Endeca® MDEX Engine

Performance Tuning Guide

Version 6.2.1 • December 2011

Contents

Copyright and disclaimer

Trademarks

Preface

Endeca® InFront enables businesses to deliver targeted experiences for any customer, every time, in any channel. Utilizing all underlying product data and content, businesses are able to influence customer behavior regardless of where or how customers choose to engage — online, in-store, or on-the-go. And with integrated analytics and agile business-user tools, InFront solutions help businesses adapt to changing market needs, influence customer behavior across channels, and dynamically manage a relevant and targeted experience for every customer, every time.

InFront Workbench with Experience Manager provides a single, flexible platform to create, deliver, and manage content-rich, multichannel customer experiences. Experience Manager allows non-technical users to control how, where, when, and what type of content is presented in response to any search, category selection, or facet refinement.

At the core of InFront is the Endeca MDEX Engine,™ a hybrid search-analytical database specifically designed for high-performance exploration and discovery. InFront Integrator provides a set of extensible mechanisms to bring both structured data and unstructured content into the MDEX Engine from a variety of source systems. InFront Assembler dynamically assembles content from any resource and seamlessly combines it with results from the MDEX Engine.

These components — along with additional modules for SEO, Social, and Mobile channel support — make up the core of Endeca InFront, a customer experience management platform focused on delivering the most relevant, targeted, and optimized experience for every customer, at every step, across all customer touch points.

About this guide

This guide provides information about tuning your Endeca Dgraphs and Agraphs to provide optimal performance. It also includes hardware provisioning recommendations, and performance recommendations related to various aspects of your implementation, including performance of Dgidx. This guide includes operating system support, as well as storage, memory, and networks support and recommendations.

Who should use this guide

This guide is intended for system administrators and developers responsible for the performance of an Endeca implementation.

Conventions used in this guide

This guide uses the following typographical conventions:

Code examples, inline references to code elements, file names, and user input are set in `monospace` font. In the case of long lines of code, or when inline monospace text occurs at the end of a line, the following symbol is used to show that the content continues on to the next line: ¬

When copying and pasting such examples, ensure that any occurrences of the symbol and the corresponding line break are deleted and any remaining space is closed up.

Contacting Endeca Customer Support

The Endeca Support Center provides registered users with important information regarding Endeca software, implementation questions, product and solution help, training and professional services consultation as well as overall news and updates from Endeca.

You can contact Endeca Standard Customer Support through the Support section of the Endeca Developer Network (EDeN) at *http://eden.endeca.com*.

Chapter 1

Before You Begin

This section provides background information you should know before you begin to diagnose performance problems in your Endeca implementation.

About the Dgraph and the Agraph

A typical Endeca implementation includes one or more Dgraphs. Optionally, it can include an Agraph that connects to a number of Dgraphs. This topic reviews these programs.

Introduction to the Dgraph

The Dgraph is the name of the process for the MDEX Engine, which is the query engine that provides the backbone for all Endeca solutions. The Dgraph uses proprietary data structures and algorithms that allow it to provide real-time responses to client requests. Because the Dgraph is key to every Endeca implementation, its performance is critical.

Introduction to the Agraph

A distributed configuration requires a program called the Agraph in addition to the Dgraph. The Agraph typically resides on a separate machine.

Starting with the MDEX Engine version 6.0, a more powerful Dgraph that utilizes 64-bit support can accommodate much larger data sets, compared with those supported in previous versions, without the need to implement an Agraph. Prior to the capabilities of 64-bit support in the MDEX Engine, the hard platform limit on the amount of RAM available per process could require using an Agraph for large data set implementations, in order to distribute memory requirements.

The Agraph program is responsible for receiving requests from clients, forwarding the requests to the distributed Dgraphs, and coordinating the results. From the perspective of the Endeca Presentation API, the Agraph program behaves similarly to the Dgraph program.

Agraph-based implementations allow parallelization of query processing. The implementation of this parallelization results from partitioning the set of records into two or more disjoint subsets of records and then assigning each subset to its own Dgraph.

When to consider using an Agraph

You can use a single Dgraph or a set of load-balanced Dgraphs for processing an entire data set in many implementations in which an Agraph had to be used in previous versions of Endeca IAP.

Note: Starting with version 6.0, indexing time is improved for many applications, in particular, for text-heavy applications. It is worth testing such applications without an Agraph.

Consider an Agraph deployment in the following situations:

- The MDEX data is too large to fit on a single server.
- The large number of records in the MDEX Engine causes query processing to be too slow (this varies from application to application).
- The large number of records causes indexing time to be too slow.

For more information about Agraph performance, see *"Agraph performance considerations"* in this guide. To assess whether your application parameters warrant the use of an Agraph, consult Endeca Professional Services.

Important concepts

There is a small set of terms and concepts you should be familiar with as you read this guide.

The following terms are used to discuss the performance of the MDEX Engine:

- *Throughput* is the number of requests processed by the MDEX Engine per unit of time. In this guide, unless otherwise specified, it is expressed as query operations per second (ops/sec). Throughput is measured with the performance tool Eneperf using an MDEX Engine request log.
- *Dgraph sustained throughput* is the measure of query capacity, that is, the maximum number of requests that can be consistently processed by the MDEX Engine per second.
- *Latency* is how fast the MDEX Engine responds to queries, or the time it takes for a query to be returned by the Engine, typically in milliseconds.
- *Maximum latency* is the maximum time it takes for the longest query to be returned by the MDEX Engine.

  Note: Though latency and throughput are related, they are not directly derivable from one another. The inverse of the average latency is a lower bound on the maximum throughput. For example, if the average latency for a shopper in a supermarket checkout line is 5 minutes, we know that the checkout throughput of the store must be at least 0.2 shoppers per minute. In addition, latency and throughput are tied together by concurrency. Using the same example, the real maximum throughput may be 10 shoppers per minute because there are many checkout lanes.

- An *operation* is defined as a single request to the MDEX Engine.

  Such a request may have one of the following types:

  - Navigation (possibly including record search, analytics, and so on)
  - Dimension search
  - Record search
  - Aggregated record
  - Administration (such as a Web Service invocation for administrative purposes, statistics, configuration update, partial update, and so on)

- *Memory bandwidth* is the rate at which data can be read from or stored in memory by a processor. It is measured in bytes per second. In relation to MDEX Engine performance, you may be interested in the memory bandwidth that a system can sustain while running a Dgraph or multiple Dgraphs.
- The *virtual process size (or address space)* for the Dgraph is the total amount of virtual memory allocated by the operating system to the MDEX Engine process at any point in time. This includes the Dgraph code, the MDEX Engine data as represented on disk, the Dgraph cache and any temporary work space.
- *Resident set size (RSS)* is the amount of physical memory currently allocated and used by the MDEX Engine process. As the MDEX Engine process runs, the active executable code and data are brought into RAM, becoming part of the RSS for the MDEX Engine.

  You can view the resident set size of a process on Linux by using `ps -o pid`,`ucomm`, or `rss` commands, `ucomm`, or by using the `top` program which reports the RSS size.
- The *working set size (WSS)* of the MDEX Engine process is the amount of physical memory needed for those parts of the process that have been most recently and frequently accessed. In other words, the Dgraph WSS is the amount of memory a Dgraph process is consuming now and that is needed to avoid paging.

  The WSS of the Dgraph process directly affects RAM usage. As the working set increases, the Dgraph process memory demand increases. With a larger WSS, a process needs more memory to run with acceptable performance.

  You cannot measure the WSS, but you can make assumptions about it when you measure the resident set size and observe performance; performance tends to degrade if the RSS cannot equal the WSS.
- The *Dgraph cache* is an area of memory set aside for dynamically saving the partial and complete results of processing queries.
- *Warming* is the process during which the MDEX Engine performance gradually increases to a steady state. A gradual increase in performance takes place either as the MDEX Engine starts up and processes queries or following a partial update.
- *Utilization* is the percentage of the total capacity of a resource that is actually being used.
- *The number of concurrent users* is the number of site users engaging the MDEX Engine at any given time. When planning for Dgraph capacity based on the number of concurrent users, take into account the consideration that multiple users on a site do not continually issue queries. Between queries, a user has some "think time" before issuing another query.

Location of additional information

The table below lists additional information in the Endeca documentation set to support your performance tuning efforts.

If you want	Look here
A listing and brief description of all Dgraph and Agraph flags	*Endeca Administrator's Guide*
Information on how to develop back-end Endeca features (primarily pipeline activities)	*Endeca Forge Guide*, *Endeca Partial Updates Guide*, and *Endeca CAS Server Guide*
Information on how to develop front-end Endeca features (primarily API activities)	*Endeca Basic Development Guide*, *Endeca Advanced Development Guide*, *Endeca Developer’s Guide for Endeca RAD Toolkit for*

If you want	Look here
	ASP.NET, and the *Endeca Web Services and XQuery Developer's Guide*
Details on the Endeca Application Controller	*Endeca Application Controller Guide*

Chapter 2

System Characteristics and Hardware

This section provides recommendations for hardware used for an Endeca implementation and discusses typical hardware-based issues that affect performance of the MDEX Engine.

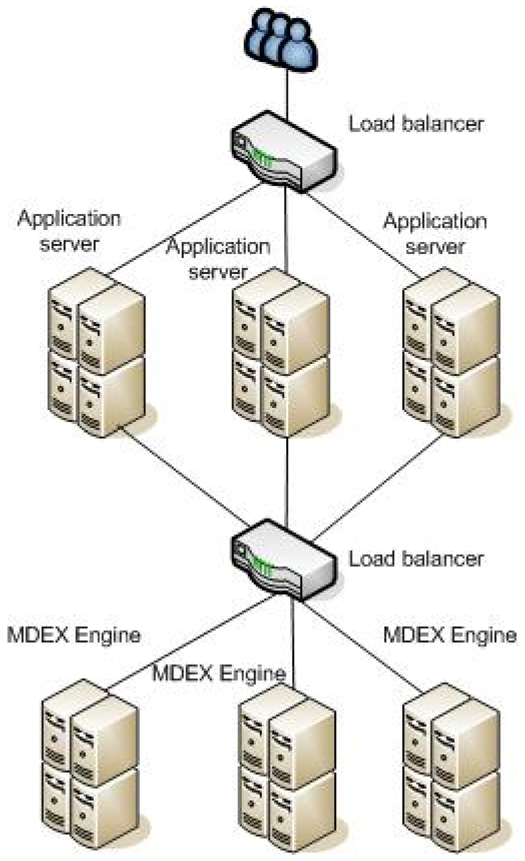
MDEX Engine architecture and performance

The MDEX Engine is optimized for performance. This section reviews those characteristics of the Engine that have a direct impact on its performance.

Hardware architecture diagram

The following diagram represents a typical MDEX Engine deployment architecture. It shows a set of application servers and MDEX Engines, each with a dedicated hardware load balancer. The Information Transformation Layer (ITL) that supplies data to the MDEX Engine index is not shown.

In this diagram, a load balancer directs query requests to one of the MDEX Engines. If you are using servers with dual-core or quad-core processors, multiple multithreaded MDEX Engines can be configured on the same machine, with two or more threads configured for each MDEX Engine.



Resource utilization

The MDEX Engine stores index structures in system memory to provide rapid access during query execution. Less frequently accessed structures and record data are stored on disk; these are only pulled into RAM as needed.

Storage locality

The data and indexes are stored in memory and on disk in a manner that provides optimal locality for common access patterns. When queries have to access disk to retrieve information, they find all the data required with the minimum number of seek operations. This decreases the cumulative disk access seek times thereby decreasing the time needed for query processing and increasing query throughput.

Unified Dgraph cache

The MDEX Engine has a unified dynamic cache where it stores intermediate results and index structures for future processing. When similar requests are made to the Engine with slight changes (example: sorting by price, then ranking, then popularity), the Engine stores intermediate results in the cache. This allows for the optimal reuse of data previously retrieved from slower sources, such as disk. The cache is dynamically managed by the MDEX Engine to keep the optimal data cached for the current query patterns.

Stateless architecture combined with load balancing

The Endeca implementation has a stateless server architecture. Query processing does not require any state information about prior queries from this client or other clients. Because of this, when multiple identical MDEX Engines are placed in parallel behind a load balancer, the response will be identical regardless of which server receives the request. The throughput of such a system is equal to the throughput of a single server times the number of parallel servers.

Multithreaded mode

The MDEX Engine always runs in multithreaded mode with the total number of threads set to 1 by default. Endeca recommends to increase this number to maximize the use of system resources. On processors that are multithreaded or multicore, multiple query threads can use a single processor at the same time.

64-bit architecture

The MDEX Engine utilizes 64-bit operating systems and processors, and can store and access larger volumes of data with scale. The MDEX Engine can utilize as much physical memory as can be placed in a server. Running in the 64-bit environment, the MDEX Engine can service many memory-intensive requests simultaneously without the risk of running out of memory address space. This, combined with a large Dgraph cache (1GB), provides a significant performance benefit.

Storage considerations

Endeca recommends using one of two storage approaches with Endeca IAP implementations, RAID or SAN-backed network-attached storage (if using RAID is not possible).

Locally attached RAID storage (RAID 5/6, RAID 10, or RAID 0)

For RAID disks, use these Endeca recommendations.

Storage availability after disk failure is usually a requirement for your RAID configuration. In this case, you may opt for either a read/write balanced configuration or a more purely read-oriented configuration.

For most implementations, a configuration that balances the demands of disk read and write activities is the best choice.

- RAID 5/6. For some implementations, disk read speed is paramount and write speed is much less important to performance. For example, suppose the baseline index is never modified by partial updates, and new baseline indexes are moved into production only infrequently. In these implementations, a RAID 5 (or RAID 6) configuration improves availability with the least cost in spindles.
- RAID 10 (also known as RAID 1+0) is an excellent choice for devices that are partitioned across a disk array of four or more spindles. RAID 10 provides the performance benefits of striping and the redundancy of mirroring.
- RAID 0. The RAID 0 configuration is useful when storage availability after disk failure is not a concern. This is because both read and write activities are parallelized across all available spindles to decrease access latency and increase read and write throughput.

In any RAID configuration, high rotational speeds (such as 15k RPM or 10k RPM) are very beneficial to performance. Performance-oriented RAID controller features, such as battery-backed write caching, or a large cache size within the RAID controller, are also very beneficial to performance.

SAN-backed network-attached storage

As an alternative to using RAID disks, you can also use SAN-backed storage with a Fibre Channel backplane network from the MDEX Engine server to the SAN.

A storage area network (SAN) is a network to which remote storage devices are attached, usually accessible by a single machine in a one-to-one relationship. The storage devices appear to the operating system as locally attached to the server, as opposed to network- attached disks.

Note: To enable a successful Endeca implementation, ensure that the SAN is properly configured. It is also preferable that the MDEX Engine has dedicated access to its own SAN disk arrays.

In Endeca implementations, a SAN is in many cases faster and easier to work with than local storage. SAN-backed storage provides the following benefits:

- Faster promotion of index images from staging to production
- Faster backup of index images in production
- Faster copying of data from staging to production server
- Simpler backups of Endeca index files due to built-in functions for backups and snapshots in SAN

Note: Network-attached storage with NFS is known to cause performance issues and is not recommended in Endeca implementations.

Memory considerations

This section discusses the relationship between the amount of RAM, the Dgraph process virtual memory usage, the Dgraph cache, the working set size (WSS), and the resident set size (RSS) for the Dgraph process and their impact on performance.

In general, storing information on disk, instead of in memory, increases disk activity, which slows down the server. Although all the information the MDEX Engine may need is stored on disk, the running MDEX Engine aims to store in memory as many of its structures it currently needs as possible.

The decisions on what to keep in memory at any given time are based on which parts of the Dgraph are most frequently used. This affects the resident set size and the working set size of the running Dgraph, which, as they increase, lead to the increase of RAM being consumed.

Related Links

Dgraph virtual memory vs. RAM: use cases on page 19

While the amount of virtual memory consumed by the Dgraph process may grow and even exceed RAM at times, it is important for performance reasons that the working set size of the Dgraph process does not exceed RAM.

About Dgraph process memory usage

The Dgraph performs best when the working set of its process fits in RAM without swapping memory pages to disk.

The working set of the Dgraph process is a collection of pages in the virtual address space of the process that is resident in physical memory. The pages in the working set have been most recently and frequently referenced. In other words, the Dgraph working set is the amount of memory a Dgraph process is consuming now. This is the amount of memory that is needed to avoid paging.

In general, depending on the query load, the virtual memory process size of the Dgraph fluctuates. In some cases, it can exceed physical memory to a degree without affecting performance.

The section *"Dgraph virtual memory vs. RAM: use cases"* illustrates these statements.

Many factors affect the amount of memory needed by the Dgraph process. The number of records in the source data and their complexity are the most obvious factors, but the use of almost any feature will cause some increase in RAM usage.

The amount of memory needed for the Dgraph process also depends on other aspects of the query mix, such as which of the items that typically constitute Guided Navigation are being used and requested (records, dimensions, refinements, or other), and their particular usage in the query mix.

Memory usage recommendations for optimizing performance

Use the following recommendations to measure memory and optimize its usage for best performance.

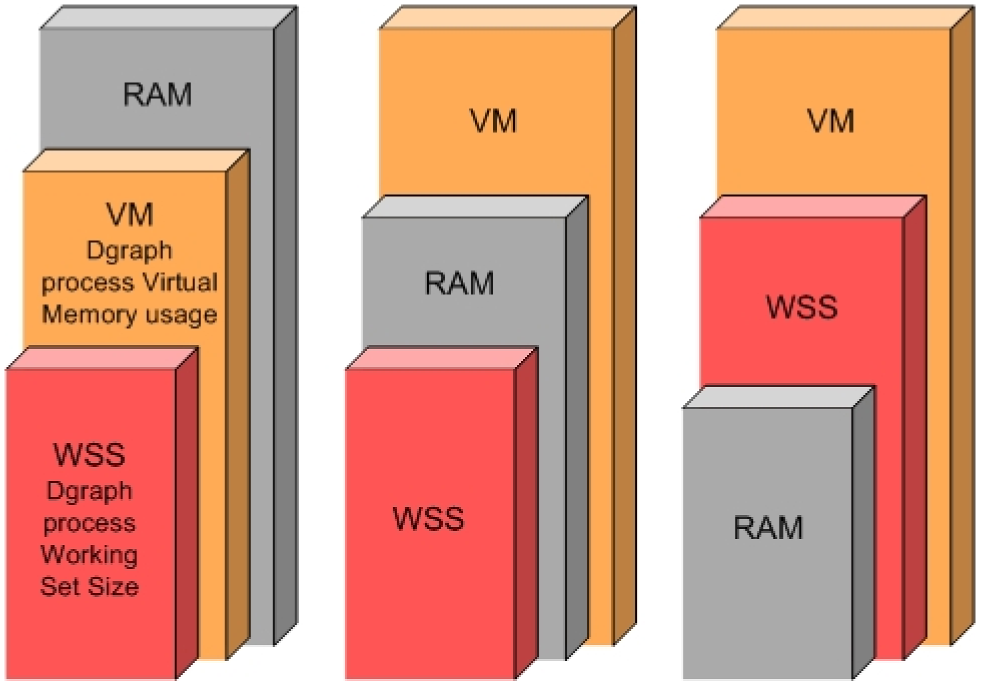
- Periodically measure the virtual memory process size of the MDEX Engine and its resident set size. The goal for these tests is to check whether the working set size (WSS) of the MDEX Engine starts to significantly exceed physical memory (it may exceed physical memory to a degree). The WSS cannot be computed, although it is always less than or equal to the amount of virtual process size for the MDEX Engine.
- Determine the WSS experimentally: if you notice that increasing RSS (by adding RAM or subtracting competing processes) improves performance of the MDEX Engine, this means that the WSS was previously larger than the RSS. This was likely the cause of the performance degradation.
- If the size of the WSS grows too close to the amount of RAM, or starts to exceed it, paging to disk begins and you will notice rapid decreases in performance.

The most noticeable symptom of paging is a large increase in Dgraph query latency. You can directly measure the amount of paging by using tools. For a list of some commonly used tools, see *"Useful Third-Party Tools"* in this guide.

Dgraph virtual memory vs. RAM: use cases

While the amount of virtual memory consumed by the Dgraph process may grow and even exceed RAM at times, it is important for performance reasons that the working set size of the Dgraph process does not exceed RAM.

The following diagram illustrates this relationship:


RAM
VM
Dgraph process Virtual Memory usage
WSS
Dgraph process Working Set Size
VM
RAM
WSS
VM
WSS
RAM

In this diagram:

- RAM is the amount of physical memory
- VM is the Dgraph process virtual memory usage
- WSS is the Dgraph process working set size

The diagram illustrates three distinct use cases:

- **Typical operation with normal memory saturation.** The graph on the left side illustrates the case where the amount of virtual memory used by the Dgraph process completely fits into RAM and thus the working set size of the Dgraph process also fits into RAM. This is a standard situation under which the Dgraph maintains its optimal performance.
- **Typical operation in an out-of-memory situation**. The graph in the middle illustrates the case where, while the amount of virtual memory exceeds RAM, the working set size of the Dgraph process fits into RAM. In this case, the Dgraph also maintains its optimal performance.
- **Potentially I/O bound operation with poor performance where WSS starts to exceed RAM**. The graph on the right side illustrates a situation that you should avoid. In this case, both the amount of virtual memory consumed by the Dgraph and the working set size of the Dgraph exceed RAM. Two situations are possible in this scenario that are of particular interest to you: the WSS can start to exceed RAM mildly or significantly. Subsequently, the degradation in I/O performance can also be mild or significant. Identify the level of I/O performance that is acceptable to your implementation. Depending on the acceptable I/O performance, you can decide whether you need to address the situation with WSS exceeding RAM. In general, if WSS starts to considerably exceed RAM, this causes Dgraph performance to drop dramatically.

Solutions for memory-based Dgraph performance problems

Use the following tips when addressing paging or out-of-memory situations with the Dgraph process.

- Add more RAM to the server hosting a single Dgraph or multiple Dgraphs. This is the simplest solution to paging issues with the Dgraph. If multiple Dgraphs are sharing a machine, you can spread them out over a larger number of machines, thus giving each Dgraph a larger share of RAM. This solution has limits based on your hardware capabilities.

In addition, you can take a conservative approach, and add additional RAM in cases where the Dgraph memory consumption (WSS) approaches the amount of RAM available for the Dgraph, but does not exceed it yet. In such cases, while additional RAM may not be necessary to create an environment free of I/O contention, it provides a buffer and ensures that memory is available when needed.

- Defragment the file system periodically. Note that this may alleviate some performance problems.
- Consider tuning the `read_ahead_kb` kernel parameter on Linux. For example, a large data scale implementation that is operating out of memory can be a candidate for tuning this parameter.
- Explore how you use features such as wildcard search, multi-assign for dimensions, and others.

Related Links

Dgraph and Agraph Analysis and Tuning on page 57
: This section describes Dgraph and Agraph performance tuning tips feature by feature. Features are not presented in order of severity of system impact.

Memory usage recommendations for optimizing performance on page 19
: Use the following recommendations to measure memory and optimize its usage for best performance.

Solutions for memory-based Dgraph performance problems on page 20
: Use the following tips when addressing paging or out-of-memory situations with the Dgraph process.

Cache-tuning recommendations for optimizing performance on page 22
: There is no hard and fast rule for how to best allocate memory between internal Dgraph cache and FS cache, if you do not have enough memory to maximize both. For example, if you are operating at large data scale, it is likely that you will not have enough memory to maximize both the FS cache and the Dgraph cache. In practice it is easier to determine the right answer experimentally.

Tuning the read_ahead_kb kernel parameter on page 30
: Endeca recommends setting the `read_ahead_kb` kernel parameter to 64 kilobytes on all Linux machines (RHEL 5). This setting controls how much extra data the operating system reads from disk when performing I/O operations.

About the Dgraph cache

The MDEX Engine cache (or the Dgraph cache) is a storage area in memory that the Dgraph uses to dynamically save potentially useful data structures, such as partial and complete results of processing queries.

Since the Dgraph has direct access to the structures it needs, it does not need to repeat the computational work previously done. The structures that are chosen for storing enable the Dgraph to answer queries faster by using fewer server resources.

The Dgraph cache is unified and adaptive:

- The cache is unified in that all sorts of data structures go into the same cache. The whole Dgraph has one cache, and all the threads share it.
- The cache is adaptive in that it uses a dynamic mechanism for evicting those data structures that it finds no longer useful. Its eviction algorithm attaches value to each cache object based on the empirical evidence of how useful the object actually is to the Engine, based on your current data,

and responding to your visitors' current queries. When this information changes, the Dgraph cache detects the change and adjusts, but you do not have to retune it.

The default Dgraph cache size (specified by the `--cmem` flag) is 1024MB (1GB).

The Dgraph cache improves both throughput and latency by taking advantage of similarities between processed queries. When a query is processed, the Dgraph checks to see whether processing time can be saved by looking up the results of some or all of the query computation from an earlier query.

The Dgraph cache is used to dynamically cache query results as well as partial or intermediate results. For example, if you perform a text search query the result will be stored, if it was not already, in the cache. If you then refine the results by selecting a dimension value, your original text search query is augmented with a refinement. It is likely that the Dgraph can take advantage of the cached text search result from your original query and avoid recomputing that result. If the navigation refinement result is also in the cache, the Engine does not need to do that work either.

To a large extent, the contents of the Dgraph cache are self-adjusting: what information is saved there and how long it is kept is decided automatically.

However, when deploying a Dgraph you need to decide how much memory to allocate for the Dgraph cache.

Allocating more memory to the cache improves performance by increasing the amount of information that can be stored in it. Thus, this information does not have to be recomputed.

Your MDEX Engine is well-tuned only when the Dgraph cache and the file system cache are well-balanced; therefore you need to understand them both.

About the File System Cache

The file system (FS) cache is a mechanism that the operating system is using to speed up disk read and write operations.

FS caching is very beneficial for the MDEX Engine, and it is important to tune the file system cache and the Dgraph cache on the server that runs the Dgraph.

For example, consider read acceleration since it is the aspect of the FS cache that matters most when tuning the MDEX Engine for performance. The FS cache speeds up reads by holding recently accessed information in RAM (on the theory that your process will need this data again), and by proactively reading ahead beyond the area recently accessed, and holding that information in RAM too (on the theory that your process will likely ask for that data next).

Related Links

Tuning the read_ahead_kb kernel parameter on page 30

Endeca recommends setting the `read_ahead_kb` kernel parameter to 64 kilobytes on all Linux machines (RHEL 5). This setting controls how much extra data the operating system reads from disk when performing I/O operations.

Cache-tuning recommendations for optimizing performance

There is no hard and fast rule for how to best allocate memory between internal Dgraph cache and FS cache, if you do not have enough memory to maximize both. For example, if you are operating at large data scale, it is likely that you will not have enough memory to maximize both the FS cache and the Dgraph cache. In practice it is easier to determine the right answer experimentally.

Use the following practices for optimizing the Dgraph and the file system caches for best performance:

- Examine the Cache tab of the MDEX Engine Stats page, especially if you need to tune the cache. In particular, pay attention to these columns in the Cache tab:
  - “Number of rejections”. Examining this column is useful if you want to see whether you need to increase the amount of disk space used for the MDEX cache. Counts greater than zero in the "Number of rejections" column indicate that the cache is undersized and you may want to increase it.
  - “Number of reinsertions”. Examining this column is useful if you want to examine your queries for similarities and improve performance by considering the redesign of the front-end application. Large counts in the "Number of reinsertions" column indicate that simultaneous queries are computing the same values, and it may be possible to improve performance by sequencing queries, if the application design permits.
  - "Total reinsertion time". Examining this column is useful for quantifying the overall performance impact of queries that contribute to the "Number of reinsertions" column. This column represents the aggregated time that has been spent calculating identical results in parallel with other queries. This is the amount of compute time that potentially can be saved by sequencing queries in a re-design of the front-end application.
- Experiment and increase the size of the Dgraph cache as your hardware allows. However, do not set the Dgraph cache to use all the free memory available on your server, since you also want to allocate memory for the file system cache and query working memory.

  Use the Dgraph `--cmem` flag to experimentally tune the Dgraph cache. It specifies the size of the cache in megabytes of RAM, and is the major mechanism for tuning the Dgraph cache. By default, if `--cmem` is not specified, the size of the cache is 1024MB (1GB) for the Dgraph.

  If you want the Engine to perform better, and you have physical memory to spare, you can increase the cache size and see if it works. Once the MDEX Engine obtains extra memory for its cache, the cache algorithm identifies the best strategy for storing the most useful data structures and eviction of those structures that are less likely to be needed frequently.
- For a specific MDEX Engine on any server, find the optimal performance point experimentally:

  Gradually increase the size of the Dgraph cache until it no longer improves performance. When you notice that performance stops improving and starts degrading, you will know you have gone too far.

  Back off the Dgraph cache setting by a fair amount (such as 500MB). The right answer depends on both raw data size and some subtle characteristics of the workload (such as, how much disk-backed information the average query needs, and how similar or different queries are from each other).
- Review your query mix to see if it exhibits a high degree of similarity between queries (either because of a highly constrained user interface or a highly homogeneous user base). This is one of the cases where performance improvements from a larger Dgraph cache may not be noticeable. If all your queries are similar, a large Dgraph cache is unlikely to be valuable.
- Find the right balance between the Dgraph cache and the FS cache. When tuning the size of the Dgraph cache, ensure that you do not accidentally displace the amount of memory allocated to the FS cache.

  In general, the Dgraph cache may contain a slightly larger number of objects useful to the Dgraph compared with the FS cache. This is often beneficial for the Dgraph performance. However, this causes a significant performance degradation when information that is not in the FS cache is needed. This is because real disk access (not just access to the FS cache reads from RAM) will be needed more often, and disk reads are quite slow relative to the reads from the FS cache.
- Be aware of the paging situation when you experimentally determine the best strategy for allocating RAM to the Dgraph internal cache and the file system cache.

If you increase the Dgraph cache size in large increments between experiments, you may create a configuration where the Dgraph process memory (including the Dgraph cache) does not fit into physical RAM. In this situation not only there is not enough room for the FS cache, but the Dgraph process starts paging and performance degrades significantly.

- As your hardware permits, experiment with increasing the FS cache, along with the Dgraph cache. In general, performance gains from using the FS cache vary depending on what processes you are running and what they are doing with the disk.

  For information on the file system caching mechanism, refer to the online sources of information that are specific to y our operating system and the file system that you use.

The Dgraph cache and its impact on virtual process size

The amount of memory allocated to the Dgraph cache directly affects the virtual process size of the Dgraph. An example in this topic shows how to adjust the Dgraph cache.

Furthermore, since the cache is accessed frequently, the amount of virtual memory allocated to it affects the working set size of the Dgraph. This may cause virtual memory paging, which can adversely affect throughput and especially the maximum latency. Whether this is a problem depends on your deployment scenario.

Example: Adjusting the Dgraph cache

Consider a scenario where a single Dgraph runs on a machine with 8GB of physical memory:

- If the virtual process size of the Dgraph is 6GB with a default (1GB) Dgraph cache, and the machine is not being used for any other processes, it makes sense to experiment with increasing the Dgraph cache size to 2.5GB to improve performance. The resulting 8.5GB virtual process size will not cause undue memory pressure.
- If the virtual process size of the Dgraph is 9GB, this exceeds the amount of RAM (8GB) and creates significant memory pressure. However, it may still make sense to increase the Dgraph cache size above the default, if the increase is not aggressive. Although in such a situation, increasing the cache size further will slow down those queries that are not assisted by the Dgraph cache, that may be acceptable if the effect of speeding up queries by providing a larger cache is greater than the effect of slowing down queries by causing virtual memory paging.

  To make the right trade-off in this situation, increase the cache size while watching throughput, average latency, and maximum latency. At some point you will see that throughput is improving but average latency has gotten worse. Whether you are willing to trade latency degradation for throughput improvement will depend on the specific performance numbers, on your application, and on the expectations of your users.

Estimating the MDEX Engine RAM requirements

This topic provides recommendations for estimating the requirements for physical memory for an Endeca 6.1.x system given the anticipated growth of your data set.

The size of the Dgraph process is impacted by:

- The size of the Dgraph index generations in memory
- the size of the precomputed sorts in memory (if precomputed sorts are used)
- the size of the Dgraph cache
- Other factors, such as the size of the in-flight data

Each of these areas is discussed below in a separate section.

Impact of the MDEX Engine cache on WSS

Use `--cmem` to identify (or change) the Dgraph cache, and take it into account when estimating the projected amount of RAM needed for the MDEX Engine operations in view of the projected growth of the data set.

Impact of partial updates on WSS

Partial updates can have a significant impact on RSS and WSS. The precise details of the Endeca generation merging strategy are complex and proprietary. However, the rough pattern of memory usage that you can expect to see from a Dgraph running with partial updates is as follows:

- Expect a jump in address space usage each time a partial update is applied. The size of the jump depends on the size of the update. Each partial update causes one or more index generation files to be created.
- When merges of partial update generations occur, the MDEX Engine allocates space for a new generation file and merges two or more existing generations into that new generation file. This allocation causes a spike in the address space usage. Since some of the merged operations may cancel each other (for example, adding a record in generation file N is canceled by the deletion of that record in generation file N+1), the new total generation size may be smaller after the partial merge.
- Additionally, when full generation merges occur, all existing generations are merged into a single new generation file. Since the new generation file is roughly the same size as the sum of all pre-existing generation files (minus any canceled operations), the WSS roughly doubles during this period.
- While a full merge may cause the WSS to increase significantly, the effects on WSS are muted by the paging behavior of the operating system. Based on Endeca's recommendations, it is unnecessary for an MDEX Engine server to have a quantity of RAM equal to twice the generation file sizes when partial or full merge is occurring.
- It is fairly easy to detect the occurrence of a full merge. Watch the generations directory, found in `<dgidx_output>/<`*`dataset_prefix`*`>_indexes/generations/`, and notice when the number of generation files drops to 1.
- During this testing, push enough of partial updates through the system to trigger the full merge. This will provide you with a good enough estimate of how much RAM you need for handling partial updates.

Note: Beginning with version 6.1.4 of the MDEX Engine, you can set the partial updates merges to use a balanced or aggressive merge strategy. For details on the merge policy, see the *Partial Updates Guide*.

Impact of sorting strategies on WSS

When measuring WSS, account for the sorting strategies used by the MDEX Engine. To ensure that you measure the full "eventual" WSS of the Dgraph in 6.1.x, include a wide range of queries in your testing logs, ensuring that a portion of your queries utilizes sorting strategies, including precomputed sorts.

Note: You can confirm whether your sorting queries utilize precomputed sort by checking whether any of your properties is configured in Developer Studio so that it can be used for record sort, or by checking the `<RECORD_SORT_CONFIG>` element in your application's XML configuration files. This element lists properties that are configured to use precomputed sort.

Precomputed sort techniques may be used by the MDEX Engine in the default sort queries. Therefore, to verify whether any of your sorting queries use precomputed sort, you can check the **Index Preparation Tab** of the Stats page that contains **Precomputed Sorts** statistics. This metric displays how much time the Dgraph has spent computing sorts, including computing sorts and incremental sort updates.

Impact of in-flight data on WSS

In addition to the types of impact that are already listed in this topic, other factors, such as in-flight processing and data can have an effect on WSS. These factors cannot be measured directly, but you should be aware of their effect.

Recommendations for estimating projected RAM requirements

Important: Use the following recommendations with the understanding that estimating RSS and WSS depends to a large degree on the operating system processing, and is also highly dependent on the context of your Endeca implementation. The size of the Dgraph process is affected by several aspects, such as the size of the index in memory, cache, and other computations. These artifacts, in turn, depend on the features you are using, such as types of updates you run (partial or baseline), or whether the application relies on precomputed sorts. To summarize, while estimating requirements for physical memory, use these recommendations in the context of your own implementation, to account for variability in the RSS size due to these factors.

To estimate projected requirements for physical memory for an Endeca 6.1.x system, use the following recommendations:

- Measure RSS. Perform evaluations of your average resident set size for your indexes, and peak resident set size, while noting the record set size on disk. For example, you may find it useful to identify various ratios between average record size on disk, average resident set size of your indexes, and peak resident set size. For testing these numbers, employ tests with varying levels of request activity sent to the Dgraph. For example, send a considerable number of requests to the 6.1.x MDEX Engine with periodic cache flushes to force the Dgraph to go to memory or disk as needed to fulfill some of the requests (this is true if you replay request logs for your test).
- If your implementation uses partial updates, account for this fact in your MDEX Engine testing. Include in your tests large enough files that contain records which will be updated through partial updates. For more information, see the section in this topic on Impact of partial updates on RSS.
- Similarly, account for the size of the Dgraph cache, for sorting queries that utilize precomputed sorts, and for the size of in-flight data (see sections in this topic on each of these aspects of the RSS).
- Identify the ratio of the RSS to on-disk representation of the record set, and confirm that with different tests this ratio remains the same.
- Based on these evaluations, draw conclusions and identify the following numbers:
  - The average on-disk record set size and the largest on-disk record set size.
  - The peak resident set size observed with the current record set.

    Note: If you are not using partial updates, this number could be roughly equivalent to the on-disk representation of the MDEX Engine data plus the size of the cache for each of your Dgraphs, the size of the in-flight processing and data, and the fact whether precomputed sort is being used. If you are using partial updates, see a section in this topic for their impact on WSS and RSS.

- Using these recommendations, you can identify the following numbers for the MDEX Engine 6.1.x:
  - The average on disk record size that is used for your number of records.
  - The peak resident set size (RSS) of the Dgraph.
  - The peak virtual memory usage.
- Predict the growth of the RSS that you will need. You can do so based on the projected growth of the on-disk representation of the data set and the numbers that you obtain for the peak resident size, peak virtual memory usage and their ratios to your data set size.

Once you predict the growth of the resident set size, you can estimate memory requirements for your Endeca implementation. This will make it possible to provision enough hardware to support the MDEX Engines with the projected data set growth.

Network considerations

Endeca recommends that you use 100Mbit or Gigabit Ethernet. Also, make sure that all NICs in your implementation use the same duplex setting. The full-duplex setting is highly recommended.

Dgidx performance recommendations

This topic provides information about performance considerations for Dgidx.

RAM and disk swap size recommendations

Although this guide deals with MDEX Engine performance, since the Dgidx program is involved in the indexing process, it is important to plan for adequate Dgidx performance as well. It is especially important to plan for Dgidx performance if you have a large data set.

Endeca recommends provisioning your hardware for running Dgidx using these estimates:

- Plan to run Dgidx with the provisioned amount of RAM that is equal to the size of the finished index size, that is the size of the `data/dgidx_output` directory after a successful Dgidx run.
- Increase the amount of swap space size to at least the amount of RAM provisioned on your system.

Troubleshooting tips for Dgidx

if a record takes longer than 60 seconds to process by Dgidx, Dgidx prints out a warning enabling you to identify and fix the record. This information can be useful to you if you need to identify a record with extremely large numbers of property assignments. This may occur as a result of an issue with the ETL process. After you identify the record, you can review it to decide whether all of its assignments are required by the application.

Operating system considerations

This section discusses various tuning changes on the Operating system level that you can perform on the server running the MDEX Engine to optimize its performance.

Windows 2008 performance considerations

If you experience poor performance on an Intel Xeon processor-based servers running Windows Server 2008, Endeca recommends changing the default BIOS setting for power management from "Dynamic" mode to "Static High Performance" mode.

The BIOS has a mode setting that controls the power regulator. In the default "Dynamic" mode, the system attempts to balance high performance with power savings. Setting the regulator to "Static High Performance" mode forces the system to always favor performance.

This issue has been observed only on some Xeon-based servers.

VMware performance considerations

This topic discusses performance expectations of MDEX Engine deployments on VMware (all supported versions) and provides recommendations for such deployments.

Virtualizing Endeca deployments on VMware is motivated by cost management reduction that is typically associated with server consolidation, as well as by human cost reduction associated with simplified server administration and maintenance.

Supported guest operating systems

See the "Supported operating systems" section of the *Endeca MDEX Engine Installation Guide* for supported guest operating systems.

Configuration guidelines

Endeca recommends using the following guidelines for MDEX Engine deployments on VMware:

- Configure four VCPUs on a virtual machine.
- Specify four threads for each Dgraph. Overall, the number of threads should not exceed the number of VCPUs.
- Allocate a single Dgraph per virtual machine. Endeca does not recommend running more than one MDEX Engine per virtual machine.

Performance expectations

Overall, for server-level performance, the average and sustained throughput decrease in a VM environment, while the latency and the warmup time increase.

If you consider deploying an MDEX Engine with the Dgraph that is configured with four threads and where the MDEX Engine is assumed to be utilized at full capacity, expect a 10-30% performance overhead with a VMware-based deployment compared with a non-VM deployment. The indexing performance is also expected to be in the range of 10-30% overhead above the non-VM deployment. In some deployments, depending on your hardware, storage and implementation strategy, performance overhead can be up to 50%.

These performance expectations manifest in the decrease in sustained throughput, increase in average latency, increase in the amount of time it takes the Dgraph to reach 80% of its expected level of throughput, and increase in the latency of the longest query (99% of queries perform better than this query).

Additional performance recommendations

Performance risk associated with virtualizing the MDEX Engine is directly related to the performance and scalability requirements of your application. While Endeca recommends virtualization, customers interested in virtualizing HPC (high-performance computing) applications should analyze the risk associated with such projects and seek IT support with strong virtualization skills and experience. Endeca believes that virtualization of the MDEX Engine on VMware is most appropriate at smaller data scale.

Endeca recommends the following practices to ensure adequate performance on VMware:

- Implement vendor best practices for tuning performance of network and storage in a VM environment. For example, be aware of the limitation of four virtual CPUs per virtual machine.
- Be aware of the virtualization performance tax. The performance overhead, or "tax", of virtualizing the MDEX Engine varies by data set and by performance metric. When a deployment is properly configured and sized, the performance overhead is generally about 10%-30%. Endeca expects that the virtualization performance tax will exceed the range of 10%-30% and may reach up to 50% in the following situations:
  - Improperly configured or improperly sized deployments. Adequate memory allocation is especially important. Plan for additional memory and storage requirements due to index replication.
  - Write-heavy workloads. In particular, the following Endeca configurations are susceptible: (1) deployments where Dgidx and Forge are used heavily, and (2) Dgraphs under extensive and sustained partial update load.
- Rely on a robust deployment architecture. Most of the initial performance problems associated with deploying VMware occur due to mis-configurations or inadequate system resources.
- The approach to disk storage can be a significant factor in performance. Both locally-attached storage and network-attached storage solutions are supported. To ensure adequate performance, pay special attention to testing and tuning the bandwidth and latency of your storage solution with VMware. Consult with the documentation for your storage manufacturer for information on tuning your storage configuration for VMware.
- Expect that lower throughput will lead to longer warmup periods.
- Plan for lower ratio of query threads to update threads for applications leveraging frequent partial updates. Frequent partial updates are recommended in such implementations because each Dgraph is limited to four threads by the virtual machine limit of four virtual CPUs. On non-VM platforms, a Dgraph can be configured with significantly more threads, improving the ratio of query threads to update threads during partial update processing.

Linux considerations

This section lists recommended tuning changes on RHEL 4 and RHEL 5 configurations for the MDEX Engine.

About the read_ahead_kb kernel parameter

Starting with the MDEX Engine version 6.0, the MDEX Engine takes advantage of the readahead function.

Readahead is a technique employed by the Linux kernel that can improve file reading performance. If the kernel assumes that a particular file is being read sequentially, it attempts to read subsequent blocks from the file into memory before the application requests them. Setting the readahead can speed up the system's throughput, since the reading application does not have to wait as long for its subsequent requests, since they are served from cache in RAM, not from disk. However, in some cases the readahead setting generates unnecessary I/O operations and occupies memory pages which are needed for some other purpose. Therefore, tuning readahead for best performance is recommended.

You can tune readahead for optimum performance based on the settings recommended by Endeca.

Tuning the read_ahead_kb kernel parameter

Endeca recommends setting the `read_ahead_kb` kernel parameter to 64 kilobytes on all Linux machines (RHEL 5). This setting controls how much extra data the operating system reads from disk when performing I/O operations.

Reducing this value from the default typically increases sustained throughput for the MDEX Engine while also increasing its warmup time. Warmup is defined as initial performance of the MDEX Engine after startup (throughput and query latency), until the sustained level of performance is reached. Therefore, if you decide to tune this parameter, choose a value to balance these concerns.

Reducing `read_ahead_kb` has a noticeable effect and increases throughput for the MDEX Engine only in cases where a large data set may not fit into the MDEX Engine memory.

In cases when the index fits into memory, reducing `read_ahead_kb` from its default has no noticeable effect on the MDEX Engine performance.

When operating the MDEX Engine on a large data set that is running out of memory, consider adding more memory in addition to tuning `read_ahead_kb` to improve performance.

Setting `read_ahead_kb` to 64 kilobytes is a reasonable choice for most applications running on Linux.

To tune the `read_ahead_kb` kernel parameter on RHEL 5:

Add a command to `/etc/rc.local` as root:

```
echo 64 > /sys/block/sda/queue/read_ahead_kb
```

where `sda` is the name of the disk device for the MDEX Engine, and `64` is the number of kilobytes for the new `read_ahead_kb` setting.

Changing the I/O scheduler on RHEL 5

Endeca recommends changing the default I/O scheduler that the Linux kernel uses from CFQ to DEADLINE.

This dramatically speeds up performance of Endeca applications with large data sets in cases where both the amount of physical memory available to the MDEX Engine and disk I/O are limited. This recommendation applies to Endeca implementations on both RAID disk arrays and individual disks.

To adjust the I/O scheduler on a device:

1. Add a command similar to the following to /etc/rc.local as root:

   ```
   echo deadline > /sys/block/sda/queue/scheduler
   ```

   where `sda` is the name of the block device where the Dgraph input resides on your system. This changes the scheduler to DEADLINE.
2. Use performance tools to validate the results.

Disabling the swap token timeout on RHEL 5

Endeca recommends disabling the swap token timeout by setting it to zero. The swap token is a mechanism in Linux that allows some processes to make progress when the total working set size of all processes exceeds the size of physical RAM.

In situations when only one process is active, and the virtual memory size of that process gets close to, or exceeds the size of the available RAM, enabling the swap token negatively affects performance. In the context of the Dgraph, this can happen if the physical server is dedicated exclusively to running the MDEX Engine, and the index size is close to, or exceeds the size of the available RAM.

Endeca recommends disabling the swap token for those MDEX Engine configurations running on Linux that serve large data sets and are memory- and disk-bound.

If you choose not to disable the swap token, and experience erratic Dgraph performance, you may wish to examine the system to determine whether the swap token is causing problems. The swap token can cause "direct steal" operations.

To measure "direct steal" operations, check the contents of `/proc/vmstat`, adding `pgsteal_dma32` and `pgsteal_normal` values and subtracting `kswapd_steal`.

Note: Endeca recommends that you disable the swap token explicitly for the MDEX Engine disk devices even though you can obtain a patch for the Linux kernel that disables it.

To disable the swap token timeout on RHEL 5:

As part of the boot process, add one of the following options to your `/etc/rc.local` file as root:

```
sysctl -w vm.swap_token_timeout=0
```

or

```
echo 0 > /proc/sys/vm/swap_token_timeout
```

Or, add `vm.swap_token_timeout = 0` to `/etc/sysctl.conf`.

Load balancer considerations

For all deployment architectures, Endeca recommends the following load balancing practices.

- Use load balancers with the MDEX Engine to increase throughput and ensure availability in the event of hardware failure. Endeca recommends including two hardware-based load-balancing switches configured redundantly in your configuration. Having two load balancers ensures their availability in the event of a load balancer hardware failure.
- Use the "least connections" model as the best routing algorithm for balancing traffic to the Dgraphs. The “round robin” model can have negative consequences, especially when occasional long-running queries are possible and the site is operating near its maximum traffic load.
- Ensure that return traffic from Dgraphs to the client tier is directly transmitted, and does not pass back through the load balancer hardware.
- Use scripting for load balancers. For example, you can use `http://[host]:[port]/admin?op=ping` on the load balancer to check whether the Dgraph process is running on this port. If it is not running, the load balancer fails over to another port, and directs queries to the MDEX Engine that is currently available.

Load balancing and session affinity

In a load balancing situation, consider enabling session affinity on the application server that directs server requests to the load balanced Dgraphs.

Session affinity, also known as “sticky sessions”, is the function of the load balancer that directs subsequent requests from each unique session to the same Dgraph in the load balancer pool. Implementing session affinity makes the utilization of the Dgraph cache more effective, which improves performance of Dgraph access and the application server.

To facilitate session affinity, your application code can call `ENEQuery.setQueryInfo()` to create an `ENEQueryInfo` object. In this object, you set query-specific information in name/value pairs (such as the session ID and query ID) for the MDEX Engine to log.

Alternatively, you can also set this information by calling `HttpENEConnection.addHttpHeader()` and specifying a name/value pair.

In either approach, the Web application sends the name/value pairs to the MDEX Engine. However, the `setQueryInfo()` method adds the name/value pairs to the query object itself; while the `addHttp¬Header()` method adds the name/value pairs to the header of the HTTP GET request.

Note: The `addHttpHeader()` method works with the Dgraph but does not work with the Agraph. (The `setQueryInfo()` method works with both the Dgraph and Agraph.)

In cases where long URLs interact poorly with a load balancer, you may need to force a POST request. You can force a POST request by calling `HttpENEConnection.setMaxUrl()` and specifying an upper limit on the length of the URL. Any URLS longer than the specified value are sent to the MDEX Engine using a POST request. You can also call `setMaxUrl()` and specify a value of 0 to force a POST request for all queries regardless of URL length.

Remember that application code automatically sends a query using a POST if the URL becomes too long to send using a GET request. The `setMaxUrl()` provides a way to force the request type if necessary.

Session affinity increases the latency overhead of the load balancer. Therefore, Endeca recommends testing the load balanced environment for performance optimization. This helps to determine whether the benefit of increased leverage from the Dgraph cache exceeds the cost of increased latency in the load balancer.

High availability considerations

Endeca recommends the following practices to ensure high performance and high availability.

- Use the Dgraph in multithreaded mode and experiment with increasing the number of threads. By default, the Dgraph runs in multithreaded mode with the number of threads set to one. It can be configured to run with a larger number of threads.
- Protect your configuration from hardware failures:

  Use redundant disk drives with RAID (RAID 0, RAID 0+1, RAID 5), or SAN.

  Utilize device redundancy for servers, load balancer devices and routers.

  Use a second data center (hot or cold standby) to protect from site failures, such as power and network outages or enterprise data center failures.
- Protect your configuration from software failures:

If you have not done this already, upgrade to 64-bit systems to avoid “out of address space” failures. (Starting with its version 6.*, MDEX Engine installations are only supported on 64-bit systems.)

Use respawning monitors to protect against unexpected fatal process errors.

Watch out for paging with process memory usage.

Periodically examine your application for slow queries, or massive responses (too many results returned not all of which may be needed by the users).

Chapter 3

Using Multithreaded Mode

This section discusses MDEX Engine performance in multithreaded mode.

About multithreaded mode

The MDEX Engine always runs in multithreaded mode with the default number of threads set to 1. The multithreaded mode cannot be disabled.

The MDEX Engine always starts with a pool of threads that you can control with the `--threads` flag. These threads include query processing and partial update processing threads and additional threads that support query and update processing.

Each thread acts like an independent MDEX Engine, processing client requests one at a time and performing other tasks that support these requests, such as sorting and background index merging. It is important that the threads share data, memory, and the server network port. Essentially, this allows a multithreaded MDEX Engine with N threads to appear as a single MDEX Engine process that can work on N queries at a time. Each of the independent threads can run on independent CPUs (or cores), allowing a single multithreaded MDEX Engine to make use of multi-processor hardware.

Multiple threads can also share a processor, especially a multi-core processor, allowing an MDEX Engine running on a single-processor host to remain responsive as long-running queries are handled.

Benefits of multithreaded MDEX Engine

The MDEX Engine normally runs in multithreaded mode with the default number of threads set to 1. For many applications, Endeca recommends running the MDEX Engine with the number of threads greater than 1. These applications have the following characteristics.

- **Large index files on disk**. Only one set of index files is required for the multithreaded MDEX Engine. Thus, in addition to reduced hardware costs, the multithreaded approach reduces the hardware hosting disk space required.
- **Long-running queries**. For applications that rely on commonly used MDEX Engine features, the vast majority of queries complete in a fraction of a second. This allows the MDEX Engine to remain responsive at all times. However, many applications make use of more advanced features (such as computing complex aggregate Analytics queries) and can encounter longer running queries. For such applications, multithreaded mode allows the MDEX Engine to remain responsive while working on long-running queries.

- **Simplified system management and network architecture**. Enabling multithreading and tuning it is much simpler than adding new distinct servers that will run additional MDEX Engines, which includes reconfiguring the file system, adding load balancers and other infrastructure changes.
- **Applications with high throughput requirements with limited hardware resources**. The most efficient way to achieve simultaneous high throughput is to add MDEX Engines and run multiple MDEX Engines on distinct servers. But, when hardware resources are limited, running a multithreaded MDEX Engine on the same server requires fewer hardware resources than multiple distinct Engines, because all threads in the multithreaded MDEX Engine share resources.

  The MDEX Engine relies on in-memory index structures to provide sub-second responses to complex queries. As the scale of application data increases, so does the memory required to host a single instance of the MDEX Engine.

  Multithreaded execution mode enables more efficient utilization of RAM through SMP (Symmetric Multi-Processing) configurations. For example, if your current data scale requires 4GB of RAM, and query throughput requires four CPUs, multithreaded execution allows the site to be hosted on a single quad-processor machine with 5-6GB of RAM, rather than using more costly options, such as four single-processor machines, each with 4GB of RAM, or a 16GB machine with four Dgraphs on it.
- **Applications that heavily use the MDEX Engine dynamic cache**. Such applications cause a multithreaded MDEX Engine (with threads greater than 1) to perform better than multiple singlethreaded MDEX Engines because all threads in a multithreaded Engine share the same dynamic cache. This is especially true when that cache is cleared frequently due to restarts or partial updates, or when the cache is typically under heavy eviction pressure.

The MDEX Engine threading pool

The MDEX Engine consistently manages all processor-intensive tasks related to query processing using its preconfigured threading pool.

The `--threads` flag reflects the total number of threads in the MDEX Engine threading pool.

You define the number of threads in the threading pool at MDEX Engine startup, based on the setting for the `--threads` flag.

Recall that the recommended number of threads for the MDEX Engine is typically equal to the number of cores on the MDEX Engine server. By managing the threading pool, the MDEX Engine lets you more accurately limit the available computation resources to each core. This ensures that the system resources are used effectively for the highly prioritized tasks in the MDEX Engine all of which support query processing and high performance.

The threading pool manages the following MDEX Engine tasks:

- Query processing tasks
- Update and administrative operations
- All tasks that support query processing in the MDEX Engine. The MDEX Engine allocates these tasks for threads in the threading pool. The tasks include all high-priority, CPU-intensive, frequently performed operations the MDEX Engine runs in production. For example, they include precomputed sorting, background merging of index generations, and operations that support high performance of updates, among others.

Other MDEX Engine operations that do not have a significant impact on CPU usage are not managed by the threading pool.

Note: If you use operating system commands such as `top` to examine the number of threads used by the MDEX Engine server, you may see a number that is larger than the number you specify with the `--threads` flag. This is because in addition to this number of threads, the MDEX Engine may use additional threads for other tasks. These additional threads support tasks that are run infrequently, are less-CPU intensive, and do not affect overall MDEX Engine performance. You cannot control these additional threads.

Configuring the number of MDEX Engine threads

For most applications, Endeca recommends experimenting and increasing the number of threads.

By default, the MDEX Engine runs in multithreaded mode with the number of threads set to 1.

To increase the number of threads:

Specify it for the `--threads` flag when starting the MDEX Engine (Dgraph).
For example: `--threads 4`

This starts the MDEX Engine in multithreaded mode with four threads that are used for query processing and other MDEX Engine tasks that support query processing.

When to increase the number of threads

Endeca recommends using a higher setting for threads than in previous releases. Increasing the number of threads allows the MDEX Engine to handle more queries simultaneously.

Use the following recommendations:

- If you are using an application with a low throughput without long-running queries, this implementation can run in a singlethreaded mode in which one thread is used to process all query requests to the MDEX Engine. The same thread is used for other query-related processes of the MDEX Engine.
- If you are using a single MDEX Engine server with one thread, it is worth increasing the number of threads to improve performance.

  A simple recommendation is to configure at least one thread per core. Higher ratios may generate more throughput, but due to the potential impact on latencies, Endeca recommends running further testing to find the thread count most beneficial to the needs of a specific application.

  If increasing the number of threads stops improving query performance, this is an inflection point at which you can start considering the need to switch to a configuration with more Dgraphs.

  A typical estimate that you can use to start testing with the increased number of threads is about 1 thread per core. For example:

  - On a standard processor, enable 1 thread per processor
  - On a dual-core processor, enable 2 threads per processor
  - On a quad-core processor, enable 4 threads per processor

Multithreaded MDEX Engine performance

The performance of an MDEX Engine process is a function of a number of factors.

These factors include:

- Base, single-threaded performance, given the application data and query profile
- Number of processors on the host system
- Query characteristics
- Host operating system

Generally, on a host system with N CPUs or cores, where one single-threaded MDEX Engine can serve K operations/seconds of query load, N or more independent MDEX Engine processes will serve somewhat less than N times K, commonly in the 80-90% utilization range. In other words, given the base single-instance performance of K, the expected N-processor performance is given by $U \times K \times N \quad where \quad (0.8 \leq U \leq 0.9)$.

The expected performance for one multithreaded MDEX Engine with more than one thread is similar, but generally somewhat less. In this case, the expected performance is given by the above formula, except with utilization in the 65% to 85% range ($0.65 \leq U \leq 0.85$). However, less RAM is required for running one multithreaded MDEX Engine with threads more than one compared with running separate single-threaded MDEX Engines.

For example, if one single-threaded MDEX Engine provides 20 ops/sec on a given load, running two MDEX Engines on a dual processor may provide around 36 ops/sec (U=90%, K=20, N=2). Running the same application with an MDEX Engine with threads more than one may provide 32 ops/sec (U=80%, K=20, N=2).

Similarly, if a single MDEX Engine requires 16GB of RAM, two Engines will require 32GB. Whereas a single MDEX Engine with more than one processing thread will only require slightly more than 16GB of RAM.

To summarize, Endeca recommends that you run a single MDEX Engine with the number of threads set to more than one, as opposed to multiple MDEX Engines. (Running multiple MDEX Engines introduces implementation complexity and also requires a load balancer.)

Recommended threading strategies and OS platform

The size of the thread pool and the host operating system impact performance and processor utilization.

In general, Endeca recommends using one thread per processor or core for good performance in most cases. The actual optimal number of threads for a given application depends on many factors, and is best determined through experimental performance measurements using expected query load on production data.

If high performance is required, enable more than one thread. Determine the optimal number of threads through load testing of different configurations.

As a starting point, enable the following number of threads:

- On a quad-core processor, enable 4 threads per processor
- On a hyperthreaded processor, enable 2 threads per processor
- On a standard processor, enable 1 thread per processor

For example, consider a server with two hyperthreaded processors and sufficient disk resources and RAM, on which a high-performance application will be deployed. The appropriate starting point for such an architecture would be one MDEX Engine running multithreaded with 4 threads.

Multithreaded MDEX Engine on Linux and Solaris

On Linux and Solaris, the MDEX Engine uses the POSIX Thread Library, `Pthreads`. You can examine the thread count using standard tools, such as `top`.

Multithreaded MDEX Engine on Windows

On Windows, the MDEX Engine uses native Windows threads. The thread count for an MDEX Engine can be examined in the Windows Task Manager in the Threads column.

Note: The number of threads listed may be greater than the value specified for the `--threads` flag; the additional threads that could be listed are those that are used infrequently by processes that are not CPU-intensive and represent internal maintenance tasks. All the CPU-intensive, query processing-related threads are controlled by the `--threads` flag.

Multithreaded MDEX Engine on VMware

On VMware, use the following configuration:

- Be aware of the limitation of four virtual CPUs per virtual machine.
- Specify four threads for each Dgraph. Overall, the number of threads should not exceed the number of VCPUs.

Chapter 4

Diagnosing Dgraph and Agraph Problems

This section discusses techniques for determining the root cause of apparent poor MDEX Engine performance. It walks you through some example scenarios and points you in the appropriate direction, based upon the problems that may be present in your Endeca implementation.

Information you need

This section lists the information you need and the tools you can use to gather information in order to analyze and optimize the MDEX Engine performance.

Use the following sources of information:

- System state characteristics
- The MDEX Engine request log
- The Cheetah utility
- Eneperf

Sometimes poor application performance is the symptom of an operational problem (with the hardware, network, connections, or the application server). At other times, it may require you to review and revise the application coding, the Dgraph or Agraph settings that were chosen previously and may need to be adjusted, or interactions between different features.

The first step in performance tuning is to find out what is causing the application to run more slowly than expected.

As you gather information about system performance, Endeca recommends that you note what steps you take and any changes you make to your environment, to ensure that you can analyze them or revert to your previous settings if needed.

When testing performance, make sure that the types of operations used to produce a load against the Dgraph or Agraph are representative of an actual application usage scenario.

System state characteristics

The first clues to identifying the source of a performance problem are found in the system state. The following characteristics are easy to extract and may immediately indicate a direction in which to concentrate further investigation.

- The `Dgraph_input` directory.
- **Information about changes in the configuration.** This includes:

  - Can the issue be replicated in the staging environment?
  - Could the issue be caused by changes in network traffic or other network-related performance issues?
  - Have there been any changes to the incoming data, pipeline or configuration files?

- **CPU utilization, disk I/O activity, and internal resource use.** This includes:
  - Physical number of CPUs available and the number of cores per CPU
  - The number of threads the Dgraph has been started with, and the total number of Dgraphs started on one machine
  - The type of disk I/O connection
  - CPU utilization statistics from the Dgraph host (especially when the performance problem is exhibited, if it is transient)
  - CPU utilization statistics from the front-end application host
  - disk I/O activity: processes other than the Dgraphs running on the machine that are not standard daemons or services (for example, a periodic backup process may interfere with disk access)
- **Memory utilization.** This includes:
  - Amount of allocated memory on the application server
  - Amount of physical memory (RAM) available on the Dgraph machine
  - Memory footprint of the Dgraph process. This includes the Dgraph cache (obtain it with `--cmem`), resident set size, and the amount of virtual memory available for the Dgraph process.
- **Storage capacity and configuration.** This includes:
  - Disk capacity in GB and disk rotation speed
  - Configuration and number of disks holding the index
  - Whether network-attached storage is used (SAN with Fibre Channel is recommended) versus local storage
  - Whether RAID configuration is used (the simultaneous use of two or more hard disk drives to achieve greater levels of performance)
  - If RAID is used, the configuration of the read-ahead policy for RAID. If the policy you have allows read-ahead, this lets the disk controller read additional data into the disk cache, which in turn increases the Dgraph performance.
  - Whether mirrored disks are used

This information defines the basic parameters for the performance problem. Typically, you base initial hypotheses on these findings, and confirm them with the next steps of the investigation.

Note: It is likely that you already have many of the tools you need to assess system state.

Related Links

Useful Third-Party Tools on page 145
This section lists some third-party tools that you may find useful during the Endeca performance monitoring process. The tools listed here are not supported by Endeca and are subject to change. In addition, these suggestions are not meant to overrule your choice of other tools.

Performance tools overview

You can use the following performance tools.

- The MDEX Engine Request Log

- The MDEX Engine Statistics page
- The MDEX Engine Auditing page
- The Cheetah utility
- Eneperf

The following sections describe these tools in detail.

Related Links

The MDEX Engine Request Log on page 85
: This section describes the MDEX Engine (Dgraph) request log, which you can use to analyze Endeca application performance.

MDEX Engine Statistics and Auditing on page 133
: The MDEX Engine Statistics page displays MDEX Engine (Dgraph) performance statistics. You can also view the Agraph Statistics page. The MDEX Engine Auditing page tracks usage for licensing and performance purposes. This section describes these pages.

The Eneperf Tool on page 117
: Eneperf is a performance testing tool that is included in your Endeca installation. This section describes how to use Eneperf.

The MDEX Engine request log

The MDEX Engine request log captures per-query metrics from a running Dgraph.

You can sort, filter, or otherwise manipulate the Dgraph request log to collect performance information. For example, you can sort the Dgraph request log based on query processing time to get the list of most expensive queries, or sort it on response duration to track latency trends.

The MDEX Engine Statistics page

The MDEX Engine Statistics page (also called the Dgraph Stats page) provides aggregated metrics since startup, and creates a detailed breakdown of what a running Dgraph is doing.

If performance is an issue, this page can help you to figure out which features are at fault.

Typically the feature in the Hot-spot Analysis section with the highest total is the best place to start your investigation. You can use the figures in the Dgraph Stats page to calculate useful metrics.

For example, to determine your application's network usage, you can multiply the number of ops/second by the average result page size.

The MDEX Engine Auditing page

The MDEX Engine Auditing page lets you view the aggregate MDEX Engine metrics over time and provides output of XML reports that track ongoing usage statistics.

These statistics persist through process restarts. This data can be used to verify compliance with licensing terms, and is also useful for tracking product usage. Each Dgraph in an implementation is audited separately.

The Cheetah utility

Use the Cheetah utility for processing request logs to analyze query load metrics for the MDEX Engine. Cheetah reports actual performance, not the expected performance.

Use the Cheetah utility together with Eneperf to investigate whether you have performance under load. The Cheetah utility and documentation are available from the Endeca Support Center.

Here are some of the ways you can use this utility:

- Isolate requests within a specific time range with the `--timelower` and `--timeupper` flags.
- Focus your attention on user-generated requests, by excluding admin, invalid, empty and error requests with the `--ignore` flag.
- Ensure that all statistics are logged. Request metrics in Endeca log reports do not correspond directly to query load metrics for the MDEX Engine. Differences in request metrics can arise from pages that issue multiple queries and from caching. For example, run Cheetah with `--showAll` flag to ensure all statistics are logged:

  ```
  perl cheetah.pl --showAllGraph1.log > Graph1.stats
  ```

- Determine whether the performance bottleneck is caused by the Dgraph by comparing the statistics for “Engine-Only Processing Time” with “Round-Trip Response Time”.
- Show statistics based on threading with the `--showthreading` flag. This is useful when tuning your Dgraph threading configuration to increase the number of query threads.

Eneperf

Eneperf is a lightweight performance testing tool that is included in your Endeca installation. It makes Presentation API queries and XQuery-based queries against the Dgraph based on your Dgraph request logs and reveals how many operations per second the Dgraph responds with.

Dgraph performance issues

This section discusses locating and addressing Dgraph performance issues.

Improving the speed of Dgraph startup

Starting with the 6.1.x version of the MDEX Engine, Web services are loaded by default at startup. For this reason, Dgraph startup takes slightly longer than it did in the version 6.0.1. The Dgraph startup is typically faster than in Endeca IAP 5.1.

In most cases this increase in startup time is not an issue. However, if you find the startup time a problem and you are not planning to use Web services, you can turn off Web services and thus avoid the startup penalty. To do this, start the Dgraph with the `--disable_web_services` flag. (This flag is particularly useful during development, when you might be starting and stopping the Dgraph frequently.)

Tips for troubleshooting long processing time

You can use the Cheetah utility available from the Endeca Support site to determine whether the performance bottleneck is caused by the Dgraph by comparing the statistics for “Engine-Only Processing Time” with “Round-Trip Response Time”.

If "Engine-Only Processing Time" as returned by the Cheetah tool is long, look further into specific query features to identify possible causes of the problem. This list identifies which problems you may want to isolate first:

- Is the long processing time for the Engine caused by limitations of hardware resources? Identify whether long query time is caused by CPU, memory, or disk I/O utilization.
- Is a high number of records being returned by the MDEX Engine? Identify how many records are being returned per query by looking for large `nbins` values in queries as reported by Cheetah. This value indicates the maximum number of records that can be returned in the query. If this number is high, this can be expensive to compute and affects performance. Consider implementing paging control methods. For information on using paging control methods, see the *Endeca Advanced Development Guide*.
- Are all dimension refinements (dimension values) exposed for navigation? That is, examine whether your queries are spending most of their time in refinement computation. Identify whether all dimension refinements are exposed by looking for `allgroups=1` in the Dgraph request log (request URL parameter) or in Cheetah reports.

  This setting corresponds to `NavAllRefinements` value of the `ENEQuery` method.

  If the `allgroups=1` setting is present in the URL parameter, review this configuration setting for your application to decide whether it is necessary. Exposing all refinements for navigation can decrease performance because the MDEX Engine has to examine each dimension value in the dimensions and determine whether or not that dimension value is a valid refinement given a current navigation state. Exposing all dimension refinements for navigation is not recommended.

  For dimensions with many dimension values, Endeca recommends introducing a hierarchy (for example, a sift dimension hierarchy for automatically generated dimensions), so that the MDEX Engine has fewer dimension values to consider at one time.

- Are your longest queries similar? Check the longest queries for similarities, such as whether they all use the same search interface with relevance ranking, wildcard search, or record filters. See the sections in this guide about tuning performance of each of these features.
- Is record search being used? Identify whether a record search is being used by any queries by looking for "`attrs=search_interface_name`" in a query. This indicates that a record search is being used which means that possibly expensive relevance ranking modules can be contributing to high computation time.
- Which relevance ranking strategies are being used? Check the `app_prefix.relrank_strategies.xml` file for the presence of Exact, Phrase and Proximity ranking modules and test the same query with these modules removed.
- Is sorting enabled for properties or dimensions? Identify whether sorting with sort keys is enabled, for which properties and dimensions it is being used and whether it is needed. The first time a sort key is issued to a Dgraph after startup the key must be computed which can slow down performance. To isolate this problem, test the query in the staging environment by removing the sort key. If you confirm sort keys are the issue, consider using sort keys in a representative batch of queries used to warm up the Dgraph after startup. The sorts will become cached and these queries will be faster.

  Note: Also, identify if sorting for properties and dimensions is necessary. In particular, it is not necessary to flag all sortable properties as sort keys in the project. This is often a performance problem itself.

Related Links

The MDEX Engine Parameter Listing on page 93

This section describes the parameters in the MDEX Engine request logs and provides mappings between the URL that is sent from the application to the Endeca Presentation API, and the URL that is sent from the API to the MDEX Engine.

CPU recommendations for optimizing performance on page 48

Use the following recommendations to optimize CPU performance.

I/O recommendations for optimizing performance on page 48
: If you are testing the Dgraph maximum throughput using Eneperf with an adequate `num connections` and the CPU is still not fully utilized, I/O could be a problem, especially if your application is search intensive but light on other features.

Disk access recommendations for optimizing performance on page 47
: To optimize disk access performance, consider the following recommendations.

Relevance ranking on page 80
: Relevance ranking can impose a significant computational cost in the context of affected search operations (that is, operations where relevance ranking is enabled).

Warming performance vs. steady state performance

When a Dgraph starts, its performance will gradually increase until it reaches a steady state. This process is known as Dgraph warming.

It is important to distinguish between the warming performance of the Dgraph and the steady state performance. Many of the techniques discussed in this guide address either one or the other, while others address both types of performance diagnostics and optimization.

The following considerations apply specifically to diagnosing and optimizing the warming performance of the Dgraph:

- Disk I/O problems can sometimes cause slow warming.
- It is helpful to run a Dgraph warming script at startup. For example, you can use a request log of characteristic queries played against the Dgraph to help warm it to a steady state.

About planning for peak Dgraph load

It is important that you plan your capacity to handle peak load. Sustained load above the projected peak load results in requests being queued for a long time. The system cannot keep up, and as a result, site performance (in particular latency) degrades.

About tuning the number of threads

Standard system diagnostic tools can tell you how busy CPUs on the machine are. If performance is poor and the CPUs are not very busy, try to increase the number of threads.

By default, starting with the MDEX Engine version 6.0, the Dgraph is running in multithreaded mode, with the `--threads` setting set to 1.

If increasing the number of threads does not help, one of the following is happening:

- You are using too many threads in one process. This is unlikely unless you exceed four threads, in which case consider using multiple Dgraphs.
- You have an I/O problem.
- There is an underlying network problem that needs to be investigated.

Multithreaded Dgraphs on machines with multithreaded processors

Processors with multithreading is a feature that allows a single microprocessor to act like two or more separate processors to the operating system and the application programs that use it.

Hyperthreading is a feature of Intel® Xeon® processors, as well as of Pentium 4® processors that support this technology.

Similarly, SPARC® Chip Multithreading (CMT) processors provide the technology for processor multithreading.

If your machine features hyperthreading or CMT, adding threads to your Dgraph can improve peak throughput by up to 30% per processor.

Multiple Dgraphs on one machine vs. multithreaded Dgraphs

You can run more than one Dgraph on a single machine, add additional threads to a single Dgraph, or run several Dgraphs with several threads enabled for each. Depending on your application, one choice might be better than the other.

The following use cases describe these choices:

- In most cases, the following recommendation applies: Dgraphs with a large memory footprint, especially in search-intensive applications, should be run in multithreaded mode with the number of threads greater than one for best performance.

  For example, suppose you have a four-processor 16GB machine and a 3GB Dgraph. You could run four identical separate Dgraphs. A better alternative is to run one four-threaded Dgraph and thus reap the benefits of having more disk cache.

  By running with more than one thread, I/O and computation can be overlapped. Although the time to process an individual request isn't improved (and can actually increase slightly due to contention for shared resources), overall throughput is significantly boosted.

- Likewise, in many cases it is appropriate to run two or more Dgraphs on one machine, each with several threads. Two four-threaded Dgraphs on one machine is an especially common configuration. The trade-off between thread contention and memory depends on the memory footprint that you estimate is needed for each Dgraph and the amount of memory available on the machine that will host multiple Dgraphs.

Disk access recommendations for optimizing performance

To optimize disk access performance, consider the following recommendations.

- Use a dedicated storage device with low latency and high IO ops/sec for all your Endeca indices and files. Locally-attached storage with a RAID controller is preferred. Only in cases where that is not possible, SAN using a Fibre Channel will typically provide strong performance assuming it has been configured correctly.
- If you are using an array controller, Endeca recommends using a striped disk configuration, such as RAID 5/6 or RAID 0+1 that enable you to avoid having redundant disks but ensures fault tolerance.
- Do not use disks with NFS, or other file system protocols. They are known to slow down performance.
- Ensure that the log files are saved locally. Turning off verbose mode, which prints information about each request to `stdout`, can sometimes help performance.
- Ensure that you have a fast disk subsystem and plenty of memory available for disk cache managed by the operating system, since the Dgraph keeps its various text search indices on disk, including search and navigation indexes.

CPU recommendations for optimizing performance

Use the following recommendations to optimize CPU performance.

- If the CPU is under-utilized, increase the number of threads for the Dgraph.
- If the CPU is over-utilized and you are not satisfied with throughput, investigate which activities make it busy. Add machines or make the queries less taxing by tuning individual features.

Related Links

Dgraph and Agraph Analysis and Tuning on page 57
This section describes Dgraph and Agraph performance tuning tips feature by feature. Features are not presented in order of severity of system impact.

I/O recommendations for optimizing performance

If you are testing the Dgraph maximum throughput using Eneperf with an adequate `num connections` and the CPU is still not fully utilized, I/O could be a problem, especially if your application is search intensive but light on other features.

There is no absolute threshold that indicates that an application is I/O bound, but typical symptoms include very high numbers of I/O hits per second or KB per second. If I/O is below the specifications for the hardware, it is less likely to be a problem. In some cases, it is even possible to go beyond a device’s theoretical maximum because of disk caching.

To determine the level of I/O activity, use the following tools:

- On Solaris, run `iostat -2`
- On Linux, run `sar -b`
- On Windows, do the following:

  On the **Task Manager**, open the **Processes** tab.

  From the menus, select **View** > **Select Columns**.

  Check **I/O Reads**, **I/O Read Bytes**, **I/O Writes**, and **I/O Write Bytes**. These options enable new columns in the **Processes** pane that provide similar information to `sar -b` on UNIX.

Agraph performance considerations

Agraph implementations add extra complexity to performance assessment. This section contains recommendations for CPU, memory, network, and disk resources provisioning, with the goal to optimize Agraph performance. It also lists specific issues that affect Agraph performance and recommends ways to improve it.

Agraph use of server resources

The following points describe at a high level how the Agraph process uses key server resources.

- **Network resources**. The Agraph process handles queries by dispatching child queries to several Dgraph processes in parallel, and integrating the results of these queries. Therefore, the Agraph process makes significant use of networking resources, and its demand for networking resources grows linearly with the number of Dgraph child processes. Consider server hardware for an Agraph with multiple network connections, especially for Agraph implementations with a large number of

Dgraph child processes. Gigabit Ethernet connections between Agraph and Dgraph tiers are always recommended.

- **CPU usage**. The Agraph makes significant use of CPU resources, in particular, when integrating the set of responses from child Dgraph processes. As discussed in this guide, your choice of features has a significant impact on server resource demands at the Dgraph tier. Your choice of features also determines whether Agraph tier processing is CPU-intensive in your implementation. Examples of operations that place a significant computation load on the Agraph process are deep pagination into result sets, and Analytics operations that involve heavy cross-child coordination, such as MEDIAN.
- **Memory usage**. The Agraph process requires adequate RAM, but does not require or benefit from large amounts of RAM to the extent that the Dgraph process does. (With the exception of Analytics, for which the Agraph uses large amounts of RAM, similar to the Dgraph.) RAM requirements for an Agraph grow with use of features that put significant coordination load on the Agraph process. Deep pagination and the Analytics operations such as MEDIAN are examples of features that cause an Agraph to use more memory.
- **Disk usage**. The size of the index used by the Agraph process is generally a small fraction of the size of the child Dgraph process indexes. In addition, this index is accessed primarily at Agraph process startup. Therefore, provisioning high-performance disk storage for an Agraph server, (such as a large locally-attached RAID 0 or RAID 5 array, or a connection to a SAN backplane), is not usually necessary, nor does fast disk storage yield a significant performance benefit.

Recommendations for higher throughput with an Agraph

You can achieve higher throughput of Agraph queries from a single server in one of two ways.

- On Linux or Solaris platforms, you can run the Agraph process in forking mode. This is similar to the Dgraph multi-threaded mode, and allows you to start up one Agraph process that is able to handle multiple Agraph queries in parallel by forking the Agraph process. On Windows platforms, you can provision more than one Agraph process on the server.

  Server resource requirements, such as networking bandwidth, the number of CPU cores, and the amount of RAM, grow linearly with the number of Agraph processes, or process forks, running concurrently on the server.

- Beyond this point, you can reach higher throughput levels by scaling through replication: adding additional Agraph tier servers, and supporting separate Agraph processes, with the entire Agraph tier accessed through a load balancer.

About the Agraph in --fork mode

A high performance Agraph solution on UNIX should use the `--fork` command line flag for the Agraph. It causes the Agraph to fork off a new process to handle each request.

The default `--fork-max` setting is 4. You can experiment with higher values (such as 8 or 16), which may increase throughput, though at the cost of latency. Be sure to increase the number of virtual clients (the `num connections`) used by Eneperf so that all these potential Agraph child processes will be used.

Even if the Agraph machine is a multiprocessor machine, you only need to run a single Agraph if you are using the `--fork` flag. The child processes in `--fork` mode are all independent processes, which the OS schedules onto the processors.

Note: On Windows, where the Agraph does not have the `--fork` flag, the solution is to add more Agraph instances and use a load balancer to balance the load across them.

Identifying the Agraphs to Dgraphs ratio

One common consideration when considering server resources for an Agraph implementation is finding the optimal ratio of Agraph tier CPU resources to Dgraph tier CPU resources.

Outside this optimal range, either the Agraph or Dgraph tier will consistently be the bottleneck on performance; either Dgraph processes will be idle while Agraph processes are saturated while working on response integration from Dgraphs, or Agraph processes will be idle while Dgraph processes are saturated producing responses.

The optimal point for any particular implementation depends on many factors, and can be accurately found through load testing. Once you find the optimal range at a particular data scale, then the optimal range for a larger data scale is roughly predictable by holding steady the ratio of Agraph tier CPU cores to Dgraph tier CPU cores.

As an example, consider an Agraph implementation that provisions 16 child Dgraph processes across 4 Dgraph tier servers, where each Dgraph tier server has 8 CPU cores. In addition, we will assume that through load testing, we have found that at the current data scale and feature selections we reach optimal throughput by provisioning 2 Agraph tier servers, where each Agraph tier server also has 8 CPU cores and supports a single forking Agraph process.

At this data scale, the Agraph tier has 16 CPU cores relative to the 32 CPU cores at the Dgraph tier: this is a one-to-two CPU ratio.

Note: Depending on your selection of features, the optimal ratio of Agraph tier cores to Dgraph tier cores may be different from this example.

At this step, suppose that we want to increase the data scale of the application by 50%. As a testing hypothesis, we would expect that this requires 8 additional child Dgraph processes, provisioned on 2 additional Dgraph tier servers of the current specification. Prior to load testing at higher data scale, we would also assume that optimal performance requires the addition of a third Agraph tier server. If we did not provision an additional Agraph tier server, we would expect that the Agraph tier would become the consistent performance bottleneck, and that overall throughput yield of the two existing Agraph tier servers would degrade, due to the additional resource demands of coordinating work across 50% more child Dgraph processes. Load testing this hypothesis should provide you with the knowledge on whether adding a third Agraph tier server will optimize performance.

Identifying performance problems in Agraph deployments

Use the following recommendations to identify performance problems and optimize Agraph performance.

- Evaluate a slow Agraph system. Use Eneperf to measure the performance of a single Dgraph. If that Dgraph is too slow, the aggregate Agraph system is certainly going to be too slow, so there is no point in looking further until the individual Dgraph problems have been resolved.
- Isolate performance issues that could be caused by the network. Ensure that you test the Dgraph performance with Eneperf running on the Dgraph machine and also on the Agraph machine.
- Configure the Agraph on a different machine from the Dgraphs.

Testing Agraph network problems with Eneperf

Large Agraph deployments (that is, those with more than 10 or 15 Dgraphs) can saturate the network, because all of the Dgraphs send data back to the same Agraph machine. Your Agraph implementation may have performance problems with the network bandwidth.

Each request to an Agraph causes at least one and often two requests from the Agraph to each of the Dgraphs. Although the amount of data returned by the Agraph is usually equal to the sum of the responses from the Dgraphs, in some cases each of the Dgraphs can send as much data as the Agraph. This leads to a significant increase in the network traffic. The Agraph response is not sent until all the Dgraph responses are received by the Agraph.

If you notice that the Dgraphs take a long time responding to the Agraph, measure the amount of data sent over the network to the Agraph by each of the Dgraphs.

Use the Dgraph request logs and Eneperf to determine how many bytes each Dgraph is sending back to the Agraph. Remember that the Agraph machine has to receive that many bytes from n separate Dgraphs, which may overload the Agraph machine's network connection (this is why Endeca recommends using Gigabit Ethernet).

One way to test whether bandwidth to the Agraph machine is sufficient is to run multiple instances of Eneperf simultaneously on the Agraph machine. Each Eneperf instance should point at a different Dgraph.

To test the network bandwidth to the Agraph machine from multiple Dgraphs:

1. On a Unix system, run the command from `sh` as shown in this example:

   ```
   for i in dg01 dg02 ; do eneperf $i 5555 logfile 4 1 & done
   ```

   where `dg01` and `dg02` are the machine names of the Dgraph servers, `5555` is the port number, and `4` is the setting for `num connections` for Eneperf.
2. On Windows, run the commands as shown in this example:

   ```
   C:\> start /B eneperf dg01 5555 log.txt 4 1
   C:\> start /B eneperf dg02 5555 log.txt 4 1
   ```

   where `dg01` and `dg02` are host names, and `5555` is the port number.

   This produces a lot of screen output, but you will be able to see whether all the Eneperf instances are getting the performance you expect from each Dgraph.

Determining whether the Agraph CPU is saturated

The Agraph process can saturate the CPU resources on its host server, especially in a large Agraph installation with many Dgraphs. In this situation, Agraph performance can be the bottleneck on overall system throughput, since the Dgraph processes could each individually support higher throughput.

Watching CPU utilization on the Agraph server typically provides useful information. If the Agraph CPU is not running out, this indicates that the implementation is likely not CPU-bound on the Agraph server.

The best way to determine whether additional Agraph CPU resources will improve system throughput is to directly test the Agraph process using a server with more CPUs. If you do not have access to such a server, an alternative is to use the Eneperf load driver to simulate Agraph query traffic on two servers, where these two servers have in aggregate more CPU resources than the current Agraph server.

Note: The Agraph CPU can become saturated in a large Agraph system with many Dgraphs. This is evident when Eneperf is running with sufficient connections to saturate Agraph CPU but the Dgraphs are not running at full capacity.

In this testing process, prepare to run multiple instances of Eneperf simultaneously. Configure one Eneperf process for each Dgraph child process, so that each Eneperf instance sends queries to a different Dgraph. All Eneperf instances can use the same query log. Ideally, this log is derived from the request log of the Agraph process.

To test whether the Agraph CPU is saturated:

1. Test the throughput of the two Eneperf processes, running all in parallel, when all load drivers are hosted on the current Agraph server.

   This test requires substantially less CPU resources than the Agraph process, because the results of each query performed by Eneperf are not coordinated, aggregated, or merged by the Agraph.
2. Test the throughput of the two Eneperf processes, running all in parallel, when half of the load drivers are hosted on each of the Agraph testing servers. The two servers provide, in aggregate, more CPU resources to the set of load drivers.
3. Compare the throughput numbers of both tests.

   If the throughput numbers are similar, the bottleneck for this test is the tier of Dgraph child processes.

   If the throughput numbers for the second test are substantially higher, this is a strong evidence that in the first test the Agraph server CPU was the bottleneck. Because the first test consumes less CPU resources than a comparable Agraph process, this is by extension strong evidence that the Agraph process is CPU saturated and would perform better if you add more CPUs.

Identifying problems with resource usage by the application

Use the following recommendations to identify performance problems associated with resource usage.

- Isolate performance testing for those parts of the application that specifically use the Endeca MDEX Engine from testing for other parts of the application. In other words, measure the performance of those parts of the application that use the Endeca MDEX Engine separately from the performance of those parts that use other software that may cause performance problems, such as a relational database. For example, if the latency is high, consider testing the interaction of the application with the database, if you are using one.
- If you are sending a lot of requests to the front-end application and performance is slow but the MDEX Engine servers are idle, the front-end application and its resource usage is probably the issue. There are two possible fixes: you can reduce consumption of resources by the application by reviewing your coding practices for the front-end application, or add resources.

Coding practices for the front-end application

Reviewing your front-end application code can help reduce resource usage performance issues that affect it. Review your Web application to check for any of the following problems.

- Creating or discarding objects unnecessarily.
- Excessive looping, particularly over properties that are not going to be displayed.
- Creating too many variables.

Web application ephemeral port contention

Each client/server connection has a unique identifier (known as a quad) that includes an ephemeral port number that will later be reassigned. Each operating system has a range of numbers that it uses as ephemeral ports (for example, on Windows the range is 1024 through 4999).

The operating system allocates ephemeral ports when a new socket is set up.

If the range is relatively small and you are making several requests per page in parallel, you can run out of port numbers. At that point the ephemeral port numbers assigned by the operating system start colliding with ones already in use as they are recycled too quickly, and subsequent connections will be aborted.

To address this problem, try one of the following:

- Reduce the two-minute time interval that the system waits between a connection close and port reassignment. The minimum recommended time is 30 seconds.
- Change the ephemeral port range. The method varies depending on your operating system; however, details are easily obtained on the Web.

Recommendations for identifying network problems

Often the diagnosis of slow performance comes from a query load played against the front-end application. The front-end application, or the configuration of its application server, may be the reason for the poor performance.

Alternatively, the network may be the problem, although this is less likely. (In the case of a Dgraph, unlike an Agraph, it is unusual for the network to be the bottleneck.)

To identify whether the network is a performance issue:

- Compare Eneperf performance on the local host and a remote host. First, run Eneperf against the Dgraph on the Dgraph machine. Next, run the same Eneperf against the same Dgraph, but from the front-end machine (if possible), or somewhere on the other side of the network. If the difference is negligible, the network is not a problem. If Eneperf across the network is slow, you need to consider both the network itself and the application configuration.
- Alternatively, you can run the Cheetah tool and compare the “Round-Trip Response Time” with the “Engine-Only Processing Time”. If “Round-Trip Response Time” is long but the “Engine-Only Processing Time” is short, this can indicate a network problem or a configuration of an application server for the front-end application.
- Measure network performance using Netperf, a freely available tool that can be used to measure bandwidth. Alternatively, you can FTP some large files across the network link. If these tools show poor throughput across the network, this can indicate a network hardware problem such as a failing network interface card (NIC) or cable.
- In addition, check Eneperf statistics, the Dgraph request logs, or the Dgraph Stats page to see how much data is being transmitted back from the Dgraph on an average request. Large average result page size can saturate the network.

If it seems as if your application is trying to move too much data, it is likely that you may need to change the configuration of your application. To determine if changes are needed, consider the following:

- Is all of the data actually being used by the application? In other words, does the MDEX Engine return record fields that are then ignored by the front-end application? This is an especially serious problem with large documents.
- Is your application returning unnecessary fields with the Select feature (described in “Controlling Record Values with the Select Feature” in the *Endeca Advanced Development Guide*)?

- Is your application returning navigation pages that are too large? (Navigation pages are result list pages, as opposed to record detail pages.) If the application returns a lot of detailed information in the result list pages, consider reserving the details for a click-through and reducing the size of the result list pages your application returns on initial requests.
- Is your application returning large numbers of records without using the bulk record API (described in “Bulk Export of Records” in the *Endeca Advanced Development Guide*)?
- Is the network saturated? Upgrade to Gigabit Ethernet and identify the transmission speed being used. Ensure there is ample network bandwidth between the front-end application and the Dgraph. To identify Gigabit Ethernet transmission speeds, work with your network administrator.
- What is the configuration of NIC cards? Ensure that NIC duplex settings match between the Dgraph host and the web application client host and that both are set to full duplex. A mismatch can cause latency issues.
- Could large response sizes returned by the Dgraph be saturating the network? Use Cheetah analysis to confirm large response s izes returned by the Dgraph, which can be caused by the query features you use. The way certain features are used can cause slow processing time and also saturate the network.
- Do you have queries waiting in the Dgraph queue to be processed? Check "Threading/Queuing Information" summary in Cheetah for the number of items experiencing queue issues and the number of HTTP Error request 408 timeouts. Review the Dgraph setting for the number of worker threads and consider increasing it, if it is set to 1. Queuing can also be caused by spikes in traffic.
- Does the front-end application process the responses returned by the Dgraph quickly enough? Check CPU, memory, and disk I/O utilization on the front-end application server. Ensure the application server does not need to be tuned and that large responses are not being returned by the Dgraph.

Related Links

Useful Third-Party Tools on page 145
: This section lists some third-party tools that you may find useful during the Endeca performance monitoring process. The tools listed here are not supported by Endeca and are subject to change. In addition, these suggestions are not meant to overrule your choice of other tools.

Tips for troubleshooting long processing time on page 44
: You can use the Cheetah utility available from the Endeca Support site to determine whether the performance bottleneck is caused by the Dgraph by comparing the statistics for “Engine-Only Processing Time” with “Round-Trip Response Time”.

Troubleshooting connection errors

This topic discusses how to debug connection errors with ENEQuery exceptions.

Problem - The application server does not seem to connect to the Endeca server. The Endeca reference application has no difficulty connecting. A connection to the port works as confirmed by JUnit tests. A problem exists connecting to the server once all the reference application libraries are packaged into the EAR file that is run inside the WebSphere application server.

Solution - In general, the `HttpENEConection.query ENEQuery` method is used to issue a query against the Dgraph. In the `HttpENEConnection.query` method in the Java version of the Endeca Presentation API, any connections problems are raised as an `ENEQueryException`. (There is an equivalent in .NET version of the Endeca Presentation API).

To diagnose a connection problem from an application server to an Endeca server, the following assumptions are made:

- The Java version of the Endeca Presentation API is being used.
- The connection from the application server to the MDEX Engine is running on HTTP, not HTTPS.
- The application server and the MDEX Engine on the Endeca server are configured on separate machines.

To troubleshoot the connection problem, do the following:

1. Verify from the application server machine that you can connect to the port on the Endeca server. Using telnet on Windows or Unix can help you determine if you can successfully make a connection:

   ```
   telnet <hostname> <dgraph port>
   ```

   a) If you cannot establish a connection with telnet, check that the Dgraph process is running with the specified port. Check the Dgraph `stderr` log to confirm the Dgraph was able to successfully bind to the port and another process is not using the port. You can also verify the Endeca server machine is listening on a socket with the specified port using `netstat -a`. Check that a valid network route exists from the application server to the Endeca server. You can also use `ping`. Also, use `tracert` on Windows, `tracepath` on Linux, or `traceroute` on Solaris. If no valid network paths exist, check with your network administrator to eliminate possible problems with a firewall or routing configuration.
   b) If you can obtain a connection from telnet, verify that the application server can talk to the Endeca server. Write a Java program with a `static void main` method to make a connection to the MDEX Engine on the Endeca server. Make sure the Endeca Navigation JAR file is included in your classpath. If this program makes a connection successfully, the problem should only occur within the application server.
2. Write a utility JSP page that connects to the MDEX Engine on the Endeca application server and place it on the application server to verify the connection. Alternatively, you can run the Reference Application on the application server.
3. If everything works correctly, to troubleshoot further check the application server configuration. For Websphere, do the following:
   a) Check all log files in `IBM/Websphere/AppServer/profiles/AppSrv01/logs/server1`.
   b) Verify that the Reference application is correctly packaged as EAR file.
   c) Make sure Websphere deployed the EAR file and the application is running in the WAS admin console.

   Assuming that you have WAS 6.1, go to **Security** > **Secure Administration, application and infrastructure** and check whether Java 2 security is enabled. If it is enabled, make sure your `was.policy` file is saved in the META-INF directory.

Next steps

Your hardware needs should be based on the number of ops/second revealed by Eneperf testing. If you feel that the resulting hardware requirements are too great, the next thing to do is identify costly features in your front-end application and see what you can do about them.

Modifications you can make to your Dgraph or Agraph settings in order to improve the performance of your Endeca application are discussed in the next chapter.

Chapter 5

Dgraph and Agraph Analysis and Tuning

This section describes Dgraph and Agraph performance tuning tips feature by feature. Features are not presented in order of severity of system impact.

Feature performance overview

Once you have determined that the Dgraph or Agraph is the bottleneck using the techniques described in this guide, there are many things you can do to tune performance. In many cases, unnecessary complexity slows performance, so small changes can yield big returns.

It is best to begin making adjustments with a conservative strategy that you understand well. Do not modify too many features at once—it makes it difficult to assess the impact of any one change.

Details on tuning specific features can be found in the following sections. Where applicable, they discuss problematic feature interactions. Likewise, each section indicates whether the kind of data you are processing (for example, large text fields as opposed to many part numbers) significantly impacts a feature's performance.

This chapter calls out only those aspects of a feature that affect application performance. For more general information about implementing these features, see the *Endeca Forge Guide*, *Endeca Basic Development Guide* and the *Endeca Advanced Development Guide*.

Endeca record configuration

This section discusses the performance implications of some aspects of Endeca record configuration.

Record select

The Select feature prevents the transfer of unneeded properties and dimension values when they are not used by the front-end Web application.

It therefore makes the application more efficient because the unneeded data does not take up network bandwidth and memory on the application server. This may be relevant if your logs are showing large result pages.

You set the selection list on the `ENEQuery.setSelection()` method (Java), or the `ENEQuery.Se¬lection` property (.NET).

Aggregated records

Aggregated Endeca records are not necessarily an expensive feature in the MDEX Engine. However, use them only when necessary, because they add organizational and implementation complexity to the application (particularly if the rollup key is different from the display information).

Using aggregated records slows down the performance of sorting and paging.

Note also that dynamic statistics on regular and aggregated records (controlled with the `--stat-abins` Dgraph flag) are expensive computations for the Endeca MDEX Engine. See the topic in this section for more details.

Derived properties on aggregated records

Some overhead is introduced to calculate derived properties on aggregated records. In most cases this should be negligible. However, large numbers of derived properties and, more importantly, aggregated records with many member records may degrade performance.

The number of records returned with an aggregated record and performance

You can use the `Np` parameter to specify the number of records to be returned with an aggregated records. For example, `Np=1` means that a single representative record is returned with each aggregate record, and `Np=2` brings back all records.

Utilizing `Np=2` may adversely affect your performance, as it causes the MDEX Engine to serialize more records for each query. The degree to which performance is affected is proportional to the number of base records for each aggregate record that is returned.

In most cases, it is not recommended to bring back all records in each query and aggregate all records with `Np=2` as this computation could be expensive for the MDEX Engine to serialize the result. However, `Np=2` can be useful in some cases. The impact on performance is proportional to the number of records that will be returned as aggregates.

For example, if each aggregate record contains only 2 records, the record serialization time is only twice the time as it is for `Np=1`. If, however, each aggregated record has 100 records associated with it, it is 100 times more expensive to perform the record serialization for Np=2 than for Np=1.

Record serialization time is typically only a large portion of the query processing time in very low latency applications or with very large numbers of returned records.

Note also that in many cases, a 100-fold increase in record serialization time is barely noticeable. You can examine the `Prefetching horizontal records` statistics in the `Hotspot Analysis` section of the Stats page to determine whether their performance issue is due to returning many records.

For example, if you have a very small data set with queries served almost entirely from the cache, where most of the computation done by the Dgraph for each query consists of assembling the records to be returned, the negative effect on performance is reflected in the `Prefetching horizontal records` statistics being very large in this case which indicates that `Np=2` should not be used.

Dimensions and dimension values

This section discusses tuning features related to dimensions and dimension values.

Hidden dimensions

You prevent a dimension from appearing in the navigation controls by designating it as a hidden dimension.

Hidden dimensions, like regular dimensions, are composed of dimension values that allow the user to refine a set of records. The difference between regular dimensions and hidden dimensions is that regular dimensions are returned for both navigation and record queries, while hidden dimensions are only returned for record queries and dimension search.

In cases where certain dimensions in an application are composed of many values, marking such dimensions as hidden improves Dgraph performance to the extent that queries on large dimensions are limited, reducing the processing cycles and amount of data the Dgraph must return.

Dimensions and dimension values with high record coverage

Consider a case where records have dimensions that have almost—but not quite—full coverage over the records. For example, 99% of the records have a dimension value for a Location dimension, but the remaining 1% do not.

While this factor does not affect performance significantly, you can add an “n/a” dimension value to fill the gap and make the dimension have 100% coverage, if you want to let users explicitly refine to records that do not have an assignment for that dimension.

Flat dimension hierarchy

In general, avoid using large, flat dimensions (that is, dimensions with thousands of dimension values at the same level of hierarchy).

This is doubly true if statistics are enabled for those dimensions. It is better to design dimensions that contain sensible levels of hierarchy.

For some applications with extremely large, non-hierarchical dimensions, larger values for `--esampmin` can meaningfully improve dynamic refinement ranking quality with minor performance cost.

Displaying multiselect dimensions

When making decisions about whether to configure a dimension as multiselect, keep in mind that users may take longer to refine the list of results, because the user can continue to refine a multiselect dimension until all leaf dimensions have been selected.

In particular, refinements for dimensions tagged as multiselect OR are expensive.

Multi-assign dimensions

A dimension is considered to be multi-assign if there exists a record which has more than one dimension value assigned to it from that dimension.

Making a dimension multi-assign can slow down refinement computation. To improve performance, you can use multi-assign only for those dimensions for which you need it, and avoid making dimensions multi-assign where it is not useful.

Displaying refinement dimension values

Run-time performance of the MDEX Engine is sometimes directly related to the number of refinement dimension values being computed for display. If any refinement dimension values are being computed by the MDEX Engine but not being displayed by the application, use the `Ne` parameter more strictly.

The worst-case scenario for run-time performance is having a data set with a large number of dimensions, each dimension containing a large number of refinement dimension values, and setting the `ENEQuery.setNavAllRefinements()` method (Java), or `ENEQuery.NavAllRefinements()` property (.NET) to `true`. This combination is slow to compute and creates a page with an overwhelming number of refinement choices for the user. Endeca does not recommend using this strategy.

In general, you may want to reconsider the number of refinements you display, as well as consider implementing precedence rules.

Related Links

Precedence rules on page 79
This section discusses precedence rules and explains their performance impact.

Dynamic statistics on dimension values

You should only enable a dimension for dynamic statistics if you intend to use the statistics in your Endeca-enabled Web application. Because the Dgraph performs additional computation for the statistics, there is a performance cost to enabling statistics that your application does not use.

Using dynamic refinement ranking can greatly speed up refinement computation by displaying only the top refinements for a dimension, rather than computing the exhaustive list of refinements.

To decide whether or not dynamic refinement count statistics are likely to be appropriate for a project, consider the following aspects of your configuration:

- The number of dimension value refinements per page, especially dimension values assigned to large numbers of records. The more refinements are returned on each page, the more counts that need to be computed, and the bigger the performance impact.

  For example, if the data set has a large number of dimensions, and/or the application uses `ENE¬Query.setNavAllRefinements` (true), then the performance impact will be larger. This is especially true if many of the dimension values are assigned to large numbers of records. This frequently happens with hierarchical dimensions. For example, it is more expensive to count Red Wines than it is to count Merlots.

- The number of records in the data set. Data sets with large numbers of records will see a proportionally higher performance impact from record count statistics.
- The average number of results per query. Applications that tend to perform searches that match larger numbers of records will see proportionally higher impact from refinement count statistics.

As a simple rule, add up the counts for all of the refinements on the page. The performance impact of record count statistics grows proportionally with that sum over all refinements. All of the above considerations are aspects of the application that can make that sum larger, and increase your performance slowdown related to record counts.

You can speed up computation of dynamic statistics for refinements by doing the following:

- Set the following options in the `STATS` subelement in the `refinement_config.xml` file:
  - `RECORD_COUNT_DISABLE_THRESHOLD` specifies the maximum number of records in a result set above which the MDEX Engine does not compute or return any dynamic statistics for that query. This speeds up processing if you do not need the counts in this case.

- `MAX_RECORDS_COUNT` causes the MDEX Engine to stop computing dynamic statistics for a particular dimension value when it has reached the specified value. The count returned in this case is the minimum of the actual count and `MAX_RECORDS_COUNT`. Thus, you can set this parameter to a specific value if you do not need to know the count for a particular dimension value once it is sufficiently high.

Aggregated refinement counts

Dynamic statistics on regular and aggregated records are expensive computations for the Endeca MDEX Engine.

You should only enable a dimension for dynamic statistics if you intend to use the statistics in your Endeca-enabled Web application.

Similarly, you should only use the `--stat-abins` flag with the Dgraph to calculate aggregated record counts if you intend to use the statistics in your Endeca-enabled Web application. Because the Dgraph does additional computation for additional statistics, there is a performance cost for those that you are not using.

In applications where record counts or aggregated record counts are not used, these lookups are unnecessary. The MDEX Engine takes more time to return navigation objects for which the number of dimension values per record is high.

The `--stat-abins` flag for the Dgraph lets you calculate aggregated record counts beneath a given refinement. For more information on using this flag, see the *Endeca Basic Development Guide*.

Dynamic refinement ranking and performance

You can use `--esampmin` with the Dgraph, to specify the minimum number of records to sample during refinement computation. The default is `0`.

For most applications, larger values reduce performance without improving dynamic refinement ranking quality. For some applications with extremely large, non-hierarchical dimensions (if they cannot be avoided), larger values for `--esampmin` can meaningfully improve dynamic refinement ranking quality with minor performance cost.

Disabled refinements

Performance impact from displaying disabled refinements falls into three categories. They are discussed in the order of importance.

- The cost of computation involved in determining the base and default navigation states.

  The base and default navigation states are computed based on the top-level filters that may belong to these states. These filters are text searches, range, EQL and record filters and selections from dimensions. The types and numbers of these top-level filters in the base and default navigation states affect the MDEX Engine processing involved in computing the default navigation state. The more filters exist in the current navigation state, the more expensive is the task; some filters, such as EQL, are more expensive to take into account than others.

- The trade off between using dynamic refinement ranking and disabled refinements.

  In general, these two features pursue the opposite goals in the user interface — dynamic ranking allows you to intelligently return less information to the users based on most popular dimension values, whereas disabled refinements let you return more information to the users based on those refinements that are not available in the current navigation state but would have been available if some of the selections were not made by the users.

Therefore, carefully consider your choices for the user interface of your front-end application and decide for which of your refinements you would like to have one of these user experiences:

- Dynamically ranked refinements
- Disabled refinements

If, for example, for some dimensions you want to have only the most popular dimension values returned, you need dynamic ranking for those refinements. For it, you set the sampling size of records (with `--esampin`), which directly affects performance: the smaller the sampling, the quicker the computation. However, for those dimensions, the MDEX Engine then does not compute (and therefore, does not return) disabled refinements.

If, on the other hand, in your user experience you would like to show grayed out (disabled) refinements, and your performance allows it, you can decide to enable them, instead of dynamic ranking for those dimensions. This means that for those dimensions, you need to disable dynamic ranking. As a side effect, this involves a performance cost, since computing refinements without dynamic ranking is more expensive. In addition, with dynamic ranking disabled, the MDEX Engine will need to compute refinement counts for more dimension values.

- The cost of navigation queries.

  Disabled refinements computation slightly increases the navigation portion of your query processing. This increase is roughly proportional to the number of dimensions for which you request the MDEX Engine to return disabled refinements.

Displaying dimension value properties

Dimension value properties (that is, key-value pairs that the Dgraph passes back along with a dimension value) could slightly increase the processing or querying time because additional data is moved through the system, but this effect is generally minimal.

If your Endeca application does complex formatting on the properties, this could slow down page loads. If the properties are used to add formatting HTML or perform other trivial operations, they have minimal impact on performance.

Collapsible dimension values

Collapsible dimension values have a negative impact on performance.

Mapping source properties

Automatically mapping source properties is a feature that, while it can be used in the staging environment to facilitate testing, is not recommended for using in the production environment.

The Property Mapper in Developer Studio allows you to automatically map source properties to Endeca properties or Endeca dimensions, if no mapping is found. (This feature is also known as Automapper). The option of the Property Mapper that lets you map source properties to Endeca properties or dimensions defines the setting that Forge uses to handle source properties that have neither explicit nor implicit mappings.

Use this option with caution because each source property that is mapped uses system resources. Ideally, you should only map source properties that you intend to use in your implementation. Many production-level implementations automatically pull and process new data when it is available. If this data has new source properties, they will be mapped and included in your MDEX Engine indices, which uses system resources unnecessarily. As a result, the Forge output is larger, the indexer is larger and the MDEX Engine has additional indices to process.

Indexing all properties with Dgidx

The `--nostrictattrs` flag for Dgidx allows you to index every property found on a record, including those properties that do not have corresponding property mapper settings. Using this flag may negatively affect performance of Dgidx and the MDEX Engine.

If a large number of unused properties are sent to Dgidx, they will get indexed and will consume system resources during the indexing process and at run-time. These properties can also affect performance of the front-end application API, because the amount of information communicated between the MDEX Engine and the API increases.

Record sorting and filtering

This section discusses the performance impact of record sorting and filtering.

Sorting records by dimension or property

Enabling dimensions and properties for sorting increases the size of the Dgraph process and may negatively affect partial update latency. The specific size of the increase is related to the number of records included in the data set.

Therefore, in Developer Studio, enable only those dimensions or properties for sorting which are specifically needed by an application. Sorting gets slower as the process size grows and paging gets deeper.

In general, the MDEX Engine explicitly uses precomputed sorts for properties that you specifically configure as sort keys in Developer Studio, using the **“Prepare sort offline”** option.

Sorting can be done on any property, whether configured for sort or not. Configuring for sort mainly controls the generation of a precomputed sort (an internal optimization done by the MDEX Engine), and secondarily enables the field to be returned in the API sort keys function. In cases where the precomputed sort is rarely or never used (such as when the number of search results is typically small), the memory can be saved.

If the Dgraph has to compute precomputed sort objects to answer queries, the precomputed sort process in the Dgraph can be time-consuming. As a side effect of this processing, if you issue the `admin?op=exit` command to shut down the Dgraph while the precomputed sort process is still running, the actual shutdown may be delayed from the time the command is issued. This delay occurs because the Dgraph shutdown process may still be waiting for the completion of its creating several precomputed sort objects.

Geospatial sorting and filtering

Geospatial sorting and filtering is a query-time operation. The computation time it requires increases as larger sets of records are sorted and filtered. For best performance, apply geospatial sorting and filtering once the set of records has been reduced by normal refinement or search.

To optimize performance of geofilters, consider using these recommendations:

- Examine the request log for the presence of long distance queries that contain a geofilter. If there is a noticeable percentage of such queries, remove the geofilter from them.

  In other words, if a portion of your queries represents searches in which distance is very large and thus appears to be not an important factor in a query, remove the geofilter from such queries.

  For example, for users searching for cars within a radius beyond 10, 000 miles, remove the geofilter for those queries. Removing the geofilter does not affect the records returned, but cuts the MDEX Engine response times in half.

  In general, when the MDEX Engine applies a geofilter, it first uses the area's bounding rectangle to reduce the number of records it has to consider, and then performs the computation on remaining records, to determine if the record falls within the specified radius. This computation is expensive. For queries containing a geofilter for very large distances, the bounding rectangle includes all records, which means that the MDEX Engine performs this expensive computation for each record.
- Restrict the number of records returned to speed up MDEX Engine performance.

Range filters

Range filters do not impact the amount of memory needed by the Dgraph. However, because the feature is evaluated entirely at request time, the Dgraph response times are directly related to the number of records being evaluated for a given range filter request.

You should test your application to ensure that the resulting performance is compatible with the requirements of the implementation.

Record filters

Record filters can impact the following areas.

- Spelling auto-correction and spelling Did You Mean. Record filters impose an extra performance cost on spelling auto-correction and spelling Did You Mean.
- Memory cost
- Expression evaluation
- Large `OR` filters ("part lists")
- Large scale negation
- Record filters with complex logic

Record filters: memory cost

The evaluation of record filter expressions is based on the same indexing technology that supports navigation queries in the Dgraph. Because of this, there is no additional memory or indexing cost associated with using navigation dimension values in record filters.

When using property values in record filter expressions, additional memory and indexing cost is incurred because String properties are not indexed for navigation by default.

In some cases, it may be worth replacing some of the filters with dimensions that have the same meaning. For example, if you notice that 20% of queries have a filter of "price > 0" on them, to improve performance, add a `"has price?"` dimension to your records instead of using a filter in this case.

Expression evaluation in record filters: impact on performance

Because expression evaluation is based on composition of indexed information, most expressions of moderate size (that is, tens of terms and operators) do not add significantly to request processing time. Furthermore, because the Dgraph caches the results of record filter operations, the costs of expression evaluation are typically only incurred on the first use of a filter during a navigation session. However, some expected uses of record filters have known performance bounds, which are described in the following sections.

Large OR filters (“part lists”)

One common use of record filters is to specify lists of individual records to identify data subsets (for example, custom part lists for individual customers, culled from a superset of parts for all customers).

The total cost of processing records can be broken down into two main parts: the parsing cost and the evaluation cost. For large expressions such as “part lists”, which are commonly stored as file-based filters, XML parsing performance dominates total processing cost.

XML parsing cost is linear in relation to the size of the filter expression, but incurs a much higher unit cost than actual expression evaluation. Though lightweight, expression evaluation exhibits non-linear slowdown as the size of the expression grows.

`OR` expressions with a small number of operands perform linearly in the number of results, even for large result sets. While the expression evaluation cost is reasonable into the low millions of records for large `OR` expressions, parsing costs relative to total query execution time can become too large, even for smaller numbers of records.

Part lists beyond approximately one hundred thousand records generally result in unacceptable performance (10 seconds or more load time, depending on hardware platform). Lists with over one million records can take a minute or more to load, depending on hardware. Because results are cached, load time is generally only an issue on the first use of a filter during a session. However, long load times can cause other Dgraph requests to be delayed and should generally be avoided.

Large-scale negation

In most common cases, where the `NOT` operator is used in conjunction with other positive expressions (that is, `AND` with a positive property value), the cost of negation does not add significantly to the cost of expression evaluation.

However, the costs associated with less typical, large-scale negation operations can be significant. For example, running top-level negation filtering, such as `“NOT availability=FALSE”` on a record set of several million records leads to lower throughput.

If possible, attempt to rephrase expressions to avoid the top-level use of `NOT` in Boolean expressions. For example, in the case where you want to list only available products, the expression `“availabili¬ty=TRUE”` yields better performance than `“NOT availability=FALSE”`.

Optimizing URL record filters that use complex logic

URL record filters with complex logic may cause an expected growth in memory usage for the MDEX Engine. You can create either a fast-running filter that heavily uses memory, or a slow-running filter that uses minimum memory. This section explains the trade offs and recommends which filter logic you should use.

The filter syntax dictates the sequence in which queries are being run by the MDEX Engine.

Use these recommendations:

- If your goal is to run the record filter as quickly as possible, regardless of concerns for potential memory usage growth on the MDEX Engine server, use the query logic in your filter that is as flat as possible. In other words, use `AND` and `OR` operations directly on the records, and do not use nested operations.

  For example, this filter lists several records directly without any nested operations. It maximizes query performance at the expense of memory usage:

  ```
  Nr=OR(P_WineID:89955,P_WineID:73036,P_WineID:69087,P_WineID:69993,
  P_WineID:60641,P_WineID:58831,P_WineID:44996,P_WineID:52212,
  P_WineID:81192,P_WineID:75040,P_WineID:76632)
  ```

- If your goal is to run the record filter that minimizes memory usage by the MDEX Engine, each `AND` and `OR` statements should contain at most two direct records. Since in many cases you may need to include more than two records in your filters, you can nest `AND` and `OR` operations.

  For example, this heavily nested filter minimizes memory usage at the expense of MDEX Engine query processing time:

  ```
  Nr=OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(P_WineID:89955,P_WineID:73036),
  P_WineID:69087),P_WineID:69993),P_WineID:60641),P_WineID:58831),
  P_WineID:44996),P_WineID:52212),P_WineID:81192),P_WineID:75040),
  P_WineID:76632)
  ```

To summarize, if the data set is large, the filter with flat query logic consumes more memory but runs faster than the filter with nested logic, which runs slower but consumes minimum memory.

If hardware limitations prevent you from accommodating the expected memory growth, change the logic of your existing URL record filter.

EQL expressions and Record Relationship Navigation

You can use Endeca Query Language (EQL) expressions for these purposes.

- To filter query results based on dimension values, individual property values, ranges of property values and search terms.
- To combine EQL expressions using Boolean logic.
- To enable an Endeca feature known as Record Relationship Navigation (RRN).

For more information on EQL and Record Relationship Navigation, see the *Endeca Advanced Development Guide*.

When to use EQL-based filters vs. other filter types

You can use EQL expressions to express all of the filter capabilities that are also supported by range filters (`Nf`), text search filters (`Ntt`, `Ntk`, `Ntx`) and navigation refinements. This topic helps you decide which type of filters to use, EQL-based or regular.

In general, due to their Boolean logic capabilities, EQL expressions offer more flexibility than regular filters expressed through other `UrlENEQuery` parameters. However, EQL expressions have different performance characteristics, and demonstrate other effects that you should take into account when considering which type of filter to implement.

Consider the following characteristics when deciding which type of filters to use, EQL-based or regular:

- **Unless you need EQL filter functionality, use regular filters.**

  In general, when it is possible to express a query using regular filters (range filters and other types), use those methods instead of EQL expressions, as they often provide better query performance. Use EQL expressions after you have evaluated using other features for expressing your query logic.

  In particular:

  - EQL-based filters may be slower than record filters (`Nr`).

    Use record filters (`Nr`) for large filters. (Large filters are used to filter out lists of individual records that identify data subsets, for example custom part lists created for individual customers that are culled from a superset of parts for all customers.) Large filters are better expressed with file-based record filter expressions than with EQL expressions.

  - EQL-based range filters are slower than range filters (`Nf`).

- **To utilize merchandising rules or other supplementary information generated by regular filters, use them alone or in combination with EQL filters.**

  EQL-based filters do not trigger the same supplementary information as a similar refinement navigation or a text search filter. For example, a navigation refinement may trigger merchandising rules, but an EQL filter does not.

  In cases when you want to take advantage of additional information, such as search reports, merchandising rules, DYM and `--whymatch`, use either of the following solutions:

  - Use regular filters.
  - Use EQL expressions in conjunction with other query parameters (such as `N`, `Ntt`, and `Nr` filtering parameters, and `Nf`, `Nrk`, `Nrt`, `Nrr`, and `Nrm` relevance ranking parameters).

    EQL combined with these parameters provides such actions as triggering merchandising rules, sorting, search reports or relevance ranking.

    For examples and information on the feature interaction possibilities, see the *Endeca Advanced Development Guide*.

- **To implement security, use record filters.**

  Use record filters instead of EQL-based filters to implement security filtering, such as filtering based on user role or catalog type. Record filters (`Nr`) are useful also in cases when you want to use file-based filters. (File-based filters are the recommended method for filtering out large numbers of included or excluded records.)

- **To maximize the use of the Dgraph cache, use record filters.**

  Use the `Nr` parameter instead of EQL for those parts of the filter that are static across many queries. This is because static parts of the filter are faster with `Nr` than with EQL, due to the maximized use of the filter cache.

  EQL caches the results of the entire filter, as well as those of a few limited sub expressions. Record filters (`Nr`) also cache the full results of each filter. Thus, if some part of an EQL filter is static across many queries and can be expressed in the language of the `Nr` parameter, it can be advantageous to use `Nr` for that part of the filter so as to maximize use of the cache.

- **For more flexibility, use filters in combination.**

  Use EQL-based filters instead of record filters when you do not require security implications, or when you need more flexibility in expressing filter logic. In this case you may want to improve EQL filter performance by using record filters in conjunction with EQL-based filters, as explained in the next bullet.

- **To narrow down the set of records, use record filters first.**

  Record filters act as pre-filters and narrow down the working set of records for future evaluation by the MDEX Engine. Other expressions in such a query operate only on records returned by a record filter. By comparison, EQL-based filters do not narrow down the working set of records in this way. This has performance implications.

  When evaluating a query, the MDEX Engine first evaluates record filters of type `Nr`, and then all other filters.

Performance impact of EQL-based filters

Use the following recommendations to optimize query performance of EQL-based filters.

- To optimize the performance of EQL-based filters, use record filters in conjunction with EQL-based filters. Use record filters first (`Nr`) if you can, to narrow down the working set, and then use EQL logic to filter within the smaller working set of records.
- Monitor the size of the standard Dgraph request log file. EQL-based filters have verbose syntax. Since all Endeca queries are logged to the standard Dgraph request log, the size of EQL-based queries affects disk space due to the growing size of the Dgraph logs. As an alternative, consider using file-based record filters.
- Identify slow queries during testing. To determine whether an EQL-based filter is slowing down your navigation queries, set the EQL statistics logging in the Dgraph. For example:

  ```
  --log_stats <file_name>
  --log_stats_thresh N
  ```

  This file contains timing for queries taking longer than the specified threshold. Endeca recommends setting a low threshold value during development, and a more conservative value for testing. Do not use statistics logging in production since the verbosity of the logs can cause heavy disk writes and consume available disk space. Look for nodes with large `self_time_ms` values to identify the total time, in milliseconds, spent in this query node and its descendants.

- To optimize EQL query performance, use EQL for queries based on property value instead of queries based on range. For example, if the application's price property contains only 0 or positive values, using an EQL expression to query for “not (price = 0)” provides a better query performance than using queries of type “price > 0”. (This recommendation is true for regular range filters as well.)
- To speed up the MDEX Engine processing of queries, consider implementing the filtering logic in the Forge pipeline. For more complex range expressions, it is more efficient to implement the filtering logic in the Forge pipeline. Use expression logic in a record manipulator or Java manipulator to create a new property with a Boolean value.

  For example, create an “onsale = true” property value if the record has “price > 0” and “price < listprice” properties, and then use the EQL expression to perform a query based on the property value for the newly created property (that is, for “onsale = true”), rather than using EQL for computing range filter expressions on the original properties.

Performance impact of RRN

You can use EQL expressions for Record Relationship Navigation (RRN).

Important: You must configure the MDEX Engine in order to enable Record Relationship Navigation. This capability is an optional module that extends the MDEX Engine. Endeca customers who are entitled by their license to use Record Relationship Navigation can find instructions on the Endeca Support site. Contact your Endeca representative if you need to obtain a Record Relationship Navigation license.

Use the following recommendations to speed up the RRN queries:

- When writing RRN filters, take into account that RRN filter expressions work from the inside out. That is, the innermost, or most nested, expressions are evaluated by the MDEX Engine before the outer ones. The following example illustrates this bottom-up processing:

  ```
  collection()/record
    [
      author_bookref = collection()/record
      [
        book_year = "1843"
      ]
      /book_id
    ]
  ```

  The MDEX Engine first finds the records that have the book_year property set to “1843”. Then it finds the list of all of the values in the book_id property for that set of records. Finally, it finds the set of records with the author_bookref property set to any of the values in that list.

- To speed up RRN queries, assign different property names for records representing different concepts. This is because RRN query performance depends on the number of records in the “nested” EQL query. Keep the number of records that match results for the innermost expression of the RRN filter relatively small. For example, in this query:

  ```
  collection()/record[
   record_type = "Film"
   and
   endeca:matches(., "title", "Godfather")
   and
   actor_id = collection()/record[
     record_type = "Actor"
     and
     gender = "male"
     and
     nationality = "Italian"
  ]
  ```

  the MDEX Engine uses its bottom-up query execution strategy in the following way:

  It first evaluates the inner query and finds the set of records for which the record_type property has the value "Actor," the gender property has the value "male," and the nationality property has the value "Italian."

  It then creates a collection of all the values of the id property for this set of records.

Next, it iterates over the set of “/id” values to filter the set of "Film" records. Thus, if the size of the collection of “/id” values is really large, the iteration can be relatively slow.

In this example, if the number of film IDs that are returned from the innermost filter to the Actor filter is relatively small, the RRN filter that will evaluate these records will be fast; if the number of IDs returned is large, the RRN evaluation will be slow.

To generalize, when you know that the number of records that will have to be evaluated for a RRN filter is quite large (in this example, it is the number of Italian male actors), a query could be slow. To solve this problem, one solution is to use the user interface and force the users to narrow down the set of records early on in the navigation process.

If this is not a reasonable solution for your application, and you cannot guarantee that the user’s navigation path will necessarily limit the set of records, you can narrow down this set by limiting the number of records that match in the innermost query, as shown in this example:

```
collection()/record[
 record_type = "Film"
 and
 endeca:matches(., "title", "Godfather")
 and
 actor_id = collection()/record[
   record_type = "Actor"
   and
   gender = "male"
   and
   nationality = "Italian"
   and
   film_id = collection()/record[
     record_type = "Film"
     and
     endeca:matches(., "title", "Godfather")
    ]/id
  ]/id
]
```

This method is mimicking a top-down execution of a query.

While building an application, test the performance of this inner query with EQL statistics logging to evaluate the time spent in it.

- To speed up RRN queries, assign different property names for different record types of the RRN `collection()/record` function.

  For example, consider this generic RRN query:

  ```
  collection()/record[propertyKey1 = recordPath/propertyKey2]
  ```

  where:

  `propertyKey1` is the NCName of an Endeca property on a record type to be filtered, such as record of type Vineyard. The resulting records will have this property.

  `recordPath` is one or more of the collection()/record functions.

  `propertyKey2` is the NCName of an Endeca property on another record type, such as record of type Wine, that will be compared to `propertyKey1`. Records that satisfy the comparison will be added by the MDEX Engine to the returned set of records.

  In this example, instead of assigning the same value of “ID” for `propertyKey1` and `proper¬tyKey2`, assign two different property names— “wine_reference_ID” on a record representing a vineyard, and “wine_ID” on a record representing a wine. As the number of records evaluated for the RRN query increases, having the naming convention with different property names for different record types has a greater effect on performance.

When properties with the same name are assigned on each side of the RRN query, this negatively affects RRN query performance.

For more information about RRN, see the *Endeca Advanced Development Guide*.

Tips for troubleshooting EQL filters

To detect queries with errors in EQL, check the Dgraph standard error log located at `$`*`ENDECA_PROJECT_DIR/`*`logs/dgraphs/DgraphN/DgraphN.reqlog`.

We recommend using `tail -f` to follow the log during query development.

To troubleshoot EQL filters, use the following recommendations:

- **Watch for disk space limitations**. All Endeca queries are logged to the standard Dgraph request log. Be careful to monitor the size of this log file; there is a risk to run out of disk space due to large log files.
- **Watch for filter length limitations**. The Dgraph process has no limits on the length of a request. The Endeca APIs, however, may have limitations stemming from the programming languages in which they are implemented.
- **Detect slow EQL queries with a dedicated statistics log**. Use these Dgraph flags to enable a special EQL statistics log:
  - `--log_stats` *`[path_to_file]`*
  - `--log_stats_thresh N`

  The log contains an execution plan, including timing, for queries taking longer than the specified threshold. To identify slow EQL queries, in the log, look for nodes with large `self_time_ms` values.

  This statistics logging is turned off by default. Specifying a target for `--log_stats` implicitly turns it on.

  Endeca recommends placing this log in the same directory as all other Dgraph logs, such as in: `$`*`ENDECA_PROJECT_DIR/`*`logs/dgraphs/DgraphN/DgraphN.eqllog`.

  You can specify values for the optional `--log_stats_thresh` argument either as seconds or milliseconds, such as `1s` or `500`. If unspecified, the default is 60 seconds. Endeca recommends setting a low threshold value during development and a more conservative value for testing to capture queries that take longer than the threshold. In general, do not use statistics logging in production, as additional logs can cause operational issues due to heavy disk usage and consumption of available disk space.

Typical causes of EQL filter errors

EQL filter errors are logged into the Dgraph standard error log. This topic lists the most frequent causes of EQL filter errors.

When errors with parsing or syntax occur in EQL filters, they are logged to the Dgraph standard error log located at `$`*`ENDECA_PROJECT_DIR/`*`logs/dgraphs/DgraphN/DgraphN.reqlog`.

Note: When EQL filter errors occur, the query returns zero results and no messages are included in the API response. Therefore, it is important to look into the Dgraph standard error log.

The top issues that may cause errors in the EQL filters are the following:

- **Missing brackets**. Make sure your expressions have matching brackets `[]` and parentheses `()`.
- **Case-sensitivity**. All fields and values are case-sensitive. This includes boolean operators which must be lower case.
- **Property is not indexed properly**. Ensure that you enable properties for record filters in Developer Studio. For any property enabled for record filtering, the Dgidx process creates an inverted index. If a property is not enabled, you may receive an error message like this: `Property "p_name" is not invertible; comparison will fail`.
- **Property or dimension is not an NCName**. For example, `"Wine Type"` is not correct, `"Wine_Type"` or `"WineType"` are correct.
- **Whitespace is present in values**. For example, this is applicable to property value filters: `"Foo " != "Foo"`.
- **You are using an Agraph**. EQL filters are supported only in the Dgraph.

Snippeting

You can minimize the performance impact of snippeting by limiting the number of words in a property that the MDEX Engine evaluates to identify the snippet.

This approach is especially useful in cases where a snippet-enabled property stores large amounts of text. Provide the `--snip_cutoff <num words>` flag to the Dgraph to restrict the number of words that the MDEX Engine evaluates in a property. For example, `--snip_cutoff 300` evaluates the first 300 words of the property to identify the snippet.

If the `--snip_cutoff` Dgraph flag is not specified, or is specified without a value, the snippeting feature defaults to a cutoff value of 500 words.

Spelling auto-correction and Did You Mean

This section discusses tuning the spelling auto-correction and spelling Did You Mean features.

Spelling auto-correction

Spelling auto-correction performance is impacted by the size of the dictionary in use. Spell-corrected keyword searches with many words, in systems with very large dictionaries, can take a disproportionately long time to process relative to other Dgraph requests.

It is important to carefully analyze the performance of the system together with application requirements prior to production application deployment.

Performance of admin?op=updateaspell

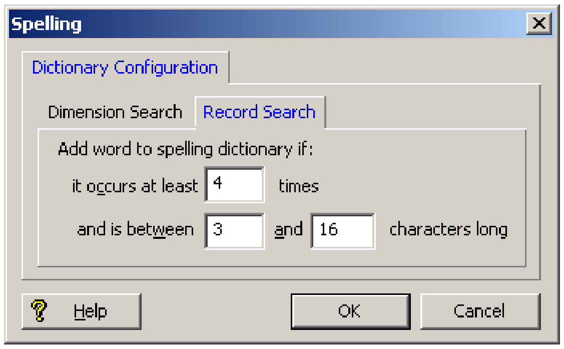
You can use the `admin?op=updateaspell` administrative query to make changes to the Aspell spelling dictionary without having to stop and restart the MDEX Engine. This administrative query causes the MDEX Engine to temporarily stop processing other regular queries, update the spelling dictionary and then resume its regular processing.

If the total amount of searchable text is large, this increases the latency of the `admin?op=updatea¬spell` operation, especially at large data scale.

Dictionary pruning

The performance of spelling correction in the Dgraph depends heavily on the size of the dictionary. An unnecessarily large dictionary can slow response times and provide less focused results.

Dictionary pruning techniques allow you to reduce the size of the dictionary without sacrificing much in the way of usefulness. To improve spelling correction performance, consider making the following adjustments in Developer Studio's Spelling editor:



Add word to spelling dictionary if:

it occurs at least 4 times

and is between 3 and 16 characters long

- Set the minimum number of word occurrences to a number greater than one.

  The first setting in the Spelling editor indicates the number of times a word must occur in the source data in order for it to be included in the dictionary. For record search, the default value is four, which means only words that appear four or more times are included in the dictionary.

- Set the minimum word length to a number greater than one.

  The second setting in the Spelling editor specifies the minimum length (number of characters) of a word for inclusion in the dictionary. By default, words that are longer than three characters and shorter than sixteen characters are included.

  While less dramatic than tuning the minimum word occurrences, adjusting the minimum word length can result in a cleaner, more useful dictionary.

Tuning word break analysis

Word-break analysis allows you to consider alternate queries computed by changing the word divisions in the user's query. The performance impact of word-break analysis can be considerable, depending on your data. Seemingly small deviations from default values, such as increasing the value of `--wb_maxbrks` from one to two or decreasing the value of `--wb-minbrklen` from two to one, can have a significant impact, because they greatly increase the workload on the MDEX Engine. Endeca suggests that you tune this feature carefully and test its impact thoroughly before exposing it in a production environment.

Did You Mean

Lowering the value for `--dym_hthresh` (a Dgraph spelling option) may improve the performance of Did You Mean.

The option `--dym_hthresh` indicates when spelling Did You Mean engages. The default is 20, meaning that spelling Did You Mean engages even if there are up to 20 results.

Depending upon your data, making Did You Mean suggestions at this point may be unnecessary or even overwhelming to your end users. Setting `--dym_hthresh` to 2 or 4 is often a better choice.

Stemming and thesaurus

Stemming and thesaurus equivalences generally introduce little memory overhead (beyond the amount of memory required to store the raw string forms of the equivalences).

In terms of online processing, both features expand the set of results for typical user queries.

While this generally slows search performance (search operations require an amount of time that grows linearly with the number of results), typically these additional results are a required part of the application behavior and cannot be avoided.

The overhead involved in matching the user query to thesaurus and stemming forms is generally low, but could slow performance in cases where a large thesaurus (tens of thousands of entries) is asked to process long search queries (dozens of terms).

Because matching for stemming entries is performed on a single-word basis, the cost for stemming-oriented query expansion does not grow with the size of the stemming database or with the length of the query. However, the stemming performance of a specific language is affected by the degree to which the language is inflected. For example, German nouns are much more inflected than English nouns.

Guidelines for thesaurus development

To avoid performance problems related to expensive and non-useful thesaurus search query expansions, consider the following thesaurus clean-up rules.

- Use `--thesaurus_cutoff <limit>` to set a limit on the number of words in a user's search query that are subject to thesaurus replacement. The default value of `<limit>` is 3. Up to 3 words in a user's search query can be replaced with thesaurus entries. If there are more terms in the query that match thesaurus entries, these terms are not replaced by thesaurus expansion. This option serves as a performance guard against very expensive thesaurus queries. Lower values improve thesaurus engine performance.
- Do not create a two-way thesaurus entry for a word with multiple meanings. For example, *khaki* can refer to a color as well as to a style of pants. If you create a two-way thesaurus entry for *khaki* = *pants*, then a user's search for *khaki towels* could return irrelevant results for pants.
- Do not create a two-way thesaurus entry between a general and several more-specific terms, such as *top* = *shirt* = *sweater* = *vest*. This increases the number of results the user has to go through while reducing the overall accuracy of the items returned.

  In this instance, better results are attained by creating individual one-way thesaurus entries between the general term top and each of the more specific terms.
- Use care when creating thesaurus entries that include a term that is a substring of another term in the entry. Consider the following example with a two-way equivalency between *Adam and Eve* and *Eve*.

  If users type *Eve*, they get results for Eve or (*Adam and Eve*) (that is, the same results they would have gotten for *Eve* without the thesaurus). If users type *Adam and Eve*, they get results for (*Adam and Eve*) or *Eve*, causing the Adam part of the query to be ignored.

There are times when this behavior might be desirable (such as in an equivalency between *George Washington* and *Washington*), but not always.

- Do not use stop words such as and or the in single-word thesaurus forms.

  For example, if the has been configured as a stop word, thesaurus equivalency between thee and the is not useful.

  You can use stop words in multi-word thesaurus forms, because multi-word thesaurus forms are handled as phrases. In phrases, a stop word is treated as a literal word and not a stop word.
- Avoid multi-word thesaurus forms where single-word forms are appropriate.

  In particular, avoid multi-word forms that are not phrases that users are likely to type, or to which phrase expansion is likely to provide relevant additional results. For example, the two-way thesaurus entry *Aethelstan, King Of England (D. 939) = Athelstan, King Of England (D. 939)* should be replaced with the single-word form *Aethelstan = Athelstan*.
- Thesaurus forms should not use non-searchable characters. For example, the one-way thesaurus entry *Pikes Peak* > *Pike's Peak* should only be used if apostrophe (') is enabled as a search character.
- Use `--thesaurus_multiword_nostem` to specify that words in a multiple-word thesaurus form should be treated like phrases and should not be stemmed. This may increase performance for some query loads. Single-word terms will be subject to stemming regardless of whether this flag is specified.

  This flag prevents the Dgraph from expanding multi-word thesaurus forms by stemming. Thesaurus entries continue to match any stemmed form in the query, but multi-word expansions only include explicitly listed forms. To get the multi-word stemmed thesaurus expansions, the various forms must be listed explicitly in the thesaurus.

Record, phrase, and dimension search

This section discusses the performance impact of various kinds of search.

Record search

Because record search is an indexed feature, each property enabled for record search increases the size of the Dgraph process. The specific size of the increase is related to the size of the unique word list generated by the specific property in the data set.

Therefore, only properties that are needed by an application for record searching should be configured as such.

Boolean search

The performance of Boolean search is a function of the number of terms and operators in the query and also the number of records associated with each term in the query.

As the number of records increases and as the number of terms and operators increase, queries become more expensive.

Proximity search impacts the system in various ways. The performance of proximity searches is as follows:

- Searches using the proximity operators will be slower than searches using the other Boolean operators.
- Proximity searches that operate on phrases will be slower than other proximity searches and slower than normal phrase searches.

Note: If you notice unexpected behavior while using Boolean search, use the Dgraph `-v` flag when starting the Dgraph. This flag prints detailed output to `stderr` describing the running Boolean query process.

Phrase search

The cost of phrase search operations depends mostly on how frequently the query words appear in the data and the number of words in the phrase. You can improve performance of phrase search by limiting the number of words in a phrase with the `--phrase_max <num>` flag for the Dgraph.

Searches for phrases containing relatively infrequent words (such as proper names) are generally very rapid.

You can use the `--phrase_max <num>` flag for the Dgraph to specify the maximum number of words in each phrase for text search. Using this flag improves performance of text search with phrases. The default number is 10. If the maximum number of words in a phrase is exceeded, the phrase is truncated to the maximum word count and a warning is logged.

Wildcard search

The MDEX Engine uses a mechanism for wildcard search that simplifies user configuration. In most cases, the size of the on-disk index is reduced considerably, and indexing performance is improved compared with previous releases. This topic provides recommendations for optimizing your wildcard search performance.

To optimize performance of wildcard search, use the following recommendations:

- **Account for increased time needed for indexing**. In general, if wildcard search is enabled in the MDEX Engine (even if it is not used by the users), it increases the time and disk space required for indexing. Therefore, consider first the business requirements for your Endeca application to decide whether you need to use wildcard search.

  Note: To optimize performance, the MDEX Engine performs wildcard indexing for words that are shorter than 1024 characters. Words that are longer than 1024 characters are not indexed for wildcard search.

- **Do not use "low information" queries**. For optimal performance, Endeca recommends using wildcard search queries with at least 2-3 non-wildcarded characters in them, such as `abc*` and `ab*de`. Avoid wildcard searches with one non-wildcarded character, such as `a*`, since they are more expensive to process. Also be aware that the MDEX Engine ignores queries that contain only wildcards, such as `*`. Similarly, wildcard queries that contain only punctuation symbols, spaces and wildcards, such as `*.`, `*'`, or `* *`, are ignored.
- **Analyze the format of your typical wildcard query cases.** This lets you be aware of performance implications associated with one specific wildcard search pattern. Examine your queries to identify whether you have queries that contain punctuation syntax in between strings of text, such as

`ab*c.def*`. For strings with punctuation, the MDEX Engine generates lists of words that match each of the punctuation-separated wildcard expressions. In this case, the MDEX Engine uses the `--wildcard_max <count>` setting to optimize its performance. This setting does not affect wildcard searches for strings which do not contain punctuation.

You enable wildcard search in Developer Studio.

Wildcard search with punctuation and performance

The number of terms to which the MDEX Engine matches the wildcard search strings is limited by the `--wildcard_max <count>` number (the default is 100). This flag lets you specify to the MDEX Engine the maximum number of terms that can match a wildcard term in a wildcard search query that contains punctuation.

When a search reaches the `--wildcard_max` limit, the verbose Dgraph error log records a message similar to the following: `Wildcard term 1*0*.234* is too general: returns 1618 words, which is greater than max of 100. Using the most frequent 100 terms, which took 46.2 ms. to compute.`

Increasing the `--wildcard_max <count>` improves the completeness of results returned by wildcard search for strings with punctuation, but negatively affects performance. Thus you may want to find the number that provides a reasonable trade-off.

If your wildcard search queries contain punctuation, such as `1*0*.234*`, the MDEX Engine generates lists of words that match each of the punctuation-separated wildcard expressions, and uses these non-wildcard terms to locate related results in the documents (records).

This means that if the corpus of data contains other possible matches beyond the `--wildcard_max <count>` (and beyond the results that are already found), the MDEX Engine may not return them as results. Thus, the list of results returned by the Engine in a wildcard search with punctuation may not be exhaustive. This creates a trade-off situation in which you need to optimize performance cost versus business value of maximum completeness of returned results.

To summarize, if the business requirements of your application require a nearly 100% complete list of results even on very "low-information" wildcard queries with punctuation, such as `1*0*.234*`, increase the value of `wildcard_max`. Next, pay attention to the information returned in the search report. From it, you can estimate whether it may make sense to increase the `wildcard_max` value further.

Gradually increase the `--wildcard_max` value, while watching the performance of the MDEX Engine.

Note: If search queries contain only wildcards and punctuation, such as `*.*`, the MDEX Engine rejects them for performance reasons and returns no results.

Preventing expensive wildcard searches

Certain types of wildcard queries may cause the MDEX Engine to grow in memory footprint and take a long time to complete. Even though these types of queries are legitimate searches that would eventually return, they can cause the appearance of a timeout and potentially cause a site outage. As a best practice, Endeca recommends preventing these types of wildcard queries in your front-end application code.

The behavior of such wildcard queries does not typically indicate an actual timeout of the MDEX Engine; instead, it may indicate, for example, that the query search term is so broad that it takes a very long time to compute results. For example, to process a search for `"a*"`, the MDEX Engine must return every record containing any word beginning with a; this is a more time-intensive query for the Dgraph to compute.

The following types of wildcard queries are potentially very expensive to compute for the MDEX Engine:

- Wildcard queries with short search terms, such as `*a*`, `*/*`, or `* *`.
- Wildcard queries with search terms that contain non-searchable characters, such as punctuation or dashes.
- Wildcard queries with search terms that have quoted phrases in them, such as `*"pizza pie"*`.

To prevent users from issuing such types of wildcard queries, utilize front-end application code to circumvent these scenarios for all queries that contain a wildcard character (`*`).

Note: If search queries contain only wildcards and punctuation, such as `*.*`, the MDEX Engine rejects them for performance reasons and returns no results.

Use the following recommendations in the front-end application, by utilizing application code at query time:

1. Remove all non-searchable characters from each wildcard query before issuing it to the MDEX Engine.

   Stripping non-searchable characters should make little difference in your search results because the MDEX Engine treats non-searchable characters as white space both when indexing and when retrieving word matches.

2. Parse the queries to calculate their search term length to avoid very low information queries, such as `"a*"`. For, example, you may want to prevent issuing to the MDEX Engine wildcarding queries that contain fewer than 3 non-wildcarded characters.

   Filtering out such queries should make no difference in your search results because wildcard search for two characters or less would bring back an unusable results set in almost all instances.

3. Exclude wildcard queries with quoted phrase searches. This will not affect your search results because when users issue quoted phrase search, most likely they expect exact matches and do not require wildcards in this case.

You can accomplish these recommendations in the front-end application tier by programmatically analyzing search terms entered by the users before issuing them to the MDEX Engine, determining whether a query will be issued, and prompting the user to submit a better query (or using logic of your choice to handle this situation).

Note: In the majority of cases, none of these changes should impact the user experience.

Dimension search

The runtime performance of dimension search directly corresponds to the number of dimension values and the size of the resulting set of matching dimension values. In general, this feature performs at a much higher number of operations per second than navigation requests.

The most common performance problem occurs when the resulting set of dimension values is exceptionally large (greater than 1,000), thus creating a large results page. Always use the advanced dimension search and query parameters to limit the number of results per request. For details, see “Using Dimension Search” in the *Endeca Basic Development Guide*.

Compound dimension search requests are generally more expensive than non-compound requests, and are comparable in performance to record search requests.

To summarize, if you submit a default dimension search query, the query is generally very fast. If you submit a compound dimension search query, performance is not as fast as for the default dimension search. In both cases, the query will be faster if you limit the results by using any of the advanced dimension search parameters. For example, you can use the `Di` parameter to specify the specific dimension (in the case of the default dimension search), or a list of dimension value IDs (in the case of compound dimension search) for which you expect matches returned by the MDEX Engine.

Note: Do not confuse the Dgraph configuration for dimension search with the Dgraph configuration to enable record search.

Precedence rules

This section discusses precedence rules and explains their performance impact.

About precedence rules

Precedence rules let you limit the presentation of certain Guided Navigation dimensions only to specified navigation states.

You configure precedence rules in Developer Studio.

Each precedence rule lets you identify a trigger dimension value and a target dimension, and presents the target dimension for Guided Navigation only in those query contexts in which:

- Users explicitly select the trigger dimension value as a refinement, or
- The trigger dimension value is assigned to all records in the current result set.

Example of a precedence rule

For example, suppose that an application includes a precedence rule linking the trigger dimension value “Part Category > Passives > Resistors” to a target dimension “Resistance”, which might contain refinements such as “10 ohms” and “22 ohms”.

In a navigation query where, for example, the user performs a search matching records tagged with a variety of values from “Part Category” including “Resistors” and other values, and where the user does not explicitly or implicitly select the dimension value “Part Category > Passives > Resistors”, the “Resistance” dimension is not returned for Guided Navigation.

This prevents the presentation of a contextually irrelevant navigation dimension to the user. Before the user has indicated some interest in resistors, presenting “Resistance” navigation choices may be unexpected, clutter the presentation of more relevant navigation choices, and detract from the overall experience.

If the user subsequently selects the “Part Category > Passives > Resistors” dimension value as a refinement, the “Resistance” dimension is presented for Guided Navigation (assuming that there are valid, available navigation refinements available for “Resistance”). Similarly, if the user performs a search that triggered “Part Category > Passives > Resistors” as an implicit refinement, for example if the user performed a text search for a manufacturer who only makes resistors, the “Resistance” dimension is returned for navigation.

This unique behavior provided by the MDEX Engine allows the contextual presentation of appropriate navigation dimensions to be more automatic and adaptive, as the front-end application need not be aware that the user's search has implied “Part Category > Passives > Resistors” for the “Resistance” dimension to be presented automatically as a navigation dimension.

Relevance ranking

Relevance ranking can impose a significant computational cost in the context of affected search operations (that is, operations where relevance ranking is enabled).

Important: The relevance ranking chapter in the *Endeca Advanced Development Guide* outlines recommended strategies for both retail catalogs and document repositories. If you are developing an Endeca application on your own, you should start with the recommended relevance ranking strategy. Later, if the recommended strategy is not sufficient, you can experiment carefully with strategy tuning.

The set of modules that will provide acceptable performance depends heavily on the size and characteristics of the application data set.

In general, Endeca recommends testing the set of modules used for relevance ranking in a staging environment before using it in production. This is because the qualities of the data set may affect relevance ranking performance in unexpected ways. The following characteristics of the data set may negatively affect performance:

- The data set is too large to fit into RAM
- It contains large file content used in search
- It uses stemming or thesaurus heavily
- It has many dimensions or properties per record
- It frequently produces large result set sizes

Minimizing the performance impact of relevance ranking

You can minimize the performance impact of relevance ranking in your implementation by making module substitutions when appropriate, and ordering the modules you do select sensibly within your relevance ranking strategy.

Making module substitutions

Because of the linear cost of relevance ranking in the size of the result set, the actual cost of relevance ranking depends heavily on the set of ranking modules used. In general, modules that do not perform text evaluation introduce significantly lower computational costs than text-matching-oriented modules.

Although the relative cost of the various ranking modules is dependent on the nature of your data and the number of records, the modules can be roughly grouped into four tiers:

- Exact is very computationally expensive.
- Proximity, Phrase with Subphrase or Query Expansion options specified, and First are all high-cost modules, presented in the order of decreasing cost.
- WFreq can also be costly in some situations.
- The remaining modules (Static, Phrase with no options specified, Freq, Spell, Glom, Nterms, Interp, Numfields, Maxfields and Field) are generally relatively cheap.

In order to maximize the performance of your relevance ranking strategy, consider a less expensive way to get similar results. For example, replacing Exact with Phrase may improve performance with relatively little impact on results.

Note: Choose the set of modules used for relevance ranking most carefully when the data set is large or contains large file content that is used for search operations.

Ordering modules sensibly

Relevance ranking modules are only evaluated as needed. When higher-priority modules determine the order of records, lower-priority modules do not need to be calculated. This can have a dramatic impact on performance when higher-cost modules have a lower priority than a lower-cost module.

To optimize performance, make sure that the cheaper modules are placed before the more expensive ones in your strategy.

Dynamic business rules

Dynamic business rules (used in merchandising and content spotlighting) require very little data processing or indexing, so they do not impact the Dgraph memory footprint.

However, because the MDEX Engine evaluates dynamic business rules at query time, the larger the number of rules, the longer the evaluation and response time.

To improve query response-time performance of the Dgraph with dynamic business rules:

- Monitor and limit the number of rules that are evaluated for each request. Each rule that is evaluated for a request impacts the response time for that request.

  To do this, specify the number of records returned in the Maximum Records text box of the Styles editor in Developer Studio. Setting the Maximum Records value prevents business rules from returning an entire set of matching records, potentially overloading the network, memory, and page size limits for a request. If the Maximum Records value is set to a large number, such as 1,000, then as many as 1,000 promoted records could be returned with each navigation request, causing significant performance degradation.
- Use `Nmrf` to specify the syntax for the rule filter. Rule filters restricts which rules can promote records for a navigation query. The `Nmrf` query parameter controls the use of a rule filter. `Nmrf` has a corresponding ENEQuery method and parameter.
- Set a rule limit for each rule zone.
- Configure triggers for all business rules. Business rules without triggers are evaluated for every navigation query and negatively affect performance.
- Review how rule sorting is used. Rule sorting allows you to sort the rule’s promoted records by a specified property or dimension value. Per-rule sorts can increase the performance cost of dynamic business rules.

Agraph performance considerations

Ideally, the Agraph speeds up the processing of requests in the MDEX Engine by a factor of the number of partitions. The MDEX Engine achieves close to the ideal speed-up for handling expensive requests, especially analytics requests.

For smaller requests, the overhead of the Agraph tends to nullify the benefits of parallelizing the computation.

Dynamic business rules and the Agraph

If you are using dynamic business rules with the Agraph, you may experience a loss of performance if you are using unique zones combined with high maximum values (for the number of records returned) or large numbers of rules. To avoid a slowdown in Dgraph operation, you may need to reduce the rule count, reduce the number of records returned or abandon uniqueness.

Paging and the Agraph

The combination of paging (particularly deep paging) and the Agraph can be slow.

Analytics performance considerations

This section explores issues related to optimizing performance of Analytics queries.

Important: Endeca Analytics is a separate module that extends the MDEX Engine. You should configure the MDEX Engine in order to enable Analytics. Endeca customers who are entitled by their license to use Analytics can find instructions on the Endeca Support site. Contact your Endeca representative if you need to obtain an Endeca Analytics license.

For more information about how to use Endeca Analytics functions, and for examples and best practices, see the *Endeca Analytics Guide*, and the solution article "Analytics Considerations and Best Practices" available online from the Endeca Developer Network (EDeN).

Each of the following considerations has an impact on the Analytics query performance:

- Review existing Analytics queries to understand their processing order and Analytics statement dependencies. For example, you may improve query performance if you narrow down the working record set which Analytics statements must process.

  When a query contains an Analytics query, the Analytics processing is one of the last steps in the overall query processing order. The Analytics statements are calculated on the resulting record set (`NavStateRecords`) after any search, navigation, or filtering has been applied by the Endeca query. This has performance benefits, since the fewer records the Analytics statements need to process, the better.

- Test Analytics queries that contain a `GROUP BY` operation to measure RAM footprint and query response time. This will help identify the size of a result set that does not negatively affect performance.

  `GROUP BY` operations result in a large number of aggregated records that are stored within the Dgraph RAM. This may cause an increase in the RAM footprint and the Dgraph processing time. It may be necessary to tune `GROUP BY` operations within Analytics statements in your queries.

- Build Analytics queries in a way that lets them utilize the caching of Analytics statements used in more than one query.

  The Dgraph dynamic cache stores Analytics statements. If statement dependencies exist in your queries, you can utilize previously computed data within other Analytics statements. If one Analytics query includes multiple Analytics statements, each statement is cached separately, which results in a significant performance gain in cases when specific Analytics statements are shared across multiple queries.

Analytics and the Agraph performance considerations

If you are using an Agraph, consider which Analytics functions should be performed by an Agraph versus those that should be processed by child Dgraph processes. From a performance standpoint, the more work the Dgraphs can perform, the more efficient the query is.

Consider the following cases:

- Be aware that Analytics statement dependencies that occur in an Agraph require that all of the lookup processing occurs in the Agraph. This results in slower query processing.
- Review how you use different types of aggregate functions. The aggregate functions are calculated at different stages of Analytics processing. Understanding the differences between the functions can be useful when optimizing the performance of the Analytics statements. In particular, logically distribute `SUM`, `COUNT`, `AVG` and `COUNTDISTINCT` functions that are used in your Analytics queries between Dgraphs and an Agraph to ensure that they are optimized for performance.

  For example:

  - Each of the Dgraphs can process its own `SUM`, while an Agraph computes the resulting `SUM` of `SUMS`.
  - Similarly, Dgraphs can perform `SUM` and `COUNTS` on their records, while an Agraph computes the resulting `SUM of SUMs` and `SUM of COUNTs`, as well as `AVG`.
  - For more complex functions such as `COUNTDISTINCT`, each Dgraph can send a full list of its unique values to the Agraph, while the Agraph determines the `COUNTDISTINCT` across all Dgraphs responses.

  Note: For more information about aggregate functions, see the solution article *“Analytics Considerations and Best Practices”* available online from the Endeca Developer Network (EDeN).

- Revise your data partitioning strategy to account for Analytics queries by choosing the most appropriate partition key.

  In an Agraph architecture, you can partition the data by specifying a total number or partitions and a partition key based on which the data will be split.

  If this partition key is included in the `GROUP BY` clause, the Dgraphs perform significantly more of the work, which optimizes Analytics performance. Therefore, when possible, the partition key should be a property that is used within a `GROUP BY` clause most frequently, or that is expected to return a large number of results.

  For this optimization to work properly, the Endeca configuration files must have only one `Rollup` key defined, and it must be the key that the data was partitioned on. If the configuration is not created properly, the Agraph may try to optimize the query when it should not, resulting in the Agraph returning multiple aggregate records for the same `GROUP BY` value.

Appendix A

The MDEX Engine Request Log

This section describes the MDEX Engine (Dgraph) request log, which you can use to analyze Endeca application performance.

About the MDEX Engine request log

The MDEX Engine request log (also called the Dgraph request log) is the file that captures Web application query information.

The MDEX Engine always generates a request log with a default name `dgraph.reqlog`. You use the `--log` option when running the MDEX Engine to specify a different path to store the request log.

You can extract queries from this log file and use them with the Endeca Eneperf tool to analyze Web application performance. You can also use Perl to extract useful information from Dgraph request logs.

In addition, depending upon the size of your log files, you can import them into a tool that allows you to manipulate column-based data, such as Microsoft Excel.

Related Links

Extracting information from request logs on page 89

MDEX Engine request logs can be very large and difficult to read. You might find it useful to sort them on fields you are interested in, such as Processing Time or Total Request Duration. You can then look for a pattern or feature in the most time-consuming queries that might be the origin of the performance issue.

Request log file format

The content of the request log file varies slightly, depending upon whether it is treating Presentation API queries or Web services invocations.

Note: If a field is not relevant to the query in question, the request log entry for that query contains a dash (-) in that location.

Each entry has the following 14 columns:

```
[Timestamp] [Client IP Address] [Agraph Transaction ID]
[HTTP Exchange ID] [Response Size] [Total Request Time]
[Total Processing Time] [HTTP Return Code] [Number of Results]
```

```
[Queue Status] [Thread ID] [Query String] [Query Body]
[HTTP Headers]
```

These entries are listed in the order of the timestamp. Because of this, the entries are listed in the response order, not in the request order. The following table describes the log entries in more detail:

Column	Presentation API Queries or Web Services Invocations	Description
Timestamp	Both	Time stamp indicating the time the request was completed, in milliseconds, since the epoch (January 1, 1970, 00:00:00 UTC). For example: 1208947882000=2008-04-23 10:51:22 AM GMT The time is recorded in GMT (not the localized time of the server). You can convert it using a UTC epoch converter utility, such as UTC.
Client IP	Both	IP address of the requesting client.
Agraph Transaction ID	Presentation API Queries	Agraph transaction identifier. **Note:** This field is always empty unless the Dgraph is running under an Agraph.
HTTP Exchange ID	Both	Unique query identifier. This identifier allows you to correlate Dgraph request log items with error messages in the Dgraph log. In addition, it is used by the MDEX Server Statistics page to compose most expensive query statistics. **Note:** The identifier is only unique within a single Dgraph instance, and is not persistent across Dgraph shutdown.
Response Size	Both	Number of bytes written to the client. May be less than or equal to the intended result size, for example, due to a premature session end.
Total Request Time	Both	The request lifetime, in milliseconds. Equal to the total amount of time between when the Dgraph reads the request from the network and finishes sending the result. May include queuing time, such as time spent waiting for earlier requests to be completed. **Note:** In previous releases, the request lifetime ended when the connection was closed. If connection close did not time out, this lifetime would include the time to transport the response to the client, and the time for the client to read the response. Starting with 6.1.0, the request lifetime ends when the response has been successfully delivered to the socket layer.

Column	Presentation API Queries or Web Services Invocations	Description
Total Processing Time	Both	Processing time, in milliseconds. Equal to the total computation time required for the Dgraph to handle the request, excluding network and wait time. This value gives an accurate measure of how expensive the request was to compute, given current system state. (That is, if the machine in question was busy with other threads or processes, the time may be longer than on an otherwise unused machine.) For any given query, Processing Time is always smaller than Total Request Time.
HTTP Status Code	Both	The HTTP return code. A status code of 200 (OK) is returned if the request was successful. For details on other codes that can appear in this field, see the table below.
Number of Results	Presentation API Queries	Number of results from your query (or "`-`" if the HTTP request was not a query). **Note:** This number reflects the number of results, not necessarily the number of results returned. That is, this is the number of results from your query, not accounting for your `nbins` and `offset` settings. `nbins` and `offset` are used to specify how many of the results are actually returned.
Queue Status	Both	The number of threads busy when the request was received. This number is calculated when the request was received, thus this request is not included in this number. In previous releases, this column reported the number of threads busy (when Q was a positive number), or the number of query threads that are idle (when Q was a negative number). Starting with the MDEX Engine version 6.1.2, this column does not report the number of query threads that are idle because there is no longer a one-to-one relationship between threads and queries. Specifically, when you use the `--threads` flag to specify the number of threads to the MDEX Engine, the number you specify determines the total number of threads available to the MDEX Engine, which includes query processing threads and other threads that support query processing. As an implication, there is a greater chance that a non-saturated Dgraph could experience minor queuing, even in the case when the number of query requests in the queue is less than the number of threads specified. For more information, see the chapter in this guide about using the multithreaded mode.

Column	Presentation API Queries or Web Services Invocations	Description
Thread ID	Both	The thread ID of the thread that was assigned the request (or “`-`” in single-threaded mode).
Query String	Both	The URL of the Presentation API query or of the Web service.
Query Body	Web Services Invocations	The URL-encoded POST body of the query. The actual entry in the request log is a single token, even though POST body can contain multiple lines of text.
HTTP Headers	Both	The URL-encoded HTTP headers that were sent with the query. The actual entry in the request log is a single token, even though HTTP headers can contain multiple lines of text.

Non-OK HTTP Status Codes

This table details the non-OK HTTP Status Codes that might appear in the Request Log.

Status Code	Name	Condition
100	Continue	In response to HTTP request header Expect: 100-continue (not an error)
400	Bad Request	Admin or config request with unsupported op
400	Bad Request	HTTP request line parse error, or HTTP request header parse error, or HTTP request Transfer-Encoding other than chunked
400	Bad Request	HTTP request with invalid chunk size or missing chunk terminator
400	Bad Request	HTTP request with invalid trailing header format
400	Bad Request	HTTP request with wildcard URL ("*") not valid for METHOD
400	Bad Request	HTTP request URL includes protocol other than "http", or protocol but no host, or neither protocol nor host and path does not start with "/"
400	Bad Request	HTTP request with version 1.1 but no Host
400	Bad Request	HTTP request with more data than expected
400	Bad Request	Conversion of POST body to string failed for web service request
403	Forbidden	Admin ops are disabled for the Dgraph, and `admin?op=exit` or `ad¬min?op=restart` is requested
404	Not Found	Presentation API request with URI parse error or processing error
404	Not Found	Request has empty path, or admin or config request has additional path steps
404	Not Found	File server request for non-existent file or for a directory, or file outside of allowed root directory
404	Not Found	Web service request for unknown Web service

Status Code	Name	Condition
408	Request Time-out	Queue timeout exceeded for the request, or I/O timeout reading HTTP request
410	Gone	Presentation API request for unsupported feature
411	Length Required	HTTP POST request with Content-Length missing or empty or not a non-negative integer
412	Precondition Failed	HTTP request with "If-None-Match" header
415	Unsupported Media Type	Content-Type parse error in Web service request
500	Internal Server Error	Attempt to return informational status code to HTTP 1.0 client
500	Internal Server Error	Exception from XQuery evaluation in Web service request
500	Internal Server Error	Unhandled exception during request processing
500	Internal Server Error	`admin?op=update` is requested and no update directory was specified for the Dgraph
501	Not Implemented	HTTP request for unsupported Method (such as PUT)
501	Not Implemented	HTTP request includes an unsupported header that must not be ignored: ("Authorization", "Content-Encoding", "Content-Transfer-Encoding", "Range", "Content-Range", "If-Range")
501	Not Implemented	Presentation API request for disabled feature
503	Service Unavailable	HTTP request to server that is closed (in the process of shutting down)
505	HTTP Version Not Supported	HTTP request with version not "1.0" and not "1.1"

Related Links

List of request log parameters on page 96
This section lists request log parameters.

Extracting information from request logs

MDEX Engine request logs can be very large and difficult to read. You might find it useful to sort them on fields you are interested in, such as Processing Time or Total Request Duration. You can then look for a pattern or feature in the most time-consuming queries that might be the origin of the performance issue.

Here are two approaches to extract information from request logs:

- Run the Cheetah script available from the Endeca Developer Network (EDeN).

- Write your own Perl code.

The Cheetah script reads one or more MDEX Engine logs and reports on the nature and performance of the queries recorded in those logs. This report provides information on what actually happened in the past, instead of reporting on potential performance or capacity planning for the future. This script can be run manually in order to debug performance problems, and should also be run on a regular basis to continually monitor performance and call out trends in Dgraph traffic load, latency, throughput, and application behavior.

The Cheetah script is available from the Downloads section of EDeN under Tools and Utilities.

If you write Perl to extract, manipulate, and analyze the information in a request log, you may find the following setting useful in Perl scripts:

```
perl -nae
```

where:

- n indicates that it is a loop processing each line of the input file(s) in turn
- a turns on autosplit
- e indicates that it should execute the next argument, which should be Perl code

This script shows how many queries took more than five seconds. It splits the line on whitespace into an array called F. The sixth element in the array ([5]) corresponds to the Total Request Time and represents the amount of time the query took.

```
perl -nae 'print if $F[5] > 5000' logfile
```

If you are tracking system trends by time, you may find it useful to correlate the epochal time that the log displays with human-readable time. This script is used to convert the time stamps into a more readable form.

```
perl -nae 'print scalar localtime $F[0]," $_"'
```

Note: In this script, `Localtime` is set to the location where you are doing analysis, so if you are looking at a log from a different time zone, you may want to change the time zone. On UNIX systems the TZ environment variable can be set to effect this change. For example, `TZ=US/Pacific`.

Storing logs on a separate physical drive

There can be disk contention between MDEX logging and update processing that can cause sporadic increases in query processing latencies. Update processing includes both partial update processing and merging generations. One way to minimize disk contention is to store MDEX logs, such as the error log, request log, and update log on a separate physical drive from where the MDEX indices are stored.

To store logs on a separate physical drive:

- For the error log, specify the `--out <stdout/stderrfile>` flag to the Dgraph with a path to a different physical drive from the MDEX indices.
- For the request log, specify the `--log` flag to the Dgraph with a path to a different physical drive from the MDEX indices.
- For the update log, specify the `--updatelog` flag to the Dgraph with a path to a different physical drive from the MDEX indices.

Request log rolling

The MDEX Engine request log is subject to log rotation when it goes over one gigabyte.

When this occurs, the existing logfile is renamed from, say, `dgraph.reqlog` to `dgraph.reqlog.PID.N`, where:

- PID is the Dgraph process ID
- N is the number of logs that this Dgraph has already rotated. N=0 the first time the Dgraph does log rotation, and then goes up by 1 each time.

Appendix B

The MDEX Engine Parameter Listing

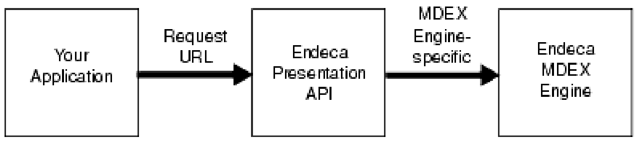
This section describes the parameters in the MDEX Engine request logs and provides mappings between the URL that is sent from the application to the Endeca Presentation API, and the URL that is sent from the API to the MDEX Engine.

Understanding the URL parameter mapping

Typically, when you analyze the MDEX Engine request query logs for troubleshooting purposes, you investigate a log entry for a query in question, and identify an MDEX Engine parameter in the query's log entry.

Next, you want to trace this log parameter to its corresponding settings in the user-visible URL that is sent from the application to the Endeca Presentation API and the URL that is sent from the API to the MDEX Engine. There is not a one-to-one correlation between the two URLs.

The Presentation API transforms the URL it receives from the application into an MDEX Engine-specific URL before sending it to the MDEX Engine.



Mappings between request log and UrlENEQuery URL parameters

This explains a mapping between the URL that is sent from the application to the Endeca Presentation API, and the URL that is sent from the API to the MDEX Engine.

It helps you translate the MDEX Engine request log file, which tells you exactly which URLs the MDEX Engine has processed. By extension, these are the URLs that the Presentation API has sent to the MDEX Engine. If the API has sent an incorrect URL to the MDEX Engine, it is a good indication that the API received an incorrect URL from the Web application in the first place.

Note: For a complete description of the ENE URL query parameters, see the *Endeca Advanced Development Guide*.

Example mappings

Here are some sample mappings:

Web Application to API	API to MDEX Engine
`/controller.jsp?N=0`	`/graph?node=0`
`/controller.jsp?N=0&Ntk=DESC& Ntt=merlot`	`/graph?node=0+attrs=DESC+merlot`

Mapping parameters

The table in this section establishes a mapping between those MDEX Engine request log parameters that have corresponding `UrlENEQuery` URL parameters, such as `N` and `Ntt`.

Not all request log parameters have corresponding `UrlENEQuery` URL parameters. This table does not list those MDEX Engine request log parameters that do not have directly corresponding end-user parameters. It also does not indicate which methods or properties of the `ENEQuery` objects can be used to produce the specified request log parameters.

In this table, the ENE parameters in bold are the primary parameters, while those in non-bold are secondary parameters.

MDEX Engine parameter	Description	Maps to...
graph?	**Navigation query**	**N**
node	Navigation query parameter, navigation descriptors	N
offset	Navigation query parameter, record offset	No
offset	Navigation query parameter, aggregated record offset	Nao
group	Navigation query parameter, exposed refinements	Ne
allbins	Navigation query parameter, records per aggregated record	Np
analytics	Navigation query parameter, analytics expression to apply to a query	Na
sort	Navigation query parameter, sort	Ns
sort	Navigation query parameter, sort order	Nso
groupby	Navigation query parameter, rollup	Nu
attrs	Navigation query parameter, record search key, terms, and options	Ntk, Ntt, Ntx
relrank	Navigation query parameter, search interface, relevance	Nrk, Nrt, Nrr, Nrm

MDEX Engine parameter	Description	Maps to...
	ranking terms, relevance ranking strategy and match mode	
dym	Navigation query parameter, Did You Mean	Nty
autophrase	Navigation query parameter, compute phrasings	Ntpc
autophrasedwim	Navigation query parameter, rewrite query	Ntpr
merchpreviewtime	Navigation query parameter, merchandising preview time	Nmpt
merchrulefilter	Navigation query parameter, merchandising rule filter	Nmrf
pred	Navigation query parameter, range filters	Nf
filter	Navigation query parameter, record filters	Nr
structured	Navigation query parameter, Endeca Query Language	Nrs
refinement	Navigation query parameter, dynamic refinement ranking	Nrc
search?	**Dimension search query**	**D**
terms	Dimension search query parameter, search terms	D
options	Dimension search query parameter, options	Dx
node	Dimension search query parameter, dimension search scope	Dn
model	Dimension search query parameter, search dimension	Di
num	Dimension search query parameter, number of results	Dp
offset	Dimension search query parameter, offset	Do
rank	Dimension search query parameter, rank	Dk
pred	Dimension search query parameter, range filters	Df
filter	Dimension search query parameter, record filters	Dr

MDEX Engine parameter	Description	Maps to...
structured	Dimension search query parameter, Endeca Query Language	Drs
abin?	**Aggregated record query**	**A**
id	Aggregated record query parameter, record ID	A
node	Aggregated record query parameter, descriptors	An
groupby	Aggregated record query parameter, rollup	Au
pred	Aggregated record query parameter, range filters	Af
filter	Aggregated record query parameter, record filters	Ar
structured	Aggregated record query parameter, Endeca Query Language	Ars
bin?	**Record query**	**R**
id	Record query parameter, record ID	R

List of request log parameters

This section lists request log parameters.

It provides the following information:

- Lists the request log parameters and explains what they do.
- Identifies how the request log parameters correspond with the end user visible URL parameters. In other words, a mapping is established between the parameters that are visible in the end-user URL, known as the `UrlENEQuery` URL parameters, such as `N` and `Ntt`, and the parameters that are present in the request log, such as node and attrs.
- Lists those request log parameters that do not have directly corresponding end-user parameters, such as allgroups and nbins.
- Indicates which methods or properties of the `ENEQuery` objects can be used to produce the specified request log parameters.

  In general, in your application, you use either the `UrlENEQuery` URL parameters, such as `N` and `Ntt`, or the methods or properties of the `ENEQuery` object class. In either case, both methods produce the MDEX request log parameters described in this section.

Example: interpreting error log messages

This example illustrates how to interpret the messages found in the MDEX Engine error log.

Suppose the following messages appear in your MDEX Engine error log:

```
ERROR 06/04/08 18:13:33.250 UTC DGRAPH {dgraph}: Bad dimension or property
 name
[WineType] in select
```

To troubleshoot, look through the corresponding MDEX request log for entries that contain “select” and “WineType”. The results are as follows:

```
1212603213 127.0.0.1 - 3378 105.54 7.49 200 56300 -2 10
/graph?node=0&select=P_Name+P_Score+WineType&group=0&offset=0&nbins=10&pred=
P_Score%7CGTEQ+70&irversion=510
```

Check the documentation in this section for the select parameter that appears in the MDEX Engine URL, in the request log. You will find that it corresponds to the Java API `ENEQuery.setSelection()` method; there is no corresponding `UrlENEQuery` URL parameter. This means that the incorrect value is set through this method. You can now look through the application code and find the `setSelec¬tion()` call to try to determine why it is specifying an incorrect property or dimension name as part of the value for this method. In this example, it is because the code is specifying "WineType" rather than "Wine Type" with a space.

Description of query types

The parameters in the MDEX request log use the query type names that correspond to the types of user queries. This section and the table below list the query types and maps them to user queries.

Query type as indicated in the request log	Description of the corresponding user query type
`admin`, `config`	Administrative query
`bin`, `abin`	Record query
`graph`	Navigation and record search queries that return navigation data
`search`	Dimension search queries only

agreq

Description	Identifier string for an Agraph query; is used to correlate entries in Dgraph request logs with those from the Agraph request log. `agreq` is set automatically by the Agraph.
Valid in query types	`graph`, `search`, `bin`, `abin`
ENEQuery method or property	N/A
UrlENEQuery URL parameters	N/A
Format	Underscore-separated string values

Values (order)	UNIX timestamp, IP address, port, sequential count
Example	`agreq=1184080225_172.30.20.117_8888_2`

allbins

Description	Specifies the number of representative records returned with each aggregated record.
Valid in query types	graph
ENEQuery method or property	Java: `ENEQuery.setNavErecsPerAggrERec()` .NET: `ENEQuery.NavERecsPerAggrERec`
UrlENEQuery URL parameters	Np
Format	Numeric value
Values (order)	0 (no representative records), 1 (one representative record), 2 (all records associated with aggregated record). Value "0" equates to API constant ENEQuery.ZERO_ERECS_PER_AGGR, "1" to `ENEQuery.ONE_EREC_PER_AGGR` "2" to `ENEQuery.ALL_ERECS_PER_AGGR`
Example	N/A

allgroups

Description	Specifies whether child refinements are exposed for all dimension values. Takes precedence over group if both are specified. The API includes one parameter or the other. **Note:** allgroups=1 in the Dgraph URL can cause significant impact on performance of the MDEX Engine and indicates that all refinements are exposed for navigation. If you notice this setting in the queries check the validity of this setting for the application.
Valid in query types	graph
ENEQuery method or property	Java: `ENEQuery.setNavAllRefinements()` .NET: `ENEQuery.NavAllRefinements`

UrlENEQuery URL parameters	N/A
Format	Numeric Boolean value
Values (order)	0 (false), 1 (true)
Example	N/A

analytics

Description	Specifies an analytics expression to apply to a query.
Valid in query types	graph
ENEQuery method or property	Java: `ENEQuery.setAnalyticsQuery()` .NET: `ENEQuery.AnalyticsQuery`
UrlENEQuery URL parameters	`Na`
Format	String analytics expression
Values (order)	N/A

Example	`analytics=Q%28A%28Test%28T%29SL`
	`%28S%28%28Vintage%29KEY%28Vin¬ tage%29%29%29%29%29`

attrs

Description	Specifies search key, terms, and options for record searches
Valid in query types	`graph`
ENEQuery method or property	Java: `ENEQuery.setNavERecSearches()` .NET: `ENEQuery.NavERecSearches`
UrlENEQuery URL parameters	`Ntk`, `Ntt`, `Ntx`
Format	Space-separated string values for search key, literal plus character separator, space-separated string values for search terms, pipe character separator, space-separated string values for search options (`mode`, `rel`, and `autoforce`).
Values (order)	See above
Example	`attrs=Inter¬ face+search+terms|mode+matchall+rel+ex¬ act+autoforce+correction`

autoforce

Description	Is set automatically by the Agraph. **Note:** Starting with the Endeca IAP v.5.1, `autoforce` is a search option specified in `attrs`, and is no longer a separate URL parameter. Specifies to a child Dgraph to force a spelling auto-correction to the specified term or terms. Used on Agraph re-queries to unify autocorrection results across all child Dgraphs.
Valid in query types	N/A
ENEQuery method or property	N/A
UrlENEQuery URL parameters	N/A
Format	N/A
Values (order)	N/A
Example	N/A

autophrase

Description	Specifies whether the MDEX Engine computes autophrase matches for search terms.
Valid in query types	graph
ENEQuery method or property	Java: ENEQuery.`setNavERecSearchComputeAlternativePhrasings()` .NET: `ENEQuery.NavERecSearchComputeAlternativePhrasings`
UrlENEQuery URL parameters	`Ntpc`
Format	Numeric Boolean value
Values (order)	0 (false), 1 (true)
Example	N/A

autophrasedwim

Description	Specifies whether the MDEX Engine replaces phrases found in search terms with computed autophrase matches. Is functional only if the autophrase parameter is also set to 1 (true).

Valid in query types	graph
ENEQuery method or property	Java: `ENEQuery.setNavERecSearchRewrite¬ QueryWithAnAlternativePhrasing()` .NET: `ENEQuery.NavERecSearchRewrite¬ QueryWithAnAlternativePhrasing`
UrlENEQuery URL parameters	`Ntpr`
Format	Numeric Boolean value
Values (order)	0 (false), 1 (true)
Example	N/A

compound

Description	Specifies whether dimension search is performed as a compound dimension search.
Valid in query types	search
ENEQuery method or property	Java: `ENEQuery.setDimSearchCompound()` .NET: `ENEQuery.DimSearchCompound`
UrlENEQuery URL parameters	N/A
Format	Numeric Boolean value
Values (order)	0 (false), 1 (true)
Example	N/A

dym

Description	Specifies whether "did you mean" (DYM) spelling correction is enabled for a record search.
Valid in query types	graph
ENEQuery method or property	Java: `ENEQuery.setNavERecSearchDidY¬ ouMean()` .NET: `ENEQuery.NavERecSearchDidYouMean`
UrlENEQuery URL parameters	`Nty`
Format	Numeric Boolean value
Values (order)	(order) 0 (false), 1 (true)
Example	N/A

filter

Description	Specifies record filter to apply for navigation, dimension-search, or aggregated-record (abin) queries.
Valid in query types	graph, search, abin
ENEQuery method or property	(graph) Java: `ENEQuery.setNavRecordFilter()` .NET: `ENEQuery.NavRecordFilter` (search) Java: `ENEQuery.setDimSearchNavRecordFil¬ter()` .NET: `ENEQuery.DimSearchNavRecordFilter` (abin) Java: `ENEQuery.setAggrERecNavRecordFil¬ter()` .NET: `ENEQuery.AggrERecNavRecordFilter`
UrlENEQuery URL parameters	`Nr` (graph), or `Dr` (search), or `Ar` (abin)
Format	String values separated by plus signs
Values (order)	String values
Example	`filter=P_Region%3aPortugal, fil¬ter=8021`

format

Description	Description Specifies result object return format for a query. **Note:** Format can only be set by hand. XML schema is unsupported and is subject to change.
Valid in query types	graph, search, bin, abin
ENEQuery method or property	N/A
UrlENEQuery URL parameters	N/A
Format	String value

Values (order)	binary (default) or XML
Example	N/A

group

Description	Specifies dimension values for which child refinements should be exposed; Overridden by allgroups if both are specified. The API includes one parameter or the other. Only a single dimval from any given dimension can be specified (even if the dimension is configured for multiselect).
Valid in query types	graph
ENEQuery method or property	Java: `ENEQuery.setNavExposedRefine¬ments()` .NET: `ENEQuery.NavExposedRefinements`
UrlENEQuery URL parameters	Ne
Format	Space-separated numeric dimval IDs
Values (order)	Numeric dimval IDs
Example	`group=123+3893+1232123`

groupby

Description	Specifies rollup (aggregation) key to apply for navigation or aggregated-record queries.
Valid in query types	graph, abin
ENEQuery method or property	(graph) Java: `ENEQuery.setNavRollupKey()` .NET: `ENEQuery.NavRollupKey` (abin) Java: `ENEQuery.setAggrERecRollupKey()` .NET: `ENEQuery.AggrERecRollupKey`
UrlENEQuery URL parameters	`Nu` (graph), or `Au` (abin)
Format	Space-separated string property or dimension names
Values (order)	String property or dimension names

Example	`groupby=My+DimName, groupby=P_Winery`

id

Description	Specifies a record to return (by record spec value or other identifier). **Note:** Aggregated-record (abin) queries only support a single record identifier, not a space-separated list.
Valid in query types	bin, abin
ENEQuery method or property	(bin) Java: `ENEQuery.setERecs()` .NET: `ENEQuery.ERecs` (abin) Java: `ENEQuery.setAggrERecSpec()` .NET: `ENEQuery.AggrERecSpec`
UrlENEQuery URL parameters	`R` (bin), or `A` (abin)
Format	Space-separated string values
Values (order)	String values
Example	`id=18114, id=Record+23, id=2+73`

ignore

Description	Specifies whether the Dgraph ignores missing dimension value IDs in a query. When set to false, queries with missing dimval IDs fail with "Invalid category id… in query" errors; when set to true, such queries return successfully with "Detected missing category… (query will return zero results)" messages. **Note:** Set to 1 by Agraph for queries to child Dgraphs.
Valid in query types	graph
ENEQuery method or property	N/A

UrlENEQuery URL parameters	N/A
Format	Numeric Boolean value
Values (order)	0 (false), 1 (true, default)
Example	N/A

irversion

Description	Specifies a major version of API; set automatically by API and should not be changed.
Valid in query types	graph, search, bin, abin
ENEQuery method or property	N/A
UrlENEQuery URL parameters	N/A
Format	Three-digit numeric value
Values (order)	N/A
Example	`irversion=500` (5.0.x), `irversion=510` (5.1.x), `irversion=601` (6.0.1)

keyprops

Description	Specifies whether to return key properties with the query results.
Valid in query types	graph
ENEQuery method or property	Java: `ENEQuery.setNavKeyProperties()` .NET: `ENEQuery.NavKeyProperties`
UrlENEQuery URL parameters	`Nk`
Format	String value
Values (order)	none (default), all “All” equates to API constant `ENE¬ Query.KEY_PROPS_ALL` “None” equates to `ENEQuery.KEY_PROPS_NONE`
Example	N/A

lang

Description	Specifies a language to use for a query.

Valid in query types	graph, search
ENEQuery method or property	Java: `ENEQuery.setLanguageId()` .NET: `ENEQuery.LanguageId`
UrlENEQuery URL parameters	`LanguageId`
Format	N/A
Values (order)	Standard language code string value
Example	`lang=en` for English, `lang=zn_CH` for simplified Chinese

log

Description	Specifies session and query ID values.
Valid in query types	graph, search, bin, abin
ENEQuery method or property	Java: `ENEQuery.setQueryInfo()` .NET: `ENEQuery.QueryInfo`
UrlENEQuery URL parameters	N/A
Format	String containing one or more URL-encoded key=value pairs, separated by ampersands.
Values (order)	key=value pairs
Example	`log=sid%3d11586B%26rid%3d11586`

merchdebug

Description	Specifies debugging output for business rule evaluation in the Dgraph error log. Configured by the `--merch_debug` flag.
Valid in query types	graph
ENEQuery method or property	Java: `ENEQuery.setMerchDebugOn()` .NET: `ENEQuery.MerchDebugOn`
UrlENEQuery URL parameters	N/A
Format	Numeric Boolean value
Values (order)	0 (false), 1 (true)
Example	N/A

merchpreviewtime

Description	Specifies preview time to use for business rules.
Valid in query types	graph
ENEQuery method or property	Java: `ENEQuery.setNavMerchPreviewTime()` .NET: `ENEQuery.NavMerchPreviewTime`
UrlENEQuery URL parameters	`Nmpt`
Format	String value
Values (order)	now (current time), or a date expressed in yyyy-mm-ddTmm:ss format (such as, 2007-07-12T08%3a15 for 8:15am, 12 July 2007).
Example	`merchpreviewtime=now`, `merchpreview¬ time=2007-08-28T12%3a51`

merchrulefilter

Description	Specifies the filter for business rules.
Valid in query types	graph
ENEQuery method or property	Java: `ENEQuery.setNavMerchRuleFilter()` .NET: `ENEQuery.NavMerchRuleFilter`
UrlENEQuery URL parameters	`Nmrf`
Format	String value, formatted per record filters.
Values (order)	N/A
Example	`merchrulefilter=endeca.internal.work¬ flow.state%3aACTIVE`

model

Description	Specifies dimension(s) to which dimension search will be restricted. Multiple values are only usable for compound dimension searches (such as, search for "ford tempo" against intersection of Make and Model dimensions). Simple dimension searches are restricted to a single dimension only, and return 0 results if multiple dimval IDs are specified.
Valid in query types	search

ENEQuery method or property	(search, simple) Java: `ENEQuery.setDimSearchDimension()` .NET: `ENEQuery.DimSearchDimension` (search, compound) Java: `ENEQuery.setDimSearchDimensions()` .NET: `ENEQuery.DimSearchDimensions`
UrlENEQuery URL parameters	`Di`
Format	Numeric dimval ID (simple dimension search), or space-separated list of numeric dimval IDs (compound dimension search).
Values (order)	N/A
Example	`model=2344` (simple dimension search), `mod¬ el=1+18+25` (compound dimension search)

nbins

Description	Specifies maximum number of ERec objects to return for a navigation query, assuming that a query can be on non-aggregated records and on aggregated records. Does not map to any `UrlENEQuery` URL parameter.
Valid in query types	graph
ENEQuery method or property	• In non-aggregated navigation queries: Java: `ENEQuery.setNavNumERecs()` .NET: `ENEQuery.NavNumERecs` • In aggregated navigation queries: Java: `ENEQuery.setNavNumAggrERecs()` .NET: `ENEQuery.NavNumAggrERecs`
UrlENEQuery URL parameters	N/A
Format	Numeric value
Values (order)	10 (default)
Example	`nbins=10` (default), `nbins=500`

nbulkbins

Description	Specifies maximum number of `ERec` objects to be returned via bulk export.

	This parameter corresponds to different methods when querying aggregated records, that is, when a rollup key is applied.
Valid in query types	graph
ENEQuery method or property	(graph) Java: `ENEQuery.setNavNumBulkERecs()` .NET: `ENEQuery.NavNumBulkERecs` (graph, aggregated records) Java: `ENEQuery.setNavNumBulkAggrERecs()` .NET: `ENEQuery.NavNumBulkAggrERecs`
UrlENEQuery URL parameters	N/A
Format	Numeric value
Values (order)	Values (order) 0 (default), positive values, -1 (all records, or `ENEQuery.MAX_BULK_ERECS_AVAIL¬ABLE`) **Note:** "-1" is equivalent to all records, or to setting `ENE¬Query.MAX_BULK_ERECS_AVAILABLE` (that is, bulk-exporting all records matching the query) for the relevant methods.
Example	N/A

node

Description	Specifies selected (descriptor) dimension values.
Valid in query types	graph, search, abin
ENEQuery method or property	(graph) Java: `ENEQuery.setNavDescriptors()` .NET: `ENEQuery.NavDescriptors` (search) Java: `ENEQuery.setDimSearchNavDescrip¬tors()` .NET: `ENEQuery.DimSearchNavDescriptors` (abin) Java: `ENEQuery.setAggrERecNavDescrip¬tors()`

	.NET: `ENEQuery.AggrERecNavDescriptors`
UrlENEQuery URL parameters	`N` (graph), `Dn` (search), `An` (abin)
Format	Space-separated numeric dimval IDs.
Values (order)	N/A
Example	`node=0`, `node=125+234423+87`

num

Description	Specifies the number of dimension value matches to return from each dimension as results of dimension search, per dimension.
Valid in query types	search
ENEQuery method or property	Java: `ENEQuery.setDimSearchNumDimVal¬ues()` .NET: `ENEQuery.DimSearchNumDimValues`
UrlENEQuery URL parameters	`Dp`
Format	Numeric value
Values (order)	N/A
Example	`num=5`

offset

Description	Specifies the number of values to skip before beginning to return record objects (for record search), or dimension value objects (for dimension search).
Valid in query types	graph, search
ENEQuery method or property	(graph) Java: `ENEQuery.setNavERecsOffset()` .NET: `ENEQuery.NavERecsOffset` (graph, aggregated records) Java: `ENEQuery.setNavAggrERecsOffset()` .NET: `ENEQuery.NavAggrERecsOffset` (search) Java: `ENEQuery.setDimSearchResultsOff¬set()` .NET: `ENEQuery.DimSearchResultsOffset`

UrlENEQuery URL parameters	`No` (graph) or `Nao` (graph, aggregated records), or `Do` (search)
Format	Numeric value
Values (order)	N/A
Example	`offset=20` (begins returning objects from record or dimension value starting with 21 and onward).

op

Description	Specifies an operation to perform for command-type (non-query) URLs.
Valid in query types	`admin`, `config`
ENEQuery method or property	N/A
UrlENEQuery URL parameters	N/A
Format	String value
Values (order)	The following `admin` operations are supported: `audit`, `au¬ditreset`, `exit`, `flush`, `help`, `logroll`, `ping`, `restart`, `update`, `updatehistory`, `reload-services`, `stats`, and `statsreset`. The following `config` operations are supported: `help`, `log-disable`, `log-enable`, `log-status`, and `update`. **Note:** The `config` `log-enable` and `log-disable` operations can take several logging variables, which are documented in the MDEX Engine Logging Variables appendix to the *Endeca Advanced Developement Guide*.
Examples	`admin?op=update`, `admin?op=stats`, `config?op=up¬date`

opts

Description	Specifies options, such as match mode, for dimension search. Also specifies a `spell+nospell` option for disabling spelling correction and DYM suggestions on individual queries.
Valid in query types	search

ENEQuery method or property	Java: `ENEQuery.setDimSearchOpts()` .NET: `ENEQuery.DimSearchOpts`
UrlENEQuery URL parameters	`Dx`
Format	Space-separated string values
Values (order)	N/A
Example	`opts=mode+matchall+spell+nospell`

pred

Description	Specifies a range filter expression for a query.
Valid in query types	graph, search, abin
ENEQuery method or property	(graph) Java: `ENEQuery.setNavRangeFilters()` .NET: `ENEQuery.NavRangeFilters` (search) Java: `ENEQuery.setDimSearchNavRange¬ Filters()` .NET: `ENEQuery.DimSearchNavRangeFilters` (abin) Java: `ENEQuery.setAggrERecNavRange¬ Filters()` .NET: `ENEQuery.AggrERecNavRangeFilters`
UrlENEQuery URL parameters	`Nf` (graph), or `Df` (search), or `Af` (abin)
Format	Space-separated string value
Values (order)	property or dimension name key, pipe character separator, operator (such as BTWN, GT), values.
Example	`pred=P%5FPrice%7CBTWN+8+12` (restricts query to records where P_Price value is between 8 and 12).

pretendtime

Description	Specifies time value to use for time-triggered business rules.

Valid in query types	graph
ENEQuery method or property	N/A
UrlENEQuery URL parameters	N/A
Format	String time value (m/ d/ yyyy hh:mm)
Values (order)	**Note:** Set by the Agraph for queries to child Dgraphs. Value is the time of Agraph query.
Example	`pretendtime=+2%2F+1%2F2007+11%3A49`

profiles

Description	Specifies user profiles to apply to a query (used to restrict triggering of business rules).
Valid in query types	graph
ENEQuery method or property	Java: ENEQuery.setProfiles() .NET: ENEQuery.Profiles
UrlENEQuery URL parameters	N/A
Format	Space-separated list of string profile names
Values (order)	String profile names
Example	`profiles=free_shipping+USA`

rank

Description	Specifies whether to use relevance ranking to order dimension values returned by dimension search.
Valid in query types	search
ENEQuery method or property	Java: `ENEQuery.setDimSearchRankRe¬sults()` .NET: `ENEQuery.DimSearchRankResults`
UrlENEQuery URL parameters	Dk
Format	Numeric Boolean value
Values (order)	0 (default dimension value ranking), 1 (relevance ranking)
Example	N/A

refinement

Description	Specifies query-time dynamic refinement ranking settings.
Valid in query types	graph
ENEQuery method or property	Java: `ENEQuery.setNavRefinementCon¬ figs()` .NET: `ENEQuery.NavRefinementConfigs`
UrlENEQuery URL parameters	`Nrc`
Format	Colon-separated list of space-separated values
Values (order)	string, number key, value pairs.
Example	`refinement=dimvalid:6300+dynrank:1+ exposed:1+dynorder:0+dyncount:4`

relrank

Description	Specifies query-time relevance ranking settings.
Valid in query types	graph
ENEQuery method or property	• Through IAP 5.1.1: Java: `ENEQuery.setNavRelRankERec¬ Search()` .NET: `ENEQuery.NavRelRankERecSearch` • IAP 5.1.2 and later: Java: `ENEQuery.setNavRelRankERe¬ cRank()` .NET: `ENEQuery.NavRelRankERecRank`
UrlENEQuery URL parameters	`Nrk Nrt Nrr Nrm`
Format	Pipe-separated list of space-separated values
Values (order)	search key, search terms, relevance-ranking strategy, search mode
Example	`relrank=All|napa+valley|ex¬ act|matchall` **Note:** Match mode and `Nrm` parameter are only supported as of IAP 5.1.2. Using older API libraries with newer versions of the MDEX Engine may produce unexpected results.

select

Description	Specifies fields (properties and dimensions) to return on ERec objects from navigation query.
Valid in query types	graph
ENEQuery method or property	Java: `ENEQuery.setSelection()` .NET: `ENEQuery.Selection`
UrlENEQuery URL parameters	N/A
Format	Space-separated list of string property/dimension name values
Values (order)	String property/dimension name values
Example	`select=P_Name+Vintage`

sort

Description	Description Specifies sort key(s) and order to use for records returned by a query. **Note:** Current version only uses the `Ns` parameter (`Nso` is deprecated).
Valid in query types	graph
ENEQuery method or property	Java: `ENEQuery.setNavActiveSortKeys()` .NET: `ENEQuery.NavActiveSortKeys`
UrlENEQuery URL parameters	`Ns` `Nso` (deprecated)
Format	Pipe-separated list of string key| order value pairs (two pipes between pairs)
Values (order)	asc (ascending), desc (descending)
Example	`sort=P_Price|asc||Vintage|desc`

structured

Description	Specifies an Endeca Query Language (EQL) expression to apply to a query.
Valid in query types	graph, search, abins
ENEQuery method or property	(graph)

	Java: `ENEQuery.setNavRecordStructureEx¬pr()` .NET: `ENEQuery.NavRecordStructureExpr` (search) Java: `ENEQuery.setDimSearchNavRecord¬StructureExpr()` .NET: `ENEQuery.DimSearchNavRecordStruc¬tureExpr` (abin) Java: `ENEQuery.setAggrERecStructureEx¬pr()` .NET: `ENEQuery.AggrERecStructureExpr`
UrlENEQuery URL parameters	`Nrs` (graph), `Drs` (search) `Ars` (abin)
Format	String EQL expression
Values (order)	EQL expression
Example	`structured=collec¬tion%28%29%2frecord%5bP_Re¬gion%3d%22Sonoma%22%5d`

terms

Description	Specifies search terms for dimension search.
Valid in query types	search
ENEQuery method or property	Java: `ENEQuery.setDimSearchTerms()` .NET: `ENEQuery.DimSearchTerms`
UrlENEQuery URL parameters	`D`
Format	Space-separated list
Values (order)	String values for terms
Example	`terms=my+search+terms`

Appendix C

The Eneperf Tool

Eneperf is a performance testing tool that is included in your Endeca installation. This section describes how to use Eneperf.

About Eneperf

Eneperf is a performance, analytics and debugging tool that can measure throughput to help you identify system bottlenecks. Eneperf makes HTTP queries against the MDEX Engine (Dgraph) based on your MDEX Engine request logs and gathers the resulting statistics, without processing the results in any way.

Because Eneperf is lightweight, it has a very slight impact on performance. In most cases, it can be run on the same machine as the Dgraph or Agraph being tested. It can also be run on a remote machine.

Eneperf drives a substantial load at the MDEX Engine and reveals how many operations per second the MDEX Engine responds with. Eneperf lets you measure both query latency and throughput. You specify the log file and specify to Eneperf how many times to run through it, as well as the number of client connections to simulate.

Eneperf understands Endeca MDEX Engine URLs, which use the pipe symbol (|). Because the pipe symbol is not a legal character in the URL/URI standards, other programs, such as `wget`, may transform it inappropriately.

Using Eneperf

Eneperf is installed in the Endeca IAP `bin` directory. It has the following usage.

```
usage: eneperf [-v]
  [--header <header file path>]
  [--help] [--gzip]
  [--list] [--nreq <n>]
  [--nodnscache>] [--msec-between-updates]
  [--progress] [--pidcheck <pid>]
  [--prelude <log file path>]  [--postlude <log file path>]
  [--quitonerror] [--rcvbuf <size bytes>]
  [--record <recording file prefix>] [--record_hdr]
  [--record_ord] [--record_roll <max KB per recording file>]
  [--reqstats] [--reqtimeout <secs>]
  [--runtime <max runtime (minutes)>]
  [--seek <n>] [--seekrepeat] [--sleeponerror <secs>]
  [--stats <num reqs>] [--throttle <max req/sec>]
  [--updates-log]  [--version]
  [--warn <max req time warning threshold (msecs)>]
  <host> <port> <log> <num connections> <num iterations>
```

Eneperf has both required and optional settings.

Required settings

The required settings (shown in order) are as follows.

```
<host> <port> <log> <num connections> <num iterations>
```

Their usage is as follows.

Setting	Description
`<host>`	Target host for requests.
`<port>`	Port on which the target host is listening for requests.
`<log>`	Log file of the query portion of the MDEX Engine URLs and optional associated information (that is, the portion that resides in the last three columns of the MDEX Engine request log). This log file is used for HTTP request generation. URLs and associated information from the `<log>` file are replayed in order. Each line of the `<log>` file contains three columns: • A URL (required) • A POST body (URL-encoded and optional) • HTTP headers (URL-encoded and optional). If a dash (-) is found in an optional column, the column is ignored.
`<num connections>`	Maximum number of outstanding requests to allow before waiting for replies. In other words, the number of simultaneous HTTP connection streams to keep open at all times. This number emulates multiple clients for the target server. For example, using `<num connections>` of 16 emulates 16 concurrent clients querying the target server at all times.
`<num iterations>`	Number of times to replay the URL query log. All outstanding requests are processed before a new iteration is started.

Host and port settings for running Eneperf locally or remotely

You can run Eneperf locally or from a remote machine.

- Running Eneperf locally. Eneperf is lightweight and has a very slight impact on performance. It can usually be run on the same machine as the Dgraph or Agraph being tested with no impact on results.

  To run Eneperf on the same machine as the Dgraph or Agraph, you point it to `localhost` and `<port>`. This configuration is useful for isolating MDEX Engine performance from any potential networking issues.

- Running Eneperf on a remote host. Eneperf can also be run from a remote host. Using Eneperf to test the same MDEX Engine from the local machine and from across the network can expose networking problems if the throughputs are significantly different.

Note: Eneperf can be run on a machine with a different architecture than one you are testing.

Log file settings suitable for Eneperf input

MDEX Engine request logs can be used as Eneperf input with some modifications.

URLs in the log should not include any machine connection parameters such as protocol, host, or port. These are added automatically. For example, a log entry of the following form is valid:

```
/graph?node=0
```

But a log entry of the following form is not valid:

```
http://myhost:5555/graph?node=0
```

You can achieve higher concurrent load by using a single large request log file (which might simply be repeated concatenations of a smaller log file) than by using multiple iterations of a small log file. The log file should preferably be at least 100 lines, even if it consists of the same query repeated over and over. Because Eneperf drains all connections between each iteration, running a one-line log file through Eneperf 100 times results in skewed throughput statistics.

If you are planning to measure performance of partial updates with Eneperf, (as opposed to measuring performance of regular queries), create a separate updates log based on your existing request log.

That is, suppose your MDEX Engine request log contains both regular queries and updates operations. Then your updates log should contain only `config?op=update` operations. You can create this updates log manually, by extracting these operations from a regular log. You can then run Eneperf against the updates log and the regular log, to measure the performance of your updates, by using the `--updates-log` and the `--log` settings together.

Note: This is only one way to measure performance of updates and should only be used in cases when you care about the time between the updates. (If you do not care about the timing between updates, you can use the regular log for your testing.)

About the number of connections and iterations

Eneperf load is driven by the `num connections` setting, which indicates the number of simultaneous connections Eneperf tries to maintain at a time.

For example, if `num connections` is set to 4, it sends four requests to the MDEX Engine. When one returns, another is sent out to replace it.

To adequately measure performance of the MDEX Engine, you need to identify the number of connections for Eneperf that saturates the MDEX Engine thread pool.

The number of connections needed to saturate the MDEX Engine depends on the MDEX Engine threading configuration and the server characteristics, and generally correlates with the number of the MDEX Engine threads in use, (assuming the MDEX Engine is configured with enough threads). However, an MDEX Engine with four threads might be saturated by only three connections if the queries are complex and all CPUs are being fully utilized.

To identify an appropriate setting for `num connections`, Endeca recommends running tests with the following settings:

- For debugging, run a test with `num connections` set to one. This test sends only one request to the MDEX Engine at a time. Each query is processed alone; no other query computations are contending for the machine’s resources. This test generates an MDEX Engine request log showing the canonical time for each query. You can examine the request log to identify slow queries without the concern that they happened to be slow because other queries were processed simultaneously. Note that using a log file with just one entry limits num connections to one.
- For stress testing, run a test with `num connections` set to the number of threads for the MDEX Engine. In this test, no requests are waiting in the queue. This lets you obtain an estimate of the maximum expected MDEX Engine performance. Because no queuing occurs, this test offers a conservative bias for throughput.

  In addition, you can run a test with `num connections` set to the “number of threads + one”. In this test case, a minimal waiting in the queue for the MDEX Engine request may occur. This also lets you obtain an estimate of the maximum expected MDEX Engine performance. Because queuing does not occur, this test offers an aggressive bias for throughput.
- Do not use a small log with a large number of `num connections`. Also, do not run a small log many times to simulate a large log.

Example: Selecting the number of connections

Commonly, you will wish to perform the load testing of the MDEX Engine to a level below saturation. Use the following examples to help you select an appropriate number of connections for Eneperf that will saturate MDEX Engine performance to the desired levels.

Typically, front-end applications have different requirements for response times and peak loads. Such as:

- An application that is used steadily across the year. For applications of this type, MDEX Engine performance must support average query response time under average loads. Occasional slowdowns under peak load are acceptable. Therefore, you need to measure average response time under average load.
- An application that is used during the peak seasons. For applications of this type, MDEX Engine performance must support peak response time under peak loads. It is acceptable for this application to have extra performance capacity during non-peak seasons.

To identify the projected throughput for the MDEX Engine, use the following formulas.

These formulas represent a highly simplified approach to calculating throughput. Although you can use more thorough methods, these formulas provide reasonable estimates, and can be used for initial assessment:

```
concurrent users / (expected page latency + think time) = page views/sec
page views / second x MDEX queries/page = ops/second for the MDEX Engine
```

Where:

- The number of concurrent users is the estimated number of users currently logged in to the application
- The number of simultaneous requests is the number of users currently making a request to the application. Typically, it is 20-30% of the number of concurrent users.
- Peak load is the expected maximum number of simultaneous requests, such as during a specific time period
- Think time is the time between requests issued by a single user. It is used to calculate simultaneous requests based on the estimated number of concurrent users.

For example, 100 concurrent users with a 5 second think time and a 1 second expected page latency will yield 17 pages/sec. 17 pages/second with 2 MDEX Engine queries per page will yield 34 ops/sec for the expected performance of the MDEX Engine. This means that to support 100 concurrent users in this application, the MDEX Engine must perform at 34 ops/sec.

In another example, if your implementation includes a load balancer serving four application servers, and two MDEX Engines with another load balancer, the following calculations provide you with the estimated performance for each of the MDEX Engines:

- 600 concurrent users are distributed across 4 application servers. This means 150 users per server.
- 150 users divided by 5 (4 sec think time and 1 sec expected page latency) yields 30 simultaneous page views per server.
- 30 page views with 2 MDEX Engine queries per page yield 60 MDEX Engine queries per server.
- 60 queries per server multiplied by 4 application servers yield 240 queries total.
- 240 queries are sent to the load balancer that distributes them across two MDEX Engines. Each MDEX Engine serves 120 queries.

This means that to support 100 concurrent users in this application, each MDEX Engine must perform at 120 ops/sec.

To summarize, you can use these recommendations to identify the number of connections (equal to the number of simultaneous requests in these examples) that you need to provide to Eneperf to achieve the desired MDEX Engine performance.

Optional settings

Eneperf contains the following optional settings.

Setting	Description
`-v`	Verbose mode. Print query URLs as they are requested.
`--gzip`	Add `Accept-encoding: gzip` to the HTTP request header.
`--header <head¬er_file_path>`	Specify path of file containing HTTP header text, one header field per line. This setting, if used, overrides headers from the log file (which you can also specify).
`--help`	Print the help usage and exit.
`--list`	Treat the `<log>` parameter as the name of a file containing the names of a sequence of request logs, rather than directly naming a single request log. As a result, Eneperf iterates over the sequence of logs. Each line in the `<log>` names a request log file to be replayed against Eneperf in sequence, during each iteration.

Setting	Description
`--msec-between-updates`	If you use this setting with `--updates-log`, it specifies the minimum time interval between sending partial update requests, in milliseconds. Before sending a new update request, Eneperf waits for a free connection (after the specified time interval expires). This setting must not be used together with `--list`, `--seek`, `--seekrepeat`, `--prelude`,`--postlude`, and `--throttle`. **Note:** The `--msec-between-updates` setting is optional. If you use only the `--updates-log` setting, Eneperf processes updates one after another. Eneperf waits for the current update to finish and immediately sends another update. It does not wait for any period of time between sending individual updates to the Dgraph.
`--nreq <n>`	Stop after n requests.
`--nodnscache`	Disable caching of DNS hostname lookups. By default, Eneperf caches these lookups to improve performance.
`--pidcheck <pid>`	On a connection error, check the specified Dgraph or Agraph process to see if it is running. If the process is not running, terminate Eneperf.
`--prelude <log_file_path>`	Specify a `<log_file_path>` of the file with URLs to replay before those of the `<log>` parameter, for each iteration. Use this flag together with the `--list` flag to avoid repetition of requests in the several log files named in the `<log>` parameter.
`--postlude <log_file_path>`	Specify a `<log_file_path>` of the file with URLs to replay after those of the `<log>` parameter, for each iteration. Use this flag together with the `--list` flag to avoid repetition of requests in the several log files named in the `<log>` parameter.
`--progress`	Display the percentage of the query log file processed. **Note:** If you run Eneperf in the two-stream mode for testing updates performance, it displays the progress only for the regular queries log, not for the updates log.
`--quitonerror`	Terminate the Eneperf process if it encounters a fatal HTTP connection error. By default, errors are ignored and do not stop the Eneperf run.
`--rcvbuf <size_bytes>`	Override the default TCP receive buffer size, set with the `SO_RCVBUF` socket option.
`--record <rec_file_prefix>`	Record a log of all HTTP responses. Recorded data is placed in output files with the prefix `<rec_file_prefix>`. Data files are

Setting	Description
	given the suffixes .dat1, .dat2, and so on. An index file with the suffix .idx is also produced.
`--record_hdr`	In `--record` mode, record HTTP header information along with page content.
`--record_ord`	In `--record` mode, ensure that log entries are recorded in the same order that they are listed in the `<log>` file, even if they are processed out of order.
`--record_roll <max_KB>`	Set the maximum number of KB per recording file. Default is 1024 KB.
`--reqstats`	Maintain and report per-request timing statistics. **Note:** This option produces accurate results only if you specify `<num connections>` as 1.
`--reqtimeout`	Places a limit on the time for any individual request. Default is 600 seconds.
`--runtime <max_runtime>`	Place a limit on the run time for Eneperf. Eneperf exits after `<max_runtime>` minutes. Minutes are the default unit.
`--seek <n>`	Skip a specified number of requests in the specified log file and start with log entry n. For example, in a log containing 100 requests, if you run Eneperf with `--seek 50`, it issues 50 requests from 50 to 100.
`--seekrepeat`	Use in conjunction with `--seek`. Start each iteration with the log entry specified by `--seek`. `--seekrepeat` has an impact only if the number of iterations specified is greater than one. If it is so, when Eneperf reaches the end of the log file, `--seekrepeat` indicates that it should start the next iteration from the log entry specified as a value to `--seek` (50 in the example above). The behavior without `--seekrepeat` and with `--seek` specified is to seek only on the first iteration and restart from the beginning of the file on subsequent iterations.
`--sleeponerror <secs>`	Sleep for a specified number of seconds before sending any new requests after a connection error occurs.
`--stats <num_reqs>`	Print statistics after the specified `<num reqs>` are processed (sent and received).
`--throttle <max_req/sec>`	Place an approximate limit on the number of requests per second that Eneperf generates.
`--updates-log`	Specifying the updates log allows running Eneperf in a two-stream mode with two logs: regular query request logs and update request logs. In this mode, Eneperf sends update requests from the updates log at regular intervals while sending queries from the query log.

Setting	Description
	This setting can be used either together with the `--msec-be¬tween-updates` setting, or without it: • If this setting is used together with `--msec-between-up¬dates`, it specifies the updates log file that contains partial update requests. These requests are replayed at every interval in milliseconds specified with `--msec-between-updates`. • If this setting is used without `--msec-between-updates`, updates are sent to the Dgraph one after another, that is, Eneperf waits for the current update to finish and immediately sends another update. It does not wait for any period of time between sending individual updates to the Dgraph. This setting must not be used together with `--list`, `--seek`, `--seekrepeat`, `--prelude`,`--postlude`, and `--throttle`. Before running Eneperf in the two-stream mode, you need to create a separate log that contains only partial update requests. You should create such a log with several partial update requests pointing to a single update file using the `admin?op=update&up¬datefile=filename` command. For more information on running partial updates on a single file, see the *Partial Updates Guide*.
`--version`	Add the version of Eneperf that is used for this iteration. The version information is always displayed at the beginning of Eneperf output, as follows: Endeca eneperf version `<number>`.
`--warn <max_req_threshold>`	Print a warning message for any requests that take longer than the specified threshold time limit to return (useful for finding the “slow” requests in a log file). The threshold time limit is specified in milliseconds.

About generating incremental statistics

You use the `--stats` setting to specify how many queries you want to see statistics reported on.

Typical values are 500 or 100. The `--reqstats` setting provides a finer level of detail.

Generating statistics on the fly

Eneperf can run for hours. If you neglected to set `--stats` yet want to obtain a statistics printout without stopping the process, you can send Eneperf a `usr1` signal.

For example, on UNIX, you could use the kill command to send a signal like this:

```
kill -usr1 pid
```

About setting the number of queries sent to the Dgraph

By default, Eneperf drives load as fast as the MDEX Engine can handle it. However, there is a setting, `--throttle`, that allows you to place an approximate limit on the number of queries per second sent to the MDEX Engine. That means you can drive load at a rate you select.

The `--throttle` setting is useful when you want to approximate a special case. For example, imagine you expect high-traffic load during the holiday season. You want to calculate maximum load, while maintaining a comfortable margin of error for the MDEX Engine by running it at 80% utilization.

You might prepare an estimate by multiplying the maximum load by 0.8. Alternatively, you could use `--throttle` to try different numbers of queries per second and to capture the CPU performance on the MDEX Engine machine, using a tool such as vmstat on Solaris. You could then calculate the average CPU utilization from these numbers, or plot a chart of utilization over time in Microsoft Excel.

The mapping of the `--throttle` setting to queries per second is not exact. Eneperf uses a simple method to calculate the waiting times to insert between queries. You get a real number of operations per second but it might be significantly lower than you want or expect. The `--throttle` setting to Eneperf can generate performance results that exceed the maximum throughput of the MDEX Engine and still result in throughput results for the MDEX Engine that are less than its maximum. Experiment with this setting to identify the best strategy for your situation.

Example of Eneperf output

This topic contains an example of Eneperf output and describes it briefly.

```
Running iteration 1...
Done:
58881 sent, 58881 received, 0 errors.
22 minutes, 42.63 seconds for iteration 1, 43.2112 req/sec.
22 minutes, 42.63 seconds elapsed (user: 6.20 seconds, system: 15.24 sec¬
onds).
Net: 1.18389e+06 KB (868.829 KB/sec).
Page Size: avg=91.34 KB, std dev=142.81 KB, max=1238.37 KB, min=0.16 KB.
Latency: avg=92.36 ms, std dev=238.27 ms, max=13441.11 ms, min=0.18 ms. 250
 queries longer than 1s.
Eneperf completed:
58881 sent, 58881 received, 0 errors.
22 minutes, 42.63 seconds elapsed (user: 6.20 seconds, system: 15.24 sec¬
onds).
Net: 1.18389e+06 KB (868.829 KB/sec).
Page Size: avg=91.34 KB, std dev=142.81 KB, max=1238.37 KB, min=0.16 KB.
Latency: avg=92.36 ms, std dev=238.27 ms, max=13441.11 ms, min=0.18 ms. 250
 queries longer than 1s.
        Best iteration time: 22 minutes, 42.63 seconds.
        Peak rate: 43.2112 req/sec.
        Avg iteration time: 22 minutes, 42.63 seconds.
        Avg rate: 43.2112 req/sec.
        Total rate: 43.2112 req/sec.
```

The entries from Eneperf output are described in the following table:

Sample Eneperf output entry	Description
`Running iteration 1...`	Is printed as each iteration begins.

Sample Eneperf output entry	Description
	The numbers following this line, until "Eneperf completed:" occur for each iteration requested. The number of iterations requested is the last Eneperf parameter.
`done:`	Is printed once the iteration finishes.
`58881 sent, 58881 received, 0 errors.`	“Sent” is the number of queries sent. It is the sum of “Received” and “Errors” and the number of errors, where errors is the number of 404 or 400 HTTP codes that the Dgraph returns, (rather than errors in the Dgraph log). “Received” is the number of queries with a 200 HTTP status code that the Dgraph returns. “Errors” is the number of queries with 404 or 400 HTTP status code that the Dgraph returns, rather than errors in the Dgraph log.
`22 minutes, 42.63 seconds for iteration 1, 43.2112 req/sec.`	The time for the specific iteration, and the throughput for this iteration.
`22 minutes, 42.63 seconds elapsed (user: 6.20 seconds, system: 15.24 seconds).`	The total runtime up until this point. System time is the time spent in the operating system on behalf of the Dgraph. User time is the time spent in the Dgraph itself.
`Net: 1.18389e+06 KB (868.829 KB/sec).`	The total amount of data returned for the entire test (not just for one iteration).
`Page Size: avg=91.34 KB, std dev=142.81 KB, max=1238.37 KB, min=0.16 KB.`	Cumulative statistics on the amount of data returned for each query.
`Latency: avg=92.36 ms, std dev=238.27 ms, max=13441.11 ms, min=0.18 ms. 250 queries longer than 1s.`	Cumulative statistics on the latencies. The statistics include previous iterations. Latency information may be inaccurate when multiple connections are in use, particularly if the network is slow. If accuracy is critical, consider obtaining latency information from the Dgraph request log.
`Peak rate: 43.2112 req/sec.`	The processing rate of the iteration in the test with the best performance, but should not be confused with "peak" performance in the sense of a single second that showed the highest throughput. It is the total number of requests processed in that iteration divided by the time of the iteration in seconds.

Sample Eneperf output entry	Description
	If the test includes only one iteration, peak rate is the processing rate for that iteration.
`Avg iteration time: 22 minutes, 42.63 seconds.`	The average time of the iterations in the test.
`Avg rate: 43.2112 req/sec.`	The average rate of the iterations in the test, in requests processed per second.
`Total rate: 43.2112 req/sec.`	The total of requests processed for all of the iterations in the test, divided by the total time of all of the iterations in the test.
`Eneperf completed:`	All information after this statement is cumulative over the entire run. This line is printed once all iterations have completed.

About the format of logs for use with Eneperf

In order to use Eneperf, you need a log of URLs in the correct format. The lines in the log file you use with Eneperf should not specify the run-time statistics, hostname and the port.

There are numerous ways that you can obtain such logs; this section provides you with guidelines and a few examples.

The Request Log Parser

In order to use Eneperf, you need a log of URLs in the correct format. The Request Log Parser is an Endeca utility that converts the MDEX Engine log format into Eneperf log format.

Alternatively, you can convert URLs yourself. For more information, see Converting a MDEX Engine request log file for Eneperf.

You can download the Request Log Parser from EDeN.

Recommendations for generating a representative log for Eneperf

The test log that you will use with Eneperf determines the contents and the results of your performance testing. Because the test log serves as input to Eneperf, it should be representative of those aspects of the MDEX Engine performance you want to test.

Use these recommendations to create a representative log:

- Add queries of various types to your log to account for a variety of queries. Depending on the query type, some queries are processed much faster than others.

  For example, dimension and record search queries are the fastest, queries on aggregate records, or navigation and search queries take longer, whereas navigation with Analytics, or navigation queries with RRN may take more time. Even within queries of the same type, individual queries can have large performance differences, depending on the query parameters.

- If you want to test a particular feature configuration for performance, ensure that your query log contains a fair percentage of queries of this type.
- If you are planning to test updates that run at regular intervals, create a separate updates log from your regular log that contains only `config?op=update` operations, and run Eneperf against this updates log and the regular log at the same time. Use the `--updates-log` setting together with `log` and `--msec-between-updates` settings.
- If queries are repeated in the log, or parts of them are repeated, this makes the log less useful for performance testing, since a large percentage of queries may be served entirely from the MDEX Engine cache. Therefore, do not replay a short query log multiple times.
- For a full-scale performance test, generate a log that runs for 30 minutes or more. In addition, you may want to create a smaller log that runs for 5-10 minutes to use it as a quick test.
- To create a representative log, use the existing MDEX Engine logs from the production system. Use the Request Log Parser to strip undesired columns and queries. For information, see “The Request Log Parser”.
- Translate existing Web application logs into the MDEX Engine format. For example:

  ```
  /results.jsp?searchterm=ipod
  ```

  turns into:

  ```
  graph?node=0&group=0&offset=0&nbins=10&attrs=All+ipod|mode+matchall&dym=1
  ```

- Translate existing traffic reports, such as a list of top search terms, into the MDEX Engine format by programmatically generating URLs as produced by the MDEX Engine. For example, for the term “iron man”, generate:

  ```
  graph?node=0&group=0&offset=0&nbins=10&at¬
  trs=All+iron+man|mode+matchall&dym=1
  ```

- Use the Request Log Parser to remove all `admin` queries from a request log (use the default or `-q gb` options for the parser). Typically, process health requests of type `/admin?op=ping` can run every few seconds, are typically very fast and not generated by end users. However, requests of type `/admin?op=exit` stop and restart the process and will impact your log.
- Remove dimension search queries from your Eneperf log. This is because a single API request that includes a dimension search is turned into two MDEX Engine requests. For example, the following request:

  ```
  ?N=0&Ntk=All&Ntt=plum&Nty=1&D=plum
  ```

  turns into:

  ```
  /graph?node=0&group=0&offset=0&nbins=10&attrs=All+plum
  /search?terms=plum&rank=0&offset=0&compound=1
  ```

  From the application perspective, this request constitutes one query, since the presentation API waits for both responses and recombines them into a single response object to the front-end application. However, the MDEX Engine and performance tools, such as Eneperf and Cheetah, treat such dimension search requests as two queries.

  If you remove these dimension search queries, which are known to be fast, from the Eneperf log and replace them with other queries, you can use Eneperf to measure the MDEX Engine performance against this log. If the desired level of performance is achieved with such a log, you will achieve or exceed that performance when dimension searches are included again.

Running Eneperf in two-stream mode: regular logs and logs with updates

You can run Eneperf in a two-stream mode using two streams of request logs — regular query request logs and logs that contain partial update requests. This lets you test MDEX Engine performance with partial updates applied at regular intervals while running a regular query load.

To run Eneperf in the two-stream mode, use the following Eneperf settings together:

- `--updates-log`
- `--msec-between-updates`
- `--log`

When used in this mode, Eneperf sends update requests from the updates log at regular intervals while sending queries from the query log.

In more detail, Eneperf runs in the following way:

1. It uses the log file (specified with `--log`) and sends requests from this file for the duration that you specify by the `--msec-between-updates` setting.
2. At the specified time interval, it sends an update request from the updates log file (specified with `--updates-log`) and uses one of its connections for this request.
3. It continues to send query requests from the query log (`--log`), using the other connections.

   Note: This behavior assumes that you are running Eneperf with the number of connections set to more than one. If you use only one connection, Eneperf will switch between update and regular query requests.

4. This process continues until either the regular query log or the updates log has been completely processed. For example:
   - If Eneperf sends the last update request from the updates log, but the query log still contains queries, Eneperf will send additional queries for the time interval specified with `--msec-between-updates` and then stop. (Since the two-stream mode is designed specifically to test updates performance, Eneperf does not process regular queries after the last update in the updates log has been processed.)
   - If Eneperf sends the last query from the regular log, but the updates log still contains additional update requests, it will not send these updates to the Dgraph. Therefore, ensure that the regular query log contains sufficient number of requests to last for the duration of your two-stream Eneperf testing session.

The format of the updates request log is the same as the format of a regular query log for Eneperf, except that the updates log should contain only `config?op=update` operations in order to provide meaningful performance results. (If your updates log contains regular queries, Eneperf still processes this log successfully. However, the results are not meaningful for measuring updates performance.)

Using `--updates-log` and `--log` settings is useful to measure performance of those updates that run at regular intervals. To test updates that run at random times, you can continue using your regular log with Eneperf.

Note: The actual time interval between sending update requests may be equal to or greater than the time specified with `--msec-between-updates`. This is because Eneperf uses the same `num connections` setting while processing the regular query log and updates log. This causes Eneperf to wait for a preceding request to complete before it can process the next updates log request.

Before running Eneperf in the two-stream mode, you need to create a separate log that contains only partial update requests. You should create such a log with several partial update requests pointing to a single update file using the `admin?op=update&updatefile=filename` command. For more information on running partial updates on a single file, see the *Partial Updates Guide*.

Note: The `--msec-between-updates` flag is optional. In other words, if you only specify the `--updates-log` flag, the updates are sent to the Dgraph one after another. Eneperf waits for the current update to finish and immediately sends another update. It does not wait for any period of time between sending individual updates to the Dgraph.

Converting an MDEX Engine request log file for Eneperf

In order to use Eneperf, you need a log of URLs in the correct format. You can manually convert the log to the desired format, or use the Request Log Parser available from EDeN.

The lines in the log file you use with Eneperf should not specify the run-time statistics, hostname and the port. For example, raw URL requests could be formatted like these:

```
/search?terms=blackberry&rank=0&opts=mode+matchall&offset=0&compound=1
&irversion=510
/graph?node=0&group=10&offset=0&nbins=10&attrs=All+berry|mode+matchall
&dym=1&irversion=510
```

To convert a complete MDEX Engine request log file for Eneperf use:

Run the following command:

```
sed -e '/DGRAPH STARTUP/d' <logfile> |
sed -e '/\/admin.*$/d' |
cut -d ' ' -f 12-
```

This does the following:

- It deletes DGRAPH STARTUP lines, because these lines contain no commands.
- It removes `admin` requests, such as `admin?op=stats` or `admin?op=exit`, that can cause problems in an Eneperf run.
- It obtains the last three columns in the log (the URL, POST body, and HTTP headers).

Performance testing .NET 2.0 applications that contain long or complex queries

In rare cases, if your .NET 2.0 (or later) application uses very complex record filters or Analytics statements, you may find that your Eneperf results differ from what is seen in production.

This discrepancy results from the way the .NET 2.0 API to the MDEX Engine handles very long or complex queries. Instead of the usual HTTP GET request to the MDEX Engine, it uses an HTTP POST request. However, the MDEX Engine logs the query as if it were a GET request. The different processing and validation that occurs for POST requests may result in performance differences.

To better simulate the performance of applications that contain such queries, you can use the Request Log Parser to pre-process the logs used to run the Eneperf test. For each request in the log that is longer than 65,000 characters, prepend `'/graph'` with a space after it to the request. Use the subsequent log as the input to Eneperf.

Note: This behavior only manifests itself in the case of very long or complex queries. Most applications never use queries of this sort.

Creating a log file by hand using substitute search terms

You can also approximate a log file to be used with Eneperf. This method is useful when you do not have a running MDEX Engine and archives of logs to work with.

For example, you may want to test the performance of search terms culled from some other system.

To create a log file by hand:

1. Create a list of search terms that you want to test.
2. Copy or create a URL and optional HTTP POST body in the appropriate format.
3. Compose a new log file by substituting your search terms into URL requests containing suitable options.

Debugging Eneperf

Eneperf generates error messages in various error conditions.

- If you make an error while typing the command line argument, Eneperf returns its help message.
- if you accidentally mistype the MDEX Engine port, Eneperf generates numerous failed connection error messages.
- If Eneperf encounters socket connection errors, it reports error messages.

It is also possible for error messages to be displayed during normal operation. For example, if the log file contains a request to retrieve a record that is not present in the MDEX Engine data set, Eneperf (as expected) presents a 404 (file not found) message.

Note: Queries that cause HTTP errors are not counted towards ops/sec performance results displayed by Eneperf.

Appendix D

MDEX Engine Statistics and Auditing

The MDEX Engine Statistics page displays MDEX Engine (Dgraph) performance statistics. You can also view the Agraph Statistics page. The MDEX Engine Auditing page tracks usage for licensing and performance purposes. This section describes these pages.

About the MDEX Engine Statistics page

The MDEX Engine Statistics page provides a detailed breakdown of what the Dgraph is doing, and is a useful source of information about your Endeca implementation's configuration and performance.

The statistics page is also called the Dgraph Stats page or Admin Stats page.

It provides information such as startup time, last data indexing time, and indexing data path. This allows you to focus your tuning and load-balancing efforts. By examining this page, you can see where the Dgraph is spending its time. Begin your tuning efforts by identifying the features in the Hot Spot Analysis section with the highest totals.

Note: In addition to the Dgraph Stats page, if you are using an Agraph implementation, you can access the Agraph Stats page.

Viewing the MDEX Engine Statistics page

You can request the MDEX Engine Statistics page for the Dgraph with the URL listed below.

```
http://DgraphServerNameOrIP:DgraphPort/admin?op=stats
```

For example, if your Dgraph is running on your local machine and listening on port 8000, specify this:

```
http://localhost:8000/admin?op=stats
```

You can determine the host and port on the EAC Admin Console of Endeca Workbench by opening the MDEX Engine component or by exploring your Deployment Template `AppConfig.xml` file.

To reset the statistics, make the following request:

```
http://DgraphServerNameOrIP:DgraphPort/admin?op=statsreset
```

To view the statistics information for a single request, clear statistics, issue a request and inspect statistics again.

The statistics page information is valid as long as the MDEX Engine keeps running; it is cleared upon the MDEX Engine restart.

The source data for the Dgraph statistics is stored in XML. By default, the MDEX Engine Statistics page is rendered into HTML through an Endeca XSLT stylesheet, `stats.xslt`, that is installed in the `ENDECA_MDEX_ROOT/conf/dtd/xform directory`.

If your browser supports XSLT transformations (for example, Internet Explorer 6 and later), you can view the statistics as transformed by `stats.xslt`, or you can modify the shipped stats.xslt stylesheet to provide a different transformation of the data.

If your browser does not support XSLT transformations, or if you want to see the raw XML, rename or remove `ENDECA_MDEX_ROOT/conf/dtd/xform/stats.xslt`.

To ensure that the statistics page displays properly in your browser, JavaScript must be enabled, as part of the settings for the browser. (If your browser's security setting is set to high, this may disable JavaScript.)

Sections of the MDEX Engine Statistics page

The MDEX Engine Statistics page for the Dgraph is divided into tabs. Information on all of the tabs is presented through the URL of the statistics page as described in the following sections.

The Performance Summary tab

The **Performance Summary** tab contains the highest level statistics. They reflect and help to monitor those characteristics that are external to the actual processing of queries, such as the queue of incoming queries, the thread pool, and the overall throughput of the process.

The **Performance Summary** tab contains the following sections:

Section	Description
Performance	Various statistics (average, standard deviation, minimum, maximum, and total) on: • The total number of requests received • Total CPU usage (in seconds of total user time and total system time). • The memory resource usage. • Resident Set Size (RSS) statistics.
Throughput (req/sec)	Five-minute, one-minute, and ten-second average throughput statistics (only for multithreaded mode). When thread becomes available, the throughput statistics is measured.

The General Information tab

The **General Information** tab contains the following sections.

Section	Description
Information	Basic connection and machine details, such as process ID, parent process ID, user ID, user name, effective user ID, group ID, effective group ID, current working directory, hostname, server port for the Dgraph, start time, information about the data (path, tag and date), and the number of index generations.
Arguments	A list of all arguments the Dgraph was started with.

The Index Preparation tab

The **Index** tab tracks index preparation and precomputed sorts statistics, including timing.

It contains the following sections:

Section	Description
Update Totals	The number of non-XQuery updates run against the Dgraph, and performance of updates (count, average, standard deviation, min, max and total), on the following items: • Record changes, including the number of adds, updates, deletes and replacements • Dimension changes • Record change errors • Dimension change errors • Update latency, including various finer-grained performance statistics of indexing processing.
XQuery Update Totals	The number of XQuery updates run against the Dgraph, and performance of updates (count, average, standard deviation, min, max and total), on the following items: • Record changes, including the number of adds, updates, deletes and replacements • Dimension changes • Record change errors • Dimension change errors • Update latency, including various finer-grained performance statistics of indexing processing. **Note:** The XQuery update feature is Early Access in this release. For details, see the *Web Services and XQuery Developer's Guide.*
Precomputed Sorts	Displays how much time the Dgraph has spent computing sorts, including computing sorts and incremental sort updates.

Note: For some of the statistics on this page, it is possible to drill down for further information by clicking on the black arrow that appears outside the rightmost column.

The Cache tab

The **Cache** tab contains information about the MDEX Engine cache.

Section	Description
Main Cache	Provides details on totals, including number of entries in the cache, size of entries, number or lookups in the cache, number of rejections, percentage of hit rate and miss rate, number and size of evictions from the cache, number of reinsertions, total reinsertion time and average creation and eviction times. In particular, if you need to analyze the MDEX Engine cache, examine the results in the following columns. Analyzing these results may help you tune your cache and re-design your front-end application to improve performance of the MDEX Engine. • Number of rejections. Counts greater than zero in this column indicate that the cache is undersized and you may want to increase it. • Number of reinsertions. Large counts in this column indicate that simultaneous queries are computing the same values, and it may be possible to improve performance by sequencing queries, if the application design permits. • Total reinsertion time. Examining this column is useful for quantifying the overall performance impact of queries that contribute to the "Number of reinsertions" column. This column represents the aggregated time that has been spent calculating identical results in parallel with other queries. This is the amount of compute time that potentially can be saved by sequencing queries in a re-design of the front-end application.

The Details tab

The **Details** tab contains the following sections:

Section	Description
Most Expensive Queries	The URL and total time in milliseconds for the ten queries with the largest total computation time (that is, queue time plus Dgraph processing time plus write time) made in the session. The queries are ordered by processing time. Each time a new Dgraph transaction that yields results is completed, this tab may become updated with a new query, if it makes the list of current top ten most expensive queries. Each query is described with these characteristics: • Query rank • Computation processing time (in milliseconds) • URL Unlike in Presentation API mode, where the URL contains all of the information about the query, in Web services mode the URL only contains the service name. The bulk of the query is contained in the POST body. Therefore, if the Dgraph is running in Web services mode, a serial number is appended to the URL, as in the following example: `/ws/myservice:57`.

Section	Description
	This serial number corresponds to the HTTP Exchange ID in the MDEX Engine Request Log. You can use it to retrieve additional information about the contents of the query from the Request Log's Query Body field.
Hotspots	Details on the performance of specific features, such as clustering, record search, record filter, range filter, content spotlighting and snippeting. This section also contains the following page render and record sorting statistics: • **Page render total**. After the MDEX Engine knows which records and values must be returned, this time represents the total time spent generating and returning those results to the Presentation API. This time includes retrieving records from memory or disk, ordering them based on the specified sort or relevance ranking strategies, as well as other information returned to the API, such as content spotlighting results. • **Prefetching horizontal record**. The cost to retrieve records from the data layer of the MDEX Engine. • Statistics related to various sorting strategies. The MDEX Engine examines information about the data being returned and selects the best sorting strategy. **Note:** These statistics may change. They are used for internal debugging and tuning of the MDEX Engine sorting selection strategy and are not useful to the end user.
Results	The following items are listed in the Results section. The statistics includes count, average, standard deviation, min, max and total, where applicable: • Number of records in result set • Result page size in bytes • Result page format performance in milliseconds
Server	Statistical information for the MDEX Engine server: • HTTP: Total request time • HTTP: Time reading request • HTTP: Time in scheduler • HTTP: Time writing response • HTTP: Request bytes read (including HTTP overhead) • HTTP: Response body size (including HTTP overhead) • Scheduler: Queue time before processing • Scheduler: Processing time • Scheduler: Queue time after processing • Scheduler: Queries queued This metric describes the queue length. • Scheduler: Queries in process This metric describes the number of queries that are in process. • Scheduler: Update queue time

Section	Description
	• XQuery: Total time in XQuery engine • XQuery: Total time in XQuery external functions **Note:** This statistic only includes the time spent in the following functions: `internal:query()`, `mdex:dimension-value-id-from-path()`, and `mdex:add-navigation-descriptors()`. • XQuery: Time retrieving documents with `fn:doc()` • XQuery: Time storing documents with `fn:put()` • XQuery: Result serialization time • Most expensive MAX invocations • Custom timing list This metric, which can list things like expensive queries, only appears when you implement custom metric gathering with the `ep:stats-timing` pragma. See the *Web Services and XQuery Developer's Guide* for more information.
Navigation	Information about the number of navigation pages, as well as navigation performance, query size, and result size by average, standard deviation, minimum, maximum, and total.
Record Sorting	The number and type of sorts performed (does not include timing), and the percentage of those sorts for each sort type.
Analytics	Information pertaining to the analytics features in Endeca Analytics, such as total processing time, query parsing, time checking and evaluation times.
Disk usage	Disk usage statistics for the indices: • current total disk usage value (MB) • disk usage high water mark value (MB)
Search	A finer-grained analysis of the performance of individual features. This information is used for the internal analysis by Endeca.
Data Layer Performance	Statistical information about the data layer performance. This information is used for the internal analysis by Endeca.

Note: For some of the statistics on this page, it is possible to drill down for further information by clicking the black arrow that appears outside the rightmost column.

Note: If you modified the `stats.xslt` style sheet that is included in the installation, the information might display differently.

About the Agraph Statistics page

In an Agraph implementation, you can access the Agraph Statistics page.

To view the Agraph Statistics page, use the syntax as shown in the following example:

```
http://AgraphServerNameOrIP:AgraphPort/admin?op=stats
```

where `AgraphServerNameOrIP:AgraphPort` are the host name and the port number on which the Agraph is running.

You can determine the host and port of the Agraph on the EAC Admin Console of Endeca Workbench by opening the MDEX Engine component, or by exploring your Deployment Template `AppConfig.xml` file and information in it about the Agraph.

To reset the statistics, make the following request:

```
http://AgraphServerNameOrIP:DgraphPort/admin?op=statsreset
```

To view the statistics information for a single request, clear statistics, issue a request and inspect statistics again.

The statistics page information is valid as long as the MDEX Engine keeps running; it is cleared upon the MDEX Engine restart.

Note: Unlike the Dgraph Stats page, the source for the Agraph Stats page is stored in an HTML file, not an XML file.

Example of the Agraph Statistics page

This table describes parameters listed on the Agraph Stats page. It is followed by an example of the Agraph Stats page:

Agraph Stats page parameter	Description
`Version`	The version of the MDEX Engine that is being used
`PID`	Process ID
`UID`	User ID
`GID`	Group ID
`CWD`	Path to the current working directory
`Data Path`	Path to the updates directory of Agidx
`Num requests`	Number of query requests made to the Agraph
`Num requeries`, `Num re¬queries failed`	Information about Agraph performance. This information is used for the internal analysis by Endeca.
`current_process_size (MB)`	The size of the Agraph process in memory

Agraph Stats page parameter	Description
`argc`, `argv`	The `argc` parameter indicates the number of arguments listed with `argv`. The `argv` parameter lists the arguments with which the Agraph has been started, such as the Agraph flags, location of the log file, the Agidx data file, and the hosts and ports of the child Dgraphs.
`Result page stats`, `System usage`	Various statistics on Agraph query performance and system usage. This information is used for the internal analysis by Endeca.

This example illustrates typical results returned by the Agraph Statistics page:

```
Endeca Navigation Engine (agraph) Server Statistics

--------------------------------------------------------------------------
-------
Current time: [Date and Time]
Avg. Throughput (10 sec.): 0.67 req/sec
Avg. Throughput (1 min.): 0.67 req/sec
Avg. Throughput (5 min.): 0.67 req/sec

General Information
Version: [version used]
PID: 1742 (PPID: 1)
UID: 10094, userID (EUID: 10094)
GID: 10094 (EGID: 10094)
CWD: /current_working_directory/ene/agraph
Host: agraph_host
Server Port: 5555
Start Time: [Date and Time]
Data Path: /localdisk/ene/agraph/updates/agidx
Data Tag: unknown
Data Date: [Date and Time]
Last Request Time: [Date and Time]
Num Requests: 6
Num Requeries: 2
Num Requeries Failed: 0
current_process_size (MB): 85.6523

argc: 14
argv[0]: /localdisk/dev/x86_64pc-linux/bin/agraph
argv[1]: --no-partial
argv[2]: --stat-brel
argv[3]: --port
argv[4]: 5555
argv[5]: --pidfile
argv[6]: /localdisk/dev/ene/agraph/updates/_agraph.pid
argv[7]: --log
argv[8]: /localdisk/dev/ene/agraph/updates/_agraph.log
argv[9]: /localdisk/dev/ene/agraph/updates/agidx
argv[10]: --child
argv[11]: localhost:5590
argv[12]: --child
argv[13]: localhost:5591
```

```
Result Page Stats
Num result pages served: 3
Result page format performance (milliseconds): avg=0.113037 stdDev=0.0800197
 min=0.0219727 max=0.172119
Result page size (bytes): avg=6684.67 stdDev=5314.51 min=548 max=9753

------------------------------------------------------------------------
-------

System Usage
Rusage (user time): 0.0560 seconds
Rusage (system time): 0.0120 seconds
```

About the Endeca MDEX Engine Auditing page

The MDEX Engine Auditing page lets you view the aggregate MDEX Engine metrics over time. It provides the output of XML reports that track ongoing usage statistics.

These statistics persist through process restarts.

This data can be used to verify compliance with licensing terms, and is also useful for tracking product usage.

Note: Each Dgraph in an implementation is audited separately.

Viewing the MDEX Engine Auditing page

You can request the MDEX Engine Auditing page with the URL below.

To view the MDEX Engine Auditing page:

Access the following URL:

```
http://DgraphServerNameOrIP:DgraphPort/admin?op=audit
```

For example, if your Dgraph is running on your local machine and listening on port 8000, specify this:

```
http://localhost:8000/admin?op=audit
```

The information on the MDEX Engine Auditing page is persistent and is valid across the MDEX Engine restarts.

The source data for the auditing reports is stored in XML. By default, the MDEX Engine Auditing page is rendered into HTML through an Endeca XSLT stylesheet, `audit.xslt`, that is installed in the `ENDECA_MDEX_ROOT/conf/dtd/xform directory`.

Audit persistence file details

The naming convention for the audit persistence file is: `audit-<data_prefix>-<agidx_persistence_number>.xml`.

For example, an audit persistence file on the sample wine implementation might look like this: `audit-wine-0.xml`.

This convention ensures that each Dgraph creates a unique file. It makes it possible to maintain the audit persistence files for numerous Dgraphs in an application in the same directory without contention.

By default, the audit persistence file is written to a directory called persist that is located in the application's working directory. To direct it elsewhere, use the Dgraph flag `--persistdir` when you first create the Dgraph. Do not move or rename this directory after it has been created.

You should not delete the audit persistence file or attempt to edit it manually. Upon startup, the Dgraph checks for the presence of this file, and if it cannot find it or read it, it issues a warning message and creates a new one. However, if you see such a warning message when you first create a Dgraph, you can safely disregard it.

Note: The auditing function adds information prefixed by the word Endeca.* to records. This namespace is reserved for administrative use and should not be used for other purposes.

Sections of the MDEX Engine Auditing page

The MDEX Engine Auditing page consists of two tabs: Audit Stats and General Information.

Auditing statistics are gathered in one of two ways:

- The Query Load statistic tracks the hour with the most queries in each calendar week, starting when you first run the Dgraph and persisting through process restarts.
- All other auditing statistics constantly monitor the peak value over the course of a calendar week, and report the exact time when a value greater than the current peak value appears, starting when you first run the Dgraph and persisting through process restarts.

  Because these metrics are calculated over the course of a week, a change such as a deleted record is not reflected until the following week, when the peak value count is reset.

The Audit Stats tab

The Audit Stats tab contains the following information.

Section	Description
Query Load	The peak number, in the week beginning at the displayed time, of queries that the Dgraph has received in any single hour, plus the time at which that peak occurred. This field contains the sum of the next two fields, Net Query Load and WS Query Load. Depending on the modes in which you run your Dgraph, there may be values in both of these fields or only one of them.
Net Query Load	The peak number, in the week beginning at the displayed time, of queries that the Dgraph has received in any single hour while running in Presentation API mode, plus the time at which that peak occurred.

Section	Description
WS Query Load	The peak number, in the week beginning at the displayed time, of queries that the Dgraph has received in any single hour while running in Web services mode, plus the time at which that peak occurred.
Number of Records	The peak number, in the week beginning at the displayed time, of records, plus the time at which that peak was reached.
Number of Columns	The peak value, in the week beginning at the displayed time, for the total number of properties and dimensions across all records, plus the time at which that peak was reached.
Number of Words	The peak value, in the week beginning at the displayed time, for the total number of words (counting multiple occurrences of the same word) across all records, plus the time at which that peak was reached.
Number of Assignments	The peak value, in the week beginning at the displayed time, for the total number of populated dimension and property values across all records, plus the time at which that peak was reached.
Size of Data	The peak value, in the week beginning at the displayed time, for the total size occupied by all records, plus the time at which that peak was reached. **Note:** This may vary, depending on operating system platform.

The General Information tab

The General Information tab contains the following sections.

Section	Description
Information	Basic connection and machine details.
Arguments	A list of all arguments the Dgraph was started with.

Note: This tab is identical to the one of the same name on the MDEX Engine Server Statistics page.

Appendix E

Useful Third-Party Tools

This section lists some third-party tools that you may find useful during the Endeca performance monitoring process. The tools listed here are not supported by Endeca and are subject to change. In addition, these suggestions are not meant to overrule your choice of other tools.

Cross-platform tools

The following tools are available in both UNIX and Windows versions.

Tool	Description
Wireshark	Wireshark is an open source network protocol analyzer for both UNIX and Windows. It allows you to examine data from a live network or a capture file on disk. For information and downloads, see *http://www.wireshark.org/download.html*.
Tcpdump/Windump	Tcpdump (and its Windows version, Windump) are network traffic analysis tools. These tools can be used to watch and diagnose network traffic according to various complex rules. You can download Tcpdump from *http://www.tcpdump.org*. You can download Windump from *http://www.winpcap.org/windump*. **Note:** Tcpdump comes with most Linux distributions by default.

Solaris and Linux tools

The following tools are available for both Solaris and Linux.

Tool	Description

Netperf	Netperf is a network benchmarking tool that can be used to measure the throughput of many different types of TCP and UDP connections. Netperf provides tests for both unidirectional throughput, and end-to-end latency.
	
	Note: Be sure to compile netperf with histogram support.
	
	To simulate the network traffic to a MDEX Engine with average result pages of 50,000 bytes, run netperf like this:
	`netperf -l 600 -v 2 -H remotehost -p 8899`
	`-t TCP_CRR -- -r 200, 50000`
	
	where:
	• `-l` is the length of the test in seconds
	• `-v` specifies verbose output level
	• `-H` indicates the host where netserver is running
	• `-p` indicates the port that was given to the netserver process
	• `-t` indicates the test to run. TCP_CRR is the TCP test that opens a new TCP connection for each request/response
	• `-r` specifies the request/response characteristics, in this case a 200 byte request (approximately the size of a URL) and a 50K result
	
	For information and downloads, see *http://www.netperf.org*.

Top	Top is a UNIX utility you can use to quickly identify top CPU-using processes. It is a popular and common tool for monitoring system-wide process activity. For information and downloads, see *http://www.groupsys.com/top*.
Sar	Sar reports system activity on single processor systems. It reports the status of counters in the operating system that are incremented as the system performs various activities. These include counters for CPU utilization, buffer usage, disk I/O activity, TTY device activity, switching and system-call activity, file access, queue activity, inter-process communications, swapping and paging. On Solaris, sar is part of the system activity reporter package. On Linux, it is part of the downloadable sysstat package.
iostat	The iostat utility iteratively reports terminal, disk, and tape I/O activity, as well as CPU utilization. On Solaris, iostat is built in to the operating system. On Linux, it is part of the downloadable sysstat package.

Solaris-specific tools

The following utilities are built into Solaris.

Tool	Description
prstat	On Solaris the prstat command displays information about active processes on the system. By default, prstat displays information about all processes sorted by CPU usage.
cpusar and mpsar	On multiprocessor machines, cpusar reports per-CPU statistics, and mpsar reports system-wide statistics.
Kstat	Kstat reports many kernel parameters and statistics.
lockstat	The lockstat utility gathers and displays kernel locking and profiling statistics. It allows you to identify what are the processes and kernel really doing. Lockstat allows you to specify which events to watch, how much data to gather for each event, and how to display the data.
SE Toolkit	The SE Toolkit is a collection of scripts for performance analysis that gives advice on performance improvements.

Linux-specific tools

The following tools are available for Linux.

Tool	Description
sysstat	The sysstat utilities package is a download for Linux that contains performance monitoring tools such as iostat, sar, and mpstat. Iostat and sar are described in “Solaris and Linux tools” on ...; mpstat is described below. For information and downloads, see *http://perso.wanadoo.fr/sebastien.godard*.
Mpstat	Mpstat is the Linux multiprocessor load display utility. It displays system processor activity information on your screen for each of the processors serialized on your system.

Windows tools

The following tools are available for Windows.

Tool	Description
Task Manager	The Windows Task Manager provides information about programs and processes running on your computer. It also displays the most commonly used performance measures for processes. You can access the Task Manager by right-clicking an empty area on the task bar on your Windows machine.
Performance Monitor	The Performance Monitor provides details about the resources used by specific components of the operating system and by programs that have been designed to collect performance data.

Tool	Description
	You can access the Performance Monitor from the Control Panel by selecting Administrative Tools > Performance.
Other performance tools	Sysinternals (*http://www.sysinternals.com*) offers useful freeware tools, including the following: • Process Explorer, which shows you information about which handles and DLL processes have opened or loaded. • TCPView, which shows you detailed listings of all TCP and UDP endpoints on your system, including the local and remote addresses and state of TCP connections. On Windows NT, 2000, and XP TCPView also reports the name of the process that owns the endpoint.

Appendix F

Tuning the Network Performance

You only need to perform the procedures described in this appendix if you are installing in a production environment—they are not required for a typical developer installation. You will not see the benefits of this tuning until the Endeca server is placed under very heavy load.

Tuning network performance on Windows

Endeca provides two registry scripts that you can use, singly or in combination, to tune your server's network performance.

- The tcp_time_wait_tune.reg script tunes the server's network performance by changing the default time wait interval from 240,000 to 60,000 milliseconds. This change accelerates the rate at which the server re-uses ports when establishing TCP connections.

  To determine if you need to run the tuning script, open the Registry Editor and look for the following key:

  ```
  HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\
  Services\Tcpip\Parameters\TcpTimedWaitDelay
  ```

  Note: In the Registry Editor Explorer pane, expand the folders until you reach Parameters. Then click on the Parameters folder and look for the **TcpTimedWaitDelay** setting in the right pane.

  If this key does not exist, that means that the system is using the default time-out of 240,000 milliseconds.

- The tcp_max_ports_tune.reg script increases the number of ports available for TCP connections from 5,000 to 65,534. The affected key appears in the Registry Editor as follows:

  ```
  HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\
  Services\Tcpip\Parameters\MaxUserPort
  ```

To tune network performance on Windows:

1. In the `%ENDECA_MDEX_ROOT%\bin` directory, double-click one of the following scripts:
   - `tcp_time_wait_tune.reg`
   - `tcp_max_ports_tune.reg`
2. When the information box reading “Are you sure you want to add the information in *script name* to the registry?” appears, click **Yes**.

3. The system displays a confirmation message that reads “Information in *script name* has been successfully added to the registry.” Click **OK**.
4. Optionally, repeat these steps for the other tuning script.
5. Reboot the server for the registry changes to take effect.

Tuning network performance on Solaris

This section applies only to Solaris installations, not to Linux installations.

The Endeca installation includes a script that tunes the server’s network performance by changing the default time wait interval from 240,000 to 30,000 milliseconds. This change accelerates the rate at which the server re-uses ports when establishing TCP connections.

To determine if you need to run the tuning script, run the following command:

```
netstat -an | grep TIME_WAIT | wc -l
```

If the resulting number is consistently greater than 5,000, apply the tuning script and wait 4 minutes. The number of connections in a time wait state will drain off and you should find that the 5,000+ number drops by at least a factor of two.

To run the tuning script:

1. Change directories to the `$ENDECA_MDEX_ROOT/bin` directory.
2. As root, type the following at the prompt:
   ```
   ./tcp_time_wait_tune.sh
   ```
3. Press **Enter**.

A message appears indicating that the `tcp_time_wait_interval` has been set to 30,000.

Configuring the FIN_WAIT_2 timeout interval

The FIN_WAIT_2 timeout interval is the number of seconds that the HTTP server waits after sending the response for the client to close down its end of the socket. If this timeout expires, the server forcibly shuts down the connection.

This timeout interval is important for two reasons:

- Waiting for some time before shutting down the socket ensures that clients get complete responses.
- Timing out after certain period protects against buggy clients, which may never close their end of the socket. This can tie up resources on the server machine, leading to performance degradation and, in the extreme case, denial of service.

When the MDEX Engine finishes sending a response to a client, it does a "soft close" of the socket. This allows the client to finish reading data, and to close its end of the socket whenever it is ready. The state of the server-side socket during the interval between the server closing one end, and the client closing the other, is known as FIN_WAIT_2. All operating systems supported in this release automatically clean up sockets that stay in FIN_WAIT_2 for too long.

In general, you should not need to change this from the default value. If you do need to change the setting, follow the instructions below for your operating system.

Configuring FIN_WAIT_2 timeout on Linux

On Linux systems, the `tcp_fin_wait` timeout is stored in `/proc/sys/net/ipv4/tcp_fin_timeout`.

You can change the value of this parameter using the `sysctl` command.

To get the value, issue the following command:

```
/sbin/sysctl net.ipv4.tcp_fin_timeout
```

To set the value, issue the following command:

```
/sbin/sysctl -w net.ipv4.tcp_fin_timeout=30
```

Note: Root permissions are typically required to set this value.

Configuring FIN_WAIT_2 timeout on Solaris

On Solaris systems, you can modify the FIN_WAIT_2 timeout interval in `/dev/tcp`.

The default value is 675000ms.

To get the value, issue the following command:

```
ndd -get /dev/tcp tcp_fin_wait_2_flush_interval
```

To set the value, issue the following command:

```
ndd -set /dev/tcp tcp_fin_wait_2_flush_interval 30000
```

Note: Root permissions are typically required to set this value.

Configuring FIN_WAIT_2 timeout on Windows

On Windows systems, the variable to control the `FIN_WAIT_2` timeout interval can be modified in the Windows Registry.

The Registry entry that controls this setting is `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters`. You need to specify the `TcpFinWait2Delay` value for the above entry in the registry. The default value is 240s.

Note: Administrator privileges are required to set this value.

1. In the Windows Registry, go to `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters`
2. If the `TcpFinWait2Delay` value already appears in the details window, tune the value. The valid range is between 30s and 300s.
3. If the value does not exist, right click and select **Add a new DWORD value**. Add `TcpFinWait2Delay` and set its value.
4. Restart your system for the change to take effect.

Index