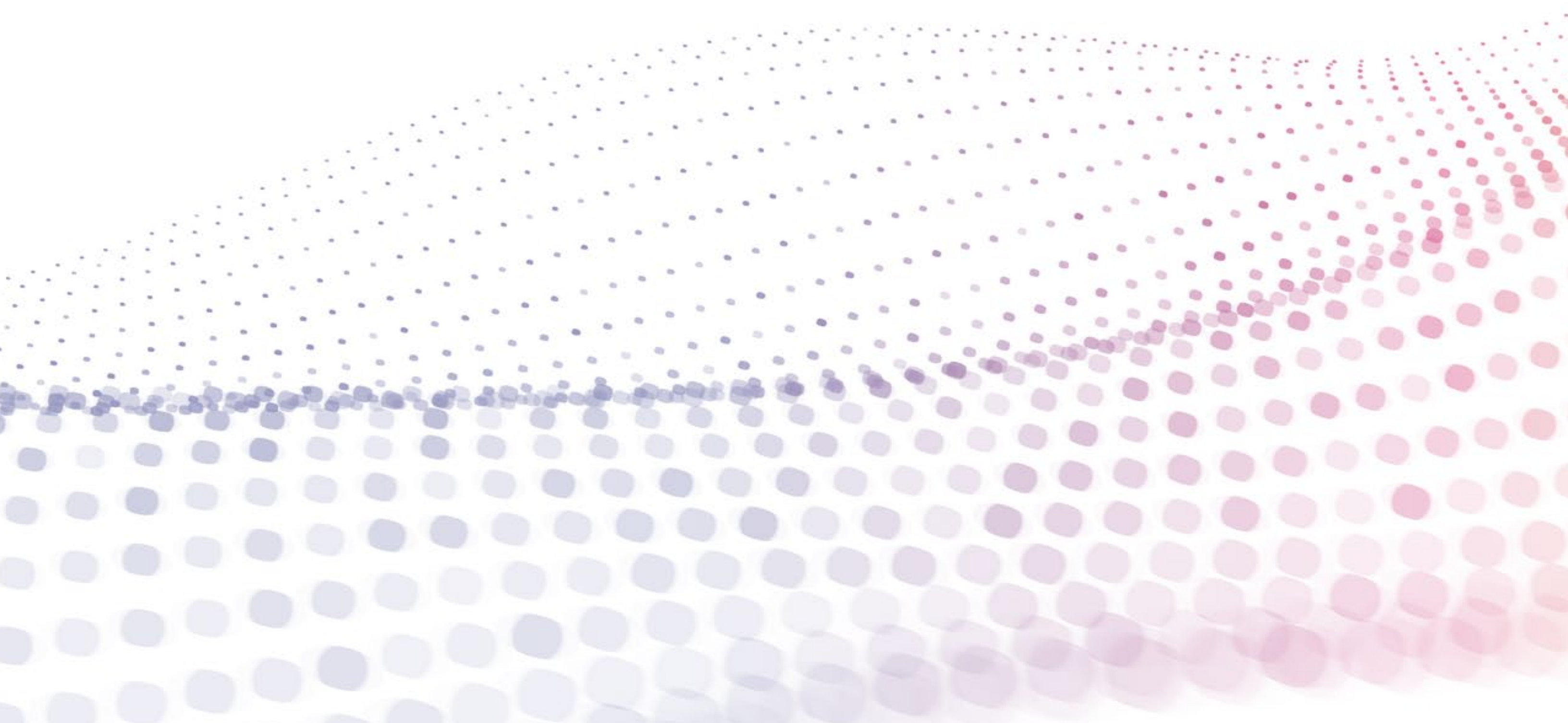
MarkLogic®

Performance Testing With MarkLogic

MARKLOGIC WHITE PAPER · OCTOBER 2018

Database performance testing is critical to deploying and running high performance enterprise applications. This technical white paper provides guidelines for performance testing with MarkLogic that are based on industry best practices and over a decade of experience with large-scale systems. It covers how to write scalable requests, design and execute performance tests, and understand the results.



About the Author

Erin Miller

Sr. Manager, Performance Engineering

Erin Miller is a lead engineer for the MarkLogic Performance Engineering team. In her tenure at MarkLogic, she has worked with many of MarkLogic's largest and most complex deployments. Erin is an expert in performance and application tuning and served as a key contributor at enterprise software companies for over 20 years. Prior to MarkLogic, Erin specialized in enterprise search and has deep expertise in Java and J2EE technologies. Erin holds a Bachelor's Degree from Harvard College.

Contents

Introduction

Will your code hold up at scale? How should you monitor your queries? Where are your bottlenecks? Managing performance for massively scalable systems is daunting. But, with careful planning, testing, and tweaking, your system will perform gracefully as capacity needs change. Optimizing the performance of MarkLogic is a big priority for us, and we strive to provide both a product and the guidance necessary that will lead to success.

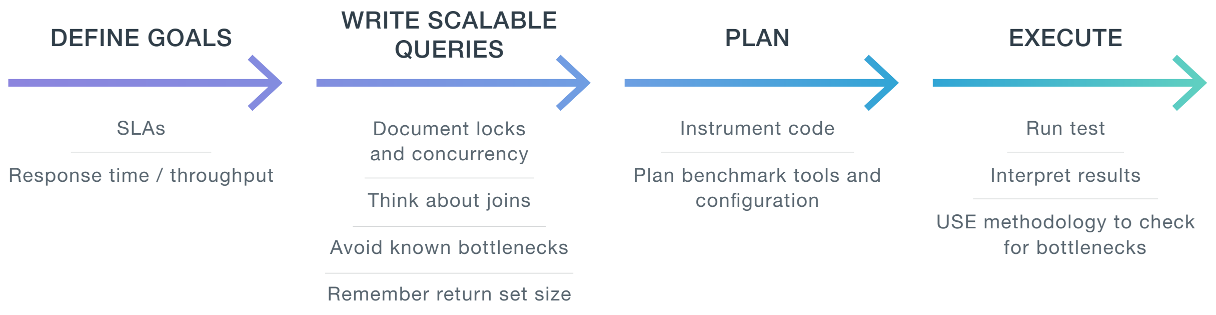
MarkLogic is an *operational and transactional* Enterprise NoSQL database platform used by organizations around the world to integrate critical data and build innovative applications with a 360-degree view. Many organizations use MarkLogic as a centralized Operational Data Hub to handle massive amounts of data. With such a database, it is no surprise that performance is critical to get right.

Demand on MarkLogic may increase because more business units want to load data into MarkLogic. Or, a primary application might have new data streams that need to be added. Or, a different business unit wants to deploy a new application that accesses data in MarkLogic. In any case, it is critical to meet performance SLAs as the system grows.

The goal of this technical white paper is to provide a better understanding of the best practices for load testing applications and modeling future capacity. It is written for application developers, systems engineers, and solution architects who want to better understand performance testing with MarkLogic, and are doing the actual work to keep their MarkLogic system humming.

The Lifecycle of a MarkLogic Performance Test

The diagram below explains the steps to go through when planning a MarkLogic performance test.



In this white paper, we walk through each of these four stages: defining goals, writing scalable queries, planning your topology and benchmark, executing your test, and diagnosing the results. By the end of this process, you will have clear methodology to apply to any MarkLogic project, regardless of use case.

Defining Goals

Are you load testing an application before it goes to production to validate that it will handle a live workload? Or, are you trying to evaluate how many more documents can be added to the database before it needs another host?

In either case, the fundamental metrics are the same: *response time* and *throughput*.

Your organization may have other critical metrics that vary in complexity, including SLAs (Service Level Agreements) and KPIs (Key Performance Indicators). But, for the purposes of this white paper, we assume that the two most important metrics that guide the structure of your load test are *response time* and *throughput*.

Key Metrics

Response time is the time it takes for a query to execute. The complexity of queries can be highly variable, which is why many organizations define acceptable percentiles to measure response times. An example of a response time requirement is, “95% of queries must be less than one second (or, sub-second).”

Throughput is how many queries can execute concurrently in a given timeframe. This requirement usually comes from business SLAs. An example might be: “The application must support 100 concurrent queries per second.”

Targets for response time and throughput should be defined in terms of statistical distribution. So, putting both together, the example requirement is: “At 100 concurrent requests, 95% of queries must be less than one second.” (Note that this is just an example, not an approximation of MarkLogic performance.)

It is important to understand the requirements ahead of time so you can reason about inflection points and actions to take as you reach limits. Once you define your requirements around response time and throughput, you can then define how to measure these metrics.

Measurement & Instrumentation

Many load or performance testing tools like JMeter or LoadRunner have built-in components to measure throughput and response time. Many organizations using MarkLogic rely on LoadRunner to test the full application stack end-to-end, though there are many other performance testing tools also available.

It is important to choose a tool that measures, at a minimum, response time and throughput. It is also important to implement your code so that you understand MarkLogic’s contributions to overall response times, as well as the metrics around MarkLogic component usage, such as caches.

xdmp:trace()

Although `xdmp:log()` can be used to log to file, you should consider using trace events instead. That way, you can decorate your code with trace events as needed, and then enable and disable the event on the Group level without changing your code.

xdmp:elapsed-time()

This is a lightweight API call that provides the elapsed time of each query. It is useful to tie a query to an ID, either a session ID or some other way to identify the query. It is also helpful to log the query itself along with the elapsed time:

```
xdmp:trace("query-meters-example", fn:concat("Query id: ", $id, "Query: ",
$query, "Response time: ", xdmp:elapsed-time))
```

xdmp:query-meters()

This is a much more detailed API call that provides a great deal of information about your query performance. The full API documentation can be found in the product documentation.

At times, MarkLogic's Support may ask you to decorate your code with query-meters calls in order to better understand resource utilization. Some key metrics that may be important include: elapsed-time, request, cache-hits, cache-misses, and round-trip-times. The cache hits and misses may be important depending on your application requirements. In very general terms, you want more hits than misses. Hits mean that you are reading data from memory, and misses mean that you are reading data from disk.

Round-trip times can be particularly useful when you have a cluster. Each node (or d-node, if your system has separated e-nodes and d-nodes) will send back the round-trip time and counts. This enables you to see whether you have a poorly performing node, or a hotspot where you have more data on one node than expected. Starting with MarkLogic 9.0-7, it is possible to use the **Request Monitoring** feature to decorate your code with query metrics. Prior to 9.0-7, you can use `xdmp:query-meters()` to gather lots of information about each query.

Analyzing Logged Results

Using query meters to log metrics about each individual query will result in a large amount of log data. There are many other languages and tools that available to parse log data: Perl, Python, and R are common, as well as PowerShell or bash scripting. A good first step for using query meters results is to look for queries that exceed your SLA, and then dig into the metrics for that individual query. We discuss analyzing metrics more in the "Understanding Results" section of this paper.

Writing Scalable Requests

When writing your application, you should make sure your code issues scalable requests. This may involve some tuning before you begin performance testing. If not, performance testing will reveal application issues. Either is preferable to production traffic revealing issues. In this section, we cover best practices for writing scalable requests.

Keep Requests Independent

Make sure your requests are independent of each other so that requests are as isolated as possible. This helps you isolate resource use of other concurrently running requests.

Watch Your Locks

Where possible, we recommend not requiring more than one request to update the same document and to lock ONLY the documents you update. Read/write locks can be a tricky area that developers can stumble over. There has been more than one developer to exclaim, "I thought I only locked the document I'm updating—not all the other documents in my query module!" Read this blog post to better understand query isolation and locking.

Avoid Unnecessary Bottlenecks

There are some features that seem like a good idea at the time, but end up introducing a bottleneck. The classic example of this is the sequential ID requirement: "I want each document in my database to have a sequentially incrementing ID." That is okay. But, how are you going to keep track of the latest ID… server field? That gets really big and starts using lots of memory. You might then say, "How about I put the ID in a document and look up the current ID at runtime?" See the "Watch your Locks" section above—you have just implemented a massively deadlocking application. Instead, consider how you might meet the requirement (unique ID) without introducing a bottleneck. (Hint: Use a UUID or GUID from an upstream system that is built for efficiency, or `xdmp:random()` or `xdmp:hash64($something as xs:string))`.

Limit the Request's Resource Utilizations

A well-known anti-pattern is to write code that scales well in development and falls apart in production. To avoid this pitfall, make sure that your requests scale in both environments.

- Do not request a result set that grows with the size of your data (e.g., every document in the database)
- Put a limit on the number of results at some reasonable maximum
- Paginate results. This way, you do not put a limit on the total number, but you do put a limit on the number of results per request
- When doing joins or using SPARQL or SQL, limit the scope of the request, if possible, using `cts` or Optic API search clauses

Use Optimistic Concurrency for REST Requests

For a detailed description of optimistic concurrency (also known as optimistic locking), take a look at the REST Developers Guide.

Use Multi-Statement Transactions Wisely

If you need multi-statement transactions, by all means, use them! But, multi-statement transactions may have unanticipated locking side effects (see "Watch your Locks" above). So, only use multi-statement transactions (and XA transactions) if you really need them—using them for convenience is not a good reason.

Planning Your Test Topography

The best practice for performance testing with MarkLogic is to run your performance test on the same hardware and with the same configuration and same data as your production system. If you can exactly replicate your production hardware and configuration in a test environment, you get the best coverage and the best confidence you are running the right tests. In the cases where it is not possible to match testing and production environments, we provide some additional considerations below.

Production Host Configuration

Make sure that your performance environment has the same host configuration as your production cluster. This means using the same number of forests per host, the same setup for replication, the same setup for high availability (e.g., If you are using local-disk forest failover in production, do not skip it in performance testing), etc.

Production Test Data Set

It is likely that your application uses many different data sources. The best practice is to use a complete copy of data in your performance testing environment. If you choose to use a subset, make sure that use a representative subset. For example, imagine you have a database that includes trade transactions. You might have different categories of transactions to track (e.g. Equity trades and OTC derivatives trades). Make sure that your subset of data contains the same ratio of categories that your production data does. Why does this matter? Imagine you have a query that computes the value traded with a particular counterparty. OTC derivatives trade records are much more complex than listed equity trades, so if your performance test environment does not have a representative proportion of them, your query may run much faster in test than in production.

Forest Data Density

Data density, or how much data you have per host, is a critical part of your test infrastructure. Your hosts should replicate the production environment, not just in the number of forests per host (configuration), but also in the amount and size of data per forest. The best practice is to have the data in your test environment exactly mimic production. So, you should have the database configured the same way, with the same forests and data. A database backup serves you well here.

If you decide not to follow the best practice and to instead use a subset of data for performance testing, it is critically important to carefully choose your data. If you have data types that vary significantly in size (where one type of data might be significantly larger or smaller than others), then make sure to consider this as part of your data density strategy. Just as with the data distribution, you should make sure that your performance test nodes contain about the same amount of data and have forests that are about the same size as your production environment.

Finally, if you have a few documents that are outliers in terms of size, it is best to include them. It is not common for organizations to have a record that is orders of magnitude larger than any other record in the database. Including those records in performance test clusters helps ensure there will not be surprises when this record is involved in a query or update.

A Word on Data Security & Privacy

Many organizations using MarkLogic have data that is subject to strict security or privacy controls. If that is the case with your system, you should consider using MarkLogic's **Redaction** feature to anonymize data when moving it from your production to test environments.

Planning Your Benchmark

Now that we have laid the groundwork of best practices, it is time to plan the performance benchmark itself. But, there are a few tips to keep in mind here as well. Let us discuss them before going further.

Choose the Right Benchmarking Tool

There are many benchmarking tools available, both commercially and as open source. JMeter and LoadRunner, which we already mentioned, are popular load generation tools. At an absolute minimum, you should be able to measure response time and throughput, record the results, and ramp up your load over time.

Choose Representative Queries

When designing your benchmark, be sure to choose a representative query load. How do you define a representative query load? The best way to generate the workload is to pull logs from your current production system. That guarantees the representative query load replicates real-world traffic. However, if that is not possible (perhaps because you are not yet in production), then it is important to choose queries that are representative of the workloads you expect your users to run against MarkLogic, and to do sensitivity analysis with different workload mixes to understand the boundaries of performance for reasonable scenarios. See Sensitivity Parameters, below.

Balance Your Workload

Many customers have application endpoints to fulfill business requirements. For instance, an application that helps users pick insurance plans might have an endpoint that looks up a user’s state, an endpoint that returns all available plans in that state, an age eligibility endpoint, etc. To make sure that you are properly testing your application, you should exercise all these endpoints in the same ratio as expected production traffic. For instance, you should not test to hit the state lookup endpoint 97 times and the age eligibility endpoint 3 times. You should balance traffic so that it hits the endpoints in the same way that you would expect users to exercise your application.

Another aspect of balancing your workload is choosing the right mix of parameters to send to these endpoints. If you have endpoints that do significantly different work depending on how you call them, you want to approximate your production usage not only in which endpoints are called, but also in what parameters are passed to them. For example, one organization using MarkLogic has an endpoint that does an aggregate calculation over a set of records that conforms to one or more criteria. The amount of work this endpoint does depends on whether the criteria are tight (and therefore result in fewer records in scope) or loose (and therefore result in many more records in scope).

Isolate Your Environment (The parts that are not shared!)

Make sure you are running your test at a time when other applications or developers are not using the hardware and application. In the often-frantic dash to production deployment, sometimes signals get crossed and you discover that some users are hitting your performance cluster right as you are trying to run your test. However, this assumes that you plan to use isolated hardware. If you are using any shared components, such as a SAN for forest data, make sure you run the test under conditions replicating production load. For example, if you are on a shared SAN and you run your test off-hours, make sure that the SAN is loaded as it would be during business hours. Your test might run fine off-hours, but when you run in production, if the SAN is oversubscribed, you may end up with a performance problem. With shared infrastructure, make sure that when you run your tests, you replicate the expected production load.

Choose the Right Test Parameters

Decide how you want to run the test. Do you have a fixed set of queries you plan to run through? Or, do you want to add load over time? Do you want to start new threads every minute, or ramp up by a percentage as you go?

Match Sensitivity Parameters to Your Risk

You should vary your sensitivity parameters depending upon the risk in your system and/or workload characterization estimates. You are trying to find the edge of the envelope here. You have your workload, your code, your proposed deployment configuration. Now, you should push on the parts of your workload that you are worried about. Are you concerned about query response times during a resource-intensive ingest? If so, make sure you have that scenario covered.

For a given system with a standard workload, there will only be one bottleneck at a time. But, if your workload is variable there may be more than one bottleneck. Scenarios that have variable workloads include:

- Some batch jobs;
- Some interactive job, with different APIs doing different things, or;
- Using the same API, but once a month you request *everything* instead of a subset

It is important to understand what your workload looks like at the time you reach capacity. Probe the envelope of your workload to find any unexpected issues.

Executing the Test

There are many excellent industry articles and tools related to test methodology. MarkLogic’s Performance Engineering team often refers to Brendan Gregg’s work and his book Understanding System Resources.

Here is the basic methodology we propose for running your test:

1. Ramp up until you hit a bottleneck (even if it is beyond your SLA). You can tell when you hit a bottleneck when you see a "knee" in the performance curve that maps response time to request volume.

2. Diagnose the bottleneck

3. Decide what to do about the bottleneck
   - Increase a resource
   - Decrease the demand (perhaps indirectly via a code change)
   - Leave *as is* and scale the cluster instead

Depending upon the outcome of the above process, you have two directions—repeat or scale. You should repeat if you have remediated a bottleneck by adding resources (like memory) or decreasing demand either by ramping down test threads or changing your code to use fewer resources. So, you alleviated that bottleneck—it's time to re-run.

There is something to think about though. If you tune every single resource perfectly for that workload, the whole node will be tuned—*for that precise workload*. What happens when you add a new workload? You will not necessarily be able to predict the resource utilization and therefore will not be able to effectively monitor and scale the cluster as a whole.

In general, it is best to tune each node so that there is one bottleneck resource that you can monitor for all reasonable workloads. To do this, you may need to provision excess capacity in other resources on that node. Think of this excess capacity as the safety breakdown lane on a highway. With that excess capacity, what you are paying for is not performance but *stability* and *predictability*.

When should you scale out? This is a great choice if you have a bottleneck that is hard to scale up—like CPU. You may have fixed hardware (or instance/VM types, in the cloud) and you know that you need more CPU. You will have to scale the cluster out. To do this, add hosts one and a time and re-run the test as before.

As you ramp your test, eventually your response time will increase and throughput will decrease. Your response time graphs will look like a knee. Congratulations, you found a bottleneck! Now you can advance to the next step—diagnosing the bottleneck.

Understanding Test Results

In his book Systems Performance: Enterprise and the Cloud, Brendan Gregg outlines what he calls the "USE methodology", which is a great way to think about test results. The USE methodology is really an acronym. For every resource, check Utilization, Saturation, and Errors."

"Resources" includes computing or hardware resources like CPU, memory, and storage. It can also mean MarkLogic resources such as caches, app server latency, or other resources. So, this system provides a way to outline your resources and then for each resource, check the three parameters. Here are the resources that you should keep in mind as you use this methodology:

Physical Resources

- CPU
- RAM
- Disk I/O
- Storage capacity
- Network

MarkLogic Resources

- Caches – hits and misses
- Expanded Tree Cache (ETC)
- Compressed Tree Cache (CTC)
- List Cache
- Triple and Triple Value Caches
- Caches – Memory utilization (same cache list as above)
- Merge Disk Throughput
- Journal read/write rates
- Forest Memory
- Registered Query Memory
- Huge Pages Usage
- External KMS (if applicable)
- Locks (read/write/hold locks, deadlocks)
- Database Replication Rate
- D-node Roundtrips (if applicable)

These MarkLogic resources map to system resources. (For more information, see the MarkLogic technical white paper called Understanding System Resources.) These MarkLogic components may map to several resources. For instance, app server latency may be related to disk I/O, network I/O, and available RAM. So, you need to put the pieces together in order to create an understanding of where your bottleneck lies. If you work your way through this list of resources, you should get an accurate picture of where the bottleneck might be.

If you look at your Monitoring Dashboard (Monitoring History), you will find graphs for all of these MarkLogic-specific resources. Some of the details, like registered query memory, show up in MarkLogic 9.0-5 and later, but you can find output in the ErrorLog starting in MarkLogic 9.0-4 that provides information about some of these metrics. Here is an example of the memory logging that appears in the ErrorLog:

```
2018-05-30 01:12:00.081 Info: Memory 33% phys=96504 virt=75325(78%)
rss=688(0%) huge=31460(32%) anon=631(0%) file=162(0%) forest=1236(1%)
cache=30720(31%)
```

This log entry tells you the amount of physical, virtual and resident memory, as well as huge pages, anonymous huge pages, files, forest memory, and caches. It also provides percentages so that you can monitor your system for memory usage.

After you run your benchmark, you should use the Monitoring History, benchmark results, and MarkLogic logs to understand the performance characteristics of your testing.

If you have used this methodology and there are things that are difficult to understand or if you are having trouble identifying your bottleneck, email support@marklogic.com for more assistance understanding your benchmark results.

Diagnosing Common Performance Problems

In this section, we discuss some performance problems to consider as you run tests. Of course, there are other problems not included here, but these are some of the more common ones.

Check I/O Throughput Limitations

This can manifest as query response times increasing during ingest, and especially during merges. This can also manifest as “hung” messages in the logs, or as “slow background thread” messages. These symptoms can mean that you do not have enough I/O throughput to both do ingestion and handle queries. These clues should point you to system-level metrics to verify, where you can check disk utilization and diagnose I/O bottlenecks. You can remediate the problem by adding I/O capacity (better/faster disks) or by scaling out with more hosts (which includes more forests and thus more I/O capacity).

Check for Inefficient Code

This can manifest as high CPU-utilization, although your test may be designed to push the CPU as high as possible. Remember that even if you have high utilization, it does not mean there is a problem. It may also be that you need to write more efficient code in order to reduce the CPU utilization and better utilize the cores available.

Review the guidelines in Writing Scalable Requests for ideas. Or, even better, use the profiler built into MarkLogic’s Query Console to run a query at the same time that you are running your performance test. The profiler will show you the functions that are taking the most time and also the counts of the functions. This can be particularly helpful in finding code defects. For instance, you may realize that the code was not supposed to count an element 12 million times (e.g. counting the element across all the chapters in your entire corpus of XML documents instead of just counting the chapter elements in your book).

Check Memory Limits

If you have a substantial MarkLogic implementation, you may have many range indexes. All of these indexes are kept in memory—in addition to the list data, expanded and compressed trees, and triples and triple values. Assuming that you are configured with swap, per the Installation Guide, you should see your servers begin to swap as your system hits memory limits. This results in severely degraded performance. If all of a sudden you see the performance grind to a halt, and you know that you are close to your memory limits, make sure you check to see if there is swapping. If so, you will likely need to add memory or scale out your cluster to do less work on each node.

You may also run into swapping when you are doing inefficient joins, either with XQuery or SPARQL/SQL. In that case, you may want to tune your code and/or add memory.

As an ancillary topic, **make the best use of your caches**. You may find that you do not have a great hit/miss ratio with your caches. If you are doing SPARQL or SQL queries, you will want to see a hit rate on your triples and triples values caches. In general, the List cache is the other cache to look at. This hit ratio should be as close to 100% as possible. If you need to load up too many results from disk to resolve queries, your performance is going to suffer. If the list cache is running at a low ratio, this could be an indication that you have not properly sized that cache. We recommend contacting MarkLogic Technical

Support for assistance in changing cache sizes. Finally, if you are on Linux, make sure that the OOM killer (Out-of-Memory Killer) is not restarting MarkLogic (check `/var/log/messages` for details). This should only happen if you are running without swap configured.

Check the Slowest D-Node

We mentioned the use of `xdmp:query-meters()` earlier. That code shows you the d-node roundtrip times and counts. This means that for each query that executes, the e-node (the requesting host) needs to send the request to all of the hosts with forest data (or d-nodes). Those d-nodes then resolve the query locally and send results back to the e-node. Sometimes, you might have some sort of disk or another issue on a single host that can slow this whole process down. If you have one d-node that is three times slower, then all of your queries will run slowly—the queries can only run as fast as the slowest d-node. So, check query-meters output to make sure that d-nodes are returning results in roughly the same amount of time.

This is very general guidance—you might have some d-nodes that have a lot more work to do than others. But, even so, if you have a complex query that needs to do lots of work to resolve, you should see several slower d-nodes, not just one. Whenever you see a pattern where a single d-node is always slower, you likely have a hardware issue on that host.

If on Linux, Check Whether *Transparent Huge Pages* is Disabled

Running with *Transparent Huge Pages* enabled can cause all sorts of transient performance issues. If you are running into problems that you cannot isolate, make sure that you have disabled THP. For more information on how to do this, see the section in MarkLogic documentation on Linux Huge Pages and Transparent Huge Pages.

Check Your Network

Generally speaking, a 10Gig Ethernet network is sufficient for clusters up to several hundred nodes. However, that does not mean that the network is never a bottleneck. In addition to checking the d-node roundtrip time in `xdmp:query-meters()`, there are lots of details about network traffic in MarkLogic's Monitoring dashboard. This shows up as XDQP traffic (send and receive).

Check Your Database Replica

Also related to network traffic is database replication. If you are using database replication, remember that you have a configured lag limit. Queries on the replica must run at a timestamp that lags the primary, called the lag limit. The default is 15 seconds. See the Database Replication Guide for more info. If your replica has a hardware problem and is unable to keep up with your primary, the ingest processes will be throttled so that documents are not lost. Make sure that you check the database replication traffic in the Monitoring Dashboard and check the Error Logs.

Check for Lock Problems

Sometimes, it might appear that MarkLogic is not working very hard—the CPU utilization seems relatively low, and yet the query throughput rate is crawling and response times are long. Make sure that you do not have deadlocks in your code. It may be that MarkLogic is just waiting to resolve deadlocks, or simply waiting on an exclusive lock. You can find info about read/write locks and deadlocks in the Monitoring Dashboard and also in the Error Logs.

How to Monitor for System Resource Limitations

Two common causes of problems are resource limitations with disk (including I/O) and memory. So, here are a few notes about those resources and how to monitor them. For most resources, you never hit a ceiling that causes failure. For instance, with CPU, you might be running your CPU at 100% but that indicates you are using all the available resources. Could you go faster with more CPU cycles? Maybe. But, hitting 100% does not cause MarkLogic to restart or shut down. However, you should carefully monitor disk and memory utilization to avoid production downtime because your system ran out of disk or memory.

Disk Storage Capacity: Limits and Monitoring

MarkLogic requires disk space for merges. For a non-HA (no forest replicas), non-DR (no database replication) environment, the storage required for a merge is the total indexed forest size plus 96 GB per forest. There may be multiple merges occurring, but the merges are never more than 2x the merge-max-size, which is 48 GB by default in MarkLogic 8.0 and higher. Storage requirements can dramatically increase if HA and DR are configured. For example, if replication is paused, all of the unshipped changes need to be retained on the master. This can mean 2-4x the indexed data size. For database restores where data needs to be restored to an active database, plan for 2x indexed data size and 96 GB per forest.

If you do not have enough disk space to merge, there will be server errors and the database will become unavailable. This might tempt you to disable merging, but resist doing that! It simply prolongs the problem—forests have a maximum stand size of 64, and the database will shut down after that. So, you would end up with an even bigger problem—the database would be down and to get it back up you would need to merge all the changes. It is much better to let the merges happen gradually. Remember, merges are good. They are part of a self-tuning system. (For more details on merges, see the white paper called Understanding System Resources.)

Knowing this, it is easy to understand why you would want to monitor available storage on each host in your cluster. There are a number of tools you can use to do this—one of the easiest ways to see the storage available is to use **MarkLogic Ops Director** (See https://docs.marklogic.com/guide/opsdir/intro for more information; available in MarkLogic 9.0-2 and higher). Ops Director monitors available disk space on each host and sends alerts when the system approaches a threshold.

You can also monitor available storage per host using bash or Powershell scripts.

Memory Utilization: Limits and Monitoring

Running out of memory can cause the server to restart. Configuring swap space can alleviate this problem. Swap space is storage on disk that becomes available should the available memory on a computer be exhausted.

We recommend configuring swap space this way:

- Linux: If the host has 32GB of RAM or more, then swap space should be 32 GB. For less than 32 GB of RAM, swap should be equal to 1x RAM
- Windows: Swap space should be set to 2x RAM

Why use swap space? Is it a waste of storage? No, it is not. Configuring swap space provides a soft landing—instead of MarkLogic shutting down, you will see performance degrade and you will be able to control the restart. In other words, rather than crashing the plane, you get to do a safe emergency landing.

On both Linux and Windows, if you have not configured swap, the operating system will terminate the MarkLogic process when it is close to exhausting system memory.

So what can cause MarkLogic to run out of memory? Many components in MarkLogic use memory—there are caches, the in-memory stand, huge pages, range indexes, lexicons, registered queries, and more. Everything that helps MarkLogic run efficiently uses memory. Also, note that if using SQL or Semantics, there can be lots of in-memory joins, which can use large amounts of memory.

Because of the problems caused by running out of memory, it is critical to monitor the hosts in your environment to make sure there is enough available memory. Once again, MarkLogic Ops Director makes this easy and is available with MarkLogic 9.0-2 and higher. You can get a quick view of all the hosts in your cluster and see the percentage of available memory (see Hosts for more information). For a more fine-grained look at memory utilization, you can use the MarkLogic Monitoring Dashboard with MarkLogic 9.0-5 and higher to see the different types of memory utilization. As mentioned earlier, you can also see this information in the logs.

Here is an example of a system that is about to run out of memory:

```
2018-06-18 07:58:35.250 Info: Memory 96% phys=524288 virt=935913(178%)
rss=277896(53%) huge=230000(43%) anon=239535(45%) swap=1324652(252%)
file=173554(33%) forest=426118(81%) cache=147456(28%) registry=1(0%)
```

To get this data, you could look at your logs. You could also use bash or Powershell scripts to monitor each host and use vmstat or dstat or a similar utility to check memory utilization.

Since arbitrary queries can use large amounts of memory, particularly with Semantic joins, it may be important to lock down Query Console in production and limit the ability to execute arbitrary queries. A rogue query might cause memory usage to spike. This is hard to monitor for—by the time you discover that memory spiked, the system may already be swapping or have restarted. So, it is best to limit arbitrary queries if possible.

In general, if the memory used by MarkLogic server exceeds 90% of the physical memory, your system is entering a situation where it may start swapping or will have the Out-of-Memory (OOM) killer restart MarkLogic.

Conclusion

Methodology is your friend when it comes to performance tuning. Know your target SLAs. Find your limits. Alleviate your bottleneck. Retest. You should tune your configuration and application to best utilize your available computing resources.

Like all databases, MarkLogic should be tuned to provide the best performance to your application. Hopefully this technical white paper provided guidance to tune your system's configuration and your code. As always, we recommend that you reach out to MarkLogic Support if you have a performance problem that has you stumped. MarkLogic engineers are experts in tuning MarkLogic and they work closely with MarkLogic's Performance Engineering team.

MARKLOGIC CORPORATION
999 Skyway Road, Suite 200 San Carlos, CA 94070
+1 650 655 2300 | +1 877 992 8885 | www.marklogic.com | sales@marklogic.com

MarkLogic®

999 Skyway Road, Suite 200 San Carlos, CA 94070

+1 650 655 2300 | +1 877 992 8885

www.marklogic.com | sales@marklogic.com