BEA Liquid Data for WebLogic™

Building Queries and Data Views

Version 8.1
Document Date: December 2003
Revised: December 2003

Contents

About This Document

1. Introduction

10.Accessing SQL Calls: Stored Procedures and SQL Queries

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, "Introduction," introduces key concepts such as XQuery, ad hoc queries, and Builder-generated queries.
- Chapter 2, "Data View Builder GUI Reference," explains how to start the Data View Builder and provides a graphical user interface (GUI) tour and reference.
- Chapter 5, "Building Queries," explains how to design a query using the Data View Builder to define 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, "Schemas and Namespaces in Liquid Data," describes how to use source and target schemas to generate queries in Data View Builder.

- Chapter 7, "Analyzing and 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 6, "Running, Saving, and Deploying Queries," describes how you run the query and view the results.
- Chapter 8, "Using Data Views," has information and examples about saving and reusing data views as new query resources.
- Appendix A, "Type Casting Reference,"describes how the Data View Builder implements data type transformation for automatic type casting.

What You Need to Know

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

e-docs Web Site

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

How to Print the Document

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

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

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

Related Information

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

Contact Us!

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

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

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

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

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

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

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.

Introduction

This section introduces the Data View Builder, a programming tool you can use to plan, design, build, and test queries using BEA Liquid Data for WebLogic.

The following topics are covered:

- Data View Builder Overview
- Key Concepts of Query Building
- Next Steps

The Liquid Data Data View Builder is a major component in an end-to-end query development and testing system. The Data View Builder is ideal for trying out and refining queries in relationship to *target XML schemas*, or more usually referred to as *target schemas*, that provide additional, needed instructions as to how the results of your query should be represented.

For the complete overview of the Liquid Data system see the *Concepts Guide*.

Data View Builder Overview

The Data View Builder is a GUI-based tool for designing and generating *target schemas* and XQueries (in W3C XQuery syntax). You can then run the queries against multiple diverse data sources to retrieve and consolidate data. The Data View Builder provides a graphical, drag-and-drop *mapping* approach to query design and construction. You can use it to free yourself from having to manage the intricacies of query languages such as SQL while allowing you to give full attention to:

- Information design

- The conceptual synthesis of information coming from multiple sources
- The content and shape of the information you want in the query result.
- Using Liquid Data, you can directly access distributed, heterogeneous data sources as integrated logical views and build up a picture of how you want that consolidated information to be structured.

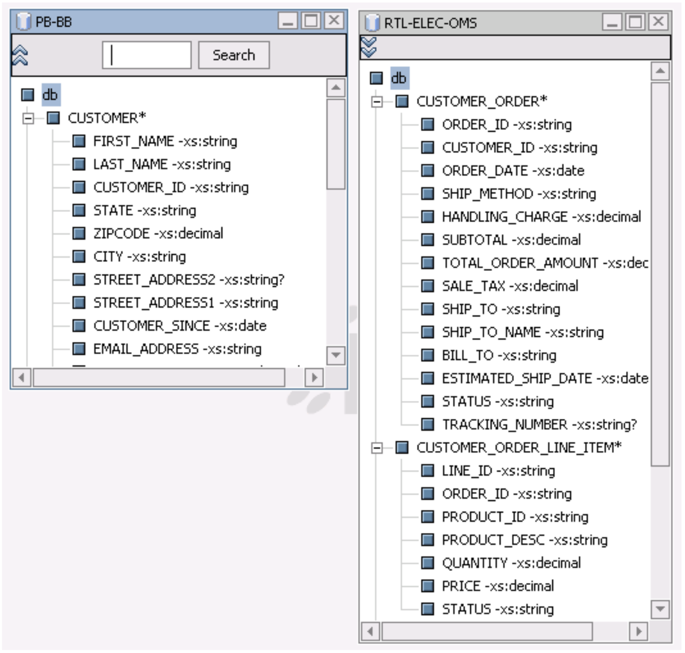
Benefits of the Data View Builder

The Data View Builder lets you create queries using an intuitive, drag-and-drop mapping strategy. The XML schema representations and mappings of source and target data are packaged and saved as a project. Using Liquid Data projects you can retrieve the full picture of the query, complete with source schemas and target mappings. Queries can also be stored as *data views* that can be treated as data sources themselves in Liquid Data and re-used.

How the Data View Builder Works

Once a data source has been configured in Liquid Data using the Liquid Data node of the WebLogic Administration Console (see the Liquid Data *Administration Guide*), it becomes available to the Data View Builder. The structure of the data stored in relational databases, Web services, application views, data views, delimited files, and XML files themselves are all represented as XML schemas.

Figure 1-1 Sample Relational and XML Source Schema



By representing disparate data sources as *XML schemas*, Data View Builder makes it easy for you to represent relationships among different types of data sources.

To build up your query you simply drag and drop elements and attributes among XML schema representations of data sources to create joins, unions, and so on. The default source condition is a join. It uses a built-in equality [`eq`] function; more complex XQuery functions are also available.

You can also map elements from source schemas to a target schema to shape the structure of the query. Target schemas can be created "on the fly" from existing data source representations or created externally. Optimization hints can be added to queries to improve query performance.

Queries can be automatically generated at any point in the development process. When you run the query the underlying data sources are accessed and the results appear in tree-view or XML. The query can also be deployed for use in BEA WebLogic Workshop web applications, among other technologies.

Key Concepts of Query Building

The following terms and concepts are introduced here:

- Data Sources

- Source and Target Schemas
- Queries and Query Joins, Unions, Aggregates, and Functions
- Stored Queries
- Ad Hoc Queries
- Query Plans

Data Sources

Liquid Data supports multiple types of data sources in addition to RDBMS (relational database management systems) including:

- XML Files
- Web Services
- Application Views
- Data Views
- SQL Calls
- Delimited Files

See Chapter 3, "Data Sources," for a complete survey.

Source and Target Schemas

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

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

Liquid Data queries require identified data sources. Each data source requires an associated schema. For relational databases, the schema is automatically created based on the metadata available through the database JDBC driver. For XML files, Views, Complex Parameter Types, Stored Procedures, or Web Services, you specify the schema when you define the data source to Liquid Data.

For more information on source and target schemas in Liquid Data, see "Source and Target Schemas" on page 4-2.

Queries and 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 problem.

A number of tools and techniques are available to develop or expand business logic around the data base schemas. These tools and techniques include:

- Joins
- Unions
- XQuery Functions
- Query Parameters
- Constants

Joins

A query with a *join* operation combines information in two data source schemas when there is a match on a common field.

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

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 type query to consolidate elements in source schemas into a tailored view of the data. For example, you could construct a query that reports all customer orders from multiple sources into a single result. This is a very typical use of Liquid Data target schemas.

XQuery Functions

Liquid Data provides a large number of XQuery functions. These built-in functions can by used by any Liquid Data query.

Aggregates

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

Other Built-in Functions

All conditions are set through functions. The default function for a simple equality function [`eq`[. If you drag and drop one data source element onto another 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 Data View Builder-generated XQuery.

For information on W3C standard functions supported by Liquid Data, see "Functions Reference" in the *XQuery Reference Guide*.

You can also create and use custom functions. See "Configuring Access to Custom Functions" in the Liquid Data *Administration Guide*.

Query Parameters

There are two types of query parameters:

- A function may have parameters that become elements in a source schema.
- You can define generic placeholders for a variable value and specify that value at query run time. For example, a query parameter could be defined as *lastname*, which is a placeholder for a real last name that you identify when the query runs.

Constants

Although not as flexible as query parameters, numeric and string constants are easily created and used.

Stored Queries

A *stored query* is a query that has been saved to the `stored_queries` folder Liquid Data repository. Queries must be saved with a `.xq` extension to be recognized as stored queries in Liquid Data.

The *query result* for a stored query can be cached.

Caching of query results for stored queries is configurable through the Liquid Data node of the WebLogic 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, but rather is passed to the Liquid Data server at run time.

Query Plans

A *query plan* is a compiled query. Before a query is run, Liquid Data compiles the XQuery into an optimized query plan. Optionally, you can compile the query plan without running the query.

At runtime, Liquid Data executes the query plan against physical data sources and returns the query results.

How This Book is Organized

Building Queries and Data Views can be thought of as having several sections:

Focus	Chapters
Introduction	• Chapter 1, "Introduction" • Chapter 2, "Data View Builder GUI Reference"
Query Components	• Chapter 3, "Data Sources" • Chapter 4, "Schemas and Namespaces in Liquid Data"

Focus	Chapters
Query Design	• Chapter 5, "Building Queries" • Chapter 6, "Running, Saving, and Deploying Queries" • Chapter 5, "Building Queries" • Chapter 7, "Analyzing and Optimizing Queries"
Special Data Sources	• Chapter 8, "Using Data Views" • Chapter 9, "Using Complex Parameter Types in Queries" • Chapter 10, "Accessing SQL Calls: Stored Procedures and SQL Queries"

This book does assume some familiarity with the purpose and operation of the Liquid Data graphical query development tool called Data View Builder. You can familiarize yourself with this tool in several ways:

- *Getting Started* provides a details, stepped example showing how to set up both the Data View Builder and Liquid Data node of the WebLogic Administration Console.
- A Getting Started Demo can be found on the Liquid Data Demo page. This animated demo runs 10-15 minutes and shows the development of a sample Liquid Data query and web application that uses that query.

Next Steps

- If you have not already done so, try 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 an Order Query example.
- To learn how to start the Data View Builder and understand the GUI tools and views, see Chapter 2, "Data View Builder GUI Reference."
- To learn more about planning and designing queries and using the Data View Builder to build them, see Chapter 5, "Building Queries."
- For information on query optimization and performance, see Chapter 7, "Analyzing and Optimizing Queries."
- For information on defining stored procedures to Liquid Data, see Chapter 10, "Accessing SQL Calls: Stored Procedures and SQL Queries."

- For details on creating queries by using custom functions, see "Using Custom Functions" in the *Application Developer's Guide*.
- For numerous examples of building different types of queries using advanced functions and tools, see "*Liquid Data by Example*."

Data View Builder GUI Reference

This chapter provides a graphical interface reference for the Data View Builder, including menus and other visual components used in accessing data sources, target XML schemas, query parameters, constants, XQuery functions and other features that are used in designing, optimizing, testing, and deploying a Builder-generated query and saving any resulting projects.

The following topics are covered:

Starting the Data View Builder

Note: The Liquid Data *Getting Started* guide contains detailed instructions on starting and using the Data View Builder to create a sample query and then use that query in a WebLogic Workshop application.

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

1. Start the Data View Builder.

   – On a Windows platform, choose the menu item:
     Star t —> Programs —> BEA WebLogic Platform 8.1 —> BEA Liquid Data for WebLogic 8.1 —> 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 where your data sources are located. Initially you may want to connect to the Liquid Data Samples server. See the Liquid Data *Getting Started* guide for details.

   a. The username and password for the Data View Builder is specified in the WebLogic Server (WLS) Compatibility Security via the Liquid Data node of the WebLogic Administration Console for the 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.

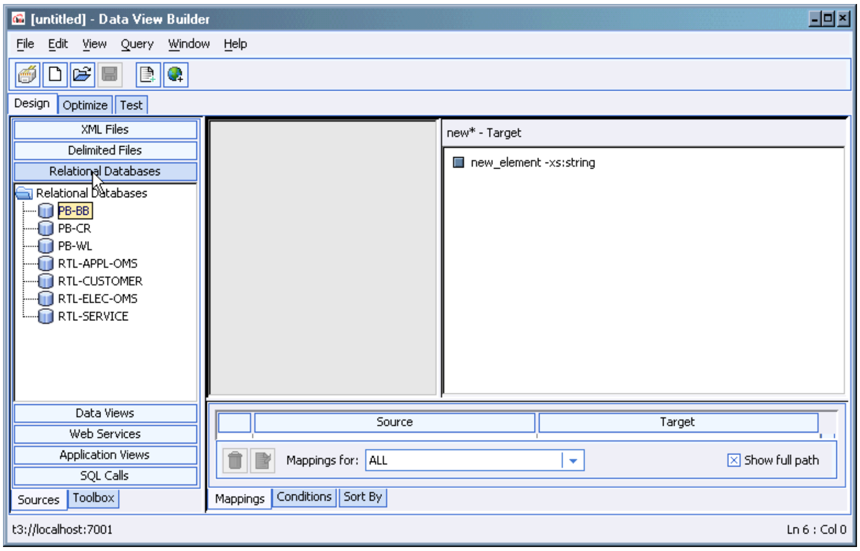
   b. Enter the URL for the Liquid Data server. For example, to connect to a server running on your own machine as a local host you enter the following:

      ```
      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 the query, respectively. Some queries benefit from optimization, available 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 conditions and source-to-target mappings. The following sections describe the features available on the Design tab.

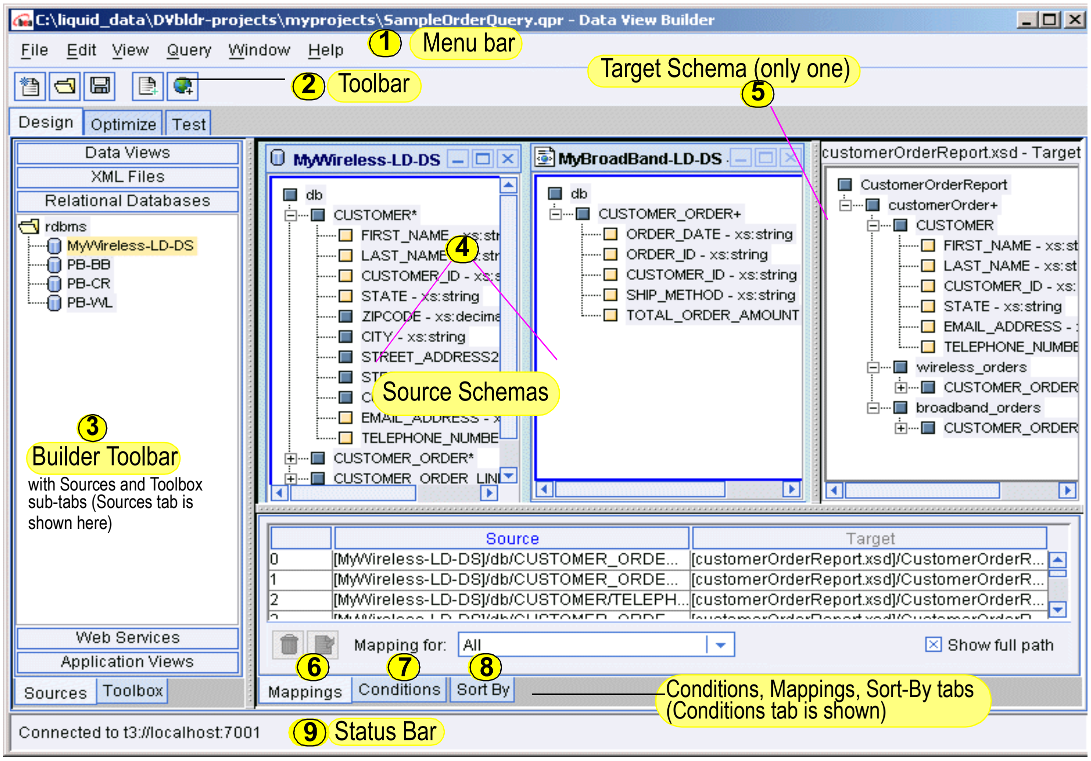
- Overview Picture of Design Tab Components
- 1. Menu Bar for the Design Tab

- 2. Toolbar for the Design Tab
- 3. Builder Toolbar
- 4. Source Schemas
- 5. Target Schema
- 6. Conditions Tab
- 7. Mappings Tab
- 8. Sort By Tab
- 9. Status Bar

Overview Picture of Design Tab Components

The following figure and accompanying numerically-coded sections describe the components on the Design tab.

Figure 2-2 Design Tab



	Source	Target
0	[MyWireless-LD-DS]/db/CUSTOMER_ORDE...	[customerOrderReport.xsd]/CustomerOrderR...
1	[MyWireless-LD-DS]/db/CUSTOMER_ORDE...	[customerOrderReport.xsd]/CustomerOrderR...
2	[MyWireless-LD-DS]/db/CUSTOMER/TELEPH...	[customerOrderReport.xsd]/CustomerOrderR...

Note: Menus, horizontal shortcut toolbar and status bar are also covered in detail in this section. Although most menu options and shortcuts are available in all modes, others are mode specific and described in the appropriate section.

1. Menu Bar for the Design Tab

The menus provide File, Schema, View, and Window menus as detailed inTable 2-3.

Table 2-3 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, see "Working With Liquid Data Projects" on page 2-29.) • **Connect...** Resets your connection to a **Liquid Data Server.** Choosing this command closes any open projects or schemas. • **New Project.** Creates a new project. If you choose this option when you have an unsaved project in the workspace you are given the option to first save your current work to a project. If you choose not to save, any previously generated query and associated conditions and schema mappings will be lost. • **Open Project.** Opens an existing project that you specify through the file browser. • **Close Project**. Closes the current project. If you have not saved your work, you are given an opportunity to do so. • **Open Query.** Opens an existing saved query. When you open a saved query, you only see the Test Tab; the Design and Optimize tabs are not available. You can then edit, run, and save the query. • **Save Project**. Saves the current project. Data View Builder projects are saved with a `.qpr` filename extension and may be saved to any directory. • **Save Project As.** Saves the current project under a different file name. • **Add Selected Schema**. Adds/opens the source schema that is selected in the Builder Toolbar to the current project. This is the same as dragging a data source schema into the work area. • **Set Target Schema.** Brings up a file browser from which you can select a schema file from your local system, a network drive, or a 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 from the list of data sources (Builder Toolbar) to be set as the target schema in the current project. (A right-click short-cut "Set as target schema" is also available when you click on a data source.) • **Save Target Schema.** Saves the current target schema to the Liquid Data repository or to a folder location and filename you choose. If you choose Repository when saving a target schema, a relative path to the file is saved in the project file. This makes the target schema available to other Liquid Data users and servers. If you save a target schema to a local file, the fully qualified path is saved in the project file, making the schema accessible only on the local machine. • **Save Query.** For a description of this option, see Table 2-18, "Menu Bar for the Test Tab," on page 2-25. • **Exit.** Closes the Data View Builder application.

Table 2-3 Menu Bar for the Design Tab (Continued)

Menu	Description of Menu Options
Edit Menu	Provides standard edit features. Availability of these commands is based on previous actions you may have takes and what item is selected. For example, if you highlight an element in a target schema and select Delete, the highlighted element will be deleted. • **Cut** • **Copy** • **Paste** • **Delete** • **Select All**

Table 2-3 Menu Bar for the Design Tab (Continued)

Menu	Description of Menu Options
View Menu	As an alternative to using the Design mode tabs the View menu provides a means for you to **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 attach notes to particular queries. • **Data Types.** Toggle to show/hide data types for all source and target elements 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. • **Lines.** Includes a submenu of options to show: – all lines between data sources and your target schema – no lines – only lines of a selected data source If the Show lines option is enabled and you highlight a schema element that is mapped to a target schema (or highlight a target schema element), a dashed yellow line will show the correction and the names of the elements will be highlighted. Lines are drawn only for mappings where the source and target elements are both visible. A solid gray lines represents a mapping from a source element whose containing window is in the background on the desktop.

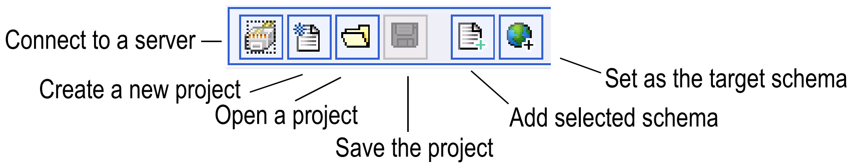
Table 2-3 Menu Bar for the Design Tab (Continued)

Menu	Description of Menu Options
Query Menu	• **Compile Query**. Compiles the current query without executing it. This is particularly useful when trying to validate large queries. If you do not see an error message after running Compile Query, it means that your query has compiled successfully and can be run at your convenience.The Compile Query option can also be accessed through a button on the menu bar. After you compile a query it will be cached. The query will be recompiled if it does not exactly match the text in the cache. • **Run Query.** Runs the current query. (See "Table , "6. Run Query," on page 2-28) • **Stop Query Execution.** Stops a running query. (See Table , "Stopping a Running Query," on page 2-29.) • **Allow Existential Condition Generation.** Toggle to turn existential condition generation on or off. A checkmark next to this option indicates that existential condition generation is *on*. For more information see "Using Existential Condition Checking in Queries" on page 5-53. • **Automatic Type Casting.** Toggle to turn automatic type casting on or off. A checkmark next to this option indicates that automatic type casting is *on*. For more information see "Using Automatic Type Casting" on page 5-59. • **Automatic Treat-as.** Toggle to turn automatic treat-as on or off. A checkmark next to this option indicates that automatic treat-as is *on*. When automatic treat-as is on, `treat` functions are automatically placed in the query whenever there is a type mismatch. For details on the treat functions, see "Treat Functions" in the "Functions Reference" section of the *XQuery Reference Guide*. • **Condition Targets —> Advanced View.** Toggle to turn Advanced View for manual scoping on/off. For more information on using scoping in Advanced View see "Managing Target Schema Properties" on page 5-26. • **Target Namespace.** Opens a dialog box where you can enter a prefix and URI for the target schema. For a description of the other options in the Query menu (Compile Query, Run Query, or Stop Query Execution) that are relevant only for running/testing a query, see Table 2-3, "Menu Bar for the Design Tab," on page 2-7.
Window Menu	The Window menu provides various options for window management such as next, previous, close, and close all. Source schema windows that you open are listed in the Window menu.
Help Menu	Provides links to Data View Builder online documentation.

2. Toolbar for the Design Tab

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

Figure 2-4 Toolbar



3. Builder Toolbar

The Builder Toolbar includes two subtabs:

- **Sources.** 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.** Provides access to functions, constants, query parameters, and other components used in query design.

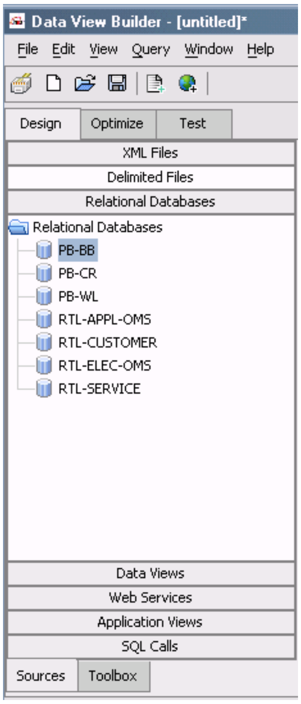
Sources Tab

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 Liquid Data Server to which you are connecting. Potentially available data sources include:

- XML Files
- Delimited Files
- Web Services
- Application Views
- Data Views
- SQL Calls
- Delimited Files

Note: For a detailed introduction to Liquid Data data sources, see Chapter 3, "Data Sources." See also Chapter 4, "Schemas and Namespaces in Liquid Data," for details on using data source schemas in constructing queries.

Figure 2-5 Builder Toolbar: Sources Tab



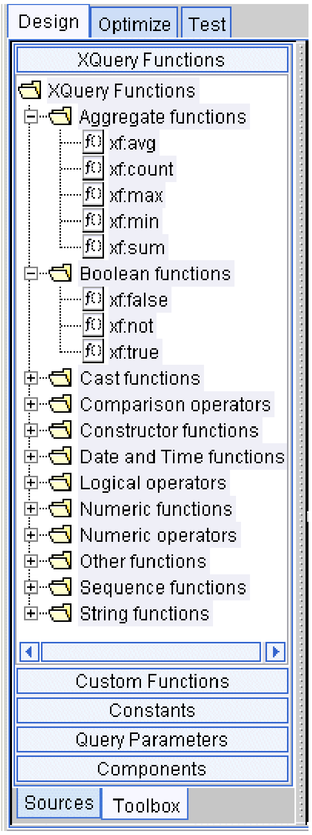
Toolbox Tab

The Toolbox tab on the Builder Toolbar provides the following tools you can use in constructing and tailoring your query:

- XQuery Functions (information on the Function Editor is included here)
- Custom Functions
- Constants
- Query Parameters
- Complex Parameter Types

- Components

Figure 2-6 Builder Toolbar: Toolbox Tab



XQuery Functions

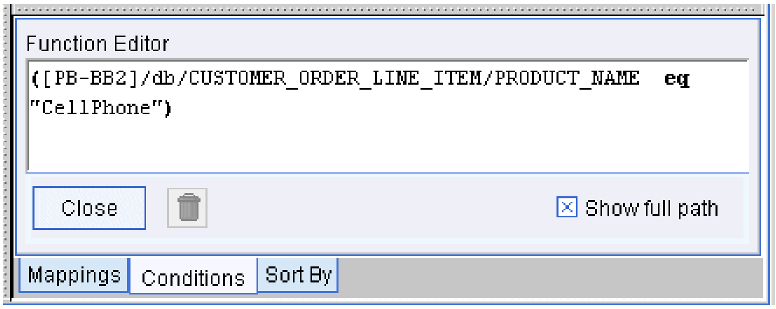
XQuery Functions are built-in code modules that return a value when they run. The XQuery Functions panel provides a library of standard W3C functions compliant with the W3C *XQuery 1.0 and XPath 2.0 Functions and Operators* specification. (See Figure 2-6, "Builder Toolbar: Toolbox Tab," on page 2-13 for an example of the Functions panel.)

In the Data View Builder, Functions are displayed in the Builder Toolbar on the Toolbox tab XQuery Functions panel by category names such as Aggregate Functions, Boolean Functions, Cast Functions, and so on. To view all the functions in a category or group, expand the group element. For details on using XQuery functions see "Using XQuery Functions" on page 5-14.

Function Editor

The functions editor provides the ability to create functions using drag-and-drop and to view existing functions in your project.

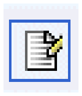
Figure 2-7 Function Editor



```
([PB-BB2]/db/CUSTOMER_ORDER_LINE_ITEM/PRODUCT_NAME  eq
"CellPhone")
```

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 2-8 Button to Access the Functions Editor



For details see "Using the Function Editor" on page 5-5.

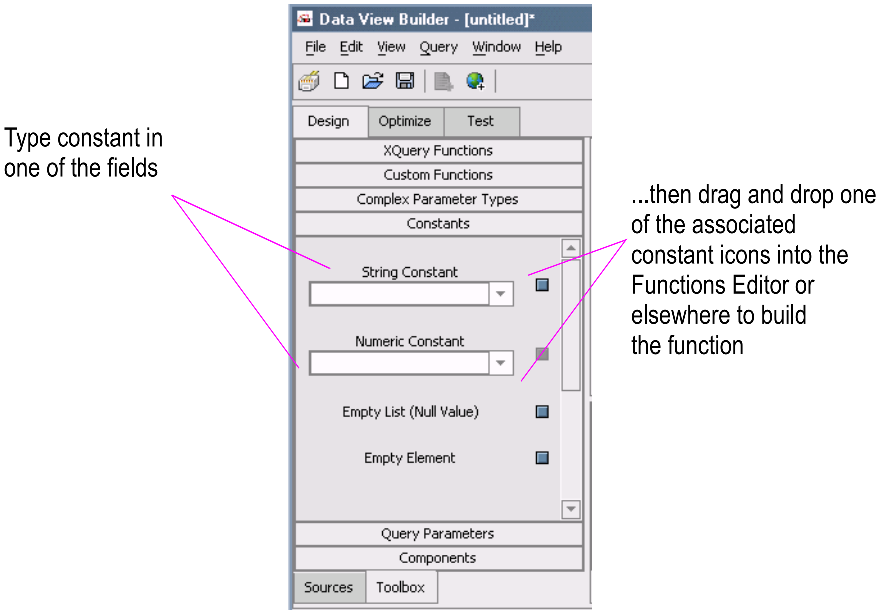
Custom Functions

If you have custom functions configured through the Liquid Data node of the WebLogic Administration Console, these will appear in the Custom Functions section of the Data View Builder toolbar. For details on creating queries by using custom functions, see "Using Custom Functions" in the *Application Developer's Guide*.

Constants

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

Figure 2-9 Setting Constants Dialog Box


Type constant in one of the fields
...then drag and drop one of the associated constant icons into the Functions Editor or elsewhere to build the function
Data View Builder - [untitled]*
File Edit View Query Window Help
Design Optimize Test
XQuery Functions
Custom Functions
Complex Parameter Types
Constants
String Constant
Numeric Constant
Empty List (Null Value)
Empty Element
Query Parameters
Components
Sources Toolbox

For details see "Creating and Using Constants" on page 5-8.

Query Parameters

You can create named query parameters and associate them with a data type. For details see "Creating and Using Query Parameters" on page 5-10.

Complex Parameter Types

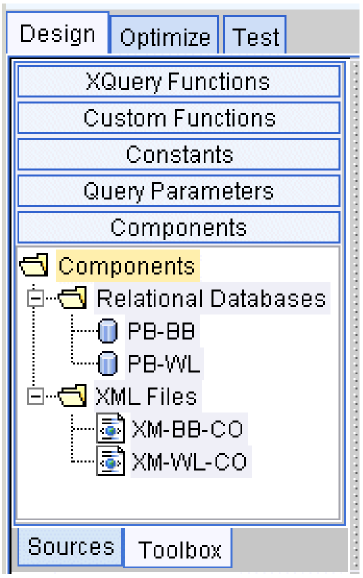
If you have complex parameter types configured through the WebLogic Administration Console, these will appear in the Data View Builder on the Toolbar Functions tree under Complex Parameter Types. For more information, see Chapter 9, "Using Complex Parameter Types in Queries" and Configuring Access to Complex Parameter Types in the *Administration Guide*.

Components

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

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

Figure 2-10 Builder Toolbar: Toolbox Tab: Components



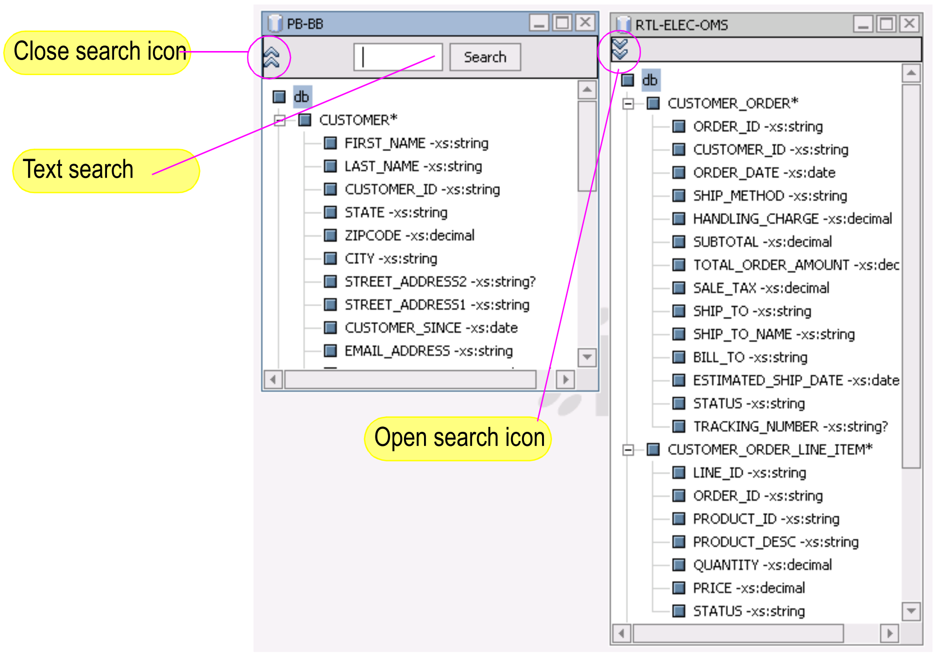
Any component that appears in this panel can be minimized on the Liquid Data desktop by double-clicking the appropriate node. Click again and the component reappears on the desktop.

You can hide the Builder Toolbar using a checkbox located under View —-> Toolbars.

4. Source Schemas

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

Figure 2-11 Sample Source Schemas



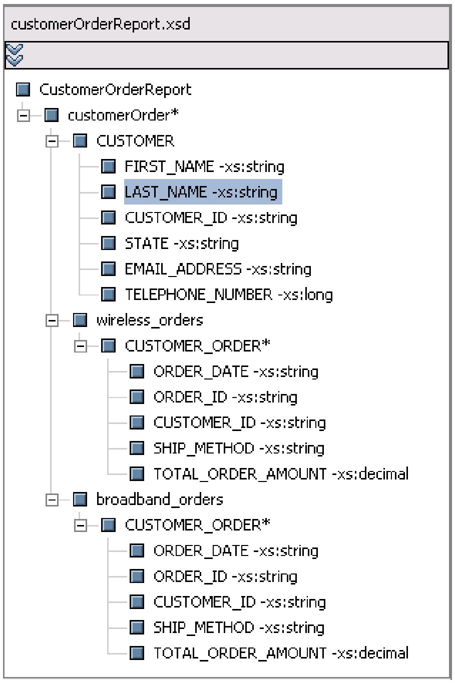
```
PB-BB
db
  CUSTOMER*
    FIRST_NAME -xs:string
    LAST_NAME -xs:string
    CUSTOMER_ID -xs:string
    STATE -xs:string
    ZIPCODE -xs:decimal
    CITY -xs:string
    STREET_ADDRESS2 -xs:string?
    STREET_ADDRESS1 -xs:string
    CUSTOMER_SINCE -xs:date
    EMAIL_ADDRESS -xs:string

RTL-ELEC-OMS
db
  CUSTOMER_ORDER*
    ORDER_ID -xs:string
    CUSTOMER_ID -xs:string
    ORDER_DATE -xs:date
    SHIP_METHOD -xs:string
    HANDLING_CHARGE -xs:decimal
    SUBTOTAL -xs:decimal
    TOTAL_ORDER_AMOUNT -xs:dec
    SALE_TAX -xs:decimal
    SHIP_TO -xs:string
    SHIP_TO_NAME -xs:string
    BILL_TO -xs:string
    ESTIMATED_SHIP_DATE -xs:date
    STATUS -xs:string
    TRACKING_NUMBER -xs:string?
  CUSTOMER_ORDER_LINE_ITEM*
    LINE_ID -xs:string
    ORDER_ID -xs:string
    PRODUCT_ID -xs:string
    PRODUCT_DESC -xs:string
    QUANTITY -xs:decimal
    PRICE -xs:decimal
    STATUS -xs:string
```

5. Target Schema

The Target Schema window shows the XML schema representation for the structure of the *target* data (query result). For additional information see "Source and Target Schemas" on page 4-2.

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 2-12 Target Schema



```
CustomerOrderReport
customerOrder*
  CUSTOMER
    FIRST_NAME -xs:string
    LAST_NAME -xs:string
    CUSTOMER_ID -xs:string
    STATE -xs:string
    EMAIL_ADDRESS -xs:string
    TELEPHONE_NUMBER -xs:long
  wireless_orders
    CUSTOMER_ORDER*
      ORDER_DATE -xs:string
      ORDER_ID -xs:string
      CUSTOMER_ID -xs:string
      SHIP_METHOD -xs:string
      TOTAL_ORDER_AMOUNT -xs:decimal
  broadband_orders
    CUSTOMER_ORDER*
      ORDER_DATE -xs:string
      ORDER_ID -xs:string
      CUSTOMER_ID -xs:string
      SHIP_METHOD -xs:string
      TOTAL_ORDER_AMOUNT -xs:decimal
```

The Target Schema can be hidden using a checkbox in View —> Panels.

6. Conditions Tab

The Conditions tab shows:

- In Basic mode, conditions defined for the source data (see "Conditions Section" on page 2-20)
- In Advanced mode, conditions defined that potentially define Scope for the target data or query result (see "Advanced View (Setting Condition Scope Manually)" on page 5-41)

The Conditions area lists underlying query conditions that can inspect or change. Whenever you do a drag-and-drop operation that changes query conditions, the Conditions tab is automatically updated and displayed.

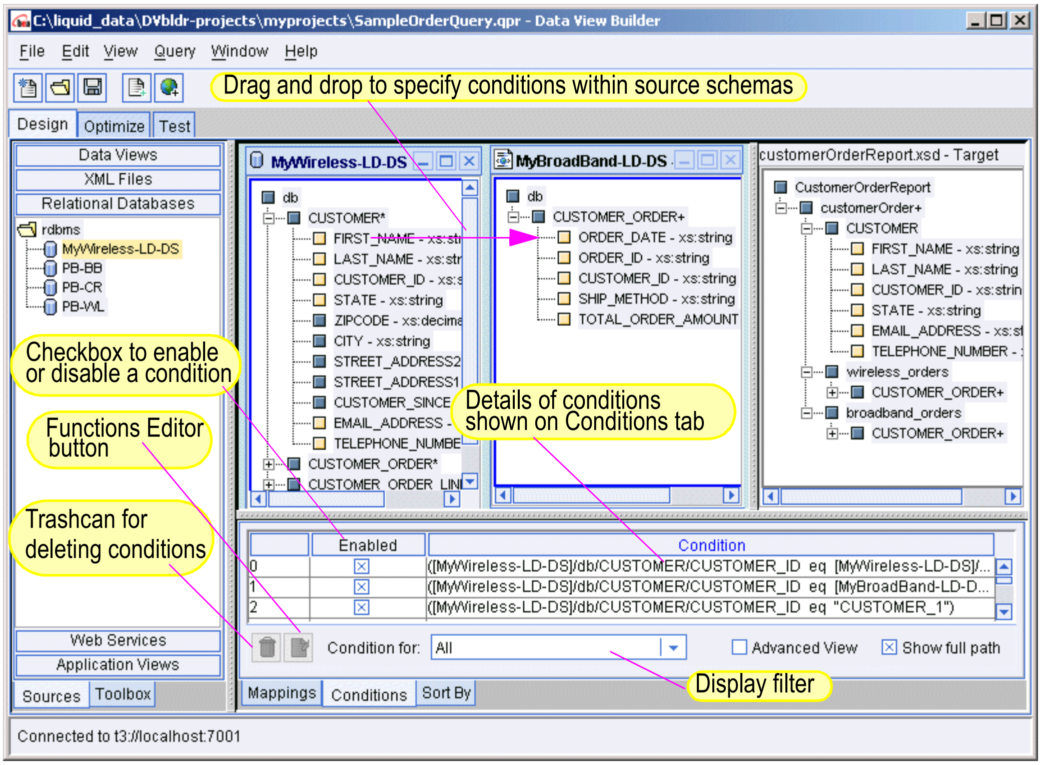
The Conditions tab includes the following features:

- **Conditions Section.** List of conditions (filters) that can optionally be applied when the query is generated.

- **Function editor.** To edit an existing condition, select it and click on the Function editor icon, to the right of the Trashcan. You can also access the Functions editor by dragging and dropping a function from the Functions panel on the Toolbox panel into an empty Conditions row. See also "XQuery Functions" on page 2-13 and ""Function Editor" on page 2-14.
- **Trashcan.** To remove a condition, select the row that contains the condition you want to remove and click the Trashcan.
- **Enabled checkbox.** Each query condition can be enabled (or disabled) using an Enabled checkbox. By default all conditions are enabled. To disable a condition click the checkbox.

  Disabled conditions are not part of the query.

Figure 2-13 Conditions Tab on the Design tab



	Enabled	Condition
0	☒	([MyWireless-LD-DS]/db/CUSTOMER/CUSTOMER_ID eq [MyWireless-LD-DS]/...
1	☒	([MyWireless-LD-DS]/db/CUSTOMER/CUSTOMER_ID eq [MyBroadBand-LD-D...
2	☒	([MyWireless-LD-DS]/db/CUSTOMER/CUSTOMER_ID eq "CUSTOMER_1")

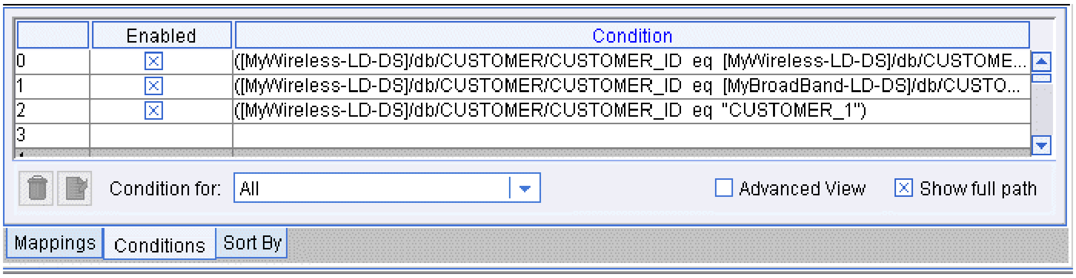
Condition for: All ☐ Advanced View ☒ Show full path

Connected to t3://localhost:7001

Conditions Section

The Conditions section displays conditions for source data. As you build up the query by creating drag-and-drop source-to-source element relationships among data source schemas, the implied condition statements are recorded and reflected as *equality joins* (eq) under the Conditions.

Figure 2-14 Conditions Tab in Basic View



	Enabled	Condition
0	☒	([MyWireless-LD-DS]/db/CUSTOMER/CUSTOMER_ID eq [MyWireless-LD-DS]/db/CUSTOME...
1	☒	([MyWireless-LD-DS]/db/CUSTOMER/CUSTOMER_ID eq [MyBroadBand-LD-DS]/db/CUSTO...
2	☒	([MyWireless-LD-DS]/db/CUSTOMER/CUSTOMER_ID eq "CUSTOMER_1")
3		

Condition for: All ☐ Advanced View ☒ Show full path

For details on using the query Conditions section in basic and Advanced View mode see "Managing Target Schema Properties" on page 5-26.

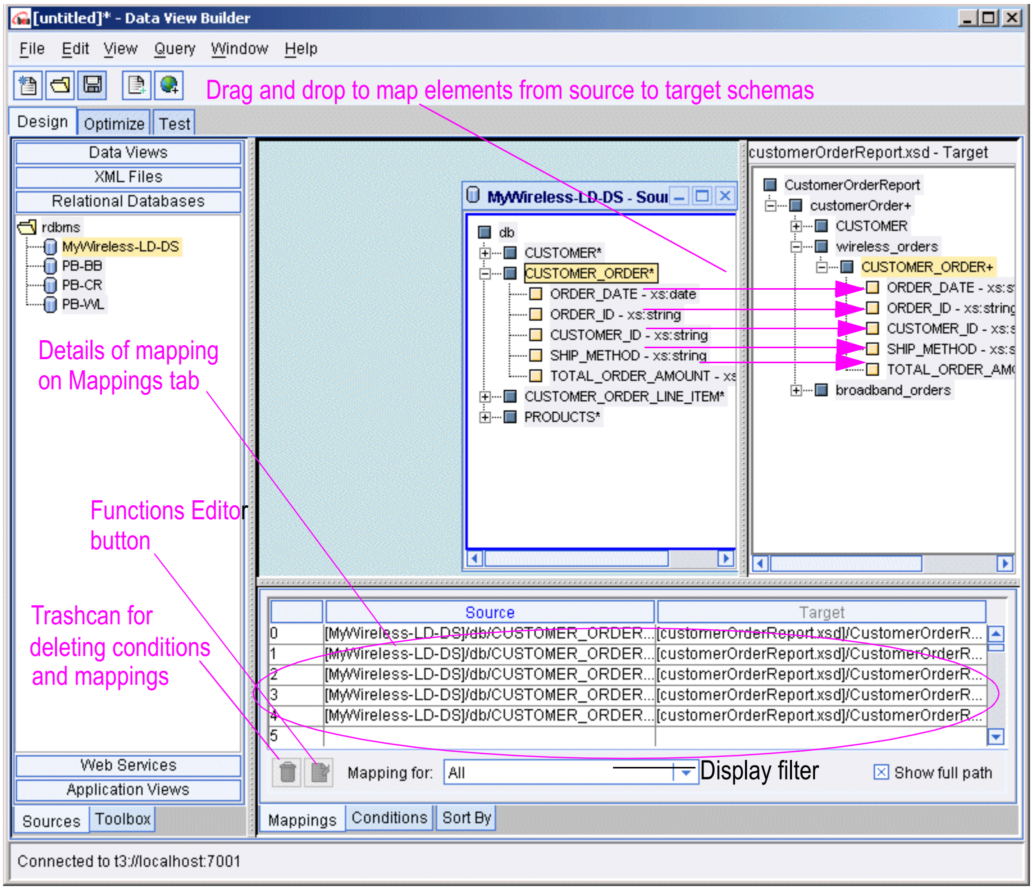
7. Mappings Tab

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.

A checkbox below the list of query conditions allows you to optionally show or hide the full path to the mapping elements.

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

Figure 2-15 Mappings Tab



	Source	Target
0	[MyWireless-LD-DS]/db/CUSTOMER_ORDER...	[customerOrderReport.xsd]/CustomerOrderR...
1	[MyWireless-LD-DS]/db/CUSTOMER_ORDER...	[customerOrderReport.xsd]/CustomerOrderR...
2	[MyWireless-LD-DS]/db/CUSTOMER_ORDER...	[customerOrderReport.xsd]/CustomerOrderR...
3	[MyWireless-LD-DS]/db/CUSTOMER_ORDER...	[customerOrderReport.xsd]/CustomerOrderR...
4	[MyWireless-LD-DS]/db/CUSTOMER_ORDER...	[customerOrderReport.xsd]/CustomerOrderR...
5		

The Conditions Tab can be hidden using a checkbox in View —> Panels.

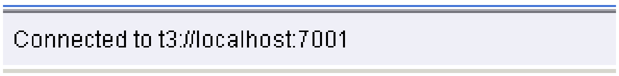
8. Sort By Tab

Target schema elements associated with complex elements with the repeatable attribute set can be sorted in ascending or descending order. In addition, the order that elements are sorted can be easily changed. See "Sorting Query Results" on page 5-51 for details.

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. The Status Bar can optionally be hidden using a checkbox in View —> Panels.

Figure 2-16 Status Bar

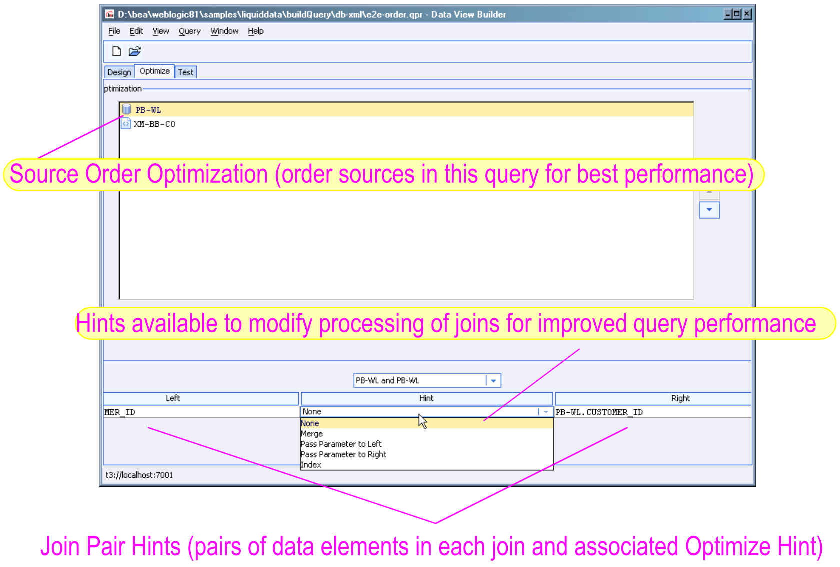


Connected to t3://localhost:7001

Optimize Tab

Use the Optimize tab to add clarifying hints to improve query performance.

Figure 2-17 Optimize Tab


D:\bea\weblogic81\samples\liquiddata\buildQuery\db-xml\e2e-order.qpr - Data View Builder
File Edit View Query Window Help
Design Optimize Test
ptimization
PB-WL
XM-BB-CO
Source Order Optimization (order sources in this query for best performance)
Hints available to modify processing of joins for improved query performance
PB-WL and PB-WL
Left
Hint
Right
MER_ID
None
PB-WL.CUSTOMER_ID
None
Merge
Pass Parameter to Left
Pass Parameter to Right
Index
t3://localhost:7001
Join Pair Hints (pairs of data elements in each join and associated Optimize Hint)

For detailed information on how to optimize a query by ordering source schemas, see Chapter 7, "Analyzing and Optimizing Queries."

Test Tab

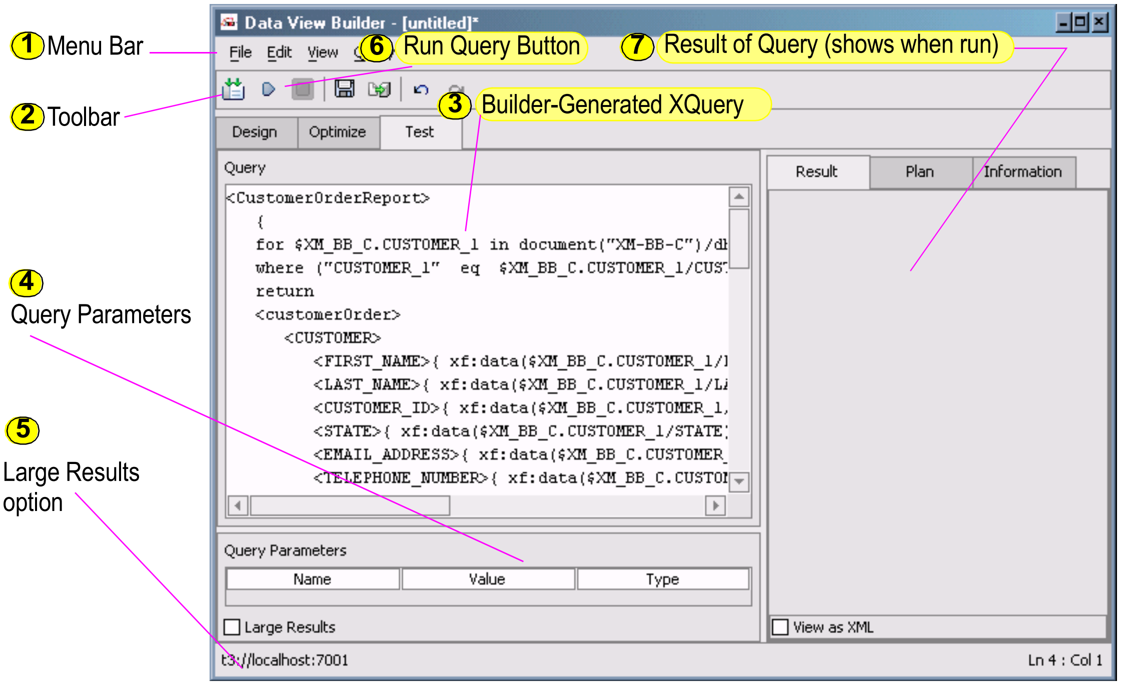
The Test tab is where you view the generated XQuery from the query elements you developed on the Design and Optimize tabs. From this view, you can provide different parameters to the query before you run it.The following sections (numerically keyed to Figure 2-1, "Data View Builder Test Mode," on page 2-24) describe the graphical features available on the Test tab:

1. Menu Bar for the Test Tab
2. Toolbar for the Test Tab
3. Builder-Generated XQuery
4. Query Parameters: Submitted at Query Runtime
5. Query Results - Large Results
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 2-1 Data View Builder Test Mode



```
<CustomerOrderReport>
   {
   for $XM_BB_C.CUSTOMER_1 in document("XM-BB-C")/db
   where ("CUSTOMER_1"  eq  $XM_BB_C.CUSTOMER_1/CUST
   return
   <customerOrder>
      <CUSTOMER>
         <FIRST_NAME>{ xf:data($XM_BB_C.CUSTOMER_1/F
         <LAST_NAME>{ xf:data($XM_BB_C.CUSTOMER_1/LA
         <CUSTOMER_ID>{ xf:data($XM_BB_C.CUSTOMER_1,
         <STATE>{ xf:data($XM_BB_C.CUSTOMER_1/STATE)
         <EMAIL_ADDRESS>{ xf:data($XM_BB_C.CUSTOMER_
         <TELEPHONE_NUMBER>{ xf:data($XM_BB_C.CUSTOM
```

1. Menu Bar for the Test Tab

Table 2-18 Menu Bar for the Test Tab

Menu	Description of Menu Options
File Menu	Many of the File menu commands available in Test mode, including: • Connect... • New Project • Open Project • Open Query • Save Query • Exit For a complete description of File menu items, including the above, see "Menu Bar for the Design Tab" on page 2-7.
Edit Menu	Provides standard edit features. • **Cut** • **Copy** • **Paste** • **Delete** • **Select All** For a complete description of File menu items, including the above, see "Menu Bar for the Design Tab" on page 2-7.
View Menu	As an alternative to using the tabs the View menu lets you **navigate** to the following UI views. For a complete description of View menu items, including the above, see "Menu Bar for the Design Tab" on page 2-7.

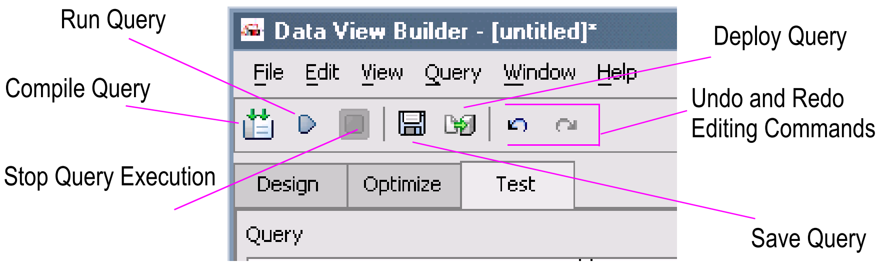
Table 2-18 Menu Bar for the Test Tab

Menu	Description of Menu Options
Query Menu	Provides the following options related to running a query: • **Compile Query.** Compiles the current query without executing it. This is particularly useful when trying to validate large queries. If you do not see an error message after running Compile Query, it means that your query has compiled successfully and is ready to be run.The Compile Query option can also be accessed through a button on the menu bar. • **Run Query.** Runs the query. (See "6. Run Query" on page 2-28) • **Stop Query Execution.** Stops a running query. (See "Stopping a Running Query" on page 2-29.) • **Deploy Query...** Stores the current query with its target schema in the `stored_query` directory. Options allow simultaneous deployment as a data view. See "Deploying Your Query" on page 6-10. **Note:** If Liquid Data security is enabled, you must log into the Data View Builder as a user who is a member of either the LDConsoleUsers or LDAdministrators group. If the user is not a member of one of these groups, attempts to deploy a query will fail with a security error. • The Query menu options for Automatic Type Casting and Condition Targets—>Advanced View are described in "1. Menu Bar for the Design Tab" on page 2-6.
Window Menu	The Window menu provides various options for window management such as next, previous, close, and close all. 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 links to online documentation for the Data View Builder.

2. Toolbar for the Test Tab

The toolbar, located directly below the menus, provides shortcuts to commonly used actions also available from the menus in addition to Undo and Redo commands.

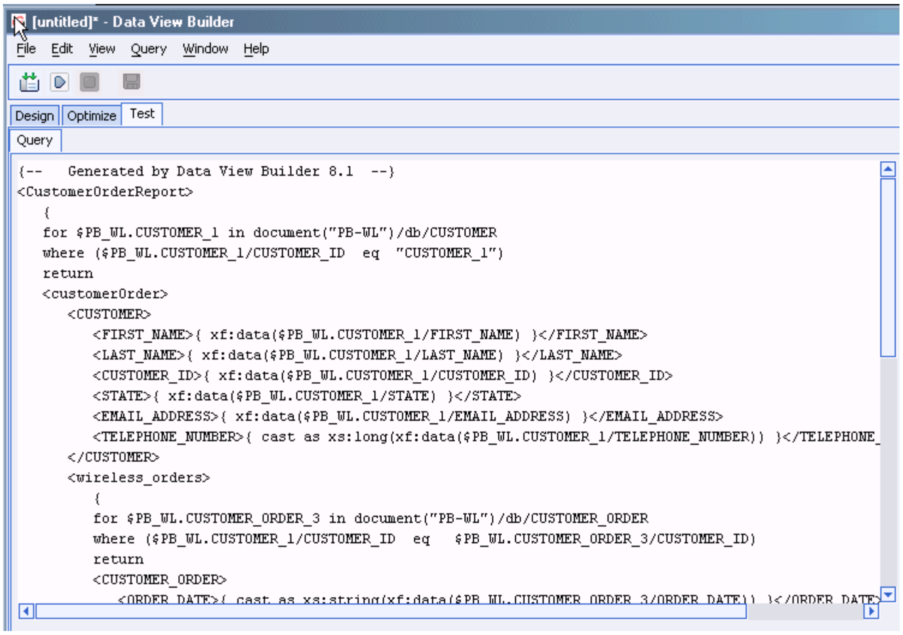
Figure 2-19 Test Tab Toolbar Icons


Run Query
Compile Query
Stop Query Execution
Data View Builder - [untitled]*
File Edit View Query Window Help
Design Optimize Test
Query
Deploy Query
Undo and Redo Editing Commands
Save Query

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 2-20 Builder-Generated XQuery Shown in Query Window

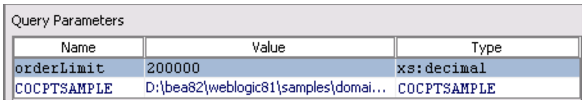

[untitled]* - Data View Builder
File Edit View Query Window Help
Design Optimize Test
Query

```
{--   Generated by Data View Builder 8.1  --}
<CustomerOrderReport>
   {
   for $PB_WL.CUSTOMER_1 in document("PB-WL")/db/CUSTOMER
   where ($PB_WL.CUSTOMER_1/CUSTOMER_ID  eq  "CUSTOMER_1")
   return
   <customerOrder>
      <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>
         <CUSTOMER_ID>{ xf:data($PB_WL.CUSTOMER_1/CUSTOMER_ID) }</CUSTOMER_ID>
         <STATE>{ xf:data($PB_WL.CUSTOMER_1/STATE) }</STATE>
         <EMAIL_ADDRESS>{ xf:data($PB_WL.CUSTOMER_1/EMAIL_ADDRESS) }</EMAIL_ADDRESS>
         <TELEPHONE_NUMBER>{ cast as xs:long(xf:data($PB_WL.CUSTOMER_1/TELEPHONE_NUMBER)) }</TELEPHONE_
      </CUSTOMER>
      <wireless_orders>
         {
         for $PB_WL.CUSTOMER_ORDER_3 in document("PB-WL")/db/CUSTOMER_ORDER
         where ($PB_WL.CUSTOMER_1/CUSTOMER_ID  eq  $PB_WL.CUSTOMER_ORDER_3/CUSTOMER_ID)
         return
         <CUSTOMER_ORDER>
            <ORDER_DATE>{ cast as xs:string(xf:data($PB_WL.CUSTOMER_ORDER_3/ORDER_DATE)) }</ORDER_DATE>
```

4. Query Parameters: Submitted at Query Runtime

You can use the Query Parameters panel (located below the Query area) to change 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 (see "Creating and Using Query Parameters" on page 5-10).

Figure 2-21 Query Parameters Settings on Test Tab



Query Parameters

Name	Value	Type
orderLimit	200000	xs:decimal
COCPTSAMPLE	D:\bea82\weblogic81\samples\domai...	COCPTSAMPLE

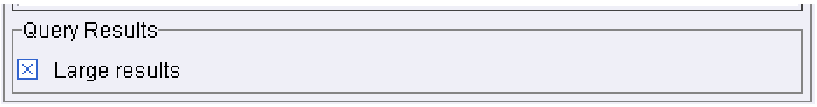
Note: Complex Parameter Type data sources are also identified in the Query Parameter Settings area. For more information, see "Using Complex Parameter Types in Queries" on page 9-1 and Configuring Access to Complex Parameter Types in the *Administration Guide*.

5. Query Results - Large Results

If you anticipate a large set of data coming back when the query is run, 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 errors.

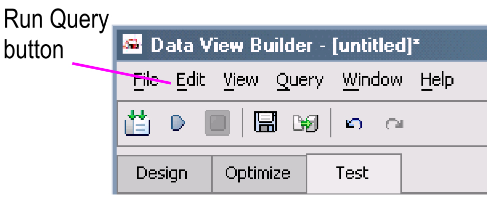
Figure 2-22 Specifying Large Results



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 2-23 Run Query Button

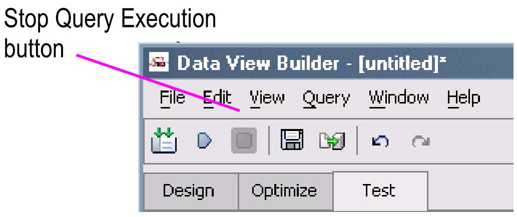


The query is run against your data sources and the result is displayed in the Results panel in XML format.

Stopping a Running Query

You can stop a running query before it has completed by clicking the Stop Query Execution button in the Toolbar. (Alternatively choose the Stop Query option from the Query menu.)

Figure 2-24 Stop Query Execution Button


Stop Query Execution button
Data View Builder - [untitled]*
File Edit View Query Window Help
Design Optimize Test

7. Result of a Query

Query results are reported in several forms. By default, results appear in structure XML.You can also view the query plan and statistics on the query once it has been run. For details see "Running, Saving, and Deploying Queries" on page 6-1.

Working With Liquid Data Projects

It is a good practice to save your project file frequently since it will allow you to immediately restore your query and the schemas and other relationships that were defined to create it.

To save a project choose File —> Save Project or File —> Save Project As 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 "1. Menu Bar for the Design Tab" on page 2-6.

Notes:

- 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. However, saving a project does not make the query in that project available as a *stored query*. For more information on stored queries see "Using the stored_queries Folder" on page 6-7.
- A copy of any mapped function is saved automatically with the project. The saved function (with associated parameters) appears in the Components panel when you reopen the project.

- If you are working with a project that was last saved under Liquid Data 8.1 SP1 or earlier, the Query menu's Allow Existential Condition Generation option will be selected. For more information see "Using Existential Condition Checking in Queries" on page 5-53.

Using Schemas Saved With Projects

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

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

Offline resources are available only to the previously associated project.

Save Target Schema to Repository

In order for your project to be portable you should save your target schema to the Liquid Data Server repository on the server where the project will be used.

Next Steps: Building and Testing Sample Queries

If you have not already done so, consider 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 an Order Query sample. (For more information about Liquid Data samples, see the Samples introduction page.) Working through the sample in *Getting Started* is a good, 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 are ready to get started on building some other basic queries see *Liquid Data by Example*. It provides examples of queries of 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.

Data Sources

The best known and most pervasive traditional data source is the relational database. An RDBMS can be thought of as a tabular data storage and retrieval resource.

The reality is that the development of global business and distributed systems has generated information in many other forms as well such as in packaged enterprise information system (EIS) applications (PeopleSoft, Siebel, etc.), and in emerging net-based technologies like Web services and XML documents.

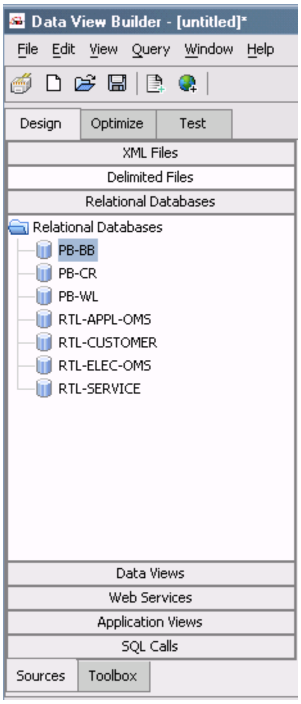
Liquid Data and Data View Builder give you the ability to query and create views into data that resides in all these types of information sources.

In this chapter the following LD-supported data sources are briefly described:

- Relational Databases
- XML Files
- Web Services
- Application Views
- Data Views
- SQL Calls
- Delimited Files

Data source descriptions available in the connected Liquid Data Server are easily accessed from the Data View Builder.

Figure 3-1 Liquid Data Sample Data Sources As Displayed in the Data View Builder



Relational Databases

All types of businesses and other organizations use a RDBMS (relational database management system) to store information. *Relational* refers to the way the database maintains information — 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*.

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

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

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

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

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

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 unit 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. Web services are becoming important resources of global business information. Liquid Data and the 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 the results as *application view* data sources in your data integration solution.

Data Views

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 if your underlying 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 learn more about Data Views see Chapter 8, “Using Data Views”.

SQL Calls

Two types of SQL queries can appear as data sources under SQL Calls:

- **Stored Procedures.** For relational databases that support stored procedures, you can create stored procedures in the RDBMS and expose them to Liquid Data. Liquid Data treats a stored procedure as a function which requires one or more inputs to produce zero or more outputs. A stored procedure allows database programmers to combine business logic with database queries, and they provide a powerful way to efficiently and securely produce information from relational databases. Also, stored procedures allow the database administrators to tune the queries run by the stored procedures, thus ensuring good performance and minimizing impact on database performance.
- **SQL Statements.** These are stored SQL queries available for execution against relational data sources. These statements can be used as virtual tables as you would any other Liquid Data data source.

Delimited Files

Spreadsheets provide a useful means of storing and manipulating information, especially information which needs to be changed quickly. You can use spreadsheet data that has been saved in comma separated value (CSV) file format in Liquid Data queries and data views. Although the separator field is conventionally referred to as a comma you can set the separator to be any ASCII character using the Liquid Data node of the WebLogic Administration Console. See "Configuring Access to Delimited Files" in the Liquid Data *Administration Guide*.

CHAPTER 4

Schemas and Namespaces in Liquid Data

This chapter describes source and target XML schemas, also called target schemas, as used in the Data View Builder to define queries. It also describes how XML namespaces can be used in your queries.

The following topics are covered:

Source and Target Schemas

XML *schemas* are used in Liquid Data to represent the hierarchical structure of various data sets and the query structure. The Data View Builder uses XML schema representations as follows:

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

For relational databases accessed through JDBC drivers, a schema is automatically generated based on available metadata.

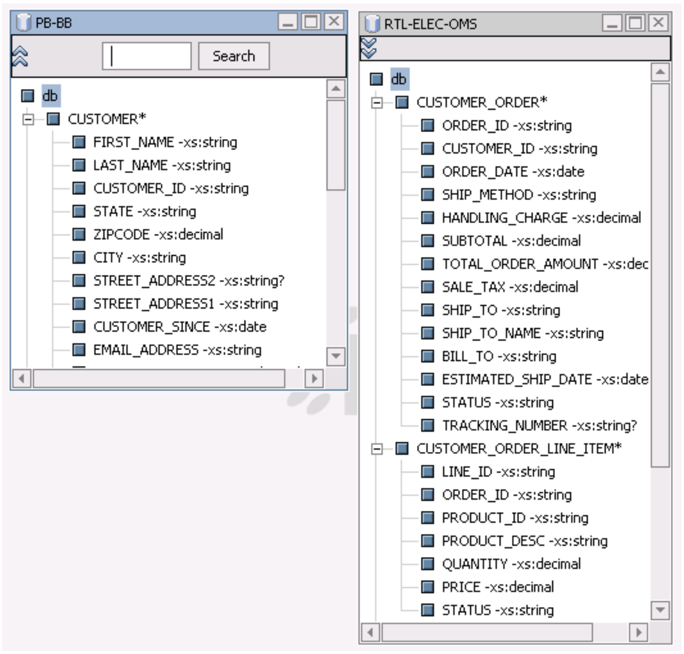
For XML files, views, complex parameter types (CPTs), stored procedures, delimited files, or web services, you first develop and then specify the schema using the WebLogic Administration Console.

Note: For the versions of the XQuery and XML specifications implemented in Liquid Data see "Supported XQuery and XML Schema Versions In Liquid Data" in the *XQuery Reference Guide*.

Source Schemas

The Data View Builder provides graphical representation of source schemas in a tree structure format. The visual representations can be expanded and collapsed for convenience and readability.

Figure 4-1 Sample source schemas



If you are building a query that depends upon more than one data source, you will use multiple source schemas (one for each data source).

Searching Text in Schemas and Other Work Area Elements

You can apply a keyword search to any source or target schema, as well as to functions. Simply click the Open search icon at the top of the pane and a search field will appear. Enter any valid search string (case does not matter) and if it exists in the pane the string will be highlighted.

Wildcard symbols (? or *) are not allowed. However, any word or partial word will be found if it appears in the pane. For example, if you search on the string `TAT` the element `STATE` will be found it if exists in the pane.

Text search is circular beginning at the currently highlighted element. In other words, if the search will be satisfied by an element above the currently highlighted line it will eventually be found if you keep clicking the Search button.

Using Source Schemas Multiple Times in Constructing Queries

In the Data View Builder you can use source schemas as many times as needed, simply by dragging an additional copy of the data source scheme into the work area.

A source is said to be *replicated* if the source schema appears multiple times in a query. In XQuery, a source is replicated if *document_name* appears multiple times in the XQuery, usually appearing in two different `for` clauses. Similarly, in SQL a source is said to be *replicated* if the source (table) appears twice in a `FROM` clause (or in two different `FROM` clauses).

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

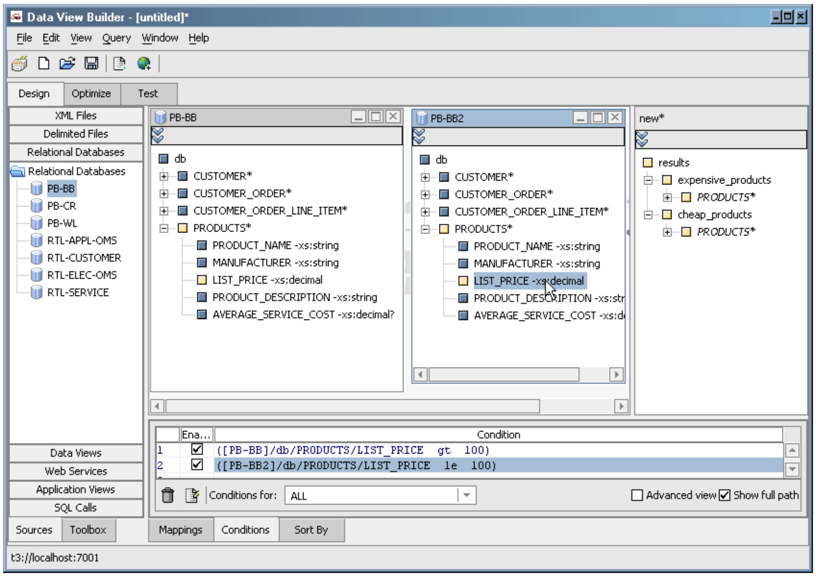
When More Than a Single Copy of a Source Schema is Needed

Sometimes it is helpful to use more than one copy of a data source schema to improve query performance. In other cases, however, having more than one copy of a data source schema is necessary.

Take, for example, a very simple problem: you want to build a target schema that lists product list prices over a certain amount and under a certain amount. XQuery functions exist for the greater-than [`gt`] and less-than-or-equal-to [`le`] test conditions. Using Advanced view you could disable `where` clause conditions to make the query valid (see "Sorting Query Results" on page 5-51).

But a clearer and cleaner approach would be to use two source instances that each reference the same data source. From the first instance of the source schema, PB-BB, `PRODUCTS` would be projected under Expensive products. From the second instance, PB-BB2, `PRODUCTS` would be projected under Cheap products (Figure 5-34). In both cases, Copy and Paste and Map are used. (See "Mapping to Target Schemas" on page 5-18 for more information on mapping of complex elements to target schemas.)

Figure 4-2 Project Illustrating Use of Two Copies of a Data Source Schema



	Ena...	Condition
1	☑	([PB-BB]/db/PRODUCTS/LIST_PRICE gt 100)
2	☑	([PB-BB2]/db/PRODUCTS/LIST_PRICE le 100)

Then it is a simple matter of creating the [ge]/[lt] conditions as described in "To resolve this problem click Advanced view in the Conditions section. You will notice that instead of the two conditions you created, four are listed. This is because Advanced view shows you the actual where clause conditions used in the query, based on application of the Data View Builder best-guess autoscope rules." on page 5-47.

The XQuery generated by this project (Listing 4-1) illustrates this approach.

Listing 4-1 XQuery returning product list prices in two groups

```
<results>
 <expensive_products>
  {
  for $PB_BB.PRODUCTS_15 in document("PB-BB")/db/PRODUCTS
  where ($PB_BB.PRODUCTS_15/LIST_PRICE gt 100)
  return
  $PB_BB.PRODUCTS_15
  }
 </expensive_products>
 <cheap_products>
  {
  for $PB_BB2.PRODUCTS_21 in document("PB-BB")/db/PRODUCTS
```

```
   where ($PB_BB2.PRODUCTS_21/LIST_PRICE le 100)
   return
   $PB_BB2.PRODUCTS_21
   }
  </cheap_products>
</results>
```

In the above query `PRODUCTS` with list prices greater than or equal to [`gt`] 100 are returned for the PB_BB data source. Similarly, `PRODUCTS` with list prices less than 100 are returned for the PB_BB2 data source. Of course the underlying data source is the same.

The Self-Join

Another example of necessary source replication would be a self-join in SQL. In the classic example of a self-join, the query retrieves all employee names that match a particular manager ID.

```
SELECT emp.name, mgr.name
FROM employee emp, employee mgr
WHERE emp.manager_id = mgr.id
```

In XQuery, the query would appear similar to that in Listing 4-2.

Listing 4-2 Query retrieves records where employee manager ID field matches a particular manager ID

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

Enhancing Readability or Code Efficiency With Duplicate Source Schemas

In ambiguous cases, both replicating and not replicating a source would lead to reasonable queries. For example, a self-join to get employee-manager pairs was shown in a previous example. Without replicating the source, you could:

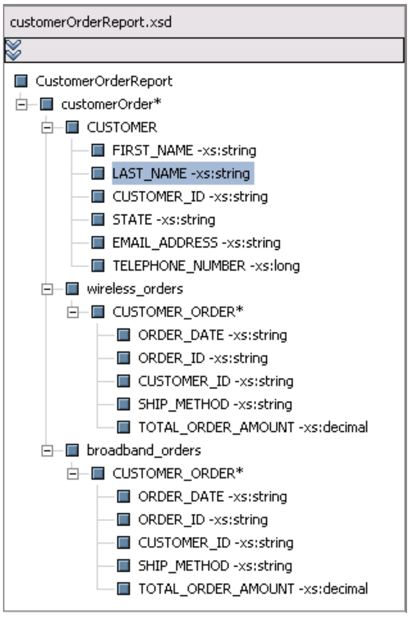
1. Map name to the target (get the employee name)
2. Join `manager_id` with `id` (join to get the manager)
3. Map name to the target (get the manager name)

Of course, the Data View Builder would interpret this query as: "give me all employees who are their own manager". Under such circumstances the option of creating multiple copies of a source schema reduces possible confusion or confusing results.

Target Schemas

A *target schema* describes the structure of a query result that will be produced when the query runs. As with source schemas, the Data View Builder provides a graphical representation of target schemas in a tree structure format.

Figure 4-3 Sample Target Schema



```
customerOrderReport.xsd

CustomerOrderReport
  customerOrder*
    CUSTOMER
      FIRST_NAME -xs:string
      LAST_NAME -xs:string
      CUSTOMER_ID -xs:string
      STATE -xs:string
      EMAIL_ADDRESS -xs:string
      TELEPHONE_NUMBER -xs:long
    wireless_orders
      CUSTOMER_ORDER*
        ORDER_DATE -xs:string
        ORDER_ID -xs:string
        CUSTOMER_ID -xs:string
        SHIP_METHOD -xs:string
        TOTAL_ORDER_AMOUNT -xs:decimal
    broadband_orders
      CUSTOMER_ORDER*
        ORDER_DATE -xs:string
        ORDER_ID -xs:string
        CUSTOMER_ID -xs:string
        SHIP_METHOD -xs:string
        TOTAL_ORDER_AMOUNT -xs:decimal
```

Target schemas have these main purposes:

- Provide a template for the mapping data from source schemas in order to generate a query.
- Provide a schema for Liquid Data Web Service and Data View definitions.
- Determine the structure and order of the XML document generated by the XQuery.

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

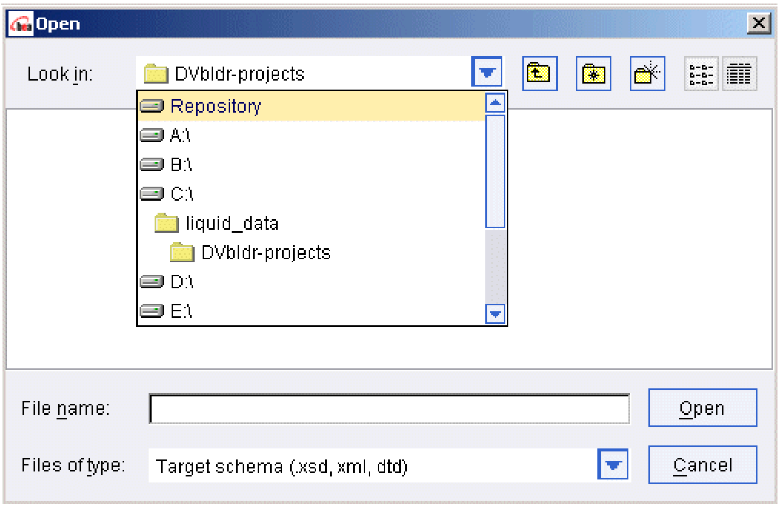
- Select a schema from the Liquid Data repository.
- Build a new target schema from scratch (using right-click menu commands).

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.

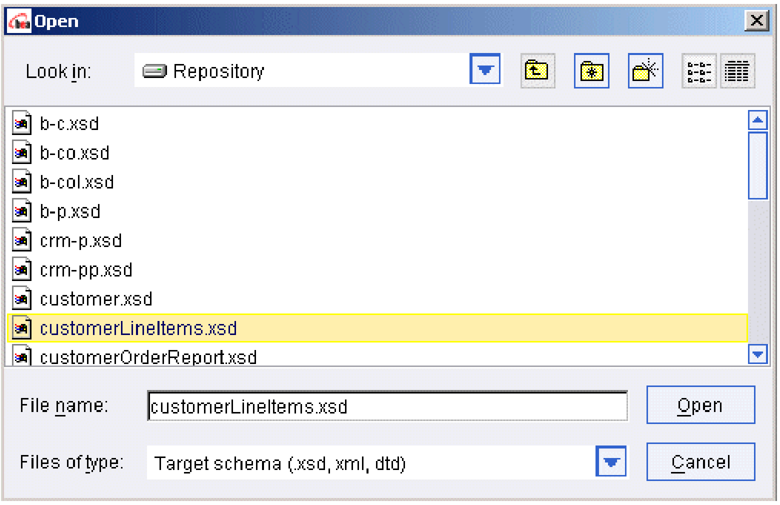
Figure 4-4 Liquid Data Repository Highlighted in File Browser



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

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

Figure 4-5 Schema File Selected



The target schema is displayed and docked on the right side of the Design tab work area.

(You can also choose the menu item File —> Set Selected Source Schema as Target Schema to create a target schema that is, at least initially, based exactly on a source schema.)

Guidelines for Working With Target Schemas

Use these guidelines when working with target schemas:

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 "Building Queries" on page 5-1 demonstrate this guideline.

2. Project at least one element from each data source that is part of the query to the target schema.

3. It is generally unnecessary to map every element in a target schema. For example, you could choose the same schema for both the source and target data structure, but then map only some of the source elements to the target schema. The query result will show only those source data elements that are actually mapped to elements in the target schema.

4. Understand how target schema conformity works and use it efficiently. (See "Managing Target Schema Properties" on page 5-26 for details on schema element property settings.)

   – A plus sign [+] next to a element indicates that the element is repeatable and *required*. (In other words, there must be one or more occurrences of this.) Since this setting requires extra checking of the data, most queries that use it pay a performance penalty.

   – An asterisk [*] next to an element indicates that the element is repeatable and *optional*. (In other words, there can be 0 or more occurrences of this.)

     Use this setting when possible 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.

   Of the several examples included in this section the following particularly demonstrate these guidelines:

   – "Example 1: Retrieve All BroadBand Customers, Returning Their Wireless Orders, If Any (ORDER is Repeatable and Optional)" on page 5-32

   – "Example 2: Retrieve Only BroadBand Customers Who Have At Least One Wireless Order; Return Their Wireless orders (ORDER Is Repeatable And Required)" on page 5-34

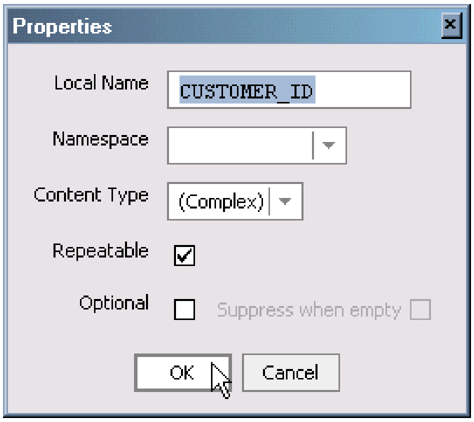
5. If your plan is to create a data view from your query, your target schema should only contain required elements that are utilized in the query. For example, if the `Customer` table contains `first_name`, `last_name`, `email`, and **`phone`** elements and each of those elements is required in the target schema, then you need to map each element of your query before saving it.

For a detailed description of target schemas, see "Schemas and Namespaces in Liquid Data" on page 4-1.

Managing Target Schemas

Target schemas are composed of *complex elements*, *simple elements* (*child elements*), and *attributes*. You can set element properties using the Properties dialog box, which you access by right-clicking on the element.

Figure 4-6 Properties Dialog



The following properties can be set:

- **Local name.** Allows you to change the element name.
- **Namespace.** Allows you to specify a previously created XML namespace. (See "Understanding XML Namespaces" on page 4-13 for a discussion of namespaces.)
- **Content type.** Allows you to choose a content type such as a string or Boolean from a drop-down list.
- **Repeatable attribute.** Provides for schema elements and their sub-elements to be repeatable.
- **Optional attribute.** Provides for schema elements to be optional rather than required.

Using Schemas Saved With Projects

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

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

Offline resources are available only to the previously associated project.

Schema Import Resolution Rules

If a schema file has an import statement with a relative path to another schema file, Liquid Data resolves the location of the imported files according to the following rules:

1. Attempt to resolve the filename in the `<ldrepository>/schemas` directory.
2. If the file is not found in the `<ldrepository>/schemas` directory, attempt to resolve it relative to the directory in which the schema file (the first one with the import statement) is saved.
3. If imported schema files in turn import other schema files, they are resolved first from the `<ldrepository>/schemas` directory.

   In the case of the Liquid Data Server Samples repository, the first attempt to resolve the search will be in the following directory:

   ```
   <WL_HOME>/samples/domain/liquiddata/ldrepository/schemas
   ```

   and then from the location relative to the original schema file (the first one with the import statement).

For example, if you have a schema file in the following location in the repository:

```
<ldrepository>/schemas/dir1/dir2/s.xsd
```

and it contains the following import statement:

```
import dir3/file.xsd
```

then Liquid Data first looks for a schema file named:

```
<ldrepository>/schemas/dir3/file.xsd
```

and, if it does not find it relative to the root level of the repository, Liquid Data looks for it in:

```
<ldrepository>/schemas/dir1/dir2/dir3/file.xsd
```

As a further example, assume the `file.xsd` import was resolved in:

```
<ldrepository>/schemas/dir3/file.xsd
```

If `file.xsd` in turn has the following import statement:

```
import dir4/another.xsd
```

then Liquid Data first attempts to resolve this import statement relative to the root of the repository:

```
<ldrepository>/schemas/dir4/another.xsd
```

If the file is not there, it then resolves it relative to the original `<ldrepository>/schemas/dir1/dir2/s.xsd` file, as follows:

```
<ldrepository>/schemas/dir1/dir2/dir4/another.xsd
```

Understanding XML Namespaces

XML namespaces are a mechanism by which you can ensure that there are no name conflicts (or ambiguity) when combining XML documents or referencing an XML element.

Liquid Data supports XML namespaces and includes namespaces in the queries generated in Data View Builder.

This section includes the following topics:

- XML Namespace Overview
- Using XML Namespaces in Liquid Data Queries and Schemas
- Migrating Liquid Data 1.0 Queries

XML Namespace Overview

XML namespaces appear in queries as a string followed by a colon. For example, the `xs:integer` data type uses the XML namespace `xs`. Actually, `xs` is an alias (called a *prefix*) for the URI name of the namespace. (See Table 4-7 for the full set of predefined XQuery namespaces.)

Prefix	URI Name
`xs`	`http://www.w3.org/2001/XMLSchema`

XML namespaces ensure that names do not collide when combining data from heterogeneous XML documents.

For example, there could be an element `<tires>` in a document related to automobile manufacturers. In a document related to bicycle tire manufacturers, there is also a `<tires>` element. Obviously, combining these elements would be problematic under most circumstances. XML namespaces easily avoid such name collisions by referring to the elements as `<automobile:tires>` and `<bicycle:tires>`.

In a XML schema namespaces — including the *target namespace* — are declared in the schema tag. Here is an example:

```
<schema   xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:bea="http://www.bea.com/public/schemas"
          targetnamespace="http://www.bea.com/public/schemas"
          ...
```

The first line of the above schema contains the *default namespace*, which is the namespace of all the unqualified elements in the schema.

For example, if you see the following element in a schema document:

```
<element name="appliance" type="string"/>
```

the element `element`, and the attribute `name` and `type` all belong to the default namespace, as do unprefixed types such as `string`.

The second line of the schema contains a namespace declaration — `bea` — which is simply an association of a URI with a prefix. There can be any number of such declarations in a schema.

Lastly, comes the target namespace, declared with the `targetNamespace` attribute. It this case, the target namespace is bound to the namespace declared on the second line, meaning that all element and attribute names declared in this document belong to:

```
http://www.bea.com/public/schemas
```

References to types declared in this schema document must be prefixed. For example:

```
<complexType name="AddressType">
         <sequence>
         <element name="street_address" type="string"/>

         ...

         </sequence>
</complexType>

<element name="address" type="bea:AddressType"/>
```

Predefined Namespaces in XQuery

The following table shows predefined namespaces used in XQuery:

Table 4-7 Predefined Namespaces in XQuery

Namespace Prefix	Description	Examples

`xf`	The prefix for XQuery functions.	`xf:data`
		`xf:sum`
		`xf:substring`

`xfext`	The prefix for Liquid Data-specific extensions to the standard set of XQuery functions.	`xfext:match` `xfext:trim`
`xs`	The prefix for XML schema types.	`xs:element` `xs:string`
`xsext`	The prefix for Liquid Data-specific extensions to the standard set of XML schema types.	`xsext:myownstringtype`

Other XML Namespace References

The following are some Internet links where you can find more information on XML namespaces:

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

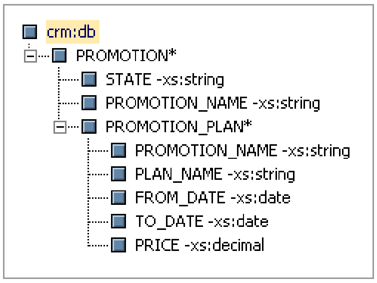
See also "Supported XQuery and XML Schema Versions In Liquid Data" in the *XQuery Reference Guide*.

Using XML Namespaces in Liquid Data Queries and Schemas

The Data View Builder automatically generates the correct namespace declarations when generating a query.

However, when a target schema is created in the Data View Builder, its elements and attributes are `unqualified`, meaning that the target namespace is not automatically part of the element or attribute name.

Figure 4-8 Example of a schema with unqualified attributes and elements



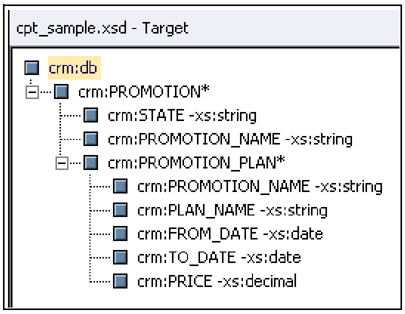
If you want elements and attributes appear as qualified, you need to use an editor outside Data View Builder to modify the generated schema for either or both `attributeFormDefault` and `elementFormDefault` to be set to *qualified*. See Listing 4-3.

Listing 4-3 Schema Tag Setting Elements and Attributes to Qualified (emphasis added)

```
<xsd:schema targetNamespace="urn:schemas-bea-com:ld-cocpt"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:cocpt="urn:schemas-bea-com:ld-cocpt" attributeFormDefault="qualified"
elementFormDefault="qualified">
```

Once attributes and elements have been set to qualified, they will appear as such in the Data View Builder when the target schema is set to your newly edited file.

Figure 4-9 Example of a schema with qualified attributes and elements



Note: If you are hand-coding your queries (not using the Data View Builder as a query generator), you must include the necessary namespace declaration(s) to satisfy Liquid Data server requirements. For a list of data sources that require namespace declarations, see "Data Sources that Require Namespace Declarations" on page 4-18.

Namespace Declarations in XQuery Prolog

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

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

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

Consider the following simple query:

```
namespace view = "urn:views"

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

The line in the prolog:

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

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

You must define namespaces in the XQuery prolog in order to use them in a query (except for the predefined namespaces described in "Predefined Namespaces in XQuery" on page 4-15). If you do not define namespaces in the XQuery prolog, the query will fail with a compilation error.

Defining Namespaces in Target Schema

When you use the Data View Builder to create or modify target schemas, you can specify a namespace for an element or an attribute. Such a specified namespace is added to the XML markup in the query (and therefore to the query results).

You can set or change a target namespace using the Target Namespace menu option, available from the Data View Builder Query menu when in Design mode.

Figure 4-10 Target Namespace Dialog Box

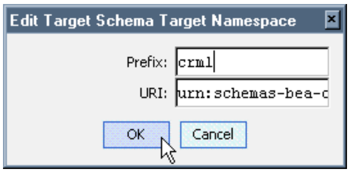
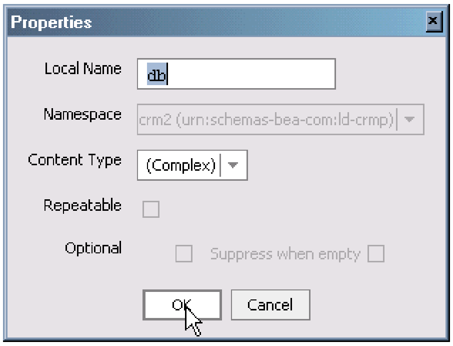


Figure 4-11 shows adding a local name called `db` to an element of the target schema named `crm2` from the Properties dialog box. If multiple namespaces are available, you can select one from the drop-down list box.

You can access the Properties dialog box by right-clicking on an element in your target schema.

Figure 4-11 Properties Dialog Box



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

```
<crm2:db xmlns:crm2="urs:schemas-bea-com:ld-crmp"> 100.0 </crm2:db>
```

Data Sources that Require Namespace Declarations

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

- Data Views
- Web Services
- Application Views
- Stored Procedures
- Complex Parameter Types

Migrating Liquid Data 1.0 Queries

Liquid Data 1.0 did not support XML namespaces, and any queries used in Liquid Data 1.0 must be migrated to work in Liquid Data 8.1. If you have queries that are generated in a Data View Builder project file, you can open the project file in Data View Builder 8.1. When you click the Test tab, the Data View Builder automatically generates the new query with the proper namespace declarations in the query prolog.

If you have stored queries and data views, you must use the `queryMigrate` tool to migrate the queries so they work properly in Liquid Data 8.1. For information on the `queryMigrate` tool, see Migrating from Liquid Data 1.0 to 8.1 in the Liquid Data *Migration Guide*.

Building Queries

This chapter explains how to design and build a BEA Liquid Data for WebLogic query using the Data View Builder, including manually applying condition scoping rules. The following topics are covered:

- Defining Query Requirements
- Managing Query Components
- Working With Source and Target Schema Elements
- Setting Query Conditions
- Sorting Query Results
- Using Existential Condition Checking in Queries
- Using Automatic Type Casting

Defining Query Requirements

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

- What is the description of the data integration problem to be solved?
- How do I want the query result to look? In other words, how do I want to structure the output?
- What types of data sources does my query need?

- What is the structure of each data source; that is, what are the input (if any) and output XML schemas for the source?
- How does source data map to the target?
- What conditions do I need to define? (Conditions filter source data in a specific way.)
- What target XML schema *design pattern* should I use? More specifically, what is the appropriate cardinality of each element in the target schema? Proper design of the target XML schema is a key factor in building a successful and efficient query.

Once you have designed the query and defined an outline strategy for accomplishing the information mapping and filtering, you are ready to build a test version of your query. For other than very simple queries, you will probably revise, refine and test the query several times.

Examples Set-up

This chapter contains several illustrated, stepped examples. If you want to work through these examples in the Data View Builder, you can easily do so.

Unless otherwise indicated each example requires the following set-up instructions.

1. Move schemas for the following two data sources into the Data View Builder work area:

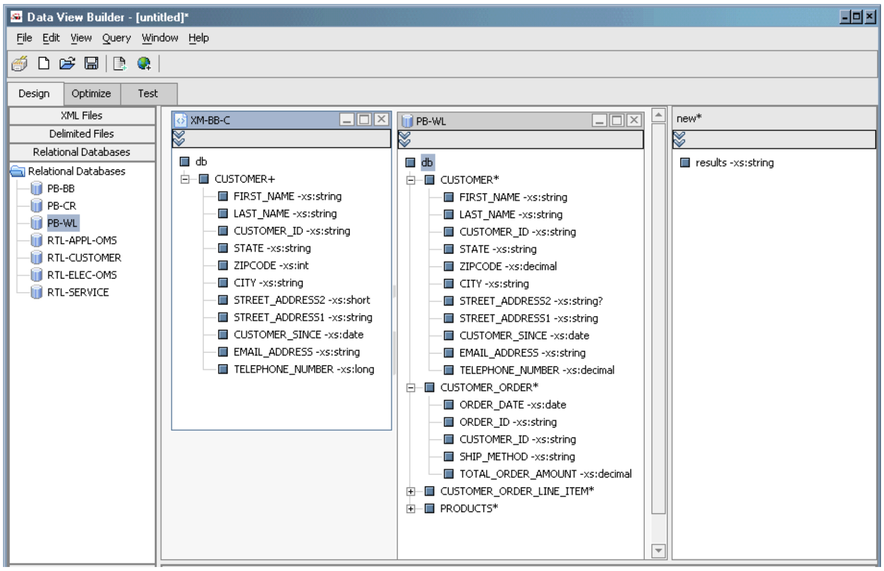
- Relational Database: `PB-WL`
- XML File: `XM-BB-C`

2. With the Data View Builder open click the Design tab.
3. On the Builder Toolbar, click the Sources tab (on the bottom of the left vertical panel).
   - Click Relational Databases and then double-click on `PB-WL` data source to open the associated XML schema showing Wireless customers.
   - Click XML Files and then double-click on `XM-BB-C` data source to open the associated XML schema showing BroadBand customers.

   The schemas for each of the data sources appear.

   Position the schema windows so you can view the data elements in each schema. You can expand the data elements by clicking the plus `[+]` sign next to the element name. For example, in the `PB-WL` data source, CUSTOMER is a *complex element* with subordinate *simple elements*.

Figure 5-1 Example with Data Sources Expanded



4. To create and set the target schema for this example cut-and-paste the XML in Listing 5-1 into a plain text file and save it to the Liquid Data Server repository under the file name:

   ```
   amtByState.xsd
   ```

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

   ```
   <ldrepository>/schemas
   ```

   An example of a full path to the Liquid Data Server repository is:

   ```
   /bea81/weblogic81/samples/domains/liquiddata/ldrepository
   ```

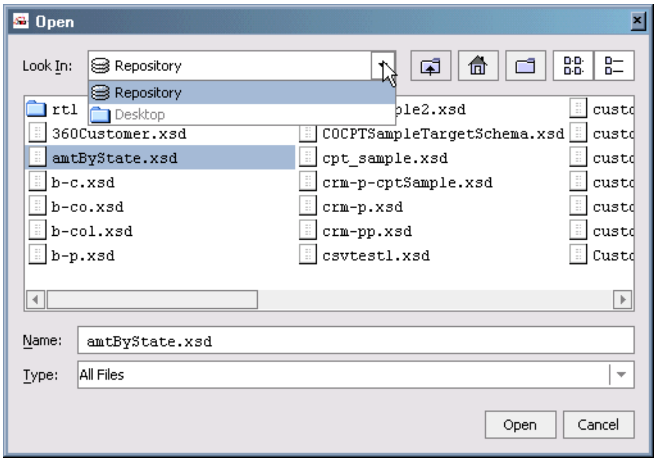
Listing 5-1 XML source for amtByState.xsd target schema

```
<?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>
```

5. Navigate to the Liquid Data Server repository, the topmost directory in the browser.
6. Choose `amtByState.xsd` and click Open.

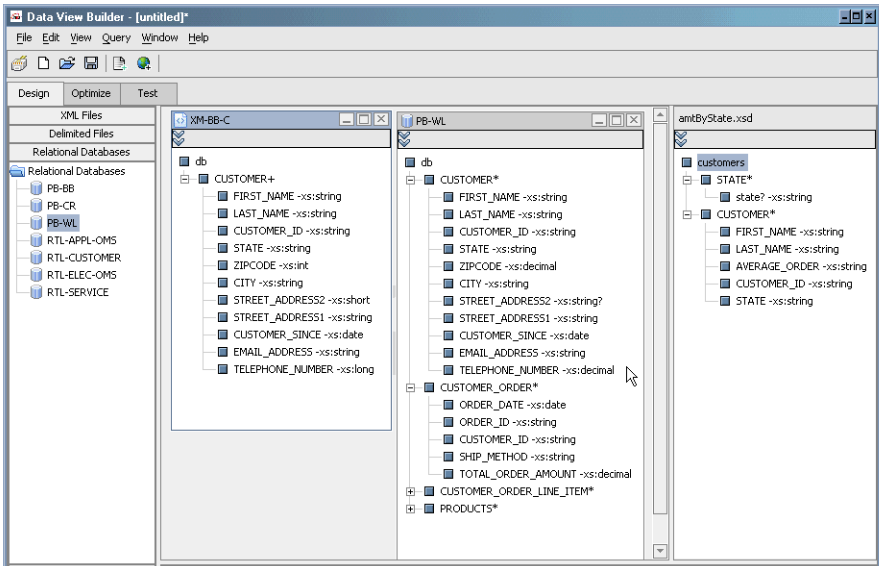
Figure 5-2 Selecting the Newly Created Schema as Target Schema



Name: amtByState.xsd
Type: All Files

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

Figure 5-3 Example Showing Data Sources and Target Schema



```
Relational Databases
  PB-BB
  PB-CR
  PB-WL
  RTL-APPL-OMS
  RTL-CUSTOMER
  RTL-ELEC-OMS
  RTL-SERVICE
```

```
db
  CUSTOMER+
    FIRST_NAME -xs:string
    LAST_NAME -xs:string
    CUSTOMER_ID -xs:string
    STATE -xs:string
    ZIPCODE -xs:int
    CITY -xs:string
    STREET_ADDRESS2 -xs:short
    STREET_ADDRESS1 -xs:string
    CUSTOMER_SINCE -xs:date
    EMAIL_ADDRESS -xs:string
    TELEPHONE_NUMBER -xs:long
```

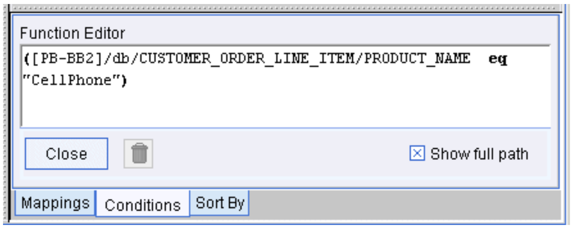
```
db
  CUSTOMER*
    FIRST_NAME -xs:string
    LAST_NAME -xs:string
    CUSTOMER_ID -xs:string
    STATE -xs:string
    ZIPCODE -xs:decimal
    CITY -xs:string
    STREET_ADDRESS2 -xs:string?
    STREET_ADDRESS1 -xs:string
    CUSTOMER_SINCE -xs:date
    EMAIL_ADDRESS -xs:string
    TELEPHONE_NUMBER -xs:decimal
  CUSTOMER_ORDER*
    ORDER_DATE -xs:date
    ORDER_ID -xs:string
    CUSTOMER_ID -xs:string
    SHIP_METHOD -xs:string
    TOTAL_ORDER_AMOUNT -xs:decimal
  CUSTOMER_ORDER_LINE_ITEM*
  PRODUCTS*
```

```
customers
  STATE*
    state? -xs:string
  CUSTOMER*
    FIRST_NAME -xs:string
    LAST_NAME -xs:string
    AVERAGE_ORDER -xs:string
    CUSTOMER_ID -xs:string
    STATE -xs:string
```

Using the Function Editor

You can use the Data View Builder Function Editor to build up XQuery functions (see "Using XQuery Functions" on page 5-14 for more information).

Figure 5-4 Function Editor



```
([PB-BB2]/db/CUSTOMER_ORDER_LINE_ITEM/PRODUCT_NAME  eq
"CellPhone")
```

To use the Function Editor:

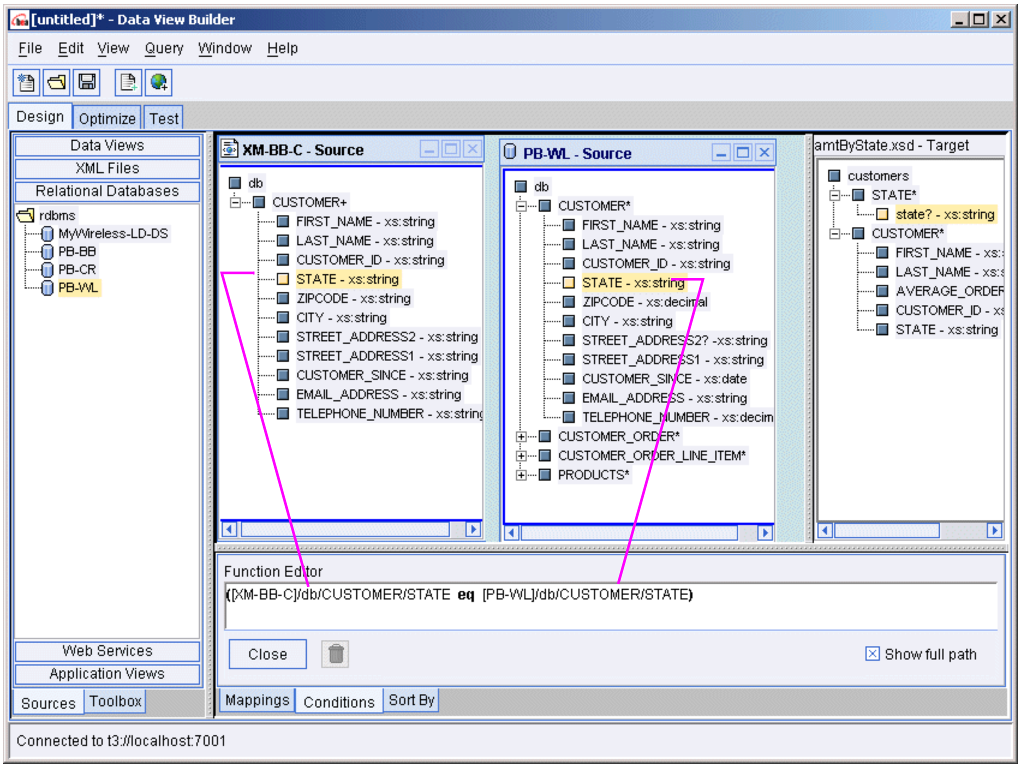
1. Drag and drop an XQuery equals function [`eq`] from the Toolbox panel to the first empty row in the Conditions tab.
2. Drag a source schema element and drop it into the same row of the Condition column. Drag a second source schema element and drop it into the same row of the Condition column.

To edit an existing function:

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

2. Edit the statement as needed. You will need to delete the current parameters or function using the Trashcan or Delete key. Then drag and drop a new function or source elements/attributes to the Functions Editor.

Figure 5-5 Mapping Elements to Functions



([XM-BB-C]/db/CUSTOMER/STATE **eq** [PB-WL]/db/CUSTOMER/STATE)

To get the view shown in Figure 5-5, click on the Conditions tab, select the row with the condition to be edited, then click the Edit button.

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

For more information about using the Functions Editor and working with functions see "XQuery Functions" on page 2-13 and "Function Editor" on page 2-14.

Managing Query Components

If you think of selected *data elements* as nouns (what you want to work on), the *functions* as verbs (the action), then the *mapping* among the data elements creates a logical sentence that expresses the *query*.

Results and query performance can change significantly depending on how you:

- Map (or *project*) source data from one or more sources to the target schema
- Specify conditions (filter source data)
- Tune the target schema

Although you can simply type in an XQuery and run it from the Data View Builder, the more common way to create a query is build it up through the following operations:

- Map simple or complex elements from source schemas to target schemas
- Define constants and/or query parameters
- Create join relationships between source schema elements
- Transform information using built-in or custom functions
- Filter data using conditions

In the Data View Builder these operations can occur in any order and are fully reversible.

If you have taken the time to outline a design for the query first, constructing it will be a matter of drag-and-drop query building. Then you can test, fine-tune, and modify your project as needed to produce variations on the results, or to optimize the query for better performance.

In addition to data sources (see "Data Sources" on page 3-1), constants, query parameters, and XQuery functions are used in constructing a Liquid Data XQuery graphically.

Data Sources

A data source is represented in the Data View Builder through a *source XML schema*. You can use multiple data source schemas in your query. In some cases you may need to use a single source schema multiple times. Some data sources require input data as well.

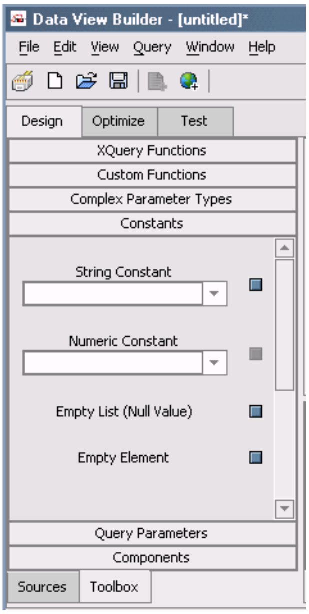
Using the Sources tab on the Data View Builder Toolbar you can access available data sources, grouped by type, that are configured on the Liquid Data Server to which you are connected. Note that a data source group (such as Relational Databases) appears only if at least one source of that type has been configured in the Liquid Data Server to which you are connected.

See Liquid Data *Getting Started* and *Administration Guide* for examples of configuring and using data sources.

Creating and Using Constants

You can add constants to functions or use constants as part of any query condition.

Figure 5-6 Toolbox Constants Panel



To create a constant choose Toolbox —>Constants. Four options are available:

- **String Constant.** 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.
- **Number Constant.** Numbers can be integers (positive or negative), decimal values, or floating point expressions.

- **Empty List (null value).** Creates an empty list in the generated XQuery such as:

  ```
  <CUSTOMER_ID>{ () }</CUSTOMER_ID>
  ```

- **Empty Element.** Creates an empty element in the generated XQuery such as:

  ```
  <STATE/>
  ```

Using Constants with Functions

To include a constant as a function parameter, follow steps similar to those in the following example:

1. Drag the desired XQuery function to a row on the Condition tab or to the Liquid Data desktop. For example: choose the `starts-with` function. You get the following placeholder in the Functions Editor:

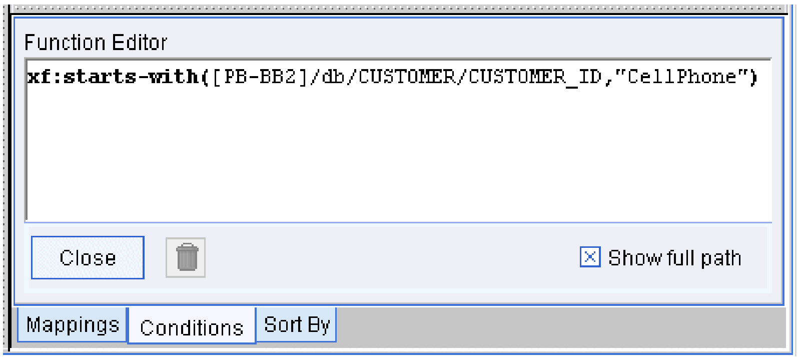
   xf:starts-with(*str1*,*str2*)

2. Drag an appropriate source element 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, `CellPhone`. Drag the Constants icon onto the second string placeholder (*str2*).

   The condition appears in the Functions Editor as shown in Figure 5-7.

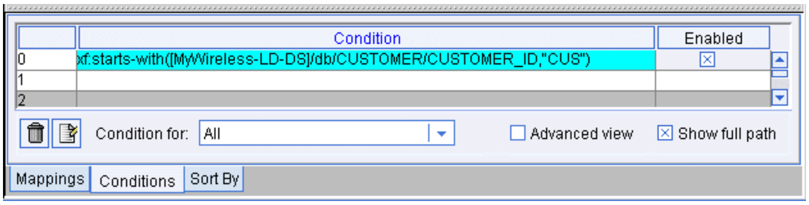
Figure 5-7 Condition with starts-with Constant in Functions Editor



```
xf:starts-with([PB-BB2]/db/CUSTOMER/CUSTOMER_ID,"CellPhone")
```

Close the Functions Editor by clicking the Close button. The new condition you created appears in the Source column on the Condition tab.

Figure 5-8 Condition with starts-with Constant in Row on Conditions Tab



	Condition	Enabled
0	xf:starts-with([MyWireless-LD-DS]/db/CUSTOMER/CUSTOMER_ID,"CUS")	☒
1		
2		

Condition for: All

☐ Advanced view ☒ Show full path

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 queries will differ somewhat even though the functionality will be the same.

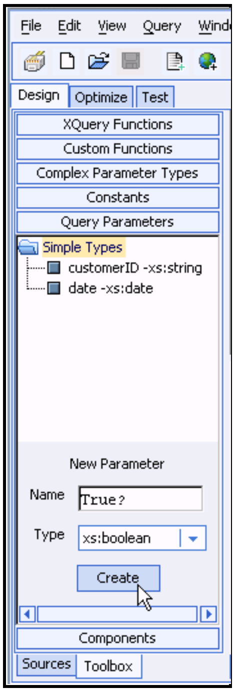
Creating and Using Query Parameters

Using a query parameter you can change a value in your query each time it is run. This is ideal for *ad hoc* queries based around changes in a customer name or order number.

The Query Parameter section of the Toolbox provides a text field where you can enter a new parameter name. To create a query parameter:

1. Name your query parameter.
2. Select the parameter type from the drop-down list (See Table 5-10, “Query Parameter Types,” on page 5-12).
3. Click Create. Your new parameter will appear in the Query Parameter resource tree (Figure 5-9).

Figure 5-9 Query Parameters Dialog Box



To expand the list of query parameters right-click on the Simple Types folder. You can then right-click on the query parameter name to rename or delete the parameter.

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

For an example showing use of a query parameter, see the Getting Started demo:

`http://e-docs.bea.com/liquiddata/docs81/interm/demopage.html`

Table 5-10 Query Parameter Types

Parameter Type	Examples
`xs:boolean` (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.
`xs:byte` (byte)	A positive or negative whole number. The maximum value is `127` and the minimum value is `-128`. For example: • `-1` • `0` • `126` • `+100`
`xs:date` (date)	Input must be in this format: *`MMM dd, YYYY`* For example: `JUN 12, 2002`

`xs:dateTime` (datatype)	Input must be in this format:
	`MMM dd, YYYY HH:MM:SS AM/PM`
	For example:
	`MAY 12, 2002 12:12:11 AM`

`xs:decimal` (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.`
`xs:double` (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).

Table 5-10 Query Parameter Types

Parameter Type	Examples
`xs:float` (floating point)	A real number (negative or positive) that can contain a fractional part. For example: • `100.0` • `-100.5` **Note:** Liquid Data does not support floating point formats expressed in fractions (½) or IEEE floating point notation (3E-5).
`xs:int` (int)	A positive or negative whole number. The maximum value is 2147483647 and minimum value is -2147483648. For example: • `-1` • `0` • `126789675` • `+100000`
`xs:integer` (integer)	A positive or negative whole number. The maximum value is 2147483647 and minimum value is -2147483648. For example: • `1` • `-100` • `+100`
`xs:long` (long)	A positive or negative whole number. The maximum value is 9223372036854775807 and minimum value is -9223372036854775808. For example: • `-1` • `0` • `12678967543233` • `+100000`
`xs:short` (short)	A positive or negative whole number. The maximum value is 32767 and minimum is -32768. For example: • `-1` • `0` • `126789` • `+10000`

Table 5-10 Query Parameter Types

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

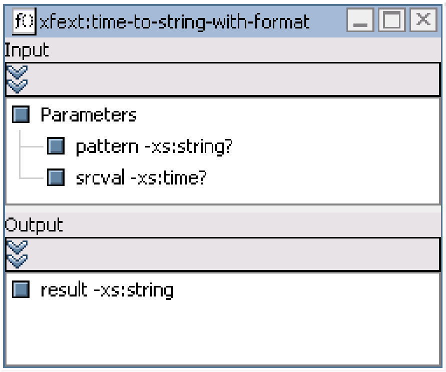
Using XQuery Functions

In Liquid Data, XQuery functions are a set of built-in functions that allow you to graphically establish functional relationships between data elements or to apply business logic to data.

You can double-click or drag and drop a function to move it the Liquid Data desktop. The function will appear in a structured format that displays the number and type of input parameters required, as well as the output parameter.

For most XQuery functions you drag-and-drop one or more information element to the function. The information element can be source data, variables, or constant values. Functions return results based on input and the output element with which the results are associated.

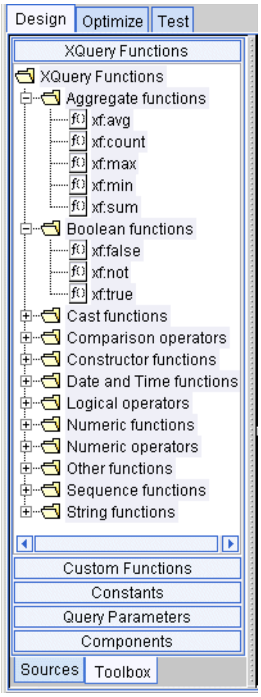
Figure 5-11 Sample XQuery Function as it Appears in the Data View Builder Work Area



For example, if you want to find out how many customer IDs in the BroadBand database are *not equal to* those in the Wireless database you can use the [`ne`] (not-equal-to) function.

To access this function go to Builder Toolbar —> Toolbox tab —> XQuery Functions area, expand the `Operators` element, and drag the [`ne`] function into the work area.

Figure 5-12 XQuery Functions Panel Showing Aggregate and Boolean Functions Tab Expanded



Note: Automatic type casting is available to help ensure that input parameters used in functions and mappings are appropriate to the function in which they are used. When Automatic Type Casting is active, 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 5-59.

Most XQuery functions in the Data View Builder are standard XQuery functions supported by the W3C. (For related information about using functions, see "Functions Reference" in the *XQuery Reference Guide*. For more detailed information, see the W3C *XQuery 1.0 and XPath 2.0 Functions and Operators* specification.)

Mapping Elements to Functions

When you drag and drop a source element onto another source element (either within the same source schema or among different source schemas) you are automatically creating a join which is represented in the Data View Builder as an equality relationship between the two elements/attributes using the [`eq`] (equals) function.

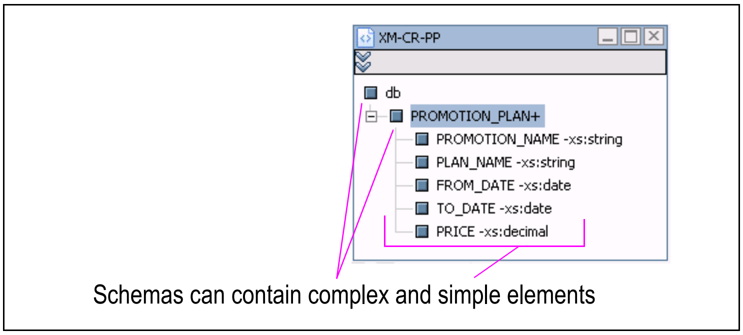
You can also create the same equality relationship by dragging and dropping the `eq` function onto a row in the Conditions tab and then dragging and dropping two source elements/attributes into the same row.

Working With Source and Target Schema Elements

Mapping schema elements involves establishing a visual relationship among data source elements, attributes, and functions and to a target schema.

There are two types of schema elements: *simple* and *complex*. *Complex elements* contain elements and/or attributes. *Simple elements* can hold content and have attributes, but do not contain other elements.

Figure 5-13 Expanded Schema Showing Complex and Simple Elements



To expand a complex element, right-click on it and choose Expand (or just double-click). If you do this for the topmost element in the schema, all the complex elements will be expanded.

Supported Drag-and-Drop Actions in the Data View Builder

The Data View Builder supports the drag-and-drop actions that are described in the following table.

Table 5-14 Supported Mapping Relationships

Action	Description
Map simple element from one source to another simple element in another source	Creates an equality [`eq`] relationship between the two elements/attributes using the [eq] (equals) function. These can be in the same or different source schemas.

Action	Description
Map simple element to a function	A data element is used as 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. See Example 2: Aggregates in *Liquid Data by Example* (specifically, the `Add the count` function within the Aggregates example).
Map simple element to a target element	Projects data element onto the target schema. Most query examples provided in this documentation show how to map source schema elements/attributes to target elements/attributes.
Map function output to target schema	A function (*f1*) output can be input to an element in a source schema.
Copy, then Paste a complex element to a target schema	Copies the structure of the complex element, including its simple elements and attributes, to the target schema. In order for these elements to be included in the generated query they must first be individually mapped.
Copy, then Paste and Map a complex element to a target schema	Copies the complex element to the target schema. Content of the element are shown in italics for information only. A generated query will treat the complex element as a unit. See "Complex Element Mappings" on page 5-20.

Mapping to Target Schemas

The Data View Builder automatically generates queries based on target schemas and the mappings into them. (See Liquid Data *Getting Started* for an example.)

The Data View Builder supports two types of mappings: *value mappings* and *complex element mappings*. Value mappings map (assign) only the value of an element or attribute from a source to the value of its target element or attribute. Element mappings allow mapping source elements (simple or complex) to target elements.

For more details on creating source and target schemas see "Source and Target Schemas" on page 4-2.

Mapping Elements and Attributes Between Source and Target Schema

Value mappings of elements and attributes allow you to map source contents to corresponding target elements.

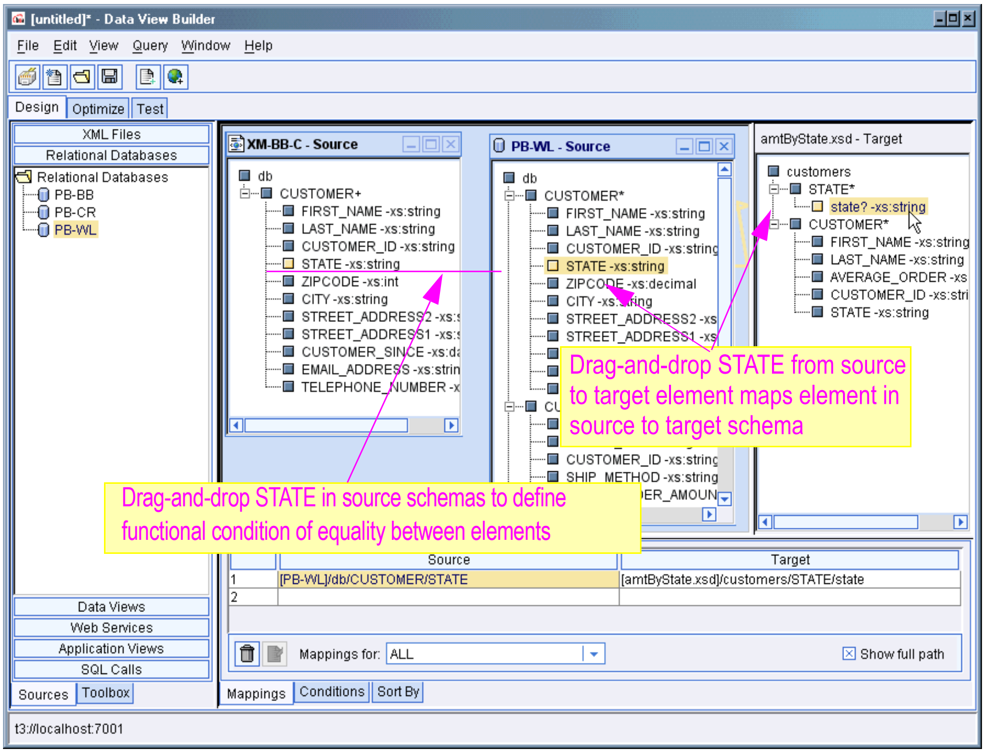
Figure 5-15 illustrates a simple join of the source element `STATE` in the BroadBand source schema (`XM-BB-C`) with a source element `STATE` in the Wireless source schema (`PB-WL`). This action joins the common elements in each schema and disregards those that do not occur in both schemas.

To *project* a result, you can designate how the output of this relationship should look when the query runs. Because you are collecting only information about states and defining only one element in the target schema, you are in effect asking the Data View Builder to fill only that data element in the result when the query runs.

If you are following along in the Data View Builder, drag and drop the `STATE` element in PB-WL source schema onto the `state?` element (under `STATE*`) in the target schema.

Symbols next to the element name such as [`*`], [`?`], and [`+`] represent repeatable and optional conditions. For details see "Managing Target Schema Properties" on page 5-26.

Figure 5-15 Mapping Elements in the Data View Builder



	Source	Target
1	[PB-WL]/db/CUSTOMER/STATE	[amtByState.xsd]/customers/STATE/state
2		

Complex Element Mappings

Complex element mappings provide a quick and efficient way to copy entire sub-parts of source elements to your target schema. This is useful where parts of the target result are (or should be) identical or nearly identical to parts of the sources.

There are many situations which you will find it convenient to map elements to your target schema, including:

- When you are creating a target schema from scratch.
- When you are sure that your source schema matches your target schema in terms of both elements and attributes.
- When you want elements individually mapped but it is easier to map complex elements, expand the mappings to include values, and then add or delete some mappings using right-click target schema management commands.

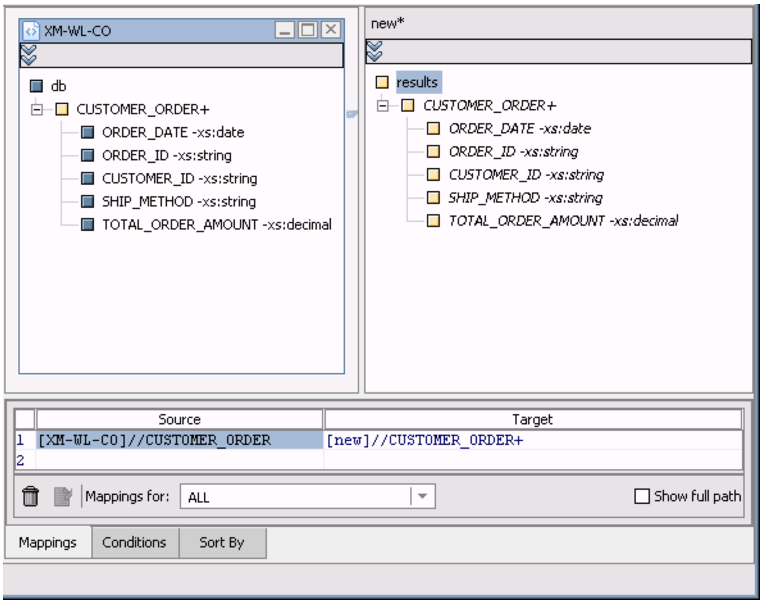
There are several benefits of mapping or projecting elements:

1. One-to-one mapping of multiple elements is less often needed.
2. The query is easier to read compared to a query where individual elements are mapped.
3. The query runs faster, due to fewer element or attribute constructors.
4. If the underlying structure of the complex element changes — an element is added, deleted, or an attribute is changed — the generated query does not change.

Figure 5-16 shows the results of the mapping of a complex element to the target schema. The mapping was accomplished by:

1. Choosing File —> New Project in the Data View Builder. This clears any data sources, target schema or other settings.
2. In Design mode double-clicking on the `XM-WL-CO` XML data source.
3. Right-clicking on the complex element named `CUSTOMER_ORDER`.
4. Choosing Copy.
5. Right-clicking on `results`.
6. Choosing Paste and Map. Mapped elements appear in italics, indicating that the elements are mapped without values.

Figure 5-16 Mapping a Complex Element



	Source	Target
1	[XM-WL-CO]//CUSTOMER_ORDER	[new]//CUSTOMER_ORDER+
2		

When you select Test mode, an XQuery is generated that returns all the child elements of CUSTOMER_ORDER.

Listing 5-2 XQuery resulting from mapping of CUSTOMER_ORDER complex element

```
<results>
   {
   for $XM_WL_CO.CUSTOMER_ORDER_1 in
document("XM-WL-CO")/db/CUSTOMER_ORDER
      return
      $XM_WL_CO.CUSTOMER_ORDER_1
   }
</results>
```

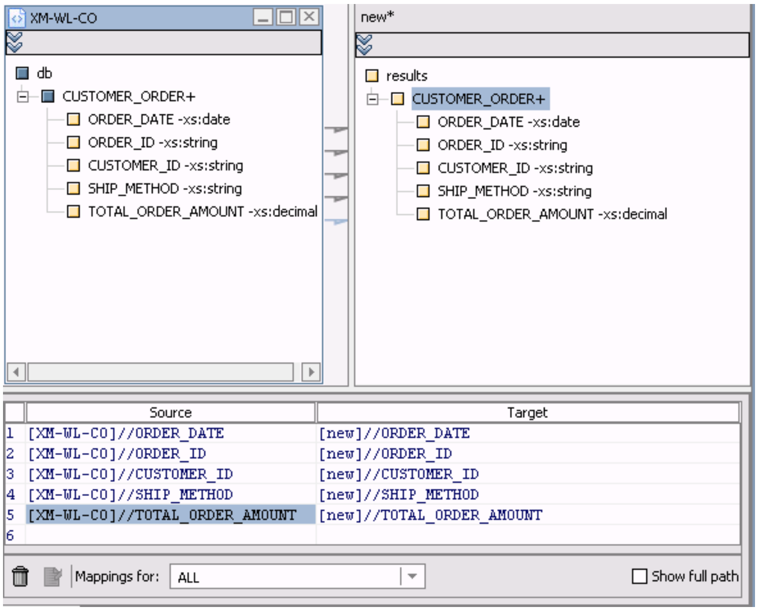
Expanding Mapped Complex Elements

Here are two examples where you might find it useful to use element mapping even when there is not an exact match between the source and target schema:

- A complex element called `Customer` may contain `FIRST_NAME`, `LAST_NAME`, `PHONE`, `ADDRESS`, and so on. Even if you don't want every one of these elements mapping it may be easier to map all first and then delete a few mappings.
- You may have a target schema that is close to the source, but not an exact match. In such a case it may be easier to:
  - Delete the target schema element(s)
  - Copy, then paste-and-map the source element(s)
  - Expand the mapping to include values
  - Edit the target schema as needed

Figure 5-17 shows the results of the mapping a set of simple elements to their corresponding elements in the target schema. Although this mapping could have been accomplished by drag-and-drop of each element individually, it was easier to follow the steps for mapping a complex element (Figure 5-16) and then to right-click on the complex element name and select `Expand complex mapping`. The results is exactly as if you had individually mapped all the simple elements from source to target schema. In this case no further editing of the target schema was done.

Figure 5-17 Mapping Simple Elements



	Source	Target
1	[XM-WL-CO]//ORDER_DATE	[new]//ORDER_DATE
2	[XM-WL-CO]//ORDER_ID	[new]//ORDER_ID
3	[XM-WL-CO]//CUSTOMER_ID	[new]//CUSTOMER_ID
4	[XM-WL-CO]//SHIP_METHOD	[new]//SHIP_METHOD
5	[XM-WL-CO]//TOTAL_ORDER_AMOUNT	[new]//TOTAL_ORDER_AMOUNT
6		

When Test mode is selected a query is generated based on the *value mapping* of all sub-elements associated with `CUSTOMER_ORDER`.

Listing 5-3 XQuery resulting from the mapping of individual CUSTOMER_ORDER elements

```
<results>
   {
   for $XM_WL_CO.CUSTOMER_ORDER_1 in document("XM-WL-CO")/db/CUSTOMER_ORDER
   return
   <CUSTOMER_ORDER>
        <ORDER_DATE>{ xf:data($XM_WL_CO.CUSTOMER_ORDER_1/ORDER_DATE)
}</ORDER_DATE>
        <ORDER_ID>{ xf:data($XM_WL_CO.CUSTOMER_ORDER_1/ORDER_ID)
}</ORDER_ID>
        <CUSTOMER_ID>{ xf:data($XM_WL_CO.CUSTOMER_ORDER_1/CUSTOMER_ID)
}</CUSTOMER_ID>
        <SHIP_METHOD>{ xf:data($XM_WL_CO.CUSTOMER_ORDER_1/SHIP_METHOD)
}</SHIP_METHOD>
        <TOTAL_ORDER_AMOUNT>{
xf:data($XM_WL_CO.CUSTOMER_ORDER_1/TOTAL_ORDER_AMOUNT)
}</TOTAL_ORDER_AMOUNT>
   </CUSTOMER_ORDER>
   }
</results>
```

Notes:

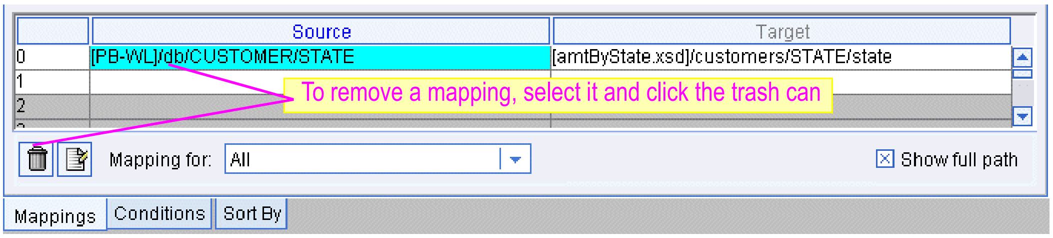
- You cannot clone a complex element using the Paste and Map option. Instead either rename or delete the complex element in the target schema first or map it to a different location in the schema.
- You cannot map the same element from more than one source schema to a single element in the target schema. For example, if you map `STATE` (under `CUSTOMER`) from the BroadBand database to `cust_state` in the target schema and then map `STATE` from a second source schema to `cust_state` in the target schema, only the latter mapping will apply.

Removing Mappings

Mapped elements/attributes in a query are displayed on the Mappings tab. You can delete mappings between elements and attributes by:

1. Highlighted the element or attribute row you want to delete.
2. Clicking the Trashcan icon or pressing the Delete key (see Figure 5-11).

Figure 5-18 Removing a Mapping



	Source	Target
0	[PB-WL]/db/CUSTOMER/STATE	[amtByState.xsd]/customers/STATE/state
1		
2		

Modifying Target Schemas

You can make changes to a target schema by right-clicking an element. A pop-up menu displays available options.

Option	Effect
Expand	Expands to show any hidden child elements.
Properties	Allows you to set or inspect element properties. Depending on the element selected, properties that may be changed include local name, namespace, content type, repeatable, and optional.
Copy	Copies the selected schema element or attribute to the clipboard.

Option	Effect
Paste	Appends the copied element and any children to the selected element. If a copied element contains cloned elements/attributes, the Data View Builder pastes them as regular elements/attributes. Only the hierarchical structure transfers. Notes: • If a pasted element is a duplicate, Data View Builder renames the element as NAME(2), NAME(3) and so on. • The Paste function works only with elements. You cannot paste an element to an attribute. • This menu item is only available when you have data on the clipboard.
Paste and Map	Appends a complex element as a child to the selected element. Properties of the copied source complex elements and its children cannot be changed. Notes: • Sub-elements are shown as mapped and in italics. Any generated query treats complex elements as a unit. • You cannot Paste and Map a complex element of the same name to the same level of the target schema. If you try to do so you will get a "Clones of mapped element types are not allowed" error. Options include deleting or renaming the original complex element in the target schema.
Expand Complex Mappings	Converts an element mapped to a set of individual value mappings. Notes: • After expanding complex mappings you can delete individual mappings using the Trashcan or Delete key. • The only way to Undo this operation is to delete the mappings and Paste and Map again.
Add Attribute	Allows you to add an attribute to the selected element. Attribute properties include local name, namespace, content type, and optional. By default the name of the new attribute is `new_attribute`. Add Attribute works only on an element.
Add Child	Appends a new element as a child to the selected element. By default the name of the new attribute is `new_attribute`. Add Child works only on an element.

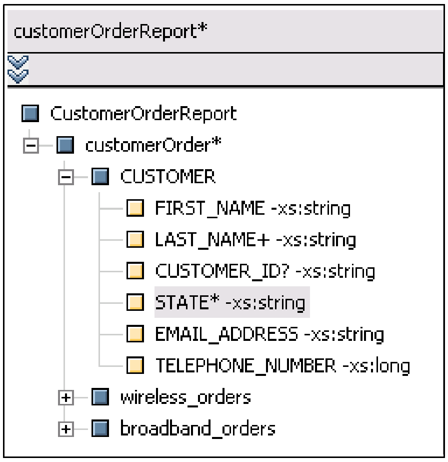
Option	Effect
Add Parent	Creates a new element as a parent of the selected element or attribute. This also increases the nesting level of the selected element. The name of the new element is, by default, `new_element`.
Delete	Removes a selected element/attribute. If the element/attribute to be deleted is mapped, Data View Builder will first display a warning.
Move up	Moves the element/attribute (and children, if any) higher in the list of siblings in the schema tree. An element or attribute can only move up or down among siblings.
Move down	Moves the element/attribute (and children, if any) lower in the list of sibling on the schema tree. An element or attribute can only move up or down among siblings.
Clone	Duplicates the selected element to the same level of the schema hierarchy. Unlike a Copy/Paste operation, cloning does not change your physical schema. You would use cloning if you were, for example, adding a second data source for the same type of information (such as customer orders). The Union example in *Liquid Data by Example* illustrates a use of the clone command.

Managing Target Schema Properties

Liquid Data provides for the setting of combinations of Optional and Repeatable properties on target schema elements. The Data View Builder uses these properties settings to determine the shape of the result set when generating a query.

The following modified version of the customerOrderReport sample schema has FIRST_NAME taking the default condition (no repeat, mandatory), followed by examples of elements with repeatable [+], optional [?], and optional *and* repeatable [*] properties.

Figure 5-19 Various Target Schema Element Attribute Settings



Listing 5-4 shows how these settings are rendered in the generated target XML schema:

Listing 5-4 Various element attribute settings in a generated target schema

```
<xsd:element name="CUSTOMER">
      <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="FIRST_NAME" type="xsd:string"/>
         <xsd:element name="LAST_NAME" type="xsd:string" maxOccurs="unbounded"/>
         <xsd:element name="CUSTOMER_ID" type="xsd:string" minOccurs="0"/>
         <xsd:element name="STATE" type="xsd:string" minOccurs="0"
maxOccurs="unbounded"/>
         <xsd:element name="EMAIL_ADDRESS" type="xsd:string"/>
         <xsd:element name="TELEPHONE_NUMBER" type="xsd:long"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

The following table summarizes the rendering of element properties in the Data View Builder and the generated target schema.

Table 5-20 Rendering of Element Attributes in Data View Builder and Target Schema XML

Symbol	Repeatable?	Optional?	XML Equivalent
[None]	No	No	[None]
+ [plus symbol]	Yes	No	`maxOccurs="unbounded"`
? [question-mark symbol]	No	Yes	`minOccurs="0"`
* [asterisk symbol]	Yes	Yes	`minOccurs="0"` `maxOccurs="unbounded"`

Repeatable Property Settings

When you set a simple or complex element in a target schema to Repeatable (plus [+] or asterisk [*]) it means that the element can repeat within the confines of its enclosed parent in the form:

```
<groupA>
   <item1>
   <item2>
<groupB>
   <item1>
   <item2>
..
```

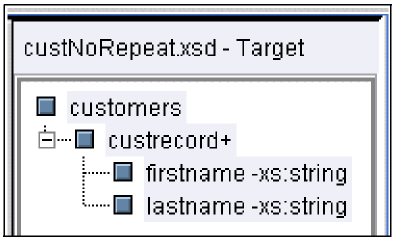
If the Repeatable (+ or *) attribute is not selected, then query results would appear in the form:

```
<groupA>
   <item1>
<groupB>
   <item1>
<groupA>
   <item2>
<groupB>
   <item2>
..
```

Thus the Repeatable element setting is important in maximizing the readability of your query results.

Consider the following target schema:

Figure 5-21 Target schema with a non-repeatable elements

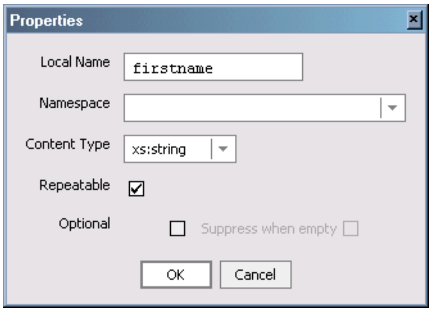


In this target schema, the `firstname` and `lastname` elements are non-repeatable and the `custrecord` element is defined as repeatable and required. If you map data to the `firstname` and `lastname` elements, this target schema will generate results similar to the following:

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

If you modify the target schema so the `firstname` and `lastname` elements are also repeatable (see example in Figure 5-22), the resulting schema for the generated query will be different.

Figure 5-22 Target Schema Properties Dialog with Repeatable Attribute Selected



Local Name: firstname
Namespace:
Content Type: xs:string
Repeatable: ✓
Optional: ☐ Suppress when empty ☐

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

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

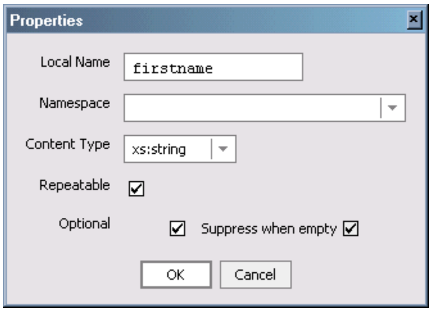
In this case it is likely that the query designer would want the result set to display the first and last names together for the same customer, and would therefore desire the `firstname` and `lastname` elements to be non-repeatable.

Optional Attribute Settings

By default target schema elements are *required*.

If a complex or simple element in a target schema is set to Optional, a question mark [?] or asterisk [*] appears next to its name, meaning that the element can occur *zero or more* times. If the Suppress when empty checkbox is selected (see Figure 5-23), then the element can only occur *one or more times*; in other words, the element will not appear if it has no content.

Figure 5-23 Target Schema Properties Dialog with Suppress When Empty Option Selected



Properties

Local Name: firstname
Namespace:
Content Type: xs:string
Repeatable: ✓
Optional: ✓ Suppress when empty ✓

Listing 5-5 shows an XQuery that is generated with the firstname element set to Optional and Suppress when empty. Not that the `for` loop associated with firstname will affect query efficiency.

Listing 5-5 XQuery returning all BroadBand customers and their Wireless order items, if any. The first_name element is optional and suppressed when empty (code emphasis added)

```
<customers>
 {
 for $PB_BB.CUSTOMER_1 in document("PB-BB")/db/CUSTOMER
 return
 <customer>
   {
   for $firstname_3 in $PB_BB.CUSTOMER_1/FIRSTNAME/text()
   return
   <firstname>{ xf:data($PB_BB.CUSTOMER_1/FIRSTNAME) }</first_name>
   }
   <lastname>{ xf:data($PB_BB.CUSTOMER_1/LASTNAME) }</last_name>
   <orders>
      <order>
         {
         for $PB_WL.CUSTOMER_ORDER_LINE_ITEM_4 in
document("PB-WL")/db/CUSTOMER_ORDER_LINE_ITEM
         where xf:not(xf:empty(
            for $PB_WL.CUSTOMER_ORDER_5 in document("PB-WL")/db/CUSTOMER_ORDER
            where ($PB_BB.CUSTOMER_1/CUSTOMER_ID  eq
$PB_WL.CUSTOMER_ORDER_5/CUSTOMER_ID)
              and ($PB_WL.CUSTOMER_ORDER_5/ORDER_ID  eq
$PB_WL.CUSTOMER_ORDER_LINE_ITEM_4/ORDER_ID)
            return
            xf:true()))
         return
         <line_item product={$PB_WL.CUSTOMER_ORDER_LINE_ITEM_4/PRODUCT_NAME}
expected_ship_date={$PB_WL.CUSTOMER_ORDER_LINE_ITEM_4/EXPECTED_SHIP_DATE} />
         }
      </order>
   </orders>
 </customer>
 }
</customers>
```

Optional Attribute and Data Views

In additional to performance considerations — described in "Building Queries" on page 5-1 — you should use the Optional attribute if you plan to use the target schema as part of a data view. The Optional attribute will prevent an unmapped element from appearing as a data source element in the data view.

Caution: If you attempt to use a data view unmapped element as a source element in a new query, the query will fail with a "not mapped" error.

To understand how these attributes affect the query results, experiment with different property settings, run the queries, and compare the results.

Examples Illustrating How Repeatable and Optional Properties Can be Used to Better Filter Query Results

The following two examples show how the combination of elements and joins can be used to filter out data that does not match the query requirements.

Example Set-up

To set up Data View Builder for the following examples, follow these steps:

1. Create a new project.
2. Move the following relational database schemas into the work area:
   - PB-BB (BroadBand orders RDBMS)
   - PB-WL (Wireless orders RDBMS)
3. Set your target schema to `customerLineItems.xsd`

Example 1: Retrieve All BroadBand Customers, Returning Their Wireless Orders, If Any (ORDER is Repeatable and Optional)

In this case, the target schema is `CUSTOMERS(CUSTOMER*(ORDER*))`. The target schema allows for customers with zero orders. This means that the query returns customers even if they have no orders. Practically, this makes the query a *left outer-join* between customers and orders.

By following these steps you can create this query:

1. Map the following elements from the BroadBand source to the target schema:

Source: [PB-BB]/db/	Target: [customerLineItems.xsd]/customers/
CUSTOMER/**FIRST_NAME**	CUSTOMER/**FIRST_NAME**
CUSTOMER/**LAST_NAME**	CUSTOMER/**LAST_NAME**

2. Map the following elements from the Wireless source to the target schema:

Source: [PB-WL]/db/	Target: [customerLineItems.xsd]/customers/
CUSTOMER_ORDER/**ORDER_DATE**	orders/order/**date**
CUSTOMER_ORDER/**ORDER_ID**	orders/order/**id**

3. Create an equal joins [eq] between the following pair of elements by dragging one element over the other:

Join Element	Join Element
[PB-BB]/db/CUSTOMER/**CUSTOMER_ID**	[PB-WL]/db/CUSTOMER_ORDER/**CUSTOMER_ID**

4. Enter Test mode. You should see a query similar to that shown in Listing 5-6.

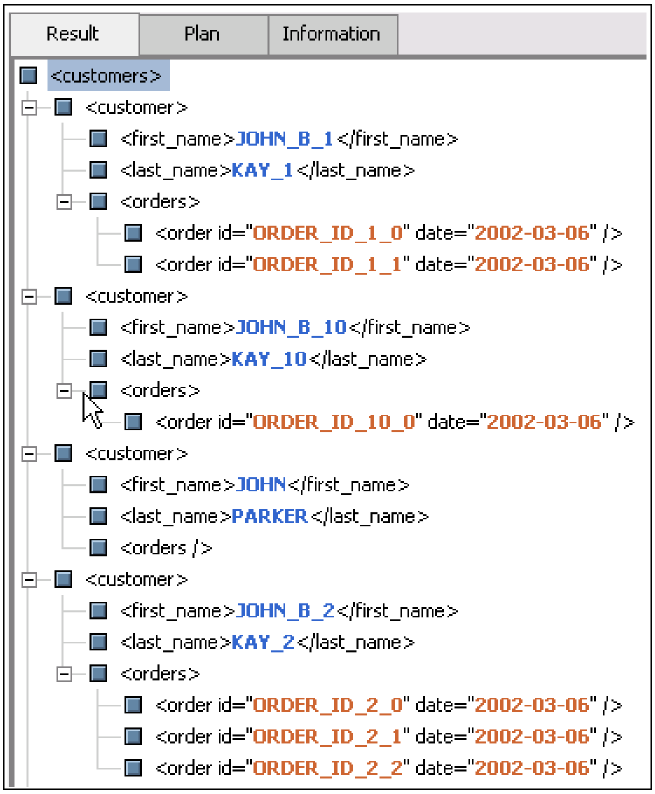
Listing 5-6 Xquery returning all BroadBand customers and returns Wireless orders, if any

```
<customers>
  {
  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>
    <orders>
     {
     for $PB_WL.CUSTOMER_ORDER_6 in document("PB-WL")/db/CUSTOMER_ORDER
     where ($PB_BB.CUSTOMER_1/CUSTOMER_ID  eq
$PB_WL.CUSTOMER_ORDER_6/CUSTOMER_ID)
     return
     <order id={$PB_WL.CUSTOMER_ORDER_6/ORDER_ID}
date={$PB_WL.CUSTOMER_ORDER_6/ORDER_DATE} />
     }
    </orders>
  </customer>
  }
</customers>
```

Results

Notice that the third customer, John Parker, has no orders (Figure 5-24).

Figure 5-24 Example 1: Query Results (First Four Complex Elements Shown)



```
<customers>
  <customer>
    <first_name>JOHN_B_1</first_name>
    <last_name>KAY_1</last_name>
    <orders>
      <order id="ORDER_ID_1_0" date="2002-03-06" />
      <order id="ORDER_ID_1_1" date="2002-03-06" />
  <customer>
    <first_name>JOHN_B_10</first_name>
    <last_name>KAY_10</last_name>
    <orders>
      <order id="ORDER_ID_10_0" date="2002-03-06" />
  <customer>
    <first_name>JOHN</first_name>
    <last_name>PARKER</last_name>
    <orders />
  <customer>
    <first_name>JOHN_B_2</first_name>
    <last_name>KAY_2</last_name>
    <orders>
      <order id="ORDER_ID_2_0" date="2002-03-06" />
      <order id="ORDER_ID_2_1" date="2002-03-06" />
      <order id="ORDER_ID_2_2" date="2002-03-06" />
```

Example 2: Retrieve Only BroadBand Customers Who Have At Least One Wireless Order; Return Their Wireless orders (ORDER Is Repeatable And Required)

In this example the goal is to be able to check for existence of at least one element before you generate the parent. Generation of required repeatable elements is *promoted* 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, the result (list) of the `let` clause is checked to see if it is empty or not before producing the rest of the result.

The ORDER element is required so you need to check for the existence of orders before producing a customer. This means that you need to generate the list of orders for each customer, and output the customer only if this list is not empty.

The only change needed to the target schema used in Example 2-A is to change the order element from:

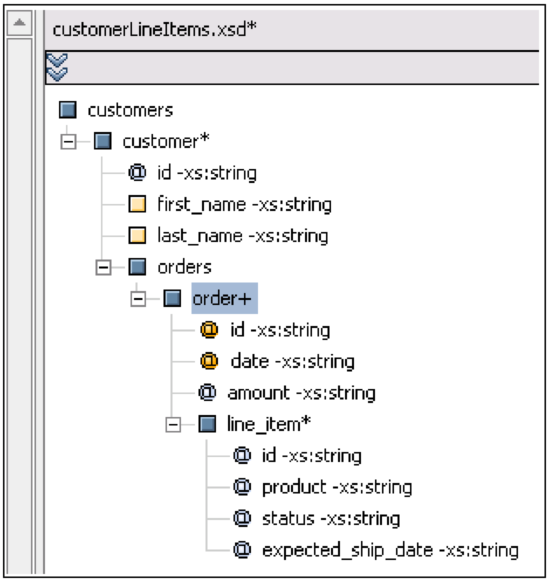
repeatable + optional

to

repeatable + required

To do so right-click on the order element (*below* orders). When the Properties dialog box appears, de-select the Optional checkbox.

Figure 5-25 Orders Element Set to Repeatable and Required



```
customerLineItems.xsd*

customers
  customer*
    id -xs:string
    first_name -xs:string
    last_name -xs:string
    orders
      order+
        id -xs:string
        date -xs:string
        amount -xs:string
        line_item*
          id -xs:string
          product -xs:string
          status -xs:string
          expected_ship_date -xs:string
```

Now your target schema no longer allows for customers with zero orders. This means that the query will not return customers without orders. This makes the query a *natural join* between customers and orders.

When you enter Test mode a query similar to that shown in Listing 5-7 will appear.

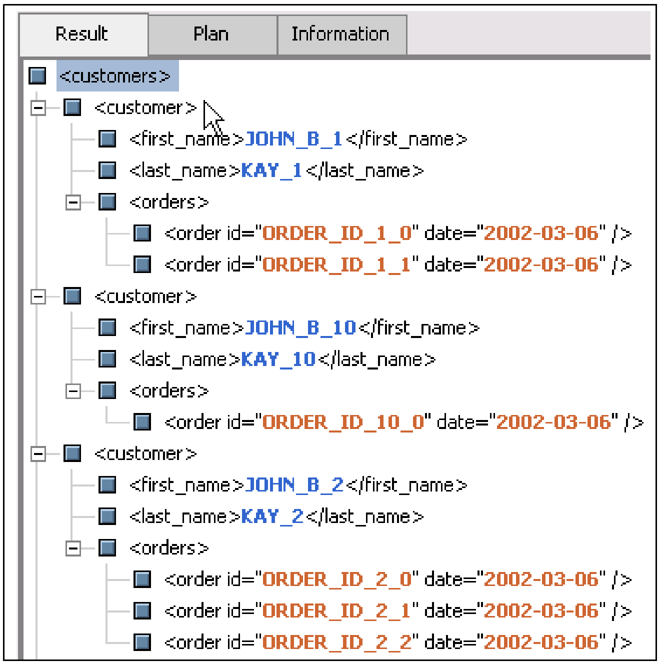
Listing 5-7 XQuery returning only BroadBand customers with at least one Wireless order (emphasis added)

```
<customers>
  {
  for $PB_BB.CUSTOMER_1 in document("PB-BB")/db/CUSTOMER
  let $order_6 :=
                for $PB_WL.CUSTOMER_ORDER_7 in document("PB-WL")/db/CUSTOMER_ORDER
                   where ($PB_BB.CUSTOMER_1/CUSTOMER_ID  eq
$PB_WL.CUSTOMER_ORDER_7/CUSTOMER_ID)
                   return
                   <order id={$PB_WL.CUSTOMER_ORDER_7/ORDER_ID}
date={$PB_WL.CUSTOMER_ORDER_7/ORDER_DATE} />
  where xf:not(xf:empty($order_6))
  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_6 }
    </orders>
  </customer>
  }
</customers>
```

Results

The first four elements returned by the query are shown in Figure 5-26. Since the query filtered out customers without Wireless orders, John Parker no longer appears in the list of customers.

Figure 5-26 Example 2: Query Results (First Three Result Sets Only)



```
<customers>
  <customer>
    <first_name>JOHN_B_1</first_name>
    <last_name>KAY_1</last_name>
    <orders>
      <order id="ORDER_ID_1_0" date="2002-03-06" />
      <order id="ORDER_ID_1_1" date="2002-03-06" />
  <customer>
    <first_name>JOHN_B_10</first_name>
    <last_name>KAY_10</last_name>
    <orders>
      <order id="ORDER_ID_10_0" date="2002-03-06" />
  <customer>
    <first_name>JOHN_B_2</first_name>
    <last_name>KAY_2</last_name>
    <orders>
      <order id="ORDER_ID_2_0" date="2002-03-06" />
      <order id="ORDER_ID_2_1" date="2002-03-06" />
      <order id="ORDER_ID_2_2" date="2002-03-06" />
```

Setting Query Conditions

In the Data View Builder you can define query conditions through XQuery functions in conjunction with constants, query parameters, and custom functions.

You can create *conditions* (or filters) on source data in two ways:

- Drag-and-drop a source element/attribute onto another source element/attribute to build a conditional statement that defines the default [`eq`] (equality) functional relationship between the mapped elements/attributes. (See "Supported Drag-and-Drop Actions in the Data View Builder" on page 5-17.)
- 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 XQuery functions available from Design tab —> Toolbox tab —> Functions panel.

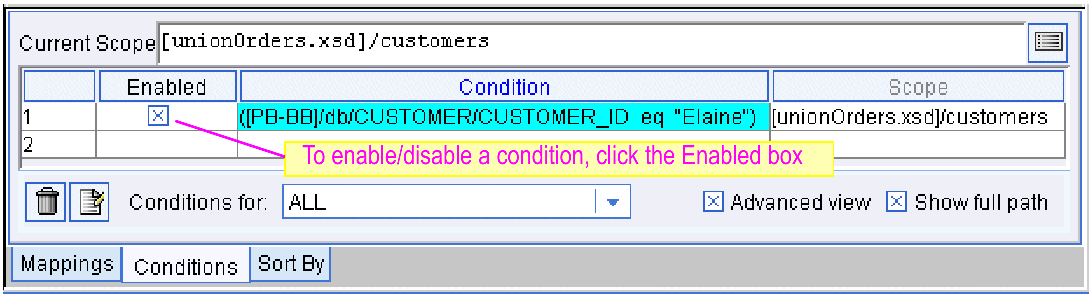
Working With the Conditions Panel

Conditions are displayed in a panel accessed by the Conditions tab (Figure 5-27). It is in the Conditions area that you can view and change query scoping rules (see "Understanding Condition Scoping" on page 5-39.)

Enabling or Disabling Conditions

To enable or disable a condition, click the Enabled box to the left of the Condition (see Figure 5-27.)

Figure 5-27 Enabling or Disabling a Condition



Current Scope [unionOrders.xsd]/customers

	Enabled	Condition	Scope
1	☒	([PB-BB]/db/CUSTOMER/CUSTOMER_ID eq "Elaine")	[unionOrders.xsd]/customers
2			

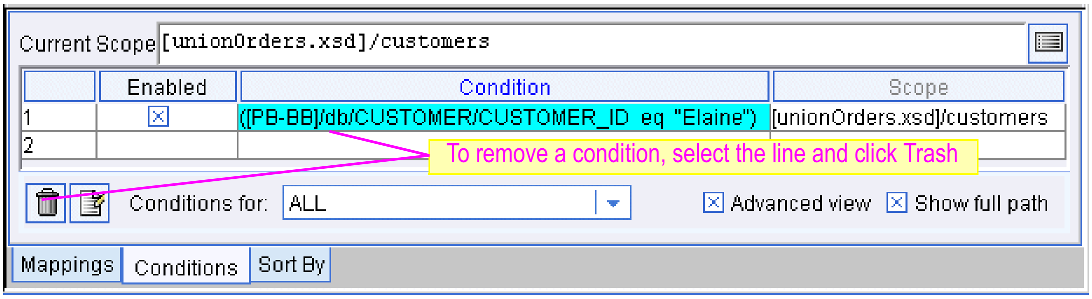
To enable/disable a condition, click the Enabled box

Conditions for: ALL ☒ Advanced view ☒ Show full path

Removing Conditions

Conditions are displayed in the Design view on the Conditions tab. You can remove a condition by selecting the row that contains it and then clicking the Trashcan button or Delete key. (See Figure 5-28.)

Figure 5-28 Removing a Condition



Current Scope [unionOrders.xsd]/customers

	Enabled	Condition	Scope
1	☒	([PB-BB]/db/CUSTOMER/CUSTOMER_ID eq "Elaine")	[unionOrders.xsd]/customers
2			

To remove a condition, select the line and click Trash

Conditions for: ALL ☒ Advanced view ☒ Show full path

Editing Conditions

To add or delete a condition 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 Trashcan, Delete key, or use the Cut, Copy, Paste options on the Edit menu to modify the condition statement.

For additional information see "Using the Function Editor" on page 5-5.

Understanding Condition Scoping

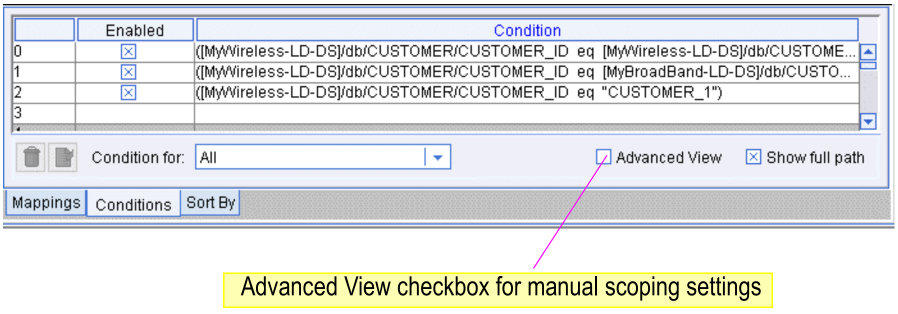
When you add a condition to a query, the Data View Builder makes a "best guess" as to the parts of the target schema to which the condition applies. This is known as *automatic condition scoping* or *autoscope*, and is determined by:

- Structure of the target schema
- Mappings from source schemas to the target schema
- The conditions themselves

Autoscope should be sufficient for most cases. However, there may be situations in which you want to control condition scoping explicitly. In such cases, you should switch to manual scoping by clicking the checkbox next to Advanced view in the Conditions panel (Figure 5-29).

In Advanced view you can explicitly control the extent that a particular condition applies to the result. For example, you can set scope manually in order to specify which part of a data view is the focal point for a particular condition in the query.

Figure 5-29 Conditions Tab in Basic View



	Enabled	Condition
0	☒	([MyWireless-LD-DS]/db/CUSTOMER/CUSTOMER_ID eq [MyWireless-LD-DS]/db/CUSTOME...
1	☒	([MyWireless-LD-DS]/db/CUSTOMER/CUSTOMER_ID eq [MyBroadBand-LD-DS]/db/CUSTO...
2	☒	([MyWireless-LD-DS]/db/CUSTOMER/CUSTOMER_ID eq "CUSTOMER_1")
3		

Condition for: All

☐ Advanced View ☒ Show full path

The following sub-topics are discussed in this section:

- Where Scope Applies
- Setting Condition Scope
- Scoping Example

Where Scope Applies

There are three areas where conditions can be scoped:

- Repeatable elements in the target schema
- Repeatable input elements in functions
- Root of the target schema

Note: A repeatable element is identified with either an asterisk [*] or plus [+] sign. (See "Managing Target Schema Properties" on page 5-26.)

Setting Condition Scope

A common case involving scoping issues occurs when a condition logically applies in two places, but you only want it to appear in one place. You may first notice this when examining the XQuery *where* clauses or when running the query.

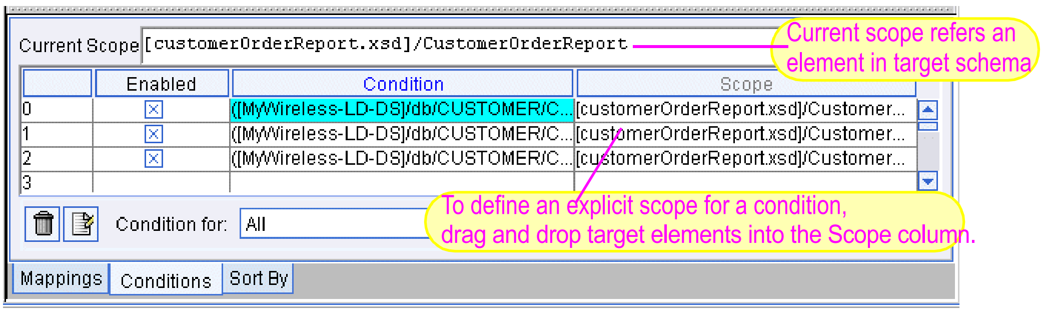
A less common case occurs when you want to create an *assertion*. For example, you may want to devise your query so that the Liquid Data Server returns a result only when a certain condition occurs. You can accomplish this if you switch to the Advanced view, create the condition, and set the scope for the condition to be the root of the target schema.

Advanced View (Setting Condition Scope Manually)

When Advanced view is selected, the Conditions tab expands to show scoping information. The initial display corresponds to the autoscope setting provided by the Data View Builder.

As an example of Advanced view scope setting in Figure 5-30 the first line (line 0) is selected. The current scoping for that line appears near the top of the Conditions pane: (`[customerOrderReport.xsd]/CustomerOrderReport`).

Figure 5-30 Conditions Tab in Advanced View Showing Explicit Scope


Current scope refers an element in target schema
To define an explicit scope for a condition, drag and drop target elements into the Scope column.
Condition for: All
Mappings
Conditions
Sort By

Current Scope `[customerOrderReport.xsd]/CustomerOrderReport`

	Enabled	Condition	Scope
0	☒	([MyWireless-LD-DS]/db/CUSTOMER/C...	[customerOrderReport.xsd]/Customer...
1	☒	([MyWireless-LD-DS]/db/CUSTOMER/C...	[customerOrderReport.xsd]/Customer...
2	☒	([MyWireless-LD-DS]/db/CUSTOMER/C...	[customerOrderReport.xsd]/Customer...
3			

Note: When you switch to Advanced view, it is unnecessary to change any of the explicit scope settings. However, if you add new conditions when in Advanced view, or change existing conditions, you need to manually set the scope for each query condition.

Here are some things to keep in mind when manually setting scope using Advanced view:

- When switching to Advanced view the Current Scope settings initially show the target schema root. Before you map schema sections and create conditions, you can drag a repeatable target schema or function input element to the Current 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.

  You can also drag the appropriate repeatable element in the target schema to the Scope column of a particular row. This permits you to refine your query by narrowing where the condition applies.

- The Enabled column contains a switch to include or exclude a condition when the query runs. This operates the same whether in autoscope or manual mode.

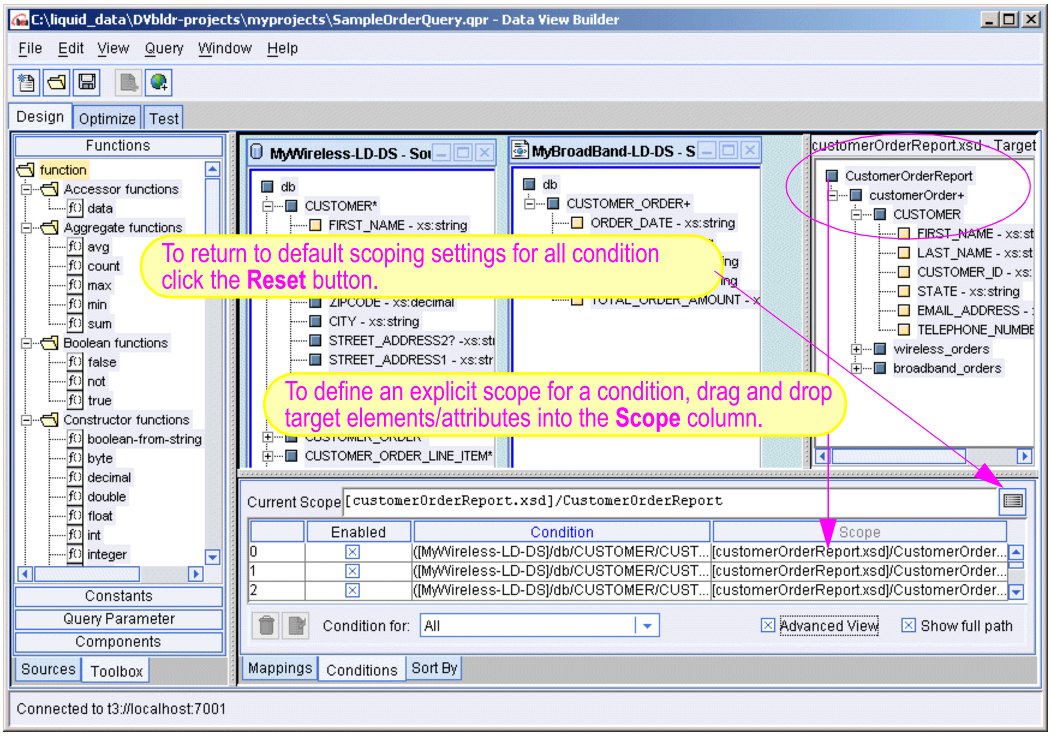
- The Condition column shows the source element, condition, and condition target element. This information is the same whether in autoscope or manual mode.

- The Scope column shows which target schema element Liquid Data will use to focus the result. You can also drag a repeatable target schema element directly to a cell in the Scope column to change the scope for that condition.
- The Reset button in the upper right area of the Conditions panel (Figure 5-31) recalculates all scope settings and returns them to the autoscope settings selected by Liquid Data.

When you explicitly define scope you are forcing the XQuery `where` clause to a specific place in the query or, perhaps, forcing it to be there at all.

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

Figure 5-31 Advanced View



To return to default scoping settings for all condition click the **Reset** button.

To define an explicit scope for a condition, drag and drop target elements/attributes into the **Scope** column.

Current Scope `[customerOrderReport.xsd]/CustomerOrderReport`

	Enabled	Condition	Scope
0	☒	([MyWireless-LD-DS]/db/CUSTOMER/CUST...	[customerOrderReport.xsd]/CustomerOrder...
1	☒	([MyWireless-LD-DS]/db/CUSTOMER/CUST...	[customerOrderReport.xsd]/CustomerOrder...
2	☒	([MyWireless-LD-DS]/db/CUSTOMER/CUST...	[customerOrderReport.xsd]/CustomerOrder...

Condition for: All ☒ Advanced View ☒ Show full path

The Current Scope field shows the default scope setting for every condition that you add. If you add a new condition in Advanced view, the default scope is the target schema root until you change that value.

Returning to Autoscope

Caution: When you toggle Advanced view *off*, Data View Builder returns to autoscope mode. Any changes you made in Advanced view mode are lost and the Current Scope field and Targets column disappear. You will see an alert to this effect when deselecting Advanced view.

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:

1. Select the `xf:count` function and map a source element to be the input of `xf:count`.
2. Create a condition that uses the output of the `xf:count` function.
3. In 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 in autoscope mode. However, if you set scope manually, it is possible such a circular dependency can happen. Data View Builder cancels the action and generates the error message:

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

Recommended Action

If the recursion error message appears, consider resetting all condition scope targets using the Reset button (see Figure 5-31). Or override the automatic settings one at a time, switch to Test view to examine the query, run it, and assess the results.

Scoping Example

This section contains an example illustrating uses of manual scoping.

Resolving Extraneous Joins Through Advanced View Manual Scoping

Advanced View can be used to resolve ambiguous joins.

If you want to create a query that divides products into two groups based on their list price you would create two conditions:

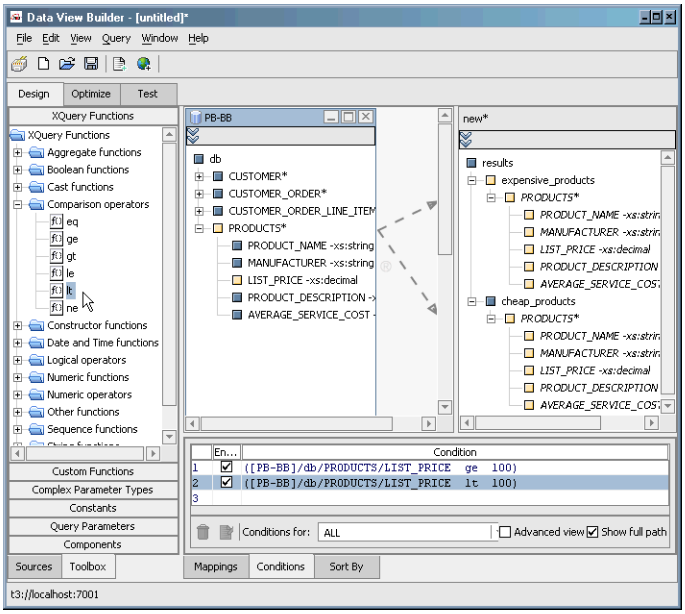
list_price greater-than-or-equal-to $100 *list_price less-than $100*

Obviously, if both these conditions are applied to the same set of data no data will be returned.

One way to resolve this is to create a second version of the data source schema (see "Using Source Schemas Multiple Times in Constructing Queries" on page 4-4 for an example of this approach). However Advanced Scoping can be used to the same effect, as the following steps show:

1. Open the Data View Builder and drag the relational data source PB-BB onto the desktop. Expand the data source.
2. Right-click on the `new` element in the target schema area; select Add Child.
3. Enter `expensive_products` as the local name; click Ok.
4. Repeat Steps 2 and 3, using `cheap_products` as the local name.
5. Back in the data source right-click on the `Products` complex element; choose Copy.
6. In the target schema area, right-click on `expensive_products`; choose Paste and Map. The `PRODUCTS` complex element will be projected to the target schema under `expensive_products`.
7. Right-click on `cheap_products`; choose Paste and Map. The `cheap_products` complex element is similarly associated with data.

Figure 5-32 List Price Comparison Project



	En...	Condition
1	✓	([PB-BB]/db/PRODUCTS/LIST_PRICE ge 100)
2	✓	([PB-BB]/db/PRODUCTS/LIST_PRICE lt 100)
3		

Once you create the project, you need to set your query conditions:

1. Click on the Conditions tab below the Data View Builder work area.
2. Click on XQuery Functions. Under Comparison Operators locate the greater-than-or-equal-to `[ge]` function and drag it to the first Conditions line.
3. When the Functions editor appears, map the `LIST_PRICE` element and `100` constant as shown below. Then click Close.

Source	Target: Comparison Function
`[PB-BB]/db/PRODUCTS/`**`LIST_PRICE`**	`[ge]/`**`anyValue1`**
`[CONSTANT]/`**`100`**	`[ge]/`**`anyValue2`**

4. From the Comparison Operator function list locate the less-than `[lt]` function and drag it to the first empty Conditions line.

5. In the Functions editor map the `LIST_PRICE` element and `100` constant appropriately. Then click Close.

Source	Target: Comparison Function
`[PB-BB]/db/PRODUCTS/`**`LIST_PRICE`**	`[lt]/`**`anyValue1`**
`[CONSTANT]/`**100**	`[lt]/`**`anyValue2`**

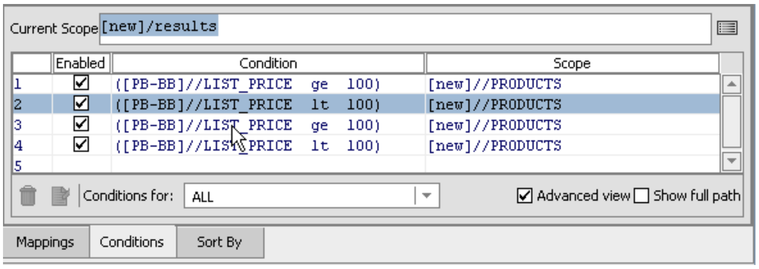
If you just Test the query (Listing 5-8) at this point no results will appear, as expected, since nothing can fulfill both `where` clause conditions.

Listing 5-8 List Price XQuery illustrating self-cancelling conditions (emphasis added)

```
<results>
   <expensive_products>
       {
       for $PB_BB.PRODUCTS_15 in document("PB-BB")/db/PRODUCTS
       where ($PB_BB.PRODUCTS_15/LIST_PRICE  ge  100)
         and ($PB_BB.PRODUCTS_15/LIST_PRICE  lt  100)
       return
       $PB_BB.PRODUCTS_15
       }
   </expensive_products>
   <cheap_products>
       {
       for $PB_BB.PRODUCTS_21 in document("PB-BB")/db/PRODUCTS
       where ($PB_BB.PRODUCTS_21/LIST_PRICE  ge  100)
         and ($PB_BB.PRODUCTS_21/LIST_PRICE  lt  100)
       return
       $PB_BB.PRODUCTS_21
       }
   </cheap_products>
</results>
```

6. To resolve this problem click Advanced view in the Conditions section. You will notice that instead of the two conditions you created, four are listed. This is because Advanced view shows you the actual `where` clause conditions used in the query, based on application of the Data View Builder best-guess autoscope rules.

Figure 5-33 Advanced View of Conditions in List Price Project



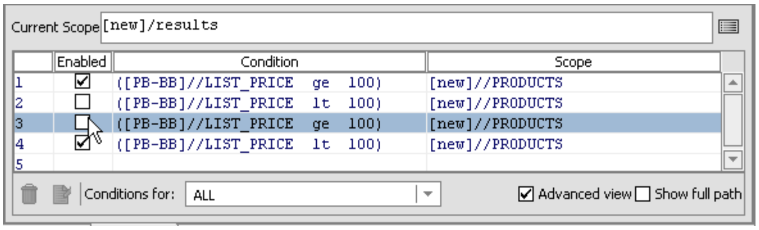
Current Scope [new]/results

	Enabled	Condition	Scope
1	☑	([PB-BB]//LIST_PRICE ge 100)	[new]//PRODUCTS
2	☑	([PB-BB]//LIST_PRICE lt 100)	[new]//PRODUCTS
3	☑	([PB-BB]//LIST_PRICE ge 100)	[new]//PRODUCTS
4	☑	([PB-BB]//LIST_PRICE lt 100)	[new]//PRODUCTS
5			

Conditions for: ALL ☑ Advanced view ☐ Show full path

If you disable the inappropriate conditions (Figure 5-34), Advanced View will appear as we expected it should, with a single WHERE condition for each section of the query.

Figure 5-34 Advanced View With Two Conditions Disabled



Current Scope [new]/results

	Enabled	Condition	Scope
1	☑	([PB-BB]//LIST_PRICE ge 100)	[new]//PRODUCTS
2	☐	([PB-BB]//LIST_PRICE lt 100)	[new]//PRODUCTS
3	☐	([PB-BB]//LIST_PRICE ge 100)	[new]//PRODUCTS
4	☑	([PB-BB]//LIST_PRICE lt 100)	[new]//PRODUCTS
5			

Conditions for: ALL ☑ Advanced view ☐ Show full path

The newly generated XQuery is correct (Listing 5-9).

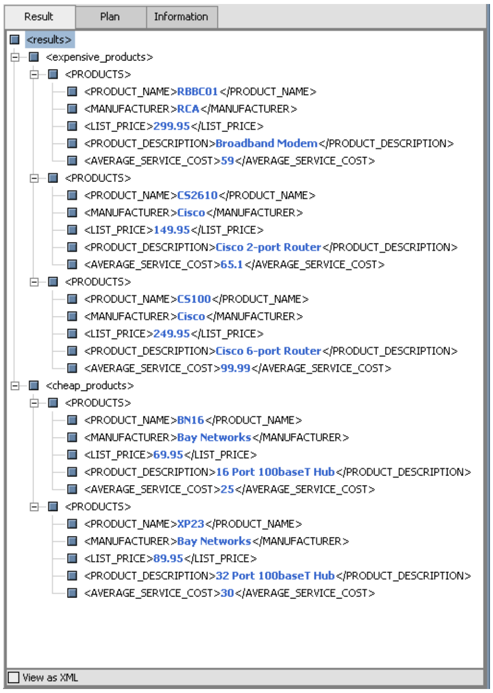
Listing 5-9 List Price XQuery after extraneous conditions are disabled

```
<results>
   <expensive_products>
      {
      for $PB_BB.PRODUCTS_15 in document("PB-BB")/db/PRODUCTS
      where ($PB_BB.PRODUCTS_15/LIST_PRICE  ge  100)
      return
      $PB_BB.PRODUCTS_15
      }
   </expensive_products>
```

```
    <cheap_products>
      {
      for $PB_BB.PRODUCTS_21 in document("PB-BB")/db/PRODUCTS
      where ($PB_BB.PRODUCTS_21/LIST_PRICE  lt   100)
      return
      $PB_BB.PRODUCTS_21
      }
    </cheap_products>
</results>
```

And the results (Figure 5-35) conform with the query.

Figure 5-35 List Price XQuery Results Show Three “Expensive” Products and Two “Cheap” Products

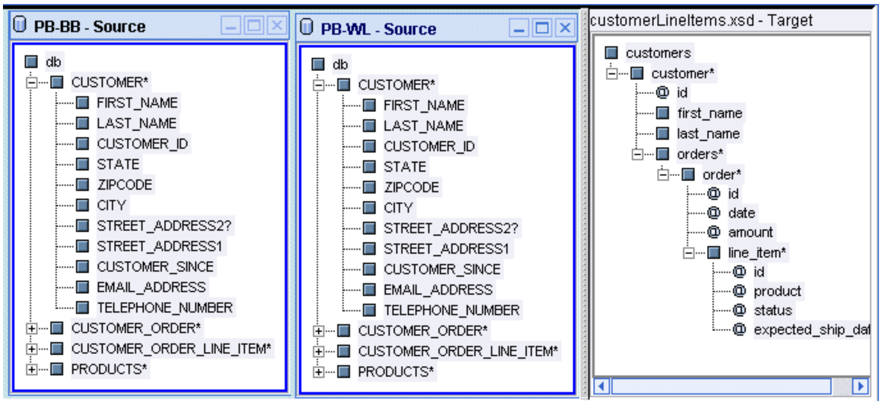


```
<results>
  <expensive_products>
    <PRODUCTS>
      <PRODUCT_NAME>RBBC01</PRODUCT_NAME>
      <MANUFACTURER>RCA</MANUFACTURER>
      <LIST_PRICE>299.95</LIST_PRICE>
      <PRODUCT_DESCRIPTION>Broadband Modem</PRODUCT_DESCRIPTION>
      <AVERAGE_SERVICE_COST>59</AVERAGE_SERVICE_COST>
    <PRODUCTS>
      <PRODUCT_NAME>CS2610</PRODUCT_NAME>
      <MANUFACTURER>Cisco</MANUFACTURER>
      <LIST_PRICE>149.95</LIST_PRICE>
      <PRODUCT_DESCRIPTION>Cisco 2-port Router</PRODUCT_DESCRIPTION>
      <AVERAGE_SERVICE_COST>65.1</AVERAGE_SERVICE_COST>
    <PRODUCTS>
      <PRODUCT_NAME>CS100</PRODUCT_NAME>
      <MANUFACTURER>Cisco</MANUFACTURER>
      <LIST_PRICE>249.95</LIST_PRICE>
      <PRODUCT_DESCRIPTION>Cisco 6-port Router</PRODUCT_DESCRIPTION>
      <AVERAGE_SERVICE_COST>99.99</AVERAGE_SERVICE_COST>
  <cheap_products>
    <PRODUCTS>
      <PRODUCT_NAME>BN16</PRODUCT_NAME>
      <MANUFACTURER>Bay Networks</MANUFACTURER>
      <LIST_PRICE>69.95</LIST_PRICE>
      <PRODUCT_DESCRIPTION>16 Port 100baseT Hub</PRODUCT_DESCRIPTION>
      <AVERAGE_SERVICE_COST>25</AVERAGE_SERVICE_COST>
    <PRODUCTS>
      <PRODUCT_NAME>XP23</PRODUCT_NAME>
      <MANUFACTURER>Bay Networks</MANUFACTURER>
      <LIST_PRICE>89.95</LIST_PRICE>
      <PRODUCT_DESCRIPTION>32 Port 100baseT Hub</PRODUCT_DESCRIPTION>
      <AVERAGE_SERVICE_COST>30</AVERAGE_SERVICE_COST>
```

Task Flow Model for Advanced View Manual Scoping

If you decide to override automatic scope settings, there is a workflow model that can 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 project shown in Figure 5-36 which has two source schemas: PB-BB and PB-WL, and the target schema `customerLineItems.xsd`.

Figure 5-36 Schemas for Manual Scope Example



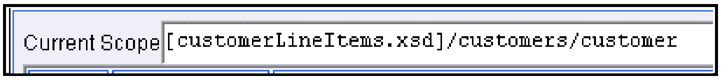
The target schema, `customerLineItems.xsd`, has a hierarchical structure. There are three distinct sections in the schema that represent repeatable data. Elements `customer` and `order` each have an asterisk [`*`] as the occurrence indicator. The element `line_item` has a plus sign [`+`] as its occurrence 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 the 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 you want to collect `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.)



[customerLineItems.xsd]/customers/customer

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`

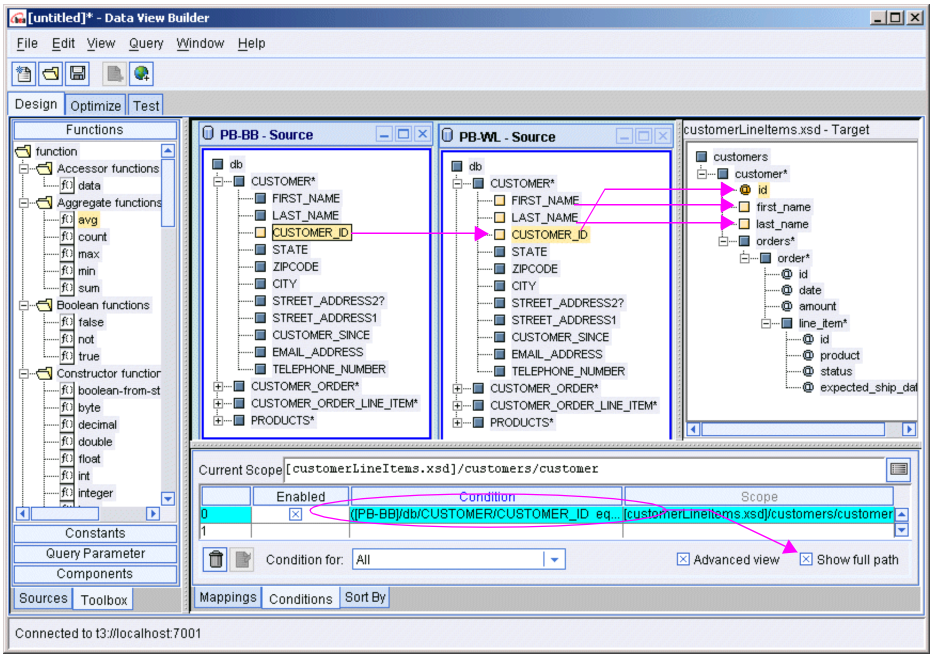
4. Set any conditions that connect and filter the mapped sources.

   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 this example, you set as a condition a join between `CUSTOMER_ID` in the PB-BB schema and `CUSTOMER_ID` in the PB-WL schema (Figure 5-37).

Figure 5-37 Project Showing Join on CUSTOMER_ID



Current Scope [customerLineItems.xsd]/customers/customer

	Enabled	Condition	Scope
0	☒	([PB-BB]/db/CUSTOMER/CUSTOMER_ID eq...	[customerLineItems.xsd]/customers/customer
1			

Condition for: All

☒ Advanced view ☒ Show full path

Connected to t3://localhost:7001

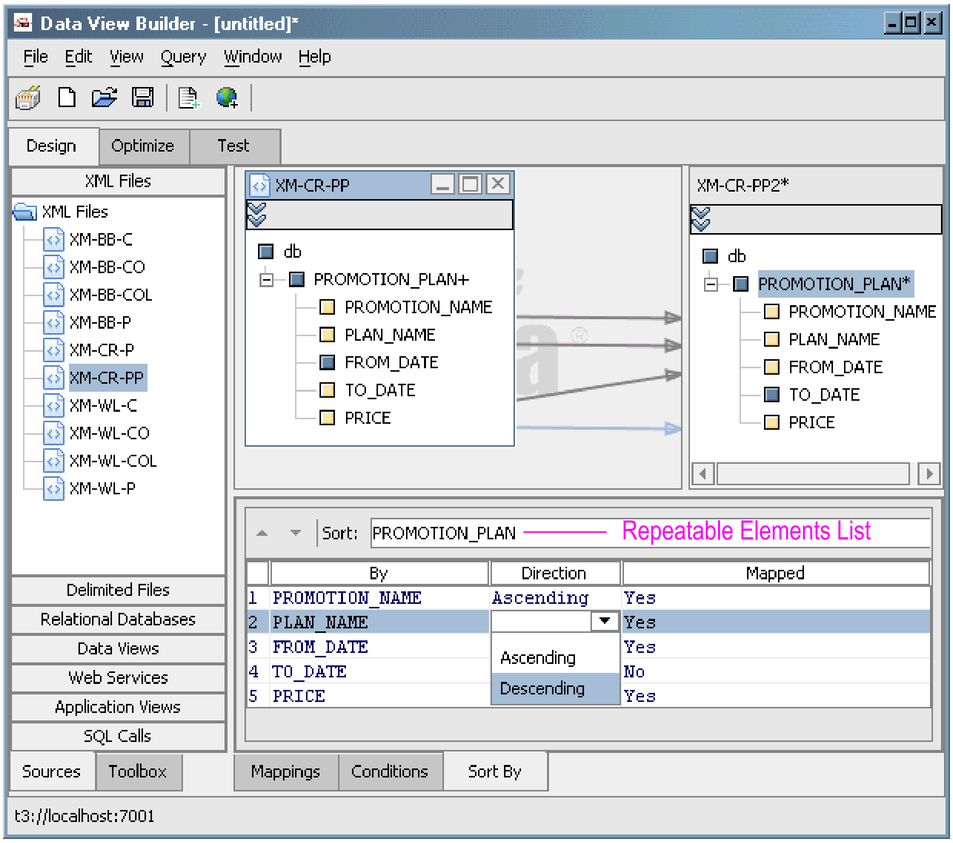
5. Repeat these steps for each section of the target schema where you want data to appear in the result. Work on one section at a time and work from the outside (more general) to the inside (most specific). Ensure that you set the default target, map, and define the conditions, before you move to the 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.

Sorting Query Results

The Sort By tab allows you to specify how query results should be ordered. The screen shot of the Sort By Tab Dialog Box (Figure 5-38) contains a single data source with a repeatable and optional complex element called PROMOTION_PLAN.

Figure 5-38 Sort By Tab Dialog Box



Sort: PROMOTION_PLAN

	By	Direction	Mapped
1	PROMOTION_NAME	Ascending	Yes
2	PLAN_NAME		Yes
3	FROM_DATE	Ascending	Yes
4	TO_DATE	Descending	No
5	PRICE		Yes

The Sort By tab allows you to define the output order for any repeatable element, as identified by a plus [+] or asterisk [*] next to its name. An element can be sorted by one or more sub-elements (including itself in the case of a simple element). (You can change an attribute setting of a complex element to repeatable. For details see "Managing Target Schema Properties" on page 5-26.)

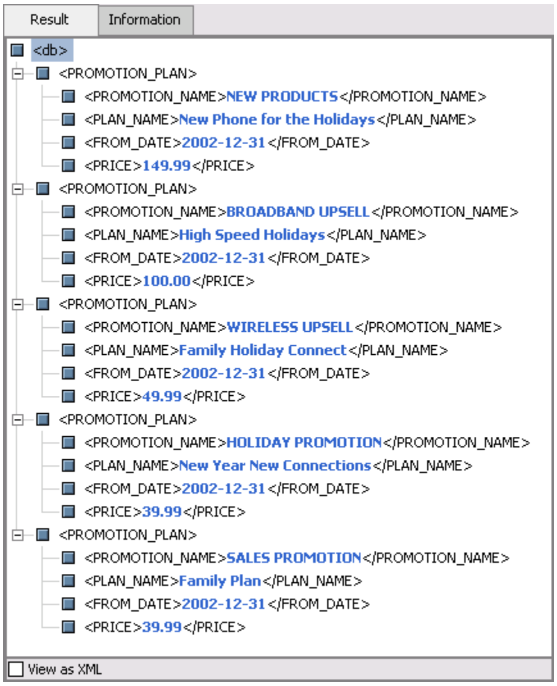
Follow these steps to change sorting order of an element:

1. Select an element from the Sort drop-down list.
2. To specify a sub-element to sort by, select the sub-element from the By column, then set the direction.
3. The relevant sorting order can be modified by selecting a line and using the Up or Down arrows.

In the case of the project shown in Figure 5-38, you are sorting elements in PROMOTION_PLAN first by PROMOTION_NAME and then by PLAN_NAME. The PROMOTION_NAME element will be sorted in ascending order while PLAN_NAME will be Descending.

If you set the topmost sort element to PRICE and the direction to Descending, the result of the query will be ordered appropriately. See Figure 5-39.

Figure 5-39 Results Sorting by Price in Descending Order



```
<db>
  <PROMOTION_PLAN>
    <PROMOTION_NAME>NEW PRODUCTS</PROMOTION_NAME>
    <PLAN_NAME>New Phone for the Holidays</PLAN_NAME>
    <FROM_DATE>2002-12-31</FROM_DATE>
    <PRICE>149.99</PRICE>
  <PROMOTION_PLAN>
    <PROMOTION_NAME>BROADBAND UPSELL</PROMOTION_NAME>
    <PLAN_NAME>High Speed Holidays</PLAN_NAME>
    <FROM_DATE>2002-12-31</FROM_DATE>
    <PRICE>100.00</PRICE>
  <PROMOTION_PLAN>
    <PROMOTION_NAME>WIRELESS UPSELL</PROMOTION_NAME>
    <PLAN_NAME>Family Holiday Connect</PLAN_NAME>
    <FROM_DATE>2002-12-31</FROM_DATE>
    <PRICE>49.99</PRICE>
  <PROMOTION_PLAN>
    <PROMOTION_NAME>HOLIDAY PROMOTION</PROMOTION_NAME>
    <PLAN_NAME>New Year New Connections</PLAN_NAME>
    <FROM_DATE>2002-12-31</FROM_DATE>
    <PRICE>39.99</PRICE>
  <PROMOTION_PLAN>
    <PROMOTION_NAME>SALES PROMOTION</PROMOTION_NAME>
    <PLAN_NAME>Family Plan</PLAN_NAME>
    <FROM_DATE>2002-12-31</FROM_DATE>
    <PRICE>39.99</PRICE>
```

Using Existential Condition Checking in Queries

An existential condition tests for the existence of an underlying data relationship that fits specific criteria.

The Data View Builder offers an option that potentially introduces additional existential conditions in a XQuery. This condition or conditions can be used to further filter query results such as eliminating duplicates being returned by a query. Because extra processing is involved, adding existential conditions can impact query performance.

To activate the option select Allow Existential Condition Generation from the Query menu. A checkmark next to the option indicates that it is active.

The following pseudocode shows an existential condition test. The *where-for* routine will return an `xf:true()` if the enclosed conditions are fulfilled and execution will proceed. If the conditions are not fulfilled, the `return data` will not be executed.

```
 ...

    where xf:not(xf:empty(
      for ...
         where ...
      return
      xf:true()))
return
  data
```

An Existential Example

The following example illustrates a case where the existential condition generation option affects query results.

To set up Data View Builder, follow these steps:

1. Create a new project.
2. Move the following relational database schemas into the work area:
   – PB-BB (BroadBand orders RDBMS)
   – PB-WL (Wireless orders RDBMS)
3. Set your target schema to `customerLineItems.xsd`
4. Map the following elements from the BroadBand source to the target schema:

Source: [PB-BB]/db/	**Target: [customerLineItems.xsd]/customers/**
CUSTOMER/**LAST_NAME**	CUSTOMER/**last_name**
CUSTOMER/**CUSTOMER_ID**	CUSTOMER/**id**

5. Map the following elements from the Wireless source to the target schema:

Source: [PB-WL]/db/	Target: [customerLineItems.xsd]/customers
PRODUCTS/**PRODUCT_NAME**	orders/order/line_item/**product**
CUSTOMER/**CUSTOMER_ID**	CUSTOMER/**id**

6. To ensure referential integrity create joins between the following pairs of elements by dragging one element over the other:

Join Element	Join Element
[PB-BB]/db/CUSTOMER/**CUSTOMER_ID**	[PB-WL]/db/CUSTOMER/**CUSTOMER_ID**
[PB-WL]/db/CUSTOMER/**CUSTOMER_ID**	[PB-WL]/db/CUSTOMER_ORDER/**CUSTOMER_ID**
[PB-BB]/db/CUSTOMER_ORDER/**ORDER_ID**	[PB-WL]/db/CUSTOMER_ORDER_LINE_ITEM/**ORDER_ID**
[PB-BB]/db/CUSTOMER_ORDER_LINE_ITEM/**PRODUCT_NAME**	[PB-WL]/db/PRODUCTS/**PRODUCT_NAME**

Note that no `order_id` is projected to the target schema. Therefore all products ordered by a particular customer will be returned in a single group. The generated query makes this relationship clear.

Listing 5-10 Example XQuery With Existential Condition Generation Off (Default Condition)

```
<customers>
   {
   for $PB_BB.CUSTOMER_1 in document("PB-BB")/db/CUSTOMER
   return
   <customer id={$PB_BB.CUSTOMER_1/CUSTOMER_ID}>
      <last_name>{ xf:data($PB_BB.CUSTOMER_1/LAST_NAME) }</last_name>
      <orders>
         <order>
            {
            for $PB_WL.PRODUCTS_4 in document("PB-WL")/db/PRODUCTS
            for $PB_WL.CUSTOMER_5 in document("PB-WL")/db/CUSTOMER
            for $PB_WL.CUSTOMER_ORDER_6 in document("PB-WL")/db/CUSTOMER_ORDER
            for $PB_WL.CUSTOMER_ORDER_LINE_ITEM_7 in
document("PB-WL")/db/CUSTOMER_ORDER_LINE_ITEM
            where ($PB_BB.CUSTOMER_1/CUSTOMER_ID  eq
$PB_WL.CUSTOMER_5/CUSTOMER_ID)
              and ($PB_WL.CUSTOMER_5/CUSTOMER_ID  eq
$PB_WL.CUSTOMER_ORDER_6/CUSTOMER_ID)
              and ($PB_WL.CUSTOMER_ORDER_6/ORDER_ID  eq
$PB_WL.CUSTOMER_ORDER_LINE_ITEM_7/ORDER_ID)
              and ($PB_WL.CUSTOMER_ORDER_LINE_ITEM_7/PRODUCT_NAME  eq
$PB_WL.PRODUCTS_4/PRODUCT_NAME)
            return
            <line_item product={$PB_WL.PRODUCTS_4/PRODUCT_NAME} />
            }
         </order>
      </orders>
   </customer>
   }
</customers>
```

The query in Listing 5-10 returns a list of every item ordered by the particular customer, including duplicates, if any. (See Listing 5-11.)

Listing 5-11 Query Results for CUSTOMER_1 With Existential Condition Checking Inactive

```
<customers>
  <customer id="CUSTOMER_1">
    <last_name>KAY_1</last_name>
    <orders>
      <order>
        <line_item product="E110"/>
        <line_item product="E110"/>
        <line_item product="E900"/>
        <line_item product="E900"/>
        <line_item product="NOK9250"/>
        <line_item product="NOK9250"/>
        <line_item product="S625"/>
        <line_item product="S625"/>
        <line_item product="SS8"/>
        <line_item product="SS8"/>
      </order>
```

```
        </orders>
    </customer>
...
</customers
```

When the Allow Existential Condition Generation option is active, a `where xf:not(xf:empty)` condition is applied that effectively filters out the return of duplicate order items. The resulting query is shown in Figure 5-12.

Listing 5-12 Example XQuery With Existential Condition Generation On (emphasis added)

```
<customers>
  {
  for $PB_BB.CUSTOMER_1 in document("PB-BB")/db/CUSTOMER
  return
  <customer id={$PB_BB.CUSTOMER_1/CUSTOMER_ID}>
    <last_name>{ xf:data($PB_BB.CUSTOMER_1/LAST_NAME) }</last_name>
    <orders>
      <order>
        {
        for $PB_WL.PRODUCTS_4 in document("PB-WL")/db/PRODUCTS
        where xf:not(xf:empty(
          for $PB_WL.CUSTOMER_5 in document("PB-WL")/db/CUSTOMER
          for $PB_WL.CUSTOMER_ORDER_6 in document("PB-WL")/db/CUSTOMER_ORDER
          for $PB_WL.CUSTOMER_ORDER_LINE_ITEM_7 in
document("PB-WL")/db/CUSTOMER_ORDER_LINE_ITEM
          where ($PB_BB.CUSTOMER_1/CUSTOMER_ID  eq
$PB_WL.CUSTOMER_5/CUSTOMER_ID)
             and ($PB_WL.CUSTOMER_5/CUSTOMER_ID  eq
$PB_WL.CUSTOMER_ORDER_6/CUSTOMER_ID)
             and ($PB_WL.CUSTOMER_ORDER_6/ORDER_ID  eq
$PB_WL.CUSTOMER_ORDER_LINE_ITEM_7/ORDER_ID)
             and ($PB_WL.CUSTOMER_ORDER_LINE_ITEM_7/PRODUCT_NAME  eq
$PB_WL.PRODUCTS_4/PRODUCT_NAME)
          return
          xf:true()))
      return
      <line_item product={$PB_WL.PRODUCTS_4/PRODUCT_NAME} />
      }
     </order>
    </orders>
   </customer>
   }
  </customers>
```

As noted above, both results are valid. For performance reasons it is recommended that where appropriate queries be run without the additional existential condition generation. In many cases duplicate results reporting may be sought or acceptable. In other cases the underlying data may make such existential condition checks unnecessary.

Listing 5-13 Query Results for CUSTOMER_1 With Existential Condition Checking Active

```
<customers>
  <customer id="CUSTOMER_1">
    <last_name>KAY_1</last_name>
    <orders>
      <order>
        <line_item product="E110"/>
        <line_item product="E900"/>
        <line_item product="NOK9250"/>
        <line_item product="S625"/>
        <line_item product="SS8"/>
      </order>
    </orders>
  </customer>
..
</customers>
```

Note: Opening a project saved with Liquid Data 8.1 SP1 or earlier will make the Allow Existential Condition Generation active in order to preserve backward compatibility.

Using Automatic Type Casting

Automatic type casting helps ensure that input parameters used in functions and mappings are appropriate to the function in which they are used.

Select Automatic Type Casting on the Query menu to ensure that Liquid Data will assign (cast) a new data type when:

1. The source element data type does not match the mapped target element data type and
2. The source element is eligible to be type cast to the target element data type.

An checkmark 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.

Table 5-40 Numeric Data Type Promotions

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

If the type mismatch requires casting in reverse order, the Liquid Data Server does not attempt type casting. In this case, the Data View Builder attempts to type cast but the results may be unpredictable.

An 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. However, if the input function type is `xs:double` or `xs:float` and the required input type is *xs:integer* or *xs:byte*, the Data View Builder tries to cast, but there may be unpredictable rounding or truncating of the result. 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.

Automatic Type Casting Transformations

This section provides specifics on how the Data View Builder implements data type transformation for automatic type casting. The following topics are included:

- Automatic Type Casting to a Numeric Target
- Automatic Type Casting to a Non-Numeric Target
- Automatic Type Casting Function Parameters

You can use the information in the following sections to predict automatic type casting behavior.

Automatic Type Casting to a Numeric Target

The following table shows whether Liquid Data transforms a source element data type to the numeric data type of the target element.

	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

	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: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

Automatic Type Casting to a Non-Numeric Target

The following table shows whether Liquid Data transforms a source element data type to the non-numeric data type of the target element.

	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

	Target: xs:byte	Target: xs:boolean	Target: xs:date	Target: xs:time	Target: xs:dateTime	Target: xsext:anyValue xsext:anyType xsext:item
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`.

Automatic 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

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 or to target schema elements/attributes that require the following data types:

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

If the source data is not compatible with the data type of the target element, automatic type casting will not improve query results. For example, mapping a date to a numeric type may not produce useful results.

Note: You may not see an error on a type mismatch until the Liquid Data Server tries to run the query.

Running, Saving, and Deploying Queries

This topic describes how to run BEA Liquid Data for WebLogic queries, as well as how to save and deploy queries. The following sections are included:

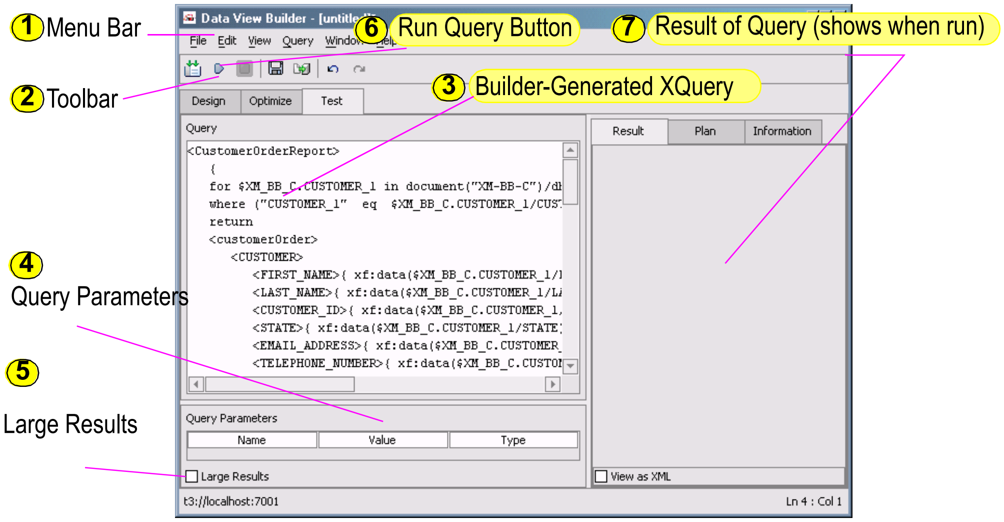
Test Mode

Test mode in the Data View Builder allows you to:

- View the generated XQuery that is based upon your mapping and optimization inputs
- Enter any needed query parameters
- Enter complex parameter type (CPT) reference file names
- View and analyze the query plan
- Run your query against data sources and verify the result
- Get statistical information on the query after it has run
- Experiment with hand editing a generated query or write and test a query from scratch

Click the Test tab to switch to the Test mode.

Figure 6-1 Test Tab



```
<CustomerOrderReport>
   {
   for $XM_BB_C.CUSTOMER_1 in document("XM-BB-C")/dl
   where ("CUSTOMER_1"  eq  $XM_BB_C.CUSTOMER_1/CUST
   return
   <customerOrder>
      <CUSTOMER>
         <FIRST_NAME>{ xf:data($XM_BB_C.CUSTOMER_1/I
         <LAST_NAME>{ xf:data($XM_BB_C.CUSTOMER_1/LA
         <CUSTOMER_ID>{ xf:data($XM_BB_C.CUSTOMER_1,
         <STATE>{ xf:data($XM_BB_C.CUSTOMER_1/STATE]
         <EMAIL_ADDRESS>{ xf:data($XM_BB_C.CUSTOMER_
         <TELEPHONE_NUMBER>{ xf:data($XM_BB_C.CUSTON
```

Viewing a Generated Query

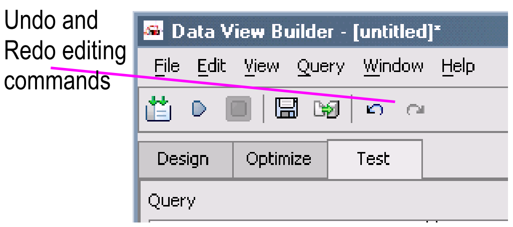
Whenever you enter Test mode an XQuery is automatically created based upon the input provided through the Design and Optimize tabs. (See "Target Schemas" on page 4-7 for details.)

The query appears in the upper-left pane of the Test tab (Figure 6-1). Whenever you save a query, it is saved exactly as it appears.

Editing a Generated Query

Although queries are generally automatically generated, you can make changes to the generated query or create one from scratch. In addition to standard Cut, Copy, and Paste, and Delete editing functions, buffered Undo and Redo editing commands are available through the Test tab toolbar (Figure 6-2).

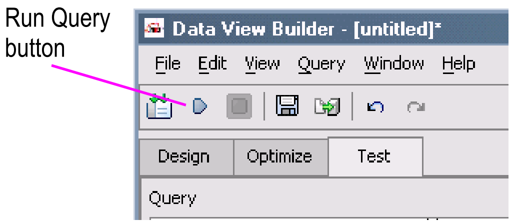
Figure 6-2 Click the Run Query Button to Run the Query


Undo and Redo editing commands
Data View Builder - [untitled]*
File Edit View Query Window Help
Design Optimize Test
Query

Running a Query

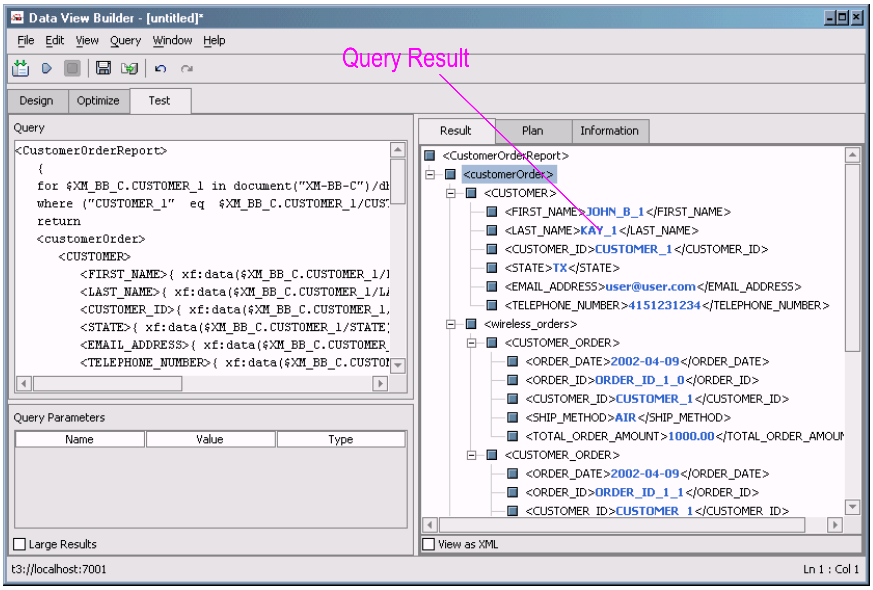
To run your query, click the Run Query button on the Toolbar or select Run Query from the Query menu.

Figure 6-3 Run Query Button


Run Query button
Data View Builder - [untitled]*
File Edit View Query Window Help
Design
Optimize
Test
Query

Queries are run against available data sources made available through a Liquid Data Server. Results appear in the Results panel as an XML tree (Figure 6-3).

Figure 6-4 Expanded Query Results

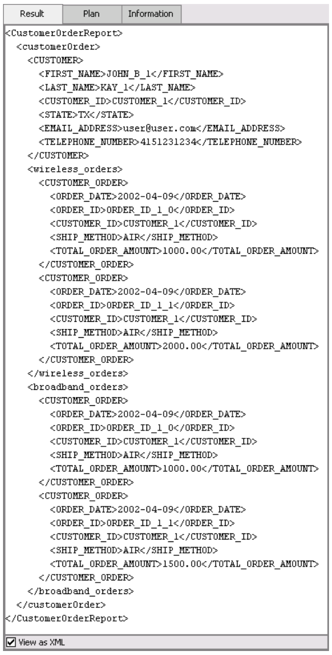


```
<CustomerOrderReport>
   {
   for $XM_BB_C.CUSTOMER_1 in document("XM-BB-C")/db
   where ("CUSTOMER_1"  eq  $XM_BB_C.CUSTOMER_1/CUST
   return
   <customerOrder>
      <CUSTOMER>
         <FIRST_NAME>{ xf:data($XM_BB_C.CUSTOMER_1/I
         <LAST_NAME>{ xf:data($XM_BB_C.CUSTOMER_1/LA
         <CUSTOMER_ID>{ xf:data($XM_BB_C.CUSTOMER_1,
         <STATE>{ xf:data($XM_BB_C.CUSTOMER_1/STATE]
         <EMAIL_ADDRESS>{ xf:data($XM_BB_C.CUSTOMER_
         <TELEPHONE_NUMBER>{ xf:data($XM_BB_C.CUSTOI
```

```
<CustomerOrderReport>
  <customerOrder>
    <CUSTOMER>
      <FIRST_NAME>JOHN_B_1</FIRST_NAME>
      <LAST_NAME>KAY_1</LAST_NAME>
      <CUSTOMER_ID>CUSTOMER_1</CUSTOMER_ID>
      <STATE>TX</STATE>
      <EMAIL_ADDRESS>user@user.com</EMAIL_ADDRESS>
      <TELEPHONE_NUMBER>4151231234</TELEPHONE_NUMBER>
    <wireless_orders>
      <CUSTOMER_ORDER>
        <ORDER_DATE>2002-04-09</ORDER_DATE>
        <ORDER_ID>ORDER_ID_1_0</ORDER_ID>
        <CUSTOMER_ID>CUSTOMER_1</CUSTOMER_ID>
        <SHIP_METHOD>AIR</SHIP_METHOD>
        <TOTAL_ORDER_AMOUNT>1000.00</TOTAL_ORDER_AMOUN
      <CUSTOMER_ORDER>
        <ORDER_DATE>2002-04-09</ORDER_DATE>
        <ORDER_ID>ORDER_ID_1_1</ORDER_ID>
        <CUSTOMER_ID>CUSTOMER_1</CUSTOMER_ID>
```

Alternatively, you can view the query results as XML source (Figure 6-5). To view as XML select the checkbox at the bottom of the query Results panel. You can also right-click on the results pane to change the results format to XML source.

Figure 6-5 Query Result in XML Source Format



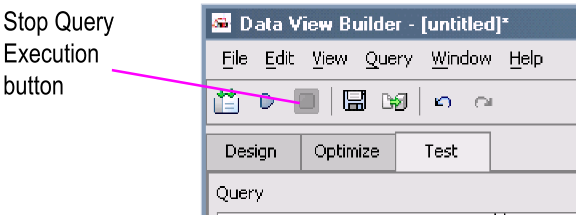
```
<CustomerOrderReport>
  <customerOrder>
    <CUSTOMER>
      <FIRST_NAME>JOHN_B_1</FIRST_NAME>
      <LAST_NAME>KAY_1</LAST_NAME>
      <CUSTOMER_ID>CUSTOMER_1</CUSTOMER_ID>
      <STATE>TX</STATE>
      <EMAIL_ADDRESS>user@user.com</EMAIL_ADDRESS>
      <TELEPHONE_NUMBER>4151231234</TELEPHONE_NUMBER>
    </CUSTOMER>
    <wireless_orders>
      <CUSTOMER_ORDER>
        <ORDER_DATE>2002-04-09</ORDER_DATE>
        <ORDER_ID>ORDER_ID_1_0</ORDER_ID>
        <CUSTOMER_ID>CUSTOMER_1</CUSTOMER_ID>
        <SHIP_METHOD>AIR</SHIP_METHOD>
        <TOTAL_ORDER_AMOUNT>1000.00</TOTAL_ORDER_AMOUNT>
      </CUSTOMER_ORDER>
      <CUSTOMER_ORDER>
        <ORDER_DATE>2002-04-09</ORDER_DATE>
        <ORDER_ID>ORDER_ID_1_1</ORDER_ID>
        <CUSTOMER_ID>CUSTOMER_1</CUSTOMER_ID>
        <SHIP_METHOD>AIR</SHIP_METHOD>
        <TOTAL_ORDER_AMOUNT>2000.00</TOTAL_ORDER_AMOUNT>
      </CUSTOMER_ORDER>
    </wireless_orders>
    <broadband_orders>
      <CUSTOMER_ORDER>
        <ORDER_DATE>2002-04-09</ORDER_DATE>
        <ORDER_ID>ORDER_ID_1_0</ORDER_ID>
        <CUSTOMER_ID>CUSTOMER_1</CUSTOMER_ID>
        <SHIP_METHOD>AIR</SHIP_METHOD>
        <TOTAL_ORDER_AMOUNT>1000.00</TOTAL_ORDER_AMOUNT>
      </CUSTOMER_ORDER>
      <CUSTOMER_ORDER>
        <ORDER_DATE>2002-04-09</ORDER_DATE>
        <ORDER_ID>ORDER_ID_1_1</ORDER_ID>
        <CUSTOMER_ID>CUSTOMER_1</CUSTOMER_ID>
        <SHIP_METHOD>AIR</SHIP_METHOD>
        <TOTAL_ORDER_AMOUNT>1500.00</TOTAL_ORDER_AMOUNT>
      </CUSTOMER_ORDER>
    </broadband_orders>
  </customerOrder>
</CustomerOrderReport>
```

Once generated in XML, query results can easily be formatted in a variety of printed reports or displayed in web pages. For an example of the use of Liquid Data data as part of a web application see the Getting Started Demo and "Understanding the Avitek Customer Service Sample Application" in *Liquid Data by Example*.

Stopping a Running Query

You can stop a running query before it has finished by clicking the Stop Query Execution button in the Toolbar (Figure 6-6).

Figure 6-6 Stop Query Execution Button



Specifying Large Results Sets

If you expect your query to produce a result set that may exceed available memory, choose the Large Results option, available from the Test mode page (lower left, Figure 6-1) prior to running your query.

If the Large Results option is selected for a query, Liquid Data will use swap files to temporarily store intermediate results. Although your query will run more slowly, this option will avoid possible out-of-memory errors.

Note: Before using this option, increase heap size on the server. This is further discussed in the chapter "Tuning Performance" in the Liquid Data *Deployment Guide.*

You can explicitly specify a directory for swap files using the Administration Console. For more information, see "Configuring Liquid Data Server Settings" in the *Administration Guide*.

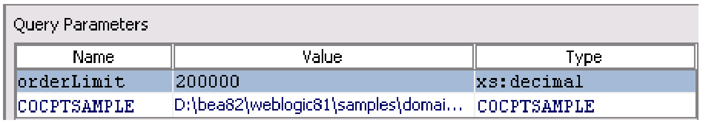
Specifying Query Parameters

You can use the Query Parameters panel (located below the generated query area) to enter or change variable values prior to running a query.

The types of variables that may appear include:

- Query parameters defined in Design mode (see "Creating and Using Query Parameters" on page 5-10).
- Complex Parameter Type (CPT) data source file names, which are selected using a file browser (see Chapter 9, "Using Complex Parameter Types in Queries" and "Configuring Access to Complex Parameter Types" in the *Administration Guide*).

Figure 6-7 Sample Query Parameter and Complex Parameter Type (CPT) Settings


Query Parameters

Name	Value	Type
orderLimit	200000	xs:decimal
COCPTSAMPLE	D:\bea82\weblogic81\samples\domai...	COCPTSAMPLE

Setting and Changing Query Parameters

To set or change query parameters click in a cell in the Value column. You can either delete what is currently there or edit it using simple character insert and delete commands.

For examples of using query parameters see the following example queries in "*Liquid Data by Example*."

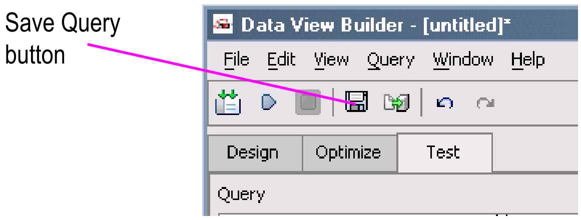
- "Example 1: Simple Joins"

- "Example 2: Aggregates"
- "Example 3: Date and Time Duration"

Saving a Query

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

Figure 6-8 Click the Save Query Button


Save Query button
Data View Builder - [untitled]*
File Edit View Query Window Help
Design
Optimize
Test
Query

Security Considerations

In order to save or update files to a Liquid Data repository that has security restrictions in place, you need to either belong to the proper security group or specifically have appropriate access permissions granted to you by your Liquid Data administrator.

If you have such permissions, whenever you start the Data View Builder you need to enter the exact, case-sensitive user name and password that was assigned to you by your Liquid Data administrator.

Liquid Data security is based on the WebLogic Server security policy system. For details see "Security in Liquid Data" in the *Administration Guide* which contains additional references to WebLogic Server security documentation.

Note: By default, the Liquid Data Samples repository does not have any security restrictions so you can freely create or delete items, including queries in the `stored_queries` repository directory without having to enter a user name or password.

Query Naming Conventions

There are a few Liquid Data query naming conventions:

- **Stored queries always have an `.xq` extension.** Queries saved to the Liquid Data repository much have an `.xq` extension. 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. This is because the query name is converted to an XML tag in the web service-generation process. (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*.)

  XML naming conventions require that a 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 a hyphen) 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.

  For a complete description of naming conventions for schema tags see *W3C XML Schema* document at http://www.w3.org/XML/Schema

Using the stored_queries Folder

The advantages of saving your queries to the `stored_queries` folder of the Liquid Data repository include:

- Providing the Liquid Data administrator with the ability to create access control at the stored query level
- The query can be used in a WebLogic Workshop
- The query can be turned into a web service
- The query can be turned into a data view

Caching Query Results

When your query resides in the Liquid Data repository you also have the option to configure caching on the query result. Caching of query results for stored queries is configurable through the Liquid Data node of the WebLogic Administration Console (see "Configuring the Query Results Cache" in the Liquid Data *Administration Guide*).

Steps to Save a Query to the Repository

If the query is saved into the `<ld_repository>/stored_queries` folder, it becomes a stored query in Liquid Data.

To save a query follow these steps:

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.)
2. Use the file browser to navigate to the Liquid Data Repository; it is the topmost directory listed.
3. Enter a name for the query in the File name field of the file browser and click Save. The query is saved to the `stored_queries` folder in the Repository with a `.xq` extension.

You can reload a stored query using the Liquid Data File —> Open Query command after navigating to the directory containing your stored queries.

Once a stored query has been loaded, it is ready to be run. See "Running a Query" on page 6-3.

Deploying a Query

In a Liquid Data query, data is drawn from heterogeneous sources and then normalized (through automatic and manual casting) to elements in the Data View Builder target XML schema. This information is automatically available when a query is run from the Data View Builder. However, in order for it to be available to applications such as the WebLogic Workshop Liquid Data Control the relationship must be formalized in the Liquid Data Server.

A *deployed query* is a query that has been saved to the Liquid Data repository and then associated with a target schema that is also in the repository. See "To Configure (Deploy) a Stored Query" in the *Administration Guide*.

A query can be deployed through the Data View Builder or the Liquid Data node of the WebLogic Administration Console. In the Administration Console this operation is known as *query configuration*.

A *data view* is functionally identical to a deployed query. The difference is that a data views function as data sources for other queries. See Chapter 8, "Using Data Views."

Deploy Query Command

The Deploy Query command is available from the Query menu when in Test mode. There are two requirements for deploying queries:

- You can only deploy a query if you have adequate permissions. See "Security Considerations" on page 6-6.
- You cannot "re-deploy" a query. In essence deploying a stored query means that the stored query has been configured to be associated with a specific target schema. The Liquid Data Server tracks the association of these two Liquid Data repository components: a query and a schema, using the name of the query. The association is dynamic so any changes you make to

the schema or query file in the repository will automatically be available to the consuming application.

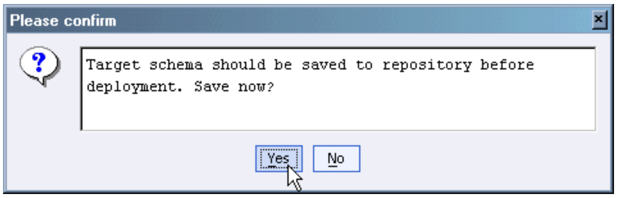
Note: Data views follow the same “no re-deployment” model. You can always delete and recreate a data view of the same name with different components.

To redeploy a stored query or data view, use the Liquid Data node of the WebLogic Administration Console. See “Configuring Stored Queries” in the Liquid Data *Administration Guide* for additional information on managing stored queries.

Saving the Current Schema and Current Query

When you initially select the Deploy Query menu option, you are asked if you want to save your current target schema and query since only saved target schemas and queries can be part of a deployed query.

Figure 6-9 Schema is Current Confirmation



Please confirm

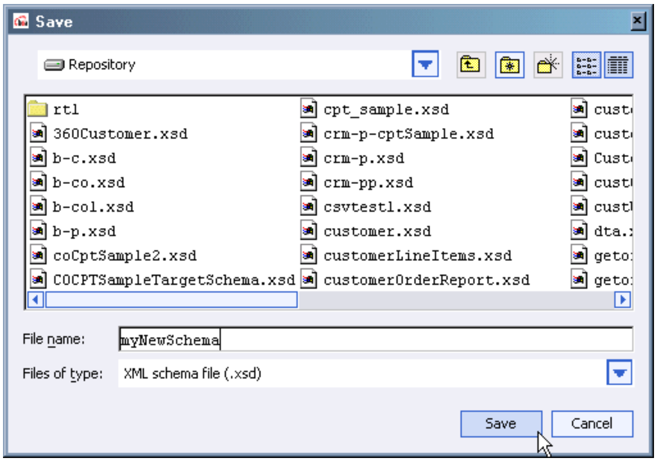
```
Target schema should be saved to repository before
deployment. Save now?
```

- Answer Yes unless you do not want your current schema to be part of the deployed query.

  If you select Yes a file browser will appear, open to the `<LD_REPOSITORY>/schemas` directory. Select a current schema or save the schema under a new name.

  If you select No you will be able to select a schema file in the Deploy Query dialog box.

Figure 6-10 Save Schema Dialog Box



File name: myNewSchema

Files of type: XML schema file (.xsd)

Follow the same approach in deciding whether to save your query to your `<LD_REPOSITORY>/stored_queries` directory.

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

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

it could be executed from a `.JSP` with:

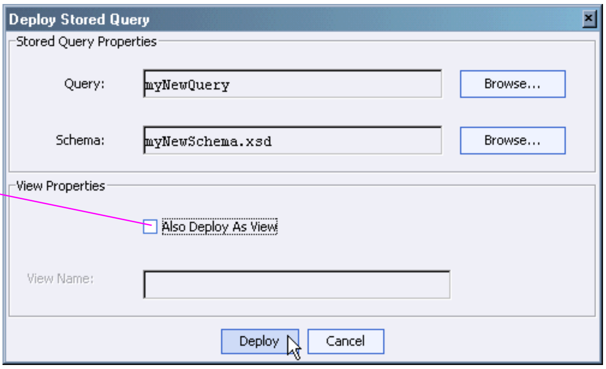
```
<lds:query name="uCustomer.custQuery">
</lds:query>
```

Although a query can be deployed immediately upon entering Test mode, it is a good practice to first run the query to make sure that the results match your expectations.

Deploying Your Query

The Deploy Stored Query dialog box gives you the option of creating a data view at the same time you deploy your query.

Figure 6-11 Deploy Query Dialog



Query: myNewQuery

Schema: myNewSchema.xsd

Follow these steps to deploy your query:

1. **Select Your Query.** If you selected to save your query, the Query field will already contain the name of the query you saved. Optionally, browse to another query in the `stored_query` directory of the repository.

2. **Select a Schema**. If you selected to save your schema, the Schema field will contain the name of the schema you saved. Optionally, you can use the Select... button to browse to a different schema.

3. Optionally, select a name for your data view. For details see "Deploying a Stored Query with a Data View" on page 6-11.

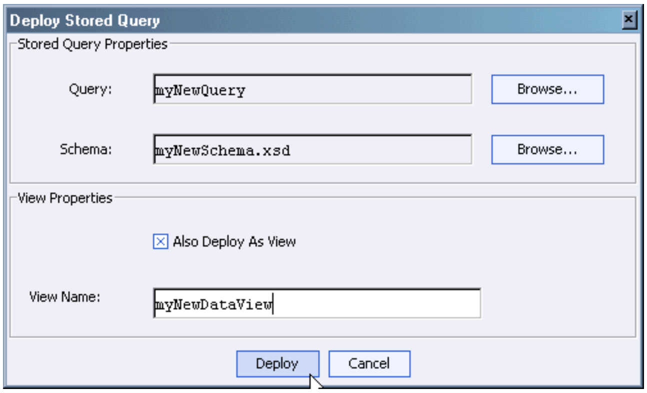
4. **Deploy Your Query.** Click the Deploy button to resave your query with schema information.

Note: Once you have deployed a query to a particular name, you cannot "redeploy" to the same name from the Data View Builder. There is also no need to do this. As long as you are happy with the particular schema you have associated with the stored query, any changes made to these files will automatically be made available from the Liquid Data Server to the application program, including the Data View Builder.See "Configuring Stored Queries" in the Liquid Data *Administration Guide* for additional information on managing stored queries.

Deploying a Stored Query with a Data View

When you deploy a query, you can optionally create a data view. (The difference between a deployed query and a Data View is that your Data View can be used as a data source in the Data View Builder. See Chapter 8, "Using Data Views," for additional information.)

Figure 6-12 Deploy Query Dialog with Create Data View Option Enabled



Query: myNewQuery

Schema: myNewSchema.xsd

[✓ Also Deploy As View]

View Name: myNewDataView

The additional steps to deploying a query and a Data View are:

1. Check the Also Deploy As View checkbox.
2. Enter a name for your Data View (choose a name that is not already in use).
3. Click the Deploy button. In addition to deploying your stored query, a Data View will be created in the Liquid Data `<ld_repository>/dataview` directory.

For information on creating and managing data views see "Configuring Access to Data Views" in the Liquid Data *Administration Guide*.

Analyzing and Optimizing Queries

This chapter describes techniques for optimizing Liquid Data queries.

Note: Tuning Performance in the *Deployment Guide* contains a general discussion of factors related to tuning and performance of Liquid Data including query design, data sources, and platform considerations.

The following sections are included:

- Query Analysis
- Factors in Query Performance
- Optimizing Queries

Query Analysis

Two tools are available to help you analyze how the query you created executes on the Liquid Data Server and to measure the performance of the query against its various data sources:

- Query Plan
- Performance Information

Viewing the Query Plan

The query plan is designed to help you:

- Understand how a query is executed \\

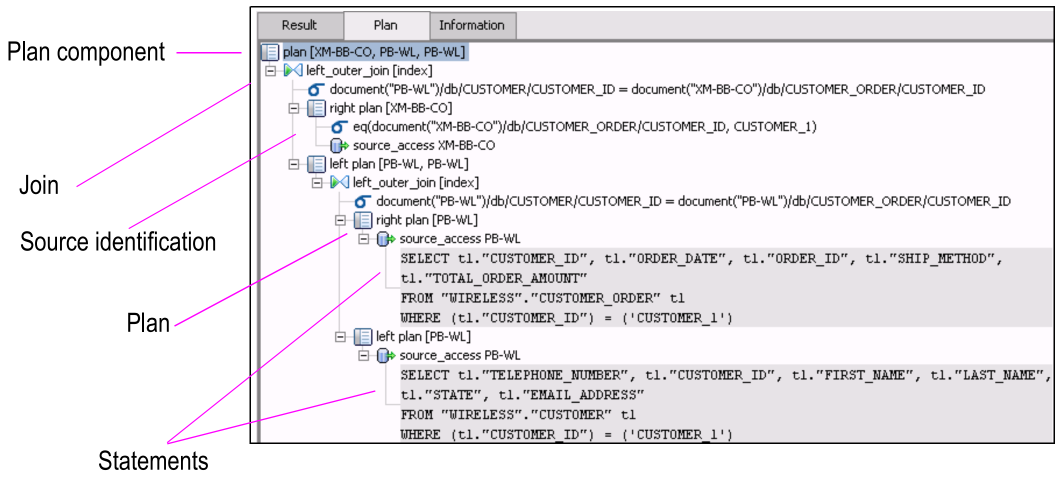
- Identify areas where query tuning may improve performance

Only the parts of the plan that may have significant performance impact on execution time are displayed when you select the Plan tab in the Results pane. The returned plan identifies the following query components:

- Join
- Outer join
- Select
- Sources
- Custom function calls
- Order-bys
- Remove duplicates
- Source access operator

The query plan also appears if you select the Compile Query menu option or, in Design mode, the Compile Query icon.

Figure 7-1 Query Plan (DB-XML Sample Project: e2e-order.qpr)



```
plan [XM-BB-CO, PB-WL, PB-WL]
  left_outer_join [index]
    document("PB-WL")/db/CUSTOMER/CUSTOMER_ID = document("XM-BB-CO")/db/CUSTOMER_ORDER/CUSTOMER_ID
    right plan [XM-BB-CO]
      eq(document("XM-BB-CO")/db/CUSTOMER_ORDER/CUSTOMER_ID, CUSTOMER_1)
      source_access XM-BB-CO
    left plan [PB-WL, PB-WL]
      left_outer_join [index]
        document("PB-WL")/db/CUSTOMER/CUSTOMER_ID = document("PB-WL")/db/CUSTOMER_ORDER/CUSTOMER_ID
        right plan [PB-WL]
          source_access PB-WL
            SELECT t1."CUSTOMER_ID", t1."ORDER_DATE", t1."ORDER_ID", t1."SHIP_METHOD",
            t1."TOTAL_ORDER_AMOUNT"
            FROM "WIRELESS"."CUSTOMER_ORDER" t1
            WHERE (t1."CUSTOMER_ID") = ('CUSTOMER_1')
        left plan [PB-WL]
          source_access PB-WL
            SELECT t1."TELEPHONE_NUMBER", t1."CUSTOMER_ID", t1."FIRST_NAME", t1."LAST_NAME",
            t1."STATE", t1."EMAIL_ADDRESS"
            FROM "WIRELESS"."CUSTOMER" t1
            WHERE (t1."CUSTOMER_ID") = ('CUSTOMER_1')
```

Getting Information on the Query

Information available on a query after it has been compiled or executed in the Data View Builder includes:

- Compilation time
- Execution time
- Retrieval time and number of invocations for each data source
- Query statement

Listing 7-1 Sample Information After Running a Query (DB-XML Sample Project: e2e-order.qpr)

```
Compilation time: 2.403 sec
Execution time: 1.052 sec

Source: PB-WL{1}
Data retrieval time: 0.02 sec
Invocations: 1
Statement: SELECT t1."CUSTOMER_ID", t1."ORDER_DATE", t1."ORDER_ID",
t1."SHIP_METHOD", t1."TOTAL_ORDER_AMOUNT"

FROM "WIRELESS"."CUSTOMER_ORDER" t1
WHERE (t1."CUSTOMER_ID") = ('CUSTOMER_1')

Source: PB-WL{0}
Data retrieval time: 0.111 sec
Invocations: 1
Statement: SELECT t1."TELEPHONE_NUMBER", t1."CUSTOMER_ID", t1."FIRST_NAME",
t1."LAST_NAME", t1."STATE", t1."EMAIL_ADDRESS"

FROM "WIRELESS"."CUSTOMER" t1 WHERE (t1."CUSTOMER_ID") = ('CUSTOMER_1')

Source: XM-BB-CO{2}
Data retrieval time: 0.631 sec
Invocations: 0
Statement: parser
```

Factors in Query Performance

If you have a good understanding of your data sources and relationships, you will be in a good position to try to improve query performance. It help greatly if you:

- Know your data sources from a size and schema point of view.
- Evaluate the relationships between your data. Consider factors like expected query result size, memory requirements, and ability to leverage stored queries, as appropriate.

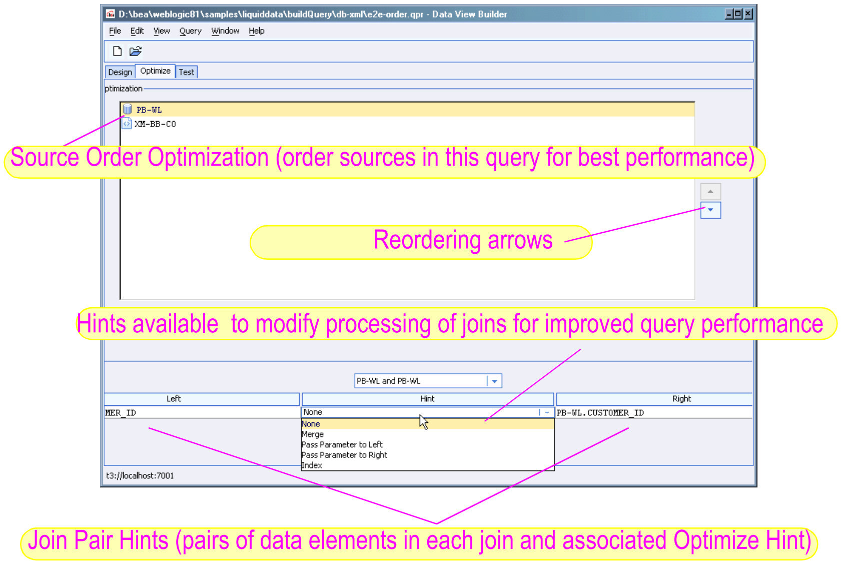
Taking such factors into account you are in a position to add effective optimization hints that may greatly improve query performance.

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.

Optimizing Queries

To access tools to improve query performance, click on the Optimize tab. (See Figure 7-2.)

Figure 7-2 Optimize Tab


D:\bea\weblogic81\samples\liquiddata\buildQuery\db-xml\e2e-order.qpr - Data View Builder
File Edit View Query Window Help
Design Optimize Test
ptimization
PB-WL
XM-BB-CO
Source Order Optimization (order sources in this query for best performance)
Reordering arrows
Hints available to modify processing of joins for improved query performance
PB-WL and PB-WL
Left
Hint
Right
MER_ID
None
PB-WL.CUSTOMER_ID
None
Merge
Pass Parameter to Left
Pass Parameter to Right
Index
t3://localhost:7001
Join Pair Hints (pairs of data elements in each join and associated Optimize Hint)

Source Order Optimization

When a query uses data from several sources, the Liquid Data Server creates intermediate results into memory combining data from the different sources. The size of these intermediate results depends on the amount of data retrieved from each data source. If you specify more than two sources, the Liquid Data Server combines 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, retrieved by 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 sources and so on. A query is generally more efficient when it minimizes the size of intermediate results.

The order of the peer XQuery `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 size. That is, the smallest resource should appear first in the list and the largest resource should appear last.

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.

Example of Source Order Optimization

Consider a query designed to find all managers and the departments they manage that contains a join across three sources: Employees, Employees2 (Employees opened a second time), and Departments.

Note: You will notice that *join selectivity* is not considered in the following discussion. This is for the sake of simplicity.

This query joins the Employees schema `ID` field and the Employees2 schema `Manager_id` field to return all managers. It 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 for a total of 150,000. More accurately, the query would generate a fraction of this number, depending on the *join selectivity* of the two sources.

A better plan would be to combine Employees with Departments first, then combine that result with Employees2. This is easily accomplished in the Source Order Optimization pane by clicking on the "Department" data source and then on the up-arrow (see Figure 7-2).

The effect is to generates (100 * 5) + (500 * 100) intermediate results for a total of 50,500 intermediate results, a considerable potential processing reduction.

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

A critical factor in query performance is the way disparate data sources are joined by the Liquid Data Server. The Liquid Data Server offers three different join methods. The Liquid Data Server optimizer applies heuristics to determine the best method for each case. However, you can apply a *join hint* in cases where you wish to override the method chosen by the optimizer. In some cases query performance can be greatly improved by properly applying query hints. In some cases query hints can greatly improve performance.

The Optimize tab on the Data View Builder provides a drop-down list of data source pairs and a table that shows the joins that have been applied to each pair. For each join in the table you can provide a hint about how to join the data most efficiently. (See Figure 7-2.)

When used, query hints appear in the query as special-purpose strings enclosed within comment brackets: `{--! `*`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.

You can easily experiment with different query hints to determine the optimal settings.

Determining When Hints Are Needed

By default no hints are specified, meaning that the Liquid Data built-in optimizer is used. To add a hint to a particular join, select the join and choose a hint from the drop-down Query Hints list. The available hints are shown in Table 7-3.

Table 7-3 Optimization Hints

Hint	Description	Syntax
None (optimizer)	The Liquid Data optimizer takes a best guess at optimizing the statement.	n/a
Left	Parameter Pass to the Left (ppleft)	`{--! ppleft !--}`
Right	Parameter Pass to the Right (ppright)	`{--! ppright !--}`
Merge	Merge	`{--! merge !--}`
Index	Index(es) will be used.	`{--! index !--}`

Apply these rules to determine the correct hint to choose.

Table 7-4 When to Use Which Hint

Hint	When To Use
None (optimizer)	• Uses the built-in Liquid Data query optimizer. In many cases this will yield the best results.
Merge	• Both relational database sources are large and cannot fit into memory.
Parameter Passing (Left *or* Right)	• One of the sources has fewer objects than the other. See "Using Parameter Passing Hints (ppleft or ppright)" on page 7-8.
Index	• The size of the source identified on the right side of the hint is small enough to fit into memory. • The left and right sources are generally equal in size. • There is at least one non-relational source used in the join.

Notes:

- Using optimization hints can help you improve performance on equijoin conditions, which contain only one equality. If the query performs complex join conditions such as (A eq B) OR (C eq D), the join conditions will not appear on the Optimizer tab as the engine does not need a hint for these join conditions. Only the join conditions that are equi-join such as (A eq B), can be given a hint in the Optimizer pane or by manually editing a query.
- Choosing the wrong direction for a parameter passing hint can degrade performance.

The following sections provide more detail regarding each type of hint setting.

Using the Liquid Data Built-in Optimizer

When no hints are provided Liquid Data attempts to optimize the query based on an analysis of the query plan based on the frequently correct premise that the most selective kinds of conditions becomes the driving source that passes parameters to the rest of the query, as appropriate.

Using Parameter Passing Hints (ppleft or ppright)

Choose a Parameter Passing hint when one of the sources has a fairly small number of data objects. 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" in *Liquid Data by Example*.

Listing 7-2 XQuery with ppright Hints

```
{--Generated by Data View Builder 8.1 --}

<customers>

{for $PB-WL.CUSTOMER_1 in document("PB-WL")/db/CUSTOMER
where ($#first_name of type xs:string  eq  $PB-WL.CUSTOMER_1/FIRST_NAME)
return
   <customer id={$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>
```

Note: 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. This assumes that the PB-BB data source has a larger amount of customer IDs. You 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.

Using Merge Hints

Choose a merge hint when both relational database sources are large and cannot fit into memory.

The following example shows the XQuery for a merge hint.

Listing 7-3 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, the input must be sorted on join attributes. A query using a merge join might have a slower response time than a query without a hint, but the memory footprint is typically much smaller with the merge join.

Note: A merge join in a character column might yield unexpected results because the collating sequence for each database may be vary. See Table 7-5 for an example of how incompatible ordering sequences for strings from two different vendors can affect query results.

Table 7-5 Collation Sequences for Some Data Types Vary by Database Vendor

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

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

Using Data Views

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

The Enterprise and the Data View

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

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

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

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

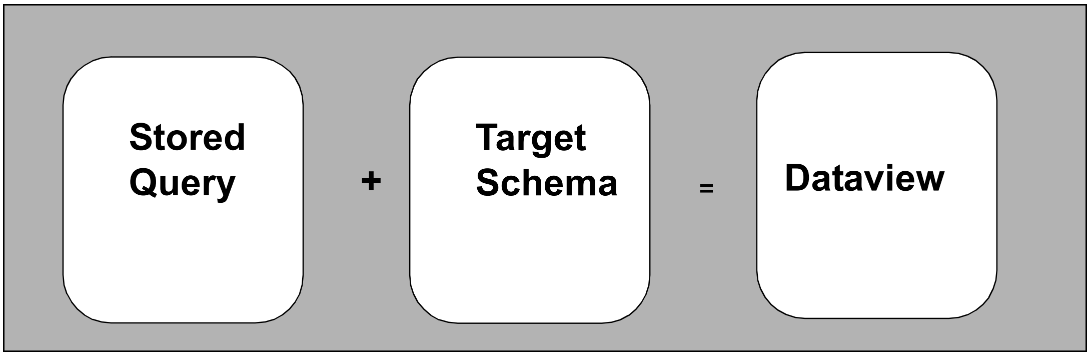
- Once validated, a reusable representation of each data view is developed through the Data View Builder and Liquid Data node of the WebLogic Administration Console as a new data view.
- Then the Liquid Data data view can be used throughout the enterprise as a *virtual data source* for queries. For example,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

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

Understanding Data Views

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

Figure 8-1 Components of a Data View



To create a data view from a query:

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

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

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

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

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

A Data View Use Case

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

– To enable HR to get answers to employees quickly and economically, an Information Technology data architect creates a query using Liquid Data that can access relevant information from the multiple payroll systems and the company's incentive bonus databases.

– Once satisfied that the query works, the IT architect creates a data view and makes it available to an HR data specialist. This specialist can then use Liquid Data to quickly get answers to inquiries from individual employees about their bonus payments.

The benefits of this approach are significant:

– A single integrated view can be created for use throughout an enterprise. Access to sensitive information is controlled and consistency is maintained.

– HR can quickly get the information it needs without having to either staff up with its own data architect or get in the queue for expensive and low-availability IT custom programming services.

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

Simple and Parameterized Data Views

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

Using Data Views as Data Sources

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

Creating a Data View

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

- Creating and Saving the Query to the Liquid Data Repository
- Configuring a Data View Data Source Description
- Adding a Data View as a Data Source

Creating and Saving the Query to the Liquid Data Repository

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

1. Construct the query in the Design view as described in Chapter 5, "Building Queries."
2. Test the query in the Test Query view as described in Chapter 6, "Running, Saving, and Deploying Queries."
3. Save the query to the Liquid Data repository as a stored query as described in Steps to Save a Query to the Repository in Chapter 6, "Running, Saving, and Deploying Queries."

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

Alternatively you can load queries and target schemas into the Liquid Data repository directly using the Liquid Data node of the WebLogic Administration Console. See Uploading Files to the Server Repository for details.

Configuring a Data View Data Source Description

In the WebLogic Administration Console, configure a data view for the query as described in Configuring Access to Data Views in the Liquid Data *Administration Guide*. Then follow these steps:

1. In the Liquid Data node of the WebLogic Administration Console click the Repository tab.

2. Double-click on the Stored Queries folder.
3. Find the stored query you want to use and inspect the Data Source Configuration column.
4. If Create Data View is available, click on that link to create you data view. (If a data view already exists based on the stored query, you will see Data View Created.)

   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 name you select for your data view.

See "Managing the Liquid Data Server Repository" in the Liquid Data *Administration Guide* for additional details.

Adding a Data View as a Data Source

After you have created the data view, reconnect to the Liquid Data server using the `File -> Connect` menu command. Your new data view should appear under Data Views when the Sources tab in Design mode is selected (see Figure 8-4).

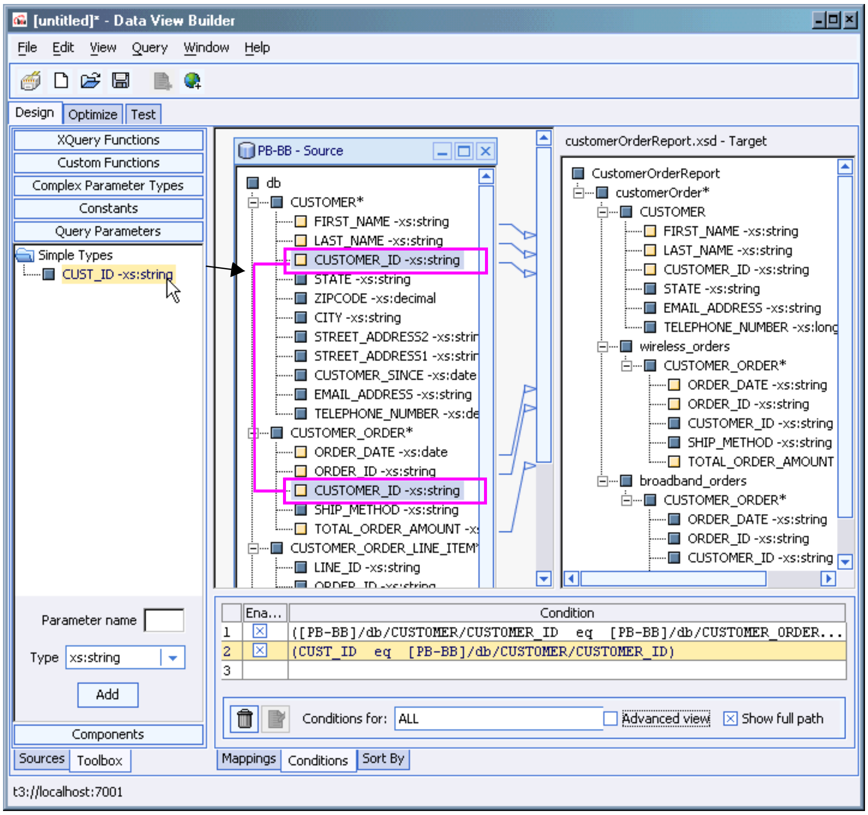
Creating a Parameterized Data View

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

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

1. Open the Data View Builder, drag the relational database source `pb-bb` onto the Liquid Data desktop. Set your target schema to `customerOrders.xsd`. Map elements to your target schema as shown in Figure 8-2.

Figure 8-2 Creating a Parameterized Query in the Data View Builder



Parameter name

Type xs:string

	Ena...	Condition
1	☒	([PB-BB]/db/CUSTOMER/CUSTOMER_ID eq [PB-BB]/db/CUSTOMER_ORDER...
2	☒	(CUST_ID eq [PB-BB]/db/CUSTOMER/CUSTOMER_ID)
3		

Conditions for: ALL ☐ Advanced view ☒ Show full path

1. From the Liquid Data Toolbox tab, choose Query Parameter. Create a single query parameter, `CUST_ID` and using the pulldown Type menu. Assign it a type string of `xs:string`.

2. Drag `cust_id` to the CUSTOMER_ID field of the CUSTOMER table in the PB_BB data source (also shown in Figure 8-2).

3. Drag CUSTOMER_ID in the Customer table to CUSTOMER_ID in the Customer Order table to create a join.

4. In Test mode supply `CUSTOMER_1` as a value for CUST_ID and run the query.

   Note: Values are case-sensitive.

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

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

6. Now you can use the Liquid Data node of the WebLogic Administration Console to create your data view from your newly saved query.

   a. Start the WebLogic Administration Console.

   b. Click the Liquid Data node.

   c. Select the Repository tab.

   d. Go to stored_queries.

   e. Choose the query `param-dv.xq`

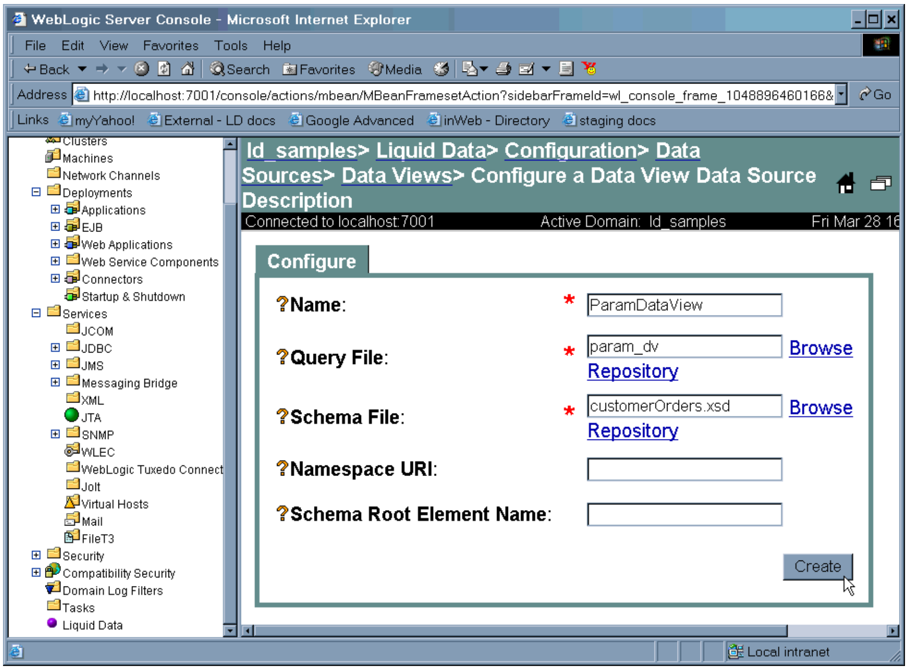
   f. Select the `Data View Data Source` option.

   g. Enter information as shown in Figure 8-3.

   h. Click Create.

   Your new data view should appear in the Liquid Data node of the WebLogic Administration Console list of available data views.

Figure 8-3 Creating the Data View in the Administration Console

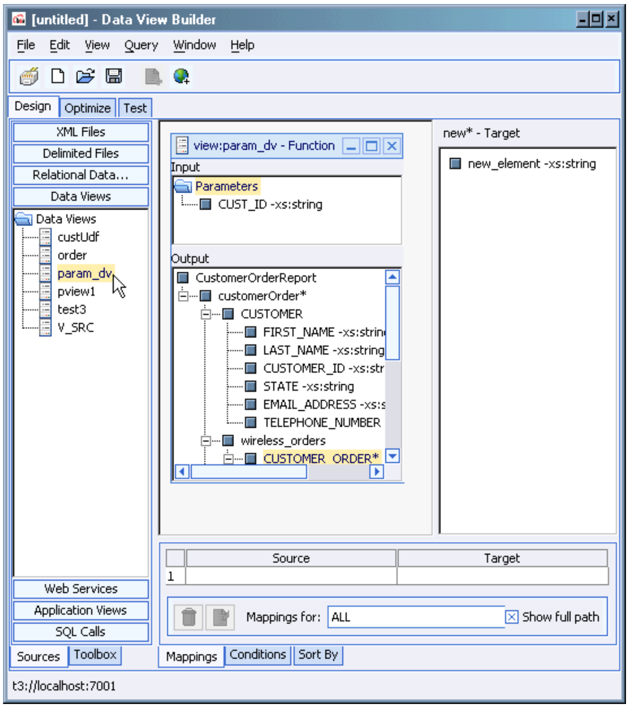

WebLogic Server Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost:7001/console/actions/mbean/MBeanFramesetAction?sidebarFrameId=wl_console_frame_1048896460166&
Links myYahoo! External - LD docs Google Advanced inWeb - Directory staging docs
Clusters
Machines
Network Channels
Deployments
Applications
EJB
Web Applications
Web Service Components
Connectors
Startup & Shutdown
Services
JCOM
JDBC
JMS
Messaging Bridge
XML
JTA
SNMP
WLEC
WebLogic Tuxedo Connect
Jolt
Virtual Hosts
Mail
FileT3
Security
Compatibility Security
Domain Log Filters
Tasks
Liquid Data
ld_samples> Liquid Data> Configuration> Data Sources> Data Views> Configure a Data View Data Source Description
Connected to localhost:7001 Active Domain: ld_samples Fri Mar 28 16
Configure
Browse
Repository
Create
Local intranet

Name: *	ParamDataView
Query File: *	param_dv
Schema File: *	customerOrders.xsd
Namespace URI:	
Schema Root Element Name:	

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

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

Figure 8-4 Data View on the Liquid Data Desktop

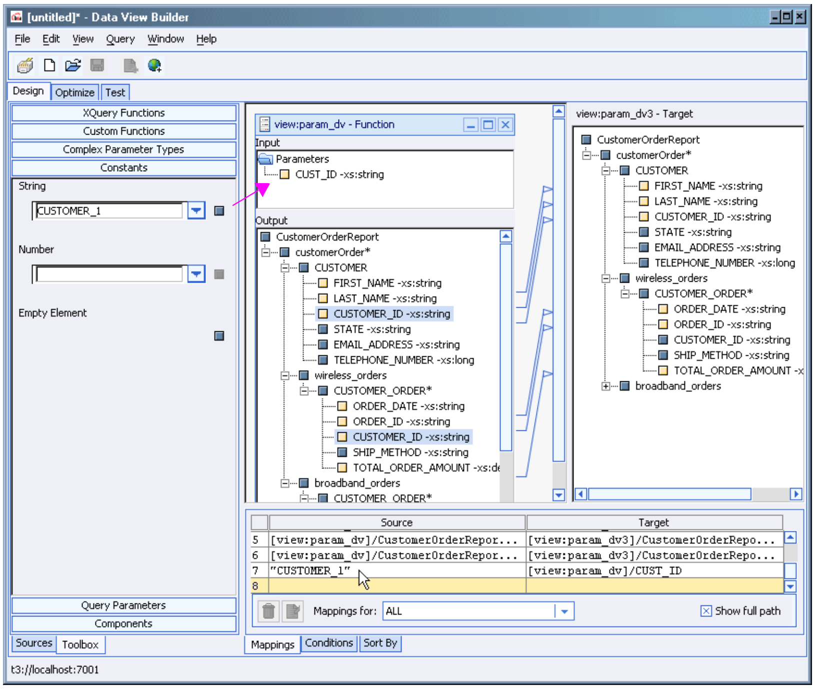


8. Drag the data view onto the Liquid Data desktop as you would any other data source (Figure 8-4).

9. Add a valid target schema. In this case, you can use the File menu `Set Selected Source as Target Schema` command. (The generated schema is shown in Figure 8-5.)

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

Figure 8-5 Setting Input and Associating Columns With Target Schema



	Source	Target
5	[view:param_dv]/CustomerOrderRepor...	[view:param_dv3]/CustomerOrderRepo...
6	[view:param_dv]/CustomerOrderRepor...	[view:param_dv3]/CustomerOrderRepo...
7	"CUSTOMER_1"	[view:param_dv]/CUST_ID
8		

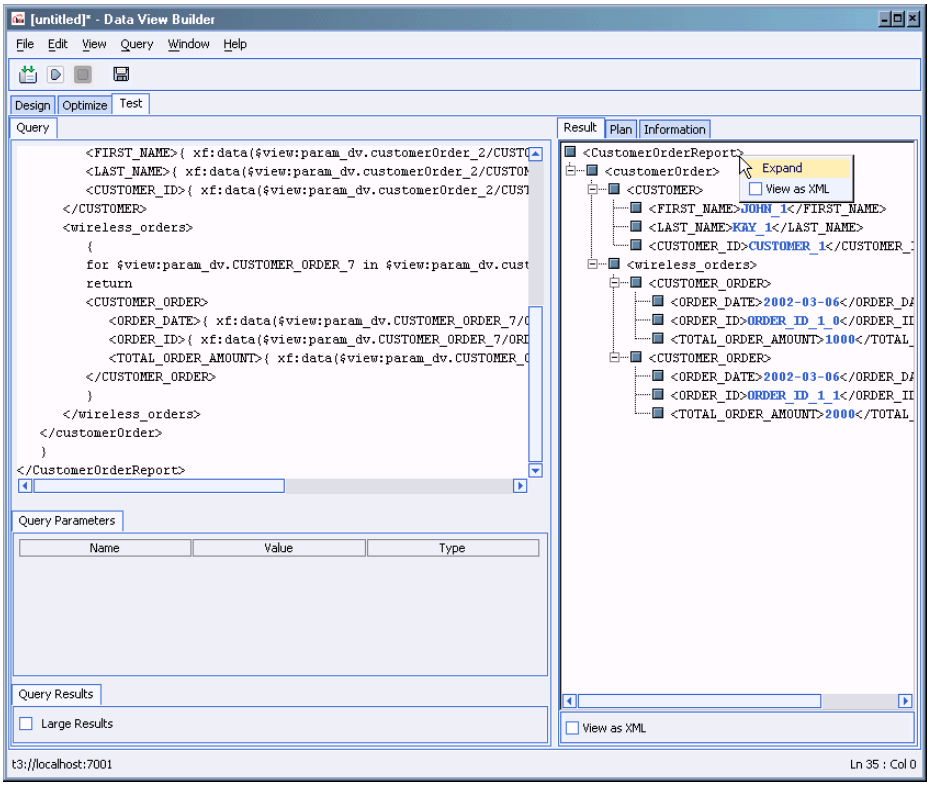
10. Using the Data View Builder toolbox create a string constant called `CUSTOMER_3` and drag it into the data view input `CUST_ID` (see Figure 8-5).

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

11. Map all the elements in your target schema.

12. Test, then run your new query.

Figure 8-6 XQuery and Generated XML Report



```
            <FIRST_NAME>{ xf:data($view:param_dv.customerOrder_2/CUSTO
            <LAST_NAME>{ xf:data($view:param_dv.customerOrder_2/CUSTON
            <CUSTOMER_ID>{ xf:data($view:param_dv.customerOrder_2/CUST
        </CUSTOMER>
        <wireless_orders>
            {
            for $view:param_dv.CUSTOMER_ORDER_7 in $view:param_dv.cust
            return
            <CUSTOMER_ORDER>
                <ORDER_DATE>{ xf:data($view:param_dv.CUSTOMER_ORDER_7/C
                <ORDER_ID>{ xf:data($view:param_dv.CUSTOMER_ORDER_7/ORD
                <TOTAL_ORDER_AMOUNT>{ xf:data($view:param_dv.CUSTOMER_C
            </CUSTOMER_ORDER>
            }
        </wireless_orders>
    </customerOrder>
    }
</CustomerOrderReport>
```

```
<CustomerOrderReport>
  <customerOrder>
    <CUSTOMER>
      <FIRST_NAME>JOHN_1</FIRST_NAME>
      <LAST_NAME>KAY_1</LAST_NAME>
      <CUSTOMER_ID>CUSTOMER_1</CUSTOMER_I
    <wireless_orders>
      <CUSTOMER_ORDER>
        <ORDER_DATE>2002-03-06</ORDER_DA
        <ORDER_ID>ORDER_ID_1_0</ORDER_II
        <TOTAL_ORDER_AMOUNT>1000</TOTAL_
      <CUSTOMER_ORDER>
        <ORDER_DATE>2002-03-06</ORDER_DA
        <ORDER_ID>ORDER_ID_1_1</ORDER_II
        <TOTAL_ORDER_AMOUNT>2000</TOTAL_
```

Data View Query Samples

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

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

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

Also see the Liquid Data Samples page for more information on other available query samples.

CHAPTER 9

Using Complex Parameter Types in Queries

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

The following subjects are discussed in this chapter:

- Understanding Complex Parameter Types
- Creating a Complex Parameter Type
- Complex Parameter Type Query Samples

Understanding Complex Parameter Types

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

In order to use CPTs, you need:

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

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

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

- "Setting Complex Parameter Types" and "Invoking Queries in EJB Clients" in the *Application Developer's Guide*.
- Liquid Data 8.1 API Reference (Javadoc).

There is also a Liquid Data EJB API sample in the directory:

```
<WL_HOME>/samples/<liquiddata>/ejbAPI
```

A CPT Use Case

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

Use Case

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

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

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

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

- customer identification
- item
- quantity
- salesperson

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

Understanding CPT Schema and Data

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

```
<WL_HOME>/samples/<liquiddata>/buildQuery/db-cptco
```

The project file is `coCPTSample`.

Sample CPT Schema

The CPT schema shown in Listing 9-1 (`coCptSample2.xsd`) is from the DB-CPTCO sample. It is located in the Liquid Data repository `schemas` directory.

```
<WL_HOME>/samples/domains/<liquiddata>/ldrepository/schemas
```

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

Listing 9-1 DB-CPTCO Sample CPT Schema (`coCptSample2.xsd`)

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

Components of the sample schema shown in Listing 9-1 include:

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

Declares a namespace `cocpt` associated with the URI.

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

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

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

Declares `CustOrder` as the schema root element.

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

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

Sample XML Data Stream

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

Listing 9-2 DB-CPTCO Sample XML Data Stream (`coCptSample2.xml`)

```
<?xml version="1.0" encoding="UTF-8"?>
<cocpt:CustOrder xmlns:cocpt="urn:schemas-bea-com:ld-cocpt"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:schemas-bea-com:ld-cocpt
      coCptSample2.xsd">
   <CUSTOMER_ORDER>
      <CUSTOMER_ID>CUSTOMER_1</CUSTOMER_ID>
      <NEW_ORDER_LINE_ITEM>
        <PRODUCT_NAME>RBBC01</PRODUCT_NAME>
        <QUANTITY>1000</QUANTITY>
        <PRICE>20</PRICE>
      </NEW_ORDER_LINE_ITEM>
      <NEW_ORDER_LINE_ITEM>
        <PRODUCT_NAME>CS2610</PRODUCT_NAME>
        <QUANTITY>1000</QUANTITY>
        <PRICE>20</PRICE>
      </NEW_ORDER_LINE_ITEM>
   </CUSTOMER_ORDER>
</cocpt:CustOrder>
```

The DB-CPTCO sample XML file is located in the Liquid Data repository `xml_files` directory.

```
<WL_HOME>/samples/domains/<liquiddata>/ldrepository/schemas
```

Components in the sample XML data stream shown in Listing 9-2 include:

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

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

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

Declares the standard XML schema instance URI.

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

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

Notes on Hand-Crafting CPT XQueries

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

- Unique Namespace
- XQuery of type element Declaration

Unique Namespace

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

XQuery of type element Declaration

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

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

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

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

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

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

Creating a Complex Parameter Type

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

Step 1. Create a CPT Schema

Step 2. Create Your Runtime Source

Step 3. Define Your CPT in the Administration Console

Step 4. Build Your Query

Step 5. Run Your Query

Step 1. Create a CPT Schema

Create a schema that models the runtime source. See the "Sample CPT Schema" on page 9-3 for a small schema example.

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

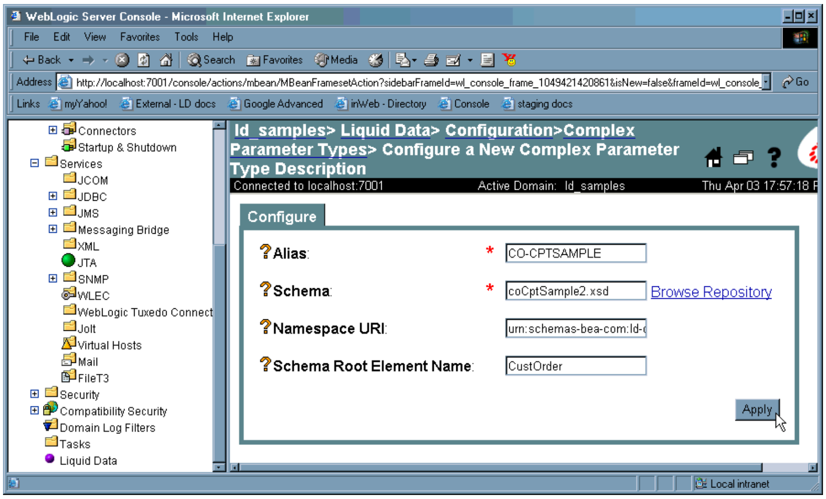
Step 2. Create Your Runtime Source

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

Step 3. Define Your CPT in the Administration Console

Through the Liquid Data node of the WebLogic Administration Console, define a complex parameter type. for a detailed procedure, see Configuring Access to Complex Parameter Types in the *Administration Guide*.

Figure 9-1 Creating a Complex Parameter Type in the Administration Console


WebLogic Server Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:7001/console/actions/mbean/MBeanFramesetAction?sidebarFrameId=wl_console_frame_1049421420861&isNew=false&frameId=wl_console_
Go
Links myYahoo! External - LD docs Google Advanced inWeb - Directory Console staging docs
Connectors
Startup & Shutdown
Services
JCOM
JDBC
JMS
Messaging Bridge
XML
JTA
SNMP
WLEC
WebLogic Tuxedo Connect
Jolt
Virtual Hosts
Mail
FileT3
Security
Compatibility Security
Domain Log Filters
Tasks
Liquid Data
ld_samples> Liquid Data> Configuration>Complex Parameter Types> Configure a New Complex Parameter Type Description
Connected to localhost:7001 Active Domain: ld_samples Thu Apr 03 17:57:18
Configure
Apply
Local intranet

Alias: *	CO-CPTSAMPLE
Schema: *	coCptSample2.xsd Browse Repository
Namespace URI:	urn:schemas-bea-com:ld-c
Schema Root Element Name:	CustOrder

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

Step 4. Build Your Query

Create your query either using the Data View Builder or by hand. See "Key Concepts of Query Building" on page 1-3 for information on designing queries in Liquid Data.

Step 5. Run Your Query

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

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

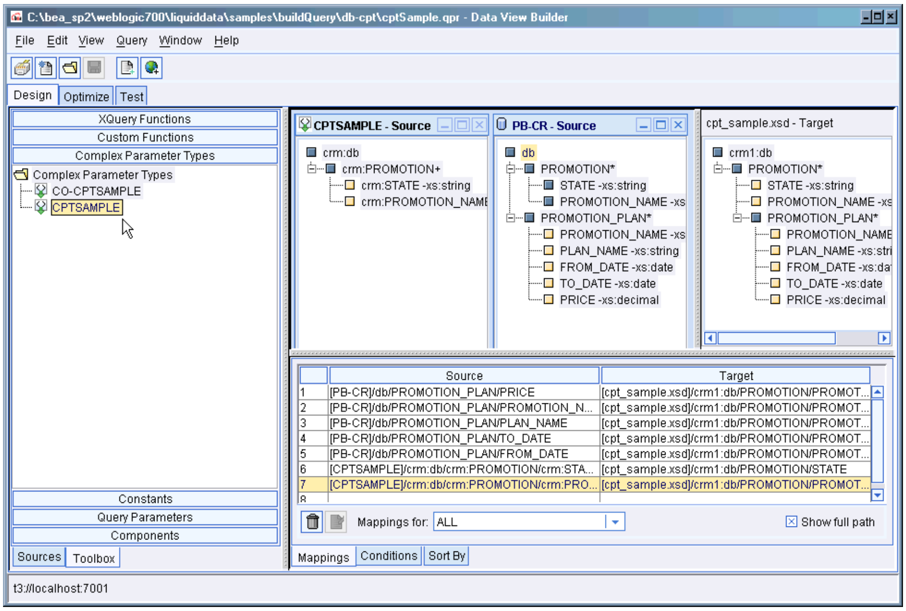
The Liquid Data DB-CPT sample `cptSample.qpr` uses a CPT to supply a promotion plan name for a given state. The sample is installed in the following directory:

```
<WL_HOME>/samples/<liquiddata>/buildQuery/db-cptco
```

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

Figure 9-2 shows the DB-CPT project. Notice that there are two complex parameter types available for use. `CPTSAMPLE` is the complex parameter type used in the query.

Figure 9-2 DB-CPT Project (CPTSAMPLE.QPR) with Complex Parameter Types Displayed



	Source	Target
1	[PB-CR]/db/PROMOTION_PLAN/PRICE	[cpt_sample.xsd]/crm1:db/PROMOTION/PROMOT...
2	[PB-CR]/db/PROMOTION_PLAN/PROMOTION_N...	[cpt_sample.xsd]/crm1:db/PROMOTION/PROMOT...
3	[PB-CR]/db/PROMOTION_PLAN/PLAN_NAME	[cpt_sample.xsd]/crm1:db/PROMOTION/PROMOT...
4	[PB-CR]/db/PROMOTION_PLAN/TO_DATE	[cpt_sample.xsd]/crm1:db/PROMOTION/PROMOT...
5	[PB-CR]/db/PROMOTION_PLAN/FROM_DATE	[cpt_sample.xsd]/crm1:db/PROMOTION/PROMOT...
6	[CPTSAMPLE]/crm:db/crm:PROMOTION/crm:STA...	[cpt_sample.xsd]/crm1:db/PROMOTION/STATE
7	[CPTSAMPLE]/crm:db/crm:PROMOTION/crm:PRO...	[cpt_sample.xsd]/crm1:db/PROMOTION/PROMOT...
8		

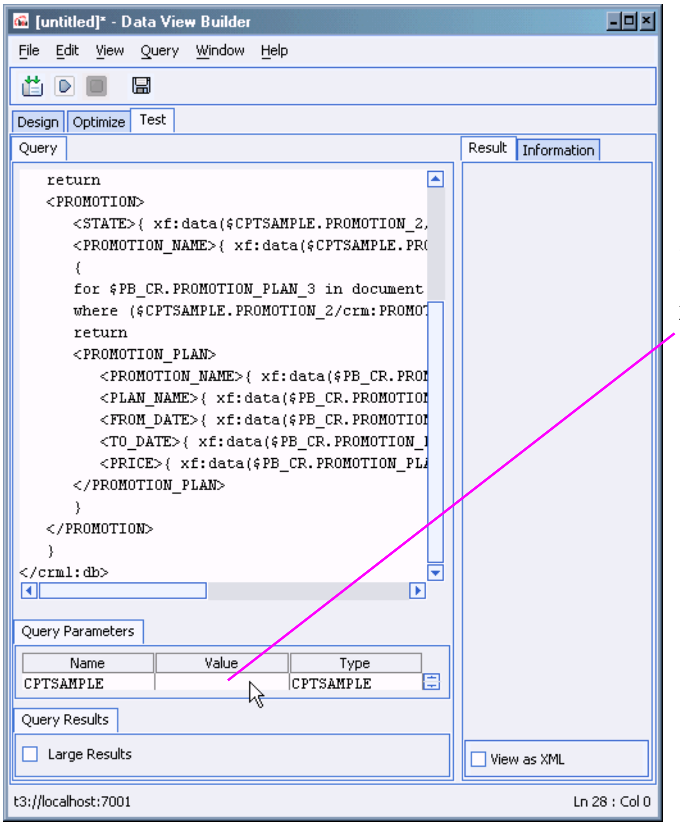
Mappings for: ALL

☒ Show full path

To test your query the name of an XML data file that is modeled on the CPT schema must be entered in the Data View Builder (see Figure 9-3). To locate the Liquid Data sample XML file click on the Value field to open the Liquid Data file browser to the following directory:

```
<WL_HOME>/samples/domains/<liquiddata>/ldrepository/xml_files
```

Figure 9-3 DB-CPT Project in Test Mode



```
    return
    <PROMOTION>
       <STATE>{ xf:data($CPTSAMPLE.PROMOTION_2,
       <PROMOTION_NAME>{ xf:data($CPTSAMPLE.PRO
       {
       for $PB_CR.PROMOTION_PLAN_3 in document
       where ($CPTSAMPLE.PROMOTION_2/crm:PROMOT
       return
       <PROMOTION_PLAN>
          <PROMOTION_NAME>{ xf:data($PB_CR.PROM
          <PLAN_NAME>{ xf:data($PB_CR.PROMOTIO
          <FROM_DATE>{ xf:data($PB_CR.PROMOTIO
          <TO_DATE>{ xf:data($PB_CR.PROMOTION_I
          <PRICE>{ xf:data($PB_CR.PROMOTION_PLA
       </PROMOTION_PLAN>
       }
    </PROMOTION>
    }
</crml:db>
```

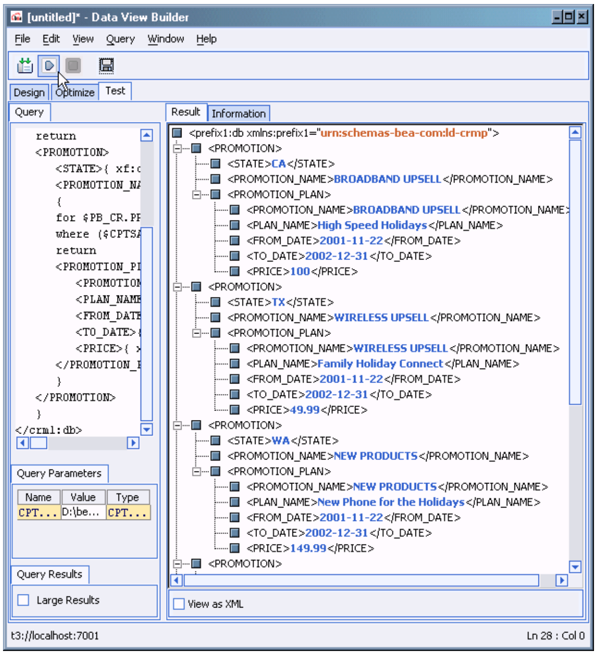
Name	Value	Type
CPTSAMPLE		CPTSAMPLE

Supply the CPT data source file name from the repository xml_files directory:

`crm-p-cptSample.xml`

Figure 9-4 shows the DB-CPT (`CPTSAMPLE.QPR`) project when the query is run.

Figure 9-4 DB-CPT Project (CPTSAMPLE.QPR) in Run Mode



```
    return
    <PROMOTION>
       <STATE>{ xf:c
       <PROMOTION_NA
       {
       for $PB_CR.PF
       where ($CPTSA
       return
       <PROMOTION_PI
          <PROMOTION
          <PLAN_NAME
          <FROM_DATE
          <TO_DATE>{
          <PRICE>{ x
       </PROMOTION_F
       }
    </PROMOTION>
    }
</crml:db>
```

Name	Value	Type
CPT...	D:\be...	CPT...

```
<prefix1:db xmlns:prefix1="urn:schemas-bea-com:ld-crmp">
  <PROMOTION>
    <STATE>CA</STATE>
    <PROMOTION_NAME>BROADBAND UPSELL</PROMOTION_NAME>
    <PROMOTION_PLAN>
      <PROMOTION_NAME>BROADBAND UPSELL</PROMOTION_NAME>
      <PLAN_NAME>High Speed Holidays</PLAN_NAME>
      <FROM_DATE>2001-11-22</FROM_DATE>
      <TO_DATE>2002-12-31</TO_DATE>
      <PRICE>100</PRICE>
  <PROMOTION>
    <STATE>TX</STATE>
    <PROMOTION_NAME>WIRELESS UPSELL</PROMOTION_NAME>
    <PROMOTION_PLAN>
      <PROMOTION_NAME>WIRELESS UPSELL</PROMOTION_NAME>
      <PLAN_NAME>Family Holiday Connect</PLAN_NAME>
      <FROM_DATE>2001-11-22</FROM_DATE>
      <TO_DATE>2002-12-31</TO_DATE>
      <PRICE>49.99</PRICE>
  <PROMOTION>
    <STATE>WA</STATE>
    <PROMOTION_NAME>NEW PRODUCTS</PROMOTION_NAME>
    <PROMOTION_PLAN>
      <PROMOTION_NAME>NEW PRODUCTS</PROMOTION_NAME>
      <PLAN_NAME>New Phone for the Holidays</PLAN_NAME>
      <FROM_DATE>2001-11-22</FROM_DATE>
      <TO_DATE>2002-12-31</TO_DATE>
      <PRICE>149.99</PRICE>
  <PROMOTION>
```

t3://localhost:7001 Ln 28 : Col 0

Complex Parameter Type Query Samples

For a step by step example of building a query with a CPT, see "Example 6: Complex Parameter Type (CPT)" in *Liquid Data by Example*.

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

Instructions for running the samples can be found under "Complex Parameter Type (CPT) Sample Queries" in *Liquid Data by Example*. The examples can be found under:

- `<WL_HOME>/samples/<liquiddata>/buildQuery/db-cpt`
- `<WL_HOME>/samples/<liquiddata>/buildQuery/db-cptco`

Also see the Liquid Data Samples page for information on other available query samples.

Accessing SQL Calls: Stored Procedures and SQL Queries

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

You can also expose any query from the database as a data source. You expose these "SQL Calls" to Liquid Data through a SQL Call Description File. This chapter describes how to define stored procedures and SQL queries in a SQL Call Description File, and contains the following sections:

- Defining Stored Procedures to Liquid Data
- SQL Call Description File
- Rules for Specifying SQL Call Description Files
- Sample SQL Call Description Files
- Stored Procedure Support by Database
- Using Stored Procedures in Queries

For an example and a demo of defining a stored procedure and using it in a query, see "Example: Defining and Using a Customer Orders Stored Procedure" on page 10-34. For information on samples installed with Liquid Data, including a stored procedure sample and a SQL Call example, see "Samples Installed with Liquid Data" in *Liquid Data by Example*.

Defining Stored Procedures to Liquid Data

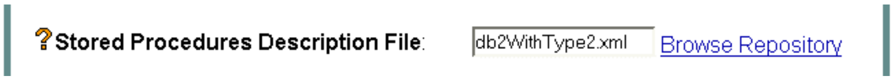
To use stored procedures in Liquid Data, you must create a SQL Call Description File. The SQL Call Description File is an XML schema file that defines the types and the functions for a set of stored procedures. For details on defining a SQL Call Description File, see "SQL Call Description File" on page 10-3 and "Rules for Specifying SQL Call Description Files" on page 10-8. For database-specific information, see "Stored Procedure Support by Database" on page 10-27.

To Define Stored Procedures to Liquid Data

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

1. Create your stored procedures in the underlying database, if they do not already exist. For details about Liquid Data support of stored procedures for your database, see "Stored Procedure Support by Database" on page 10-27.
2. In the WebLogic Console, create a JDBC Connection Pool to access your database, if one does not already exist.
3. In the WebLogic Console, create a JDBC Data Source for the connection pool created in the previous step.
4. Create a SQL Call Description File for your stored procedures and save it to the `sql_calls` directory of the Liquid Data repository. For details, see "SQL Call Description File" on page 10-3 and "Rules for Specifying SQL Call Description Files" on page 10-8.
5. In the WebLogic Administration Console (to access the Liquid Data Console, click the Liquid Data link at the bottom of the list on the WebLogic Administration Console), click the Data Sources tab.
6. Click the Relational Databases tab.
7. Click the Configure a New Relational Data Source Description Link (or open an existing Data Source to modify it).
8. If you are creating a new data source, enter values for Name, Data Source Name, and Schema fields in the Configure Relational Data Source Description screen. For more details on configuring relational data sources, see Configuring Access to Relational Databases in the *Administration Guide*.
9. In the Configure Relational Data Source Description screen, specify a SQL Call Description File by clicking the Browse Repository link next to the SQL Call Description File field.

10. In the Repository Browser, select the file you created containing your stored procedure definitions. After making your selection, click the Select button.



Stored Procedures Description File: db2WithType2.xml Browse Repository

11. In the Configure Relational Data Source Description screen, click the Apply button to save your Data Source definition.

12. Check the WebLogic Server log file for any errors, and correct them as necessary.

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

SQL Call Description File

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

- Basic Structure
- Schema Definition File for SQL Call Description File
- Element and Attribute Reference for SQL Call Description File
- Supported Datatypes

For sample SQL Call Description Files, see "Sample SQL Call Description Files" on page 10-17.

Basic Structure

The SQL Call Description File has the following main sections:

- Type Definitions
- Function Definitions

Type Definitions

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

Function Definitions

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

Schema Definition File for SQL Call Description File

The following is the schema definition file for the SQL Call Description File:

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

Element and Attribute Reference for SQL Call Description File

Table 10-1 lists and describes the elements and attributes of the SQL Call Description File.

Table 10-1 SQL Call Description File XML elements and descriptions

Element	Attribute	Description
`<definitions>`	`targetNameSpace`	The namespace declared for the stored procedures.
`<types>`		Declares any primitive or complex data types used in the stored procedure.
`<functions>`		Contains the function definitions for all of the stored procedures represented in this file.
`<function>`		Function definition for a single stored procedure.
	`name`	Name of the stored procedure as defined in the database. If the stored procedure is part of a package, the name is the fully qualified name of the stored procedure (for example, `packagename.sp_name`). If you are using procedure groups in Sybase or Microsoft SQL Server, see "Rules for Procedure Names Containing a Semi-Colon" on page 10-10.
	`return_type`	Return type of the stored procedure. The type is defined in the `<types>` element of this file. Note that this type differs from the type which the stored procedure returns. If you are hand-coding your own XQueries, you must perform a function signature transformation; for details, see "Rules for Transforming the Function Signature When Hand Writing an XQuery" on page 10-14.

Table 10-1 SQL Call Description File XML elements and descriptions

Element	Attribute	Description
`<sql_statement>`		Contains the SQL statement text for a query against the data source. The query can then be used as a data source.
`<argument>`		Contains the argument declarations for the inputs and/or outputs of the stored procedure.
	`label`	The name of the argument input or output. This name is used in queries and is displayed in clients such as the Data View Builder.
	`type`	Type of the argument. The type can be one of the types listed in "Supported Datatypes" on page 10-8, or it can be a complex type declared in the SQL Call Description File.
	`mode`	Lists whether the argument is part of the input, output, or both. Possible values are: • `input_only` • `output_only` • `input_output`
`<presentation group>`		Currently not supported.
`<description>`		Comment text describing the stored procedures used in the SQL Call Description File.

Note: An element within the `types` definition with a name specified with `name="return_value"` is reserved to specify the return value from a function or a procedure. For an example, see "Example 2: Type Definition with Simple Return Value" on page 10-11.

Supported Datatypes

The stored procedures you define in the SQL Call Description Files must use the XML data types shown in Table 10-2. You must map the database data types to one of the types in this table. For data type support by database, see "Stored Procedure Support by Database" on page 10-27.

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

Table 10-2 XML data types and their Java equivalents for SQL Call Description Files

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

For JDBC and database-specific type mapping, see "Supported Datatypes" in the *XQuery Reference Guide*.

Rules for Specifying SQL Call Description Files

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

- Rules for Element and Attribute Names
- Rules for Procedure Names Containing a Semi-Colon
- Rules and Examples of <type> Declarations to Use in the <function> return_type Attribute
- Rules for the mode Attribute output_only <argument> Element

- Rules for Transforming the Function Signature When Hand Writing an XQuery

Rules for Element and Attribute Names

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

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

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

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

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

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

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

For some notes on relational database object names and how to specify them so they can be used in an XQuery, see "Relational Databases" on page 3-2.

Rules for Procedure Names Containing a Semi-Colon

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

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

When you specify these procedures in the SQL Call Description File, use the database name (the name with the semicolon character). When you use these procedure names in an XQuery, however, you must substitute an underscore character (_) for the semicolon character. The Data View Builder automatically substitutes the underscore character for the semicolon character in the XQuery it generates.

For example, consider the following definition for a stored procedure in a SQL Call Description File:

```
<function name="MY_SP;2" return_type="Results">
       <argument label="COLUMN_123" mode="input_only"
              type="xs:string"/>
       <argument label="ANOTHER_COLUMN" mode="output_only"
                            type="xs:int"/>
</function>
```

When you reference this function in an XQuery, it is referred to as follows:

```
MY_SP_2
```

Rules and Examples of <type> Declarations to Use in the <function> return_type Attribute

The `return_type` attribute of the `<function>` element specifies the complex type for the stored procedure. The complex type must be declared in the `<types>` section of the SQL Call Description File. For example, the following element opening tag shows a function named `myFunction` with a `return_type` of `myReturnType`:

```
<function name="myFunction" return_type="myReturnType" >
```

The return type `myReturnType` must be declared in the `<types>` section. The type must contain the actual return value of the stored procedure (if it has a return value) and the row set definitions (if applicable). The row set definitions must appear in the order in which the stored procedure returns them.

When a stored procedure returns a primitive type, you must declare the primitive type using the `return_value` keyword for the `name` attribute. For an example of this, see "Example 2: Type Definition with Simple Return Value" on page 10-11.

Example 1: Type Definition with No Return Value

The following is a type definition for a stored procedure that has no return value and returns no row sets.

```
<types>
      <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
             <xs:element name="EmptyOutput">
                    <xs:complexType>
                          <xs:sequence>
                          </xs:sequence>
                    </xs:complexType>
             </xs:element>
      </xs:schema>
</types>
```

Use a similar type definition in a stored procedure that does not have any return value. This `EmptyOutput` type definition is required for all stored procedures and functions that do not return anything.

Example 2: Type Definition with Simple Return Value

The following is a type definition for a stored procedure that has a simple return value (`xs:integer`) and returns no row sets.

```
<types>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="SimpleOutput">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="return_value"
             type="xs:integer" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>
</types>
```

Use a similar type definition with a stored procedure that returns a status code or a single value.

Example 3: Type Definition for Complex Row Set Type

The following is a type definition for a stored procedure that returns a row set, which is a complex type.

```
  <types>
        <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
               <xs:element name="customerTable">
                      <xs:complexType>
                             <xs:sequence>
                                    <xs:element name="CUSTOMER"
minOccurs="0"
                                                  maxOccurs="unbounded" >
                                           <xs:complexType>
                                                  <xs:sequence>
                                                         <xs:element
name="C_NAME"

type="xs:string"/>
                                                         <xs:element
name="C_ACCTBAL"

type="xs:decimal"/>
                                                  </xs:sequence>
                                           </xs:complexType>
                             </xs:sequence>
                      </xs:complexType>
               </xs:element>
        </xs:schema>
  </types>
```

Use a similar type definition with a stored procedure that returns a result set (for example, in Sybase, Microsoft SQL Server, or DB2).

Example 4: Type Definition with Complex Return Value

The following is a type definition for a stored procedure that returns a complex type. Assume that the `customerTable` complex type (shown in the previous example) is defined in the same Stored Procedure Definition file.

```
<types>
      <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
             <xs:element name="return_value" type="customerTable">
             </xs:element>
      </xs:schema>
</types>
```

Use a similar type definition with an Oracle stored procedure that returns a cursor with a complex type.

Example 5: Type Definition with Simple Return Value and Two Row Sets

The following is a type definition for a stored procedure that has a simple return value (`xs:integer`) and returns two row sets.

```
  <types>
        <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
......
......{-- Definitions for complex types customerTable and
          ordersTable go here --}
......
               <xs:element name="OutputName">
                      <xs:complexType>
                             <xs:sequence>
                                    <xs:element name="return_value"
                                                  type="xs:integer" />
                                    <xs:element ref="customerTable" />
                                           {-- customerTable defined
above--}
                                    <xs:element ref="ordersTable" />
                                           {-- ordersTable defined above --}
                             </xs:sequence>
                      </xs:complexType>
               </xs:element>
```

```
      </xs:schema>
</types>
```

Rules for the mode Attribute output_only <argument> Element

If you define a function that has an `<argument>` element that has the mode `output_only`, then you need only reference the type definition in the function definition. The following example references the `customerTable` type (defined in the `<types>` section of the SQL Call Description File, as described in "Example 3: Type Definition for Complex Row Set Type" on page 10-12). Assume that the `customerTable` type maps to a cursor returned from an Oracle stored procedure.

```
<function name="GetAllCustomersByState" return_type="EmptyOutput">
       <argument label="state" mode="input_only" type="xs:string"/>
       <argument label="CustomersOutput" mode="output_only"
                            type="customerTable"/>
</function>
```

Rules for Transforming the Function Signature When Hand Writing an XQuery

There are two issues to remember when hand crafting an XQuery that accesses a stored procedure:

- Namespace Declaration
- Function Transformation

Namespace Declaration

All queries that access stored procedures must have a unique namespace with a URI of the of the following form:

urn:*<Liquid_Data_Relational_data_source_name>*

Declare the namespace in the query prolog. The namespace declaration has the following syntax:

namespace *<alias>*="*<URI>*"

For example:

```
namespace SY_WL_NS="urn:SY-WL"
```

You can then access the stored procedure using the namespace alias and the name of the stored procedure object. For example:

```
SY_WL_NS:wireless.dbo.RetAndOpParamTransformation("CUSTOMER_11")
```

Function Transformation

For stored procedures that return both a return value (for example, an integer return value) and have `output` or `input_output` parameters, the function signature in the SQL Call Description File is different from the signature that is used to write queries that access the stored procedure. If you look at the schema that displays in the Stored Procedures palette of the Data View Builder, you will see the transformed signature.

The transformed signature combines the return value and any `output` or `input_output` parameters.

For example, consider an example using a Sybase stored procedure with the following signature:

```
create proc RetAndOpParamTransformation (@custidPattern varchar(64),
@custCount numeric(10) output )
 as
      select @custCount = count(*)  from CUSTOMER
      where CUSTOMER.CUSTOMER_ID Like '%' +  @custidPattern  + '%'
RETURN 1
```

The following is the SQL Call Description File for this stored procedure:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions>
 <types>
       <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<!-- The stored procedure returns an integer, which is mapped as the
return_value, a reserved element name for stored procedure with a return.
-->
              <xs:element name="RetAndOpParamTransformation">
                     <xs:complexType>
                            <xs:sequence>
                                   <xs:element name="return_value"
                                 type="xs:integer"/>
                            </xs:sequence>
                     </xs:complexType>
              </xs:element>
       </xs:schema>
  </types>

<!-- The stored procedure signature mapping. The element
RetAndOpParamTransformation wraps the return_value of the stored procedure.
This stored procedure has custidPattern as an input parameter. custCount is
defined as an output parameter of type integer (because it returns an integer
count).
-->
  <functions>
       <function name="wireless.dbo.RetAndOpParamTransformation"
                  return_type="RetAndOpParamTransformation">
              <argument label="custidPattern" mode="input_only"
                     type="xs:string"/>
              <argument label="custCount" mode="output_only"
                     type="xs:integer"/>
              <presentation group="Sample to show transformation of
return_value and output prameter in a stored procedure"/>
       </function>
  </functions>
</definitions>
```

Because this stored procedure has a return value and an output parameter, the output of the function is transformed to the following schema:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="RetAndOpParamTransformation">
       <xsd:complexType>
              <xsd:sequence>
                     <xsd:element name="return_value"
                           type="xsd:integer"/>
                     <xsd:element name="custCount" type="xsd:integer"/>
              </xsd:sequence>
       </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

where the `return_value` is the return from the stored procedure and `custCount` is the output parameter. If you view this stored procedure in the Data View Builder, you will see the transformed schema.

The following is a query against this stored procedure:

```
namespace SY-WL-NS = "urn:SY-WL"

let $SY_WL_SP_Return :=
SY-WL-NS:wireless.dbo.RetAndOpParamTransformation("CUSTOMER_11")
return

<RetAndOpParamTransformation>
 <return_value>{
xf:data($SY_WL_SP_Return/RetAndOpParamTransformation/return_value) }
 </return_value>
   <custCount>{
xf:data($SY_WL_SP_Return/RetAndOpParamTransformation/custCount) }
   </custCount>
</RetAndOpParamTransformation>
```

In this query, the XPath expressions for `return_value` and for `custCount` have the same parent element.

Sample SQL Call Description Files

This section shows several sample SQL Call Description Files. For simplicity and readability, each of the examples shown defines a single stored procedure and its supporting type; your SQL Call Description Files can define multiple stored procedures and multiple types. For information on samples installed with Liquid Data, including a stored procedure sample and a SQL Call example, see "Samples Installed with Liquid Data" in *Liquid Data by Example*.

This section includes the following examples:

- DB2 Simple input_only, output_only, and input_output Example
- Oracle Cursor Output Parameter Example
- DB2 Multiple Result Set Example
- Oracle Cursor as return_value
- Oracle SQL Statement With Subquery

DB2 Simple input_only, output_only, and input_output Example

The following SQL Call Description File describes a DB2 stored procedure that returns simple data types with `input_only`, `output_only`, and `input_output` parameters.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions>
  <types>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name="EmptyOutput">
        <xs:complexType>
          <xs:sequence>
          </xs:sequence>
        </xs:complexType>
    </xs:schema>
  </types>
  <functions>
      <!-- given a customer id if valid, returns as output
             parameters all  the customer details -->
      <function name="CALLINCALLOUT" return_type="EmptyOutput">
        <argument label="custid" mode="input_only" type="xs:string"/>
        <argument label="fname" mode="output_only" type="xs:string"/>
        <argument label="lname" mode="output_only" type="xs:string"/>
        <argument label="telephoneNumber" mode="output_only"
                   type="xs:long"/>
        <argument label="customerSinceAsData" mode="output_only"
                   type="xs:date"/>
        <argument label="customerSinceAsTimeStamp"
                   mode="output_only" type="xs:dateTime"/>
      <presentation group="DB2 stored procedures"/>
             </function>
  </functions>
</definitions>
```

The following is the stored procedure signature for this SQL Call Description File. This signature creates a procedure that has one simple input and returns five simple outputs.

```
CREATE PROCEDURE DB2ADMIN.CALLINCALLOUT (
    IN CUSTID varchar(64), OUT FNAME varchar(4000),
    OUT LNAME varchar(4000), OUT TELEPHONENUMBER bigint,
    OUT CUSTOMERSINCE date, OUT CUSTOMERSINCE1 timestamp )
EXTERNAL NAME
  '"DB2ADMIN".SQL30205005750980:db2test.CallInCallOut.callInCallOut'
SPECIFIC DB2ADMIN.CALLINCALLOUT
RESULT SETS 0
LANGUAGE JAVA
PARAMETER STYLE JAVA
NOT DETERMINISTIC
FENCED NO
DBINFO NULL
CALL MODIFIES SQL DATA
```

Oracle Cursor Output Parameter Example

The following SQL Call Description File describes an Oracle stored procedure that returns an output parameter as a cursor.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions>
  <types>
      <xs:element name="OutCursor">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="CUSTOMER" minOccurs="0"
                        maxOccurs="unbounded">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="C_CUSTKEY" type="xs:integer"/>
                  <xs:element name="C_FNAME" type="xs:string"/>
                  <xs:element name="C_LNAME" type="xs:string"/>
                  <xs:element name="C_STATE" type="xs:string"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Output">
        <xs:complexType>
          <xs:sequence/>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </types>
  <functions>
    <function name="TEST_PACKAGE.GETCUSTOMER" return_type="Output">
      <argument label="CUSTID" mode="input_only" type="xs:string"/>
      <argument label="customer_OUT" mode="output_only"
                type="OutCursor"/>
      <presentation group="OR-TEST stored procedures"/>
    </function>
```

```
 </functions>
</definitions>
```

The following is the stored procedure signature for this SQL Call Description File. This signature creates a procedure that returns a cursor as an output parameter.

```
create or replace package test_package as
-- Stored procedure that returns a cursor as an output parameter
  procedure getCustomer
     ( CUSTOMERID IN VARCHAR, cust_cursor1 OUT ref_cursor );
end test_package ;
```

DB2 Multiple Result Set Example

The following SQL Call Description File describes a DB2 stored procedure that returns multiple result sets.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions>
 <types>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="CustomerAndOrders">
     <xs:complexType>
      <xs:sequence>
        <xs:element ref="resultSetCustomer"/>
        <xs:element ref="resultSetCustomerOrders"/>
      </xs:sequence>
     </xs:complexType>
   </xs:element>
   <xs:element name="resultSetCustomer">
     <xs:complexType>
      <xs:sequence>

       <xs:element name="customerRow" minOccurs="1"
                maxOccurs="unbounded">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="C_CUSTKEY" type="xs:string"/>
          <xs:element name="C_FNAME" type="xs:string"/>
          <xs:element name="C_LNAME" type="xs:string"/>
          <xs:element name="C_STATE" type="xs:string"/>
```

```
            <xs:element name="C_SINCE" type="xs:date"/>
            <xs:element name="C_TELEPHONENO" type="xs:long"/>
           </xs:sequence>
          </xs:complexType>
         </xs:element>
        </xs:sequence>
       </xs:complexType>
      </xs:element>

      <xs:element name="resultSetCustomerOrders">
       <xs:complexType>
        <xs:sequence>
         <xs:element name="orderRow" minOccurs="1"
                   maxOccurs="unbounded">
          <xs:complexType>
           <xs:sequence>
            <xs:element name="C_CUSTKEY" type="xs:string"/>
            <xs:element name="CO_ORDERKEY" type="xs:string"/>
            <xs:element name="CO_ORDERDATE" type="xs:date"/>
            <xs:element name="CO_SHIPMETHOD" type="xs:date"/>
            <xs:element name="CO_TOTALORDERAMT" type="xs:decimal"/>
           </xs:sequence>
          </xs:complexType>
         </xs:element>
        </xs:sequence>
       </xs:complexType>
      </xs:element>

     </xs:schema>
    </types>
    <functions>
         <!-- result one returns a customer, result 2 has the
              orders for that customer -->
     <function name="CALLMULTIPLERESULTSET"
                   return_type="CustomerAndOrders">
       <argument label="custid" mode="input_only" type="xs:string"/>
      <presentation group="DB2 stored procedures"/>
     </function>
```

```
  </functions>
</definitions>
```

The following is the stored procedure signature for this SQL Call Description File. This signature creates a procedure that returns multiple result sets.

```
CREATE PROCEDURE DB2ADMIN.CALLMULTIPLERESULTSET (
    IN CUSTID varchar(64) )
EXTERNAL NAME
     '"DB2ADMIN".SQL30206110348560:db2test.CallMultipleResultSet.callMulti
pleResultSet'
SPECIFIC DB2ADMIN.CALLMULTIPLERS
RESULT SETS 2
LANGUAGE JAVA
PARAMETER STYLE JAVA
NOT DETERMINISTIC
FENCED NO
DBINFO NULL
CALL MODIFIES SQL DATA
```

Oracle Cursor as return_value

The following SQL Call Description File describes an Oracle stored procedure that returns a cursor as a `return_value`.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions>
  <types>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name="Output_TEST_PACKAGE.GETCUSTOMERBYID">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="return_value">
              <xs:complexType>
                <xs:sequence>
                  <xs:element maxOccurs="unbounded" minOccurs="0"
                              name="customer">
                    <xs:complexType>
                      <xs:sequence>
                        <xs:element name="FIRST_NAME" type="xs:string"/>
                    <xs:element name="LAST_NAME" type="xs:string"/>
                    <xs:element name="CUSTOMER_ID" type="xs:string"/>
                    <xs:element name="STATE" type="xs:string"/>
                    <xs:element name="ZIPCODE" type="xs:string"/>
                    <xs:element name="CITY" type="xs:string"/>
                    <xs:element name="STREET_ADDR2"
                                type="xs:string"/>
                    <xs:element name="STREET_ADDR1"
                                type="xs:string"/>
                    <xs:element name="CUSTOMER_SINCE"
                                type="xs:dateTime"/>
                    <xs:element name="EMAIL_ADDRESS"
                                type="xs:string"/>
                    <xs:element name="TELEPHONE_NUMBER"
                                type="xs:string"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  </xs:schema>
 </types>
 <functions>
  <function name="TEST_PACKAGE.GETCUSTOMERBYID"
            return_type="Output_TEST_PACKAGE.GETCUSTOMERBYID">
    <argument label="CUSTID" mode="input_only" type="xs:string"/>
  </function>
 </functions>
</definitions>
```

The following is the stored procedure signature for this SQL Call Description File. This signature creates a procedure that returns a cursor.

```
create or replace package body test_package  as
-- SP that returns a cursor
  FUNCTION  getCustomerByID (custID varchar)
      RETURN CUST_CURSOR IS cur CUST_CURSOR;
  BEGIN
    open  cur for
        select first_name, last_name, customer_id, state,
               zipCode,city, street_address2, street_address1,
               customer_since, email_address, telephone_number
        from wireless.customer
        where customer_id = custID;
    return cur;
  END;
end test_package ;
```

Oracle SQL Statement With Subquery

The following SQL Call Description File defines a SQL statement with a subquery and syntax specific to Oracle.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions>
<types>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

 <xs:element name="Customers">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="resultSetCustomer"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name="resultSetCustomer">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="customerRow" minOccurs="0"maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="CUSTOMER_ID" type="xs:string"/>
       <xs:element name="ORDER_ID" type="xs:string"/>
       <xs:element name="LINE_ID" type="xs:string"/>
```

```
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>

</types>

  <functions>


    <function name="GetOrderInfoSQL" return_type="Customers" >
    <sql_statement>
    SELECT t1.customer_id, t2.order_id, t3.line_id from
      customer t1,
      (SELECT * from customer_order where customer_id != 'CUSTOMER_1' ) t2,
      customer_order_line_item t3
      where t1.customer_id = ? and t2.customer_id (+) = t1.customer_id and
t3.order_id(+)=t2.order_id
    </sql_statement>
      <argument label="customer_id" mode="input_only" type="xs:string"/>
      <presentation group="Oracle SQL Call"/>
    </function>

  </functions>

</definitions>
```

Stored Procedure Support by Database

This section lists stored procedure support by database vendor. Each vendor supports the data types supported in their respective databases. The following databases are supported:

- Oracle
- Microsoft SQL Server
- Sybase
- IBM DB2
- Informix

Oracle

Table 10-3 describes the stored procedure support for Oracle databases. Table 10-4 describes Oracle stored procedures return values.

Table 10-3 Oracle Stored Procedure parameter support

Parameter Mode	Data Types Supported	Notes and Restrictions
`input_only`	Only database data types that you can map to one of the Liquid Data primitive types defined in "Supported Datatypes" on page 10-8.	• The PL/SQL `%TYPE` definitions must be translated to the XML schema types defined in "Supported Datatypes" on page 10-8.
`output_only`	• A Cursor • Only database data types that you can map to one of the Liquid Data primitive types defined in "Supported Datatypes" on page 10-8.	
`input_output`	Only database data types that you can map to one of the Liquid Data primitive types defined in "Supported Datatypes" on page 10-8.	

Table 10-4 Oracle Stored Procedure returned values support

Return Value	Types Supported
Primitive type	An primitive type such as an integer, a string, etc.
Return cursor	See Table 10-3.

Microsoft SQL Server

Table 10-5 describes the stored procedure support for Microsoft SQL Server databases. Table 10-6 describes Microsoft SQL Server stored procedures return values.

Table 10-5 Microsoft SQL Server Stored Procedure parameter support

Parameter Mode	Data Types Supported	Notes and Restrictions
`input_only`	Only database data types that you can map to one of the Liquid Data primitive types defined in "Supported Datatypes" on page 10-8.	• You must map `TINYINT` values to `xs:short` in the SQL Call Description File.
`output_only`	Only database data types that you can map to one of the Liquid Data primitive types defined in "Supported Datatypes" on page 10-8.	• You must map `TINYINT` values to `xs:short` in the SQL Call Description File.

Table 10-6 Microsoft SQL Server Stored Procedure returned values support

Return Value	Types Supported	Notes and Restrictions
Return Status code	An integer value.	
Row Set	Single or multiple result sets.	• You must map `TINYINT` values to `xs:short` in the SQL Call Description File.

If you are using procedure groups, see "Rules for Procedure Names Containing a Semi-Colon" on page 10-10 for information on mapping the procedure names to the SQL Call Description File and using the names in an XQuery.

Microsoft SQL Server parameter names begin with the @ character, but the name must appear in the SQL Call Description File without the @ character. For example, a parameter named `@myInputParameter` must be mapped as `myInputParameter`.

Sybase

Table 10-7 describes the stored procedure support for Sybase databases. Table 10-8 describes Sybase stored procedures return values.

Table 10-7 Sybase Stored Procedure parameter support

Parameter Mode	Data Types Supported	Notes and Restrictions
`input_only`	Only database data types that you can map to one of the Liquid Data primitive types defined in "Supported Datatypes" on page 10-8.	• You must map `TINYINT` values to `xs:short` in the SQL Call Description File.
`output_only`	Only database data types that you can map to one of the Liquid Data primitive types defined in "Supported Datatypes" on page 10-8.	• You must map `TINYINT` values to `xs:short` in the SQL Call Description File.

Table 10-8 Sybase Stored Procedure returned values support

Return Value	Types Supported	Notes and Restrictions
Return Status code	An integer value.	
Row Set	Single or multiple result sets.	• You must map `TINYINT` values to `xs:short` in the SQL Call Description File.

If you are using procedure groups, see "Rules for Procedure Names Containing a Semi-Colon" on page 10-10 for information on mapping the procedure names to the SQL Call Description File and using the names in an XQuery.

Sybase parameter names begin with the @ character, but the name must appear in the SQL Call Description File without the @ character. For example, a parameter named `@myInputParameter` must be mapped as `myInputParameter`.

IBM DB2

Table 10-9 describes the stored procedure support for IBM DB2 databases. Table 10-10 describes IBM DB2 stored procedures return values.

Table 10-9 IBM DB2 Stored Procedure parameter support

Parameter Mode	Data Types Supported
`input_only`	Only database data types that you can map to one of the Liquid Data primitive types defined in "Supported Datatypes" on page 10-8.
`output_only`	Only database data types that you can map to one of the Liquid Data primitive types defined in "Supported Datatypes" on page 10-8.
`input_output`	Only database data types that you can map to one of the Liquid Data primitive types defined in "Supported Datatypes" on page 10-8.

Table 10-10 IBM DB2 Stored Procedure returned values support

Return Value	Types Supported
Primitive type	An primitive type such as an integer, a string, etc.
Row Set	Single or multiple result sets.

Informix

Table 10-11 describes the stored procedure support for Informix databases. Table 10-12 describes Informix stored procedures return values.

Table 10-11 Informix Stored Procedure parameter support

Parameter Mode	Data Types Supported
`input_only`	Only database data types that you can map to one of the Liquid Data primitive types defined in "Supported Datatypes" on page 10-8.

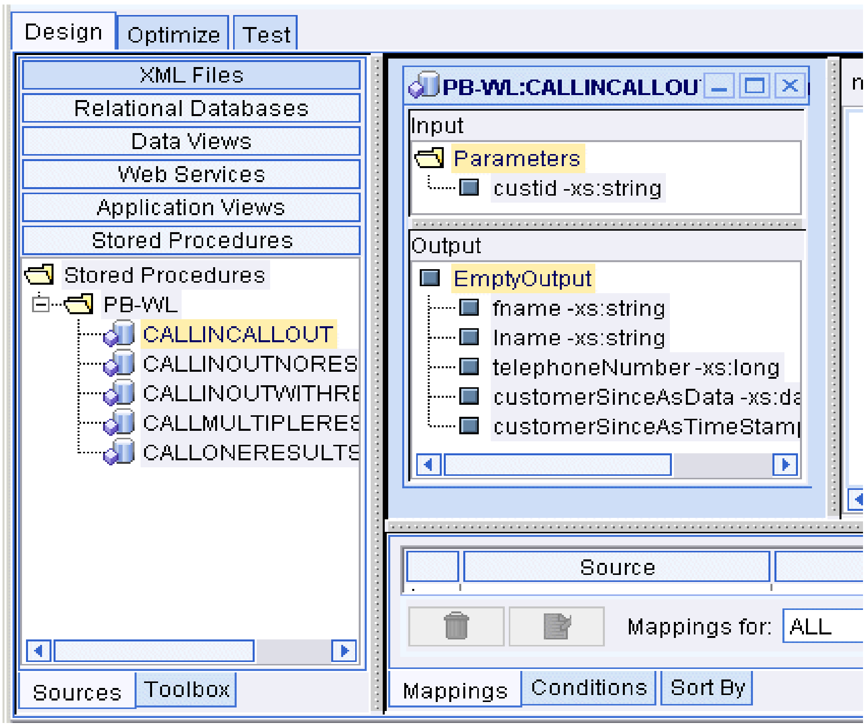
Table 10-12 Informix Stored Procedure returned values support

Return Value	Types Supported
Row Set	Single or multiple result sets.

Using Stored Procedures in Queries

You can use stored procedures to build queries in the Data View Builder just like you use other data sources. Drag and drop input elements into the inputs of the procedure and drag and drop output elements to combine with other sources or to map onto a target XML schema.

Figure 10-13 Drag and Drop Input Elements into the Elements of the Stored Procedure



This section shows an example of defining a stored procedure and then using it in a query, and is divided into the following sections:

- Define Stored Procedures to Liquid Data
- Example: Defining and Using a Customer Orders Stored Procedure

Define Stored Procedures to Liquid Data

You must define the stored procedures to Liquid Data before you can use them in queries. For details, see "To Define Stored Procedures to Liquid Data" on page 10-2. To use a stored procedure in the Data

View Builder, select Stored Procedures from the Sources tab, navigate to your stored procedure, then drag and drop it into the design workspace. You can then connect data by dragging and dropping inputs and outputs.

Example: Defining and Using a Customer Orders Stored Procedure

This example details the steps to define a stored procedure to Liquid Data and then use it in a query. This example is similar to the example installed in the following directory:

`BEA_HOME/liquiddata/samples/buildQuery/stored-procedure`

The demo in this directory includes the SQL Call Description File and a Data View Builder project file.

Business Scenario

The stored procedure in this example answers the following business question: For all orders greater than or equal to $500.00, find the number of orders and the total value of all of those orders for a given customer.

View a Demo

Stored Procedure Demo… If you are looking at this documentation online, you can click the "Demo" button to see a viewlet demo showing how to define a stored procedure and use it in a query. This demo previews the steps described in detail in the following sections.

Step 1: Create the Stored Procedure in the Database

You must have stored procedures defined in your database before you can access them through Liquid Data.

Every database has its own way of creating stored procedures. This sample uses a Pointbase database, and Pointbase uses Java stored procedures. The source code for the sample stored procedure is installed with Liquid Data in the following file:

`BEA_HOME/liquiddata/samples/buildQuery/stored-procedure/pbsp.java`

The signature for this stored procedure is created with the following SQL statements:

```
create procedure
         GetOrderInfo( IN P1 VARCHAR(20),  IN P2 INTEGER,
                       OUT P3 INTEGER, OUT P4 INTEGER)
LANGUAGE JAVA
```

```
SPECIFIC GetOrderInfo
EXTERNAL NAME  "com.bea.ldi.sample.pbsp::GetOrderInfo"
PARAMETER STYLE SQL;
```

Step 2: Create the SQL Call Description File

For details on the structure of the SQL Call Description File, see “SQL Call Description File” on page 10-3 and “Rules for Specifying SQL Call Description Files” on page 10-8.

The SQL Call Description File for this example defines an empty complex type in the `<types>` section and defines a function that returns that complex type in the `<functions>` section. The function definition contains `<argument>` elements for each input and output argument. The `<argument>` elements specify the name (`label` attribute), parameter type (`mode` attribute), and data type (`type` attribute) for each input and output of the stored procedure.

The following is a code listing of the SQL Call Description File for this example.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions>
 <types>
      <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
             <xs:element name="Results">
                    <xs:complexType>
                           <xs:sequence>
                          </xs:sequence>
                    </xs:complexType>
             </xs:element>
      </xs:schema>
 </types>
 <functions>


      <function name="GetOrderInfo" return_type="Results">
             <argument label="customer_id" mode="input_only"
                   type="xs:string"/>
             <argument label="order_amount" mode="input_only"
                   type="xs:integer"/>
             <argument label="totalsum" mode="output_only"
                   type="xs:integer"/>
             <argument label="totalorder" mode="output_only"
```

```
                    type="xs:integer"/>
              <presentation group="Pointbase stored procedures"/>
       </function>

  </functions>
</definitions>
```

Step 3: Specify the SQL Call Description File in the Liquid Data Console

Perform the following steps to specify the SQL Call Description File in the data source description:

1. In the WebLogic Administration Console (to access the Liquid Data Console, click the Liquid Data link at the bottom of the list on the WebLogic Administration Console), click the Data Sources tab.
2. Click the Relational Databases tab.
3. Select an existing relational data source and edit it or create a new relational data source.

   If you are creating a new data source, you must also configure a JDBC Connection Pool to access your database and a JDBC Data Source for the connection pool.
4. In the Configure Relational Data Source Description screen, enter values for Name, Data Source Name, and Schema fields, if they are not already entered. For more details on configuring relational data sources, see Configuring Access to Relational Databases in the *Administration Guide*.
5. In the Configure Relational Data Source Description screen, click the Browse Repository link next to the SQL Call Description File field.
6. In the Repository Browser, select the file you created containing your stored procedure definitions. After making your selection, click the Select button.
7. In the Configure Relational Data Source Description screen, click the Apply button to save your Data Source definition.

Step 4: Open the Data View Builder to See Your Stored Procedures

Start the Data View Builder and connect to the Liquid Data server. If you are already connected, run the File > Connect command to reconnect. If you configured the Stored procedure correctly, it appears in the Sources tab as one of the stored procedures.

Step 5: Use the Stored Procedure in a Query

Perform the following steps in the Data View Builder to create a query that uses the stored procedure.

1. Start a new Data View Builder project (File —> New Project).
2. Open the source and target schemas.
   - Drag and drop Source —> Stored Procedure —> PB-WL —> `getOrderInfo` into the design area.
   - Set the target schema to `getorderinfo.xsd` (in the repository).
3. Create a query parameter named `CUST_ID` of type `xs:string` for `customer_id`.
4. Drag the `CUST_ID` query parameter into the `customer_id` stored procedure input.
5. Create a numeric constant of 500 and drag it into the `order_amount` input parameter.
6. Drag the `totalsum` stored procedure output to the `totalsum` element of the target schema.
7. Drag the `totalorder` stored procedure output to the `totalorder` element of the target schema.

Step 6: Run the Query

Perform the following to run this query:

1. Click the test tab in the Data View Builder.
2. Enter a value for the `CUST_ID` query parameter. For example, enter `CUSTOMER_1`.
3. Click the Run button. The results will look similar to the following:

```
<Results>
    <totalsum>7000</totalsum>
    <totalorder>3</totalorder>
</Results>
```

Index