PERFORMANCE

Understanding System Resources

MARKLOGIC WHITE PAPER · OCTOBER 2018

There are many aspects involved in understanding resource consumption. To make sense of them, this paper will break down the overall problem by describing common MarkLogic functions and then provide detail on system impact (I/O, CPU, memory, storage size, network). We also show examples of how to monitor these various system components, how to remediate problems, and important limits to keep in mind.

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

Overview

MarkLogic®, the only Enterprise NoSQL database, is designed to scale to large amounts of content. The intent of this guide is to provide details about MarkLogic core functions and operations so that DBAs, Infrastructure Architects, IT Managers, Developers and Engineers can configure MarkLogic for optimal success.

Because there are many aspects involved in understanding resource consumption, this guide attempts to break down the overall problem by describing common MarkLogic functions and then providing detail on system impact (I/O, CPU, memory, storage size, network). We also show examples of how to monitor these various system components, how to remediate problems, and important limits to keep in mind.

The whitepaper introduces basic MarkLogic terms for those readers who might be new to the product and concepts, but we also assume that readers will consult the MarkLogic Documentation for background on specific functions or operations. This guide views MarkLogic through the lens of resource consumption and infrastructure planning.

This guide applies to MarkLogic version 7 and 8.

MarkLogic Server Architecture at a Glance

MarkLogic Server has many different components, all of which require varying levels of system resources.

Throughout this paper, refer this diagram to help understand what types of system resources are being used for any given process flow.

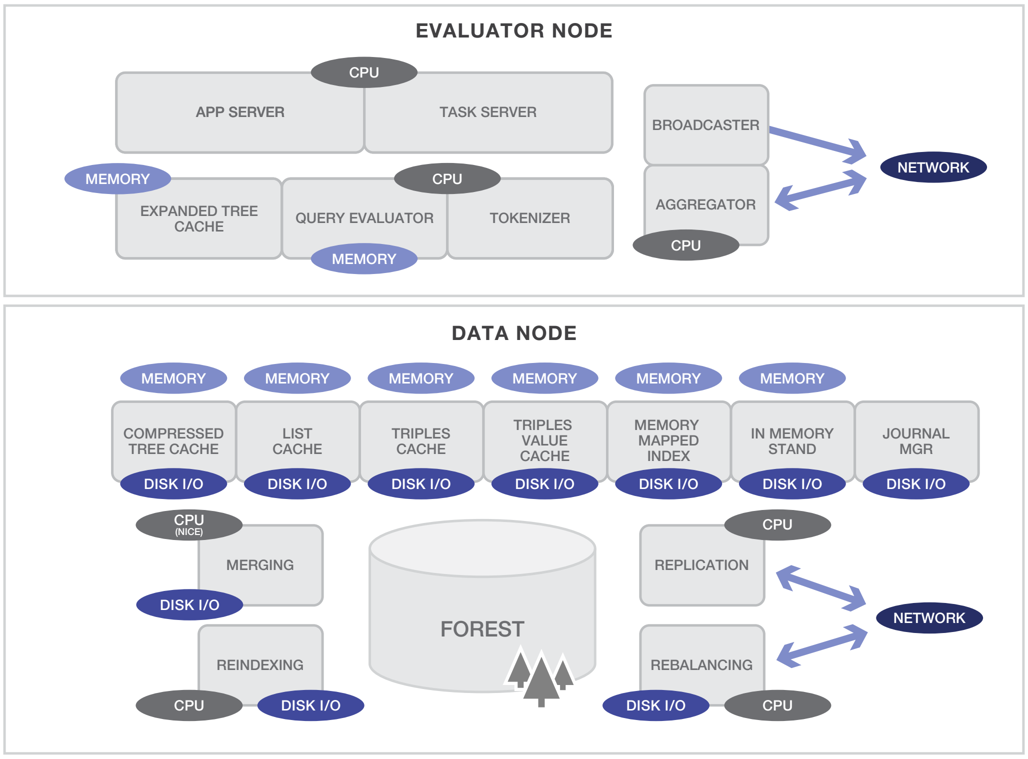


Figure 1: E-and D-nodes

Terminology and General Concepts

A database consists of one or more *forests*. A system can have multiple databases, and generally will, but each query or update request executes against a particular database. A *forest* contains a set of records called *documents*, along with all their indexes, mapped to a physical location on disk. Documents within a forest are organized into *stands*. Each forest holds one or more *stands*. A stand holds a subset of the forest data and exists as a physical subdirectory under the forest directory.

Each MarkLogic server instance runs a process that can perform *Evaluator (E-node)* and *Data Manager (D-node)* tasks. E-nodes have application servers, parse requests, and return responses. E-nodes are also responsible for executing JavaScript or XQuery server-side code.

D-nodes hold data and indexes and are responsible for maintaining transactional integrity during insert, update, and delete operations as well as other data management tasks such as forest recovery, backup operations, and on-disk forest management. D-nodes are also responsible for providing forest optimization (merges), index maintenance, and content retrieval.

Any specific host running MarkLogic can be configured as *either* an E- or D-node or a combined E/D node. This is a configuration option. D-nodes are defined by the presence of an attached forest on the node; E-nodes **don't** have attached forests, but they do have application servers. So the components are much the same for a shared E/D node – it's just that all functions run on a single host. We'll talk more later about the reasons to create an architecture with separated E/D nodes, but for now, just consider that either configuration is fine.

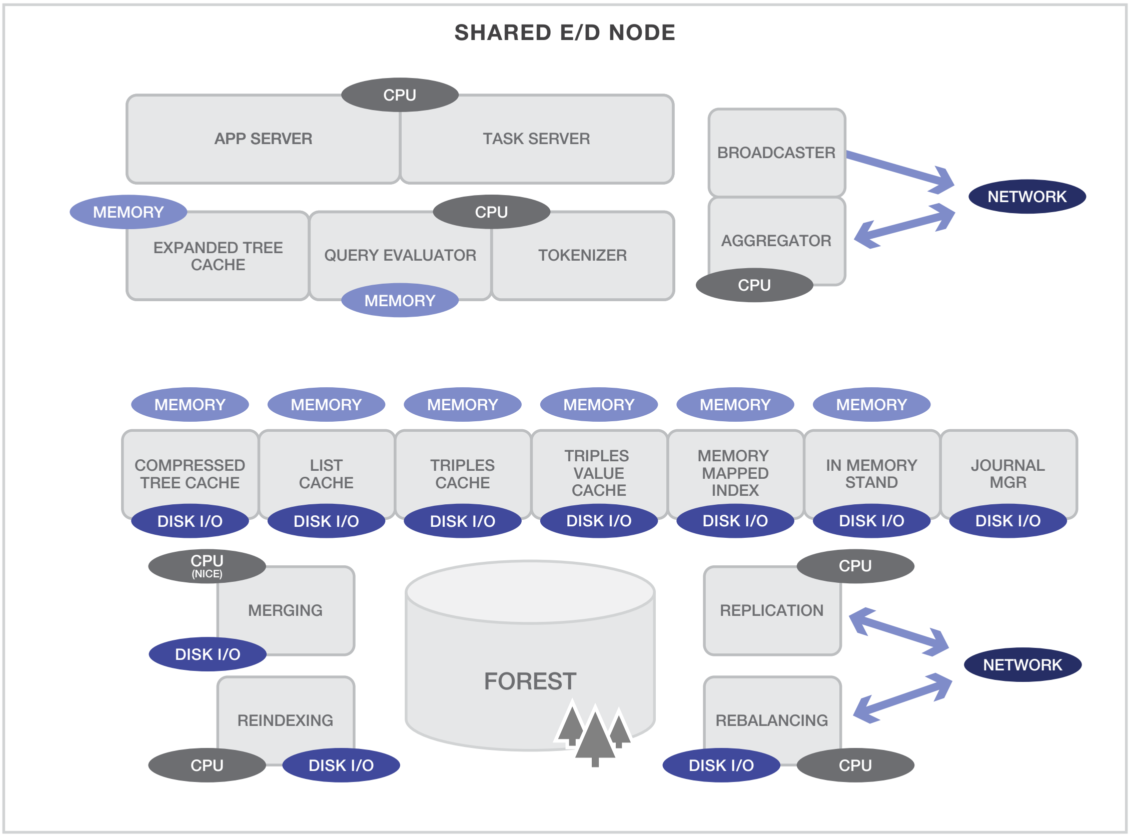


Figure 2: Shared E/D-node

What is a Database Server?

MarkLogic is a database server – but one with many features not found in traditional database systems. It has the flexibility to store structured, unstructured, or semi-structured information. It can run both database-style queries and search-style queries, or a combination of both. It can run highly analytical queries too. It can scale horizontally. It's a platform, purpose-built from the ground up, that makes it dramatically easier to author and deploy today's data-rich applications.

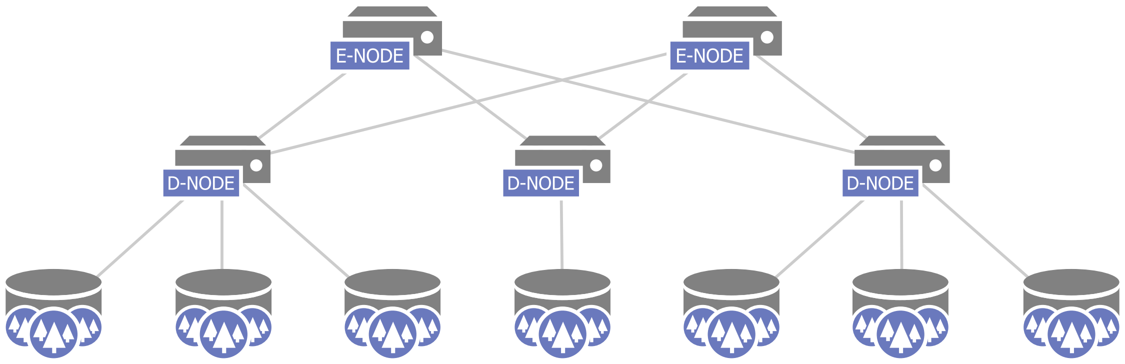


Figure 3: Cluster

MarkLogic Server is both a database and an application server. When you issue a query against MarkLogic, you send the query to the application server, which then broadcasts the query out to the forests. If you've got a multi-host environment, the E-node process will broadcast to all the D-node processes. The D-nodes resolve the query against their indexes and return a result set composed of documents. These documents are in a compressed binary format; they are first saved in-memory into the compressed tree cache, and then sent to the E-node where they are expanded into their original format in the expanded tree cache. The E-node will aggregate the results from all the D-nodes and sort the result set if needed. The application server then returns the result set to the calling client.

When we use the term "application server" here, we are specifically referring to the MarkLogic Application Server – which is simply a communication mechanism for the database backend. Just as a relational database might provide a web services endpoint that parses URL parameters, translates those into query terms, executes the query against the database and then returns a response, MarkLogic application servers provide HTTP-based endpoints for data services.

To explain how MarkLogic components use system resources, we'll walk through the lifecycle of our data and explain process flows in the context of our architectural diagram.

Part I will cover ***ingestion***. We'll start by looking at what happens at a system level when documents are inserted into the database, explaining the in-memory stand, merges and journal writes. Part II will cover ***tokenization* and *indexing***. We'll explain how documents are broken up into tokens and then indexed when ingested. Part III will focus on ***queries*** and what happens when queries are executed against the server. Part IV will cover ***operational concerns*** as they pertain to system resources: cluster topologies, backups, rebalancing and reindexing, tiered storage, high availability and swap recommendations.

Part I: Ingestion

Ingestion

Ingestion involves system resources from the D-node functionality of MarkLogic Server.

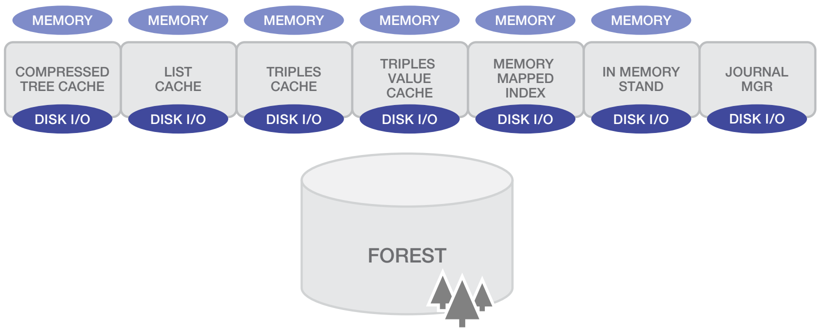


Figure 4: D-node

When content is ingested, we're concerned primarily with CPU and I/O. The forest at the bottom of the diagram represents the data on disk. We're also writing journals as well, represented by the Journal Manager on the right. **So ingestion is a heavy I/O and CPU intensive activity. Memory is also involved, since we are writing into the in-memory stand and populating in-memory caches.**

30,000 Foot View: Ingest Summary

Before we go too deep into the internals of what happens during ingestion, here's the high level view: when documents are ingested into MarkLogic, they are loaded into memory and the transactions recorded into an on-disk journal. Periodically, a forest checkpoint occurs where the data in-memory is flushed to disk. In addition, MarkLogic runs a background process called *merging* that combines and optimizes the stands and index structures on disk.

There are multiple operations that consume resources in the ingest process. Some of those operations must happen in the foreground, immediately: writing to the journal, for example. Other operations happen in the background, prioritized behind foreground operations (and subject to throttling through administrative settings.) You will find that MarkLogic utilizes resources – particularly I/O and CPU – even at times when no queries are issued. The system is constantly optimizing for the next read or write operation.

This means that you'll observe the following, all of which is ***normal and means the system is operating properly:***

- **Spiky I/O writes.** Big sequential writes, lots of smaller background I/O CPU activity (seen as nice percentage); this is different from a traditional relational database where the I/O may be relatively stable
- **High I/O and CPU activity, as well as increased memory utilization.** If the documents need to be transformed in any way on input, MarkLogic will use CPU resources to do the transformations

That's the high level; if that's all you want to know about ingest, skip to the merge section. Otherwise, continue on to understand these on-disk and in-memory structures and all of the associated details.

Ingest in Detail

When a new document is loaded into the database, the E-node first tokenizes the document. MarkLogic puts this document in an in-memory stand and writes the action to an on-disk journal to maintain transactional integrity in case of system failure.

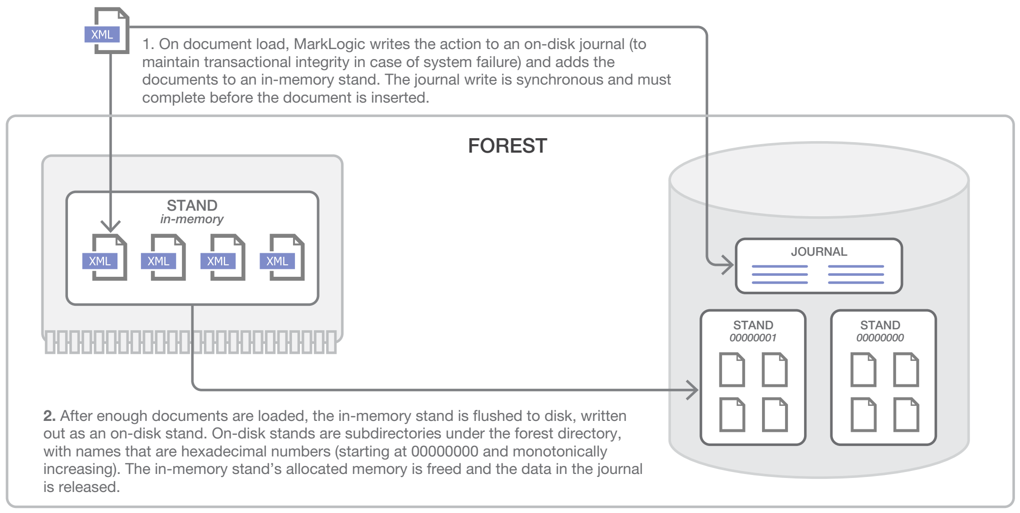
After enough documents are loaded, the in-memory stand will fill up and be flushed to disk, written out as an on-disk stand. Each new stand gets its own subdirectory under the forest directory, with names that are monotonically-increasing hexadecimal numbers (ie, `00000000, 00000001,` etc.).That on-disk stand contains all the data and indexes for the documents loaded since the last in-memory stand was written out to disk. It's written from the in-memory stand out to disk as a sequential write for maximum efficiency. Once it's written, the in-memory stand's allocated memory can be freed.

As more documents are loaded, they go into a new in-memory stand. This in-memory stand will eventually fill up, and the in-memory stand gets written as a new on-disk stand. Sometimes, under load, you have two in-memory stands at once, when the first stand is still writing to disk as a new stand is created for additional documents – ingest does not stop while the in-memory stand is being written out to disk.

The mechanism continues with in-memory stands filling up and writing to on-disk stands. However, the in-memory stand size is much smaller than the optimum on-disk data structure for resolving queries. As documents are added to stands, terms are added to the stand's term list, which is an index used to resolve search queries. To read a single term list, MarkLogic must read the term list data from each individual stand and unify the results. To keep the number of stands to a manageable level where that unification isn't a performance concern, MarkLogic runs merges in the background. A merge takes some of the stands on disk and creates a new singular stand out of them, coalescing and optimizing the indexes and data, as well as removing any previously deleted documents. After the merge finishes and the new on-disk stand has been fully written, and after all the current requests using the old on-disk stands have completed, MarkLogic deletes the old on-disk stands. It's important to note that this process happens not only on ingest but as part of any update.

Each forest has its own in-memory stand and set of on-disk stands. Loading and indexing content is a largely parallelizable activity so splitting the loading effort across forests and potentially across hosts in a cluster can help scale the ingestion work.

WRITING DOCUMENTS TO STANDS



1. On document load, MarkLogic writes the action to an on-disk journal (to maintain transactional integrity in case of system failure) and adds the documents to an in-memory stand. The journal write is synchronous and must complete before the document is inserted.

2. After enough documents are loaded, the in-memory stand is flushed to disk, written out as an on-disk stand. On-disk stands are subdirectories under the forest directory, with names that are hexadecimal numbers (starting at 00000000 and monotonically increasing). The in-memory stand's allocated memory is freed and the data in the journal is released.

Figure 5: Writing Documents to Stands

Important Limits

More than 96 million fragments in a forest is considered to be an error condition.
If forests are approaching this size, add forests and/or hosts and rebalance the content. The server writes messages about these conditions to the ErrorLog:

Fragment Count	Log Level
96 million or more	Error
120 million or more	Critical
144 million or more	Alert
160 million or more	Emergency

More Detail About the In-memory Stand

The in-memory stand size and in-memory caches are all configurable. If the group is set to "Info" log level, you may see messages in the log related to the in-memory stand.

The **XDMP-INMMTREEFULL, XDMP-INMMLISTFULL, XDMP-INMMINDXFULL XDMP-INMREVIDXFULL, and XDMP-INMTRPLIDXFULL** messages are *informational only*. These messages indicate that in-memory storage is full, resulting in the forest stands being written out to disk. There is no error as MarkLogic Server is working as expected.

- **XDMP-INMMTREEFULL** indicates the in-memory tree storage is full
- **XDMP-INMMLISTFULL** indicates the in-memory list storage is full
- **XDMP-INMMINDXFULL** indicates the in-memory range index storage is full
- **XDMP-INMREVIDXFULL** indicates the in-memory reverse index storage is full
- **XDMP-INMTRPLIDXFULL** indicates the in-memory triple index storage is full

If these messages consistently appear more frequently than once per minute, increasing the 'in-memory' settings in the affected **database** may be appropriate. If memory is at capacity, then adding servers and rebalancing content across the new hosts is another way to increase cache capacity.

XDMP-INMMTREEFULL corresponds to the "in-memory tree size" setting. "in-memory tree size" specifies the amount of memory to be allocated for managing document data for an in-memory stand.

XDMP-INMMLISTFULL corresponds to the "in-memory list size" setting. "in-memory list size" specifies the amount of memory to be allocated for managing termlist data for an in-memory stand.

XDMP-INMMINDXFULL corresponds to the "in-memory range index size" setting. "in-memory range index size" specifies the amount of memory to be allocated for managing range index data for an in-memory stand.

DMP-INMREVIDXFULL corresponds to the "in-memory reverse index size" setting. "in-memory reverse index size" specifies the amount of memory to be allocated for managing reverse index data for an in-memory stand.

XDMP-INMTRPLIDXFULL corresponds to the "in-memory triple index size" setting. "in-memory triple index size" specifies the amount of memory to be allocated for managing triple index data for an in-memory stand.

Since the in-memory stand size is configurable, why not make it as big as possible? Then you'd load all your content into memory and ingestion would be very fast, with the benefit of durability provided by the journal writes. While that may be true, this is not the best approach if you also need to query the data while ingesting. The on-disk data structures are optimized for queries; the in-memory structures are optimized for ingestion. Having small in-memory stands and larger on-disk stands optimizes for overall performance.

Resource Consumption Summary for Ingestion

Now that we understand the mechanisms involved in ingestion and writing the in-memory stand out to disk, let's look at the system resources involved. Remember we said that ingestion is an I/O and memory intensive process. If transformations are involved, it's also a CPU intensive process. Here's a summary of what to expect:

System Resource	Impact
Storage	For a non-HA (no High Availability forest replicas) non-DR (no database replication) environment, and with the default 32 GB merge-max-size, the storage required for a merge is the total indexed forest size + 64 GB per forest. There may be multiple merges occurring, but the merges never require more than 2x the merge-max-size. Storage requirements can increase significantly if HA and DR are configured; for example, if replication is paused, all of the unshipped changes need to be retained on the master, so this can mean 2x to 4x the indexed data size. For database restores where data needs to be restored to an active database, plan for 2x indexed data size + 64 GB per forest.
I/O throughput	When the in-memory stand is flushed to disk, MarkLogic does sequential writes in 512KB blocks. Journal writes are of varying sizes, but the largest write is also 512KB.
CPU	If ingest does any transformation (XSLT, or a special XQuery or JavaScript module that runs on each document as it comes into the database), CPU utilization will increase. Mileage will vary, but CPU should be monitored during ingest.
Memory	As documents are added to the in-memory stand, index structures are also created. Memory is allocated for the tree structure for the documents themselves, for the term list indexes for those documents, and for the range index, reverse index and triple index. The memory is not pre-allocated; instead the memory is allocated as documents are loaded, and then released when the in-memory stand is flushed to disk. You should therefore expect to see memory utilization increase as content is ingested.
Network	Can vary depending upon the ingest mechanism. Using MLCP and the fastload ingest option causes the client to insert data into the in-memory stand directly on the data node that hosts the target forest; otherwise, the evaluator node sends the data to the data node, which requires a network hop.

More About Merges

30,000 Foot View: Merge Summary

Merging is a background process that consolidates stands on disk and optimizes indexes. It runs periodically when certain thresholds are met. When ingesting or updating content, you'll observe the following, **all of which is normal and means the system is operating properly:**

- **Spiky I/O.** This happens when periodic merges run and do big I/O operations to combine files on disk
- **CPU.** Merges will show up as nice % in CPU statistics

When ingesting content, you should expect to see heavy I/O and CPU activity related to merges.

What Happens During a Merge

When a document is updated, new versions of all of the fragments associated with the document update are created in a new stand. Any old versions of the fragment remain in the old stand with a system timestamp that lets MarkLogic Server know that they are old versions of the fragments. Similarly, when a document is deleted, its fragments remain in the old stand with a system timestamp that lets MarkLogic Server know that they are old versions of the fragments.

Merges occur to move any unchanged fragments from an old stand into a new stand, deleting any old versions of fragments (including deleted fragments), freeing up disk space and compacting the usable fragments so they are all together on disk. Additionally, merges combine index data for all of the fragments in a stand, optimizing the indexes. Merges are a normal part of database operation, and they ensure that the system continues to perform at its best as updates and deletes occur.

To summarize, as part of merging, the following occurs:

- Multiple stands are combined into one for improved performance
- Disk space is reclaimed
- Indexes and lexicons are combined and re-optimized based on their new size

Merges are a way of self-tuning the performance of the system, and MarkLogic Server continuously assesses the state of each database to see if it would benefit from self-tuning through a merge. In most cases, the default merge settings and the dynamic nature of merges will keep the database tuned optimally at all times.

But How Does the Algorithm Work?

As mentioned above, MarkLogic uses an algorithm to determine when to merge (consolidate stands and remove deleted documents). The cost of merging is proportional to the size of the stands being merged, but the benefit is proportional to the number of stands being merged. This is why we merge smaller stands together more frequently, and bigger stands less frequently, and why we try not to merge small stands with big stands (prefer to merge small stands with other small stands).

There are two kinds of merges in MarkLogic:

- Manual Merge (executed by user request)
- Automatic Merge

A manual merge takes all stands in a forest and merges them down to the minimum number of stands that satisfies the merge max size, taking into account the cost and benefit of merges.

Estimated Post-merge Size

The size of stands plays an important role in whether and how they are merged. When referring to the stand size, the algorithm uses an estimated post-merge size, rather than the current on-disk size. This is because merging will result in the removal of deleted fragments and reclamation of that space. Because it's expensive to count up the total size of all non-deleted fragments in a stand, a heuristic is applied based on the percentage of deleted fragments in the stand to estimate the post-merge stand size.

Automatic Merge

MarkLogic evaluates all stands in all forests once per second to determine whether they should be merged. It does this by ordering all stands by size, then, starting with the biggest stand that is smaller than the max merge size (the candidate stand), it looks to see if the sum of all other stands' sizes exceeds the merge min ratio.

If the candidate stand doesn't qualify for a merge, the next-biggest stand is made the candidate, and so on until either a candidate qualifies or all stands have been evaluated.

If a candidate qualifies for merge, the merge process takes place on that candidate and all smaller stands.

Merge Process

The merge itself happens by taking the set of stands to merge (which in the case of a manual merge is all stands in the forest), and going through them in size order.

```
For each stand (in estimated size order from largest to smallest):
  If the stand size is >= max merge size, then
     If (1 – the percentage of deleted fragments) < merge min ratio, then merge this stand with itself (just
     to remove deleted fragments).
     Else find the set of next-smallest stands that can be merged with this stand while still keeping it
     under the max merge size, and merge these all into one stand.
Continue with remaining stands until all have been merged.
```

Merges and Memory

Although merges are generally not memory-intensive operations, it is important to understand how memory utilization might be impacted by merges. In particular, there is a database setting that can impact the memory utilization and performance of merges: **preload mapped data**. When this setting is enabled, then memory-mapped data (range indexes and lexicons, for instance) are loaded immediately into memory when a stand opens. If mapped data is not preloaded, then it will be paged into memory dynamically as needed.

What does this mean for merges? At the end of a merge, if you have preload-mapped data enabled, all memory-mapped file data pages are preloaded. If the newly merged memory-mapped file data pages are still in the Linux kernel buffer cache at that point then there are no major faults. If the newly merged memory-mapped file data pages are no longer in the Linux kernel buffer cache then they are faulted in. So if you notice a high number of page faults at the end of a merge, this is the likely reason.

Obsolete Stands

The files that make up a stand reside in a directory on the file system. The directory for a stand that has been marked as obsolete contains a file with the name `Obsolete`. When the stand is no longer needed, the directory and contents will be deleted.

Obsolete stands are created during the normal operation of MarkLogic Server. Stands in a forest are marked as obsolete so that MarkLogic Server can recover the forest from an unexpected outage. There are many reasons a stand can be marked as obsolete:

1. It is the output stand of a merge that has not yet completed. Once the merge finishes, the stand is unmarked and is available to the forest.
2. The stand is in the process of being created from an in-memory stand but has not yet been completely saved on disk. Once the in-memory stand is completely saved to disk, the stand is unmarked and is available to the forest.
3. All the data has been moved to a new stand through merging but this stand is still in use by a query. Once the last query that is using the stand completes, the stand is deleted.
4. It has been replaced by a merge but is still in use by a forest or database backup. Once all backups that are using the stand completes, the stand is deleted.

If a forest has been disabled, when it is subsequently re-enabled, any stand in the forest that is marked as obsolete will be deleted.

Obsolete stands should only exist at startup in the situation where the forest is disabled unexpectedly. This may occur:

- If MarkLogic Server was stopped unexpectedly
- If the stand resides on a network attached device and the device became unreachable
- If there were in-flight queries or backups when MarkLogic Server was stopped

A forest can be started by:

- (Re)starting MarkLogic Server
- (Re)starting (Disabling and Enabling) the Forest

Resource Consumption Summary for Merges

System Resource	Impact
Storage	For a non-HA (no High Availability, no forest replicas) non-DR (no database replication) environment, the storage required for a merge is the total indexed forest size + 64 GB per forest. There may be multiple merges occurring, but the merges are never more than 2x the merge-max-size, which is 32 GB by default. 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, so this can mean 2x to 4x the indexed data size. For database restores where data needs to be restored to an active database, plan for 2x indexed data size + 64 GB per forest.
I/O throughput	Merges write in sequential 512KB units of work. This is not configurable and should be taken into consideration when configuring stripe and block sizes for the file system. As stands are combined and written to disk, I/O throughput demands increase significantly. Merges use background I/O that will show up as nice % in CPU reports. Merge priority can be changed to either dial back merges or to promote more aggressive merges (slow things down, freeing up system resources, or speed things up, resulting in a faster merge but more I/O and CPU utilization).
CPU	Merges can be CPU intensive; the process of determining which stands should be merged and which fragments removed may cause CPU to increase during the merge process.
Memory	Merges generally don't use a great deal of memory; indexes are optimized on disk and then loaded into memory as needed. However, if **preload mapped data** is enabled on the database, then all memory-mapped file data pages will be loaded into memory at the end of a merge. If your database has lots of range indexes and preload mapped data is enabled, memory will spike at the end of the merge.
Network	Merges do not utilize the network unless the file system itself is reached via the network (NFS, FCoE).

Important Limits

MarkLogic Server limits the maximum number of stands in a forest to 64. If a single forest in a database reaches 64 stands, the database will become unavailable, so you need merges to be able to keep up with ingestion. The merge priority helps you balance ingest and merging. Typical reasons for merges to not keep up with ingestion include lack of I/O bandwidth on the storage system or too low a background-io-limit (see below).

Merges and Background I/O

Merges run as background I/O. This can be configured as a setting on the group. For example:

```
background-io-limit = 100
```

This will limit the background I/O for that group to 100 MB/sec per host across all hosts in that group. This should only be configured if merges are causing problems – it is a way of throttling back the I/O used by the merging process. Setting background I/O to 100 is a good starting point if you find that your merges are progressing too slowly. Background-io-limit is generally not used for direct attached storage.

Journal Writes and The Fast Data Directory

Journal Writes and the Fast Data Directory: 30,000 Foot View

MarkLogic uses an on-disk journal to maintain transactional integrity during updates. Writes to the journal are multi-threaded and there are multiple journals in a database. Writes will be no larger than 512 KB. The fast data directory is an option on a forest whereby you can use SSD storage to write the in-memory forest and journals out to the substantially faster SSD disk. This is helpful in an I/O constrained environment – but if enough I/O is available in your infrastructure, using SSDs for the fast data directory may not be cost effective.

Journal Writes: Details

When you load content, MarkLogic Server performs updates transactionally, locking documents as needed and persisting the transaction in the journal before the transaction commits. By default, all documents involved in a transaction are locked during an update and the journal is used to preserve committed transactions even if the MarkLogic Server process ends unexpectedly. Like all ACID-compliant databases, locking can have a significant effect on performance. Overly aggressive holding of locks can slow performance and appear as an I/O contention issue at first glance. There are detailed locking statistics available in Performance History.

The database settings *locking* and *journaling* control how fine-grained and robust you want this transactional process to be. By default, it is set up to be a good balance between speed and data integrity. All documents being loaded are locked, making it impossible for another transaction to update the same document being loaded or updated in a different transaction.

There is a journal write to disk on transaction commit (in addition to the journal write on transaction begin), and by default the system relies on the operating system to perform the disk write. Therefore, even if the MarkLogic Server process ends, the write to the journal occurs, unless the computer crashes before the operating system can perform the disk write. Protecting against the MarkLogic Server process ending unexpectedly while depending on the operating system to complete the write is the fast setting for the journaling option. If you want to protect against the whole computer crashing unexpectedly, you can set the journaling to strict. A setting of strict forces a filesystem sync before the transaction is committed. This takes a little longer for each transaction, but protects your transactions against the computer failing. Often, the disk subsystem does not actually complete the filesystem sync before reporting it back as complete – we recommend RAID cards with batteries or capacitors to make sure filesystem sync actually completes in the event of a crash.

Understanding the Fast Data Directory

Solid-state drives (SSDs) have performance that's dramatically better than spinning hard disks, at a price that is higher. When considering SSD utilization for MarkLogic, consider the realized throughput of large sequential writes on the drive or subsystem you are considering. While random reads are almost always faster on SSD, some SAS/SATA-based SSD's can actually be slower at writing data than spinning disks, particularly as they fill up and age.

However, many SSD systems are PCIe based and can offer massive performance gains vs. spinning disk. While the cost per TB is higher than spinning disk, the cost of throughput is quickly dropping below that of spinning disk.

In order to realize the benefits from high throughput SSD systems without the cost of provisioning 100% of the database on SSD, MarkLogic has a configurable "fast data directory" for each forest, which you setup to point to a directory built on a fast filesystem, such as one using SSDs. It's completely optional; if not present then nothing special happens and all the data is placed in the regular "data directory." But if it is present, then each time a forest does a merge (saving an in-memory stand included), MarkLogic will attempt to merge onto the fast data directory. When it can't because there's no room, it will use the regular data directory. Of course merging onto the "data directory" will then free up room on the fast data directory for future stands. The journals for the forest will also be placed on the fast data directory. As a result, all the smaller and more frequent merges will happen on SSD, improving utilization of disk I/O bandwidth. Note that frequently-updated documents tend to reside in the smaller stands and thus are more likely to reside on the SSD.

The overall philosophy of the Fast Data Directory (FDD) is to try to put as many writes and seeks to it as we can, so in addition to journals, we try to put as many stands as possible on the FDD. That's because for each term in a termlist that is not in List Cache, we have to seek and read that term once per stand. Having more stands on FDD means more of these seeks happen on FDD.

The way to have more stands on FDD is to put smaller stands on FDD. The smallest stands are checkpointed in-memory stands, so we always try to leave room for those on FDD. Given room for those, we then use the rest of the FDD to put new stands as we merge, however to preserve space for future small stands, we are conservative in deciding whether to put merge destination stands on FDD (so even if there's room, if it's getting crowded, we may merge to spinning disk instead).

Monitoring Journal Writes

Journal writes can be monitored using the MarkLogic Monitoring Dashboard. The Journal and Save Write is located on the Rates and Loads Overview page.

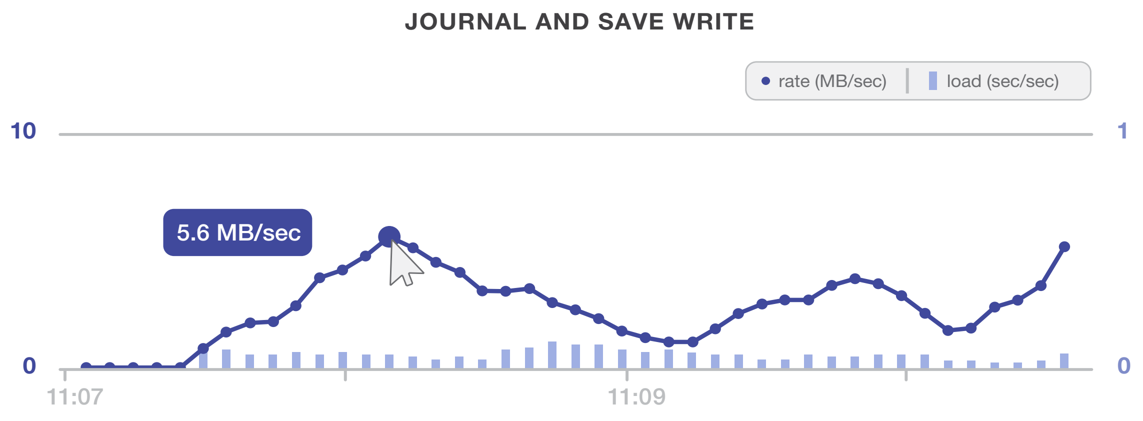


Figure 6: Journal and Save Writes in the Monitoring History dashboard

In addition, you can find historical journal write information in the Monitoring History dashboard.

Part II: Indexing and Tokenization

This part of the discussion also happens at the D-node layer – so here's that diagram again:

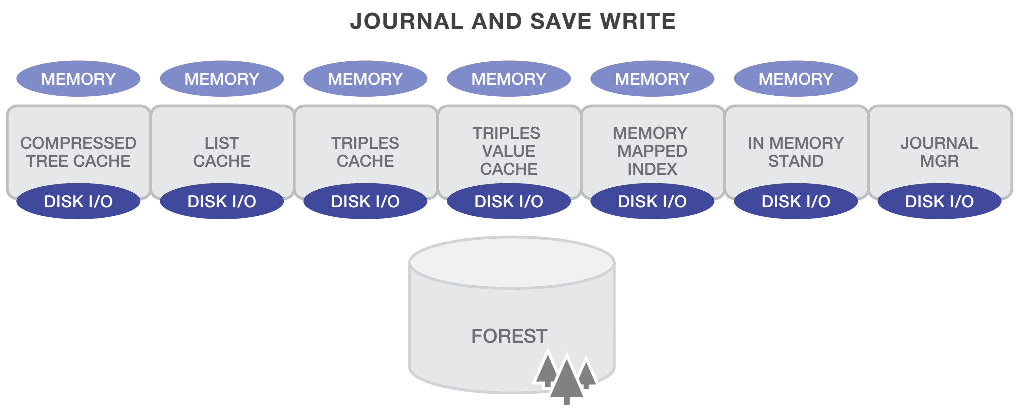


Figure 7: D-node

In MarkLogic, the indexes are stored with the data -- index files for on-disk stands are stored on disk with the stand, and are cached in-memory on the host where the forest resides. So you end up with a highly distributed system where the indexes are physically coupled with the data. In this part of the paper, we'll be talking about how these indexes are created and updated.

Tokenization

In order to create meaningful indexes, MarkLogic must first tokenize the incoming content. When you load content (which is made up of text strings) into MarkLogic Server, the string is broken down to a set of parts, each of which is called a *token*. Each token is classified as a word, as punctuation, or as whitespace. The process of breaking down strings into tokens is called tokenization. Tokenization occurs during document loading as well as during query evaluation, and they are independent of each other.

Tokenization is language-specific; that is, a given string is tokenized differently depending on the language in which it is tokenized. The language is determined based on the language specified at load or query time (or the database default language if no language is specified) and on any `xml:lang` attributes in the content.

There are different tokenizers for different languages. There's also a custom tokenizer, should none of the out-of-the box tokenizers meet application requirements. More details on how to override the tokenizer can be found in the Custom Tokenization chapter of the Search Developer's Guide

MarkLogic Indexes

When documents are ingested into a MarkLogic database, we create a number of indexes based upon the database configuration settings. There are five basic categories of indexes in MarkLogic:

- Universal Index
- Range Indexes
- Lexicons
- Reverse Index
- Triple Index

MarkLogic Indexes at 30,000 Feet

MarkLogic indexes consist of both in-memory and on-disk data structures. Range indexes and lexicons are stored in-memory-mapped files – if your application uses them, you'll see equivalent memory usage. *Term lists* are part of what's known as the Universal Index, and those are both in-memory (in the List Cache) and in files on disk. The Triple Index also uses memory and disk resources; although not memory mapped, the Triple Cache will grow and shrink as needed to support semantics queries.

Generally, utilization of indexes means you'll need both more storage space on-disk, and potentially more space utilized in-memory, in the case of lexicons and range indexes. More indexes mean larger index files, and slower ingestion – more work needs to be done as content is ingested to create the index files. Of course, more indexes, particularly when residing in-memory, can result in query performance 100X-1000X faster than if the query needs to be resolved through additional work at query time.

MarkLogic Indexes: the Details

Universal Index

The Universal Index contains entries for both terms and structure of documents. Depending upon database settings, the Universal Index also contains entries for phrases, specific element values, and positions (e.g. "brown" near "fox"). A number of different database settings (e.g. "Fast element queries") can impact which options are enabled within the Universal Index. As more options are selected, the index files are created with more entries per document and the indexes are larger overall; the need to add more items to an index slows down ingestion somewhat. Index options are almost always a trade-off between ingest speed, storage requirements and query speed.

Range Indexes

Range queries are designed to constrain searches on ranges of a value. In order to constrain searches on ranges, a range index must be configured for the JSON property or XML element you'd like to search. For example, if you want to find all articles that were published in 2005, and if your content has an element (or an attribute or a property) named `PUBLISHDATE` with type `xs:date`, you can create a range index on the element `PUBLISHDATE`, then specify in a search that you want all articles with a `PUBLISHDATE` greater than December 31, 2004 and less than January 1, 2006. Because that element has a range index, MarkLogic Server can resolve the query extremely efficiently.

Range indexes are memory-mapped files. You can control when these files are loaded into memory: by setting preload mapped data (and preload replica mapped data) to true, you specify that the indexes are loaded into memory when the forest is mounted. If set to false, these are loaded dynamically into memory.

Lexicon Index

Lexicons are lists of unique words or values, either throughout an entire database (words only) or within named elements or attributes (words or values). Also, you can define lexicons that allow quick access to the document and collection URIs in the database, and you can create word lexicons on named fields. The collection and URI lexicons are just range indexes. Both the word and value lexicons are memory mapped like range indexes, but have less associated information (for instance, no position information or IDs of documents containing the lexicon).

Reverse Index

Typically, a search starts with a query and finds the set of matching documents. A reverse query does the opposite: you start with a document and find all matching queries – the set of stored queries that if executed would match this document. Enabling the reverse index makes this search possible.

Triple Index
The triple index is used to index schema-valid `sem:triple` elements found anywhere in a document.

Internally, MarkLogic stores triples in two ways: triple values and triple data. The triple values are the individual values from every triple, including all typed literal, IRIs, and blank nodes. The triple data holds the triples in different permutations, along with a document ID and position information. The triple data refer to the triple values by ID, making for very efficient lookup. Triple data is stored compressed on disk, and triple values are stored in a separate compressed value store.

When triple data is needed (for example during a lookup), the relevant block is cached. Unlike other MarkLogic caches, the triple cache and triple value cache shrink and grow, only using memory when needed, rather than having all memory pre-allocated.

Understanding Index Expansion and Compression Patterns

Why Don't we Just Flip all Index Settings to True and Index Everything?

While this would certainly make searches faster at run time, enabling indexes has a cost. More indexes means slower ingest and greater index expansion, but faster and more accurate searches. Adding an index setting generally means that the server will need to do more work on ingest; in addition to storing the document itself, the indexes will need to be updated with information from/about that document. So it makes sense to be judicious about index settings and to use what is needed. This will be an iterative process; you cannot determine the optimum trade-offs in indexing vs. query time processing without having both the queries, query patterns and the content on hand. As with all databases, this tuning work cannot be complete until the system is built and may need to be revisited as query patterns or data changes substantially. The good news is that you can change your index settings at any time and MarkLogic will re-index your data for you without taking your application offline. MarkLogic is smart enough to only re-index documents that are impacted by the new index settings.

How much do different index settings impact disk size and performance? The only real way to tell is to enable indexes and test the results, but there are some general patterns that may be helpful to understand. Field Indexes can be significantly expensive, both in terms of slowing down ingest and index sizes on disk, so planning for index expansion with Field Indexes is important. *Fast-phrase* or *fast-case-sensitive* indexes have a minor impact in terms of disk storage, so these may be good options for applications that need these features.

Document Compression

When a document is ingested into MarkLogic server, it's converted to a highly compact serialization of the XML or JSON document. The data is stored in a TreeData file, and the tree structure of the document gets saved using a compact binary encoding. The end result is a highly compact serialization of the file, much smaller than the content you see in a regular file. This serialization and tokenization is automatic and cannot be turned off. Shortening of attribute names, use of coding vs. readable terms is often not necessary and might not result in the same realized cost savings as other systems without the advanced compression of MarkLogic.

Index Expansion

Even as documents are compressed, different combinations of index settings can contribute to a fairly wide range of index expansion ratios. Because there are so many different combinations of settings that can influence index expansion, looking at some common patterns may help in understanding index expansion.

Pattern: Positions

MarkLogic uses positions on various indexes to resolve `NEAR` queries, phrase searches, and some XPath expressions related to parent/child node relationships with multiple levels of nesting. Position calculations require time and memory proportional to the number of terms being examined, so MarkLogic uses position information to reduce the number of documents and terms that need to be processed. Within the Universal Index, the following settings all involve positions:

- word positions
- element word positions
- element value positions
- attribute value positions
- field value positions
- trailing wildcard word positions

In addition, positions indexes can be added to range values and to triples.

For each positions index, MarkLogic adds positions information about each term to the term list. This makes the termlist size proportional to the number of *instances* of a term in the database, versus the number of *documents* containing that term. This can be a very significant expansion of the index size.

Pattern: Wildcards

If the trailing wildcard index is enabled to support queries like “Marklo*” or “Lindbl??”, the size of the index will be dependent on the average size of the words.

For trailing wildcards, every word longer than 3 characters will have many terms. An example: “adventure” will become:

adv*
adve*
adven*
advent*
adventu*
adventur*
adventure*

Using trailing wildcards, then, in conjunction with other index settings, can cause significant index expansion.

What if you need to support an arbitrary wildcard search like “Ma????gic”? In this case, the expansion is further amplified. Instead of trailing wildcards, you’d need to use the three-character index. The three-character index breaks each term up into trigrams; “Adventure” becomes:

adv
dve
ven
ent
ntu
tur
ure

To achieve both the best coverage to support an arbitrary wildcard requirement, and also the most efficient index settings, we recommend that the following indexes be configured:

- word searches
- three-character word searches
- word positions
- word lexicon in the codepoint collation
- three-character word positions

Use of either trailing wildcards or the three-character index can result in significant index expansion.

How to Properly Estimate Index Expansion Ratio

Compression, like index expansion, varies greatly based on the terms in the content being stored. Frequency and distribution, not term length, is the primary driver of compression. Index expansion together with term-list compression drives storage space requirements.

To compute the raw to on-disk storage ratio for any combination of content and index settings, ingest **at least 32GB of representative content**. After loading data, detach and re-attach the forest, which will force a checkpoint and will flush the in-memory stand to disk. Comparing the raw on-disk size to the size of the stand after data has been loaded will give you the compression or expansion ratio for *that combination of content and index settings*. If the content isn't representative of the content to be stored at scale or the index settings aren't identical to those used in production, the ratio will not hold.

Raw data size *is not* an indicator of on-disk size. Depending upon merge policies, also reserve some space for deleted fragments (fragments marked for deletion but not yet cleaned up by a merge).

Resource Summary for Indexes

System Resource	Impact
Storage	Index settings dramatically impact the storage footprint. When estimating overall storage costs, it is critical to understand index expansion for your use case. The easiest way to do this is to load sample content with the desired index settings.
I/O throughput	More index settings means more work to do at ingest time – which means potentially slower ingestion, depending on overall I/O capacity.
CPU	More index settings means more work to do at ingest time – which means more CPU utilization at ingest.
Memory	Range indexes and lexicons are memory mapped, so using these indexes will increase overall memory utilization.
Network	N/A

Part III: Queries

So now we have indexed data inside the MarkLogic database. It's time to ask some questions – and by that, we mean queries. Now we're going to get into the Evaluator services, so let's look at that part of our diagram again:

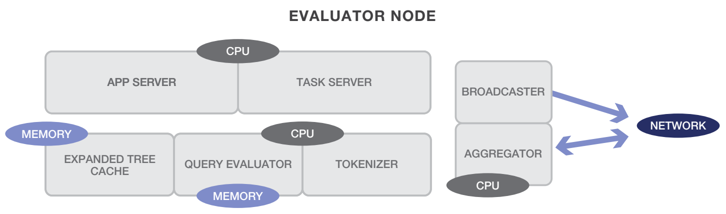


Figure 8: E-node

A client request comes in to MarkLogic Server to a port on the App Server. The App Server directs the request to the Query Evaluator that runs appropriate code as needed in the higher-level language: XSLT, XQuery, JavaScript, SPARQL or SQL. Think of the Query Evaluator as the mechanism that executes stored procedures or eval'ed queries on the database. If anything in those stored procedures needs data, then the Evaluator Node needs to go get the data from the forests and the Data Nodes. The Broadcaster will ask all the D-nodes for the data that it needs (example: I'm searching for documents with the word "MarkLogic"). In a cluster, this will be more than one host, so there will be a network cost. The D-nodes individually process the query request and send back compressed data to the Evaluator Node—at that point, the Aggregator uncompresses and collates all the responses and returns them back to the App Server and to the calling client.

There are all sorts of caches involved as the request makes its way through the cluster, and we'll want to understand those as well. Generally speaking, with queries, you'll get the fastest performance with warm caches, so what MarkLogic caches, and where they reside, is critically important to understanding query performance.

Query Processing: Filtered and Unfiltered Searches

30,000 Foot View

Resolving queries accurately – without false positives – typically depends on a filtering process to resolve the element in the query within the document. Filtered queries execute that process against the documents. Unfiltered queries can resolve to answers accurately by resolving the query directly against the indexes, assuming the correct index settings. Executing searches as "unfiltered" is an important performance optimization, particularly for large result sets. Filtered searches result in high I/O utilization and tend to be long running, particularly on large datasets.

Understanding Searches

When evaluating search expressions (and also when resolving XPath expressions), 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 search. Fragment IDs are ordered according to relevance criteria. This step is called index resolution.
2. Optionally, if specified in the query, 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 search, to determine if the fragment contains a result that matches the given search expression. This step is called filtering.

Index resolution happens on the D-nodes and filtering happens on the E-nodes.

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. The filter step removes false positives from the candidate list by opening each candidate document and verifying that the candidate matches the query.

Because filtering has a non-trivial performance cost for large datasets, consider using filtering only if the application use case is intolerant of false positives and tuning index settings is not enough to eliminate false positives.

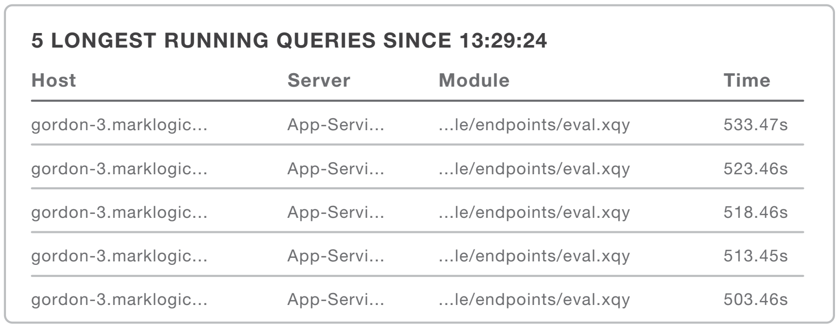
Disk I/O, CPU and Filtered Searches

Filtered searches, particularly searches that are unbounded (i.e. "find all documents containing the word 'tree'" instead of "find the first ten documents containing the word 'tree'") can have significant impact on disk I/O and on CPU. Disk I/O and network traffic will dramatically increase, as each candidate document is opened by the D-node and sent to the E-node. Similarly, CPU utilization will increase, as the E-node needs to do the work of validating that each candidate matches the query.

Mitigating the Impact of Filtered Searches

By default, `cts:search` runs *filtered*. So if a developer doesn't specify the "unfiltered" option to the search, the filter step will occur. This may not be ideal for applications that don't need the additional filtering, but developers may not be aware of the impact. For all other clients, including the Node.js and the Java clients, searches run *unfiltered* by default.

For a server under load, filtered searches can exhibit as long-running queries, particularly if an I/O bottleneck is generated by the work of opening and checking the documents for matches. Long-running queries can be observed in the Monitoring Dashboard.


5 LONGEST RUNNING QUERIES SINCE 13:29:24

Host	Server	Module	Time
gordon-3.marklogic...	App-Servi...	...le/endpoints/eval.xqy	533.47s
gordon-3.marklogic...	App-Servi...	...le/endpoints/eval.xqy	523.46s
gordon-3.marklogic...	App-Servi...	...le/endpoints/eval.xqy	518.46s
gordon-3.marklogic...	App-Servi...	...le/endpoints/eval.xqy	513.45s
gordon-3.marklogic...	App-Servi...	...le/endpoints/eval.xqy	503.46s

Figure 9: Long Running Queries

Filtered searches will also result in increased I/O utilization.

If false-positives are possible, given the index settings for an application, and if false-positives cannot be tolerated by the application, then filtered searches may be intentional and desired. If so, additional I/O and CPU capacity should be planned. In particular, as the database size grows, the impact of filtered searches on system resources should be measured and monitored.

Optimizing Caches for Performant Queries

30,000 Foot View

In a cluster, you'll have network traffic related to queries; queries enter the system by making a request via an App Server port – the request then gets farmed out to all the D-nodes and forests, and the response is returned back to the client. Each query can do the following:

- Run server side code (essentially stored procedures) which will use CPU
- Use I/O to open documents and check for term matches
- Use caches to resolve, which uses dynamic memory
- Use memory-mapped files to resolve

How and in what ratios these things happen depend in part upon the query. There are some general things that can help improve performance – making sure that caches are sized properly is a key component here.

Lifecycle of a Query

Let's look at the big picture again:

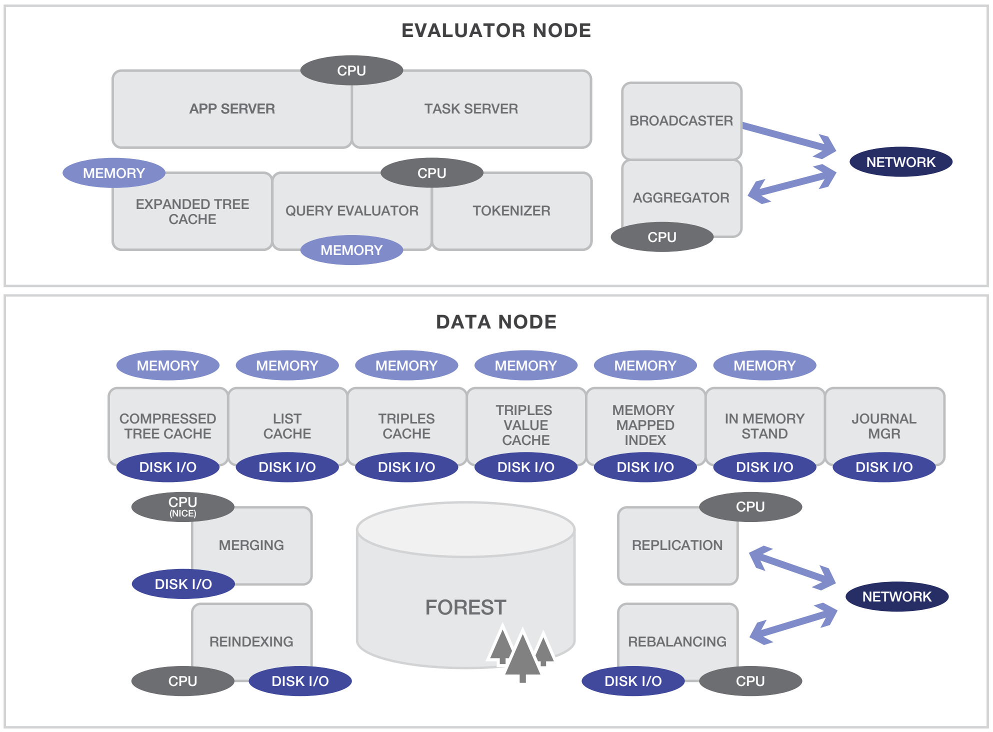


Figure 10: E- and D-nodes

Our query comes into the App Server, and then the Query Evaluator executes whatever server-side code needs to run. If that query requires data, the Broadcaster sends the request out to all the D-nodes that host forests currently enabled for that database. (If hosts are configured as shared E/D-nodes, then this usually means all the nodes in the cluster, depending upon which nodes host forests for that database.) The request then hits the D-node, and several things happen. If the request is simply "return me this document," the D-node will check to see if it has that document in any of the local forests.

If it does, it will check the Compressed Tree Cache to see if the document is there. If it is there, the D-node just sends that in-memory document back to the E-node. The E-node checks the Expanded Tree Cache to see if the document is already in-memory. If not, the E-node expands that in-memory compressed tree data into the full XML or JSON and places the fully expanded tree into the Expanded Tree Cache on the E-node. The document is then returned to the client via the App Server port. Meanwhile, all the other D-nodes in the cluster are responding to the Aggregator, saying, “No, I don’t have that document.”

But what if the query is substantially more complex than just “Give me this document”? What if we want to find all documents that contain the word “apple” and have a PUBDATE before January 1, 1970? Then things get a little more complex.

The same Broadcaster request goes out to the D-nodes, but this time the D-nodes need to look in the List Cache to see if there are termlist matches. If not, we’ll need to check the termlist files on disk. This query also happens to use a range index, so the D-node will use the memory mapped index to figure out if there are any documents that meet that criteria. We might also be doing processing using semantic triples, in which case we’d hope that our doc IDs would be in our triples values and triples caches (otherwise, we’ll read from disk); and we might be using lexicons, which means we’ll want to look back in those memory-mapped indexes. Since memory will always be faster than disk (at least for now, although affordable SSD cards might change this in the future), we want to make best use of our available caches.

It’s also important to note that the in-memory stand is part of all this query processing – the documents in the in-memory stand are included in the result set if they match the query.

Tuning MarkLogic Caches

On database creation, MarkLogic assigns default cache sizes optimized for your hardware, using the assumption that the server will be acting as both E-node and D-node.

Depending upon your topology, you may want to modify the cache sizes. For instance, if your cluster has separate E- and D-nodes, you can improve performance by optimizing each group's cache sizes. With an E-node group, increase the caches related to request evaluation, and decrease those related to data management. For a D-node group, do the opposite. More details can be found in *"Part IV: Operational Concerns" on page 26*. Like most database tuning, it is very important to have a representative set of data, queries and load on representative hardware. Effects at scale, particularly with caches can be pronounced and unexpected when tuning is not performed with full-scale data, query volume and hardware.

Cache Partitions

Along with setting each cache size, you can also set a cache partition count. Each cache defaults to one, two or sometimes four partitions, depending on your memory size. Increasing the count can improve cache concurrency but will reduce the size available for caches by providing more, smaller caches. More threads can access the cache, but there will be less space in each partition.

An excessive number of partitions reduces the efficiency of the cache, so generally these need to be kept to a smaller number. Each cache partition has to manage its own aging out of entries, and can only elect to remove the most stale entries from itself, even if there's a more stale entry in another partition.

List Cache

This cache holds term lists after they've been read off disk. Since index resolution happens on D-nodes, in a separated E/D environment, this cache should be bigger on D-nodes and smaller on E-nodes.

Compressed Tree Cache

This cache holds the binary representation of documents after they've been read off disk. They're stored compressed, to reduce space and improve I/O efficiency. Reading documents off disk is solely a D-node task; in a separated E/D environment, this cache should be bigger on D-nodes and smaller on E-nodes.

Expanded Tree Cache

Each time a D-node sends an E-node a fragment over the network, it sends it in the same compressed format in which it was stored. The E-node then expands the fragment into a usable data structure. The Expanded Tree Cache stores the expanded tree instances.

You'll want to raise the Expanded Tree Cache size on E-nodes and greatly reduce it on D-nodes. Why not reduce to zero? D-nodes need their own Expanded Tree Cache as a workspace to support background reindexing. Also, if the D-node group includes an admin port on 8001, which is a good idea in case you need to administer the host directly should it leave the cluster, it needs to have enough Expanded Tree Cache to support the administration work.

Triple Cache and Triple Value Cache

These caches hold the references and values for triples. If triples are not in use, these caches can be decreased. Keep in mind that many internal MarkLogic databases, such as Meters, use triples so changing triples cache sizes for those databases is not recommended.

Caching Binary Documents

Large and external binary documents receive special treatment regarding the tree caches. Large binaries (those above a configurable size threshold, and thus managed in a special way on disk by MarkLogic) go into the Compressed Tree Cache, but only in chunks. The chunking ensures even a massive binary won't overwhelm the cache. These binaries never go into the Expanded Tree Cache. There's no point, as the data is already in its native binary form in the Compressed Tree Cache.

Note that external binaries are pulled in from the external source by the E-nodes.

This means a system with external binaries should make sure the E-node has a sufficiently large Compressed Tree Cache.

Monitoring Cache Utilization

How well are your caches being used? The answer to this question can lead you to changes that can result in dramatic performance improvements. This section will show you how to monitor cache utilization and how to make changes to improve cache usage.

Cache utilization can be monitored using the Admin API or Admin console. The Expanded Tree Cache can also be monitored using the Monitoring Dashboard.

Use the Admin API function `xdmp:cache-status` to get information about the cache hit/miss/status. Although this API call can be resource intensive (so use it judiciously), it provides information about cache usage and partitions. The values returned are per host, defaulting to the current host. The function takes an optional host-id to allow you to gather values from a specific host in the cluster.

The output of `xdmp:cache-status` will look something like this:

```
<cache-status xmlns="http://marklogic.com/xdmp/status/cache">
   <host-id>18349804367231394552</host-id>
   <host-name>macpro-2113.local</host-name>
   <compressed-tree-cache-partitions>
     <compressed-tree-cache-partition>
       <partition-size>512</partition-size>
       <partition-table>0.2</partition-table>
       <partition-used>0.8</partition-used>
       <partition-free>99.2</partition-free>
       <partition-overhead>0</partition-overhead>
     </compressed-tree-cache-partition>
   </compressed-tree-cache-partitions>
   <expanded-tree-cache-partitions>
     <expanded-tree-cache-partition>
       <partition-size>1024</partition-size>
       <partition-table>0.7</partition-table>
       <partition-busy>0</partition-busy>
       <partition-used>30.4</partition-used>
       <partition-free>69.6</partition-free>
       <partition-overhead>0</partition-overhead>
     </expanded-tree-cache-partition>
   </expanded-tree-cache-partitions>
   <list-cache-partitions>
     <list-cache-partition>
       <partition-size>1024</partition-size>
       <partition-table>0.2</partition-table>
       <partition-busy>0</partition-busy>
       <partition-used>0</partition-used>
       <partition-free>100</partition-free>
       <partition-overhead>0</partition-overhead>
     </list-cache-partition>
   </list-cache-partitions>
   <triple-cache-partitions>
     <triple-cache-partition>
       <partition-size>1024</partition-size>
       <partition-busy>0</partition-busy>
       <partition-used>0</partition-used>
      <partition-free>100</partition-free>
     </triple-cache-partition>
   </triple-cache-partitions>
  <triple-value-cache-partitions>
     <triple-value-cache-partition>
     <partition-size>512</partition-size>
       <partition-busy>0</partition-busy>
       <partition-used>0</partition-used>
       <partition-free>100</partition-free>
     </triple-value-cache-partition>
   </triple-value-cache-partitions>
 </cache-status>
```

- `partition-size`: The size of a cache partition, in MB
- `partition-table`: The percentage of the table for a cache partition that is currently used. The table is a data structure that has a fixed overhead per cache entry, for cache admin. This will fix the number of entries that can be resident in the cache. If the partition table is full, something will need to be removed before another entry can be added to the cache
- `partition-busy`: The percentage of the space in a cache partition that is currently used and cannot be freed
- `partition-used`: The percentage of the space in a cache partition that is currently used
- `partition-free`: The percentage of the space in a cache partition that is currently free
- `partition-overhead`: The percentage of the space in a cache partition that is currently overhead

The `partition-busy` value is the most useful indicator of getting a cache-full error. It tells you what percent of the cache partition is locked down and cannot be freed to make room for a new entry.

In addition to `xdmp:cache-status`, which gives you utilization statistics, you can also monitor cache hits and misses using the Monitoring Dashboard. The Expanded Tree Cache hits/misses is available in the Monitoring History dashboard, and details on the List, Compressed, and Triple store caches are available either through the Admin UI or by calling xdmp:forest-status.

When do hits/misses matter? For the Expanded Tree Cache (ETC), depending upon your application, maybe not that much. If you have lots of misses in the ETC that means that you are getting a large number of unique documents back. The same applies to the Compressed Tree cache. But if you're getting lots of misses on the List Cache or the triples caches (assuming a use of the semantics features), *then* you may well want to increase those cache sizes or add RAM and then increase the cache sizes.

Mitigation and Remediation

It's a good idea to monitor error logs for cache errors. Useful keywords include: `notice`, `error`, `exception`, `SVC-`, `XDMP-`, & `start`. Over time, you may want to refine the keywords, but these may indicate that something is wrong.

Cache full errors may indicate a resource issue. Caches are shared by all concurrently running queries, so the occasional single cache full error indicates that a query retried and succeeded. It doesn't require any immediate action but it is an indicator that capacity limits may be approaching and so some capacity planning is in order. Frequent cache full errors do indicate a serious problem and may be significantly impacting the performance of a cluster, so frequent errors should be treated as an urgent issue.

Expanded Tree Cache Full Errors (XDMP-EXPNTREECACHEFULL)

The error message `XDMP-EXPNTREECACHEFULL: Expanded tree cache full` means that MarkLogic has run out of room in the expanded tree cache during query evaluation, and that consequently it cannot continue evaluating the complete query.

Possible solutions:

- Change the problem query so that it does not need to use as much data. For instance, return only 100 results instead of 500
- Tune the problem query so that it does not need to simultaneously cache as much content. An example: you might be returning 500 complete documents when all you really want is the value of a single element in each document. Better usage of the indexes and queries can help here
- Alternatively, if there is not sufficient memory to increase total cache size, you can increase the size of the cache partitions by decreasing the number of partitions. More partitions allow more concurrency, but make each individual cache partition smaller, which could make it more likely for the cache to fill up

List Cache Full Errors (XDMP-LISTCACHEFULL)

MarkLogic Server uses its List Cache to hold search term lists in-memory. This error occurs when a query fails due to the size of the search term lists exceeding the allocated List Cache.

Very occasional instances of `XDMP-LISTCACHEFULL`, especially during periods of high concurrency, are not generally critical problems since queries that fail will retry and succeed. Frequent instances of `XDMP-LISTCACHEFULL` indicate an unhealthy system, where some combination of queries is failing frequently, and remedial action should be taken to either resize the cache or tune queries.

Possible Solutions:

- **Increase cache size.** If memory is available on a host, increasing the cache size may be a good option.
- **Add hosts.** This would increase overall cache size by adding per-host memory.
- **Simplify data model to reduce the size of termlists.**
- **Reduce the number of List Cache partitions,** as long as doing so would increase the size of each partition. Note that this may reduce concurrency and shouldn't be the first action.

Resource Consumption Summary for Queries

System Resource	Impact
Storage	N/A
I/O throughput	Filtered searches can result in very high I/O utilization because the E-nodes will need to open each document and inspect for matches. If the caches are not efficiently used, you will also see high I/O utilization on D-nodes.
CPU	CPU will be used on E-nodes to execute server-side code (XSLT, JavaScript and/or XQuery). D-nodes will also have some CPU utilization – in particular, positions processing (near queries) has significant CPU impact.
Memory	Memory will be allocated dynamically for caches – so all cache sizes will impact overall memory. As documents are fetched from disk and loaded into caches, memory utilization will increase.
Network	Depending upon the size of the cluster, network traffic can be substantial (in the case of 50 or greater hosts) or small (1-3 hosts). Query workload can also impact network – if queries are requesting large numbers of documents, this can impact network. Network should be monitored; if network throughput becomes a bottleneck, a 10Gig E can alleviate network concerns.

Part IV: Operational Concerns

There are a number of operational concerns that don't fit neatly into the "ingest or query" model. Database backups, intracluster communication and topology decisions are some examples. This section addresses operational concerns and makes recommendations based upon best practices.

Optimal Number of Forests

Forests are a unit of concurrency in MarkLogic Server: more forests means more concurrent writes and reads. However, at some point, concurrency benefits will plateau.

There are also many different factors at play when determining the optimal number of forests for an application:

- How large are the documents?
- How many documents are there?
- What is the query and update frequency?
- How many concurrent users are expected?
- What are the query response time and throughput requirements?
- Is tiered storage in place and do you need to align forests with partitions?
- How many forests are optimal from a manageability perspective?

We always recommend using more than one forest. Beyond that, it makes sense to decide on number of forests depending upon the factors listed above. For example, if response times need to meet a certain SLA and the application uses large numbers of facets, the number of documents in a forest should be kept on the smaller side, and therefore more forests may be needed.

Topology Decisions: E-nodes and D-nodes in a Cluster

E-nodes communicate with D-nodes to retrieve data. If a request does not need any forest data to complete, then an E-node request is evaluated entirely on the E-node. If the request needs forest data (for example, a document in a database), then it communicates with one or more D-nodes to service the forest data. Once it gets the content back from the D-node, the E-node finishes processing the request (performs the filter portion of query processing) and sends the results to the application.

In single host configurations, a single host carries out both E-node and D-node activities. In a cluster, it is also possible for some or all of the hosts to have shared E-node and D-node duties.

E-nodes and D-nodes run exactly the same MarkLogic software – the separation is logical and functional. For an E-node, configure the host with application servers, but no forests. For a D-node, configure with forests, but no application servers.

When to Separate

Separating nodes into D- and E-nodes provides clear workload isolation. However, either shared E/D nodes or separated D- and E-nodes are valid configurations. In general, separating E- and D-nodes in larger scale clusters remains an easier way to monitor resource consumption and understand bottlenecks, but many successful MarkLogic customers run in production with large-scale (greater than 100) nodes using a combined E/D approach.

There are some clear cases where separation or combined E/D make sense.

Analytic Applications/Semantics

For analytic applications where workload isolation is required, use separate E- and D-nodes. Semantics applications using SPARQL or MarkLogic built-ins to do semantic queries should be considered analytic applications; semantics applications should use separate E- and D-nodes. Semantics E-nodes need a significant amount of memory to execute in-memory joins – a minimum of 64 GB of RAM is recommended for E-nodes in semantics applications.

More Data, Fewer Requests

Some applications have a great deal of data but very small numbers of concurrent requests. In this case, it makes sense to separate nodes and allocate many more nodes as D-nodes.

Less Data, More Requests

Other applications might have a small amount of data but large numbers of concurrent requests. In this case, separating into E- and D-nodes makes sense, with more hosts allocated for E-nodes.

Arbitrary Queries

Another use case for splitting nodes is an application that runs arbitrary queries (either via Query Console or via the application API). This can cause significant E-node utilization and so it might make sense in this case to separate E- and D-nodes.

The Gray Area

For applications where these scenarios don't apply, understanding the pros and cons of each approach is important. In general, once a cluster has reached about 16 nodes, it makes sense to start separating E- and D- nodes, *unless there is a compelling reason (financial, logical, functional) not to*. Again, both types of configurations are valid. This is an enterprise infrastructure decision: given your particular resources, which configuration makes sense?

Beyond "Query Evaluation" or "Data Services"

Understanding the workload managed by each type of node can also be part of the decision to split or run with combined E/D nodes.

Functions Impacting E-node Workload

- **Snippets**. Snippeting code needs to run on the E-nodes and can have significant impact on memory and CPU. To create a snippet, the entire document is read into memory and then parsed to look for similar terms.
- **Server-side XQuery or JavaScript code that does NOT involve database calls.** All server-side code is evaluated on the E-node. In many cases, this code will need to manipulate data that has been returned or run reports. All of these functions consume CPU.
- **Filtering.** For some applications, filtering search results is a requirement. The filtering step 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. If a search is filtered, the E-node will expand the results from disk if the document is not already in cache, and then find a match; filtering can therefore significantly increase I/O and CPU utilization.
- **Expanding tree data.** When reading from the database, the E-node receives the compressed tree binary data. The E-node then expands the binary data into the Expanded Tree representation (either XML or JSON) which is eventually returned to the calling function.
- **Determining which forest to insert into.** When data is loaded into MarkLogic Server, the E-node is in charge of determining what forest to load content into, based upon the database assignment policy. Note that this step is skipped if using mlcp (MarkLogic Content Pump) with the fastload option.
- **Tokenization.** When documents are loaded, the E-node breaks the document up into tokens.
- **Task Server.** If using the Content Processing Framework (CPF), tasks run on the E-node task server.
- **Semantics joins and semantics aggregations** (not cts:aggregate functions)**.** All SPARQL join statements happen on the E-node.
- **SQL joins and aggregations.**

Functions Impacting D-node Workload

- **Data reads.** When data is requested from the D-node, the data is loaded from disk into the Compressed Tree Cache (if not already there) and then sent to the E-node.
- **Data writes.** The D-node is responsible for writing data to the local forest.
- **Index resolution.** The D-node is responsible for resolving searches against local indexes and returning results to the E-node. Queries against all indexes, including lexicons, happen here.
- **Rebalancing.** All rebalancing work happens on the D-node. The E-node is not at all involved. This can increase CPU and I/O utilization.
- **Reindexing.** When data is reindexed, the I/O, CPU, and memory required are all used on the D-node.

Separate E- and D-nodes: Pros

- **Workload isolation.** E-nodes are stateless, so if a query consumes a lot of E-node resources (lots of filtering, co-occurrence, complex code, etc.) then having it run on an E-node isolates that from other queries that need to access data that would otherwise be on that node.
- **Ability to leverage available machines/VMs that lack disk bandwidth**
- **Ability to dedicate CPU resources to evaluation separately from data/ingest**
- **If an E-node machine becomes unavailable for some reason** (for example due to poorly performant ad-hoc queries), **it won't cause a failover or bring down the whole cluster** (if failover isn't enabled)
- **Separating E-nodes makes it quicker to unmount or restart forests,** which can be important during state transitions like entering flash backup

Separate E- and D-nodes: Cons

- **Separation of concerns in a bad way:** you can't use available E-node CPU or RAM to help on D-nodes or vice versa
- **Disk bandwidth now even more focused on D-nodes,** which may reduce total disk bandwidth available depending on infrastructure

Using Performance Monitoring to Understand Constraints

Whether your implementation uses shared D/E nodes or separated D- and E-nodes, using the Monitoring Dashboard and Monitoring History can help you understand bottlenecks and what to do next.

Example: Low Disk Utilization, High CPU Utilization
In this example, the disks are moderately utilized under load (ingest and query), but the CPU is at 75% utilized. It's likely that CPU resources will be exhausted in the near future, since you plan to roll the application out to additional users.

Remediation:

- Tune code. Is there a way to make better use of MarkLogic caches and reduce E-node operations?
- Add E-nodes/cores. Adding additional capacity should alleviate the overall CPU bottleneck
  - If the configuration uses combined E/D nodes, this particular problem will leave disk cycles unused

Example: High Disk Utilization, Low CPU Utilization
In this example, the disks are maxed out under load (ingest and query), but the CPU is quite low. Customers are complaining about slow response times and you're missing your SLAs.

Remediation:

- Improve underlying storage
  - May require deep dive into storage infrastructure, RAID controllers, SAN config
- Add hosts. More storage may be the best solution here; if using local disk forests, you will need to add more hosts to increase storage. Note that this would also suggest the need for an E/D separation – you'd want to have separate D-nodes to use all available resources for data or you'll be adding CPU resources that you don't need.

Summary

Overall, there may be important reasons to use shared E/D nodes, but as your cluster grows, using separate E- and D-nodes will make diagnosing problems and cluster management somewhat easier.

Full and Incremental Backups

Full backup/restore operations are I/O intensive and should be scheduled during off-hours, when possible. Backup is a background operation and can be throttled. Backup I/O competes with background Merge I/O for throughput and cannot be prioritized except through scheduling backups and merge blackouts.

The backup/restore operations with journal archiving enabled provide a point-in-time recovery option that enables you to restore database changes to a specific point in time between full backups with the input of a wall clock time. When journal archiving is enabled, journal frames are written to backup directories by near synchronously streaming frames from the current active journal of each forest.

When journal archiving is enabled, you will experience longer restore times and slightly increased system load as a result of the streaming of journal frames.

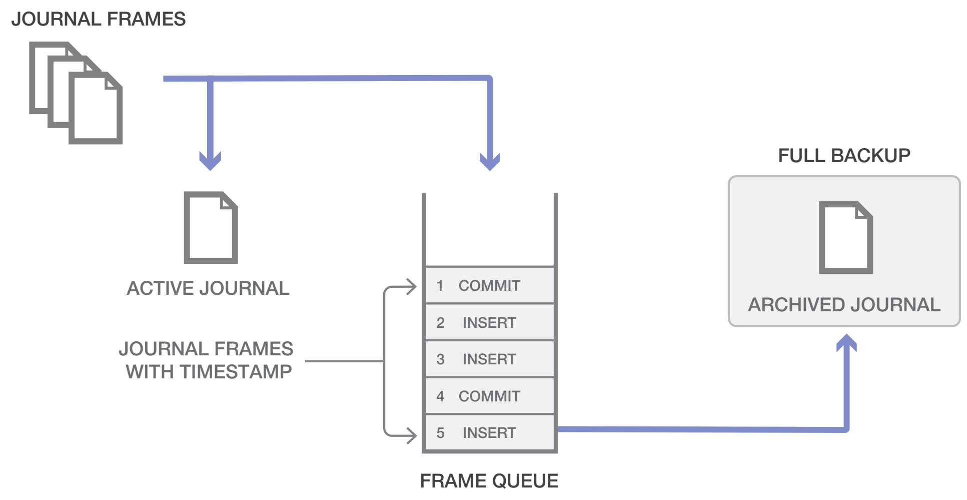


Figure 11: Journal Archiving

Journal archiving can only be enabled at the time of a full backup. If you restore a backup and want to re-enable journal archiving, you must perform a full backup at that time.

When journal archiving is enabled, you can set a lag limit value that specifies the amount of time (in seconds) in which frames being written to the forest's journal can differ from the frames being streamed to the backup journal. For example, if the lag limit is set to 30 seconds, the archived journal can lag behind a maximum of 30 seconds worth of transactions compared to the active journal. If the lag limit is exceeded, transactions are throttled until the backup journal has caught up.

The active and backup journal are synchronized at least every 30 seconds. If the lag limit is less than 30 seconds, synchronization will be performed at least once in that period. If the lag limit is greater than 30 seconds, synchronization will be performed at least once every 30 seconds. The default lag limit is 15 seconds.

The decision on setting a lag limit time is determined by your Recovery Point Objective (RPO), which is the amount of data you can afford to lose in the event of a disaster. A low RPO means that you will restore the most data at the cost of performance, whereas a higher RPO means that you will potentially restore less data with the benefit of less impact to performance. In general, the lag limit you choose depends on the following factors:

A lower lag limit implies:

- Accurate synchronization between active and backup journals at the potential cost of system performance
- Use when you have an archive location with high I/O bandwidth and your RPO objective is low

A higher lag limit implies:

- Delayed synchronization between active and backup journals, but lesser impact on system performance
- Higher server memory utilization due to pending frames being held in-memory
- Use when you have an archive location with low I/O bandwidth and your RPO objective is high

Understanding Reindexing and Rebalancing

Reindexing

Indexing in MarkLogic is usually accomplished when loading a document. However, if you wish to change the index settings after you've already ingested content, you'll need to reindex the existing documents if you wish them to pick up the new index settings. When adding a new index, the server runs an estimate of all the fragments that match and then proceeds to reload those URIs that match.

As a result, indexing/reindexing can be a very CPU and disk-I/O intensive operation that creates a lot of new documents, which then need to be merged. If the system doesn't have adequate I/O throughput capacity, all of this activity can hurt query performance.

For range indexes, range queries cannot execute at all until reindexing is complete; in this case, if the new index is needed faster, then the reindexing process should be given priority.

Mitigating Reindexing Impact

There are several steps that can be taken to mitigate reindexing impact. These steps are ordered; it makes the most sense to first lower priority and throttle. If that isn't sufficient, then scheduling during slower times may be better.

Configure Throttle and Background I/O

Reindexing can lead to heavy merge activity and may lead to disk bottlenecks if not managed carefully. To reduce the impact of merges on queries, you should consider throttling the reindexer in order to keep the I/O to a minimum. For example, use the following settings:

```
reindexer-throttle = 3
```

Make sure to leave the large-size-threshold at default or larger to prevent binary documents from being indexed.

You can also adjust the following group settings to help limit I/O:

- `background-io-limit = 100`

This will limit the background I/O for that group to 100 MB/host. This is a good starting point, and may be increased if you find that your merges are progressing too slowly. If you are I/O limited, make sure to set both the reindexer-throttle and background I/O to balance priorities between reindexing and merges.

Schedule During Slower Times

For non-urgent index changes, it may make sense to schedule reindexing during a time when your cluster is not busy. For a production system, when new range indexes might be needed to support application changes, this would not be a good practice and reindexing should be given priority. But for non-urgent production changes, it may make sense to work through a gradual reindexing process. For instance::

- Change your index configuration on a Friday night
- Let it run for most of the weekend
- Set the `reindexer-enable` field to 'false' for the database being reindexed
- Be sure to do the previous step a few hours before your cluster begins to see heavy usage. This will allow the merging to complete in a timely fashion
- If need be, reindexing can continue over the next weekend and so on. The reindexer process is able to pick up where it left off before it was disabled

Avoid Unused Range Indexes, Fields, and Path Indexes

In addition to taking up extra disk space, the above indexes require extra work when it's time to reindex. This is especially true for Field and Path indexes, which may require extra loading passes.

Rebalancing

Rebalancing does not run as background I/O. It uses the assignment policy for the database to decide where to place documents, and all operations happen on the D-node. Note that this is similar to ingestion – so documents are inserted, marked for deletion, and stands are merged. This can have a significant impact on disk and CPU.

Using the rebalance throttle and enabling rebalancing at times when the cluster is not busy can mitigate rebalancing impact.

Summary

Backups, reindexing and rebalancing all use significant resources, particularly CPU and I/O. Plan for these activities carefully and use tools available to throttle to reduce impact on your live query traffic.

Memory Utilization in MarkLogic Server

There are a number of components and data structures that use memory in MarkLogic:

- **Range Indexes and Lexicons.** These are memory mapped and are loaded either at forest mount time (preload mapped data) or dynamically as needed.
- **Caches.** Memory for MarkLogic caches is allocated at start time.
- **In-memory stand.** The in-memory stand and associated in-memory structures is allocated as needed when documents are ingested. When the stand is flushed to disk, memory is released.
- **Queries – semantic joins.** SPARQL queries use in-memory joins to execute.

MarkLogic and Huge Pages

MarkLogic uses Huge Pages to optimize memory utilization. MarkLogic will try to utilize Huge Pages in some cases where it needs to allocate large blocks of memory, such as allocating memory for the group level caches and in-memory stands.

In addition to this, MarkLogic will also try to use Huge Pages for range index tables for inversions. It can also allocate Huge Pages for large termlist and fragment read buffers.

All range indexes map values to corresponding fragment ids; the range index inversion performs the inverse mapping (mapping fragment ids to values). When MarkLogic first needs to map a fragment id to a value, it generates this inversion table from the mapped range index files. Once this table is created, it is held in-memory until the corresponding stand is closed -- so the upfront cost of this work is offset through reuse.

Note that queries that make use of Huge Pages may not be the same queries that are utilizing a large amount of anonymous memory; many queries will run through without allocating any Huge Pages and instead will rely on allocating large numbers of small blocks of memory.

MarkLogic will never fail if it is unable to allocate Huge Pages; anonymous pages will always be allocated in this situation and this is why MarkLogic recommends using either of the two values output by MarkLogic when the node first starts up.

Configuring Huge Pages

When MarkLogic starts up, the server logs a suggested low and high range for Huge Pages:

```
ErrorLog_1.txt:2015-06-16 11:36:44.949 Info: Linux Huge Pages: detected 0, recommend 9760 to 10816
```

The low number is the sum of all cache sizes. The high number also includes in-memory settings for forests. If forest configuration settings change, this can cause changes to the recommended Huge Pages.

This KnowledgeBase article explains how to configure Huge Pages. *Be especially careful to DISABLE Transparent Huge Pages and use Huge Pages instead.*

RedHat Enterprise Linux, CentOS, and Suse Linux provide support for Huge Pages. Amazon Linux, Windows Server and MacOS do not provide any support for Huge Pages.

Tiered Storage

MarkLogic Server allows you to manage your data at different *tiers* of storage and computation environments, with the top-most tier providing the fastest access to your most critical data and the lowest tier providing the slowest access to your least critical data. Infrastructures such as Hadoop and public clouds make it economically feasible to scale storage to accommodate massive amounts of data in the lower tiers. Segregating data among different storage tiers allows you to optimize trade-offs among cost, performance, availability, and flexibility.

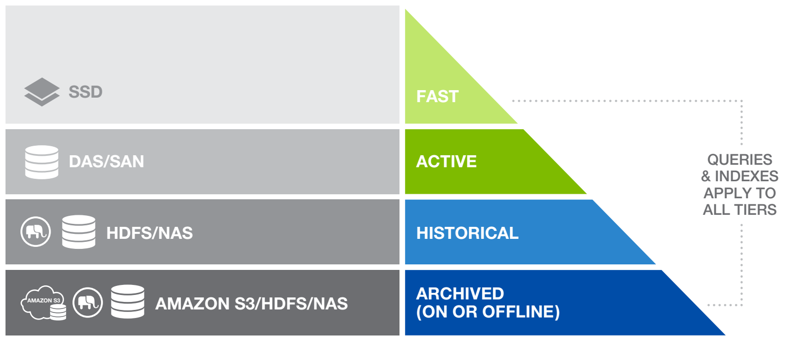


Figure 12: Tiered Storage

In the example above, the data has been organized into four categories: fast, active, historical and archived. Depending upon the importance of the data (as defined by application requirements), the data is partitioned into one of these categories. Each category has its own hardware, corresponding to how quickly queries need to return. “Importance” in this context depends upon application requirements; for example, an application that mines social media will consider content freshness to be most important, whereas a different application might consider most frequently accessed content to be the most important.

High Availability

Local-disk Failover

Local-disk failover creates one or more replica forests for each failover forest. The replicas contain the exact same data as the primary forest, and are kept up to date transactionally as updates to the forest occur. Each replica forest should be on a different host from the primary forest so that, in the event the host for the primary forest goes down, another host with a copy of the primary forest's data can take over.

Each forest has its own host, and each host has disk space allocated for the forest. The primary forest is the forest that is attached to the database. Any replica forests configured for the primary forest have their own local disk. As updates happen to the database, the primary forest is updated as well as each configured replica forest.

In the event that Host1 goes down or an I/O error occurs on the F1 forest, the MarkLogic Server cluster will wait until the specified timeout and then it will automatically remove the F1 forest from the cluster. Because all of the forests in a database must be available for queries to run, the database is unavailable at this time. At this point, the system fails over the forest to the first available replica forest, and the Replica1 forest is automatically attached to the database. The database once again becomes available. If the failed forest comes back online, it will resume as a replica.

While each replica forest maintains all of the committed documents that are in the primary forest, it is not an exact byte-for-byte copy of the forest. It is a forest in its own right, and will merge when it needs to merge, which might not be at the same times that the primary forest merges. So assuming the replica forest is replicating normally (*sync replicating*, as described in Forest Mount States), it will contain the same committed documents as the primary forest. The host that services each replica forest must meet the needed disk space requirements and be sized appropriately so that merges can occur as needed on the forest.

Remember that memory utilization on the D-node might vary greatly after a failover and you should size accordingly. For example, if preload is turned off for range indexes, a host that properly served 6 primary forests and 6 failover forests could find itself with inadequate memory when it's serving 9 primary forests and 3 failover forests after a node failure. Likewise, those failover forests might not have impacted cache utilization on that host before the failover but once active, are consuming cache resources. When load testing, testing during failover is recommended also.

Shared-disk Failover

Shared-disk failover uses a clustered filesystem to store the forest data. The clustered filesystem must be available with the same path on each host that is configured as a failover host. In the event of a host that is assigned a shared-disk failover forest going down, another host can take over the assignment of that forest.

After the cluster has determined that a host is down and disconnected it from the cluster, if failover is enabled and configured for that forest, then one of the failover hosts will attempt to mount the forest locally. The host that attempts to mount the forest is determined based on the list of failover hosts in the forest configuration, and the host that is highest on the list will be the first to attempt to mount the forest. If that host is not available, then the next one in the list will attempt to mount the forest, and so on until the forest is either mounted or there are no failover hosts available. After the forest is mounted locally, the other hosts in the cluster will mount it remotely.

If the forest on Host2 comes back online, it will automatically connect to the cluster. However, Host1 will continue to host the primary forest until the forest is restarted. This avoids having the forest 'ping pong' between hosts in the event that the primary host has a recurring problem that takes some time to solve.

In local disk failover, each copy of the forest has its own in-memory and on-disk stands. In shared-disk failover, the in-memory stand is not available to the failover host after the primary host has failed. In order to populate the in-memory forest, the journal has to playback from the time of the last complete in-memory stand write.

Choosing Between Local-disk and Shared-disk Failover for HA

Advantages of Local-disk Failover

- Typically higher performance throughput at lower cost
- Typically fewer points of failure
- More rapid failover

Disadvantages of Local-disk Failover

- Increased disk usage
- More forests to manage (2X)
- Additional CPU capacity and memory to update and merge the forest, even when the forest is not failed over

Advantages of Shared-disk Failover

- There is only a single filesystem to manage
- It takes advantage of the high-availability features of a clustered filesystem.
- Can be used with forest data directories mounted with Network File System (NFS)

Disadvantages of Shared-disk Failover

- Typically slower performance for everyday usage and failover
- Typically more complex to configure and monitor
- Typically more expensive than local disk to acquire and manage
- Shared disk failover does not keep multiple copies of the forest on different filesystems (although clustered filesystems tend to have redundancy built in). This means failover will only work if the host that has the forest mounted locally fails or if the XDQP communication between the host and disk is interrupted. Unlike local-disk failover, if the shared forest fails due to a disk failure, there is no backup forest
- Forests that are configured for shared-disk failover always perform their updates using strict journaling mode, which explicitly performs a file synchronization after each commit. While strict journaling is safer because it protects against the computer unexpectedly going down and not just against MarkLogic Server unexpectedly going down, it makes updates slower

Clusters and Network Communication

As the number of hosts in a cluster grows, network communications become more important in understanding resource consumption patterns. This section describes network communication.

Communication Between Hosts

A cluster is a group of hosts running MarkLogic Server; a cluster is configured at installation time. Each host in a cluster communicates with all of the other hosts in the cluster at periodic intervals. This periodic communication, known as a heartbeat, circulates key information about host status and availability between the hosts in a cluster. Through this mechanism, the cluster determines which hosts are available and communicates configuration changes with other hosts in the cluster. If a host goes down for some reason, it stops sending heartbeats to the other hosts in the cluster.

If a host stops sending heartbeats, the other hosts in the cluster decide whether to vote the non-responsive host out of the cluster. To vote a host out of the cluster, there must be a *quorum* of hosts voting to evict a host. A quorum occurs when more than 50% of the *total* number of configured hosts in the cluster (including any hosts that are down) vote the same way. Therefore, you need at least 3 hosts in the cluster to reach a quorum.

The voting that each host performs is done based on how long it has been since it last had a heartbeat from the other host. If more than half of the hosts in the cluster determine that a host is down, then that host is disconnected from the cluster. If failover is configured, the replica forests for that host then take over so that availability is maintained.

Each host in the cluster continues listening for the heartbeat from the disconnected host to see if it has come back up, and if a quorum of nodes in the cluster is getting heartbeats from the node, then it automatically rejoins the cluster.

The heartbeat mechanism enables the cluster to recover gracefully from things like hardware failures or other events that might make a host unresponsive. This occurs automatically; hosts can go down and automatically come back up without requiring intervention from an administrator (although failback does require manual intervention). If the host that goes down stores content in a forest, then the database to which that forest belongs goes offline until the forest either comes back up or is detached from the database. If you have failover enabled and configured for that forest, it attempts to fail over the forest to a secondary host (that is, one of the secondary hosts will attempt to mount the forest). Once that occurs, the database will come back online.

Communication Between Clusters
While local-disk failover replication happens within a cluster, replication of data to a completely separate cluster for disaster recovery purposes (called Database Replication) uses inter-cluster communication. Communication between clusters uses the XDQP protocol. Before you can configure database replication, each cluster in the replication scheme must be aware of the configuration of the other clusters. This is accomplished by coupling the local cluster to the foreign cluster. For more information, see the Database Replication Guide.

Tuning Application Server Threads and Backlog

MarkLogic Application Servers have several parameters that can be tuned to improve performance. These settings are specific to the E-node and control the HTTP or XDBC requests handled by the server.

- Threads: specifies maximum number of Application Server threads
- Backlog: specifies the maximum number of pending connections on the HTTP server socket

Although the defaults are a good starting point, different applications may need different settings depending upon whether E- and D-nodes are separated and whether additional E-node resources are available on the host.

To tune these settings, first run a performance test with the defaults and measure the response time and throughput. Then, increase the number of threads. When the number of threads is low, the server should be able to provide the fastest response time. When the number of threads is increased, the response time for individual requests may increase, but the throughput should increase as well.

Using the Monitoring History, observe available system resources (CPU, memory, disk bandwidth). As the number of threads is increased, usage of system resources should increase; the goal is to find the ideal number of threads before system resources are exhausted.

Increasing the backlog is a way to queue requests without incurring significant additional resource cost. Adjusting the backlog may provide response times that are equivalent (or better) than what users experience with a high thread count.

The backlog should be chosen to queue about one second of connection requests, or roughly the number of requests per second expected on that App Server.

Swap Recommendations

We strongly recommend configuring swap space on Linux as a conservative measure against memory exhaustion. If the MarkLogic process uses all available memory, the OS will use swap space as a temporary memory store. Once this happens, things will get slow – very slow. Having swap space will allow you to degrade over time instead of just crashing when the OS can’t allocate more memory.

If your hardware has less than 32 GB RAM, then use 1x available RAM for swap. If your hardware has more than 32 GB RAM, use 32 GB RAM.

Note that swapping is a critical error condition – if the host begins swapping, MarkLogic performance will severely degrade. Swap activity is a symptom of a problem – not having enough memory. You may need to increase the RAM on each host, or to increase the number of hosts to support your memory requirements.

Conclusion

This paper provides an initial look at resource consumption for MarkLogic Server. Because characteristics can vary greatly between applications, it is important to understand what MarkLogic does and how it consumes resources in order to properly estimate and plan. This whitepaper covered ingestion, tokenization and indexing, queries, and operational concerns with the goal of providing an overview of MarkLogic resource consumption.

Additional Reading

- Monitoring MarkLogic Server
- Query Performance and Tuning Guide
- Scalability, Availability, and Failover Guide
- Inside MarkLogic Server
- Performance Testing With MarkLogic

All representations of system configuration, storage capacity, query latency or other performance data in the sizing communications are estimates and averages based on certain assumptions and conditions. No representation is made that these throughputs, capacities, response times or other performance data will be accurate or achieved in any given deployment of MarkLogic® software: Customer results will vary depending a variety of factors. Any configuration recommended by the sizing information communication should be tested and verified by Customer subsequent to tuning the final production code and queries.

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