAME`.

Note: In this version of the CO-CPTSAMPLE, automatic scoping is used. See [“Understanding Scope in Basic and Advanced Views”](#) on page 3-30 for more information.

Although your query is not complete, you can test run it again.

Listing 9-21 Interim Results (2) from CPT Example Query

```
<cocpt:CustOrderStatus xmlns:cocpt="urn:schemas-bea-com:ld-cocpt">
  <CUSTOMER>
    <FIRST_NAME>JOHN_B_1</FIRST_NAME>
    <LAST_NAME>KAY_1</LAST_NAME>
    <CUSTOMER_ID>CUSTOMER_1</CUSTOMER_ID>
    <CUSTOMER_ORDER>
      <NEWORDER_TOTAL_AMOUNT>40000</NEWORDER_TOTAL_AMOUNT>
      <TOTAL_OPEN_ORDERLIMIT>200000</TOTAL_OPEN_ORDERLIMIT>
    </CUSTOMER_ORDER>
  </CUSTOMER>
</cocpt:CustOrderStatus>
```

Ex 6: Step 8. Determine the Amount of Currently Open Orders

Follow these steps to find the total amount of open orders in the sample PB-BB database:

1. Click the XQuery Functions.
2. Drag and drop the multiply (*) operator into the design area.
3. Drag and drop [PB-BB]/db/CUSTOMER_ORDER_LINE_ITEM/QUANTITY and [PB-BB]/db/CUSTOMER_ORDER_LINE_ITEM/PRICE to the function operands.
4. Drag and drop another `xf:sum` into the design area. Drag the output result of the `f()*2 - Function(*2 being the second use of a multiplication function)` to the input parameter of `xf:sum2`.
5. Drag and drop output from `xf:sum2` onto the target schema [COCPTSampleTargetSchema.xsd]/cocpt:CustOrderStatus/CUSTOMER/CUSTOMER_ORDER/OPEN_ORDER_TOTAL_AMOUNT.

Finally, we need to restrict results to open orders:

1. In the Data View Builder select `Design`→`Toolbox`.
2. Click on `Constants`.
3. Create a string constant called `OPEN` and drag the icon to the right of `OPEN` to the [PB-BB]/db/CUSTOMER_ORDER_LINE_ITEM/STATUS item.

If we run our current query, the amount of open orders should be 150,000.

Ex 6: Step 9: Determine the Total Amount of All Open and New Orders

1. Click `xQuery Functions`.
2. Drag and drop the plus (+) numeric operator into the design area.
3. Use `xf:sum` (total new orders) and `xf:sum2` (total open orders) as the operands to obtain the total open and new order amount.

Ex 6: Step 10: Test If Open Orders + New Orders Exceeds the Order Limit

1. Click on `xQuery Functions`.
2. Under Comparison operators locate the `gt` (greater than) function. Drag it into the design area.
3. Use the output of the sum of open and newly arrived orders as the first input parameter.
4. Use the query parameter `orderLimit` as input for `operand2`. The result is a Boolean value that is True if the sum of open and newly arrived orders is greater than the overall order limit.

Ex 6: Step 11: Determine If the Order is Accepted or Rejected

Now that the relationships and conditions are established, set up an if-then-else test to solve the business problem. See [“The Problem” on page 9-41](#).

1. Under Other functions locate the `xfext:if-then-else` function. Drag it into the design area.
2. Drag the output of the `gt` function to the `if-then-else` input condition parameter.
3. Click on the Toolbox `Constants` tab. In the String field enter `REJECT`, drag the String field icon to the `then` parameter in the `xfext:if-then-else` function.
4. Change the String field to `ACCEPT`, then drag the String field icon to the `else` parameter.

5. Drag the output result onto the target schema [COCPSTSampleTargetSchema.xsd]/cocpt:CustOrderStatus/CUSTOMER/CUSTOMER_ORDER/ORDER_REVIEW_STATUS.

Ex 6: Step 12: View the XQuery

The generated XQuery is shown in the following code listing.

Listing 9-22 XQuery for Example 6: Complex Parameter Type (CPT)

```
{--Generated by Data View Builder 1.1--}
namespace cocpt = "urn:schemas-bea-com:ld-cocpt"
<cocpt:CustOrderStatus>
  {
    for $CO_CPTSAMPLE.CUSTOMER_ORDER_2 in ($#CO-CPTSAMPLE of type element
cocpt:CustOrder)/CUSTOMER_ORDER
      for $MyBroadBand_LD_DS.CUSTOMER_3 in document ("MyBroadBand-LD-DS")/db/CUSTOMER
        let $srcval_4 :=
          for $MyBroadBand_LD_DS.CUSTOMER_ORDER_LINE_ITEM_5 in
document ("MyBroadBand-LD-DS")/db/CUSTOMER_ORDER_LINE_ITEM
          let $srcval_9 :=
            for $MyBroadBand_LD_DS.CUSTOMER_ORDER_10 in
document ("MyBroadBand-LD-DS")/db/CUSTOMER_ORDER
              where ($MyBroadBand_LD_DS.CUSTOMER_ORDER_10/ORDER_ID
eq $MyBroadBand_LD_DS.CUSTOMER_ORDER_LINE_ITEM_5/ORDER_ID)
                and ($MyBroadBand_LD_DS.CUSTOMER_3/CUSTOMER_ID eq
$MyBroadBand_LD_DS.CUSTOMER_ORDER_10/CUSTOMER_ID)
                  return
                    xf:true()
                where xf:not (xf:empty($srcval_9))
                  and ("OPEN" eq $MyBroadBand_LD_DS.CUSTOMER_ORDER_LINE_ITEM_5/STATUS)
                    return
                      $MyBroadBand_LD_DS.CUSTOMER_ORDER_LINE_ITEM_5/QUANTITY *
$MyBroadBand_LD_DS.CUSTOMER_ORDER_LINE_ITEM_5/PRICE
                  let $xf:sum2_11 := xf:sum($srcval_4)
                  let $srcval_12 :=
                    for $CO_CPTSAMPLE.NEW_ORDER_LINE_ITEM_14 in
$CO_CPTSAMPLE.CUSTOMER_ORDER_2/NEW_ORDER_LINE_ITEM
                      let $cast_as_xs:decimal2_17 := cast as
xs:decimal($CO_CPTSAMPLE.NEW_ORDER_LINE_ITEM_14/QUANTITY)
                      let $cast_as_xs:decimal_20 := cast as
xs:decimal($CO_CPTSAMPLE.NEW_ORDER_LINE_ITEM_14/PRICE)
                      return
                        $cast_as_xs:decimal2_17 * $cast_as_xs:decimal_20
                  let $xf:sum_22 := xf:sum($srcval_12)
                  let $v_23 := $xf:sum2_11 + $xf:sum_22
                  let $gt_24 := $v_23 gt $#orderLimit of type xs:decimal
                  where ($CO_CPTSAMPLE.CUSTOMER_ORDER_2/CUSTOMER_ID eq
$MyBroadBand_LD_DS.CUSTOMER_3/CUSTOMER_ID)
```

```
return
<CUSTOMER>
  <FIRST_NAME>{ xf:data($MyBroadBand_LD_DS.CUSTOMER_3/FIRST_NAME) }</FIRST_NAME>
  <LAST_NAME>{ xf:data($MyBroadBand_LD_DS.CUSTOMER_3/LAST_NAME) }</LAST_NAME>
  <CUSTOMER_ID>{ xf:data($CO_CPTSAMPLE.CUSTOMER_ORDER_2/CUSTOMER_ID)
}</CUSTOMER_ID>
  <CUSTOMER_ORDER>
    <OPENORDER_TOTAL_AMOUNT>{ cast as xs:decimal($xf:sum2_11)
}</OPENORDER_TOTAL_AMOUNT>
    <NEWORDER_TOTAL_AMOUNT>{ cast as xs:decimal($xf:sum_22)
}</NEWORDER_TOTAL_AMOUNT>
    <TOTAL_OPEN_ORDERLIMIT>{ $#orderLimit of type xs:decimal
}</TOTAL_OPEN_ORDERLIMIT>
    <ORDER_REVIEW_STATUS>{ cast as xs:string(xfext:if-then-else( treat as
xs:boolean($gt_24), "REJECT", "ACCEPT")) }</ORDER_REVIEW_STATUS>
  </CUSTOMER_ORDER>
</CUSTOMER>
}
</cocpt:CustOrderStatus>
```

Ex 6: Step 13. Run the XQuery to Verify the Result

When you run this query on the sample data sources as described in this example, the result is an accepted order.

Listing 9-23 Result of Example 6: Complex Parameter Type (CPT)

```
<cocpt:CustOrderStatus xmlns:cocpt="urn:schemas-bea-com:ld-cocpt">
  <CUSTOMER>
    <FIRST_NAME>JOHN_B_1</FIRST_NAME>
    <LAST_NAME>KAY_1</LAST_NAME>
    <CUSTOMER_ID>CUSTOMER_1</CUSTOMER_ID>
    <CUSTOMER_ORDER>
      <OPENORDER_TOTAL_AMOUNT>150000</OPENORDER_TOTAL_AMOUNT>
      <NEWORDER_TOTAL_AMOUNT>40000</NEWORDER_TOTAL_AMOUNT>
      <TOTAL_OPEN_ORDERLIMIT>200000</TOTAL_OPEN_ORDERLIMIT>
      <ORDER_REVIEW_STATUS>ACCEPT</ORDER_REVIEW_STATUS>
    </CUSTOMER_ORDER>
  </CUSTOMER>
</cocpt:CustOrderStatus>
```

A 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. (See also “[XQuery Functions](#)” on page 2-11 in Chapter 2, “Starting the Builder and Touring the GUI.”)

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 in Liquid Data XQuery Functions](#)
 - [Naming Conventions](#)
 - [Occurrence Indicators](#)
 - [Data Types](#)
 - [Date and Time Patterns](#)
- [Accessor and Node Functions](#)
- [Aggregate Functions](#)
- [Boolean Functions](#)
- [Cast Functions](#)
- [Comparison Operators](#)
- [Constructor Functions](#)
- [Date and Time Functions](#)
- [Logical Operators](#)
- [Numeric Operators](#)
- [Numeric Functions](#)
- [Other Functions](#)
- [Sequence Functions](#)
- [String Functions](#)
- [Treat Functions](#)

About in Liquid Data XQuery Functions

You can browse the Liquid Data XQuery function in the Data View Builder. The functions are located in 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 `xsect:datatype` notation. When you encounter the `xsect:` 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 A-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.
xsex: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 xsex:anyValue. A value that supports the mathematical concept of binary-valued logic: true or false.
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

Table A-1 Data Types

Data Type Name	Description
xs:date	<p>A subset of <code>xsect:anyValue</code>. Represents the leftmost component of <code>dateTime</code> <code>YYYY-MM-DD</code> where:</p> <ul style="list-style-type: none"> ■ <code>YYYY</code> is the year ■ <code>MM</code> is the month ■ <code>DD</code> is the day <p>May be preceded by a leading minus (-) sign to indicate a negative number. If the sign is omitted, plus (+) is assumed.</p> <p>May be immediately followed by a <code>Z</code> 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 <code>hh:mm</code>.</p> <p>Example:</p> <p>To specify 1:20 pm on May the 31st, 1999, write: <code>1999-05-31</code>.</p>
xs:dateTime	<p>A subset of <code>xsect:anyValue</code>. Represents the format <code>YYYY-MM-DDThh:mm:ss</code> where:</p> <ul style="list-style-type: none"> ■ <code>YYYY</code> is the year ■ <code>MM</code> is the month ■ <code>DD</code> is the day ■ <code>T</code> is the date/time separator ■ <code>hh</code> is the hour ■ <code>mm</code> is the minute ■ <code>ss</code> is the second <p>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 (<code>ss.ss...</code>) with any number of digits after the decimal point is supported.</p> <p>May be immediately followed by a <code>Z</code> 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 <code>hh:mm</code>.</p> <p>Example:</p> <p>To specify 1:20 pm on May the 31st, 1999 EST, which is five hours behind Coordinated Universal Time (UTC), write: <code>1999-05-31T13:20:00-05:00</code>.</p>

Table A-1 Data Types

Data Type Name	Description
xs:decimal	<p>A subset of <code>xsect:anyValue</code>. Includes all integer types, such as <code>xs:integer</code>, <code>xs:long</code>, <code>xs:short</code>, <code>xs:int</code>, or <code>xs:byte</code>.</p> <p>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.</p> <p>Examples: -1.23, 12678967.543233, +100000.00, 210</p>
xs:double	<p>A subset of <code>xsect:anyValue</code>. There are no subordinate data types; however, <code>xs:float</code> and <code>xs:decimal</code>, and all derived types, can be promoted to <code>xs:double</code> in certain cases, such as function calls.</p> <p>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).</p>
xs:float	<p>A subset of <code>xsect:anyValue</code>. There are no subordinate data types; however, <code>xs:decimal</code>, and all derived types, can be promoted to <code>xs:float</code> in certain cases, such as function calls.</p> <p>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).</p>
xsect:item	<p>A subset of <code>xs:anyType</code>. Includes <code>xsect:anyValue</code> and <code>xsect:node</code>. Excludes any sequence. Represents a list element, individual value, or attribute.</p>
xs:int	<p>A subset of <code>xs:long</code>. Represents a finite-length sequence of decimal digits (0–9). An optional leading sign is allowed. If the sign is omitted, plus (+) is assumed.</p> <p>Examples: -1, 0, 12678967543233, +100000</p>
xs:integer	<p>A subset of <code>xs:decimal</code>. Represents a finite-length sequence of decimal digits (0–9). An optional leading sign is allowed. If the sign is omitted, plus (+) is assumed.</p> <p>Examples: -1, 0, 12678967543233, +100000</p>

Table A-1 Data Types

Data Type Name	Description
xs:long	<p>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.</p> <p>Examples: -1, 0, 12678967543233, +100000</p>
xs:node	<p>A subset of xs:node. A component in a tree structure that represents a data element.</p>
xs:short	<p>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.</p> <p>Examples: -1, 0, 12678, +10000</p>
xs:string	<p>A subset of xs:string. A sequence that contains alphabetic, numeric, or special characters.</p>
xs:time	<p>A subset of xs:string. Represents the rightmost segment of the dateTime format where:</p> <ul style="list-style-type: none"> ■ <i>hh</i> is the hour ■ <i>mm</i> is the minute ■ <i>ss</i> is the second <p>May contain an optional following time zone indicator.</p> <p>Examples:</p> <ul style="list-style-type: none"> ■ 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 9-3 Date and Time Patterns

This Symbol	Represents This Data	Produces This Result
G	Era	AD
y	Year	1996
M	Month of year	July, 07
d	Day of the month	19
h	Hour of the day (1–12)	10
H	Hour of the day (0–23)	22
m	Minute of the hour	30
s	Second of the minute	55
S	Millisecond	978
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 functions 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: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: `xsect:node?`
- Returned data type: `xsect: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})` returns the numeric value 3.
 - `xf:data(<a/>)` returns an empty list ().
 - `xf:data((<a>{3}, <a>{7}))` 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: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: *xsect: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: *xsex:item**
- Returned data type: *xsex: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}, 3, (), {2}))` returns `<a>{4}`.
-

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}, 3, (), {2}))` returns `{2}`.
 - `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: *xsect:anyValue**
- Returned data type: *xsect: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}, 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 successfully 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 July 23rd, 2002.
 - 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), -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

- 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), -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: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 `xsex: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) 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: *xsect:anyValue?*
- Parameter2 data type: *xsect: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: *xs:ext:anyValue?*
- Parameter2 data type: *xs:ext: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” operator (previous entry in this table).

gt

Returns true if `Parameter1` is greater than `Parameter2`.

Data Types

- `Parameter1` data type: *xsect:anyValue?*
- `Parameter2` data type: *xsect: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: *xsect:anyValue?*
- `Parameter2` data type: *xsect: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).

lt

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

Data Types

- Parameter1 data type: *xsect:anyValue?*
- Parameter2 data type: *xsect: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).

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: *xsect:boolean?*
- Parameter2 data type: *xsect: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 the “eq” operator (previous entry in this table).

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: *xsect: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: *xsect: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: *xsect: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: *xsex: 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: *xsect: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: *xsect: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: *xsect: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 `xsex: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)` 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`
- `xfext:date-from-dateTime`
- `xfext:date-from-string-with-format`
- `xf:dateTime`
- `xfext:dateTime-from-string-with-format`
- `xf:get-hours-from-dateTime`
- `xf:get-hours-from-time`
- `xf:get-minutes-from-dateTime`
- `xf:get-minutes-from-time`
- `xf:get-seconds-from-dateTime`
- `xf:get-seconds-from-time`
- `xf:time`
- `xfext:time-from-dateTime`
- `xfext:time-from-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

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.
-

xfext:date-from-dateTime

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 A-3](#). For more information about valid formats for `dateTime`, see [“xf:dateTime” on page A-56](#).

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 A-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 A-3](#).

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.
-

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-DDT+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
-

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 A-3](#), and see [“Date and Time Patterns” on page A-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.
-

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-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:time

Returns a time 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: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 A-3](#). For more information about valid formats for `dateTime`, see [“xf:dateTime” on page A-56](#).

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 A-3](#), and see [“Date and Time Patterns” on page A-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.
-

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/>,)` 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/>, ` 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), -0, 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), -0, 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), -0, 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), -0, 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), -0, 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 hundredths, 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 hundredths, 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 A-3](#).

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

- If `(xf:true())` then 3 else "10" returns the value 3.
- If `(xf:false())` then 3 else "10" returns the value "10."
- If `("true")` then 3 else "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](#)
- [xf:empty](#)
- [xf:subsequence \(format 1\)](#)
- [xf:subsequence \(format 2\)](#)

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: *xsect:item**
- Returned data type: *xsect: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 sequence ("a", "b", "c").
- `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: *xsect: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 sequence ("b", "c", "d", "e").
 - `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: *xsect:item**
- Parameter2 data type: *xs:integer*
- Parameter3 data type: *xs:integer*
- Returned data type: *xsect: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 sequence ("b", "c", "d", "e").
- `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. 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 \(format 1\)](#)
- [xf:substring \(format 2\)](#)
- [xf:substring-after](#)
- [xf:substring-before](#)
- [xf:upper-case](#)
- [xfext:match](#)
- [xfext:trim](#)

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 `xsex: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 9-4 Regular expression syntax examples for the `xfext:match` function

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

Table 9-4 Regular expression syntax examples for the xtext:match function

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 9-4 Regular expression syntax examples for the `xfext:match` function

Syntax Example	Description
Logical Operators	
<code>AB</code>	Matches expression A followed by expression B.
<code>A B</code>	Matches expression A or expression B.
<code>(A)</code>	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("abccdee", "[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 A-3](#).

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

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 between the 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](#).

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](#).

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](#).

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](#).

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](#).

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

This function uses the `xf:double` 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.
-

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](#).

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

This function uses the `xf:int` function.

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](#).

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](#).

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](#).

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](#).

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](#).

XQuery Specification Compliance

Conforms to the current specification.

B Supported Data Types

This section provides information about the data types supported in BEA Liquid Data for WebLogic™ and the Data View Builder. The following topics are included:

- [Overview](#)
 - [JDBC Types](#)
 - [JDBC Names](#)
- [Database-Specific Names](#)
 - [Oracle Names](#)
 - [Microsoft SQL Server Names](#)
 - [DB2 Names](#)
 - [Sybase Names](#)
 - [Informix Names](#)

Overview

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, and 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.

B Supported Data Types

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.

JDBC Types

The following table maps the JDBC type to the appropriate data type that you should use with Liquid Data.

JDBC Type	Liquid Data Data Type
Types.ARRAY	<i>not supported</i>
Types.BIGINT	xs:long
Types.BINARY	xs:string
Types.BIT	xs:boolean
Types.BLOB	<i>not supported</i>
Types.CHAR	xs:string
Types.CLOB	<i>not supported</i>
Types.DATE	xs:date
Types.DECIMAL	xs:decimal
Types.DOUBLE	xs:double
Types.FLOAT	xs:double
Types.INTEGER	xs:integer
Types.JAVA_OBJECT	<i>not supported</i>

JDBC Type	Liquid Data Data Type
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	<i>not supported</i>
Types.TIME	xs:time
Types.TIMESTAMP	xs:dateTime
Types.TINYINT	xs:byte
Types.VARBINARY	xs:string
Types.VARCHAR	xs:string

JDBC Names

The following table maps the JDBC name to Liquid Data data types.

JDBC Name	Liquid Data Data Type
ARRAY	<i>not supported</i>
BIGINT	xs:long
BINARY	xs:string
BIT	xs:boolean
BLOB	<i>not supported</i>
CHAR	xs:string

B Supported Data Types

JDBC Name	Liquid Data Data Type
CLOB	<i>not supported</i>
DATE	xs:date
DEC	xs:decimal
DECIMAL	xs:decimal
DOUBLE	xs:double
FLOAT	xs:double
INTEGER	xs:integer
JAVA_OBJECT	<i>not supported</i>
LONGVARBINARY	xs:string
LONGVARCHAR	xs:string
NUM	xs:decimal
NUMERIC	xs:decimal
REAL	xs:float
REF	xs:string
SMALLINT	xs:short
STRUCT	<i>not supported</i>
TIME	xs:time
TIMESTAMP	xs:dateTime
TINYINT	xs:byte
VARBINARY	xs:string
VARCHAR	xs:string

Database-Specific Names

This section includes tables showing the database-specific names and the corresponding Liquid Data data types. This section includes the following:

- [Oracle Names](#)
- [Microsoft SQL Server Names](#)
- [DB2 Names](#)
- [Sybase Names](#)

Oracle Names

The following table maps Oracle names to Liquid Data data types.

Oracle Name	Liquid Data Data Type
FLOAT	xs:float
BFILE	<i>not supported</i>
LONG	<i>not supported</i>
LONG RAW	<i>not supported</i>
NCHAR	xs:string
NCLOB	<i>not supported</i>
NUMBER	xs:decimal
NVARCHAR2	xs:string
RAW	xs:string

B Supported Data Types

Oracle Name	Liquid Data Data Type
ROWID	xs:string
UROWID	<i>not supported</i>
VARCHAR2	xs:string

Microsoft SQL Server Names

The following table maps Microsoft SQL Server names to Liquid Data data types.

SQL Name	Liquid Data Data Type
DATETIME	xs:dateTime
IMAGE	<i>not supported</i>
INT	xs:integer
MONEY	xs:float
NTEXT	xs:string// too big?
NVARCHAR	xs:string
SMALLDATETIME	xs:dateTime
SMALLMONEY	xs:float
SQL_VARIANT	xs:string
UNIQUEIDENTIFIER	xs:string

DB2 Names

The following table maps DB2 data types to Liquid Data data types.

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 Names

The following table maps Sybase data types to Liquid Data data types.

Sybase Name	Liquid Data Data Type
SYSNAME	xs:string
TEXT	xs:string

Informix Names

The following table maps Informix data types to Liquid Data data types.

Informix Name	Liquid Data Data Type
BLOB	<i>not supported</i>
BYTE	<i>not supported</i>
BOOLEAN	xs:boolean
CHAR (n)	xs:string
CHARACTER (n)	xs:string
CLOB	<i>not supported</i>
DATE	xs:date
DATETIME	xs:dateTime
DEC/DECIMAL	xs:decimal
DOUBLE PRECISION/FLOAT	xs:double
INT/INTEGER	xs:integer
INT8	xs:long
INTERVAL	<i>not supported</i>
LVARCHAR	xs:string
MONEY	xs:float
NCHAR	xs:string
NUMERIC	xs:decimal
REAL	xs:float
SMALLFLOAT	xs:float
SMALLINT	xs:short
TEXT	xs:string

C Type Casting Reference

This section provides details on how Data View Builder implements data type transformation for automatic type casting. The following topics are included:

- [Type Casting to a Numeric Target](#)
- [Type Casting to a Non-Numeric Target](#)
- [Type Casting Function Parameters](#)

When you request Automatic Type Casting, Liquid Data can reassign a data type if the data type of the source node does not match the data type of the mapped target node but the data types are compatible. Use the information in the following sections to predict the automatic type casting behavior when this occurs.

Note: For information on how to set the option for automatic type casting in the Data View Builder, see [“Using Automatic Type Casting” on page 3-17](#).

Type Casting to a Numeric Target

The following table shows whether Liquid Data transforms a source node data type to the numeric data type of the target node.

	Target: xs:byte	Target: xs:short	Target: xs:int	Target: xs:long	Target: xs:integer	Target: xs:decimal	Target: xs:float	Target: xs:double
xs:byte	N	Y	Y	Y	Y	Y	Y	Y
xs:short	Y	N	Y	Y	Y	Y	Y	Y
xs:int	Y	Y	N	Y	Y	Y	Y	Y
xs:long	Y	Y	Y	N	Y	Y	Y	Y
xs:integer	Y	Y	Y	Y	N	Y	Y	Y
xs:decimal	Y	Y	Y	Y	Y	N	Y	Y
xs:float	Y	Y	Y	Y	Y	Y	N	Y
xs:double	Y	Y	Y	Y	Y	Y	Y	N
xs:string	Y	Y	Y	Y	Y	Y	Y	Y
xs:boolean	Y	Y	Y	Y	Y	Y	Y	Y
xs:date	N	N	N	N	N	N	N	N
xs:time	N	N	N	N	N	N	N	N
xs:dateTime	N	N	N	N	N	N	N	N
xsect:anyValue xsect:anyType xsect:item	Y	Y	Y	Y	Y	Y	Y	Y

Type Casting to a Non-Numeric Target

The following table shows whether Liquid Data transforms a source node data type to the non-numeric data type of the target node.

	Target: xs:byte	Target: xs:boolean	Target: xs:date	Target: xs:time	Target: xs:dateTime	Target: xsect:anyValue xsect:anyType xsect:item
xs:byte	Y	Y	N	N	N	N
xs:short	Y	Y	N	N	N	N
xs:int	Y	Y	N	N	N	N
xs:long	Y	Y	N	N	N	N
xs:integer	Y	Y	N	N	N	N
xs:decimal	Y	Y	N	N	N	N
xs:float	Y	Y	N	N	N	N
xs:double	Y	Y	N	N	N	N
xs:string	N	Y	Y	Y	Y	N
xs:boolean	Y	N	N	N	N	N
xs:date	Y	N	N	N	N	N
xs:time	Y	N	N	N	N	N
xs:dateTime	Y	N	Y (see note)	Y (see note)	N	N
xsect:anyValue xsect:anyType xsect:item	Y	Y	Y	Y	Y	N

Note: The type cast from xs:dateTime to xs:date and xs:time uses xfext:date-from-dateTime() and xfext:time-from-dateTime.

Type Casting Function Parameters

In some cases, Liquid Data can transform the data type for a function parameter when a mismatch occurs.

	Target: xs:byte	Target: xs:short	Target: xs:int	Target: xs:long	Target: xs:integer	Target: xs:decimal	Target: xs:float	Target: xs:double
xs:byte	N	N	N	N	N	N	N	N
xs:short	Y	N	N	N	N	N	N	N
xs:int	Y	Y	N	N	N	N	N	N
xs:long	Y	Y	Y	N	N	N	N	N
xs:integer	Y	Y	Y	Y	N	N	N	N
xs:decimal	Y	Y	Y	Y	Y	N	N	N
xs:float	Y	Y	Y	Y	Y	Y	N	N
xs:double	Y	Y	Y	Y	Y	Y	Y	N

Index

Symbols

../design.html#1049351 3-41

A

ad hoc query 1-7
advanced view
 on Conditions tab 2-27
 understanding scope in 3-30
aggregate
 definition 1-16
 in example query 9-8
application view
 as supported data source 1-9
automatic type casting 3-17

B

BEA corporate Web site iii-iv

C

Complex Parameter Types, Using 7-1
components
 accessing components of a query from
 Toolbox tab 2-11
constants
 accessing from Toolbox tab 2-10
count function
 in example query 9-35
CPT (Complex Paramater Type) 7-1
custom functions

 accessing from Toolbox tab 2-10
 use cases for 7-3
customer support contact information iii-v

D

data sources
 order optimization 4-3
 supported in Liquid Data 1-7
data view
 as supported data source 1-10
data views
 simple and parameterized 6-4
 using as data sources 6-4, 7-8
date-time
 example query 9-18
DB2
 names for Liquid Data data types B-7
Design tab 2-4
documentation, where to find it iii-iv

F

functions
 accessing from Toolbox tab 2-10
 count used in example query 9-35
 date and time in example query 9-18
 introduction to use of in Data View
 Builder 1-16
 W3C XQuery links A-1

H

- hints
 - for parameter passing 4-6
 - merge 4-8
 - optimizing queries with 4-5
 - ppleft 4-6
 - ppright 4-6

I

- IBM DB2
 - stored procedure support 8-30
- Informix
 - names for Liquid Data data types B-8
 - stored procedure support 8-31

J

- JDBC
 - names for Liquid Data data types B-3
 - supported data types for Liquid Data B-2
- join
 - adding hints for optimizing query
 - performance 4-5
 - definition 1-15
 - in example query 9-2

L

- Liquid Data documentation Home page iii-v

M

- Microsoft SQL Server
 - stored procedure support 8-28
- minus
 - in example query 9-35
- MSQL
 - server names for Liquid Data data types B-6

N

- namespaces, XML 1-17
- naming conventions
 - for queries 5-7
 - for stored queries to be generated as Web services 5-7

O

- optimization
 - data source order in query 4-3
 - hints for joins 4-5
- Optimize tab 2-30
- Oracle
 - names for Liquid Data data types B-5
 - stored procedure support 8-27

P

- parameter
 - types 2-17
- parameters
 - introduction to use of in functions 1-17
- ppleft 4-6
- ppright 4-6
- print, how to iii-v
- printing product documentation iii-v

Q

- query
 - ad hoc 1-7
 - optimizing source order in 4-3
 - plans 1-6
 - result 5-5
 - running 5-4
 - saving 5-6
 - saving as a stored query 5-6
 - stopping while running 5-4
 - stored 1-6
 - testing 5-4

query parameters
 defining 2-15
 defining in Toolbox tab 2-10
 introduction to use of in functions 1-17
 submitting at query runtime 2-35
 types 2-17
query plan
 definition 1-6

R

related information iii-v

S

schemas
 off-line (unavailable) 2-39
 source-introduction 1-10
 target-introduction 1-10
scope
 defining in Advanced view on
 Conditions tab 2-27
source
 order optimization 4-3
source schema
 introduction 1-10
stored procedure
 support by database 8-26
Stored Procedure Description file
 DB2 multiple result sets 8-23
 elements and attributes 8-6
 Oracle cursor as a return_value 8-25
 Oracle cursor output parameter 8-22
 overview 8-4
 rules for specifying 8-10
 sample files 8-19
 schema definition file 8-5
stored procedures
 as supported data source 1-10
 defining to Liquid Data 8-2
 example 8-33

 overview 8-1
 Stored Procedure Description file 8-4
 using in queries 8-32
stored query
 definition 1-6
 saving as 5-6
support
 technical iii-vi
Sybase
 names for Liquid Data data types B-7
 stored procedure support 8-29

T

target schema
 introduction 1-10
 namespaces 1-20
 understanding 1-11
Test tab 2-32
type casting
 automatic in Data View Builder 3-17

U

union
 definition 1-16
 in example query 9-28

W

W3C
 relationship to XQuery and Liquid Data
 1-2
Web services
 definition 1-9
 naming conventions for queries 5-7
World Wide Web Consortium (W3C) iii-iii

X

XML iii-iii, iii-iv
 role in XQuery 1-2

- XML files as supported data source 1-9
- XML file
 - definition 1-9
- XML namespaces 1-17
- XQuery iii-iii
 - as used in Liquid Data 1-3
 - definition 1-2
 - links to more information 1-4