MarkLogic Server

Query Performance and Tuning Guide

MarkLogic 9
May, 2017

Last Revised: 9.0-11, November, 2019

Copyright © 2019 MarkLogic Corporation. All rights reserved.

Table of Contents

Query Performance and Tuning Guide

1.0 Tuning Query Performance in MarkLogic Server 7
- 1.1 Overview of Query Performance 7
- 1.2 General Techniques to Tune Performance 8
  - 1.2.1 Search Built-In APIs 8
  - 1.2.2 Lexicons For Unique Word or Value Lookups 9
  - 1.2.3 Range Queries for Constraining Searches to a Range of Values 9
  - 1.2.4 Positions Indexes Can Help Speed Phrase Searches 9
  - 1.2.5 Use Query Meters and Query Trace to Characterize Performance 9
  - 1.2.6 Profiler API 9
  - 1.2.7 Monitoring API and Status Screens 9
  - 1.2.8 Index Options, Range Indexes, Fields 10
- 1.3 Understanding MarkLogic Server Caches 10
- 1.4 Rules of Thumb for Sizing 11

2.0 Fast Pagination and Unfiltered Searches 13
- 2.1 Understanding the Search Process 13
- 2.2 Understanding Unfiltered Searches 14
- 2.3 Using Unfiltered Searches for Fast Pagination 16
- 2.4 Example: Determining the Number of False-Positive Matches 17

3.0 Tuning Queries with query-meters and query-trace 19
- 3.1 Indexes, XPath Expressions, and Query Performance 19
- 3.2 Understanding query-meters Output 20
  - 3.2.1 Output From xdmp:query-meters 20
  - 3.2.2 Understanding the Cache Statistics 20
- 3.3 Understanding query-trace Output 22
  - 3.3.1 What query-trace Logs 22
    - 3.3.1.1 XPath Expression Analysis Messages 23
    - 3.3.1.2 Constraint Analysis Messages 23
    - 3.3.1.3 Search Execution Messages 24
  - 3.3.2 Interpreting the Log Messages 24
  - 3.3.3 Fully Searchable Paths and cts:search Operations 25
- 3.4 Using xdmp:plan to View the Evaluation Plan 27
- 3.5 Examples 27
  - 3.5.1 Sample xdmp:query-meters Output 28
  - 3.5.2 Sample xdmp:query-trace Output 29
  - 3.5.3 Logging Both query-meters and query-trace Output 30

4.0 Sorting Searches Using Range Indexes 35
4.1 Using a cts:order Specification in a cts:search 35
4.1.1 Creating a cts:order Specification 35
4.1.2 Using the cts:order Specification in a Search 36
4.2 Optimizing Order By Expressions With Range Indexes 36
4.2.1 Speed Up Order By Performance 36
4.2.2 Rules for Order By Optimization 36
4.2.3 Creating Range Indexes 38
4.2.4 Example Order By Queries 39
4.2.4.1 Order by a Single Element 39
4.2.4.2 Order by Multiple Elements 40

5.0 Profiling Requests to Evaluate Performance 41
5.1 Enabling Profiling on an App Server 41
5.2 Understanding XQuery Profiling 41
5.2.1 Definitions and Terminology for the XQuery Profiling 42
5.2.2 XQuery Profiling Overview 42
5.2.3 XQuery Profiling API 43
5.3 Understanding Server-Side JavaScript Profiling 44
5.4 Profiling Examples 45
5.4.1 Simple Enable and Disable XQuery Example 45
5.4.2 Returning a Part of the XQuery Profile Report 47
5.4.3 JavaScript Profile Example 47

6.0 Disk Storage Considerations 51
6.1 Disk Storage and MarkLogic Server 51
6.2 Fast Data Directory on Forests 51
6.3 Large Data Directory on Forests 52
6.4 HDFS, MapR-FS, and S3 Storage on Forests 52
6.4.1 HDFS Storage 52
6.4.2 MapR-FS Storage 54
6.4.3 S3 Storage 54
6.4.3.1 S3 and MarkLogic 54
6.4.3.2 Entering Your S3 Credentials for a MarkLogic Cluster 56
6.5 Windows Shared Disk Registry Settings and Permissions 56

7.0 Monitoring MarkLogic Server Performance 57
7.1 Ways to Monitor Performance and Activity 57
7.1.1 Monitoring History Dashboard 57
7.1.2 Server Logs 58
7.1.3 Status Screens in the Admin Interface 59
7.1.4 Create Your Own Server Reports 61
7.2 Server Monitoring APIs 61

8.0 Endpoints and Request Monitoring 63
8.1 Monitoring Requests 63
8.2 App Server Request Monitoring 63
8.3 XDBC Server Request Monitoring 64
8.3.1 XDBC Invoke Requests 64
8.3.2 XDBC Eval Requests 64
8.4 Task Server Monitoring 64
8.5 Creating Endpoint Declarations 64
8.5.1 The Endpoint Declaration File 65
8.5.2 Constraints on Meters 70
8.5.3 Controlling Request Logging With Thresholds 71
8.5.4 Enabling Request Monitoring 72
8.5.5 The Default Declaration File 74
8.5.6 Request Logs 75
8.6 Request Cancelling 75
8.7 Request Monitoring APIs 76

9.0 Technical Support 77

10.0 Copyright 79
10.0 COPYRIGHT 79

1.0 Tuning Query Performance in MarkLogic Server

This chapter describes some general issues involving query performance in MarkLogic Server, and includes the following sections:

- Overview of Query Performance
- General Techniques to Tune Performance
- Understanding MarkLogic Server Caches
- Rules of Thumb for Sizing

1.1 Overview of Query Performance

MarkLogic Server is designed to search extremely large content sets, while providing fine-grained control over the search and access of the content. Performance is always an important component in a search application. In many cases, applications will be extremely fast with no tuning whatsoever. There are, however, many tools and techniques to help make queries faster.

There are several things to consider when looking at query performance:

- Application requirements: how fast does performance need to be for your application? It is often useful to quantify this at application design time. Factors such as who will be using the application, what any user expectations for performance are, and whether the application will be publicly available are important considerations in defining performance requirements.
- Indexing options: what indexes are defined for the database? Indexing options play an important role in how well queries can be resolved from the indexes. The fastest way to resolve a query is directly from the indexes. For details on database options, see the chapters Databases and Text Indexing in the *Administrator's Guide*.
- XQuery code: is your code written in the most efficient way possible? Sometimes, code runs more slowly than necessary because there are redundant or unneeded function calls. Or there may be a MarkLogic XQuery built-in function that performs an equivalent task more efficiently. Functions such as `xdmp:estimate`, `cts:search`, lexicon functions, and so on are all designed for fast performance.
- More indexes and lexicons: can range indexes and lexicons speed up your queries? For queries that access values and/or do comparisons on those values, range indexes can greatly speed performance. Range indexes are memory mapped structures, so they can retrieve the values without ever needing to access the documents. Lexicons are lists of words or values, and they too can greatly speed up certain types of queries.
- Server tuning: are your server parameters set appropriately for your system? In most cases, the parameters set during installation work well for the system in which MarkLogic Server is installed. Nevertheless, there are cases where you might need to change some parameters, either for a short-term need or for ongoing needs.

- Scalability: is your system sufficiently large for your needs? Memory, disk space and quality, swap space, number of processors, and number of servers all contribute to the overall scalability of a MarkLogic Server system. MarkLogic Server is designed to scale to very large clusters with extremely large amounts of content.
- Access patterns and resource requirements differ for analytic workloads. In general, analytic workloads access and aggregate more data per transaction, increasing the baseline memory requirements. Although there are stated minimum memory requirements for MarkLogic Server, the memory requirements for analytics should be higher than those stated.

This chapter and this book, as well as the *Application Developer's Guide*, provide information and techniques on tuning a system for optimal performance. The nature of tuning exercises is that they tend to be content-specific, so you cannot always pinpoint a particular recipe that will work for every situation. Getting to know the tools available, the XQuery APIs, and how MarkLogic Server works is the best way to make your applications run extremely fast.

1.2 General Techniques to Tune Performance

This section lists some general techniques useful in tuning performance, and provides links to places in the documentation where there is more information on a subject. It contains the following parts:

- Search Built-In APIs
- Lexicons For Unique Word or Value Lookups
- Range Queries for Constraining Searches to a Range of Values
- Positions Indexes Can Help Speed Phrase Searches
- Use Query Meters and Query Trace to Characterize Performance
- Profiler API
- Monitoring API and Status Screens
- Index Options, Range Indexes, Fields

1.2.1 Search Built-In APIs

The search built-in XQuery APIs are designed to provide very fast searches. The APIs (`cts:search`, `xdmp:estimate`, `cts:element-values`, and so on) use the indexes for fast search performance. The composable `cts:query` constructors make it easy to compose complex search queries with fast performance. For details on the search built-in XQuery APIs, see *MarkLogic XQuery and XSLT Function Reference*. For details on the constructors, see Composing cts:query Expressions in the *Search Developer's Guide*.

1.2.2 Lexicons For Unique Word or Value Lookups

MarkLogic Server allows you to create lexicons, which are lists of unique words or values in a database. Lexicons allow for very fast lookups, and in the case of values, also provide very fast counts. For details on lexicons, see the chapter Browsing With Lexicons in the *Search Developer's Guide*.

1.2.3 Range Queries for Constraining Searches to a Range of Values

Range queries allow you to specify queries that use range indexes in a `cts:query` expression. Range queries can both improve performance and make it easier to build applications that constrain on values. For details on range queries, see Using Range Queries in cts:query Expressions in the *Search Developer's Guide*.

1.2.4 Positions Indexes Can Help Speed Phrase Searches

If you specify `word positions` in the database configuration, it can speed phrase searches. During the index resolution phase of query processing, MarkLogic Server determines if words are next to each other based on their positions. For example, if you search for the phrase `"to be or not to be"`, MarkLogic Server can eliminate as possible matches, based on positions, most occurrences of these common words because they do not have the proper word next to it. This speeds performance in two ways: it lowers the number of I/Os needed to retrieve candidate fragments, and it makes the filtering phase faster because there are less candidate fragments to filter. For details about how search processing works, see "Understanding the Search Process" on page 13.

1.2.5 Use Query Meters and Query Trace to Characterize Performance

There are two XQuery functions to help you characterize the performance of queries: `xdmp:query-meters` and `xdmp:query-trace`. The former provides timing of a query and the latter logs details of the query evaluation to the `ErrorLog.txt` file. For details on these APIs, see "Tuning Queries with query-meters and query-trace" on page 19 and the *MarkLogic XQuery and XSLT Function Reference*.

1.2.6 Profiler API

MarkLogic Server has a profiler to help determine where a query is spending time processing. For details on the profiler, see "Profiling Requests to Evaluate Performance" on page 41 and the *MarkLogic XQuery and XSLT Function Reference*.

1.2.7 Monitoring API and Status Screens

There are APIs and status screens in the Admin Interface to monitor activities on your system. These can be useful in identifying bottlenecks on your system. For details, see "Monitoring MarkLogic Server Performance" on page 57.

1.2.8 Index Options, Range Indexes, Fields

There are many types of index options, including several types of wildcard indexes, element indexes, stemmed indexes, element and attribute range indexes, and so on. Depending on your needs, these indexes can help speed performance. Indexes tend to take more disk space and increase loading times, but can greatly improve performance.

Fields are another way of improving performance, especially if you are only interested in searching through certain included elements, or you want your searches to exclude particular elements. For details on fields, see Fields Database Settings in the *Administrator's Guide*.

1.3 Understanding MarkLogic Server Caches

MarkLogic Server has several caches used in query processing, defined on the group configuration page. The *list cache* stores termlists in memory, the *compressed tree cache* stores compressed fragment data in memory, and the *expanded tree cache* stores uncompressed fragment data in memory. Additionally, there are several other caches used for security objects, modules, schemas, and so on; these other caches cannot be configured. In most cases, if the caches fill up, they will move older data out to make room for newer content.

In some cases, however, it is possible to run a query that will fail because a cache was full. Particularly, when the expanded tree cache gets full, a query can fail with an `XDMP-TREECACHEFULL` exception. The following are some guidelines to avoid `XDMP-TREECACHEFULL` errors:

- Avoid queries that return the entire database. Instead, return the results in batches (a page at a time, like a classic search page, for example).
- Try to rewrite the query in a more efficient way.
- Make sure swap space is configured properly on your server.
- If you do not have sufficient memory on your server, consider adding more memory to the system.
- You can raise the sizes of the caches, but that might be a temporary fix.
- 64-bit systems are recommended. 64-bit systems can hold a lot more memory, and more memory means larger caches.

1.4 Rules of Thumb for Sizing

The following are some rule-of-thumb sizing recommendations. These recommendations are best practices based on experience with MarkLogic Server implementations. Also, some of these recommendations are content specific. Performing experiments on your own content is a good way to validate any expansions of these rules of thumb, but these provide a good starting point.

- For each node, provide memory based either on the formula: (threads x cores x 4GB) or 64GB. Whichever is higher.
- Allot approximately 1 GB of RAM for each 10-20 GB of forest data. More memory will help, too, especially if you have a lot of range indexes and/or lexicons.
- 64-bit systems greatly increase the address space so you can address more than 4GB of RAM.
- Plan to configure no more than 512 primary forests per database. There is a hard limit of 1024 primary forests in a single database.
- The swap space should be at least 2x memory (or the recommended amount for your platform, as described in Memory, Disk Space, and Swap Space Requirements in the *Installation Guide*). This is important to make sure MarkLogic Server does not run out of memory. At query time, MarkLogic Server asks the operating system to reserve both memory and swap space. If there is not enough of either, the query can fail with `SVC-MEMALLOC` messages. These messages can happen if you do not have the recommended amount of swap space. If you do have enough swap space and still get these errors, it can indicate that you either need to increase the amount of memory in the system or lower the amount of memory being used, either by modifying your queries or lowering some of the sizes of server caches, lowering the number of threads the server can service, and so on.
- For updates, make the journal size larger if you have a lot of range index data. A symptom of this as a problem is journal-full errors.
- For updates, make the journal size larger if your transactions span multiple forests. The journals must keep the lock information for all documents involved in the transaction, not just for the documents in the journal for the forest in which the document exists. A symptom of this as a problem is journal-full errors.
- There is a limit of 65k for the size of a string literal or a token in an XQuery program. If you need to input a string longer than 65k, use an external variable with the `xdmp:invoke` API. External variables are limited to a single node or a string, and in XCC are limited to string only. In XCC, if you need to input a node as an external variable, you must quote it as a string on input and then unquote it (`xdmp:unquote`) into a node in your XQuery function. Note that this limit is only for the size of a string literal or a token; XQuery program sizes are limited only by the cache size.

2.0 Fast Pagination and Unfiltered Searches

MarkLogic Server provides a powerful XML-enabled search capability through the `cts:search` XQuery built-in function. As part of the search capability, MarkLogic Server allows you to issue `cts:search` expressions that return results directly from the indexes, without performing the filtering necessary to ensure there are no false-positive results. This chapter describes the search process, including unfiltered searches, and includes the following sections:

- Understanding the Search Process
- Understanding Unfiltered Searches
- Using Unfiltered Searches for Fast Pagination
- Example: Determining the Number of False-Positive Matches

2.1 Understanding the Search Process

When evaluating `cts:search` expressions (and also when resolving XPath expressions within XQuery code), MarkLogic Server performs a two-step process.

1. A list of candidate fragment IDs is generated directly from the indexes, based on the index-resolvable criteria incorporated in the various parameters passed to `cts:search`. Fragment IDs are ordered according to relevance criteria. This step is called *index resolution*.

2. The candidate fragment IDs are used to load fragments from disk. Each fragment is then examined in order, using the complete criteria incorporated in the various parameters passed to `cts:search`, to determine if the fragment contains zero, one, or more than one result that matches the given `cts:search` expression. This step is called *filtering*.

The purpose of index resolution is to narrow the set of candidate fragments to as small a set as possible, without missing any. In some circumstances, the index resolution step can yield a precisely correct set of candidate fragments, rendering the filtering step redundant. In other circumstances, index resolution can reduce the set of candidate fragments somewhat, but in the candidate fragment list there are still false-positive results (that is, candidate fragments that in fact contain no matching results). In still other circumstances, the candidate fragment list can contain fragments that contain more than one matching result.

To better understand false-positive results, imagine a database configuration which has not enabled fast case-sensitive indexes (`fast case sensitive searches` on the database configuration page). This means that the full-text indexes only maintain direct lookups for words independent of their case. In this scenario, if you are searching for `"Dog"`, the indexes can only tell you what fragments contain the word `"dog"`, in any case-combination of text (for example, `"dog"`, `"DOG"`, `"Dog"`, `"doG"`, and so on). So when index resolution generates a candidate fragment list for `"Dog"`,

that list could include a fragment that has the word `"dog"` but not the word `"Dog"`. This is where filtering comes in—by loading that fragment and examining it prior to returning it as a match, MarkLogic Server is able to determine that it is not in fact a match for the specified query, and rule it out. The fragment is a false-positive result, and should not be returned to the query.

To understand how a candidate fragment can contain more than one match, consider a single-fragment document that contains multiple `<author>` elements as follows:

```
<author>Bruce Smith</author>
<author>Betty Smith</author>
<author>Gordon Blair</author>
```

Now consider the following query:

```
cts:search(//author, "Smith")
```

During index resolution, this query generates the fragment for that document as a single candidate fragment. In fact, that single document should generate two results—one for each of Bruce and Betty Smith. During the filtering step, MarkLogic Server identifies that there is more than one element in this document that matches the search requirements, and returns both of the first two `<author>` elements as results.

As you can see from these examples, the combination of index resolution and filtering combine to provide both performance and accuracy. The algorithm is designed to allow you to write complex queries, and have MarkLogic Server determine the most efficient path providing accurate results.

There are disadvantages to the algorithm, however. Sometimes, you might know better than the search engine, and through careful design of your XML and your fragmentation, you might know that filtering is simply unnecessary. In this case, filtering takes unneeded cycles. In another situation, if you want to jump deep into a result set (for example, retrieving the 1,000,000th result from a really large result set), the emphasis MarkLogic Server has on accuracy through filtering might provide an impractical constraint for your application, because filtering the first 999,999 results will take far too long. Furthermore, it might not matter to your application if, when you jump to the 1,000,000th result, you actually end up at *approximately* that result (even if in reality it is the 949,237th result).

Consequently, MarkLogic Server provides you with tools to influence the evaluation of `cts:search` expressions, indicating whether or not filtering is required.

2.2 Understanding Unfiltered Searches

An *unfiltered* search omits the filtering step, which validates whether each candidate fragment result actually meets the search criteria. Unfiltered searches, therefore, are guaranteed to be fast, while filtered searches are guaranteed to be accurate. By default, searches are filtered; you must specify the `"unfiltered"` option to `cts:search` to return an unfiltered search. Unfiltered searches have the following characteristics:

- They determine the results directly from the indexes, without filtering for validation. This makes unfiltered results most comparable to traditional search-engine style results.
- They include false-positive results. False-positive results can originate from a number of situations, including phrase searches containing 3 or more words, certain wildcard searches, punctuation-sensitive, diacritic-sensitive, and/or case-sensitive searches.
- They will be significantly affected by fragmentation policy.

The following are some useful guidelines for when to use unfiltered searches:

- You should only perform unfiltered searches on top-level nodes or on fragment roots, otherwise you might get unexpected answers. This is because, for queries below the fragment level, there is no guarantee that a particular unfiltered search even matches the query (that is, there is a match somewhere in the fragment, but not necessarily a match in the node you are searching).
- If you choose to specify an XPath other than a top-level node or a fragment root, your XPath expression will give correct results if there is only one possible node to match in each fragment (or if the only possible match is in the first node specified). This is because unfiltered searches stop after encountering the first node per fragment that matches the specified XPath expression. If you are sure that the node you specify only has one instance per fragment, then it will not miss any results (although it might get false-positive results). An example of this is `ABSTRACT` in MEDLINE citatation, where `ABSTRACT` is below the fragment root, but there is only one `ABSTRACT` node per fragment. If you specify below a fragment root and there are multiple nodes in the fragment, the search may miss results (it will only find results if they are in the first fragment).

Finally, it is useful to understand that `cts:contains` implements the filtering step of the two-step search process:

```
unfiltered cts:search + cts:contains = normal (filtered) cts:search
```

Breaking a `cts:search` operation into an unfiltered search and a `cts:contains` allows you to do the search so it is always fast, but then only do the false-positive result removal if you want or need to. This is true as long as the first parameter to `cts:search` is at a fragment root node. If it is below a root node, it is only true if you know that the first node is the only possible hit for the search (for example, if there is only one node, as in the `ABSTRACT` example above).

2.3 Using Unfiltered Searches for Fast Pagination

There are many useful applications for unfiltered searches. Applications of unfiltered searches tend to have one or more of the following characteristics:

- Your content and search terms are such that you know the unfiltered searches are also accurate (for example, the searches are all performed at document or fragment roots, they are single-term queries, and are not wildcard, punctuation-sensitive, diacritic-sensitive, and/or capitalization-sensitive searches).
- You do not mind if there are some false-positive results because the results are an estimate (that is, they need to be fast, but are not required to be exact).
- Your searches return a large number of results and you want efficient ways to jump to a particular portion of those results.

The last point describes the situation for fast pagination. *Fast pagination* is a way to get an approximate count of the total number of result hits and then provide efficient ways to jump deep to an arbitrary point in the result sequence. Such pagination is common in search engine-style results, where a particular result might return 1 million hits and the search interface returns them 10 at a time. There is usually some sort of counter that says something like “displaying 1-10 of 1,000,000 results,” and then there are links to go to the next page of results or to go to the tenth, twentieth, or any page of the results. Often it is not critical that going to the twentieth page actually gets you to the 200th hit; it is OK if there were some false-positive results, and when you click that link you actually get to the 176th result.

When you implement a fast pagination application, you will need to jump into a position in an unfiltered search. To maximize the efficiency of this search, you must immediately follow the unfiltered `cts:search` expression with the position predicate, with no XPath steps in between. For example, to jump into the 1,000,001st hit of an unfiltered search for the phrase “one two three”, the search might look like the following:

```
cts:search(fn:doc(), "one two three", "unfiltered")[1000001 to 1000010]
```

This search will skip directly to the 1,000,001st unfiltered hit and return the 10 fragments specified in the position predicate; it will not need to fetch any other fragments.

2.4 Example: Determining the Number of False-Positive Matches

The following code sample prints out the number of false-positive matches from a search.

```
xquery version "1.0-ml";
declare boundary-space preserve;
declare namespace qm="http://marklogic.com/xdmp/query-meters";

let $trueCounter := 0
let $falseCounter := 0
let $searchTerms := "one! two three"
let $x :=
  for $result in cts:search(fn:doc(), $searchTerms, "unfiltered")
  return
  (
  if ( cts:contains($result, $searchTerms) )
  then ( xdmp:set($trueCounter, $trueCounter + 1) )
  else ( xdmp:set($falseCounter, $falseCounter + 1) )
  )
return
<results>
  <resultTotal>{$trueCounter}</resultTotal>
  <false-positiveTotal>{$falseCounter}</false-positiveTotal>
  <elapsed-time>{xdmp:query-meters()/qm:elapsed-time/text()}
  </elapsed-time>
</results>
```

Because the specified `$searchTerms` contains punctuation in the middle of the phrase, any document that has the phrase “one two three” will prove to be a false-positive result. If you substitute in your query terms for the `$searchTerms` variable, you can see if your own unfiltered search yields false-positive results.

The above code uses the `xdmp:set` function to keep track of the number of matches and the number of false-positive results. It runs the unfiltered search and then uses `cts:contains` to check if each result is actually a match. If it is a match, then increment the `$trueCounter` variable, otherwise increment the `$falseCounter` variable.

3.0 Tuning Queries with query-meters and query-trace

MarkLogic Server is designed for very fast query performance over large amounts of data. While query performance is usually very fast, sometimes you will issue queries that do not perform as well as you would like. MarkLogic Server includes functions to help you optimize the performance of queries.

This chapter describes how to use the `xdmp:query-meters` and `xdmp:query-trace` functions to understand and tune the performance of queries. It includes the following sections:

- Indexes, XPath Expressions, and Query Performance
- Understanding query-meters Output
- Understanding query-trace Output
- Using xdmp:plan to View the Evaluation Plan
- Examples

3.1 Indexes, XPath Expressions, and Query Performance

When you load data into a MarkLogic Server database, indexes are created based on the index configuration for that database. The indexes help to optimize searches, XPath expressions, and other query patterns.

Sometimes, however, a query cannot use the indexes, and that leads to slower performance. In these cases, there are two main types of things you can do to speed up the query performance:

- Rewrite the query so it makes better use of the indexes.
- Add more indexes.

The `xdmp:query-meters` and `xdmp:query-trace` functions provide information to help you determine where the problem areas in the query are, and can help you determine ways to easily and, in many cases, dramatically improve query performance. Understanding the output of these functions is the key to analyzing a query and tuning it for maximum performance.

To use these functions in a query:

- Add `xdmp:query-meters()` to the end of a query, with the concatenate operator (`,`) before the function.
- Add `xdmp:query-trace(true())` to the beginning of the portion of the query you want to analyze, with the concatenate operator (`,`) after the function. Then add `xdmp:query-trace(false())` at the end of the portion of the query you want to analyze, with the concatenate operator (`,`) before the function.

3.2 Understanding query-meters Output

The `xdmp:query-meters` function provides statistics about query execution. To use `xdmp:query-meters`, concatenate the `xdmp:query-meters()` function to the end of your query. For example, the following query produces both the initial query results and the `query-meters` output:

```
doc("/myDocuments/hello.xml")//a/b/c
, xdmp:query-meters()
```

The result is a sequence of `c` nodes from the `/myDocuments/hello.xml` document followed by a `qm:query-meters` node containing the `query-meters` output.

For its function signature, see the `xdmp:query-meters` function in *MarkLogic XQuery and XSLT Function Reference*.

The following subsections describe the output of the `xdmp:query-meters` function:

- Output From xdmp:query-meters
- Understanding the Cache Statistics

3.2.1 Output From xdmp:query-meters

The `xdmp:query-meters` function produces an XML document that conforms to the `query-meters.xsd` schema. The `query-meters.xsd` schema is loaded into the schemas database and is copied to the `<install_dir>/Config` directory at installation time.

The output shows elapsed time for the query, hits and misses from the various query caches, and information about fragments and documents the query accessed. The fragment output prints one element per fragment root name (not one element per fragment). The document output prints one element per document URI. For sample `xdmp:query-meters` output, see “Sample xdmp:query-meters Output” on page 28.

3.2.2 Understanding the Cache Statistics

There are several elements in the `xdmp:query-meters` output that list the number of hits and misses on the query caches. Cache hits are good, and indicate the query is running in an optimized fashion. Cache misses indicate that the query could not retrieve its results directly from the cache, and had to read the data from disk. Because disk I/O is expensive relative to reading from memory, cache misses indicate that the query might be able to be optimized, either by rewriting the parts of the query that have cache misses to better take advantage of the indexes or by adding indexes that the query can use.

MarkLogic Server has several different caches used for query processing. In general, these caches load index data into memory, providing optimized query processing for a large variety of queries.

The `xdmp:query-meters` function lists hits and misses for the following caches:

- list cache

  The *list cache* holds search term lists in memory and helps optimize XPath expressions and text searches.

- expanded tree cache

  The *expanded tree cache* holds the uncompressed XML data in memory (in its expanded format).

- compressed tree cache

  The *compressed tree cache* holds compressed XML tree data in memory. The data is cached in memory in the same compressed format that is stored on disk.

- in-memory cache

  The *in-memory cache* holds data that was recently added to the system and is still in an in-memory stand; that is, it holds data that has not yet been written to disk.

- value cache

  The *value cache* exists only for the duration of a query. It holds typed values and optimizes queries that perform frequent conversion of nodes to typed values. Each miss for the value cache indicates that an XML node must be converted to a typed value.

- regular expression cache

  The *regular expression cache* (`regexp-cache`) exists only for the duration of a query. It holds compiled regular expressions, and optimizes queries that use a regular expression multiple times.

- link cache

  The *link cache* exists only for the duration of a query. The link cache holds the relationships between parent and child nodes, reusing that relationship throughout the query execution to optimize query processing.

The cache hits and misses are also broken down by fragment and by document. Each fragment element represents all of the fragments with the specified name. Each document element represents a document with the specified URI. The fragment and document elements of the `xdmp:query-meters` output show cache hits and misses for the expanded tree cache. These statistics can help you isolate which documents or fragments are being optimally processed. If a given document or fragment gets cache misses, you might be able to add indexes or rewrite the query to speed performance.

To help tune query performance, run the `xdmp:query-meters` function with your query and look for cache misses in the `xdmp:query-meters` output; cache misses indicate areas where the query can be tuned (either by rewriting or by adding indexes) for better performance.

3.3 Understanding query-trace Output

The `xdmp:query-trace` function logs output to the `<data_dir>/Logs/ErrorLog.txt` file during query execution. To start query tracing, concatenate the `xdmp:query-trace(true())` function at the part of your query where you want the tracing to begin, and add `xdmp:query-trace(false())` where you want tracing to stop. For example, the following query produces results for the query and logs the `query-trace` output to the `ErrorLog.txt` file:

```
xdmp:query-trace(true()),
doc("/myDocuments/hello.xml")//a/b/c
, xdmp:query-trace(false())
```

For its function signature, see the `xdmp:query-trace` function in *MarkLogic XQuery and XSLT Function Reference*.

The following subsections describe the output of the `xdmp:query-trace` function:

- What query-trace Logs
- Interpreting the Log Messages
- Fully Searchable Paths and cts:search Operations

3.3.1 What query-trace Logs

The `xdmp:query-trace` function prints INFO-level messages to the log file while a query is executing. It prints one log message for each XPath expression, and at least one log message for each step in the XPath expression. It also prints messages for predicates and other parts of query evaluation. Therefore, `xdmp:query-trace` can potentially log a large number of messages to the log file, particularly for complex queries that contain very deep XPath expressions and many searches.

The `xdmp:query-trace` function logs the following information about the query processing and execution:

- XPath Expression Analysis Messages
- Constraint Analysis Messages
- Search Execution Messages

3.3.1.1 XPath Expression Analysis Messages

The `xdmp:query-trace` function prints INFO-level messages to the log file about the XPath expressions in the query. The messages log whether an XPath expression is `searchable`. A `searchable` expression is one which can be optimized by using the indexes. The `query-trace` output shows which steps in the XPath expression are or are not `searchable` with the indexes.

For query tuning, the most important thing the log output has is the information about whether an expression is `searchable` or not. In general, `searchable` expressions can use the indexes to execute, and therefore execute fast. Expressions that are `unsearchable` cannot use the indexes, and must fetch the data from disks. For a summary of how to read the log messages, see "Interpreting the Log Messages" on page 24.

3.3.1.2 Constraint Analysis Messages

The constraint analysis phase of the `query-trace` output prints log messages about predicates in XPath expressions and `where` clauses. At the beginning of each constraint analysis section, you will see a message similar to the following:

```
2004-12-06 11:57:18.325 Info: line 21: Gathering constraints.
```

The output logs one message for each step in the XPath expression that contributes to the constraint. It only prints messages about constraints that can be evaluated using the indexes; unoptimized constraints do not generate any `query-trace` output. When the predicate constraint is reached, the log shows a message similar to the following:

```
2004-12-15 10:44:57.734 Info: line 2: Comparison contributed hash value
constraint: Heading-2 = "hello"
```

This message corresponds to an XPath expression with a predicate like the following:

```
doc("/myDocuments/hello.xml")/XML//Heading-2[. = "hello"]
```

The log message text `hash value constraint` indicates that the optimizer used the standard indexes (word search, stemmed search, and so on, as set up in the database configuration) to evaluate this predicate. Equality constraints on predicates use the standard indexes for evaluation, and this makes the evaluation perform fast.

Inequality constraints such as greater than (`gt` or `>`) and less than (`lt` or `<`) cannot be evaluated using the standard indexes. For inequality constraints to be optimized, you must have an element (range) index on the element used in the comparison. If you have an inequality constraint and have an element index on the element used in the comparison, the log shows a message similar to the following for the constraint `Heading-2 > "hello"`:

```
2004-12-15 10:44:57.734 Info: line 2: Comparison contributed range
value constraint: Heading-2 > "hello"
```

The log message text `range value constraint` indicates that the optimizer used an element index to evaluate the query.

Note: If neither the standard indexes nor an element index is used to evaluate a constraint, no such log message appears, and the constraint is not optimized.

3.3.1.3 Search Execution Messages

The `xdmp:query-trace` function also logs detailed information about how many fragments are used to evaluate a query. These messages show the number of fragments that are filtered. When a fragment is filtered, it means that the indexes found a possible match for the query in that fragment, and the fragment must then be retrieved to make sure it meets all of the query criteria. In a well-optimized query, the number of fragments filtered will be close to the number of fragments that satisfy the query.

If a query returns no results, or if it can be answered directly from the indexes, there will be no fragments filtered, and the log shows messages similar to the following:

```
2004-12-15 10:44:57.367 Info: line 2: Executing search.
2004-12-15 10:44:57.367 Info: line 2: Selected 0 fragments to filter
```

If the query results come from a single fragment, and the query uses either the standard or element (range) indexes for its evaluation, the log shows messages similar to the following:

```
2004-12-15 11:14:10.926 Info: line 2: Executing search.
2004-12-15 11:14:10.926 Info: line 2: Selected 1 fragment to filter
```

The line that says `Selected 1 fragment to filter` indicates how many candidate fragment references were returned from the index resolution stage of query processing. For a query that makes good use of the indexes, the number of fragments filtered is close to the number of fragments returned in the query results. For example, if there are 45 fragments that match a given query, and if `xdmp:query-trace` shows 45 fragments filtered, then that query is making good use of the indexes (because it does not have to filter any fragments that end up not contributing to the query result).

In most cases, the smaller the number of fragments selected to filter, the faster the query performs. An exception to this is if you are doing unfiltered searches, as unfiltered search skip the filtering stage of query processing. For details on unfiltered searches, see “Fast Pagination and Unfiltered Searches” on page 13.

3.3.2 Interpreting the Log Messages

The messages written to the log from the `xdmp:query-trace` function help you to determine if there are ways to optimize the performance of a query. The following is a summary of some important things to look for when interpreting the `xdmp:query-trace` output:

- The output is written to the `ErrorLog.txt` file.
- Log messages with the term `searchable` are good—this means indexes can be used to execute this part of the query (which in turn means the query will execute fast).

- Suspect problem areas when you see log messages with the term `unsearchable`—this means the indexes cannot be used to execute this part of the query.
- Log messages with the term `does not use indexes` mean that there might be XPath steps below this step that are `searchable`, but this step or predicate will not be resolved directly from the indexes (known as *conditionally searchable*). This is not necessarily bad, as searches with steps that do not use the indexes can still be fast, but it is not as good as `searchable`.
- Log messages with the term `comparison contributed **hash** value` constraint indicate that this predicate used the standard indexes to execute (which in turn indicates an optimized predicate evaluation).
- Log messages with the term `comparison contributed **range** value` constraint indicate that this predicate used an element (range) index to execute (which in turn indicates an optimized predicate evaluation).
- No `hash` or `range` message in the constraint section indicates that the constraint needed to scan the fragment to execute, and could not be optimized from the indexes.
- In the execution phase, the `xdmp:query-trace` output has a log message indicating the number of fragments filtered. In a fully optimized query, that number is equal to the number of fragments that the query returns (the number you would get if you wrapped the search portion of the query in an `xdmp:estimate` call). As the number of fragments filtered increases, and particularly as the number of fragments filtered grows past the number of fragments that ultimately match the query, the amount of work needed to execute the query increases (which in turn causes performance to slow).
- XPath predicates that cross fragment boundaries are `unsearchable` (cannot use indexes). For example, if a document is fragmented at the `b` element, then you should make sure predicates do not cross the b boundary. Therefore, the following expression:

  ```
  /a/b[c="1"]/../d
  ```

  will run faster than the following expression:

  ```
  /a[b/c="1"]/d
  ```

3.3.3 Fully Searchable Paths and cts:search Operations

A fully searchable XPath expression is one that can be efficiently resolved using the indexes. The following are examples of contexts requiring a fully searchable expression:

- The first parameter of the `cts:search` XQuery function. This parameter identifies the nodes to which MarkLogic applies the search query.
- The `searchable-expression` query option usable with the Search API and the Client APIs.

  Note: Due to security and performance considerations, beginning in MarkLogic 9.0-10, the `searchable-expression` property/element in query options is deprecated.

Please see Search API searchable-expression Deprecated in the Release Notes for more information.

- Optimization of the XQuery `order by` clause of a FLOWR expression; for details, see "Sorting Searches Using Range Indexes" on page 35.

(Search operations such as the `cts.search` or `jsearch.documents` JavaScript functions or the `search:search` XQuery function implicitly use the (fully searchable) expression `fn:doc()` for the same purpose as the first parameter of `cts:search`.)

A *searchable* XPath expression or path step is one that can be fully resolved out of the indexes. An XPath expression is searchable if it meets the following criteria:

- Rooted in an `fn:doc`, `fn:collection`, or `xdmp:document-properties` call in XQuery, or in an `fn.doc`, `cts.doc`, `fn.collection`, or `xdmp.document-properties` call in Server-Side JavaScript.
- Uses only forward element axes, such as "/" and "//.

A path step is searchable if it can be resolved out of the indexes. Generally, this means a relatively simple expression. For example, if it uses predicates, the predicates are searchable and the step uses only forward axes.

A path step can be *searchable* (resolvable out of the indexes), *unsearchable* (not resolvable out of the indexes), or *conditionally searchable* (might or might not be resolvable out of the indexes). A searchable expression can include predicates, but some predicates will make a path step conditionally searchable.

A *fully searchable* XPath expression is one in which no path steps are unsearchable and the last step is searchable. In a context such as the first parameter of `cts:search`, the XPath expressions must be fully searchable. In other contexts, such as when traversing to a node, MarkLogic will still attempt to satsify unsearchable or conditionally searchable path steps without using the indexes, but performance will suffer.

You can often make an XPath expression fully searchable by rewriting the expression or adding new indexes.

A *partially searchable* XPath expression is one in which the first path step is searchable, but the rest of the expression does not meet the requirements for fully searchable. For example, a partially searchable XPath expression might contain an unsearchable path step. A partially searchable expression cannot be evaluated as efficiently as a fully searchable expression.

You cannot use a partially searchable expression as the first parameter of `cts:search`, but you can use one to select nodes in other contexts, such as when selecting nodes via XPath for a non-search operation. You can also use a partially searchable expression as the value of the `-document_selector` option of an mlcp export command.

The best way to determine if an XPath expression and it's path steps are searchable is to examine the output of `xdmp:query-trace` (XQuery) or `xdmp.queryTrace` (JavaScript). If the trace output contains no entries tagged as unsearchable, then the expression is fully searchable. For an example, see "Sample xdmp:query-trace Output" on page 29.

3.4 Using xdmp:plan to View the Evaluation Plan

You can use the `xdmp:plan` built-in function to see the search and execution plan for a query. It takes an XQuery expression, and it returns an XML report providing information about how the indexes will be used if you were to run the expression. It provides much of the information shown in `xdmp:query-trace`, as well as some more information about the query terms selected from the index. The `xdmp:plan` output is useful in determining if an expression is optimized properly and if your range indexes are being used as you expect them to be.

Running an `xdmp:plan` on a search is similar to running an `xdmp:estimate` on a search, and the results of the estimate are included in the `xdmp:plan` output. If the search cannot be run in a plan or estimate, then it will throw an `XDMP-UNSEARCHABLE` exception. For more details and the signature of `xdmp:plan`, see the *MarkLogic XQuery and XSLT Function Reference*.

3.5 Examples

This section shows sample output from the `xdmp:query-meters` and `xdmp:query-trace` functions. The following examples are included:

- Sample xdmp:query-meters Output
- Sample xdmp:query-trace Output
- Logging Both query-meters and query-trace Output

3.5.1 Sample xdmp:query-meters Output

The following listing shows sample output from the `xdmp:query-meters` function:

```
<qm:query-meters xsi:schemaLocation="http://marklogic.com/xdmp/query-m
eters query-meters.xsd" xmlns:qm="http://marklogic.com/xdmp/query-mete
rs" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <qm:elapsed-time>PT0.000283S</qm:elapsed-time>
  <qm:requests>0</qm:requests>
  <qm:list-cache-hits>0</qm:list-cache-hits>
  <qm:list-cache-misses>0</qm:list-cache-misses>
  <qm:list-size>0</qm:list-size>
  <qm:in-memory-list-hits>0</qm:in-memory-list-hits>
  <qm:triple-cache-hits>0</qm:triple-cache-hits>
  <qm:triple-cache-misses>0</qm:triple-cache-misses>
  <qm:triple-value-cache-hits>0</qm:triple-value-cache-hits>
  <qm:triple-value-cache-misses>0</qm:triple-value-cache-misses>
  <qm:expanded-tree-cache-hits>0</qm:expanded-tree-cache-hits>
  <qm:expanded-tree-cache-misses>0</qm:expanded-tree-cache-misses>
  <qm:compressed-tree-cache-hits>0</qm:compressed-tree-cache-hits>
  <qm:compressed-tree-cache-misses>0</qm:compressed-tree-cache-misses>
  <qm:compressed-tree-size>0</qm:compressed-tree-size>
  <qm:in-memory-compressed-tree-hits>0
      </qm:in-memory-compressed-tree-hits>
  <qm:value-cache-hits>0</qm:value-cache-hits>
  <qm:value-cache-misses>0</qm:value-cache-misses>
  <qm:regexp-cache-hits>0</qm:regexp-cache-hits>
  <qm:regexp-cache-misses>0</qm:regexp-cache-misses>
  <qm:link-cache-hits>0</qm:link-cache-hits>
  <qm:link-cache-misses>0</qm:link-cache-misses>
  <qm:filter-hits>0</qm:filter-hits>
  <qm:filter-misses>0</qm:filter-misses>
  <qm:fragments-added>0</qm:fragments-added>
  <qm:fragments-deleted>0</qm:fragments-deleted>
  <qm:fs-program-cache-hits>0</qm:fs-program-cache-hits>
  <qm:fs-program-cache-misses>0</qm:fs-program-cache-misses>
  <qm:db-program-cache-hits>0</qm:db-program-cache-hits>
  <qm:db-program-cache-misses>1</qm:db-program-cache-misses>
  <qm:env-program-cache-hits>0</qm:env-program-cache-hits>
  <qm:env-program-cache-misses>0</qm:env-program-cache-misses>
  <qm:fs-main-module-sequence-cache-hits>0
      </qm:fs-main-module-sequence-cache-hits>
  <qm:fs-main-module-sequence-cache-misses>0
```

```
        </qm:fs-main-module-sequence-cache-misses>
      <qm:db-main-module-sequence-cache-hits>0
        </qm:db-main-module-sequence-cache-hits>
      <qm:db-main-module-sequence-cache-misses>0
        </qm:db-main-module-sequence-cache-misses>
      <qm:fs-library-module-cache-hits>0</qm:fs-library-module-cache-hits>
      <qm:fs-library-module-cache-misses>0
        </qm:fs-library-module-cache-misses>
      <qm:db-library-module-cache-hits>0</qm:db-library-module-cache-hits>
      <qm:db-library-module-cache-misses>0
        </qm:db-library-module-cache-misses>
      <qm:read-locks>0</qm:read-locks>
      <qm:write-locks>0</qm:write-locks>
      <qm:lock-time>0</qm:lock-time>
      <qm:contemporaneous-timestamp-time>1.1e-06
        </qm:contemporaneous-timestamp-time>
      <qm:compile-time>0.0001972</qm:compile-time>
      <qm:commit-time>0</qm:commit-time>
      <qm:run-time>0</qm:run-time>
      <qm:indexing-time>0</qm:indexing-time>
      <qm:fs-schema-cache-hits>0</qm:fs-schema-cache-hits>
      <qm:fs-schema-cache-misses>0</qm:fs-schema-cache-misses>
      <qm:db-schema-cache-hits>0</qm:db-schema-cache-hits>
      <qm:db-schema-cache-misses>0</qm:db-schema-cache-misses>
      <qm:env-schema-cache-hits>0</qm:env-schema-cache-hits>
      <qm:env-schema-cache-misses>0</qm:env-schema-cache-misses>
      <qm:fragments></qm:fragments>
      <qm:documents></qm:documents>
      <qm:hosts></qm:hosts>
    </qm:query-meters>
```

3.5.2 Sample xdmp:query-trace Output

The following sample query:

XQuery	Server-Side JavaScript

`xquery version "1.0-ml";`
 	
`xdmp:query-trace(fn:true()),`
`fn:doc("/myDocs/file.xml")//Node-2,`
`xdmp:query-trace(fn:false())`
	`'use strict';`
	
	`xdmp.queryTrace(true);`
	`cts.doc("/myDocs/file.xml")//Node-2;`
	`xdmp.queryTrace(false);`

produces `xdmp:query-trace` output similar to the following in the `ErrorLog.txt` file of your App Server. The timestamp and "Info:" message prefix has been elided for readability.

```
... Analyzing path: doc("/myDocs/file.xml")/descendant::Node-2
... Step 1 is searchable:      doc("/myDocs/file.xml")
... Step 2 axis does not use indexes: descendant
... Step 2 test is searchable: Node-2
... Step 2 is searchable: descendant::Node-2
... Path is searchable.
... Gathering constraints.
... Step 1 contributed 1 constraint: fn.doc("/myDocs/file.xml")
... Executing search.
... Selected 1 fragment to filter
```

3.5.3 Logging Both query-meters and query-trace Output

You can use the `xdmp:log` function to write the `xdmp:query-meters` output to the log file with the `xdmp:query-trace` output as follows:

```
xdmp:log("
****
**** Begin query trace and meter log
****
"),
xdmp:query-trace(true()),
doc("/myDocs/file.xml")//Heading-2[. = "hello"]
,
xdmp:log(xdmp:query-meters())
,
xdmp:log("
****
**** End query trace and meter log
****
")
```

This query produces log output in the `ErrorLog.txt` file like the following:

```
2004-12-08 15:48:01.502 Info:

****
**** Begin query trace and meter log
****

004-12-08 15:48:01.502 Info: line 9: Analyzing path:
     doc("/myDocs/file.xml")/descendant::Node-1
2004-12-08 15:48:01.502 Info: line 9: Step 1 is searchable:
     doc("/myDocs/file.xml")
004-12-08 15:48:01.502 Info: line 2: Step 2 axis does not use
     indexes: descendant
004-12-08 15:48:01.502 Info: line 2: Step 2 test is searchable: Node-2
004-12-08 15:48:01.502 Info: line 2: Step 2 is searchable:
     descendant::Node-2
```

```
004-12-08 15:48:01.502 Info: line 2: Path is searchable.
004-12-08 15:48:01.502 Info: line 2: Gathering constraints.
2004-12-08 15:48:01.502 Info: line 2: Step 2 test contributed 1
     constraint: Node-2
2004-12-08 15:48:01.502 Info: line 2: Executing search.
004-12-08 15:48:01.502 Info: line 2: Selected 1 fragment to filter
2004-12-08 15:48:01.502 Info: <qm:query-meters xsi:schemaLocation="htt
p://marklogic.com/xdmp/query-meters query-meters.xsd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:qm="http://marklogic.com/xdmp/query-meters">
<qm:elapsed-time>PT0.0000685S
  </qm:elapsed-time>
  <qm:requests>0
  </qm:requests>
  <qm:list-cache-hits>0
  </qm:list-cache-hits>
  <qm:list-cache-misses>0
  </qm:list-cache-misses>
  <qm:list-size>0
  </qm:list-size>
  <qm:in-memory-list-hits>0
  </qm:in-memory-list-hits>
  <qm:triple-cache-hits>0
  </qm:triple-cache-hits>
  <qm:triple-cache-misses>0
  </qm:triple-cache-misses>
  <qm:triple-value-cache-hits>0
  </qm:triple-value-cache-hits>
  <qm:triple-value-cache-misses>0
  </qm:triple-value-cache-misses>
  <qm:expanded-tree-cache-hits>0
  </qm:expanded-tree-cache-hits>
  <qm:expanded-tree-cache-misses>0
  </qm:expanded-tree-cache-misses>
  <qm:compressed-tree-cache-hits>0
  </qm:compressed-tree-cache-hits>
  <qm:compressed-tree-cache-misses>0
  </qm:compressed-tree-cache-misses>
  <qm:compressed-tree-size>0
  </qm:compressed-tree-size>
  <qm:in-memory-compressed-tree-hits>0
  </qm:in-memory-compressed-tree-hits>
  <qm:value-cache-hits>0
  </qm:value-cache-hits>
  <qm:value-cache-misses>0
  </qm:value-cache-misses>
  <qm:regexp-cache-hits>0
  </qm:regexp-cache-hits>
  <qm:regexp-cache-misses>0
  </qm:regexp-cache-misses>
  <qm:link-cache-hits>0
  </qm:link-cache-hits>
  <qm:link-cache-misses>0
  </qm:link-cache-misses>
```

```
<qm:filter-hits>0
</qm:filter-hits>
<qm:filter-misses>0
</qm:filter-misses>
<qm:fragments-added>0
</qm:fragments-added>
<qm:fragments-deleted>0
</qm:fragments-deleted>
<qm:fs-program-cache-hits>0
</qm:fs-program-cache-hits>
<qm:fs-program-cache-misses>0
</qm:fs-program-cache-misses>
<qm:db-program-cache-hits>0
</qm:db-program-cache-hits>
<qm:db-program-cache-misses>1
</qm:db-program-cache-misses>
<qm:env-program-cache-hits>0
</qm:env-program-cache-hits>
<qm:env-program-cache-misses>0
</qm:env-program-cache-misses>
<qm:fs-main-module-sequence-cache-hits>0
</qm:fs-main-module-sequence-cache-hits>
<qm:fs-main-module-sequence-cache-misses>0
</qm:fs-main-module-sequence-cache-misses>
<qm:db-main-module-sequence-cache-hits>0
</qm:db-main-module-sequence-cache-hits>
<qm:db-main-module-sequence-cache-misses>0
</qm:db-main-module-sequence-cache-misses>
<qm:fs-library-module-cache-hits>0
</qm:fs-library-module-cache-hits>
<qm:fs-library-module-cache-misses>0
</qm:fs-library-module-cache-misses>
<qm:db-library-module-cache-hits>0
</qm:db-library-module-cache-hits>
<qm:db-library-module-cache-misses>0
</qm:db-library-module-cache-misses>
<qm:read-locks>0
</qm:read-locks>
<qm:write-locks>0
</qm:write-locks>
<qm:lock-time>0
</qm:lock-time>
<qm:contemporaneous-timestamp-time>0
</qm:contemporaneous-timestamp-time>
<qm:compile-time>0.0001729
</qm:compile-time>
<qm:commit-time>0
</qm:commit-time>
<qm:run-time>0
</qm:run-time>
<qm:indexing-time>0
</qm:indexing-time>
<qm:fs-schema-cache-hits>0
</qm:fs-schema-cache-hits>
```

```
    <qm:fs-schema-cache-misses>0
    </qm:fs-schema-cache-misses>
    <qm:db-schema-cache-hits>0
    </qm:db-schema-cache-hits>
    <qm:db-schema-cache-misses>0
    </qm:db-schema-cache-misses>
    <qm:env-schema-cache-hits>0
    </qm:env-schema-cache-hits>
    <qm:env-schema-cache-misses>0
    </qm:env-schema-cache-misses>
    <qm:fragments>
    </qm:fragments>
    <qm:documents>
    </qm:documents>
    <qm:hosts>
    </qm:hosts>
</qm:query-meters>
2004-12-08 15:48:01.502 Info:

****
**** End query trace and meter log
****
General Methodology for Tuning a Query
```

The following are general steps you can take to analyze and tune query performance. These steps represent a methodology; the actual steps you take will depend on your application and queries.

1. Identify the application where you see query performance slower than you expect.

2. In the application, break apart different parts of the query into separate queries and run them separately.

3. If you identify code that appears to run slowly, append `xdmp:query-meters()` to the end of the code and run it again. For details, see “Understanding query-meters Output” on page 20.

4. In the `xdmp:query-meters` output, record the elapsed time and look for cache misses.

5. Run the query several times and compare the `xdmp:query-meters` output between the different runs. There are some query caches that are populated when a query runs the first time, and can improve the performance of subsequent query runs.

6. Continue to try and simplify the query, helping to isolate where it might be running slow.

7. When you have isolated the query down to as simple a case as possible, add `xdmp:query-trace(true())` to the beginning of the query and run it again. For details, see “Understanding query-trace Output” on page 22.

8. Examine the `query-trace` output in the `ErrorLog.txt` file. Look for XPath steps that are `unsearchable`.

9. If you find `unsearchable` steps, see if there are ways to rewrite the query so those steps become `searchable`.

10. Examine the `constraints` entries of the `query-trace` log output. For details, see "Constraint Analysis Messages" on page 23.

11. Check the `query-trace` log output for the number of fragments used to filter. This number should be the close to or the same as the number of fragments that match the searchable expression (the number returned from `xdmp:estimate`) if the query is well optimized.

12. Check your indexing options. Add indexes if the proper indexes are not built. For example, if stemmed or word indexes are not built, many XPath steps will be `unsearchable`. Also, if your query contains inequality constraints, you will need element (range) indexes to optimize those constraints.

13. After making query and/or index changes, rerun the query with `xdmp:query-meters()` to see if the execution time has decreased and the number of cache misses has decreased.

14. Continue iteratively with this methodology until you are satisfied that the query execution is fast and well optimized.

4.0 Sorting Searches Using Range Indexes

Range indexes are value indexes that are typed and are sorted in type order. You can run searches in MarkLogic and efficiently sort the search using a range index value. There are two ways to specify the range indexes in a search:

- Using a cts:order Specification in a cts:search
- Optimizing Order By Expressions With Range Indexes

The first way, introduced in MarkLogic 8, is the easier way; the second way still works for backward compatibility.

4.1 Using a cts:order Specification in a cts:search

By default, a `cts:search` is sorted in relevance order. If you want to instead sort the search by a value in the documents returned, you can create a range index on the sort value and then specify that index in the `cts:search`. The easiest way to specify a sort order in a search is by adding a `cts:order` specification to your `cts:search` statment. This section describes how to construct such searches, and includes the following parts:

- Creating a cts:order Specification
- Using the cts:order Specification in a Search

4.1.1 Creating a cts:order Specification

The `cts:order` type is a native type in MarkLogic. You can create `cts:order` specifications using the following constructors:

- `cts:index-order`
- `cts:score-order`
- `cts:confidence-order`
- `cts:fitness-order`
- `cts:quality-order`
- `cts:document-order`
- `cts:unordered`

You can specify a sequence of `cts:order` constructors and it will result ordering by the first in the sequence, followed by the next, and so on. For example, you might want to order first on a path range index of `/a/b/c`, with a secondary ordering on `//title`.

> **Note:** Any order you specify with a cts:index-order constructor requires the appropriate range index to be created in MarkLogic, otherwise the search will throw an exception.

The default sort order is equivalent to `(cts:score-order("descending"),cts:document-order("ascending"))`.

4.1.2 Using the cts:order Specification in a Search

You can use the cts:order specification in a `cts:search` in XQuery or a `cts.search` in Javascript. The `cts:order` is part of the `$options` parameter.

4.2 Optimizing Order By Expressions With Range Indexes

When you have queries that include an `order by` expression, you can create range indexes (for example, element indexes, attribute indexes, or path indexes) on the element(s) or attributes(s) in the `order by` expression to speed performance of those types of queries. This chapter describes this optimization and how to use it in your queries, and includes the following parts:

- Speed Up Order By Performance
- Rules for Order By Optimization
- Creating Range Indexes
- Example Order By Queries

Note: Starting with MarkLogic 8, you can get the same sorting results by specifying a `cts:order` specification in a search. For details, see “Using a cts:order Specification in a cts:search” on page 35.

4.2.1 Speed Up Order By Performance

MarkLogic Server allows you to create indexes on elements to speed up performance of queries that order the results based on that element. The `order by` clause is the “O” in the XQuery `FLWOR` expression, and it allows you to sort the results of a query based on one ore more elements. The `order by` optimization speeds up queries that order the results and then return a subset of those results (for example, the first 10 results).

4.2.2 Rules for Order By Optimization

The following rules apply to a query in order for the `order by` optimization to apply:

- Optimizes subsets of `order by` queries. For example:

  ```
  (FLWOR)[1 to 20]
  ```

  where `FLWOR` is an XQuery `FLWOR` expression.

- Uses range indexes (for example, element, attribute, and/or path range indexes).
- The sequence bound to the `for` variable must be fully searchable XPath expression or a `cts:search` expression. See “Fully Searchable Paths and cts:search Operations” on page 25.
- The order by expression must be on variables bound in the `for` clause; queries that have `order by` expressions on variables bound to a sequence in a `let` clause are *not* optimized.

- There must be a range index on the last step of the order by expression. For example:

  ```
  order by $x/bar/foo
  ```

  needs a range index on `foo` to execute with the `order by` optimization.

- The type of the order by expression must be the type of the range index, either implicitly, through a schema, or through an explicit cast. If there is a cast in the order by expression, than it must be to the type of the range index.
- You can have order by expressions with multiple items, as long as there is a range index on each item. For example:

  ```
  order by $x/foo, $x/bar
  ```

  as long as there are range indexes for `foo` and `bar`.

- The XPath expression in the order by expression must be a simple relative path; no math or other expressions are allowed.
- It does not matter what the `let`, `where`, or `return` clauses are; these do not effect the optimization.
- If you order by `cts:score($x)`, `cts:confidence($x)`, or `cts:quality($x)`, no range index is required.
- You can specify either `ascending` or `descending` orders (optionally).

  With the `cts:search` parameter `cts:index-order`, results with no comparable index value are always returned at the end of the ordered result sequence. With an XQuery `order by` clause, results with no comparable value are normally returned by MarkLogic at the end of the ordered result sequence.

  You can specify either `empty greatest` or `empty least`, but empties always need to be at the end for the order by optimizations to work. For example, `empty greatest` is optimized with `ascending`; `empty least` is optimized with `descending`. If neither is specified, MarkLogic chooses the order that is optimized. The following example goes against the default. It orders the list by `$doc/document/modified_date`, `ascending` order, with `empty least`:

  ```
  xquery version "1.0-ml";
  for $doc in fn:doc()
  order by $doc/document/modified_date ascending empty least
  return $doc
  ```

- Optimized `order by` clauses implicitly add `order by` expressions for `cts:score` and document order to the end of the `order by` expression.
- If you have a function that is a `FLWOR` expression (with the required fully searchable path and so forth), subsets of that will be optimized. For example:

```
xquery version "1.0-ml";
declare function local:foo()
{
for $x in //a/b/c
order by $x/d
return $x
};
( local:foo() )[1 to 10]
```

- The search or XPath expression must be part of the `FLWOR`, not bound to a variable that is referenced in the `FLWOR`. For example, the following will not be optimized:

  ```
  let $x := cts:search(/foo, "hello")
  return
  (for $y in $x
  order by $y/element
  return $y)[1 to 10]
  ```

  but the following will (given the other rules are followed):

  ```
  (for $y in cts:search(/foo, "hello")
  order by $y/element
  return $y)[1 to 10]
  ```

- You can use `xdmp:query-trace` to determine if a query is using the range indexes to optimize an order by expression. For details on using `xdmp:query-trace`, see "Understanding query-trace Output" on page 22.
- When using an order by with a `cts:search`, further optimization occurs if you specify the `"unfiltered"` option to `cts:search`. For example, if you order by a simple XPath expression and that expression returns a sequence, if the `cts:search` is `"filtered"` (which is the default) then the search will throw an exception (because it is illegal to order by a sequence of more than one item), but if you use the `"unfiltered"` option to `cts:search`, the search will complete and will use the range index. If there are multiple values that match the order by expression in an unfiltered `cts:search`, then it will use the maximum value (`fn:max($result/item())`) for `order by ascending` and the minimum value (`fn:min($result/item())`) for `order by decending`. For more details about unfiltered `cts:search`, see "Fast Pagination and Unfiltered Searches" on page 13.

4.2.3 Creating Range Indexes

You must create range indexes over the elements or attributes in which you order your result by in the `order by` expression. You create range indexes using the Admin interface by going to the Databases > *database_name* > Element Indexes or Attribute Indexes or Path Range Index page. Be sure to select the proper type for the element or attribute, or specify a path defining the element(s) and/or attributes(s) you want to index. For more details on creating indexes, see the *Administrator's Guide*.

4.2.4 Example Order By Queries

This section shows the following simple queries that use the order by optimizations:

- Order by a Single Element
- Order by Multiple Elements

4.2.4.1 Order by a Single Element

The following query returns the first 100 `lastname` elements. In order for this query to run optimized, there must be a range index defined on the `lastname` element.

```
(for $x in //myNode
order by $x/lastname
return
$x/lastname)[1 to 100]
```

If you enabled query tracing on this query (by adding `xdmp:query-trace(fn:true()),` to the beginning of the query, for example), the query trace output will show if the range index is being used for the optiomization. If the range index is not being used, the query-trace output looks similar to the following:

```
2009-05-15 15:56:05.046 Info: myAppServer: line 2:
xdmp:eval("xdmp:query-trace(fn:true()),&#13;&#10;(for $x in
//myNode&#13;&#...", (), <options
xmlns="xdmp:eval"><database>66188263795947693 4</database><modules>0</m
odules><defa...</options>)
2009-05-15 15:56:05.068 Info: myAppServer: line 2: Analyzing path for
$x: collection()/descendant::myNode
2009-05-15 15:56:05.068 Info: myAppServer: line 2: Step 1 is
searchable: collection()
2009-05-15 15:56:05.068 Info: myAppServer: line 2: Step 2 is
searchable: descendant::myNode
2009-05-15 15:56:05.068 Info: myAppServer: line 2: Path is fully
searchable.
2009-05-15 15:56:05.068 Info: myAppServer: line 2: Gathering
constraints.
2009-05-15 15:56:05.068 Info: myAppServer: line 2: Step 2 test
contributed 1 constraint: myNode
2009-05-15 15:56:05.068 Info: myAppServer: line 2: Executing search.
2009-05-15 15:56:05.089 Info: myAppServer: line 2: Selected 6 fragments
to filter.
```

The above output does not show that the range index is being used. This could be because the range index does not exist or it could indicate that one of the criteria for the order by optimizations is not met, as described in “Rules for Order By Optimization” on page 36.

When the correct range index is in place and the query is being optimized, the query-trace output will look similar to the following:

```
2009-05-15 15:58:04.145 Info: myAppServer: line 2:
xdmp:eval("xdmp:query-trace(fn:true()),&#13;&#10;(for $x in
//myNode&#13;&#...", (), <options
xmlns="xdmp:eval"><database>661882637959476934</database><modules>0</m
odules><defa...</options>)
2009-05-15 15:58:04.145 Info: myAppServer: line 2: Analyzing path for
$x: collection()/descendant::myNode
2009-05-15 15:58:04.145 Info: myAppServer: line 2: Step 1 is
searchable: collection()
2009-05-15 15:58:04.145 Info: myAppServer: line 2: Step 2 is
searchable: descendant::myNode
2009-05-15 15:58:04.145 Info: myAppServer: line 2: Path is fully
searchable.
2009-05-15 15:58:04.146 Info: myAppServer: line 2: Gathering
constraints.
2009-05-15 15:58:04.146 Info: myAppServer: line 2: Step 2 test
contributed 1 constraint: myNode
2009-05-15 15:58:04.146 Info: myAppServer: line 2: Order by clause
contributed 1 range ordering constraint for $x: order by $x/lastname
ascending
2009-05-15 15:58:04.146 Info: myAppServer: line 2: Executing search.
2009-05-15 15:58:04.183 Info: myAppServer: line 2: Selected 6 fragments
to filter.
```

Notice the line that says `Order by clause contributed 1 range constraint`. That line indicates that the query is being optimized by the range index (which is good).

4.2.4.2 Order by Multiple Elements

The following query returns the first 100 `myNode` elements, ordered by `lastname` and then `firstname`. For this query to run optimized, there must be a range index defined on the `lastname` and `firstname` elements.

```
(for $x in //myNode
order by $x/lastname, $x/firstname
return
$x)[1 to 100]
```

If you run query-trace with this query, that will verify whether the range indexes are being used.

5.0 Profiling Requests to Evaluate Performance

This chapter describes how to use the Performance Profiler built-in functions to examine the performance characteristics of XQuery and Server-Side JavaScript requests in MarkLogic Server. The profiling APIs enable you to gather statistics about the evaluation of your code on a per-request basis.

This chapter covers the following topics:

- Enabling Profiling on an App Server
- Understanding XQuery Profiling
- Understanding Server-Side JavaScript Profiling
- Profiling Examples

You can also use Query Console to profile your programs, as described in Profiling a Query in the *Query Console User Guide*.

5.1 Enabling Profiling on an App Server

To use the profiling API, you must first enable profiling on the App Server hosting your XQuery or Server-Side JavaScript program (or on the task server if you are profiling spawned queries).

The profile allow option must be set to true on the App Server configuraton page in the Admin UI. For example:



5.2 Understanding XQuery Profiling

This section describes profiling XQuery code and includes the following topics:

- Definitions and Terminology for the XQuery Profiling
- XQuery Profiling Overview
- XQuery Profiling API

5.2.1 Definitions and Terminology for the XQuery Profiling

The following table lists some terms and their definitions used in describing the XQuery profile API. Several of these terms are used in the data you can generate with the profiling API.

Term	Definition
XQuery Program	The XQuery main module fully expanded with any XQuery library modules needed for its evaluation. An XQuery program is sometimes referred to as a query, a statement, or a request. For more details on this terminology, see Understanding Transactions in MarkLogic Server in the *Application Developer's Guide*.
profiler	An application which measures the performance characteristics of a running program (in the case MarkLogic Server, of an XQuery program).
expression	The basic parse element of an XQuery program. Expressions can represent literal values, arithmetic operations, functions, function calls, and so on. Expressions can contain other expressions.
shallow time	The time spent evaluating a specific expression, not including time spent evaluating any expressions contained within the specific expression.
deep time	The total time spent evaluating an expression, including time spent evaluating any expressions contained within the specific expression.
elapsed time	Both shallow and deep time are expressed in elapsed wall clock time.
profile report	An XML report containing statistics for all of the expressions evaluated while profiling was enabled. For a sample profile report, see "Simple Enable and Disable XQuery Example" on page 45.

5.2.2 XQuery Profiling Overview

The XQuery profiling API in MarkLogic Server is not a query profile application (a *profiler*), but you could use the profiling API to build such an application. A profiler would measure the performance characteristics of an XQuery or JavaScript program.

You can use Query Console to generate and review a profiling report; for details, see Profiling a Query in the *Query Console User Guide*. You can also simply use the profiling API to generate profiling data and then either manually analyze the data or use XQuery to extract details and format a report.

When profiling is enabled, you can use the `prof:enable`, `prof:report`, and other profile functions to generate profiling statistics for the evaluation of individual XQuery programs.

The XQuery profiling API cannot "see into" a Server-Side JavaScript function. If your code calls a JavaScript function, you will only see data for the top level call, not any JavaScript functions called by that function. The time spent appears as a single block.

Profiling helps you see where a query is spending its processing time. MarkLogic gathers statistics during the evaluation portion of the query, at the individual expression level. Time spent in the data node portion of the query (time spent gathering content from the forests) is included in the expression time for the expression that requested the content from the forest (for example, a `cts:search`). For each expression, the profile report shows the shallow time and the deep time (see "Definitions and Terminology for the XQuery Profiling" on page 42 for these definitions).

All profiling information is gathered on a per-request basis; there is no notion of profiling a set of requests, although it is possible to write an application that performs such aggregation.

5.2.3 XQuery Profiling API

The following functions are included in the XQuery Performance Profiler API:

- `prof:allowed`
- `prof:disable`
- `prof:enable`
- `prof:eval`
- `prof:invoke`
- `prof:report`
- `prof:reset`

For details on the APIs and for their function signatures, see the *MarkLogic XQuery and XSLT Function Reference*. Note the following about the XQuery profile APIs.

- If profiling is not enabled on the App Server and in the XQuery program (via `prof:enable`), then profile APIs do not do anything except return the empty sequence.
- You can profile the currently running request (`prof:enable(xdmp:request())`), an evaluated request (`prof:eval`), or an invoked module (`prof:invoke`). To profile other requests, you need to debug the request; if you try to profile another request, MarkLogic Server throws the `DBG-NOTSTOPPED` exception.
- Constants (for example, `47` or `"hello"`) do not show up in the profile report.
- Constructed elements do not show up in the profile report.
- Profile time starts after the static analysis phase of query evaluation; it does not include the query parsing time.

- All of the profile APIs are in the `http://marklogic.com/xdmp/profile` namespace. The `prof` prefix is bound to this namespace, and is pre-configured in MarkLogic Server (so there is no need to define this namespace in your XQuery prolog).
- If you profile a request besides the currently running request, or if you start a new request for profiling using `prof:eval` or `prof:invoke`, you need one of the privileges `http://marklogic.com/xdmp/privileges/profile-my-requests` (to profile a request issued by the same user ID) or `http://marklogic.com/xdmp/privileges/profile-any-requests` (to profile a request issued by any user ID). If you are profiling the currently running request, no privileges are required.

5.3 Understanding Server-Side JavaScript Profiling

Profiling for Server-Side JavaScript is only available through the `prof.eval` built-in function or through Query Console. Profiling must be enabled for your App Server, as described in “Enabling Profiling on an App Server” on page 41. To learn about generating a profiling report using Query Console, see Profiling a Query in the *Query Console User Guide*.

When you profile Server-Side JavaScript, MarkLogic collects a CPU Profile from the JavaScript engine. The profiling data generated for JavaScript is based on sampling, so it will not reflect all functions called or accurate function hit counts. However, those areas of your code that consume the most CPU will still be readily apparent.

The JavaScript profiler cannot profile XQuery code. If your JavaScript code calls into XQuery, the time spent inside the XQuery function is a single block. The called XQuery function name will not include a namespace identifier in the profiling data.

You can save the output from profiling JavaScript to a file and import the data into the Profiles tab of the Chrome developers to analyze the data outside of MarkLogic. Query Console enables you to download a profiling report from within its UI.

You can also generate and save your own profiling report from `prof.eval` with code similar to the following:

```
'use strict';

const myQuery = "your code (as string) here";
xdmp.save(
  '/space/rest/junk.cpuprofile',
  fn.head(prof.eval(myQuery))
);
```

5.4 Profiling Examples

The following are code examples showing some simple usage patterns for the profile API. For details on the APIs, see the *MarkLogic XQuery and XSLT Function Reference*. This section shows the following examples:

- Simple Enable and Disable XQuery Example
- Returning a Part of the XQuery Profile Report
- JavaScript Profile Example

5.4.1 Simple Enable and Disable XQuery Example

The following examples show simple uses for the profile API.

```
let $req := xdmp:request()
let $dummy1 := prof:enable($req)
let $version := xdmp:version()
let $node :=
  <foo>
    <version>{$version}</version>
    <request>{$req}</request>
  </foo>
let $dummy2 := prof:disable($req)
let $dummy3 := fn:current-dateTime()
let $dummy4 := prof:enable($req)
let $dummy5 := 47
let $dummy6 := $node/foo/request
let $dummy6 := prof:disable($req)

return
prof:report($req)
```

This query returns the following report:

```
<prof:report
  xsi:schemaLocation="http://marklogic.com/xdmp/profile profile.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:prof="http://marklogic.com/xdmp/profile">
  <prof:metadata>
    <prof:overall-elapsed>PT0S</prof:overall-elapsed>
    <prof:created>2014-03-19T14:20:18.763-07:00</prof:created>
    <prof:server-version>7.0-4</prof:server-version>
  </prof:metadata>
  <prof:histogram>
    <prof:expression>
      <prof:expr-id>6467258264555963988</prof:expr-id>
      <prof:expr-source>xdmp:version()</prof:expr-source>
      <prof:uri/>
      <prof:line>3</prof:line>
      <prof:count>1</prof:count>
      <prof:shallow-time>PT0S</prof:shallow-time>
```

```
      <prof:deep-time>PT0S</prof:deep-time>
    </prof:expression>
    <prof:expression>
      <prof:expr-id>127913224145561857</prof:expr-id>
      <prof:expr-source>prof:disable($req)</prof:expr-source>
      <prof:uri/>
      <prof:line>9</prof:line>
      <prof:count>1</prof:count>
      <prof:shallow-time>PT0S</prof:shallow-time>
      <prof:deep-time>PT0S</prof:deep-time>
    </prof:expression>
    <prof:expression>
      <prof:expr-id>8377097069965042614</prof:expr-id>
      <prof:expr-source>prof:disable($req)</prof:expr-source>
      <prof:uri/>
      <prof:line>14</prof:line>
      <prof:count>1</prof:count>
      <prof:shallow-time>PT0S</prof:shallow-time>
      <prof:deep-time>PT0S</prof:deep-time>
    </prof:expression>
    <prof:expression>
      <prof:expr-id>10959125668150171452</prof:expr-id>
      <prof:expr-source>prof:enable($req)</prof:expr-source>
      <prof:uri/>
      <prof:line>2</prof:line>
      <prof:count>1</prof:count>
      <prof:shallow-time>PT0S</prof:shallow-time>
      <prof:deep-time>PT0S</prof:deep-time>
    </prof:expression>
    <prof:expression>
      <prof:expr-id>1823871875827665493</prof:expr-id>
      <prof:expr-source>$node/child::foo/child::request
        </prof:expr-source>
      <prof:uri/>
      <prof:line>14</prof:line>
      <prof:count>1</prof:count>
      <prof:shallow-time>PT0S</prof:shallow-time>
      <prof:deep-time>PT0S</prof:deep-time>
    </prof:expression>
    <prof:expression>
      <prof:expr-id>16669888445989754369</prof:expr-id>
      <prof:expr-source>prof:enable($req)</prof:expr-source>
      <prof:uri/>
      <prof:line>11</prof:line>
      <prof:count>1</prof:count>
      <prof:shallow-time>PT0S</prof:shallow-time>
      <prof:deep-time>PT0S</prof:deep-time>
    </prof:expression>
  </prof:histogram>
</prof:report>
```

You might note that the times are all zero. That is because this example does a trivial amount of work, and each expression took less than a millisecond to complete. But it does illustrate some things that are useful when looking at the report:

- The `fn:current-dateTime()` function does not appear in the report, because profiling was disabled at this stage of the XQuery program.
- The expressions are not necessarily in the order they are in the XQuery program. If you want them in order, you can take the `prof:expression` elements and order them by the `prof:line` element (or one of the time elements, or whatever makes sense for your reports).
- While the expression 47 occurs while profiling is enabled, it does not show up in the output because it is just a constant, and constants do not appear in the profile report.

5.4.2 Returning a Part of the XQuery Profile Report

Because the profile report is XML, you can use XQuery to manipulate the report to suit your needs. The following example returns only a expression elements of the profile report, wraps it in an element, and orders it by the deep time element.

```
xquery version "1.0-ml";

let $req := xdmp:request()
let $dummy := prof:enable($req)
let $foo := for $i in fn:doc() return xdmp:node-uri($i)
let $dummy2 := prof:disable($req)
return <foo>{
   for $j in prof:report($req)//*:expression
     order by xs:dayTimeDuration($j/*:deep-time)
     return $j
}</foo>
```

To see the results, copy the code and run it against MarkLogic Server. You will need to enable profiling on the HTTP Server Configuration page, otherwise the report will be empty.

5.4.3 JavaScript Profile Example

In Server-Side JavaScript, you can profile a JavaScript program as shown below. You can load this output into the Profiles tab of the Chrome developer tools to for further analysis:

```
prof.eval('cts.search("hello")')

=>
{"head":
  {"functionName":"(root)",
   "scriptId":0,
   "url":"",
   "lineNumber":0,
   "columnNumber":0,
   "hitCount":0,
```

```
"callUID":1,
"children":[
  {"functionName":"",
   "scriptId":42,
   "url":"/qconsole/endpoints/evaljs.sjs",
   "lineNumber":1,
   "columnNumber":1,
   "hitCount":0,
   "callUID":2,
   "children":[
    {"functionName":"addResponseHeader",
     "scriptId":0,
     "url":"",
     "lineNumber":0,
     "columnNumber":0,
     "hitCount":0,
     "callUID":3,
     "children":[
       {"functionName":"doEval",
        "scriptId":49,
        "url":
           "/MarkLogic/appservices/qconsole/qconsole-js-amped.sjs",
        "lineNumber":3,
        "columnNumber":20,
        "hitCount":0,
        "callUID":4,
        "children":[
          {"functionName":"eval",
           "scriptId":0,
           "url":"",
           "lineNumber":0,
           "columnNumber":0,
           "hitCount":0,
           "callUID":5,
           "children":[
             {"functionName":"",
              "scriptId":50,
              "url":"",
              "lineNumber":1,
              "columnNumber":1,
              "hitCount":0,
              "callUID":6,
              "children":[
                {"functionName":"eval",
                 "scriptId":0,
                 "url":"",
                 "lineNumber":0,
                 "columnNumber":0,
                 "hitCount":1,
                 "callUID":5,
                 "children":[],
                 "id":7}],
               "id":6}],
             "id":5}],
```

```
                "id":4}],
              "id":3}],
            "id":2}],
          "id":1},
    "startTime":3800158243229,
    "endTime":3800158265495,
    "samples":[7],
    "timestamps":[3800158243229]
  }
```

6.0 Disk Storage Considerations

This chapter describes how disk storage can affect performance of MarkLogic Server, and some of the storage options available for forests. It includes the following sections:

- Disk Storage and MarkLogic Server
- Fast Data Directory on Forests
- Large Data Directory on Forests
- HDFS, MapR-FS, and S3 Storage on Forests
- Windows Shared Disk Registry Settings and Permissions

6.1 Disk Storage and MarkLogic Server

MarkLogic Server applications can be very disk-intensive in their system demands. It is therefore very important to size your hardware appropriate for your workload and performance requirements. The topic of disk storage performance is complicated; there are many factors that can influence performance including disk controllers, network latency, the speed and quality of the disks, and other disk technologies such as storage area networks (SANs) and solid state drive (SSD). As with most performance issues, there are price/performance trade-offs to consider.

For example, SSDs are quite expensive compared with rotating drives. Conversely, HDFS (Hadoop Distributed Filesystem) or Amazon S3 (Simple Storage Service) storage can be quite inexpensive, but might not offer all of the speed of conventional disk systems.

6.2 Fast Data Directory on Forests

In the forest configuration for each forest, you can configure a Fast Data Directory. The Fast Data Directory is designed for fast filesystems such as SSDs with built-in disk controllers. The Fast Data Directory stores the forest journals and as many stands as will fit onto the filesystem; if the forest never grows beyond the size of the Fast Data Directory, then the entire forest will be stored in that directory. If there are multiple forests on the same host that point to the same Fast Data Directory, MarkLogic Server divides the space equally between the different forests.

When the Fast Data Directory begins to approach its capacity, during periodic merges, MarkLogic Server will start to put data in the regular Data Directory. By specifying a Fast Data Directory, you can get much of the advantage of using the fast disk hardware while only buying a relatively small SSD (or other fast disk system). For example, consider a scenario where you have an 8-core MarkLogic Server d-host that is hosting 4 forests. If you have good quality commodity server-class rotating disk system with many magnetic disk spindles (for example, 6 disks in some RAID configuration) having 2 terabytes of storage, and if you have a 250 gigabyte SSD (for example, a PCI I/O accelerator card) for the fast data directory, then you can get a significant amount of the benefit of having the SSD storage while keeping the cost down (because the rotating storage is several times less expensive than the SSD storage). In this scenario, each of the 4 forests could use up to 1/4 of the size of the SSD, or about 62.5 GB. Once the forest size grows close to that limit, then the Data Directory with the rotating storage is used.

6.3 Large Data Directory on Forests

Just like you might want a different class of disk for the Fast Data Directory, you might also want a different class of disk for the Large Data Directory. The Large Data Directory stores binary document that are larger than the Large Size Threshold specified in the database configuration. This filesystem is typically a very large filesystem, and it may use a different class of disk than your regular filesystem (or it may just be on a different set of the same disks). For more details about binary documents, see Working With Binary Documents in the *Application Developer's Guide*.

6.4 HDFS, MapR-FS, and S3 Storage on Forests

HDFS (Hadoop Distributed Filesystem) and Amazon S3 (Simple Storage Service) storage represent two approaches to large distributed filesystems, and it is possible to use both HDFS and S3 to store MarkLogic forest data. This section describes considerations for using HDFS and S3 for storing forest data in MarkLogic and contains the following topics:

- HDFS Storage
- MapR-FS Storage
- S3 Storage

Both HDFS and S3 can be very useful when implementing a tiered storage solution. For details on tiered storage, see Tiered Storage in the *Administrator's Guide*.

6.4.1 HDFS Storage

HDFS is a storage solution that uses Hadoop to manage a distributed filesystem. Hadoop has tools to specify how many copies of each file are replicated on how many different servers. HDFS gives you a high degree of control over your filesystem, as you can choose the disks to use, the computers to use, as well as configuration settings such as number of copies to replicate. MarkLogic can use Kerberos Secured HDFS as a file system on Linux platforms, as described in Kerberos Authentication for Secured HDFS in the *Security Guide*.

HDFS storage is supported with MarkLogic on the following HDFS platform:

- Cloudera CDH version 5.8
- Hortonworks HDP version 2.6

Internally, MarkLogic Server uses JNI to access HDFS. When you specify an HDFS path for one of the data directories, MarkLogic will write the forest data directly to HDFS according to the path specification.

When you set up an HDFS path as a forest directory, the path must be readable and writable by the user in which the MarkLogic Server process is running.

Because you can set up HDFS as a very large shared filesystem, it can be good not only for forest data, but as a destination for database backups.

An HDFS path is of the following form:

```
hdfs://<machine-name>:<port>/directory
```

so the following path would be to an hdfs filesystem accessed on a machine named `raymond.marklogic.com` on port 12345:

```
hdfs://raymond.marklogic.com:12345/directory
```

Each MarkLogic host that uses HDFS for forest storage requires access to the following:

- The Oracle/Sun Java JDK (or an Oracle/Sun JRE that includes JNI)
- Hadoop HDFS client JAR files
- Your Hadoop HDFS configuration files

The following HDFS configuration property settings are required:

- `dfs.support.append`: true. This is the default value.
- `dfs.namenode.accesstime.precision`: 1

The remainder of this section describes how to configure your hosts so that MarkLogic can find these components.

For details on the supported Java versions and how MarkLogic locates a JRE, see “Java Virtual Machine Requirements” on page 9 in the *Installation Guide*.

Though MarkLogic does not ship with HDFS client libraries, you can download client library bundles from http://developer.marklogic.com/products/hadoop.

Follow this procedure to make the bundled libraries and configuration files available to MarkLogic Server. You must follow this procedure on each MarkLogic host that uses HDFS for forest storage.

1. Download the Hadoop client bundle that corresponds to your Hadoop distribution from http://developer.marklogic.com/products/hadoop.

2. Unpack the bundle to one of the following locations: `/usr`, `/opt`, `/space`. For example, if you download the HDP bundle for MarkLogic 9.0-7 to `/opt`, then the following commands unpack the bundle to `/opt`.

   ```
   cd /opt
   gunzip hadoop-hdfs-hdp-9.0-7.tar.gz
   tar xf hadoop-hdfs-hdp-9.0-7.tar
   ```

   The bundle unpacks to a directory named “hadoop”, so the above commands create `/opt/hadoop/`. The version portion of your bundle download filename may differ.

3. Make your Hadoop HDFS configuration files available under `/etc/hadoop/conf/`. You must include at least your `log4j.properties` configuration file in this location.

4. Ensure the libraries and config files are readable by MarkLogic.

For more information on Hadoop and HDFS, see the Apache Hadoop documentation.

6.4.2 MapR-FS Storage

MarkLogic supports MapR-FS through NFS. For details on how to mount MapR-FS as NFS, see:

- http://maprdocs.mapr.com/51/DevelopmentGuide/c-mounting-the-cluster.html
- http://maprdocs.mapr.com/51/AdministratorGuide/c_set_up_mapr_nfs.html

6.4.3 S3 Storage

S3 is a cloud-based storage solution from Amazon. S3 is like a filesystem, but you access it via HTTP. MarkLogic Server uses HTTP to access S3, and you can put an S3 path into any of the data directory specifications on a forest, and MarkLogic will then write to S3 for that directory. For more details about Amazon S3, see the Amazon web site http://aws.amazon.com/s3/. This section describes S3 usage in MarkLogic and includes the following parts:

- S3 and MarkLogic
- Entering Your S3 Credentials for a MarkLogic Cluster

6.4.3.1 S3 and MarkLogic

Storage on S3 has an “eventual consistency” property, meaning that write operations might not be available immediately for reading, but they will be available at some point. Because of this, S3 data directories in MarkLogic have a restriction that MarkLogic does not create Journals on S3. Therefore, MarkLogic recommends that you use S3 only for backups and for read-only forests, otherwise you risk the possibility of data loss. If your forests are read-only, then there is no need to have journals.

When you set up an S3 path as a forest directory, the path must be readable and writable by the user in which the MarkLogic Server process is running. Typically, this means you must set Upload/Delete, View Permissions, and Edit Permissions on the AWS S3 bucket. This is true for both forest paths and for backup paths.

Because S3 is a very large shared filesystem, it can be good not only for forest data, but as a destination for database backups.

To specify an S3 path in MarkLogic, use a URL of the following form:

```
s3://<bucket-name>/<path-to-location>
```

so the following path would be to an S3 filesystem with a bucket named `my-bucket` and a path named `my-directory`:

```
s3://my-bucket/my-directory
```

Note: Amazon has other ways to set up S3 URLs, but use the form above to specify the S3 paths in MarkLogic; for more information on S3, see the Amazon documentation.

6.4.3.2 Entering Your S3 Credentials for a MarkLogic Cluster

S3 requires authentication with the following S3 creadentials:

- AWS Access Key
- AWS Secret Key

The S3 creadentials for a MarkLogic cluster are stored in the security database for the cluster. You can only have one set of S3 credentials per cluster. You can set up security access in S3, you can access any paths that are allowed access by those credentials. Because of the flexibility of how you can set up access in S3, you can set up any S3 account to allow access to any other account, so if you want to allow the credentials you have set up in MarkLogic to access S3 paths owned by other S3 users, those users need to grant access to those paths to the AWS Access Key set up in your MarkLogic Cluster.

To set up the AW credentials for a cluster, enter the keys in the Admin Interface under Security > Credentials. You can also set up the keys programmatically using the following Security API functions:

- `sec:credentials-get-aws`
- `sec:credentials-set-aws`

The credentials are stored in the Security database. Therefore, you cannot use S3 as the forest storage for a security database.

6.5 Windows Shared Disk Registry Settings and Permissions

If you are using remote machine file paths on Windows (for example, a path like `\\machine-name\dir`, where `machine-name` is the name of the host and `dir` is the path it exposes as a share), you must set the following registry settings to ZERO, as shown in https://technet.microsoft.com/en-us/library/ff686200.aspx:

- FileInfoCacheLifetime
- FileNotFoundCacheLifetime
- DirectoryCacheLifetime

These DWORD registry keys settings are in the following registry:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Lanmanworkstation
\Parameters
```

Additionally, the directory path must have read and write permissions for the SYSTEM user, or whichever user under which `MarkLogic.exe` runs.

7.0 Monitoring MarkLogic Server Performance

This chapter provides an overview to various ways to monitor the status in MarkLogic Server, both through the Admin Interface and through Server Monitoring APIs designed to report status of various parts of the system. This chapter includes the following sections:

- Ways to Monitor Performance and Activity
- Server Monitoring APIs

7.1 Ways to Monitor Performance and Activity

This section describes the following ways to monitor various activity on MarkLogic Server:

- Monitoring History Dashboard
- Server Logs
- Status Screens in the Admin Interface
- Create Your Own Server Reports

7.1.1 Monitoring History Dashboard

The Monitoring History dashboard is used to capture and make use of historical performance data for a MarkLogic cluster. Once the performance data has been collected, you can view the data in the Monitoring History page. The top-level Monitoring History page provides an overview of the performance metrics for all of the key resources in your cluster. For each resource, you can drill down for more detail. You can also adjust the time span of the viewed data and apply filters to view the data for select resources to compare and spot exceptions.

For details on the Monitoring History dashboard, see the MarkLogic Server Monitoring History chapter in the *Monitoring MarkLogic Guide*.

7.1.2 Server Logs

The logs for MarkLogic Server are an important source of information about activity on the server, particularly information about error conditions. The logs are all stored in the `Logs` directory under the MarkLogic Server data directory (typically `c:\Program Files\MarkLogic\Data\Logs` in windows, `/var/opt/MarkLogic/Logs` under UNIX-based systems). There are two types of logs:

- `ErrorLog.txt`, which logs MarkLogic Server exceptions, startup activity, and so on.
- *`port_no`*`_AccessLog.txt`, which logs access requests (for example, HTTP requests) for the App Server running at the specified port.

You can configure how long to keep a log before starting a new one, at which level to log activity, and how many old log files to keep before deleting (they have a number appended to their name, for example, `ErrorLog_5.txt` indicates 5 new log files have been created since this one was used). For more details on configuring the log files, see the *Administrator's Guide*.

Another option you can configure, at the App Server level, is for any uncaught application exceptions to be written to the `ErrorLog.txt` file. This way, if your application throws an exception (for example, if it has a syntax error), the error message is logged in addition to being sent to the client. This is useful in debugging, especially if queries are being generated via user activity on a browser or through a WebDAV client.

You can also code your own log messages into an application using `xdmp:log`. You can use `xdmp:log` to log any message at any level, and that message is written to the `ErrorLog.txt` file when it is called. Log messages are useful in debugging during development, and are also useful in logging certain activities in production.

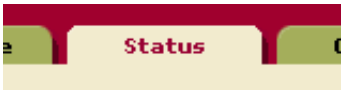
For operational purposes, some developers write scripts or programs to monitor the logs for specific messages. Then, if the specific message is logged, the script or program can send some sort of alert out (for example, page someone or send a message to someone).

The logs contain important information that can be used in monitoring MarkLogic Server. The logs can be a powerful tool in an overall monitoring policy. How you use that information depends on your requirements.

> **Note:** There must be sufficient disk space on the filesystem in which the log files reside. If there is no space left on the log file device, MarkLogic Server will abort. Additionally, if there is no disk space available for the log files, MarkLogic Server will fail to start.

7.1.3 Status Screens in the Admin Interface

The Admin Interface includes many pages that list current activity and status for various parts of MarkLogic Server. To access the status pages, click the status tab in the part of the Admin Interface to which you want to find current system information.



The following table lists the status pages available in the Admin Interface, along with the location path to the status tab in the Admin Interface and a description of each page.

Name	Location	Description
System Status	Configure > Status	Provides a summary of all information throughout the entire MarkLogic Server cluster, including host, App Server, database, and forest information. Also includes buttons to restart or shutdown all of the hosts in the cluster.
Group Status	Groups > *group_name* > Status	Provides a summary of all information throughout the MarkLogic Server group, including host, App Server, database, and forest information. Also includes buttons to restart or shutdown all of the hosts in the group.
Host Status	Hosts > *host_name* > Status	Provides a summary of the current conditions on the host, including information about App Servers, forests, and active queries. Also includes buttons to enter a new license key, restart, or shutdown the host.
App Server Status	Groups > *group_name* > App Servers > *app_server_name* > Status	Shows information about activity on the App Server, including queries active on each host and the ability to cancel the queries.

Name	Location	Description
Task Server Status	Groups > *group_name* > Task Server > Status	Shows information about activity on the task servers for the group, including the number of requests being processed and the number of tasks queued on the task server.
Database Status	Databases > *db_name* > Status	Shows activity on the specified database. Shows if any merges are in progress and if reindexing is in progress, and gives estimates about how long they will take. Shows information about the database, including the number of documents, number of fragments, and its size. Optionally shows forest status information.
Forest Status	Forests > *forest_name* > Status	Shows information and activity about the specified forest, including merge activity, number of stands, size, space available, and so on. Also includes a button to restart the forest, which forces the forest to go offline. Once it is offline, it will automatically attempts to rejoin the host, resulting in a restart of the forest. If failover is enabled, a restarted forest will first attempt to join the primary host.

7.1.4 Create Your Own Server Reports

The status pages in the Admin Interface provide a lot of detail about many parts of the system. If you want information about MarkLogic Server status in a different form, however, or if you want to combine it with some other application-specific information, you can build your own server reports using the Server Monitoring APIs. For more details, see the next section.

7.2 Server Monitoring APIs

The Server Monitoring APIs are XQuery functions that return XML representations of current activity of various parts of MarkLogic Server. Because they are XQuery functions, you can build any application you deem necessary with them. For example, perhaps you want a summary page that shows some different information then the system status page in the Admin Interface, or perhaps you want to combine some of that information with some content from your application. Perhaps you want to integrate it with a site-wide monitoring system. Whatever your requirements, because the APIs return XML, it is easy to write an application to display the information in whatever way fits your needs.

The following are the monitoring functions available:

- `xdmp:forest-counts`
- `xdmp:forest-status`
- `xdmp:host-status`
- `xdmp:request-cancel`
- `xdmp:request-status`
- `xdmp:server-status`

The `xdmp:forest-counts` function can take some time to compute on large systems, as it must query the forest to determine some of the counts. You can limit the work it performs with the optional second argument. For more details about syntax and usage of these functions, see the Server Monitoring functions in the *MarkLogic XQuery and XSLT Function Reference*.

8.0 Endpoints and Request Monitoring

The Request Monitoring feature enables you to configure logging of information related to requests, including metrics collected during request execution. This feature lets you enable logging of internal preset metrics for requests on specific endpoints. You can also log custom request data by calling the provided Request Logging APIs. This logged information may help you evaluate server performance.

This chapter provides an overview of creating endpoint declarations and using them to monitor requests on MarkLogic Server. This chapter includes the following sections:

- Monitoring Requests
- App Server Request Monitoring
- XDBC Server Request Monitoring
- Task Server Monitoring
- Creating Endpoint Declarations
- Request Cancelling
- Request Monitoring APIs

8.1 Monitoring Requests

The Request Monitoring feature enables you to configure logging of information request-related information, including metrics collected during request execution. You can switch on logging of internal preset metrics for requests on specific endpoints, or you can choose to log additional custom request data by calling the request logging APIs. The custom request data might contain a query plan, traces, or whatever information you want to collect and log for a request. This logged information may help you spot offending requests and evaluate request history.

8.2 App Server Request Monitoring

You can trigger request logging for an App Server through one or more of the following options:

- For targeted endpoints (main modules), by switching on monitoring in their endpoint declaration.
- For all requests on the App Server, by using a special server declaration.
- By calling request logging APIs in modules.

To switch on monitoring for an endpoint, you must add a `monitoring` section to the App Server Endpoint Declaration file and specify which metrics will be logged. Request monitoring is switched off by default for all metrics.

8.3 XDBC Server Request Monitoring

XDBC Server enables XCC and XDBC applications to communicate with MarkLogic Server. You can configure request monitoring for XDBC requests for specific endpoints and globally for the XDBC Server. There are two types of XDBC requests where request monitoring are enabled:

- XDBC Invoke Requests
- XDBC Eval Requests

8.3.1 XDBC Invoke Requests

You can enable request monitoring at both the endpoint level and at the server level for XDBC invoke request. For a specific endpoint, you must add a `monitoring` section to the XDBC Server Endpoint Declaration file and specify which metrics will be logged. To configure monitoring on a global level, you must add a `default.api` file in the modules root directory for the XDBC Server. For the Task Server, as there is no port number, the output request log file will be logged into a new type of log file called `TaskServer_RequestLog.txt`

8.3.2 XDBC Eval Requests

For XDBC eval requests, request monitoring is only available at the server level, as there are no real endpoints. To seperate the monitoring configuration between the invoke and eval request, and to add more control over the monitoring of the eval request, you can add an `eval.api` file to the module root in addition to the `default.api` file. The `eval.api` file has the same format as the `default.api` file, which contains only a monitoring section, but the settings in `eval.api` override those in `default.api`.

8.4 Task Server Monitoring

The Task Server processes request that has been spawned, such as from `xdmp:spawn()` or from a post-commit trigger action. Since each task is handled as a module, you can configure request monitoring for both endpoint level and server level. To configure request monitoring for a specific endpoint, add a `monitoring` section to the Task Server Endpoint Declaration file (`*.api`) and specify which metrics are to be logged. For monitoring on a global level, add a `monitoring` section to the `default.api` file in the modules root directory for the Task Server. The configuration in a specific endpoint declaration file overrides the settings in the global `default.api`.

8.5 Creating Endpoint Declarations

An Endpoint Declaration is a JSON file with the extension `.api` that resides in the **module** database or file directory of an HTTP server. The App Server uses the declarations in this file to dispatch requests to corresponding main modules. The declarations in this file also determine which requests are to be logged.

8.5.1 The Endpoint Declaration File

The name, parameters, and return value for each endpoint are declared in the `*.api` file. The `*.api` file contains a JSON data structure with the following properties:

Property	Declares
`functionName`	The name used to call the endpoint, which must match the name (without the `.api` extension) of the declaration file.
`desc`	Optional; plain text documentation for the endpoint.
`params`	Optional; an array specifying the parameters of the endpoint. This is omitted for endpoints with no parameters. Parameter objects have `name`, `desc`, `datatype`, `nullable`, and `multiple` properties.
`return`	Optional; an object specifying the endpoint return value. This is omitted for endpoints with no return value. The child object has `desc`, `datatype`, `nullable`, and `multiple` properties.
`errorDetail`	Optional; specifies a value from the following enumeration to control whether error responses include stack traces: • `log` (the default): log the stack trace on the server but do not return the stack trace to the middle tier. • `return`: include the stack trace in the exception on the middle tier as well as log it on the server.

Note: When monitoring a module that is not defined as an endpoint, none of the properties defined in the preceding table are needed

The following is a list of meters that can be logged with the parameters that control them:

Monitoring Flag	Data Type	Default Value	Parameters
`general`	`object`		Enables all the general (non-custom) meters. The list of parameters on which you can set constraints is in the next table.
`enabled`	`boolean`	`false`	Controls the logging of meters.
`custom`	`boolean`	`true`	Custom meters manipulated with the `xdmp:request-log-*` APIs.

Monitoring Flag	Data Type	Default Value	Parameters
`fragments`	`integer`	`0`	The maximum number of items to log. For each fragment: `root`, `expandedTreeCacheHits`, `expandedTreeCacheMisses`
`documents`	`integer`	`0`	The maximum number of items to log. For each document: `uri`, `expandedTreeCacheHits`, `expandedTreeCacheMisses`
`hosts`	`integer`	`0`	The maximum number of items to log. For each host: `host`, `roundTripTime`, `roundTripCount`

The following parameters may be included in a `*.api` file:

Parameter	Description
`commitTime`	The aggregate commit phase time, represented as a double-precision value in seconds.
`compileTime`	The aggregate time spent compiling a module or a program, represented as a double-precision value in seconds.
`compressedTreeSize`	The aggregate size in bytes read from disk by unsuccessful compressed tree cache lookups. Each unsuccessful compressed tree cache lookup is followed by a disk access to load the compressed tree into the cache.
`compressedTreeCacheHits`	The number of successful compressed tree cache lookups. The compressed tree cache holds XML document data in the compressed representation stored on disk.
`compressedTreeCacheMisses`	The number of unsuccessful compressed tree cache lookups. Each unsuccessful compressed tree cache lookup was followed by a disk access to load the compressed tree into the cache.
`contemporaneousTimestampTime`	The time spent by queries waiting for the contemporaneous timestamp for which any transaction is known to have committed, represented as a double-precision value in seconds. When the multi-version concurrency control is set contemporaneous, queries can block waiting for the contemporaneous transactions to fully commit.

Parameter	Description
dbLibraryModuleCacheHits	The number of library module cache hits from library modules from the modules database.
dbLibraryModuleCacheMisses	The number of library module cache misses from library modules from the modules database.
dbMainModuleSequenceCacheHits	The number of main module cache hits from main modules in a database.
dbMainModuleSequenceCacheMisses	The number of main module cache misses from main modules in a database.
dbProgramCacheHits	The number of module cache hits from the entire program made from modules in a database (may contain library modules from the special Modules directory).
dbProgramCacheMisses	The number of module cache misses from the entire program made from modules in a database (may contain library modules from the special Modules directory).
elapsedTime	The time elapsed since the start of the processing of this query, in the form of a duration. Use this parameter instead of the deprecated `xdmp:set-request-time-limit` function.
envProgramCacheHits	The number of module cache hits from the entire program made from ad hoc XSLT stylesheet nodes.
envProgramCacheMisses	The number of module cache misses from the entire program made from ad hoc XSLT stylesheet nodes.
expandedTreeCacheHits	The number of successful expanded tree cache lookups. The expanded tree cache cache holds XML document data in the expanded representation used by the XQuery evaluator.
expandedTreeCacheMisses	The number of unsuccessful expanded tree cache lookups. Each unsuccessful expanded tree lookup was followed by a compressed tree cache lookup to load the expanded tree into the cache.
filterHits	The number of successful search filter matches.
filterMisses	The number of unsuccessful search filter matches.
fragmentsAdded	The number of XML fragments added to the database by an update.

Parameter	Description
fragmentsDeleted	The number of XML fragments deleted from the database by an update.
fsLibraryModuleCacheHits	The number of library module cache hits from library modules on the file system.
fsLibraryModuleCacheMisses	The number of library module cache misses from library modules on the file system.
fsMainModuleSequenceCacheHits	The number of main module cache hits from main modules on the file system.
fsMainModuleSequenceCacheMisses	The number of main module cache misses from main modules on the file system.
fsProgramCacheHits	The number of module cache hits from the entire program made from modules on the file system.
fsProgramCacheMisses	The number of module cache misses from the entire program made from modules on the file system.
inMemoryCompressedTreeHits	The number of successful compressed tree lookups in in-memory stands.
inMemoryListHits	The number of successful list lookups in in-memory stands.
indexingTime	The indexing time of documents before they are inserted into the database, represented as a double-precision value in seconds.
linkCacheHits	The number of successful link cache lookups. The link cache is a transient cache that exists only for the duration of one query. It holds pointers to expanded trees, and is used to accelerate the frequent dereferencing of link nodes.
linkCacheMisses	The number of unsuccessful link cache lookups. Each unsuccessful link cache lookup was followed by a search for the link target tree.
listSize	The aggregate size in bytes read from disk by unsuccessful list cache lookups. Each unsuccessful list cache lookup is followed by a disk access to load the search term list into the cache.

Parameter	Description
listCacheHits	The number of successful list cache lookups. The list cache holds search termlists used to accelerate path expressions and text searches.
listCacheMisses	The number of unsuccessful list cache lookups. Each unsuccessful list cache lookup was followed by a disk access to load the search termlist into the cache.
lockTime	The aggregate time forests spend waiting for transactional read and write locks, represented as a double-precision value in seconds. This time can exceed the run-time.
readLocks	The number of read locks.
regexpCacheHits	The number of successful regular expression cache lookups. The regular expression cache is a transient cache that exists only for the duration of one query. It holds compiled regular expressions, and is used to accelerate the frequent use of regular expressions during the evaluation of a query.
regexpCacheMisses	The number of unsuccessful regular expression cache lookups. Each unsuccessful regular expression cache lookup was followed by a compilation of a regular expression from source text.
requests	The number of requests.
runTime	The aggregate time spent evaluating or running a module or a program, represented as a double-precision value in seconds.
valueCacheHits	The number of successful value cache lookups. The value cache is a transient cache that exists only for the duration of one query. It holds typed values, and is used to accelerate the frequent conversion of nodes to typed values.
valueCacheMisses	The number of unsuccessful value cache lookups. Each unsuccessful value cache lookup was followed by a conversion of an XML node to a typed value.
writeLocks	The number of write locks.

8.5.2 Constraints on Meters

To control the number of meters that are logged, you can put the following constraints on meters:

Operator	Description
`lt`	Less than
`gt`	Greater than
`le`	Less than or equal to
`ge`	Greater than or equal to

The declaration format of a constraint is:

```
meter_name : {"operator":value, "operator":value, ...}
```

For example:

```
"constraints": {
  "tripleCacheHits" : { "ge":1 }
}
```

In this example, `tripleCacheHits` is logged only if the the value of `tripleCacheHits` is >= 1.

Meters with zero or empty values are not normally logged. This is done to minimize the size of the Request Log file. To log a zero or empty value, use the following code:

```
"constraints": {
  "meter_name" : { "ge":0 }
}
```

The default constraint value on any meter is:

```
"gt":0
```

8.5.3 Controlling Request Logging With Thresholds

You can add thresholds to specify that a request is logged only when the threshold conditions are satisfied. To declare a threshold, create a "thresholds" section in your `*.api` file as follows:

```
{
  "monitoring":{
    "thresholds" : {
      "meter_name" : { "operator":value, "operator":value, ...}
    }
...
  }
}
```

Note: The only threshold that is currently supported is `elapsedTime`.

The following operators are allowed in thresholds:

Operator	Description
`lt`	Less than
`gt`	Greater than
`le`	Less than or equal to
`ge`	Greater than or equal to

In the following example, the `general` meters for the request are logged only when the total runtime of the request is greater than one second:

```
{
  "monitoring":{
    "thresholds" : {
      "elapsedTime": { "gt": 1.0 }
    }
    "general" : {
      "enabled": true
    }
  }
}
```

8.5.4 Enabling Request Monitoring

Request monitoring is enabled by default on the following default MarkLogic application servers:

- The default App-Services application server, normally Port 8000:
  - For all requests related to the Query Console.
  - For requests with runtime exceeding one second that are not related to the Query Console. This mainly covers the REST Client API.
- The default Manage application server, normally Port 8002, for requests running longer than one second. This covers the Configuration Manager and Monitoring Dashboard.

If request monitoring is not already enabled, you can enable request monitoring by calling any server-side JavaScript (`*.sjs`) or XQuery (`*.xqy`) functions in files that reside in the **modules** directory as declared on the HTTP server via the Admin interface, and you can create other `*.api` files in the same directory or, if using the file system, in the same subdirectory as the JavaScript or XQuery file being called. For more information on configuring HTTP servers and the **modules** directory, see the "HTTP Servers" chapter in the *Administrator's Guide*.

The following example enables request monitoring:

```
{
"monitoring":{
  "general":{
    "enabled":true
  },
  "custom":true
}
}
```

The following example logs `tripleCacheHits` if the the value of `tripleCacheHits` is >= 0:

```
"constraints": {
     "tripleCacheHits" : { "ge":0 }
    }
```

The following code fragment logs up to 10 documents, 10 fragments, and 4 hosts:

```
"documents": 10,
"fragments" : 10,
"hosts" : 4
```

The following example is called `countdocs.api`. The function `getCount` is defined in a file called `countdocs.sjs` that resides in the same directory. The `general` object under the `monitoring` section of the file has `enable` and `custom` set to `true`; this enables request logging.

```
        {
          "functionName":"getCount",
          "params":[
            {
              "name":"collection",
              "datatype":"string",
              "multiple":false,
              "nullable":false
            },
            {
              "name":"method",
              "datatype":"int",
              "multiple":false,
              "nullable":false
            }

          ],
          "return":{
            "datatype":"string",
            "nullable":false,
            "multiple":false
          },

          "monitoring":{
            "general": {
              "enabled": true,
              "constraints": {
                "tripleCacheHits" : { "ge":1 }
              }
            },
            "custom": true ,
            "documents": 10,
            "fragments" : 0,
            "hosts" : 4
          }
        }
```

8.5.5 The Default Declaration File

You can configure request monitoring globally for all requests on an App Server by adding a `default.api` file where the **modules** root is configured for the App Server (either to the **modules** database or to the file system). The `default.api` file only contains a monitoring section.

The following is a sample `default.api` file:

```
{
"monitoring":{
    "general":  {
      "enabled": true,
      "constraints": {
        "tripleCacheHits" : { "ge":0 }
      }
    },
    "custom": true,
    "documents": 10,
    "fragments" : 10,
    "hosts" : 5  }
}
```

When you make a request to an endpoint and do not specify an App Server Endpoint Declaration file, the `default.api` file in the module database or file directory is used if it exists. If a module has a `*.api` file associated with it, the monitoring settings in the `*.api` file for that module are used instead of those in the `default.api` file.

If you want to save the `default.`api file, use JavaScript:

```
declareUpdate();
xdmp.documentInsert("/default.api",{
 "monitoring":{
  "general": {
   "enabled": true
   },
  "custom": true,
  "documents": 10,
  "fragments" : 10,
  "hosts" : 5 }
});
```

8.5.6 Request Logs

The logs for MarkLogic Server containing information about the requests you have chosen to log are stored in the `Logs` directory under the MarkLogic Server data directory (typically `c:\Program Files\MarkLogic\Data\Logs` on Windows, `/var/opt/MarkLogic/Logs` on UNIX-based systems) in the file *port_no*`_RequestLog.txt`. Each RequestLog.txt file will contain the meters from multiple monitored endpoints that:

- are configured on an App Server with some monitoring switched on.
- have calls to `xdmp:request-log-put` in their module.

The collected data is logged in JSON format at the rate of one line per request information. Even if monitoring is completely switched off on an endpoint, calls to `xdmp:request-log-put(`*key*`,`*value*`)` during a request will result in data being logged for all the `(`*key*`,`*value*`)` pairs that have been stored during the request.

The following is an example of a request log where different meters are logged for two endpoints:

```
{"time":"2018-11-09T14:07:14-08:00",
"url":"/booleanApiDecl.sjs?booleanArg=true", "user":"admin",
"result":10, "elapsedTime":0.007145, "requests":1,
"dbProgramCacheMisses":1, "dbMainModuleSequenceCacheMisses":1,
"dbLibraryModuleCacheMisses":0}

{"time":"2018-11-09T14:10:18-08:00",
"url":"/booleanApiDecl.xqy?booleanArg=true", "user":"admin",
"result":10, "elapsedTime":0.001603, "requests":1,
"dbProgramCacheMisses":1, "dbMainModuleSequenceCacheMisses":1,
"dbLibraryModuleCacheMisses":0}
```

8.6 Request Cancelling

This section describes the procedure to setup and enable request cancelling for an endpoint, a main module, or globally on an App Server or an XDBC Server. Request cancelling is disabled by default for all meters. You can enable request cancelling by adding a `limits` section to the monitoring section in the endpoint declaration, as in the following example:

```
{
  "monitoring": {
    "limits" : {
      "lockCount" : 100,
      "readSize" : 1000000
    }
  }
}
```

The following limits are available:

Meter	Unit	Description
`elapsedTime`	seconds	Equivalent to calling `xdmp:set-request-time-limit()`
`readSize`	bytes	Combined size read from disk (`listSize` + `compressedTreeSize`)
`lockCount`	count	Combined count for the number of times a read or a write lock was acquired (`readLocks` + `writeLocks`)

You can update the `limits` configuration while the server is running without having to restart the server.

8.7 Request Monitoring APIs

The Request Monitoring APIs are XQuery and JavaScript functions enable you to log additional information in request logs. The following are the request monitoring functions available:

- `xdmp:request-log-put`
- `xdmp:request-log-get`
- `xdmp:request-log-delete`
- `xdmp:set-request-limit`
- `xdmp.requestLogPut`
- `xdmp.requestLogGet`
- `xdmp.requestLogDelete`
- `xdmp:request-status`

Whatever is logged with `xdmp:request-log-put` displays in the log files unless the `custom` flag is set to `false`. When the `custom` flag is set to `false`, all custom logging is muted.

For more details about syntax and usage of these functions, see the AppServer functions in the *MarkLogic XQuery and XSLT Function Reference* and the *MarkLogic JavaScript Reference Guide*.

9.0 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.

10.0 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.