O'REILLY®

MarkLogic Cookbook

Implementing XQuery: Practical Solutions to Real-World Problems

Part 1

Dave Cassel

Your data deserves better.

There's nothing wrong with your data
that a better database can't fix.

- Rely on a 100% trusted, enterprise-grade multi-model database
- Load your data *as-is*, no upfront data modeling required
- Unify your structured and unstructured data
- Build and deploy your apps in days, not months

www.marklogic.com

MarkLogic®

MarkLogic Cookbook

Implementing XQuery: Practical Solutions to Real-World Problems

David M. Cassel

Beijing • Boston • Farnham • Sebastopol • Tokyo

MarkLogic Cookbook
by Dave Cassel

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editor: Shannon Cutt
Production Editor: Kristen Brown
Copyeditor: Sonia Saruba
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

June 2017: First Edition

Revision History for the First Edition
2017-06-09: Part 1

978-1-491-99458-0

[LSI]

Table of Contents

Foreword

This book comes at MarkLogic from the opposite direction of my own book, *Inside MarkLogic Server* (recently updated by Mike Wooldridge). In my book, I aimed to describe MarkLogic's internals: its data model, indexing system, and operational behaviors. I made the decision to *avoid* getting into how exactly to accomplish specific goals, because to do so would have to be a book of its own.

This is that book!

In *MarkLogic Cookbook*, Dave documents a set of MarkLogic recipes: ways to do common things that can be a bit too tricky to remember without a reference by your side. This first installment covers XQuery. Over time, this book will issue additional installments with more recipes and topics.

What you'll find here today is:

- Getting the best performance
- Manipulating maps with the `map:map` data type
- Viewing security details on documents
- Managing tasks on the Task Server

We hope you enjoy it. If you have your own ideas (favorite tricks!) that you think should be included in future installments, please send them to *recipes@marklogic.com*.

— Jason Hunter
Somewhere over the Pacific Ocean
April 2017

Introduction

MarkLogic is a powerful multi-model database platform with a very broad set of capabilities—all designed to help you integrate data from silos faster. It does take some time to learn how to harness that power, though. The recipes in this book will move you along this process faster—you can learn from others who have taken the time to learn how to get the most out of MarkLogic, and add some of their tools to your toolbelt.

In this, the first volume of a three-part series, we are covering XQuery recipes. For much of MarkLogic's history, XQuery was the primary language used to interact with MarkLogic (more recently, MarkLogic has added support for JavaScript). This W3C-standard functional language is well-suited for working with hierarchical data structures, like XML, which in turn is a descriptive medium for describing document data.

Recipes are a useful way to distill simple solutions to common problems—copy and paste these into MarkLogic's Query Console or your source code, and you've solved the problem. In choosing recipes for this book, I looked for a couple of factors. First, I wanted problems that occur with some frequency. Some problems in this book are more common than others, but all occur often enough in real-world situations that one of my colleagues wrote down a solution. Second, I looked for techniques that aren't commonly known, such as using the `fn:fold-left` function when working with a sequence of maps. Finally, some recipes require explanations that provide insight into how to approach programming with MarkLogic. Each recipe provides some combination of these factors.

Developers will get the most value from these recipes and the accompanying discussions after they've worked with MarkLogic for at least a few months and built an application or two. If you're just getting started, I suggest spending some time on MarkLogic University classes first, then come back to this material.

The recipes in this book were submitted by a variety of MarkLogic employees: sales engineers, who demonstrate the value of MarkLogic; consultants, who work with customers to build production applications; and members of the Engineering team, who build MarkLogic Server itself. Check *http://developer.marklogic.com/recipes* for additional recipes or to suggest your own to the broader community.

Acknowledgments

My thanks to Diane Burley, for doing the hounding necessary for me to have a shot at my deadlines.

I'd like to thank the many members of the MarkLogic Community who contributed recipes, including Bill Holmes, Tyler Replogle, Jason Hunter, Paxton Hare, Geert Josten, Mark Plotnick, and Julio Solis.

CHAPTER 1

Peak Performance

Many MarkLogic installations store large amounts of data, but still provide fast searches. The key to performance is understanding how MarkLogic works—specifically understanding query and update modes, and the use of indexes. These two recipes help ensure you're getting the speed you need for your applications.

Assert Query Mode

Problem

All MarkLogic requests run in either *query* or *update* mode, based on a static analysis of the code. The mode is important, because query requests are able to run without locking database content. Accidentally running in update mode is a common cause of requests running slower than expected.

Verify that a MarkLogic statement is running in *query* mode.

Solution

Applies to MarkLogic versions 7 and higher

Place this snippet as early in the code path as you can to make sure it is executed before MarkLogic spends too much time on other parts of your request:

```
let $assert-query-mode as xs:unsignedLong :=
  xdmp:request-timestamp()
```

If a request that includes this line is run in update mode, then this error will be thrown:

```
> XDMP-AS: (err:XPTY0004) let $assert-query-mode as
xs:unsignedLong := xdmp:request-timestamp() -- Invalid coercion:
() as xs:unsignedLong
```

Discussion

Sometimes MarkLogic's static analysis may see something that triggers update mode, even if that was not the developer's intent. The code in this recipe will throw an exception if it is run as an update, making it easy to notice the problem. Once this problem has been seen, find the code that caused the statement to run as an update. If the statement really should be running as an update, remove the assertion. If the update can be removed or isolated into an `xdmp:invoke()` call, do that to allow the statement to run as a query. Using this function, we can specify the `different-transaction` option, causing the update to be separated from the main request.

See the Transaction Type section of the Application Developer's Guide for more information about *query* or *update* modes.

Note that we don't need the same approach for Server-side JavaScript (SJS). With SJS, there is no static analysis; the developer must explicitly declare update mode.

It's important to see that we can't just call `xdmp:request-timestamp()` and get the same effect. The magic is in the `as xs:unsignedLong`—because that clause is present, MarkLogic will expect the value to be an unsigned long, or convertible to one. If the code returns the empty sequence, the conversion can't happen, and the error is thrown.

The name is important too, in order to be self-documenting. What we *don't* want to happen is that a developer runs into this exception and realizes that it can be "fixed" by removing the `as xs:unsignedLong`, or by changing it to `as xs:unsignedLong?` (making it optional). The presence of the word `assert` in the name provides a clue that we're expecting something here, and silencing the message would be contrary to the original developer's intent.

What do you do if this exception gets thrown? If that's happening, MarkLogic sees that updates might be made. Check whether those updates can be made in a different transaction using `xdmp:invoke` or `xdmp:invoke-function`. Consider whether those updates need to be made at all. If updates really should be part of a request, you can remove the assertion—but make sure you aren't locking too many documents.

Fast Distinct Values

Problem

You want to quickly find the distinct values in a particular element or JSON property.

Solution

Build a range index on the element or property, then call:

```
let $ref :=
  (: call one of the cts:*-reference functions to create a
     reference to your index
   :)
return cts:values($ref)
```

Required Index

Range index on the target element or property.

Discussion

Wanting a list of the distinct values in an element or property is a common problem. Developers who are new to MarkLogic often turn to `fn:distinct-values()`, like this:

```
fn:distict-values(/content/author/full-name)
```

While this approach will work fine for small numbers of values, it doesn't scale. As written, MarkLogic will retrieve all fragments that the `/content/author/full-name` path matches, put the `full-name` elements into a sequence, and pass that to `fn:distinct-values()`. Because `distinct-values` expects a sequence of strings, each element is converted to a string. The function will then loop through each string it was given in order to find the unique values.

Consider a database that has just 1,000 matching documents, but just 10 distinct values. Even such a small example is enough to illustrate how much effort MarkLogic has to waste by loading all 1,000 fragments to get just those 10 values. To see how many fragments

MarkLogic would need to load to answer this query on your data, run this in Query Console: `xdmp:plan(/content/author/full-name)`, substituting your XPath for `/content/author/full-name`.

Conversely, if a range index is available, then the work has already been done. An element range index on `full-name`, or a path range index on `/content/author/full-name`, will have a list of distinct values, along with identifiers of fragments that hold the values. By calling `cts:values()`, we directly access the index and don't need to load any of the fragments.

CHAPTER 2

Fun with Maps

Maps (known as associative arrays in some languages) are a useful data structure, allowing fast, key-based access to a value. MarkLogic provides a common set of map operators, but the recipes in this chapter make them even easier to work with.

Check Whether Two Maps Are Equal

Problem

Sometimes you need to see if two maps are equal, but don't want to loop through all the keys and compare them. If you do an equals (=), you'll get an error called `XDMP-COMPARE` saying "Items not comparable."

Solution

Applies to MarkLogic versions 7 and higher

If you serialize the map into XML, then you can use `fn:deep-equal()`. Here is an example of how this can be done:

```
let $mapA :=
  map:new((
    map:entry("a", "aardvark"),
    map:entry("b", "badger")
  ))
let $mapB :=
  map:new((
    map:entry("a", "aardvark"),
    map:entry("b", "badger")
  ))
let $mapC :=
  map:new((
    map:entry("c","candidate")
  ))
return
 (
   (: ($mapA eq $mapB), will cause the XDMP-COMPARE error :)
   fn:deep-equal(<x>{$mapA}</x>, <x>{$mapB}</x>),
   fn:deep-equal(<x>{$mapA}</x>, <x>{$mapC}</x>)
 )
```

Discussion

MarkLogic represents maps as XML, so:

```
map:new((
  map:entry("a", "aardvark"),
  map:entry("b", "badger")
))
```

becomes:

```
<map:map xmlns:map="http://marklogic.com/xdmp/map"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <map:entry key="b">
    <map:value xsi:type="xs:string">badger</map:value>
  </map:entry>
  <map:entry key="a">
    <map:value xsi:type="xs:string">aardvark</map:value>
  </map:entry>
</map:map>
```

With that XML representation, `fn:deep-equal()` is able to make the comparison.

Find the Intersection of a Sequence of Maps

Problem

The intersection of two maps is the set of key/value pairs that are the same in both maps. To find the intersection of two maps, you can use the map intersection operator (`*`), like this: `$mapA * $mapB`. But what if you have an arbitrarily long sequence of maps?

Solution

Applies to MarkLogic versions 7 and higher

This is where folding becomes very handy. The `fn:fold-left` function applies an operation to a sequence of values:

```
declare function local:intersect($maps as map:map*)
as map:map* {
  fn:fold-left(
    function($left, $right) { $left * $right },
    fn:head($maps),
    fn:tail($maps)
  )
};
let $mapA :=
  map:new((
    map:entry("a", "aardvark"),
    map:entry("b", "badger")
  ))
let $mapB :=
  map:new((
    map:entry("a", "aardvark"),
    map:entry("b", "badger"),
    map:entry("d", "duck")
  ))
let $mapC :=
  map:new((
    map:entry("a", "aardvark"),
    map:entry("b", "badger"),
    map:entry("c", "candidate")
  ))
return
  (
    local:intersect(($mapA, $mapB, $mapC))
  )
```

The result is:

```
<map:map
  xmlns:map="http://marklogic.com/xdmp/map"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <map:entry key="b">
    <map:value xsi:type="xs:string">badger</map:value>
  </map:entry>
  <map:entry key="a">
    <map:value xsi:type="xs:string">aardvark</map:value>
  </map:entry>
</map:map>
```

Discussion

The fn:fold-left() function applies a function to a series of values, with the result of one operation being input to the next. For instance:

```
fn:fold-left(
  function($left, $right) { $left + $right },
  1,
  (2, 3)
)
```

This applies the specified function to the 1 and the first item in the sequence, 2. These are added together, producing 3. That accumulated value and the next value in the sequence are then passed to the function. The new accumulated value becomes 3 + 3 = 6. The sequence is empty now, so fn:fold-left is finished.

With the maps, the local:intersect() function will use the intersect operator ("*") to combine $mapA and $mapB, then combine that result with $mapC.

Apply a Function to All Values in a Map

Problem

Generate a new map by applying a function to each value in a map.

Solution

Applies to MarkLogic versions 7 and higher

The local:apply-to-map() function takes a function to apply to each value, as well as a map to work on:

```
declare function local:apply-to-map(
  $function as xdmp:function,
  $mapIN as map:map
) as map:map
{
  map:new(
    (: Uses the simple map operator; see discussion below :)
    map:keys($mapIN) !
      map:entry(., xdmp:apply($function, map:get($mapIN, .)))
  )
};

declare function local:plus-one($n)
```

```
{
  $n + 1
};

(: example run :)
let $map :=
  map:new((
    map:entry("foo", 1),
    map:entry("bar", 2),
    map:entry("stuff", 3),
    map:entry("nonsense", 4)
  ))
return local:apply-to-map(
  xdmp:function(xs:QName("local:plus-one")),
  $map
)
```

Discussion

In XQuery, as in a number of other languages, functions are items that we can pass around. This allows us to set up a function that will apply another function in some way. In this case, we're looping through the keys of an input map, applying the specified function to each value.

Notice that the function returns a new map with the changed values. It's also possible to write a function like this that will modify the map in place, but returning a new map is more in keeping with functional programming.

The function to be applied can do whatever you want. The key element is that it needs to take a single value and return a new value. In the example, these values are simple numbers, but they could be XML nodes, sequences, strings, or whatever your application calls for. The key line in `local:apply-to-map` is:

```
map:keys($mapIN) !
  map:entry(., xdmp:apply($function, map:get($mapIN, .)))
```

This line uses the simple map operator (`!`), which applies some code to each item in a sequence. The same line can be written as a FLWOR statement, which is equivalent, but a bit less succinct:

```
for $item in map:keys($mapIN)
return
  map:entry($item,
    xdmp:apply($function, map:get($mapIN, $item)))
```

With the simple map operator, the period acts as the current item.

CHAPTER 3

Document Security

MarkLogic provides a robust, role-based security model. Most of the functions expect to work with the IDs of roles or users, but names are much easier for humans to process. These recipes provide easier insight into who can see what.

List User Permissions on a Document

Problem

You want a list of a particular user's permissions on a document.

Solution

Applies to MarkLogic versions 7 and higher

The `xdmp:document-get-permissions()` function will get all permissions, but you can narrow this down after identifying the user's roles:

```
let $roles := xdmp:user-roles("some-user")
return
  xdmp:document-get-permissions("/content/some-doc.json")
    [sec:role-id = $roles]/sec:capability/fn:string()
```

The result will be a sequence of permission strings from among `read`, `update`, `insert`, and `execute`.

Discussion

Permissions are assigned to a document by role. Users are also assigned roles, and through them gain access to documents.

The first step of this recipe is to gather the roles that the specified user has. The `xdmp:user-roles()` function returns both the roles that the user has been directly granted and any inherited roles.

With the roles in hand, we can retrieve all the permissions on the target document, then use some XPath to retrieve just the ones we are interested in.

Note that the `sec` namespace is available by default—you do not need to declare it.

Get Permissions with Role Names

Problem

Get the permissions on a document, decorated with the names of the roles.

Solution

Applies to MarkLogic versions 7 and higher

We want to get not just the IDs of the roles, but their names as well. This requires calling `sec:get-role-names()`, which must be run against the Security database. However, `xdmp:document-get-permissions()` must be run against the database containing the document about which we want the information.

```
import module namespace sec="http://marklogic.com/xdmp/security"
  at "/MarkLogic/security.xqy";
declare function local:dump-perms($uri)
{
  for $perm in xdmp:document-get-permissions($uri)
  let $role-name :=
    xdmp:invoke-function(
      function() {
        try {
          sec:get-role-names($perm/sec:role-id)
        }
        catch($ex) {()}
      },
      <options xmlns="xdmp:eval">
```

```
          <database xmlns="http://www.w3.org/1999/xhtml">{
            xdmp:security-database()
          }</database>
        </options>
      )
    return
      <role
        id="{$perm/sec:role-id}"
        name="{$role-name}"
        capability="{$perm/sec:capability}"></role>
};
local:dump-perms("/content/doc1.json")
```

Sample Output

```
(
  <role id="7054712474775191582" name="RunDMC-author"
    capability="update"></role>
  <role id="13533095337080026511" name="RunDMC-role"
    capability="read"></role>
)
```

Required Privileges

- *http://marklogic.com/xdmp/privileges/get-role-names*
- *http://marklogic.com/xdmp/privileges/xdmp-invoke*

Discussion

When we get permissions for a document, we typically get something like this:

```
(
  <sec:permission>
    <sec:capability>read</sec:capability>
    <sec:role-id>324978243</sec:role-id>
  </sec:permission>,
  <sec:permission>
    <sec:capability>read</sec:capability>
    <sec:role-id>32493478578243</sec:role-id>
  </sec:permission>,
  <sec:permission>
    <sec:capability>update</sec:capability>
    <sec:role-id>32493478578243</sec:role-id>
  </sec:permission>
)
```

That provides the essential information, but to be useful to people, we really need the role names, not just the IDs. This recipe looks up the names. sec:get-role-names() gives us the role names, with the requirement that the function be run against the Security database. In order to do that, we're calling xdmp:invoke-function(). We could have used xdmp:eval() here; either function allows us to run a block of code in a different execution context. There's a big advantage to invoke: the function has access to the local variables, so we don't need to pass in the role ID to look up as an external variable, as we would with xdmp:eval. We also avoid having code in a string, which is generally harder to maintain.

Notice the try/catch. sec:get-role-names() will throw an error if called with a role ID that is not in the Security database. How can this happen?

Suppose we have a role, role-1. We insert a document, giving role-1 read and update permissions:

```
xquery version "1.0-ml";

xdmp:document-add-permissions(
  "/example.xml",
  (xdmp:permission("role-2", "read"),
   xdmp:permission("role-2", "update"))
)
```

Right now, if we run the recipe above, here's the output we get:

```
<role id="3480302512589563034" name="role-2"
  capability="update"></role>
<role id="3480302512589563034" name="role-2"
  capability="read"></role>
<role id="3480302512512133719" name="role-1"
  capability="read"></role>
<role id="3480302512512133719" name="role-1"
  capability="update"></role>
```

Now suppose that role-1 gets deleted, due to changing security requirements or implementation. When a role is deleted, it is removed from all users, and the record of it is removed from the Security database. However, the indexes are not updated to reflect that the role no longer exists—doing so could be a very large operation if the role had permissions on many documents. Note that this is not a security problem, because no user has that role anymore. However, it does mean that our document still lists permissions for this orphaned role. If an invalid ID gets passed to sec:get-role-names(), then the function will throw an error. This is why we have the try/catch in place: to allow us to continue gathering information on known roles. After removing role-1, here is the result of calling the recipe:

```
<role id="3480302512589563034" name="role-2"
  capability="update"></role>
<role id="3480302512589563034" name="role-2"
  capability="read"></role>
<role id="3480302512512133719" name=""
  capability="read"></role>
<role id="3480302512512133719" name=""
  capability="update"></role>
```

The empty name indicates an orphaned role. If we prefer to suppress those results, we can add `where $role-name ne ""` to the FLWOR statement. We can also use this to discover orphaned roles, which can be cleaned up by using `xdmp:document-set-permissions()` with the valid ones.

CHAPTER 4

Working with Documents

Documents are the fundamental data structure for MarkLogic. This chapter addresses some common problems: generating unique identifiers for documents and finding binary documents (which can't be directly searched by content).

Generate a Unique ID

Problem

Generate a unique identifier for each document.

Solution

Use the built-in `sem:uuid-string()` function:

```
xdmp:document-insert(
 "/content/" || sem:uuid-string() || ".xml",
 $new-doc
)
```

Discussion

This recipe generates identifiers that are unique. When generating unique IDs, many people start with the idea that they should be monotonically increasing numbers. While this is conceptually a reasonable thing to do, it has a hidden requirement: a single place where the next value is stored and updated. For instance, consider having a document that tracks the next available number. A process that wants to insert a new document must create a write-lock on the number-tracking document. Any other process wanting to insert must wait until it can get a write-lock on that same document. This single resource prevents MarkLogic from being able to work on noninterfering inserts in parallel. When faced with a requirement to generate monotonically increasing numbers for IDs, ask whether they really need to be that, or simply be unique. Most often, the real requirement is to be unique. The `sem:uuid()` and `sem:uuid-string()` functions provide a fast way to accomplish that.

Another thought to consider is where to use the unique identifier. In a database of students, each new student's information is stored in a document. Suppose the developer decided to use `let $uri := "/student/" || $student-name || ".xml"` as the URI, where `$student-name` is the student's first and last names, joined by a hyphen. To avoid collisions, the developer adds a `<student-id>` element, populated with `sem:uuid-string()`. When inserting a new student record, the application code will need to check whether a student already has the URI that would be built using the standard process. If so, then the code would need to modify the URI somehow; perhaps by adding a "-2" to the student's name. Of course, to do that, the code must check whether that URI has already been taken, and so on. A much simpler approach is to use the unique ID in the URI, thus avoiding the concern: `let $uri := "/student/" || sem:uuid-string() || ".xml"`.

Find Binary Documents

Problem

Find the URIs of binary documents.

Solution

Applies to MarkLogic versions 7 and higher

```
xquery version "1.0-ml";

declare namespace qry  = "http://marklogic.com/cts/query";

let $binary-term :=
  xdmp:plan(/binary())//qry:term-query/qry:key/text()
return cts:uris((), (), cts:term-query($binary-term))
```

Required Privileges

- *http://marklogic.com/xdmp/privileges/xdmp-plan*

Discussion

This recipe returns a sequence of URIs for all the binary documents in the target database.

The implementation relies on how the `/binary()` XPath is interpreted. `xdmp:plan()` tells us how MarkLogic sees a query. Part of the result is the final plan:

```
<qry:final-plan xmlns:qry="http://marklogic.com/cts/query">
  <qry:and-query>
    <qry:term-query weight="0">
      <qry:key>7908746777995149422</qry:key>
      <qry:annotation>document-format(binary)</qry:annotation>
    </qry:term-query>
  </qry:and-query>
</qry:final-plan>
```

Notice the `term-query` part—in addition to storing a document, MarkLogic stores metadata about a document, and that metadata is queryable, too. Sometimes the trick is just figuring out how to specify that query. In this case, we use information from `xdmp:plan` to get the job done.

You might ask, "Why not just use XPath, such as `/binary()`?" This would also work, but it works by retrieving the binaries themselves. You could take it a step further with `/binary() ! fn:base-uri(.)` to get just the URIs (which is what the recipe provides), but again, this requires loading up the actual documents and doing something with them. The beauty of the recipe is that it works on indexes.

There's one sneaky bit with this recipe: `cts:term-query` isn't a published function. That means you should be careful where you use it, but for this recipe, it gets the job done.

Find Recently Modified Binary Documents

Problem

Find binary documents that have been recently modified.

Solution

Applies to MarkLogic versions 7 and higher

```
xquery version "1.0-ml";

declare namespace qry  = "http://marklogic.com/cts/query";

let $binary-term :=
  xdmp:plan(/binary())//qry:term-query/qry:key/text()
let $query-start :=
  (fn:current-dateTime() - xs:dayTimeDuration("P1D"))
let $query-stop := fn:current-dateTime()
let $query := cts:and-query((
  cts:properties-fragment-query(
    cts:and-query((
      cts:element-range-query(
        xs:QName("prop:last-modified"), ">", $query-start),
      cts:element-range-query(
        xs:QName("prop:last-modified"), "<", $query-stop)
    ))
  ),
  cts:term-query($binary-term)
))
return (
  text{
    "Estimate:",
    xdmp:estimate(cts:search(fn:doc(), $query))
  },
  cts:uris((), ("limit=100"), $query)
)
```

Required Privileges

- *http://marklogic.com/xdmp/privileges/xdmp-plan*

Required Indexes

- `"maintain last modified"` must be on
- `dateTime` range index on `prop:last-modified`

Discussion

Recently modified binaries can be found if the `"maintain last modified"` option on the target database is active. You must also have a `dateTime` range index set up on `prop:last-modified`, so that the `cts:element-range-query` will work.

The `xs:dayTimeDuration` chosen for `$query-start` defines what "recent" means in this case. Notice that the last-modified date is stored in a property fragment, so this recipe uses a `cts:properties-fragment-query` to look for it.

The recipe returns an estimate of the number of recently modified binaries, as well as the URIs of the first 100. We can count on the estimate to be accurate, as the query is targeting indexes.

CHAPTER 5

The Task Server

MarkLogic's Task Server provides a way to schedule work asynchronously. By default, up to 16 tasks will run at the same time on the standard queue, along with 16 tasks on the high-priority queue. Recipes in this chapter show how to remove tasks from the queue, which normally only happens if the queue is wiped by a MarkLogic restart.

Cancel Active Tasks on the Task Server

Problem

There are active tasks being executed on the Task Server, and you'd like to cancel some or all of them.

Solution

Applies to MarkLogic versions 7 or higher

```
xquery version "1.0-ml";

declare namespace hs="http://marklogic.com/xdmp/status/host";
declare namespace ss="http://marklogic.com/xdmp/status/server";

let $max-task-duration := xs:dayTimeDuration("PT7M")
let $min-retries := 2
(: set $user-ids to desired sequence of ids :)
let $user-ids := xdmp:get-current-userid()
let $compare-time := fn:current-dateTime() - $max-task-duration
for $host as xs:unsignedLong in xdmp:hosts()
```

```
let $task-server-id :=
  xdmp:host-status($host)//hs:task-server-id
let $requests :=
  xdmp:server-status($host, $task-server-id)//ss:request-status[
    not(xs:boolean(ss:canceled))
    (: and ss:user = 7071164300007443533 :)
    (: and ss:retry-count >= $min-retries :)
    and ss:request-text = "/some/module/uri.xqy"
    and ss:start-time < $compare-time
]
return (
  text{
    "There are currently", fn:count($requests),
    "matching requests on host", xdmp:host-name($host)
  },
  for $request in $requests
  let $request-id as xs:unsignedLong := $request/ss:request-id
  let $start as xs:dateTime := $request/ss:start-time
  return (
    text{
      $request-id, "started:", $start,
      "duration:", (fn:current-dateTime() - $start),
      if (fn:true()) then (
        xdmp:request-cancel(
          $host,
          $request/ss:server-id,
          $request-id),
        "-- cancel issued"
      )
      else ()
    }
  )
)
```

Required Privileges

- *http://marklogic.com/xdmp/privileges/status*
- Either of:
  - *http://marklogic.com/xdmp/privileges/cancel-any-request*
  - *http://marklogic.com/xdmp/privileges/cancel-my-requests*

Discussion

Under certain circumstances, it may become necessary to cancel all the tasks on the Task Server that match a specific pattern. This may be due to human error, such as someone performing a “re-replicate all documents in domain," or due to specific conditions occurring within a production application. Note that this recipe only cancels tasks that are actively running. Queued tasks are not affected.

The `http://marklogic.com/xdmp/privileges/status` privilege is required. The user running this script must also have either `http://marklogic.com/xdmp/privileges/cancel-my-requests` (to cancel requests running as that user) or `http://marklogic.com/xdmp/privileges/cancel-any-request` (to cancel any other requests).

As written, this script uses multiple filters to only show tasks that have been executing for over seven minutes, have already been retried twice, and are associated with a specific module. Other criteria that may be useful include the user that the task is running under. Also note the usage of a fixed conditional that can be toggled to perform the `xdmp:request-cancel()`. This allows you to test your query criteria iteratively until you are satisfied that you will only cancel the desired tasks.

Below is an example of a request record returned by `xdmp:server-status`. Filters can be written for any of these fields.

```
<request-status xmlns="http://marklogic.com/xdmp/status/server">
  <request-id>796436023172923809</request-id>
  <server-id>14409436176295478539</server-id>
  <host-id>15405276691316718307</host-id>
  <transaction-id>4091631258594104359</transaction-id>
  <canceled>false</canceled>
  <modules>11527541000394886112</modules>
  <database>17515328177061313217</database>
  <root>/</root>
  <request-kind>invoke</request-kind>
  <request-text>/</request-text>
  <update>false</update>
  <start-time>2017-03-08T15:53:22.543552Z</start-time>
  <time-limit>600</time-limit>
  <max-time-limit>3600</max-time-limit>
  <user>7071164303237443533</user>
  <trigger-depth>0</trigger-depth>
  <expanded-tree-cache-hits>0</expanded-tree-cache-hits>
  <expanded-tree-cache-misses>0</expanded-tree-cache-misses>
  <request-state>running</request-state>
  <profiling-allowed>true</profiling-allowed>
  <profiling-enabled>false</profiling-enabled>
  <debugging-allowed>true</debugging-allowed>
  <debugging-status>detached</debugging-status>
  <retry-count>0</retry-count>
</request-status>
```

Cancel Active and Queued Tasks on the Task Server

Problem

You need to cancel requests that have queued on the Task Server, without wanting to clear the entire queue or restart the host.

Solution

Applies to MarkLogic versions 7 and higher

```
xquery version "1.0-ml";

declare namespace hs="http://marklogic.com/xdmp/status/host";
declare namespace ss="http://marklogic.com/xdmp/status/server";

(: set debug to fn:false() to actually cancel tasks :)
let $debug := fn:true()
let $seconds-to-run as xs:integer := 3600
let $seconds-to-pause as xs:integer := 3
(: Cancel tasks running longer than: :)
let $max-task-duration := xs:dayTimeDuration("PT7M")
let $max-iterations :=
  xs:integer($seconds-to-run div $seconds-to-pause)
let $_ := xdmp:set-request-time-limit($seconds-to-run)
for $i in (1 to $max-iterations)
return (
  if ($debug) then text{"Iteration", $i, "of", $max-iterations}
  else (),
  xdmp:invoke-function(
    function () {
      for $host as xs:unsignedLong in xdmp:hosts()
      let $task-server-id :=
        xdmp:host-status($host)//hs:task-server-id
      let $compare-time :=
        fn:current-dateTime() - $max-task-duration
      for $request in
        xdmp:server-status($host, $task-server-id)
          //ss:request-status[
            not(xs:boolean(ss:canceled))
            (: and ss:user = 7071164300007443533 :)
            (: and ss:retry-count >= 2 :)
            (: and ss:request-text = "/some/module/uri.xqy" :)
            and ss:start-time < $compare-time
          ]
      let $request-id as xs:unsignedLong :=
        $request/ss:request-id
      return (
        if ($debug) then (
          text{
            "Cancelling request:", $request-id,
            "with duration:",
            fn:current-dateTime() -
              xs:dateTime($request/ss:start-time)
          }
        ) else (),
        try {
          xdmp:request-cancel(
            $host, $task-server-id, $request-id
          )
        } catch ($e) { }
      )
    },
    <options xmlns="xdmp:eval">
      <isolation>different-transaction</isolation>
    </options>
  ),
  if ($debug) then
    text{"pausing for", $seconds-to-pause, "seconds"}
  else (),
  xdmp:sleep($seconds-to-pause * 1000)
)
```

Required Privileges

- *http://marklogic.com/xdmp/privileges/xdmp-invoke*
- *http://marklogic.com/xdmp/privileges/status*
- Either of:
  - — *http://marklogic.com/xdmp/privileges/xdmp-set-request-time-limit-any*
  - — *http://marklogic.com/xdmp/privileges/xdmp-set-request-time-limit-my*
- Either of:
  - — *http://marklogic.com/xdmp/privileges/cancel-any-request*
  - — *http://marklogic.com/xdmp/privileges/cancel-my-requests*

Discussion

When we ask for status information about the Task Server, we can find out how many tasks are queued, along with the task IDs of the tasks that are currently running, but there's no way to inspect the queued tasks. We can clear the queue by restarting MarkLogic, but often that's not desirable. Also, we might only want to remove some tasks from the queue. This recipe allows us to specify which tasks to remove.

Because we can only see the running tasks, we can only cancel tasks that have started running. This also means that if there are as many good tasks running (ones that we want to continue running) as there are Task Server threads (typically 16), we won't be able to cancel any tasks until some of the good ones finish. For example, suppose we set some selection criteria, and among the currently running tasks 12 of them do not match those criteria. That leaves just four that we can look at and try to cancel. During the next iteration of the loop, those four will have been replaced by four queued tasks (if there are some queued). If none of the 12 have finished, then we can only cancel 4 more.

CHAPTER 6

Administration

MarkLogic takes less work to administer than many databases, but there are still things to do. The first recipe helps out those who don't have admin privileges; the second helps track the progress of the MarkLogic version 9 rolling upgrade feature.

Find Hostnames in a Cluster

Problem

Someone with access to Query Console, but not to the Admin UI, wants to know the names of hosts in the cluster.

Solution

Applies to MarkLogic versions 7 and higher

```
xdmp:hosts() ! xdmp:host-name(.)
```

Discussion

The Admin UI provides a lot of information about a MarkLogic cluster: configuration of the databases, application servers, groups, and other information. It also provides a means to change all these things. As such, access to it is limited to those who have the admin privilege. This permits a user to do anything in MarkLogic, including seeing and updating any data, so this role should be held tightly to an administration team that uses procedures to ensure mistakes don't happen, especially in production.

Query Console, on the other hand, can be much more widely available. Developers working on a project will likely be able to use Query Console to try out queries. Their rights will likely be limited to what they need, but it doesn't take much to run Query Console.

Sometimes team members may want to get the list of hostnames in a cluster. This recipe is a simple one-liner, but very useful all the same. `xdmp:hosts()` returns the IDs of all hosts in the cluster. We then feed those values to `xdmp:host-name` to return the human-readable names.

Find Current and Effective MarkLogic Versions During Rolling Upgrade

Problem

During a rolling upgrade, some servers will have the original version, while others will have the new version but act as if they still had the old. Generate a report showing which servers have which actual and effective versions.

Solution

Applies to MarkLogic versions 8.0-7 and higher

```
xquery version "1.0-ml";

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

declare namespace hosts = "http://marklogic.com/manage/hosts";
declare namespace http = "xdmp:http";

<hosts>
  <cluster-version>{
    admin:cluster-get-effective-version(
      admin:get-configuration()
    )
  }</cluster-version>
  {
    for $id in xdmp:hosts()
    let $response :=
      xdmp:http-get(
        "http://localhost:8002/manage/v2/hosts/" || $id ||
          "?view=status&amp;format=xml",
        <options xmlns="xdmp:http">
```

```
              <authentication method="digest">
                <username>admin</username>
                <password>admin</password>
              </authentication>
              <headers>
                <content-type>application/json</content-type>
              </headers>
            </options>)
        return
          if ($response[1]/http:code = 200) then
            <host>
              <name>{
                $response[2]/hosts:host-status/hosts:name/
                  fn:string()
              }</name>
              <software-version>{
                $response[2]/hosts:host-status/hosts:version/
                  fn:string()
              }</software-version>
              <effective-version>{
                $response[2]/hosts:host-status/
                  hosts:effective-version/fn:string()
              }</effective-version>
            </host>
          else
            <host>
              <name>{xdmp:host-name($id)}</name>
              <code>{$response[1]/http:code/fn:string()}</code>
            </host>
      }
    </hosts>
```

This produces output like:

```
    <hosts>
      <cluster-version>8000600</cluster-version>
      <host>
        <name>ml3.local</name>
        <software-version>8.0-6</software-version>
        <effective-version>8000600</effective-version>
      </host>
      <host>
        <name>ml1.local</name>
        <software-version>8.0-6</software-version>
        <effective-version>8000600</effective-version>
      </host>
      <host>
        <name>ml2.local</name>
        <software-version>8.0-6</software-version>
        <effective-version>8000600</effective-version>
      </host>
    </hosts>
```

Required Privileges

- *http://marklogic.com/xdmp/privileges/manage*
- *http://marklogic.com/xdmp/privileges/admin-module-read*

Discussion

Rolling upgrades allow a single cluster to roll forward incrementally to a new release. This is done by taking down one node at a time, upgrading that node, and bringing it back up. When the node is restarted, it will talk to the rest of the cluster in the lowest level of the other nodes in the cluster. When the last server in the cluster gets upgraded, then all nodes switch over to the new version.

To see the current state of upgrades across a cluster, we need to ask each of the servers for some data—there's no function we can run on one server that will tell us the version running on another. To get the information we want, we send a Management API request to each of the hosts asking for current status, then extract the version information.

When we send the HTTP request, we get back a two-item sequence. The first item gives details on the response itself—response code, and so on. The second item has the content of the response. If XML produced by the recipe has a `<code>` element in it, that is the HTTP error code from the REST call.

About the Author

David Cassel is the Technical Community Manager for MarkLogic, where he educates developers, architects, and DBAs about how to use MarkLogic to implement data integration solutions. David has more than 20 years experience as a developer, building applications ranging from quick proof-of-concepts to production systems for customers across such verticals as public sector, financial services, medical, and telecommunications. He has also built a number of developer productivity tools, some of which can be found on GitHub.

Besides building applications, Dave creates and delivers educational material in a variety of formats, including blog posts, YouTube videos, and Meetup presentations.