Application Developer's Guide — Chapter 12

Using the map Functions to Create Name-Value Maps

This chapter describes how to use the map functions and includes the following sections:

- Maps: In-Memory Structures to Manipulate in XQuery
- map:map XQuery Primitive Type
- Serializing a Map to an XML Node
- Map API
- Map Operators
- Examples

Maps: In-Memory Structures to Manipulate in XQuery

Maps are in-memory structures containing name-value pairs that you can create and manipulate. In some programming languages, maps are implemented using hash tables. Maps are handy programming tools, as you can conveniently store and update name-value pairs for use later in your program. Maps provide a fast and convenient method for accessing data.

MarkLogic Server has a set of XQuery functions to create manipulate maps. Like the xdmp:set function, maps have side-effects and can change within your program. Therefore maps are not strictly functional like most other aspects of XQuery. While the map is in memory, its structure is opaque to the developer, and you access it with the built-in XQuery functions. You can persist the structure of the map as an XML node, however, if you want to save it for later use. A map is a node and therefore has an identity, and the identity remains the same as long as the map is in memory. However, if you serialize the map as XML and store it in a document, when you retrieve it will have a different node identity (that is, comparing the identity of the map and the serialized version of the map would return false). Similarly, if you store XML values retrieved from the database in a map, the node in the in-memory map will have the same identity as the node from the database while the map is in memory, but will have different identities after the map is serialized to an XML document and stored in the database. This is consistent with the way XQuery treats node identity.

The keys take `xs:string` types, and the values take `item()*` values. Therefore, you can pass a string, an element, or a sequence of items to the values. Maps are a nice alternative to storing values an in-memory XML node and then using XPath to access the values. Maps makes it very easy to update the values.

map:map XQuery Primitive Type

Maps are defined as a map:map XQuery primitive type. You can use this type in function or variable definitions, or in the same way as you use other primitive types in XQuery. You can also serialize it to XML, which lets you store it in a database, as described in the following section.

Serializing a Map to an XML Node

You can serialize the structure of a map to an XML node by placing the map in the context of an XML element, in much the same way as you can serialize a cts:query (see Serializing a cts:query to XML in the Composing cts:query Expressions chapter of the *Search Developer's Guide*). Serializing the map is useful if you want to save the contents of the map by storing it in the database. The XML conforms to the *<marklogic-dir>*`/Config/map.xsd` schema, and has the namespace `http://marklogic.com/xdmp/map`.

For example, the following returns the XML serialization of the constructed map:

```
let $map := map:map()
let $key := map:put($map, "1", "hello")
let $key := map:put($map, "2", "world")
let $node := <some-element>{$map}</some-element>
return $node/map:map
```

The following XML is returned:

```
<map:map xmlns:map="http://marklogic.com/xdmp/map"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <map:entry key="1">
    <map:value xsi:type="xs:string">hello</map:value>
  </map:entry>
  <map:entry key="2">
    <map:value xsi:type="xs:string">world</map:value>
  </map:entry>
</map:map>
```

Map API

The map API is quite simple. You can create a map either from scratch with the map:map function or from the XML representation (map:map) of the map. The following are the map functions. For the signatures and description of each function, see the *MarkLogic XQuery and XSLT Function Reference*.

- map:clear
- map:count
- map:delete
- map:get
- map:keys
- map:map
- map:put

Map Operators

Map operators perform a similar function to set operators. Just as sets can be combined in a number of ways to produce another set, maps can be manipulated with map operators to create combined results. The following table describes the different map operators:

Map Operator	Description
`+`	The union of two maps. The result is the combination of the keys and values of the first map (Map A) and the second map (Map B). For an example, see Creating a Map Union.
`*`	The intersection of two maps (similar to a set intersection). The result is the key-value pairs that are common to both maps (Map A and Map B) are returned. For an example, see Creating a Map Intersection.
`-`	The difference between two maps (similar to a set difference). The result is the key-value pairs that exist in the first map (Map A) that do not exist in the second map (Map B) are returned. For an example, see Applying a Map Difference Operator. This operator also works as an unary negative operator. When it is used in this way, the keys and values become reversed. For an example, see Applying a Negative Unary Operator.
`div`	The inference that a value from a map matches the key of another map. The result is the keys from the first map (Map A), and values from the second map (Map B), where the value in Map A is equal to key in Map B. For an example, see Applying a Div Operator.
`mod`	The combination of the unary negative operation and inference between maps. The result is the reversal of the keys in the first map (Map A) and the values in Map B, where a value in Map A matches a key in Map B. In summary, `Map A mod Map B` is equivalent to `-Map A div Map B.` For an example, see Applying a Mod Operator.

Examples

This section includes example code that uses maps and includes the following examples:

- Creating a Simple Map
- Returning the Values in a Map
- Constructing a Serialized Map
- Add a Value that is a Sequence
- Creating a Map Union
- Creating a Map Intersection
- Applying a Map Difference Operator
- Applying a Negative Unary Operator
- Applying a Div Operator
- Applying a Mod Operator

Creating a Simple Map

The following example creates a map, puts two key-value pairs into the map, and then returns the map.

```
let $map := map:map()
let $key := map:put($map, "1", "hello")
let $key := map:put($map, "2", "world")
return $map
```

This returns a map with two key-value pairs in it: the key '1' has a value 'hello', and the key '2' has a value 'world'.

Returning the Values in a Map

The following example creates a map, then returns its values ordering by the keys:

```
let $map := map:map()
let $key := map:put($map, "1", "hello")
let $key := map:put($map, "2", "world")
return
  for $x in map:keys($map)
  order by $x return
  map:get($map, $x)
(: returns hello world :)
```

Constructing a Serialized Map

The following example creates a map like the previous examples, and then serializes the map to an XML node. It then makes a new map out of the XML node and puts another key-value pair in the map, and finally returns the new map.

```
let $map := map:map()
let $key := map:put($map, "1", "hello")
let $key := map:put($map, "2", "world")
let $node := <some-element>{$map}</some-element>
let $map2 := map:map($node/map:map)
let $key := map:put($map2, "3", "fair")
return $map2
```

This returns a map with three key-value pairs in it: the key '1' has a value 'hello', the key '2' has a value 'world', and the key '3' has a value 'fair'. Note that the map bound to the `$map` variable is not the same as the map bound to `$map2`. After it was serialized to XML, a new map was constructed in the `$map2` variable.

Add a Value that is a Sequence

The values that you can put in a map are typed as an `item()*`, which means you can add arbitrary sequences as the value for a key. The following example includes some string values and a sequence value, and then outputs each results in a `<result>` element:

```
let $map := map:map()
let $key := map:put($map, "1", "hello")
let $key := map:put($map, "2", "world")
let $seq := ("fair",
             <some-xml>
                 <another-tag>with text</another-tag>
             </some-xml>)
let $key := map:put($map, "3", $seq)
return
  for $x in map:keys($map) return
  <result>{map:get($map, $x)}</result>
```

This returns the following elements:

```
<result>fair
        <some-xml>
            <another-tag>with text</another-tag>
        </some-xml>
</result>
<result>world</result>
<result>hello</result>
```

Creating a Map Union

The following creates a union between two maps and returns the key-value pairs:

```
let $mapA := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>1</map:key>
            <map:value>1</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
        </map:entry>
</map:map>)
let $mapB := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>2</map:key>
            <map:value>2</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
            <map:value>3.5</map:value>
        </map:entry>
        </map:map>)
return $mapA + $mapB
```

Any key-value pairs common to both maps are included only once. This returns the following:

```
<xml version="1.0" encoding="UTF-8">
<results warning="atomic item">
    <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="1">
            <map:value>1</map:value>
        </map:entry>
        <map:entry key="2">
            <map:value>2</map:value>
        </map:entry>
        <map:entry key="3">
            <map:value>3</map:value>
            <map:value>3.5</map:value>
        </map:entry>
    </map:map>
</results>
```

Creating a Map Intersection

The following example creates an intersection between two maps:

```
xquery version "1.0-ml";
let $mapA := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">

        <map:entry>
            <map:key>1</map:key>
            <map:value>1</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
        </map:entry>
        </map:map>)
```

```
        </map:map>)
let $mapB := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>2</map:key>
            <map:value>2</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
            <map:value>3.5</map:value>
        </map:entry>
</map:map>)
return $mapA * $mapB
```

The key-value pairs common to both maps are returned. This returns the following:

```
<xml version="1.0" encoding="UTF-8">
<results warning="atomic item">
    <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="3">
            <map:value>3</map:value>
        </map:entry>
    </map:map>
</results>
```

Applying a Map Difference Operator

The following example returns the kep-value pairs that are in Map A but not in Map B:

```
let $mapA := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>1</map:key>
            <map:value>1</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
        </map:entry>
</map:map>)
let $mapB := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>2</map:key>
            <map:value>2</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
            <map:value>3.5</map:value>
        </map:entry>
        </map:map>)
return $mapA - $mapB
```

This returns the following:

```
<xml version="1.0" encoding="UTF-8">
<results warning="atomic item">
    <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="1">
            <map:value>1</map:value>
        </map:entry>
    </map:map>
</results>
```

Applying a Negative Unary Operator

The following example uses the map difference operator as a negative unary operator to reverse the keys and values in a map:

```
xquery version "1.0-ml";
let $mapA := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>1</map:key>
            <map:value>1</map:value>

        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
        </map:entry>
        </map:map>)
let $mapB := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>2</map:key>
            <map:value>2</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
            <map:value>3.5</map:value>
        </map:entry>
</map:map>)
return -$mapB
```

This returns the following:

```
<xml version="1.0" encoding="UTF-8">
<results warning="atomic item">
    <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="3.5">
            <map:value>3</map:value>
        </map:entry>
        <map:entry key="2">
            <map:value>2</map:value>
        </map:entry>
        <map:entry key="3">
            <map:value>3</map:value>
        </map:entry>
    </map:map>
</results>
```

Applying a Div Operator

The following example applies the inference rule that returns the keys from Map A and the values in Map B, where a value of Map A is equal to a key in Map B:

```
xquery version "1.0-ml";
let $mapA := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>1</map:key>
            <map:value>1</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
        </map:entry>
        </map:map>)
let $mapB := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
```

```
        <map:entry>
            <map:key>2</map:key>
            <map:value>2</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
            <map:value>3.5</map:value>
        </map:entry>
</map:map>)
return $mapA div $mapB
```

This returns the following:

```
<xml version="1.0" encoding="UTF-8">
<results warning="atomic item">
    <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="3">
            <map:value>3</map:value>
            <map:value>3.5</map:value>
        </map:entry>
    </map:map>
</results>
```

Applying a Mod Operator

The following example perform two of the operations mentioned. First, the keys and values are reversed in Map A. Next, the inference rule is applied to match a value in Map A to a key in Map B and return the values in Map B.

```
xquery version "1.0-ml";
let $mapA := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>1</map:key>
            <map:value>1</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
        </map:entry>
        </map:map>)
let $mapB := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>2</map:key>
            <map:value>2</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
            <map:value>3.5</map:value>
        </map:entry>
</map:map>)
return $mapA mod $mapB
```

This returns the following:

```
<xml version="1.0" encoding="UTF-8">
<results warning="atomic item">
    <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="3.5">
            <map:value>3</map:value>
        </map:entry>
        <map:entry key="3">
            <map:value>3</map:value>
        </map:entry>
    </map:map>
</results>
```