008

Best Practices for Sitecore

Performance and scalability tips for an optimal online experience

Author: Glen McInnis

nonlinear digital

a division of non-linear creations inc.

Nonlinear Digital is the full-service digital agency division of non-linear creations, with extensive knowledge and technical expertise in various CMS and marketing/analytics solutions, particularly the Sitecore platform. Our teams, working with you, use an agile methodology to create unique website and digital experiences that drive business results in a measurable way.

For more information on our services or to get in touch, please visit:

www.nonlinearcreations.com

Contents

Tables 4

Figures 4

Introduction 5
- For the speed reader

Sitecore terminology 7

How fast is fast enough? Defining performance requirements 8

Infastructure and setup 9
- What you need to know
- Hardware and network topology
- Configuration tasks
  - Storage locations
  - Caching
  - Prefetch

Development considerations for performance in Sitecore 14
- What you need to know
- Content structures
- Sorting
- Related content
- Sitecore and .NET coding practices
- XSL
- Sitecore debugger, code profilers and stress testing
- Caching specifics
- Static publication

Maintenance and operational procedures 24
- What you need to know
- Clearing old versions of content
- Monitoring and adjusting
- The sky is falling

Conclusion 27

About Nonlinear 28

Appendix A: Lucene Example 29

Tables

This whitepaper organizes important information about Sitecore settings and infrastructure into a number of tables:

Table 1: Measuring performance of Sitecore authoring tools 7
Table 2: Measuring performance of the website 8
Table 3: Site cache settings 12
Table 4: VaryBy cache parameters 24
Table 5: Threshold settings 27
Table 6: Error-handler settings 28

Figures

We've also included a number of figures and screenshots:

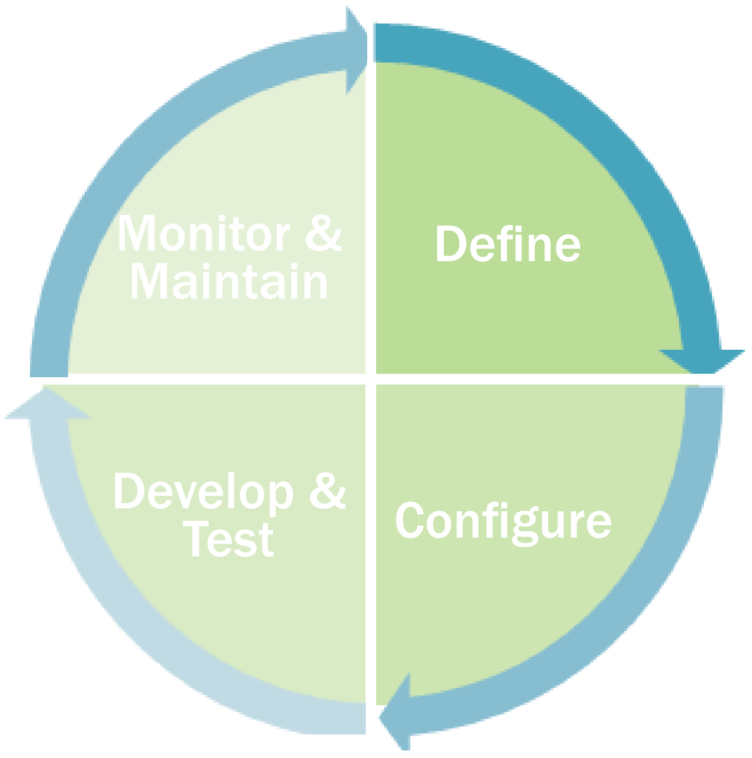
Figure 1: Deployment diagram 10
Figure 2: Prefetch configuration 14
Figure 3: Content organization: wide vs. deep trees 15
Figure 4: Sorting options dialog 17
Figure 5: Sitecore Debugger Hot Spots 21
Figure 6: Typical Sitecore caching model 22
Figure 7: Sitecore Debugger profile section 22
Figure 8: Rendering statistics (stats.aspx) 23
Figure 9: Cache settings for presentation objects 24

Introduction

All content management systems (CMS) offer promises of reduced IT expenditures, a streamlined content lifecycle, and a return of content control to the subject matter experts. Sitecore, an incredibly flexible CMS that fits well in the mid-market, can deliver on those promises, whether you're using version 5.3 or the newest incarnation of Sitecore—version 6.0.

Our previous whitepaper shared "lessons learned" during Nonlinear's real-world implementations of Sitecore, but it did not cover two very important considerations in a Sitecore build: performance and scalability. As a CMS that dynamically serves content from a database, users and potential purchasers of Sitecore are often concerned with how to create a best-of-breed implementation that accounts for both performance and scalability. In this whitepaper, we offer you the steps that your implementation and operations teams will need to take to ensure a high-performing and reliable web presence within the Sitecore CMS.

If you only have a few minutes to take in the key points of this document, I suggest reviewing the following diagram, which speaks to the overarching process required to ensure a responsive and scalable solution:



First, define the performance criteria for your Sitecore deployment. Second, purchase and configure a server & network environment that can deliver on these requirements. Third, the development team designs, implements and (most importantly!) tests the Sitecore solution to ensure it performs well in your technical environment. Fourth, the operational team monitors and proactively responds to any changing conditions, perhaps revisiting the original requirements and making some tweaks. Now, if you still have some time and are interested in learning more, read on!

For the speed reader

Each section in this document begins with a short summary of its key points. If you are short on time, read these first and then dig deeper when it seems relevant.

We sincerely hope you take advantage of Nonlinear's whitepaper series to give you a more complete view of the online environment.

Nonlinear's other whitepapers are available here: http://www.nonlinearcreations.com/Whitepapers

Sitecore terminology

This document assumes a familiarity with the Sitecore content management system. If you are comfortable with Sitecore concepts and web programming terminology, feel free to skip this section. If, on the other hand, you have little or no experience with the Sitecore CMS, then this brief overview of Sitecore concepts should render the rest of the document much more readable.

Branch Template (Version 6)

Used by content authors to create new content items Branch templates have the ability to also create "children" (descendants) of the new item at the same time the parent item is created.

Command Template (Version 6)

Used by content authors to create a new content item through the use of a custom wizard or dialog box.

Content Authors

Users of the Sitecore CMS, who are responsible for the input and approval of content. All content authors have a username and password to access the Sitecore CMS.

Content Item

A piece of content that has been created using a template and assigned to a location in the content tree. A content item can exist in one or more languages and may also be controlled by a workflow process.

Content Tree (Hierarchy)

The content tree is the method of organizing content items in Sitecore, using a folder-like structure.

Device

Like many CMS products, Sitecore allows you to separate content from its presentation layer. In addition, Sitecore allows you to define multiple presentation schemes for your content. These schemes are defined using a Device.

In Sitecore version 5.3.1, there are two default devices. The first is the standard HTML device, and the second is the print-friendly device. It is the responsibility of the Sitecore developer to create and assign the required presentation objects, which allow the devices to render content appropriately.

Layout

This is the base presentation object in Sitecore, which defines the general look-and-feel of a web page.

Masters (Version 5.3)

What content authors use to create content items. Masters include additional security information, which creates default settings for the content items.

Page Editor (Version 6)

This authoring interface is similar to the Web Edit mode of version 5.3 but includes improved functionality, such as inline editing and content author control over presentation items.

Rendering / Sub-layout

A presentation object assigned to areas of a layout, which renders specific content. Renderings are created using XSLT; sub-layouts using .NET code.

Templates

Templates define the structure and fields of content items. In Version 5.3, they are used by administrators to create content, but, normally, content authors make use of masters. In Version 6, templates are also used by content authors to create content.

Web Edit (Version 5.3)

The most common authoring interface used by content authors to browse the site and edit content. Making use of green dots and a floating menu, content authors are able to create and edit content.

How fast is fast enough? Defining performance requirements

In order to ensure your organization is getting what it needs from the CMS and website, you need to first define the level of performance you require. Performance can be separated into two areas: (i) performance of the website and (ii) performance of the content authoring interfaces. In both cases, the performance of the system will be greatly influenced by the estimated number of users.

To define your performance requirements for each system, simply complete the following tables. We have added some sample information for you, as well as a description of what each metric means:

Table 1: Measuring performance of Sitecore authoring tools

BACKGROUND INFORMATION	
How many users will have accounts in the Sitecore CMS?	50
Of the above users, how many of them will be logged in and working at the same time?	20
How many web pages does your site contain?	2, 000
How many images, documents and other files will be used in your site?	500
What is the approximate size of your site (in MB or GB)?	3GB
PERFORMANCE METRICS	
What is the system's maximum allowable time for publishing a single page?	5 minutes
What is the system's maximum allowable time for publishing the entire site?	10 minutes
What is the system's maximum allowable time for locking an item for editing?	5 seconds
What is the system's maximum allowable time for rendering a page in Page Editor?	30 seconds

Table 2: Measuring performance of the website

What is the average number of simultaneous visitors to your website?	10, 000
What is the maximum number of simultaneous visitors that you need to support?	30, 000
What is an acceptable average time to load a page that is 10KB?	5 seconds
What is an acceptable maximum time to load a page that is 10KB?	10 seconds

Filling in the above table will provide you with a baseline when developing test procedures for your CMS and website. Note that the above criteria are offered as a guide only; you may want to develop additional criteria for your project to account for your most important pages, major entry or landing pages or functionally-complex areas of the site.

Infrastructure and setup

Like any software application, Sitecore's performance is reliant on the following:

- Proper installation and configuration of the product
- Appropriate hardware
- A suitable network topology

Final choices on hardware and networks will depend on the size of your site, the level of functionality it requires and the number of visitors you need to support.

What you need to know

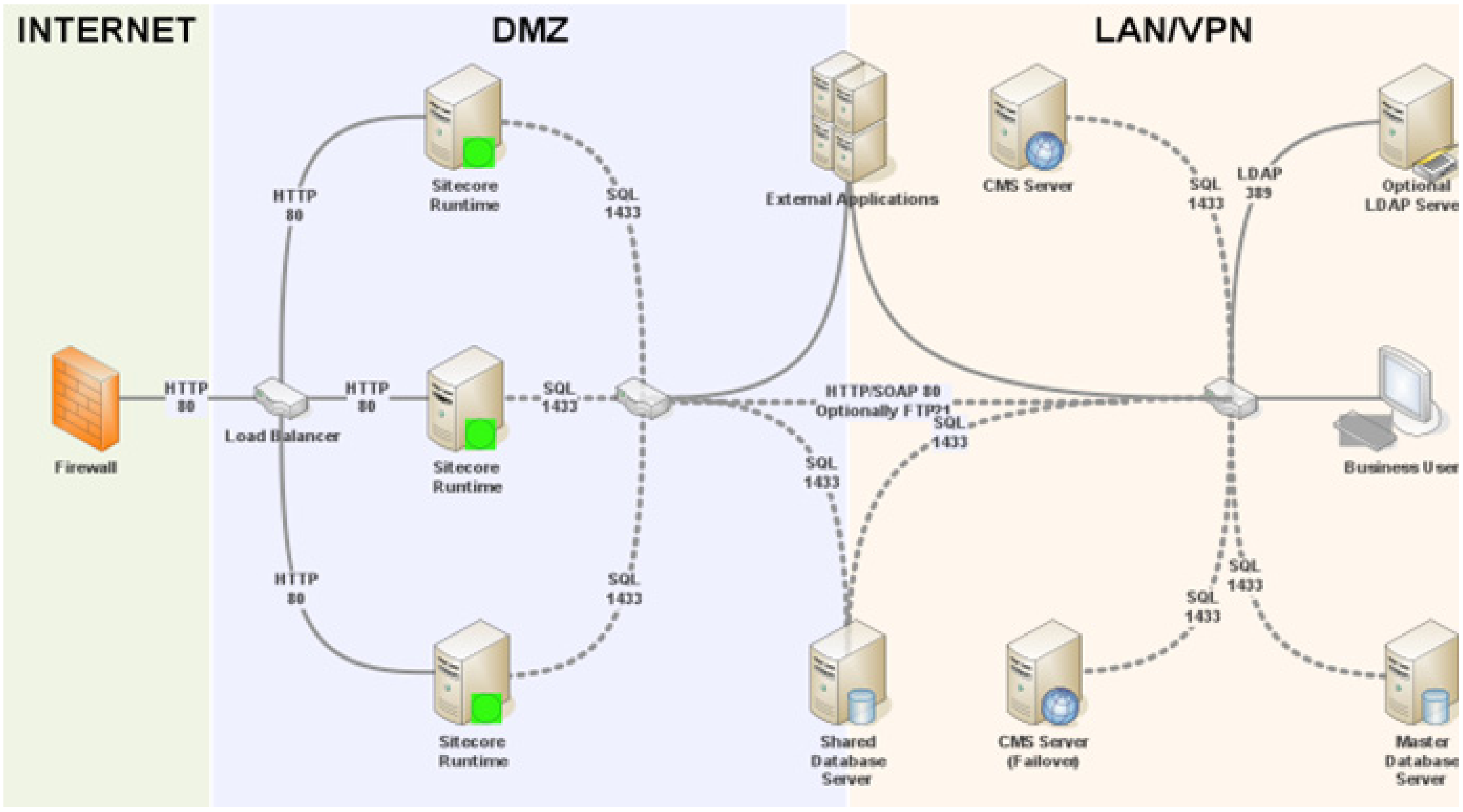
For optimal performance:

1. The content authoring server and content delivery installations should reside on separate servers — this is best for both security and performance
2. The database should not reside on the same server as Sitecore
3. Media and other large files should never be stored in the database

Hardware and network topology

In this section, we describe the ideal deployment topology of a productive Sitecore system. Not all organizations will require this level of complexity and performance. Consider the network diagram on the next page to see how it all functions.

Figure One: Deployment Diagram



In the above diagram, you can see that the Sitecore CMS authoring environment is separate from the actual website delivery nodes. This allows for the separation of performance requirements for content authors and site visitors. This configuration also increases security by removing the authoring environment from the public site. This deployment architecture includes multiple, load-balanced web nodes for added performance and fault tolerance.

The diagram also shows that all instances of Microsoft SQL Server (MS SQL) have been separated from the both the authoring servers and the web nodes. We strongly recommend the separation of the database from Sitecore, for a few different reasons:

1. MS SQL server does not perform well in most virtualized environments, whereas Sitecore can perform quite well. Virtualizing the Sitecore web nodes allows for the rapid deployment of additional capacity, if required.
2. The hardware requirements of MS SQL, when compared to Sitecore, are typically greater. By separating these specialized applications, you can make appropriate hardware choices for each.
3. When the database is isolated from Sitecore, it is much easier to diagnose performance issues.

Configuration tasks

Once Sitecore is installed, there are a number of configuration settings that can be adjusted, which tend to impact performance. The first category of settings defines storage location. The second category controls the caching behavior of the CMS.

Storage locations

There are a number of directory locations in the web.config file. The following excerpt from a Sitecore 6 web.config file shows the most relevant entries:

```
<sc.variable name="dataFolder" value="C:\Inetpub\uoa\data\" />
<sc.variable name="mediaFolder" value="/upload" />
<sc.variable name="tempFolder" value="/temp" />
<setting name="Media.CacheFolder" value="/App_Data/MediaCache" />
```

The web.config file references the above entries to control the majority of file location settings.

These are the two major elements of file system configuration:

- The datafolder is responsible for logs, viewstates, diagnostic counters, debug traces and packages—making this location a high-write situation for both the CMS and the web nodes. Ensuring the datafolder is located on a disk that can handle the write requirements aids in performance. You will want to consider this set-up for the tempFolder, as well, which is used as a temporary processing location.
- The Media.CacheFolder, which is used to store media that has been extracted from the database, is most relevant for the published site. Writes to disk are generally infrequent for this folder, making high-speed reading a greater priority.

Caching

As you probably already know, caching is one of the most complex topics related to performance. The impact of caching is subject to available hardware resources, the caching algorithm of Sitecore, the configuration approach you have taken and the behavior of your users. The only way you can know with 100% certainty what will happen is to try it. This makes planning and realistic pre-launch testing critical.

In this section, we outline the major types of caching in Sitecore and where the settings are configured. In the development section, we cover tech specifics for developers. Our intention here is not to dive into the nitty-gritty detail, but instead give you a solid overview of your options so that you are able to test the performance of your chosen strategy.

There are three levels of caching in Sitecore: the browser-level cache, the site-level cache and the database-level cache.

Site-level caching

Site-level caching is enabled by settings on sublayouts and renderings inside the Sitecore CMS. We recommend that you maintain the cache settings for renderings and sublayouts using the standard values for your template layout settings. This ensures that you do not mistakenly cache sublayouts and renderings that require dynamic generation on each load. These settings can be adjusted to suit the needs of the particular control, in terms of desired rendering, functionality and usage. The importance of this in relation to end usage is that the more of these variations you use, potentially the more cache space each rendering or sublayout will use, as a separate copy of the rendered version will be stored for each specific variation.

The site nodes have a number of settings that control caching, which are listed in the following table:

Table 3: Site cache settings

PARAMETER	DESCRIPTION
filterItems	If set to true, the site will always show the current version of an item (without publishing it).
filteredItemsCacheSize	Refers to the size of the cache used to store filtered items. The value should be specified in bytes, KB, MB or GB.
cacheHtml	If set to true, HTML caching will be enabled. If set to false, no HTML will be cached for any rendering. The default value for this is false.
htmlCacheSize	Refers to the size of the html cache. Specify the value in bytes or append the value with KB, MB or GB.
cacheMedia	If set to true, media caching will be enabled. If set to false, no media will be cached. The default value for this is true.
mediaCachePath	Refers to the path to the folder where media data will be cached. The default value for this is: {temp folder}/{site name}/mediacache.
filterItems	If set to true, the site will always show the current version of an item (without publishing it).
filteredItemsCacheSize	Refers to the size of the cache used to store filtered items. The value should be specified in bytes, KB, MB or GB.
cacheHtml	If set to true, HTML caching will be enabled. If set to false, no HTML will be cached for any rendering. The default value for this is false.
htmlCacheSize	Refers to the size of the html cache. Specify the value in bytes, KB, MB or GB.

Note that Sitecore differentiates a cache size for HTML vs. XSLT controls, allowing you increased granularity for tuning the performance needs of your particular site, but this is dependent on usage of pages heavy in sublayouts (HTML) or renderings (XSLT).

Sitecore also allows for a Media cache, which holds media files extracted from the database in a flat file format on the web server. This allows for faster response times than accessing the associated database blob. We don't recommend that you store media in the database.

For each of these groups it is important to spell out their mandate and responsibilities. The key group in this model is the ECM Governance Board; we provide some insight into how it is frequently structured.

Database-level caching

Each database has its own caching parameters. These parameters define the allowable cache size for data, items, paths and standard values. The following extract from the web.config entries of Sitecore 6 show an example of possible settings:

```
<cacheSizes hint="setting">
            <data>20MB</data>
            <items>10MB</items>
            <paths>500KB</paths>
            <standardValues>500KB</standardValues>
</cacheSizes>
```

This review of cache settings will be further expanded in the Development section of this whitepaper.

Prefetch

In version 5, Sitecore introduced a new feature called Prefetch. Prefetch allows you to configure Sitecore to pre-load the item cache upon start-up. The benefit is a smoother experience for users, as they do not have to endure the waiting time of the initial non-cached request. The trade-off here is a longer start-up time.

In Sitecore, prefetch options are configured for each database. There is also a single, shared configuration. The common configuration file and the .config files for each database are located in the App_Config/Prefetch directory.

For optimal site performance, adjust the prefetch entries of the web database (App_Config/Prefetch/web.config). The following is a slightly modified listing of the web database default entries found in Sitecore 6. The additional inline comments explain what each entry does.

Figure 2: Prefetch configuration

```
<configuration>
   <!— Size of the cache -->
   <cacheSize>10MB</cacheSize>

   <!—
          Loads all items based on system template:
          /sitecore/content/templates/system/alias
   -->


   <template desc="alias">{<alias-guid>}</template>
   <!—
          Loads all items based on system template:
          /sitecore/content/templates/system/Layout/Layout
   -->
   <template desc="layout">{<layout-guid>}</template>

   <!—
          Loads all items based on system template:
          /sitecore/content/templates/system/Layout/Renderings/Xsl Rendering
   -->
   <template desc="xsl rendering">{<xsl-guid>}</template>

   <!—
          Loads the "home" node
   -->
   <item desc="home">{<home-guid>}</item>
   <!—
          Loads all children of the "home" node
   -->
   <children desc="main items">{<home-guid>}</children>
</configuration>
```

Development considerations for performance in Sitecore

The performance of any software application is most significantly affected by the quality of its configuration and coding. Sitecore has a few specific areas where the development team needs to take performance into careful consideration. In this section, we discuss these areas and what your implementation team needs to do in order to ensure a robust and stable solution.

What you need to know

1. In Sitecore, the way content is structured and organized is very important, so watch out for shallow or broad content trees
2. Relationships between content can cause unexpected looping (and the related challenges). Using a Lucene index can prove a more efficient means of determining content relationships
3. Caching is an art form. Watch your site's performance and tweak settings as required
4. Make full use of the Sitecore Debugger, and be sure to stress-test your implementation before going live
5. Profile your code. This can show you how it is interacting with Sitecore
6. Rely on Sitecore sorting rather than customized .NET sorting code to sort on the published site
7. If all else fails, static publication is an option

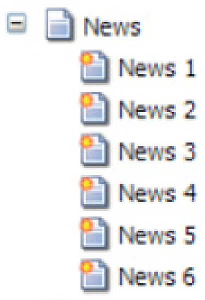
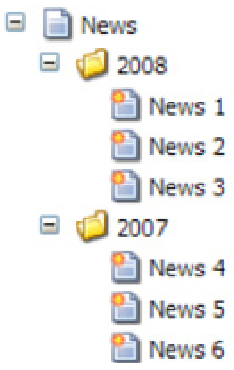
Content structures

Content in Sitecore is organized using the Content Hierarchy. There are a few basic considerations to take into account when designing your content tree.

When the tree is too broad, the authoring interfaces are often required to load a large number of items at once. This can also create performance problems in the rendering of the website. If possible, structure the content so that no node in the tree has more than 100 siblings.

To go deeper into this concept, let's consider a simple example: news releases.

How do we best structure this content? One option is to organize all news releases into single folder. The second option is to organize the releases into subfolders by year.

NEWS IN ONE FOLDER	NEWS ORGANIZED BY YEAR
 News News 1 News 2 News 3 News 4 News 5 News 6	 News 2008 News 1 News 2 News 3 2007 News 4 News 5 News 6

Consider the news release page for 2008. When the news releases are organized into a single folder, the page must load all news articles and then filtre out the news releases that are not for 2008. This forces Sitecore to make a lot of unnecessary calls to either the database or the cache, and then perform extra processing to eliminate the unneeded data.

When news releases are stored by year, Sitecore can simply load all the releases from a single folder and display them. We can see how this choice can significantly impact site performance.

If we consider how product pages relate to news items, we can also see how the choice of content structure can also impact the performance of the authoring environment. Using the single folder option, the amount of information that Sitecore needs to load is much greater. While it's not an issue with only six news items, 6000 news items would be quite problematic. Organizing news releases by year would alleviate the problem.

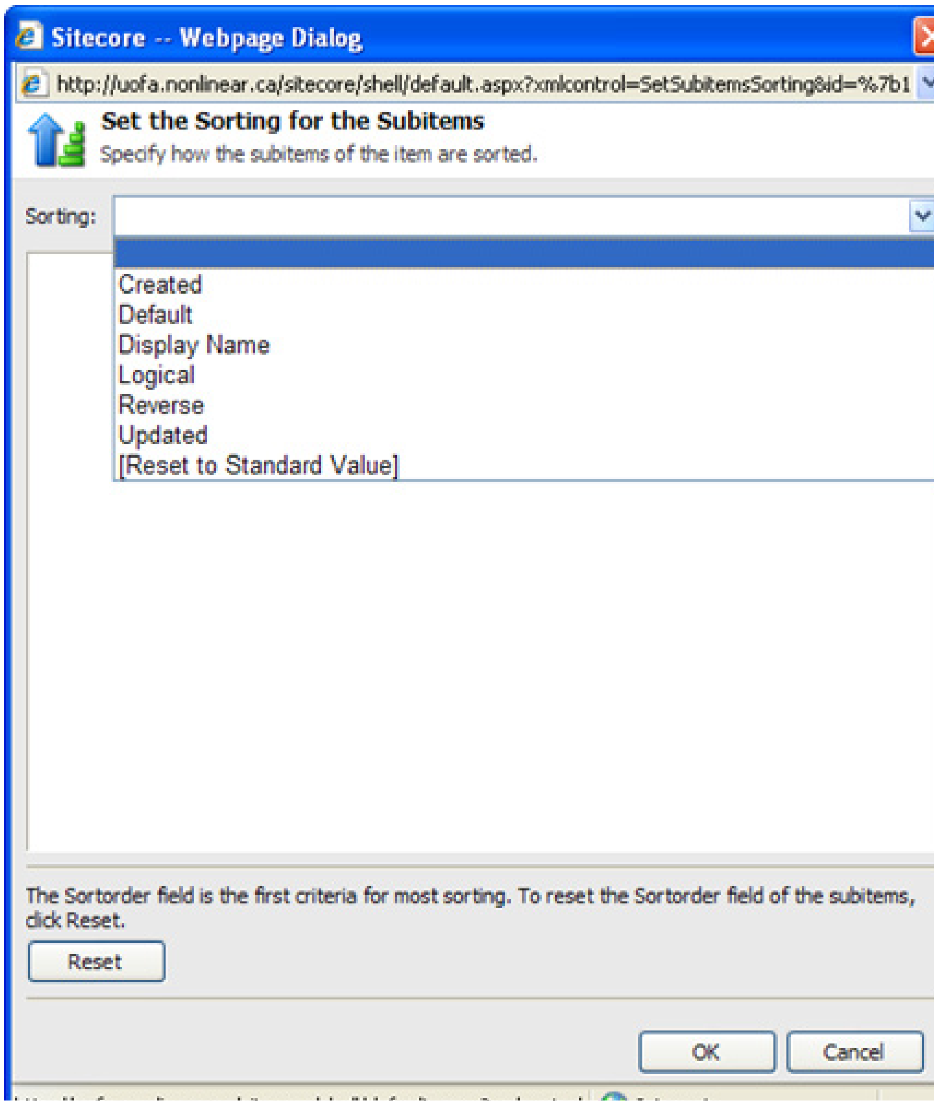
In short, be careful of the way content is organized in the Sitecore tree, and remember that content organizing strategies have implications for both the published site and the content authoring environment.

Sorting

Organizations often require their published sites to sort content items based on a field value. Common examples would be alphabetical sorts based on title, or chronological sorts based on a date or time value. Understanding how Sitecore performs this sorting of content items is an important part of developing the most efficient code.

Basically, Sitecore sorts children of any given item using the SortOrder field. If this field is not filled or the values are all equal, then Sitecore sorts according to the Subitems Sorting field of the parent item. This can be seen clearly in the following sorting dialog screenshot.

Figure 4: Sorting options dialog


Sitecore -- Webpage Dialog
http://uofa.nonlinear.ca/sitecore/shell/default.aspx?xmlcontrol=SetSubitemsSorting&id=%7b1
Sorting:
Created
Default
Display Name
Logical
Reverse
Updated
[Reset to Standard Value]
Reset
OK
Cancel

Set the Sorting for the Subitems

Specify how the subitems of the item are sorted.

The Sortorder field is the first criteria for most sorting. To reset the Sortorder field of the subitems, click Reset.

Note that it is also possible to create your own sorting behaviors and register them with Sitecore by implementing a customized class based on the Comparer abstract class. If you look to the **Core** database, then/Sitecore/content/Settings/Subitems Sorting you will find the registration of the existing sorting methods. Once you have created your own sorter class, simply register it in this location by creating a new item.

At Nonlinear, we prefer this method of sorting, as it provides explicit sorting control for the content authors and performance benefits for the published site (which no longer needs to run the sorting code). The sorting code option inside Sitecore only runs in two circumstances:

(a) When interacting with the content-editing interfaces, and, (b) During the publication of content items. Once content items have been published to the live site, the order is explicitly set. This means that there is no sorting code or logic executed in the published website.

Related content

Many of our clients at Nonlinear request that content authors be able to tag content during the publishing process. Using these tags, the content is automatically pulled into related positions on the website. If we continue with our news release example from before, that would mean that news releases could be tagged according to related products. By tagging content in this way, the related news items would automatically appear on each product page without the intervention of the content author. There is no problem with this, so long as the right approach is employed. To maximize performance, at Nonlinear we make use of customized Lucene indexes on the tag fields. This allows us to create efficient Lucene queries to quickly retrieve content with particular tags as opposed to scanning large numbers of content items.

The first step in this process is to set up the appropriate Lucene index:

```
      <index id="relatedLinks" singleInstance="true" type="Sitecore.Data.Indexing.Index,
Sitecore.Kernel">
        <param desc="name">$(id)</param>
        <!--<templates hint="list:AddTemplate">
          <template>Section Search</template>
        </templates>-->
        <fields hint="raw:AddField">
          <field target="id" storage="unstored">@id</field>
          <field target="tags">tags</field>
          <field target="includeinsearchresults">include in search results</field>
        </fields>
      </index>
```

In this example, a Single-Line Text field called “Tags” is being used to populate a Lucene index call relatedLinks. The sample .NET code that would be used to query this index has been included in Appendix A.

Sitecore and .NET coding practices

We won’t bore you to death by exploring all the ins and outs of optimal coding in .NET — if you need this kind of information, there is excellent content available from Microsoft, and the web at-large. What we want to focus on instead is a discussion of the problem areas we often encounter when auditing existing Sitecore implementations:

1. NET string manipulation capabilities (like string concatenation, String.Format and StringBuilder) need to be used correctly. Though, what is correct for your situation depends on the performance characteristics you are trying to achieve. Lower memory use? Faster execution? Both?
2. Sitecore does cache, but don’t forget you can still access the Cache features of the .NET framework. This can be especially helpful if you want to explicitly control the caching of complex queries to Sitecore or other systems. Complex navigational systems (such as sites using faceted navigation) will often utilize such an approach.
3. Creating new hooks or pipeline additions in Sitecore is a common means to extend functionality; however, before deciding on this approach, be sure you understand when this new code fires. If you are unsure, create a simple stub and have it log out, or trace it through your debugger.

1. Data providers are an excellent way to integrate external data into Sitecore; however, when they're chained into the Master database (the way it's described in articles on the Sitecore Developers' Network, or SDN), you will add significant overhead to each item request, burdening the master database. Instead, consider using the Sharepoint Connector, which has been architected with its own database and cross-proxy. We have found this to be much more efficient.
2. Make use of automated tools to test your .NET code. At Nonlinear, we really like the Red Gate suite of products for profiling our .NET code and locating potential problem points. The free FxCop scanner from Microsoft can also help you to find problematic code.

XSL

The XSL vs. .NET debate is always a lively one on the SDN forums. All being equal, at Nonlinear we take the approach that XSL is more than acceptable for content that has no complex rendering logic, such as conditionals or advanced queries. In our experience, the performance improvement gained by going to .NET in such cases is minimal and, sometimes undetectable. We prefer XSL in these instances, as renderings are much easier to update and do not require a recompile of the solution.

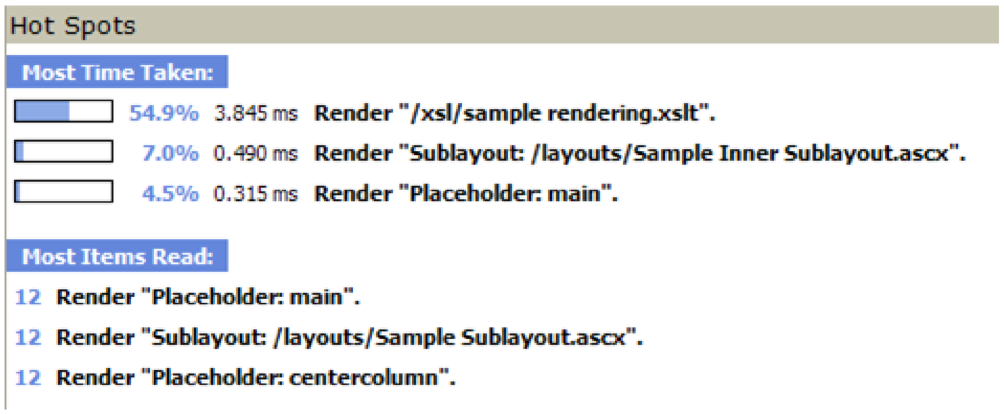
One quick tip: the Rendering Wizard in Sitecore automatically inserts a $home variable. Remove it unless you need it. If you are concerned about the performance of a rendering, use the Sitecore Debugger to determine the bottleneck. If required, you can always convert it to .NET.

Sitecore debugger, code profilers and stress testing

In previous sections, we talked about writing good code, which is fundamentally based on testing and validation.

There are three steps required of any Sitecore developer when validating code for performance:
1. Use the Sitecore Debugger as a means of identifying any underperforming code early on
2. Make use of .NET profilers to find .NET issues
3. Stress-test your code with realistic volumes of data and users—make the test scenario as close as you can to the reality of your eventual implementation environment

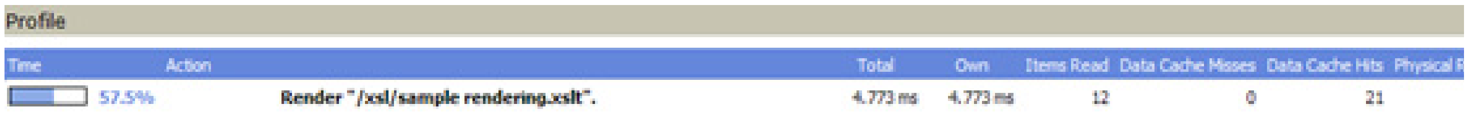
Now, let's look at the Sitecore Debugger. The first thing that needs your attention, on any given page, is the "Hot Spots" sections of the trace:



```
Hot Spots
Most Time Taken:
54.9%  3.845 ms  Render "/xsl/sample rendering.xslt".
 7.0%  0.490 ms  Render "Sublayout: /layouts/Sample Inner Sublayout.ascx".
 4.5%  0.315 ms  Render "Placeholder: main".

Most Items Read:
12  Render "Placeholder: main".
12  Render "Sublayout: /layouts/Sample Sublayout.ascx".
12  Render "Placeholder: centercolumn".
```

There are two main pieces of information to take notice of here. First, the time taken to display various renderings and sublayouts. Second, the “most items read” info. If a rendering is taking a long time but there are few item reads for that rendering, this often points to an algorithmic problem in the code. If, however, the item reads are high, this can point to excess Sitecore queries, unnecessary loading of items or a poorly-designed content tree. As part of Sitecore’s profile section, the Debugger provides counts of cache hits, cache misses and physical reads. If we look at the detailed profile information, we can see the performance information for sample rendering.xslt:



Profile

Time	Action	Total	Own	Items Read	Data Cache Misses	Data Cache Hits	Physical R
57.5%	Render "/xsl/sample rendering.xslt".	4.773 ms	4.773 ms	12	0	21	

This information tells us that 12 items were read and 21 cache hits were made in order to output this rendering. Examination of the logic in this rendering may give you leads for potential performance improvements.

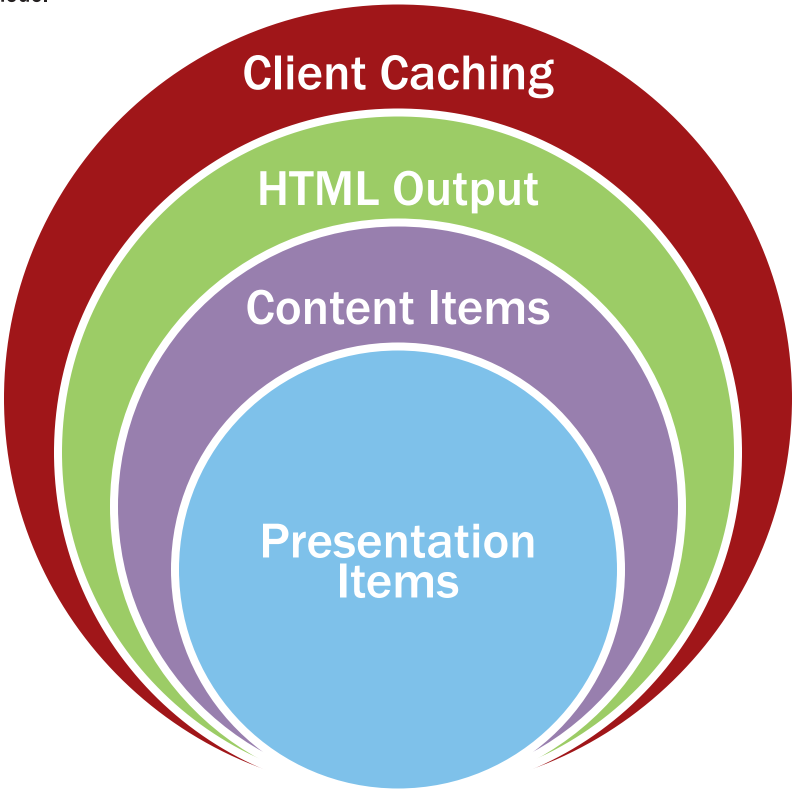
I am not going to dive into .NET profiling and stress-testing here, but I would suggest you take a serious look at the RedGate products for .NET profiling. They have been invaluable for us at Nonlinear in diagnosing underperforming .NET code during audits of existing implementations, as well as during our own development projects.

If you don’t have the budget for commercial stress-testing tools, there are also a number of free and open source tools. OpenSTA (http://opensta.org/) is probably our favorite. Microsoft also offers a number of tools for website stress-testing, including the free Web Application Stress Tool.

Caching specifics

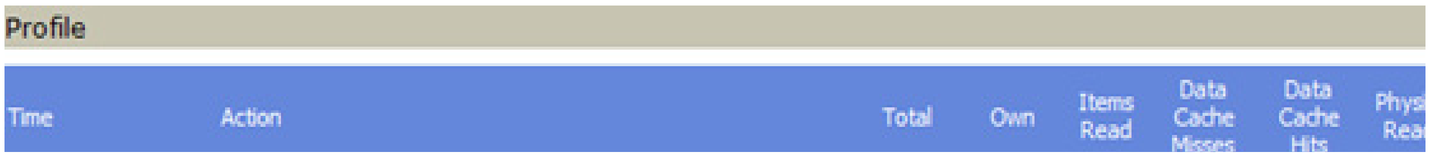
I have already discussed some of the basics of Sitecore caching. Now, I would like to delve into some of the greater detail Namely, what level of caching is appropriate for your circumstances? At its most basic, caching in Sitecore relies on caching the presentation objects themselves. From there, you simply need to decide how many additional layers of caching to add. See below for a figure that outlines a typical caching structure for Sitecore.

Figure 6: Typical Sitecore caching model



Sitecore is configured, by default, to cache items; however, what you will need to consider is how large the cache should be for each database/site. As your site grows, more content items will be used within the same timeframe. In order to properly maintain all items in the cache, you will need to increase the allocated cache size. There are a number of tools that you can use to monitor cache-hit performance. The first is the Sitecore Debugger. Part of Sitecore's profile section, the Debugger provides counts of cache hits, cache misses and physical reads.

Figure 7: Sitecore Debugger profile section



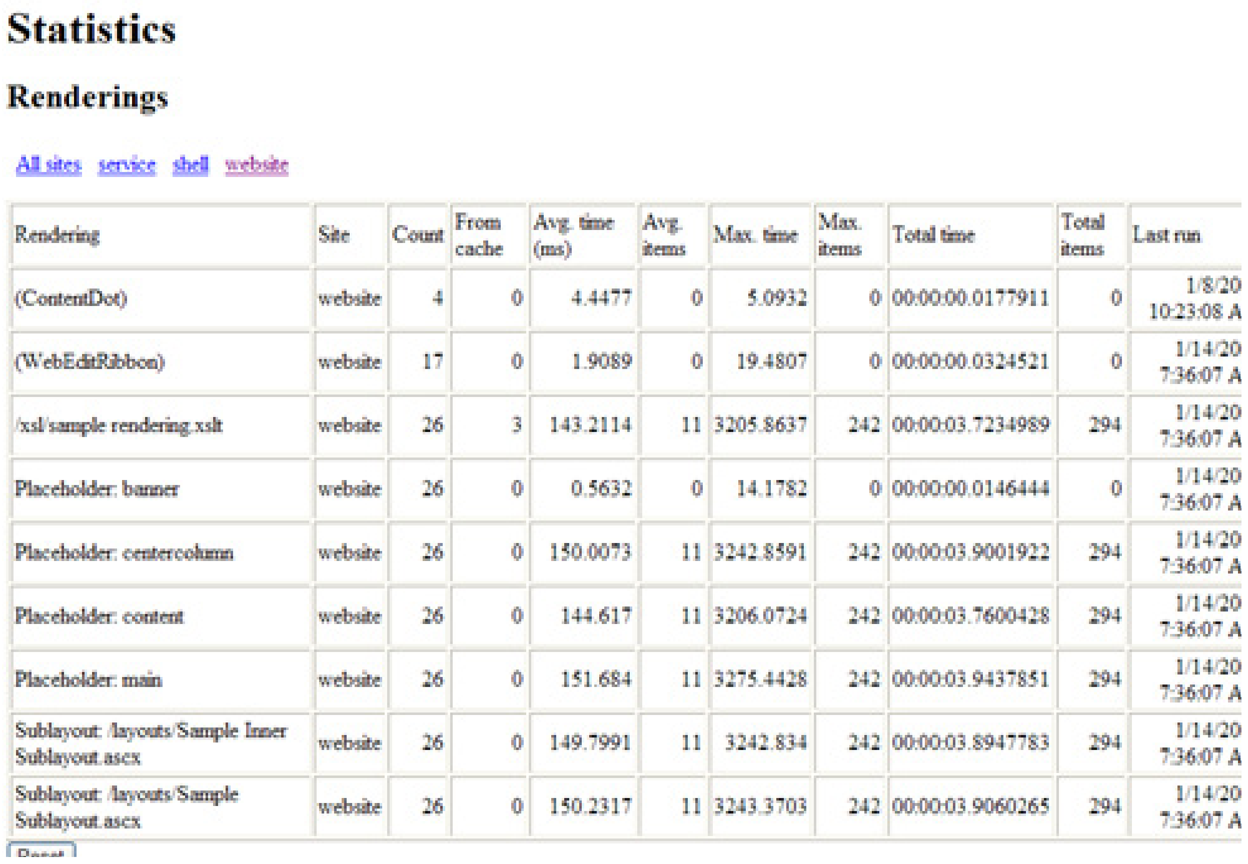
Profile

Time	Action	Total	Own	Items Read	Data Cache Misses	Data Cache Hits	Phys Rea

Ideally, item are always read from cache. As the Debugger is simply a snapshot in time, it does not give you a complete picture of your development environment.

If you are looking for a more complete summary of rendering statistics, the stats.aspx page, which is part of Sitecore's admin tools (/Sitecore/admin/stats.aspx), provides an excellent summary of presentation objects for each site. Using this tool, you can easily see average and maximum rendering times as well as the number of times the rendering was drawn from cache as opposed to being regenerated. See next page.

Figure 8: Rendering statistics (stats.aspx)



Statistics

Renderings

All sites service shell website

Rendering	Site	Count	From cache	Avg. time (ms)	Avg. items	Max. time	Max. items	Total time	Total items	Last run
(ContentDot)	website	4	0	4.4477	0	5.0932	0	00:00:00.0177911	0	1/8/20 10:23:08 A
(WebEditRibbon)	website	17	0	1.9089	0	19.4807	0	00:00:00.0324521	0	1/14/20 7:36:07 A
/xsl/sample rendering.xslt	website	26	3	143.2114	11	3205.8637	242	00:00:03.7234989	294	1/14/20 7:36:07 A
Placeholder: banner	website	26	0	0.5632	0	14.1782	0	00:00:00.0146444	0	1/14/20 7:36:07 A
Placeholder: centercolumn	website	26	0	150.0073	11	3242.8591	242	00:00:03.9001922	294	1/14/20 7:36:07 A
Placeholder: content	website	26	0	144.617	11	3206.0724	242	00:00:03.7600428	294	1/14/20 7:36:07 A
Placeholder: main	website	26	0	151.684	11	3275.4428	242	00:00:03.9437851	294	1/14/20 7:36:07 A
Sublayout: /layouts/Sample Inner Sublayout.ascx	website	26	0	149.7991	11	3242.834	242	00:00:03.8947783	294	1/14/20 7:36:07 A
Sublayout: /layouts/Sample Sublayout.ascx	website	26	0	150.2317	11	3243.3703	242	00:00:03.9060265	294	1/14/20 7:36:07 A

The most important thing a developer can learn is how to structure rendering and sub-layouts to maximize the benefits of caching. There are no hard-and-fast rules for this because the choices you make will depend on the size of your available cache and the design of your site. That said, here are some good rules of thumb:

1. Building a site with a higher number of small renderings and sublayouts gives you greater flexibility to control the caching of HTML

2. Navigational controls tend to use the VaryByData option unless content is personalized.

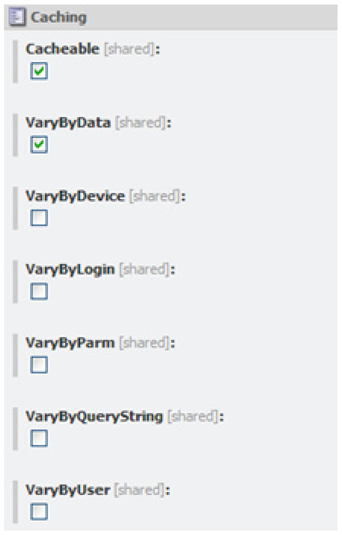
3. Portions of the header and footer often require no variation. Structuring renderings to allow for a single cached instance of such segments will greatly reduce requirements for cache size and make for better site performance.

4. Over-riding cache settings at an item level often makes sense for high-traffic pages, such as main section pages or campaign landing pages. But watch your analytics and use your own judgment.

5. If you have a high number of content items based on a single template, carefully consider how you will set up the caching. If caching is applied to all items, you may consume a large portion of the cache—especially if the traffic is distributed amongst the items (such as with news releases). Instead, you may consider removing the caching from the news renderings altogether. Or, if there are some high-hit releases, consider an item-by-item approach.

Now we come to the caching of the rendered HTML. As discussed earlier, Sitecore developers can control this in the standard values of the presentation objects, in the presentation configuration of templates or in an item-by-item configuration. In Sitecore, HTML caching must first be enabled for each site. This is done by setting the cacheHTML attribute to true. In the following screenshot, we can see the available Sitecore 6 caching options:

Figure 9: Cache settings for presentation objects



Caching

Cacheable [shared]: ✓

VaryByData [shared]: ✓

VaryByDevice [shared]: ☐

VaryByLogin [shared]: ☐

VaryByParm [shared]: ☐

VaryByQueryString [shared]: ☐

VaryByUser [shared]: ☐

To enable caching for any rendering or sub-layout, the Cacheable option must be selected. Then, optionally select from the six other “VaryBy” cache behaviors. If none of the VaryBy parameters are selected, the rendering will present the same content in all circumstances.

Table 4: VaryBy cache parameters

VARY	DESCRIPTION	MOST USEFUL WHEN
Data	Sitecore caches the output based on the item accessed. When the same item is accessed for the second time, the HTML will be loaded from the cache.	Content is highly consistent, such as headers and footers.
Device	Sitecore caches copies of the output for each Device being used.	A Device has been created to serve smart phones.
Login	Sitecore caches two copies of the output. One is for authenticated Extranet users and one is for unauthenticated Extranet users.	Sites require registration but present the same content to all users once registered.
Param	Sitecore caches the output for each parameter accepted by the rendering.	You are making use of rendering parameters.
QueryString	Sitecore caches the output for each unique combination of query string parameters.	Most commonly needed when forms or other functionality have been embedded into the site.
User	Sitecore caches the output for each authenticated user.	Personalized sites that deliver unique experiences to each user.

That should provide you with enough information to get started on caching. Next, we discuss what to do when caching isn’t enough.

Static publication

Sitecore is designed to serve and render database content. However, there are circumstances in which you may want to consider static publication of HTML files from Sitecore. While uncommon, we do see this in cases where a new promotion or event landing page is being set up and heavily promoted. Instead of loading the CMS with this (hopefully) overwhelming traffic, a static snapshot of the content is taken and stored on the web nodes.

At Nonlinear, we prefer to automate static publication as much as possible. By hooking into the publication process, it is possible to capture the results of the page and then store the resulting HTML and media to a specialized location for static serving of the content.

This process involves two major complications. Both are related to managing the mixture of content—which is to say, content that's dynamically served by Sitecore and content that is statically served by IIS from a disk.

In order to allow IIS to serve the static files, it is easiest to place those files in a specialized location. This allows for an optimal configuration of IIS, removing all Sitecore handlers from the execution.

There is also the issue of managing URLs in content. If a dynamically-served page links to a page that is static, the dynamic page must be aware of the alternate URL (to modify it) or URL redirects need to be put in place. Personally, I prefer the redirect imposed by the IIS service; however, this does require that Sitecore publish redirect rules to IIS. While not a complex task, this requires a hook into the Sitecore publication, or whatever program is being used, to grab the static HTML.

Maintenance and operational procedures

It may be routine and boring, but maintenance tasks and operational procedures keep everything running smoothly. Once you get these set, automate them as much as possible. Below, I've listed a few key tips:

What you need to know

1. Remove unneeded content, including old or unneeded versions.
2. Monitor the logs. Watch for warning signs of poor cache utilization and long render times.
3. Be prepared for the worst case. Have appropriate error messages and a fall-back site.

Clearing old versions of content

Large volumes of content place added stress on the database. In the authoring environment, this is particularly evident when content items have a significant number of versions. We generally prune old versions of content using a scheduled task and recommend that no content item have more than 15 versions.

But the question is, do you simply delete the old versions of content, or do you move them to an archival location? You'll have to weigh the benefits of having the instant availability of old versions—probably most critical if you have a requirement for e-discovery—against the options of delayed retrieval or outright deletion. With the exception of our clients in the financial sector, most will opt to archive old content in a separate location, accepting a slightly increased retrieval time in the unlikely event that the content is required.

Monitoring and adjusting

Sitecore and Windows server logs can provide a wealth of information and allow you to catch a potential issue before it becomes a real problem. If you examine the web.config file, you will find a number of threshold settings. Defining appropriate thresholds allows you to look for slow performance before it becomes critical.

In httpRequestEnd, you will find:

```
<TimingThreshold desc="Milliseconds">1000</TimingThreshold>
<ItemThreshold desc="Item count">1000</ItemThreshold>
<MemoryThreshold desc="KB">10000</MemoryThreshold>
```

In Settings you will find:

```
<setting name="Profiling.RenderFieldThreshold" value="100" />
<setting name="Profiling.SheerUIWarningThreshold" value="800" />
```

These threshold settings dictate under what conditions Sitecore will create a log entry for page execution. The complete list of options can be found in the table on the following page.

Table 5: Threshold settings

SETTING	PURPOSE
TimingThreshold	Defines the threshold value in milliseconds for logging long executing pages.
ItemThreshold	Defines the threshold for logging page executions attempting to read more than the specified number of items.
MemoryThreshold	Defines the threshold for logging page executions attempting to use more than the specified amount of memory.
Profiling.RenderFieldThreshold	Defines the threshold value in milliseconds for logging long field render operations.
Profiling.SheerUIWarningThreshold	Defines the number of milliseconds before logging a long Sheer UI request. Requires that Profiling.SheerUI = "true".
TimingThreshold	Defines the threshold value in milliseconds for logging long executing pages.

Ideally, your log-monitoring programs will be watching the Sitecore logs and can notify you when required.

The sky is falling

You've done your best but, for some unknown reason, everything has gone offline. Blame it on the gremlins if you like, but I always advocate for worst-case-scenario preparation.

First, make sure that you have configured Sitecore and IIS to return friendly error messages. If your site is going down for performance reasons, IIS is likely to return a HTTP error in the 500 range. If you look into the web.config file, you will find a number of settings that are relevant to error-handling. You will want to ensure the designated handlers are displaying appropriate messaging and performing any required notifications.

The following example shows the generic error handler in Sitecore:

```
<!--  ERROR HANDLER
Url of page handling generic errors
-->
<setting name="ErrorPage" value="/sitecore/service/error.aspx" />
```

The complete list of error settings can be found in the table below:

Table 6: Error-handler settings

NAME	DEFAULT VALUE	DESCRIPTION
RequestErrors.UseServerSideRedirect	False	If set to True, Sitecore will use Server. Transfer redirect requests to service pages when an error occurs. If False Response, Redirect is used.
ErrorPage	/sitecore/service/error.aspx	URL of generic page-handling errors
LayoutNotFoundUrl	/sitecore/nolayout.aspx	URL of 'Layout not found' page-handling errors
ItemNotFoundUrl	/sitecore/service/notfound.aspx	URL of 'Item not found' page-handling errors
LinkItemNotFoundUrl	/sitecore/service/notfound.aspx	URL of 'Link item not found' page-handling errors
NoAccessUrl	/sitecore/service/noaccess.aspx	URL of 'Access denied' page-handling errors

Even better than well-configured error messages, though, is having a hot standby site, which allows you to maintain your online presence no matter what. This standby site does not have to be a copy of the whole site, but it should provide enough information so your visitors know what options are available to them — such as contacting a call center or returning to the site once it is restored in a few hours. You get the idea.

There are two ways of going about standby sites. The first option is to have a separate Sitecore installation (preferably, in a separate datacenter in a different part of the country or world). This kind of installation typically involves only the web nodes but may also include the authoring environment. This environment is a (perhaps delayed) mirror of the actual site. In the event of catastrophic failure in your primary installation and datacenter, you switch over to this secondary system.

Obviously, the cost of the first approach is relatively high. A more moderate approach that also works well is to make use of the static publication described earlier to create snapshots of the site. This way, you can deploy limited, flat versions of the site. As very few sites are static, we would recommend that you also publish a simple message to indicate that site functionality is temporarily reduced.

Conclusion

Performance and scalability issues relating to any large-scale website are almost always complex. Luckily, Sitecore provides straightforward tools that deliver a high-performance and robust web presence. If you follow the general guidelines in this whitepaper, you should be well on your way to a successful implementation.

As always, additional information on Sitecore is available on the Sitecore Developer Network (http://sdn5.sitecore.net) and the Nonlinear Blog (http://blog.nonlinearcreations.com/).

If you find that you need further assistance in your Sitecore upgrade or implementation, you can reach us by email at info@ nonlinear.ca

About Nonlinear

Who is Nonlinear Digital?

Nonlinear Digital is the full-service digital agency division of non-linear creations. With offices across Canada, the USA and Brazil, Nonlinear has a track record of successfully planning, executing and measuring the effectiveness of digital marketing and outreach solutions for a range of clients.

What does Nonlinear Digital do?

We write digital strategy. We build eye-catching websites and we do it with a technical proficiency that is second to none. We help organize your digital team and educate your stakeholders. We help you extract meaning from data so you know if you're really driving business value from your investment.

We are a full-service digital agency with extensive knowledge and technical expertise in various CMS and marketing/analytics solutions. Our teams, working with you, use an agile methodology to create unique website and digital experiences that drive business results in a measurable way.

What makes Nonlinear different?

We put business first. After almost 20 years in the game, we have seen technologies and trends come and go. What has remained constant, however, is the need to drive tangible business outcomes from your digital spend. Today our focus is on creating an engaging online presence that:

- serves your customers' needs by being memorable, relevant and increasingly personalized and accessible on any device
- serves your business' needs by generating meaningful data from which you can extract real insights and metrics tied to your business goals

Contact us today

http://www.nolinearcreations.com

info@nonlinear.ca

APPENDIX A
Lucene Example

```
private void SearchInternal(string search)
{
    //Get index for the database by index name

    Database db = currentDatabase;

    Index index = db.Indexes[indexName];

    //If index doesn’t exist for the database then index == null
    Sitecore.Diagnostics.Error.AssertNotNull(index,
                                             "There is no " + indexName + " index on the current
database (" +
                                             db.Name + ")”);

    //IndexSearcher is the object which performs search through the index
    IndexSearcher searcher = index.GetSearcher(db);
    try
    {
        /*
         *An Analyzer builds TokenStreams to analyze the text.
         *The analyzer represents a policy for extracting index terms from the text.
         *A common implementation is to build a Tokenizer, which breaks the stream
         *of characters from the Reader into raw Tokens.
         *One or more TokenFilters may then be applied to the output of the Tokenizer.
         *spurious (stop) words using English dictionary
         */
        Analyzer analyzer = new StandardAnalyzer();

        /*
         * QueryParser obtains token stream using the Analizer and creates the Query
         * baseing on internal rules.
         * QueryParser need to know content field in index to be able to parse queries like
         * "Search” or "Example”. These queries will be transformed into
         * [content_field]:Search and [content_field]:Example
        */

        QueryParser qp = new QueryParser(Index.ContentFieldName, analyzer);

        /*
         * In default mode (OR_OPERATOR) terms without any modifiers are considered optional:
         * for example Test search is equal to Test OR search.
         * In AND_OPERATOR mode terms are considered to be in conjuction: the above mentioned
         * query is parsed as Test AND search
         */
        qp.SetDefaultOperator(QueryParser.OR_OPERATOR);
        /*
         * This method converts string query to object Query that can be used in searching
         */
        Query query = qp.Parse(search);

        TermQuery includeInSearch = new TermQuery(new 
Lucene.Net.Index.Term("includeinsearchresults", "1"));

        BooleanQuery fullQuery = new BooleanQuery();

        includeInSearch.SetBoost(0);

        fullQuery.Add(query, BooleanClause.Occur.SHOULD);
        fullQuery.Add(includeInSearch, BooleanClause.Occur.MUST);

        /*
         * Execute search using query and search objects
         * This method returns Hits object. It contains documents (search results) collection
         * and some additional result information.
         */
        //Hits hits = searcher.Search(query);
        Hits hits = searcher.Search(fullQuery);
        int docsCount = 0;

        //Hits.Length contains overall number of documents found
        for (int i = 0; i < hits.Length(); i++)
        {
            /*Hits.Doc(i) returns one Document (search result) by index
             *
             *Sitecore.Data.Indexing.Index has static method that can return Item
             * based on Document (search result) and database.
             *Be aware that it can be done only because during the indexing
             * indexer writes to the field _docID item ID. So if you want to override
             * default indexer class do not forget to put this info.
             */

            Item item = Index.GetItem(hits.Doc(i), db);
            if (item != null)
            {
                /*
                 * The easiest way to put search result on the page is creating
                 * Literal, filling it with propper value and adding it to the
                 * result placeholder
                */
                Literal lable = new Literal();

                lable.Text = string.Format("<b>{0}</b><br/><a href=\"{1}{2}\">Link</a><br/>", item.
Name,
                    item.Paths.IsMediaItem ? item.Fields["Path"].Value : Sitecore.Links.LinkManager.
GetItemUrl(item), "?sc_database=" + db.Name);
                SearchResults.Controls.Add(lable);
                docsCount++;
            }
        }
        //Add some overall search inforamtion
        SetResultsSummary(search, docsCount);
    }
    catch
    {
        //If search is failed for some reason it's necessary to inform user about it
        string errorText = Translate.Text("Error executing search request") + ". ";
        errorText += Translate.Text("Please input a valid search string") + ".";
        LiteralControl errorControl =
           new LiteralControl("<span style=\"color:Red;
        SearchResults.Controls.Add(errorControl);
    }
    finally
    {
        // DO NOT forget to close searcher. Because it may lock index.
        // Be careful not to call this method while you are still using objects like Hits.
        searcher.Close();
    }
}
```