MarkLogic Server

Application Developer's Guide

Release 4.1
June, 2009

Last Revised: 4.1-9, January, 2011

© Copyright 2002-2011 by MarkLogic Corporation. All rights reserved worldwide.

Copyright

© Copyright 2002-2011 by MarkLogic Corporation. All rights reserved worldwide.

This Material is confidential and is protected under your license agreement.

Excel and PowerPoint are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. This document is an independent publication of MarkLogic Corporation and is not affiliated with, nor has it been authorized, sponsored or otherwise approved by Microsoft Corporation.

Contains LinguistX, from Inxight Software, Inc. Copyright © 1996-2006. All rights reserved. www.inxight.com.

Antenna House OfficeHTML Copyright © 2000-2008 Antenna House, Inc. All rights reserved.

Argus Copyright ©1999-2008 Iceni Technology Ltd. All rights reserved.

Contains Rosette Linguistics Platform 6.0 from Basis Technology Corporation, Copyright © 2004-2008 Basis Technology Corporation. All rights reserved.

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/) Copyright © 1995-1998 Eric Young (eay@cryptsoft.com). All rights reserved. Copyright © 1998-2001 The OpenSSL Project. All rights reserved. Contains software derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm. Copyright © 1991-1992, RSA Data Security, Inc. Created 1991. All rights reserved.

Contains ICU with the following copyright and permission notice:
Copyright © 1995-2010 International Business Machines Corporation and others. All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

Table of Contents

Application Developer's Guide

Copyright 2

1.0 Developing Applications in MarkLogic Server 10
1.1 Overview of MarkLogic Server Application Development 10
1.2 Skills Needed to Develop MarkLogic Server Applications 10
1.3 Where to Find Specific Information 11

2.0 Understanding Transactions in MarkLogic Server 12
2.1 Overview of Terminology 12
2.2 System Timestamps and Fragment Versioning 13
2.3 Query and Update Statements 14
2.3.1 Query Statements 14
2.3.2 Update Statements—Readers/Writers Locks 15
2.3.3 Example Scenario 16
2.4 Semi-Colon as a Transactional Separator 17
2.5 Interactions with xdmp:eval/invoke 17
2.5.1 Isolation Option to xdmp:eval/invoke 17
2.5.2 Preventing Undetectable Deadlocks 18
2.5.3 Seeing Updates From eval/invoke Later in the Transaction 20
2.6 Functions With Non-Transactional Side Effects 21
2.7 Example: Incrementing the System Timestamp 22

3.0 Loading Schemas 23
3.1 Configuring Your Database 23
3.2 Loading Your Schema 24
3.3 Referencing Your Schema 25
3.4 Working With Your Schema 25
3.5 Validating XML Against a Schema 26

4.0 Content Repair 27
4.1 General-Purpose Tag Repair 27
4.1.1 How General-Purpose Tag Repair Works 27
4.1.2 Pitfalls of General-Purpose Tag Repair 28
4.1.3 Scope of Application 30
4.1.4 Disabling General-Purpose Tag Repair 30
4.2 Empty Tag Auto-Close 30
4.2.1 What Empty Tag Auto-Close Does 31

4.2.2 How Empty Tag Auto-Close Works 32
4.2.3 How to Perform Empty Tag Auto-Close 33
4.2.4 Scope of Application 35
4.2.5 Disabling Empty Tag Auto-Close 35
4.3 Schema-Driven Tag Repair 35
4.3.1 What Schema-Driven Tag Repair Does 35
4.3.2 Loading Content with Schema-Driven Tag Repair 38
4.3.3 Scope of Application 39
4.3.4 Disabling Schema-Driven Tag Repair 39
4.4 Load-Time Default Namespace Assignment 39
4.4.1 How Default Namespace Assignments Work 40
4.4.2 Scope of Application 40
4.5 Load-Time Namespace Prefix Binding 40
4.5.1 How Load-Time Namespace Prefix Binding Works 41
4.5.2 Interaction with Load-Time Default Namespace Assignment 43
4.5.3 Scope of Application 44
4.5.4 Disabling Load-Time Namespace Prefix Binding 45
4.6 Query-Driven Content Repair 45
4.6.1 Point Repair 45
4.6.2 Document Walkers 46

5.0 Loading Documents into the Database 48
5.1 Document Formats 48
5.1.1 XML Format 49
5.1.2 Binary (BLOB) Format 49
5.1.3 Text (CLOB) Format 49
5.2 Setting Formats During Loading 50
5.2.1 Implicitly Setting the Format Based on the Mimetype 50
5.2.2 Explicitly Setting the Format with xdmp:document-load 50
5.2.3 Determining the Format of a Document 51
5.3 Built-In Document Loading Functions 51
5.4 Specifying a Forest in Which to Load a Document 52
5.4.1 Advantages of Specifying a Forest 53
5.4.2 Example: Examining a Document to Decide Which Forest to Specify 53
5.4.3 More Examples 53
5.5 Using WebDAV to Load Documents 54
5.6 Permissions on Documents 55

6.0 Importing XQuery Modules and Resolving Paths 56
6.1 XQuery Library Modules and Main Modules 56
6.1.1 Main Modules 56
6.1.2 Library Modules 56
6.2 Rules for Resolving Import, Invoke, and Spawn Paths 57
6.3 Example Import Module Scenario 59

7.0 Library Services Applications 60
7.1 Understanding Library Services 60
7.2 Building Applications with Library Services 62
7.3 Required Range Element Indexes 62
7.4 Library Services API 63
7.4.1 Library Services API Categories 64
7.4.2 Managed Document Update Wrapper Functions 64
7.5 Security Considerations of Library Services Applications 65
7.5.1 dls-admin Role 65
7.5.2 dls-user Role 65
7.5.3 dls-internal Role 65
7.6 Putting Documents Under Managed Version Control 66
7.7 Checking Out Managed Documents 66
7.7.1 Displaying the Checkout Status of Managed Documents 67
7.7.2 Breaking the Checkout of Managed Documents 67
7.8 Checking In Managed Documents 67
7.9 Updating Managed Documents 68
7.10 Defining a Retention Policy 69
7.10.1 Purging Versions of Managed Document 69
7.10.2 About Retention Rules 70
7.10.3 Creating Retention Rules 70
7.10.4 Retaining Specific Versions of Documents 72
7.10.5 Multiple Retention Rules 73
7.10.6 Deleting Retention Rules 75
7.11 Managing Modular Documents in Library Services 76
7.11.1 Creating Managed Modular Documents 76
7.11.2 Expanding Managed Modular Documents 78
7.11.3 Managing Versions of Modular Documents 79

8.0 Transforming XML Structures With a Recursive typeswitch Expression ..82
8.1 XML Transformations 82
8.1.1 XQuery vs. XSLT 82
8.1.2 Transforming to XHTML or XSL-FO 82
8.1.3 The typeswitch Expression 83
8.2 Sample XQuery Transformation Code 83
8.2.1 Simple Example 84
8.2.2 Simple Example With cts:highlight 85
8.2.3 Sample Transformation to XHTML 86
8.2.4 Extending the typeswitch Design Pattern 88

9.0 Document and Directory Locks 89
9.1 Overview of Locks 89
9.1.1 Write Locks 89
9.1.2 Persistent 89
9.1.3 Searchable 90

9.1.4 Exclusive or Shared 90
9.1.5 Hierarchical 90
9.1.6 Locks and WebDAV 90
9.1.7 Other Uses for Locks 90
9.2 Lock APIs 91
9.3 Example: Finding the URI of Documents With Locks 92
9.4 Example: Setting a Lock on a Document 92
9.5 Example: Releasing a Lock on a Document 93
9.6 Example: Finding the User to Whom a Lock Belongs 93

10.0 Properties Documents and Directories 94
10.1 Properties Documents 94
10.1.1 Properties Document Namespace and Schema 94
10.1.2 APIs on Properties Documents 96
10.1.3 XQuery property Axis 97
10.1.4 Protected Properties 98
10.1.5 Creating Element Indexes on a Properties Document Element 98
10.1.6 Sample Properties Documents 98
10.2 Using Properties for Document Processing 98
10.2.1 Using the property Axis to Determine Document State 99
10.2.2 Document Processing Problem 100
10.2.3 Solution for Document Processing 100
10.2.4 Basic Commands for Running Modules 101
10.3 Directories 102
10.3.1 Properties and Directories 102
10.3.2 Directories and WebDAV Servers 102
10.3.3 Directories Versus Collections 103
10.4 Permissions On Properties and Directories 103
10.5 Example: Directory and Document Browser 104
10.5.1 Directory Browser Code 104
10.5.2 Setting Up the Directory Browser 105

11.0 Point-In-Time Queries 107
11.1 Understanding Point-In-Time Queries 107
11.1.1 Fragments Stored in Log-Structured Database 107
11.1.2 System Timestamps and Merge Timestamps 108
11.1.3 How the Fragments for Point-In-Time Queries are Stored 108
11.1.4 Only Available on Query Statements, Not on Update Statements 109
11.1.5 All Auxiliary Databases Use Latest Version 109
11.1.6 Database Configuration Changes Do Not Apply to Point-In-Time Fragments 110
11.2 Using Timestamps in Queries 110
11.2.1 Enabling Point-In-Time Queries in the Admin Interface 110
11.2.2 The xdmp:request-timestamp Function 112
11.2.3 Requires the xdmp:timestamp Execute Privilege 112

11.2.4 The Timestamp Parameter to xdmp:eval, xdmp:invoke, xdmp:spawn ..112
11.2.5 Timestamps on Requests in XCC 113
11.2.6 Scoring Considerations 113
11.3 Specifying Point-In-Time Queries in xdmp:eval, xdmp:invoke, xdmp:spawn, and XCC 114
11.3.1 Example: Query Old Versions of Documents Using XCC 114
11.3.2 Example: Querying Deleted Documents 114
11.4 Keeping Track of System Timestamps 115

12.0 Using the map Functions to Create Name-Value Maps 117
12.1 Maps: In-Memory Structures to Manipulate in XQuery 117
12.2 map:map XQuery Primitive Type 117
12.3 Serializing a Map to an XML Node 118
12.4 Map API 119
12.5 Examples 119
12.5.1 Creating a Simple Map 119
12.5.2 Returning the Values in a Map 120
12.5.3 Constructing a Serialized Map 120
12.5.4 Add a Value that is a Sequence 121

13.0 Function Values 122
13.1 Overview of Function Values 122
13.2 xdmp:function XQuery Primitive Type 122
13.3 XQuery APIs for Function Values 123
13.4 When the Applied Function is an Update from a Query Statement 123
13.5 Example of Using Function Values 123

14.0 Reusing Content With Modular Document Applications 126
14.1 Modular Documents 126
14.2 XInclude and XPointer 127
14.2.1 Example: Simple id 128
14.2.2 Example: xpath() Scheme 128
14.2.3 Example: element() Scheme 129
14.2.4 Example: xmlns() and xpath() Scheme 129
14.3 CPF XInclude Application and API 129
14.3.1 XInclude Code and CPF Pipeline 130
14.3.2 Required Security Privileges—xinclude Role 130
14.4 Creating XML for Use in a Modular Document Application 131
14.4.1 <xi:include> Elements 131
14.4.2 <xi:fallback> Elements 131
14.4.3 Simple Examples 132
14.5 Setting Up a Modular Document Application 133

15.0 Controlling App Server Access, Output, and Errors 135
15.1 Creating Custom HTTP Server Error Pages 135
15.1.1 Overview of Custom HTTP Error Pages 135
15.1.2 Error XML Format 136
15.1.3 Configuring Custom Error Pages 136
15.1.4 Execute Permissions Are Needed On Error Handler Document for Modules Databases 137
15.1.5 Example of Custom Error Pages 138
15.2 Setting Up URL Rewriting for an HTTP App Server 138
15.2.1 Overview of URL Rewriting 139
15.2.2 Creating a URL Rewrite Script 140
15.2.3 Configuring an HTTP App Server to use the URL Rewrite Script 140
15.2.4 More URL Rewrite Script Examples 142
15.2.5 Prohibiting Access to Internal URLs 143
15.2.6 URL Rewriting and Page-Relative URLs 143
15.2.7 Using the URL Rewrite Trace Event 143
15.3 Outputting SGML Entities 145
15.3.1 Understanding the Different SGML Mapping Settings 145
15.3.2 Configuring SGML Mapping in the App Server Configuration 146
15.3.3 Specifying SGML Mapping in an XQuery Program 147
15.4 Specifying the Output Encoding 147
15.4.1 Configuring App Server Output Encoding Setting 147
15.4.2 XQuery Built-In For Specifying the Output Encoding 148

16.0 JSON: Serializing To and Parsing From 149
16.1 Serializing and Parsing JSON To and From XQuery Types 149
16.2 JSON API 149
16.3 JSON Parsing Restrictions 150
16.4 Examples 150
16.4.1 Serializing to a JSON String 150
16.4.2 Parsing a JSON String into a List of Items 151

17.0 Using Triggers to Spawn Actions 152
17.1 Overview of Triggers Used in the Content Processing Framework 152
17.2 Pre-Commit Versus Post-Commit Triggers 153
17.2.1 Pre-Commit Triggers 153
17.2.2 Post-Commit Triggers 153
17.3 Trigger Events 154
17.4 Trigger Scope 155
17.5 Modules Invoked or Spawned by Triggers 155
17.5.1 Difference in Module Behavior for Pre- and Post-Commit Triggers 155
17.5.2 Module External Variables trgr:uri and trgr:trigger 156
17.6 Creating and Managing Triggers With triggers.xqy 156
17.7 Simple Trigger Example 157

18.0 Training the Classifier 159
18.1 Understanding How Training and Classification Works 159
18.1.1 Training and Classification 159
18.1.2 XML SVM Classifier 160
18.1.3 Hyper-Planes and Thresholds for Classes 161
18.1.4 Training Content for the Classifier 164
18.2 Classifier API 164
18.2.1 XQuery Built-In Functions 164
18.2.2 Data Can Reside Anywhere or Be Constructed 165
18.2.3 API is Extremely Tunable 165
18.2.4 Supports Versus Weights Classifiers 165
18.2.5 Kernels (Mapping Functions) 166
18.2.6 Find Thresholds That Balance Precision and Recall 166
18.3 Leveraging XML With the Classifier 166
18.4 Creating a Training Set 167
18.4.1 Importance of the Training Set 167
18.4.2 Defining Labels for the Training Set 167
18.5 Methodology For Determining Thresholds For Each Class 168
18.6 Example: Training and Running the Classifier 170
18.6.1 Shakespeare’s Plays: The Training Set 170
18.6.2 Comedy, Tragedy, History: The Classes 171
18.6.3 Partition the Training Content Set 171
18.6.4 Create Labels on the First Half of the Training Content 171
18.6.5 Run cts:train on the First Half of the Training Content 172
18.6.6 Run cts:classify on the Second Half of the Content Set 172
18.6.7 Use cts:thresholds to Compute the Thresholds on the Second Half 173
18.6.8 Evaluating Your Results, Make Changes, and Run Another Iteration ...173
18.6.9 Run the Classifier on Other Content 174

19.0 Technical Support 175

1.0 Developing Applications in MarkLogic Server

This chapter describes application development in MarkLogic Server in general terms, and includes the following sections:

- Overview of MarkLogic Server Application Development
- Skills Needed to Develop MarkLogic Server Applications
- Where to Find Specific Information

This *Application Developer's Guide* provides general information about creating XQuery applications using MarkLogic Server. For information about developing search application using the powerful XQuery search features of MarkLogic Server, see the *Search Developer's Guide*.

1.1 Overview of MarkLogic Server Application Development

MarkLogic Server provides a platform to build content application. Developers build applications using XQuery both to search the content and as a programming language in which to develop the applications. The applications can integrate with other environments via web services or via an XCC interface from Java or .NET. But it is possible to create entire applications using only MarkLogic Server, and programmed entirely in XQuery.

This *Application Developer's Guide* focuses primarily on techniques, design patterns, and concepts needed to use XQuery to build content and search applications in MarkLogic Server.

1.2 Skills Needed to Develop MarkLogic Server Applications

The following are skills and experience useful in developing applications with MarkLogic Server. You do not need to have all of these skills to get started, but these are skills that you can build over time as you gain MarkLogic application development experience.

- Web development skills (xHTML, HTTP, cross-browser issues, CSS, Javascript, and so on), especially if you are developing applications which run on an HTTP App Server.
- Overall understanding and knowledge of XML.
- XQuery skills. To get started with XQuery, see *XQuery Reference Guide*.
- Understanding of search engines and full-text queries.
- Java or .Net experience, if you are using XCC to develop applications.
- General application development techniques, such as solidifying application requirements, source code control, and so on.
- If you will be deploying large-scale applications, administration on operations techniques such as creating and managing large filesystems, managing multiple machines, network bandwidth issues, and so on.

1.3 Where to Find Specific Information

MarkLogic Server includes a full set of documentation, available at online at http://developer.marklogic.com/pubs. This *Application Developer's Guide* provides concepts and design patterns used in developing MarkLogic Server applications. The following is a list of pointers where you can find technical information:

- For information about installing and upgrading MarkLogic Server, see the *Installation Guide*. Additionally, for a list of new features and any known incompatibilities with other releases, see the *Release Notes*.
- For information about creating databases, forests, App Servers, users, privileges, and so on, see the *Administrator's Guide*.
- For information on how to use security in MarkLogic Server, see *Understanding and Using Security*.
- For information on creating pipeline processes for document conversion and other purposes, see *Content Processing Framework*.
- For syntax and usage information on individual XQuery functions, including the XQuery standard functions, the MarkLogic Server built-in extension functions for updates, search, HTTP server functionality, and other XQuery library functions, see the *MarkLogic Built-In and Module Functions Reference*.
- For information on using XCC to access content in MarkLogic Server from Java or .Net, see the *XCC Developer's Guide*.
- For information on how languages affect searches, see Language Support in MarkLogic Server in the *Search Developer's Guide*. It is important to understand how languages affect your searches regardless of the language of your content.
- For information about developing search applications, see the *Search Developer's Guide*.
- For information on what constitutes a transaction in MarkLogic Server, see "Understanding Transactions in MarkLogic Server" on page 12 in this *Application Developer's Guide*.
- For other developer topics, review the contents for this *Application Developer's Guide*.
- For performance-related issues, see *Query Performance and Tuning*.

2.0 Understanding Transactions in MarkLogic Server

MarkLogic Server is a transactional system that ensures data integrity. This chapter describes the transaction model of MarkLogic Server, and includes the following sections:

- Overview of Terminology
- System Timestamps and Fragment Versioning
- Query and Update Statements
- Semi-Colon as a Transactional Separator
- Interactions with xdmp:eval/invoke
- Functions With Non-Transactional Side Effects
- Example: Incrementing the System Timestamp

2.1 Overview of Terminology

Although transactions are a core feature of most database systems, various systems support subtly different transactional semantics. Clearly defined terminology is key to a proper and comprehensive understanding of these semantics. To avoid confusion over the meaning of any of these terms, this section provides definitions for several terms used throughout this chapter and throughout the MarkLogic Server documentation. The definitions of the terms also provide a useful starting point for describing transactions in MarkLogic Server.

- system timestamp

  The *system timestamp* is a number maintained by MarkLogic Server that increments by 1 every time a change or a set of changes occurs in any of the databases in a system (including configuration changes from any host in a cluster). Each fragment stored in a database has system timestamps associated with it to determine the range of timestamps during which the fragment is valid.

- statement

  A *statement* is a unit of XQuery code to be evaluated by MarkLogic Server.

- transaction

  A *transaction* is a set of statements which either all fail or all succeed.

- program

  A *program* is the expanded version of some XQuery code that is submitted to MarkLogic Server for evaluation, such as a query expression in a `.xqy` file or XQuery code submitted in an `xdmp:eval` statement. The program consists not only of the code in the calling

module, but also any imported modules that are called from the calling module, and any modules they might call, and so on.

- request

  A *request* is any invocation of a program, whether through an App Server, through a task server, through `xdmp:eval`, or through any other means. In addition, certain client calls to App Servers (for example, loading an XML document through XCC, downloading an image through HTTP, or locking a document through WebDAV) are also requests.

- query statement

  A *query statement* is a statement that contains no update calls. The existence of any update calls is determined statically through lexical analysis prior to beginning the statement evaluation. Query statements run at a particular system timestamp, and have a read-consistent view of the database.

- update statement

  An *update statement* is a statement that contains the potential to perform updates (that is, it contains one or more update calls). The existence of any update calls is determined statically through lexical analysis prior to beginning the statement evaluation. Consequently, a statement may be categorized as an update statement regardless of whether the statement actually performs any updates. Update statements run with readers/writers locks, obtaining locks as needed for documents accessed in the statement.

- readers/writers locks

  During update statements, MarkLogic Server uses *readers/writers locks*, which are a set of read and write locks that lock documents for reading and update only when the documents are accessed, not at the beginning of a transaction. Because update statements only obtain locks as they are needed, update statements might see a newer version of a document than the time when the update statement began. The view will be still be consistent for any given document from the time the document is locked. Once a document is locked, any update statements in other transactions will have to wait for the lock to be released before reading or updating the document. For more details, see “Update Statements—Readers/Writers Locks” on page 15.

2.2 System Timestamps and Fragment Versioning

To understand how transactions work in MarkLogic Server, it is important to understand something about how documents are stored. Documents are made up of one or more fragments. After a document is created, each of its fragments are stored in one or more stands. The stands are part of a forest, and the forest is part of a database. A database contains one or more forests.

Each fragment in a stand has system timestamps associated with it, which correspond to the range of system timestamps in which that version of the fragment is valid. When a document is updated, the update process creates new versions of any fragments that are changed. The new versions of the fragments are stored in a new stand and have a new set of valid system timestamps associated with them. Eventually, the system merges the old and new stands together and creates a new stand with only the latest versions of the fragments. Point-in-time queries will also effect which versions of fragments are stored and preserved during a merge. After the merge, the old stands are deleted.

The range of valid system timestamps associated with fragments are used when a statement determines which version of a document to use during a transaction. The next section describes how query statements and update statements work. For more details about how merges work, see the "Understanding and Controlling Database Merges" chapter of the *Administrator's Guide*. For more details on how point-in-time queries effect which versions of documents are stored, see "Point-In-Time Queries" on page 107. For an example of how system timestamps change as updates occur to the system, see "Example: Incrementing the System Timestamp" on page 22.

2.3 Query and Update Statements

As defined in the previous section, a statement in MarkLogic Server is either a *query statement* or an *update statement*. To ensure maximum concurrency and minimum contention for resources, query statements and update statements are optimized differently. Query statements are read-only and never obtain any locks on documents. Update statements must ensure transactional integrity and obtain locks on documents. This section describes the semantics of these two types of statements and includes the following parts:

- Query Statements
- Update Statements—Readers/Writers Locks
- Example Scenario

2.3.1 Query Statements

If there are no updates found in a statement during lexical analysis, then the statement is a query statement. This section describes query statements and has the following parts:

- Queries Run at a Timestamp (No Locks)
- Queries See Latest Version of Documents Up To Timestamp of Query

2.3.1.1 Queries Run at a Timestamp (No Locks)

Query statements run at the system timestamp corresponding to the time in which the query statement initiates. Calls to `xdmp:request-timestamp` will return the same system timestamp at any point during a query statement; they will not return the empty sequence. Query statements do not obtain locks on any documents, so other transactions can read or update the document while the query is executing.

2.3.1.2 Queries See Latest Version of Documents Up To Timestamp of Query

At the beginning of a query statement evaluation, MarkLogic Server gets the current system timestamp (the number returned when calling the `xdmp:request-timestamp` function) and uses only the latest versions of documents whose timestamp is less than or equal to that number. Even if any of the documents that the query is accessing are updated or deleted while the query is being executed, the use of timestamps ensures that a query statement always sees a consistent view of the documents it accesses.

2.3.2 Update Statements—Readers/Writers Locks

If the potential for updates is found in a statement during lexical analysis, then the statement is an update statement. Depending on the specific logic of the statement, an update statement might not actually end up updating anything, but a statement that is determined (during lexical analysis) to be an update statement is run as an update statement, not a query statement.

Update statements run with readers/writers locks, not at a timestamp like query statements. Because update statements do not run at a set timestamp, they see the latest view of any given document at the time it is first accessed by the statement. Because an update statement must successfully obtain locks on all documents it reads or writes in order to complete evaluation, there is no chance that a given update statement will see “half” or “some” of the updates made by some other transactions; the statement is indeed transactional.

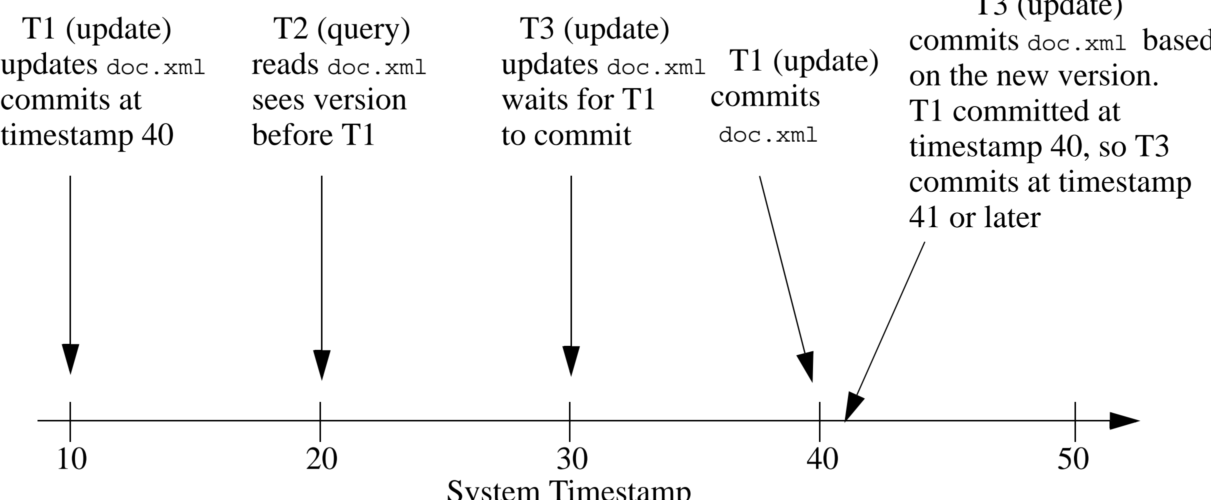
Calls to `xdmp:request-timestamp` will always return the empty sequence during an update statement; that is, if `xdmp:request-timestamp` returns a value, the statement is a query statement, not an update statement.

During the update statement evaluation, any updates the statement performs are not visible to itself or to other transactions. The updates only become visible once the transaction commits, which occurs automatically when the statement completes. If you want a transaction to use a newly-updated document, you must separate that transaction into two transactions, first updating the document in one transaction and then accessing the new version in a subsequent transaction.

An update statement cannot perform an update to the same document that will conflict with other updates occurring in the same statement (for example, you cannot update a node and add a child element to that node in the same statement). Any attempts to perform such conflicting updates to the same document in a single update statement will fail with an `XDMP-CONFLICTINGUPDATES` exception.

2.3.3 Example Scenario

The following figure shows three different transactions, T1, T2, and T3, and how the transactional semantics work for each one:



Assume T1 is a long-running update transaction which starts when the system is at timestamp 10 and ends up committing at timestamp 40 (meaning there were 30 updates or other changes to the system while this update statement runs). When T2 reads the document being updated by T1 (`doc.xml`), it sees the latest version that has a system timestamp of 20 or less, which turns out to be the same version T1 uses before its update. When T3 tries to update the document, it finds that T1 has readers/writers locks on it, so it waits for them to be released. After T1 commits and releases the locks, then T3 sees the newly updated version of the document, and performs its update which is committed at a new timestamp of 41.

2.4 Semi-Colon as a Transactional Separator

MarkLogic Server has extended the XQuery language to include the semi-colon (`;`) in the XQuery body as a separator between transactions in an XQuery module. Each part of the program separated by a semi-colon is evaluated as its own transaction. It is possible to have a program where some semi-colon separated parts are evaluated as query statements and some are evaluated as update statements. The statements are evaluated in the order in which they appear, and in the case of update statements, a statement will commit before the next one begins.

Note that semi-colon separated statements are not multi-statement transactions. Although static analysis is performed on the entire module first and will catch some errors (syntax errors, for example), it will not catch any runtime errors that might occur during execution. If one update statement commits and the next one throws a runtime error, the first transaction is not rolled back. If you have logic that requires a rollback if subsequent transactions fail, you must either add that logic to your XQuery code or use a pre-commit trigger. For information about triggers, see “Using Triggers to Spawn Actions” on page 152.

2.5 Interactions with xdmp:eval/invoke

The `xdmp:eval` and `xdmp:invoke` functions allow you to start one transaction from the context of another. The `xdmp:eval` function submits a string to be evaluated and the `xdmp:invoke` function evaluates a stored module. You can control the semantics of `xdmp:eval` and `xdmp:invoke` with options to the functions, and this can subtly change the transactional semantics of your program. This section describes some of those subtleties and includes the following parts:

- Isolation Option to xdmp:eval/invoke
- Preventing Undetectable Deadlocks
- Seeing Updates From eval/invoke Later in the Transaction

2.5.1 Isolation Option to xdmp:eval/invoke

The `xdmp:eval` and `xdmp:invoke` functions take an options node as the optional third parameter. The `isolation` option determines the behavior of the transaction that results from the eval/invoke operation, and it must be one of the following values:

- `same-statement`
- `different-transaction`

When you set the `isolation` to `same-statement`, the code that is run when you call `xdmp:eval` or `xdmp:invoke` is run as part of the same statement as the calling statement, which means it is also run as part of the same transaction (if the calling statement is an update statement). The effect is that any updates done in the eval/invoke operation with `same-statement` isolation are performed as part of the same statement, and therefore are not visible to subsequent parts of the statement. Because query statements run at a timestamp, you cannot run update statements with `same-statement` isolation in eval/invoke operations that are called from query statements. Doing

so would specify that the statement switch between timestamp mode and readers/writers locks mode in the middle of a transaction, and that is not allowed; statements that do so will throw an exception. Additionally, `same-statement` eval/invoke operations are not allowed when specifying a different database in which the eval/invoke operates against.

When you set the `isolation` to `different-transaction`, the code that is run when you call `xdmp:eval` or `xdmp:invoke` is run as a separate transaction from the calling statement, and the eval/invoke statement will commit before continuing to the rest of the calling statement. The effect is that any updates done in the eval/invoke operation with `different-transaction` isolation are indeed visible to subsequent parts of the statement. However, if you use `different-transaction` isolation (which is the default isolation level), you need to ensure that you do not get into an undetectable deadlock situation (see “Preventing Undetectable Deadlocks” on page 18).

The following table shows which isolation options are allowed from query statements and update statements.

Calling Statement	Called Statement (`xdmp:eval`, `xdmp:invoke`)			
	`same-statement` isolation		`different-transaction` isolation	
	query statement	update statement	query statement	update statement
query statement (timestamp mode)	Yes	No (throws exception)	Yes	Yes
update statement (readers/writers locks mode)	Yes*	Yes	Yes	Yes (possible deadlock if updating a document with any lock)

Note: This table is slightly simplified. For example, if an update statement calls a query statement with `same-statement` isolation, the “query statement” is actually run as part of the update statement (because it is run as part of the same transaction as the calling update statement), and it therefore runs with readers/writers locks, not in a timestamp.

2.5.2 Preventing Undetectable Deadlocks

A deadlock is where two processes are each waiting for the other to release a lock, and neither process can continue until the lock is released. Deadlocks are a normal part of database operations, and when a system detects them, it can deal with them (for example, by retrying one or the other transactions, by killing one or the other or both requests, and so on).

There are, however, some deadlock situations that MarkLogic Server cannot detect. When you run an update statement that calls an `xdmp:eval` or `xdmp:invoke` statement, and the eval/invoke in turn is an update statement, you run the risk of creating an undetectable deadlock condition. These undetectable deadlocks can only occur in update statements; query statements will never cause a deadlock.

An undetectable deadlock condition occurs when a transaction acquires a lock of any kind on a document and then an eval/invoke statement called from that transaction attempts to get a write lock on the same document. These undetectable deadlock conditions can only be resolved by restarting MarkLogic Server; cancelling the query does not clear the deadlock.

To be completely safe, you can prevent these undetectable deadlocks from occurring by setting the `prevent-deadlocks` option to `true`, as in the following example:

```
xquery version "1,0-ml";
(: the next line ensures this runs as an update statement :)
declare option xdmp:update "true";
xdmp:eval("xdmp:node-replace(doc('/docs/test.xml')/a,
<b>goodbye</b>)",
          (),
          <options xmlns="xdmp:eval">
            <isolation>different-transaction</isolation>
            <prevent-deadlocks>true</prevent-deadlocks>
          </options>) ,
doc("/docs/test.xml")
```

This statement will then throw the following exception:

```
XDMP-PREVENTDEADLOCKS: Processing an update from an update with
different-transaction isolation could deadlock
```

In this case, it will indeed prevent a deadlock from occurring because this statement runs as an update statement, due to the `xdmp:document-insert` call, and therefore uses readers/writers locks. In line 2, a read lock is placed on the document with URI `/docs/test.xml`. Then, the `xdmp:eval` statement attempts to get a write lock on the same document, but it cannot get the write lock until the read lock is released. This creates an undetectable deadlock condition, and the only way to clear the deadlock it to restart MarkLogic Server. Therefore the `prevent-deadlocks` option stopped the deadlock from occurring.

If you remove the `prevent-deadlocks` option, then it defaults to `false` (that is, it will *allow* deadlocks). Therefore, the following statement results in a deadlock:

Warning This code is for demonstration purposes; if you run this code, it will cause an undetectable deadlock and you will have to restart MarkLogic Server to clear the deadlock.

```
(: the next line ensures this runs as an update statement :)
if ( 1 = 2) then ( xdmp:document-insert("foobar", <a/>) ) else (),
doc("/docs/test.xml"),
xdmp:eval("xdmp:node-replace(doc('/docs/test.xml')/a,
<b>goodbye</b>)",
          (),
          <options xmlns="xdmp:eval">
            <isolation>different-transaction</isolation>
          </options>) ,
doc("/docs/test.xml")
```

This is a deadlock condition, and the only way to clear the deadlock it to restart MarkLogic Server. Note that if you take out the first call to `doc("/docs/test.xml")` in line 2 of the above example, the statement will not deadlock because the read lock on `/docs/test.xml` is not called until after the `xdmp:eval` statement completes.

2.5.3 Seeing Updates From eval/invoke Later in the Transaction

If you are sure that your update statement in an eval/invoke operation does not try to update any documents that are referenced earlier in the calling statement (and therefore does not result in an undetectable deadlock condition, as described in “Preventing Undetectable Deadlocks” on page 18), then you can set up your statement so updates from an eval/invoke are visible from the calling transaction. This is most useful in transactions that have the eval/invoke statement before the code that accesses the newly updated documents.

Note: If you want to see the updates from an eval/invoke operation later in your statement, the statement must be an update statement. If the statement is a query statement, it runs in timestamp mode and will always see the version of the document that existing before the eval/invoke operation committed.

For example, consider the following example, where `doc("/docs/test.xml")` returns `<a>hello</a>` before the transaction begins:

```
(: doc("/docs/test.xml") returns <a>hello</a> before running this :)
(: the next line ensures this runs as an update statement :)
if ( 1 = 2 ) then ( xdmp:document-insert("fake.xml", <a/>) ) else (),
xdmp:eval("xdmp:node-replace(doc('/docs/test.xml')/node(),
<b>goodbye</b>)", (),
          <options xmlns="xdmp:eval">
            <isolation>different-transaction</isolation>
            <prevent-deadlocks>false</prevent-deadlocks>
          </options>) ,
doc("/docs/test.xml")
```

The call to `doc("/docs/test.xml")` in the last line of the example returns `<a>goodbye</a>`, which is the new version that was updated by the `xdmp:eval` operation.

2.6 Functions With Non-Transactional Side Effects

Update statements use various update built-in functions which, at the time the transaction completes, update documents in a database. These updates are technically known as *side effects*, because they cause a change to happen outside of what the statement returns. The side effects from the update built-in functions (`xdmp:node-replace`, `xdmp:document-insert`, and so on) are transactional in nature; that is, they either complete fully or are rolled back to the state at the beginning of the update statement.

There are several functions that evaluate asynchronously as soon as they are called, regardless of whether called from an update statement or a query statement. Examples of these functions are `xdmp:spawn`, `xdmp:http-get`, and `xdmp:log`. These functions have side effects outside the scope of the transaction (*non-transactional* side effects).

When evaluating an XQuery module that performs an update transaction, it is possible for the update to either fail or retry. That is the normal, transactional behavior, and the database will always be left in a consistent state if a transaction fails or retries. If your update statement calls one of these functions with non-transactional side effects, however, that function will evaluate even if the calling update statement fails and rolls back.

Use care or avoid calling any of these functions from an update statement, as they are not guaranteed to only evaluate once (or to not evaluate if the transaction rolls back). If you are logging some information with `xdmp:log` in your transaction, it might or might not be appropriate for that statement to be logged on retries (for example, if the transaction is retried because a deadlock is detected). Even if it is not what you intended, it might not do any harm.

Other side effects, however, can cause problems in updates. For example, if you use `xdmp:spawn` in this context, the action might be spawned multiple times if the calling statement retries, or the action might be spawned even if the transaction fails; the `xdmp:spawn` call evaluates asyncronously as soon as it is called. Similarly, if you are calling a web service with `xdmp:http-get` from an update statement, it might evaluate when you did not mean for it to evaluate. If you do use these functions in updates, your application logic must handle the side effects appropriately. These types of use cases are usually better suited to triggers and the Content Processing Framework. For details, see “Using Triggers to Spawn Actions” on page 152 and the *Content Processing Framework* manual.

2.7 Example: Incrementing the System Timestamp

The following example shows how updates to the system increment the system timestamp. For background on system timestamps, see “System Timestamps and Fragment Versioning” on page 13.

```
<results>{
  <start-of-query>{xdmp:request-timestamp()}</start-of-query>,
for $x in (1 to 10)
return
(xdmp:eval("xdmp:document-insert('test.xml', <test/>)"),
 xdmp:eval("xdmp:document-delete('test.xml')" ) ),
<end-of-query>{xdmp:eval('xdmp:request-timestamp()')}</end-of-query>,
  <number-of-transactions>{xdmp:eval('xdmp:request-timestamp()') -
  xdmp:request-timestamp()}</number-of-transactions>,
<elapsed-time>{xdmp:query-meters()/*:elapsed-time/text()}
  </elapsed-time>
}
</results>
```

This XQuery program executes as a query statement, which runs at a timestamp. Each of the `xdmp:eval` statements is evaluated as a separate transaction, and they are each update statements. The query starts off by printing out the system timestamp, then it starts a FLWOR expression which evaluates 10 times. Each iteration of the FLWOR creates a document called `test.xml` in a separate transaction, then deletes the document in another transaction. Finally, it calculates the number of transactions that have occurred since the query started by starting a new transaction to get the current system timestamp and subtracting that from the timestamp in which the query runs. Therefore, this query produces the following results, which show that 20 transactions have occurred.

```
<results>
  <start-of-query>382638</start-of-query>
  <end-of-query>382658</end-of-query>
  <number-of-transactions>20</number-of-transactions>
  <elapsed-time>PT0.102211S</elapsed-time>
</results>
```

Note: This example executes 20 update statements with `xdmp:eval`, each of which starts its own transaction, and each of which increments the system timestamp by 1. If there are other transactions happening concurrently on the system while these transactions are executing, those transactions might also increment the system timestamp, and it is possible that the last `xdmp:eval` statement which executes the `xdmp:request-timestamp` function might return a number more that 20 greater than the first call to `xdmp:request-timestamp`.

3.0 Loading Schemas

MarkLogic Server has the concept of a *schema database*. The schema database stores schema documents that can be shared across many different databases within the same MarkLogic Server cluster. This chapter introduces the basics of loading schema documents into MarkLogic Server, and includes the following sections:

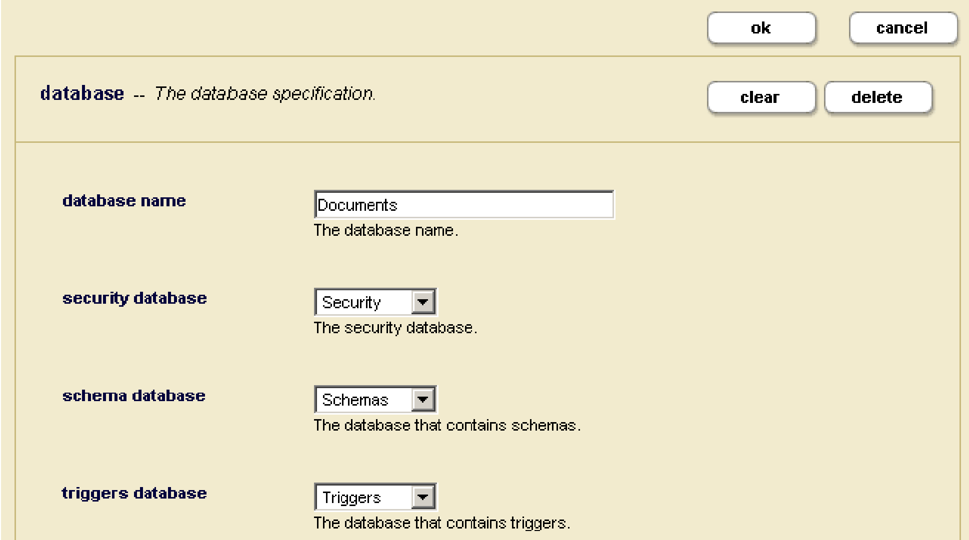
- Configuring Your Database
- Loading Your Schema
- Referencing Your Schema
- Working With Your Schema
- Validating XML Against a Schema

For more information about configuring schemas in the Admin Interface, see the "Understanding and Defining Schemas" chapter of the *Administrator's Guide*.

3.1 Configuring Your Database

MarkLogic Server automatically creates an empty schema database, named *Schemas*, at installation time.

Every document database that is created references both a schema database and a security database. By default, when a new database is created, it automatically references *Schemas* as its schema database. In most cases, this default configuration (shown below) will be correct:



In other cases, it may be desirable to configure your database to reference a different schema database. It may be necessary, for example, to be able to have two different databases reference different versions of the same schema using a common schema name. In these situations, simply select the database from the drop-down schema database menu that you want to use in place of the default *Schemas* database. Any database in the system can be used as a schema database.

In select cases, it may be efficient to configure your database to reference itself as the schema database. This is a perfectly acceptable configuration which can be set up through the same drop-down menu. In these situations, a single database stores both content and schema relevant to a set of applications.

> **Note:** To create a database that references itself as its schema database, you must first create the database in a configuration that references the default *Schemas* database. Once the new database has been created, you can change its schema database configuration to point to itself using the drop-down menu.

3.2 Loading Your Schema

HTTP and XDBC Servers connect to document databases. Document insertion operations conducted through those HTTP and XDBC Servers (using `xdmp:document-load()`, `xdmp:document-insert()` and the various XDBC document insertion methods) insert documents into the document databases connected to those servers.

This makes loading schemas slightly tricky. Because the system looks in the schema database referenced by the current document database when requesting schema documents, you need to make sure that the schema documents are loaded into the current database's *schema database* rather than into the current *document database*.

There are several ways to accomplish this:

1. You can use the Admin Interface's load utility to load schema documents directly into a schema database. Go to the Database screen for the schema database into which you want to load documents. Select the load tab at top-right and proceed to load your schema as you would load any other document.

2. You can create an XQuery program that uses the `xdmp:eval` built-in, specifying the `<database>` option to load a schema directly into the current database’s schema database:

```
xdmp:eval('xdmp:document-load("sample.xsd")', (),
        <options xmlns="xdmp:eval">
            <database>{xdmp:schema-database()}</database>
        </options>)
```

3. You can create an XDBC or HTTP Server that directly references the schema database in question as its document database, and then use any document insertion function to load one or more schemas into that schema database. This approach should not be necessary.

4. You can create a WebDAV Server that references the Schemas database and then drag-and-drop schema documents in using a WebDAV client.

3.3 Referencing Your Schema

Schemas are automatically invoked by the server when loading documents (for conducting content repair) and when evaluating queries (for proper data typing). For any given document, the server looks for a matching schema in the schema database referenced by the current document database.

1. If a schema with a matching target namespace is not found, a schema is not used in processing the document.
2. If one matching schema is found, that schema is used for processing the document.
3. If there are more than one matching schema in the schema database, a schema is selected based on the precedence rules in the order listed:

   a. If the `xsi:schemaLocation` or `xsi:noNamespaceSchemaLocation` attribute of the document root element specifies a URI, the schema with the specified URI is used.

   b. If there is an import schema prolog expression with a matching target namespace, the schema with the specified URI is used. Note that if the target namespace of the import schema expression and that of the schema document referenced by that expression do not match, the import schema expression is not applied.

   c. If there is a schema with a matching namespace configured within the current HTTP or XDBC Server's Schema panel, that schema is used. Note that if the target namespace specified in the configuration panel does not match the target namespace of the schema document, the Admin Interface schema configuration information is not used.

   d. If none of these rules apply, the server uses the first schema that it finds. Given that document ordering within the database is not defined, this is not generally a predictable selection mechanism, and is not recommended.

3.4 Working With Your Schema

It is sometimes useful to be able to explicitly read a schema from the database, either to return it to the outside world or to drive certain schema-driven query processing activities.

Schemas are treated just like any other document by the system. They can be inserted, read, updated and deleted just like any other document. The *difference* is that schemas are usually stored in a secondary schema database, not in the document database itself.

The most common activity developers want to carry out with schema is to read them. There are two approaches to fetching a schema from the server explicitly:

1. You can create an XQuery that uses the `xdmp:eval-in()` built-in to read a schema directly from the current database's schema database. For example, the following expression will return the schema document loaded in the code example given above:

   ```
   xdmp:eval-in('doc("sample.xsd")', xdmp:schema-database())
   ```

   The use of the `xdmp:schema-database()` built-in ensures that the sample.xsd document will be read from the current database's schema database.

2. You can create an XDBC or HTTP Server that directly references the schema database in question as its document database, and then submit any XQuery as appropriate to read, analyze, update or otherwise work with the schemas stored in that schema database. This approach should not be necessary in most instances.

Other tasks that involve working with schema can be accomplished similarly. For example, if you need to delete a schema, an approach modeled on either of the above (using `xdmp:document-delete("sample.xsd")`) will work as expected.

3.5 Validating XML Against a Schema

You can use the XQuery `validate` expression to check if an element is valid according to a schema. For details on the validate expression, see Validate Expression in the *XQuery Reference Guide* and see the W3C XQuery recommendation (http://www.w3.org/TR/xquery/#id-validate).

If you want to validate a document before loading it, you can do so by first getting the node for the document, validate the node, and then insert it into the database. For example:

```
xquery version "1.0-ml";

let $node := xdmp:document-get("c:/tmp/test.xml")
return
try { xdmp:document-insert("/my-valid-document.xml",
        validate lax { $node } )
    }
catch ($e) { "Validation failed: ",
             $e/error:format-string/text() }
```

4.0 Content Repair

MarkLogic Server includes the ability to correct content that is not well-formed (according to the XML specification) at load time. MarkLogic Server can also be used to fix poorly or inconsistently structured content. This section discusses each of the MarkLogic Server native content repair capabilities and how they can be used in your environment.

MarkLogic Server supports five different content repair models:

- General-Purpose Tag Repair
- Empty Tag Auto-Close
- Schema-Driven Tag Repair
- Load-Time Default Namespace Assignment
- Load-Time Namespace Prefix Binding
- Query-Driven Content Repair

MarkLogic Server does not validate content against predetermined XSchema or DTDs.

4.1 General-Purpose Tag Repair

By default, MarkLogic Server applies a general-purpose stack-driven tag repair algorithm to every XML document loaded from an external source. The algorithm is triggered whenever the loader encounters a closing tag (for example, `</tag>`) that does not match the most recently opened tag on the stack.

4.1.1 How General-Purpose Tag Repair Works

For example, consider the following simple markup document example:

```
<p>This is <b>bold and <i>italic</i></b> within the paragraph.</p>
```

Each of the variations below introduces a tagging error common to hand-coded markup:

```
<p>This is <b>bold and <i>italic</b> within the paragraph.</p>

<p>This is <b>bold and <i>italic</i></b></u> within the paragraph.</p>
```

In the first variation, the italic element is never closed. And in the second, the underline element is never opened.

When the MarkLogic Server encounters an unexpected closing tag, it performs one of the following actions:

1. If the QName of the unexpected closing tag (ie. both the tag's namespace *and* its localname) matches the QName of a tag opened earlier and not yet closed, the loader will automatically close all tags until the matching opening tag is closed.

   Consequently, in the first sample tagging error shown above, the loader will automatically close the italic element when it encounters the tag closing the bold element:

   ```
   <p>This is <b>bold and <i>italic</i></b> within the paragraph.</p>
   ```

   The bold characters in the markup shown above indicate the close tag dynamically inserted by the loader.

2. If there is no match between the QName of the unexpected closing tag and all previously opened tags, the loader ignores the closing tag and proceeds.

   Consequently, in the second sample tagging error shown above, the loader will ignore the "extra" underline closing tag and proceed as if it were not present:

   ```
   <p>This is <b>bold and <i>italic</i></b></u> within the paragraph.</p>
   ```

   The italic tag shown above indicates the closing tag that the loader is ignoring.

Both rules work together to repair even more complex situations. Consider the following variation, in which the bold and italic closing tags are mis-ordered:

```
<p>This is <b>bold and <i>italic</b></i> within the paragraph.</p>
```

In this circumstance, the first rule automatically closes the italic element when the closing bold tag is encountered. When the closing italic tag is encountered, it is simply discarded as there are no previously opened italic tags still on the loader's stack. The result is more than likely what the markup author intended:

```
<p>This is <b>bold and <i>italic</i></b> within the paragraph.</p>
```

4.1.2 Pitfalls of General-Purpose Tag Repair

While these two general repair rules produce sound results in most situations, their application can also lead to repairs that may not match the original intent. For example, consider the following two examples.

1. This snippet contains a markup error – the bold element is never closed.

   ```
   <p>This is a <b>bold and <i>italic</i> part of the paragraph.</p>
   ```

   The general-purpose repair algorithm will fix this problem by inserting a closing bold tag just in front of the closing paragraph tag, because this is the point at which it becomes apparent to the system that there is a markup problem:

```
<p>This is a <b>bold and <i>italic</i> part of the paragraph.</b></p>
```

In this situation, the entire remainder of the paragraph is emboldened, because it is not otherwise apparent to the system where else the tag should have been closed. In fact, for cases other than this simple example, even a human will not always be able to make the right decision.

2. Rule 1 can also cause significant “unwinding” of the stack if a tag, opened much earlier in the document, is mistakenly closed mid-document. Consider the following mark up error where `</d>` is mistyped as `</b>`.

```
<a>
  <b>
    <c>
      <d>...content intended for d...</b>
      ...content intended for c...
    </c>
    ...content intended for b...
  </b>
  ...content intended for a...
</a>
```

The erroneous `</b>` tag triggers rule 1 and the system closes all intervening tags between `<b>` and `<d>`. Rule 2 then discards the actual close tags for `<b>` and `<c>` that have now been made redundant (since they have been closed by rule 1). This results in an incorrectly “flattened” document as shown below (some indentation and line breaks have been added for illustrative purposes):

```
<a>
  <b>
    <c>
      <d>...content intended for d...</d>
    </c>
  </b>
  ...content intended for c...
  ...content intended for b...
  ...content intended for a...
</a>
```

General-purpose tag repair is not always able to correctly repair structure problems, as shown in the two examples above. MarkLogic Server offers more advanced content repair capabilities that can be used to repair a wider range of problems, including the examples above. These advanced content repair capabilities and techniques are described in later sections of this document.

4.1.3 Scope of Application

General-purpose tag repair is carried out on all documents loaded from external sources. This includes documents loaded using the XQuery built-in `xdmp:document-load()` and the XDBC document insertion methods. XML content loaded using `xdmp:document-insert()`, `xdmp:node-replace()`, `xdmp:node-insert-before()`, `xdmp:node-insert-after()` and `xdmp:node-insert-child()`, is not processed through general-purpose tag repair. For each of these functions, the XML node provided as a parameter is either dynamically-generated by the query itself (and is consequently guaranteed to be well-formed) or is explicitly defined within the XQuery code itself (in which case the query will not successfully be parsed for execution unless it is well-formed).

Note that general-purpose tag repair will not insert a missing closing root element tag into an XML document. Previous (2.0 and earlier) versions of MarkLogic Server would repair missing root elements, making it effectively impossible to identify truncated source content. MarkLogic Server reports an error in these conditions.

4.1.4 Disabling General-Purpose Tag Repair

By default, the server attempts general-purpose tag repair for all documents loaded from external sources.

MarkLogic Server allows the developer to disable general-purpose tag repair during any individual document load through an optional parameter to `xdmp:document-load()`, `xdmp:unquote()` and XDBC's `openDocInsertStream()` method. See the *MarkLogic Built-In and Module Functions Reference* or the XDBC javadocs for more details.

4.2 Empty Tag Auto-Close

The server can automatically close tags that are identified as empty tags in a schema.

This approach addresses a common problem found in SGML and HTML documents. SGML and HTML both regard tags as markup rather than as the hierarchical element containers defined by the XML specification. In both the SGML and HTML worlds, it is quite acceptable to use a tag as an indication of some formatting directive, without any need to close the tag.

For example, an `<hr>` tag in an HTML document indicates a horizontal rule. Because there is no sense to containing anything within a horizontal rule, the tag is interpreted by browsers as an empty tag. Consequently, while HTML documents may be littered with `<hr>` tags, you will rarely find a `</hr>` tag or even a `<hr/>` tag unless someone has converted the HTML document to be XHTML-compliant. The same can frequently be said of `<img>` and `<meta>` tags, to name just two. In SGML documents, you can easily find `<pgbrk>`, `<xref>` and `<graphic>` used similarly.

MarkLogic Server enables the developer to use a specially-constructed schema to identify empty tags that should be automatically closed at load-time. Applying this type of content repair enables you to avoid the false nesting of content within otherwise-unclosed empty tags that frequently results from the liberal use of these empty tags within SGML and HTML content.

4.2.1 What Empty Tag Auto-Close Does

For example, consider the following simple SGML document snippet:

```
<book>
<para>This is the first paragraph.</para>
<pgbrk>
<para>This paragraph has a cross-reference <xref id="f563t001"> in some
<italic>italic</italic> text.</para>
</book>
```

This snippet incorporates two tags, `<pgbrk>` and `<xref>`, that are traditionally viewed as empty tags. However, working under default settings, MarkLogic Server will view each of these two tags as opening tags that will at some point later in the document be closed, and will consequently incorrectly view the following content as children of those tags. This will result in a falsely nested document (indentation and line breaks added for clarification):

```
<book>
  <para>
    This is the first paragraph.
  </para>
  <pgbrk>
    <para>
      This paragraph has a cross-reference
      <xref id="f563t001">
        in some
        <italic>italic</italic>
        text.
      </xref>
    </para>
  </pgbrk>
</book>
```

The bold characters in the markup shown above indicate closing tags automatically inserted by the general-purpose tag repair algorithm.

This example demonstrates how unclosed empty tags can distort the structure of a document. Imagine how much worse this example could get if it had fifty `<pgbrk>` tags in it.

To understand the ramifications of this, consider how the markup applied above would be processed by a query that specifies an XPath such as `/doc/para`. The first paragraph will match this XPath, but the second will not, because it has been loaded incorrectly as the child of a pgbrk element. While alternative XPath expressions such as `/doc//para` will gloss over this difference, it would be more desirable simply to load the content correctly in the first place (indentation and line breaks added for clarification):

```
<book>
  <para>
    This is the first paragraph.
  </para>
  <pgbrk/>
  <para>
    This paragraph has a cross-reference
    <xref id="f563t001"/>
    in some
    <italic>italic</italic>
    text.
  </para>
</book>
```

4.2.2 How Empty Tag Auto-Close Works

Empty tag auto-close is a special case of schema-driven tag repair and is supported in all versions of MarkLogic Server.

To take advantage of the empty tag auto-close functionality, you must first define a schema that describes which tags should be assumed to be empty tags. Given this information, when the XML loader is loading content from an external source, it will automatically close these tags as soon as they are encountered. If some of the specified tags are in fact accompanied by closing tags, these closing tags will be discarded by the general-purpose tag repair algorithm.

Here is an example of a schema that instructs the loader to treat as empty tags any `<xref>`, `<graphic>` and `<pgbrk>` tags found in documents governed by the `http://www.mydomain.com/sgml` namespace:

```
<xs:schema
  targetNamespace="http://www.mydomain.com/sgml"
  xsi:schemaLocation="http://www.w3.org/2001/XMLSchema XMLSchema.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <xs:complexType name="empty"/>
  <xs:element name="xref" type="empty"/>
  <xs:element name="graphic" type="empty"/>
  <xs:element name="pgbrk" type="empty"/>
</xs:schema>
```

If the sample SGML document shown earlier were to be loaded under the control of this simple schema, it would be corrected as desired.

To use either of these two schemas for content repair, two things are required:

- The schema must be loaded into MarkLogic Server.
- The content to be loaded must properly reference the schema at load-time.

4.2.3 How to Perform Empty Tag Auto-Close

There are multiple ways to invoke the empty tag auto-close functionality. The recommended course of action is as follows:

1. Author a schema that specifies which tags should be treated as empty tags. The simple schema shown above is a good starting point.

2. Load the schema into MarkLogic Server. See “Loading Schemas” on page 23 for instructions on properly loading schemas in MarkLogic Server.

3. Make sure that the content to be loaded references the namespace of the schema you have loaded. For the first simple schema shown above, the root element could take one of two forms. In the first form, the document implicitly references the schema through its namespace:

   ```
   <document
     xmlns="http://www.mydomain.com/sgml">
     ...
   </document>
   ```

   MarkLogic Server automatically looks for matching schema whenever a document is loaded. See “Loading Schemas” on page 23 for an in-depth discussion of the precedence rules that are applied in the event that multiple matching schemas are found.

   The following example shows how one of multiple matching schemas can be explicitly referenced by the document being loaded:

   ```
   <document
     xmlns="http://www.mydomain.com/sgml"
     xsi:schemaLocation="http://www.mydomain.com/sgml /sch/SGMLEmpty.xsd"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema">
     ...
   </document>
   ```

   The construct shown above explicitly references the schema stored at URI `/sch/SGMLEmpty.xsd` in the current schema database. If there is no schema stored at that URI, or the schema stored at that URI has a target namespace other than `http://www.mydomain.com/sgml`, no schema will be used.

4. Load the content using `xdmp:document-load()` or any of the XDBC document insertion methods.

After the content has been loaded, you can inspect it to see that the content repair has been performed. If auto-close repair has not been carried out, then you should troubleshoot the placement, naming and cross-referencing of your schema, as this is the most likely source of the problem.

If it is not feasible to modify the content so that it properly references a namespace in its header, there are other approaches that will yield the same result:

1. Author a schema that specifies which tags should be treated as empty tags. Because the root `xs:schema` element lacks a `targetNamespace` attribute, the document below specifies a schema that applies to documents loaded in the unnamed namespace:

```
<xs:schema
  xsi:schemaLocation="http://www.w3.org/2001/XMLSchema XMLSchema.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <xs:complexType name="empty"/>
  <xs:element name="xref" type="empty"/>
  <xs:element name="graphic" type="empty"/>
  <xs:element name="pgbrk" type="empty"/>
</xs:schema>
```

2. Load the schema into MarkLogic Server, remembering the URI name under which you store the schema. See “Loading Schemas” on page 23 for instructions on properly loading schema in MarkLogic Server.

3. Construct an XQuery statement that temporarily imports the schema into the appropriate namespace and loads the content within that context. Following is a simple example of importing a schema into the unnamed namespace:

```
xquery version "0.9-ml"
import schema namespace "myNS" at "schema-uri-you-specified-in-step-2";
xdmp:document-load("content-to-be-repaired.sgml", ...)
```

   Be careful to restrict the content loading operations you carry out within the context of this `import schema` directive, as all documents loaded in the unnamed namespace will be filtered through the empty tag auto close algorithm under the control of this schema.

   Note: The target namespace specified in the `import schema` prolog statement and in the schema document itself must be the same, otherwise the schema import will fail silently.

4. Run the query shown above to load and repair the content.

4.2.4 Scope of Application

Once a schema has been configured and loaded for empty tag auto-closing, any content that references that schema and is loaded from an external source will automatically be repaired as directed by that schema.

4.2.5 Disabling Empty Tag Auto-Close

There are several ways to turn off load-time empty-tag auto-closing:

1. Disable content repair at load-time using the appropriate parameter to `xdmp:document-load()`, `xdmp:unquote()` or to the XDBC `openDocInsertStream()` method.

2. Remove the corresponding schema from the database and ensure that none of the content to be loaded in the future still references that schema.

3. Modify the referenced schema to remove the empty tag definitions.

Removing the schema from the database will not impact documents already loaded under the rubric of that schema, at least with respect to their empty tags being properly closed. To the extent that the schema in question contains other information about the content that is used during query processing, you should consider the removal of the schema from the database carefully.

4.3 Schema-Driven Tag Repair

MarkLogic Server supports the use of schemas for empty tag auto-closing and for more complex schema-driven tag repair.

This functionality allows the developer to provide, through a schema, a set of general rules that govern how various elements interact hierarchically within a document. This can help with more complex content repair situations.

4.3.1 What Schema-Driven Tag Repair Does

For example, consider the following SGML document snippet:

```
<book>
<section><para>This is a paragraph in section 1.
<section><para>This is a paragraph in section 2.
</book>
```

This snippet illustrates one of the key challenges created by interpreting markup languages as XML. Under default settings, the server will repair and load this content as follows (indentation and line breaks added for clarification):

```
<book>
  <section>
    <para>
    This is a paragraph in section 1.
      <section>
        <para>This is a paragraph in section 2.</para>
      </section>
    </para>
  </section>
</book>
```

The repaired content shown above is almost certainly not what the author intended. However, it is all that the server can accomplish using only general-purpose tag repair.

Schema-driven content repair improves the situation by allowing the developer to indicate, via a schema, constraints in the relationships between elements. In this case, the developer may indicate that a `<section>` element may only contain `<para>` elements. Therefore, a `<section>` element cannot be a child of another `<section>` element. In addition, the developer can indicate that `<para>` element is a simple type that only contains text. Armed with this knowledge, the server can improve the quality of content repair that it performs. For example, the server can use the schema to know that it should check to see if there is an open `<section>` element on the stack whenever it encounters a new `<section>` element.

The resulting repair of the SGML document snippet shown above will be closer to the original intent of the document author:

```
<book>
  <section>
    <para>
      This is a paragraph in section 1.
    </para>
  </section>
  <section>
    <para>
      This is a paragraph in section 2.
    </para>
  </section>
</book>How it works
```

To take advantage of schema-driven tag repair, you must first define a schema that describes the constraints on the relationships between elements. Given this information, when the XML loader is loading content from an external source, it will automatically close tags still open on its stack when it encounters an open tag that would violate the specified constraints.

Unlike general-purpose tag repair, which is triggered by unexpected *closing* tags, schema-driven tag repair is triggered by unexpected *opening* tags, so the two different repair models interoperate cleanly. In the worst case, schema-driven tag repair may, as directed by the governing schema for the document being loaded, automatically close an element sooner than that element is explicitly closed in the document itself. This case will only occur when the relationship between elements in the document is at odds with the constraints described in the schema, in which case the schema is used as the dominating decision factor.

The following is an example of a schema that specifies the following constraints:

- `<book>` elements in the `http://www.mydomain.com/sgml` namespace can only contain `<section>` elements.
- `<section>` elements in the `http://www.mydomain.com/sgml` namespace can only contain `<para>` elements.
- `<para>` elements in the `http://www.mydomain.com/sgml` namespace can only contain text.

```
<xs:schema
  targetNamespace="http://www.mydomain.com/sgml"
  xsi:schemaLocation="http://www.w3.org/2001/XMLSchema XMLSchema.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <xs:complexType name="book">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="section"/>
    </xs:choice>
  </xs:complexType>

  <xs:complexType name="section">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="para"/>
    </xs:choice>
  </xs:complexType>

  <xs:element name="book" type="book"/>
  <xs:element name="section" type="section"/>
  <xs:element name="para" type="xs:string"/>
</xs:schema>
```

If the sample SGML document shown above were to be loaded under the control of this simple schema, it would be corrected as desired.

To make this happen, two things are required:

- The schema must be loaded into MarkLogic Server.
- The content to be loaded must properly reference this schema at load-time.

4.3.2 Loading Content with Schema-Driven Tag Repair

As with many such things, there are multiple ways to perform schema-driven correction. The recommended course of action is as follows:

1. Author a schema that describes the relationships between the elements.

2. Load the schema into MarkLogic Server. See "Loading Schemas" on page 23 for instructions on properly loading schema in MarkLogic Server.

3. Ensure that the root element of the content to be loaded properly references the appropriate schema. See "How to Perform Empty Tag Auto-Close" on page 33 for examples of referencing schema from within content.

4. Load the content using `xdmp:document-load()` or any of the XDBC document insertion methods.

After the content has been loaded, you can inspect it to see that the content repair has been performed. If the appropriate content repair has not occurred, then you should troubleshoot the placement, naming and cross-referencing of your schema.

If it is not feasible to modify the content so that it properly references the schema in its header, there are other approaches that will yield the same result:

1. Author a schema that describes the relationships between the elements, and omit a `targetNamespace` attribute from its `xs:schema` root element.

2. Load the schema into MarkLogic Server, remembering the URI name under which you store the schema. See "Loading Schemas" on page 23 for instructions on properly loading schema in MarkLogic Server.

3. Construct an XQuery statement that temporarily imports the schema into the appropriate namespace and loads the content within that context. Following is a simple example of importing a schema into the unnamed namespace:

```
xquery version "0.9-ml"
import schema namespace "myNS" at "schema-uri-you-specified-in-step-1";
xdmp:document-load("content-to-be-repaired.sgml", ...)
```

Be careful to restrict the content loading operations you carry out within the context of this `import schema` directive, as all documents loaded will be filtered through the same schema-driven content repair algorithm.

Note: The target namespace specified in the `import schema` prolog statement and in the schema document itself must be the same, otherwise the schema import will fail silently.

4. Run the query shown above to load and repair the content.

4.3.3 Scope of Application

Once a schema has been configured and loaded for schema-driven tag repair, any content that references that schema and is loaded from an external source will automatically be repaired as directed by that schema.

4.3.4 Disabling Schema-Driven Tag Repair

There are several ways to turn off load-time schema-driven correction:

1. Disable content repair at load-time using the appropriate parameter to `xdmp:document-load()`, `xdmp:unquote()` or to XDBC's `openDocInsertStream()` method.

2. Remove the corresponding schema from the database and ensure that none of the content to be loaded in the future still references that schema.

3. Modify the referenced schema to remove the empty tag definitions.

Removing the schema from the database will not impact documents already loaded under the rubric of that schema. To the extent that the schema in question contains other information about the content that is used during query processing, you should consider the removal of the schema from the database carefully.

4.4 Load-Time Default Namespace Assignment

When documents are loaded into MarkLogic Server, every element is stored with a complete QName comprised of a namespace URI and a localname.

However, many XML files are authored without specifying a default namespace or a namespace for any of their elements. When these files are loaded from external sources into MarkLogic Server, the server applies the default unnamed namespace to all the nodes that do not have an associated namespace.

However, in some situations, this is not the desired result. Once the document has been loaded without a specified namespace, it is difficult to remap each QName to a different namespace. The developer would rather have the document loaded into MarkLogic Server with the desired default namespace in the first place.

The best way to specify a default namespace for a document is by adding a default namespace attribute to the document’s root node directly. When that is not possible, MarkLogic Server's load-time namespace substitution capability offers a good solution. You can specify a default namespace for the document at load-time, provided that the document root node does not already contain a default namespace specification.

Note: This function is performed as described below if a default namespace is specified at load time, even if content repair is turned off.

4.4.1 How Default Namespace Assignments Work

The `xdmp:document-load()` function and the XDBC document insertion method both allow the developer to optionally specify a namespace as the default namespace for an individual document loading operation.

The server uses that namespace definition as follows:

1. If the root node of the document does not contain the default namespace attribute, the server uses the provided namespace as the default namespace for the root node. The appropriate namespaces of descendant nodes are then determined through the standard namespace rules.

2. If the root node of the document incorporates a default namespace attribute, the server ignores the provided namespace.

Note that rule 2 means that the default namespace provided at load time cannot be used to override an explicitly specified default namespace at the root element

4.4.2 Scope of Application

You can optionally specify default namespaces at load-time as a parameter to `xdmp:document-load()` and to the XDBC `openDocInsertStream()` method. See the corresponding documentation for further details. All other methods for loading, updating or replacing content in the database do not have access to this functionality.

4.5 Load-Time Namespace Prefix Binding

The original XML specifications allow the use of colons in element names e.g. `<myprefix:a>`. However, according to the XML Namespace specifications (developed after the initial XML specifications), the string before a colon in an element name is interpreted as a namespace prefix. The use of prefixes that are not bound to namespaces is deemed as non-compliant with the XML Namespace specifications.

Prior to version 2.1, MarkLogic Server drops unresolved prefixes from documents loaded into the database in order to conform to the XML Namespace specifications. Consider a document named `mybook.xml` that contains the following content:

```
<publisher:book>
  <section>
    This is a section.
  </section>
</publisher:book>
```

If publisher is not bound to any namespace, `mybook.xml` will be loaded into the database as:

```
<book>
  <section>
    This is a section.
  </section>
<book>
```

Starting in 2.1, MarkLogic Server supports more powerful correction of XML documents with unresolved namespace bindings. If content repair is on, `mybook.xml` will be loaded with a namespace binding added for the publisher prefix.

```
<publisher:book
 xmlns:publisher="appropriate namespace-see details below">
  <section>
    This is a section.
  </section>
</publisher:book>
```

If content repair is off, MarkLogic Server will return an error if unresolved namespace prefixes are encountered at load time.

4.5.1 How Load-Time Namespace Prefix Binding Works

If content repair is on, MarkLogic Server allows a developer to create namespace bindings at load time for namespace prefixes that would otherwise be unresolved.

Namespace prefixes are resolved using the rules below. The rules are listed in order of precedence:

1. If the prefix is explicitly bounded in the document, that binding is retained. For example, in the example below, the binding for publisher to `"http://publisherA.com"` is specified in the document and is retained.

```
<publisher:book xmlns:publisher="http://publisherA.com">
  <section>
    This is a section.
  </section>
</publisher:book>
```

2. If the prefix is declared in the XQuery environment, that binding is used. Suppose that `mybook.xml`, the document being loaded, contains the following content:

```
<publisher:book>
  <section>
    This is a section.
```

```
  </section>
</publisher:book>
```

In addition, suppose that publisher is bound to `http://publisherB.com` in the XQuery environment:

```
declare namespace publisher = "http://publisherB.com"

xdmp:document-load("mybook.xml")
```

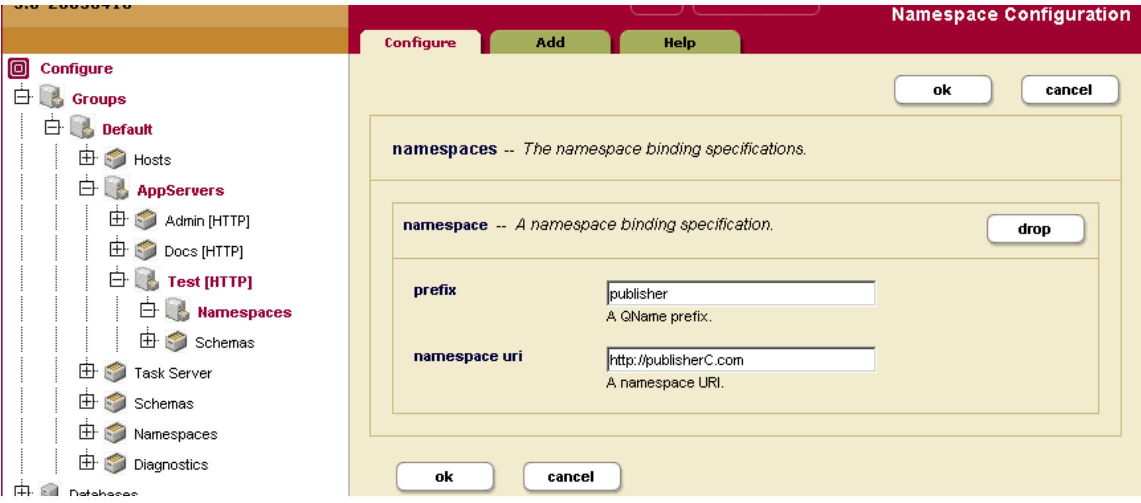
The code snippet will load the `mybook.xml` as:

```
<publisher:book xmlns:publisher="http://publisherB.com">
  <section>
    This is a section.
  </section>
</publisher:book>
```

Note: This does not apply for XDBC document insertion methods.

3. If the prefix is declared in the Admin Interface for the HTTP or XDBC server through which the document is loaded, that binding is used.

   For example, imagine a scenario in which the namespace prefix publisher is defined on the HTTP server named *Test*.



Then, suppose that the following code snippet is executed on Test:

```
xdmp:document-load("mybook.xml")
```

The initial document `mybook.xml` as shown in the second case will be loaded as:

```
<publisher:book xmlns:publisher="http://publisherC.com">
  <section>
    This is a section.
  </section>
</publisher:book>
```

4. If no binding for the prefix is found, the server creates a namespace that is the same as the prefix and binds it to the prefix. In this instance, `mybook.xml` will be loaded as:

```
<publisher:book xmlns:publisher="publisher">
  <section>
    This is a section.
  </section>
</publisher:book>
```

4.5.2 Interaction with Load-Time Default Namespace Assignment

While both load-time default namespace assignment and load-time namespace prefix binding involve document namespaces, the two features work independently. The former allows the assignment of a default namespace at the root element level, while the latter creates bindings for namespaces that are otherwise unresolved.

Consider the examples below:

1. This document has neither a binding for the `publisher` prefix, nor a default namespace.

```
<publisher:book>
  <section>
    This is a section.
  </section>
</publisher:book>
```

Suppose a default namespace `http://publisher.com/default-namespace` is specified at load time, and the publisher prefix resolves to `http://publisher.com/prefix` according to the rules described in the previous section. The document will be loaded as:

```
<publisher:book xmlns:publisher="http://publisher.com/prefix"
 xmlns="http://publisher.com/default-namespace">
  <section>
    This is a section.
  </section>
</publisher:book>
```

In this case, `<book>` is in the `"http://publisher.com/prefix"` namespace, while `<section>` is in the `"http://publisher.com/default-namespace"` namespace.

2. This document has a binding for the `publisher` prefix, but does not specify a default namespace in the root node.

```
<publisher:book xmlns:publisher="http://publisher.com/prefix">
  <section>
    This is a section.
  </section>
</publisher:book>
```

If `http://publisher.com/default-namespace` is specified as the default namespace at load time, the loaded document will be the same as the document loaded in the example above.

3. This document specifies a default namespace, but does not contain a binding for the publisher prefix, this time, associated with the `<section>` element.

```
<book xmlns="http://publisher.com/original-namespace">
  <publisher:section>
    This is a section.
    <paragraph>
      This is a paragraph.
    </paragraph>
  </publisher:section>
</book>
```

If a default namespace `http://publisher.com/default-namespace` is specified at load time, it will be ignored. Assume that publisher resolves to `publisher`. The document will be loaded as shown below:

```
<book xmlns="http://publisher.com/original-namespace">
  <publisher:section xmlns:publisher="publisher">
    This is a section.
    <paragraph>
      This is a paragraph.
    </paragraph>
  </publisher:section>
</book>
```

In this case, the `<book>` and `<paragraph>` elements are in the default namespace `http://publisher.com/original-namespace`, while the `<section>` element is in the `publisher` namespace.

4.5.3 Scope of Application

Load-time namespace prefix binding is carried out on all documents loaded from external sources. This includes documents loaded using the XQuery built-in `xdmp:document-load()` and the XDBC document insertion methods.

4.5.4 Disabling Load-Time Namespace Prefix Binding

By default, the server attempts to create bindings for unresolved namespace prefixes as a form of content repair for all documents loaded from external sources.

MarkLogic Server allows you to disable content repair during any individual document load through an optional parameter to `xdmp:document-load()`, `xdmp:unquote()` and XDBC's `openDocInsertStream()` method. See the *MarkLogic Built-In and Module Functions Reference* or the XDBC javadocs for more details.

4.6 Query-Driven Content Repair

All the content repair models described above influence the content as it is loaded, trying to ensure that the structure of the poorly or inconsistently formatted content is as close to the author's intent as possible when it is first stored in the database.

When a situation requires content repair that is beyond the scope of some combination of these four approaches, it is important to remember that MarkLogic Server's schema-independent core makes XQuery itself a powerful content repair mechanism.

Once a document has been loaded into MarkLogic Server, queries can be written to specifically restructure that content as required, without needing to reconfigure the database in any way. There are two fundamental approaches to query-driven content repair: point repair and document walkers:

4.6.1 Point Repair

Point repair uses XPath-based queries to identify document subtrees of interest, create repaired content structures from the source content, and then call `xdmp:node-replace()` to replace the document subtree of interest. A simple example of such a query is shown below:

```
for $node-to-be-repaired in doc($uri-to-be-repaired)//italic
return
  xdmp:node-replace($node-to-be-repaired,
    <i>{ $node-to-be-repaired/* }</i>)
```

This sample code finds every element with localname italic in the default element namespace and changes its QName to localname i in the default element namespace. All of the element's attributes and descendants are inherited as is.

An important constraint of the XQuery shown above lies in its assumption that italic elements cannot be descendants of other italic elements, a constraint that should be enforced at load-time using schema-driven content repair. If such a situation occurs in the document specified by $uri-to-be-repaired, the above XQuery will generate an error.

4.6.2 Document Walkers

Document walkers use recursive descent document processing functions written in XQuery to traverse either the entire document or a subtree within it, create a transformed (and appropriately repaired) version of the document, and then call `xdmp:document-insert()` or `xdmp:node-replace()` to place the repaired content back into the database.

Queries involving document traversal are typically more complex than point repair queries, because they deal with larger overall document context. Because they can also traverse the entire document, the scope of repairs that they can address is also significantly broader.

The `walk-tree()` function shown below uses a recursive descent parser to traverse the entire document:

```
xquery version "1.0-ml";
declare function local:walk-tree(
  $node as node())
as node()
{
  if (xdmp:node-kind($node) = "element") then
    (: Reconstruct node and its attributes; descend to its children :)
    element { fn:node-name($node) }  {
      $node/@*,
      for $child-node in $node/node()
      return
        local:walk-tree($child-node)
    }
  else if (xdmp:node-kind($node) = "comment" or
           xdmp:node-kind($node) = "processing-instruction" or
           xdmp:node-kind($node) = "text") then
    (: Return the node as is :)
    $node
  else if (xdmp:node-kind($node) = "document") then
    document {
    (: Start descent from the document node's children :)
       for $child-node in $node/node()
       return
         local:walk-tree($child-node)
    }
  else
    (: Should never get here :)
    fn:error(
      fn:concat("Error:  Could not process node of type '",
             xdmp:node-kind($node), "'")
    )
};

let $node := text {"hello"}
return
local:walk-tree($node)
(: returns the text node containing the string "hello" :)
```

This function can be used as the starting point for any content repair query that needs to walk the entire document in order to perform its repair. By inserting further checks in each of the various clauses at will, this function can transform both the structure and the content at will. For example, consider the following modification of the first if clause:

```
if (xdmp:node-kind($node) = "element") then
  (: Reconstruct node and its attributes; descend to its children :)
  element {
    if (fn:local-name($node) != "italic") then
      fn:node-name($node)
    else
      fn:QName(fn:namespace-uri($node), "i")
  } {
    $node/@*,
    for $child-node in $node/node()
    return
      local:walk-tree($child-node)
  }
```

Inserting this code into the `walk-tree()` function will enable the function to traverse a document, finding any element whose local-name is `italic`, regardless of that element's namespace, and change that element's local-name to `i`, keeping its namespace unchanged.

You can use the above document walker as the basis for complex content transformations, effecting content repair using the database itself as the repair tool once the content has been loaded into the database.

Another common design pattern for recursive descent is to use a `typeswitch` expression. For details, see "Transforming XML Structures With a Recursive typeswitch Expression" on page 82.

5.0 Loading Documents into the Database

This section describes how to load XML files, binary files (BLOBs), and character text files (CLOBs) into the database using a variety of techniques, including using built-in XQuery functions and using a WebDAV client. The following topics are included:

- Document Formats
- Setting Formats During Loading
- Built-In Document Loading Functions
- Specifying a Forest in Which to Load a Document
- Using WebDAV to Load Documents
- Permissions on Documents

5.1 Document Formats

Every document in a MarkLogic Server database has a *format* associated with it. The format is based on the root node of the document, and is one of the following types:

- XML Format
- Binary (BLOB) Format
- Text (CLOB) Format

Once a document has been loaded as a particular format, you cannot change the format unless you replace the root node of the document with one of a different format. You can accomplish this in a number of ways, including reloading the document specifying a different format, deleting the document and then loading it again with the same URI, or replacing the root node with one of a different format.

> **Note:** Documents loaded into a MarkLogic Server database in XML or text format are always stored in UTF-8 encoding. Documents loaded in MarkLogic Server must either be in UTF-8 or you can specify the encoding of the document during loading in the load API (for example, use the `<encoding>` option to xdmp:document-load). For details about encodings, see Encodings and Collations in the *Search Developer's Guide*.

5.1.1 XML Format

Documents loaded with XML format have special characteristics that allow you to do more with them. For example, you can use XPath expressions to search through to particular parts of the document and you can use the whole range of `cts:query` constructors to to fine-grained search, including element-level search.

XML documents are indexed when they are loaded. The indexing speeds up query response time. The type of indexing is determined by the options set in the database to which the document is loaded. You can fragment XML documents by various elements in the XML, allowing you to load extremely large XML documents. The maximum size of a single XML fragment is 128 MB for 32-bit machines, 512 MB for 64-bit machines. For more details about fragmenting documents, see the *Administrator's Guide*.

5.1.2 Binary (BLOB) Format

Binary large object (BLOB) documents are loaded into the database as binary nodes. Each binary document is a single node with no children. Binary documents are typically not textual in nature and require another application to read them. Some typical binary documents are image files (for example, `.gif`, `.jpg`), Microsoft Word `.doc` files, executable program file, and so on.

For 32-bit machines, binary documents have a 128 MB size limit. For 64-bit machines, binary documents have a 512 MB size limit. The in memory tree size limit database property (on the database configuration screen in the Admin Interface) should be at least 1 or 2 megabytes larger than the largest binary document you plan on loading into the database.

Binary documents are not indexed when they are loaded.

5.1.3 Text (CLOB) Format

Character large object (CLOB) documents, or *text* documents, are loaded into the database as text nodes. Each text document is a single node with no children. Unlike binary documents, text documents are textual in nature, and you can therefore perform text searches on them. Because text documents only have a single node, however, you cannot navigate through the document structure using XPath expressions like you can with XML documents.

Some typical text documents are simple text files (`.txt`), source code files (`.cpp`, `.java`, and so on), non well-formed HTML files, or any non-XML text file.

For 32-bit machines, text documents have an 16 MB size limit. For 64-bit machines, text documents have a 64 MB size limit. The `in memory tree size` limit database property (on the database configuration screen in the Admin Interface) should be at least 1 or 2 megabytes larger than the largest text document you plan on loading into the database.

The text indexing settings in the database apply to text documents (as well as XML documents), and the indexes are created when the text document is loaded.

5.2 Setting Formats During Loading

Whenever a document is loaded into a database, it is loaded as one of the three formats: XML, binary, or text. The format is determined in the following ways:

- Implicitly Setting the Format Based on the Mimetype
- Explicitly Setting the Format with xdmp:document-load
- Determining the Format of a Document

5.2.1 Implicitly Setting the Format Based on the Mimetype

Unless the format is explicitly set in the `xdmp:document-load` command, the format of a document is determined based on the mimetype that corresponds to the URI extension of the new document. The URI extension mimetypes, along with their default formats, are set in the Mimetypes section of the Admin Interface. For example, with the default mimetype settings, documents loaded with the `xml` URI extension are loaded as XML files; therefore loading a document with a URI `http://marklogic.com/file.xml` will result in loading an XML document.

You can also use the Mimetypes section of the Admin Interface to modify any of the default content setting, create new mimetypes, or add new extensions and associate a format. For example, if you know that all of your HTML files are well-formed (or clean up nicely with content repair), you might want to change the default content loading type of URIs ending with `.html` and `.htm` to XML.

5.2.2 Explicitly Setting the Format with xdmp:document-load

When you load a document, you can explicitly set the format with on optional argument to the `xdmp:document-load` function. Explicitly setting the format overrides the default settings specified in the Admin Interface mimetypes configuration screen. For example, HTML files have a default format of text, but you might have some HTML files that you know are well-formed, and can therefore be loaded as XML.

> **Note:** It is a good practice to explicitly set the format in the `xdmp:document-load` function rather than relying on implicit format settings based on the MIME types. When you explicitly set the format, you always know for certain the format in which a document is loaded into the database, eliminating any surprises for a document that you might want in one format but have MIME type extensions which result in the document loading in a different format.

The following example shows a load function which explicitly sets the document type to XML:

```
xdmp:document-load("c:\myFiles\file.html",
          <options xmlns="xdmp:document-load">
            <uri>http://myCompany.com/file.html</uri>
            <permissions>{xdmp:default-permissions()}</permissions>
            <collections>{xdmp:default-collections()}</collections>
            <format>xml</format>
          </options>)
```

There are similar ways to explicitly set the format when loading documents using the XDBC libraries. For details, see the XDBC javadoc or the .NET XDBC API documentation.

5.2.3 Determining the Format of a Document

After a document is loaded into a database, you cannot necessarily determine by its URI if it is an XML, text, or binary document (for example, a document could have been loaded as XML even if it has a URI that ends in `.txt`). To determine the format of a document in a database, you must do a node test on the root node of the document. XQuery includes node tests to determine if a node is text (`text()`) or if a node is an XML element (`element()`). MarkLogic Server has added a node test extension to XQuery to determine if a node is binary (`binary()`).

The following code sample shows how you can use a typeswitch to determine the format of a document.

```
(: Substitute in the URI of the document you want to test :)
let $x:= doc("/my/uri.xml")/node()
return
typeswitch ( $x )
 case element() return "xml element node"
 case text() return "text node"
 case binary() return "binary node"
default return "don't know"
```

5.3 Built-In Document Loading Functions

The `xdmp:document-load`, `xdmp:document-insert`, and `xdmp:document-get` functions can all be used as part of loading documents into a database. The `xdmp:document-load` function allows you to load documents from the filesystem into the database. The `xdmp:document-insert` function allows you to insert an existing node into a document (either a new or an existing document). The `xdmp:document-get` function loads a document from disk into memory. If you are loading a new document, the combination of `xdmp:get` and `xdmp:document-insert` is equivalent to `xdmp:document-load` of a new document.

Note: The version 2.x `xdmp:load` and `xdmp:get` functions are deprecated in the current version of MarkLogic Server; in their place, use the `xdmp:document-load` and `xdmp:document-get` functions.

The basic syntax of `xdmp:document-load` is as follows:

```
xdmp:document-load(
   $location as xs:string,
   [$options as node()]
) as empty-sequence()
```

The basic syntax of `xdmp:document-insert` is as follows:

```
xdmp:document-insert(
   $uri as xs:string],
   $root as node()
   [$permissions as element(sec:permission)*],
   [$collections as xs:string*],
   [$quality as xs:integer],
   [$forest-ids as xs:unsignedLong*]
) as empty-sequence()
```

The basic syntax of `xdmp:document-get` is as follows:

```
xdmp:document-get(
   $location as xs:string],
   [$options as node()]
) as xs:node()
```

See the online XQuery Built-In and Module Function Reference for a more detailed syntax description.

5.4 Specifying a Forest in Which to Load a Document

When loading a document, you can use the `<forests>` node in an options node for `xdmp:document-load`, or the `$forest-id` argument to `xdmp:document-insert` (the sixth argument) to specify one or more forests to which the document is loaded. Specifying multiple forest IDs loads the document into one of the forests specified; the system decides which one of the specified forests to load the document. Once the document is loaded into a forest, it stays in that forest unless you delete the document, reload it specifying a different forest, or clear the forest.

Note: In order to load a document into a forest, the forest must exist and be attached to the database into which you are loading. Attempting to load a document into a forest that does not belong to the context database will throw an exception.

5.4.1 Advantages of Specifying a Forest

Because backup operations are performed at either the database or the forest level, loading a set of documents into specific forests allows you to effectively perform backup operations on that set of documents (by backing up the database or forest, for example).

Specifying a forest also allows you to have more control over the filesystems in which the documents reside. Each forest configuration includes a directory where the files are stored. By specifying the forest in which a document resides, you can control the directories (and in turn, the filesystems) in which the documents are stored. For example, you might want to place large, frequently accessed documents in a forest which resides on a RAID filesystem with complete failover and redundancy, whereas you might want to place documents which are small and rarely accessed in a forest which resides in a slower (and less expensive) filesystem.

> **Note:** Once a document is loaded into a forest, you cannot move it to another forest. If you want to change the forest in which a document resides, you must reload the document and specify another forest.

5.4.2 Example: Examining a Document to Decide Which Forest to Specify

You can use the `xdmp:document-get` function to a load a document into memory. One use for loading a document into memory is the ability to perform some processing or logic on the document before you load the document onto disk.

For example, if you want to make a decision about which forest to load a document into based on the document contents, you can put some simple logic in your load script as follows:

```
let $memoryDoc := xdmp:document-get("c:\myFiles\newDocument.xml")
let $forest :=
     if( $memoryDoc//ID gt "1000000" )
     then xdmp:forest("LargeID")
     else xdmp:forest("SmallID")
return
     xdmp:document-insert("/myCompany/newDocument.xml",
                          $memoryDoc,
          xdmp:default-permissions(),
          xdmp:default-collections(),
                          0,
                          $forest)
```

This code loads the document `newDocument.xml` into memory, finds the `ID` element in the in-memory document, and then inserts the node into the forest named `LargeID` if the `ID` is greater than 1,000,000, or inserts the node into the forest named `SmallID` if the `ID` is less than 1,000,000.

5.4.3 More Examples

The following command will load the document into the forest named `myForest`:

```
xdmp:document-load("c:\myFile.xml",
       <options xmlns="xdmp:document-load">
          <uri>/myDocs/myDocument.xml</uri>
          <permissions>{xdmp:default-permissions()}</permissions>
          <collections>{xdmp:default-collections()}</collections>
          <repair>full</repair>
          <forests>
            <forest>{xdmp:forest("myForest")}</forest>
          </forests>
        </options> )
```

The following command will load the document into either the forest named `redwood` or the forest named `aspen`:

```
xdmp:document-load("c:\myFile.xml",
       <options xmlns="xdmp:document-load">
          <uri>/myDocs/myDocument.xml</uri>
          <permissions>{xdmp:default-permissions()}</permissions>
          <collections>{xdmp:default-collections()}</collections>
          <repair>full</repair>
          <forests>
            <forest>{xdmp:forest("redwood")}</forest>
            <forest>{xdmp:forest("aspen")}</forest>
          </forests>
        </options> )
```

5.5 Using WebDAV to Load Documents

If you have configured a WebDAV server, you can use a WebDAV client to load documents into the database. WebDAV clients such as Windows Explorer allow drag and drop access to documents, just like any other documents on the filesystem. For details on setting up WebDAV servers, see the chapter on WebDAV in the *Administrator's Guide*.

5.6 Permissions on Documents

When you load any documents in a database, make sure you either explicitly set permissions in the document loading API (for example, `xdmp:document-load` or `xdmp:document-insert`) or have set default permissions on the user (or on roles that the user has) who is loading the documents. Default permissions specify what permissions a document has when it is loaded if you do not explicitly set permissions.

Permissions on a document control access to capabilities (`read`, `insert`, `update`, and `execute`) on the document. Each permission consists of a capability and a corresponding role. In order to have access to a capability for a document, a user must have the role paired with that capability on the document permission. Default permissions are specified on roles and on users in the Admin Interface. If you load a document without the needed permissions, users might not be able to read, update, or execute the document (even by the user who loaded the document). For an overview of security, see *Understanding and Using Security*. For details on creating privileges and setting permissions, see the Security chapter of the *Administrator's Guide*.

> **Note:** When you load a document, ensure that a named role has update permissions. For any document created by a user who does not have the `admin` role, the document must be created with at least one update permission or it will throw an `XDMP-MUSTHAVEUPDATE` exception during document creation. If all named roles on the document permissions do not have update permissions, or if the document has no permissions, then only users with the `admin` role will be able to update or delete the document.

6.0 Importing XQuery Modules and Resolving Paths

You can import XQuery modules from other XQuery modules in MarkLogic Server. This chapter describes the two types of XQuery modules and specifies the rules for importing modules. It includes the following sections:

- XQuery Library Modules and Main Modules
- Rules for Resolving Import, Invoke, and Spawn Paths
- Example Import Module Scenario

6.1 XQuery Library Modules and Main Modules

There are two types of XQuery modules (as defined in the XQuery specification, http://www.w3.org/TR/xquer//#id-query-prolog):

- Main Modules
- Library Modules

For more details about the XQuery language, see the *XQuery Reference Guide*.

6.1.1 Main Modules

A main module can be executed as an XQuery program, and must include a query body consisting of an XQuery expression (which in turn can contain other XQuery expressions, and so on). The following is a simple example of a main module:

```
"hello world"
```

Main modules can have prologs, but the prolog is optional. As part of a prolog, a main module can have function definitions. Function definitions in a main module, however, are only available to that module; they cannot be imported to another module.

6.1.2 Library Modules

A library module has a namespace and is used to define functions. Library modules cannot be evaluated directly; they are imported, either from other library modules or from main modules with an `import` statement. The following is a simple example of a library module:

```
xquery version "1.0-ml";
module namespace hello = "helloworld";

declare function helloworld()
{
  "hello world"
};
```

If you save this module to a file named `c:/code/helloworld.xqy`, and you have an App Server with filesystem root `c:/code`, you can import this in either a main module or a library module and call the function as follows:

```
xquery version "1.0-ml";
import module namespace hw="helloworld" at "/helloworld.xqy";

hw:helloworld()
```

This XQuery program will import the library module with the `helloworld` namespace and then evaluate its `helloworld()` function.

6.2 Rules for Resolving Import, Invoke, and Spawn Paths

In order to call a function that resides in an XQuery library module, you need to import the module with its namespace. MarkLogic Server resolves the library paths similar to the way other HTTP and application servers resolve their paths. Similarly, if you use `xdmp:invoke` or `xdmp:spawn` to run a module, you specify access to the module with a path.

The XQuery module that is imported/invoked/spawned can reside in any of the following places:

- In the Modules directory.
- In a directory relative to the calling module.
- Under the App Server root, which is either the specified directory in the Modules database (when the App Server is set to a Modules database) or the specified directory on the filesystem (when the App Server is set to find modules in the filesystem).

The paths in `import/invoke/spawn` statements are resolved as follows:

1. When an import/invoke/spawn path starts with a leading slash, first look under the Modules directory (on Windows, typically `c:\Program Files\MarkLogic\Modules`). For example:

   ```
   import module "foo" at "/foo.xqy";
   ```

   In this case, it would look for the module file with a namespace `foo` in `c:\Program Files\MarkLogic\Modules\foo.xqy`.

2. If the import/invoke/spawn path starts with a slash, and it is not found under the Modules directory, then start at the App Server root. For example, if the App Server root is `/home/mydocs/`, then the following import:

```
import module "foo" at "/foo.xqy";
```

will look for a module with namespace `foo` in `/home/mydocs/foo.xqy`.

Note that you start at the App Server root, both for filesystem roots and Modules database roots. For example, in an App Server configured with a modules database and a root of `http://foo/`:

```
import module "foo" at "/foo.xqy";
```

will look for a module with namespace `foo` in the modules database with a URI `http://foo/foo.xqy` (resolved by appending the App Server root to `foo.xqy`).

3. If the import/invoke/spawn path does not start with a slash, look relative to the location of the module that called the function. For example, if a module at `/home/mydocs/bar.xqy` has the following import:

```
import module "foo" at "foo.xqy";
```

it will look for the module with namespace `foo` at `/home/mydocs/foo.xqy`.

Note that you start at the calling module location, both for App Servers configured to use the filesystem and for App Servers configured to use modules databases. For example, a module with a URI of `http://foo/bar.xqy` that resides in the modules database and has the following `import` statement:

```
import module "foo" at "foo.xqy";
```

will look for the module with the URI `http://foo/foo.xqy` in the modules database.

4. If the import/invoke/spawn path contains a scheme or network location, then the server throws an exception. For example:

```
import module "foo" at "http://foo/foo.xqy";
```

will throw an invalid path exception. Similarly:

```
import module "foo" at "c:/foo/foo.xqy";
```

will throw an invalid path exception.

6.3 Example Import Module Scenario

Consider the following scenario:

- There is an HTTP server with a root defined as `c:/mydir`.
- In a file called `c:/mydir/lib.xqy`, there is a library module with the function to import. The contents of the library module are as follows:

```
xquery version "1.0-ml";
module namespace hw="http://marklogic.com/me/my-module";

declare function hello()
{
"hello"
};
```

- In a file called `c:/mydir/main.xqy`, there is an XQuery main module that imports a function from the above library module. This code is as follows:

```
xquery version "1.0-ml";

declare namespace my="http://marklogic.com/me/my-module";
import module "http://marklogic.com/me/my-module" at "lib.xqy";

my:hello()
```

The library module `lib.xqy` is imported relative to the App Server root (in this case, relative to `c:/mydir`).

7.0 Library Services Applications

This chapter describes how to use Library Services, which enable you to create and manage versioned content in MarkLogic Server in a manner similar to a Content Management System (CMS). This chapter includes the following sections:

- Understanding Library Services
- Building Applications with Library Services
- Required Range Element Indexes
- Library Services API
- Security Considerations of Library Services Applications
- Putting Documents Under Managed Version Control
- Checking Out Managed Documents
- Checking In Managed Documents
- Updating Managed Documents
- Defining a Retention Policy
- Managing Modular Documents in Library Services

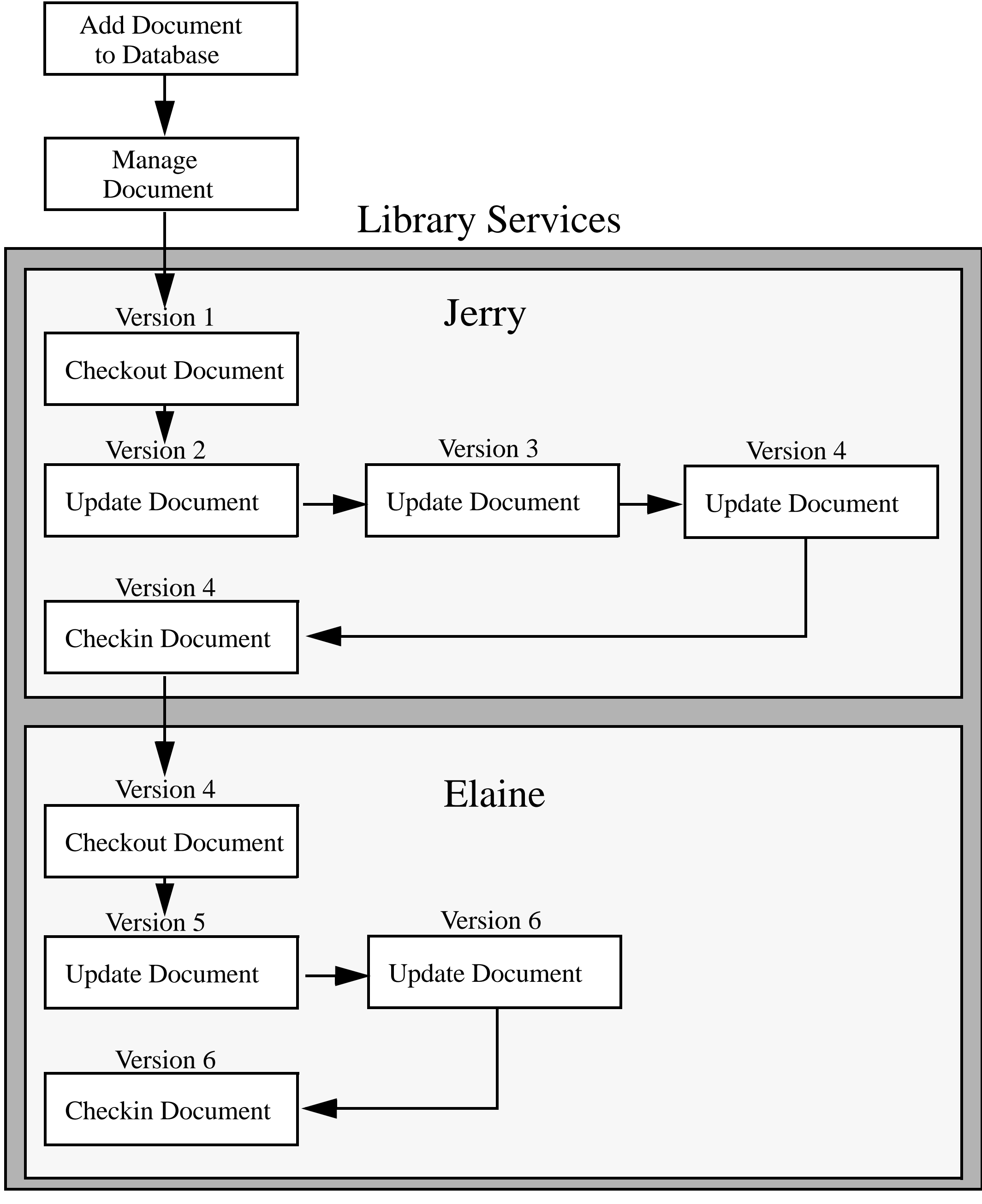
7.1 Understanding Library Services

The Library Services enable you to create and maintain versions of managed documents in MarkLogic Server. Access to managed documents is controlled using a check-out/check-in model. You must first check out a managed document before you can perform any update operations on the document. A checked out document can only be updated by the user who checked it out; another user cannot update the document until it is checked back in and then checked out by the other user.

When you initially put a document under Library Services management, it creates Version 1 of the document. Each time you update the document, a new version of the document is created. Old versions of the updated document are retained according to your retention policy, as described in "Defining a Retention Policy" on page 69.

The Library Services include functions for managing modular documents so that various versions of linked documents can be created and managed, as described in "Managing Modular Documents in Library Services" on page 76.

The following diagram illustrates the workflow of a typical managed document. In this example, the document is added to the database and placed under Library Services management. The managed document is checked out, updated several times, and checked in by Jerry. Once the document is checked in, Elaine checks out, updates, and checks in the same managed document. Each time the document is updated, the previous versions of the document are purged according to the retention policy.



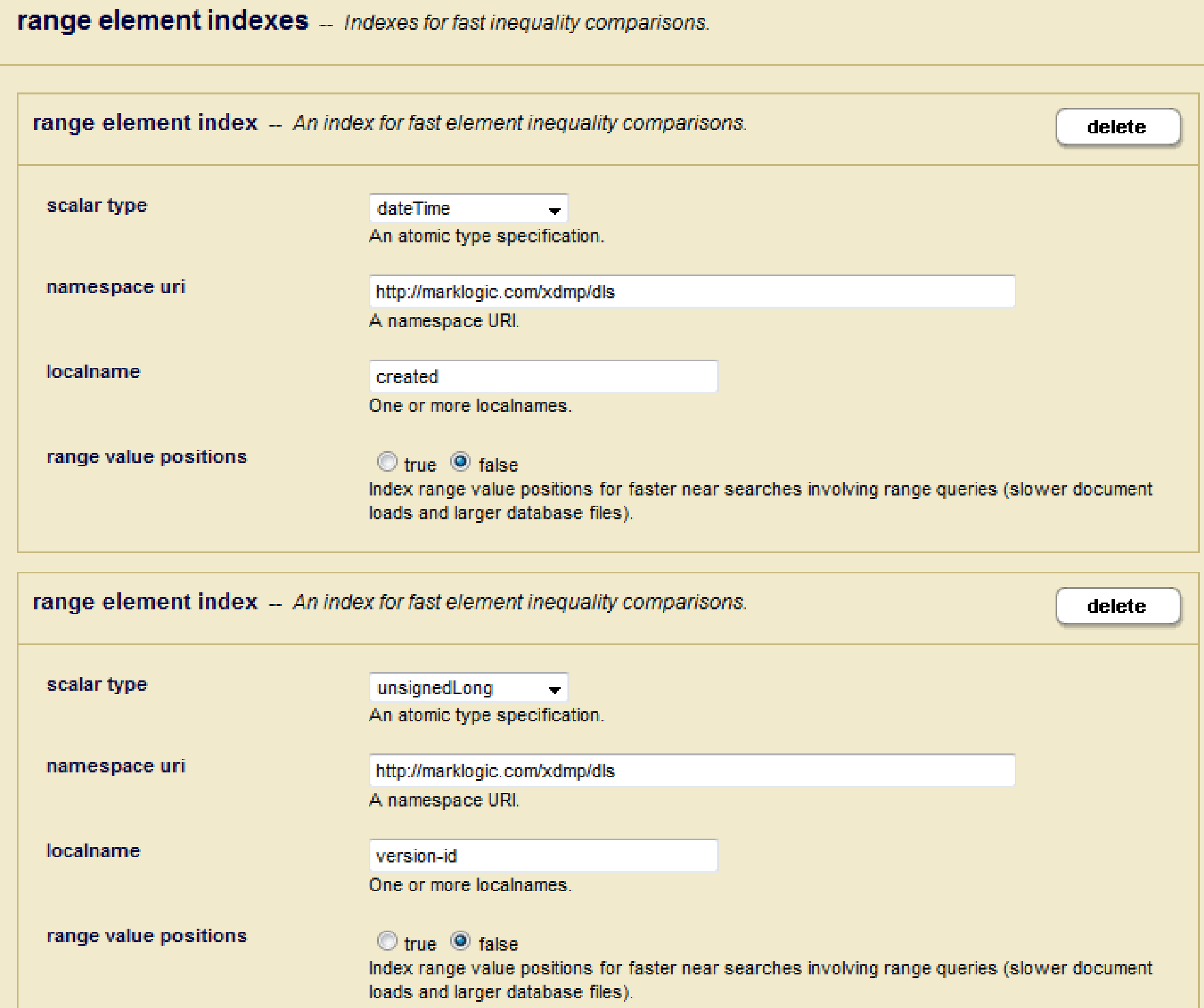
7.2 Building Applications with Library Services

The Library Services API provides the basic tools for implementing applications that store and extract specific drafts of a document as of a particular date or version. You can also use the Library Services API, along with the other MarkLogic Server APIs, to provide structured workflow, version control, and the ability to partition a document into individually managed components. The security API provides the ability to associate user roles and responsibilities with different document types and collections. And the search APIs provide the ability to implement powerful content retrieval features.

7.3 Required Range Element Indexes

The range element indexes shown in the table and figure below must be set for the database that contains the documents managed by the Library Services. These indexes are automatically set for you when you create a new database. However, if you want to enable the Library Services for a database created in an earlier release of MarkLogic Server, you must manually set them for the database.

Scalar Type	Namespace URI	Localname	Range value position
dateTime	http://marklogic.com/xdmp/dls	created	false
unsignedLong	http://marklogic.com/xdmp/dls	version-id	false



7.4 Library Services API

This section describes the Library Services API and contains the following sections:

- Library Services API Categories
- Managed Document Update Wrapper Functions

7.4.1 Library Services API Categories

The Library Services functions are described in the *MarkLogic Built-In and Module Functions Reference*. The Library Services functions fall into the following categories:

- Document management functions for putting documents under version management, checking documents in and out of version management, and so on. For usage information, see "Putting Documents Under Managed Version Control" on page 66, "Checking Out Managed Documents" on page 66 and "Checking In Managed Documents" on page 67.
- Document update functions for updating the content of documents and their properties. For usage information, see "Updating Managed Documents" on page 68 and "Managed Document Update Wrapper Functions" on page 64.
- Retention policy functions for managing when particular document versions are purged. For usage information, see "Defining a Retention Policy" on page 69.
- XInclude functions for creating and managing linked documents. For usage information, see "Managing Modular Documents in Library Services" on page 76.
- `cts:query` constructor functions for use by `cts:search`, Library Services XInclude functions, and when defining retention rules. For usage information, see "Defining a Retention Policy" on page 69.

7.4.2 Managed Document Update Wrapper Functions

All update and delete operations on managed documents must be done through the Library Services API. The Library Services API includes the following "wrapper" functions that enable you to make the same updates on managed documents as you would on non-managed document using their XDMP counterparts:

- `dls:document-add-collections`
- `dls:document-add-permissions`
- `dls:document-add-properties`
- `dls:document-set-collections`
- `dls:document-set-permissions`
- `dls:document-set-properties`
- `dls:document-remove-properties`
- `dls:document-remove-permissions`
- `dls:document-remove-collections`
- `dls:document-set-property`
- `dls:document-set-quality`

7.5 Security Considerations of Library Services Applications

There are two pre-defined roles designed for use in Library Services applications, as well as an internal role that the Library Services API uses:

- dls-admin Role
- dls-user Role
- dls-internal Role

7.5.1 dls-admin Role

The `dls-admin` role is designed to give administrators of Library Services applications all of the privileges that are needed to use the Library Services API. It has the needed privileges to perform operations such as inserting retention policies and breaking checkouts, so only trusted users (users who are assumed to be non-hostile, appropriately trained, and follow proper administrative procedures) should be granted the `dls-admin` role. Assign the `dls-admin` role to administrators of your Library Services application.

7.5.2 dls-user Role

The `dls-user` role is a minimally privileged role. It is used in the Library Services API to allow regular users of the Library Services application (as opposed to `dls-admin` users) to be able to execute code in the Library Services API. It allows users, with document update permission, to manage, checkout, and checkin managed documents.

The `dls-user` role only has privileges that are needed to run the Library Services API; it does not provide execute privileges to any functions outside the scope of the Library Services API. The Library Services API uses the `dls-user` role as a mechanism to amp more privileged operations in a controlled way. It is therefore reasonably safe to assign this role to any user whom you trust to use your Library Services application. Assign the `dls-user` role to all users of your Library Services application.

7.5.3 dls-internal Role

The `dls-internal` role is a role that is used internally by the Library Services API, but you should not explicitly grant it to any user or role. This role is used to amp special privileges within the context of certain functions of the Library Services API. Assigning this role to users would give them privileges on the system that you typically do not want them to have; do not assign this role to any users.

7.6 Putting Documents Under Managed Version Control

In order to put a document under managed version control, it must be in your content database. Once the document is in the database, users assigned the `dls-user` role can use the `dls:document-manage` function to place the document under management. Alternatively, you can use the `dls:document-insert-and-manage` function to both insert a document into the database and place it under management.

For example, the following query inserts a new document into the database and places it under Library Services management:

```
(: Insert a new managed document into the database. :)
xquery version "1.0-ml";

import module namespace dls = "http://marklogic.com/xdmp/dls"
        at "/MarkLogic/dls.xqy";

dls:document-insert-and-manage(
        "/engineering/beta_overview.xml",
        fn:true(),
        <TITLE>Project Beta Overview</TITLE>)
```

7.7 Checking Out Managed Documents

You must first use the `dls:document-checkout` function to check out a managed document before performing any update operations. For example, to check out the `beta_overview.xml` document, along with all of its linked documents, specify the following:

```
xquery version "1.0-ml";

import module namespace dls = "http://marklogic.com/xdmp/dls"
        at "/MarkLogic/dls.xqy";

dls:document-checkout(
        "/engineering/beta_overview.xml",
        fn:true(),
        "Updating doc")
```

You can specify an optional `timeout` parameter to `dls:document-checkout` that specifies how long (in seconds) to keep the document checked out. For example, to check out the `beta_overview.xml` document for one hour, specify the following:

```
dls:document-checkout(
        "/engineering/beta_overview.xml",
        fn:true(),
        "Updating doc",
        3600)
```

7.7.1 Displaying the Checkout Status of Managed Documents

You can use the `dls:document-checkout-status` function to report the status of a checked out document. For example:

```
dls:document-checkout-status("/engineering/beta_overview.xml")
```

Returns output similar to:

```
<dls:checkout xmlns:dls="http://marklogic.com/xdmp/dls">
  <dls:document-uri>/engineering/beta_overview.xml</dls:document-uri>
  <dls:annotation>Updating doc</dls:annotation>
  <dls:timeout>0</dls:timeout>
  <dls:timestamp>1240528210</dls:timestamp>
  <sec:user-id xmlns:sec="http://marklogic.com/xdmp/security">
    10677693687367813363
  </sec:user-id>
</dls:checkout>
```

7.7.2 Breaking the Checkout of Managed Documents

Users with `dls-admin` role can call `dls:break-checkout` to “un-checkout” documents. For example, if a document was checked out by a user who has since moved on to other projects, the Administrator can break the existing checkout of the document so that other users can check it out.

7.8 Checking In Managed Documents

Once you have finished updating the document, use the `dls:document-checkin` function to check it, along with all of its linked documents, back in:

```
dls:document-checkin(
        "/engineering/beta_overview.xml",
        fn:true() )
```

7.9 Updating Managed Documents

You can call the `dls:document-update` function to replace the contents of an existing managed document. Each time you call the `dls:document-update` function on a document, the document's version is incremented and a purge operation is initiated that removes any versions of the document that are not retained by the retention policy, as described in "Defining a Retention Policy" on page 69.

For example, to update the "Project Beta Overview" document, enter:

```
let $contents :=
<BOOK>
  <TITLE>Project Beta Overview</TITLE>
  <CHAPTER>
    <TITLE>Objectives</TITLE>
    <PARA>
    The objective of Project Beta, in simple terms, is to corner
    the widget market.
    </PARA>
  </CHAPTER>
</BOOK>

return
    dls:document-update(
       "/engineering/beta_overview.xml",
       $contents,
       "Roughing in the first chapter",
       fn:true())
```

Note: The `dls:document-update` function replaces the entire contents of the document.

7.10 Defining a Retention Policy

A *retention policy* specifies what document versions are retained in the database following a purge operation. A retention policy is made up of one or more *retention rules*. If you do not define a retention policy, then none of the previous versions of your documents are retained.

This section describes:

- Purging Versions of Managed Document
- About Retention Rules
- Creating Retention Rules
- Retaining Specific Versions of Documents
- Multiple Retention Rules
- Deleting Retention Rules

7.10.1 Purging Versions of Managed Document

Each update of a managed document initiates a purge operation that removes the versions of that document that are not retained by your retention policy. You can also call `dls:purge` to purge all of the documents or `dls:document-purge` to run purge on a specific managed document.

You can also use `dls:purge` or `dls:document-purge` to determine what documents *would* be deleted by the retention policy without actually deleting them. This option can be useful when developing your retention rules. For example, if you change your retention policy and want to determine specifically what document versions will be deleted as a result, you can use:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:purge(fn:false(), fn:true())
```

7.10.2 About Retention Rules

Retention rules describe which versions of what documents are to be retained by the purge operation. When using `dls:document-update` or `dls:document-extract-part` to create a new version of a document, previous versions of the document that do not match the retention policy are purged.

You can define retention rules to keep various numbers of document versions, to keep documents matching a `cts-query` expression, and/or to keep documents for a specified period of time. Restrictions in a retention rule are combined with a logical AND, so that all of the expressions in the retention rule must be true for the document versions to be retained. When you combine separate retention rules, the resulting retention policy is an OR of the combined rules (that is, the document versions are retained if they are matched by any of the rules). Multiple rules do not have an order of operation.

Warning The retention policy specifies what is *retained*, not what is purged. Therefore, anything that does not match the retention policy is removed.

7.10.3 Creating Retention Rules

You create a retention rule by calling the `dls:retention-rule` function. The `dls:retention-rule-insert` function inserts one or more retention rules into the database.

For example, the following retention rule retains all versions of all documents because the empty `cts:and-query` function matches all documents:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-insert(
dls:retention-rule(
    "All Versions Retention Rule",
    "Retain all versions of all documents",
    (),
    (),
    "Locate all of the documents",
    cts:and-query(()) ) )
```

The following retention rule retains the last five versions of all of the documents located under the `/engineering/` directory:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-insert(
dls:retention-rule(
    "Engineering Retention Rule",
    "Retain the five most recent versions of Engineering docs",
    5,
    (),
    "Locate all of the Engineering documents",
    cts:directory-query("/engineering/", "infinity") ) )
```

The following retention rule retains the latest three versions of the engineering documents with “Project Alpha” in the title that were authored by Jim:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-insert(
dls:retention-rule(
    "Project Alpha Retention Rule",
    "Retain the three most recent engineering documents with
     the title ‘Project Alpha’ and authored by Jim.",
    3,
    (),
    "Locate the engineering docs with 'Project Alpha' in the
     title authored by Jim",
    cts:and-query((
       cts:element-word-query(xs:QName("TITLE"), "Project Alpha"),
       cts:directory-query("/engineering/", "infinity"),
       dls:author-query(xdmp:user("Jim")) )) ) )
```

The following retention rule retains the five most recent versions of documents in the “specs” collection that are no more than thirty days old:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-insert(
dls:retention-rule(
    "Specs Retention Rule",
    "Keep the five most recent versions of documents in the ‘specs’
    collection that are 30 days old or newer",
    5,
    xs:duration("P30D"),
    "Locate documents in the 'specs' collection",
    cts:collection-query("http://marklogic.com/documents/specs") ) )
```

7.10.4 Retaining Specific Versions of Documents

The `dls:document-version-query` and `dls:as-of-query` constructor functions can be used in a retention rule to retain *snapshots* of the documents as they were at some point in time. A snapshot may be of specific versions of documents or documents as of a specific date.

For example, the following retention rule retains the latest versions of the engineering documents created before 5:00pm on 4/23/09:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-insert(
dls:retention-rule(
    "Draft 1 of the Engineering Docs",
    "Retain each engineering document that was update before
     5:00pm, 4/23/09",
    (),
    (),
    (),
    cts:and-query((
       cts:directory-query("/documentation/", "infinity"),
       dls:as-of-query(xs:dateTime("2009-04-23T17:00:00-07:00")) )) ))
```

If you want to retain two separate snapshots of the engineering documents, you can add a retention rule that contains a different `cts:or-query` function. For example:

```
cts:and-query((
    cts:directory-query("/documentation/", "infinity"),
    dls:as-of-query(xs:dateTime("2009-25-12T09:00:01-07:00")) ))
```

7.10.5 Multiple Retention Rules

In some organizations, it might make sense to create multiple retention rules. For example, the Engineering and Documentation groups may share a database and each organization wants to create and maintain their own retention rule.

Consider the two rules shown below. The first rule retains the latest 5 versions of all of the documents under the `/engineering/` directory. The second rule, retains that latest 10 versions of all of the documents under the `/documentation/` directory. The ORed result of these two rules does not impact the intent of each individual rule and each rule can be updated independently from the other.

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-insert((
dls:retention-rule(
    "Engineering Retention Rule",
    "Retain the five most recent versions of Engineering docs",
    5,
    (),
    "Apply to all of the Engineering documents",
    cts:directory-query("/engineering/", "infinity") ),

dls:retention-rule(
    "Documentation Retention Rule",
    "Retain the ten most recent versions of the documentation",
    10,
    (),
    "Apply to all of the documentation",
    cts:directory-query("/documentation/", "infinity") ) ))
```

As previously described, multiple retention rules define a logical OR between them, so there may be circumstances when multiple retention rules are needed to define the desired retention policy for the same set of documents.

For example, you want to retain the last five versions of all of the engineering documents, as well as all engineering documents that were updated before 8:00am on 4/24/09 and 9:00am on 5/12/09. The following two retention rules are needed to define the desired retention policy:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-insert((
dls:retention-rule(
    "Engineering Retention Rule",
    "Retain the five most recent versions of Engineering docs",
    5,
    (),
    "Retain all of the Engineering documents",
    cts:directory-query("/engineering/", "infinity") ),

dls:retention-rule(
    "Project Alpha Retention Rule",
    "Retain the engineering documents that were updated before
     the review dates below.",
    (),
    (),
    "Retain all of the Engineering documents updated before
     the two dates",
    cts:and-query((
      cts:directory-query("/engineering/", "infinity"),
      cts:or-query((
        dls:as-of-query(xs:dateTime("2009-04-24T08:00:17.566-07:00")),
        dls:as-of-query(xs:dateTime("2009-05-12T09:00:01.632-07:00"))
        ))
     )) ) ))
```

It is important to understand the difference between the logical OR combination of the above two retention rules and the logical AND within a single rule. For example, the OR combination of the above two retention rules is not same as the single rule below, which is an AND between retaining the last five versions and the as-of versions. The end result of this rule is that the last five versions are not retained and the as-of versions are only retained as long as they are among the last five versions. Once the revisions of the last five documents have moved past the as-of dates, the AND logic is no longer true and you no longer have an effective retention policy, so no versions of the documents are retained.

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-insert(
dls:retention-rule(
    "Project Alpha Retention Rule",
    "Retain the 5 most recent engineering documents",
    5,
    (),
    "Retain all of the Engineering documents updated before
     the two dates",
    cts:and-query((
      cts:directory-query("/engineering/", "infinity"),
      cts:or-query((
        dls:as-of-query(xs:dateTime("2009-04-24T08:56:17.566-07:00")),
        dls:as-of-query(xs:dateTime("2009-05-12T08:59:01.632-07:00"))
       )) )) ) )
```

7.10.6 Deleting Retention Rules

You can use the `dls:retention-rule-remove` function to delete retention rules. For example, to delete the “Project Alpha Retention Rule,” use:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-remove("Project Alpha Retention Rule")
```

To delete all of your retention rules in the database, use:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-remove(fn:data(dls:retention-rules("*")//dls:name))
```

7.11 Managing Modular Documents in Library Services

As described in "Reusing Content With Modular Document Applications" on page 126, you can create modular documents from the content stored in one or more linked documents. This section describes:

- Creating Managed Modular Documents
- Expanding Managed Modular Documents
- Managing Versions of Modular Documents

7.11.1 Creating Managed Modular Documents

As described in "Reusing Content With Modular Document Applications" on page 126, you can create modular documents from the content stored in one or more linked documents. The `dls:document-extract-part` function provides a shorthand method for creating modular managed documents. This function extracts a child element from a managed document, places the child element in a new managed document, and replaces the extracted child element with an XInclude reference.

For example, the following function call extracts Chapter 1 from the "Project Beta Overview" document:

```
dls:document-extract-part("/engineering/beta_overview_chap1.xml",
    fn:doc("/engineering/beta_overview.xml")//CHAPTER[1],
    "Extracting Chapter 1",
    fn:true() )
```

The contents of `/engineering/beta_overview.xml` is now as follows:

```
<BOOK>
  <TITLE>Project Beta Overview</TITLE>
  <xi:include href="/engineering/beta_overview_chap1.xml"/>
</BOOK>
```

The contents of `/engineering/beta_overview_chap1.xml` is as follows:

```
<CHAPTER>
  <TITLE>Objectives</TITLE>
  <PARA>
   The objective of Project Beta, in simple terms, is to corner
   the widget market.
  </PARA>
</CHAPTER>
```

Note: The newly created managed document containing the extracted child element is initially checked-in and must be checked out before you can make any updates.

The `dls:document-extract-part` function can only be called once in a transaction for the same document. There may be circumstances in which you want to extract multiple elements from a document and replace them with XInclude statements. For example, the following query creates separate documents for all of the chapters from the "Project Beta Overview" document and replaces them with XInclude statements:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

declare namespace xi="http://www.w3.org/2001/XInclude";

let $includes := for $chap at $num in
   doc("/engineering/beta_overview.xml")/BOOK/CHAPTER

return (
   dls:document-insert-and-manage(
     fn:concat("/engineering/beta_overview_chap", $num, ".xml"),
     fn:true(),
     $chap),

   <xi:include href="/engineering/beta_overview_chap{$num}.xml"
    xmlns:xi="http://www.w3.org/2001/XInclude"/>
  )

let $contents :=
   <BOOK>
      <TITLE>Project Beta Overview</TITLE>
      {$includes}
   </BOOK>

return
   dls:document-update(
      "/engineering/beta_overview.xml",
      $contents,
      "Chapters are XIncludes",
      fn:true() )
```

This query produces a "Project Beta Overview" document similar to the following:

```
<BOOK>
   <TITLE>Project Beta Overview</TITLE>
   <xi:include href="/engineering/beta_overview_chap1.xml"
    xmlns:xi="http://www.w3.org/2001/XInclude"/>
   <xi:include href="/engineering/beta_overview_chap1.xml"
    xmlns:xi="http://www.w3.org/2001/XInclude"/>
   <xi:include href="/engineering/beta_overview_chap2.xml"
    xmlns:xi="http://www.w3.org/2001/XInclude"/>
</BOOK>
```

7.11.2 Expanding Managed Modular Documents

Module documents can be "expanded" so that you can view the entire node, complete with its linked nodes, or a specific linked node. You can expand a modular document using `dls:node-expand`, or a linked node in a modular document using `dls:link-expand` .

> **Note:** When using the `dls:node-expand` function to expand documents that contain XInclude links to specific versioned documents, specify the `$restriction` parameter as an empty sequence.

For example, to return the expanded `beta_overview.xml` document, you can use:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

let $node := fn:doc("/engineering/beta_overview.xml")

return dls:node-expand($node, ())
```

To return the first linked node in the `beta_overview.xml` document, you can use:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

declare namespace xi="http://www.w3.org/2001/XInclude";

let $node := fn:doc("/engineering/beta_overview.xml")

return dls:link-expand(
               $node,
               $node/BOOK/xi:include[1],
               () )
```

The `dls:node-expand` and `dls:link-expand` functions allow you to specify a `cts:query` constructor to restrict what document version is to be expanded. For example, to expand the most recent version of the "Project Beta Overview" document created before 1:30pm on 4/6/09, you can use:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

let $node := fn:doc("/engineering/beta_overview.xml")

return dls:node-expand(
         $node,
         dls:as-of-query(
            xs:dateTime("2009-04-06T13:30:33.576-07:00")) )
```

7.11.3 Managing Versions of Modular Documents

Library Services can manage modular documents so that various versions can be created for the linked documents. As a modular document's linked documents are updated, you might want to take periodic snapshots of the entire node.

For example, as shown in "Creating Managed Modular Documents" on page 76, the "Project Beta Overview" document contains three chapters that are linked as separate documents. The following query takes a snapshot of the latest version of each chapter and creates a new version of the "Project Beta Overview" document that includes the versioned chapters:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

declare namespace xi="http://www.w3.org/2001/XInclude";

(: For each chapter in the document, get the URI :)
let $includes :=
  for $chap at $num in doc("/engineering/beta_overview.xml")
//xi:include/@href

(: Get the latest version of each chapter :)
let $version_number :=
  fn:data(dls:document-history($chap)//dls:version-id)[last()]

let $version :=  dls:document-version-uri($chap, $version_number)

(: Create an XInclude statement for each versioned chapter :)
  return
    <xi:include href="{$version}"/>

(: Update the book with the versioned chapters :)
let $contents :=
  <BOOK>
    <TITLE>Project Beta Overview</TITLE>
    {$includes}
  </BOOK>

return
  dls:document-update(
    "/engineering/beta_overview.xml",
    $contents,
    "Latest Draft",
    fn:true() )
```

The above query results in a new version of the “Project Beta Overview” document that looks like:

```
<BOOK>
  <TITLE>Project Beta Overview</TITLE>
  <xi:include 
href="/engineering/beta_overview_chap1.xml_versions/4-beta_overview_
chap1.xml" xmlns:xi="http://www.w3.org/2001/XInclude"/>
  <xi:include 
href="/engineering/beta_overview_chap2.xml_versions/3-beta_overview_
chap2.xml" xmlns:xi="http://www.w3.org/2001/XInclude"/>
  <xi:include 
href="/engineering/beta_overview_chap3.xml_versions/3-beta_overview_
chap3.xml" xmlns:xi="http://www.w3.org/2001/XInclude"/>
</BOOK>
```

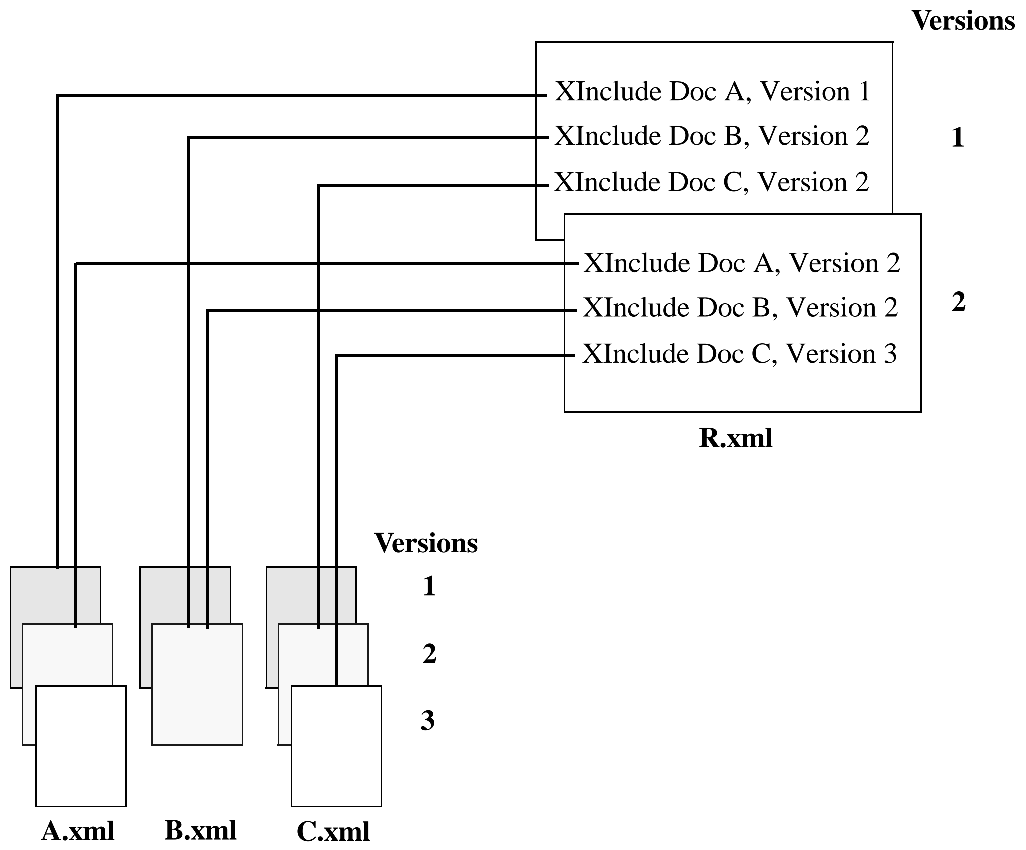
Note: When using the `dls:node-expand` function to expand modular documents that contain XInclude links to specific versioned documents, specify the `$restriction` parameter as an empty sequence.

You can also create modular documents that contain different versions of linked documents. For example, in the illustration below, Doc R.xml, Version 1 contains the contents of:

- Doc A.xml, Version 1
- Doc B.xml, Version 2
- Doc C.xml, Version 2

While Doc X, Version 2 contains the contents of:

- Doc A.xml, Version 2
- Doc B.xml, Version 2
- Doc C.xml, Version 3



8.0 Transforming XML Structures With a Recursive typeswitch Expression

A common task required with XML is to transform one structure to another structure. This chapter describes a design pattern using the XQuery `typeswitch` expression which makes it easy to perform complex XML transformations with good performance, and includes some samples illustrating this design pattern. It includes the following sections:

- XML Transformations
- Sample XQuery Transformation Code

8.1 XML Transformations

Programmers are often faced with the task of converting one XML structure to another. These transformations can range from very simple element name change transformations to extremely complex transformations that reshape the XML structure and/or combine it with content from other documents or sources. This section describes some aspects of XML transformations and includes the following sections:

- XQuery vs. XSLT
- Transforming to XHTML or XSL-FO
- The typeswitch Expression

8.1.1 XQuery vs. XSLT

XSLT is commonly used in transformations, and it works well for many transformations. It does have some drawbacks for certain types of transformations, however, especially if the transformations are part of a larger XQuery application.

XQuery is a powerful programming language, and MarkLogic Server provides very fast access to content, so together they work extremely well for transformations. MarkLogic Server is particularly well suited to transformations that require searches to get the content which needs transforming. For example, you might have a transformation that uses a lexicon lookup to get a value with which to replace the original XML value. Another transformation might need to count the number of authors in a particular collection.

8.1.2 Transforming to XHTML or XSL-FO

A common XML transformation is converting documents from some proprietary XML structure to HTML. Since XQuery produces XML, it is fairly easy to write an XQuery program that returns XHTML, which is the XML version of HTML. XHTML is, for the most part, just well-formed HTML with lowercase tag and attribute names. So it is common to write XQuery programs that return XHTML.

Similarly, you can write an XQuery program that returns XSL-FO, which is a common path to build PDF output. Again, XSL-FO is just an XML structure, so it is easy to write XQuery that returns XML in that structure.

8.1.3 The typeswitch Expression

There are other ways to perform transformations in XQuery, but the `typeswitch` expression used in a recursive function is a design pattern that is convenient, performs well, and makes it very easy to change and maintain the transformation code.

For the syntax of the `typeswitch` expression, see The typeswitch Expression in *XQuery Reference Guide*. The `case` clause allows you to perform a test on the input to the `typeswitch` and then return something. For transformations, the tests are often what are called *kind tests*. A kind test tests to see what kind of node something is (for example, an element node with a given QName). If that test returns true, then the code in the `return` clause is executed. The `return` clause can be arbitrary XQuery, and can therefore call a function.

Because XML is an ordered tree structure, you can create a function that recursively walks through an XML node, each time doing some transformation on the node and sending its child nodes back into the function. The result is a convenient mechanism to transform the structure and/or content of an XML node.

8.2 Sample XQuery Transformation Code

This section provides some code examples that use the `typeswitch` expression. For each of these samples, you can cut and paste the code to execute against an App Server. For a more complicated example of this technique, see the Shakespeare Demo Application on developer.marklogic.com/code.

The following samples are included:

- Simple Example
- Simple Example With cts:highlight
- Sample Transformation to XHTML
- Extending the typeswitch Design Pattern

8.2.1 Simple Example

The following sample code does a trivial transformation of the input node, but it shows the basic design pattern where the `default` clause of the `typeswitch` expression calls a simple function which sends the child node back into the original function.

```
xquery version "1.0-ml";
(: This function takes the children of the node and passes them
   back into the typeswitch function.  :)
declare function local:passthru($x as node()) as node()*
{
for $z in $x/node() return local:dispatch($z)
};

(: This is the recursive typeswitch function :)
declare function local:dispatch($x as node()) as node()*
{
typeswitch ($x)
  case text() return $x
  case element (bar) return <barr>{local:passthru($x)}</barr>
  case element (baz) return <bazz>{local:passthru($x)}</bazz>
  case element (buzz) return <buzzz>{local:passthru($x)}</buzzz>
  case element (foo) return <fooo>{local:passthru($x)}</fooo>
  default return <temp>{local:passthru($x)}</temp>
};

let $x :=
<foo>foo
  <bar>bar</bar>
  <baz>baz
    <buzz>buzz</buzz>
  </baz>
  foo
</foo>
return
local:dispatch($x)
```

This XQuery program returns the following:

```
<fooo>
  foo
  <barr>bar</barr>
  <bazz>baz
    <buzzz>buzz</buzzz>
  </bazz>
  foo
</fooo>
```

8.2.2 Simple Example With cts:highlight

The following sample code is the same as the previous example, except it also runs `cts:highlight` on the result of the transformation. Using `cts:highlight` in this way is sometimes useful when displaying the results from a search and then highlighting the terms that match the `cts:query` expression. For details on `cts:highlight`, see Highlighting Search Term Matches in the *Search Developer's Guide*.

```
xquery version "1.0-ml";
(: This function takes the children of the node and passes them
   back into the typeswitch function.  :)
declare function local:passthru($x as node()) as node()*
{
for $z in $x/node() return local:dispatch($z)
};

(: This is the recursive typeswitch function :)
declare function local:dispatch($x as node()) as node()*
{
typeswitch ($x)
  case text() return $x
  case element (bar) return <barr>{local:passthru($x)}</barr>
  case element (baz) return <bazz>{local:passthru($x)}</bazz>
  case element (buzz) return <buzzz>{local:passthru($x)}</buzzz>
  case element (foo) return <fooo>{local:passthru($x)}</fooo>
  default return <booo>{local:passthru($x)}</booo>
};

let $x :=
<foo>foo
  <bar>bar</bar>
  <baz>baz
    <buzz>buzz</buzz>
  </baz>
  foo
</foo>
return
cts:highlight(local:dispatch($x), cts:word-query("foo"),
   <b>{$cts:text}</b>)
```

This XQuery program returns the following:

```
<fooo><b>foo</b>
  <barr>bar</barr>
  <bazz>baz
    <buzzz>buzz</buzzz>
  </bazz>
  <b>foo</b>
</fooo>
```

8.2.3 Sample Transformation to XHTML

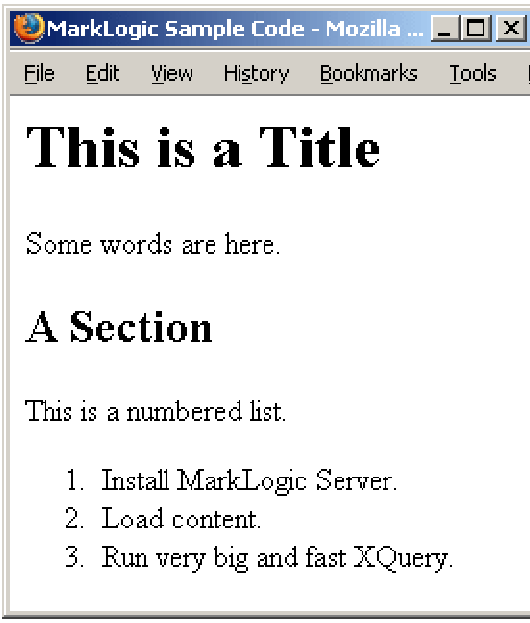
The following sample code performs a very simple transformation of an XML structure to XHTML. It uses the same design pattern as the previous example, but this time the XQuery code includes HTML markup.

```
xquery version "1.0-ml";
declare default element namespace "http://www.w3.org/1999/xhtml";

(: This function takes the children of the node and passes them
   back into the typeswitch function.  :)
declare function local:passthru($x as node()) as node()*
{
for $z in $x/node() return local:dispatch($z)
};

(: This is the recursive typeswitch function :)
declare function local:dispatch($x as node()) as node()*
{
typeswitch ($x)
  case text() return $x
  case element (a) return local:passthru($x)
  case element (title) return <h1>{local:passthru($x)}</h1>
  case element (para) return <p>{local:passthru($x)}</p>
  case element (sectionTitle) return <h2>{local:passthru($x)}</h2>
  case element (numbered) return <ol>{local:passthru($x)}</ol>
  case element (number) return <li>{local:passthru($x)}</li>
  default return <tempnode>{local:passthru($x)}</tempnode>
};

let $x :=
<a>
 <title>This is a Title</title>
 <para>Some words are here.</para>
 <sectionTitle>A Section</sectionTitle>
 <para>This is a numbered list.</para>
 <numbered>
   <number>Install MarkLogic Server.</number>
   <number>Load content.</number>
   <number>Run very big and fast XQuery.</number>
 </numbered>
</a>
return
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>MarkLogic Sample Code</title></head>
<body>{local:dispatch($x)}</body>
</html>
```

This returns the following XHTML code:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>MarkLogic Sample Code</title>
  </head>
<body>
<h1>This is a Title</h1>
<p>Some words are here.</p>
<h2>A Section</h2>
<p>This is a numbered list.</p>
  <ol>
      <li>Install MarkLogic Server.</li>
      <li>Load content.</li>
      <li>Run very big and fast XQuery.</li>
  </ol>
</body>
</html>
```

If you run this code against an HTTP App Server (for example, copy the code to a file in the App Server root and access the page from a browser), you will see results similar to the following:



Note that the `return` clauses of the `typeswitch case` statements in this example are simplified, and look like the following:

```
case element (sectionTitle) return <h2>{local:passthru($x)}</h2>
```

In a more typical example, the `return` clause would call a function:

```
case element (sectionTitle) return local:myFunction($x)
```

The function can then perform arbitrarily complex logic. Typically, each case statement calls a function with code appropriate to how that element needs to be transformed.

8.2.4 Extending the typeswitch Design Pattern

There are many ways you can extend this design pattern beyond the simple examples above. For example, you can add a second parameter to the simple `dispatch` functions shown in the previous examples. The second parameter passes some other information about the node you are transforming.

Suppose you want your transformation to exclude certain elements based on the place in the XML hierarchy in which the elements appear. You can then add logic to the function to exclude the passed in elements, as shown in the following code snippet:

```
declare function dispatch($x as node(), $excluded as element()*)
  as node()*
{
(: Test whether $x is an excluded element, if so return empty,
   otherwise run the typeswitch expression.
:)
if ( some $node in $excluded satisfies $x )
then ( )
else ( typeswitch ($x) ..... )
};
```

There are plenty of other extensions to this design pattern you can use. What you do depends on your application requirements. XQuery is a powerful programming language, and therefore these types of design patterns are very extensible to new requirements.

9.0 Document and Directory Locks

This chapter describes locks on documents and directories, and includes the following sections:

- Overview of Locks
- Lock APIs
- Example: Finding the URI of Documents With Locks
- Example: Setting a Lock on a Document
- Example: Releasing a Lock on a Document
- Example: Finding the User to Whom a Lock Belongs

9.1 Overview of Locks

Each document and directory can have a *lock.* A lock is stored as a locks document in a MarkLogic Server database. The locks document is separate from the document or directory to which it is associated. Locks have the following characteristics:

- Write Locks
- Persistent
- Searchable
- Exclusive or Shared
- Hierarchical
- Locks and WebDAV
- Other Uses for Locks

9.1.1 Write Locks

Locks are write locks; they restrict updates from all users who do not have the locks. When a user has an exclusive lock, no other users can get a lock and no other users can update or delete the document. Attempts to update or delete documents that have locks raise an error. Other users can still read documents that have locks, however.

9.1.2 Persistent

Locks are persistent in the database. They are not tied to a transaction. You can set locks to last a specified time period or to last indefinitely. Because they are persistent, you can use locks to ensure that a document is not modified during a multi-transaction operation.

9.1.3 Searchable

Because locks are persistent XML documents, they are therefore searchable XML documents, and you can write queries to give information about locks in the database. For an example, see "Example: Finding the URI of Documents With Locks" on page 92.

9.1.4 Exclusive or Shared

You can set locks as `exclusive`, which means only the user who set the lock can update the associated database object (document, directory, or collection). You can also set locks as `shared`, which means other users can obtain a shared lock on the database object; once a user has a `shared` lock on an object, the user can update it.

9.1.5 Hierarchical

When you are locking a directory, you can specify the depth in a directory hierarchy you want to lock. Specifying `"0"` means only the specified URI is locked, and specifying `"infinity"` means the URI (for example, the directory) and all of its children are locked.

9.1.6 Locks and WebDAV

WebDAV clients use locks to lock documents and directories before updating them. Locking ensures that no other clients will change the document while it is being saved. It is up to the implementation of a WebDAV client as to how it sets locks. Some clients set the locks to expire after a time period and some set them to last until they explicitly unlock the document.

9.1.7 Other Uses for Locks

Any application can use locks as part of its update strategy. For example, you can have a policy that a developer sets a lock for 30 seconds before performing an update to a document or directory. Locks are very flexible, so you can set up a policy that makes sense for your environment, or you can choose not to use them at all.

If you set a lock on every document and directory in the database, that can have the effect of not allowing any data to change in the database (except by the user who owns the lock). Combining a application development practice of locking and using security permissions effectively can provide a robust multi-user development environment.

9.2 Lock APIs

There are basically two kinds of APIs for locks: APIs to show locks and APIs to set/remove locks. For detailed syntax for these APIs, see the online XQuery Built-In and Module Function Reference.

The APIs to show locks are:

- `xdmp:document-locks`
- `xdmp:directory-locks`
- `xdmp:collection-locks`

The `xdmp:document-locks()` function with no arguments returns a sequence of locks, one for each document lock. The `xdmp:document-locks()` function with a sequence of URIs as an argument returns the locks for the specified document(s). The `xdmp:directory-locks` function returns locks for all of the documents in the specified directory, and the `xdmp:collection-locks` function returns all of the locks for documents in the specified collection.

You can set and remove locks on directories and documents with the following functions:

- `xdmp:lock-acquire`
- `xdmp:lock-release`

The basic procedure to set a lock on a document or a directory is to submit a query using the `xdmp:lock-acquire` function, specifying the URI, the scope of lock requested (`exclusive` or `shared`), the hierarchy affected by the lock (just the URI or the URI and all of its children), the owner of the lock, the duration of the lock

> **Note:** The `owner` of the lock is not the same as the `sec:user-id` of the lock. The `owner` can be specified as an option to `xdmp:lock-acquire`. If owner is not explicitly specified, then the owner defaults to the name of the user who issued the lock command. For an example, see "Example: Finding the User to Whom a Lock Belongs" on page 93.

9.3 Example: Finding the URI of Documents With Locks

If you call the XQuery built-in `xdmp:node-uri` function on a locks document, it returns the URI of the document that is locked. The following query returns a document listing the URIs of all documents in the database that have locks.

```
<root>
{
for $locks in xdmp:document-locks()
return <document-URI>{xdmp:node-uri($locks)}</document-URI>
}
</root>
```

For example, if the only document in the database with a lock has a URI `/document/myDocument.xml`, then the above query would return the following.

```
<root>
  <document-URI>/documents/myDocument.xml</document-URI>
</root>
```

9.4 Example: Setting a Lock on a Document

The following example uses the `xdmp:lock-acquire` function to set a two minute (120 second) lock on a document with the specified URI:

```
xdmp:lock-acquire("/documents/myDocument.xml",
                   "exclusive",
                   "0",
                   "Raymond is editing this document",
                   xs:unsignedLong(120))
```

You can view the resulting lock document with the `xdmp:document-locks` function as follows:

```
xdmp:document-locks("/documents/myDocument.xml")

=>

<lock:lock xmlns:lock="http://marklogic.com/xdmp/lock">
  <lock:lock-type>write</lock:lock-type>
  <lock:lock-scope>exclusive</lock:lock-scope>
  <lock:active-locks>
    <lock:active-lock>
      <lock:depth>0</lock:depth>
      <lock:owner>Raymond is editing this document</lock:owner>
      <lock:timeout>120</lock:timeout>
      <lock:lock-token>
       http://marklogic.com/xdmp/locks/4d0244560cc3726c
```

```
      </lock:lock-token>
      <lock:timestamp>1121722103</lock:timestamp>
      <sec:user-id xmlns:sec="http://marklogic.com/xdmp/security">
         8216129598321388485
      </sec:user-id>
    </lock:active-lock>
  </lock:active-locks>
</lock:lock>
```

9.5 Example: Releasing a Lock on a Document

The following example uses the `xdmp:lock-release` function to explicitly release a lock on a document:

```
xdmp:lock-release("/documents/myDocument.xml")
```

If you acquire a lock with no timeout period, be sure to release the lock when you are done with it. If you do not release the lock, no other users can update any documents or directories locked by the `xdmp:lock-acquire` action..

9.6 Example: Finding the User to Whom a Lock Belongs

Because locks are documents, you can write a query that finds the user to whom a lock belongs. For example, the following query searches through the `sec:user-id` elements of the lock documents and returns a set of URI names and user IDs of the user who owns each lock:

```
for $x in xdmp:document-locks()//sec:user-id
return  <lock>
           <URI>{xdmp:node-uri($x)}</URI>
           <user-id>{data($x)}</user-id>
        </lock>
```

A sample result is as follows (this result assumes there is only a single lock in the database):

```
<lock>
  <URI>/documents/myDocument.xml</URI>
  <user-id>15025067637711025979</user-id>
</lock>
```

10.0 Properties Documents and Directories

This chapter describes properties documents and directories in MarkLogic Server. It includes the following sections:

- Properties Documents
- Using Properties for Document Processing
- Directories
- Permissions On Properties and Directories
- Example: Directory and Document Browser

10.1 Properties Documents

A *properties document* is an XML document that shares the same URI with a document in a database. Every document can have a corresponding properties document, although the properties document is only created if properties are created. The properties document is typically used to store metadata related to its corresponding document, although you can store any XML data in a properties document, as long as it conforms to the properties document schema. A document must exist at a given URI in order to create a properties document, although it is possible to create a document and add properties to it in a single transaction. This section describes properties documents and the APIs for accessing them, and includes the following subsections:

- Properties Document Namespace and Schema
- APIs on Properties Documents
- XQuery property Axis
- Protected Properties
- Creating Element Indexes on a Properties Document Element
- Sample Properties Documents

10.1.1 Properties Document Namespace and Schema

Properties documents are XML documents that must conform to the `properties.xsd` schema. The `properties.xsd` schema is copied to the `<install_dir>/Config` directory at installation time.

The properties schema is assigned the `prop` namespace prefix, which is predefined in the server:

```
http://marklogic.com/xdmp/property
```

The following listing shows the `properties.xsd` schema:

```
<xs:schema targetNamespace="http://marklogic.com/xdmp/property"
 xsi:schemaLocation="http://www.w3.org/2001/XMLSchema XMLSchema.xsd
                     http://marklogic.com/xdmp/security security.xsd"
 xmlns="http://marklogic.com/xdmp/property"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xhtml="http://www.w3.org/1999/xhtml"
 xmlns:sec="http://marklogic.com/xdmp/security">

  <xs:complexType name="properties">
    <xs:annotation>
      <xs:documentation>
        A set of document properties.
      </xs:documentation>
      <xs:appinfo>
      </xs:appinfo>
    </xs:annotation>
    <xs:choice minOccurs="1" maxOccurs="unbounded">
      <xs:any/>
    </xs:choice>
  </xs:complexType>

  <xs:element name="properties" type="properties">
    <xs:annotation>
      <xs:documentation>
        The container for properties.
      </xs:documentation>
      <xs:appinfo>
      </xs:appinfo>
    </xs:annotation>
  </xs:element>

  <xs:simpleType name="directory">
    <xs:annotation>
      <xs:documentation>
        A directory indicator.
      </xs:documentation>
      <xs:appinfo>
      </xs:appinfo>
    </xs:annotation>
    <xs:restriction base="xs:anySimpleType">
    </xs:restriction>
  </xs:simpleType>

  <xs:element name="directory" type="directory">
    <xs:annotation>
      <xs:documentation>
        The indicator for a directory.
      </xs:documentation>
      <xs:appinfo>
      </xs:appinfo>
    </xs:annotation>
  </xs:element>
```

```
  <xs:element name="last-modified" type="last-modified">
    <xs:annotation>
      <xs:documentation>
        The timestamp of last document modification.
      </xs:documentation>
      <xs:appinfo>
      </xs:appinfo>
    </xs:annotation>
  </xs:element>

  <xs:simpleType name="last-modified">
    <xs:annotation>
      <xs:documentation>
        A timestamp of the last time something was modified.
      </xs:documentation>
      <xs:appinfo>
      </xs:appinfo>
    </xs:annotation>
    <xs:restriction base="xs:dateTime">
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

10.1.2 APIs on Properties Documents

The APIs for properties documents are XQuery functions which allow you to list, add, and set properties in a properties document. The properties APIs provide access to the top-level elements in properties documents. Because the properties are XML elements, you can use XPath to navigate to any children or descendants of the top-level property elements. The properties document is tied to its corresponding document and shares its URI; when you delete a document, its properties document is also deleted.

The following APIs are available to access and manipulate properties documents:

- `xdmp:document-properties`
- `xdmp:document-add-properties`
- `xdmp:document-set-properties`
- `xdmp:document-set-property`
- `xdmp:document-remove-properties`
- `xdmp:document-get-properties`
- `xdmp:collection-properties`
- `xdmp:directory`
- `xdmp:directory-properties`

For the signatures and descriptions of these APIs, see the *MarkLogic Built-In and Module Functions Reference*.

10.1.3 XQuery property Axis

MarkLogic has extended the XQuery language to include the *property axis*. The property axis (`property::`) allows you to write an XPath expression to search through items in the properties document for a given URI. These expression allow you to perform joins across the document and property axes, which is useful when storing state information for a document in a property. For details on this approach, see "Using Properties for Document Processing" on page 98.

The property axis is similar to the forward and reverse axes in an XPath expression. For example, you can use the `child::` forward axis to traverse to a child element in a document. For details on the XPath axes, see the [XPath 2.0 specification].

The property axis contains all of the children of the properties document node for a given URI.

The following example shows how you can use the property axis to access properties for a document while querying the document:

Create a test document as follows:

```
xdmp:document-insert("/test/123.xml",
  <test>
    <element>123</element>
  </test>)
```

Add a property to the properties document for the `/test/123.xml` document:

```
xdmp:document-add-properties("/test/123.xml",
  <hello>hello there</hello>)
```

If you list the properties for the `/test/123.xml` document, you will see the property you just added:

```
xdmp:document-properties("/test/123.xml")
=>
<prop:properties xmlns:prop="http://marklogic.com/xdmp/property">
  <hello>hello there</hello>
</prop:properties>
```

You can now search through the proerty axis of the `/test/123.xml` document, as follows:

```
doc("test/123.xml")/property::hello
=>
<hello>hello there</hello>
```

10.1.4 Protected Properties

The following properties are protected, and they can only be created or modified by the system:

- `prop:directory`
- `prop:last-modified`

These properties are reserved for use directly by MarkLogic Server; attempts to add or delete properties with these names fail with an exception.

10.1.5 Creating Element Indexes on a Properties Document Element

Because properties documents are XML documents, you can create element (range) indexes on elements within a properties document. If you use properties to store numeric or date metadata about the document to which the properties document corresponds, for example, you can create an element index to speed up queries that access the metadata.

10.1.6 Sample Properties Documents

Properties documents are XML documents that conform to the schema described in "Properties Document Namespace and Schema" on page 94. You can list the contents of a properties document with the `xdmp:document-properties("<uri>")` function. If there is no properties document at the specified URI, the function returns the empty sequence. A properties document for a directory has a single empty `prop:directory` element. For example, if there exists a directory at the URI `http://myDirectory/`, the `xdmp:document-properties` command returns a properties document as follows:

```
xdmp:document-properties("http://myDirectory/")
=>
<prop:properties xmlns:prop="http://marklogic.com/xdmp/property">
  <prop:directory/>
</prop:properties>
```

You can add whatever you want to a properties document (as long as it conforms to the properties schema). If you run the function `xdmp:document-properties()` with no arguments, it returns a sequence of all the properties documents in the database.

10.2 Using Properties for Document Processing

When you need to update large numbers of documents, sometimes in multi-step processes, you often need to keep track of the current state of each document. For example, if you have a content processing application that updates millions of documents in three steps, you need to have a way of programatically determining which documents have not been processed at all, which have completed step 1, which have completed step 2, and so on.

This section describes how to use properties to store metadata for use in a document processing pipeline, it includes the following subsections:

- Using the property Axis to Determine Document State
- Document Processing Problem
- Solution for Document Processing
- Basic Commands for Running Modules

10.2.1 Using the property Axis to Determine Document State

You can use properties documents to store state information about documents that undergo multi-step processing. Joining across properties documents can then determine which documents have been processed and which have not. The queries that perform these joins use the `property::` axis (for details, see "XQuery property Axis" on page 97).

Joins across the properties axis that have predicates are optimized for performance. For example, the following returns `foo` root elements from documents that have a property `bar`:

```
foo[property::bar]
```

The following examples show the types of queries that are optimized for performance (where `/a/b/c` is some XPath expression):

- Property axis in predicates:

```
/a/b/c[property::bar]
```

- Negation tests on property axis:

```
/a/b/c[not(property::bar = "baz")]
```

- Continuing path expression after the `property` predicate:

```
/a/b/c[property::bar and bob = 5]/d/e
```

- Equivalent FLWOR expressions:

```
for $f in /a/b/c
where $f/property::bar = "baz"
return $f
```

Other types of expressions will work but are not optimized for performance, including the following:

- If you want the bar property of documents whose root elements are `foo`:

```
/foo/property::bar
```

10.2.2 Document Processing Problem

The approach outlined in this section works well for situations such as the following:

- "I have already loaded 1 million documents and now want to update all of them." The psuedo-code for this is as follows:

  ```
  for $d in fn:doc()
  return some-update($d)
  ```

  These types of queries will eventually run out of tree cache memory and fail.

- When iterative calls of the following form become progressively slow:

  ```
  for $d in fn:doc()[k to k+10000]
  return some-update($d)
  ```

For these types of scenarios, using properties to test whether a document needs processing is an effective way of being able to batch up the updates into manageable chunks.

10.2.3 Solution for Document Processing

This content processing technique works in a wide variety of situations This approach satisfies the following requirements:

- Works with large existing datasets.
- Does not require you to know before you load the datasets that you are going to need to further processing to them later.
- This approach works in a situations in which data is still arriving (for example, new data is added every day).
- Needs to be able to ultimately transition into a steady state "content processing" enabled environment.

The following are the basic steps of the document processing approach:

1. Take an iterative strategy, but one that does not become progressively slow.
2. Split the reprocessing activity into multiple updates.
3. Use properties (or lack thereof) to identify the documents that (still) need processing.
4. Repeatedly call the same module, updating its property as well as updating the document:

   ```
   for $p in fn:doc()/root[not(property::some-update)][1 to 10000]
   return some-update($d)
   ```

5. If there are any documents that still need processing, invoke the module again.

6. The psuedo-code for the module that processes documents that do not have a specific property is as follows:

```
let $docs := get n documents that have no properties
return
for $processDoc in $docs
return if (empty $processDoc)
       then ()
       else ( process-document($processDoc),
              update-property($processDoc) )
,
xdmp:spawn(process_module)
```

   This psuedo-code does the following:

   - gets the URIs of documents that do not have a specific property
   - for each URI, check if the specific property exists
   - if the property exists, do nothing to that document (it has already been updated)
   - if the property does not exist, do the update to the document and the update to the property
   - continue this for all of the URIs
   - when all of the URIs have been processed, call the module again to get any new documents (ones with no properties)

7. (Optional) Automate the process by setting up a Content Processing Pipeline.

10.2.4 Basic Commands for Running Modules

The following built-in functions are needed to perform automated content processing:

- To put a module on Task Server Queue:

```
xdmp:spawn($database, $root, $path)
```

- To evaluate an entire module (similar to `xdmp:eval()`, but for for modules):

```
xdmp:invoke($path, $external-vars)

xdmp:invoke-in($path, $database-id, $external-vars)
```

10.3 Directories

Directories have many uses, including organizing your document URIs and using them with WebDAV servers. This section includes the following items about directories:

- Properties and Directories
- Directories and WebDAV Servers
- Directories Versus Collections

10.3.1 Properties and Directories

When you create a directory, MarkLogic Server creates a properties document with a `prop:directory` element. If you run the `xdmp:document-properties` command on the URI corresponding to a directory, the command returns a properties document with an empty `prop:directory` element, as shown in the following example:

```
xdmp:directory-create("/myDirectory/");

xdmp:document-properties("/myDirectory/")
=>
<prop:properties xmlns:prop="http://marklogic.com/xdmp/property">
  <prop:directory/>
</prop:properties>
```

Note: You can create a directory with any unique URI, but the convention is for directory URIs to end with a forward slash (`/`). It is possible to create a document with the same URI as a directory, but this is not recommended; the best practice is to reserve URIs ending in slashes for directories.

Because `xdmp:document-properties()` with no arguments returns the properties documents for all properties documents in the database, and because each directory has a `prop:directory` element, you can easily write a query that returns all of the directories in the database. Use the `xdmp:node-uri` function to accomplish this as follows:

```
xquery version "1.0-ml";

for $x in xdmp:document-properties()/prop:properties/prop:directory
return <directory-uri>{xdmp:node-uri($x)}</directory-uri>
```

10.3.2 Directories and WebDAV Servers

Directories are needed for use in WebDAV servers. In order to create a document that will be accessed from a WebDAV client, the parent directory must exist. The parent directory of a document is the directory in which the URI is the prefix of the document (for example, the directory of the URI `http://myserver/doc.xml` is `http://myserver/`). When using a database with

a WebDAV server, ensure tht the `directory creation` setting on the database configuration is set to `automatic` (this is the default setting), which causes parent directories to be created when documents are created. For information on using directories in WebDAV servers, see “WebDAV Servers” in the *Administrator’s Guide*.

10.3.3 Directories Versus Collections

You can use both directories and collections to organize documents in a database. The following are important differences between directories and collections:

- Directories are hierarchical in structure (like a filesystem directory structure). Collections do not have this requirement. Because directories are hierarchical, a directory URI must contain any parent directories. Collection URIs do not need to have any relation to documents that belong to a collection. For example, a directory named `http://marklogic.com/a/b/c/d/e/` (where `http://marklogic.com/` is the root) requires the existence of the parent directories `d`, `c`, `b`, and `a`. With collections, any document (regardless of its URI) can belong to a collection with the given URI.
- Directories are required for WebDAV clients to see documents. In other words, in order to see a document with URI `/a/b/hello/goodbye` in a WebDAV server with `/a/b/` as the root, directories with the following URIs must exist in the database:

  ```
  /a/b/
  
  /a/b/hello/
  ```

Except for the fact that you can use both directories and collections to organize documents, directories are unrelated to collections. For details on collections, see Collections in the *Search Developer’s Guide*. For details on WebDAV servers, see “WebDAV Servers” in the *Administrator’s Guide*.

10.4 Permissions On Properties and Directories

Like any document in a MarkLogic Server database, a properties document can have permissions. Since a directory has a properties document (with an empty `prop:directory` element), directories can also have permissions. Permissions on properties documents are the same as the permissions on their corresponding documents, and you can list the permissions with the `xdmp:document-get-permissions("<document-uri>")` function. Similarly, you can list the permissions on a directory with the `xdmp:document-get-permissions("<directory-uri>")` function. For details on permissions and on security, see *Understanding and Using Security*.

10.5 Example: Directory and Document Browser

Using properties documents, you can build a simple application that lists the documents and directories under a URI. The following sample code uses the `xdmp:directory` function to list the children of a directory (which correspond to the URIs of the documents in the directory), and the `xdmp:directory-properties` function to find the `prop:directory` element, indicating that a URI is a directory. This example has two parts:

- Directory Browser Code
- Setting Up the Directory Browser

10.5.1 Directory Browser Code

The following is sample code for a very simple directory browser.

```
xquery version "1.0-ml";
(:   directory browser
            Place in Modules database and give execute permission :)

declare namespace prop="http://marklogic.com/xdmp/property";

(: Set the root directory of your AppServer for the
   value of $rootdir :)
let $rootdir := (xdmp:modules-root())
(: take all but the last part of the request path, after the
   initial slash :)
let $dirpath := fn:substring-after(fn:string-join(fn:tokenize(
                xdmp:get-request-path(), "/")[1 to last() - 1],
                "/"), "/")
let $basedir := if ( $dirpath eq "" )
                then ( $rootdir )
                else fn:concat($rootdir, $dirpath, "/")
let $uri := xdmp:get-request-field("uri", $basedir)
return if (ends-with($uri, "/")) then
<html xmlns="http://www.w3.org/1999/xhtml">
   <head>
     <title>MarkLogic Server Directory Browser</title>
   </head>
   <body>
     <h1>Contents of {$uri}</h1>
<h3>Documents</h3>
{
   for $d in xdmp:directory($uri, "1")
   let $u := xdmp:node-uri($d)
   (: get the last two, and take the last non-empty string :)
   let $basename :=
     tokenize($u, "/")[last(), last() - 1][not(. = "")][last()]
   order by $basename
   return element p {
     element a {
```

```
(:  The following should work for all $basedir values, as long
    as the string represented by $basedir is unique in the
    document URI :)
       attribute href { substring-after($u,$basedir) },
       $basename
     }
   }
}
       <h3>Directories</h3>
{
   for $d in xdmp:directory-properties($uri, "1")//prop:directory
   let $u := xdmp:node-uri($d)
   (: get the last two, and take the last non-empty string :)
   let $basename :=
     tokenize($u, "/")[last(), last() - 1][not(. = "")][last()]
   order by $basename
   return element p {
     element a {
       attribute href { concat(
                              xdmp:get-request-path(),
                              "?uri=",
                              $u) },
       concat($basename, "/")
     }
   }
}
</body>
</html>
else doc($uri)

(: browser.xqy :)
```

This application writes out an HTML document with links to the documents and directories in the root of the server. The application finds the documents in the root directory using the `xdmp:directory` function, finds the directories using the `xdmp:directory-properties` function, does some string manipulation to get the last part of the URI to display, and keeps the state using the application server `request` object built-in XQuery functions (`xdmp:get-request-field` and `xdmp:get-request-path`).

10.5.2 Setting Up the Directory Browser

To run this directory browser application, perform the following:

1. Create an HTTP Server and configure it as follows:

   a. Set the Modules database to be the same database as the Documents database. For example, if the `database` setting is set to the database named `my-database`, set the `modules` database to `my-database` as well.

b. Set the HTTP Server root to `http://myDirectory/`, or set the root to another value and modify the `$rootdir` variable in the directory browser code so it matches your HTTP Server root.

c. Set the port to 9001, or to a port number not currently in use.

2. Copy the sample code into a file named `browser.xqy`. If needed, modify the `$rootdir` variable to match your HTTP Server root. Using the `xdmp:modules-root` function, as in the sample code, will automatically get the value of the App Server root.

3. Load the `browser.xqy` file into the Modules database at the top level of the HTTP Server root. For example, if the HTTP Server root is `http://myDirectory/`, load the `browser.xqy` file into the database with the URI `http://myDirectory/browser.xqy`. You can load the document either via a WebDAV client (if you also have a WebDAV server pointed to this root) or with the `xdmp:document-load` function.

4. Make sure the `browser.xqy` document has execute permissions. You can check the permissions with the following function:

```
xdmp:document-get-permissions("http://myDirectory/browser.xqy")
```

This command returns all of the permissions on the document. It should have “execute” capability for a role possessed by the user running the application. If it does not, you can add the permissions with a command similar to the following:

```
xdmp:document-add-permissions("http://myDirectory/browser.xqy",
                              xdmp:permission("myRole", "execute"))
```

where `myRole` is a role possessed by the user running the application.

5. Load some other documents into the HTTP Server root. For example, drag and drop some documents and folders into a WebDAV client (if you also have a WebDAV server pointed to this root).

6. Access the `browser.xqy` file with a web browser using the host and port number from the HTTP Server. For example, if you are running on your local machine and you have set the HTTP Server port to 9001, you can run this application from the URL `http://localhost:9001/browser.xqy`.

You should see links to the documents and directories you loaded into the database. If you did not load any other documents, you will just see a link to the `browser.xqy` file.

11.0 Point-In-Time Queries

You can configure MarkLogic Server to retain old versions of documents, allowing you to evaluate a query statement as if you had travelled back to a point-in-time in the past. When you specify a timestamp at which a query statement should evaluate, that statement will evaluate against the newest version of the database up to (but not beyond) the specified timestamp.

This chapter describes point-in-time queries and includes the following sections:

- Understanding Point-In-Time Queries
- Using Timestamps in Queries
- Specifying Point-In-Time Queries in xdmp:eval, xdmp:invoke, xdmp:spawn, and XCC
- Keeping Track of System Timestamps

11.1 Understanding Point-In-Time Queries

To best understand point-in-time queries, you need to understand a little about how different versions of fragments are stored and merged out of MarkLogic Server. This section describes some details of how fragments are stored and how that enables point-in-time queries, as well as lists some other details important to understanding what you can and cannot do with point-in-time queries:

- Fragments Stored in Log-Structured Database
- System Timestamps and Merge Timestamps
- How the Fragments for Point-In-Time Queries are Stored
- Only Available on Query Statements, Not on Update Statements
- All Auxiliary Databases Use Latest Version
- Database Configuration Changes Do Not Apply to Point-In-Time Fragments

For more information on how merges work, see the “Understanding and Controlling Database Merges” chapter of the *Administrator’s Guide*. For background material for this chapter, see “Understanding Transactions in MarkLogic Server” on page 12.

11.1.1 Fragments Stored in Log-Structured Database

A MarkLogic Server database consists of one or more forests. Each forest is made up of one or more stands. Each stand contains one or more fragments. The number of fragments are determined by several factors, including the number of documents and the fragment roots defined in the database configuration.

To maximize efficiency and improve performance, the fragments are maintained using a method analagous to a *log-structured filesystem*. A log-structured filesystem is a very efficient way of adding, deleting, and modifying files, with a garbage collection process that periodically removes obsolete versions of the files. In MarkLogic Server, fragments are stored in a log-structured database. MarkLogic Server periodically merges two or more stands together to form a single stand. This merge process is equivalent to the garbage collection of log-structured filesystems.

When you modify or delete an existing document or node, it affects one or more fragments. In the case of modifying a document (for example, an `xdmp:node-replace` operation), MarkLogic Server creates new versions of the fragments involved in the operation. The old versions of the fragments are marked as obsolete, but they are not yet deleted. Similarly, if a fragment is deleted, it is simply marked as obsolete, but it is not immediately deleted from disk (although you will no longer be able to query it without a point-in-time query).

11.1.2 System Timestamps and Merge Timestamps

When a merge occurs, it recovers disk space occupied by obsolete fragments. The system maintains a *system timestamp*, which is a number that increases everytime anything maintained by MarkLogic Server is changed. In the default case, the new stand is marked with the current timestamp at the time in which the merge completes (the *merge timestamp*). Any fragments that became obsolete prior to the merge timestamp (that is, any old versions of fragments or deleted fragments) are eliminated during the merge operation.

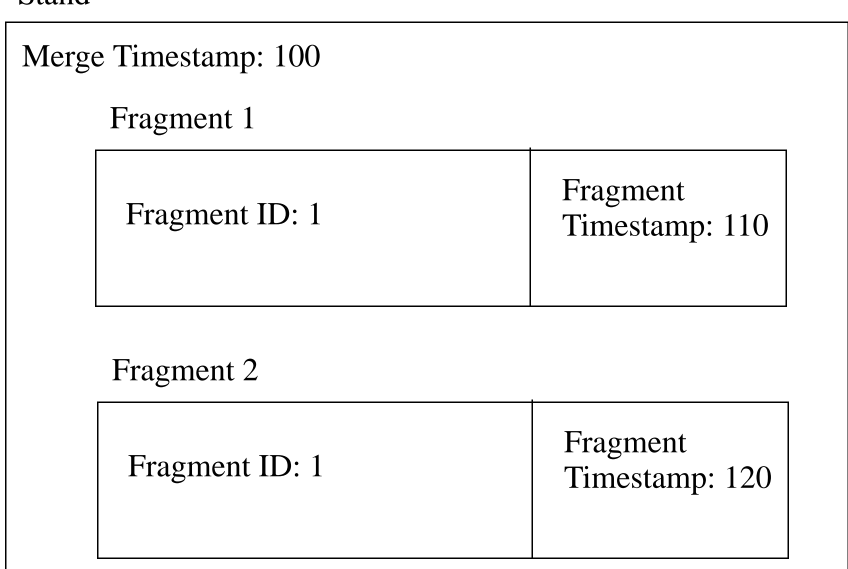
There is a control at the database level called the `merge timestamp`, set via the Admin Interface. By default, the `merge timestamp` is set to 0, which sets the timestamp of a merge to the timestamp corresponding to when the merge completes. To use point-in-time queries, you can set the `merge timestamp` to a static value corresponding to a particular time. Then, any merges that occur after that time will preserve all fragments, including obsolete fragments, whose timestamps are equal to or later than the specified `merge timestamp`.

The effect of preserving obsolete fragments is that you can perform queries that look at an older view of the database, as if you are querying the database from a point-in-time in the past. For details on setting the merge timestamp, see “Enabling Point-In-Time Queries in the Admin Interface” on page 110.

11.1.3 How the Fragments for Point-In-Time Queries are Stored

Just like any fragments, fragments with an older timestamp are stored in stands, which in turn are stored in forests. The only difference is that they have an older timestamp associated with them. Different versions of fragments can be stored in different stands or in the same stand, depending on if they have been merged into the same stand.

The following figure shows a stand with a merge timestamp of 100. Fragment 1 is a version that was changed at timestamp 110, and fragment 2 is a version of the same fragment that was changed at timestamp 120.



In this scenario, if you assume that the current time is timestamp 200, then a query at the current time will see Fragment 2, but not Fragment 1. If you perform a point-in-time query at timestamp 115, you will see Fragment 1, but not Fragment 2 (because Fragment 2 did not yet exist at timestamp 115).

There is no limit to the number of different versions that you can keep around. If the `merge timestamp` is set to the current time or a time in the past, then all subsequently modified fragments will remain in the database, available for point-in-time queries.

11.1.4 Only Available on Query Statements, Not on Update Statements

You can only specify a point-in-time query statement; attempts to specify a point-in-time query for an update statement will throw an exception. An update statement is any XQuery issued against MarkLogic Server that includes an update function (`xdmp:document-load`, `xdmp:node-replace`, and so on). For more information on what constitutes query statements and update statements, see "Understanding Transactions in MarkLogic Server" on page 12.

11.1.5 All Auxiliary Databases Use Latest Version

The auxiliary databases associated with a database request (that is, the Security, Schemas, Modules, and Triggers databases) all operate at the latest timestamp, even during a point-in-time query. Therefore, any changes made to security objects, schemas, and so on since the time specified in the point-in-time query are *not* reflected in the query.

11.1.6 Database Configuration Changes Do Not Apply to Point-In-Time Fragments

If you make configuration changes to a database (for example, changing database index settings), those changes only apply to the latest versions of fragments. For example, if you make index option changes and reindex a database that has old versions of fragments retained, only the latest versions of the fragments are reindexed. The older versions of fragments, used for point-in-time queries, retain the indexing properties they had at the timestamp in which they became invalid (that is, from the timestamp when an update or delete occured on the fragments). MarkLogic recommends that you do not change database settings and reindex a database that has the `merge timestamp` database parameter set to anything but 0.

11.2 Using Timestamps in Queries

By default, query statements are run at the system timestamp in effect when the statement initiates. To run a query statement at a different system timestamp, you must set up your system to store older versions of documents and then specify the timestamp when you issue a point-in-time query statement. This section describes this general process and includes the following parts:

- Enabling Point-In-Time Queries in the Admin Interface
- The xdmp:request-timestamp Function
- Requires the xdmp:timestamp Execute Privilege
- The Timestamp Parameter to xdmp:eval, xdmp:invoke, xdmp:spawn
- Timestamps on Requests in XCC
- Scoring Considerations

11.2.1 Enabling Point-In-Time Queries in the Admin Interface

In order to use point-in-time queries in a database, you must set up merges to preserve old versions of fragments. By default, old versions of fragments are deleted from the database after a merge. For more information on how merges work, see the “Understanding and Controlling Database Merges” chapter of the *Administrator’s Guide*.

In the Merge Policy Configuration page of the Admin Interface, there is a `merge timestamp` parameter. When this parameter is set to 0 (the default) and merges are enabled, point-in-time queries are effectively disabled. To access the Merge Policy Configuration page, click the Databases > *db_name* > Merge Policy link from the tree menu of the Admin Interface.

When deciding the value at which to set the `merge timestamp` parameter, the most likely value to set it to is the current system timestamp. Setting the value to the current system timestamp will preserve any versions of fragments from the current time going forward. To set the `merge timestamp` parameter to the current timestamp, click the `get current timestamp` button on the Merge Control Configuration page and then Click OK.

If you set a value for the `merge timestamp` parameter higher than the current timestamp, MarkLogic Server will use the current timestamp when it merges (the same behavior as when set to the default of 0). When the system timestamp grows past the specified `merge timestamp` number, it will then start using the `merge timestamp` specified. Similarly, if you set a `merge timestamp` lower than the lowest timestamp preserved in a database, MarkLogic Server will use the lowest timestamp of any preserved fragments in the database, or the current timestamp, whichever is lower.

You might want to keep track of your system timestamps over time, so that when you go to run point-in-time queries, you can map actual time with system timestamps. For an example of how to create such a timestamp record, see "Keeping Track of System Timestamps" on page 115.

Note: After the system merges when the `merge timestamp` is set to 0, all obsolete versions of fragments will be deleted; that is, only the latest versions of fragments will remain in the database. If you set the `merge timestamp` to a value lower than the current timestamp, any obsolete versions of fragments will not be available (because they no longer exist in the database). Therefore, if you want to preserve versions of fragments, you must configure the system to do so before you update the content.

11.2.2 The xdmp:request-timestamp Function

MarkLogic Server has an XQuery built-in function, `xdmp:request-timestamp`, which returns the system timestamp for the current request. MarkLogic Server uses the system timestamp values to keep track of versions of fragments, and you use the system timestamp in the `merge timestamp` parameter (described in "Enabling Point-In-Time Queries in the Admin Interface" on page 110) to specify which versions of fragments remain in the database after a merge. For more details on the `xdmp:request-timestamp` function, see the *MarkLogic Built-In and Module Functions Reference*.

11.2.3 Requires the xdmp:timestamp Execute Privilege

In order to run a query at a timestamp other than the current timestamp, the user who runs the query must belong to a group that has the `xdmp:timestamp` execute privilege. For details on security and execute privileges, see *Understanding and Using Security*.

11.2.4 The Timestamp Parameter to xdmp:eval, xdmp:invoke, xdmp:spawn

The `xdmp:eval`, `xdmp:invoke`, and `xdmp:spawn` functions all take an `options` node as the optional third parameter. The options node must be in the `xdmp:eval` namespace. The options node has a `timestamp` element which allows you to specify a system timestamp at which the query should run. When you specify a `timestamp` value earlier than the current timestamp, you are specifying a point-in-time query.

The timestamp you specify must be valid for the database. If you specify a system timestamp that is less than the oldest timestamp preserved in the database, the statement will throw an `XDMP-OLDSTAMP` exception. If you specify a timestamp that is newer than the current timestamp, the statement will throw an `XDMP-NEWSTAMP` exception.

> **Note:** If the merge timestamp is set to the default of 0, and if the database has completed all merges since the last updates or deletes, query statements that specify any timestamp older than the current system timestamp will throw the `XDMP-OLDSTAMP` exception. This is because the merge timestamp value of 0 specifies that no obsolete fragments are to be retained.

The following example shows an `xdmp:eval` statement with a `timestamp` parameter:

```
xdmp:eval("doc('/docs/mydocument.xml')", (),
  <options xmlns="xdmp:eval">
    <timestamp>99225</timestamp>
  </options>)
```

This statement will return the version of the `/docs/mydocument.xml` document that existed at system timestamp 99225.

11.2.5 Timestamps on Requests in XCC

The `xdmp:eval`, `xdmp:invoke`, and `xdmp:spawn` functions allow you to specify timestamps for a query statement at the XQuery level. If you are using the XML Content Connector (XCC) libraries to communicate with MarkLogic Server, you can also specify timestamps at the Java or .NET level.

In XCC for Java, you can set options to requests with the `RequestOptions` class, which allows you to modify the environment in which a request runs. The `setEffectivePointInTime` method sets the timestamp in which the request runs. The core design pattern is to set up options for your requests and then use those options when the requests are submitted to MarkLogic Server for evaluation. You can also set request options on the `Session` object. The following Java code snippet shows the basic design pattern:

```
// create a class and methods that use code similar to
// the following to set the system timestamp for requests

Session session = getSession();
BigInteger timestamp = session.getCurrentServerPointInTime();
RequestOptions options = new RequestOptions();

options.setEffectivePointInTime (timestamp);
session.setDefaultRequestOptions (options);
```

For an example of how you might use a Java environment to run point-in-time queries, see "Example: Query Old Versions of Documents Using XCC" on page 114.

11.2.6 Scoring Considerations

When you store multiple versions of fragments in a database, it will subtly effect the scores returned with `cts:search` results. The scores are calculated using document frequency as a variable in the scoring formula (for the default `score-logtfidf` scoring method). The amount of effect preserving older versions of fragments has depends on two factors:

- How many fragments have multiple versions.
- How many total fragments are in the database.

If the number of fragments with multiple versions is small compared with the total number of fragments in the database, then the effect will be relatively small. If that ratio is large, then the effect on scores will be higher.

For more details on scores and the scoring methods, see Relevance Scores: Understanding and Customizing in the *Search Developer's Guide*.

11.3 Specifying Point-In-Time Queries in xdmp:eval, xdmp:invoke, xdmp:spawn, and XCC

As desribed earlier, specifying a valid `timestamp` element in the `options` node of the `xdmp:eval`, `xdmp:invoke`, or `xdmp:spawn` functions initiates a point-in-time query. Also, you can use XCC to specify entire XCC requests as point-in-time queries. The query runs at the specified timestamp, seeing a version of the database that existed at the point in time corresponding to the specified timestamp. This section shows some example scenarios for point-in-time queries, and includes the following parts:

- Example: Query Old Versions of Documents Using XCC
- Example: Querying Deleted Documents

11.3.1 Example: Query Old Versions of Documents Using XCC

When making updates to content in your system, you might want to add and test new versions of the content before exposing the new content to your users. During this testing time, the users will still see the old version of the content. Then, when the new content has been sufficiently tested, you can switch the users over to the new content.

Point-in-time queries allow you to do this all within the same database. The only thing that you need to change in the application is the timestamps at which the query statements run. XCC provides a convenient mechanism for accomplishing this goal.

11.3.2 Example: Querying Deleted Documents

When you delete a document, the fragments for that document are marked as obsolete. The fragments are not actually deleted from disk until a merge completes. Also, if the `merge timestamp` is set to a timestamp earlier than the timestamp corresponding to when the document was deleted, the merge will preserve the obsolete fragments.

This example demonstrates how you can query deleted documents with point-in-time queries. For simplicity, assume that no other query or update activity is happening on the system for the duration of the example. To follow along in the example, run the following code samples in the order shown below.

1. First, create a document:

```
xdmp:document-insert("/docs/test.xml", <a>hello</a>))
```

2. When you query the document, it returns the node you inserted:

```
doc("/docs/test.xml")
(: returns the node <a>hello</a> :)
```

3. Delete the document:

```
xdmp:document-delete("/docs/test.xml")
```

4. Query the document again. It returns the empty sequence because it was just deleted.

5. Run a point-in-time query, specifying the current timestamp (this is semantically the same as querying the document without specifying a timestamp):

```
xdmp:eval("doc('/docs/test.xml')", (),
<options xmlns="xdmp:eval">
  <timestamp>{xdmp:request-timestamp()}</timestamp>
</options>)
(: returns the empty sequence because the document has been deleted :)
```

6. Run the point-in-time query at one less than the current timestamp, which is the old timestamp in this case because only one change has happened to the database. The following query statement returns the old document.

```
xdmp:eval("doc('/docs/test.xml')", (),
<options xmlns="xdmp:eval">
  <timestamp>{xdmp:request-timestamp()-1}</timestamp>
</options>)
(: returns the deleted version of the document :)
```

11.4 Keeping Track of System Timestamps

The system timestamp does not record the actual time in which updates occur; it is simply a number that is incremented by 1 each time an update or configuration change occurs in the system. If you want to map system timestamps with actual time, you need to store that information somewhere. This section shows a design pattern, including some sample code, of the basic principals for creating an application that archives the system timestamp at actual time intervals.

> **Note:** It might not be important to your application to map system timestamps to actual time. For example, you might simply set up your merge timestamp to the current timestamp, and know that all versions from then on will be preserved. If you do not need to keep track of the system timestamp, you do not need to create this application.

The first step is to create a document in which the timestamps are stored, with an initial entry of the current timestamp. To avoid possible confusion of future point-in-time queries, create this document in a different database than the one in which you are running point-in-time queries. You can create the document as follows:

```
xdmp:document-insert("/system/history.xml",
<timestamp-history>
  <entry>
    <datetime>{fn:current-dateTime()}</datetime>
    <system-timestamp>{
```

```
(: use eval because this is an update statement :)
      xdmp:eval("xdmp:request-timestamp()")}
    </system-timestamp>
  </entry>
</timestamp-history>)
```

This results in a document similar to the following:

```
<timestamp-history>
  <entry>
    <datetime>2006-04-26T19:35:51.325-07:00</datetime>
    <system-timestamp>92883</system-timestamp>
  </entry>
</timestamp-history>
```

Note that the code uses `xdmp:eval` to get the current timestamp. It must use `xdmp:eval` because the statement is an update statement, and update statements always return the empty sequence for calls to `xdmp:request-timestamp`. For details, see "Understanding Transactions in MarkLogic Server" on page 12.

Next, set up a process to run code similar to the following at periodic intervals. For example, you might run the following every 15 minutes:

```
xdmp:node-insert-child(doc("/system/history.xml")/timestamp-history,
<entry>
  <datetime>{fn:current-dateTime()}</datetime>
  <system-timestamp>{
(: use eval because this is an update statement :)
   xdmp:eval("xdmp:request-timestamp()")}
  </system-timestamp>
</entry>)
```

This results in a document similar to the following:

```
<timestamp-history>
  <entry>
    <datetime>2006-04-26T19:35:51.325-07:00</datetime>
    <system-timestamp>92883</system-timestamp>
  </entry>
  <entry>
    <datetime>2006-04-26T19:46:13.225-07:00</datetime>
    <system-timestamp>92884</system-timestamp>
  </entry>
</timestamp-history>
```

To call this code at periodic intervals, you can set up a cron job, write a shell script, write a Java or dotnet program, or use any method that works in your environment. Once you have the document with the timestamp history, you can easily query it to find out what the system timestamp was at a given time.

12.0 Using the map Functions to Create Name-Value Maps

This chapter describes how to use the map functions and includes the following sections:

- Maps: In-Memory Structures to Manipulate in XQuery
- map:map XQuery Primitive Type
- Serializing a Map to an XML Node
- Map API
- Examples

12.1 Maps: In-Memory Structures to Manipulate in XQuery

Maps are in-memory structures containing name-value pairs that you can create and manipulate. In some programming languages, maps are implemented using hash tables. Maps are handy programming tools, as you can conveniently store and update name-value pairs for use later in your program. Maps provide a fast and convenient method for accessing data.

MarkLogic Server has a set of XQuery functions to create manipulate maps. Like the `xdmp:set` function, maps have side-effects and can change within your program. Therefore maps are not strictly functional like most other aspects of XQuery. While the map is in memory, its structure is opaque to the developer, and you access it with the built-in XQuery functions. You can persist the structure of the map as an XML node, however, if you want to save it for later use. A map is a node and therefore has an identity, and the identity remains the same as long as the map is in memory. However, if you serialize the map as XML and store it in a document, when you retrieve it will have a different node identity (that is, comparing the identity of the map and the serialized version of the map would return false). Similarly, if you store XML values retrieved from the database in a map, the node in the in-memory map will have the same identity as the node from the database while the map is in memory, but will have different identities after the map is serialized to an XML document and stored in the database. This is consistent with the way XQuery treats node identity.

The keys take `xs:string` types, and the values take `item()*` values. Therefore, you can pass a string, an element, or a sequence of items to the values. Maps are a nice alternative to storing values an in-memory XML node and then using XPath to access the values. Maps makes it very easy to update the values.

12.2 map:map XQuery Primitive Type

Maps are defined as a `map:map` XQuery primitive type. You can use this type in function or variable definitions, or in the same way as you use other primitive types in XQuery. You can also serialize it to XML, which lets you store it in a database, as described in the following section.

12.3 Serializing a Map to an XML Node

You can serialize the structure of a map to an XML node by placing the map in the context of an XML element, in much the same way as you can serialize a `cts:query` (see Serializing a cts:query to XML in the Composing cts:query Expressions chapter of the *Search Developer's Guide*). Serializing the map is useful if you want to save the contents of the map by storing it in the database. The XML conforms to the `<marklogic-dir>/Config/map.xsd` schema, and has the namespace `http://marklogic.com/xdmp/map`.

For example, the following returns the XML serialization of the constructed map:

```
let $map := map:map()
let $key := map:put($map, "1", "hello")
let $key := map:put($map, "2", "world")
let $node := <some-element>{$map}</some-element>
return $node/map:map
```

The following XML is returned:

```
<map:map xmlns:map="http://marklogic.com/xdmp/map">
  <map:entry>
    <map:key>2</map:key>
    <map:value xsi:type="xs:string"
         xmlns:xs="http://www.w3.org/2001/XMLSchema"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
         world</map:value>
  </map:entry>
  <map:entry>
    <map:key>1</map:key>
    <map:value xsi:type="xs:string"
         xmlns:xs="http://www.w3.org/2001/XMLSchema"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
         hello</map:value>
  </map:entry>
</map:map>
```

12.4 Map API

The map API is quite simple. You can create a map either from scratch with the `map:map` function or from the XML representation (`map:map`) of the map. The following are the map functions. For the signatures and description of each function, see the *MarkLogic Built-In and Module Functions Reference*.

- `map:clear`
- `map:count`
- `map:delete`
- `map:get`
- `map:keys`
- `map:map`
- `map:put`

12.5 Examples

This section includes example code that uses maps and includes the following examples:

- Creating a Simple Map
- Returning the Values in a Map
- Constructing a Serialized Map
- Add a Value that is a Sequence

12.5.1 Creating a Simple Map

The following example creates a map, puts two key-value pairs into the map, and then returns the map.

```
let $map := map:map()
let $key := map:put($map, "1", "hello")
let $key := map:put($map, "2", "world")
return $map
```

This returns a map with two key-value pairs in it: the key “1” has a value “hello”, and the key “2” has a value “world”.

12.5.2 Returning the Values in a Map

The following example greates a map, then returns its values ordering by the keys:

```
let $map := map:map()
let $key := map:put($map, "1", "hello")
let $key := map:put($map, "2", "world")
return
  for $x in map:keys($map)
  order by $x return
  map:get($map, $x)
(: returns hello world :)
```

12.5.3 Constructing a Serialized Map

The following example creates a map like the previous examples, and then serializes the map to an XML node. It then makes a new map out of the XML node and puts another key-value pair in the map, and finally returns the new map.

```
let $map := map:map()
let $key := map:put($map, "1", "hello")
let $key := map:put($map, "2", "world")
let $node := <some-element>{$map}</some-element>
let $map2 := map:map($node/map:map)
let $key := map:put($map2, "3", "fair")
return $map2
```

This returns a map with three key-value pairs in it: the key “1” has a value “hello”, the key “2” has a value “world”, and the key “3” has a value “fair”. Note that the map bound to the `$map` variable is not the same as the map bound to `$map2`. After it was serialized to XML, a new map was constructed in the `$map2` variable.

12.5.4 Add a Value that is a Sequence

The values that you can put in a map are typed as an `item()*`, which means you can add arbitrary sequences as the value for a key. The following example includes some string values and a sequence value, and then outputs each results in a `<result>` element:

```
let $map := map:map()
let $key := map:put($map, "1", "hello")
let $key := map:put($map, "2", "world")
let $seq := ("fair",
             <some-xml>
                 <another-tag>with text</another-tag>
             </some-xml>)
let $key := map:put($map, "3", $seq)
return
  for $x in map:keys($map) return
  <result>{map:get($map, $x)}</result>
```

This returns the following elements:

```
<result>fair
        <some-xml>
            <another-tag>with text</another-tag>
        </some-xml>
</result>
<result>world</result>
<result>hello</result>
```

13.0 Function Values

This chapter describes how to use function values, which allow you to pass function values as parameters to XQuery functions. It includes the following sections:

- Overview of Function Values
- xdmp:function XQuery Primitive Type
- XQuery APIs for Function Values
- When the Applied Function is an Update from a Query Statement
- Example of Using Function Values

13.1 Overview of Function Values

XQuery functions take parameters, and those parameters can be any XQuery type. Typically, parameters are strings, dates, numbers, and so on, and XQuery has many types to provide robust typing support. Sometimes, however, it is convenient to pass a pointer to a named function as a parameter to another function. These function pointers are known as *function values*, and they allow you to write code that can be more robust ad more easily maintainable. Programming languages that support passing functions as parameters sometimes call those higher order functions. MarkLogic Server function values do most things that higher order functions in other languages do, except you cannot output a function and you cannot create anonymous functions; instead, you can output or input a function value, which is implemented as an XQuery primitive type.

You pass a function value to another function by telling it the name of the function you want to pass. The actual value returned by the function is evaluated dynamically during query runtime. Passing these function values allows you to define an interface to a function and have a default implementation of it, while allowing callers of that function to implement their own version of the function and specify it instead of the default version.

13.2 xdmp:function XQuery Primitive Type

Function values are defined as an `xdmp:function` XQuery primitive type. You can use this type in function or variable definitions, or in the same way as you use other primitive types in XQuery. Unlike some of the other MarkLogic Server XQuery primitive types (`cts:query` and `map:map`, for example), there is no XML serialization for the `xdmp:function` XQuery primitive type.

13.3 XQuery APIs for Function Values

The following XQuery built-in functions are used to pass function values:

- `xdmp:function`
- `xdmp:apply`

You use `xdmp:function` to specify the function to pass in, and `xdmp:apply` to run the function that is passed in. For details and the signature of these APIs, see the *MarkLogic Built-In and Module Functions Reference*.

13.4 When the Applied Function is an Update from a Query Statement

When you apply a function using `xdmp:function`, MarkLogic Server does not know the contents of the applied function at query compilation time. Therefore, if the statement calling `xdmp:apply` is a query statement (that is, it contains no update expressions and therefore runs at a timestamp), and the function being applied is performing an update, then it will throw an `XDMP-UDATEFUNCTIONFROMQUERY` exception.

If you have code that you will apply that performs an update, and if the calling query does not have any update statements, then you must make the calling query an update statement. To change a query statement to be an update statement, either use the `xdmp:update` prolog option or put an update call somewhere in the statement. For example, to force a query to run as an update statement, you can add the following to your XQuery prolog:

```
declare option xdmp:update "true";
```

Without the prolog option, any update expression in the query will force it to run as an update statement. For example, the following expression will force the query to run as an update statement and not change anything else about the query:

```
if ( fn:true() )
then ()
else xdmp:document-insert("fake.xml", <fake/>)
```

For details on the difference between update statements and query statements, see “Understanding Transactions in MarkLogic Server” on page 12.

13.5 Example of Using Function Values

The following example shows a recursive function, `my:sum:sequences`, that takes an `xdmp:function` type, then applies that function call recursively until it reaches the end of the sequence. It shows how the caller can supply her own implementation of the `my:add` function to change the behavior of the `my:sum-sequences` function. Consider the following library module named `/sum.xqy`:

```
xquery version "1.0-ml";
module namespace my="my-namespace";

(: Sum a sequence of numbers, starting with the
   starting-number (3rd parameter) and at the
   start-position (4th parameter). :)
declare function my:sum-sequence(
  $fun as xdmp:function,
  $items as item()*,
  $starting-number as item(),
  $start-position as xs:unsignedInt)
as item()
{
  if ($start-position gt fn:count($items)) then $starting-number
  else
    let $new-value := xdmp:apply($fun,$starting-number,
                          $items[$start-position])
    return
      my:sum-sequence($fun,$items,$new-value,$start-position+1)
};

declare function my:add($x,$y) {$x+ $y};
(: /sum.xqy :)
```

Now call this function with the following main module:

```
xquery version "1.0-ml";
import module namespace my="my-namespace" at "/sum.xqy";

let $fn := xdmp:function(xs:QName("my:add"))
return my:sum-sequence($fn,(1 to 100), 2, 1)
```

This returns 5052, which is the sum of all of the numbers between 2 and 100.

If you want to use a different formula for adding up the numbers, you can create an XQuery library module with a different implementation of the same function and specify it instead. For example, assume you want to use a different formula to add up the numbers, and you create another library module named `/my.xqy` that has the following code (it multiplies the second number by two before adding it to the first):

```
xquery version "1.0-ml";
module namespace my="my-namespace";

declare function my:add($x,$y) {$x+ (2 * $y)};
(: /my.xqy :)
```

You can now call the `my:sum-sequence` function specifying your new implementation of the `my:add` function as follows:

```
xquery version "1.0-ml";
import module namespace my="my-namespace" at "/sum.xqy";
```

```
let $fn := xdmp:function(xs:QName("my:add"), "/my.xqy")
return my:sum-sequence($fn,(1 to 100), 2, 1)
```

This returns 10102 using the new formula. This technique makes it possible for the caller to specify a completely different implementation of the specified function that is passed.

14.0 Reusing Content With Modular Document Applications

This chapter describes how to create applications that reuse content by using XML that includes other content. It contains the following sections:

- Modular Documents
- XInclude and XPointer
- CPF XInclude Application and API
- Creating XML for Use in a Modular Document Application
- Setting Up a Modular Document Application

14.1 Modular Documents

A *modular document* is an XML document that references other documents or parts of other documents for some or all of its content. If you fetch the referenced document parts and place their contents as child elements of the elements in which they are referenced, then that is called *expanding* the document. If you expand all references, including any references in expanded documents (recursively, until there is nothing left to expand), then the resulting document is called the *expanded document*. The expanded document can then be used for searching, allowing you to get relevance-ranked results where the relevance is based on the entire content in a single document. Modular documents use the XInclude W3C recommendation as a way to specify the referenced documents and document parts.

Modular documents allow you to manage and reuse content. MarkLogic Server includes a Content Processing Framework (CPF) application that expands the documents based on all of the XInclude references. The CPF application creates a new document for the expanded document, leaving the original documents untouched. If any of the parts are updated, the expanded document is recreated, automatically keeping the expanded document up to date.

The CPF application for modular documents takes care of all of the work involved in expanding the documents. All you need to do is add or update documents in the database that have XInclude references, and then anything under a CPF domain is automatically expanded. For details on CPF, see the *Content Processing Framework* guide.

Content can be reused by referencing it in multiple documents. For example, imagine you are a book publisher and you have boilerplate passages such as legal disclaimers, company information, and so on, that you include in many different titles. Each book can then reference the boilerplate documents. If you are using the CPF application, then if the boilerplate is updated, all of the documents are automatically updated. If you are not using the CPF application, you can still update the documents with a simple API call.

14.2 XInclude and XPointer

Modular documents use XInclude and XPointer technologies:

- XInclude: http://www.w3.org/TR/xinclude/
- XPointer: http://www.w3.org/TR/WD-xptr

XInclude provides a syntax for including XML documents within other XML documents. It allows you to specify a relative or absolute URI for the document to include. XPointer provides a syntax for specifying parts of an XML document. It allows you to specify a node in the document using a syntax based on (but not quite the same as) XPath. MarkLogic Server supports the XPointer framework, and the `element()` and `xmlns()` schemes of XPointer, as well as the `xpath()` scheme:

- `element()` Scheme: http://www.w3.org/TR/2002/PR-xptr-element-20021113/
- `xmlns()` Scheme: http://www.w3.org/TR/2002/PR-xptr-xmlns-20021113/
- `xpath()` Scheme, which is not a W3C recommendation, but allows you to use simple XPath to specify parts of a document.

The `xmlns()` scheme is used for namespace prefix bindings in the XPointer framework, the `element()` scheme is one syntax used to specify which elements to select out of the document in the XInclude `href` attribute, and the `xpath()` scheme is an alternate syntax (which looks much more like XPath than the `element()` scheme) to select elements from a document.

Each of these schemes is used within an attribute named `xpointer`. The `xpointer` attribute is an attribute of the `<xi:include>` element. If you specify a string corresponding to an `idref`, then it selects the element with that id attribute, as shown in "Example: Simple id" on page 128.

The examples that follow show XIncludes that use XPointer to select parts of documents:

- Example: Simple id
- Example: xpath() Scheme
- Example: element() Scheme
- Example: xmlns() and xpath() Scheme

14.2.1 Example: Simple id

Given a document `/test2.xml` with the following content:

```
<el-name>
  <p id="myID">This is the first para.</p>
  <p>This is the second para.</p>
</el-name>
```

The following selects the element with an `id` attribute with a value of `myID` from the `/test2.xml` document:

```
<xi:include href="/test2.xml" xpointer="myID" />
```

The expansion of this `<xi:include>` element is as follows:

```
<p id="myID" xml:base="/test2.xml">This is the first para.</p>
```

14.2.2 Example: xpath() Scheme

Given a document `/test2.xml` with the following content:

```
<el-name>
  <p id="myID">This is the first para.</p>
  <p>This is the second para.</p>
</el-name>
```

The following selects the second `p` element that is a child of the root element `el-name` from the `/test2.xml` document:

```
<xi:include href="/test2.xml" xpointer="xpath(/el-name/p[2])" />
```

The expansion of this `<xi:include>` element is as follows:

```
<p xml:base="/test2.xml">This is the second para.</p>
```

14.2.3 Example: element() Scheme

Given a document `/test2.xml` with the following content:

```
<el-name>
  <p id="myID">This is the first para.</p>
  <p>This is the second para.</p>
</el-name>
```

The following selects the second `p` element that is a child of the root element `el-name` from the `/test2.xml` document:

```
<xi:include href="/test2.xml" xpointer="element(/1/2)" />
```

The expansion of this `<xi:include>` element is as follows:

```
<p xml:base="/test2.xml">This is the second para.</p>
```

14.2.4 Example: xmlns() and xpath() Scheme

Given a document `/test2.xml` with the following content:

```
<pref:el-name xmlns:pref="pref-namespace">
  <pref:p id="myID">This is the first para.</pref:p>
  <pref:p>This is the second para.</pref:p>
</pref:el-name>
```

The following selects the first `pref:p` element that is a child of the root element `pref:el-name` from the `/test2.xml` document:

```
<xi:include href="/test2.xml"
            xpointer="xmlns(pref=pref-namespace)
                      xpath(/pref:el-name/pref:p[1])" />
```

The expansion of this `<xi:include>` element is as follows:

```
<pref:p id="myID" xml:base="/test2.xml"
   xmlns:pref="pref-namespace">This is the first para.</pref:p>
```

Note that the namespace prefixes for the XPointer must be entered in an xmlns() scheme; it does not inherit the prefixes from the query context.

14.3 CPF XInclude Application and API

This section describes the XInclude CPF application code and includes the following parts:

- XInclude Code and CPF Pipeline
- Required Security Privileges—xinclude Role

14.3.1 XInclude Code and CPF Pipeline

You can either create your own modular documents application or use the XInclude pipeline in a CPF application. For details on CPF, see the *Content Processing Framework* guide. The following are the XQuery libraries and CPF components used to create modular document applications:

- The XQuery module library `xinclude.xqy`. The key function in this library is the `xinc:node-expand` function, which takes a node and recursively expands any XInclude references, returning the fully expanded node.
- The XQuery module library `xpointer.xqy`.
- The XInclude pipeline and its associated actions.
- You can create custom pipelines based on the XInclude pipeline that use the following `<options>` to the XInclude pipeline. These options control the expansion of XInclude references for documents under the domain to which the pipeline is attached:
    - `<destination-root>` specifies the directory in which the expanded version of documents are saved. This should be a directory path in the database, and the expanded document will be saved to the URI that is the concatenation of this root and the base name of the unexpanded document. For example, if the URI of the unexpanded document is `/mydocs/unexpanded/doc.xml`, and the destination-root is set to `/expanded-docs/`, then this document is expanded into a document with the URI `/expanded-docs/doc.xml`.
    - `<destination-collection>` specifies the collection in which to put the expanded version. You can specify multiple collections by specifying multiple `<destination-collection>` elements in the pipeline.
    - `<destination-quality>` specifies the document quality for the expanded version. This should be an integer value, and higher positive numbers increase the relevance scores for matches against the document, while lower negative numbers decrease the relevance scores. The default quality on a document is 0, which does not change the relevance score.
    - The default is to use the same values as the unexpanded source.

14.3.2 Required Security Privileges—xinclude Role

The XInclude code requires the following privileges:

- `xdmp:with-namespaces`
- `xdmp:value`

Therefore, any users who will be expanding documents require these privileges. There us a predefined role called `xinclude` that has the needed privileges to execute this code. You must either assign the xinclude role to your users or they must have the above execute privileges in order to run the XInclude code used in the XInclude CPF application.

14.4 Creating XML for Use in a Modular Document Application

The basic syntax for using XInclude is relatively simple. For each referenced document, you include an `<xi:include>` element with an `href` attribute that has a value of the referenced document URI, either relative to the document with the `<xi:include>` element or an absolute URI of a document in the database. When the document is expanded, the document referenced replaces the `<xi:include>` element. This section includes the following parts:

- <xi:include> Elements
- <xi:fallback> Elements
- Simple Examples

14.4.1 <xi:include> Elements

Element that have references to content in other documents are `<xi:include>` elements, where `xi` is bound to the `http://www.w3.org/2001/XInclude` namespace. Each `xi:include` element has an `href` attribute, which has the URI of the included document. The URI can be relative to the document containing the `<xi:include>` element or an absolute URI of a document in the database.

14.4.2 <xi:fallback> Elements

The XInclude specification has a mechanism to specify *fallback* content, which is content to use when expanding the document when the XInclude reference is not found. To specify fallback content, you add an `<xi:fallback>` element as a child of the `<xi:include>` element. Fallback content is optional, but it is good practice to specify it. As long as the `xi:include href` attributes resolve correctly, documents without `<xi:fallback>` elements will expand correctly. If an `xi:include href` attribute does not resolve correctly, however, and if there are no `<xi:fallback>` elements for the unresolved references, then the expansion will fail with an `XI-BADFALLBACK` exception.

The following is an example of an `<xi:include>` element with an `<xi:fallback>` element specified:

```
<xi:include href="/blahblah.xml">
  <xi:fallback><p>NOT FOUND</p></xi:fallback>
</xi:include>
```

The `<p>NOT FOUND</p>` will be substituted when expanding the document with this `<xi:include>` element if the document with the URI `/blahblah.xml` is not found.

You can also put an `<xi:include>` element within the `<xi:fallback>` element to fallback to some content that is in the database, as follows:

```
<xi:include href="/blahblah.xml">
  <xi:fallback><xi:include href="/fallback.xml" /></xi:fallback>
</xi:include>
```

The previous element says to include the document with the URI `/blahblah.xml` when expanding the document, and if that is not found, to use the content in `/fallback.xml`.

14.4.3 Simple Examples

The following is a simple example which creates two documents, then expands the one with the XInclude reference:

```
xquery version "1.0-ml";
declare namespace xi="http://www.w3.org/2001/XInclude";

xdmp:document-insert("/test1.xml", <document>
  <p>This is a sample document.</p>
  <xi:include href="test2.xml"/>
</document>);

xquery version "1.0-ml";

xdmp:document-insert("/test2.xml",
  <p>This document will get inserted where
     the XInclude references it.</p>);

xquery version "1.0-ml";
import module namespace xinc="http://marklogic.com/xinclude"
     at "/MarkLogic/xinclude/xinclude.xqy";

xinc:node-expand(fn:doc("/test1.xml"))
```

The following is the expanded document returned from the `xinc:node-expand` call:

```
<document>
  <p>This is a sample document.</p>
  <p xml:base="/test2.xml">This document will get inserted where
     the XInclude references it.</p>
</document>
```

The base URI from the URI of the included content is added to the expanded node as an `xml:base` attribute.

You can include fallback content as shown in the following example:

```
xquery version "1.0-ml";
declare namespace xi="http://www.w3.org/2001/XInclude";

xdmp:document-insert("/test1.xml", <document>
  <p>This is a sample document.</p>
  <xi:include href="/blahblah.xml">
    <xi:fallback><p>NOT FOUND</p></xi:fallback>
  </xi:include>
</document>);

xquery version "1.0-ml";

xdmp:document-insert("/test2.xml",
  <p>This document will get inserted where the XInclude references
it.</p>);

xquery version "1.0-ml";

xdmp:document-insert("/fallback.xml",
  <p>Sorry, no content found.</p>);

xquery version "1.0-ml";
import module namespace xinc="http://marklogic.com/xinclude"
     at "/MarkLogic/xinclude/xinclude.xqy";

xinc:node-expand(fn:doc("/test1.xml"))
```

The following is the expanded document returned from the `xinc:node-expand` call:

```
<document>
  <p>This is a sample document.</p>
  <p xml:base="/test1.xml">NOT FOUND</p>
</document>
```

14.5 Setting Up a Modular Document Application

To set up a modular documents CPF application, you need to install CPF and create a domain under which documents with XInclude links will be expanded. For detailed information about the Content Processing Framework, including procedures for how to set it up and information about how it works, see the *Content Processing Framework* guide.

To set up an XInclude modular document application, perform the following steps:

1. Install Content Processing in your database, if it is not already installed. For example, if your database is named `modular`, In the Admin Interface click the Databases > modular > Content Processing link. If it is not already installed, the Content Processing Summary page will indicate that it is not installed. If it is not installed, click the Install tab and click install (you can install it with or without enabling conversion).

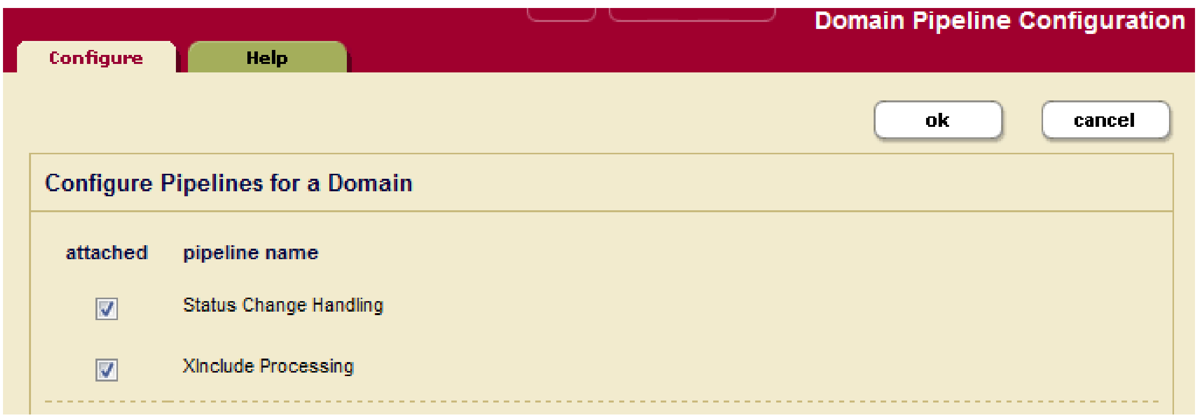
2. Click the domains link from the left tree menu. Either create a new domain or modify an existing domain to encompass the scope of the documents you want processed with the XInclude processing. For details on domains, see the *Content Processing Framework* guide.

3. Under the domain you have chosen, click the Pipelines link from the left tree menu.

4. Check the `Status Change Handling` and `XInclude Processing` pipelines. You can also attach other pipelines or detach other pipelines, depending if they are needed for your application.

   Note: If you want to change any of the `<options>` settings on the `XInclude Processing` pipeline, copy that pipeline to another file, make the changes (make sure to change the value of the `<pipeline-name>` element as well), and load the pipeline XML file. It will then be available to attach to a domain. For details on the options for the XInclude pipeline, see "CPF XInclude Application and API" on page 129.

5. Click OK. The Domain Pipeline Configuration screen shows the attached pipelines.



Any documents with XIncludes that are inserted or updated under your domain will now be expanded. The expanded document will have a URI ending in `_expanded.xml`. For example, if you insert a document with the URI `/test.xml`, the expanded document will be created with a URI of `/test_xml_expanded.xml` (assuming you did not modify the XInclude pipeline options).

Note: If there are existing XInclude documents in the scope of the domain, they will not be expanded until they are updated.

15.0 Controlling App Server Access, Output, and Errors

MarkLogic Server evaluates XQuery programs against App Servers. This chapter describes ways of controlling the output, both by App Server configuration and with XQuery built-in functions. Primarily, the features described in this chapter apply to HTTP App Servers, although some of them are also valid with XDBC Servers and with the Task Server. This chapter contains the following sections:

- Creating Custom HTTP Server Error Pages
- Setting Up URL Rewriting for an HTTP App Server
- Outputting SGML Entities
- Specifying the Output Encoding

15.1 Creating Custom HTTP Server Error Pages

This section describes how to use the HTTP Server error pages and includes the following parts:

- Overview of Custom HTTP Error Pages
- Error XML Format
- Configuring Custom Error Pages
- Execute Permissions Are Needed On Error Handler Document for Modules Databases
- Example of Custom Error Pages

15.1.1 Overview of Custom HTTP Error Pages

A custom HTTP Server error page is a way to redirect application exceptions to an XQuery program. When any 400 or 500 HTTP exception is thrown (except for a 503 error), an XQuery module is evaluated and the results are returned to the client. Custom error pages typically provide more user-friendly messages to the end-user, but because the error page is an XQuery module, you can make it perform arbitrary work.

The XQuery module can get the HTTP error code and the contents of the HTTP response using the `xdmp:get-response-code` API. The XQuery module for the error handler also has access to the XQuery stack trace, if there is one; the XQuery stack trace is passed to the module as an external variable with the name `$error:errors` in the XQuery `1.0-ml` dialect and as `$err:errors` in the XQuery `0.9-ml` dialect (they are both bound to the same namespace, but the `err` prefix is predefined in `0.9-ml` and `error` prefix is predefined in `1.0-ml`).

If the error is a 503 (unavailable) error, then the error handler is not invoked and the 503 exception is returned to the client.

If the error page itself throws an exception, that exception is passed to the client with the error code from the error page. It will also include a stack trace that includes the original error code and exception.

15.1.2 Error XML Format

Error messages are thrown with an XML error stack trace that uses the `error.xsd` schema. Stack trace includes any exceptions thrown, line numbers, and XQuery Version. Stack trace is accessible from custom error pages through the `$error:errors` external variable. The following is a sample error XML output for an XQuery module with a syntax error:

```
<error:error xsi:schemaLocation="http://marklogic.com/xdmp/error
   error.xsd"
   xmlns:error="http://marklogic.com/xdmp/error"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <error:code>XDMP-CONTEXT</error:code>
  <error:name>err:XPDY0002</error:name>
  <error:xquery-version>1.0-ml</error:xquery-version>
  <error:message>Expression depends on the context where none
               is defined</error:message>
  <error:format-string>XDMP-CONTEXT: (err:XPDY0002) Expression
    depends on the context where none is defined</error:format-string>
  <error:retryable>false</error:retryable>
  <error:expr/> <error:data/>
  <error:stack>
    <error:frame>
     <error:uri>/blaz.xqy</error:uri>
     <error:line>1</error:line>
     <error:xquery-version>1.0-ml</error:xquery-version>
    </error:frame>
  </error:stack>
</error:error>
```

15.1.3 Configuring Custom Error Pages

To configure a custom error page for an HTTP App Server, enter the name of the XQuery module in the Error Handler field of an HTTP Server. If the path does not start with a slash (`/`), then it is relative to the App Server root. If it does start with a slash (`/`), then it follows the import rules described in “Importing XQuery Modules and Resolving Paths” on page 56.

15.1.4 Execute Permissions Are Needed On Error Handler Document for Modules Databases

If your App Server is configured to use a modules database (that is, it stores and executes its XQuery code in a database) then you should put an execute permission on the error handler module document. The execute permission is paired to a role, and all users of the App Server must have that role in order to execute the error handler; if a user does not have the role, then that user will not be able to execute the error handler module, and it will get a 401 (unauthorized) error instead of having the error be caught and handled by the error handler.

As a consequence of needing the execute permission on the error handler, if a user who is actually not authorized to run the error handler attempts to access the App Server, that user runs as the default user configured for the App Server until authentication. If authentication fails, then the error handler is called as the default user, but because that default user does not have permission to execute the error handler, the user is not able to find the error handler and a 404 error (not found) is returned. Therefore, if you want all users (including unauthorized users) to have permission to run the error handler, you should give the default user a role (it does not need to have any privileges on it) and assign an execute permission to the error handler paired with that role.

15.1.5 Example of Custom Error Pages

The following XQuery module is an extremely simple XQuery error handler.

```
xquery version "1.0-ml";

declare variable $error:errors as node()* external;

xdmp:set-response-content-type("text/plain"),
xdmp:get-response-code(),
$error:errors
```

This simply returns all of the information from the page that throws the exception. In a typical error page, you would use some or all of the information and make a user-friendly representation of it to display to the users. Because you can write arbitrary XQuery in the error page, you can do a wide variety of things, including sending an email to the application administrator, redirecting it to a different page, and so on.

15.2 Setting Up URL Rewriting for an HTTP App Server

This section describes how to use the HTTP Server URL Rewriter feature and includes the following parts:

- Overview of URL Rewriting
- Creating a URL Rewrite Script
- Configuring an HTTP App Server to use the URL Rewrite Script
- More URL Rewrite Script Examples
- Prohibiting Access to Internal URLs
- URL Rewriting and Page-Relative URLs

15.2.1 Overview of URL Rewriting

You can access any MarkLogic Server resource with a URL, which is a fundamental characteristic of Representational state transfer (REST) services. In its raw form, the URL must either reflect the physical location of the resource (if it's a document in the database), or it must be of the form:

```
http://<dispatcher-program.xqy>?instructions=foo
```

Users of web applications typically prefer short, neat URLs to raw query string parameters. A concise URL, also referred to as a “clean URL,” is easy to remember, and less time-consuming to type in. If the URL can be made to relate clearly to the content of the page, then errors are less likely to happen. Also crawlers and search engines often use the URL of a web page to determine whether or not to index the URL and the ranking it receives. For example, a search engine may give a better ranking to a well-structured URL such as:

```
http://marklogic.com/technical/features.html
```

than to a less-structured, less-informative URL like the following:

```
http://marklogic.com/document?id=43759
```

In a “RESTful” environment, URLs should be well-structured, predictable, and decoupled from the physical location of a document or program. When an HTTP server receives an HTTP request with a well-structured, clean URL, it must be able to transparently map that to the internal URL of a document or program.

The URL Rewriter feature allows you to configure your HTTP App Server to enable the rewriting of internal URLs to clean URLs, giving you the flexibility to use any URL to point to any resource (web page, document, XQuery program and arguments). The URL Rewriter implemented by MarkLogic Server operates similarly to the Apache mod_rewrite module, only you write an XQuery program to perform the rewrite operation.

The URL rewriting happens through an internal redirect mechanism so the client is not aware of how the URL was rewritten. This makes the inner workings of a web site's address opaque to visitors. The internal URLs can also be blocked or made inaccessible directly if desired by rewriting them to non-existent URLs, as described in “Prohibiting Access to Internal URLs” on page 143.

15.2.2 Creating a URL Rewrite Script

Your first step in rewriting a URL is to create a URL rewrite script that reads the clean URL given to the server by the bowser and converts it to the raw URL recognized by the server.

For example, if you configured an HTTP App Server for the Shakespeare Demo Application at port 8060, you can access the demo with the URL:

```
http://localhost:8060/frames.html
```

A "cleaner," more descriptive URL would be something like:

```
http://localhost:8060/Shakespeare
```

To accomplish this URL rewrite, create script named `url_rewrite.xqy` that uses the `xdmp:get-request-url` function to read the URL given by the user and the `fn:replace` function to convert `/Shakespeare` to `/frames.html`:

```
xquery version "1.0-ml";

let $url := xdmp:get-request-url()
return fn:replace($url, "^/Shakespeare$", "/frames.html")
```

Though the URL is converted by the server to `/frames.html`, `/Shakespeare` is displayed in the browser's URL field after the page is opened.

The `xdmp:get-request-url` function returns the portion of the URL following the scheme and network location (domain name or *host_name*:*port_number*). In the above example, `xdmp:get-request-url` returns `/frames.html`. Unlike, `xdmp:get-request-path`, which returns only the query string, the `xdmp:get-request-url` function returns the query string and any anchors in the URL, all of which can be modified by your URL rewrite script.

You can create more elaborate URL rewrite scripts, as described in "More URL Rewrite Script Examples" on page 142 and "Prohibiting Access to Internal URLs" on page 143.

15.2.3 Configuring an HTTP App Server to use the URL Rewrite Script

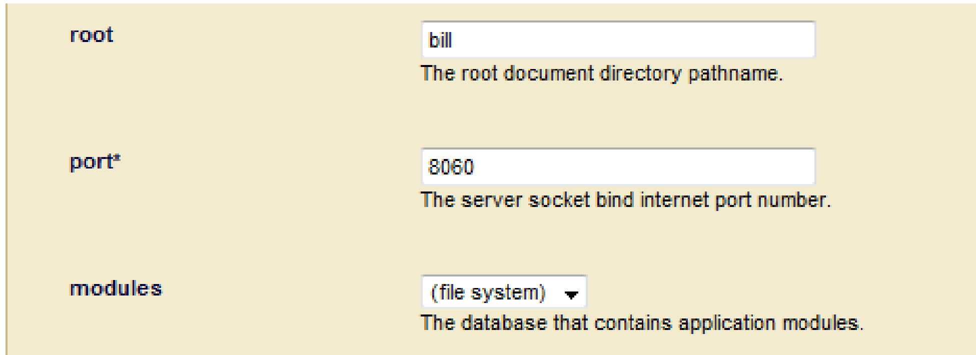
1. Click the Groups icon in the left frame.

2. Click the group in which you want to define the HTTP server (for example, Default).

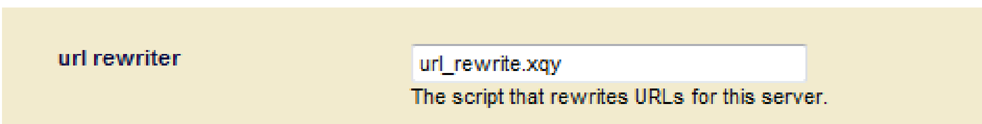
3. Click the App Servers icon on the left tree menu. and either select an exiting HTTP server or create a new one.

4. In the Root field, specify the root directory in which your executable files are located. for example, if your modules field is set to `(file system)` and the `.xqy` files for your Shakespeare demo are located in `C:\Program Files\MarkLogic\bill`, your root would be `bill`.

5. If Modules is set to `(file system)`, then place the URL rewrite script, `url_rewrite.xqy`, in the root directory. If Modules is set to a database, then load `url_rewrite.xqy` into that database under the root `bill`. For example, if using `xdmp:document-load`, the URI option would look like: `<uri>bill/url_rewrite.xqy</uri>`

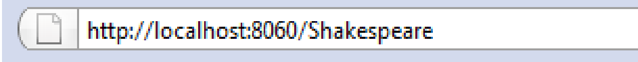


6. In the URL Rewriter field, specify the name of your URL rewrite script:



7. Click OK to save your changes.

8. From your browser, test the clean URL:



15.2.4 More URL Rewrite Script Examples

You can use the pattern matching features in regular expressions to create more flexible URL rewrite scripts.

For example, you want the user to only have to enter `/` after the scheme and network location portions of the URL (for example, `http://localhost:8060/`) and have it rewritten as `/frames.html`:

```
xquery version "1.0-ml";

let $url := xdmp:get-request-url()
return fn:replace($url,"^/$", "/frames.html")
```

In this example, you hide the `.xqy` extension from the browser's address bar and convert a static URL into a dynamic URL (containing a `?` character), you could do something like:

```
let $url := xdmp:get-request-url()

return fn:replace($url,
      "^/product-([0-9]+)\.html$",
      "/product.xqy?id=$1")
```

The product ID can be any number. For example, the URL `/product-12.html` is converted to `/product.xqy?id=12` and `/product-25.html` is converted to `/product.xqy?id=25`.

Search engine optimization experts suggest displaying the main keyword in the URL. In the following URL rewriting technique you can display the name of the product in the URL:

```
let $url := xdmp:get-request-url()

return fn:replace($url,
      "^/product/([a-zA-Z0-9_-]+)/([0-9]+)\.html$",
      "/product.xqy?id=$2")
```

The product name can be any string. For example, `/product/canned_beans/12.html` is converted to `/product.xqy?id=12` and `/product/cola_6_pack/8.html` is converted to `/product.xqy?id=8`.

If you need to rewrite multiple pages on your HTTP server, you can create a URL rewrite script like the following:

```
let $url := xdmp:get-request-url()

let $url := fn:replace($url, "^/Shaw$", "/frames1.html")
let $url := fn:replace($url, "^/Shakespeare$", "/frames2.html")
let $url := fn:replace($url, "^/Browning$", "/frames3.html")

return $url
```

15.2.5 Prohibiting Access to Internal URLs

The URL Rewriter feature also enables you to block user's from accessing internal URLs. For example, to prohibit direct access to `customer_list.html`, your URL rewrite script might look like the following:

```
let $url := xdmp:get-request-url()

return if (fn:matches($url,"^/customer_list.html$"))
       then "/nowhere.html"
       else fn:replace($url,"^/Shakespeare$", "/frames.html")
```

Where `/nowhere.html` is a non-existent page for which the browser returns a "404 Not Found" error. Alternatively, you could redirect to a URL consisting of a random number generated using `xdmp:random` or some other scheme that is guaranteed to generate non-existent URLs.

15.2.6 URL Rewriting and Page-Relative URLs

You may encounter problems when rewriting a URL to a page that makes use of page-relative URLs because relative URLs are resolved by the client. If the directory path of the clean URL used by the client differs from the raw URL at the server, then the page-relative links are incorrectly resolved.

If you are going to rewrite a URL to a page that uses page-relative URLs, convert the page-relative URLs to server-relative or canonical URLs. For example, if your application is located in `C:\Program Files\MarkLogic\myapp` and the page builds a frameset with page-relative URLs, like:

```
<frame src="top.html" name="headerFrame">
```

You should change the URLs to server-relative:

```
<frame src="/myapp/top.html" name="headerFrame">
```

or canonical:

```
<frame src="http://127.0.0.1:8000/myapp/top.html" name="headerFrame">
```

15.2.7 Using the URL Rewrite Trace Event

You can use the URL Rewrite trace event to help you debug your URL Rewrite scripts. To use the URL Rewrite trace event, you must enable tracing (at the group level) for your configuration and set the event:

1. Log into the Admin Interface.

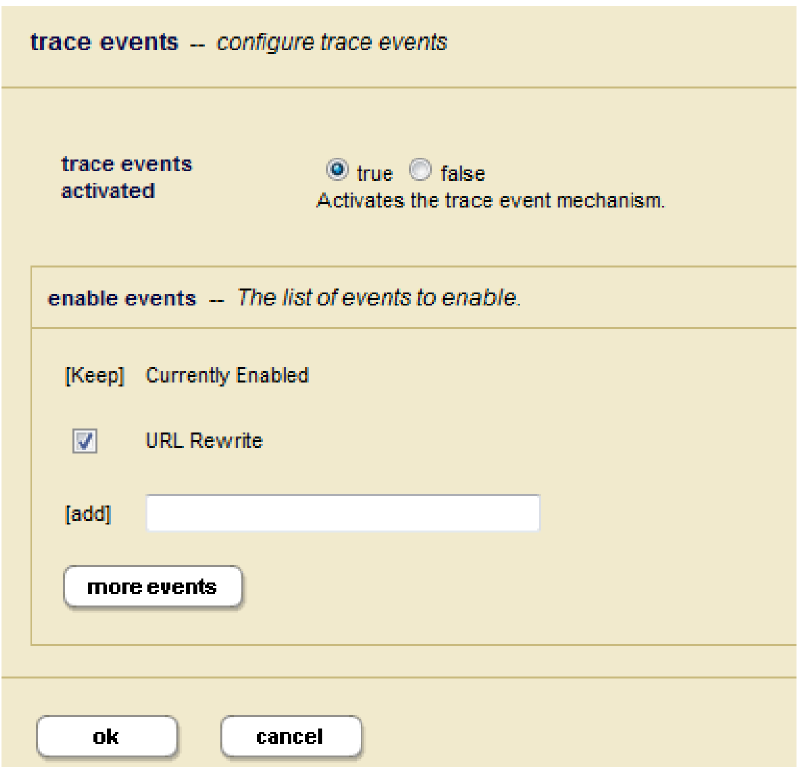
2. Select Groups > *group_name* > Diagnostics.

   The Diagnostics Configuration page appears.

3. Click the `true` button for `trace events activated`.

4. In the [add] field, enter: `URL Rewrite`

5. Click the OK button to activate the event.



After you configure the URL Rewrite trace event, when any URL Rewrite script is invoked, a line, like that shown below, is added to the `ErrorLog.txt` file, indicating the URL received from the client and the converted URL from the URL rewriter:

```
2009-02-11 12:06:32.587 Info: [Event:id=URL Rewrite] Rewriting URL
/Shakespeare to /frames.html
```

Note: The trace events are designed as development and debugging tools, and they might slow the overall performance of MarkLogic Server. Also, enabling many trace events will produce a large quantity of messages, especially if you are processing a high volume of documents. When you are not debugging, disable the trace event for maximum performance.

15.3 Outputting SGML Entities

This section describes the SGML entity output controls in MarkLogic Server, and includes the following parts:

- Understanding the Different SGML Mapping Settings
- Configuring SGML Mapping in the App Server Configuration
- Specifying SGML Mapping in an XQuery Program

15.3.1 Understanding the Different SGML Mapping Settings

An SGML character entity is a name separated by an ampersand (`&`) character at the beginning and a semi-colon (`;`) character at the end. The entity maps to a particular character. This markup is used in SGML, and sometimes is carried over to XML. MarkLogic Server allows you to control if SGML character entities upon serialization of XML on output, either at the App Server level using the Output SGML Character Entites drop down list or using the `<output-sgml-character-entities>` option to the built-in functions `xdmp:quote` or `xdmp:save`. When SGML characters are mapped (for an App Server or with the built-in functions), any unicode characters that have an SGML mapping will be output as the corresponding SGML entity. The default is `none`, which does not output any characters as SGML entites.

The mappings are based on the W3C XML Entities for Characters specification:

- http://www.w3.org/TR/2008/WD-xml-entity-names-20080721/

with the following modifications to the specification:

- Entities that map to multiple codepoints are not output, unless there is an alternate single-codepoint mapping available. Most of these entities are negated mathematical symbols (`nrarrw` from `isoamsa` is an example).
- The `gcedil` set is also included (it is not included in the specification).

The following table describes the different SGML character mapping settings:

SGML Character Mapping Setting	Description
`none`	The default. No SGML entity mapping is performed on the output.
`normal`	Converts unicode codepoints to SGML entities on output. The conversions are made in the default order. The only difference between `normal` and the `math` and `pub` settings is the order that it chooses to map entities, which only affects the mapping of entities where there are multiple entities mapped to a particular codepoint.
`math`	Converts unicode codepoints to SGML entities on output. The conversions are made in an order that favors math-related entities. The only difference between `math` and the `normal` and `pub` settings is the order that it chooses to map entities, which only affects the mapping of entities where there are multiple entities mapped to a particular codepoint.
`pub`	Converts unicode codepoints to SGML entities on output. The conversions are made in an order favoring entities commonly used by publishers. The only difference between `pub` and the `normal` and `math` settings is the order that it chooses to map entities, which only affects the mapping of entities where there are multiple entities mapped to a particular codepoint.

Note: In general, the `<repair>full</repair>` option on `xdmp:document-load` and the `"repair-full"` option on `xdmp:unquote` do the opposite of the Output SGML Character Entites settings, as the ingestion APIs map SGML entities to their codepoint equivalents (one or more codepoints). The difference with the output options is that the output options perform only single-codepoint to entity mapping, not multiple codepoint to entity mapping.

15.3.2 Configuring SGML Mapping in the App Server Configuration

To configure SGML output mapping for an App Server, perform the following steps:

1. In the Admin Interface, navigate to the App Server you want to configure (for example, Groups > Default > App Servers > MyAppServer).

2. Scroll down to the Output SGML Entity Characters drop list (it is towards the bottom).

3. Select the setting you want. The settings are described in the table in the previous section.

4. Click OK.

Codepoints that map to an SGML entity will now be serialized as the entity by default for requests against this App Server.

15.3.3 Specifying SGML Mapping in an XQuery Program

You can specify SGML mappings for XML output in an XQuery program using the `<output-sgml-character-entities>` option to the following XML-serializing APIs:

- `xdmp:quote`
- `xdmp:save`

For details, see the *MarkLogic Built-In and Module Functions Reference* for these functions.

15.4 Specifying the Output Encoding

By default, MarkLogic Server outputs content in utf-8. You can specify a different output encodings, both on an App Server basis and on a per-query basis. This section describes those techniques, and includes the following parts:

- Configuring App Server Output Encoding Setting
- XQuery Built-In For Specifying the Output Encoding

15.4.1 Configuring App Server Output Encoding Setting

You can set the output encoding for an App Server using the Admin Interface or with the Admin API. You can set it to any supported character set (see Collations and Character Sets By Language in the Encodings and Collations chapter of the *Search Developer's Guide*).

To configure output encoding for an App Server using the Admin Interface, perform the following steps:

1. In the Admin Interface, navigate to the App Server you want to configure (for example, Groups > Default > App Servers > MyAppServer).

2. Scroll down to the Output Encoding drop list (it is towards the bottom).

3. Select the encoding you want. The settings correspond to different languages, as described in the table in Collations and Character Sets By Language in the Encodings and Collations chapter of the *Search Developer's Guide*.

4. Click OK.

By default, queries against this App Server will now be output in the specified encoding.

15.4.2 XQuery Built-In For Specifying the Output Encoding

Use the following built-in functions to get and set the output encoding on a per-request basis:

- `xdmp:get-response-encoding`
- `xdmp:set-response-encoding`

Additionally, you can specify the output encoding for XML output in an XQuery program using the `<output-encoding>` option to the following XML-serializing APIs:

- `xdmp:quote`
- `xdmp:save`

For details, see the *MarkLogic Built-In and Module Functions Reference* for these functions.

16.0 JSON: Serializing To and Parsing From

This chapter describes how to use the JSON XQuery functions to parse JSON objects to XQuery and serialize XQuery types and values to JSON, and includes the following sections:

- Serializing and Parsing JSON To and From XQuery Types
- JSON API
- JSON Parsing Restrictions
- Examples

16.1 Serializing and Parsing JSON To and From XQuery Types

JSON (JavaScript Object Notation) is a data-interchange format which is designed to pass data to and from JavaScript. It is very useful in a web application to pass data back and forth between XQuery and JavaScript, and JSON is a popular mechanism for such data interchange. You could also use JSON to interchange data between XQuery and other programming environments. JSON is designed so it is both machine- and human-readable. For more details about JSON, see json.org.

To facilitate data-interchange, MarkLogic Server includes the JSON built-in XQuery functions. These functions allow you to easily take XQuery items and create JSON representations of them, and to take a JSON string and create XQuery items from it.

Most XQuery types are serialized to JSON in a way that they can be round-tripped (serialized to JSON and parsed from JSON back into a series of items in the XQuery data model) without any loss, but some types will not round-trip without loss. For example, an `xs:dateTime` value will serialize to a JSON string, but that same string would have to be cast back into an `xs:dateTime` value in XQuery in order for it to be equivalent to its original.

16.2 JSON API

There are two JSON API functions: one to serialize XQuery to JSON, and one to read a JSON string and create an XQuery data model from that string.

- `xdmp:to-json`
- `xdmp:from-json`

For the signatures and description of each function, see the *MarkLogic Built-In and Module Functions Reference*.

16.3 JSON Parsing Restrictions

There is not a one-to-one mapping between everything in JSON and everything in XQuery. Note the following when converting objects back and forth between XQuery and JSON:

- JSON null values are mapped to an empty string in XQuery (there is no null value in XQuery).
- You can have codepoints in JSON that are not legal in XQuery, such as the codepoint for the codepoint 1 (`\u0001`). If you have illegal codepoints, `xdmp:from-json` will throw an exception.
- You cannot have sequences of sequences in XQuery, and attempting to parse a JSON string that has an array of arrays will flatten the arrays into a single sequence.
- Any XML in a JSON string is parsed as text. If you have XML in a JSON string, you will have to use `xdmp:unquote` in XQuery to turn it into XML. Additionally, there is an XQuery library on the MarkLogic developer site that might help with JSON to XML conversions (http://developer.marklogic.com/svn/commons/trunk/json/json.xqy).

16.4 Examples

This section includes example code that use the JSON functions and includes the following examples:

- Serializing to a JSON String
- Parsing a JSON String into a List of Items

16.4.1 Serializing to a JSON String

The following code returns a JSON string that includes a map, a string, and an integer.

```
let $map := map:map()
let $put := map:put($map, "some-key", 45683)
let $string := "this is a string"
let $int := 123
return
xdmp:to-json(($map, $string, $int))

(:
returns:
[{"some-key":45683}, "this is a string", 123]
:)
```

For details on maps, see “Using the map Functions to Create Name-Value Maps” on page 117.

16.4.2 Parsing a JSON String into a List of Items

Consider the following, which is the inverse of the previous example:

```
let $json :=
  '[{"some-key":45683}, "this is a string", 123]'
return
xdmp:from-json($json)
```

This returns the following items:

```
map:map(
 <map:map xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:map="http://marklogic.com/xdmp/map">
  <map:entry key="some-key">
    <map:value xsi:type="xs:integer">45683</map:value>
  </map:entry>
 </map:map>)
this is a string
123
```

Note that what is shown above is the serialization of the XQuery items. You can also use some or all of the items in the XQuery data model. For example, consider the following, which adds to the map based on the other values:

```
xquery version "1.0-ml";
let $json :=
  '[{"some-key":45683}, "this is a string", 123]'
let $items := xdmp:from-json($json)
let $put := map:put($items[1], xs:string($items[3]), $items[2])
return
$items[1]

(: returns the following map:
map:map(
 <map:map xmlns:xs="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:map="http://marklogic.com/xdmp/map">
    <map:entry key="123">
       <map:value xsi:type="xs:string">this is a string</map:value>
    </map:entry>
    <map:entry key="some-key">
      <map:value xsi:type="xs:integer">45683</map:value>
    </map:entry>
 </map:map>)
:)
```

In the above query, the first item (`$items[1]`) returned from the `xdmp:from-json` call is a map, and the map is then modified, and then the modified map is returned. For details on maps, see “Using the map Functions to Create Name-Value Maps” on page 117.

17.0 Using Triggers to Spawn Actions

MarkLogic Server includes pre-commit and post-commit triggers. This chapter describes how triggers work in MarkLogic Server and includes the following sections:

- Overview of Triggers Used in the Content Processing Framework
- Pre-Commit Versus Post-Commit Triggers
- Trigger Events
- Trigger Scope
- Modules Invoked or Spawned by Triggers
- Creating and Managing Triggers With triggers.xqy
- Simple Trigger Example

The Content Processing Framework uses triggers to capture events and then set states in content processing pipelines. Any applications that use the Content Processing Framework Status Change Handling pipeline do not need to explicitly create triggers, as it automatically creates and manages the triggers as part of the Content Processing installation for a database. For details, see the *Content Processing Framework* manual.

17.1 Overview of Triggers Used in the Content Processing Framework

Conceptually, a trigger listens for certain events (document create, delete, update, or the database coming online) to occur, and then invokes an XQuery module to run after the event occurs. Trigger definitions are stored as XML documents in a database, and they contain information about the following:

- The type of event that causes the trigger to fire.
- The scope of documents to listen for events.
- Whether the trigger is of type pre-commit or post-commit.
- What XQuery module to invoke or spawn when the event occurs.

In a pipeline used with the Content Processing Framework, a trigger fires after one stage is complete (from a document update, for example) and then the XQuery module specified in the trigger is executed. When it completes, the next trigger in the pipeline fires, and so on. In this way, you can create complex pipelines to process documents.

The Status Change Handling pipeline, installed when you install Content Processing in a database, creates and manages all of the triggers needed for your content processing applications, so it is not necessary to directly create or manage any triggers in your content applications.

17.2 Pre-Commit Versus Post-Commit Triggers

There are two ways to configure the transactional semantics of a trigger: pre-commit and post-commit. This section describes each type of trigger and includes the following parts:

- Pre-Commit Triggers
- Post-Commit Triggers

17.2.1 Pre-Commit Triggers

The module invoked as the result of a pre-commit trigger is evaluated as part of the same transaction that produced the triggering event. It is evaluated by invoking the module on the same App Server in which the triggering transaction is run. It differs from invoking the module with `xdmp:invoke` in one way, however; the module invoked by the pre-commit trigger sees the updates made to the triggering document.

Therefore, pre-commit triggers and the modules from which the triggers are invoked execute in a single context; if the trigger fails to complete for some reason (if it throws an exception, for example), then the entire transaction, including the triggering transaction, is rolled back to the point before the transaction began its evaluation.

This transactional integrity is useful when you are doing something that does not make sense to break up into multiple asynchronous steps. For example, if you have an application that has a trigger that fires when a document is created, and the document needs to have an initial property set on it so that some subsequent processing can know what state the document is in, then it makes sense that the creation of the document and the setting of the initial property occur as a single transaction. As a single transaction (using a pre-commit trigger), if something failed while adding the property, the document creation would fail and the application could deal with that failure. If it were not a single transaction, then it is possible to get in a situation where the document is created, but the initial property was never created, leaving the content processing application in a state where it does not know what to do with the new document.

17.2.2 Post-Commit Triggers

The module spawned as the result of a post-commit trigger is evaluated as a separate transaction from the module that produced the triggering event. It executes asynchronously, and if the trigger module fails, it does not roll back the calling transaction. Furthermore, there is no guarantee that the trigger module will complete if it is called.

When a post-commit trigger spawns an XQuery module, it is put in the queue on the *task server*. The task server maintains this queue of tasks, and initiates each task in the order it was received. The task server has multiple threads to service the queue. There is one task server per group, and you can set task server parameters in the Admin Interface under Groups > *group_name* > Task Server.

Because post-commit triggers are asynchronous, the code that calls them must not rely on something in the trigger module to maintain data consistency. For example, the state transitions in the Content Processing Framework code uses post-commit triggers. The code that initiates the triggering event updates the property state before calling the trigger, allowing a consistent state in case the trigger code does not complete for some reason. Asynchronous processing has many advantages for state processing, as each state might take some time to complete. Asynchronous processing (using post-commit triggers) allows you to build applications that will not lose all of the processing that has already occurred should something happen in the middle of processing your pipeline. When the system is available again, the Content Processing Framework will simply continue the processing where it left off.

17.3 Trigger Events

Triggers can listen for the following events:

- document create
- document update
- document delete
- any property change (does *not* include MarkLogic Server-controlled properties such as `last-modified` and `directory`)
- specific (named) property change
- database coming online

Whether the module that the trigger invokes commits before or after the module that produced the triggering event depends if the trigger is a pre-commit or post-commit trigger. Pre-commit triggers in MarkLogic Server listen for the event and then invoke the trigger module *before* the transaction commits, making the entire process a single transaction that either all completes or all fails (although the module invoked from a pre-commit trigger sees the updates from the triggering event).

Post-commit triggers in MarkLogic Server initiate after the event is committed, and the module that the trigger spawns is run in a separate transaction from the one that updated the document. For example, a trigger on a document update event occurs *after* the transaction that updates the document commits to the database.

Because the post-commit trigger module runs in a separate transaction from the one that caused the trigger to spawn the module (for example, the create or update event), the trigger module transaction cannot, in the event of a transaction failure, automatically roll back to the original state of the document (that is, the state *before* the update that caused the trigger to fire). If this will leave your document in an inconsistent state, then the application must have logic to handle this state.

For more information on pre- and post-commit triggers, see “Pre-Commit Versus Post-Commit Triggers” on page 153.

17.4 Trigger Scope

The *trigger scope* is the scope with which to listen for create, update, delete, or property change events. The scope represents a portion of the database corresponding to one of the trigger scope values: `document`, `directory`, or `collection`.

A `document` trigger scope specifies a given document URI, and the trigger responds to the specified trigger events only on that document.

A `collection` trigger scope specifies a given collection URI, and the trigger responds to the specified trigger events for any document in the specified collection.

A `directory` scope represents documents that are in a specified directory, either in the immediate directory (depth of `1`); or in the immediate or any recursive subdirectory of the specified directory. For example, if you have a directory scope of the URI `/` (a forward-slash character) with a depth of `infinity`, that means that any document in the database with a URI that begins with a forward-slash character (`/`) will fire a trigger with this scope upon the specified trigger event. Note that in this directory example, a document called `hello.xml` is *not* included in this trigger scope (because it is not in the `/` directory), while documents with the URIs `/hello.xml` or `/mydir/hello.xml` are included.

17.5 Modules Invoked or Spawned by Triggers

Trigger definitions specify the URI of a module. This module is evaluated when the trigger is fired (when the event completes). The way this works is different for pre-commit and post-commit triggers. This section describes what happens when the trigger modules are invoked and spawned and includes the following subsections:

- Difference in Module Behavior for Pre- and Post-Commit Triggers
- Module External Variables trgr:uri and trgr:trigger

17.5.1 Difference in Module Behavior for Pre- and Post-Commit Triggers

For pre-commit triggers, the module is invoked when the trigger is fired (when the event completes). The invoked module is evaluated in an analogous way to calling `xdmp:invoke` in an XQuery statement, and the module evaluates synchronously in the same App Server as the calling XQuery module. The difference is that, with a pre-commit trigger, the invoked module sees the result of the triggering event. For example, if there is a pre-commit trigger defined to fire upon a document being updated, and the module counts the number of paragraphs in the document, it will count the number of paragraphs *after* the update that fired the trigger. Furthermore, if the trigger module fails for some reason (a syntax error, for example), then the entire transaction, including the update that fired the trigger, is rolled back to the state before the update.

For post-commit triggers, the module is spawned onto the task server when the trigger is fired (when the event completes). The spawned module is evaluated in an analogous way to calling `xdmp:spawn` in an XQuery statement, and the module evaluates asynchronously on the task server. Once the post-commit trigger module is spawned, it waits in the task server queue until it is evaluated. When the spawned module evaluates, it is run as its own transaction. Under normal circumstances the modules in the task server queue will initiate in the order in which they were added to the queue. Because the task server queue does not persist in the event of a system shutdown, however, the modules in the task server queue are not guaranteed to run.

17.5.2 Module External Variables trgr:uri and trgr:trigger

There are two external variables that are available to trigger modules:

- `trgr:uri as xs:string`
- `trgr:trigger as node()`

The `trgr:uri` external variable is the URI of the document which caused the trigger to fire (it is only available on triggers with data events, not on triggers with database online events). The `trgr:trigger` external variable is the trigger XML node, which is stored in the triggers database with the URI `http://marklogic.com/xdmp/triggers/`*`trigger_id`*, where *trigger_id* is the ID of the trigger. You can use these external variables in the trigger module by declaring them in the prolog as follows:

```
xquery version "1.0-ml";
import module namespace trgr='http://marklogic.com/xdmp/triggers'
   at '/MarkLogic/triggers.xqy';

declare variable $trgr:uri as xs:string external;
declare variable $trgr:trigger as node() external;
```

17.6 Creating and Managing Triggers With triggers.xqy

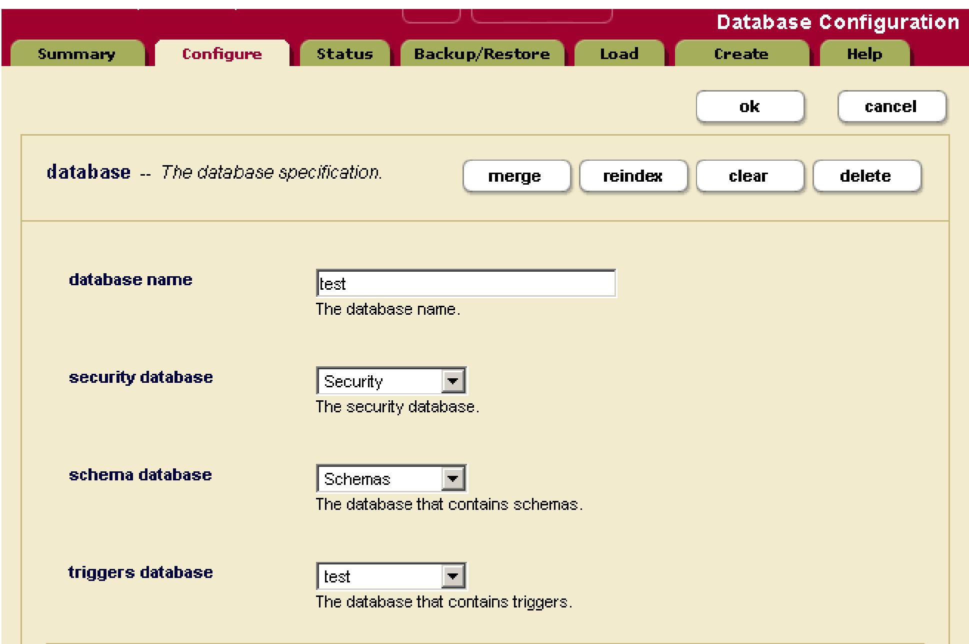
The `<install_dir>/Modules/MarkLogic/triggers.xqy` XQuery module file contains functions to create, delete, and manage triggers. If you are using the Status Change Handling pipeline, it takes care of all of the trigger details; you do not need to create or manage any triggers. For details on the trigger functions, see the *MarkLogic Built-In and Module Functions Reference*.

For real-world examples of XQuery code that creates triggers, see the `<install_dir>/Modules/MarkLogic/cpf/domains.xqy` XQuery module file. For a sample trigger example, see “Simple Trigger Example” on page 157. The functions in this module are used to create the needed triggers when you use the Admin Interface to create a domain.

17.7 Simple Trigger Example

The following example shows a simple trigger that fires when a document is created.

1. Use the Admin Interface to set up the database to use a triggers database. You can specify any database as the triggers database. The following screenshot shows the database named test used as both the database for the content and the triggers.



2. Create a trigger that listens for documents that are created under the directory `/myDir/` with the following XQuery code. Note that this code must be evaluated against the triggers database for the database in which your content is stored.

```
xquery version "1.0-ml";
import module namespace trgr="http://marklogic.com/xdmp/triggers"
   at "/MarkLogic/triggers.xqy";

trgr:create-trigger("myTrigger", "Simple trigger example",
  trgr:trigger-data-event(
      trgr:directory-scope("/myDir/", "1"),
      trgr:document-content("create"),
      trgr:post-commit()),
  trgr:trigger-module(xdmp:database("test"), "/modules/", "log.xqy"),
  fn:true(), xdmp:default-permissions() )
```

This code returns the ID of the trigger. The trigger document you just created is stored in the document with the URI `http://marklogic.com/xdmp/triggers/`*`trigger_id`*, where *trigger_id* is the ID of the trigger you just created.

3. Load a document whose contents is the XQuery module of the trigger action. This is the module that is spawned when the when the previously specified create trigger fires. For this example, the URI of the module must be `/modules/log.xqy` in the database named `test` (from the `trgr:trigger-module` part of the `trgr:create-trigger` code above). Note that the document you load, because it is an XQuery document, must be loaded as a text document and it must have execute permissions. For example, create a trigger module in the database by evaluating the following XQuery against the modules database for the App Server in which the triggering actions will be evaluated:

```
xdmp:document-insert("/modules/log.xqy",
  text{ "
xquery version '0.9-ml'
import module namespace trgr='http://marklogic.com/xdmp/triggers'
   at '/MarkLogic/triggers.xqy'

define variable $trgr:uri as xs:string external

xdmp:log(fn:concat('*****Document ', $trgr:uri, ' was created.*****'))"
} )
```

4. The trigger should now fire when you create documents in the database named `test` in the `/myDir/` directory. For example, the following:

```
xdmp:document-insert("/myDir/test.xml", <test/>)
```

will write a message to the `ErrorLog.txt` file similar to the following:

```
2007-03-12 20:14:44.972 Info: TaskServer: *****Document /myDir/test.xml
was created.*****
```

Note that this example only fires the trigger when the document is created. If you want it to fire a trigger when the document is updated, you will need a separate trigger with a `trgr:document-content` of `"modify"`.

18.0 Training the Classifier

MarkLogic Server includes an XML support vector machine (SVM) classifier. This chapter describes the classifier and how to use it on your content, and includes the following sections:

- Understanding How Training and Classification Works
- Classifier API
- Leveraging XML With the Classifier
- Creating a Training Set
- Methodology For Determining Thresholds For Each Class
- Example: Training and Running the Classifier

18.1 Understanding How Training and Classification Works

The *classifier* is a set of APIs that allow you to define *classes*, or categories of nodes. By running samples of classes through the classifier to train it on what constitutes a given class, you can then run that trained classifier on unknown documents or nodes to determine to which classes each belongs. The process of classification uses the full-text indexing capabilities of MarkLogic Server, as well as its XML-awareness, to perform statistical analysis of terms in the training content to determine class membership. This section describes the concepts behind the classifier and includes the following parts:

- Training and Classification
- XML SVM Classifier
- Hyper-Planes and Thresholds for Classes
- Training Content for the Classifier

18.1.1 Training and Classification

There are two basic steps to using the classifier: training and classification. *Training* is the process of taking content that is known to belong to specified classes and creating a classifier on the basis of that known content. *Classification* is the process of taking a classifier built with such a training content set and running it on unknown content to determine class membership for the unknown content. Training is an iterative process whereby you build the best classifier possible, and classification is a one-time process designed to run on unknown content.

18.1.2 XML SVM Classifier

The MarkLogic Server classifier implements a support vector machine (SVM). An SVM classifier uses a well-known algorithm to determine membership in a given class, based on training data. For background on the mathematics behind support vector machine (SVM) classifiers, try doing a web search for `svm classifier`, or start by looking at the information on Wikipedia.

The basic idea is that the classifier takes a set of training content representing known examples of classes and, by performing statistical analysis of the training content, uses the knowledge gleaned from the training content to decide to which classes other unknown content belongs. You can use the classifier to gain knowledge about your content based on the statistical analysis performed during training.

Traditional SVM classifiers perform the statistical analysis using term frequency as input to the support vector machine calculations. The MarkLogic XML SVM classifier takes advantage of MarkLogic Server's XML-aware full-text indexing capabilities, so the terms that act as input to the classifier can include content (for example, words), structure information (for example, elements), or a combination of content and structure (for example, element-word relationships). All of the MarkLogic Server index options that affect terms are available as options in the classifier API, so you can use a wide variety of indexing techniques to tune the classifier to work the best for your sample content.

First you define your classes on a set of training content, and then the classifier uses those classes to analyze other content and determine its classification. When the classifier analyzes the content, there are two sometimes conflicting measurements it uses to help determine if the information in the new content belongs in or out of a class:

- *Precision*: The probability that what is classified as being in a class is actually in that class. High precision might come at the expense of missing some results whose terms resemble those of other results in other classes.
- *Recall*: The probability that an item actually in a class is classified as being in that class. High recall might come at the expense of including results from other classes whose terms resemble those of results in the target class.

When you are tuning your classifier, you need to find a balance between high precision and high recall. That balance depends on what your application goals and requirements are. For example, if you are trying to find trends in your content, then high precision is probably more important; you want to ensure that your analysis does not include irrelevant nodes. If you need to identify every instance of some classification, however, you probably need a high recall, as missing any members would go against your application goals. For most applications, you probably need somewhere in between. The process of training your classifier is where you determine the optimal values (based on your training content set) to make the trade-offs that make sense to your application.

18.1.3 Hyper-Planes and Thresholds for Classes

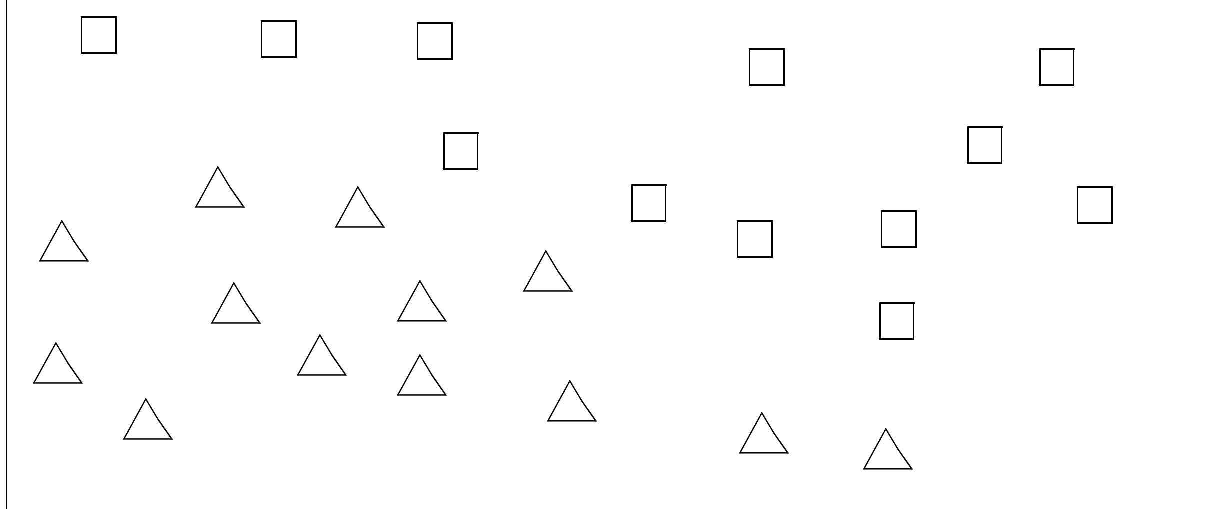
There are two main things that the computations behind the XML SVM classifier do:

- Determine the boundaries between each class. This is done during training.
- Determine the threshold for which the boundaries return the most distinctive results when determining class membership.

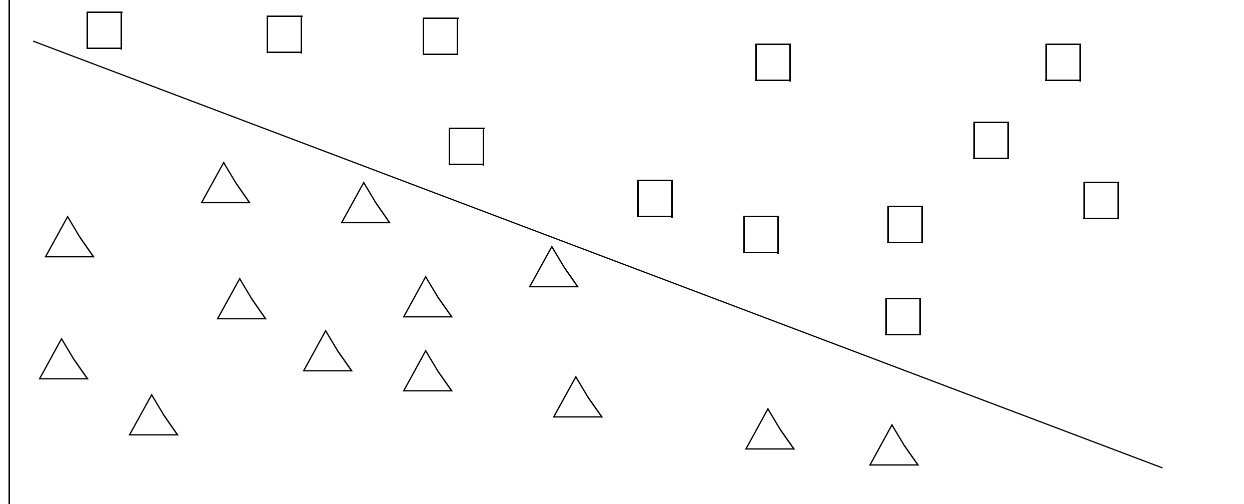
There can be any number of classes. A *term vector* is a representation of all of the terms (as defined by the index options) in a node. Therefore, classes consist of sets of term vectors which have been deemed similar enough to belong to the same class.

Imagine for a moment that each term forms a dimension. It is easy to visualize what a 2-dimensional picture of a class looks like (imagine an x-y graph) or even a 3-dimensional picture (imagine a room with height, width, and length). It becomes difficult, however, to visualize what the picture of these dimensions looks like when there are more than three dimensions. That is where *hyper-planes* become a useful concept.

Before going deeper into the concept of hyper-planes, consider a content set with two classes, one that are squares and one that are triangles. In the following figures, each square or triangle represents a term vector that is a member of either the square or triangle class, respectively.

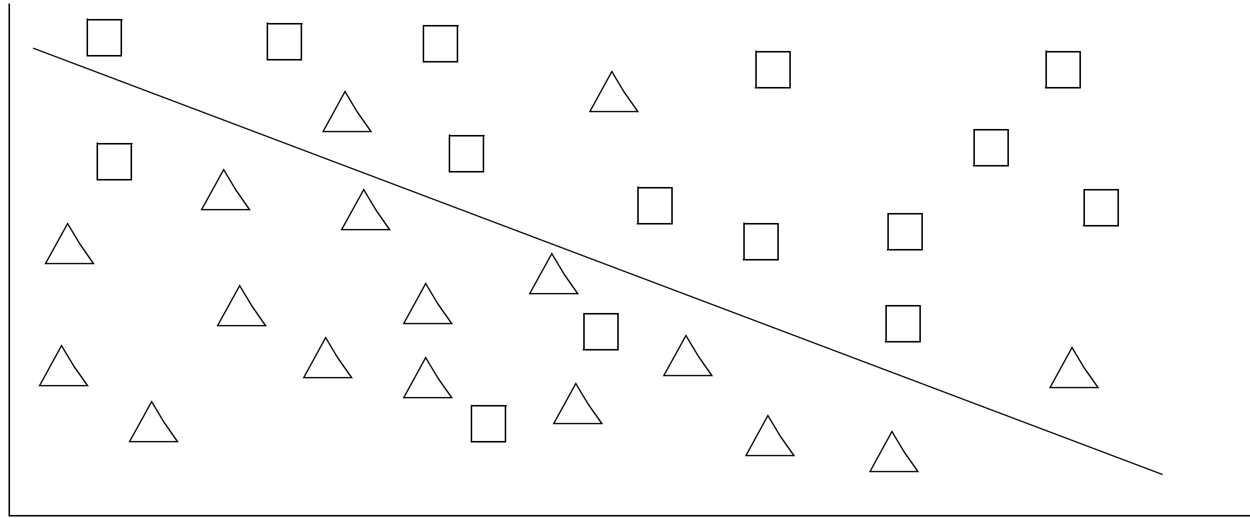


Now try to draw a line to separate the triangles from the squares. In this case, you can draw such a line that nicely divides the two classes as follows:

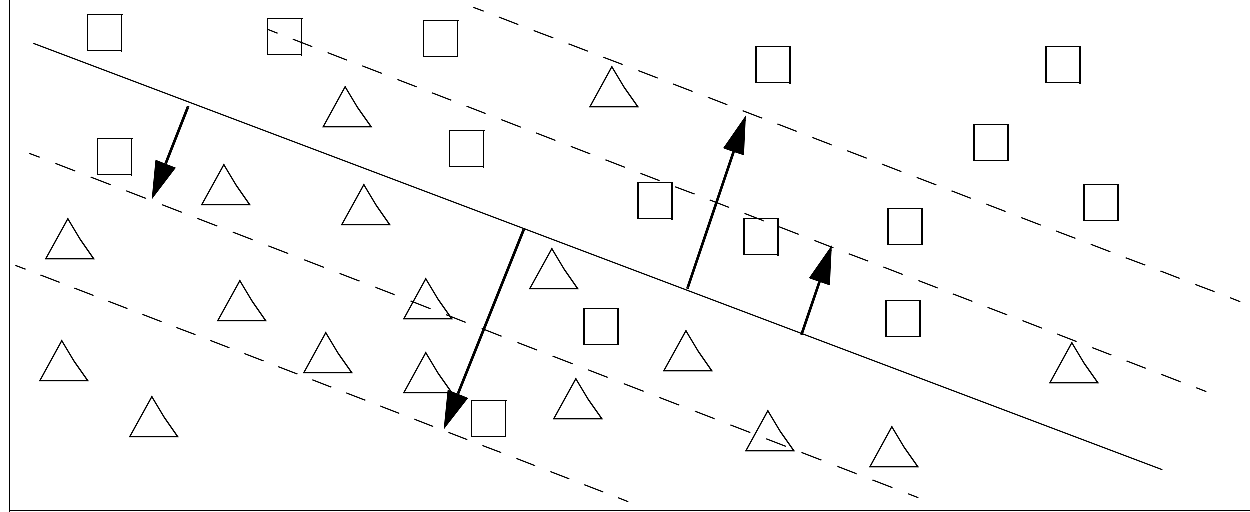


If this were three dimensions, instead of a line between the classes it would be a *plane* between the classes. When the number of dimensions grows beyond three, the extension of the plane is called a *hyper-plane*; it is the generalized representation of a boundary of a class (sometimes called the edge of a class).

The previous examples are somewhat simplified; they are set up such that the hyper-planes can be drawn such that one class is completely on one side and the other is completely on the other. For most real-world content, there are members of each class on the other side of the boundaries as follows:



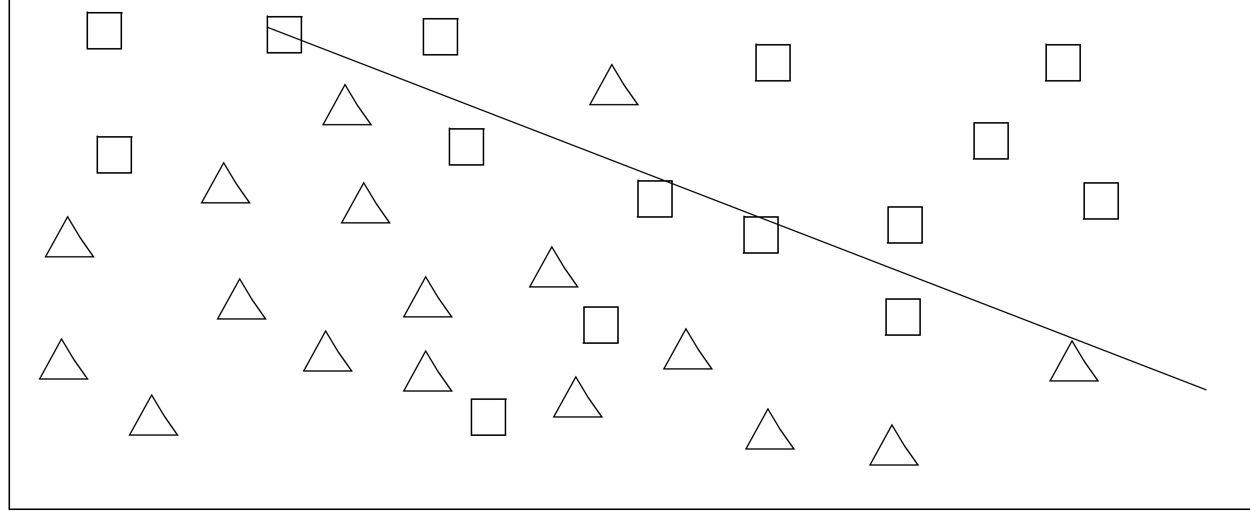
In these cases, you can draw other lines parallel to the boundaries (or in the *n*-dimensional cases, other hyper-planes). These other lines represent the *thresholds* for the classes. The distance between the boundary line and the threshold line represents the threshold value, which is a negative number indicating how far the outlier members of the class are from the class boundary. The following figure represents these thresholds.



The dotted lines represent some possible thresholds. The lines closer to the boundary represent thresholds with higher precision (but not complete precision), while the lines farther from the boundaries represent higher recall. For members of the triangle class that are on the other side of the square class boundaries, those members are not in the class, but if they are within the threshold you choose, then they are considered part of the class.

One of the classifier APIs (`cts:thresholds`) helps you find the right thresholds for your training content set so you can get the right balance between precision and recall when you run unknown content against the classifier to determin class membership.

The following figure shows the triangle class boundary, including the precision and recall calculations based on a threshold (the triangle class is below the threshold line):



Triangle Precision = 16/25 = .64
Triangle Recall = 16/17 = .94

18.1.4 Training Content for the Classifier

To find the best thresholds for your content, you need to *train* the classifier with sample content that represents members of all of the classes. It is very important to find good training samples, as the quality of the training will directly impact the quality of your classification.

The samples for each class should be statistically relevant, and should have samples that include both solid examples of the class (that is, samples that fall well into the positive side of the threshold from the class boundary) and samples that are close to the boundary for the class. The samples close to the boundary are very important, because they help determine the best thresholds for your content. For more details about training sets and setting the threshold, see "Creating a Training Set" on page 167 and "Methodology For Determining Thresholds For Each Class" on page 168.

18.2 Classifier API

The classifier has three XQuery built-in functions. This section gives an overview and explains some of the features of the API, and includes the following parts:

- XQuery Built-In Functions
- Data Can Reside Anywhere or Be Constructed
- API is Extremely Tunable
- Supports Versus Weights Classifiers
- Kernels (Mapping Functions)
- Find Thresholds That Balance Precision and Recall

For details about the syntax and usage of the classifier API, see the *MarkLogic Built-In and Module Functions Reference*.

18.2.1 XQuery Built-In Functions

The classifier API includes three XQuery functions:

- `cts:classify`
- `cts:thresholds`
- `cts:train`

You use these functions to take training nodes use them to compute classifiers. Creating a classifier specification is an iterative process whereby you create training content, train the classifier (using `cts:train`) with the training content, test your classifier on some other training content (using `cts:classify`), compute the thresholds on the training content (using `cts:threshold`), and repeat this process until you are satisfied with the results. For details about the syntax and usage of the classifier API, see the *MarkLogic Built-In and Module Functions Reference*.

18.2.2 Data Can Reside Anywhere or Be Constructed

The classifier APIs take nodes and elements, so you can either use XQuery to construct the data for the nodes you are classifying or training, or you can store them in the database (or somewhere else), whichever is more convenient. Because the APIs take nodes as parameters, there is a lot of flexibility in how you store your training and classification data.

> **Note:** There is an exception to this: if you are using the `supports` form of the classifier, then the training data must reside in the database, and you must pass in the training nodes when you perform classification (that is, when you run `cts:classify`) on unknown content.

18.2.3 API is Extremely Tunable

The classifier API has many options, and is therefore extremely tunable. You can choose the different index options and kernel types for `cts:train`, as well as specify limits and thresholds. When you change the kernel type for `cts:train`, it will effect the results you get from classification, as well as effect the performance. Because classification is an iterative process, experimentation with your own content set tends to help get better results from the classifier. You might change some parameters during different iterations and see which gives the better classification for your content.

The following section describes the differences between the supports and weights forms of the classifier. For details on what each option of the classifier does, see the *MarkLogic Built-In and Module Functions Reference*.

18.2.4 Supports Versus Weights Classifiers

There are two forms of the classifier:

- `supports`: allows the use of some of the more sophisticated kernels. It encodes the classifier by reference to specific documents in the training set, and is therefore more accurate because the whole training document can be used for classification; however, that means that the whole training set must be available during classification, and it must be stored in the database. Furthermore, since constructing a term vector is exactly equivalent to indexing, each time the classifier is invoked it regenerates the index terms for the whole training set. On the other hand, the actual representation of the classifier (the XML returned from `cts:train`) may be a lot more compact. The other advantage of the `supports` form of the classifier is that it can give you error estimates for specific training documents, which may be a sign that those are misclassified or that other parameters are not set to optimal values.
- `weights`: encodes weights for each of the terms. For mathematical reasons, it cannot be used with the Gaussian or Geodesic kernels, although for many problems, those kernels give the best results. Since there will not be a weight for every term in training set (because of term compression), this form of the classifier is intrinsically less precise. If there are a lot of classes and a lot of terms, the classifier representation itself can get quite

large. However, there is no need to have the training set on hand during classification, nor to construct term vectors from it (in essence to regenerate the index terms), so `cts:classify` runs much faster with the `weights` form of the classifier.

Which one you choose depends on your answers to several questions and criteria, such as performance (does the `supports` form take too much time and resources for your data?), accuracy (are you happy with the results you get with the `weights` form with your data?), and other factors you might encounter while experimenting with the different forms. In general, the classifier is extremely tunable, and getting the best results for your data will be an iterative process, both on what you use for training data and what options you use in your classification.

18.2.5 Kernels (Mapping Functions)

You can choose different kernels during the training phase. The kernels are mapping functions, and they are used to determine the distance of a term vector from the edge of the class. For a description of each of the kernel mapping functions, see the documentation for `cts:train` in the *MarkLogic Built-In and Module Functions Reference*.

18.2.6 Find Thresholds That Balance Precision and Recall

As part of the iterative nature of training to create a classifier specification, one of the overriding goals is to find the best threshold values for your classes and your content set. Ideally, you want to find thresholds that strike a balance between good precision and good recall (for details on precision and recall, see “XML SVM Classifier” on page 160). You use the `cts:thresholds` function to calculate the thresholds based on a training set. For an overview of the iterative process of finding the right thresholds, see “Methodology For Determining Thresholds For Each Class” on page 168.

18.3 Leveraging XML With the Classifier

Because the classifier operates from an XQuery context, and because it is built into MarkLogic Server, it is intrinsically XML-aware. This has many advantages. You can choose to classify based on a particular element or element hierarchy (or even a more complicated XML construct), and then use that classifier against either other like elements or element hierarchies, or even against a totally different set of element or element hierarchies. You can perform XML-based searches to find the best training data. If you have built XML structure into your content, you can leverage that structure with the classifier.

For example, if you have a set of articles that you want to classify, you can classify against only the `<executive-summary>` section of the articles, which can help to exclude references to other content sections, and which might have a more universal style and language than the more detailed sections of the articles. This approach might result in using terms that are highly relevant to the topic of each article for determining class membership.

18.4 Creating a Training Set

This section describes the training content set you use to create a classifier, and includes the following parts:

- Importance of the Training Set
- Defining Labels for the Training Set

18.4.1 Importance of the Training Set

The quality of your classification can only be as good as the training set you use to run the classifier. It is extremely important to choose sample training nodes that not only represent obvious examples of a class, but also samples which represent edge cases that belong in or out of a class.

Because the process of classification is about determining the edges of the classes, having good samples that are close to this edge is important. You cannot always determine what constitutes an edge sample, though, by examining the training sample. It it therefore good practice to get as many different kinds of samples in the training set as possible.

As part of the process of training the classifier, you might need to add more samples, verify that the samples are actually good samples, or even take some samples away (if they turn out to be poor samples) from some classes. Also, you can specify negative samples for a class. It is an iterative process of finding the right training data and setting the various training options until you end up with a classifier that works well for your data.

18.4.2 Defining Labels for the Training Set

The second parameter to `cts:train` is a label specification, which is a sequence of `cts:label` elements, each one having a one `cts:class` child. Each `cts:label` element represents a node in the training set. The `cts:label` elements must be in the order corresponding to the specified training nodes, and they each specify to which class the corresponding training node belongs. For example, the following `cts:label` nodes specifies that the first training node is in the class `comedy`, the second in the class `tragedy`, and the third in the class `history`:

```
<cts:label>
  <cts:class name="comedy"/>
</cts:label>
<cts:label>
  <cts:class name="tragedy"/>
</cts:label>
<cts:label>
  <cts:class name="history"/>
</cts:label>
```

Because the labels must be in the order corresponding to the training nodes, you might find it convenient to generate the labels from the training nodes. For example, the following code extracts the class name for the labels from a property names `playtype` stored in the property corresponding to the training nodes:

```
for $play in xdmp:directory("/plays/", "1")
return
  <cts:labels>
    <cts:class name={
      xdmp:document-property(xdmp:node-uri($play))//playtype/text()}/>
  </cts:labels>
```

If you have training samples that represent negative samples for a class (that is, they are examples of what does *not* belong in the class), you can label them such by specifying the `val="-1"` attribute on the `cts:class` element as follows:

```
<cts:class name="comedy" val="-1"/>
```

Additionally, you can include multiple classes in a label (because membership in one class is independent of membership in another). For example:

```
<cts:label>
  <cts:class name="comedy" val="-1"/>
  <cts:class name="tragedy"/>
  <cts:class name="history"/>
</cts:label>
```

18.5 Methodology For Determining Thresholds For Each Class

Use the following methodology to determine appropriate per-class thresholds for classification:

1. Partition the training set into two parts. Ideally, the partitions should be statistically equal. One way to achieve this is to randomize which nodes go into one partition and which go into the other.

2. Run `cts:train` on the first half of the training set.

3. Run `cts:classify` on the second half of the training set with the output of `cts:train` from the first half in the previous step. This is to validate that the training data you used produced good classification. Use the default value for the `thresholds` option for this run. The default value is a very large negative number, so this run will measure the distance from the actual class boundary for each node in the training set.

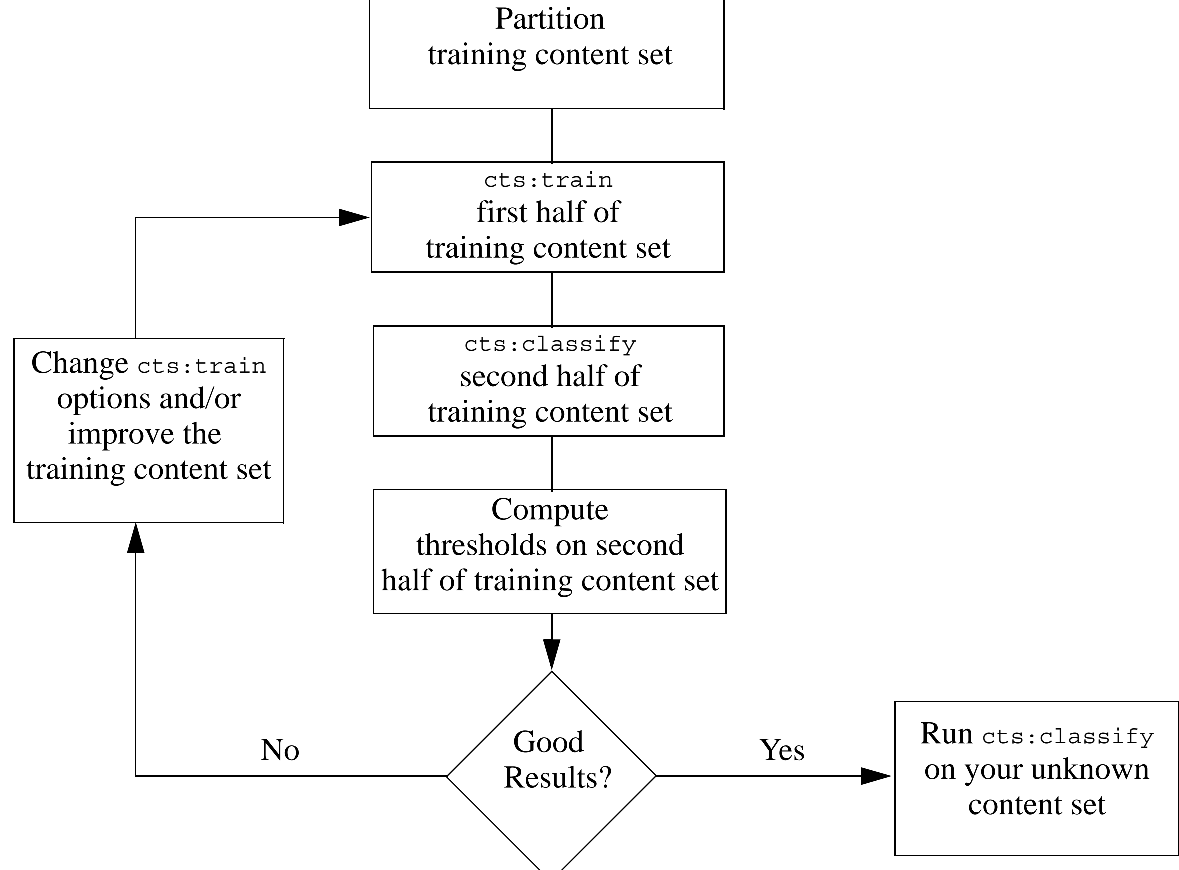
4. Run `cts:thresholds` to compute thresholds for the second half of the training set. This will further validate your training data and the parameters you set when running `cts:train` on your training data.

5. Iterate through the previous steps until you are satisfied with the results from your training content (that is, you until you are satisfied with the classifier you create). You might need to experiment with the various option settings for `cts:train` (for example, different kernels, different index settings, and so on) until you get the classification you desire.

6. After you are satisfied that you are getting good results, run `cts:classify` on the unknown documents, using the computed thresholds (the values from `cts:thresholds`) as the boundaries for deciding on class membership.

   Note: Any time you pass thresholds to `cts:train`, the thresholds apply to `cts:classify`. You can pass them either with `cts:train` or `cts:classify`, though, and the effect is the same.

The following diagram illustrates this iterative process:



18.6 Example: Training and Running the Classifier

This section describes the steps needed to train the classifier against a content set of the plays of William Shakespeare. This is meant is a simple example for illustrating how to use the classifier, not necessarily as an example of the best results you can get out of the classifier. The steps are divided into the following parts:

- Shakespeare's Plays: The Training Set
- Comedy, Tragedy, History: The Classes
- Partition the Training Content Set
- Create Labels on the First Half of the Training Content
- Run cts:train on the First Half of the Training Content
- Run cts:classify on the Second Half of the Content Set
- Use cts:thresholds to Compute the Thresholds on the Second Half
- Evaluating Your Results, Make Changes, and Run Another Iteration
- Run the Classifier on Other Content

18.6.1 Shakespeare's Plays: The Training Set

When you are creating a classifier, the first step is to choose some training content. In this example, we will use the plays of William Shakespeare as the training set from which to create a classifier.

The Shakespeare plays are available in XML at the following URL (subject to the copyright restrictions stated in the plays):

http://www.oasis-open.org/cover/bosakShakespeare200.html

This example assumes the plays are loaded into a MarkLogic Server database under the directory `/shakespeare/plays/`. There are 37 plays.

18.6.2 Comedy, Tragedy, History: The Classes

After deciding on the training set, the next step is to choose classes in which you divide the set, as well as choosing labels for those classes. For Shakespeare, the classes are COMEDY, TRAGEDY, and HISTORY. You must decide which plays belong to each class. To determine which Shakespeare plays are comedies, tragedies, and histories, consult your favorite Shakespeare scholars (there is reasonable, but not complete agreement about which plays belong in which classes).

For convenience, we will store the classes in the properties document at each play URI. To create the properties for each document, perform something similar to the following for each play (inserting the appropriate class as the property value):

```
xdmp:document-set-properties("/shakespeare/plays/hamlet.xml",
      <playtype>TRAGEDY</playtype>)
```

For details on properties in MarkLogic Server, see "Properties Documents and Directories" on page 94.

18.6.3 Partition the Training Content Set

Next, we will divide the training set into two parts, where we know the class of each node in both parts. We will use the first part to train and the second part to validate the classifier built from the first half of the training set. The two parts should be statistically random, and to do that we will simply take the first half in the order that the documents return from the `xdmp:directory` call. You can choose a more sophisticated randomization technique if you like.

18.6.4 Create Labels on the First Half of the Training Content

As we are taking the first half of the play for the training content, we will need labels for each node (in this example, we are using the document node for each play as the training nodes). To create the labels on the first half of the content, run a query statement similar to the following:

```
for $x in xdmp:directory("/shakespeare/plays/", "1")[1 to 19]
return
<cts:label>
  <cts:class name={xdmp:document-properties(xdmp:node-uri($x))
       //playtype/text()}/>
</cts:label>
```

Note: For simplicity, this example uses the first 19 items of the content set as the training nodes. The samples you use should use a statistically random sample of the content for the training set, so you might want to use a slightly more complicated method (that is, one that ensures randomness) for choosing the training set.

18.6.5 Run cts:train on the First Half of the Training Content

Next, you run `cts:train` with your training content and labels. The following code constructs the labels and runs `cts:train` to generate a classifier specification:

```
let $firsthalf := xdmp:directory("/shakespeare/plays/", "1")[1 to 19]
let $labels := for $x in $firsthalf
       return
       <cts:label>
         <cts:class name={xdmp:document-properties(xdmp:node-uri($x))
                //playtype/text()}/>
       </cts:label>
return
cts:train($firsthalf, $labels,
       <options xmlns="cts:train">
         <classifier-type>supports</classifier-type>
       </options>)
```

You can either save the generated classifier specification in a document in the database or run this code dynamically in the next step.

18.6.6 Run cts:classify on the Second Half of the Content Set

Next, you take the classifier specification created with the first half of the training set and run `cts:classify` on the second half of the content set, as follows:

```
let $firsthalf := xdmp:directory("/shakespeare/plays/", "1")[1 to 19]
let $secondhalf := xdmp:directory("/shakespeare/plays/", "1")[20 to 37]
let $classifier :=
  let $labels := for $x in $firsthalf
        return
        <cts:label>
          <cts:class name={xdmp:document-properties(xdmp:node-uri($x))
                             //playtype/text()}/>
        </cts:label>
  return
  cts:train($firsthalf, $labels,
      <options xmlns="cts:train">
        <classifier-type>supports</classifier-type>
      </options>)
return
cts:classify($secondhalf, $classifier,
      <options xmlns="cts:classify"/>,
      $firsthalf)
```

18.6.7 Use cts:thresholds to Compute the Thresholds on the Second Half

Next, calculate `cts:label` elements for the second half of the content and use it to compute the thresholds to use with the classifier. The following code runs `cts:train` and `cts:classify` again for clarity, although the output of each could be stored in a document.

```
let $firsthalf := xdmp:directory("/shakespeare/plays/", "1")[1 to 19]
let $secondhalf := xdmp:directory("/shakespeare/plays/", "1")[20 to 37]
let $firstlabels := for $x in $firsthalf
        return
        <cts:label>
          <cts:class name={xdmp:document-properties(xdmp:node-uri($x))
                                     //playtype/text()}/>
        </cts:label>
let $secondlabels := for $x in $secondhalf
        return
        <cts:label>
          <cts:class name={xdmp:document-properties(xdmp:node-uri($x))
                                     //playtype/text()}/>
        </cts:label>
let $classifier :=
    cts:train($firsthalf, $firstlabels,
      <options xmlns="cts:train">
        <classifier-type>supports</classifier-type>
      </options>)
let $classifysecond :=
  cts:classify($secondhalf, $classifier,
        <options xmlns="cts:classify"/>,
        $firsthalf)
return
cts:thresholds($classifysecond, $secondlabels)
```

This produces output similar to the following:

```
<thresholds xmlns="http://marklogic.com/cts">
  <class name="TRAGEDY" threshold="-0.00215207" precision="1"
         recall="0.666667" f="0.8" count="3"/>
  <class name="COMEDY" threshold="0.216902" precision="0.916667"
         recall="1" f="0.956522" count="11"/>
  <class name="HISTORY" threshold="0.567648" precision="1"
         recall="1" f="1" count="4"/>
</thresholds>
```

18.6.8 Evaluating Your Results, Make Changes, and Run Another Iteration

Finally, you can analyze the results from `cts:thresholds`. As an ideal, the thresholds should be zero. In practice, a negative number relatively close to zero makes a good threshold. The threshold for tragedy above is quite good, but the thresholds for the other classes are not quite as good. If you want the thresholds to be better, you can try running everything again with different parameters for the kernel, for the indexing options, and so on. Also, you can change your training data (to try and find better examples of comedy, for example).

18.6.9 Run the Classifier on Other Content

Once you are satisfied with your classifier, you can run it on other content. For example, you can try running it on SPEECH elements in the shakespeare plays, or try it on plays by other playwrights.

19.0 Technical Support

MarkLogic provides technical support according to the terms detailed in your Software License Agreement or End User License Agreement. For evaluation licenses, MarkLogic may provide support on an “as possible” basis.

For customers with a support contract, we invite you to visit our support website at http://support.marklogic.com to access information on known and fixed issues.

For complete product documentation, the latest product release downloads, and other useful information for developers, visit our developer site at http://developer.marklogic.com.

If you have questions or comments, you may contact MarkLogic Technical Support at the following email address:

support@marklogic.com

If reporting a query evaluation problem, please be sure to include the sample XQuery code.