BEA Liquid Data for WebLogic™

Building Queries and Data Views

Release: 1.0
Document Date: October 2002
Revised:

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Building Queries and Data Views

Part Number	Date	Software Version
N/A	October 2002	1.0

Contents

2. Starting the Builder and Touring the GUI

3. Designing Queries

4. Optimizing Queries

5. Testing Queries

6. Query Cookbook

7. Using Data Views as Data Sources

A. Functions Reference

B. Supported Data Types

C. Type Casting Reference

Index

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 6, "Query Cookbook," provides detailed examples about how to construct queries using some advanced techniques and functions.
- Chapter 7, "Using Data Views as Data Sources," 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.
italics	Indicates emphasis or book titles.

Convention	Item

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

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Example*s: LPT1 SIGNON OR
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.

`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...`
	`[-l` *`file-list`*`]...`

`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.

Convention	Item

. . .	Indicates one of the following in a command line:
	■ That an argument can be repeated several times in a command line
	■ That the statement omits additional optional arguments
	■ That you can enter additional parameters, values, or other information
	The ellipsis itself should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f file-list]...`
	`[-l file-list]...`

. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

CHAPTER

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 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.

How Do Liquid Data and Data View Builder Use XQuery?

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/.

Which Versions of XML Schema Does Liquid Data Support?

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 1-9.

How Can I Learn 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*.

What is the Data View Builder?

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."

Why Do I Need it?

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 Does it Work?

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.

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.

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.

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*

Liquid Data requires data sources to be configured on the server and they must have an associated schema. A default relational database schema can be obtained automatically. XML files, Views, or Web Services must be configured on the server before you try to use them in queries.

Note: For information on which versions of XML schema are supported, see "Which Versions of XML Schema Does Liquid Data Support?" on page 1-3.

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. The Data View Builder provides graphical representation of target schemas in a tree structure format. Nodes represent elements, attributes, and sub-elements and attributes that you can expand or collapse. Only one target schema per query is allowed.

A target schema is usually just that part of the hierarchy that you want to appear in the result. For example, you could choose the same schema for both the source and target data structure. The query result will show only those data elements that are actually mapped to nodes in the target schema.

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

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

Aggregate query results summarize information. You can use aggregate operations to extract summary information in different ways, such as creating totals, counts, and averages.

Functions

Liquid Data provides a functions library with built-in functions can by 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.

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 examples of building different types of queries using advanced functions and tools, see Chapter 6, “Query Cookbook.”
- 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

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.

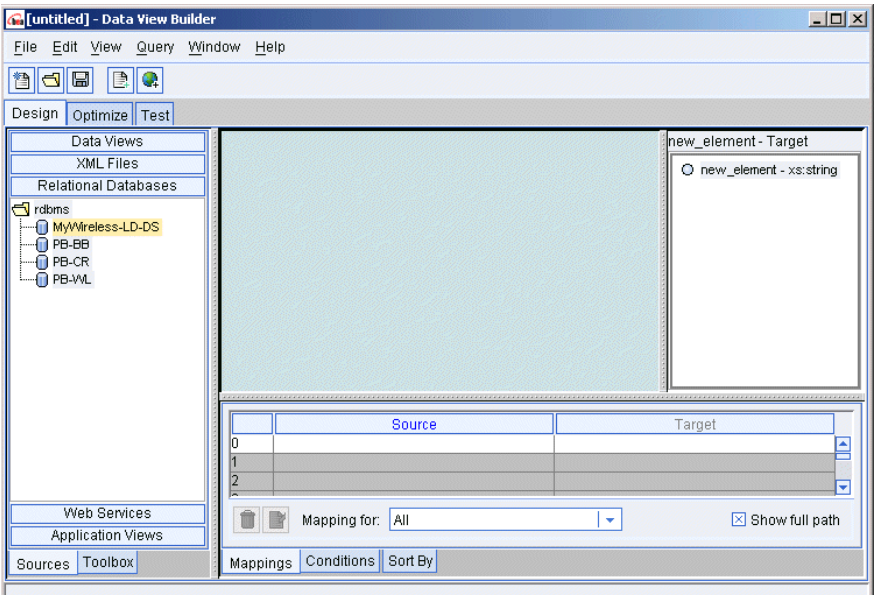
   b. Enter the URL for the Liquid Data server. For example, to connect to a server running on your own machine as a local host you would enter the following:

      ```
      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

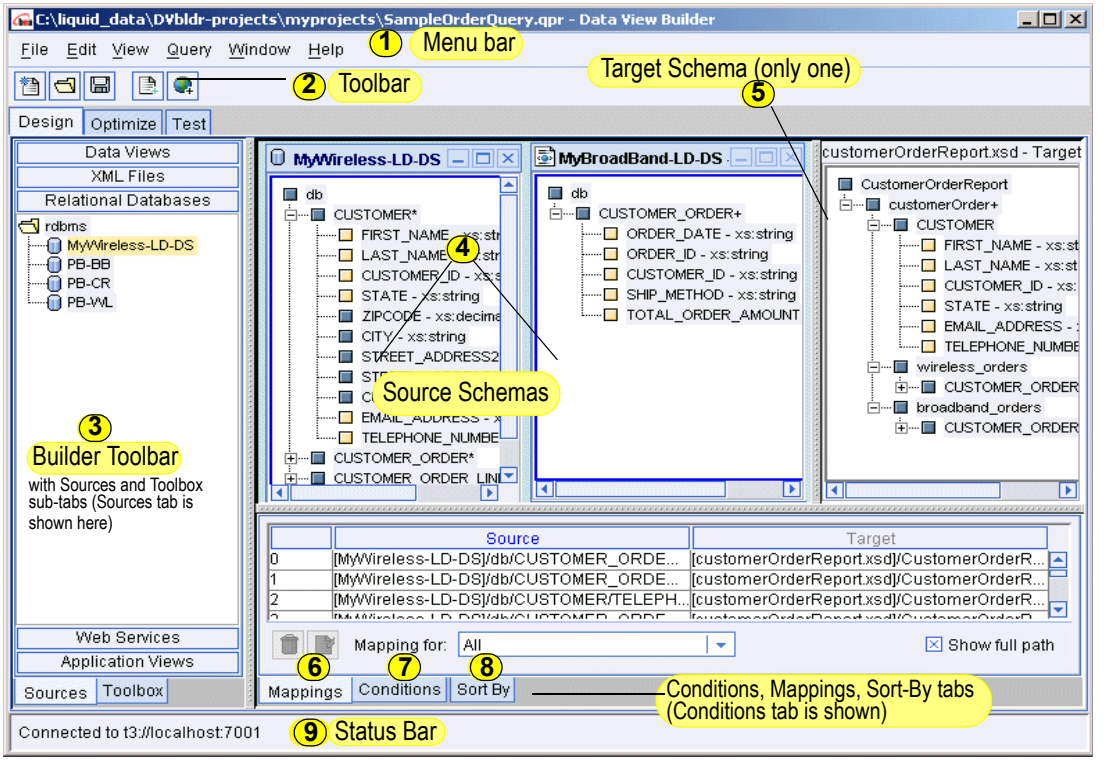
The Design tab is where you construct the query by working with source and target schemas to specify source conditions and source-to-target mappings.

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

Overview Picture of Design Tab Components

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 Menu Bar provides File, Schema, View, and Window menus as detailed in the following table.

Table 2-1 Menu Bar for the Design Tab

Menu	Description of Menu Options
File Menu	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. ■ **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. ■ **Save Projec**t—Saves the current project. Data View Builder projects are saved with a `.qpr` filename extension. ■ **Save Project As**—Saves the current project under another file name. Data View Builder projects are saved with a `.qpr` 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 folder location and filename you choose. You can save the current target schema locally or in the Liquid Data server repository, which makes it available to other Liquid Data users. ■ **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

Menu	Description of Menu Options
Edit Menu	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. ■ **Cut** ■ **Copy** ■ **Paste** ■ **Delete** ■ **Select All**
View Menu	As an alternative to using the tabs the View menu lets you **navigate** to the following UI views: ■ **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. 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: ■ **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. 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.

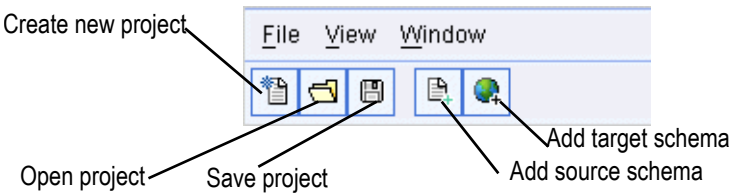
Table 2-1 Menu Bar for the Design Tab

Menu	Description of Menu Options
Query Menu	■ **Automatic Type Casting—**Toggle to turn automatic type casting on/off. (An "X" next to this option indicates that automatic type casting is *on*.) For more information about using automatic type casting see "Using Automatic Type Casting" on page 3-17. ■ **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 3-29. 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.
Window Menu	The Window menu provides various options for window management: 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.
Help Menu	Provides online documentation for the Data View Builder. **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.

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

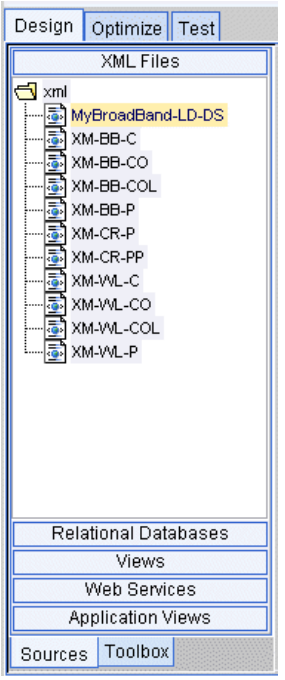
The Sources tab on the Builder Toolbar contains the data sources configured on the Liquid Data Server to which you are connected. Note that a data source type only shows up as a button on the Builder Toolbar if it has been configured in the Server to which you are connecting.

- 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."

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 desktop.

Figure 2-4 Builder Toolbar: Sources Tab



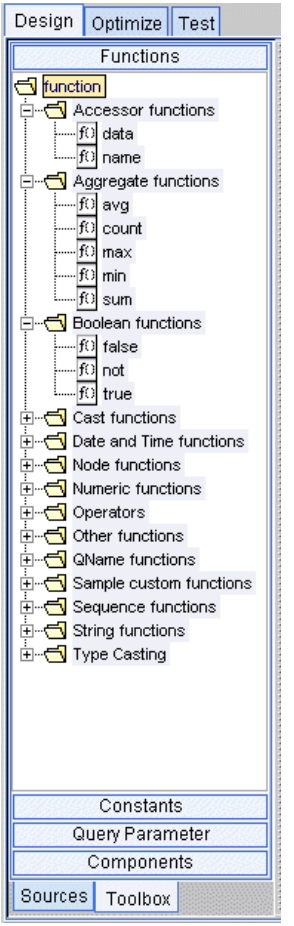
Toolbox Tab

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

- 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



Functions

Functions are built-in code modules that return a value when they run. The 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.)

In Data View Builder, the Functions are displayed in the Builder Toolbar on the Toolbox tab Functions panel by category names like Aggregate Functions, Boolean Functions, Operator 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 desktop where it appears in a tree format showing the number and type of parameters required.

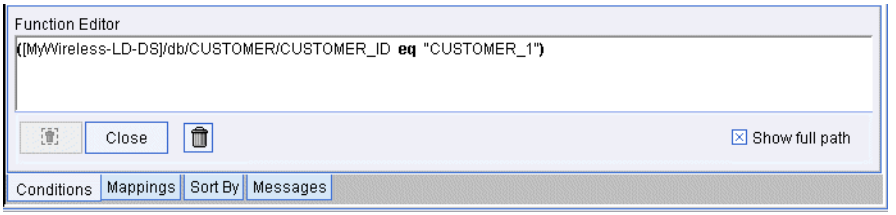
A copy of any mapped function saves automatically with the project when you close it. The saved function (with associated parameters) appears in the Components panel when you reopen the project. If you do not map the new function and you terminate the session, Data View Builder discards it and it does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

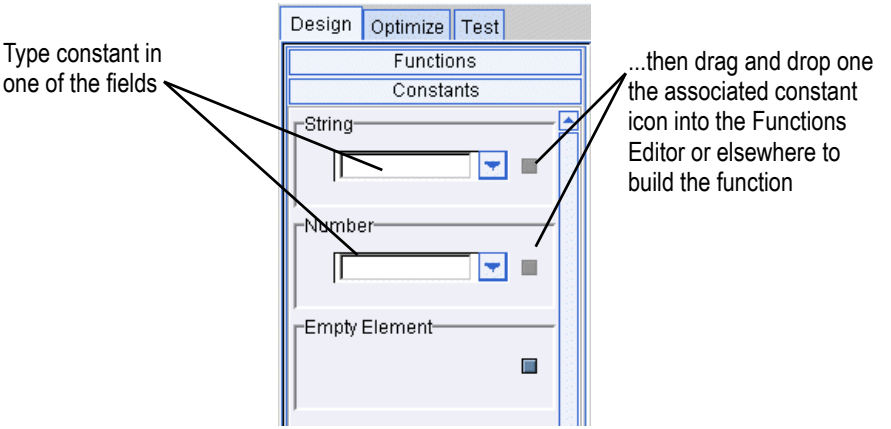
Custom Functions

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 desktop. For example: choose the `startswith` function. You get the following placeholder in the Functions Editor:

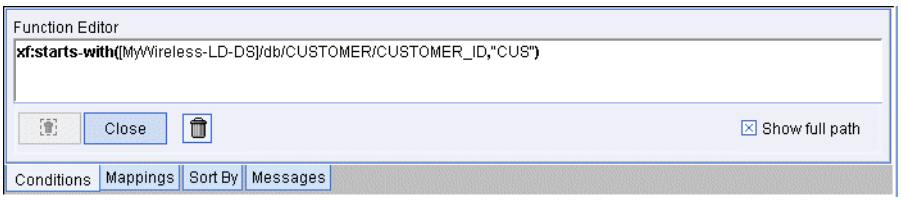
   `xf:starts-with`(*`str1`*,*`str2`*)

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`*).

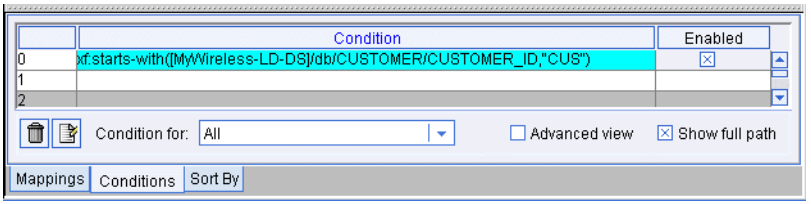
   The condition appears in the Functions Editor as shown in the following figure.

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

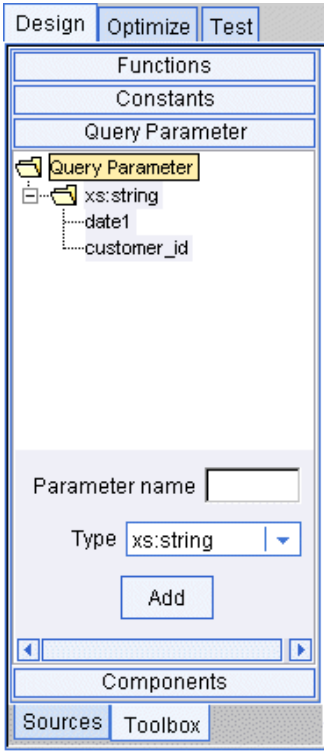
Query parameters can be strings, integers, floating point numbers, boolean expressions, or date and time types. They are variables that you define with no static value. On the Test tab, you can supply a different value each time you run the query (see "4. Query Parameters: Submitted at Query Runtime"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

The Query Parameter bar has a text box where you can enter a new parameter value to be stored.

- Type a value that is one of the four 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. The following table shows examples of supported data type values.

Table 2-2 Query Parameter Types

Parameter Type	Examples
Boolean (`xs:boolean`)	Boolean expressions test true or false. You can specify that the Boolean Query Parameter has an implicit definition of `True` or `False`, then use it as query resource.

Table 2-2 Query Parameter Types

Parameter Type	Examples
Byte (`xs:byte`)	A positive or negative whole number. The maximum value is 127 and the minimum value is -128. For example: ■ -1 ■ 0 ■ 126 ■ +100
Date (`xs:date`)	Input must be in this format: `MMM dd, YYYY` For example: JUN 12, 2002
Date and Time (`xs:dateTime`)	Input must be in this format: `MMM dd, YYYY HH:MM:SS AM/PM` For example: MAY 12, 2002 12:12:11 AM
Decimal (`xs:decimal`)	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: ■ -1.23 ■ 12678967.543233 ■ +100000.00 ■ 210.
Double (`xs:double`)	A real number (negative or positive) that can contain fractional part. For example: `3.159` Liquid Data does not support floating point formats expressed in fractions (½) or IEEE floating point notation (3E-5).
Floating Point (`xs:float`)	A real number (negative or positive) that can contain a fractional part. For example: ■ 100.0 ■ -100.5 Liquid Data does not support floating point formats expressed in fractions (½) or IEEE floating point notation (3E-5).

Table 2-2 Query Parameter Types

Parameter Type	Examples
Int (`xs:int`)	A positive or negative whole number. The maximum value is 2147483647 and minimum value is -2147483648. For example: ■ -1 ■ 0 ■ 126789675 ■ +100000
Integer (`xs:integer`)	A positive or negative whole number. The maximum value is 2147483647 and minimum value is -2147483648. For example: ■ 1 ■ -100 ■ +100
Long (`xs:long`)	A positive or negative whole number. The maximum value is 9223372036854775807 and minimum value is -9223372036854775808. For example: ■ -1 ■ 0 ■ 12678967543233 ■ +100000
Short (`xs:short`)	A positive or negative whole number. The maximum value is 32767 and minimum is -32768. For example: ■ -1 ■ 0 ■ 126789 ■ +10000
String (`xs:string`)	An alphanumeric expression such as: ■ Smith ■ Jones ■ 12345 State St. **Note:** An unspecified value for a query parameter of type String is considered an empty string.

Table 2-2 Query Parameter Types

Parameter Type	Examples
Time (`xs:time`)	Input must be in this format: `HH:MM:SS AM/PM` For example: 01:02:15 AM

You can also do one of the following:

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

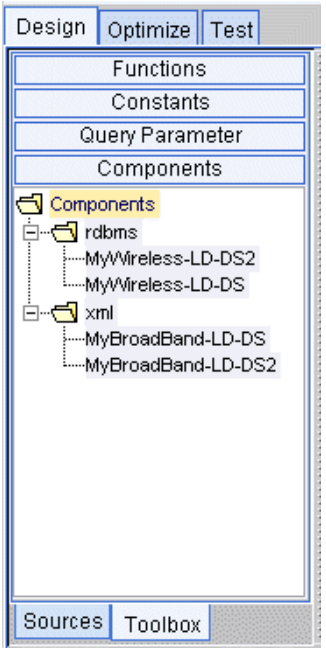
You can drag and drop a node from an input resource file to an empty row in the work area and then drag and drop a query parameter from the Query Parameter resource tree to specify a condition for the query.

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.

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.

Figure 2-12 Builder Toolbar: Toolbox Tab: Components



Any component that appears in this panel can be minimized on the 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 desktop as needed.

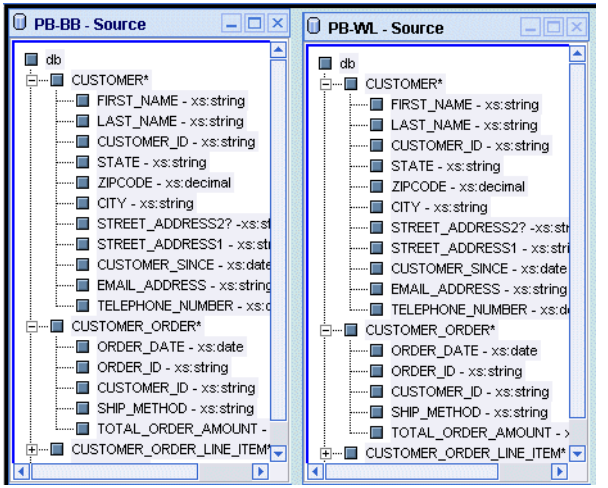
Note: For a detailed description of the special characters used to identify characteristics of schema nodes, see "Special Characters: Occurrence Indicators" on page 2-39.

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 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).

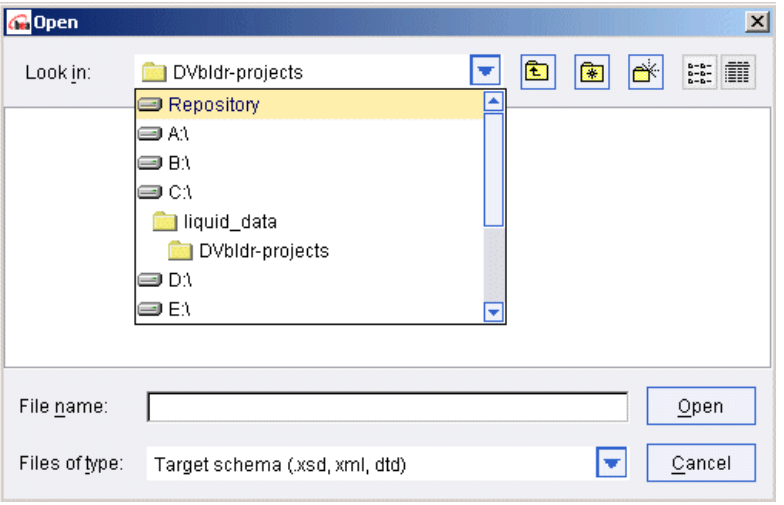
Only one target schema per project is allowed. If you have a target schema open and decide to choose another, the current target schema is closed and the new one replaces it.

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.

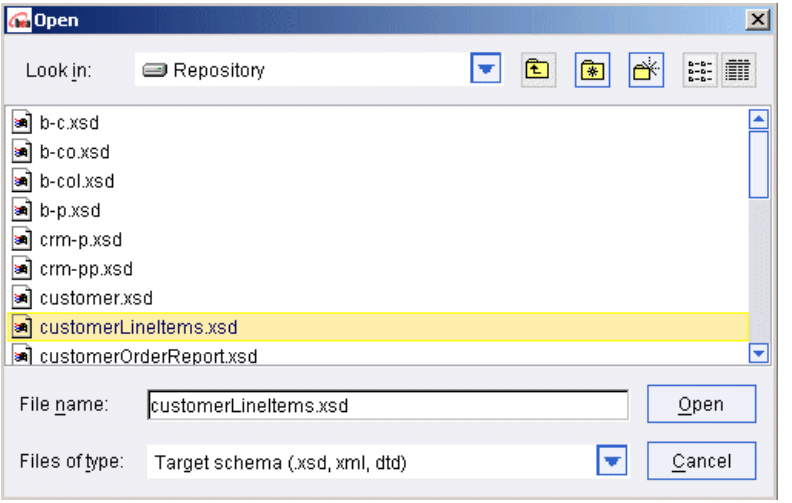
To open and set a target schema for a project:

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

   This brings up a file browser.



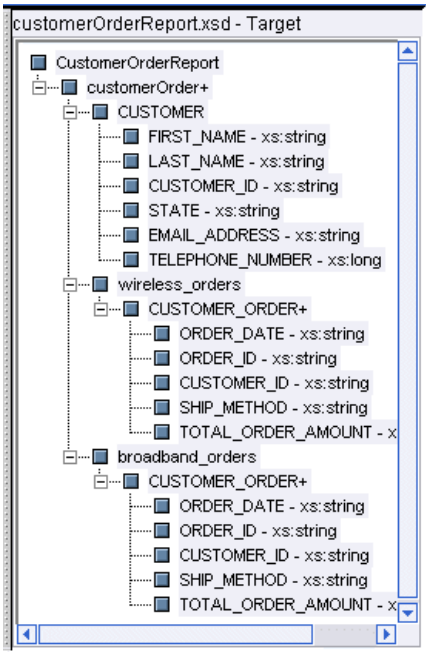
2. Navigate to the schema you want to use, select the file and click Open in the file browser.



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

(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



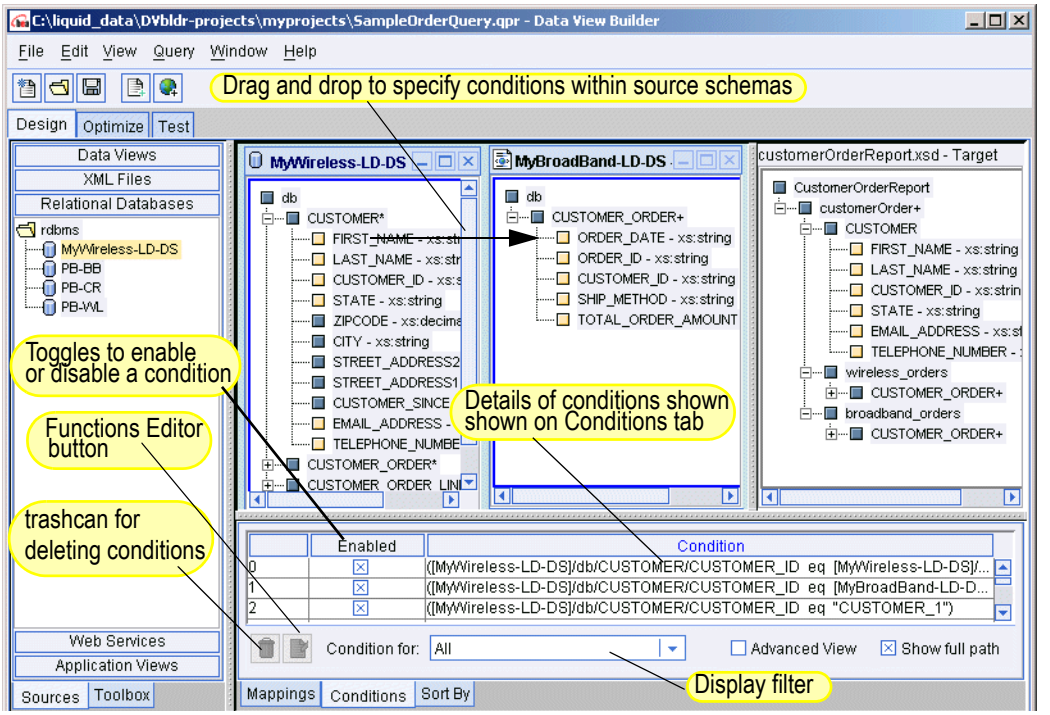
6. Conditions Tab

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 2-26)

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.

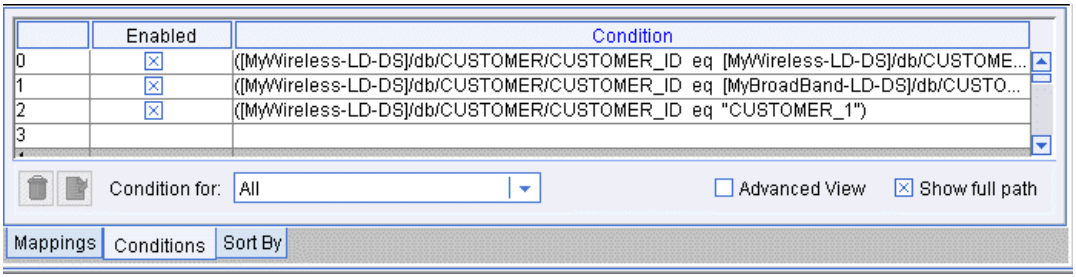
Figure 2-15 Conditions Tab on the Design tab



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.

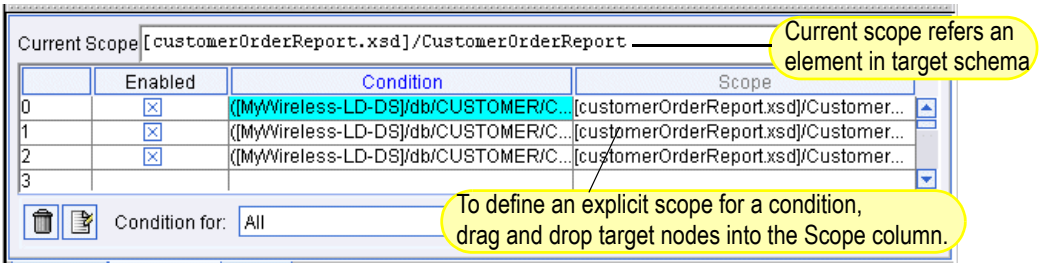
This tab includes the following features to facilitate working with conditions:

- **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 "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*). Another 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 3-29 in Chapter 3, "Designing Queries."

Returning to Basic View (Automated Scope)

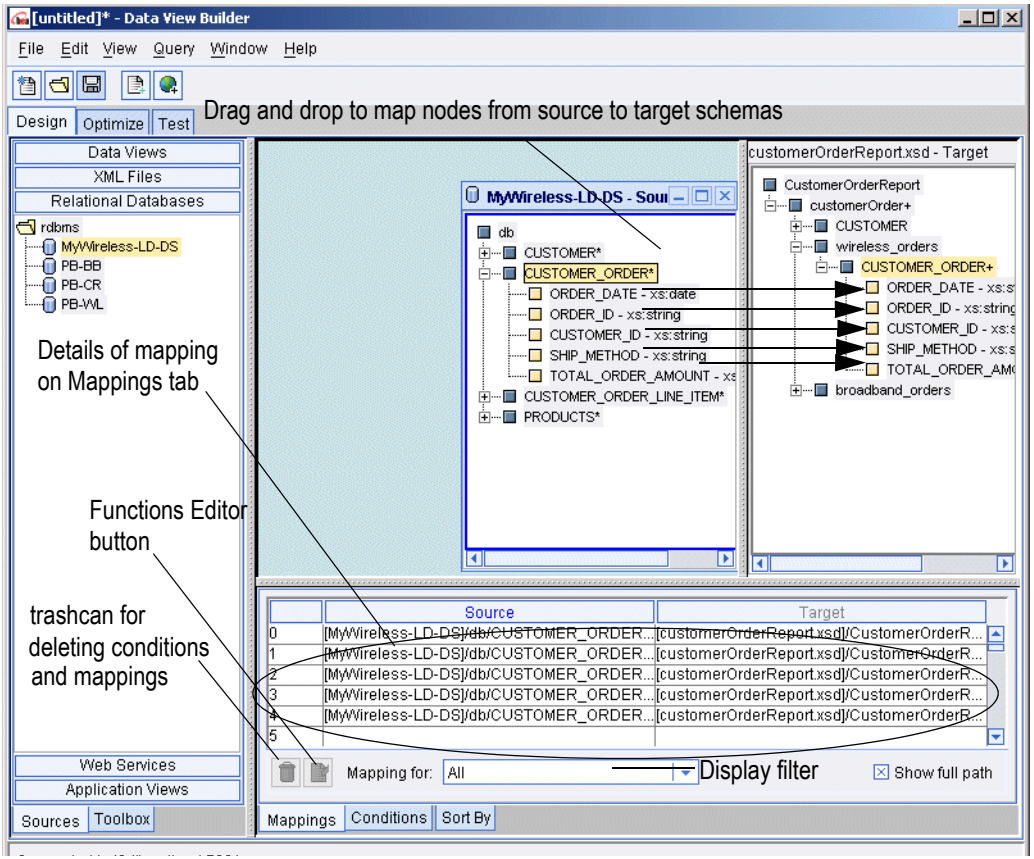
When you toggle Advanced View *off* (no X showing next to Advanced View), Data View Builder returns to automatic scoping mode and discards the changes you made in manual mode. The Current Scope text box and the Targets column disappear.

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.

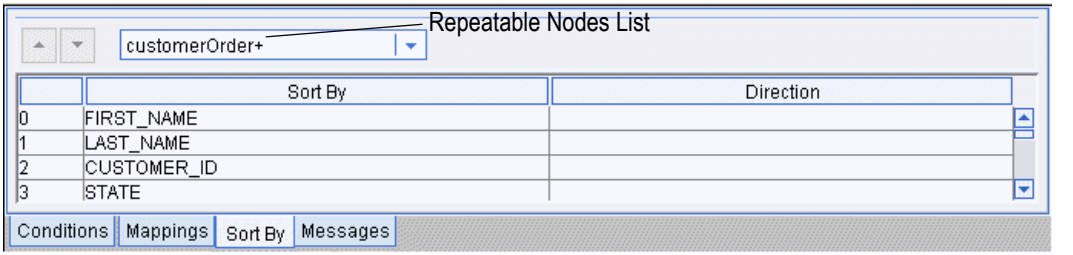
8. Sort By Tab

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

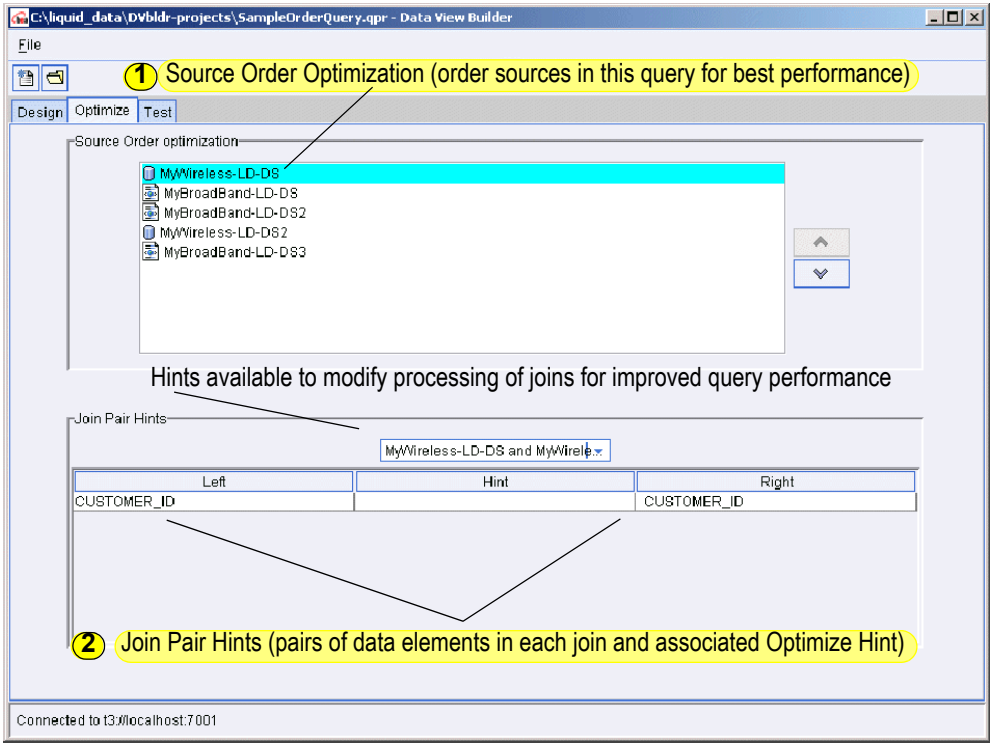
The Optimize tab is where you can optionally add more information such as “hints” to data sources to improve query performance.

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

Overview Picture of Optimize Tab Components

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.

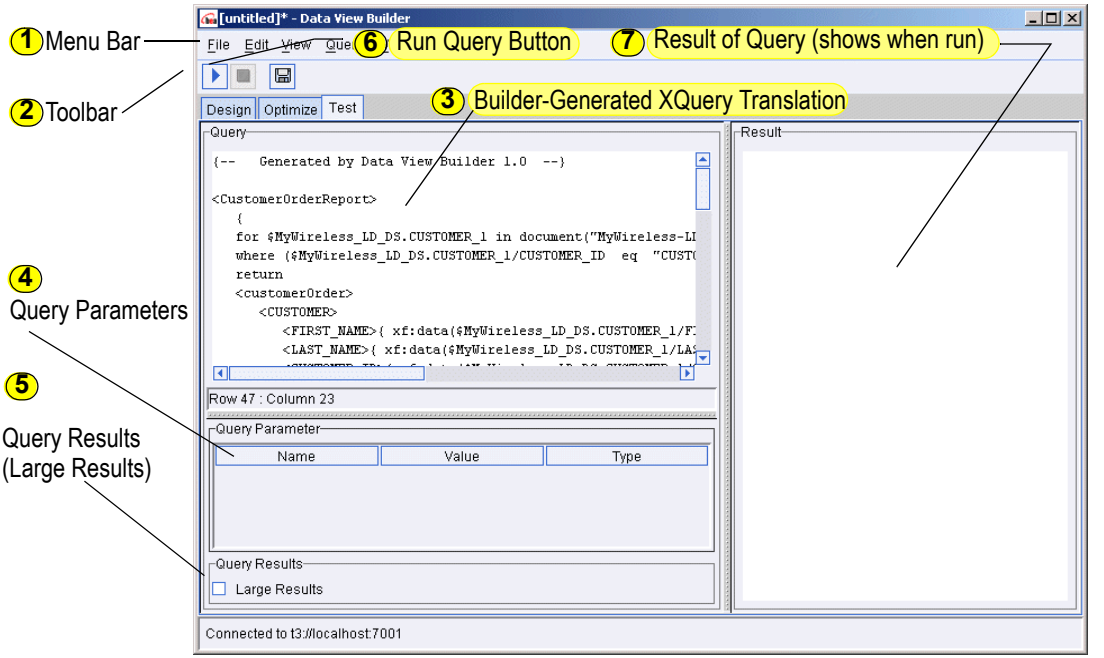
The following sections describe the features available on the Test tab.

- 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

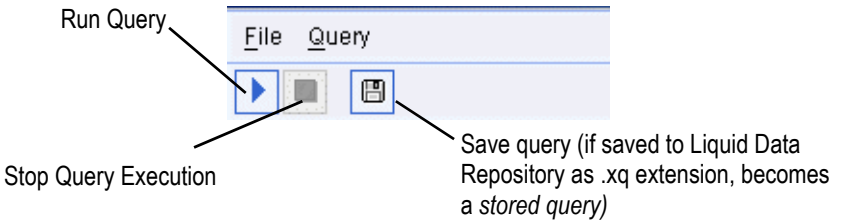
Table 2-3 Menu Bar for the Test Tab

Menu	Description of Menu Options
File Menu	Provides most of the same options as shown on the Design tab menu bar with one additional menu option as follows: ■ **Save Query**—Saves the current query to a file you specify. The file must be saved with a `.xq` extension. (If you do not add a `.xq` extension, Data View Builder will append it automatically.) If the query is saved into the `stored_queries` folder in the Liquid Data server Repository, it is considered a *stored query* in Liquid Data. For more details on saving a query, see "Saving a Query" on page 5-5 in Chapter 5, "Testing Queries." 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
Query Menu	Provides the following options related to running a query: ■ **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.) 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.

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.

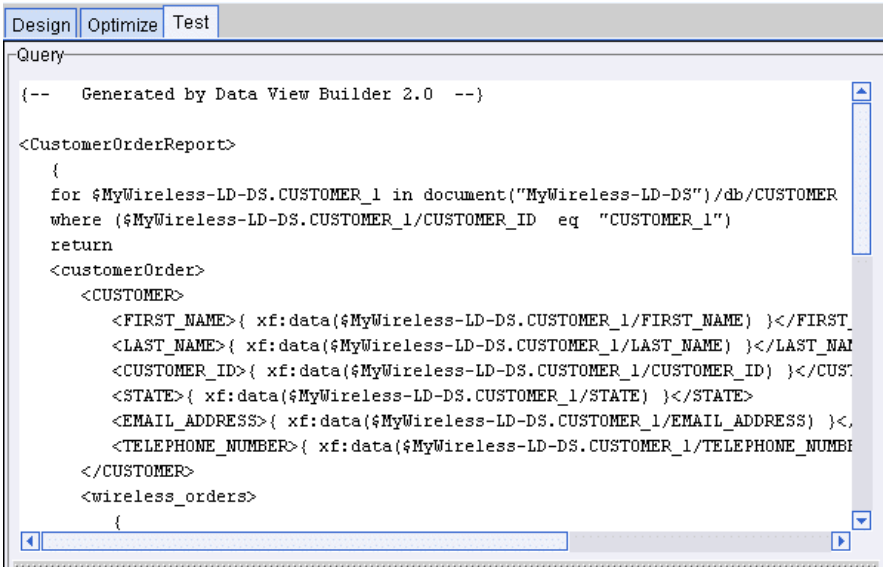
Figure 2-23 Toolbar on the Test Tab



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.

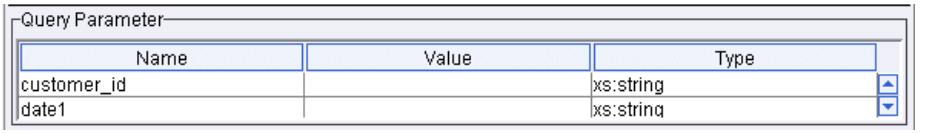
Figure 2-24 Builder-Generated XQuery Shown in Query Window



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

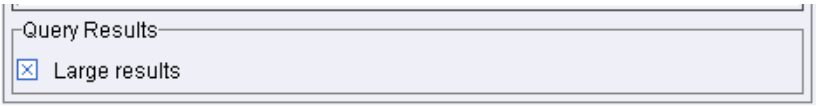


5. Query Results - Large Results

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.

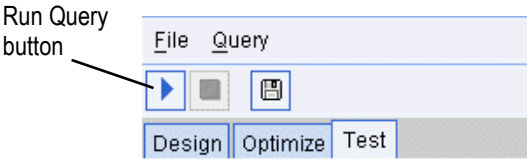
Figure 2-26 Specifying Large Results



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

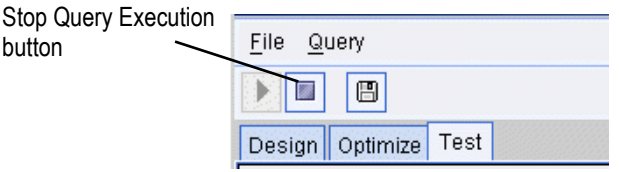


The query is run against your data sources and the result is displayed in the Results panel in XML format.

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.)

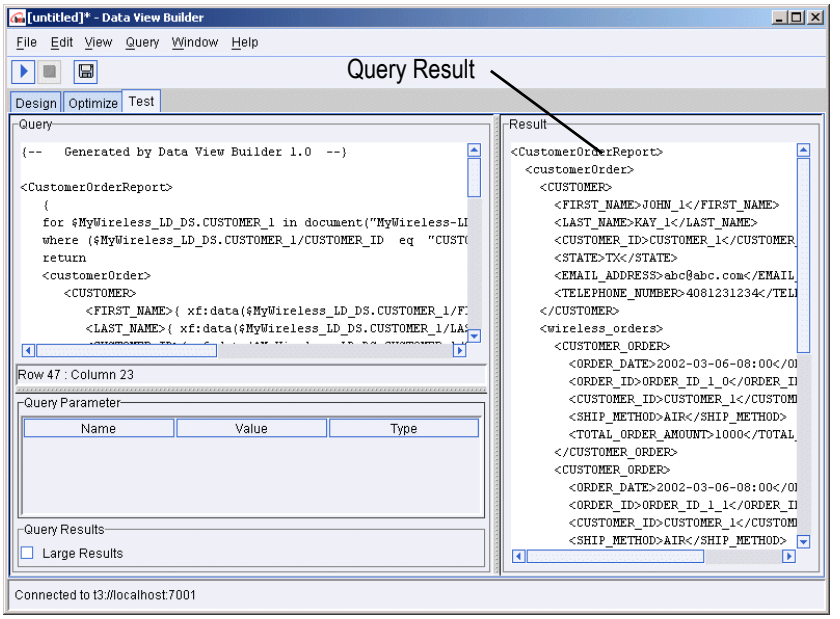
Figure 2-28 Click the "Stop Query Execution Button" to Stop a Running Query



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 5-5 in Chapter 5, "Testing Queries."

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 3-18 (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 6-1 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.

CHAPTER

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

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 3-38.
- 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">
```

```
  <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: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	You can copy both a source and a target node if: ■ The node or its parent is not a clone. ■ The node or its parent has not been cloned.
Paste	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. ■ If a pasted node is a duplicate, Data View Builder renames the node as _copy1, _copy2, 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)	Appends a new node or attribute as a child to the selected node. The name of the new element or node is *new_node* or *new_attribute.* The add function works only on an element node. You cannot add a child node to an attribute node. **Note:** Only string type data is supported.
Delete	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.
Rename	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.

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.

Or

- 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

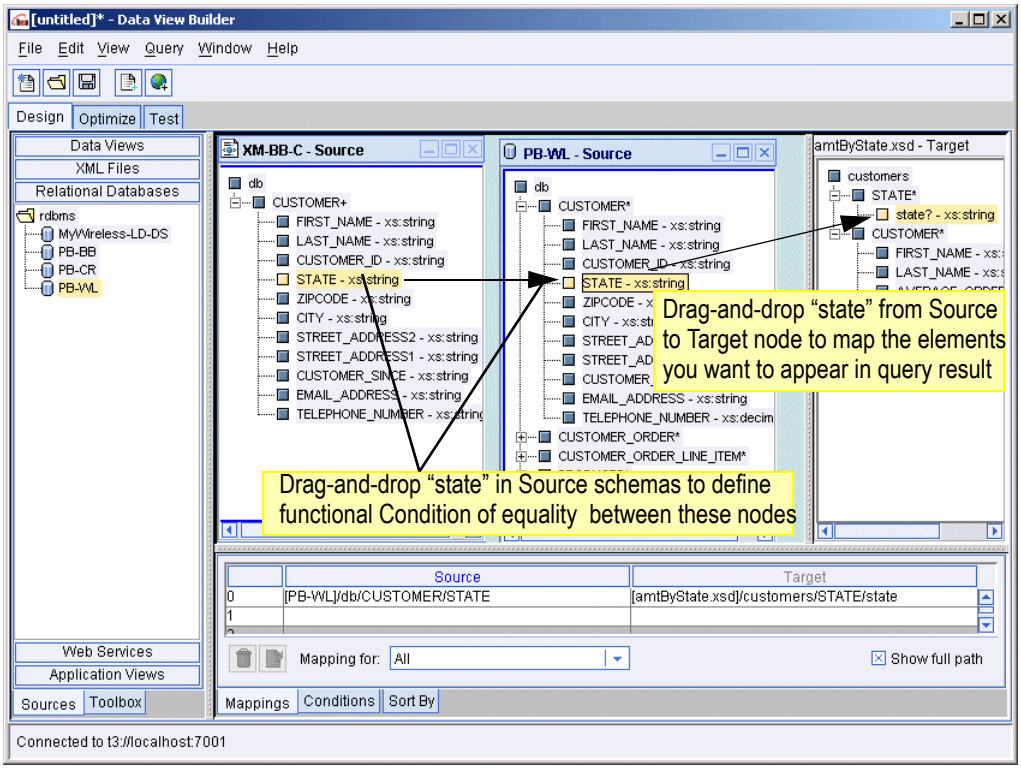
Figure 3-1 shows a simple join of the source element `STATE` in the Broadband source schema (`XM-BB-C`) with a source element `STATE` in the Wireless source schema (`PB-WL`). This action joins the common elements in each schema and disregards those that do not occur in both schemas.

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.

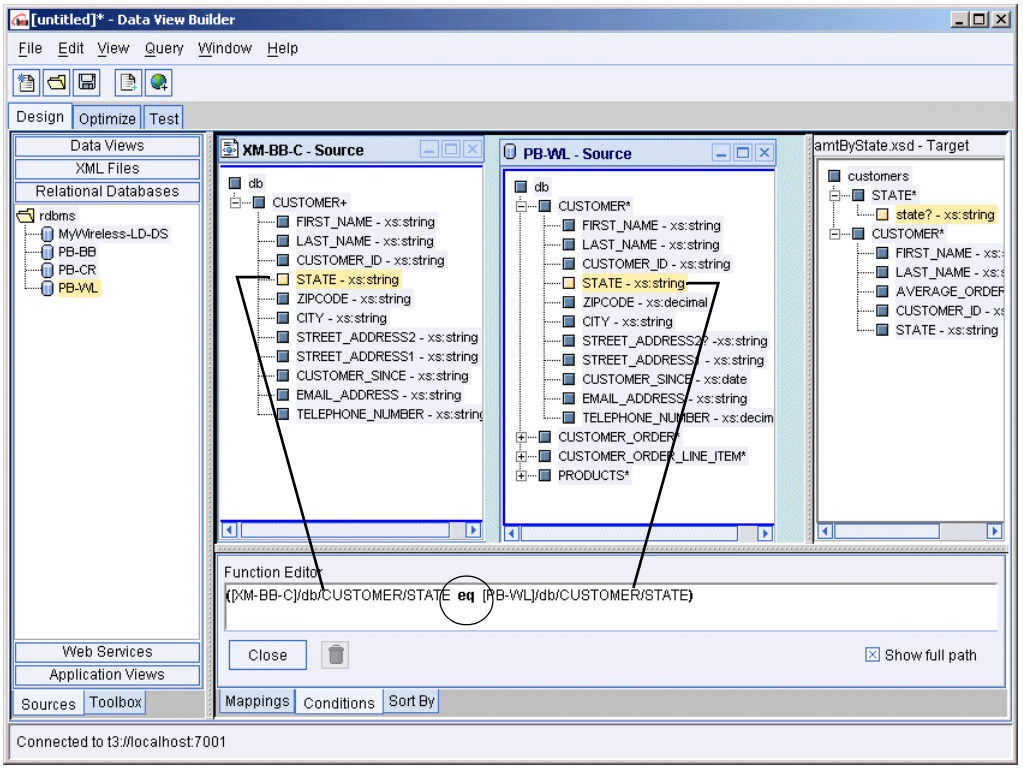
To edit an existing functional relationship:

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 (`eq`) 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 "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.

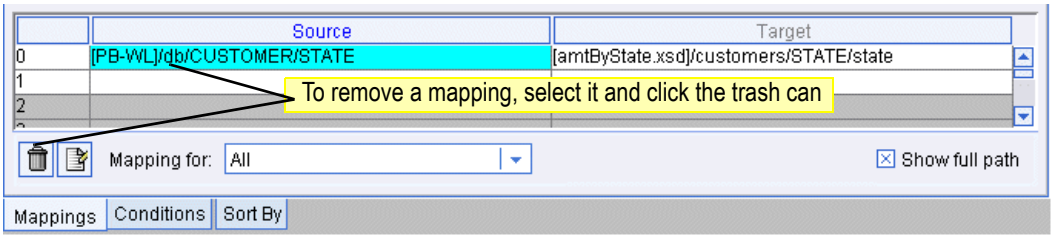
Table 3-1 Supported Mapping Relationships

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 6-8 in Chapter 6, "Query Cookbook" (specifically, the step "Ex 2: Step 8. Add the "count" Function" on page 6-14 within the Aggregates example).
Source node to a target node	By mapping a source to a target, you are *projecting*, 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 Output by State" on page 3-24.
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 6-8 in Chapter 6, "Query Cookbook" (specifically, the step "Ex 2: Step 8. Add the "count" Function" on page 6-14 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 chose 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

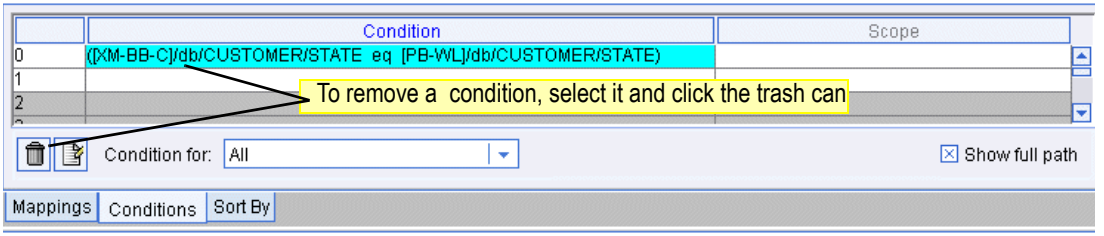
Instead of choosing an existing element/attribute as a parameter value, you can use one of these methods to specify that a constant value should be used instead of a data element from a source schema.

- 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 your 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 2-16.)
- 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.)

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 3-4.)

Figure 3-4 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 *xsext:anytype*.

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

- `xsext: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 Output 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."

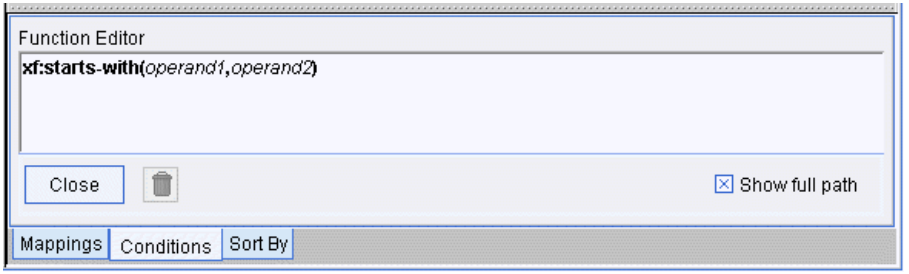
Build the Query

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.

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.

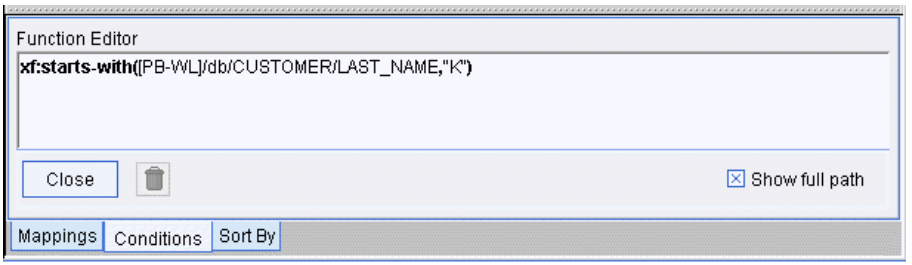


7. Drag and drop `CUSTOMER` "`LAST_NAME`" element from the Source schema onto the first parameter (*`operand1`*).

   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.

8. 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.")

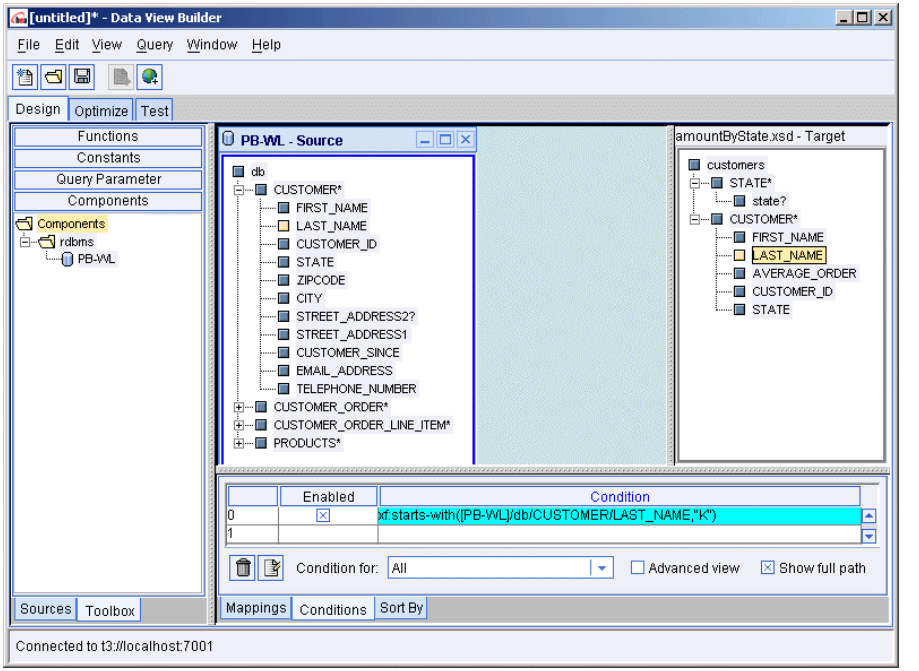
   The condition statement should look similar to that shown in following figure.



9. 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 3-5 shows the Design view of the query with conditions and source-to-target mappings completed.

Figure 3-5 Design View of Query Example: Return Customers By Name



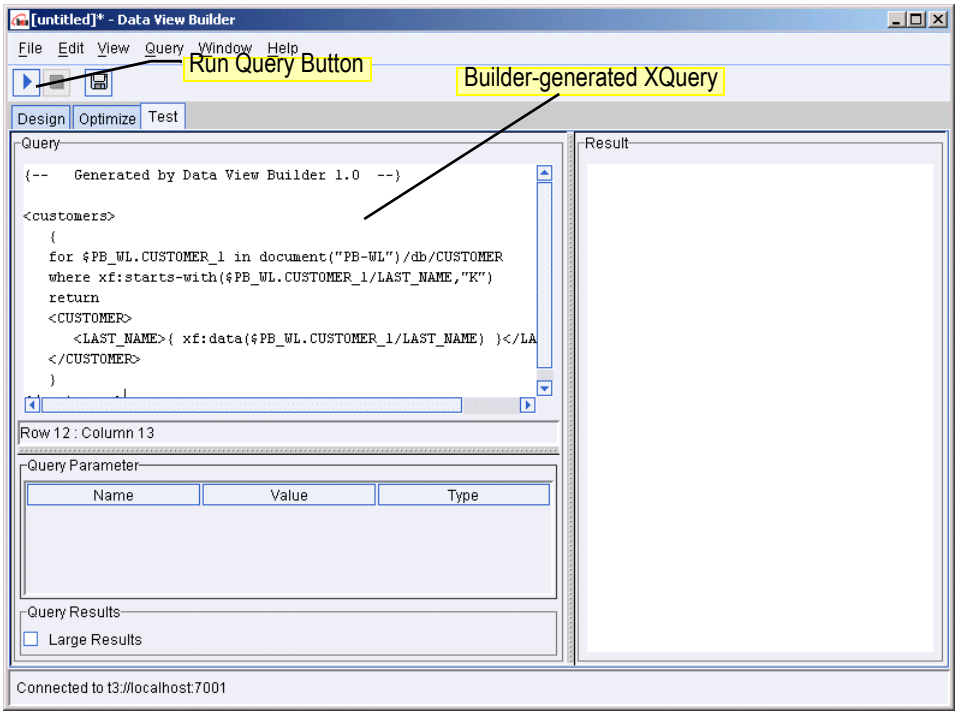
View the XQuery and Run the Query to Test it

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 3-6. The full XQuery is also provided in Listing 3-2.

Figure 3-6 XQuery for Example: Return Customers By Name



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

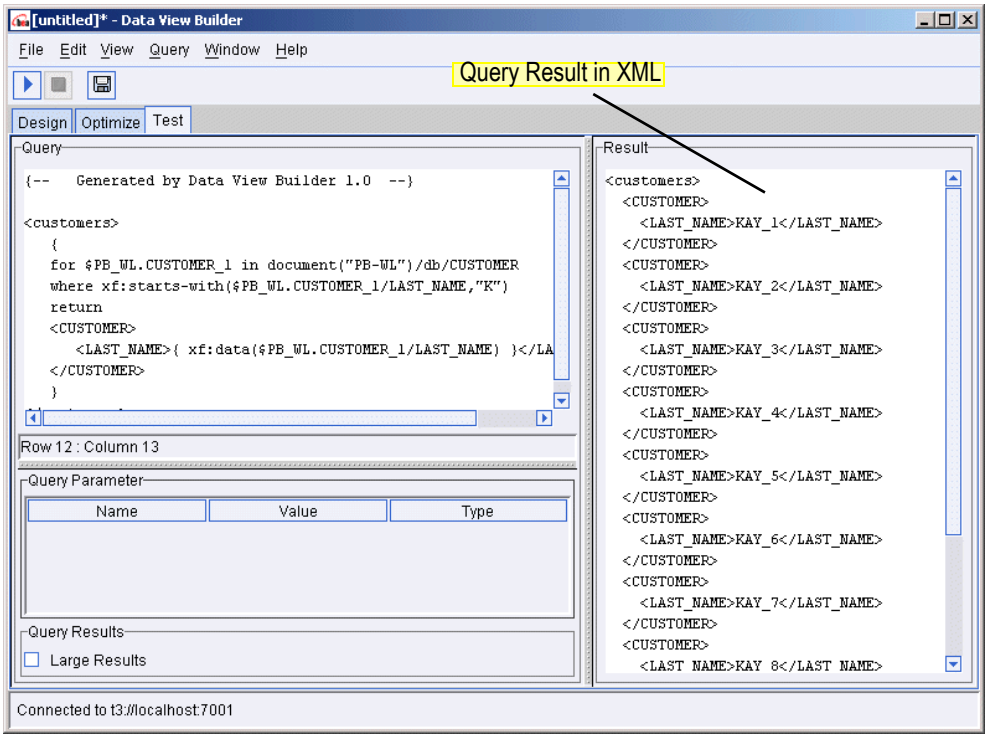
```
{--     Generated by Data View Builder 1.0 --}

<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 3-7. The full XML query result is provided in Listing 3-3.

Figure 3-7 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 Output 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 dropWireless (`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.

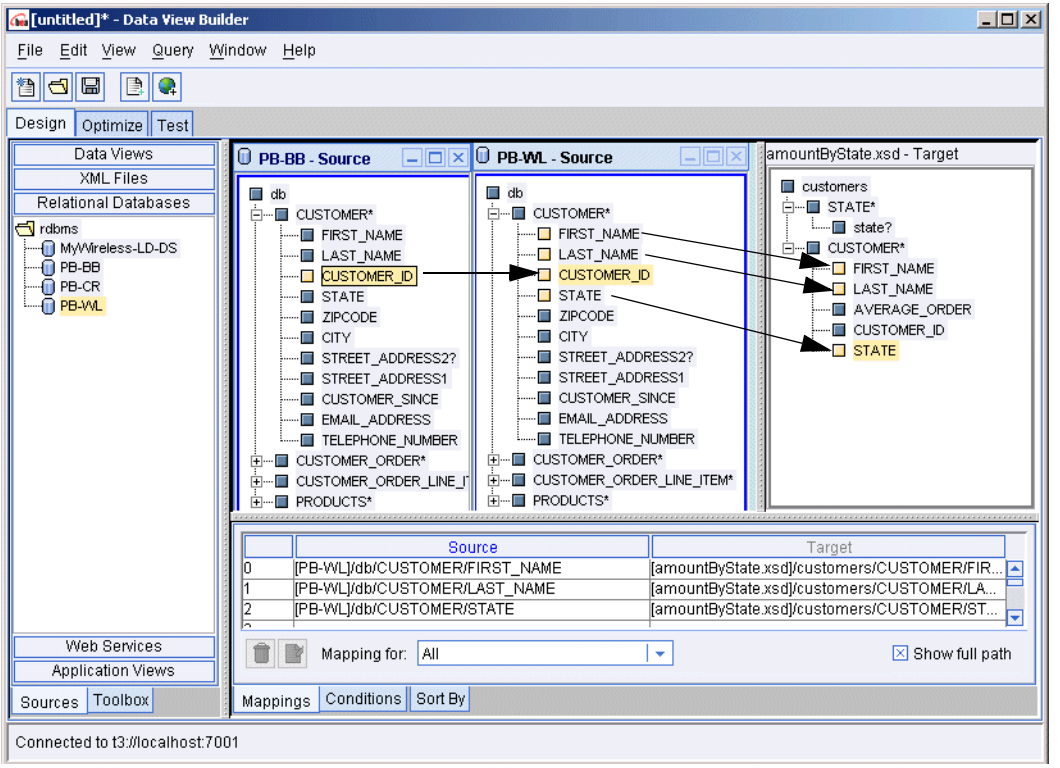
Join Two Sources

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 3-8 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 Output 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 Query Result for Example: Query Customers by ID and Sort

Output 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 2-26) 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)

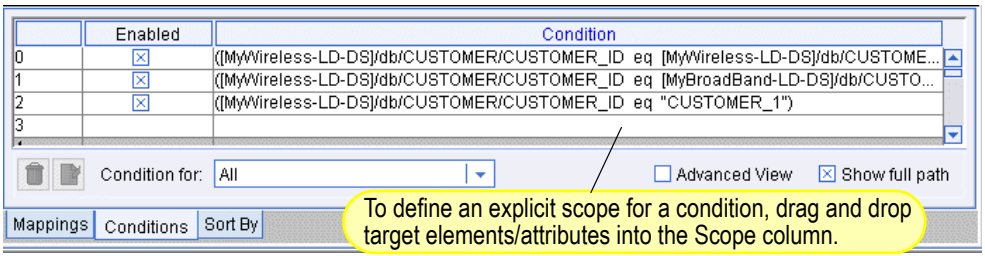
Data View Builder enables you to override the automatic scope setting by using an Advanced view of the existing scope settings. When you switching to an Advanced view, Data View Builder displays the setting it selected for automatic scoping. You can change any or all of the individual scope settings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

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.

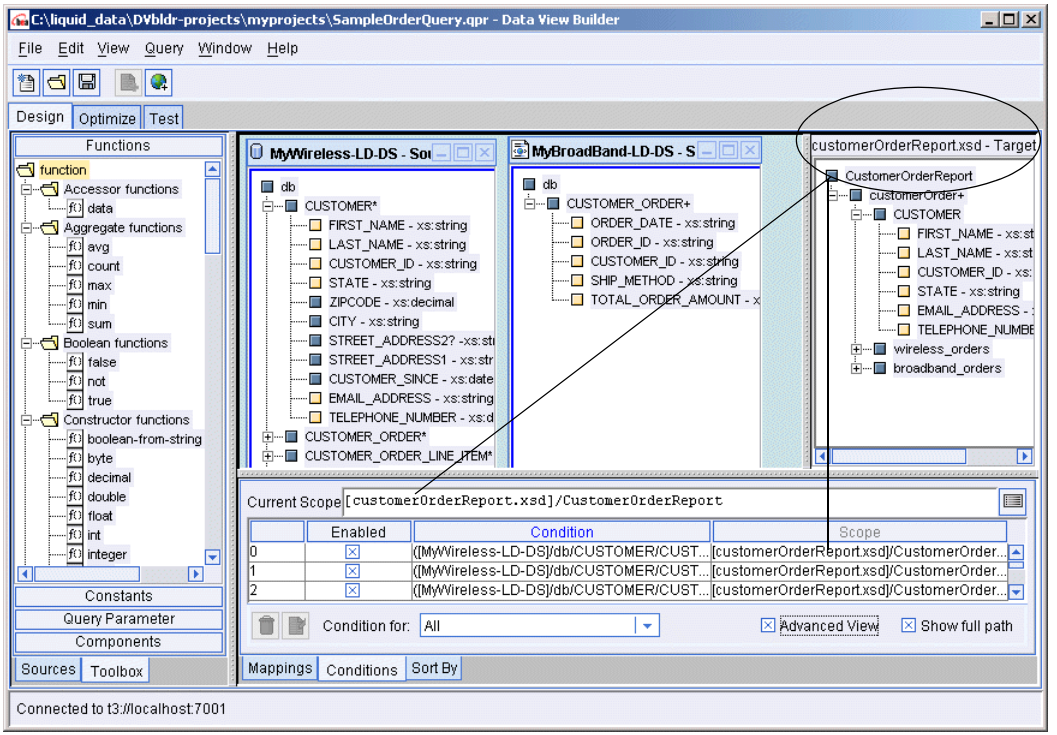
- The Current Scope initially shows the target schema root. Before you map schema sections and create conditions, you can drag a repeatable target schema or function input node to set the scope for a complete section of the target schema. Thereafter, the value in the Current Scope text box determines 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 3-9 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 3-10 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 3-33.

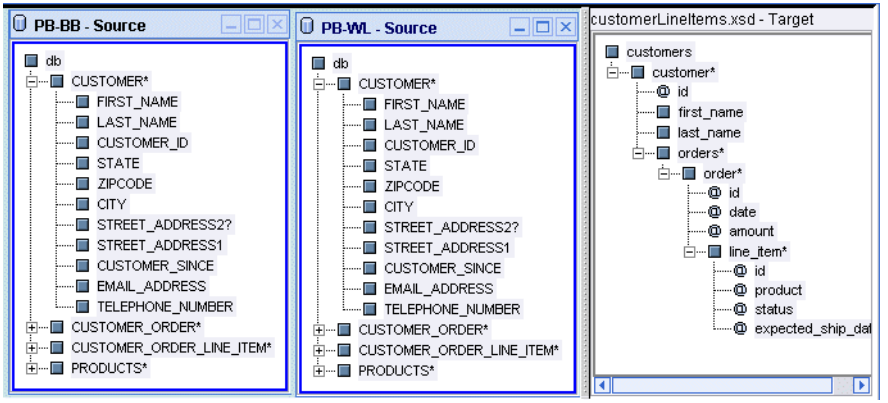
A less common case is when you want to create an assertion. For example, the Liquid Data Server should return a result only when a certain condition occurs. You can accomplish this if you switch to the Advanced view, create the condition, and set the scope for the condition to be the root of the target 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 3-11 of two source schemas: PB-BB and PB-WL, and the target schema customerLineItems.xsd.

Figure 3-11 Schemas for Manual Scope Example

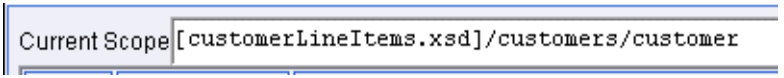


The target schema, customerLineItems.xsd, has a hierarchical structure. There are three distinct sections in the schema that represent repeatable data. customer and order each have an asterisk (*) as the occurrence indicator. line_item has a plus sign (+) as the occurrence indicator. This means that 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:

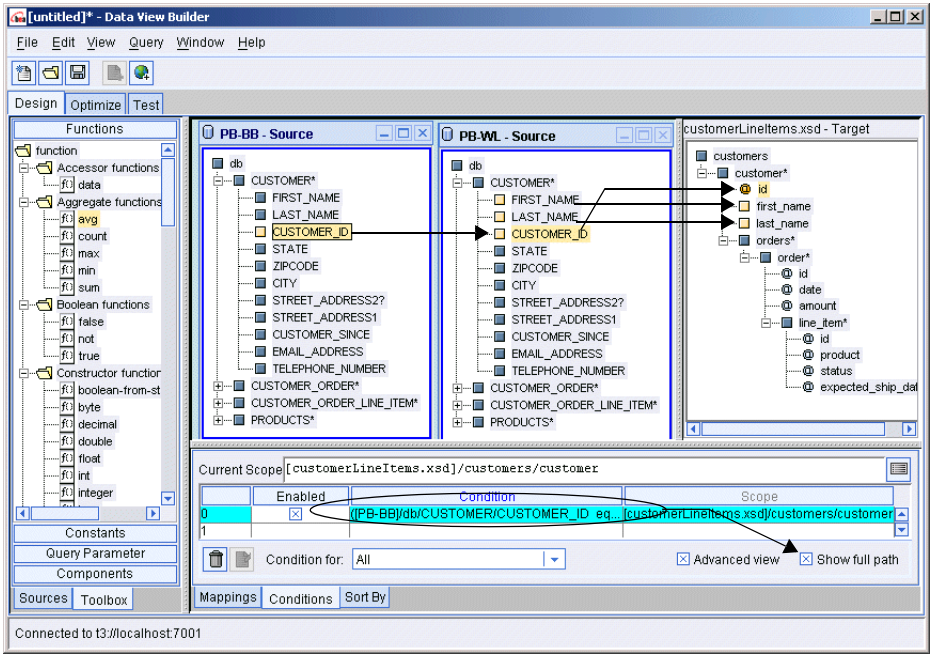
   - Map [PB-WL]/db/CUSTOMER*/FIRST_NAME to [customerLineItems.xsd]/customers/customer*/first_name.
   - Map [PB-WL]/db/CUSTOMER*/LAST_NAME to [customerLineItems.xsd]/customers/customer*/last_name.
   - Map [PB-WL]/db/CUSTOMER*/CUSTOMER_ID to [customerLineItems.xsd]/customers/customer*/id.

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.



5. 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

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

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 3-40 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 3-44
   - "Example 4: Find all Broadband customers (CUSTOMER is Repeatable and Required)" on page 3-45
   - "Example 5: Find all Broadband customers and return their Wireless orders if the customer has Wireless orders (ORDER is Required and Optional)" on page 3-46
   - "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 3-47

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:

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 and 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
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>
                     </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 3-47.

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:

```
<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_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 6, “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.”

CHAPTER

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.

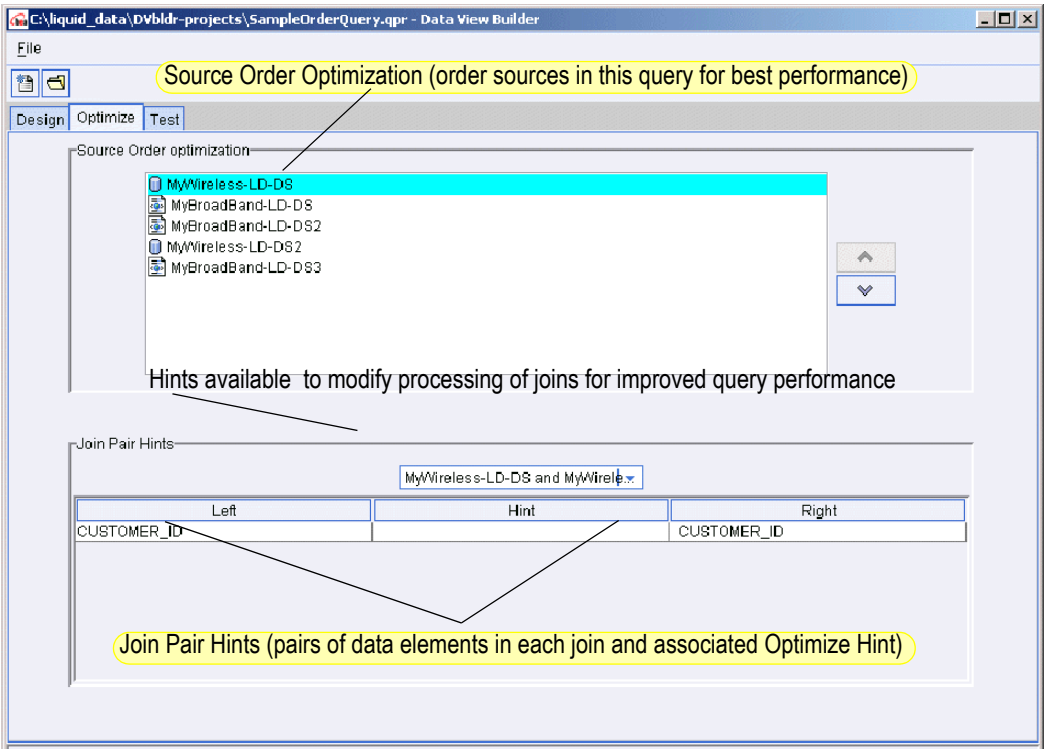
See Tuning Performance in *Deploying Liquid Data* for a broader discussion of factors related to tuning and performance of Liquid Data including query design, data sources, and platform considerations.

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 generates (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)	`{--! ppleft !--}`
Right	Parameter Pass to the Right (ppright)	`{--! ppright !--}`
Merge	Merge	`{--! merge !--}`

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 eq B) eq (C eq D). This type of conditional expression would be treated as A eq (B eq C 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 6-2 in Chapter 6, "Query Cookbook."

Listing 4-1 XQuery with ppright 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  {--! ppright !--} $PB-BB.CUSTOMER_2/CUSTOMER_ID)
                        return
                        <order id={$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  {--! ppright !--} $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 (ppright), which tells the server to retrieve the PB-WL customer

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.

CHAPTER

5 Testing Queries

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

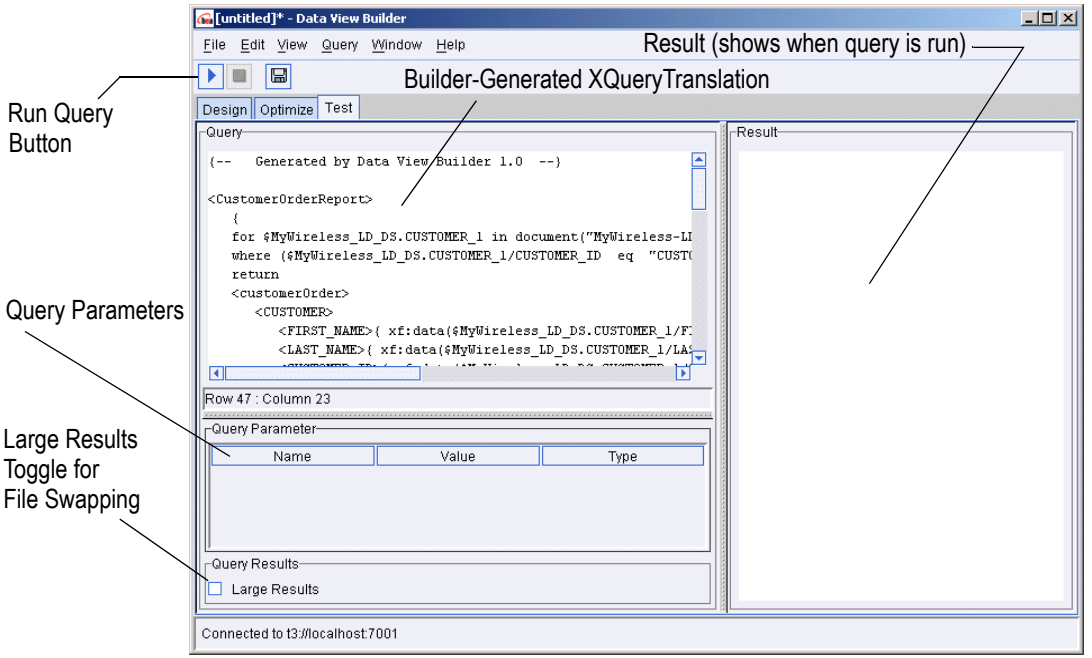
Switching to the Test View

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

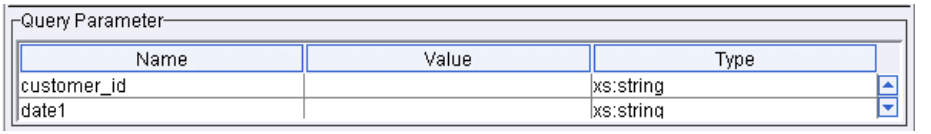


The query you developed on the Design and Optimize tabs is shown in XQuery language in the "Query" window on the upper left panel on the Test tab.

Using Query Parameters

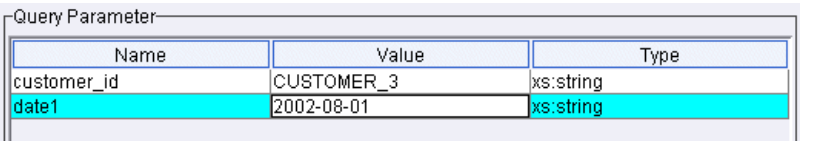
You can use the Query Parameters panel to add variable values to a query each time you run it. The list of variables depends on the number of variables you defined as Query Parameters on the Design tab and which ones appear as one or more function parameters. (For details on defining query parameters, 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 2-16.)

Figure 5-2 Query Parameters Settings on Test Tab



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

Figure 5-3 Entering Values for Query Parameters



For some examples of using query parameters see the following example queries in the Chapter 6, "Query Cookbook."

- "Example 1: Simple Joins" on page 6-2
- "Example 2: Aggregates" on page 6-8
- "Example 3: Date and Time Duration" on page 6-17

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.

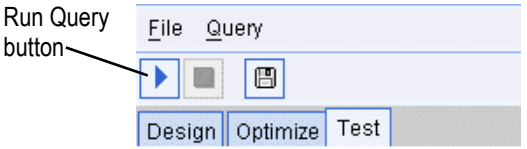
For a query that you know will produce a large result set, you can select "Large Results" in the Query Results panel on the Test tab. (An X next to Large Results indicates that the feature is *on*.) If the Large Results option is selected for a query, then Liquid Data uses swap files to temporarily store 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

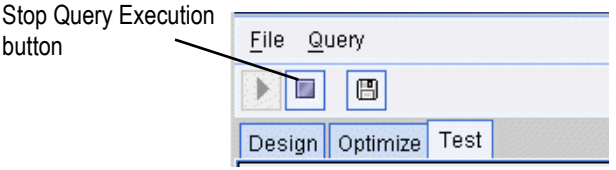


The query is run against your data sources and the result is displayed in the Results panel in XML format.

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.

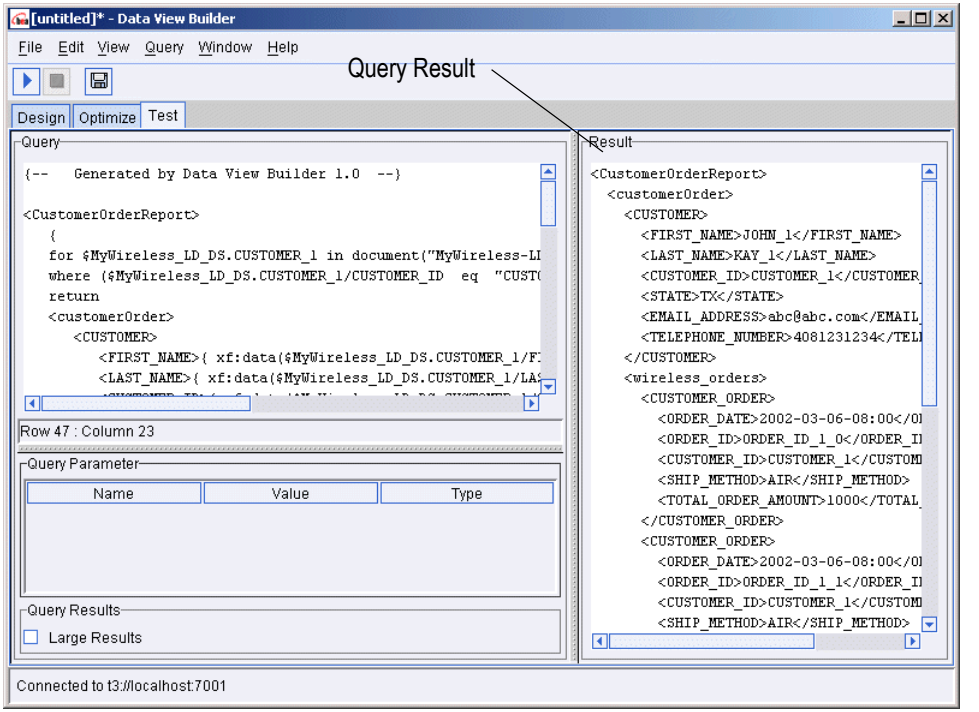
Figure 5-5 Click the "Stop Query Execution Button" to Stop a Running Query



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.

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 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)

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 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

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:

```
<WL_HOME>liquiddata/samples/config/ld_samples/ldrepository/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 6-1 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: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 Toolbar—>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 workspace.

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 Toolbar, 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 6-2 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 6-3 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

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:

```
<WL_HOME>liquiddata/samples/config/ld_samples/ldrepository/data_views/
```

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 6-4 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 the 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 6-1 User Name and Password for Samples WLS Administration Console

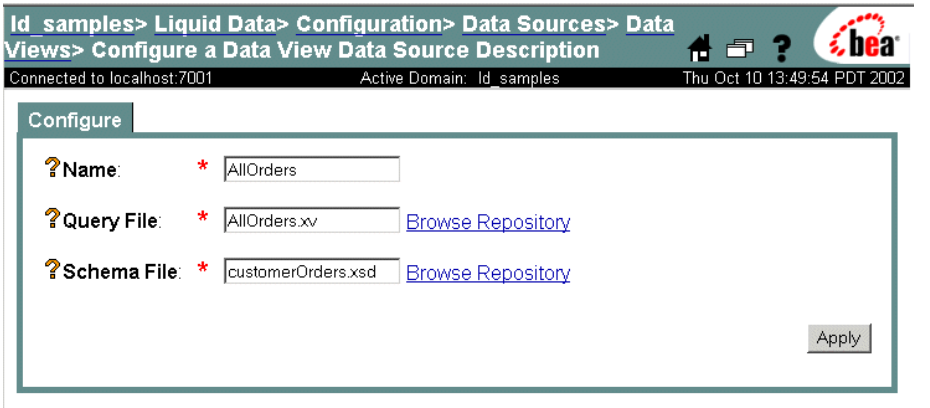
Field	Defaults
Username	`system`
Password	`security`

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. 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 6-1 Configuring Liquid Data Source Description for a Data View



7. Fill in the fields as indicated in the following table.

Table 6-2 Liquid Data Data View Configuration

Field	Description
Name	`AllOrders`
Query File	`AllOrders.xv`
Schema	`customerOrders.xsd`

8. 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 Toolbar.

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:

```
<WL_HOME>liquiddata/samples/config/ld_samples/ldrepository/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 6-5 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:attribute name="id" type="xsd:string"/>
                         <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 Toolbar—>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 workspace.

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.

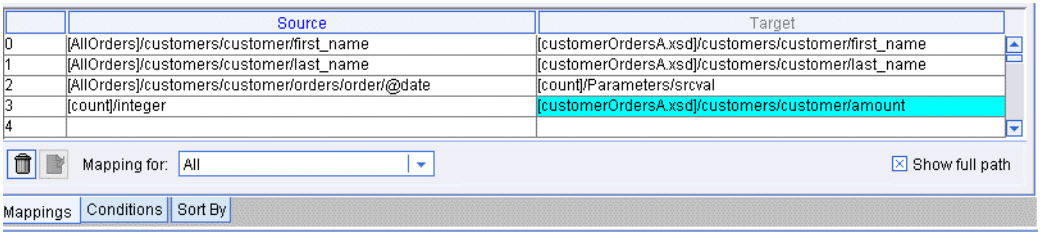
3. Drag and drop [AllOrders]/customer/orders/order/date from the AllOrders source schema onto [count-Function]input/Parameters/*srcval*.
4. Drag and drop [count-Function]Output/*`integer`* to [customerOrdersA.xsd]/customers/customer/amount in the target schema.

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

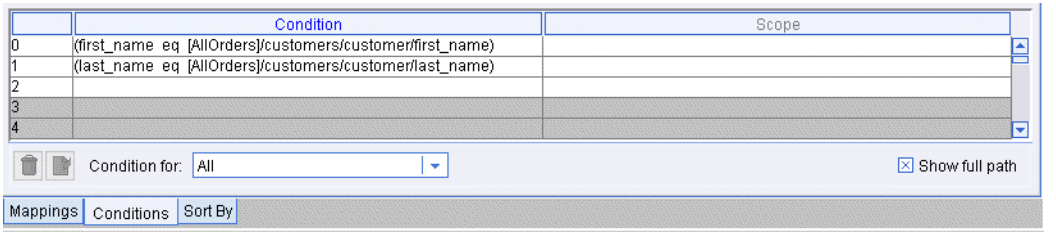
Your mappings should look like those shown in Figure 6-2.

Figure 6-2 Mappings for Example2: Aggregates



Your Conditions should like those shown in Figure 6-3.

Figure 6-3 Conditions for Example 2: Aggregates



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 6-6 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 6-7 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 "DateTime Functions" on page A-22 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 a variables that you define as query parameters to be submitted at query runtime).

To create the solution, follow these steps:

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

```
<WL_HOME>liquiddata/samples/config/ld_samples/ldrepository/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 6-8 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: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 Toolbar—>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 workspace.
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 6-9 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 6-10 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 retrieves results from two or more sources, but unlike a Join query there are no conditions across sources. 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 Broadband orders, and a repeatable list of 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. Create a Query Parameter for a Customer ID
- Ex 4: Step 4. Assign Parameters, Define Source Relationships, and Project the Output
- Ex 4: Step 5. View the XQuery and Run the Query to Test it
- Ex 4: Step 6. 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:

```
<WL_HOME>liquiddata/samples/config/ld_samples/ldrepository/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 6-11 XML Source for unionOrders.xsd Target Schema File

```
<?xml version = "1.0" encoding = "UTF-8"?>

<xsd:schema xmlns:xsd = "http://www.w3.org/2000/10/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_bb" 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:element name="orders_wl" 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—>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 workspace.

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. 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 4. Assign Parameters, Define Source Relationships, and Project the Output

1. 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.

2. 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.

3. Project the Broadband order information.

   - Drag and drop [PB-BB]/db/CUSTOMER_ORDER/TOTAL_ORDER_AMOUNT onto [UnionOrders.xsd]/customers/customer/orders_bb/order/amount.
   - Drag and drop [PB-BB]/db/CUSTOMER_ORDER/ORDER_DATE onto [UnionOrders.xsd]/customers/customer/orders_bb/order/date.

4. Assign the query parameter *customer_id* to the Wireless customer ID as follows:

   Drag and drop query parameter *customer_id* onto [PB-WL]/db/CUSTOMER/CUSTOMER_ID.

5. Within PB-WL, join the Wireless (PB-WL) Customer ID to the order Customer ID.

   Drag and drop [PB-WL]/db/CUSTOMER/CUSTOMER_ID onto [PB-WL]/db/CUSTOMER_ORDER/CUSTOMER_ID.

6. Project the Wireless (PB-WL) order information.

   - Drag and drop [PB-WL]/db/CUSTOMER_ORDER/TOTAL_ORDER_AMOUNT onto [UnionOrders.xsd]/customers/customer/orders_wl/order/amount.
   - Drag and drop [PB-WL]/db/CUSTOMER_ORDER/ORDER_DATE onto [UnionOrders.xsd]/customers/customer/orders_wl/order/date.

7. 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 5. 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 6-12 XQuery for Example 4: Union

```
{--     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($PB-BB.CUSTOMER_1/LAST_NAME) }</last_name>
        <state>{ xf:data($PB-BB.CUSTOMER_1/STATE) }</state>
        <orders_bb>
               {
```

```
        for $PB-BB.CUSTOMER_ORDER_2 in document("PB-BB")/db/CUSTOMER_ORDER
        where ($PB-BB.CUSTOMER_1/CUSTOMER_ID  eq  $PB-BB.CUSTOMER_ORDER_2/CUSTOMER_ID)
        return
        <order>
        <date>{ xf:data($PB-BB.CUSTOMER_ORDER_2/ORDER_DATE) }</date>
        <amount>{ xf:data($PB-BB.CUSTOMER_ORDER_2/TOTAL_ORDER_AMOUNT) }</amount>
        </order>
                        }
        </orders_bb>
        {
        for $PB-WL.CUSTOMER_3 in document("PB-WL")/db/CUSTOMER
        for $PB-WL.CUSTOMER_ORDER_4 in document("PB-WL")/db/CUSTOMER_ORDER
        where ($PB-WL.CUSTOMER_3/CUSTOMER_ID  eq  $PB-WL.CUSTOMER_ORDER_4/CUSTOMER_ID)
        and ($#customer_id of type xs:string  eq  $PB-WL.CUSTOMER_3/CUSTOMER_ID)
        return
        <orders_wl>
                <order>
                <date>{ xf:data($PB-WL.CUSTOMER_ORDER_4/ORDER_DATE) }</date>
                <amount>{ xf:data($PB-WL.CUSTOMER_ORDER_4/TOTAL_ORDER_AMOUNT)
        }</amount>
        </order>
        </orders_wl>
        }
        </customer>
        }
</customers>
```

2. In 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 6. 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 6-13 Result for Example 4: Union

```
<customers>
  <customer>
    <first_name>JOHN_B_4</first_name>
    <last_name>KAY_4</last_name>
    <state>NV</state>
```

```
    <orders_bb>
      <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_bb>
    <orders_wl>
      <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_wl>
  </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 counting 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, then return 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 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. Count the Equalities
- 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

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:

```
<WL_HOME>liquiddata/samples/config/ld_samples/ldrepository/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 6-14 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>
```

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—>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 workspace.
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

1. On the Builder Toolbar—>Toolbox tab, click Functions and double-click on the eq (equal) function (under Operators) to open it.
2. Drag and drop [PB-BB]/db/CUSTOMER/CUSTOMER_ID onto [eq-Function:Input]/Parameters/*anyValue1*.
3. Drag and drop [PB-WL]/db/CUSTOMER/CUSTOMER_ID onto [eq-Function:Input]/Parameters/*anyValue2*.

Ex 5: Step 4. Count the Equalities

1. On the Builder Toolbar—>Toolbox tab, click Functions and double-click on the `count` function (under Aggregate functions) to open it.

2. Drag and drop the [eq-Function:Output]/Boolean onto [count-Function:Input]/Parameters/srcval

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 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:

   ```
   ([count]/integer eq 0)
   ```

   Close the Functions Editor.
5. Project the Broadband customers.
   - 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 6-15 XQuery for Example 5: Minus

```
{--     Generated by Data View Builder 1.0 --}
<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
             let $eq_7 := $PB-BB.CUSTOMER_1/CUSTOMER_ID eq $PB-WL.CUSTOMER_3/CUSTOMER_ID
                  return


                  $eq_7
   let $count_8 := xf:count($srcval_2)
   where ($count_8  eq  0)
   return
   <CUSTOMER>
      <FIRST_NAME>{ xf:data($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 empty because the Broadband and Wireless data sources contain the same customer IDs.

CHAPTER

7 Using Data Views as Data Sources

The result of a query can be referred to as a *data view*. You can use the result of a query as a data source in BEA Liquid Data for WebLogic™. 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.

To use a data view as a data source in this way, you must create a query and save it to the Liquid Data server repository, and then configure a *data view* data source description for the query in the WebLogic Server Administration Console. We recommend 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 describe how to use the result of a query as 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.

Understanding the Relationship Between a Query and a Data View

A data view is the view into the data provided by the result of a particular query with which the data view is associated. As such, the *data view* (query result) is dynamic—it will continue to reshape itself based on any changes that occur in the data it is querying.

You can use the *result* of a query as a data source in Liquid Data. The result can change as your data changes. In this way, you can build on the queries you design to create "views on data views" for a dynamic picture that can respond to fluid information.

You can even further refine the query by creating a “views on a views” or subqueries to zoom in and get a more and more focused view of the information.

Notes:

- For this release, only two levels of data views are supported. For example, if you have a stored query (s1), you can create a data view (v1) out of it. Using v1 as a base view, another view (v2) can be created. However view v2 can not be used as part of a new view—that is, this version of Liquid Data does not support a view v3 that uses v2.
- Liquid Data does not support the use of a parameterized view as a base view.

When To Use Data Views as Data Sources

A data view based on a query constructed in the Data View Builder retains all the tools you used to construct its associated query, including the source and target schemas representations of the data, data source conditions such as joins and unions, and source-to-target mappings.

From the Data View Builder, you can access any data view as a data source for other data views. You can treat the data view just as you would any other data source in the Data View Builder. By using data views as data sources, you can retrieve only the information you need, in the format that you need it, for easier and faster reporting.

For example, suppose you want to create a variety of sales reports from a point-of-sale system that keeps detailed information about every sales transaction in a relational database. In the Data View Builder, you can create a data view that summarizes sales by store and product. After configuring the data view in the Administration Console, you can go back into the Data View Builder, select this data view as a data source, and then create data views that display sales by store, sales by product, and so on.

How to Reuse a Data View as a Data Source

The following sections explain the steps you need to follow to re-use a data view as a data source:

- Create the Query and Save it to the Liquid Data Repository
- Configure a Data View Data Source Description for the Query
- Re-start the Data View Builder and Verify the New Data View Source Shows Up

Create the Query and Save it to the Liquid Data Repository

In the Data View Builder do the following:

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"" on page 5-6 in Chapter 5, "Testing Queries."

Configure a Data View Data Source Description for the Query

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*.

Note: Before you can configure a query as a data view data source in the Administration Console, the query must be saved as a file in the Liquid Data server repository `data_views` folder. If you have already saved the query as a stored query (as described in the previous section), then you can use the Liquid Data Repository tab on the Administration Console to navigate the `stored_queries` folder, and choose "Data View Data Source" for the query you want to configure. 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 more information.

Re-start the Data View Builder and Verify the New Data View Source Shows Up

Re-start Data View Builder and click the Data View button on the Builder toolbar. (For instructions about starting the Data View Builder, see "Overview and Key Concepts" on page 1-1.)

The query shows up as a data view on the Data Views panel in the Builder toolbar in Design View. The new data view appears with the logical name provided in the WebLogic Server Administration Console at configuration time.

Data View Query Example

The Data View Query Sample shows how to create a data view, configure it as a data source, and then use that data source in other data views.

If you have installed the Liquid Data samples, the instructions for setting up and using this Data View sample are provided in the Data View Query Sample readme file. The Samples files are located in the same directory with the readme file at:

BEA_Home/WL_HOME/liquiddata/samples/buildQuery/view/readme.htm

(See the Samples page in the online documentation for more information on available query samples.)

CHAPTER

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 "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 supported in Liquid Data. This functions reference is organized by category as follows:

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.
xsext:anyValue	A subset of xs:anyType including dateTime, boolean, string, numeric values, or any single value. Does not include anyAttribute, anyElement, item, node, sequence, or anySimpleType.
xs:boolean	A subset of xsext:anyValue. A value that supports the mathematical concept of binary-valued logic: true or false.
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	A subset of xsext:anyValue. Represents the leftmost component of dateTime *YYYY-MM-DD* where: ■ *YYYY* is the year ■ *MM* is the month ■ *DD* is the day May be preceded by a leading minus (-) sign to indicate a negative number. If the sign is omitted, plus (+) is assumed. May be immediately followed by a Z to indicate Coordinated Universal Time (UTC) or, to indicate the time zone (the difference between the local time and Coordinated Universal Time), immediately followed by a sign, + or -, followed by the difference from UTC represented as *hh:mm*. **Example:** To specify 1:20 pm on May the 31st, 1999, write: 1999-05-31.
xs:dateTime	A subset of xsext:anyValue. Represents the format *YYYY-MM-DD*T*hh:mm:ss* where: ■ *YYYY* is the year ■ *MM* is the month ■ *DD* is the day ■ T is the date/time separator ■ *hh* is the hour ■ *mm* is the minute ■ *ss* is the second May be preceded by a leading minus (-) sign to indicate a negative number. If the sign is omitted, plus (+) is assumed. Additional digits can be used to increase the precision of fractional seconds if desired (ss.ss...) with any number of digits after the decimal point is supported. May be immediately followed by a Z to indicate Coordinated Universal Time (UTC) or, to indicate the time zone (the difference between the local time and Coordinated Universal Time), immediately followed by a sign, + or -, followed by the difference from UTC represented as hh:mm. **Example:** To specify 1:20 pm on May the 31st, 1999 EST, which is five hours behind Coordinated Universal Time (UTC), write: 1999-05-31T13:20:00-05:00.

Table A-1 Data Types

Data Type Name	Description
xs:decimal	A subset of xsext:anyValue. Includes all integer types, such as xs:integer, xs:long, xs:short, xs:int, or xs:byte. Represents a finite-length sequence of decimal digits (0–9) separated by an optional period as a decimal indicator. An optional leading sign is allowed. If the sign is omitted, plus (+) is assumed. Leading and trailing zeroes are optional. If the fractional part is zero, the period and following zeroes can be omitted. **Examples:** -1.23, 12678967.543233, +100000.00, 210
xs:double	A subset of xsext:anyValue. There are no subordinate data types; however, xs:float and xs:decimal, and all derived types, can be promoted to xs:double in certain cases, such as function calls. Represents a double precision 64-bit floating point value. Supports the special values positive and negative zero, positive and negative infinity and not-a-number (0, -0, INF, -INF and NaN).
xs:float	A subset of xsext:anyValue. There are no subordinate data types; however, xs:decimal, and all derived types, can be promoted to xs:float in certain cases, such as function calls. Represents a Single-precision 32-bit floating point value. Supports the special values positive and negative zero, positive and negative infinity and not-a-number (0, -0, INF, -INF and NaN).
xsext:item	A subset of xs:anyType. Includes xsext:anyValue and xsext:node. Excludes any sequence. Represents a list element, individual value, or attribute.
xs:int	A subset of xs:long. Represents a finite-length sequence of decimal digits (0–9). An optional leading sign is allowed. If the sign is omitted, plus (+) is assumed. **Examples:** -1, 0, 12678967543233, +100000
xs:integer	A subset of xs:decimal. Represents a finite-length sequence of decimal digits (0–9). An optional leading sign is allowed. If the sign is omitted, plus (+) is assumed. **Examples:** -1, 0, 12678967543233, +100000

Table A-1 Data Types

Data Type Name	Description
xs:long	A subset of xs:decimal. A sequence of decimal digits (0–9) with a range of 9223372036854775807 to -9223372036854775808. If the sign is omitted, plus (+) is assumed. **Examples:** -1, 0, 12678967543233, +100000
xsext:node	A subset of xsext:anyValue. A component in a tree structure that represents a data element.
xs:short	A subset of xs:int. A sequence of decimal digits (0–9) with a range of 32767 to -32768. If the sign is omitted, plus (+) is assumed. **Examples:** -1, 0, 12678, +10000
xs:string	A subset of xsext:anyValue. A sequence that contains alphabetic, numeric, or special characters.
xs:time	A subset of xsext:anyValue. Represents the rightmost segment of the dateTime format where: ■ *hh* is the hour ■ *mm* is the minute ■ *ss* is the second May contain an optional following time zone indicator. **Examples:** ■ To indicate 1:20 pm EST, which is five hours behind Coordinated Universal Time (UTC), write: 13:20:00-05:00. ■ Midnight is 00:00:00.

Naming Conventions

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

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

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

Occurrence Indicators

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

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

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

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

Accessor Functions

Accessor functions operate on different types of nodes. They accept single node input and return a value based on the node type.

Table A-2 Accessor Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xf:data	**Data Type:** ■ Input data type: xsext:node? ■ Returned data type: xsext:anyValue? **Description:** Returns the typed-value of each input node. **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.	**Notes:** 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.

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.

Note: In the Data View Builder, you cannot drag and drop aggregate functions to the work area. You must double-click on them to open them so that you can select input parameters.

Table A-3 Aggregate Functions

Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xf:avg	**Data Type:** ■ Input data type: *xs:double** ■ Returned data type: xs:*double?* **Description:** Returns the average of a sequence of numbers. **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.	**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.
xf:count	**Data Type:** ■ Input data type: *xs:item** ■ Returned data type: *xs:integer* **Description:** Returns the number of items in the sequence in an unsigned integer. **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).	**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.

Table A-3 Aggregate Functions

Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xf:max	**Data Type:** ■ Input data type: *xsext:item** ■ Returned data type: *xsext:item?* **Description:** 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. **Examples:** xf:max((3, 10)) returns the value 10. xf:max((<a>{4}, 3, (), {2})) returns <a>{4}.	**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.

Table A-3 Aggregate Functions

Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xf:min	**Data Type:** ■ Input data type: *xsext:item** ■ Returned data type: *xsext:item?* **Description:** 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. **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 ().	**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.

Table A-3 Aggregate Functions

Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xf:sum	**Data Type:** ■ Input data type: *xsext:anyValue** ■ Returned data type: *xsext:anyValue?* **Description:** Returns the sum of a sequence of numbers. **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.	**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*.

Boolean Functions

Boolean functions return true or false values.

Table A-4 Boolean Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Compliance
xf:false	**Data Types:** ■ Input data type: No input data required. ■ Returned data type: *xs:boolean* **Description:** Returns the boolean value false. **Examples:** ■ xf:false() returns false. ■ xf:false(34) generates a compile-time error because the function does not accept any parameters.	**XQuery Specification Compliance:** Conforms to the current specification.
xf:not	**Data Types:** ■ Input data type: *xs:boolean?* ■ Returned data type: *xs:boolean?* **Description:** Returns true if the value of the source value is false and false if the value of the source value is true. **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.	**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.

Table A-4 Boolean Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Compliance
xf:true	**Data Types:** ■ Input data type: No input data required. ■ Returned data type: *xs:boolean* **Description:** Returns the boolean value true. **Examples:** ■ xf:true() returns true. ■ xf:true("34") generates a compile-time error because the function does not accept any parameters.	**XQuery Specification Compliance:** Conforms to the current specification.

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.

Table A-5 Accessor Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xf:boolean-from-string	**Data Types:** ■ Input data type: *xs:string?* ■ Returned data type: *xs:boolean?* **Description:** Returns a boolean value of true or false from the string source value. **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.	**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.

Table A-5 Accessor Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xf:byte	**Data Types:** ■ Input data type: *xsext:anyValue?* ■ Returned data type: *xs:byte?* **Description:** Constructs a byte integer value from the string source value. **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.	**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.

Table A-5 Accessor Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xf:decimal	**Data Types:** Input data type: *xsext:anyValue?* Returned data type: *xs:decimal?* **Description:** Constructs a decimal value from the source value. **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.	**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.

Table A-5 Accessor Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xf:double	**Data Types:** ■ Input data type: *xsext:anyValue?* ■ Returned data type: *xs:double?* **Description:** Constructs a double precision value from the source value. **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.	**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.

Table A-5 Accessor Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xf:float	**Data Types:** Input data type: *xsext:anyValue?* Returned data type: *xs:float?* **Description:** Constructs a floating point value from the source value. **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.	**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.

Table A-5 Accessor Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xf:int	**Data Types:** Input data type: *xsext:anyValue?* Returned data type: *xs:integer?* **Description:** Constructs an integer value from the source value. The largest integer value is limited to a 32-bit expression. **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.	**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.
xf:integer	**Data Types:** Input data type: *xsext:anyValue?* Returned data type: *xs:integer?* **Description:** Constructs an integer value from the source value. The largest integer value is limited to a 32-bit expression. **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.	**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.

Table A-5 Accessor Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xf:long	**Data Types:** ■ Input data type: *xsext:anyValue?* ■ Returned data type: *xs:long?* **Description:** 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. **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.	**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.

Table A-5 Accessor Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xf:short	**Data Types:** Input data type: *xsext:anyValue?* Returned data type: *xs:short?* **Description:** Constructs a two-byte integer value from the source value. The largest short integer value is limited to a 16-bit expression. **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.	**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.

Table A-5 Accessor Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xf:string	**Data Types:** ■ Input data type: *xs:anyType* ■ Returned data type: *xs:string?* **Description:** Constructs a string value from the source value. The source value can be a sequence, a node of any kind, or a simple value. **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."	**Notes:** Liquid Data accepts any simple value, but supports no other accessor types, such as a sequence or other type of node. **XQuery Specification Compliance:** ■ Liquid Data treats xf:string as both a constructor and an accessor. ■ Liquid Data supports only the string format that requires one node of any type as the input. ■ Liquid Data accepts xsext:anyType input instead of a list of items. ■ Liquid Data returns an optional string. ■ Liquid Data does not recognize entities.

DateTime Functions

DateTime functions extract all or part of a dateTime expression and use it in a query.

xf:add-days

Data Types

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

Description

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

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

Data Types

No parameters required.

Returned data type: xs:*dateTime*

Description

Returns the current date and time.

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

Data Types

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

Description

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 the following is true:

- *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

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

Data Types

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

Description

Returns the leftmost date portion of a dateTime value.

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-5. For more information about valid formats for dateTime, see "xf:dateTime" on page A-28.

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

Data Types

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

Description

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-39.

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-5.

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

Data Types

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

Description

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

Notes

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

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

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

Liquid Data supports this year range: 0000–9999.

XQuery Specification Compliance

Conforms to the current specification.

Examples

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

xfext:dateTime-from-string-with-format

Data Types

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

Description

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

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-5, and see "Date and Time Patterns" on page A-39.

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

Data Types

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

Description

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

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

Data Types

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

Description

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

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

Data Types

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

Description

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

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

Data Types

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

Description

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

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

Data Types

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

Description

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

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

Data Types:

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

Description

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

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

Data Types

- Input data type: *xs:string?*

- Returned data type: *xs:time?*

Description

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

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

where the following is true:

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

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

Data Types

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

Description

Returns the time from *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-5. For more information about valid formats for dateTime, see "xf:dateTime" on page A-28.

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

Data Types

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

Description

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

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-5, and see "Date and Time Patterns" on page A-39.

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.

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 7-1 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

Table 7-1 Date and Time Patterns

This Symbol	Represents This Data	Produces This Result
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*.

Node Functions

Node functions are operations on nodes and node values.

Table A-6 Node Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xf:local-name	**Data Types:** ■ Input data type: *xsext:node* ■ Returned data type: *xs:string?* **Description:** Returns a string value that corresponds to the local name of the specified node. **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.	**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.

Numeric Functions

Numeric functions operate on numeric data types.

Table A-7 Numeric Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xf:ceiling	**Data Types:** ■ Input data type: *xs:double*? ■ Returned data type: *xs:integer?* **Description:** Returns the smallest (closest to negative infinity) integer that is not smaller than the source value. **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.	**Notes:** If the argument is an empty list, the function returns an empty list. **XQuery Specification Compliance:** Conforms to the current specification.

Table A-7 Numeric Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xf:floor	**Data Types:** ■ Input data type: *xs:double?* ■ Returned data type: xs:*integer?* **Description:** Returns the largest (closest to positive infinity) integer that is not greater than the source value. **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.	**Notes:** If the argument is an empty list, the function returns an empty list. **XQuery Specification Compliance:** Conforms to the current specification.

Table A-7 Numeric Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xf:round	**Data Types:** ■ Input data type: *xs:double?* ■ Returned data type: *xs:integer?* **Description:** Returns the integer that is closest to the source value. **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.	**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.

Comparison and Numeric Operators

XQuery has operators that are specific to types of operations, such as comparisons or numeric operations.

Table A-8 Comparison and Numeric Operators

Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
*** (Multiply)**	**Data Types:** ■ Parameter1 data type: *xs:anyValue?* ■ Parameter2 data type: *xs:anyValue?* ■ Returned data type: *xs:anyValue?* **Description:** Returns the arithmetic product of the operands: ($operand1*$operand2). **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.	**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.

Table A-8 Comparison and Numeric Operators

Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
+ (Add)	**Data Types:** ■ Parameter1 data type: *xs:anyValue?* ■ Parameter2 data type: *xs:anyValue?* ■ Returned data type: *xs:anyValue?* **Description:** Returns the arithmetic sum of the operands: ($operand1+$operand2). **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.	**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.

Table A-8 Comparison and Numeric Operators

Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
-(Subtract)	**Data Types:** ■ Parameter1 data type: *xs:anyValue?* ■ Parameter2 data type: *xs:anyValue?* ■ Returned data type: *xs:anyValue?* **Description:** Returns the arithmetic difference of the operands: ($operand1-$operand2). **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.	**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.

Table A-8 Comparison and Numeric Operators

Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
and	**Data Types:** ■ Parameter1 data type: *xs:boolean?* ■ Parameter2 data type: *xs:boolean?* ■ Returned data type: *xs:boolean?* **Description:** The result is true if both values are true, and false if one of the values is false. **Examples:** ■ xf:true() and xf:true() returns the boolean value true. ■ xf:true() and xf:false() returns the boolean value false. ■ xf:false() and xf:false() returns the boolean value false. ■ xf:true() and (<a/>, <b/>) generates a compile-time error because lists are not supported as input parameters to boolean operators. ■ xf:false() and "false" generates a compile-time error because the second parameter is not a boolean value.	**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.

Table A-8 Comparison and Numeric Operators

Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
div(ide)	**Data Types:** ■ Parameter1 data type: *xs:anyValue?* ■ Parameter2 data type: *xs:anyValue?* ■ Returned data type: *xs:anyValue?* **Description:** Returns the arithmetic quotient of the operands ($operand1/$operand2). **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.	**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.

Table A-8 Comparison and Numeric Operators

Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
eq	**Data Types:** ■ Parameter1 data type: *xsext:anyValue?* ■ Parameter2 data type: *xsext:anyValue?* ■ Returned data type: *xs:boolean?* **Description:** Returns true if Parameter1 is exactly equal to Parameter2. **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.	**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.

Table A-8 Comparison and Numeric Operators

Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
ge	**Data Types:** ■ Parameter1 data type: *xsext:anyValue?* ■ Parameter2 data type: *xsext:anyValue?* ■ Returned data type: *xs:boolean?* **Description:** Returns true if Parameter1 is greater than or equal to Parameter2. **Examples:** See the examples for "eq" operator (previous entry in this table).	**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.

Table A-8 Comparison and Numeric Operators

Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
gt	**Data Types:** ■ Parameter1 data type: *xsext:anyValue?* ■ Parameter2 data type: *xsext:anyValue?* ■ Returned data type: *xs:boolean?* **Description:** Returns true if Parameter1 is greater than Parameter2. **Examples:** See the examples for the "eq" operator (previous entry in this table).	**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.

Table A-8 Comparison and Numeric Operators

Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
le	**Data Types:** ■ Parameter1 data type: *xsext:anyValue?* ■ Parameter2 data type: *xsext:anyValue?* ■ Returned data type: *xs:boolean?* **Description:** Returns true if Parameter1 is less than or equal to Parameter2. **Examples:** See the examples for the "eq" operator (previous entry in this table).	**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.

Table A-8 Comparison and Numeric Operators

Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
lt	**Data Types:** ■ Parameter1 data type: *xsext:anyValue?* ■ Parameter2 data type: *xsext:anyValue?* ■ Returned data type: *xs:boolean?* **Description:** Returns true if Parameter1 is less than or equal to Parameter2. **Examples:** See the examples for the "eq" operator (previous entry in this table).	**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.

Table A-8 Comparison and Numeric Operators

Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
mod	**Data Types:** ■ Parameter1 data type: *xs:anyValue?* ■ Parameter2 data type: *xs:anyValue?* ■ Returned data type: *xs:anyValue?* **Description:** Returns the remainder after dividing the first operand by the second operand: ($operand1 mod $operand2). **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.	**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.

Table A-8 Comparison and Numeric Operators

Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
ne	**Data Types:** ■ Parameter1 data type: *xsext:boolean?* ■ Parameter2 data type: *xsext:boolean?* ■ Returned data type: *xs:boolean?* **Description:** 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. **Examples:** See the examples for the "eq" operator (previous entry in this table).	**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.

Table A-8 Comparison and Numeric Operators

Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
or	**Data Types:** ■ Parameter1 data type: *xs:boolean?* ■ Parameter2 data type: *xs:boolean?* ■ Returned data type: *xs:boolean?* **Description:** 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. **Examples:** ■ xf:true() or xf:true() returns the boolean value true. ■ xf:true() or xf:false() returns the boolean value true. ■ xf:false() or xf:false() returns the boolean value false. ■ xf:true() or (<a/>, <b/>) generates a compile-time error because lists are not supported as parameters to boolean operators. ■ xf:false() or "false" generates a compile-time error because the second parameter is not a boolean value.	**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.

Other Functions

Table A-9 Other Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xfext:if-then-else	**Data Types:** ■ Parameter1 data type: *xs:boolean?* ■ Parameter2 data type: *xs:anyValue?* ■ Parameter3 data type: *xs:anyValue?* ■ Returned data type: *xs:anyValue* **Description:** The xfext:if-then-else function accepts the value of a boolean parameter to select one of two other input parameters. **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.	**Notes:** The If-then-else function is an extended function. For more information about extended functions, see "Naming Conventions" on page A-5. 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.

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.

Table A-10 Sequence Functions

Data Type, Description, and Examples	Notes and XQuery Specification Compliance
Function: **xf:distinct-values**	
Data Types: ■ Input data type: *xsext:item** ■ Returned data type: *xsext:anyValue** **Description:** If the source value contains only nodes, the function removes duplicates and returns a subset of unique 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.	**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.
Function: **xf:empty**	
Data Types: ■ Input data type: *xsext:item** ■ Returned data type: *xs:boolean?* **Description:** Returns true if the specified list of items is empty; otherwise, returns false. **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.	**XQuery Specification Compliance:** Liquid Data supports an optional boolean returned value.

Table A-10 Sequence Functions

Data Type, Description, and Examples	Notes and XQuery Specification Compliance
Function: **xfext:subsequence**	
Format 1 Data Types: Parameter1 data type: *xsext:item** Parameter2 data type: *xs:integer* Returned data type: *xsext:item** **Format 2 Data Types:** Parameter1 data type: *xsext:item** Parameter2 data type: *xs:integer* Parameter3 data type: *xs:integer* Returned data type: *xsext:item** **Description:** **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. **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. **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 (().	**Notes:** **Format 1:** 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. **Format 2:** 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:** ■ 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.

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*.

Table A-11 String Functions

Data Type, Description, and Examples	Notes and XQuery Specification Compliance
Function: **xf:compare**	
Data Types: ■ Parameter1 data type: *xs:string?* ■ Parameter2 data type: *xs:string?* ■ Returned data type: *xs:integer?* **Description:** 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. **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 ().	**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.

Table A-11 String Functions

Data Type, Description, and Examples	Notes and XQuery Specification Compliance
Function: **xf:concat**	
Data Types: Parameter1 data type: *xs:string?* Parameter2 data type: *xs:string?* Returned data type: *xs:string?* **Description:** Returns a string that concatenates Parameter1 with Parameter2. **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.	**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.

Table A-11 String Functions

Data Type, Description, and Examples	Notes and XQuery Specification Compliance
Function: **xf:contains**	
Data Types: Parameter1 data type: *xs:string?* Parameter2 data type: *xs:string?* Returned data type: *xs:boolean?* **Description:** 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. **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.	**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.

Table A-11 String Functions

Data Type, Description, and Examples	Notes and XQuery Specification Compliance
Function: **xf:ends-with**	
Data Types: ■ Parameter1 data type: *xs:string?* ■ Parameter2 data type: *xs:string?* ■ Returned data type: *xs:boolean?* **Description:** Returns a boolean value or true or false indicating whether Parameter1 ends with a string that is equal to Parameter2. **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.	**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.

Table A-11 String Functions

Data Type, Description, and Examples	Notes and XQuery Specification Compliance
Function: **xf:lower-case**	
Data Types: ■ Input data type: *xs:string?* ■ Returned data type: *xs:string?* **Description:** Returns the value of the input string after translating every uppercase letter to its corresponding lower-case value. **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.	**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.

Table A-11 String Functions

Data Type, Description, and Examples	Notes and XQuery Specification Compliance
Function: **xf:starts-with**	
Data Types: ■ Parameter1 data type: *xs:string?* ■ Parameter2 data type: *xs:string?* ■ Returned data type: *xs:boolean?* **Description:** Returns a boolean value or true or false indicating whether Parameter1 starts with a string that is equal to Parameter2. **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.	**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.

Table A-11 String Functions

Data Type, Description, and Examples	Notes and XQuery Specification Compliance
Function: **xf:string-length**	
Data Types: ■ Input data type: *xs:string?* ■ Returned data type: *xs:integer?* **Description:** Returns an integer equal to the number of characters in the input source string. **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.	**Notes:** If the source value is an empty list, the function returns an empty list. Liquid Data generates an error if either parameter is not a string. **XQuery Specification Compliance:** ■ Liquid Data treats xf:string as both a constructor and an accessor. ■ Liquid Data supports only the string format that requires one node of any type as the input. ■ Liquid Data accepts xsext:anyType input instead of a list of items. ■ Liquid Data returns an optional string. ■ Liquid Data does not recognize entities.

Table A-11 String Functions

Data Type, Description, and Examples	Notes and XQuery Specification Compliance
Function: **xf:substring**	
Format 1 Data Types: ■ Parameter1 data type: *xs:string?* ■ Parameter2 data type: *xs:integer?* ■ Returned data type: *xs:string?* **Format 2 Data Types:** ■ Parameter1 data type: *xs:string?* ■ Parameter2 data type: *xs:integer?* ■ Parameter3 data type: *xs:integer?* ■ Returned data type: *xs:string?* **Description:** **Format 1:** Returns that part of the Parameter1 source string from the starting location specified by Parameter2. **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.	**Notes:** **Format 1** 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. **Format 2** 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.

Table A-11 String Functions

Data Type, Description, and Examples	Notes and XQuery Specification Compliance
Function: **xf:substring-after**	
Data Types: ■ Parameter1 data type: *xs:string?* ■ Parameter2 data type: *xs:string?* ■ Returned data type: *xs:string?* **Description:** Returns that part of the Parameter1 source string that follows the first occurrence of those characters specified in Parameter2. **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.	**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.

Table A-11 String Functions

Data Type, Description, and Examples	Notes and XQuery Specification Compliance
Function: **xf:substring-before**	
Data Types: ■ Parameter1 data type: *xs:string?* ■ Parameter2 data type: *xs:string?* ■ Returned data type: *xs:string?* **Description:** Returns that part of the Parameter1 source string that precedes the first occurrence of those characters specified in Parameter2. **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.	**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.

Table A-11 String Functions

Data Type, Description, and Examples	Notes and XQuery Specification Compliance
Function: **xf:extrim**	
Data Types: ■ Input data type: *xs:string?* ■ Returned data type: *xs:string?* **Description:** Returns the value of the input string with leading and trailing white space removed from the string. **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.	**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-5.
Function: **xf:upper-case**	
Data Types: ■ Input Parameter data type = *xs:string?* ■ Returned data type: *xs:string?* **Description:** Returns the value of the input string after translating every lower-case letter to its uppercase correspondent.	**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.

Type Casting Functions

Type cast functions process a source value as the argument. 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.

Table A-12 Type Casting Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xs:boolean	**Data Types:** ■ Input data type: *xs:anyValue* ■ Returned data type: *xs:boolean* ■ **Description:** 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. **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 ().	**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.

Table A-12 Type Casting Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xs:date	**Data Types:** ■ Input data type: *xs:anyValue* ■ Returned data type: *xs:date* **Description:** Converts the input to a date value. **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.	**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.

Table A-12 Type Casting Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xs:dateTime	**Data Types:** ■ Input data type: *xs:anyValue* ■ Returned data type: *xs:dateTime* **Description:** Converts the input to a dateTime value. **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.	**Notes:** This function uses the xf:date function. **XQuery Specification Compliance:** Conforms to the current specification.
xs:decimal	**Data Types:** ■ Input data type: *xs:anyValue* ■ Returned data type: *xs:decimal* **Description:** Converts the input to a decimal value. **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.	**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.

Table A-12 Type Casting Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xs:double	**Data Types:** ■ Input data type: *xs:anyValue* ■ Returned data type: *xs:double* **Description:** Converts the input to a double precision value. **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.	**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.
xs:float	**Data Types:** ■ Input data type: *xs:anyValue* ■ Returned data type: *xs:float* **Description:** Converts the input to a floating point value. **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.	**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.

Table A-12 Type Casting Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xs:string	**Data Types:** ■ **Input data type:** *xs:anyType* ■ **Returned data type:** *xs:string* **Description:** Converts the input to a string value. **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."	**Notes:** This function uses the xf:string function. **XQuery Specification Compliance:** ■ Liquid Data treats xf:string as both a constructor and an accessor. ■ Liquid Data supports only the string format that requires one node of any type as the input. ■ Liquid Data accepts xsext:anyType input instead of a list of items. ■ Liquid Data returns an optional string. ■ Liquid Data does not recognize entities.

Table A-12 Type Casting Functions

Function Name	Data Type, Description, and Examples	Notes and XQuery Specification Compliance
xs:time	**Data Types:** ■ Input data type: *xs:anyValue* ■ Returned data type: *xs:time* **Description:** Converts the input to a time value. **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).	**Notes:** This function uses the xf:time function. **XQuery Specification Compliance:** Conforms to the current specification.

CHAPTER

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

In relational databases, data types are described using two methods. The conventional way is to use a JDBC number. For example, an integer is 4, varchar is 12, a date is 91, an so on. These numbers are represented by constants in the java.sql.Types class, such as Types.INTEGER = 4 and Types.VARCHAR = 12. This numbering system describes all the JDBC standardized types. However, there are many vendor-specific types, and most of them use the default JDBC number 1111, meaning "other." For this method, there is a name instead of a number associated with each type.

The 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	*not supported*
Types.BIGINT	long
Types.BINARY	string
Types.BIT	boolean
Types.BLOB	*not supported*
Types.CHAR	string
Types.CLOB	*not supported*
Types.DATE	date
Types.DECIMAL	decimal
Types.DOUBLE	double
Types.FLOAT	double

JDBC Type	Liquid Data Data Type
Types.INTEGER	integer
Types.JAVA_OBJECT	*not supported*
Types.LONGVARBINARY	string
Types.LONGVARCHAR	string
Types.NUMERIC	decimal
Types.REAL	float
Types.REF	string
Types.SMALLINT	short
Types.STRUCT	*not supported*
Types.TIME	time
Types.TIMESTAMP	dateTime
Types.TINYINT	byte
Types.VARBINARY	string
Types.VARCHAR	string

JDBC Names

The following table maps the JDBC name to Liquid Data data types.

JDBC Name	Liquid Data Data Type
ARRAY	*not supported*
BIGINT	long
BINARY	string

JDBC Name	Liquid Data Data Type
BIT	boolean
BLOB	*not supported*
CHAR	string
CLOB	*not supported*
DATE	date
DEC	decimal
DECIMAL	decimal
DOUBLE	double
FLOAT	double
INTEGER	integer
JAVA_OBJECT	*not supported*
LONGVARBINARY	string
LONGVARCHAR	string
NUM	decimal
NUMERIC	decimal
REAL	float
REF	string
SMALLINT	short
STRUCT	*not supported*
TIME	time
TIMESTAMP	dateTime
TINYINT	byte
VARBINARY	string

JDBC Name	Liquid Data Data Type
VARCHAR	string

Oracle Names

The following table maps Oracle names to Liquid Data data types.

Oracle Name	Liquid Data Data Type
FLOAT	float
BFILE	*not supported*
LONG	*not supported*
LONG RAW	*not supported*
NCHAR	string
NCLOB	*not supported*
NUMBER	decimal
NVARCHAR2	string
RAW	string
ROWID	string
UROWID	*not supported*
VARCHAR2	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	dateTime
IMAGE	*not supported*
INT	integer
MONEY	float
NTEXT	string// too big?
NVARCHAR	string
SMALLDATETIME	dateTime
SMALLMONEY	float
SQL_VARIANT	string
UNIQUEIDENTIFIER	string

DB2 Names

The following table maps DB2 data types to Liquid Data data types.

DB2 Name	Liquid Data Data Type
CHARACTER	string
CHARACTER (for bit data)	string
DATALINK	string

DB2 Name	Liquid Data Data Type
LONG VARCHAR	string
LONG VARCHAR (for bit data)	string
VARCHAR (for bit data)	string

Sybase Names

The following table maps Sybase data types to Liquid Data data types.

Sybase Name	Liquid Data Data Type
SYSNAME	string
TEXT	string

CHAPTER

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
xsext:anyValue **xsext:anyType** **xsext: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: xsext:anyValue xsext:anyType xsext: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
xsext:anyValue xsext:anyType xsext: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

T

U

W

X