MarkLogic Server

Temporal Developer's Guide

MarkLogic 9
May, 2017

Last Revised: 9.0-3, September, 2017

Copyright © 2019 MarkLogic Corporation. All rights reserved.

Table of Contents

Temporal Developer's Guide

1.0 Understanding Temporal Documents 5
- 1.1 Terms 5
- 1.2 Overview of Temporal Documents 6
- 1.3 Roles and Permissions 7
- 1.4 Temporal, URI, and Latest Collections 7
- 1.5 WORM (Write Once Read Many) Support 8

2.0 Quick Start 9
- 2.1 Create Metadata Fields for the Valid and System Axes 9
- 2.2 Create Range Field Indexes for the Valid and System Axes 10
- 2.3 Create System and Valid Axes 12
- 2.4 Create a Temporal Collection 13
- 2.5 Insert Some Temporal Documents 13
- 2.6 Run some Search Queries on the Temporal Documents 17

3.0 Managing Temporal Documents 23
- 3.1 Creating Temporal Collections 23
  - 3.1.1 Bi-temporal Collections 23
  - 3.1.2 Uni-temporal Collections 28
- 3.2 Inserting and Loading Temporal Documents 29
- 3.3 Last Stable Query Time (LSQT) and Application-controlled System Time 32
- 3.4 Using MarkLogic Content Pump (MLCP) to Load Temporal Documents 35
  - 3.4.1 Importing Documents Into a Temporal Collection 35
  - 3.4.2 Copying Non-Temporal Documents Into a Temporal Collection 35
  - 3.4.3 Copying Temporal Documents Between Databases 36
- 3.5 Setting the URI for a New Version of a Temporal Document 38
- 3.6 Protecting and Archiving Temporal Documents 39
- 3.7 Deleting and Wiping Temporal Documents 40
- 3.8 Example: The Lifecycle of a Temporal Document 41

4.0 Searching Temporal Documents 53
- 4.1 Temporal Search Query Constructors 53
- 4.2 Period Comparison Operators 54
  - 4.2.1 Allen Operators 54
  - 4.2.2 ISO SQL 2011 Operators 57
- 4.3 Comparing Two Periods 61
- 4.4 Example Search Queries 62

5.0 Copyright 67
5.0 COPYRIGHT 67

6.0 Technical Support 69

1.0 Understanding Temporal Documents

You can configure MarkLogic Server to manage and query bi-temporal data. Bi-temporal documents are associated with both a *valid* time that marks when a thing is known in the real world and a *system* time that marks when the thing is available for discovery in MarkLogic Server.

Bi-temporal data is necessary whenever there is a requirement to maintain snapshots of a transaction across various time dimensions. For example, financial and insurance industries use bitemporal data to track changes to contracts, policies, and events in a manner that adheres to strict regulation and compliance requirements.

In this guide, the term *temporal* refers to both bi-temporal and uni-temporal documents and collections.

This chapter describes the basic components used to manage temporal documents, and includes the following sections:

- Terms
- Overview of Temporal Documents
- Roles and Permissions
- Temporal, URI, and Latest Collections
- WORM (Write Once Read Many) Support

1.1 Terms

To understand the temporal functions, you need to understand the meaning of the following terms:

- *Temporal Collection*: A collection created to contain either bi-temporal or uni-temporal documents.
- *Axis*: a named pair of range indexes that is the container for periods. Bi-temporal documents have both a valid and system axis. Uni-temporal documents have only a system axes.
- *Bi-temporal*: a collection of documents with both a system time and a valid time axes.
- *Uni-temporal*: a collection of documents with only a system axes.
- *Instant*: an instant of time (such as "now", "12/31/2012, 01:00:00 am").
- *Period*: an anchored duration of time (e.g. December 01, 1999 through December 31, 2000, the fall semester).
- *User-defined Time*: a time value that user provides in replacement of system start time.

- *LSQT* (Last Stable Query Time): a document with a system start time before this point can be queried and a document with a system start time after this point can be updated and ingested.
- A *split* refers to the creation of a new document that contains the same content as a previous document, but with different valid timestamps and system end time.
- *Wipe* is to delete all versions of a temporal document.
- *Protect* is to prevent certain operations (such as temporal update) from being applied to a temporal document. The definition of such prevention is a protection rule, or protection.
- *Metadata* refers to a quick and efficient mechanism to capture metadata information of a document represented as string key and string value pairs outside of the document's root node. In the context of temporal document management, metadata is used to store valid and system timestamps and archival information. Metadata of a document is stored in all fragments of the document.
- *WORM* refers to Write Once Read Many. A data storage feature in which, once data is written to the storage device, it can be read but cannot be modified.

  Note: Document quality and permissions travel with the split. Document properties do not travel with the split.

1.2 Overview of Temporal Documents

Bi-temporal data tracks information along two different time lines:

- *Valid Time*: when the information was true in the real world. Valid time may also be called application time. Valid time is provided by the user or application. The valid end time is updated by the system when the document is split.
- *System Time*: when the information was stored in the database. System time may also be called transaction time. System time is managed by the system, except in cases when the system start time is set by the application as described in “Last Stable Query Time (LSQT) and Application-controlled System Time” on page 32.

MarkLogic also supports uni-temporal documents that have only a system time. Uni-temporal documents are managed in MarkLogic in the same manner as bi-temporal documents.

In MarkLogic, a temporal document is managed as a series of versioned documents in a protected collection. The ‘original’ document inserted into the database is kept and never changes. Updates to the document are inserted as new documents with different valid and system times. A delete of the document is also inserted as a new document. In this way, a temporal document can be “rolled back” to review, at any point in time, when the information was known in the real world and when it was recorded in the database.

1.3 Roles and Permissions

The `temporal-admin` or `admin` role is required to create axes, temporal collections, and otherwise configure the temporal environment.

> **Note:** Changing permissions on a temporal document only affects the latest version of the document that is created as a result of the change. All previous versions of the document maintain their original permissions.

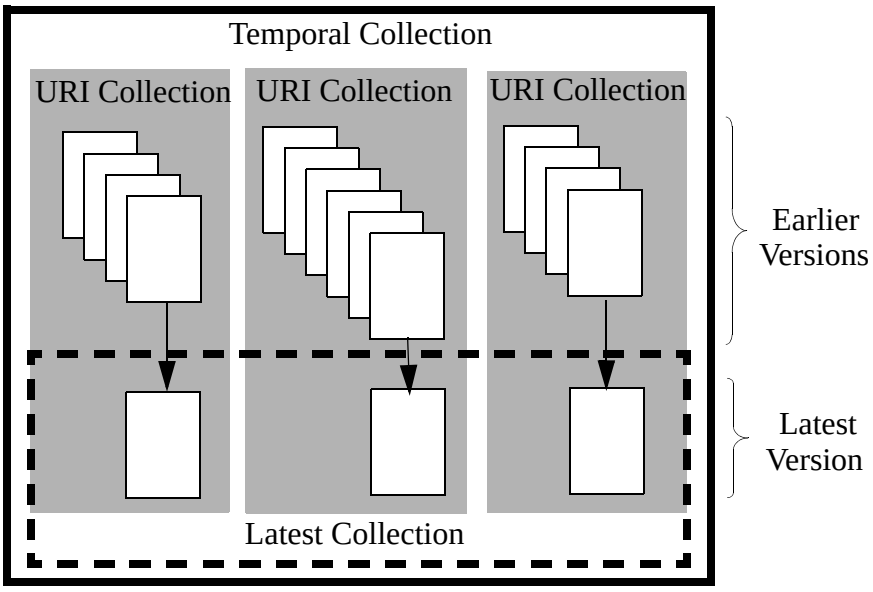
1.4 Temporal, URI, and Latest Collections

Bi-temporality and uni-temporality is defined on a protected collection, known as a *temporal collection*. A temporal collection is a logical grouping of temporal documents that share the same axes with timestamps defined by the same range indices. You can create additional temporal collections if you have documents that require a different schema for the timestamps. A temporal collection can be created for either bi-temporal documents (documents with both system and valid times) or uni-temporal documents (documents with only system times).

When a document is inserted into a temporal collection, a *URI collection* is created for that document. When the document is updated, a new document representing the update is inserted into the document's URI collection. Any new document inserted into the temporal collection will have its own unique URI collection that will hold all of the versions of that document.

Additionally, the latest version of each document will reside in a *latest collection*.

The following illustrates how three temporal documents would be organized into the temporal, URI, and latest collections:



> **Note:** When a temporal document is deleted, it is removed from the `latest` collection.

1.5 WORM (Write Once Read Many) Support

You can set an “archive date” on temporal documents at which time they can be moved to a WORM (Write Once Read Many) device. Once a temporal document is written to a WORM device, it cannot be modified or deleted. This write protection ensures that the data cannot be tampered with once it is written to the device until a pre-determined expiration date.

For details on archiving temporal documents, see “Protecting and Archiving Temporal Documents” on page 39.

2.0 Quick Start

You can configure MarkLogic Server to manage and query temporal data.

This chapter walks you through the procedures for configuring the `Documents` database to store temporal documents and for inserting and querying temporal documents. The following are the main sections:

- Create Metadata Fields for the Valid and System Axes
- Create Range Field Indexes for the Valid and System Axes
- Create System and Valid Axes
- Create a Temporal Collection
- Insert Some Temporal Documents
- Run some Search Queries on the Temporal Documents

2.1 Create Metadata Fields for the Valid and System Axes

The valid and system axis each make use of metadata fields that define the start and end times. For example, the following query creates the metadata fields to be used to store the valid and system axes.

JavaScript Example:

```
var admin = require("/MarkLogic/admin.xqy");
var config = admin.getConfiguration();
var dbid = xdmp.database("Documents");

var validStart = admin.databaseMetadataField("validStart");
var validEnd = admin.databaseMetadataField("validEnd");
var systemStart = admin.databaseMetadataField("systemStart");
var systemEnd = admin.databaseMetadataField("systemEnd");

config = admin.databaseAddField(config, dbid, validStart);
config = admin.databaseAddField(config, dbid, validEnd);
config = admin.databaseAddField(config, dbid, systemStart);
config = admin.databaseAddField(config, dbid, systemEnd);

admin.saveConfiguration(config);
```

XQuery Example:

```
xquery version "1.0-ml";

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

let $config := admin:get-configuration()
let $dbid := xdmp:database("Documents")
let $fieldspec1 := admin:database-metadata-field("validStart")
let $fieldspec2 := admin:database-metadata-field("validEnd")
let $fieldspec3 := admin:database-metadata-field("systemStart")
let $fieldspec4 := admin:database-metadata-field("systemEnd")
for $fieldspec in ($fieldspec1, $fieldspec2, $fieldspec3, $fieldspec4)
let $new-config := admin:database-add-field($config, $dbid, $fieldspec)
return admin:save-configuration($new-config)
```

2.2 Create Range Field Indexes for the Valid and System Axes

The valid and system axis each make use of `dateTime` range field indexes that define the start and end times. For example, the following query creates the field range indexes to be used to create the valid and system axes.

JavaScript Example:

```
var admin = require("/MarkLogic/admin.xqy");
var config = admin.getConfiguration();
var dbid = xdmp.database("Documents");

var validStart = admin.databaseRangeFieldIndex(
    "dateTime", "validStart", "", fn.true() );
var validEnd   = admin.databaseRangeFieldIndex(
    "dateTime", "validEnd", "", fn.true() );
var systemStart = admin.databaseRangeFieldIndex(
    "dateTime", "systemStart", "", fn.true() );
var systemEnd   = admin.databaseRangeFieldIndex(
    "dateTime", "systemEnd", "", fn.true() );

config = admin.databaseAddRangeFieldIndex(config, dbid, validStart);
config = admin.databaseAddRangeFieldIndex(config, dbid, validEnd);
config = admin.databaseAddRangeFieldIndex(config, dbid, systemStart);
config = admin.databaseAddRangeFieldIndex(config, dbid, systemEnd);

admin.saveConfiguration(config);
```

XQuery Example:

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

let $config := admin:get-configuration()
let $dbid := xdmp:database("Documents")
let $rangespec1 := admin:database-range-field-index(
                    "dateTime", "validStart", "", fn:true() )
let $rangespec2 := admin:database-range-field-index(
                    "dateTime", "validEnd", "", fn:true() )
let $rangespec3 := admin:database-range-field-index(
                    "dateTime", "systemStart", "", fn:true() )
let $rangespec4 := admin:database-range-field-index(
                    "dateTime", "systemEnd", "", fn:true() )

for $rangespec in ($rangespec1, $rangespec2, $rangespec3, $rangespec4)
let $new-config := admin:database-add-range-field-index(
                    $config, $dbid, $rangespec)

return admin:save-configuration($new-config)
```

2.3 Create System and Valid Axes

On the Documents database, create two axes, named “valid” and “system,” each to serve as a container for a named pair of field range indexes.

JavaScript Example:

```
var temporal = require("/MarkLogic/temporal.xqy");

var validResult = temporal.axisCreate(
    "valid",
    cts.fieldReference("validStart", "type=dateTime"),
    cts.fieldReference("validEnd", "type=dateTime"));

var systemResult = temporal.axisCreate(
    "system",
    cts.fieldReference("systemStart", "type=dateTime"),
    cts.fieldReference("systemEnd", "type=dateTime"));
```

XQuery Example:

```
xquery version "1.0-ml";

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

temporal:axis-create(
    "valid",
    cts:field-reference("validStart", "type=dateTime"),
    cts:field-reference("validEnd", "type=dateTime"));

xquery version "1.0-ml";

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

temporal:axis-create(
    "system",
    cts:field-reference("systemStart", "type=dateTime"),
    cts:field-reference("systemEnd", "type=dateTime"))
```

2.4 Create a Temporal Collection

Create a temporal collection, named “kool,” that uses the previously created “system” and “valid” axes.

JavaScript Example:

```
var temporal = require("/MarkLogic/temporal.xqy");

var collectionResult = temporal.collectionCreate(
"kool", "system", "valid");
```

XQuery Example:

```
xquery version "1.0-ml";

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

temporal:collection-create("kool", "system", "valid")
```

Note: Axis and temporal collection names are case-sensitive.

2.5 Insert Some Temporal Documents

Insert some documents into the temporal collection. In this example, a stock trader, John, places an order to buy some stock. The record of the trade is stored as a bi-temporal document, as follows:

1. The stock of KoolCo is trading around $12.65. John places a limit order to buy 100 shares of the stock for $12 at 11:00:00 on 3-Apr-2014 (this is the valid start time). The document for the transaction is recorded in the broker’s database at 11:00:01 on 3-Apr-2014 (this is the system start time).

   JavaScript Example:

```
declareUpdate();
var temporal = require("/MarkLogic/temporal.xqy");
var root =
    {"tempdoc":
       {"trader": "John",
        "price": 12}
    };
var options =
    {metadata:
       {validStart: "2014-04-03T11:00:00",
        validEnd: "9999-12-31T11:59:59Z"}
    };
temporal.documentInsert("kool", "koolorder.json", root, options);
```

XQuery Example:

```
xquery version "1.0-ml";

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

let $root :=
<tempdoc>
    <trader>John</trader>
    <content>12</content>
</tempdoc>

let $options :=
<options xmlns="xdmp:document-insert">
    <metadata>
       <map:map xmlns:map="http://marklogic.com/xdmp/map">
         <map:entry key="validStart">
           <map:value>2014-04-03T11:00:00</map:value>
         </map:entry>
         <map:entry key="validEnd">
           <map:value>9999-12-31T11:59:59Z</map:value>
         </map:entry>
       </map:map>
    </metadata>
</options>

return
temporal:document-insert("kool", "koolorder.xml", $root, $options)
```

Note: You can use `xdmp.documentGetMetadata` to display the metadata for a temporal document. For example: `xdmp.documentGetMetadata("koolorder.json")`

2. At 11:30:00, John changes his order to buy the stock at $13. The change is recorded as another version in the broker's database at 11:30:01.

   JavaScript Example:

```
declareUpdate();
var temporal = require("/MarkLogic/temporal.xqy");
var root =
    {"tempdoc":
       {"trader": "John",
        "price": "13"}
    };
var options =
    {metadata:
       {validStart: "2014-04-03T11:30:00",
        validEnd: "9999-12-31T11:59:59Z"}
    };
temporal.documentInsert("kool", "koolorder.json", root, options);
```

   XQuery Example:

```
xquery version "1.0-ml";

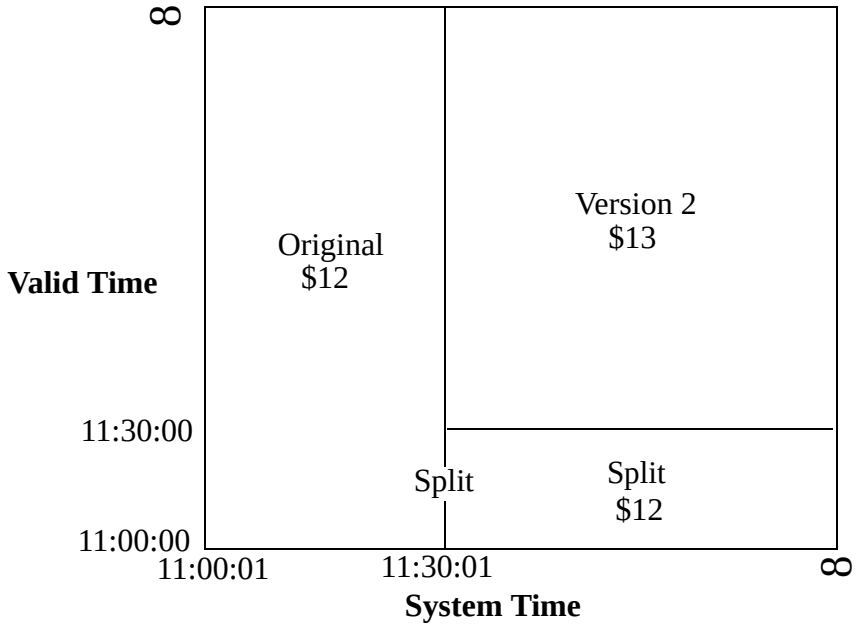
import module namespace temporal = "http://marklogic.com/xdmp/temporal"
    at "/MarkLogic/temporal.xqy";

let $root :=
<tempdoc>
    <trader>John</trader>
    <content>13</content>
</tempdoc>

let $options :=
<options xmlns="xdmp:document-insert">
    <metadata>
       <map:map xmlns:map="http://marklogic.com/xdmp/map">
         <map:entry key="validStart">
           <map:value>2014-04-03T11:30:00</map:value>
         </map:entry>
         <map:entry key="validEnd">
           <map:value>9999-12-31T11:59:59Z</map:value>
         </map:entry>
       </map:map>
    </metadata>
</options>

return
temporal:document-insert("kool", "koolorder.xml", $root, $options)
```

The result should be three documents with valid and system times as shown in the graphic below. Note that the second query resulted in a split on the Original document that resulted in a “split” document, as well as Version 2 that contains the new content.



2.6 Run some Search Queries on the Temporal Documents

The following query searches the temporal documents, using the `cts:period-range-query` function to locate the documents that were in the database between 11:10 and 11:15. `ISO_CONTAINS` is one of the comparison operators described in "ISO SQL 2011 Operators" on page 57.

In this example, only the Original Document meets the search criteria.

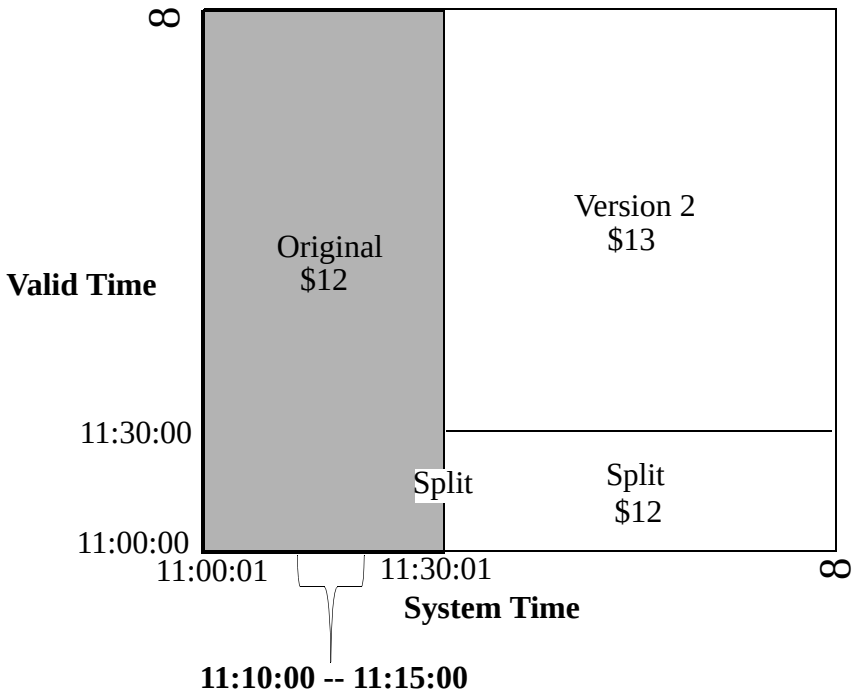
JavaScript Example:

```
cts.search(cts.periodRangeQuery(
    "system",
    "ISO_CONTAINS",
    cts.period(xs.dateTime("2014-04-03T11:10:00"),
               xs.dateTime("2014-04-03T11:15:00")) ));
```

XQuery Example:

```
xquery version "1.0-ml";

cts:search(fn:doc(), cts:period-range-query(
    "system",
    "ISO_CONTAINS",
    cts:period(xs:dateTime("2014-04-03T11:10:00"),
               xs:dateTime("2014-04-03T11:15:00")) ))
```



The following query searches the temporal documents, using the `cts:period-range-query` function to locate the documents that have a valid time period that starts after 10:30 and ends at 11:30. `ALN_FINISHES` is one of the comparison operators described in “Allen Operators” on page 54.

In this example, only the Split document meets the search criteria.

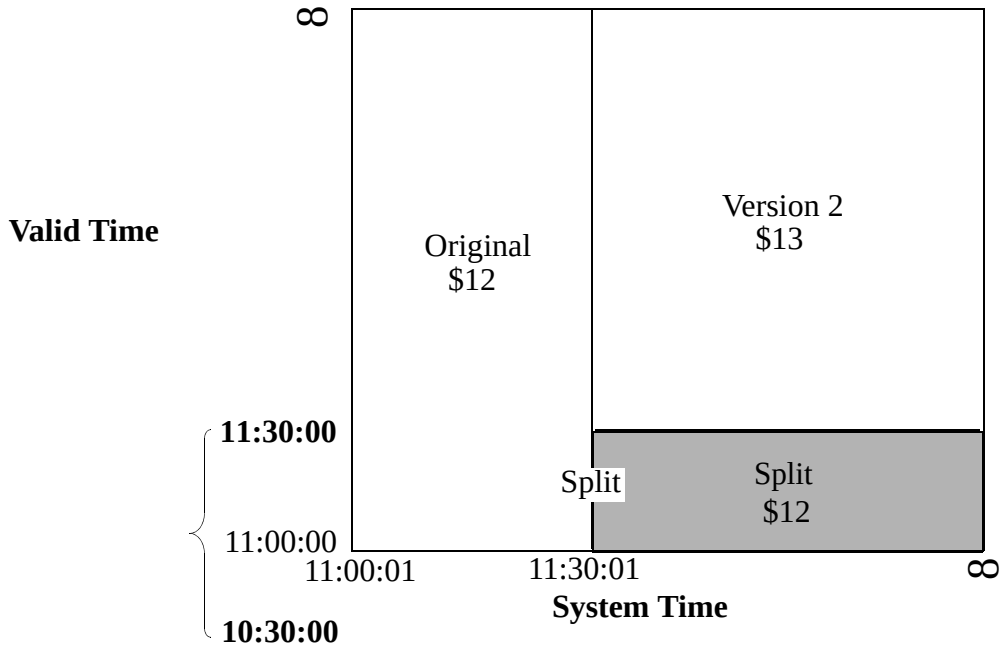
JavaScript Example:

```
cts.search(cts.periodRangeQuery(
    "valid",
    "ALN_FINISHES",
    cts.period(xs.dateTime("2014-04-03T10:30:00"),
               xs.dateTime("2014-04-03T11:30:00")) ));
```

XQuery Example:

```
xquery version "1.0-ml";

cts:search(fn:doc(), cts:period-range-query(
   "valid",
   "ALN_FINISHES",
   cts:period(xs:dateTime("2014-04-03T10:30:00"),
              xs:dateTime("2014-04-03T11:30:00")) ))
```



The following query searches the temporal documents, using the `cts:period-range-query` function to locate the documents that were in the database after 11:20. `ALN_AFTER` is one of the comparison operators described in “Allen Operators” on page 54.

In this example, both the Split and Version 2 documents meet the search criteria.

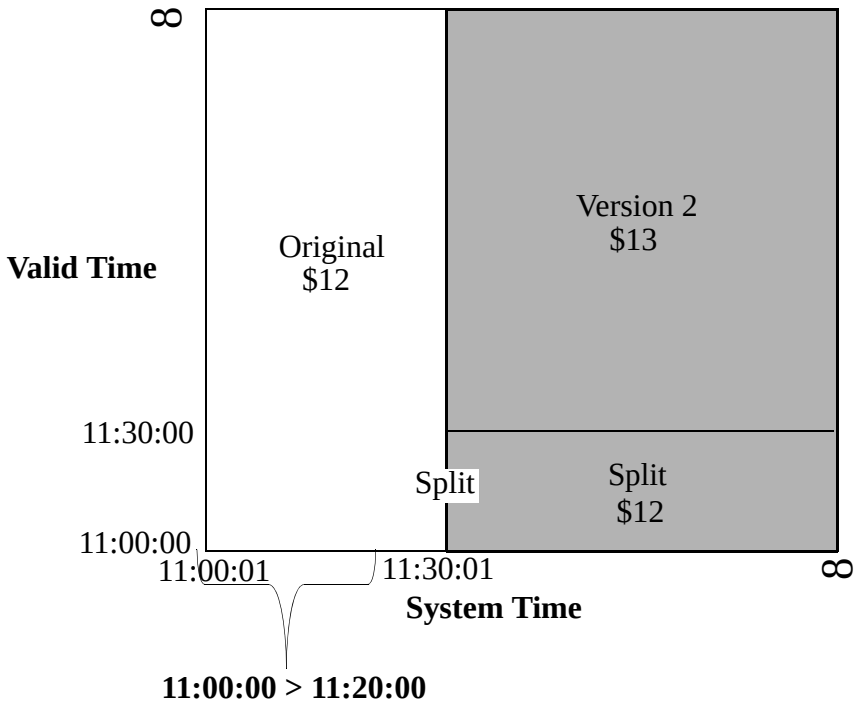
JavaScript Example:

```
cts.search(cts.periodRangeQuery(
    "system",
    "ALN_AFTER",
    cts.period(xs.dateTime("2014-04-03T11:00:00"),
               xs.dateTime("2014-04-03T11:20:00")) ));
```

XQuery Example:

```
xquery version "1.0-ml";

cts:search(fn:doc(), cts:period-range-query(
   "system",
   "ALN_AFTER",
   cts:period(xs:dateTime("2014-04-03T11:00:00"),
              xs:dateTime("2014-04-03T11:20:00")) ))
```



The following query searches the temporal documents, using the `cts:period-compare-query` function to locate the documents that were in the database when the valid time period is within the system time period. `ISO_CONTAINS` is one of the comparison operators described in “ISO SQL 2011 Operators” on page 57.

In this example, only Version 2 meets the search criteria.

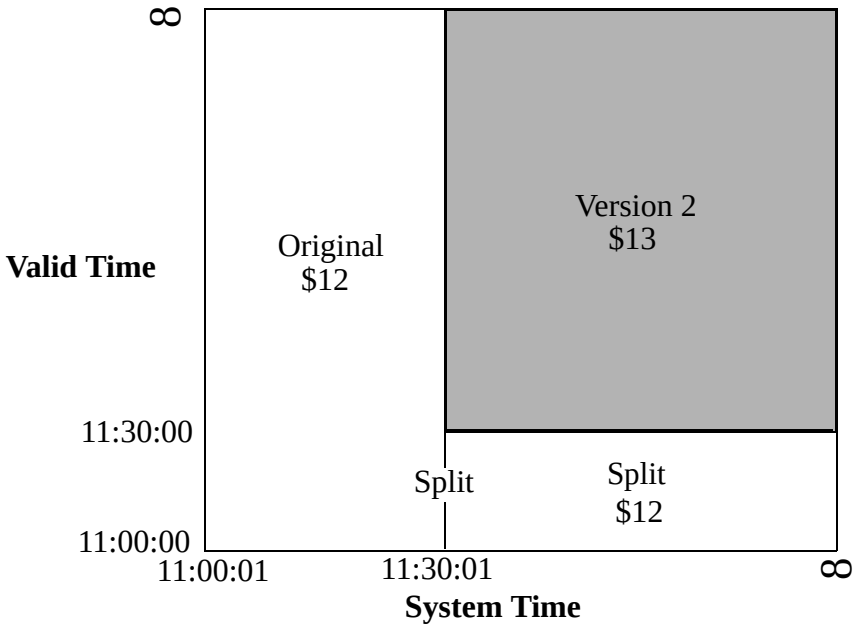
JavaScript Example:

```
cts.search(cts.periodCompareQuery(
    "system",
    "ISO_CONTAINS",
     "valid" ))
```

XQuery Example:

```
xquery version "1.0-ml";

cts:search(fn:doc(), cts:period-compare-query(
   "system",
   "ISO_CONTAINS",
   "valid" ))
```



The following query uses the `cts:and-query` to AND two `cts:collection-query` functions to return the temporal document that is in the URI collection, `koolorder.xml`, and the `latest` collection.

In this example, Ver2 meets the search criteria.

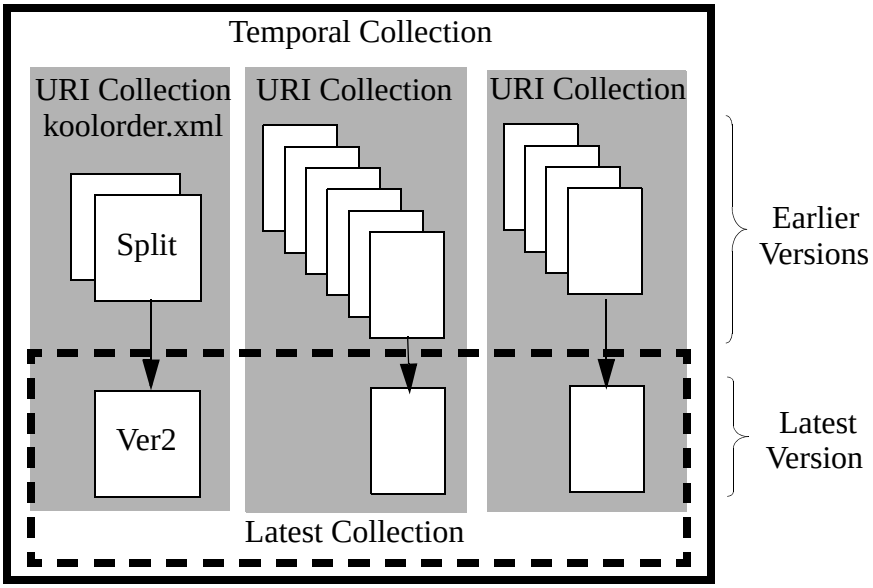
JavaScript Example:

```
cts.search(cts.andQuery([
    cts.collectionQuery("koolorder.json"),
    cts.collectionQuery("latest")]))
```

XQuery Example:

```
xquery version "1.0-ml";

cts:search(fn:doc(), cts:and-query((
    cts:collection-query(("koolorder.xml")),
    cts:collection-query(("latest")))))
```



3.0 Managing Temporal Documents

This chapter describes how to insert and update temporal documents, and includes the following sections:

- Creating Temporal Collections
- Inserting and Loading Temporal Documents
- Last Stable Query Time (LSQT) and Application-controlled System Time
- Using MarkLogic Content Pump (MLCP) to Load Temporal Documents
- Setting the URI for a New Version of a Temporal Document
- Protecting and Archiving Temporal Documents
- Deleting and Wiping Temporal Documents
- Example: The Lifecycle of a Temporal Document

3.1 Creating Temporal Collections

There are a number of ways to create a temporal collection, depending on how you want to structure your temporal documents. The fundamental types of temporal collections are:

- Bi-temporal Collections
- Uni-temporal Collections

3.1.1 Bi-temporal Collections

A bi-temporal collection is configured to store temporal documents that have both a vaild and system time axes. You can create a temporal collection to store these two axes in one of three ways:

- Both system and valid axes in metadata.
- Both system and valid axes in the document.
- Either system or valid axes in metadata and the other in the document.

The example procedures in the “Quick Start” on page 9 show how to create a collection and insert documents that store the system and valid axes in the metadata. The following sections describe how to create the other types of bi-termporal collections:

- Creating a Temporal Collection that Stores the System and Valid Axes in Documents
- Creating a Temporal Collection that Stores the System Axes in Metadata and the Valid Axes in the Document
- Creating a Temporal Collection for use with Flexible Replication

Note: The index, axes, and collection names are different for each procedure in order to accomodate all three temporal collections in the same database.

3.1.1.1 Creating a Temporal Collection that Stores the System and Valid Axes in Documents

To create a temporal collection that stores both the system and vaild axes in the document, do the following:

1. Create the range indexes for the valid and system axes. The valid and system axis each make use of `dateTime` range indexes that define the start and end times. For example, the following query creates the element range indexes to be used to create the valid and system axes.

```
const admin = require("/MarkLogic/admin.xqy");
let config = admin.getConfiguration();
const dbid = xdmp.database("Documents");

const validStart = admin.databaseRangeElementIndex(
    "dateTime", "", "valid-start", "", fn.false() );

const validEnd   = admin.databaseRangeElementIndex(
    "dateTime", "", "valid-end", "", fn.false() );

const systemStart = admin.databaseRangeElementIndex(
    "dateTime", "", "system-start", "", fn.false() );

const systemEnd   = admin.databaseRangeElementIndex(
    "dateTime", "", "system-end", "", fn.false() );

config = admin.databaseAddRangeElementIndex(config, dbid, validStart);
config = admin.databaseAddRangeElementIndex(config, dbid, validEnd);
config = admin.databaseAddRangeElementIndex(config, dbid,
systemStart);
config = admin.databaseAddRangeElementIndex(config, dbid, systemEnd);

admin.saveConfiguration(config);
```

2. Create system and valid axes. Temporal documents have both a valid and system axis. Create two axes, named “validAxes” and “systemAxes,” each to serve as a container for a named pair of range indexes.

JavaScript Example:

```
const temporal = require("/MarkLogic/temporal.xqy");

const validResult = temporal.axisCreate(
    "validAxes",
    cts.elementReference(fn.QName("", "valid-start")),
    cts.elementReference(fn.QName("", "valid-end")));

const systemResult = temporal.axisCreate(
    "systemAxes",
    cts.elementReference(fn.QName("", "system-start")),
    cts.elementReference(fn.QName("", "system-end")));
```

3. Create a temporal collection, named “kool,” that uses the previously created ““validAxes” and “systemAxes” axes.

   JavaScript Example:

```
const temporal = require("/MarkLogic/temporal.xqy");

const collectionResult = temporal.collectionCreate(
"kool", "systemAxes", "validAxes");
```

   Note: Axis and temporal collection names are case-sensitive.

4. Insert a document into the temporal collection to test.

```
const temporal = require("/MarkLogic/temporal.xqy");

const root =
    { "tempdoc": {
      "system-start": null,
      "system-end": null,
      "valid-start": "2014-04-03T11:00:00",
      "valid-end": "2014-04-03T16:00:00",
      "trader": "John",
      "price": 12
      }
    };

declareUpdate();
temporal.documentInsert("kool", "koolorder.json", root);
```

   Note: Unlike when the system axes is stored in metadata, you must include the system start and end elements or properties when they are stored in the temporal document. They can either be null or include a timestamp, which will be reset by MarkLogic.

3.1.1.2 Creating a Temporal Collection that Stores the System Axes in Metadata and the Valid Axes in the Document

To create a temporal collection that stores the system axes in metadata and the vaild axes in the document, do the following:

For example, the following query creates the metadata fields to be used to create the valid and system axes.

1. Create the range indexes for the valid axes.

```
const admin = require("/MarkLogic/admin.xqy");
let config = admin.getConfiguration();
const dbid = xdmp.database("Documents");
const validStart = admin.databaseRangeElementIndex(
    "dateTime", "", "val-start", "", fn.false() );
const validEnd   = admin.databaseRangeElementIndex(
    "dateTime", "", "val-end", "", fn.false() );
config = admin.databaseAddRangeElementIndex(config, dbid, validStart);
config = admin.databaseAddRangeElementIndex(config, dbid, validEnd);

admin.saveConfiguration(config);
```

2. Create metadata fields for the system axes.

```
const admin = require("/MarkLogic/admin.xqy");
let config = admin.getConfiguration();
const dbid = xdmp.database("Documents");
const sysStart = admin.databaseMetadataField("sys-start");
const sysEnd   = admin.databaseMetadataField("sys-end");
config = admin.databaseAddField(config, dbid, sysStart);
config = admin.databaseAddField(config, dbid, sysEnd);

admin.saveConfiguration(config);
```

3. Create range field indexes for the system axes.

```
const admin = require("/MarkLogic/admin.xqy");
let config = admin.getConfiguration();
const dbid = xdmp.database("Documents");
const systemStart = admin.databaseRangeFieldIndex(
    "dateTime", "sys-start", "", fn.true() );
const systemEnd   = admin.databaseRangeFieldIndex(
    "dateTime", "sys-end", "", fn.true() );
config = admin.databaseAddRangeFieldIndex(config, dbid, systemStart);
config = admin.databaseAddRangeFieldIndex(config, dbid, systemEnd);
admin.saveConfiguration(config);
```

4. Create system and valid axes from the indexes.

```
const temporal = require("/MarkLogic/temporal.xqy");
const validResult = temporal.axisCreate(
    "val-axes",
    cts.elementReference(fn.QName("", "val-start")),
    cts.elementReference(fn.QName("", "val-end")));
const systemResult = temporal.axisCreate(
    "sys-axes",
    cts.fieldReference("sys-start", "type=dateTime"),
    cts.fieldReference("sys-end", "type=dateTime"));
```

5. Create a temporal collection that uses the system and valid axes.

```
const temporal = require("/MarkLogic/temporal.xqy");
const collectionResult = temporal.collectionCreate(
"hybred", "sys-axes", "val-axes");
```

6. Insert a document into the temporal collection to test.

```
const temporal = require("/MarkLogic/temporal.xqy");
const root =
    { "tempdoc": {
      "val-start": "2014-04-03T11:00:00",
      "val-end": "2014-04-03T16:00:00",
      "trader": "John",
      "price": 12
      }
    };
declareUpdate();
temporal.documentInsert("hybred", "hybred.json", root);
```

3.1.1.3 Creating a Temporal Collection for use with Flexible Replication

If you plan to replicate bi-termoral documents using the Flexible Replication feature described in the *Flexible Replication Guide*, you must create the range indexes, axes, and temporal collection in target database. When creating the temporal collection, do the following:

Create the temporal collection with override priveleges:

- If you have the `admin` role, specify the `updates-admin-override` option when executing the `temporal:collection-create` function.
- If you do not have the `admin` role, specify the `override-priv` option when executing the `temporal:collection-create` function. For example: `override-priv=temporal:override`

  Then assign the override privilege to the Flexible Replication user's role. For example:

```
sec:create-privilege('temporal:override',
            'http://marklogic.com/xdmp/privileges/temporal:override',
            'execute', 'flexrep-user') "
```

3.1.2 Uni-temporal Collections

A uni-temporal collection is configured to store temporal documents that have only a system time axis. You can create a temporal collection to store system axes in one of two ways:

- The system axes in metadata.
- The system axes in the document.

The two examples below make use of the axes created in the examples previously described in “Bi-temporal Collections” on page 23.

To create a uni-temporal collection that stores the system axes in the metadata:

```
const temporal = require("/MarkLogic/temporal.xqy");

const collectionResult = temporal.collectionCreate(
"uniTemp1", "sys-axes");
```

Insert a document to test:

```
declareUpdate();
const temporal = require("/MarkLogic/temporal.xqy");
const root =
    {"tempdoc":
       {"content": "content here"}
    };
temporal.documentInsert("uniTemp1", "doc.json", root);
```

To create a uni-temporal collection that stores the system axes in the document:

```
const temporal = require("/MarkLogic/temporal.xqy");

const collectionResult = temporal.collectionCreate(
"uniTemp2", "systemAxes");
```

Insert a document to test:

```
declareUpdate();
const temporal = require("/MarkLogic/temporal.xqy");
const root =
    {"tempdoc":
       {"content": "content here"}
    };
const options =
    {metadata:
       {systemStart: "1601-01-01T13:59:00Z",
        systemEnd: "9999-12-31T11:59:59Z"}
    };
temporal.documentInsert("uniTemp2", "doc.json", root, options);
```

3.2 Inserting and Loading Temporal Documents

There are a number of ways to insert and update temporal documents in MarkLogic Server. These include:

- The XQuery functions, `temporal:document-insert` and `temporal:document-load`.
- The REST Client API resource addresses, `POST:/v1/documents` and `PUT:/v1/documents` with the `temporal-collection` parameter.
- The Java Client API, as described in Working with Temporal Documents in the *Java Application Developer's Guide*.
- The Node.js Client API, through the `documents` interface. For details, see the *Node.js Client API Reference*.
- The MarkLogic Content Pump (`mlcp`), as described in "Using MarkLogic Content Pump (MLCP) to Load Temporal Documents" on page 35.
- XCC `ContentCreateOptions` class functions: `getTemporalCollection` and `setTemporalCollection` to get and set the temporal collection object associated with the inserted documents.
- Hadoop Connector `mapreduce.marklogic.output.temporalcollection` property to specify the temporal collection that is used to ingest the documents.

Though MarkLogic manages temporal documents in the same manner regardless of the tool you use, this section describes the use of the XQuery functions, `temporal:document-insert` and `temporal:document-load`, to insert and update temporal documents into MarkLogic Server.

Calling either `temporal:document-insert` or `temporal:document-load` on an existing URI "updates" the existing temporal document. An update on a temporal document results in a new document, rather than an overwrite of the original document.

You can use `xdmp:document-set-properties` to set properties on temporal documents, but you cannot use the `xdmp:document-set-quality`, `xdmp:document-set-permissions`, or `xdmp:document-set-collections` functions to set their respective attributes on temporal documents.

As described in "Bi-temporal Collections" on page 23. the bi-temporal document being inserted or updated must specify valid start and end times either in the document or in metadata. When the system start and end times are stored in the document, they also must be included. These times must be `dateTime` values identified by elements that map to the range indexes that represent the valid start and end time period. On insert, MarkLogic sets the system start time to the current time system time and the system end time to the farthest possible time (infinity).

If you have enabled LSQT, as described in “Last Stable Query Time (LSQT) and Application-controlled System Time” on page 32, you can alternatively have your application call the `temporal:statement-set-system-time` function to specify a system start time along with your insert. The `dateTime` given must be later than the LSQT returned by the `temporal:get-lsqt` function.

JavaScript Example:

1. Set LSQT for the temporal collection.

```
declareUpdate();
const temporal = require("/MarkLogic/temporal.xqy");
temporal.setUseLsqt("kool", true);
```

2. Get the LSQT.

```
const temporal = require("/MarkLogic/temporal.xqy");
temporal.getLsqt("kool");
```

3. On document insert, set the system start time to some time after the LSQT.

```
declareUpdate();
const temporal = require("/MarkLogic/temporal.xqy");
const root =
    {"tempdoc":
       {"content": "content here"}
    };
const options =
    {metadata:
       {validStart: "2016-06-01T13:59:00Z",
        validEnd: "9999-12-31T11:59:59Z"}
    };
temporal.documentInsert("kool", "doc.json", root, options);
temporal.statementSetSystemTime(xs.dateTime("2016-06-01T14:00:00Z"));
```

XQuery Example:

1. Set LSQT for the temporal collection.

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

temporal:set-use-lsqt("kool", fn:true())
```

2. Get the LSQT.

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

temporal:get-lsqt("kool")
```

3. On document insert, set the system start time to some time after the LSQT.

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

let $root :=
<tempdoc>
    <content>v1-content here</content>
</tempdoc>

let $options :=
<options xmlns="xdmp:document-insert">
    <metadata>
       <map:map xmlns:map="http://marklogic.com/xdmp/map">
         <map:entry key="validStart">
           <map:value>2016-06-01T13:59:00Z</map:value>
         </map:entry>
         <map:entry key="validEnd">
           <map:value>9999-12-31T11:59:59Z</map:value>
         </map:entry>
       </map:map>
    </metadata>
</options>

return
temporal:document-insert("kool", "koolorder.xml", $root, $options),
temporal:statement-set-system-time(xs:dateTime("2016-06-01T14:00:00Z")
)
```

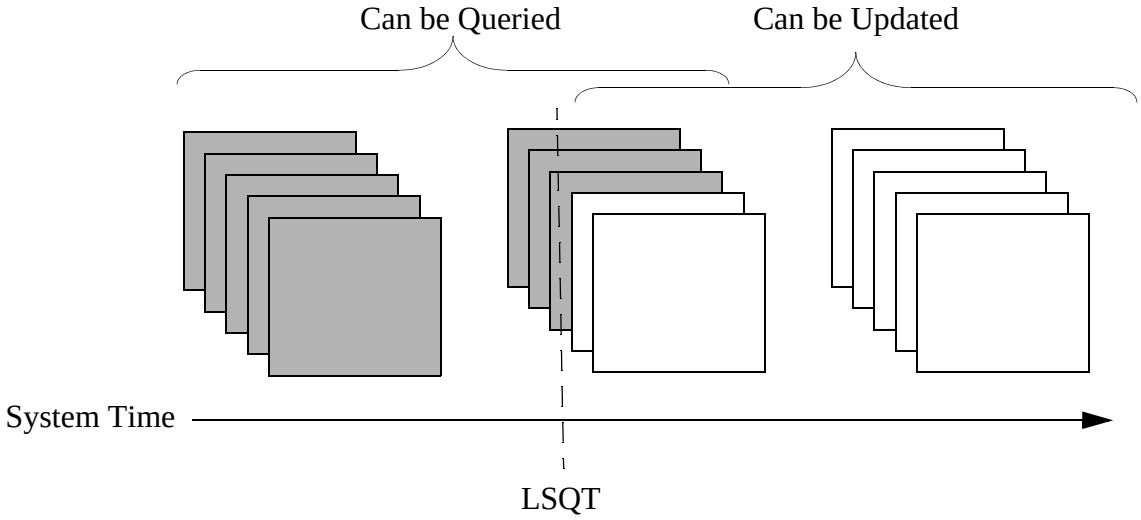
Note: Document properties are not updated on temporal documents, so do not use Content Processing Framework (CPF) or Library Services (DLS) on temporal data.

3.3 Last Stable Query Time (LSQT) and Application-controlled System Time

You can manually set the system start time when inserting or updating a document in a collection. This feature is useful when you need to maintain a "master" system time across multiple clients that are concurrently inserting and updating bi-temporal documents, without the need for the clients to communicate with one another in order to coordinate their system times.

The system start times for document versions with the same URI must progress along the system time axis, so that an update to a document cannot have a system start time that is earlier than that of the document that chronicles its last update. However, when managing documents with different URIs in a temporal collection, it is necessary to ensure that the system time progresses at the same rate for every document insert and update.

A special timestamp, called the LSQT (Last Stable Query Time), can be enabled on a temporal collection to manage system start times across documents with different URIs. A temporal document with a system start time before the LSQT can only be queried and a document with a system start time after the LSQT can be updated / ingested, but not queried. You can advance the LSQT, either manually or automatically, to manage which documents are available to be queried and which documents can be updated.



You can use the `temporal:set-use-lsqt` function to enable or disable LSQT on a temporal collection. When LSQT is enabled, the LSQT is stored in a document in the database, with a name of the form *`collection-name`*`.lsqt`. You can call the `temporal:advance-lsqt` function to manually advance the LSQT or use the `temporal:set-lsqt-automation` function to direct MarkLogic to automatically advance the LSQT at set periods.

When LSQT is enabled on a temporal collection, the LSQT value starts at 0 (lowest timestamp) When advanced, document reads and writes are quiesced until the LSQT is reset to the maximum system start time in the database. You must have LSQT enabled in order to use the `temporal:statement-set-system-time` function to set the system start time.

> **Note:** You can only call the `temporal:statement-set-system-time` function once per statement

For example, the following query first checks to make sure the application time (simulated by the current time) is greater than the LSQT:

JavaScript Example:

```
declareUpdate();
const temporal = require("/MarkLogic/temporal.xqy");

const curtime = fn.currentDateTime();
const lsqt = temporal.getLsqt("kool");
const root =
    {"tempdoc":
       {"content": "12"}
    };
const options =
    {metadata:
       {validStart: "2016-06-20T14:06:13Z",
        validEnd: "9999-12-31T11:59:59Z"}
    };

temporal.documentInsert("kool", "test.json", root, options);
if(curtime > lsqt) {
      temporal.statementSetSystemTime(xs.dateTime(curtime));}
else {fn.concat("Can't update document with specified system ",
                "time because it is earlier than the set LSQT")};
```

XQuery Example:

```
xquery version "1.0-ml";

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

let $curtime := fn:current-dateTime()
let $LSQT := temporal:get-lsqt("kool")

let $root :=
<tempdoc>
    <content>v1-content here</content>
</tempdoc>

let $options :=
<options xmlns="xdmp:document-insert">
    <metadata>
      <map:map xmlns:map="http://marklogic.com/xdmp/map">
         <map:entry key="validStart">
           <map:value>2016-06-20T14:06:13Z</map:value>
         </map:entry>
         <map:entry key="validEnd">
           <map:value>9999-12-31T11:59:59Z</map:value>
         </map:entry>
      </map:map>
    </metadata>
</options>

let $systemTime :=
      if ($curtime > $LSQT)
      then (temporal:statement-set-system-time(xs:dateTime($curtime)))
      else (fn:concat("Can't update document with specified system",
                      "time because it is earlier than the set LSQT"))

return (
   temporal:document-insert(
     "kool", "doc.xml", $root, $options),
     $systemTime )
```

3.4 Using MarkLogic Content Pump (MLCP) to Load Temporal Documents

You can use the MarkLogic Content Pump (MLCP) to import documents into MarkLogic as part of a temporal collection, copy non-temporal documents in one database into a temporal collection in another database, or copy temporal documents from one database to another. See the following topics for details:

- Importing Documents Into a Temporal Collection
- Copying Non-Temporal Documents Into a Temporal Collection
- Copying Temporal Documents Between Databases

3.4.1 Importing Documents Into a Temporal Collection

Use the mlcp `import` command with the `-temporal_collection` option to insert documents into a temporal collections with mlcp.

Note: You can only import a binary document as a temporal document if the temporal collection is uni-temporal and the system time axis is stored in the metadata.

Note: If you use MLCP to load temporal documents, the valid start and end times must be in the documents. The system start and end time will be filled in by MarkLogic, and can be either in metadata or in the document root node.

For example, to import the temporal documents in the `/etc/orders` directory on the filesystem into the temporal collection, named "kool," into the temporal database on the host, `desthost`, you would use the following MLCP command:

```
mlcp.sh import -temporal_collection kool -input_file_path /etc/orders \
-host desthost -port 8006 -username user1
```

Note: If you omit `-port`, MLCP will use port 8000.

For details on using MLCP to load documents, see Loading Content Using MarkLogic Content Pump in the *Loading Content Into MarkLogic Server Guide*.

3.4.2 Copying Non-Temporal Documents Into a Temporal Collection

Use the mlcp `copy` command with the `-temporal_collection` option to copy *non-temporal* documents from one database to another and insert them into a temporal collection in the destination database. To copy temporal documents from one database to another, see "Copying Temporal Documents Between Databases" on page 36.

Note: You can only copy a binary document as a temporal document if the temporal collection in the destination database is uni-temporal and the system time axis is stored in the metadata.

For example to migrate the non-temporal documents from the database used by the host, `srchost`, to the temporal collection named “kool” in the database used by the host, `desthost`, you use a command like the following:

```
mlcp.sh copy -mode local -input_host srchost -input_port 8006 \
-input_username user1 -input_password password1 \
-output_host desthost -output_port 8010 -temporal_collection kool \
-output_username user2 -output_password password2
```

Note: If you omit `-port`, MLCP will use port 8000.

For more details, see Copying Content Between Databases in the *mlcp User Guide*.

3.4.3 Copying Temporal Documents Between Databases

This section describes how to use mlcp to copy *temporal* documents in one database into a temporal collection in another database. To copy *non-temporal* documents in one database into a temporal collection in another database, see “Copying Non-Temporal Documents Into a Temporal Collection” on page 35.

Note: You can only copy a binary document as a temporal document if the temporal collection is uni-temporal and the system time axis is stored in the metadata.

The procedure for copying temporal documents between databases differs, depending on whether or not the temporal collection already exists in the destination database.

If the temporal collection of the source documents exists in the destination database, then your mlcp `copy` command must satisfy the following guidelines:

- Ensure the destination temporal collection is configured with an override privilege. Use the `override-priv` option of `temporal:collection-create` or `temporal.collectionCreate` to specify the override privilege for a temporal collection.
- Set the mlcp option `-output_user` to a user with the configured override privilege.
- Ensure the mlcp option `-copy_collections` is true.
- Do not use the mlcp option `-temporal_collection`.

For example, if `outuser` is a user with the override privilege, then you can use a command similar to the following to copy documents in a temporal collection on `srchost` into the same temporal collection on `desthost`. The collection must already exist on both hosts.

```
mlcp.sh copy -mode local -input_host srchost -input_port 8006 \
-input_username inuser -input_password inpassword \
-output_host desthost -output_port 8010 -copy_collections true \
-output_username outuser -output_password outpassword
```

If the temporal collection does not already exist in the destination database, then your mlcp `copy` command must satisfy the following guidelines:

- Do not use the mlcp option `-temporal_collection`.
- Ensure the mlcp option `-copy_collections` is true.
- After the copy completes, create the temporal collection in the destination database, with the `allow-nonempty` option of `temporal:collection-create` or `temporal.collectionCreate`.

For example, if you use a command such as the following to copy temporal documents from one database into another database that does not have a temporal collection named “kool”:

```
mlcp.sh copy -mode local -input_host srchost -input_port 8006 \
  -input_username inuser -input_password inpassword \
  -output_host desthost -output_port 8010 -copy_collections true \
  -output_username my_override_user -output_password outpassword \
  -copy_collections true
```

Then, after the job completes, you could create the “kool” temporal collection using a command similar to the following:

```
(: XQuery :)
xquery version "1.0-ml";
import module namespace temporal = "http://marklogic.com/xdmp/temporal"
    at "/MarkLogic/temporal.xqy";
temporal:collection-create("temporalCollection", "system", "valid"
                           ("allow-nonempty"))

// Server-Side JavaScript
var temporal = require('/MarkLogic/temporal.xqy');
temporal.collectionCreate('kool', 'system', 'valid',
                          ['allow-nonempty']);
```

For more details, see Copying Content Between Databases in the *mlcp User Guide*.

3.5 Setting the URI for a New Version of a Temporal Document

By default, MarkLogic creates a new URI of the form `filename[number].extension` for each new version of a temporal document. You can optionally set the URI when updating a temporal document by calling the `temporal:statement-set-document-version-uri` or `temporal.statementSetDocumentVersionUri` function.

For example, the initial version (the URI provided during the initial insert) of a temporal document has the following URI:

```
important.json
```

And you want the new version of the temporal document to have the following URI:

```
/version2/important.json
```

You could call the `temporal.statementSetDocumentVersionUri` function as follows:

```
declareUpdate();
const temporal = require("/MarkLogic/temporal.xqy");

const root =
    {"tempdoc":
       {"content": "content here"}
    };

const options =
    {metadata:
       {validStart: "1601-01-01T13:59:00Z",
        validEnd: "9999-12-31T11:59:59Z"}
    };

temporal.statementSetDocumentVersionUri(
     "important.json",
     "/version2/important.json")

temporal.documentInsert("kool", "important.json", root, options);
```

3.6 Protecting and Archiving Temporal Documents

Temporal documents can be protected from certain temporal operations, such as update, delete or wipe for a specified period of time. You can use the `temporal:document-protect` function to set the duration of protection, when that protection will expire (overrides duration, if different), and where to archive the temporal document.

There are three levels of protection, listed below in the descending order of restrictive level:

- `no-update` — do not allow any modification of the archived document. This includes `no-delete` and `no-wipe`.
- `no-delete` — do not allow a temporal delete document operation to delete any version of the document. This includes `no-wipe`. This is the default, if you do not specify a protection type.
- `no-wipe` — do not allow a wipe operation to delete all versions of the document.

When setting protection on a temporal document, you can specify a path to an archive, such as a WORM device. As described in “WORM (Write Once Read Many) Support” on page 8, once written to a WORM device, the temporal document can be read, but cannot be modified or deleted until its expiration date.

The archive settings are set by an XQuery, Javascript, or REST temporal document protect operation. For example, to set the protection level to `no-update` with an expire time to 2:00pm on 7/20/2016 on the `doc.xml` document in the temporal collection, `kool`, do the following:

```
declareUpdate();
const temporal = require("/MarkLogic/temporal.xqy");

temporal.documentProtect ("kool", "doc.xml",
    {level: "noUpdate", expireTime: "2016-07-20T14:00:00Z"})
```

3.7 Deleting and Wiping Temporal Documents

You can use the `temporal:document-delete` function to delete temporal documents. Deleting a temporal document maintains the document and all of its versions in the URI collection and updates the deleted document and all of its versions that have a system end time of infinity to the time of the delete.

Deleting a temporal document removes the document from the `latest` collection. So the `latest` collection is the source of all of the documents that are currrently valid and the URI collections are the source of the history of each document.

Should you insert a document using the same URI as a deleted document, the deleted document, and all of its previous versions remain in the same URI collection as the "newly" inserted document. The newly inserted document is then added to the `latest` collection.

You can also use the `temporal:document-wipe` function to remove all versions of a temporal document that has "expired." The expiration date for a document is set by the `temporal:document-protect` function, as described in "Protecting and Archiving Temporal Documents" on page 39.

> **Note:** Before all of the versions of a temporal document can be wiped, the current version of the temporal document must first have an expire time set by the `temporal.documentProtect` function.

For example, to wipe the `doc.xml` document when its protection time has expired, do the following:

```
declareUpdate();
const temporal = require("/MarkLogic/temporal.xqy");

const curtime = fn.currentDateTime()
const exptime = xs.dateTime(
        xdmp.documentGetMetadataValue("doc.xml","temporalProtectExTime"))
if (exptime.lt(curtime)) {temporal.documentWipe("kool", "doc.xml")}
else {"This document is protected or does not exist"}
```

3.8 Example: The Lifecycle of a Temporal Document

The example in this section builds on the example described in “Quick Start” on page 9. The purpose of this example is to show how new versions of a temporal document are generated and updated from a series of changes to a stock purchase order.

1. The stock of KoolCo is trading around $12.65. John places a limit order to buy 100 shares for $12 at 11:00:00 on 3-Apr-2014 (this is the valid start time). A temporal document for the order is recorded in the broker’s database at 11:00:01 on 3-Apr-2014 (this is the system start time).

2. At 11:30:00, John changes his order to buy the stock at $13. The change is recorded as another version in the broker’s database at 11:30:01.

3. At 12:10:00, John changes his mind again and decides to change his order to a limit order to buy at $12.50. This transaction is recorded as another version with a valid time of 12:10:00, but due to heavy trading, the change is not recorded in the broker’s database until 12:10:12.

4. At 13:00:00, the purchase order has not been filled and John decides he no longer wants to buy the stock, so he cancels his order. This cancellation is recorded as another version with a valid time of 13:00:00 and recorded in the broker’s database at 13:00:02.

5. However, at 13:00:01, the stock hits $12.50 and John’s order is filled.

6. The broker’s policy is to honor the valid times for all orders. At 13:00:03, the order fulfillment application reviews the valid and system times recorded at the time of the cancellation, determines that John in fact cancelled his order before it was filled, and does not debit his account for the stock purchase.

The valid and system times are each `dateTime` ranges that define a start and end time. The start time represents the time at which the information is known (as both valid and system times) and the end time represents the time at which the information is no longer true.

The above stock purchase example was kept simple for clarity. The following shows the insert query and the resulting documents with the actual valid and system times for the example, along with their respective start and end times. The graphic at the end displays the relationships between the documents in terms of valid and system times.

11:00:00 -- Initial order:

Insert Query (JavaScript):

```
declareUpdate();
const temporal = require("/MarkLogic/temporal.xqy");
const root =
    {"tempdoc":
       {"order 1": "12"}
    };
const options =
    {metadata:
       {validStart: "2014-04-03T11:00:00",
        validEnd: "9999-12-31T11:59:59Z"}
    };
temporal.documentInsert("kool", "koolorder.json", root, options);
```

Insert Query (XQuery):

```
xquery version "1.0-ml";

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

let $root :=
<tempdoc>
    <order1>12</order1>
</tempdoc>

let $options :=
<options xmlns="xdmp:document-insert">
    <metadata>
       <map:map xmlns:map="http://marklogic.com/xdmp/map">
         <map:entry key="validStart">
           <map:value>2014-04-03T11:00:00</map:value>
         </map:entry>
         <map:entry key="validEnd">
           <map:value>9999-12-31T11:59:59Z</map:value>
         </map:entry>
       </map:map>
    </metadata>
</options>

return
temporal:document-insert("kool", "koolorder.xml", $root, $options)
```

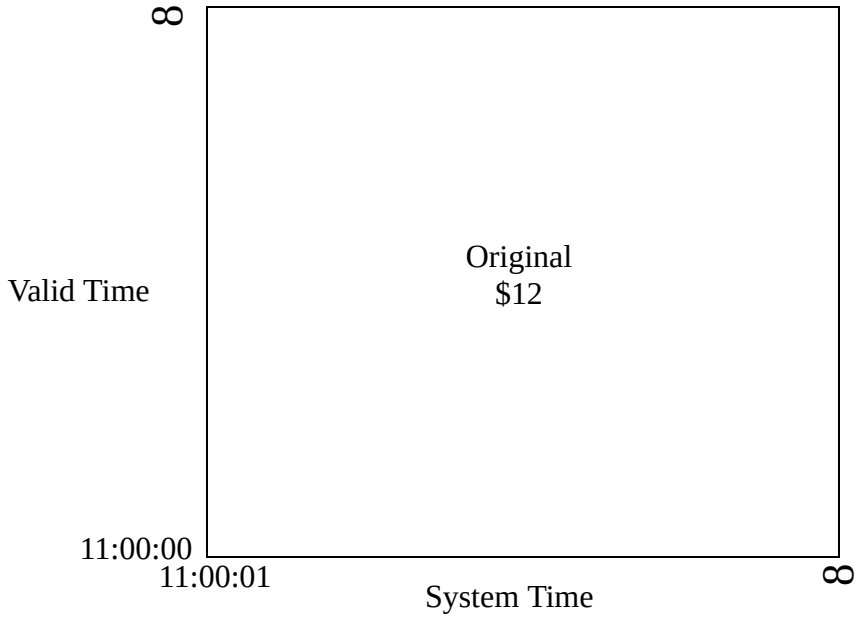
Results:

Original Document:

```
Order Price: $12

System Start: 2014-04-03T11:00:01
System End: 9999-12-31T11:59:59Z <-- infinity

Valid Start: 2014-04-03T11:00:00
Valid End: 9999-12-31T11:59:59Z <-- infinity
```



Note: The date-time `9999-12-31T11:59:59Z` represents infinity and is used when there is no valid date-time yet for that start or end time.

11:30:00 -- Changed order from $12 to $13.

Update Query (JavaScript):

```
declareUpdate();
const temporal = require("/MarkLogic/temporal.xqy");
const root =
    {"tempdoc":
       {"order 2": "13"}
    };
const options =
    {metadata:
       {validStart: "2014-04-03T11:30:00",
        validEnd: "9999-12-31T11:59:59Z"}
    };
temporal.documentInsert("kool", "koolorder.json", root, options);
```

Update Query (XQuery):

```
xquery version "1.0-ml";

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

let $root :=
<tempdoc>
    <order2>13</order2>
</tempdoc>

let $options :=
<options xmlns="xdmp:document-insert">
    <metadata>
       <map:map xmlns:map="http://marklogic.com/xdmp/map">
         <map:entry key="validStart">
           <map:value>2014-04-03T11:30:00</map:value>
         </map:entry>
         <map:entry key="validEnd">
           <map:value>9999-12-31T11:59:59Z</map:value>
         </map:entry>
       </map:map>
    </metadata>
</options>

return
temporal:document-insert("kool", "koolorder.xml", $root, $options)
```

Results:

Original Document (updated):

```
Order Price: $12

System Start: 2014-04-03T11:00:01
System End: 2014-04-03T11:30:01 <-- changed

Valid Start: 2014-04-03T11:00:00
Valid End: 9999-12-31T11:59:59Z <-- infinity
```

Split 1 from Original Document:

```
Order Price: $12

System Start: 2014-04-03T11:30:01
System End: 9999-12-31T11:59:59Z <-- infinity

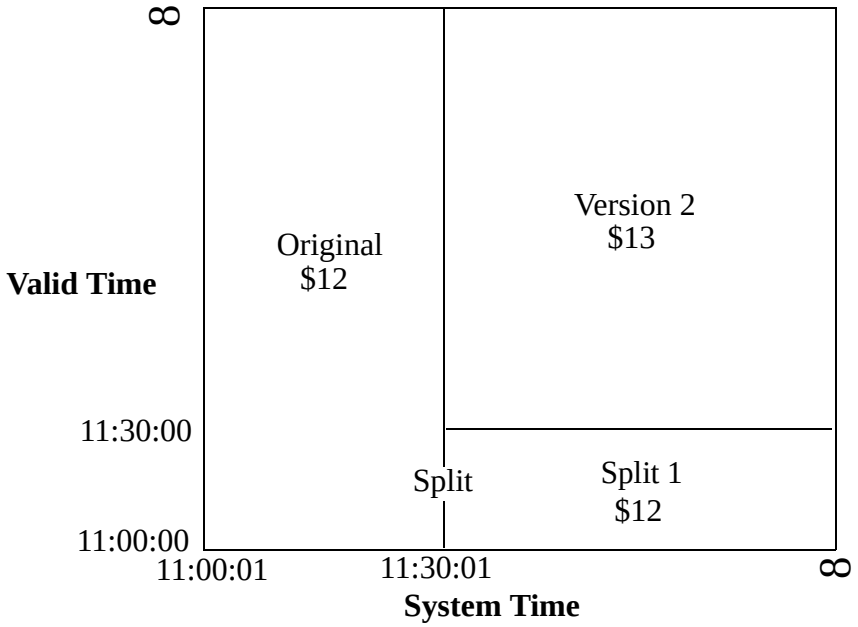
Valid Start: 2014-04-03T11:00:00
Valid End: 2014-04-03T11:30:00
```

Version 2:

```
Order Price: $13

System Start: 2014-04-03T11:30:01
System End: 9999-12-31T11:59:59Z <-- infinity

Valid Start: 2014-04-03T11:30:00
Valid End: 9999-12-31T11:59:59Z <-- infinity
```



12:10:00 -- Changed order from $13 to $12:50:

Update Query (JavaScript):

```
declareUpdate();
const temporal = require("/MarkLogic/temporal.xqy");
const root =
    {"tempdoc":
       {"order 3": "12.50"}
    };
const options =
    {metadata:
       {validStart: "2014-04-03T12:10:00",
        validEnd: "9999-12-31T11:59:59Z"}
    };
temporal.documentInsert("kool", "koolorder.json", root, options);
```

Update Query (XQuery):

```
xquery version "1.0-ml";

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

let $root :=
<tempdoc>
    <order3>12.50</order3>
</tempdoc>

let $options :=
<options xmlns="xdmp:document-insert">
    <metadata>
       <map:map xmlns:map="http://marklogic.com/xdmp/map">
         <map:entry key="validStart">
           <map:value>2014-04-03T12:10:00</map:value>
         </map:entry>
         <map:entry key="validEnd">
           <map:value>9999-12-31T11:59:59Z</map:value>
         </map:entry>
       </map:map>
    </metadata>
</options>

return
temporal:document-insert("kool", "koolorder.xml", $root, $options)
```

Results:

Original Document (no change)

Split 1 (no change)

Version 2 (update):

```
Order Price: Closing price

System Start: 2014-04-03T11:30:01
System End: 2014-04-03T12:10:12 <-- changed

Valid Start: 2014-04-03T11:30:00
Valid End: 9999-12-31T11:59:59Z <-- infinity
```

Split 2 (new)

```
Order Price: $12

System Start: 2014-04-03T12:10:12
System End: 9999-12-31T11:59:59Z <-- infinity

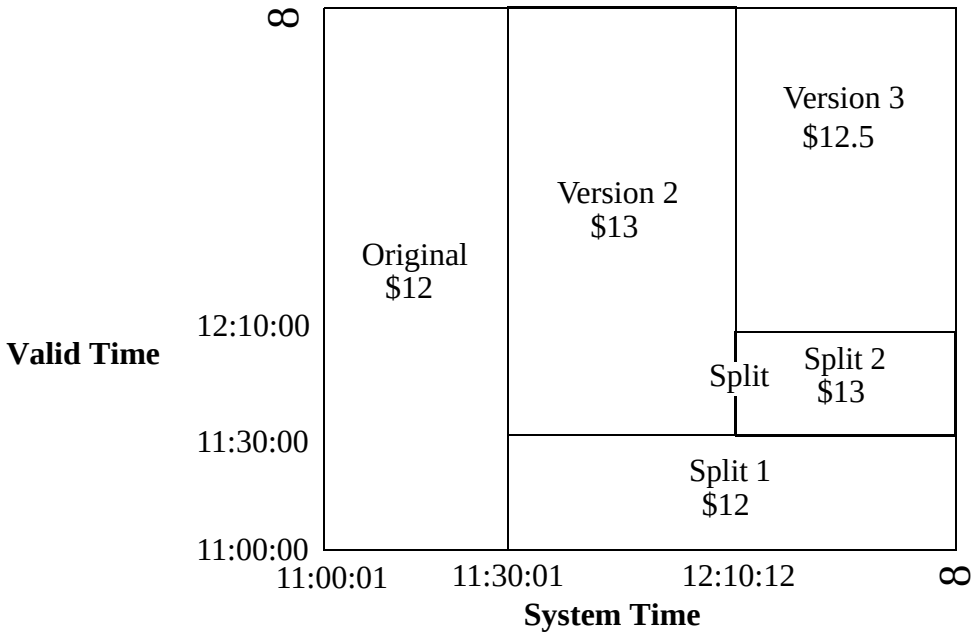
Valid Start: 2014-04-03T11:00:00
Valid End: 2014-04-03T12:10:00
```

Version 3 (new):

```
Order Price: $12.5

System Start: 2014-04-03T12:10:12
System End: 9999-12-31T11:59:59Z <-- infinity

Valid Start: 2014-04-03T12:10:00
Valid End: 9999-12-31T11:59:59Z <-- infinity
```


∞
Version 3
$12.5
Version 2
$13
Original
$12
12:10:00
Valid Time
Split
Split 2
$13
11:30:00
Split 1
$12
11:00:00
11:00:01
11:30:01
12:10:12
∞
System Time

13:00:00 -- Cancelled Order:

Update Query (JavaScript):

```
declareUpdate();
const temporal = require("/MarkLogic/temporal.xqy");
const root =
    {"tempdoc":
       {"cancel": "0"}
    };
const options =
    {metadata:
       {validStart: "2014-04-03T13:00:00",
        validEnd: "9999-12-31T11:59:59Z"}
    };
temporal.documentInsert("kool", "koolorder.json", root, options);
```

Update Query (XQuery):

```
xquery version "1.0-ml";

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

let $root :=
<tempdoc>
    <cancel>0</cancel>
</tempdoc>

let $options :=
<options xmlns="xdmp:document-insert">
    <metadata>
       <map:map xmlns:map="http://marklogic.com/xdmp/map">
         <map:entry key="validStart">
           <map:value>2014-04-03T13:00:00</map:value>
         </map:entry>
         <map:entry key="validEnd">
           <map:value>9999-12-31T11:59:59Z</map:value>
         </map:entry>
       </map:map>
    </metadata>
</options>

return
temporal:document-insert("kool", "koolorder.xml", $root, $options)
```

Results:

Original Document (no change)

Split 1 (no change)

Version 2 (no change)

Split 2 (no change)

Version 3 (updated)

```
Order Price: $12.5

System Start: 2014-04-03T12:10:12
System End: 2014-04-03T13:00:02 <-- changed

Valid Start: 2014-04-03T12:10:00
Valid End: 9999-12-31T11:59:59Z <-- infinity
```

Split 3 (new)

```
Order Price: $12.5

System Start: 2014-04-03T13:00:02
System End: 9999-12-31T11:59:59Z <-- infinity

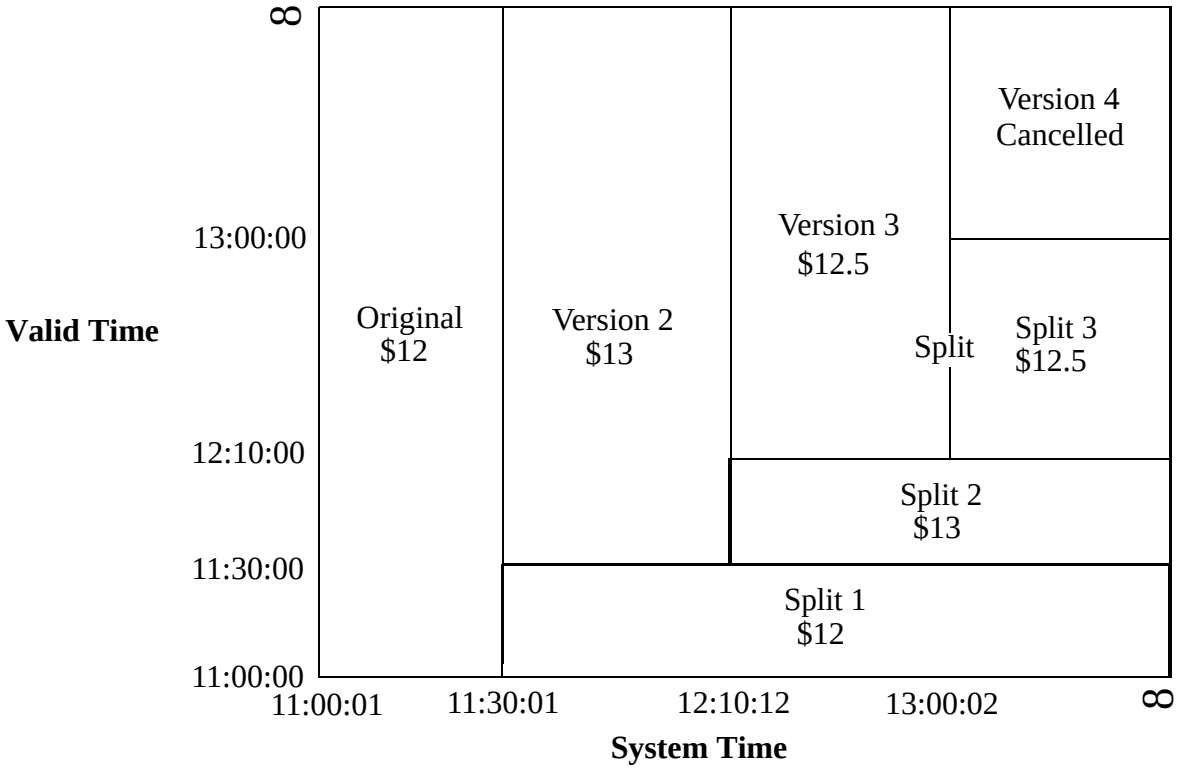
Valid Start: 2014-04-03T11:00:00
Valid End: 2014-04-03T13:00:00
```

Version 4 (new):

```
Order Price: $0

System Start: 2014-04-03T13:00:02
System End: 9999-12-31T11:59:59Z <-- infinity

Valid Start: 2014-04-03T13:00:00
Valid End: 9999-12-31T11:59:59Z <-- infinity
```


∞
13:00:00
12:10:00
11:30:00
11:00:00
Valid Time
Original
$12
Version 2
$13
Version 3
$12.5
Version 4
Cancelled
Split
Split 3
$12.5
Split 2
$13
Split 1
$12
11:00:01
11:30:01
12:10:12
13:00:02
∞
System Time

4.0 Searching Temporal Documents

This chapter describes the temporal search features, and includes the following sections:

- Temporal Search Query Constructors
- Period Comparison Operators
- Comparing Two Periods
- Example Search Queries

4.1 Temporal Search Query Constructors

The following table summarizes the available functions used to construct `cts:query` expressions for searching temporal documents. For general details on constructing `cts:query` expressions, see Composing cts:query Expressions in the *Search Developer's Guide*.

Function (XQuery and JavaScript)	Description
`cts:period-compare-query` `cts.periodCompareQuery`	Returns a `cts:query` matching documents that have relevant pair of period values.
`cts:period-compare-query-operator` `cts.periodRangeQueryOperator`	Returns the operator used to construct the specified query.
`cts:period-compare-query-options` `cts.periodCompareQueryOptions`	Returns the options for the specified query.
`cts:period-compare-query-axis-1` `cts.periodCompareQueryAxis1`	Returns the name of the first axis used to construct the specified query.
`cts:period-compare-query-axis-2` `cts.periodCompareQueryAxis2`	Returns the name of the second axis used to construct the specified query.
`cts:period-range-query` `cts.periodRangeQuery`	Returns a `cts:query` matching period by name with a period value with an operator.
`cts:period-range-query-operator` `cts.periodRangeQueryOperator`	Returns the operator used to construct the specified query.
`cts:period-range-query-options` `cts.periodRangeQueryOptions`	Returns the options for the specified query.

Function (XQuery and JavaScript)	Description
`cts:lsqt-query` `cts.lsqtQuery`	Returns only documents before LSQT or a timestamp before LSQT for stable query results.
`cts:lsqt-query-options` `cts.lsqtQueryOptions`	Returns the options for the specified query.
`cts:lsqt-query-temporal-collection` `cts.lsqtQueryTemporalCollection`	Returns the name of the temporal collection used to construct specified query.
`cts:lsqt-query-timestamp` `cts.lsqtQueryTimestamp`	Returns timestamp used to construct the specified query.
`cts:lsqt-query-weight` `cts.lsqtQueryWeight`	Returns the weight with which the specified query was constructed.

4.2 Period Comparison Operators

This section describes the Allen and ISO SQL algebra operators that can be used in search queries. Temporal queries are basically some interval operations on time period such as, period equalities, containment and overlaps. MarkLogic Server supports both Allen and SQL operators when comparing time periods. Allen's interval algebra provides the most comprehensive set of these operations. SQL 2011 also provides similar operators. However all the SQL operators can be expressed using Allen's Algebra.

4.2.1 Allen Operators

In general, Allen operators, which are identified with an ALN_ prefix, are more restrictive than ISO SQL operators, which are identified with an ISO_ prefix. The illustration below shows the relationships between the X and Y periods for each operator as used in the following period queries:

```
cts:period-range-query(X, operator, Y)
```

where X is an axis and Y is a period.

```
cts:period-compare-query(X, operator, Y)
```

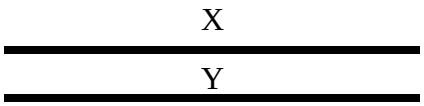
where X and Y are both axes.

```
cts:period-compare(X, operator, Y)
```

where X and Y are both periods.

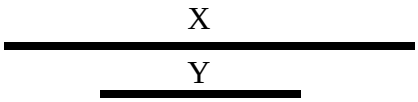
ALN_EQUALS

X-start = Y-start and X-end = Y-end



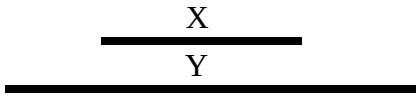
ALN_CONTAINS

X-start < Y-start and X-end > Y-end



ALN_CONTAINED_BY

X-start > Y-start and X-end < Y-end



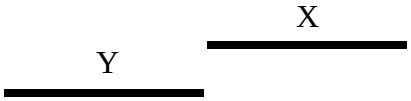
ALN_MEETS

X-end = Y-start



ALN_MET_BY

X-start = Y-end



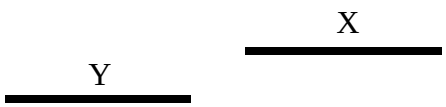
ALN_BEFORE

X-end < Y-start



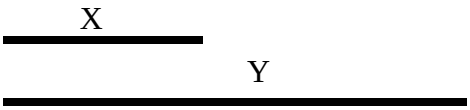
ALN_AFTER

X-start > Y-end



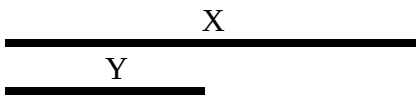
ALN_STARTS

X-start = Y-start and X-end > Y-end



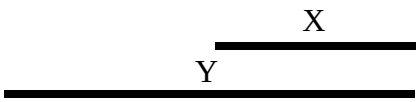
ALN_STARTED_BY

X-start = Y-start and X-end > Y-end



ALN_FINISHES

X-end = Y-end and X-start > Y-start



ALN_FINISHED_BY

X-end = Y-end and X-start < Y-start



ALN_OVERLAPS

X-start < Y-start and
X-end > Y-start and
X-end < Y-end



ALN_OVERLAPPED_BY

X-start > Y-start and
X-end > Y-end and
X-start < Y-end



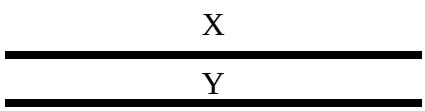
4.2.2 ISO SQL 2011 Operators

Similar to Allen operators, SQL 2011 operators can also be mapped MarkLogic range index operations. The following illustrations show the mapping from ISO SQL operators to a composition of Allen Operators.

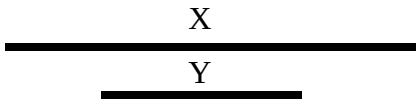
Note that the ALN operators described in “Allen Operators” on page 54 each have only one X/Y period relationship, whereas some of the less restrictive ISO SQL operators have multiple X/Y period relationships. Such ISO SQL operators have the effect of multiple Allen operators, as shown below.

ISO_OVERLAPS

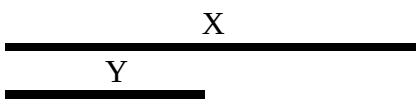
X ALN_EQUALS Y



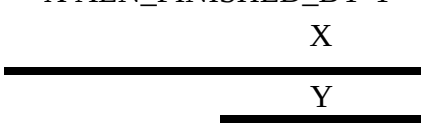
X ALN_CONTAINS Y



X ALN_STARTED_BY Y



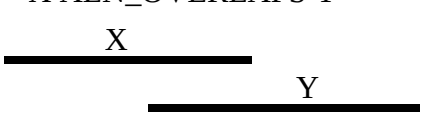
X ALN_FINISHED_BY Y



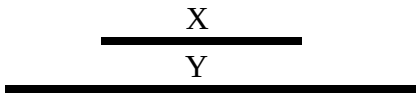
X ALN_OVERLAPPED_BY Y



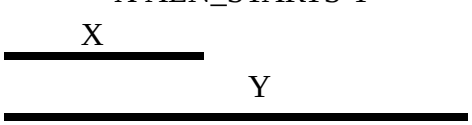
X ALN_OVERLAPS Y



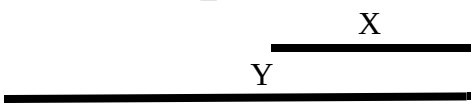
X ALN_CONTAINED_BY Y



X ALN_STARTS Y

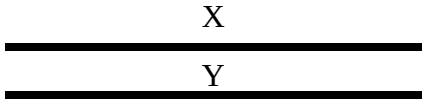


X ALN_FINISHES Y

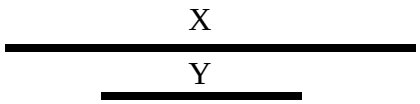


ISO_CONTAINS

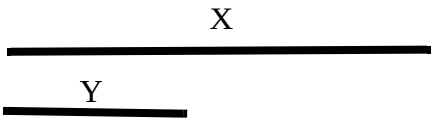
X ALN_EQUALS Y



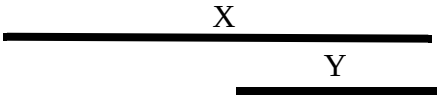
X ALN_CONTAINS Y



X ALN_STARTED_BY Y



X ALN_FINISHED_BY Y



ISO_PRECEDES

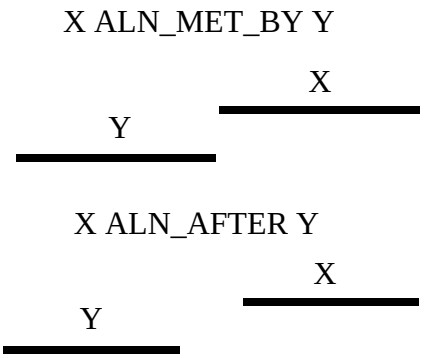
X ALN_MEETS Y



X ALN_BEFORE Y



ISO_SUCCEEDS



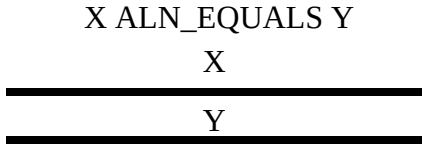
ISO_IMM_PRECEDES



ISO_IMM_SUCCEEDS



ISO_EQUALS



4.3 Comparing Two Periods

The range query on periods can be used only when there are range indexes for them on the database. However you may need to query based either on some external period values or those from a document. The following example shows how you can use the `cts:period-compare` function to determine whether two period values satisfy the conditions imposed by the comparison operator, `ALN_MEETS`. If the `ALN_MEETS` conditions are satisfied, then `true` is returned; otherwise `false` is returned.

JavaScript Example:

```
var period1 = cts.period(xs.dateTime("2000-05-31T09:30:10"),
                         xs.dateTime("2003-05-31T12:30:00"));

var period2 = cts.period(xs.dateTime("2003-05-31T12:30:00"),
                         xs.dateTime("2004-05-31T14:30:00"));

cts.periodCompare(period1, "ALN_MEETS", period2);
```

XQuery Example:

```
xquery version "1.0-ml";

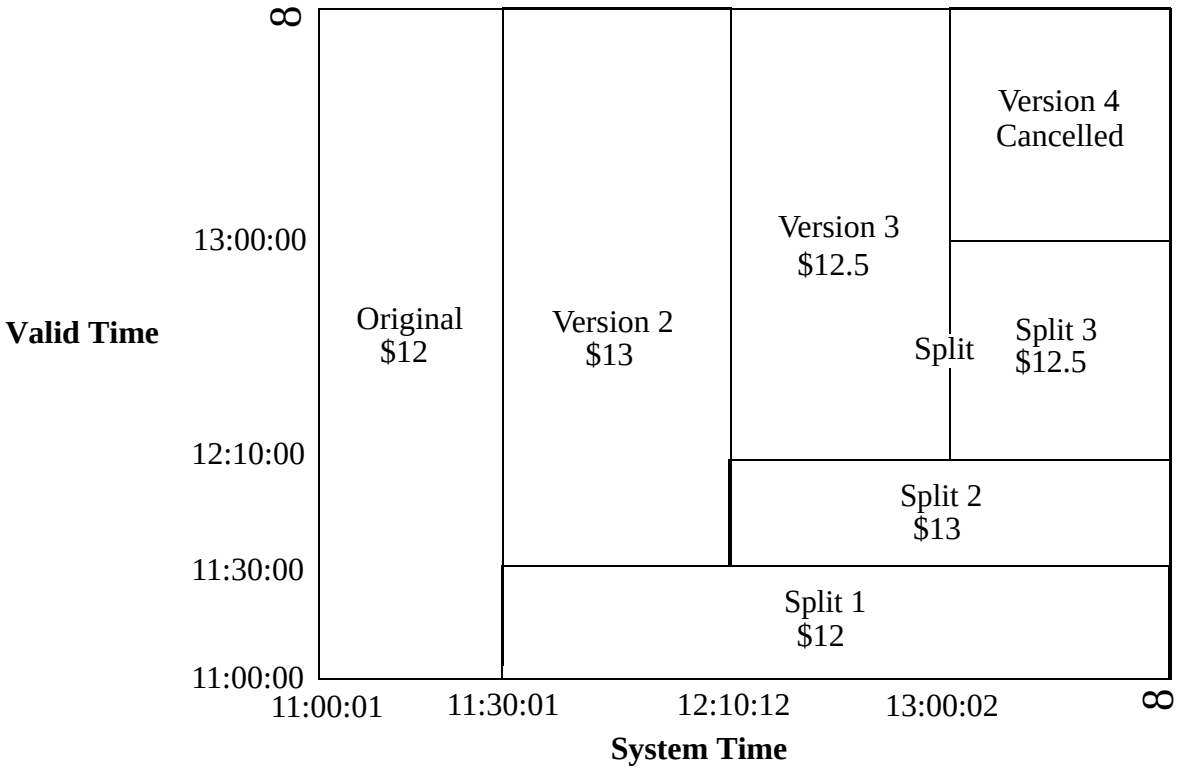
let $period1 := cts:period(xs:dateTime("2000-05-31T09:30:10"),
                           xs:dateTime("2003-05-31T12:30:00"))

let $period2 := cts:period(xs:dateTime("2003-05-31T12:30:00"),
                           xs:dateTime("2004-05-31T14:30:00"))

return cts:period-compare($period1,"ALN_MEETS",$period2)
```

4.4 Example Search Queries

This section describes some sample search queries. The searches described in this section are done on the documents described in “Example: The Lifecycle of a Temporal Document” on page 41.



The following query searches for the temporal documents that have a valid end time before `14:00`:

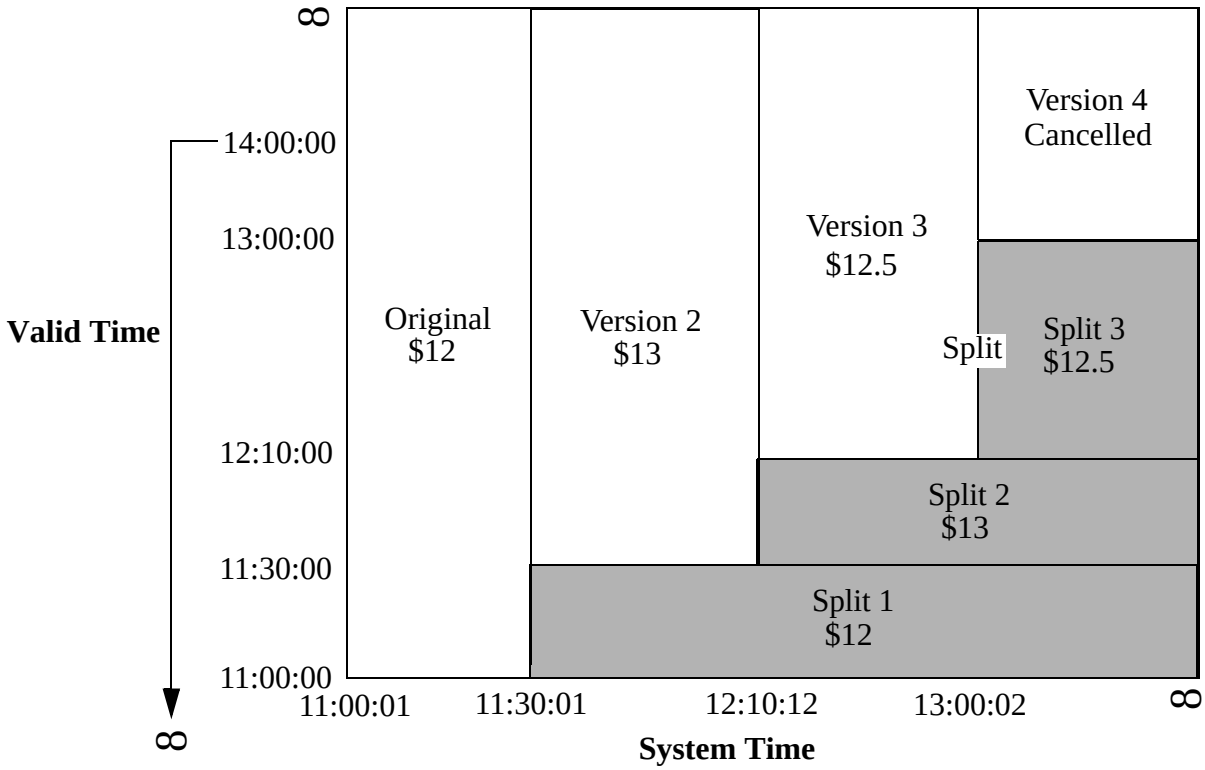
JavaScript Example:

```
cts.search(cts.periodRangeQuery(
    "valid",
    "ALN_BEFORE",
    cts.period(xs.dateTime("2014-04-03T14:00:00"),
               xs.dateTime("9999-12-31T11:59:59Z")) ))
```

XQuery Example:

```
cts:search(fn:doc(), cts:period-range-query(
   "valid",
   "ALN_BEFORE",
   cts:period(xs:dateTime("2014-04-03T14:00:00"),
              xs:dateTime("9999-12-31T11:59:59Z")) ))
```

This query returns Splits 1, 2 and 3 of the document.



The following query searches the temporal documents, using the `cts:and-query` to AND two `cts:period-range-query` functions, to locate the documents that represented the order at 11:30 when queried at 11:51.

JavaScript Example:

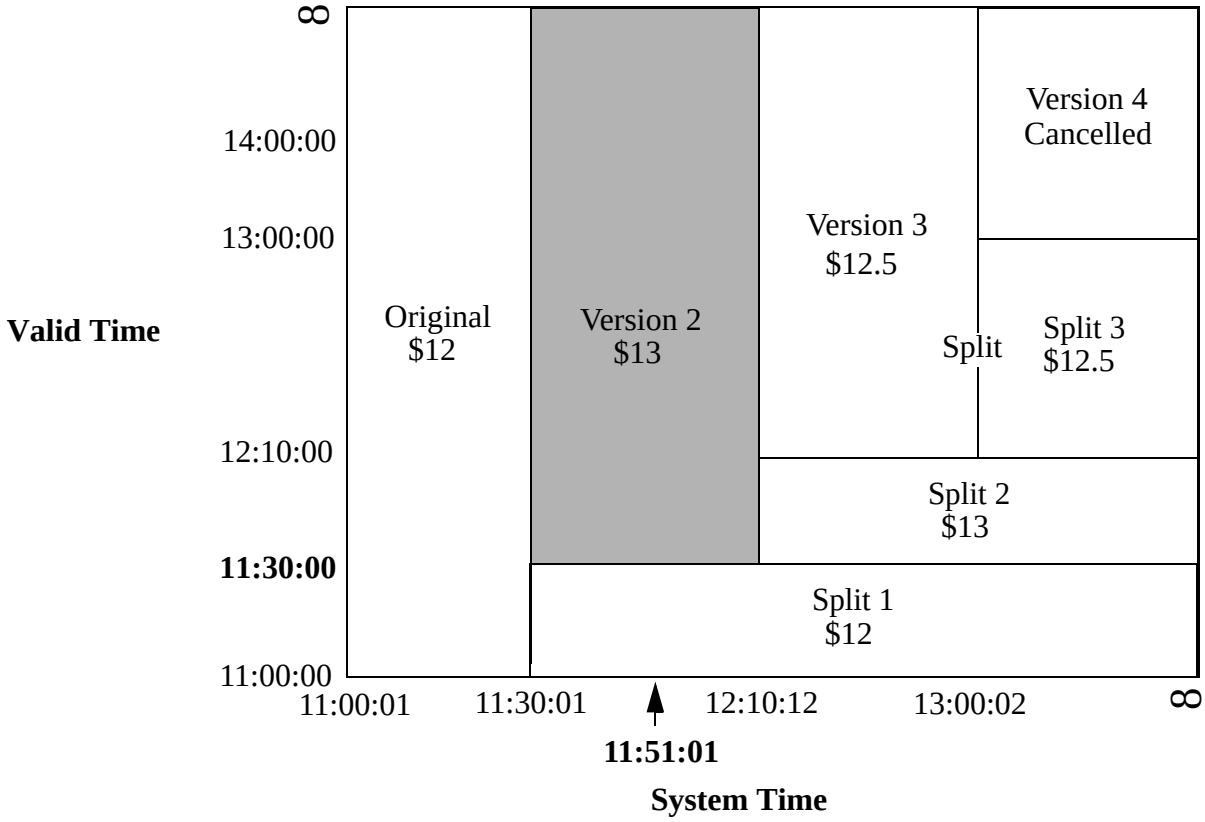
```
cts.search(cts.andQuery(
      [cts.periodRangeQuery(
          "system",
          "ISO_CONTAINS",
          cts.period(xs.dateTime("2014-04-03T11:51:00"),
                     xs.dateTime("2014-04-03T11:51:01"))),
      cts.periodRangeQuery(
          "valid",
          "ISO_CONTAINS",
          cts.period(xs.dateTime("2014-04-03T11:30:00"),
                     xs.dateTime("2014-04-03T11:30:01")))]))
```

XQuery Example:

```
xquery version "1.0-ml";

cts:search(fn:doc(), cts:and-query((
   cts:period-range-query(
      "system",
      "ISO_CONTAINS",
      cts:period(xs:dateTime("2014-04-03T11:51:00"),
                 xs:dateTime("2014-04-03T11:51:01"))),
   cts:period-range-query(
      "valid",
      "ISO_CONTAINS",
      cts:period(xs:dateTime("2014-04-03T11:30:00"),
                 xs:dateTime("2014-04-03T11:30:01"))))))
```

This query returns Version 2 of the document.



The following query searches for the temporal documents that have a valid end time of `12:10`:

JavaScript Example:

```
var period = cts.period(xs.dateTime("2014-04-03T12:10:00"),
                        xs.dateTime("2014-04-03T13:10:00"));

cts.search(cts.periodRangeQuery("valid", "ALN_MEETS", period))
```

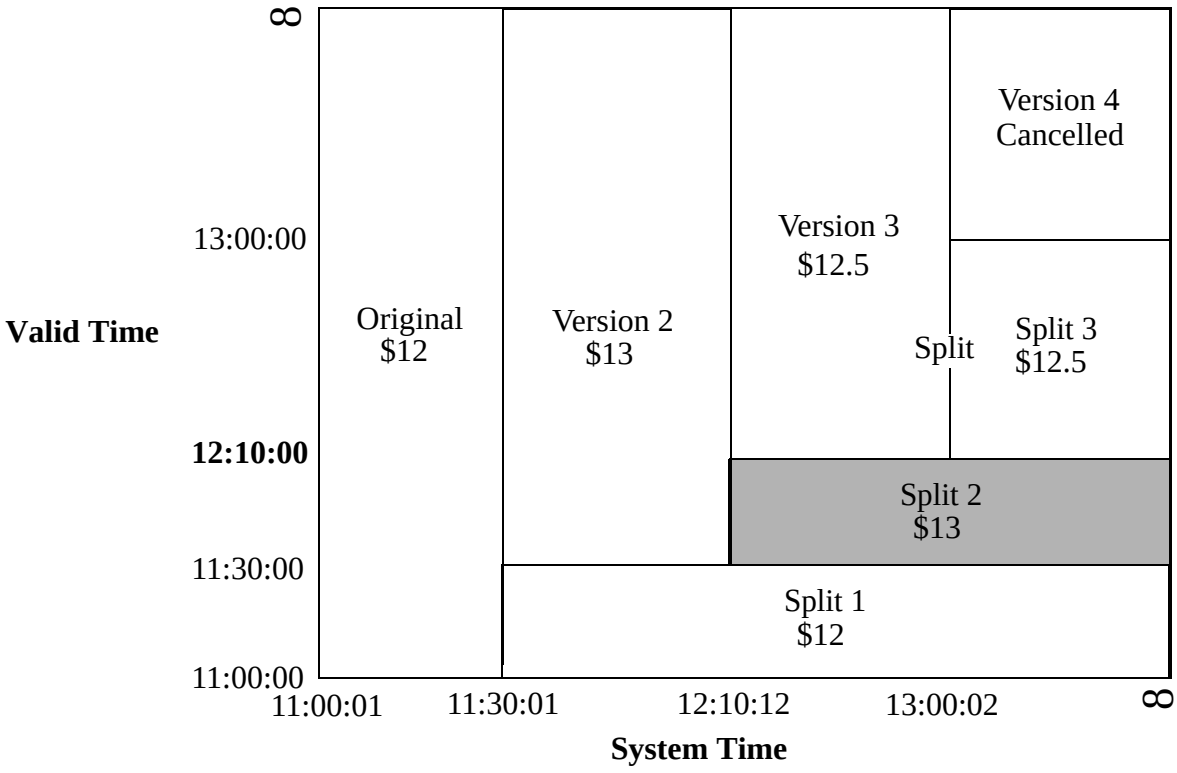
XQuery Example:

```
xquery version "1.0-ml";

let $period := cts:period(xs:dateTime("2014-04-03T12:10:00"),
                          xs:dateTime("2014-04-03T13:10:00"))

return cts:search(fn:doc(), cts:period-range-query(
     "valid", "ALN_MEETS", $period))
```

This query returns Split 2 of the document.



5.0 Copyright

MarkLogic Server 9.0 and supporting products.
Last updated: November 18, 2019

COPYRIGHT

© 2019 MarkLogic Corporation. All rights reserved.
This technology is protected by U.S. Patent No. 7,127,469B2, U.S. Patent No. 7,171,404B2, U.S. Patent No. 7,756,858 B2, and U.S. Patent No 7,962,474 B2, US 8,892,599, and US 8,935,267.

The MarkLogic software is protected by United States and international copyright laws, and incorporates certain third party libraries and components which are subject to the attributions, terms, conditions and disclaimers set forth below.

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

6.0 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 technical questions, we encourage you to ask your question on Stack Overflow.