MarkLogic Server

Loading Content Into MarkLogic Server

MarkLogic 8
February, 2015

Last Revised: 8.0-9, August, 2018

Table of Contents

Loading Content Into MarkLogic Server

1.0 Designing a Content Loading Strategy 4
1.1 Available Content Loading Interfaces 5
1.2 Loading Activities 6
1.3 What to Consider Before Loading Content 7
1.3.1 Setting Document Permissions 7
1.3.2 Schemas 7
1.3.3 Fragments 8
1.3.4 Indexing 8

2.0 Controlling Document Format 9
2.1 Terminology 9
2.2 Supported Document Formats 10
2.2.1 JSON Format 10
2.2.2 XML Format 11
2.2.3 Binary Format 11
2.2.4 Text (CLOB) Format 11
2.3 Choosing a Binary Format 12
2.3.1 Loading Binary Documents 15
2.3.2 Configuring MarkLogic Server for Binary Documents 15
2.4 Implicitly Setting the Format Based on the MIME Type 15
2.5 Explicitly Setting the Format 16
2.6 Determining the Format of a Document 17

3.0 Specifying Encoding and Language 18
3.1 Understanding Character Encoding 18
3.2 Explicitly Specifying Character Encoding While Loading 19
3.3 Automatically Detecting the Encoding 20
3.4 Inferring the Language and Encoding of a Node in XQuery with xdmp:encoding-language-detect 20
3.5 Specifying the Default Language for XML Documents 21

4.0 Loading Content Using XQuery 23
4.1 Built-In Document Loading Functions 23
4.2 Specifying a Forest in Which to Load a Document 24
4.2.1 Consider If You Really Want to Specify a Forest 24
4.2.2 Some Potential Advantages of Specifying a Forest 25
4.2.3 Example: Examining a Document to Decide Which Forest to Specify 25

4.2.4 More Examples 26
4.3 Creating External Binary References Using XQuery 26

5.0 Loading Content Using REST, Java or Node.js 28

6.0 Loading Content Using MarkLogic Content Pump 29

7.0 Loading Content Using WebDAV 30

8.0 Repairing XML Content During Loading 31
8.1 Programming Interfaces and Supported Content Repair Capabilities 32
8.2 Enabling Content Repair 32
8.3 General-Purpose Tag Repair 33
8.3.1 How General-Purpose Tag Repair Works 33
8.3.2 Pitfalls of General-Purpose Tag Repair 34
8.3.3 Limitations 35
8.3.3.1 XQuery Functions 35
8.3.3.2 Root Element 35
8.3.3.3 Previous Marklogic Versions 35
8.3.4 Controlling General-Purpose Tag Repair 35
8.4 Auto-Close Repair of Empty Tags 36
8.4.1 What Empty Tag Auto-Close Repair Does 36
8.4.2 Defining a Schema to Support Empty Tag Auto-Close Repair 37
8.4.3 Invoking Empty Tag Auto-Close Repair 38
8.4.4 Scope of Application 40
8.4.5 Disabling Empty Tag Auto-Close 41
8.5 Schema-Driven Tag Repair 41
8.5.1 What Schema-Driven Tag Repair Does 41
8.5.2 How to Invoke Schema-Driven Tag Repair 43
8.5.3 Scope of Application 44
8.5.4 Disabling Schema-Driven Tag Repair 44
8.6 Load-Time Default Namespace Assignment 45
8.6.1 How Default Namespace Assignments Work 45
8.6.2 Scope of Application 46
8.7 Load-Time Namespace Prefix Binding 46
8.7.1 How Load-Time Namespace Prefix Binding Works 47
8.7.2 Interaction with Load-Time Default Namespace Assignment 48
8.7.3 Scope of Application 50
8.7.4 Disabling Load-Time Namespace Prefix Binding 50
8.8 Query-Driven Content Repair 50
8.8.1 Point Repair 51
8.8.2 Document Walkers 51

9.0 Modifying Content During Loading 54

9.1 Converting Microsoft Office and Adobe PDF Into XML 54
9.2 Converting to XHTML 55
9.3 Automating Metadata Extraction 55
9.4 Transforming XML Structures 55

10.0 Performance Considerations 56
10.1 Understanding the Locking and Journaling Database Settings for Bulk Loads ...56
10.2 Fragmentation 57

11.0 Technical Support 58

12.0 Copyright 59
12.0 NOTICE 59

1.0 Designing a Content Loading Strategy

MarkLogic Server provides many ways to load content into a database including built-in XQuery functions, the REST Client API, and the command-line tool, MarkLogic Content Pump (mlcp). Choosing the appropriate method for a specific use case depends on many factors, including the characteristics of your content, the source of the content, the frequency of loading, and whether the content needs to be repaired or modified during loading. In addition, environmental and operational factors such as workflow integration, development resources, performance considerations, and developer expertise often need to be considered in choosing the best tools and mechanisms.

The MarkLogic mechanisms for loading content provide varying trade-offs along a number of dimensions such as the following:

- Usability and flexibility of the interface itself
- Performance, scalability, I/O capacity
- Loading frequency
- Automation or scripting requirements
- Workflow and integration requirements

This chapter lists the various tools to load content and contains the following sections:

- Available Content Loading Interfaces
- Loading Activities
- What to Consider Before Loading Content

1.1 Available Content Loading Interfaces

There are several ways to load content into MarkLogic Server. The following table summarizes content loading interfaces and their benefits.

Interface/Tool	Description	Benefits
MarkLogic Content Pump (mlcp)	A command line tool for loading content into a MarkLogic database, extracting content from a MarkLogic database, or copying content between MarkLogic databases.	Ease of workflow integration, can leverage Hadoop processing, bulk loading of billions of local files, split and load aggregate XML or delimited text files
MarkLogic Connector for Hadoop	A set of Java classes that enables loading content from HDFS into MarkLogic Server.	Distributed processing of large amounts of data
Java Client API	A Java API for creating applications on top of MarkLogic Server. The API includes document manipulation and search operations.	Leverage existing Java programming skills
Node.js Client API	A set of Node.js interfaces for creating applications on top of MarkLogic Server. The API includes document manipulation and search operations.	Leverage existing Node.js programming skills

Interface/Tool	Description	Benefits
REST Client API	A set of HTTP REST services hosted that enable developers to build applications on top of MarkLogic Server. The API includes document manipulation and search operations.	Leverage existing REST programming skills
XCC	XML Contentbase Connector (XCC) is an interface to communicate with MarkLogic Server from a Java or .NET middleware application layer	Create multi-tier applications with MarkLogic Server as the underlying content repository
XQuery API	An extensive set of XQuery functions that provides maximum control	Flexibility and expanded capabilities
Server-Side JavaScript API	An extensive set of JavaScript functions that execute on MarkLogic function and provide maximum control	Flexibility and expanded capabilities

1.2 Loading Activities

There are various things you can do with each of the loading interfaces, all resulting in ingesting data into the database. The following are some of the things you might do through the interfaces. Which interface you use is a matter of which you are most comfortable with as well as trade-offs that some might have over others (for example, ease-of-use versus extensibility). While each tool can usually accomodate each of these activities, in cases where one tool has a specific feature to make one of these activites easy, it is called out in the list.

- Load from a directory (mlcp)
- Load from compressed files (mlcp)
- Split single aggregate XML file into multiple documents (mlcp)
- Load large numbers of small files
- Load delimited text files (mlcp)
- Enrich the documents
- Extract information from the documents during ingestion (metadata, new elements)
- Extract some information and load only extracted information
- Load large binary files
- Create neutral format archive (mlcp)
- Copy from one ML database to another ML database (mlcp)

1.3 What to Consider Before Loading Content

Designing your content loading strategy depends on the complexity of your source content, the nature of the output to be inserted into the database and many other factors. This section lists some of the areas to think about with links to more detailed discussions and contains the following parts:

- Setting Document Permissions
- Schemas
- Fragments
- Indexing

1.3.1 Setting Document Permissions

When you load documents into a database, be sure you either explicitly set permissions in the document loading API or have configured default permissions on the user (or on roles for that user) who is loading the documents. Default permissions are applied to a document 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. To have a specific 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 Guide*. For details on creating privileges and setting permissions, see the Security Administration chapter of the *Administrator's Guide*.

> **Note:** When you load a document, be sure 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 MarkLogic throws an `XDMP-MUSTHAVEUPDATE` exception during document creation. If there is no role on a document's permissions with update capability, or if the document has no permissions, then only users with the `admin` role can update or delete the document.

1.3.2 Schemas

Schemas are automatically invoked by the server when loading documents (for conducting content repair) and when evaluating queries (for proper data typing). If you plan to use schemas in your content loading strategy, review the information in the Loading Schemas chapter in the *Application Developer's Guide*.

1.3.3 Fragments

When loading data into a database, you have the option of specifying how XML documents are partitioned for storage into smaller blocks of information called fragments. For large XML documents, size can be an issue, and using fragments may help manage performance of your system. For a discussion of fragments, see Fragments in the *Administrator's Guide*.

1.3.4 Indexing

Before loading documents into a database, you have the option of specifying a number of parameters that impact how the text components of those documents are indexed. These settings can affect query performance and disk usage. For details, see Text Indexing in the *Administrator's Guide*.

2.0 Controlling Document Format

Each document in a MarkLogic Server database has a *format* associated with it. The format is based on the root node of the document. 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 replace the root node of a document to change its format in a number of ways, including reloading the document while 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.

Documents loaded into a MarkLogic Server database in JSON, XML, or text format are always stored in UTF-8 encoding. Documents loaded in JSON, XML, or text format must either already be in UTF-8 encoding or the UTF-8 encoding must be explicitly specified during loading using options available in the load APIs. For example, you might use the encoding option of the `xdmp:document-load` function. For more details, see Encodings and Collations in the *Search Developer's Guide*.

The following topics are included:

- Supported Document Formats
- Choosing a Binary Format
- Implicitly Setting the Format Based on the MIME Type
- Explicitly Setting the Format
- Determining the Format of a Document

2.1 Terminology

The following terms are used in this topic.

Term	Definition
document format	Refers to how documents are stored in MarkLogic databases: JSON, XML, binary, or text format.
QName	QName stands for qualified name and defines a valid identifier for elements and attributes. QNames are used to reference particular elements or attributes within XML documents.
small binary document	A binary document stored in a MarkLogic database whose size does not exceed the large size threshold. For details, see "Choosing a Binary Format" on page 12.

Term	Definition
large binary document	A binary document stored in a MarkLogic database whose size exceeds the large size threshold. For details, see “Choosing a Binary Format” on page 12.
external binary document	A binary document that is not stored in a MarkLogic database and whose contents are not managed by the server. For details, see “Choosing a Binary Format” on page 12.
CLOB	Character large object documents, or text documents.
BLOB	Binary large object documents, binary data stored as a single entity. Typically images, audio, or other multimedia object.

2.2 Supported Document Formats

MarkLogic supports the following document formats:

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

2.2.1 JSON Format

Documents that are in JSON format have special characteristics that enable 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 perform fine-grained search operations, including property-level search.

JSON documents are indexed when they are loaded. The indexing speeds up query response time. The type of indexing is controlled by the configuration options set on your document’s destination database. JSON documents are a single fragment, and the maximum size of a fragment (and therefore of a JSON document) is 512 MB for 64-bit machines.

2.2.2 XML Format

Documents that are in XML format have special characteristics that enable 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 perform fine-grained search operations, including element-level search.

XML documents are indexed when they are loaded. The indexing speeds up query response time. The type of indexing is controlled by the configuration options set on your document's destination database. One technique for loading extremely large XML documents is to fragment the documents using various elements in the XML structure. The maximum size of a single XML fragment is 512 MB for 64-bit machines. For more details about fragmenting documents, see Fragments in the *Administrator's Guide*.

2.2.3 Binary Format

Binary 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 files (`.doc` and `.docx`), executable program files, and so on.

Binary documents are not indexed when they are loaded.

MarkLogic Server supports three kinds of binary documents: small, large (BLOBs), and external. Applications use the same interfaces to read all three kinds of binary documents, but they are stored and loaded differently. These differences may lead to tradeoffs in access times, memory requirements, and disk consumption. For more details, see "Choosing a Binary Format" on page 12.

For a discussion of the sizing and configuration options to consider when working with binary content, see Configuring MarkLogic Server for Binary Content in the *Application Developer's Guide*.

2.2.4 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 or JSON 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 or non-JSON text file.

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 database text-indexing settings apply to text documents (as well as JSON and XML documents), and MarkLogic creates the indexes when the text document is loaded.

2.3 Choosing a Binary Format

Binary documents require special consideration because they are often much larger than text, JSON, or XML content. MarkLogic Server supports three types of binary documents: *small*, *large*, and *external*. Applications use the same interfaces to read all three types of binary document, but they are stored and loaded differently. A database may contain any combination of small, large, and external binaries. Choose the format that best matches the needs of your application and the capacity of your system. The size threshold that defines small and large binary objects is configurable. For details, see Selecting a Location For Binary Content in the *Application Developer's Guide*.

The following table summarizes attributes you should consider when organizing binary content:

Binary Type	Managed By MarkLogic Server	Stored In	Considerations
Small	Yes	Stands	• Fully cached for faster access • Entire contents may be cached in memory when accessed • Size and quantity constrained by available memory • Best suited for small frequently accessed content, such as thumbnails, profile photos, and icons
Large	Yes	Large Data Directory	• Access times similar to file system reads • Cached in compressed chunks for efficient resource utilization. • Streams documents into and out of the database • Size and quantity limited only by disk space and system file size limit • Best suited for movies, music, and high definition images
External	No	File system	• Access times similar to file system reads • Cached in compressed chunks for efficient resource utilization. • Streams documents into and out of the database • Size and quantity limited only by disk space and system file size limit • External contents do not participate in transactions, backups, or replication • Best suited for read-only content managed external to MarkLogic Server

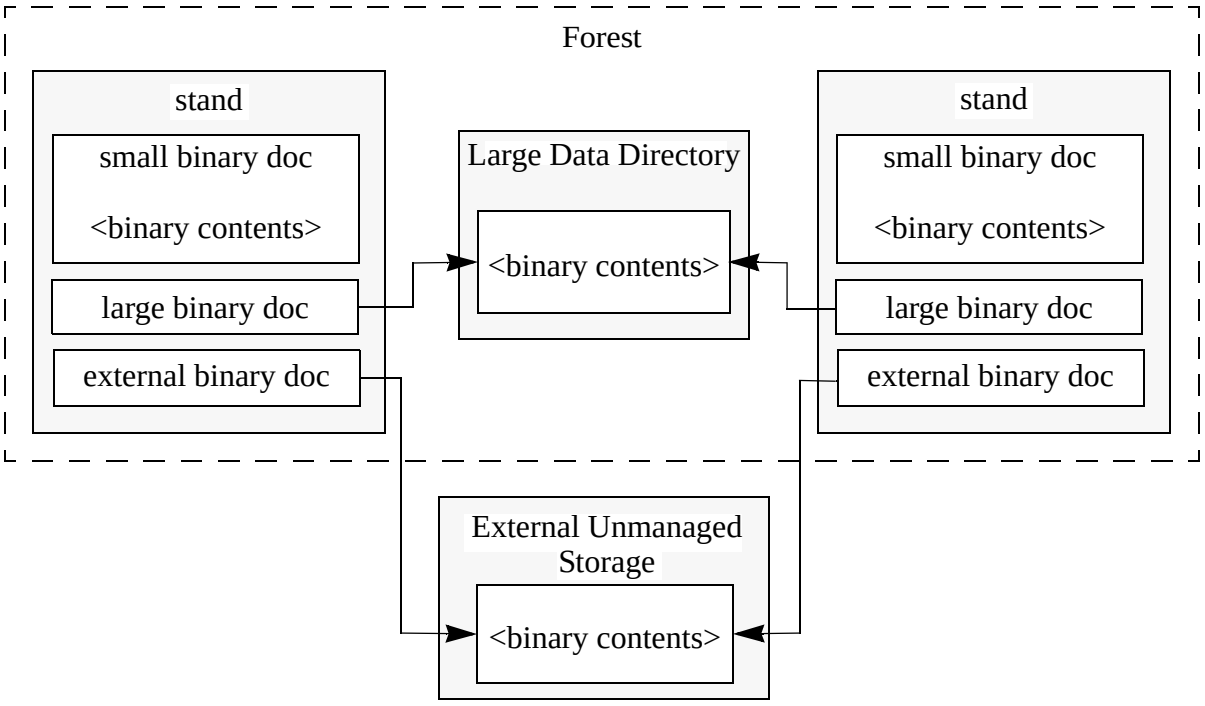
Small and large binary documents are stored in a MarkLogic database and are fully managed by MarkLogic Server. These documents fully participate in transactions, backup, and replication. Small binaries are stored directly in the stands of a forest, which means they are cached in memory. Large binaries are stored in a special Large Data Directory, with only a small reference object in the stand. The data directory containing large binary documents is located inside the forest by default. The location is configurable during forest creation. For more details, see Selecting a Location For Binary Content.

MarkLogic stores small and large binaries differently in the database to optimize resource utilization. For example, if multiple stands contain the same large binary document, only the reference fragment is duplicated. Similarly, if a new large binary document is created from a segment of an existing binary document using `xdmp:subbinary`, a new reference fragment is created, but the binary content is not duplicated. For details about stands, see Understanding Forests in the *Administrator's Guide*.

MarkLogic Server does not fully manage external binary documents because the documents are not stored in the database. The MarkLogic database contains only a small reference fragment to each external file. MarkLogic Server manages the reference fragments as usual, but does not manage the external files. For example, MarkLogic Server does not replicate or back up the external files. You must provide security, integrity, and persistence of the external files using other means, such as the underlying operating system or file system.

Large and external binary documents require little additional disk space for merges. During a merge, MarkLogic copies fragments from the old stands to a new merged stand, as described in Understanding and Controlling Database Merges in the *Administrator's Guide*. The small reference fragments of large and external binaries contribute little overhead to the merge process. The referenced binary contents are not copied during a merge.

The following diagram shows the differences in small, large, and external binaries handling. Although multiple stands may contain references fragments for the same large or external binary document, only the reference fragment is duplicated:



2.3.1 Loading Binary Documents

Loading small and large binary documents into a MarkLogic database does not require special handling, other than potentially explicitly setting the document format. Use the standard methods, such as XQuery functions or other interfaces.

External binaries require special handling at load time because they are not managed by MarkLogic. For details, see "Creating External Binary References Using XQuery" on page 26.

2.3.2 Configuring MarkLogic Server for Binary Documents

Before loading binary content, you should carefully consider the sizing and scalability implications of binary documents and configure the server appropriately. For details, see Configuring MarkLogic Server for Binary Content in the *Application Developer's Guide*.

2.4 Implicitly Setting the Format Based on the MIME Type

Unless the format is explicitly set when you load a document, the format of the document is determined based on the MIME type that corresponds to the URI extension of the new document. The URI extension MIME types, along with their default formats, are set in the Mimetypes section of the Admin Interface.

For example, with the default MIME type settings, documents loaded with the `xml` URI extension are loaded as XML files; therefore loading a document with a URI `/path/doc.xml` results in loading an XML document. The following table contains examples of applying the default MIME type mappings to output URIs with various file extensions. Many additional mappings are configured by default.

URI	Document Type
`/path/doc.json`	JSON
`/path/doc.xml`	XML
`/path/doc.jpg`	binary
`/path/doc.txt`	text

You can also use the Mimetypes configuration page of the Admin Interface to modify any of the default content setting, create new MIME types, 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.

2.5 Explicitly Setting the Format

When you load a document, you can explicitly set the format. Explicitly setting the format overrides the default settings specified on the Mimetypes configuration screen in the Admin Interface. 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 rather than relying on implicit format settings based on the MIME types because it gives you complete control over the format and eliminates surprises based on implicit MIME type mappings.

The following table summarizes the mechanisms available for explicitly setting the document format during loading. For details, see the MarkLogic interface-specific documentation.

Interface	Summary	For More Details
Content Pump (mlcp)	Set the `-document_type` import option	Importing Content Into MarkLogic Server in the *mlcp User Guide*.
Java API	`ContentDescriptor` interface.	*Java Application Developer's Guide*.
MarkLogic Connector for Hadoop	`ContentOutputFormat` class	*MarkLogic Connector for Hadoop Developer's Guide* and javadoc.
REST API	Set the `format` parameter or `Content-type` header on a PUT or POST request to the `/documents` service.	Loading Content into the Database in *REST Application Developer's Guide*, and the API reference for `PUT /v1/documents` or `POST /v1/documents` in the *REST Resources API*.
XCC	Set the format in the `ContentCreateOptions` class.	XCC Javadoc.
XQuery	Specify a value for the `<format>` element of the `<options>` node passed to `xdmp:document-load`.	The API documentation for `xdmp:document-load` in the *MarkLogic XQuery and XSLT Function Reference*.

The following XQuery example demonstrates explicitly setting the format to XML when using `xdmp:document-load`:

```
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>)
```

2.6 Determining the Format of a Document

After a document is loaded into a database, you cannot assume the URI accurately reflects the content format. For example, a document can be loaded as XML even if it has a URI that ends in `.txt`. To determine the format of a document in a database, perform 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"
```

3.0 Specifying Encoding and Language

You can specify the encoding and default language while loading a document. You can also automatically detect the encoding or manually detect the language (for example, using `xdmp:encoding-language-detect`). This section describes how to load documents with a specific encoding or language, and includes the following parts:

- Understanding Character Encoding
- Explicitly Specifying Character Encoding While Loading
- Automatically Detecting the Encoding
- Inferring the Language and Encoding of a Node in XQuery with xdmp:encoding-language-detect
- Specifying the Default Language for XML Documents

For more information about languages, see Language Support in MarkLogic Server in the *Search Developer's Guide*.

3.1 Understanding Character Encoding

MarkLogic Server stores all content in the UTF-8 encoding. If you try to load non-UTF-8 content into MarkLogic Server without translating it to UTF-8, the server throws an exception. If you have non-UTF-8 content, then you can specify the encoding for the content during ingestion, and MarkLogic Server will translate it to UTF-8. If the content cannot be translated, MarkLogic Server throws an exception indicating that there is non-UTF-8 content.

You can specify the encoding for content using either an encoding option on the ingestion function or via HTTP headers. For details, see Character Encoding in the *Search Developer's Guide*.

3.2 Explicitly Specifying Character Encoding While Loading

The table below summarizes the mechanisms available for explicitly specifying character encoding. See the interface-specific documentation for details. If no encoding is specified, MarkLogic Server defaults to UTF-8 for all non-binary documents.

Interface	Method	For Details See
MarkLogic Connector for Hadoop	The configuration property `mapreduce.marklogic.output.content.encoding`.	Configuring a MapReduce Job in the *MarkLogic Connector for Hadoop Developer's Guide* `MarkLogicConstants` in the *MarkLogic Hadoop MapReduce Connector API*
MarkLogic Content Pump (mlcp)	Character encoding cannot be controlled. Only UTF-8 is supported.	"Loading Content Using MarkLogic Content Pump" on page 29.
MarkLogic Java API	Various handles.	Conversion of Document Encoding in the *Java Application Developer's Guide*
MarkLogic REST API	The `charset` parameter of the HTTP `Content-type` header. However, Text, XML and JSON content must be UTF-8 encoded.	*REST Application Developer's Guide*
XCC	Java: `ContentCreateOptions.setEncoding` XCC: `Encoding` property of the `ContentCreateOptions` class	Javadoc for XCC dotnet for XCC (C# API)
XQuery	The `encoding` element of the `options` parameter to `xdmp:document-load`, `xdmp:document-get`, `xdmp:zip-get`, and `xdmp:http-get`.	*XQuery and XSLT Reference Guide*

The following XQuery example loads the document using the ISO-8859-1 encoding, transcoding the content from ISO-8859-1 to UTF-8 during the load:

```
xdmp:document-load("c:/tmp/my-document.xml",
  <options xmlns="xdmp:document-load">
    <uri>/my-document.xml</uri>
```

```
    <encoding>ISO-8859-1</encoding>
  </options>)
```

3.3 Automatically Detecting the Encoding

For those interfaces that support auto-detection of encoding, MarkLogic Server attempts to automatically detect the encoding of non-binary content during loading if the explicitly specified encoding is `auto`.

The automatic encoding detection chooses an encoding equivalent to the first encoding returned by the `xdmp:encoding-language-detect` XQuery function. Encoding detection is not an exact science. There are cases where content encoding is ambiguous, but as long as your document is not too small, the encoding detection is fairly accurate. There are, however, cases where auto-detect might choose the wrong encoding.

The following XQuery example demonstrates using automatic character encoding detection when loading a document using `xdmp:document-load`:

```
xdmp:document-load("c:/tmp/my-document.xml",
  <options xmlns="xdmp:document-load">
    <uri>/my-document.xml</uri>
    <encoding>auto</encoding>
  </options>)
```

For details, see the interface specific documentation or “Explicitly Specifying Character Encoding While Loading” on page 19.

3.4 Inferring the Language and Encoding of a Node in XQuery with xdmp:encoding-language-detect

If you do not want to rely on the automatic detection for the encoding or if you want to detect the language, you can use the `xdmp:encoding-language-detect` function. The `xdmp:encoding-language-detect` function returns XML elements, each of which specifies a possible encoding and language for the specified node. Each element also has a score, and the one with the highest score (the first element returned) has the most likely encoding and language.

```
xdmp:encoding-language-detect(
  xdmp:document-get("c:/tmp/session-login.css"))
=>
<encoding-language xmlns="xdmp:encoding-language-detect">
  <encoding>utf-8</encoding>
  <language>en</language>
  <score>14.91</score>
</encoding-language>
<encoding-language xmlns="xdmp:encoding-language-detect">
  <encoding>utf-8</encoding>
  <language>ro</language>
  <score>13.47</score>
</encoding-language>
<encoding-language xmlns="xdmp:encoding-language-detect">
```

```
  <encoding>utf-8</encoding>
  <language>it</language>
  <score>12.84</score>
</encoding-language>
<encoding-language xmlns="xdmp:encoding-language-detect">
  <encoding>utf-8</encoding>
  <language>fr</language>
  <score>12.71</score>
</encoding-language>
...
```

The encoding detection is typically fairly accurate when the score is greater than 10. The language detection tends to be less accurate, however, because it can be difficult to detect the difference between some languages. Because it gives you the raw data, you can use the output from `xdmp:encoding-language-detect` with whatever logic you want to determine the language. For example, if you happen to know, based on your knowlege of the content, that the language is either Italian or Spanish, you can ignore entries for other languages.

Sometimes the language or the encoding of a block of text is ambiguous, therefore detecting languages and encodings is sometimes prone to error. As a rule, the larger the block of text, the higher the accuracy of the detection. If the size of the block of text you pass into xdmp:encoding-language-detect is more than a few paragraphs of text (several hundred bytes), then the detection is typically fairly accurate.

3.5 Specifying the Default Language for XML Documents

The formal or natural language of XML content is determined by the element attribute `xml:lang`. The language affects how MarkLogic Server tokenizes content, and therefore affects searching and indexing.

When there is no explicit `xml:lang` attribute on an XML document when it is loaded, MarkLogic Server uses the configured default language for the database. Set the database-wide default language through the `language` setting in the Admin UI.

You can override the configured default language for the database using load options, as shown by the table below:

Interface	Method	For Details See
MarkLogic Connector for Hadoop	The configuration property `mapreduce.marklogic.output.content.language`.	Configuring a MapReduce Job in the *MarkLogic Connector for Hadoop Developer's Guide* `MarkLogicConstants` in the *MarkLogic Hadoop MapReduce Connector API*
MarkLogic Content Pump	`-output_language` command line option	Importing Content Into MarkLogic Server in the *mlcp User Guide*
XCC	Java: `ContentCreateOptions.setLanguage` .NET: `Encoding` property of the `ContentCreateOptions` class.	Javadoc for XCC dotnet for XCC (C# API)
XQuery	Set the `<default-language>` element of the `<options>` node passed to `xdmp:document-load`, `xdmp:document-get`, `xdmp:http-get`, or `xdmp:zip-get`.	*XQuery and XSLT Reference Guide*

For details on languages, see Language Support in MarkLogic Server in the *Search Developer's Guide*.

4.0 Loading Content Using XQuery

This chapter describes the XQuery interface for loading content and includes the following sections:

- Built-In Document Loading Functions
- Specifying a Forest in Which to Load a Document
- Creating External Binary References Using XQuery

4.1 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:document-get` and `xdmp:document-insert` is equivalent to `xdmp:document-load` of a new document.

Note: You may only load external binary documents using `xdmp:document-insert` of a constructed `external-binary` node. For details, see “Creating External Binary References Using XQuery” on page 26.

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 *XQuery and XSLT Reference Guide* for a more detailed syntax description.

4.2 Specifying a Forest in Which to Load a Document

In most situations, MarkLogic Server does a good job of determining which forest to put a document, and in general you should not need to override the defaults. When loading a document, however, 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 by explicitly specifying a forest key, 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. Additionally, the `locking` parameter must be set to `strict` on the database configuration, otherwise an XDMP-PLACEKEYSLOCKING exception is thrown.

This section describes some aspects of forest-specific loading and includes the following parts:

- Consider If You Really Want to Specify a Forest
- Some Potential Advantages of Specifying a Forest
- Example: Examining a Document to Decide Which Forest to Specify
- More Examples

4.2.1 Consider If You Really Want to Specify a Forest

For most applications, you should not specify the forest in which you want to load a document. MarkLogic Server has efficient ways of determining which forest to load a document, and those ways are almost always better than explicitly specifying the forest. The default way MarkLogic spreads documents across forests is optimized for both query and loading efficiency. If you are using Tiered Storage (for details, see Tiered Storage), it has its own way of partitioning documents that you should follow.

If you really want to specify the forest to which you load a document, the following describes some details about forest-specific loading.

4.2.2 Some Potential 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.

4.2.3 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.

4.2.4 More Examples

The following command loads 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 loads 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> )
```

4.3 Creating External Binary References Using XQuery

An *external binary node* is a special reference to a binary file managed and stored in the file system separately from MarkLogic Server. You can create an external binary node in MarkLogic and insert the node in the database, creating an external binary reference document. The external binary reference document acts like a normal binary document, except that MarkLogic never actually stores the binary data internally, and instead transparently accesses the external file every time the document is accessed. Unlike normal binary documents, you do not use `xdmp:document-load` to insert an external binary reference document in the database. To insert an external binary reference document into the database, you first create a binary node using the `xdmp:external-binary` function and then insert the node into the database using `xdmp:document-insert`.

For example, the following code creates a document representing the external binary file `/external/path/sample.jpg`, beginning at offset 1 in the file, with a length of 1M:

```
xdmp:document-insert("/docs/xbin/sample.jpg",
    xdmp:external-binary(
        "/external/path/sample.jpg", 1,1024000))
```

When you provide a length to `xdmp:external-binary`, MarkLogic Server does not verify the existence or size of the external file. If you omit a length when calling `xdmp:external-binary`, the underlying external file must exist, and MarkLogic Server calculates the length in a manner equivalent to calling `xdmp:filesystem-file-length`.

5.0 Loading Content Using REST, Java or Node.js

You can use the Node.js Client API, Java Client API, or REST Client API to load content into MarkLogic Server from a remote host. You do not need to understand XQuery to use these interfaces. For details, see the following guides:

- *Node.js Application Developer's Guide*
- *Java Application Developer's Guide*
- *REST Application Developer's Guide*

6.0 Loading Content Using MarkLogic Content Pump

MarkLogic Content Pump (mlcp) is a command line tool for getting data into and out of a MarkLogic Server database. Using mlcp, you can import documents and metadata to a database, export documents and metadata from a database, or copy documents and metadata from one database to another.

The tool supports a variety of input formats, including flat files and compressed files, on the native file system or on HDFS. You can also use mlcp with Hadoop to load large amounts of content distributed across a Hadoop cluster.

For details, see the *mlcp User Guide*.

7.0 Loading Content Using WebDAV

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. There are a number of WebDAV clients available for various platforms and information can be found by searching the Internet for WebDAV clients. Some MarkLogic users have had good success with BitKinex and with NetDrive from Novell.

For details on setting up MarkLogic WebDAV servers, see WebDAV Servers in the *Administrator's Guide*.

Directories are required for WebDAV clients to see documents. See Directories and WebDAV Servers in the *Application Developer's Guide*.

For an example of using a WebDAV client with the Default Conversion Option (a CPF example application), see Simple Drag-and-Drop Conversion in the *Content Processing Framework Guide*.

8.0 Repairing XML Content During Loading

MarkLogic Server can perform the following types of XML content repair during content loading:

- Correct content that does not conform to the well-formedness rules in the XML specification
- Modify inconsistently structured content according to a specific XML schema
- Assign namespaces and correct unresolved namespace bindings
- Restructure content using XPath or XQuery

Not all programming language interfaces support the full spectrum of XML content repair. MarkLogic Server does not validate content against predetermined XSchema (DDML) or DTDs.

This chapter includes the following topics:

- Programming Interfaces and Supported Content Repair Capabilities
- Enabling Content Repair
- Auto-Close Repair of Empty Tags
- Schema-Driven Tag Repair
- Load-Time Default Namespace Assignment
- Load-Time Namespace Prefix Binding
- Query-Driven Content Repair

8.1 Programming Interfaces and Supported Content Repair Capabilities

The MarkLogic programming interfaces support repair options as described in the following table:

Programming Interface	Content Repair Capabilities	More Details
MarkLogic Connector for Hadoop	Set the repair level.	*MarkLogic Connector for Hadoop Developer's Guide*
MarkLogic Content Pump	Tag repair and schema-driven repair, namespace prefix binding.	`-xml_repair_level` option. See Importing Content Into MarkLogic Server in the *mlcp User Guide*
MarkLogic Java API	Tag repair, schema-driven repair, namespace prefix binding.	*Java Application Developer's Guide*
REST Client API	General-purpose tag repair and schema-driven repair, namespace prefix binding.	`repair` parameter on `PUT:/v1/documents` in the REST Client API
XCC	General-purpose tag repair and schema-driven repair, namespace prefix binding.	`DocumentRepairLevel` enumeration class
XQuery	All types described in this chapter.	`<repair>` parameter in the options node of `xdmp:document-load`. Also see the *MarkLogic XQuery and XSLT Function Reference*

8.2 Enabling Content Repair

The tag repair, schema-driven repair, and namespace prefix binding mechanisms are enabled using an option to the various content loading functions as listed above.

When no repair option is explicitly specified, the default is implicitly specified by the XQuery version of the caller. In XQuery 1.0 and 1.0-ml the default is `none`. In XQuery 0.9-ml the default is `full`.

Tag repair, schema-driven repair, and namespace prefix binding can be performed on all XML documents loaded from external sources. This includes documents loaded using the XQuery built-in functions, XCC document insertion methods, or the Java or REST client APIs.

8.3 General-Purpose Tag Repair

MarkLogic Server can apply a general-purpose, stack-driven tag repair algorithm to every XML document loaded from an external source. The algorithm is triggered by encountering a closing tag (for example, `</tag>`) that does not match the most recent opening tag on the stack.

8.3.1 How General-Purpose Tag Repair Works

Consider the following simple document markup example:

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

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

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

  Consequently, in the first sample tagging error shown above, the loader automatically closes 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.

- **Rule 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 tagging error shown above, the loader ignores the "extra" underline closing tag and proceeds as if it is 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>
```

8.3.2 Pitfalls of General-Purpose Tag Repair

While these two general repair rules produce sound results in most situations, their application can lead to repairs that may not match the original intent. Consider the following 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 fixes 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 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 where the tag should have been closed. In fact, for cases other than this simple example, even a human is not always 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 markup 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 preceding examples. MarkLogic offers additional content repair capabilities that can be used to repair a wider range of problems, including the examples above. These advanced content repair techniques are described in the following sections.

8.3.3 Limitations

This section describes some known limitations of general-purpose tag repair.

8.3.3.1 XQuery Functions

For functions where 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 (in which case the query is not successfully parsed for execution unless it is well-formed), general-purpose tag repair is not performed. This includes XML content loaded using the following functions:

- `xdmp:document-insert`
- `xdmp:node-replace`
- `xdmp:node-insert-before`
- `xdmp:node-insert-after`
- `xdmp:node-insert-child`

8.3.3.2 Root Element

General-purpose tag repair does not insert a missing closing root element tag into an XML document.

8.3.3.3 Previous Marklogic Versions

Versions of MarkLogic Server 2.0 and earlier would repair missing root elements, making it effectively impossible to identify truncated source content. Later versions of MarkLogic Server reports an error in these conditions.

8.3.4 Controlling General-Purpose Tag Repair

MarkLogic Server enables you to enable or disable general-purpose tag repair during any individual document load using an optional repair parameter. The specific parameter is language specific. For example, if you use XQuery `xdmp:document-load`, `xdmp:unquote` functions, you can use the `repair` parameter on the options node and specify a value of `full` or `none`. See the language specific documentation for more details.

8.4 Auto-Close Repair of Empty Tags

Empty tag auto-close is a special case of schema-driven tag repair and is supported in all versions of MarkLogic Server. This repair mechanism automatically closes tags that are identified as empty tags in a specially-constructed XML 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 acceptable to use a tag as an indication of some formatting directive, without any need to close the tag. This frequently results in the liberal use of empty tags within SGML and HTML content.

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 rarely find a `</hr>` tag or even a `<hr/>` tag unless someone has converted the HTML document to be XHTML-compliant. The same can occur with `<img>` and `<meta>` tags, to name just two. In SGML documents, you can easily find `<pgbrk>`, `<xref>` and `<graphic>` used similarly.

Applying this type of content repair enables you to avoid the false nesting of content within otherwise unclosed empty tags.

8.4.1 What Empty Tag Auto-Close Repair Does

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. Working under default settings, MarkLogic Server views each of these two tags as opening tags that at some point later in the document will be closed, and consequently incorrectly views the following content as children of those tags. This results 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 is processed by a query that specifies an XPath such as `/doc/para`. The first paragraph matches this XPath, but the second does not, because it has been loaded incorrectly as the child of a `pgbrk` element. While alternative XPath expressions such as `/doc//para` gloss over this difference, it is better 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>
```

8.4.2 Defining a Schema to Support Empty Tag Auto-Close Repair

To use empty tag auto-close repair, you first define an XML schema that specifies which tags should be assumed to be empty tags. Using this information, when MarkLogic Server is loading content from an external source, it automatically closes these tags as soon as they are encountered. If some of the specified tags are, in fact, accompanied by closing tags, these closing tags are 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 is loaded under the control of this schema, it is repaired correctly.

To use XML 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.

8.4.3 Invoking Empty Tag Auto-Close Repair

There are multiple ways to invoke the empty tag auto-close functionality. The recommended procedure is the following:

1. Write an XML schema that specifies which tags should be treated as empty tags. The schema shown in the preceding section, "Defining a Schema to Support Empty Tag Auto-Close Repair" on page 37, is a good starting point.

2. Load the schema into MarkLogic. See Loading Schemas in the *Application Developer's Guide* for instructions.

3. Make sure that the content to be loaded references the namespace of the applicable schema that you have loaded into MarkLogic. For the schema shown above, the document's 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 a matching schema whenever a document is loaded.

   In the second form, 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>
```

This example 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 is used.

See Loading Schemas in the *Application Developer's Guide* for an in-depth discussion of the precedence rules that are applied in the event that multiple matching schemas are found.

4. Load the content using `xdmp:document-load` or one of the other language interface document insertion methods.

After the content is loaded, you can inspect it to see that the content repair was performed. If empty tag auto-close repair was not applied, then you should troubleshoot the location, naming and cross-referencing of your schema, as this is the most likely source of the problem.

When it is not feasible to modify your content so that it properly references a namespace in its root element, there are other approaches that can yield the same result:

1. Write an XMLschema 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, remembering the URI name under which you loaded the schema. See Loading Schemas in the *Application Developer's Guide* 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.

   - A simple example of importing a schema into the unnamed namespace might look like the following:

```
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 are filtered through the "empty tag auto close" repair 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 fails silently.

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

8.4.4 Scope of Application

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

8.4.5 Disabling Empty Tag Auto-Close

There are several ways to disable load-time empty tag auto-close repair:

1. Disable content repair at load-time using the applicable option for your chosen language interface.

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 does 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.

8.5 Schema-Driven Tag Repair

MarkLogic Server supports the use of XML schemas for more complex schema-driven tag repair. This enables you to use XML schemas to define a set of general rules that govern how various elements interact hierarchically within an XML document.

8.5.1 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 repairs and loads 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 you to indicate constraints in the relationships between elements by using an XML schema. In this case, you can indicate that a `<section>` element may only contain `<para>` elements. Therefore, a `<section>` element cannot be a child of another `<section>` element. In addition, you can indicate that `<para>` element is a simple type that only contains text. Using the schema, MarkLogic 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 is 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 an XML schema that describes the constraints on the relationships between elements. Using this information, when tMarkLogic Server loads content from an external source, it automatically closes 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 only occurs 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 is loaded under the control of this simple schema, it is corrected as specified.

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.

8.5.2 How to Invoke Schema-Driven Tag Repair

There are multiple ways to do schema-driven correction. The recommended procedure is the following:

1. Write an XML schema that describes the relationships between the elements.

2. Load the schema into MarkLogic Server. See Loading Schemas in the *Application Builder Developer's Guide* for instructions.

3. In the content that you need to load, ensure that the root element properly references the appropriate schema. See "Invoking Empty Tag Auto-Close Repair" on page 38 for examples of referencing the XML schema from inside the content.

4. Load the content using `xdmp:document-load` or any of the other available document insertion methods.

After the content is loaded, you can inspect it to see that the content repair was performed. If the appropriate content repair did not occur, 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 XML schema in its root element, there are other approaches that can yield the same result:

1. Write 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 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 are 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 fails silently.

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

8.5.3 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 is automatically repaired as directed by that schema.

8.5.4 Disabling Schema-Driven Tag Repair

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

1. Disable content repair at load-time using the appropriate parameter for your chosen content loading mechanism.

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

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

Removing the schema from the database does 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.

8.6 Load-Time Default Namespace Assignment

When documents are loaded into MarkLogic, every element is stored with a 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 , MarkLogic applies the default unnamed namespace to all the nodes that do not have an associated namespace.

In some situations this is not the desired result. Once the document is loaded without a specified namespace, it is difficult to remap each QName to a different namespace. It is better to load the document into MarkLogic Server with the correct default namespace in the first place.

The best way to specify a default namespace for a document is to add a default namespace attribute to the document's root node directly. When that is not possible, MarkLogic's load-time namespace substitution capability offers a good solution. If you are using XQuery or XCC for your document loading, 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.

Note: The REST and Java client APIs do not provide a default namespace option. When you use these APIs for your document loading, it is best to add the appropriate namespace attribute to your documents before loading them to the database.

8.6.1 How Default Namespace Assignments Work

The `xdmp:document-load` function and the XCC `setNamespace` method (in the `ContentCreateOptions` class) allow you to optionally specify a namespace as the default namespace for an individual document loading operation.

MarkLogic uses that namespace definition as follows:

Rule 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.

Rule 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

8.6.2 Scope of Application

You can specify default namespaces at load-time when you use XQuery or XCC to load content. See the corresponding documentation for further details.

8.7 Load-Time Namespace Prefix Binding

The original XML specifications allow the use of colons in element names, for example, `<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 dropped 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` is 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` is 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 returns an error if unresolved namespace prefixes are encountered at load time.

8.7.1 How Load-Time Namespace Prefix Binding Works

If content repair is enabled, MarkLogic can 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:

Rule 1: When the prefix is specified in the document, that binding is retained. In the following example, 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>
```

Rule 2: When the prefix is declared in the XQuery environment, that binding is used. For example, 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")
```

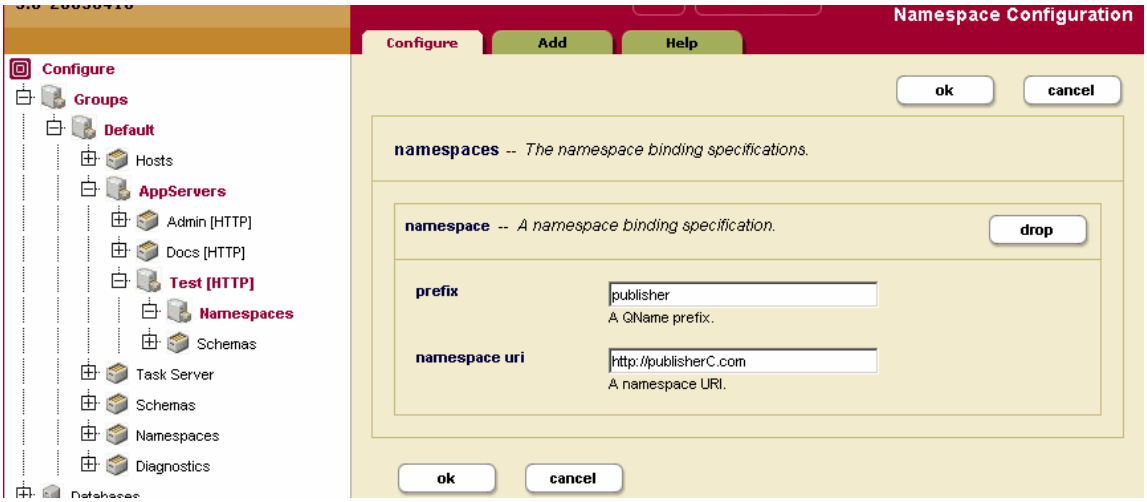
The code snippet loads the `mybook.xml` as:

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

Note: This rule only applies in the XQuery environment.

Rule 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 is loaded as:

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

Rule 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` is loaded as:

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

8.7.2 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 is 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 is 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 is ignored. Assume that publisher resolves to `publisher`. The document is 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.

8.7.3 Scope of Application

If content repair is enabled, MarkLogic attempts to create bindings for unresolved namespace prefixes as a form of content repair for all documents loaded from external sources according to the rules described in “How Load-Time Namespace Prefix Binding Works” on page 47.

8.7.4 Disabling Load-Time Namespace Prefix Binding

MarkLogic enables you to disable content repair during any individual document load using a language specific repair parameter. See “Programming Interfaces and Supported Content Repair Capabilities” on page 32.

8.8 Query-Driven Content Repair

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, MarkLogic's schema-independent core makes XQuery itself a powerful content repair mechanism.

Once a document is loaded into MarkLogic, queries can be written to specifically restructure the content as required, without needing to reconfigure the database. Two approaches to query-driven content repair -- point repair and document walkers -- are described in the following sections. If you want to do something similar from other languages, use a transformation, a feature of the REST and Java client API's that lets you install XQuery or XSLT that you can use during document loading and retrieval.

8.8.1 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 example 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 generates an error.

8.8.2 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, this function can transform both the structure and the content. 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 enables 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 in the *Application Developer's Guide*.

9.0 Modifying Content During Loading

Content can go through many stages before it is ready to use in an application. These stages might include modifying the content so that it is well-formed XML, transforming one XML structure to another, or combining the content with other content or information. The process of content going from one stage to another is called *content processing*.

Content processing can be very simple or extremely complex. You might decide to add a timestamp to a document and define a content processing stage to add the timestamp. You might have a process that translates the text from one language to another. Often, many of these stages combined together form an overall set of content processing work you need to do on a document.

While the range of problems that can be addressed is virtually unlimited, there are several core content processing capabilities required to address many of the wide-ranging issues:

- The ability to change the content from one form to another.
- The ability to tie together different pieces of content processing.
- The ability to separate different documents for different types of processing.
- The ability to automate the entire procedure so documents can move through complex processing phases automatically.
- The ability to integrate manual steps or long-running, asynchronous operations in applications.

Flexibility is important in content processing, as both the starting points of documents and their end results can vary significantly. Also, application requirements can evolve over time, forcing the content processing application to change with the requirements. It is therefore necessary to have a content processing environment that can allow for such change.

MarkLogic Server provides capabilities to modify content with workflows and pipelines. An example of a content processing application is The Default Conversion Option, which uses the components of the MarkLogic Content Processing Framework, and XQuery modules, to create a unified conversion process that converts Microsoft Office, Adobe PDF, and HTML files to well-structured XHTML and simplified DocBook format XML documents.

9.1 Converting Microsoft Office and Adobe PDF Into XML

The Default Conversion Option of the Content Processing Framework converts Microsoft Office, Adobe PDF, and HTML files to XHTML and DocBook. The Default Conversion Option only converts Microsoft Office 97 and newer documents; it cannot convert documents from Microsoft Office 95 or earlier.

9.2 Converting to XHTML

MarkLogic provides facilities for converting documents to XHTML as follows:

- `xdmp:tidy` converts HTML to XHTML
- Default Conversion Option of the Content Processing Framework converts Microsoft Office, PDF, and HTML files to XHTML
- `xdmp:pdf-convert` converts a PDF file to XHTML
- `xdmp:excel-convert` converts a Microsoft Excel document to XHTML

9.3 Automating Metadata Extraction

MarkLogic provides facilities to extract and associate metadata from binary documents as follows:

- `xdmp:document-filter`, a built-in XQuery function
- MarkLogic content pump provides commands to include or exclude metadata during copy, export, and import

9.4 Transforming XML Structures

A common task sometimes required with XML is to transform one structure to another structure. A design pattern using the XQuery typeswitch expression to transform XML to XHTML or XSL-FO is described in Transforming XML Structures With a Recursive typeswitch Expression in the *Application Developer's Guide*.

10.0 Performance Considerations

This chapter covers the following topics:

- Understanding the Locking and Journaling Database Settings for Bulk Loads
- Fragmentation

10.1 Understanding the Locking and Journaling Database Settings for Bulk Loads

When you load content, MarkLogic Server performs updates transactionally, locking documents as needed and saving the content to disk in the journal before the transaction commits. By default, all documents are locked during an update and the journal is set to preserve committed transactions, even if the MarkLogic Server process ends unexpectedly.

The database settings `locking` and `journaling` control how fine-grained and robust you want this transactional process to behave. By default, it is set up to be a good balance of speed and data-integrity. All documents being loaded are locked, making it impossible for another transaction to update the same document being loaded or updated in a different transaction, and making it impossible to create duplicate URIs in your database.

There is a journal write to disk on transaction commit, and by default the system relies on the operating system to perform the disk write. Therefore, even if the MarkLogic Server process ends, the write to the journal occurs, unless the computer crashes before the operating system can perform the disk write. Protecting against the MarkLogic Server process ending unexpectedly is the `fast` setting for the `journaling` option. If you want to protect against the computer crashing unexpectedly, you can set the journaling to `strict`. A setting of `strict` forces a filesystem sync before the transaction is committed. This takes a little longer for each transaction, but protects your transactions against the computer failing.

If you are sure that no other programs are updating content in the database, and if you are sure that your program is not updating a URI more than one time, it is possible to turn the `journaling` and/or `locking` database settings to `off`. Turning these `off` might make sense, for example, during a bulk load. You should only do so if you are sure that no URIs are being updated more than once. Be sure to turn the `directory creation` database setting to `manual` before disabling `locking` in a database, as automatic directory creation creates directories if they do not already exist, and, without locking, can result in duplicate directory URIs in some cases. The default `locking` option of `fast` locks URIs for existing documents, but not for new documents, but this is safe because the system knows where new documents will be placed and therefore does not need locks for new documents, therefore it is both safe and fast.

Warning Use extreme caution when setting these parameters to `off`, as that will disable and limit the transactional checks performed in the database, and doing so without understanding how it works can result in inconsistent data.

The advantage of disabling the `locking` or `journaling` settings is that it makes the loads faster. For bulk loads, where if something goes wrong you can simple start over, this might be a trade-off worth considering.

For more details on how transactions work, see Understanding Transactions in MarkLogic Server.

10.2 Fragmentation

Proper fragmentation is important to performance. Before you specify how to fragment the XML data being loaded, you need to plan your fragmentation strategy. For guidelines on fragmentation, see Choosing a Fragmentation Strategy in the *Administrator's Guide*.

11.0 Technical Support

MarkLogic provides technical support according to the terms detailed in your Software License Agreement or End User License Agreement.

We invite you to visit our support website at http://help.marklogic.com to access information on known and fixed issues, knowledge base articles, and more. For licensed customers with an active maintenance contract, see the Support Handbook for instructions on registering support contacts and on working with the MarkLogic Technical Support team.

Complete product documentation, the latest product release downloads, and other useful information is available for all developers at http://developer.marklogic.com. For general questions, join the general discussion mailing list, open to all MarkLogic developers.

12.0 Copyright

MarkLogic Server 8.0 and supporting products.

NOTICE

Copyright © 2018 MarkLogic Corporation.

This technology is protected by one or more U.S. Patents 7,127,469, 7,171,404, 7,756,858, 7,962,474, 8,935,267, 8,892,599 and 9,092,507.

All MarkLogic software products are protected by United States and international copyright, patent and other intellectual property laws, and incorporate certain third party libraries and components which are subject to the attributions, terms, conditions and disclaimers found at http://docs.marklogic.com/guide/copyright/legal.

MarkLogic and the MarkLogic logo are trademarks or registered trademarks of MarkLogic Corporation in the United States and other countries. All other trademarks are property of their respective owners.

For all copyright notices, including third-party copyright notices, see the Combined Product Notices for your version of MarkLogic.