PHP Applications: Increasing Performance

Thesis

Kip, R.J.
1532837

May 31, 2011

Thesis submitted to Hogeschool Utrecht

Revision 1.0

Revisions

	Description	
1.0	Compilation for hand in	May 31, 2011
0.3	Compilation for review by company supervisor	May 20, 2011
0.2	Compilation for review by school supervisor	May 17, 2011
0.1	Set-up of document structure	March 10, 2011

Revision 1.0

Printed at May 31, 2011

Company supervisor

D. J. van Roest

Reprovinci Internetdiensten bv

School supervisor, first examiner

E. Gerlofsma

Hogeschool Utrecht

Abstract

Reprovinci is a small company targeting small to medium businesses and non-profit organisations. They have their own content management system, which has, through the years, become slower. Reprovinci wished to counter this drop in performance. This was a challenge to me, as I have never had to analyse an ill-performing system. In the past months, it has been my task to analyse the cause of these performance issues, and propose and realise solutions.

Having set a bold goal, I did not achieve this goal, not having had enough time; there is still a lot to be done in order to achieve that goal. I recommend Reprovinci to implement the proces of code review to ensure future code will be of good quality, to upgrade software and change the database design of the main feature that drives their content management system.

This document starts with describing the situation and then mixes performance and optimisation theory and practice to form an informative document, serving as both my thesis and as a transferal of knowledge and recommendations to Reprovinci.

Contents

List of figures **5**

List of listings **7**

List of tables **9**

Preface **11**

1 Reprovinci **13**

2 Project summary **15**
2.1 Problem definition . . . 15
2.2 Thesis statement . . . 15
2.3 Goals . . . 15
2.4 Requirements . . . 16
2.5 Scope . . . 17
2.6 Conditions . . . 17
2.7 Stakeholders . . . 17
2.8 Assumptions . . . 18
2.9 Deliverables . . . 18

3 Management **19**
3.1 Quality assurance . . . 19
3.2 Risks . . . 19
3.3 Project timeline . . . 19

4 Current situation **21**
4.1 Functionalities . . . 21
4.2 Software . . . 22
4.2.1 Architecture . . . 22
4.3 Performance problems . . . 23

5 Performance **25**
To the reader . . . 25
5.1 What *is* performance? . . . 25
5.2 Performance analysis . . . 26
5.3 Metrics . . . 27
5.3.1 Response times . . . 27
5.3.2 Resource utilisation . . . 29
5.3.3 Throughput . . . 29
5.4 Performance before optimisation . . . 30
5.5 When to optimise . . . 31
5.6 Methodology . . . 31
5.7 Finding bottlenecks . . . 32
5.8 Design and performance . . . 33
5.9 Query optimisation . . . 34
5.10 Partial objects . . . 38
5.11 Benchmarking . . . 39

5.11.1 ApacheBench . . . 39
5.11.2 JMeter . . . 39
5.11.3 Couchmark . . . 40

6 Verification & validation **43**

7 Recommendations **49**
7.1 Improve quality of code and maintaining performance . . . 49
7.2 Reduce file caching . . . 50
7.3 Software . . . 50
7.4 Server architecture . . . 50
7.5 Alter UDO storage . . . 51
7.5.1 The entity-attribute-value model . . . 51
7.5.2 Document-oriented databases . . . 51
7.5.3 FriendFeed . . . 51
7.6 Alter UDO UX . . . 52

8 Conclusion **55**

Afterword **57**

Glossary **59**

Index **61**

A Optimisations **65**
A.1 Collection#_checkCallback() . . . 66
A.2 ObjectCollection#loadObjectInstances() . . . 67
A.3 In-progress loading . . . 69
A.4 User permissions . . . 70
A.5 `String::renderNameToKey()` . . . 72
A.6 NavigationCollection#getNode() . . . 73
A.7 Excessive calling of getNode() . . . 75
A.8 PageContentHandler#grabData() . . . 76
A.9 PageContentHandler#returnPageContentContainer() . . . 78
A.10 NavigationCollection buildup . . . 79
A.11 PublishedCollection#load() . . . 80
A.12 PublishedCollection#getNavigationNodeIds() . . . 81
A.13 Investige row parsing . . . 82
A.14 ObjectLoader: permissions . . . 83

B Testplans **85**

C Measurements **91**

List of Figures

1 My first experiences with computer programming 11

1.1 Organisational chart 13
1.2 The Reprovinci building. 14

4.1 Reprovinci's website and its administration area 22
4.2 Zend's Model-View-Controller (MVC) pattern [18] as used by the content management system (CMS), displayed in simplified form. 22

5.1 An example analytical model 26
5.2 The multiple phases of an HTTP request. 27
5.3 The three approaches to response time. 28
5.4 The content list's initial performance 30
5.5 The class `Collection`. 31
5.6 A ticket in Trac. 32
5.7 KCacheGrind shows `Authorisation->hasAccess`'s profiling information . 32
5.8 An excerpt from `ObjectCollection` 34
5.9 Situation after refactoring of `ObjectCollection` 35
5.10 A simple B-tree indexing last names 37
5.11 Simplified data structure for object instance RBAC. 38
5.12 Partial objects illustrated. 39
5.13 An excerpt from `ab`'s output. 39
5.14 JMeter's node based user interface 40
5.15 Aggregated test data in JMeter 40

6.1 Illustration of the sorting bug in `ObjectCollection` 45
6.2 A HTTP proxy recorder records HTTP traffic. 45
6.3 *small.local*'s content list, before and after 46
6.4 *medium.local*'s content list, before and after 47
6.5 *large.local*'s content list, before and after 48

7.1 An example database model for the FriendFeed approach 53

A.1 Situation after refactoring of `ObjectCollection` 68

C.1 *small.local*'s content list, before and after 92
C.2 *medium.local*'s content list, before and after 93
C.3 *large.local*'s content list, before and after 94
C.4 *small.local*'s detail page, before and after 95
C.5 *medium.local*'s detail page, before and after 96
C.6 *large.local*'s detail page, before and after 97
C.7 Editing of *small.local* content item, before and after 98
C.8 Editing of *medium.local* content item, before and after 99
C.9 Editing of *large.local* content item, before and after 100
C.10 Viewing of several pages from *small.local*, before and after 101
C.11 Viewing of several pages from *medium.local*, before and after 102
C.12 Viewing of several pages from *large.local*, before and after 103

Listings

5.1 Permissions are joined 37
5.2 Permissions are queried in a subquery 37
5.3 Example of a simple test plan using Couchmark 40
6.1 Testplan for the CMS' content list 44
A.1 Inlining of `_checkCallback` 66
A.2 Direct lookup in `Authorisation#hasAccess()` 70
A.3 Optimisation of `String::renderNameToKey()` 72
A.4 Direct lookup in `NavigationCollection#getNode()` 73
A.5 Caching strategy in `PageContentHandler#grabData()` 76
A.6 The `ContentCachingStrategy` interface 76
A.7 Change in permissions check in `ObjectLoader` 83
B.1 Testplan for the CMS' content list 85
B.2 Testplan for editing content items on the detail page 86
B.3 Testplan for viewing content items on the detail page 87
B.4 Testplan for viewing several customer website pages 88

List of Tables

2.1 The server requirements 16
2.2 Stakeholders with their interests and contributions 18

3.1 Table of risks and their probabilities and impacts 20
3.2 Project timeline 20

Preface

My passion with computer programming started around when I was nine or ten, creating message boxes using VBScript, drawing spirals and the lot in LOGO and making DOS programs for maths exercises using QuickBasic. When I was about twelve years old, I came in contact with Hypertext Markup Language (HTML) and JavaScript. Two years later, I was creating dynamic web pages using PHP: Hypertext Preprocessor (PHP). I soon decided I would study computer science.

I love the web as medium. It is a very accessible and dynamic piece of technology, both from a technological and from a human perspective. It can be driven by a plethora of platforms such as Java Enterprise Edition, PHP and Ruby. In fact, any platform that can work with sockets can serve web pages. And through the mainly standardised mark-up language, anyone with a browser can visit your creation. The end-user doesn't have to install anything but a browser to visit Google or your personal homepage.

When I started looking for a graduation internship, I applied at *Studentenbureau*. Studentenbureau acts as an intermediary between students and companies offering internships. They compare personal preference, experience and location, before pairing up a student and a company. After sending in my profile, meeting with a Studentenbureau representative, speaking out my preferences and doing a PHP test, I was invited for an interview at *Reprovinci*.

Reprovinci is a fairly small company, employing about twenty people. Due to their size, the atmosphere is informal and the organizational structure is flat; just what I like. I was explained Reprovinci has its own CMS, which has performance issues. This seemed like a challenge to me, as I had little experience with software optimisation.

During this project I have optimised their CMS and recommended several practices to prevent performance issues in the future. This thesis does not highlight every optimisation I have made, but tries to explain the theory behind performance issues and how I applied these during my internship. If you're interested, all optimisations can be viewed in short form in the appendix.

This thesis has been written in English so I don't have to come up with odd translations. Above all, I believe it enables me to write more interesting and readable text than I would be able to in Dutch.

I have found this project very versatile and enjoyed seeing the results of the optimisations. I have gathered more knowledge on how to prevent performance issues and, whenever they do pop up, how to find and ~~destroy~~ ~~smack~~ get rid of them.

For creating this opportunity and for guiding me, I would like to thank my company supervisor Dirk-Jan van Roest and my school supervisor Eric Gerlofsma. They have both

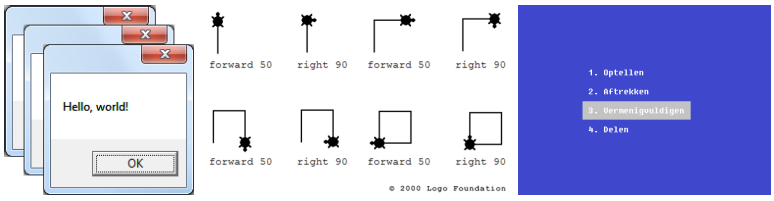


Figure 1: My first experiences with computer programming

generated helpful input.

If you come across any terms you do not understand while reading this thesis, you may find an explanation in the glossary on page 59.

Reinier
Schoonhoven, May 12, 2010

Chapter 1

Reprovinci

Reprovinci comprises two companies. The oldest of the two, *Reprovinci bv*, was founded in the year 2000 and is responsible for designing posters, brochures, cd booklets, branding, websites and much more. Their own printing office can print or press anything from posters and tickets to canvases and billboards. Large runs are outsourced.

Revasa Design was founded in the year 2001 and created websites while you and me had their evening off. Through extensive collaboration, Reprovinci bv and Revasa Design decided to engage in a closer relationship; Revasa Design was renamed to *Reprovinci Internetdiensten bv* (Reprovinci Internet Services) in 2005 and moved into the same building.

Nowadays, Reprovinci has its own CMS, and takes care of search engine optimisation (SEO) and digital newsletters, targeting the Dutch small and medium enterprises and (Christian) non-profit organisations.

In total, Reprovinci employs twenty people, three of which are full-time employees of Reprovinci Internetdiensten bv. Two of them are part-time employees. A fairly small enterprise, Reprovinci is a flat organisation. This is just what I like.

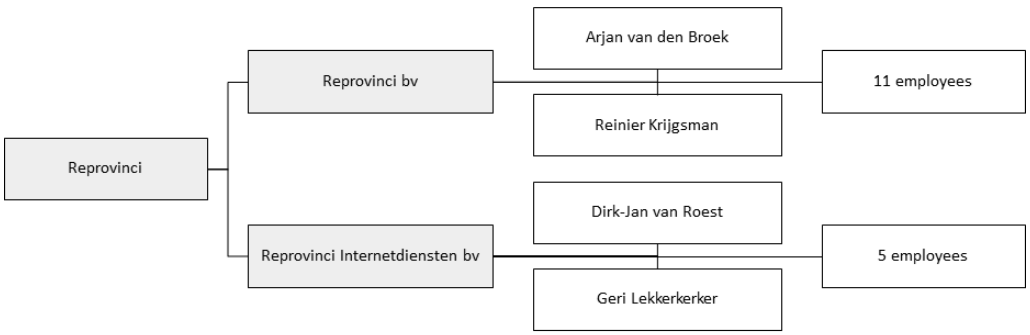


Figure 1.1: Organisational chart

Figure 1.2: The Reprovinci building.

Chapter 2

Project summary

2.1 Problem definition

Reprovinci Internetdiensten bv (from now on referred to as *Reprovinci*) decided to build its own CMS both out of stubbornness and out of want to keep full control of its features[1]. Even though it's object oriented and was initially well designed, the CMS has had may features added through the years, breaking its design. It now houses inefficient algorithms, performs too many database queries and suffers, to some degree, from clone-and-modify programming. Consequently, the performance of the system has decreased to such a level that customers started to complain. Additionally, the servers experience more stress than necessary.

To illustrate the performance of the system: a paginated list of the customer's content is shown in the administration area along with some navigation. With 50 of 1700 items, the page is generated in 5.5 seconds and reaches a peak memory usage of 40MB.

The problem definition that goes with this story reads as follows:

> "As the page structure and contents grow, the content management system becomes less usable, quickly reaching a point where it becomes unworkable."

2.2 Thesis statement

To address this issue, Reprovinci decided to initiate an optimisation project and hire an intern to fix the mess (just kidding). The thesis statement is:

> Improve the performance of the content management system.

2.3 Goals

To solve the performance issue, I am expected to:

- determine the causes of the low performance of the system;
- advise on how to solve these issues;
- address these issues;
- and to prove the performance has improved sufficiently.

The concrete goal of the project and the definition of "sufficient performance" is:

[1] For a more detailed description of the CMS, see chapter 4.

All pages must be generated within half a second.

2.4 Requirements

There are certain requirements that have to be met to validate the achievement of the project's goal. One of these requirements is the testing environment. To measure accurate performance results, I'd have to commandeer one of the production servers. They are, however, in use and indispensible. Next in line is the development server, where customer websites are developed and new CMS releases are tested. This server, however, doubles as a file server and experiences variable load throughout the day, resulting in inaccurate performance measurements. As accurate and repeatable measurements are paramount, I have chosen, in agreement with my supervisor, to setup my own workstation as testing platform. The hardware and software requirements of the server are described in table 2.1.

Hardware
2.8GHz dual-core 64-bit processor
2GB of RAM
7200rpm hard disk

Software
Apache 2.2.9
PHP 5.2.6
MySQL 5.0.51
Xdebug 2.0.5
Windows 7

Table 2.1: The server requirements

As the CMS' content can vary from customer to customer. They may use many different content types, have a lot of content entries or a combination of these two. To take these variances into account I chose to define three site blueprints. They are as follows:

- *small.local*;
  5 content types
  200 content entries
  3 versions per content entry
  10 navigation nodes in a balanced binary tree with 10 content entries per node

- *medium.local*;
  10 content types
  3 000 content entries
  4 versions per content entry
  30 navigation nodes, 50 content entries per node

- *large.local*;
  20 content types
  20 000 content entries
  5 versions per content entry
  100 navigation nodes, 500 content entries per node

The problem with this approach is that complex, relational data has to be generated that fits the database design. Generating such data is prone to error, be it with or without

tools. The data is also less representative than real data as provided by customer use. We therefore chose to ~~steal~~ ~~borrow~~ use the content of existing websites. These websites have similar characteristics to the blueprints described before. These use cases are as follows:

- *small.local*;
  19 content types
  approx. 200 content entries
  approx. 1.5 versions per content entry
  35 navigation nodes

- *medium.local*;
  26 content types
  approx. 7 500 content entries
  approx. 3 versions per content entry
  800 navigation nodes

- *large.local*;
  14 content types
  approx. 15 000 content entries
  approx. 3.5 versions per content entry
  75 navigation nodes

As you can see, *small.local* is quite a small website. *medium.local* has a large navigational structure and a lot of content types. *large.local* implements a web shop and offers many products, whose content has changes a lot, resulting in relatively many versions.

2.5 Scope

The concrete goal of generating all pages within 0.5 seconds narrows the scope considerably on one hand. On the other hand, pursuing this goal and this goal only would be very naïve. I have therefore chosen to draw up a wider scope.

To the scope will belong:

- researching the performance of the system on all fronts, focusing on response time;
- and increasing the response time of the system.

2.6 Conditions

To consider the project a success, the following conditions must be observed:

- the project's goal should be achieved before May 31, 2011;
- the thesis must be handed in before 12 AM on May 31, 2011;
- any changes made to the codebase should be backwards compatible;
- any changes that are not backwards compatible must be accompanied with documentation and/or up- and downgrade scripts;
- any changes must not be accompanied by an increase in server stress;
- and any changes must not result in an increase in memory usage.

2.7 Stakeholders

Multiple parties are involved with the project. You can see the parties and their interests and contributions in table 2.2.

[1] Just observations, not remarks.

	Interests	**Contributions**
Reprovinci	Reduced maintenance time Satisfied customer Less server capacity needed Project success	Knowledge of CMS Intern(ship) supervision Internship compensation
Customer	Reduced maintenance time Less frustration	Opinion/complaints[1] Money
Intern	Knowledge and experience Project success Internship success	Knowledge and experience Performance improvements Cheap employee[1]

Table 2.2: Stakeholders with their interests and contributions

2.8 Assumptions

During this project I assume the CMS' processes are deterministic and not subject to arbitrariness[2]. Any arbitrariness in output after (sound) optimisations I consider a pre-existing bug and is, after reporting a bug, declared backwards compatible.

2.9 Deliverables

Considering the previous sections, the following deliverables will be the result of this project:

- a performance-wise optimised content management system which is backwards compatible;
- and a thesis describing
  - applied theories and practices;
  - validity;
  - recommendations;
  - a conclusion;
  - and an appendix of causes, solutions and proof per optimisation.

[2]Except when arbitrariness is called for, like in functions that are expected to return a (pseudo)random integer.

Chapter 3

Management

3.1 Quality assurance

To ensure the project is kept within scope and adheres to its conditions:

- I meet with my company supervisor on a weekly basis;
- I meet with my school supervisor on a monthly basis;
- I evaluate made optimisations with my company supervisor;
- I work with the version control system (VCS) Subversion (SVN);
- I work in a separate branch, which was branched from the trunk;
- and I merge the trunk with my performance branch on a frequent basis to keep from straying too far from the main development branch.

Additionally, I have advised to set up Trac, which is a web-based software project manager that manages tickets[1], keeps an integrated wiki and offers tight integration with the version control system. I manage all optimisations and their progress and result in this system. This keeps me organised and "records" my work at the same time.

3.2 Risks

There are risks involved with every project. To stay ahead of these risks, a summary can be found in table 3.1.

3.3 Project timeline

The project timeline can be found in table 3.2.

[1] A ticket can be any task, including a bug or an optimisation.

Risk and measure	**Probability**	**Impact**
Reprovinci terminates the project early. *No measures can be taken to prevent such an event, except for delivering good work.*	Very small	Very high
Prolonged illness delays the project. *No measures can be taken to prevent such an event. Except for eating broccoli, maybe.*	Very small	Very high
The thesis is not completed within time. *Weekly meets with my company supervisor. Monthly meets with my school supervisor.*	Medium	Very high
Proposed performance improvement cannot be achieved. *Achieving a maximum response time of 0.5 seconds for 100% of the requests in a load-free environment is virtually impossible. In case this occurs, an explanation must be provided in the conclusion.*	High	Low

Table 3.1: Table of risks and their probabilities and impacts

	Date
Phase I: Orientation	Tue Feb 1
Finish orientation assignment	Fri Feb 11
Phase II: Optimisation	Mon Feb 14
Hand in project plan	Fri Mar 11
Phase III: Performance and advisory report	Mon May 9
Phase IV: Thesis	Mon May 16
Hand in thesis	Tue May 31
End of internship	Thu Jun 30

Table 3.2: Project timeline

Chapter 4

Current situation

In this chapter we look more closely into how the CMS came into being and how it works, both functionally and technically.

As told before, the CMS was developed both out of stubbornness and out of want to keep full control of its features. By developing your own system you can fully customise it to your needs. One must still ensure the system is stable and performs on a satisfactory level. Reprovinci could also have chosen other methods to offer their clients unique features. By extending an existing CMS through plugins, you can rely on a stable, evolving codebase. However, depending on their plugin system, not all customisations may be possible. More profound changes in functionality can be achieved by forking[1] a project and adding or changing its features. Then again, keeping up with the original project's changes may be a time-consuming and difficult process. All approaches have their merits, but the fact remains Reprovinci have developed their own system.

4.1 Functionalities

The CMS is built around several concepts, these include:

- the structure;
- content in the form of user defined objects (UDOs) and user defined forms (UDFs);
- e-commerce (pricing, stock management);
- mailing;
- templates;
- users and user rights;
- and keeping UDO content safe through versioning.

All customer websites are built from templates, as is the administration area itself. Generally speaking, these templates suffice for most websites. If this is not the case, there's always the option of creating a custom content handler, which can implement other business logic.

Virtually every page contains one or more of the previously mentioned UDOs. These UDOs can range from a news message to a shawl. They can be related (like a travel and a destination are related), they can have a special Christmas price and/or only run during a specific period (the life cycle). UDFs, on the other hand, can be created directly from the CMS' administration area.

[1] Forking is both the act of inflicting damage upon someone with a fork and the act of (legally) copying a project's source code and starting development on one's own.

Both UDOs and UDFs can be inserted into the website structure. This structure is made up out of nodes, which form a tree structure. Each node can have one or more assorted content items and each content item can be attached to multiple nodes.

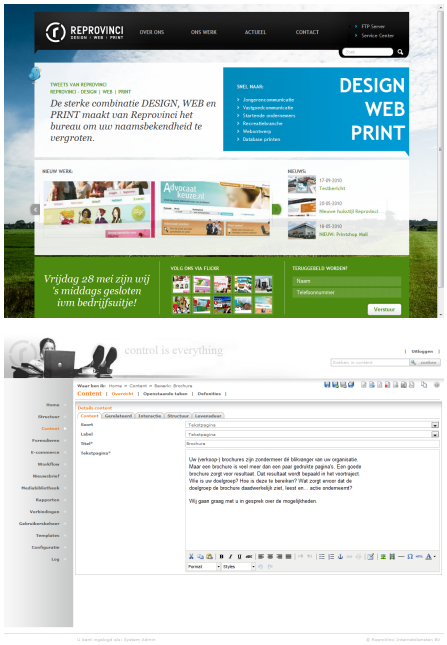


Figure 4.1: Reprovinci's website and its administration area

4.2 Software

The CMS has been built on PHP. PHP is a widely adopted interpreted language, which many consider slow, just because it's an interpreted language. Also, the platform is predominantly used by beginner programmers due to its simplicity. The truth is that PHP is quite light-weight and a hassle-free language, which can, let me stress, *in the right hands*, produce professional, well performing web applications that scale well, both horizontally[2] and vertically[3]. If you're not convinced, remember that Facebook was, until recently, a fully PHP-based web application. [10]

As its backbone the CMS uses the Zend Framework, a collection of mainly helper classes you may use – thus essentially more a library than a framework – that enable a developer to more quickly and easily develop the typical MVC based application.

4.2.1 Architecture

As I said the CMS employs the MVC-pattern, which separates data, presentation and logic, allowing for separate development and loose coupling[4]. Known as *routing*, Zend determines which controller to use from the URL. At the heart of the model layer is the `ObjectCollection` class, which is able to load UDOs, based on an array of configuration items, such as a navigation node or a search query. This construction is shown in simplified form in figure 4.2.

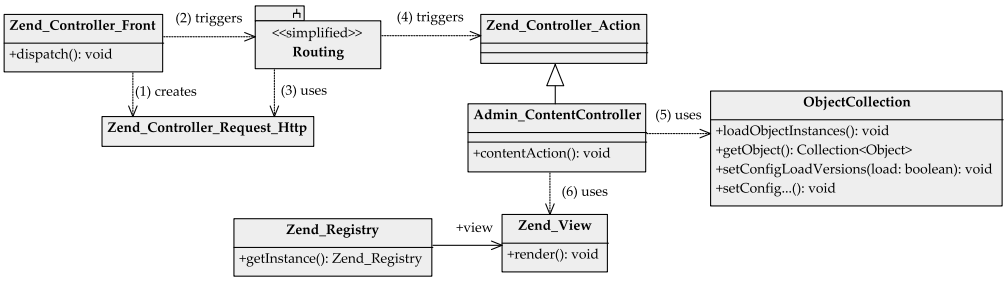


Figure 4.2: Zend's MVC pattern [18] as used by the CMS, displayed in simplified form.

[2]Scaling horizontally, or scaling out, means adding more nodes to a system. A web application's load can, for example, be distributed among more than one server. [8]

[3]Scaling vertically, or scaling up, means adding resources to a single node in a system. For example, more RAM can be added to a server in case of memory shortage. [8]

[4]In a loosely coupled system, separate components know little about other components, other than their interface (how to address them).

IMPORTANT UDOs are defined through an eXtensible Markup Language (XML) file, which is parsed by `ObjectCollection`. These blueprints are loaded into `Objects`. UDO data is *also* kept in `Objects`, but in its `VersionCollection` attribute `Object#versions`. These data are stored in the database as follows:

- `Object` stores references to UDO blueprints.
- `ObjectInstance` stores UDO instances and keeps general information like creation date and owner.
- `UDO_*` tables store UDO-specific data, such as a title and a body for a news UDO.

So as to prevent confusion, the definitions will be referred to as UDO definitions or object definitions, and the actual instances will be called UDOs, objects or content items.

To make sure the right people access the right data, the CMS implements a role-based access control (RBAC) system. This system assigns users certain roles, which in turn have certain permissions. These permissions can be assigned to virtually anything through labels. The most common use is specifying rights for UDOs.

4.3 Performance problems

The CMS currently experiences many performance problems, especially in the backend. The content list, a set of pages that display all of the underlying website's content in a list view, can become very slow (in the order of multiple seconds) as the amount of object definitions and objects grow. The content detail page, a page where one can edit a content item, is also quite slow. A change in the website's structure causes a recompilation and recaching of the navigational structure, which can take anywhere from a couple of seconds to four minutes for the largest website.

Chapter 5

Performance

Abstract

This chapter addresses the work I have done during my internship in a didactic way, describing the theory and applying it to an exemplary optimisation. It starts with explaining what performance is; how to analyse and measure performance and then tries to address various causes that underlay the CMS' performance issues and my solution to them.

To the reader

My school supervisor has urged me on many occasions to split this chapter into separate chapters, describing first theory and then optimisations I have made, to simplify finding out what I have actually done. This thesis severs both as a graduation text and as a transfer of knowledge and recommendations to Reprovinci. I also feel that the theory and the practice are to much interrelated to separate and have therefore not separated them.

5.1 What *is* performance?

When referring to computers, performance in general can be described as the amount of useful work accomplished by a computer system in relation to the time and resources used. [1] This relation can be characterised by one or more of the following quantifiers:

- short response times: the amount of time in which a resource such as an HTML-document is loaded;
- high throughput: the amount of resources that can be processed within a certain time frame;
- low utilisation of computer resources such as the central processing unit (CPU) and memory;
- high availability: a term often used in systems administration meaning the amount of time during which a user is able to use a system;
- fast or good data compression and/or decompression;
- and high bandwidth. [1]

These characteristics easily influence each other. For example, a computer system usually resorts to paging[1]during high utilisation of computer memory. As disk reads (and writes) are relatively slow [2, 3], the response time, for example, decreases.

Thus far we have discussed something called computational performance. But performance is not just numbers; the user plays a role as well. A system may have a low computational performance, yet be made to appear fast through the use of elements such as splash screens or progress bars [4]. This is called perceived performance.

An example of low perceived performance is the UDO detail page. This page uses, after the HTML-document's finishes loading, three synchronous AJAX[2] requests to load related information, blocking the user interface. To the user, the page is unusable and has thus "not finished loading", even though the page is usable without the related information.

Reprovinci's CMS has high response times, low throughput, high utilisation of computer resources and it uses no compression. Its computational performance is therefore low. There are no elements increasing its perceived performance. Its perceived performance is therefore low as well.

Not all of these characteristics apply to Reprovinci's performance problems, or are not for me to fix. Compressing and decompressing data isn't part of the CMS' day-to-day business. High bandwidth has not proven to be an issue and is hard to test locally. Lastly, high availability is more of a system administration issue.

These quantifiers will shortly be described in more detail. But before that, it is important to understand what approaches can be taken to analyse a system's performance.

5.2 Performance analysis

Performance can be analysed in a number of ways. In his thesis, Chiew describes [5, pp.31-32] three performance analysis techniques.

The first is *analytical modelling*. This technique is based on a mathematical model of the system to be tested. Based on the main aspects of the system, its accuracy is the lowest of all analysis techniques, as it cannot take details into account. The response time of *large.local*'s product page can be described as the model shown in figure 5.1.

Assuming that:
R is the time needed for routing;
a indicates whether the user is anonymous (0) or not (1);
d_{hit} is the access time for a cache item;
d_{miss} is the penalty imposed in case of a cache miss;
D is the average cache hit ratio;
p is the amount of products per page;

a simple analytical model would look like:

$$T_{products} = R + Dd_{hit} + a(p-1)Dd_{hit} + p(1-D)d_{miss}$$

Figure 5.1: An example analytical model of *large.local*'s product page. Anonymous users may see a cached page, while non-anonymous users may see p cached products.

By modifying the parameters of this model, a quick prediction of the performance of

[1]When a process accesses a certain memory location, paging transparently maps this location to a certain section (a page) in memory. This allows for non-contiguous memory allocation, but also for storing pages on secondary storage, such as hard disks. The latter is the functionality generally associated with paging.

[2]Asynchronous JavaScript and XML: a method of requesting resources from JavaScript.

a system can be made.

The second technique is called *simulation*. A simulation model can be created both using simulation software – usually tailored to a specific area, such as electronics or biology – and a programming platform. This technique allows for the construction of a more detailed model than using an analytical model, at the cost of time. The accuracy of its results are still limited.

The last technique is simply *measurements of existing systems*. By measuring the metrics of an existing system, you get the most accurate results. The downside to this technique is that, for the best results, one must measure a system in a production environment. This usually means that the system be quarantined to remove influences from other users. Additionally, it is a relatively inflexible technique, as changing of the parameters or the behaviour of the system in a production environment are difficult to realise.

Fortunately, Reprovinci already have an existing system, and because I decided to measure the system locally (as described in chapter 2.4) and because PHP code doesn't have to be recompiled and redeployed after modification, changing parameters and system behaviour can be done easily.

5.3 Metrics

As determined near the beginning of this chapter, the following characteristics remain:

- response time;
- resource utilisation;
- and throughput.

5.3.1 Response times

Response times are, when using measurements of existing systems, easily measured by requesting pages from the server the application resides on and measuring the response time. Tools, simple and more complex, are readily available for these measurements.

But what is response time? A Hypertext Transfer Protocol (HTTP) request is comprised of multiple phases: connection, request, response and rendering, as is shown in figure 5.2. A response time can consist out of a number of these phases. I will describe

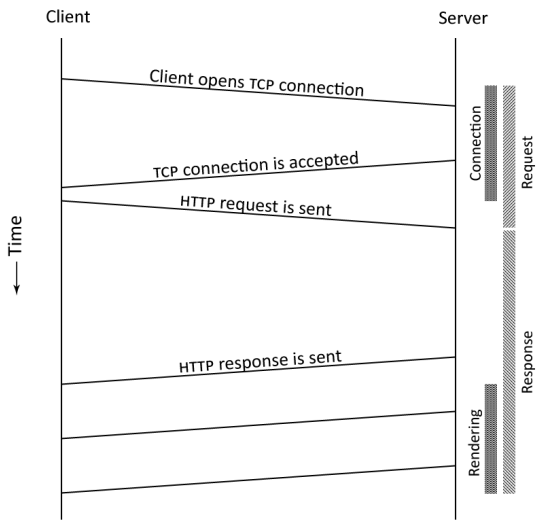


Figure 5.2: The multiple phases of an HTTP request.

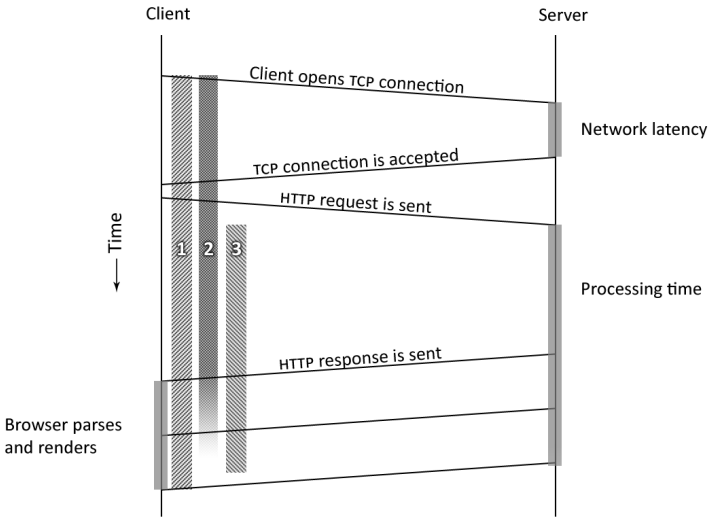


Figure 5.3: The three approaches to response time.

three approaches, illustrated in figure 5.3, having considered them during the optimisation project. The first represents the time spent from the moment the users clicks a hyperlink, for example, until the browser has finished rendering the resource. This approach considers the entire process to be the response time. The second one is the response time as it is typically experienced from the user's perspective: the user clicks on a link and is then, when the first content starts to appear, able to interact with the content. When the user is able to interact may depend on content, download speeds and user experience. The third and last interpretation concerns only the time it takes the server to construct the response based on the request.

Each of these interpretations is a valid approach to measuring the response time. In Reprovinci's CMS optimisation project we have chosen to measure the latter interpretation. Measuring the first approach has to be performed from a browser. There are extensions available for several browsers, such as Google's Chrome[3]. Unfortunately, these tools are simplistic and do not offer many options for customising the benchmarking process. Also, the measurements can vary greatly with browser, hardware and even geographic location. This makes it harder to predict how an application performs. The second approach represents the actual user's experience of performance. It is, however, even more variable than the first approach. Its measurements are strongly subject to user experience, are also time consuming to make and requires the commitment of multiple people, not lending itself for repeated testing. Choosing such an approach results in less repeatable and less reliable results, as people respond differently from time to time.

Response times can differ from request to request. They are dependent on environmental factors, such as processor and disk loads), but also on system state, user variability and cache availability. The web server may have just started up, requiring more work before the first request can be served. Different users may have different rights, requiring more or less content to be served. Cache misses also contribute to higher response times. So which combination is representative?

One's first thought may be to assume a worst case scenario: the web server has just started, the user has many rights and the cache is empty. This way, the performance can only turn out better than expected. But this type of request happens only now and then and is probably not representative of the performance of the web application.

Repeatedly testing the same page results in a higher query cache hit rate than in a real-world environment. It is very realistic that some query results may not be present in the query cache during most of the requests, as queries vary from application to application and from user to user. At Reprovinci we have therefore chosen to clear the query cache of the database server at every request during the research phase. The application

[3]Chromium benchmark extension, `http://goo.gl/SuVka`

cache, however, is not cleared, as it contains frequently used and infrequently replaced cache entries, such as the web application's object blueprints and navigational structure. During the test phase, the application cache and query cache is cleared once, after which an extensive test sequence is performed with all caches enabled, forming a representative image of the applications true performance.

This section has mainly addressed response time based on the server's perspective. Other ways of response time determination are *active probing*, which is the periodic measurement of response time by geographically distributed *agents*, and the approximation of response time by *server log analysis* [5, pp.59-61]. An advantage of these approaches is that the response times of the production servers are measured. Server logs are often analysed due to their availability.

5.3.2 Resource utilisation

A highly stress resistant web application can handle large amounts of users concurrently, while maintaining an acceptable response time. Stress resistance is primarily related to resource utilisation. Major influencers are computational performance (CPU), usage of primary memory (RAM) and the amount of reads and writes to secondary storage (e.g. hard disk). If any of these come short, response times will increase drastically [TODO: prove using Couchmark].

On UNIX systems, the load average is a good indication of the stress the CPU is under. It tells how many processes are using the CPU or waiting for the CPU. A load of 1.7 on a single-core CPU indicates the CPU should be 70% faster to handle the entire workload. On a multi-core CPU, this load is acceptable and can be handled by the CPU. The lower the load, the better an application performs. The Windows platform appears not to have such an indicator. In its task manager, the percentage of CPU utilisation (the inverse of idleness) is shown. These two indicators can best be compared using the analogy of a bank:

> *"The Windows value would be equivalent to the proportion of time that a teller spends serving customers. The* UNIX *value would be the average number of customers waiting in the queue."* – Philip Clark

While the CPU utilisation is very intuitive and easily understandable by laymen, it becomes less meaningful as soon as the number reaches 100: the CPU is probably overloaded, but by how much?

Luckily, Windows offers the Performance Monitor, which offers a large range of metrics, concerning pretty much all of the system's components, including a metric of the processor queue length, indicating how many threads are waiting for the CPU. This metric is very similar to, but not the same as, UNIX' load average. Microsoft state that, for busy systems, "the queue length should range from one to three threads per processor."[6]

Stress resistance is not directly related to Reprovinci's optimisation project, as the goal is to reduce all response times to 500 milliseconds or less. Still, I intend to keep the load on the web server at a same level as before or at a lower level than before. Serving pages within 100 milliseconds, yet completely hogging the server is not an option.

5.3.3 Throughput

Throughput is defined as the amount of resources that can be processed within a certain time frame. This metric is dependent on virtually all performance characteristics. The one influencing this number the most is stress resistance. Even though response times are sky high (say, a minute) and if the server can handle the concurrency of, say, 200 users, while sustaining the same response time it can still put through 200 requests per minute.

5.4 Performance before optimisation

Now that we have a good idea what quantifiers may be used to portray the performance of the CMS, let's give these quantifiers a value. The performance of the CMS per use case as it was before I started my internship can be found in figure 5.4. This figure only shows the performance of the content list. More measurements can be found in appendix chapter C.

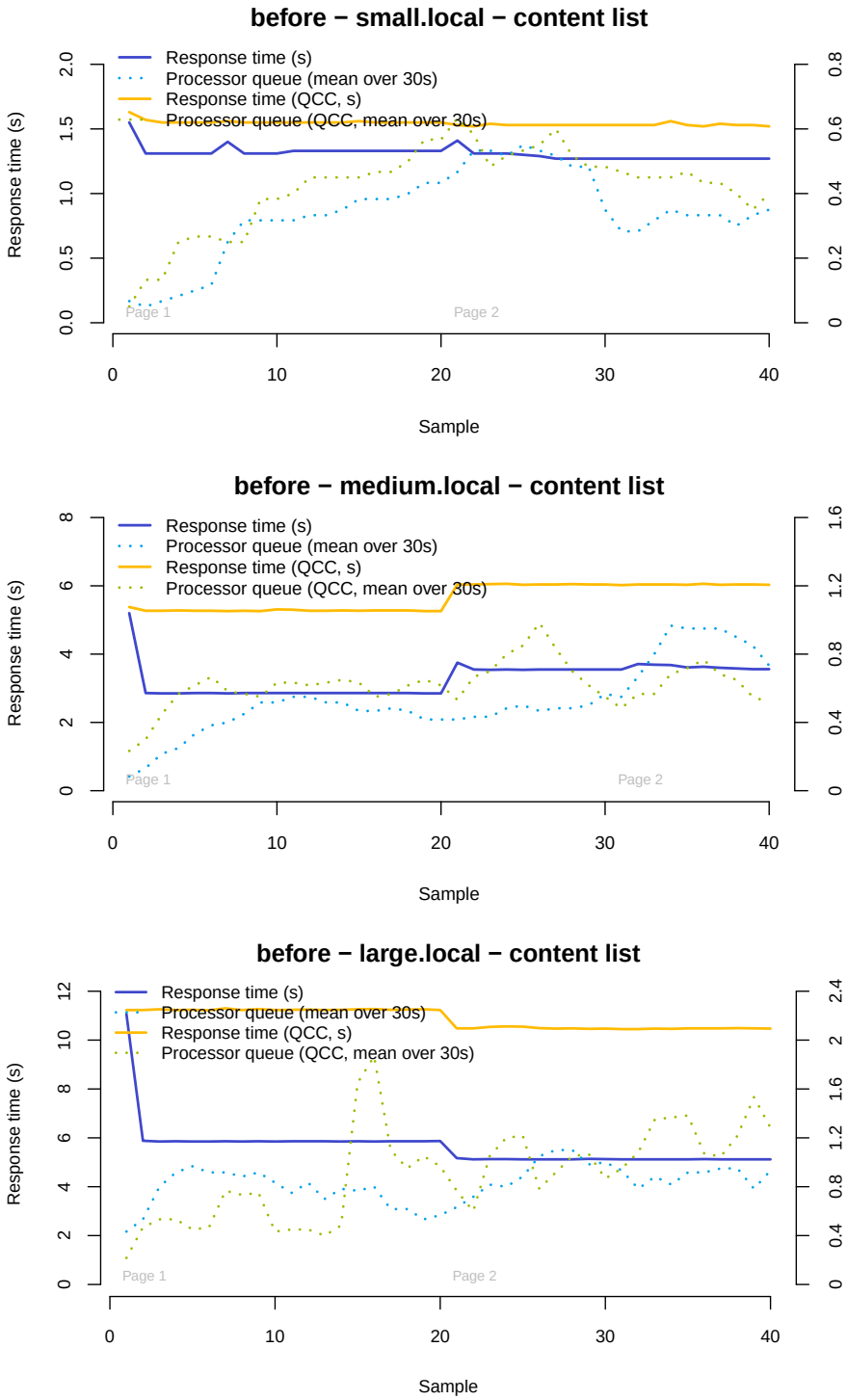


Figure 5.4: The content list's initial performance across all three websites. Thirty samples for each page.
QCC = query cache clear

5.5 When to optimise

Now that you've determined your application is really quite slow, you feel like you should optimise your application. But should you? One of the many pitfalls beginning programmers fall into is premature optimisation. While one should always work *dryly*[4] and keep an eye out for obvious slowdowns, one should typically not optimise before the application (or dependency-less module, for that matter) has been realised as it was designed. One may come across unforeseen design problems in the future with so-called code or design optimisations one has implemented in the past. Only when the application has been realised as designed can one solve performance problems correctly.

The second issue to address is that, once performance problems have been solved on the design level, optimisations on other levels are needed to increase the performance of the application.

One such a level is the optimisation of the source code. These optimisations easily reduce legibility and maintainability. This raises the question whether an optimisation is justified. An example: instead of regular PHP arrays, the CMS uses the class `Collection` (see figure 5.5), which provides an object oriented interface to arrays and essentially transparently allow arrays to be passed around by reference[5], instead of by value[6]. It also offers the feature of lazy loading by checking if the collection has been loaded and whether there's an onload function available. If so, it executes this method before accessing the member array. This functionality was implemented through the method `Collection#_checkCallback()`. As collections are used throughout the CMS, this method was called up to 33,000 times per request. For each call, PHP performed a context switch[7] which reduced performance significantly. By inlining this method[8] wherever `_checkCallback()` was called, performance was increased by almost 10%, but it also decreased maintainability, as the same block of code was present in 10 different places. Because it is a small code block which is repeated only in one class, `Collection`, I determined that this change was justified.

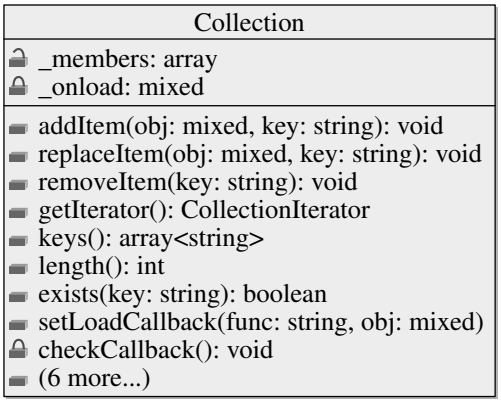


Figure 5.5: The class `Collection`.

Other levels of optimisation include the compile, assembly and run-time level. All of these are hard to apply to high-level programming languages[9] such as PHP and Java without changing their compiler or interpreter.

5.6 Methodology

Through the optimisation process, I have exercised a simple methodology. This methodology comprised seven phases:

1. Determine a page is slow.

2. Determine the cause.

[4]Don't Repeat Yourself (DRY) is a principle that states that "every piece of knowledge must have a single, unambiguous, authoritative representation within a system" [7]. To work dryly is to conform to this principle.

[5]Pass by reference: a reference to the value is passed when leaving a context, such as a method body. Changes made to the value are reflected throughout other contexts.

[6]Pass by value: the value is copied when leaving a context, such as a method body.

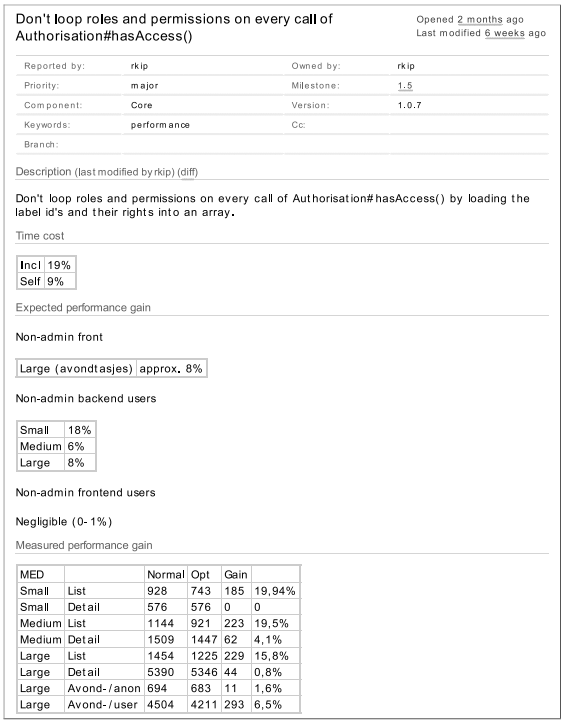
[7]Passing around parameters and return values.

[8]More about this optimisation can be found in appendix chapter A.1.

[9]High-level programming languages are languages which generally abstract computer-specific details, hide memory management and/or are easier to use than lower-level programming languages.

3. Draw up a solution.
4. Estimate how much performance can be gained (using a quick fix, for instance).
5. Implement the solution if the estimated gain is more than 5
6. Perform comparative measurements.
7. Evaluation: compare the estimated and measured performance gain.

All resulting information is kept in a Trac ticket (see chapter 3.1). This way, the entire process is kept with the actual changes in the SVN repository. In the following sections, I describe the practices I applied during the optimisation process.



Don't loop roles and permissions on every call of Authorisation#hasAccess()

Opened 2 months ago
Last modified 6 weeks ago

Reported by:	rkip	Owned by:	rkip
Priority:	major	Milestone:	1.5
Component:	Core	Version:	1.0.7
Keywords:	performance	Cc:	
Branch:			

Description (last modified by rkip) (diff)

Don't loop roles and permissions on every call of Authorisation#hasAccess() by loading the label id's and their rights into an array.

Time cost

Incl	19%
Self	9%

Expected performance gain

Non-admin front

Large (avondtasjes)	approx. 8%

Non-admin backend users

Small	18%
Medium	6%
Large	8%

Non-admin frontend users

Negligible (0-1%)

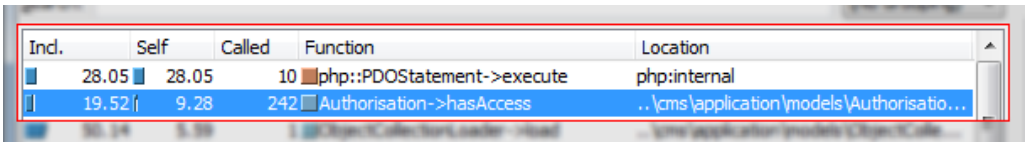
Measured performance gain

MED		Normal	Opt	Gain	
Small	List	928	743	185	19,94%
Small	Detail	576	576	0	0
Medium	List	1144	921	223	19,5%
Medium	Detail	1509	1447	62	4,1%
Large	List	1454	1225	229	15,8%
Large	Detail	5390	5346	44	0,8%
Large	Avond-/anon	694	683	11	1,6%
Large	Avond-/user	4504	4211	293	6,5%

Figure 5.6: A ticket in Trac.

5.7 Finding bottlenecks

So, we've determined our application is slow and that we actually want to optimise. The next step is to determine *why* an application performs poorly. To determine this, one may want to look into the software's execution path. Profilers record how many times a method or function is called and how long it takes to execute. Figure 5.7 illustrates this.



Incl.	Self	Called	Function	Location
28.05	28.05	10	php::PDOStatement->execute	php:internal
19.52	9.28	242	Authorisation->hasAccess	..\cms\application\models\Authorisatio...

Figure 5.7: KCacheGrind shows `Authorisation->hasAccess`'s profiling information

The table shows the cumulative time the method itself takes to execute (*Self*), the cumulative time the method takes *including* the execution times of method it itself calls added up (*Incl.*) and how many times it was called (*Called*). From this table we can

infer that the application spends approximately 28% of the request in the database[10]. Additionally, there is the method `Authorisation >hasAccess()` that is called 242 times, takes 9% to execute and calls other methods, which add 11% to its total execution time. This method is interesting, as it performs a seemingly simple, almost atomic task of checking whether a user has access to a certain UDO.

I have found that methods that are suboptimal often share one or more of these profiling characteristics:

- The method's *Self* value is equal to or more than 10%. This probably means the method is overly complicated or too long, or contains repeated code.
- The method is supposed to do something simple or an almost atomic task, yet its *Incl.* value is equal to or more than 10%.
- The method is called a lot. This may indicate redundant calls. A lot is a very relative term and it *very much* depends on the functionality of the method.

A lot of performance issues should become apparent by following these guidelines. But remember, always use your common sense; if something looks suspicious, you should investigate further.

5.8 Application design and its influence on performance

Very early in the project, `ObjectCollection` turned out to be a major culprit in the CMS' performance issues. A request for the content list (see chapter 4.1) at *medium.local* took [3.0s/5.5s][†], of which, after a query cache clear, fifty percent of the time was spent in `ObjectCollection#loadObjectInstances()`. Fifty percent doesn't have to be a problem, but in this case, that's about 2.8 seconds for just 50 objects. On inspection, it turned out that `ObjectCollection` had, through time, become the fat, omnipotent king of the model layer. Whenever you needed UDOs, you went to him. It was responsible for:

- holding `Objects` (it is a `Collection`, after all);
- loading `Objects`;
- the configuration of which `Objects` to load;
- keeping the `Object` blueprints and the caching thereof;
- and other minor tasks.

If separation of concerns (SoC)[11] were a law, the king would be sentenced to beheading.

When designing a feature, the designer should always keep SoC in classes in mind. Additionally, SoC stimulates object composition[12]. This results in smaller, loosely coupled classes and methods, which makes them the more understandable and thus maintainable. The inverse is also true: if one does *not* separate concerns, it typically results in large, tightly coupled classes and methods, which – you guessed it – are hard to understand and maintain. When the latter is the case, the class is prone to clone-and-modify programming, as abstraction possibilities become less obvious and more time-consuming. In the end, it consisted of an array of if-elses, which, query by query, narrowed down the UDOs to load.

Before optimising the loading of objects, I decided to:

[10] PDO, PHP Data Objects, is a generic software interface for accessing different databases, such as MySQL and Oracle.

[†] $[x/y]$: response time (x) versus response time with query cache cleared (y).

[11] Separation of concerns is a paradigm that, in summary, states code should typically be responsible for one task only.

[12] Object composition is the act of combining simple objects into a more complex structure to perform a complex task.

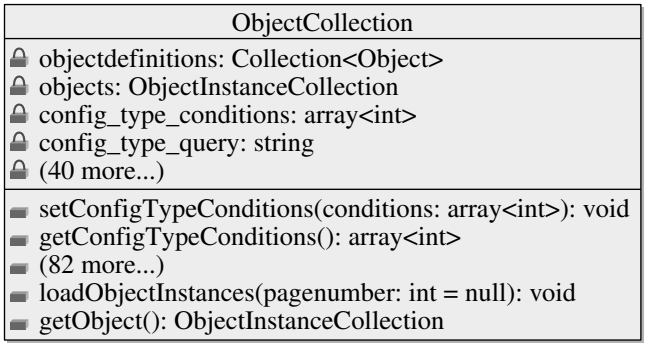


Figure 5.8: An excerpt from `ObjectCollection`

- separate it from `ObjectCollection` into `ObjectLoader#load()`;
- let `ObjectLoader#load()` return an `ObjectLoaderResult`;
- and let `ObjectCollection` implement `ObjectLoaderConfiguration`.

`ObjectCollection`'s interface is still backwards compatible, but the loading implementation has been separated and expects an implementation of `ObjectLoaderConfiguration`. Further separation of concerns is difficult without breaking compatibility. In the future, further separation of concerns should be strived for. The results can be seen in figure 5.9.

In summary, bad design leads to large blocks of code, which are difficult to understand and maintain. Inversely, good design generally leads to a more understandable and more maintainable codebase.

5.9 Query optimisation

NOTE *This chapter addresses query performance issues as they appeared using the MySQL database management system (DBMS) and the MyISAM and InnoDB storage engines. While chances are the same issues and optimisations apply to other DBMSs, each DBMS uses different storage engines and employs different (internal) optimisation techniques.*

The king of the model layer has been on a diet, but it's still performing badly. The evildoers here are the queries performed. As I explained before, `ObjectCollection#loadObjectInstances()` operates by filtering objects query by query. Depending on its configuration, it may filter on:

- rights;
- navigation node;
- whether an object is published or not;
- type(UDO);
- life cycle;
- and more.

After that, sorting is applied. The resulting array with sorted object ids is used for pagination and to load a certain amount of objects. Afterwards, additional data like prices, items in stock and related objects and files are loaded and attached to their objects.

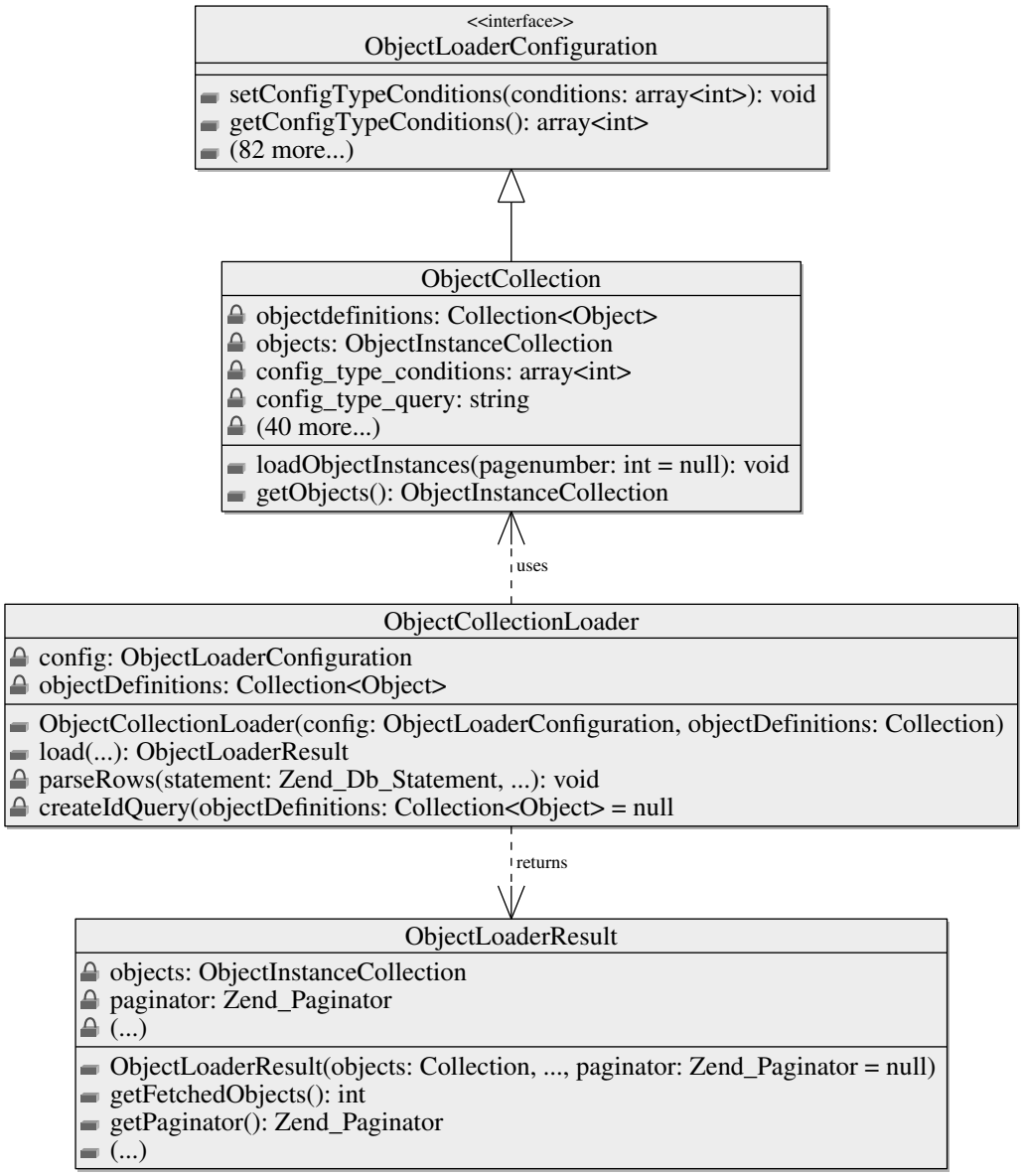


Figure 5.9: Situation after refactoring of `ObjectCollection`. The ellipses replace details that would only confuse. A more detailed description of this project can be viewed in appendix chapter A.2.

Unnecessary data

It goes without saying that you shouldn't query for data you aren't actually going to use. To some extent, this is inevitable and something you don't have to fix, but this wasn't the case with the administration.

Each object can have up to three types of versions: a published version, a version that is *in progress* (ie. being edited, may soon be published) and history items. The published versions had the flag `published` set to 1 and the in-progress version was the version with a later creation date than the published version. For the content list and detail page in the administration the in-progress version is used when available. Otherwise, the published version is shown.

`ObjectCollection`'s loading procedure had two options for versioning: loading published versions or just all versions. Thus, all versions were loaded. To determine whether there was an in-progress version available and loading the published version otherwise would query-wise be difficult and very expensive, as object versioning takes place on the UDO table level. To counteract this, I concluded we needed to query only the in-progress versions and fall back to a published version if necessary. This would need extra joins to be made per UDO table. For *medium.local*, this would have meant extra joins for twenty tables. This resulted in the conclusion that I had to add an in-progress flag. This way I could query for versions that were published or in-progress. Implementing an in-query in-progress-to-published fallback would also need extra joins, so I decided to query both in-progress and published versions and handle the fallback programmatically.

Multiple queries

Using multiple queries has disadvantages. For each query the query string is sent, the query cache is consulted, the query is performed or the cache read, and the result is sent back; you may be transfering a lot of data for what may eventually be just a short list of objects. Secondly, the database doesn't have many chances to optimise. Lastly, data integrity may be at risk; data may change in between queries, which may result in corrupt data or data the user doesn't actually have access to. It also has some advantages. For example, the query cache hit rate is generally higher with multiple queries; if only one query's cache entry is invalidated due to a mutation, the others still "hit" the cache. To improve the performance of object loading I merged the independent queries into a single query using `JOIN`s and the lot.

As queries get larger and more complex, query time is affected more and more easily. The addition of a `JOIN`, aggregate function or an `ORDER BY` clause can bring a query to a multi-minute crawl. There is never a single solution to a specific performance problem; each query needs to be individually treated.

The execution plan

Of interest to anyone creating complex queries is MySQL's execution plan. This plan can be inspected by prefixing the query with the keyword `EXPLAIN` and has proven itself to be very useful during the optimisation of `ObjectCollection`'s object loading. It can be quite cryptic, but the plan provides a lot of information about the approach MySQL takes to executing the query, such as the join order and use of indexes. For instance, a missing index is very evident, indicated by the join type `ALL` and the key `NULL`.

The profiler

Another feature of interest MySQL's profiler. One time during the optimisation of `ObjectCollection`, I had one large query. I was having problems with sorting. The addition of a `ORDER BY` clause would slow the query down to 8 seconds, or in some cases even 30 seconds. This was unacceptable. The profiler exposed the thread state (states during query processing) *copying to tmp table on disk* as the culprit. This was either due to the data set containing `TEXT` fields, forcing MySQL to writing the temporary table to

disk for sorting [21], or due to a too large a result set [22]. To resolve this issue and, coincidentally, the issue of pagination (the need to know how many objects are available in total without querying all their data), I separated the query into two queries, the id query and the data query. The id query applied all configuration options, such as permissions and sorting as well, returning all UDO ids and names. This query performed significantly better, as the data needed to sort was much smaller than before, fitting into the system's memory.

Indexes

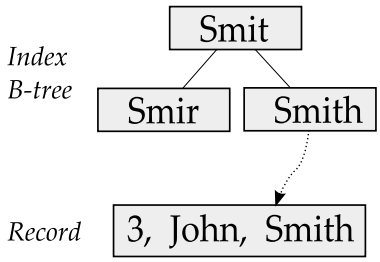


Figure 5.10: A simple B-tree indexing last names. `Smith` refers to record 3.

The first thing to check is whether you have indexes in place (you'll see why later on). The DBMS uses indexes to quickly find rows with specific column values [11]. Without indexes, the DBMS needs to perform a full table scan[13]. Indexes are usually stored in B-trees[14] [11], which allow for quick look-ups of a value (see figure 5.10). The index record in turn refers to the complete record.

Indexes are useful when the cardinality is high, i.e. the values indexed are uncommon. An index on the column `city` which always contains the text `New York` is pretty much useless when querying for `city='New York'`, as the index returns all rows. On the other hand, an index on the column `id` can be very useful, as querying for `id=3` immediately returns the associated row. In general, an index should only be placed on columns used in `JOIN`s and on columns where the index is expected to significantly *limit* the amount of results.

Almost all indexes were in place in the CMS' databases.

Subqueries

Subqueries are commonly used in the CMS. They are easy to read and understand, as they seem to perform the task step by step. However, they are often despised, as they are thought to perform badly (as did I).

The `ObjectLoader` makes sure only those objects are loaded the user has `READ` permission for. The (simplified) database structure is shown in figure 5.11. A common alternative to subqueries is joining. I decided to try a `JOIN` approach and, reluctantly, an approach using a subquery, as shown in listings 5.1 and 5.2.

```
INNER JOIN Permission ON Permission.labelid = ObjectInstanceLabel.labelid
INNER JOIN RolePermission ON RolePermission.permissionid = Permission.id
INNER JOIN Role ON Role.id = RolePermission.roleid
INNER JOIN UserRole ON UserRole.roleid = Role.id
WHERE UserRole.userid = ? AND Permission.rights & 2 = 2
```

Listing 5.1: Permissions are joined

```
WHERE ObjectInstanceLabel.labelid IN (
  SELECT DISTINCT Permission.labelid
  FROM Permission
  INNER JOIN RolePermission ON RolePermission.permissionid = Permission.id
  INNER JOIN Role ON Role.id = RolePermission.roleid
```

[13] A full table scan means all rows are read sequentially.

[14] A B-tree is a form of binary search tree that allows more than two children [12].

```
6     INNER JOIN UserRole ON UserRole.roleid = Role.id
7     WHERE UserRole.userid = ? AND Permission.rights & 2 = 2
8   )
```

Listing 5.2: Permissions are queried in a subquery

The `JOIN` approach clearly outperformed the subquery, finishing in 0.59 seconds, over the subquery's 0.87 seconds. After some further research into how subqueries work and when to use and when *not* to use subqueries, I concluded the `JOIN` approach was the way to go. However, during the writing of this chapter, I stumbled on these two pieces of MySQL documentation [13]:

> (...) for a statement that uses an `IN` subquery, the optimizer rewrites it as a correlated subquery. Consider the following statement that uses an uncorrelated subquery:
>
> `SELECT ... FROM t1 WHERE t1.a IN (SELECT b FROM t2);`
>
> The optimizer rewrites the statement to a *correlated subquery*:
>
> `SELECT ... FROM t1 WHERE EXISTS (SELECT 1 FROM t2 WHERE t2.b = t1.a);`

> `UPDATE t ... WHERE col = (SELECT * FROM (SELECT ... FROM t...) _t);`
>
> Here the result from the subquery in the `FROM` clause is *stored as a temporary table*, so the relevant rows in `t` have already been selected by the time the update to `t` takes place.

By combining these two pieces of information, I concluded that for each object, MySQL performs the permissions subquery. By wrapping these in a `SELECT`, the results of this query would be stored in a temporary table. This temporary table is then queried for each object. After this adjustment, the subquery approach finished in 0.37 seconds, outperforming the `JOIN` approach.

5.10 Partial objects

An approach I often choose during the loading of interrelated objects, is the application of partial objects. In short, a partial object is inserted whenever the data required to fill it is not yet available. It typically only contains an id and a type identifier, which can be the class the object is of.

Partial objects are heavily used in the new `ObjectLoader`. A UDO has several users related to it, among which an "owner". The owner is the person who created the object. It is very common for that person to have created other objects. If one were to load these owners along with the objects in the same query, all related data needed to create a `User`

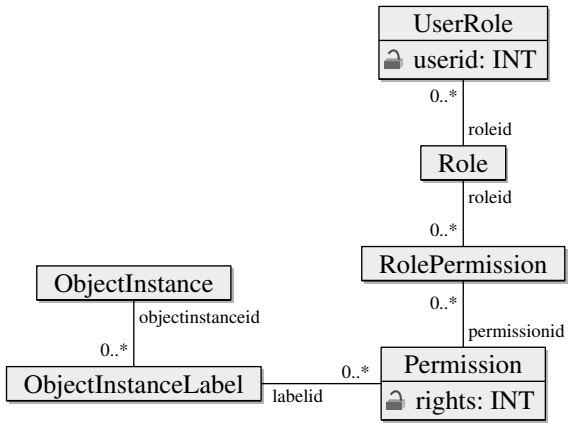
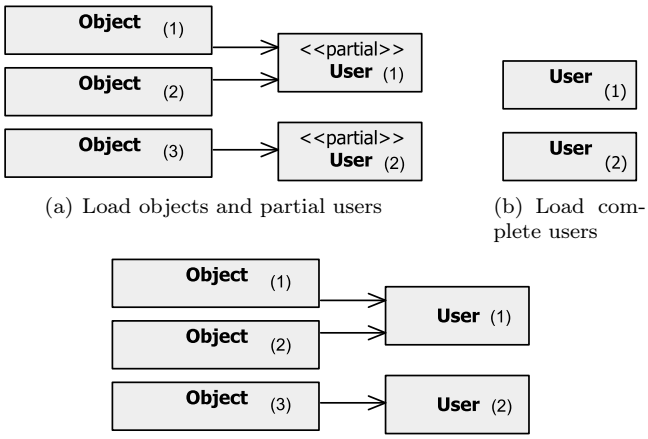


Figure 5.11: Simplified data structure for object instance RBAC.

object would be retrieved multiple times. In this case, it increased the number of rows dramatically, decreasing `ObjectLoader`'s performance. To counteract this, I decided to query only the user ids. I then created a partial `User` object for each new id, stored it centrally and attached it to the object (figure 5.12(a)). This way, the same `User` object is attached to multiple objects. When I loaded the users afterwards , I filled the existing `User` objects (5.12(b)). All objects then had complete `User` objects attached (5.12(c)).



(a) Load objects and partial users

(b) Load complete users

(c) Objects have complete users

Figure 5.12: Partial objects illustrated.

5.11 Benchmarking

After optimising a single feature, you may want to prove that the system's performance has actually improved. To do this, one can employ one of the many benchmarking tools available for this purpose.

5.11.1 ApacheBench

ApacheBench, more commonly known as `ab`, is a command line benchmarking tool that is suitable for simple tests. It can make multiple concurrent requests and report statistics such as error rate, throughput, bytes transferred and mean response times, as can be seen in figure 5.13. However, when devising more complex tests that involve, for example, authentication, `ab` comes short, as configuring cookies or HTTP authentication on the command line is troublesome. For such more complex tests other, more flexible tools can be used.

```
Concurrency Level:      10
Time taken for tests:   7.042 seconds
Complete requests:      100
Failed requests:        0
Write errors:           0
Total transferred:      455400 bytes
HTML transferred:       389100 bytes
Requests per second:    14.20 [\#/sec] (mean)
Time per request:       704.240 [ms] (mean)
Time per request:       70.424 [ms] (mean, across...)
Transfer rate:          63.15 [Kbytes/sec] received
```

Figure 5.13: An excerpt from `ab`'s output.

5.11.2 JMeter

JMeter is an Apache project that allows for complex test plans that are typically created through its node-based user interface.

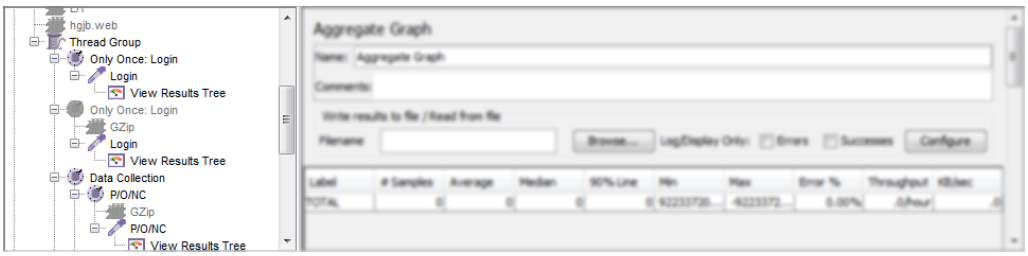


Figure 5.14: JMeter's node based user interface

JMeter's interface realises these test plans using controllers, which allow for repetition, logic, only-once-execution et cetera. The raw sample data, containing response times, bytes transferred and more, can then be saved to several output formats for later review. For a quick insight, the aggregated data can also be displayed in several tabular and graphical formats.

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
P/O/NC	66	768	745	749	191	3802	1.52%	1.3/sec	97.9
TOTAL	66	768	745	749	191	3802	1.52%	1.3/sec	97.9

Figure 5.15: Aggregated test data in JMeter

This has been the most commonly used tool during my endeavour to optimise the CMS, as it allows for both quick and complex tests. Unfortunately, JMeter does not allow for the measurement of custom metrics, such as processor or disk loads or PHP's memory usage. Measuring these metrics separately results in difficult to interpret test results. In the perfect situation, all this data is collected by the same tool. To do this, we can make use of the tool described in the next section.

5.11.3 Couchmark

My motto is: when all else fails, create it yourself (WAEFCIY?). Derived from the word benchmark, Couchmark is a Java tool I have been developing myself that essentially provides a little framework to perform benchmarks, which automagically collects samples. The test writer can create both simple and complex tests in the concise, but powerful Ruby language using simple constructs. Most importantly, the framework can be extended to, for example, allow for the measurement of PHP's memory usage through HTTP headers, which can then be incorporated in the resulting sample data.

```
include Couchmark

class Float
  def round_to(x)
    (self * 10**x).round.to_f / 10**x
  end
end

# Connect to metrics agent, residing on the web server
agent :localhost, 4300

# Defaults
http_defaults :host => 'medium.local', :port => 80

# Collection of user metrics
default_metrics do |response, sample|
  headers = response.headers
  sample.set_metric 'php-memory',
    (headers['X-Metric-Memory-Usage'].to_f()/1024/1024).round_to(2) # MB
  sample.set_metric 'php-memory-peak',
    (headers['X-Metric-Memory-Usage-Peak'].to_f()/1024/1024).round_to(2) # MB
  sample.set_metric 'parse-time',
    headers['X-Metric-Parse-Time'].to_f().round_to(2) # s
  sample.set_metric 'cpu-load',
    get_sample('cpu_load', 30)
end

# One user
users 1 do |user_id|

```

```
31    # Login
32    POST 'login', '/admin/auth/login', {:username => '*****', :password => '***********'}
33
34    # Load the content page fifty times, clearing MySQL's query cache
35    50.times { |i| GET 'content-list', '/admin/content/content', {:MYSQL_CLEAR_CACHE => 1} }
36
37  end
```

Listing 5.3: Example of a simple test plan using Couchmark

Listing 5.3 shows an example Couchmark script. I first specify where the metrics agent is located. This agent is able to monitor cpu load, disk usage, you name it. Secondly, I specify HTTP defaults, such as host and portname. The next step is to specify extra data that is added to every sample taken. In the example, I add PHP's memory usage and parse time, and the web server's cpu load to the sample. Finally, I start one user that logs in and requests the CMS' content list fifty times, clearing the query cache every time.

As you can see, Couchmark is a very flexible, easy to use tool. I believe it has potential and I intend to maintain this project in the future.

Chapter 6

Verification & validation

One of the main conditions to the success of this project was ensuring backwards compatibility as to ensure customer website templates and custom modules don't break. The code base and database has remained mainly backwards compatible.

One thing that has changed, though, is the sorting in `ObjectLoader`, due to a bug in sorting logic. Every object is versioned. The previous object loader, `ObjectCollection`, applied sorting over all versions. Each object can thus appear in different locations. The first instance found is also the location of the object in the sorted list. This is shown in figure 6.1. This bug is less apparent in the optimised version, as this version loads less versions (see chapter A.3).

The other condition was that all pages must be loaded within half a second. To test this, I created a test plan. A test plan often simulates a user as it browses a website. If you're interested in more aspects than just pure response time, you can use a HTTP proxy recorder. Such a recorder sits between the browser and the website (hence, a proxy) to test and record every HTTP transaction, including images, scripts and the lot, that takes place. The benchmarking tool can then replay these actions using high concurrency (simulating one or many users), creating an overall picture of the web application's performance and its impact on the server.

I am only interested in pure response time, while keeping an eye on server stress. To validate we've met our condition, I have created four smaller test plans that measure specific parts of the CMS that were ill-performing. These four areas are:

- the content list;
- the detail page: editing and viewing;
- and viewing pages that list objects, such as *large.local*'s product lists.

I wish to:

- make sure server stress has not increased at the cost of performance increase;
- memory usage has not increased dramatically at the cost of performance increase;
- compare the optimised CMS' performance to the original's;

and to:

- simulate a worst case scenario by clearing the query cache repeatedly;
- simulate a a common scenario by clearing the query cache once, simulating a first visit;
- while measuring:
  - parse time as response time;
  - and memory usage, to make sure it has not increased.

To do this, I have created a test plan in Couchmark. This test plan, listed in listing 6.1 applies for the content list. The test plans for the other three cases are very similar, and can all be viewed in appendix chapter B.

```
include Couchmark

debug false

class Float
  def round_to(x)
    (self * 10**x).round.to_f / 10**x
  end
end

# Connect to metrics agent, residing on the web server
agent :localhost, 4300

# Collection of user metrics
default_metrics do |response, sample|
        headers = response.headers
        sample.set_metric 'php_memory_peak',
                (headers['X-Metric-Memory-Usage-Peak'].to_f()/1024/1024).round_to(2) # MB
        sample.set_metric 'parse_time',
                headers['X-Metric-Parse-Time'].to_f().round_to(2) # s
        sample.set_metric 'cpu_load',
                get_sample('cpu_load', 30)
end


http_defaults :port => 80

for version in ['cms', 'cms-trunk'] do
        for host in ['small.local', 'medium.local', 'large.local'] do

                # Defaults
                http_defaults :host => host

                # One user
                users 1 do |user_id|

                        user_cookie 'CMS_VERSION', version
                        user_cookie 'OVERRIDE_MODE', 'prod'

                        # Clear query cache
                        GET "#{host}-admin-login_#{version}", '/admin/', {:MYSQL_CLEAR_CACHE => 1}

                        # Login
                        POST "#{host}-admin-login_#{version}", '/admin/auth/login', {:username => '******
                              ', :password => '******'}

                        sleep 30

                        # Load two content pages twenty times, no cache
                        for page in (1..2)
                                20.times do
                                        response = GET "#{host}-admin-content-list-nc_#{version}", "/admin/
                                              content/content/viewstate/page/value/#{page}", {:
                                              MYSQL_CLEAR_CACHE => 1, :filter => 'udo'}
                                        if response: response.sample.set_metric 'page', page end
                                end
                        end

                        sleep 30

                        Load two content pages twenty times
                        for page in (1..2)
                                20.times do
                                        response = GET "#{host}-admin-content-list_#{version}", "/admin/
                                              content/content/viewstate/page/value/#{page}", {:filter => '
                                              udo'}
                                        if response: response.sample.set_metric 'page', page end
                                end
                        end

                end

        end
end
```

Listing 6.1: Testplan for the CMS' content list

Using R, a programming language specificly designed for (graphical) statistics, the test plan's result set produces the graphs in figures 6.3 through 6.5:

The content list is the area where the greatest improvements were made, something Reprovinci's customers will be very happy with.

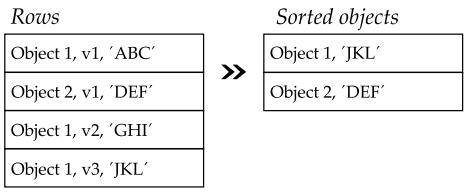


Figure 6.1: Illustration of the sorting bug in `ObjectCollection`

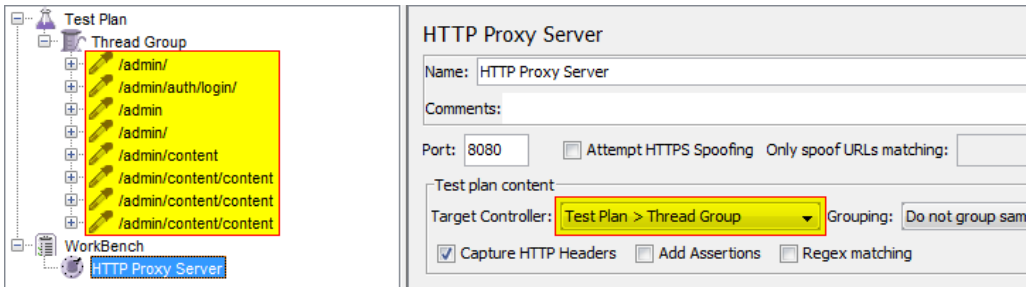


Figure 6.2: A HTTP proxy recorder records HTTP traffic.

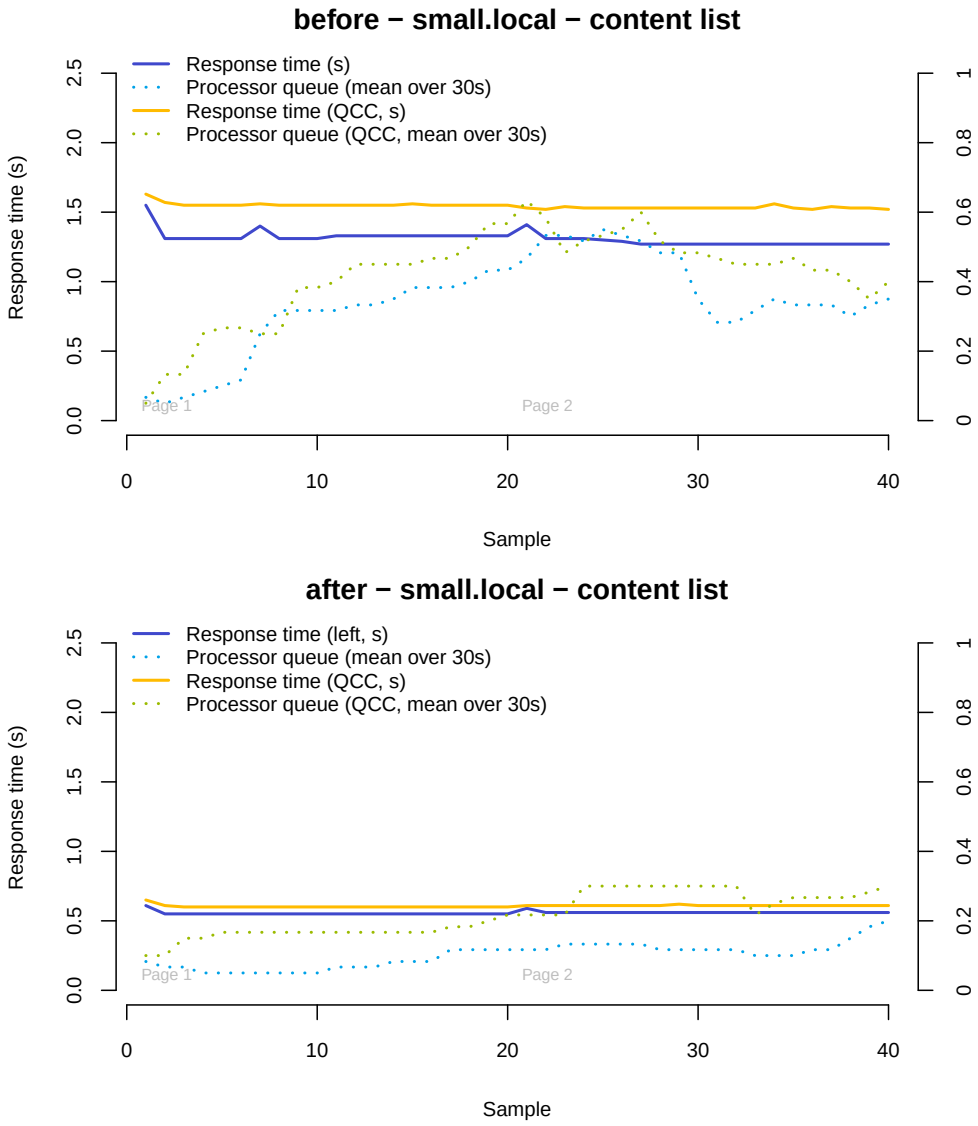


Figure 6.3: A comparison of *small.local*'s content list, before and after optimisation.

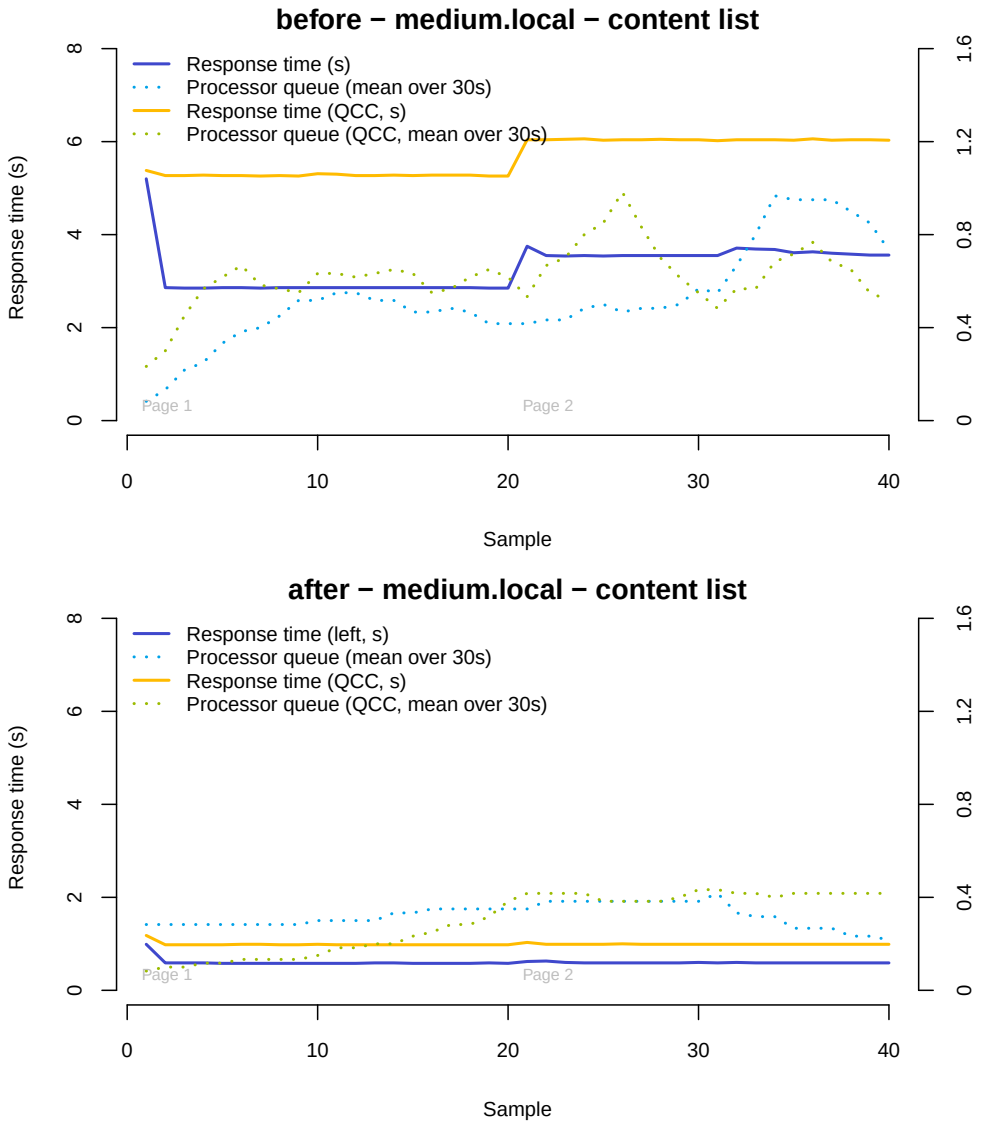


Figure 6.4: A comparison of *medium.local*'s content list, before and after optimisation.

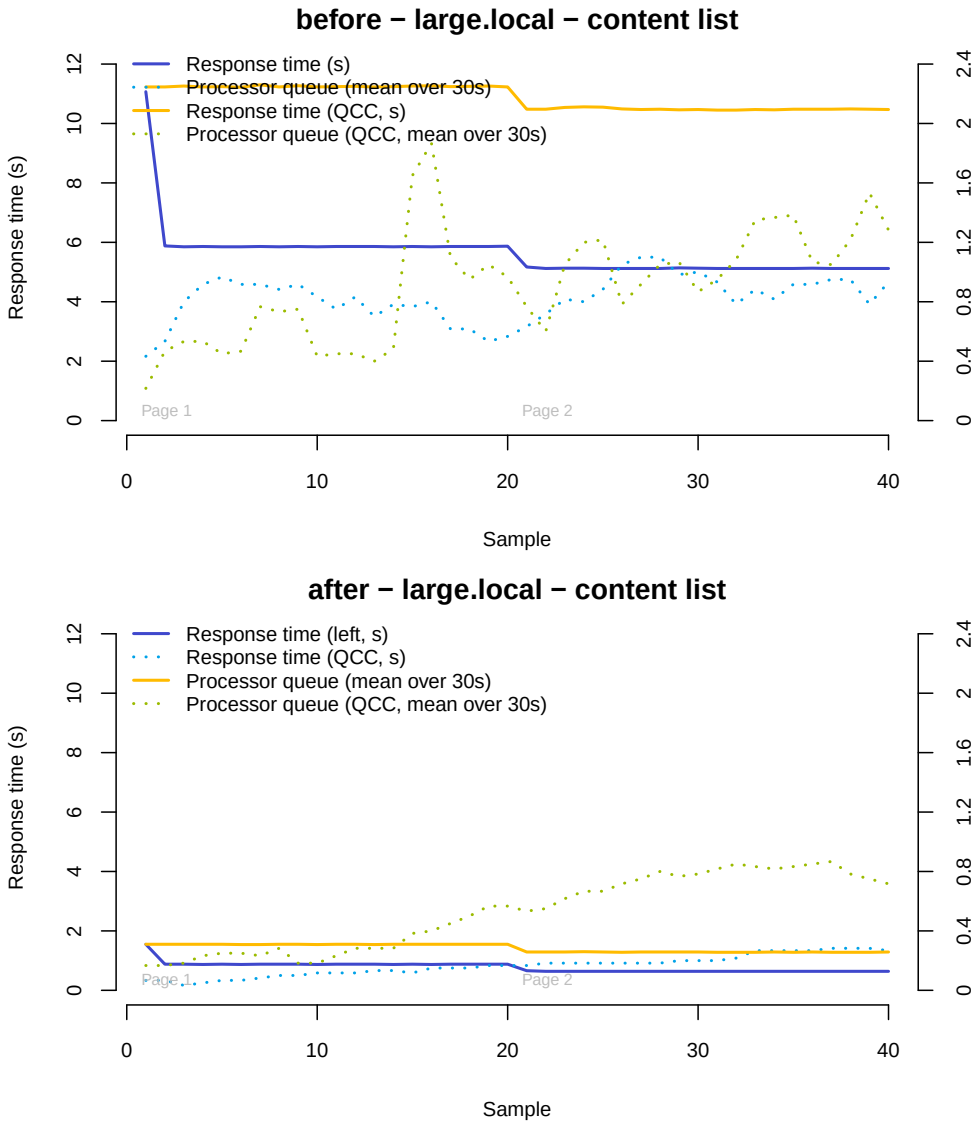


Figure 6.5: A comparison of *large.local*'s content list, before and after optimisation.

Chapter 7

Recommendations

The CMS has already become a lot faster than it was and with some effort, its performance level can be maintained and even raised further.

7.1 Improve quality of code and maintaining performance

Many of the CMS' late performance issues were due to problems with design and algorithm inefficiencies. A couple of guidelines should be followed while developing for the CMS.

- When designing a particular function, strive for separation of concerns and object composition.
- Strive to keep methods small. This improves readability, understandability and thus maintainability. Also, debugging tools like Xdebug will provide more useful information.
- Prevent repeated execution of deterministic methods or code blocks.
- Prevent look-ups through looping, but ensure there's an indexed array for a direct look-up. This concept is similar to placing indexes in DBMSs. If you *must* loop, don't forget to `break` out of the loop when you've found what you're looking for.
- Reuse objects, don't create multiple objects with the same data. This reduces memory usage and makes sure changes to the object in one place are reflected in other places. It also enables for post-loading. One way of implementing this is through the use of a so-called identity map, which contains all objects by id and type.
- Document classes, methods and the actual code. If done correctly, documentation can be generated for all classes and their methods. Code should be self-documenting (i.e. tell a story through correct naming and flow of data using parameters and return data), but comments are often very informative. Take this as a rule of thumb: if you have to think about a piece of code, it probably merits explanation through comments.

To ensure these and other programming guidelines are followed, peer reviewing should be implemented at Reprovinci. Provided it appears to perform its duty, virtually all code is currently committed to the VCS unchecked. This has led to poorly designed systems and poorly written code finding its way into the CMS. There are many forms of code review, but chiefly due to Reprovinci's size, I recommend over-the-shoulder code review, a lightweight method where a developer literally looks over the author's shoulder as he walks through the code [14]. Design and implementation should be open to discussion.

To ensure the same performance level is maintained, performance of new releases should be compared with previous, proven releases.

7.2 Reduce file caching

Currently, the file cache is large and contains many items that can be stored for a long time. Placing these items in a memory cache would need a lot of memory or cause items to be continuously discarded as they make place for other items. Improvements can be made in many of the CMS' compontents, which could reduce the need for caching, making it possible to move to a memory cache, improving performance and reducing disk usage (reducing wear and tear and the risk of disk failure).

7.3 Software

Some of the server software used is outdated. Newer releases of software, such as Apache HTTP Server and MySQL, often perform better and are more secure than previous releases. Reprovinci is a relatively small company and can easily transition to newer software.

For example, PHP can be upgraded from version 5.2 to 5.3 with minor changes. Benchmarks show a possible 5-20% increase in performance. [19, 20]

7.4 Server architecture

Currently, Reprovinci has two production servers. One server is responsible for the largest website, which is a large webshop. The other serves all other websites. The following characteristics apply to this approach:

- **Low reliability** — if one server crashes, all websites residing on that server will be inaccessible for hours or days (reliability through redundancy [15]).
- **Low maintainability** — upgrading software or hardware means downtime for all websites residing on that server.
- **Not stress-resistant** — if one website attracts a lot of visitors, all websites' performance and availability will suffer.
- **Ease of persistence** — data and sessions can be stored locally.

I believe Reprovinci should transition to *load balancing*. With load balancing, all websites reside on all servers. Load is distributed automatically. Scaling horizontally has the following characteristics:

- **High reliability** — if one server crashes, another takes over.
- **High maintainability** — software can be upgraded on one server, while the other takes over.
- **Stress-resistance** — if one website attracts a lot of visitors, load will be spread among all server.
- **Persitence is difficult** — data and sessions could be stored on a single server, but this would again lower redundancy. Implementing load-balancing often requires more than database servers configured in a master-slave configuration.

To ensure a good performance is delivered to Reprovinci customers and to assess the servers' health, I believe metrics should be collected. Such metrics include load; memory usage; availability; and request and query throughput. There are a great many tools available for Linux-based systems.

7.5 Alter storage of user defined objects

Reprovinci's CMS is able to handle user defined objects. The database design is currently highly normalised. Normalisation ensures low data redundancy and high data consistency, as data is kept only in once place. Because of this, each user defined object has its own table, containing its data. When a website, like *medium.local*, has many different user defined objects, querying these objects becomes a heavy task for *any* relational DBMS, as many tables have to be `joined` (around 30 for *medium.local*.

The content of these objects is primarily of interest to the customer and the end-user, to humans. The queries performed on the data are generally not "interested" in "human content", like blog post texts. If we were to perceive all these objects as just objects with different attributes, we can allow ourselves to approach this matter in a different way.

Each of these approaches should be checked for compatibility with Reprovinci's UDO-approach, allowing for sorting, searching, indexing of attributes, and perform better than Reprovinci's current approach.

7.5.1 The entity-attribute-value model

The entity-attribute-value (EAV)-model is one of these approaches. This approach, like the name suggests, separates entities, attributes and value. Every entity can have any number of attributes it wants, each of any type it wants. All this data is then selected in a vertical form: each row contains an attribute and its value for a certain entity. I have yet to form a reasoned opinion about this approach, but on first sight it seems like a very worrisome approach and which seems hard to query (e.g. sorting).

7.5.2 Document-oriented databases

Document-oriented databases like Apache's CouchDB and MongoDB are unstructured by nature. They allow the storage of every type of "document", a container for data. These documents are still queryable through, for example, simple comparisons or map/reduce functions, which filter and group results.

This seems like a viable alternative to the normalised approach. The downside to this approach is, first of all, having a second database to worry about, secondly, I and Reprovinci have no expertise on these databases, and lastly, being unable to directly join the documents from MySQL.

7.5.3 FriendFeed

In a 2009 blog post, FriendFeed, a social sharing site, explained how they solved their database design problem [16]. They store millions upon millions of entries, comments and "likes". Making structure changes to a table with millions of records could, they claim, lock their database for hours as it modified the table. They were happy with the proven relational DBMSs and didn't have too many confidence in document-based databases. They decided to build a non-relational storage on top of MySQL. While this issue doesn't apply to Reprovinci's CMS, their solution may.

FriendFeed stores all data *in serialised, compressed form* in a table called `entries`. To still be able to find entities by a specific attribute, they make an artificial "index". To find entities by a title, they make a table called `index_title`, which contains a reference to all entities with a title attribute and the title string. These indexes are kept as entities are modified.

Integrity between indexes and entity data is kept by what FriendFeed calls "the Cleaner". While it's typically not necessary, indexes and entities may become out of sync in transaction-less storage engines like MyISAM, as they are accessed sequentially using separate queries. The Cleaner is designed to fix this.

The same can be done for Reprovinci's UDOs. Sorting and searching can be done by keeping sortable and searchable attributes in one or two separate index tables. The database model could look like the on in figure 7.1. Attention should be paid to the storage of the entity data, as writing temporary tables to disk during querying should be prevented, and the storage of user defined collection (UDC) attributes.

7.6 Alter user interaction with user defined objects

UDOs are currently shown to the user in a large mixed list. This is the default view. The user can then choose to show only one type of UDO. I suggest making the latter the default, ie. moving the UDO choice to the side menu, with the option of showing them all in one mixed view. I believe this improves user interaction, but also performance, as only one UDO table has to be joined.

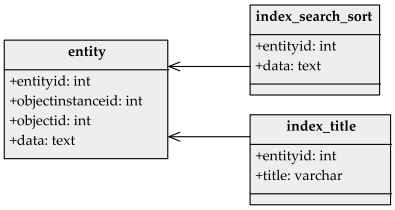


Figure 7.1: An example database model for the FriendFeed approach

Chapter 8

Conclusion

The CMS' performance has come a long way, but, even though I and Reprovinci see it as a success, has unfortunately not achieved the set goal of 500 millisecond page loads. This is due to both the four month time limit and due to the general issue underlying the CMS' code base, namely code quality. There's just a fraction too many algorithm inefficiencies, which, all added up, account for a couple of hundred milliseconds. Luckily, the limits of what can be achieved through further optimisation have been reached.

In chapter 7 I highly recommend, in order for the performance to be maintained in the future, code review should be implemented. Code should not be committed unreviewed by another developer. Furthermore, server software should be upgraded for more security and better performance; Reprovinci should implement load balancing; and Reprovinci should change the database design behind the concept driving their CMS: UDOs.

Reprovinci have a better performing CMS a lot of customers will be very happy with.

Afterword

This project has been quite the endeavour. Optimising the loading of objects was especially arduous. In the end, all this effort did pay off and I am happy to see the CMS has become a lot faster, especially for Reprovinci's direct customers. I have learnt a lot about code profiling and optimising queries, how application design has an impact on performance in the long run, and many more things.

In retrospect, I wished I would have explored more areas other than profiling code, analysing queries and the odd database design change, and I think I would have, had I had more time. I would have liked to do more research into the CMS' design, perform static code analysis and delve into the database design. I would also investigate Zend's contribution to the parse time (e.g. what is the minimum request time for a simple controller and view?).

I have much enjoyed my internship and I would again like to thank Reprovinci for the opportunity to tinker with their CMS and allowing me to spend time working on my thesis.

Glossary

Glossary

CMS content management system. 5, 11, 13, 15, 16, 18, 21–23, 25, 26, 28, 30, 31, 33, 37, 40, 41, 43, 49–51, 55, 57

CPU central processing unit. 25

DBMS database management system. 34, 37, 49, 51

DRY Don't Repeat Yourself. 31

EAV entity-attribute-value. 51

HTML Hypertext Markup Language. 11, 25, 26

HTTP Hypertext Transfer Protocol. 5, 27, 39–41, 43, 45

MVC Model-View-Controller. 5, 22

PHP PHP: Hypertext Preprocessor. 11, 16, 33, 40, 41

RAM (random-access memory) is a form of primary storage that is often volatile (loses its state on loss of power) and can be accessed in a random – i.e. non-sequential – manner. 16, 22, 29, 60

RBAC role-based access control. 5, 23, 38, 84

SEO search engine optimisation. 13

SVN Subversion. 19, 32

UDC user defined collection. 51

UDF user defined form. 21, 22

UDO user defined object. 21–23, 26, 33, 34, 36–38, 51, 52, 55, 67–69

URL uniform resource locator. 22

VCS version control system. 19, 49

XML eXtensible Markup Language. 23

B-tree A B-tree is a form of binary search tree that allows more than two children [12]. 37

clone-and-modify programming is where a piece of working, proven code is cloned to another location and modified only slightly. While this is a simple and inexpensive short-term solution, bugs can be hard to solve as they probably (but might not, or differently) affect the cloned code as well. 15, 33

database management system is a system that offers the ability to manipulate the structured data stored in its storage engine. Many database management systems support different storage engines. 34, 37, 49, 51

Don't Repeat Yourself is a principle that states that "every piece of knowledge must have a single, unambiguous, authoritative representation within a system"[7]. 31

forking is the act of (legally) copying a project's source code and starting development on one's own. 21

full table scan A full table scan means all rows are read sequentially while looking for a certain column value. 37

high-level programming language High-level programming languages are languages which generally abstract computer-specific details, hide memory management and/or are easier to use than lower-level programming languages. 31

horizontal scaling Scaling horizontally, or scaling out, means adding more nodes to a system. A web application's load can, for example, be distributed among more than one server. [8]. 22

interpreted language A language which is interpreted and converted to machine instructions (opcodes) on every execution. Interpreted language are often considered slow, because of the overhead created by interpretation. 22

loose coupling In a loosely coupled system, separate components know little about other components, other than their interface (how to address them). 22, 33

object composition is the act of combining simple objects into a more complex structure to perform a complex task. 33, 49

role-based access control is a system where individual rights, like 'editing content', are attached to roles, like 'webmaster'. Whenever content needs to be edited, access is granted or denied based on these rules.. 5, 23, 38

separation of concerns is a paradigm that, in summary, states code should typically only be responsible for one task only. 49

SoC separation of concerns. 33

user defined object A custom object, specified by an XML file, that can be edited in the CMS' administration area and included in the website structure. 51

vertical scaling Scaling vertically, or scaling up, means adding resources to a single node in a system. For example, more RAM can be added to a server in case of memory shortage. [8]. 22

Index

active probing, 29
ApacheBench, 39

cardinality, 37
clone-and-modify programming, 15, 33
computational performance, 29
content management system, 15
Couchmark, 40

data integrity, 36

forking, 21

horizontal scaling, 22

index, 37

JMeter, 39

load balancing, 50, 55

partial objects, 38, 79
peer reviewing, 49

R, 44
routing, 22

scaling
    horizontal scaling, 22, 50
    vertical scaling, 22
server log analysis, 29
stress resistance, 29

thread state, 36

url, 22

vertical scaling, 22

Zend Framework, 22

Bibliography

[1] "Computer performance", *Wikipedia*, accessed April 8, 2011, http://en.wikipedia.org/wiki/Computer_performance

[2] "Random access memory (RAM)", *ATIS Telecom Glossary*, accessed April 8, 2011, http://www.atis.org/glossary/definition.aspx?id=2189

[3] "Random-access memory: Swapping", *Wikipedia*, accessed April 8, 2011, http://en.wikipedia.org/wiki/Random-access_memory#Swapping

[4] "Perceived performance", *Wikipedia*, accessed April 12, 2011, http://en.wikipedia.org/wiki/Perceived_performance

[5] **Chiew, Thiam Kian**, "Web page performance analysis", *University of Glasgow*, 2009

[6] "Observing Processor Queue Length", *Microsoft TechNet*, accessed May 9, 2011, http://technet.microsoft.com/en-us/library/cc938643.aspx

[7] "Don't Repeat Yourself", *Cunningham & Cunningham, Inc.*, February 24, 2011, accessed May 9, 2011, http://c2.com/cgi/wiki?DontRepeatYourself

[8] "Scalability", *Wikipedia*, accessed May 13, 2011, http://en.wikipedia.org/wiki/Scalability#Scale_horizontally_vs._vertically

[9] "Preface", PHP, accessed May 13, 2011, http://nl.php.net/manual/en/preface.php

[10] **Haiping Zhao**, "HipHop for PHP: Move Fast", *Facebook Developers*, February 2, 2010, accessed May 13, 2011, http://developers.facebook.com/blog/post/358

[11] "How MySQL Uses Indexes", *MySQL Documentation*, accessed May 18, 2011, http://dev.mysql.com/doc/refman/5.0/en/mysql-indexes.html

[12] "B-tree", *Wikipedia*, accessed May 18, 2011, http://en.wikipedia.org/wiki/B-tree

[13] "Restrictions on Subqueries", *MySQL Documentation*, accessed May 19, 2011, http://dev.mysql.com/doc/refman/5.0/en/subquery-restrictions.html

[14] "Code review", *Wikipedia*, accessed May 20, 2011, http://en.wikipedia.org/wiki/Code_review

[15] "Load balancing (computing)", *Wikipedia*, accessed May 20, 2011, http://en.wikipedia.org/wiki/Load_balancing_(computing)

[16] "How FriendFeed uses MySQL to store schema-less data", *Blog of Bret Taylor*, accessed May 19, 2011, http://bret.appspot.com/entry/how-friendfeed-uses-mysql

[17] "General thread states", *MySQL Documentation*, accessed May 29, 2011, http://dev.mysql.com/doc/refman/5.0/en/general-thread-states.html

[18] **Thorsten Ruf**, "Dispatch process overview", *Nethands*, accessed May 30, 2011, last modified November 19, 2008, `http://nethands.de/download/zenddispatch_en.pdf`

[19] **Sebastian Bergmann**, "Benchmark of PHP Branches 3.0 through 5.3-CVS", accessed May 30, 2011, last modified February 7, 2008, `BenchmarkofPHPBranches3.0through5.3-CVS`

[20] **Sean Michael Kerner**, "PHP 5.3 Accelerates PHP", *internetnews.com*, accessed May 30, 2011, last modified June 30, 2009, `http://www.internetnews.com/dev-news/article.php/3827756/PHP-53-Accelerates-PHP.htm`

[21] "The BLOB and TEXT types", *MySQL Documentation*, accessed May 30, 2011, `http://dev.mysql.com/doc/refman/5.0/en/blob.html`

[22] "How MySQL Uses Internal Temporary Tables", *MySQL Documentation*, accessed May 30, 2011, `http://dev.mysql.com/doc/refman/5.0/en/internal-temporary-tables.html`

Appendix A

Optimisations

A.1 `Collection#_checkCallback()`

Description

`Collection#_checkCallback()` is called up to 30 000 times per request, performing a context switch – passing around parameters and return values – every time.

Use case	Gain
all	5%

Expected performance gain for optimisation A.1.

Process

- Inlined method's content in those places where it is called, as seen in listing A.1.
- Duplication is justified as code is duplicated in one small file.

```
<?php
\# Replaced instances like:
public function addItem(\$obj, \$key = null) {
        \$this->\_checkCallback();
        (...)
}

\# With:
public function addItem(\$obj, \$key = null) {
        if(isset(\$this->\_onload) \&\& !\$this->\_isLoaded) { \$this->\_isLoaded = true; @call\_user\
             _func(\$this->\_onload, \$this); }
        (...)
}
```

Listing A.1: Inlining of `_checkCallback`

Result

Use case	Gain
all	10%

Achieved performance gain for optimisation A.1.

A.2 `ObjectCollection#loadObjectInstances()`

Description

Rewrite the object loading process in `ObjectCollection#loadObjectInstances()`.

Use case	Gain	Argumentation
small.local	30%	Not much content and not many different UDOs.
medium.local	40%	Many different UDOs.
large.local	50%	Not many different UDOs, lot of content.

Expected performance gain for optimisation A.2.

Process

- Rewriting of multiple queries into one single query.
- Sorting is problematic, postpone until all logic is implemented.
- Realisation of need for reliable, independent testing environment to produce consistent benchmarking results.
- Specification of own workstation as basis for measurements.
- Refactoring into four separate entities, as seen in figure A.1.
  - Separation of concerns.
  - "Abstraction space".
  - Requires object-definition-specific cache entry, instead of `ObjectCollection` cache entry. "Normal" instantiation of `ObjectCollection` is then possible.
- Separate loading of object ids and actual data.
  - Allows for sorting on smaller result set.
  - Allows for pagination (determine how many objects in advance, then load actual data for page subset).
- Applying user permissions very slow.
  - Fixing of some indexes.
  - Subselect performs badly.
  - Joining of derived table, which selects permissions seems fastest.
- Joining of related users is very slow.
  - User data includes an order history, row count through the roof.
  - Post-load users.
- Loading of related objects through recursion of `ObjectLoader`
  - Keep all loaded objects, partial objects and partial users in identity map across recursions for reuse and integrity of associations.
  - Configuration is partially passed on (not everything applies to related objects)

Result

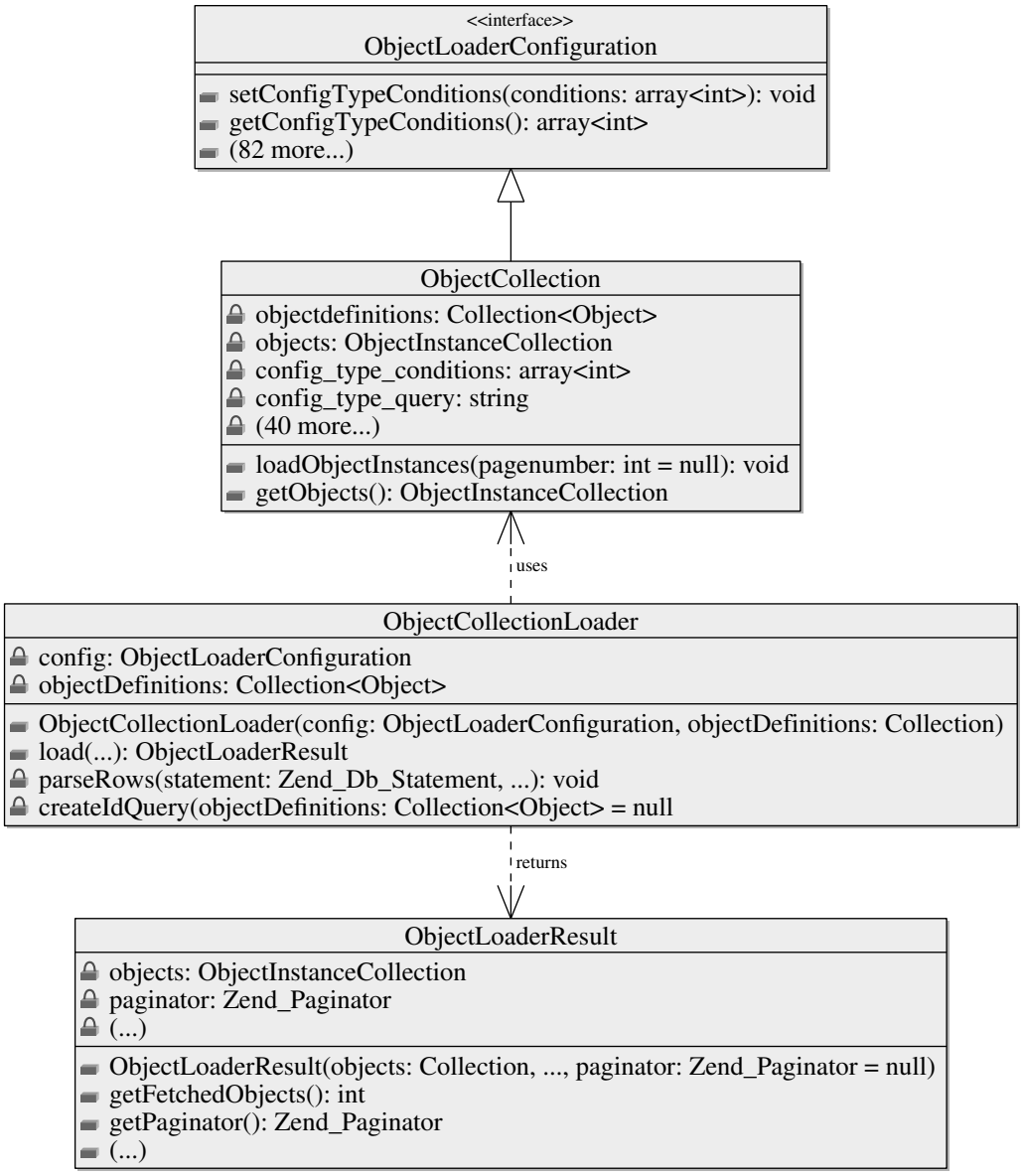


Figure A.1: Situation after refactoring of `ObjectCollection`. The ellipses replace details that would only confuse.

Use case	Gain	Argumentation
small.local	TODO	Not much content and UDOs.
medium.local	TODO	Many UDOs.
large.local	TODO	Not so much UDOs, lot of content.

Achieved performance gain for optimisation A.2.

A.3 Administration: load only in-progress with published fallback

Description

All versions of objects are loaded in many parts of the administration area, such as the content list and the content detail page. Load only in-progress versions or fallback to published versions.

Use case	Gain
small.local	12%
medium.local	5%
large.local	24%

Expected performance gain for optimisation A.3. Based on loading of only published versions compared to loading of all versions.

Process

- Concluded loading through deduction (determining if there's an in-progress version and falling back to published if necessary) is difficult and adds joins for each UDO table.
- Addition of `ObjectLoaderConfiguration#configLoadLatestVersions`.
- Addition of in-progress flag.
- Loading of in-progress *and* published versions, perform fallback programmatically.

Result

Use case	Page	Gain
small.local	content list	7.6%
medium.local	content list	25.9%
large.local	content list	13.8%
small.local	detail page	1.0%
medium.local	detail page	7.9%
large.local	detail page	32.3%

Achieved performance gain for optimisation A.3.

A.4 User permissions

Description

User permissions are tried using `Authorisation#hasAccess()`. This method loops the user's roles and permissions on each call. Create possibility for direct lookup. Currently takes up 20% of requests (registered users only).

Use case	Gain
small.local	18%
medium.local	6%
large.local	8%

Expected performance gain for optimisation A.4. Based on quick fix.

Process

- Created direct lookup array on first call, which is "cached" in a static variable for subsequent use.

```php
<?php
# Replaced instances like:
$roles = $objUser -> getRoles();
foreach ($roles as $role) {
	foreach ($role -> getPermissions() as $permission) {
		if ($permission -> getLabelId() == $labelid && (intval($rights) & intval($permission -> getRights()))) {
			return true;
		}
	}
}
return false;

# With:
$accessCache = self::getLabelRights($objUser);
return isset($accessCache[$labelid]) && ($accessCache[$labelid] & (int) $rights);

# And:

/**
 * Returns the additive rights per label per user. Sorted on label.
 * @return array<int label, int rights>
 */
public static function getLabelRights(User $user) {
	$_userId = $user->getId();
	if(isset(self::$accessCache[$_userId])) {
		return self::$accessCache[$_userId];
	} else {
		$accessCache = array();
		$roles = $user -> getRoles();
		foreach ($roles as $role) {
			foreach ($role -> getPermissions() as $permission) {
				if(isset($accessCache[$permission->getLabelId()])) {
					$accessCache[$permission->getLabelId()] |= (int) $permission->getRights();
				} else {
					$accessCache[$permission->getLabelId()] = (int) $permission->getRights();
				}
			}
		}
		ksort($accessCache);
		return self::$accessCache[$_userId] = $accessCache;
	}
}
```

Listing A.2: Direct lookup in `Authorisation#hasAccess()`

Result

Use case	**Page**	**Gain**
small.local	content list	19.9%
medium.local	content list	19.5%
large.local	content list	15.8%
small.local	detail page	0.0%
medium.local	detail page	4.1%
large.local	detail page	0.8%

Achieved performance gain for optimisation A.4.

A.5 String::renderNameToKey()

Description

String::renderNameToKey() is a method that turns a regular string in to a so-called slug. A string like "Fifa suspends Bin Hammam & Warner" becomes: "fifa-suspends-bin-hammam-warner". This method takes up 8-19% of all requests, as it is extensively used by the navigational structure and rendering of templates.

Use case	Gain
all	2-7%

Expected performance gain for optimisation A.5. Based on profiling report.

Process

- Replacing calls to ereg_replace() with a regular str_replace() is an option, but this isn't a very nice solution as it doesn't handle all characters.
- Using the *iconv* library is an option.
- Fixed some difficulties with iconv library on Windows.

```
<?php

## Replacing:
$key = strtolower($name);
$key = trim($key);

$key = ereg_replace("([é])+", "e", $key);
$key = ereg_replace("([è])+", "e", $key);
$key = ereg_replace("([ë])+", "e", $key);
# (...)
return $key;

## With:
# Replace many special characters with their ASCII counterpart
$key = iconv('UTF-8', 'ASCII//TRANSLIT//IGNORE', $name);

# Remove any other non-ASCII characters
$key = preg_replace('/[^␣\-a-z0-9]/i', '', $key);

return $key;
```

Listing A.3: Optimisation of String::renderNameToKey()

Result

Use case	Gain
all	2%

Achieved performance gain for optimisation A.5

Somewhat of a disappointment, this one. The profiling tool has proven to be inaccurate, with values adding up to a total execution time of 102%, for example. Due to measurement errors, this optimisation was largely a waste of time. However, being able to handle more characters, the functionality of String::renderNameToKey() has been improved.

A.6 NavigationCollection#getNode()

Description

large.local's product pages utilises a lot of calls to `NavigationCollection#getNode()`. This method is very inefficient, looking up a path down the navigational structure tree. The lookup can be converted to a direct lookup, using a flat, unstructured `Collection` of ids pointing to `NavigationNode`s. Method accounts for 8 to 25% of a typical product list request on *large.local*. Other use cases are largely unaffected.

Use case	Gain
large.local	5-10%

Expected performance gain for optimisation A.6.

Process

- Modified build-up of `NavigationCollection` to also create a flat, unstructured `Collection` of ids pointing to `NavigationNode`s.
- Modified `NavigationCollection#getNode()` to employ this `Collection`.

```
<?php
# Replaced:
$nodes  = $this -> findPath($id);
$nodes  = array_reverse($nodes);
$node = null;

$coll = $this -> structure;

foreach ($nodes as $key => $value) {
        if ($coll -> exists($value)) {
                $node = $coll -> getItem($value);
                $coll = $node -> getChildren();
        }
}

return $node;

# With:
return $this->allNodes->getItem($id);

# Where the build-up process is modified:
while ($value = $result -> fetch()) {
        $row = $value;

        $_nodeId = (int) $row['navigationnode_id'];
        if($allNodes->exists($_nodeId)) {
                $node = $allNodes->getItem($_nodeId);
        } else {
                $node = new NavigationNode();
                $node->setId($_nodeId);
                $node->setPartial(true);
                $allNodes->addItem($node, $_nodeId);

                if($rootNode === null) {
                        # Nodes are sorted by parentid, assume first record is the root node
                        $rootNode = $node;
                }
        }

        # (...)
}
```

Listing A.4: Direct lookup in `NavigationCollection#getNode()`

Result

Use case	**Page**	**Registered user?**	**Gain**
small.local	page w/ list view	n	0.3%
medium.local	page w/ list view	n	0.1%
large.local	page w/ list view	n	0.1%
small.local	page w/ list view	y	0.3%
medium.local	page w/ list view	y	0.1%
large.local	page w/ list view	y	6.6%

Achieved performance gain for optimisation A.6.

A.7 Excessive calling of `NavigationCollection#getNode()`

Description

large.local's product pages utilises a lot of calls to `NavigationCollection#getNode()`. This introduces unnecessary overhead. Remove any unnecessary calls. Method accounts for about 10% of a typical product list request on *large.local*. Other use cases are largely unaffected.

Use case	Gain
large.local	5-10%

Expected performance gain for optimisation A.7.

Process

- Removed unnecessary calls to `NavigationCollection#getNode()`.

Result

Use case	Page	Registered user?	Gain
large.local	page w/ list view	y	7.2%

Achieved performance gain for optimisation A.7.

A.8 PageContentHandler#grabData() is slow

Description

`PageContentHandler#grabData()` is quite slow for registered users. This is especially noticeable on *large.local*'s product lists.

Use case	Registered user?	Gain
large.local	y	65%

Expected performance gain for optimisation A.8. Based on difference between parse time for anonymous user and parse time for registered user.

Process

- Determined page caching is used for anonymous users. A simple, but inadequate solution is to create caching for registered users as well.
- Created concept of "caching strategy".
- Created two caching strategies: a default and an "aggressive one". The latter also caches for registered users.
- Complication here is that registered users can have different permissions, meaning the page can be different for each user. The aggressive content strategy is therefore only useful in environments with little roles and permissions (such as *large.local*).
- Replaced all cache calls in `PageContentHandler#grabData()` with calls to a caching strategy.
- The caching strategy determines whether to cache or not and how to cache.

```php
<?php
# In the constructor:
$_cacheConfig = $registry->config->cache;
if(isset($_cacheConfig->contentcache)) {
        $this->cacheStrategy = new $_cacheConfig->contentcache;
} else {
        $this->cacheStrategy = new DefaultContentCachingStrategy();
}

# On various places, replaced calls like:
if ($this -> anonymous && ...) {
        $cache -> save(...);
}
# With:
$strategy->savePaginatorInfo(...);
```

Listing A.5: Caching strategy in `PageContentHandler#grabData()`

```php
<?php

/**
 * Contains the strategy used for loading, parsing and caching object instances for
 * a PageContentHandler. This allows for switching strategies that can be less or more
 * efficient, depending on the site's characteristics.
 * @author Reinier Kip <rkip@reprovinci.nl>
 */
interface ContentCachingStrategy {

        /**
         * Implementation should implement no-argument constructor, as the strategy is
         * constructed automagically from the config.xml.
         */
        public function __construct();


        /**
         * Loads a paginator from cache. If no paginator was found, this method returns null.
         * @param NavigationNode $node
         * @return Zend_Paginator|null
         */
        public function loadPaginator(NavigationNode $node);
```

```
24
25          /**
26           * Saves paginator info to cache in the form of:
27           * array('items' => array(int objectinstance, ...), 'itemsperpage' => int)
28           * @param NavigationNode $node
29           * @param array<int> $items
30           * @param int $itemsPerPage
31           */
32          public function savePaginatorInfo(NavigationNode $node, array $items, $itemsPerPage);
33
34
35          /**
36           * Returns whether the caching of rendered content is enabled.
37           * @return boolean
38           */
39          public function contentCachingEnabled();
40
41          /**
42           * Saves rendered list or detail content to cache.
43           * @param NavigationNode $node
44           * @param string $type list|detail
45           * @param int $page
46           * @param string $content
47           * @param array $tags
48           */
49          public function saveContent(NavigationNode $node, $type, $page, $content, array $tags);
50
51          /**
52           * Loads rendered list content from cache.
53           * @param NavigationNode $node
54           * @param string $language Usually $registry->currentlanguage
55           * @param int $page
56           */
57          public function loadListContent(NavigationNode $node, $language, $page);
58
59          /**
60           * Loads rendered detail content from cache.
61           * @param NavigationNode $node
62           * @param string $language Usually $registry->currentlanguage
63           * @param int $page
64           */
65          public function loadDetailContent(NavigationNode $node, $language, $page);
66
67  }
```

Listing A.6: The `ContentCachingStrategy` interface

Result

Use case	Page	Registered user?	Gain	Caching strategy
small.local	page w/ list view	y	0.0%	default
medium.local	page w/ list view	y	-0.0%	default
large.local	page w/ list view	y	74.3%	aggressive

Achieved performance gain for optimisation A.8.

A.9 `PageContentHandler#returnPageContentContainer()` very slow

Description

large.local's product pages for registered users are very slow (in the order of 3-5 seconds). This is caused by multiple issues.

Use case	Registered user?	Gain
large.local	y	65%

Expected performance gain for optimisation A.9. Based on estimations for optimisations stated below.

Process

- Implemented optimisations A.4, A.6, A.7 and A.8.

Result

Use case	Page	Registered user?	Gain	Caching strategy
small.local	page w/ list view	y	0.0%	default
medium.local	page w/ list view	y	-0.0%	default
large.local	page w/ list view	y	81.8%	aggressive

Achieved performance gain for optimisation A.9.

A.10 `NavigationCollection` buildup

Description

The buildup of the `NavigationCollection` takes very long. The collection is rebuilt after changes to the navigational structure of the underlying website. This is not a common process; if possible, changes are mostly done in the cache entry as well.

Use case	Gain
all	80%

Guesstimated performance gain for optimisation A.10.

Process

- Optimised query.
  - Added `inprogress` flag.
  - Removed subqueries, replaced with `WHERE` on `inprogress`.
- Moved creation of tree of `NavigationNode`s from *after* row parsing *into* the row parsing procedure.
- Reimplemented buildup using partial objects.
- Related files for *all nodes* were loaded n_{nodes} times.

Result

Use case	Used to be	Is now	Gain
small.local	~4750ms	71ms	98.5%
medium.local	~56s	625ms	98.9%
large.local	~7m	~4800ms	98.9%

Achieved performance gain for optimisation A.10.

A.11 `PublishedCollection#load()` is very slow

Description

`PublishedCollection#load()` is very slow and is used on content item detail pages.

Use case	Gain
all	10-50%

Expected performance gain for optimisation A.11. Based on quick query test.

Process

- Added `inprogress` flag.
- Removed subqueries and add `WHERE` clause on `inprogress`.

Result

Use case	Page	Gain
small.local	content detail	2.2%
medium.local	content detail	11.3%
large.local	content detail	77.2%

Achieved performance gain for optimisation A.11.

A.12 `PublishedCollection#getNavigationNodeIds()` is quite slow

Description

`PublishedCollection#getNavigationNodeIds()` is quite slow, taking up roughly 50% (inclusive) of a detail page request on *large.local*. Possible solutions:

- Optimise lookup. Method makes use of a couple of loops to find what it's looking for.
- It's called only once per request. Replace it with a database query.

Use case	Page	Gain
all	content detail	25-45%

Expected performance gain for optimisation A.12. Based on quick query test.

Process

- Replaced lookup in entire `PublishedCollection` with database query.

Result

Use case	Page	Gain
small.local	content detail	3.7%
medium.local	content detail	24.9%
large.local	content detail	25.8%

Achieved performance gain for optimisation A.12.

A.13 Investigate `ObjectLoader`'s row parsing

Description

I had a feeling `ObjectLoader`'s row parsing could be optimised significantly.

Page	Gain
content detail	5-10%
content list	5-20%

Expected performance gain for optimisation A.13.

Process

- Abstracted logic like filling an object's version away into separate method for increase insight through profiling.
- Object cloning is sluggish.
  - Optimised deep cloning of `Collection`.

Result

Use case	Page	Gain
small.local	content detail	0.2%
medium.local	content detail	0.0%
large.local	content detail	-0.1%
small.local	content list	1.9%
medium.local	content list	1.4%
large.local	content list	1.9%

Achieved performance gain for optimisation A.13.

Apart from a few statements concerning cloning and the optimisation of a `Collection`'s cloning, there wasn't much to optimise.

A.14 Checking of permissions in `ObjectLoader`

Description

During the writing of this thesis, I stumbled on these two pieces of MySQL documentation [13]:

> (...) for a statement that uses an `IN` subquery, the optimizer rewrites it as a correlated subquery. Consider the following statement that uses an uncorrelated subquery:
>
> `SELECT ... FROM t1 WHERE t1.a IN (SELECT b FROM t2);`
>
> The optimizer rewrites the statement to a *correlated subquery*:
>
> `SELECT ... FROM t1 WHERE EXISTS (SELECT 1 FROM t2 WHERE t2.b = t1.a);`

> `UPDATE t ... WHERE col = (SELECT * FROM (SELECT ... FROM t...) _t);`
>
> Here the result from the subquery in the `FROM` clause is *stored as a temporary table*, so the relevant rows in `t` have already been selected by the time the update to `t` takes place.

By combining these two pieces of information, I concluded that for each object, MySQL performs the permissions subquery. By wrapping these in a `SELECT`, the results of this query would be stored in a temporary table. This temporary table is then queried for each object.

Use case	Gain
all	35%

Expected performance gain for optimisation A.14.

```
<?php
# Permissions were first determined using a derived table, joined to the primary query:

$labelSelect = new Select($db);
$labelSelect
        ->distinct()
        ->from('Permission', array('labelid'))
        ->join('RolePermission', 'RolePermission.permissionid␣=␣Permission.id', null)
        ->join('Role', 'Role.id␣=␣RolePermission.roleid', null)
        ->join('UserRole', 'UserRole.roleid␣=␣Role.id', null)
        ->where('UserRole.userid␣=␣?', (int) $user->getId())
        ;

$rights = (int) $config->getConfigRights();
if($rights) {
        $labelSelect->where('Permission.rights␣&␣'.$rights.'␣=␣'.$rights);
}

$select
        ->join('ObjectInstanceLabelVersion', 'ObjectInstanceLabelVersion.objectinstanceid␣=␣
             ObjectInstance.id', null)
        ->join('ObjectInstanceLabel', 'ObjectInstanceLabel.versionid␣=␣ObjectInstanceLabelVersion.id',
             null)
        ->join(array('ObjectInstanceLabelPermission' => $labelSelect), 'ObjectInstanceLabel.labelid␣=␣
             ObjectInstanceLabelPermission.labelid', null)
        ;

# Using a subquery, notice the SELECT encapsulating the subquery:

// (...)
$select
        ->join('ObjectInstanceLabelVersion', 'ObjectInstanceLabelVersion.objectinstanceid␣=␣
             ObjectInstance.id', null)
        ->join('ObjectInstanceLabel', 'ObjectInstanceLabel.versionid␣=␣ObjectInstanceLabelVersion.id',
             null)
        ;
$select->where('ObjectInstanceLabel.labelid␣IN␣(SELECT␣`labelid`␣FROM␣('.$labelSelect.')␣`permissions`)
      ');
```

Listing A.7: Change in permissions check in `ObjectLoader`

Result

Use case	Page	Gain
small.local	content list	1.9%
medium.local	content list	52.5%
large.local	content list	-0.4%

Achieved performance gain for optimisation A.14.

medium.local has a lot of roles and permissions in its RBAC system. This may explain the large performance gain, while the other two remain largely unaffected.

Appendix B

Testplans

```
include Couchmark

debug false

class Float
  def round_to(x)
    (self * 10**x).round.to_f / 10**x
  end
end

# Connect to metrics agent, residing on the web server
agent :localhost, 4300

# Collection of user metrics
default_metrics do |response, sample|
        headers = response.headers
        sample.set_metric 'php_memory_peak',
                (headers['X-Metric-Memory-Usage-Peak'].to_f()/1024/1024).round_to(2) # MB
        sample.set_metric 'parse_time',
                headers['X-Metric-Parse-Time'].to_f().round_to(2) # s
        sample.set_metric 'cpu_load',
                get_sample('cpu_load', 30)
end


http_defaults :port => 80

for version in ['cms', 'cms-trunk'] do
        for host in ['small.local', 'medium.local', 'large.local'] do

                # Defaults
                http_defaults :host => host

                # One user
                users 1 do |user_id|

                        user_cookie 'CMS_VERSION', version
                        user_cookie 'OVERRIDE_MODE', 'prod'

                        # Clear query cache
                        GET "#{host}-admin-login_#{version}", '/admin/', {:MYSQL_CLEAR_CACHE => 1}

                        # Login
                        POST "#{host}-admin-login_#{version}", '/admin/auth/login', {:username => '******
                              ', :password => '******'}

                        sleep 30

                        # Load two content pages twenty times, no cache
                        for page in (1..2)
                                20.times do
                                        response = GET "#{host}-admin-content-list-nc_#{version}", "/admin/
                                              content/content/viewstate/page/value/#{page}", {:
                                              MYSQL_CLEAR_CACHE => 1, :filter => 'udo'}
                                        if response: response.sample.set_metric 'page', page end
                                end
                        end

                        sleep 30

                        Load two content pages twenty times
                        for page in (1..2)
                                20.times do
                                        response = GET "#{host}-admin-content-list_#{version}", "/admin/
                                              content/content/viewstate/page/value/#{page}", {:filter => '
```

```
                                        udo'}
                                    if response: response.sample.set_metric 'page', page end
                                end
                        end

                end

        end
end
```

Listing B.1: Testplan for the CMS' content list

```
include Couchmark

debug false

class Float
  def round_to(x)
    (self * 10**x).round.to_f / 10**x
  end
end

# Connect to metrics agent, residing on the web server
agent :localhost, 4300

# Collection of user metrics
default_metrics do |response, sample|
        headers = response.headers
        sample.set_metric 'php_memory_peak',
                (headers['X-Metric-Memory-Usage-Peak'].to_f()/1024/1024).round_to(2) # MB
        sample.set_metric 'parse_time',
                headers['X-Metric-Parse-Time'].to_f().round_to(2) # s
        sample.set_metric 'cpu_load',
                get_sample('cpu_load', 30)
end


http_defaults :port => 80

hosts_plan = {
        'small.local' => {
                'edit_udo' => {
                        'udo' => 82,
                        'new' => {'object' => '15','labelid' => '30','authorid' => '','NL_name' => '
                                couchmark-1','NL_intro' => 'intro','NL_interview' => 'interview','
                                lifecycle_activationdate' => '08-05-2009','lifecycle_activationtime' => '
                                20:34:00','lifecycle_deactivationdate' => '','lifecycle_deactivationtime' =>
                                 '','NL_metatitle' => '','NL_metadescription' => '','NL_metakeywords' => '',
                                'sortfilesvalue' => '187,186,179','sortcategoriesvalue' => '␣','
                                sortcontentvalue' => '␣','contentsearch' => '','reactionvalue' => '','
                                structure[]' => '35','structure[]' => '13','contentid' => '82'},
                        'old' => {'object' => '15','labelid' => '30','authorid' => '','NL_name' => '
                                couchmark-1','NL_intro' => 'intro','NL_interview' => 'interview','
                                lifecycle_activationdate' => '08-05-2009','lifecycle_activationtime' => '
                                20:34:00','lifecycle_deactivationdate' => '','lifecycle_deactivationtime' =>
                                 '','NL_metatitle' => '','NL_metadescription' => '','NL_metakeywords' => '',
                                'sortfilesvalue' => '187,186,179','sortcategoriesvalue' => '␣','
                                sortcontentvalue' => '␣','contentsearch' => '','reactionvalue' => '','
                                structure[]' => '35','contentid' => '82'},
                },
        },
        'medium.local' => {
                'edit_udo' => {
                        'udo' => 1910,
                        'new' => {'object' => '15','labelid' => '75','NL_title' => 'Advies␣op␣maat','
                                NL_subtitle' => 'subtitle','NL_intro' => 'intro','NL_content' => 'content','
                                NL_quicklinkname' => '','NL_quicklinkpayoff' => '','lifecycle_activationdate
                                ' => '','lifecycle_activationtime' => '','lifecycle_deactivationdate' => '',
                                'lifecycle_deactivationtime' => '','NL_metatitle' => '','NL_metadescription'
                                 => '','NL_metakeywords' => '','sortfilesvalue' => '1105','
                                sortcategoriesvalue' => '␣','sortcontentvalue' => '␣','contentsearch' => '',
                                'mailafriend' => 'on','printobject' => 'on','reactionvalue' => '','structure
                                []' => '1259','structure[]' => '1261','contentid' => '1910'},
                        'old' => {'object' => '15','labelid' => '75','NL_title' => 'Advies␣op␣maat','
                                NL_subtitle' => 'subtitle','NL_intro' => 'intro','NL_content' => 'content','
                                NL_quicklinkname' => '','NL_quicklinkpayoff' => '','lifecycle_activationdate
                                ' => '','lifecycle_activationtime' => '','lifecycle_deactivationdate' => '',
                                'lifecycle_deactivationtime' => '','NL_metatitle' => '','NL_metadescription'
                                 => '','NL_metakeywords' => '','sortfilesvalue' => '1105','
                                sortcategoriesvalue' => '␣','sortcontentvalue' => '␣','contentsearch' => '',
                                'mailafriend' => 'on','printobject' => 'on','reactionvalue' => '','structure
                                []' => '1259','contentid' => '1910'},
                },
        },
        'large.local' => {
                'edit_udo' => {
                        'udo' => 4488,
```

```
46                      'new' => {'object' => '16','labelid' => '50','NL_articlenumber' => '2341234','
                            NL_title' => 'asdfsdfaa','NL_width' => '12','NL_length' => '31','
                            NL_composition' => 'asf','NL_description' => '<p>asdfa</p>','
                            EN_articlenumber' => '2341234','EN_title' => 'asdfas','EN_width' => '12','
                            EN_length' => '31','EN_composition' => 'asf','EN_description' => '<p>
                            asdfasfa</p>','lifecycle_activationdate' => '','lifecycle_activationtime' =>
                             '','lifecycle_deactivationdate' => '','lifecycle_deactivationtime' => '','
                            NL_metatitle' => '','NL_metadescription' => '','NL_metakeywords' => '','
                            EN_metatitle' => '','EN_metadescription' => '','EN_metakeywords' => '','
                            sortfilesvalue' => '␣','sortcategoriesvalue' => '␣','sortcontentvalue' => '␣
                            ','contentsearch' => '','reactionvalue' => '','structure[]' => '22','
                            structure[]' => '62','purchaseprice' => '','salesprice' => '','taxrateid' =>
                             '3','priceruleid' => '','minquantity' => '','weight' => '','stockquantity'
                            => '','contentid' => '4488'},
47                      'old' => {'object' => '16','labelid' => '50','NL_articlenumber' => '2341234','
                            NL_title' => 'asdfsdfaa','NL_width' => '12','NL_length' => '31','
                            NL_composition' => 'asf','NL_description' => '<p>asdfa</p>','
                            EN_articlenumber' => '2341234','EN_title' => 'asdfas','EN_width' => '12','
                            EN_length' => '31','EN_composition' => 'asf','EN_description' => '<p>
                            asdfasfa</p>','lifecycle_activationdate' => '','lifecycle_activationtime' =>
                             '','lifecycle_deactivationdate' => '','lifecycle_deactivationtime' => '','
                            NL_metatitle' => '','NL_metadescription' => '','NL_metakeywords' => '','
                            EN_metatitle' => '','EN_metadescription' => '','EN_metakeywords' => '','
                            sortfilesvalue' => '␣','sortcategoriesvalue' => '␣','sortcontentvalue' => '␣
                            ','contentsearch' => '','reactionvalue' => '','structure[]' => '22','
                            purchaseprice' => '','salesprice' => '','taxrateid' => '3','priceruleid' =>
                            '','minquantity' => '','weight' => '','stockquantity' => '','contentid' => '
                            4488'},
48                  },
49          },
50  }
51
52  for version in ['cms', 'cms-trunk'] do
53          for host, in hosts_plan do
54
55                  # Defaults
56                  http_defaults :host => host
57
58                  # One user
59                  users 1 do |user_id|
60
61                          user_cookie 'CMS_VERSION', version
62                          user_cookie 'OVERRIDE_MODE', 'prod'
63
64                          # Clear query cache
65                          GET "#{host}-admin-login_#{version}", '/admin/', {:MYSQL_CLEAR_CACHE => 1}
66
67                          # Login
68                          POST "#{host}-admin-login_#{version}", '/admin/auth/login', {:username => '******
                                ', :password => '******'}
69
70                          sleep 30
71
72                          # Attach and detach object instance to and from navigation node
73                          udo_id = hosts_plan[host]['edit_udo']['udo']
74                          post_new = hosts_plan[host]['edit_udo']['new']
75                          post_old = hosts_plan[host]['edit_udo']['old']
76                          20.times do
77                                  POST "#{host}-admin-detail-edit_#{version}", "/admin/content/content/type/
                                        detail/id/#{udo_id}", post_new
78                                  POST "#{host}-admin-detail-edit_#{version}", "/admin/content/content/type/
                                        detail/id/#{udo_id}", post_old
79                          end
80
81                  end
82
83          end
84  end
```

Listing B.2: Testplan for editing content items on the detail page

```
1   include Couchmark
2
3   debug false
4
5   class Float
6     def round_to(x)
7       (self * 10**x).round.to_f / 10**x
8     end
9   end
10
11  # Connect to metrics agent, residing on the web server
12  agent :localhost, 4300
13
14  # Collection of user metrics
15  default_metrics do |response, sample|
16          headers = response.headers
```

```
17          sample.set_metric 'php_memory_peak',
18                  (headers['X-Metric-Memory-Usage-Peak'].to_f()/1024/1024).round_to(2) # MB
19          sample.set_metric 'parse_time',
20                  headers['X-Metric-Parse-Time'].to_f().round_to(2) # s
21          sample.set_metric 'cpu_load',
22                  get_sample('cpu_load', 30)
23  end
24
25
26  http_defaults :port => 80
27
28  hosts_plan = {
29          'small.local' => {
30                  'udo' => 'blikopener',
31                  'edit_udo' => {
32                          'udo' => 82,
33                  },
34          },
35          'medium.local' => {
36                  'udo' => 'article',
37                  'edit_udo' => {
38                          'udo' => 1910,
39                  },
40          },
41          'large.local' => {
42                  'udo' => 'shawl',
43                  'edit_udo' => {
44                          'udo' => 4488,
45                  },
46          },
47  }
48
49  for version in ['cms', 'cms-trunk'] do
50          for host, in hosts_plan do
51
52                  # Defaults
53                  http_defaults :host => host
54
55                  # One user
56                  users 1 do |user_id|
57
58                          user_cookie 'CMS_VERSION', version
59                          user_cookie 'OVERRIDE_MODE', 'prod'
60
61                          # Clear query cache
62                          GET "#{host}-admin-login_#{version}", '/admin/', {:MYSQL_CLEAR_CACHE => 1}
63
64                          # Login
65                          POST "#{host}-admin-login_#{version}", '/admin/auth/login', {:username => '******
                                ', :password => '******'}
66
67                          sleep 30
68
69                          # Visit detail page, no cache
70                          udo_id = hosts_plan[host]['edit_udo']['udo']
71                          30.times do
72                                  GET "#{host}-admin-detail-view-nc_#{version}", "/admin/content/content/
                                        type/detail/id/#{udo_id}", {:MYSQL_CLEAR_CACHE => 1}
73                          end
74
75                          sleep 30
76
77                          # Visit detail page
78                          udo_id = hosts_plan[host]['edit_udo']['udo']
79                          30.times do
80                                  GET "#{host}-admin-detail-view_#{version}", "/admin/content/content/type/
                                        detail/id/#{udo_id}"
81                          end
82
83                  end
84
85          end
86  end
```

Listing B.3: Testplan for viewing content items on the detail page

```
1   include Couchmark
2
3   debug false
4
5   class Float
6     def round_to(x)
7       (self * 10**x).round.to_f / 10**x
8     end
9   end
10
11  # Connect to metrics agent, residing on the web server
```

```
12  agent :localhost, 4300
13
14  # Collection of user metrics
15  default_metrics do |response, sample|
16          headers = response.headers
17          sample.set_metric 'php_memory_peak',
18                  (headers['X-Metric-Memory-Usage-Peak'].to_f()/1024/1024).round_to(2) # MB
19          sample.set_metric 'parse_time',
20                  headers['X-Metric-Parse-Time'].to_f().round_to(2) # s
21          sample.set_metric 'cpu_load',
22                  get_sample('cpu_load', 30)
23  end
24
25
26  http_defaults :port => 80
27
28  hosts_plan = {
29          'small.local' => {
30                  'web' => ['/', '/rubrieken/aan-het-woord', '/algemeen-forum/forum/straatvraag/
                        koningshuis--45.html', '/over-blikveld/de-redactie']
31          },
32          'medium.local' => {
33                  'web' => ['/', '/voor-jou/activiteiten/vakanties/hgjb-vakanties', '/voor-jou/
                        activiteiten/vakanties/algemene-info/after-summer-praise', '/voor-jou/activiteiten/
                        vakanties/contact']
34          },
35          'large.local' => {
36                  'web' => ['/sjaals/trendy-sjaals?page=1','/sjaals/trendy-sjaals?page=2','/sjaals/trendy-
                        sjaals?page=3','/sjaals/trendy-sjaals?page=4']
37          },
38  }
39
40  for version in ['cms', 'cms-trunk'] do
41          for host, in hosts_plan do
42
43                  # Defaults
44                  http_defaults :host => host
45
46                  # One user
47                  users 1 do |user_id|
48
49                          user_cookie 'CMS_VERSION', version
50                          user_cookie 'OVERRIDE_MODE', 'prod'
51
52                          # Clear query cache
53                          GET "#{host}-admin-login_#{version}", '/admin/', {:MYSQL_CLEAR_CACHE => 1}
54
55                          # Login
56                          POST "#{host}-admin-login_#{version}", '/admin/auth/login', {:username => '******
                                ', :password => '******'}
57
58                          sleep 30
59
60                          # Load front-side webpages 20 times, no cache
61                          for url in hosts_plan[host]['web']
62                                  20.times do
63                                          response = GET "#{host}-web-nc_#{version}", url, {:
                                                MYSQL_CLEAR_CACHE => 1}
64                                          response.sample.set_metric 'body64', response.body64
65                                  end
66                          end
67
68                          next
69
70                          sleep 30
71
72                          # Load front-side webpages 20 times
73                          for url in hosts_plan[host]['web']
74                                  20.times do
75                                          GET "#{host}-web_#{version}", url
76                                  end
77                          end
78
79                  end
80
81          end
82  end
```

Listing B.4: Testplan for viewing several customer website pages

Appendix C

Measurements

This appendix displays the performance of the system before and after optimisation.

Some terms:

Processor queue Shows the thread queue length of the processor. Produces a value with a meaning similar to the load average in Linux.

QCC Query Cache Clear

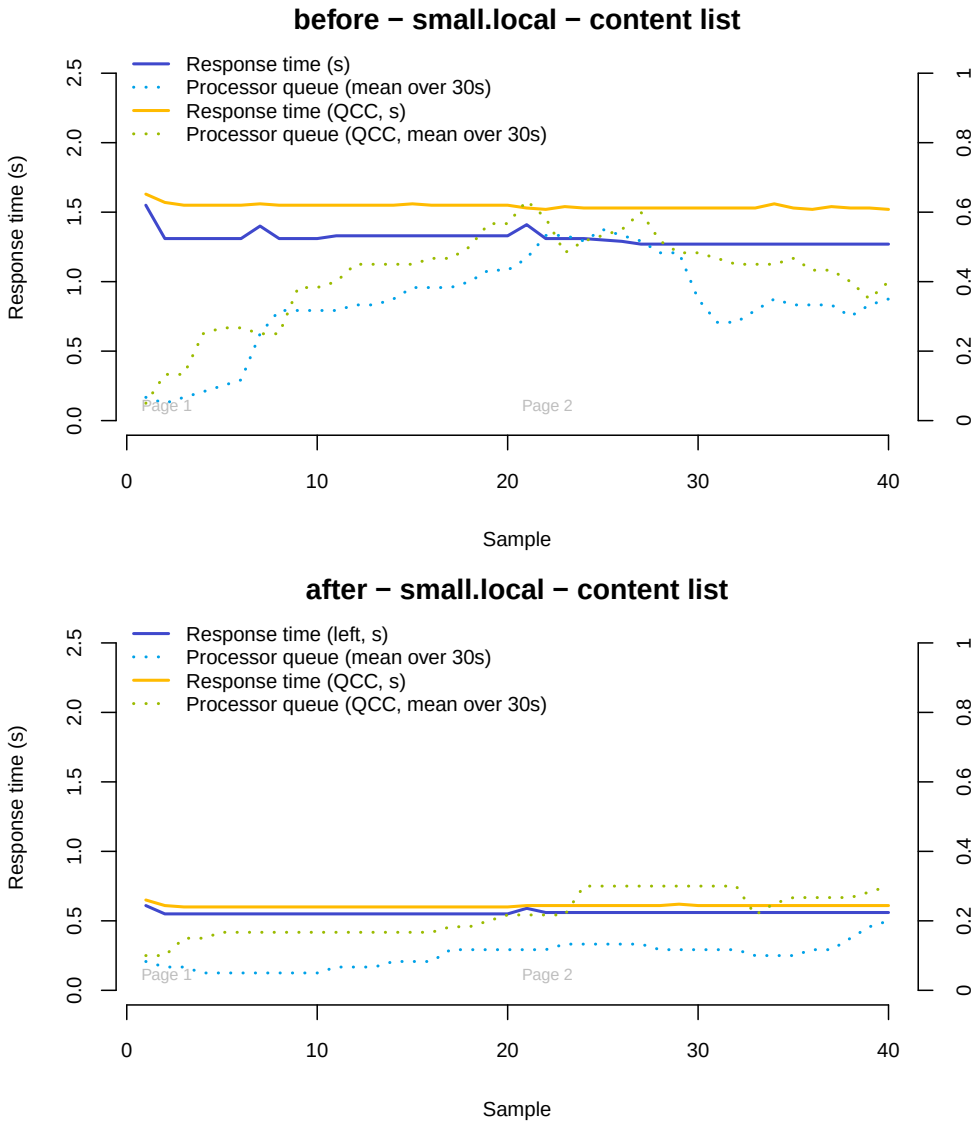


Figure C.1: A comparison of *small.local*'s content list, before and after optimisation.

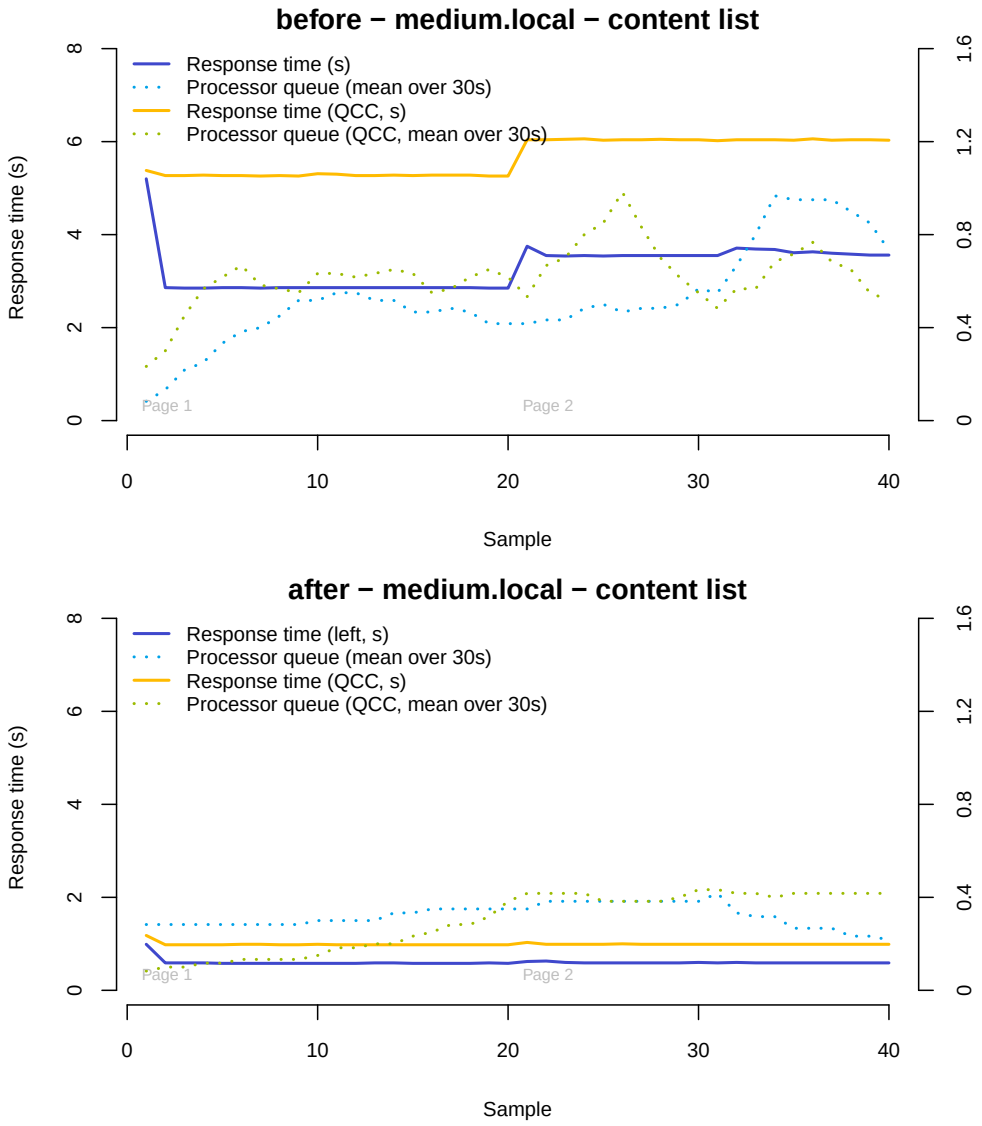


Figure C.2: A comparison of *medium.local*'s content list, before and after optimisation.

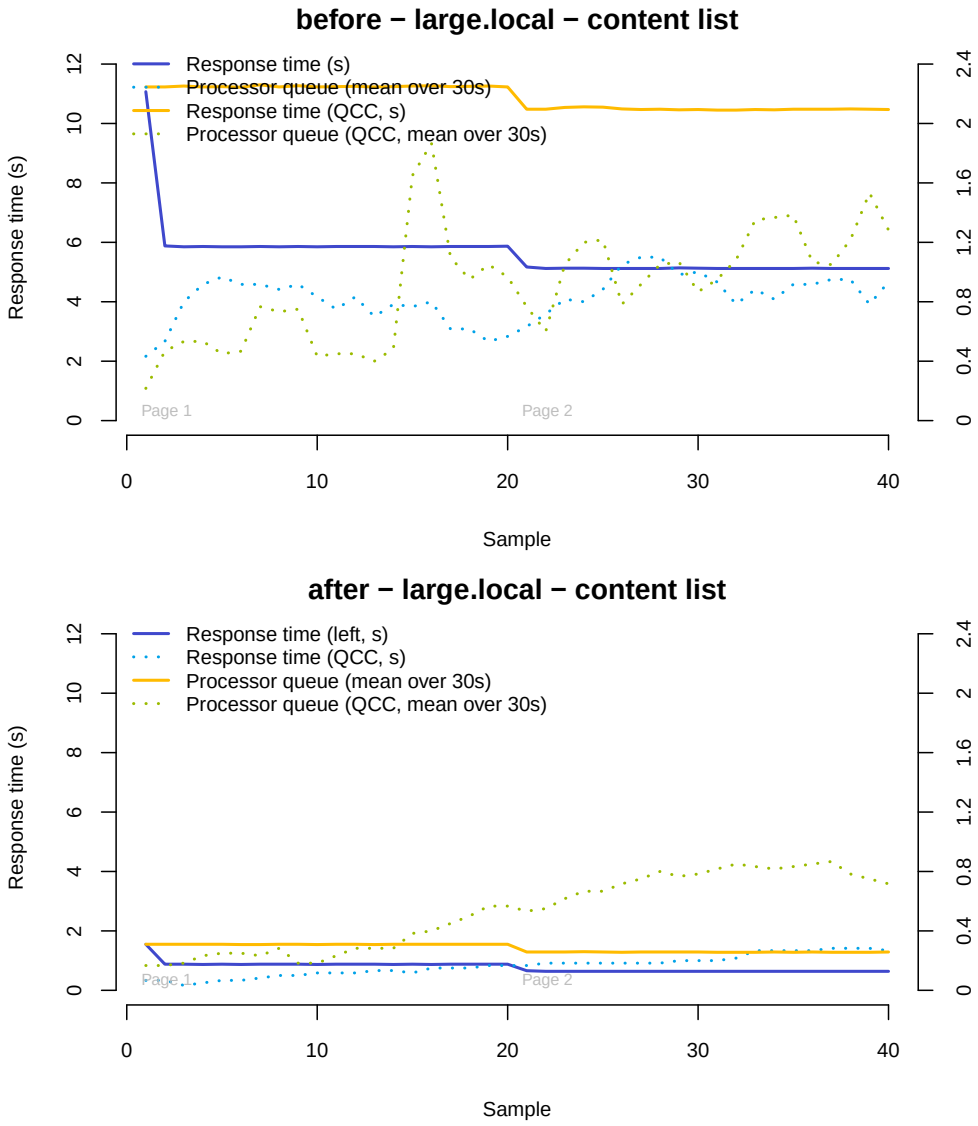


Figure C.3: A comparison of *large.local*'s content list, before and after optimisation.

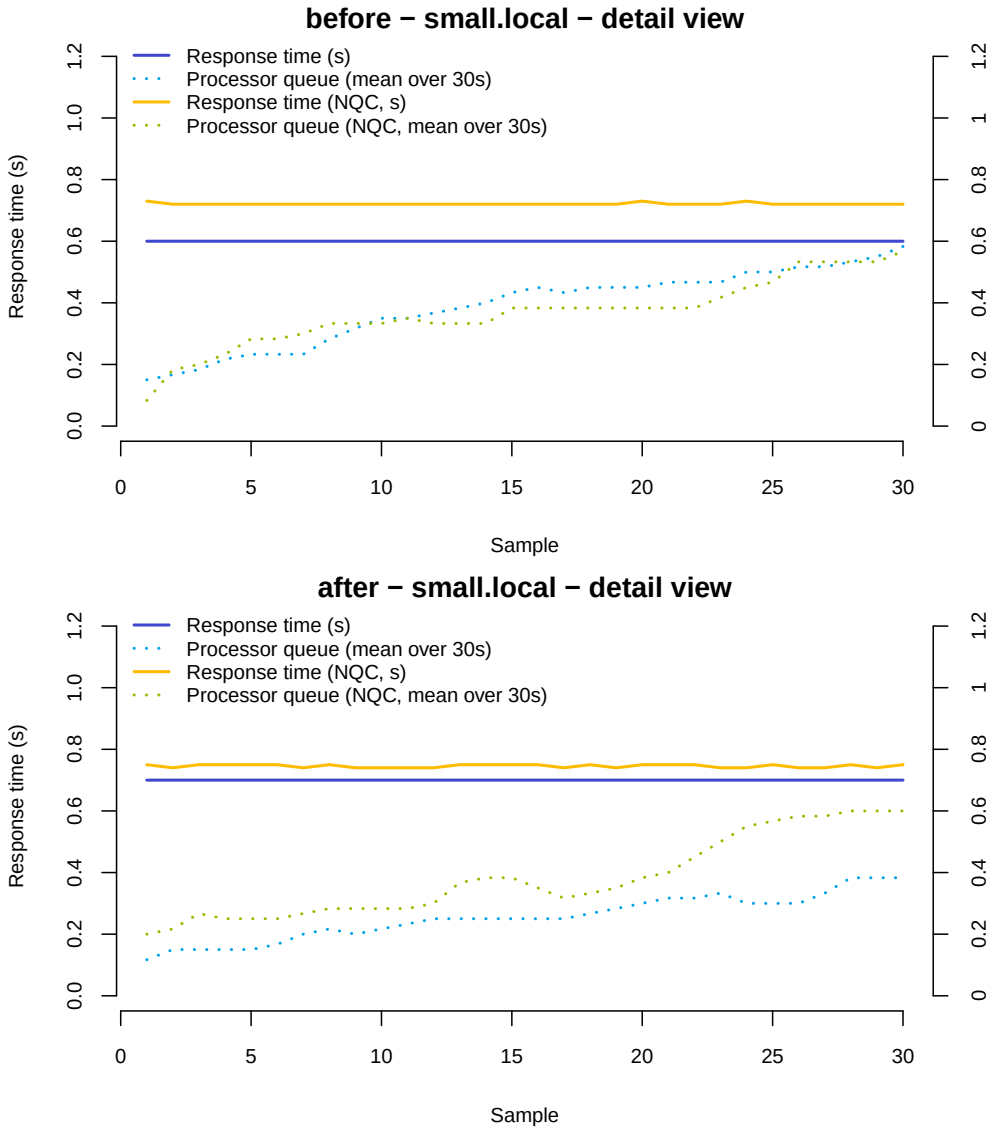


Figure C.4: A comparison of *small.local*'s detail page, before and after optimisation.

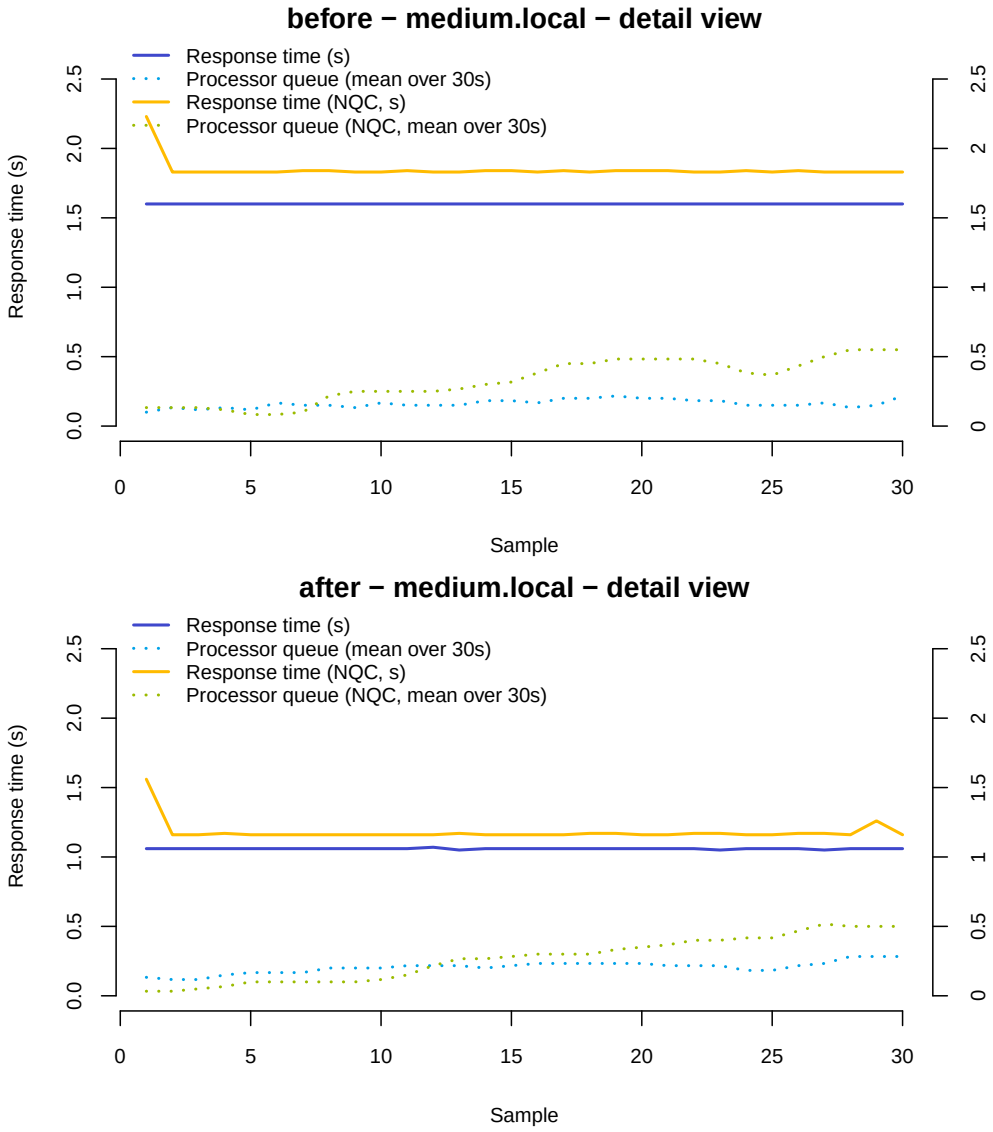


Figure C.5: A comparison of *medium.local*'s detail page, before and after optimisation.

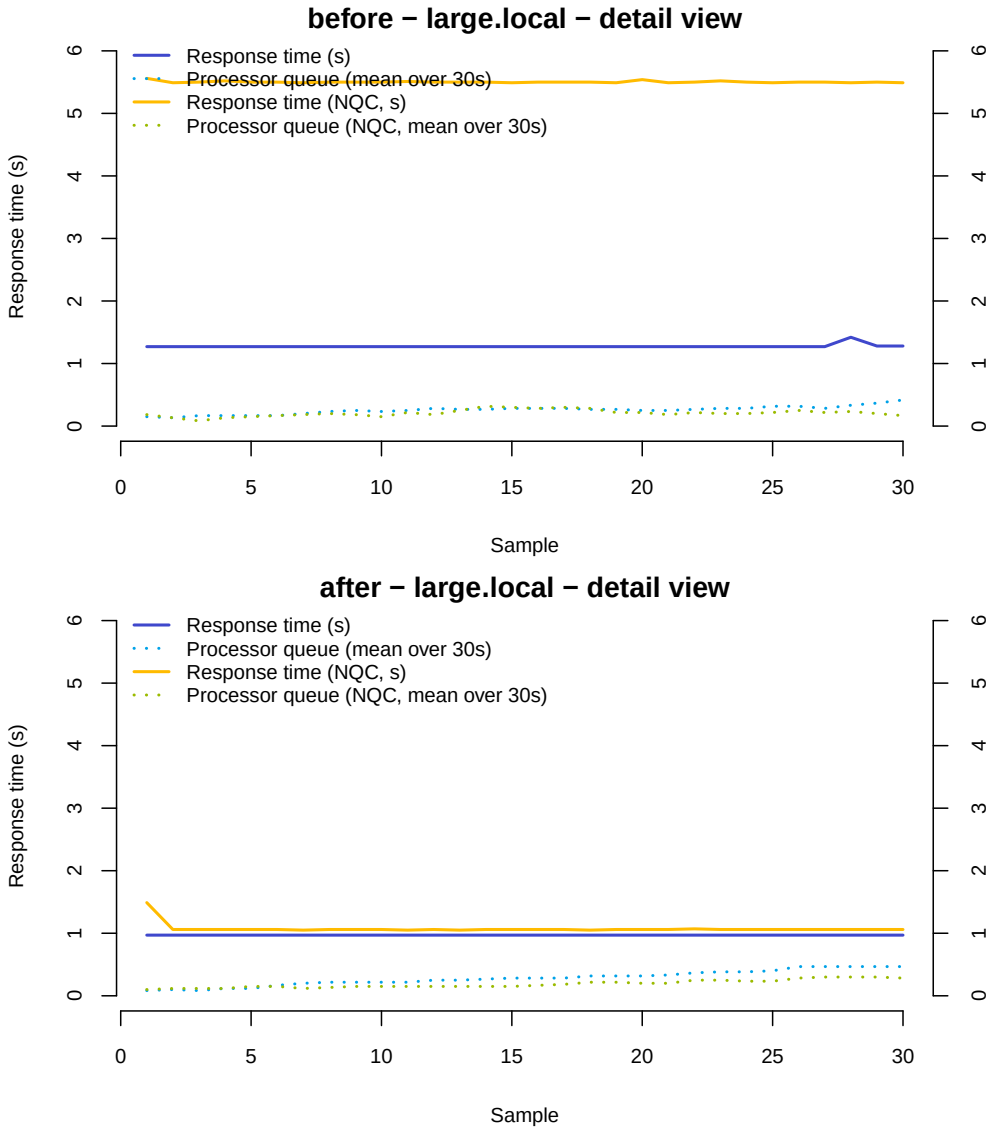


Figure C.6: A comparison of *large.local*'s detail page, before and after optimisation.

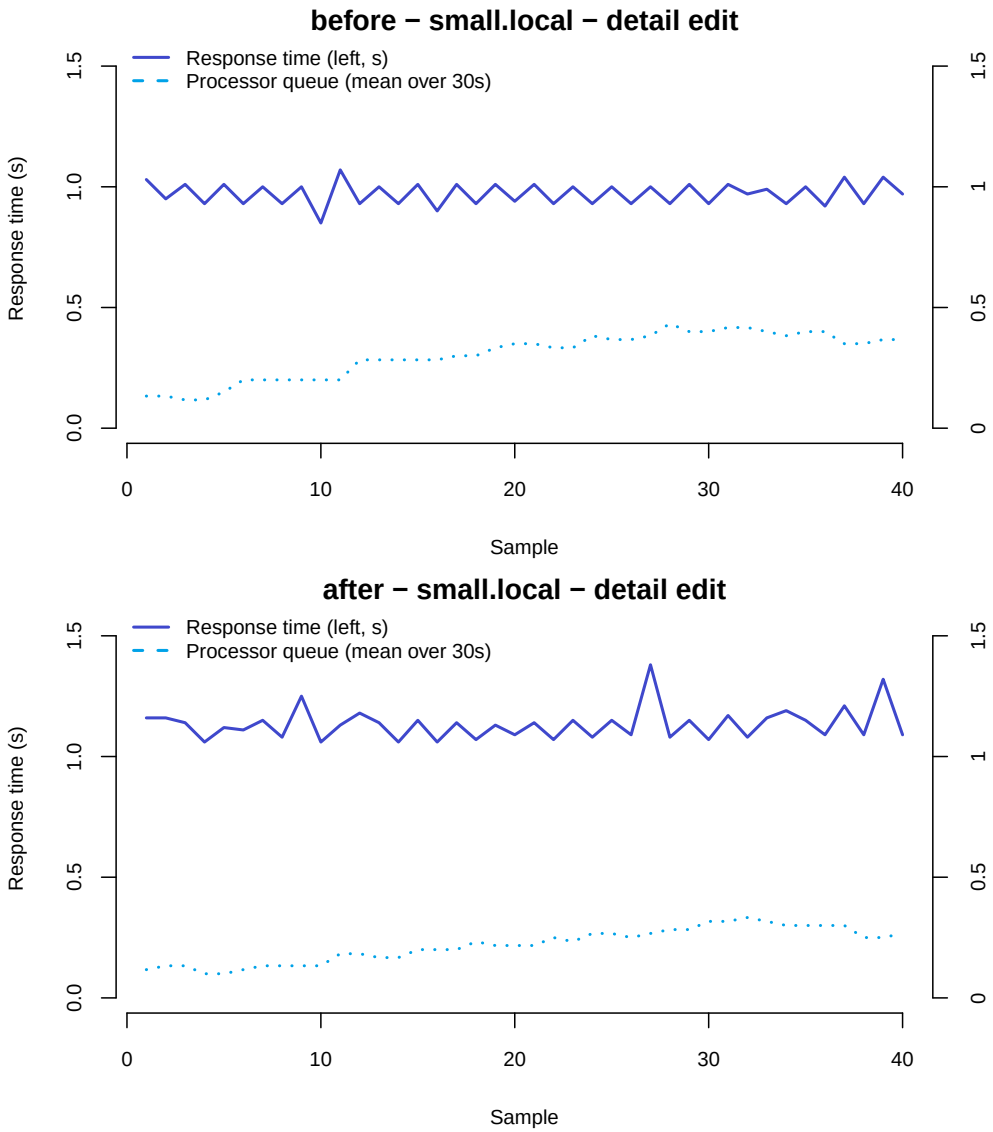


Figure C.7: A comparison of the editing of *small.local* content item, before and after optimisation. The content item is also connected to and disconnected from a node in the navigational structure. This causes the wobble.

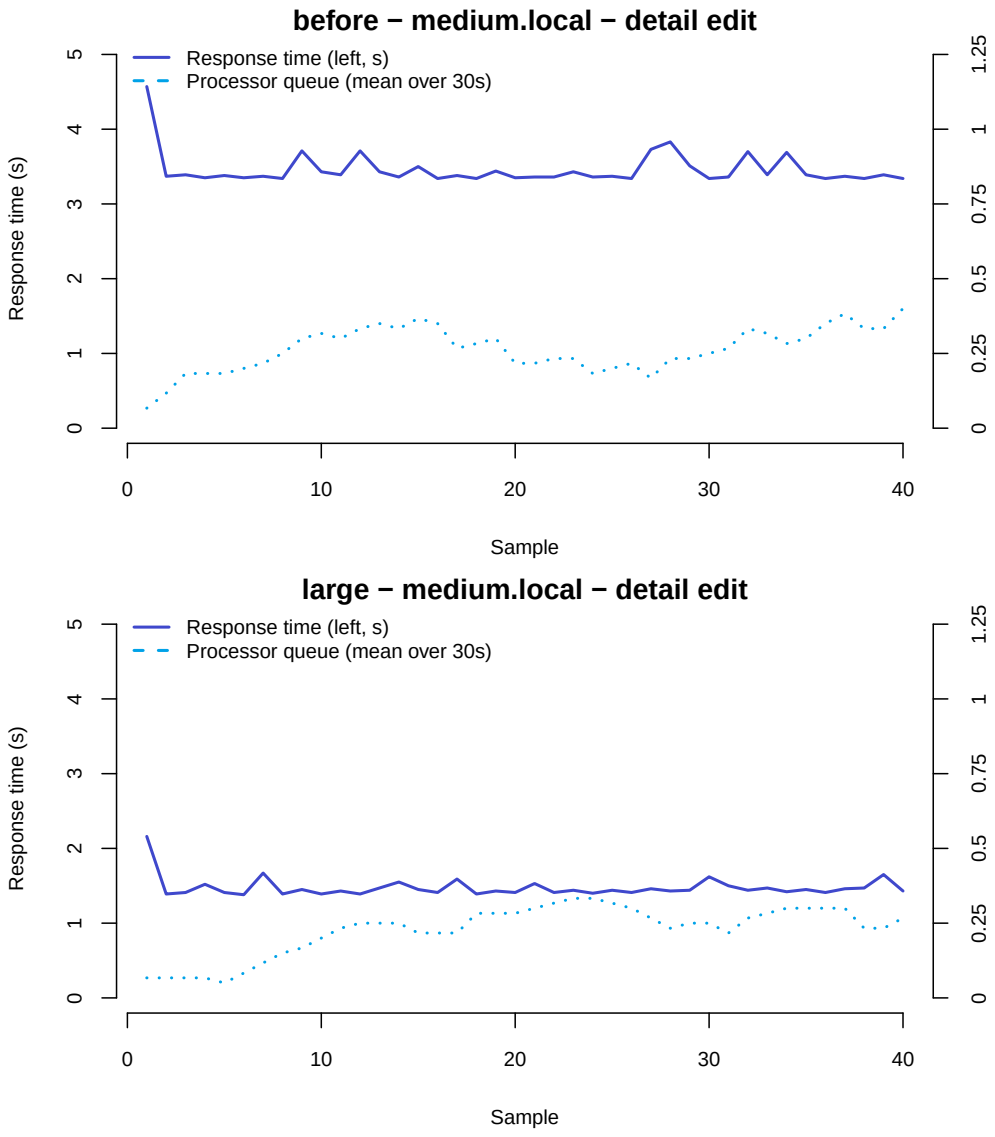


Figure C.8: A comparison of the editing of *medium.local* content item, before and after optimisation. The content item is also connected to and disconnected from a node in the navigational structure. This causes the wobble.

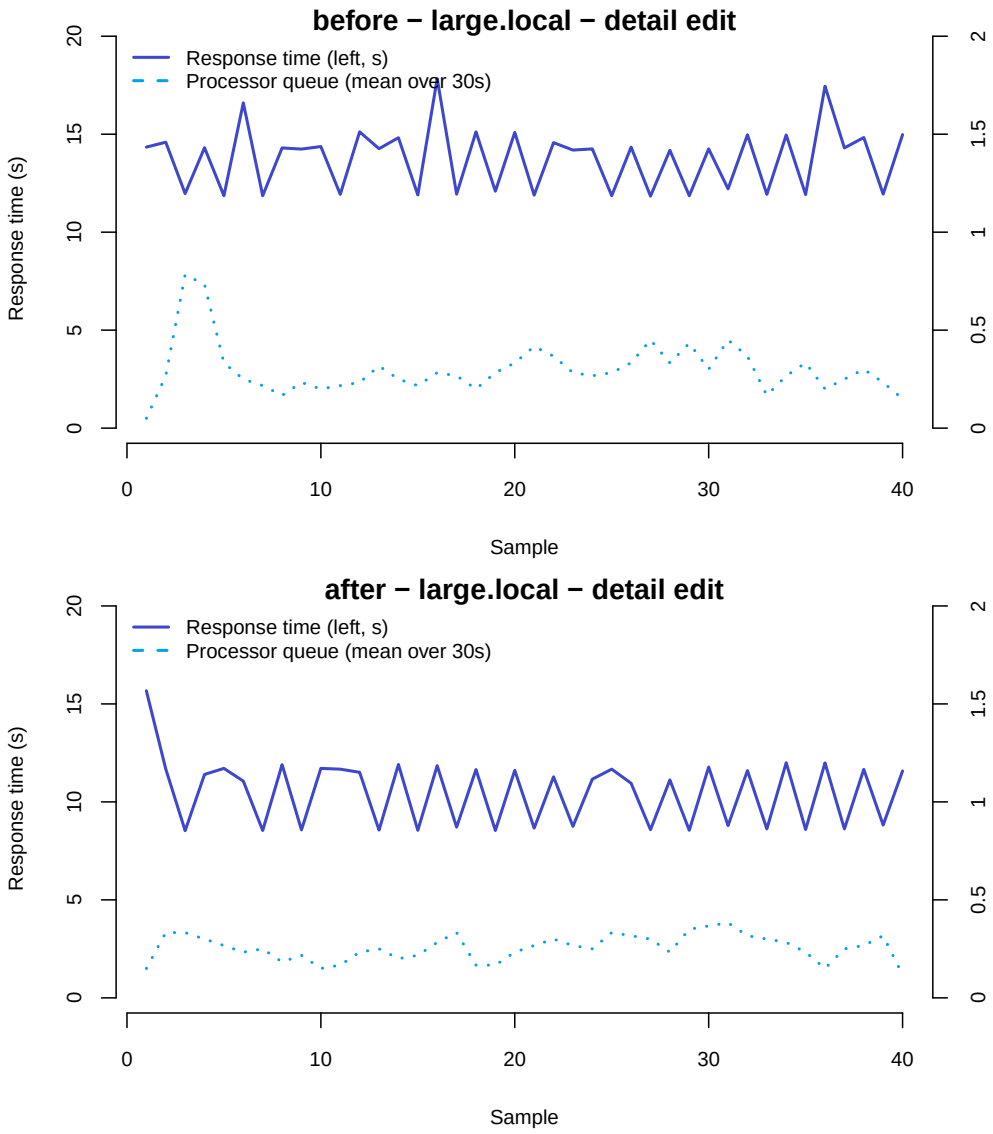


Figure C.9: A comparison of the editing of *large.local* content item, before and after optimisation. The content item is also connected to and disconnected from a node in the navigational structure. This causes the wobble.

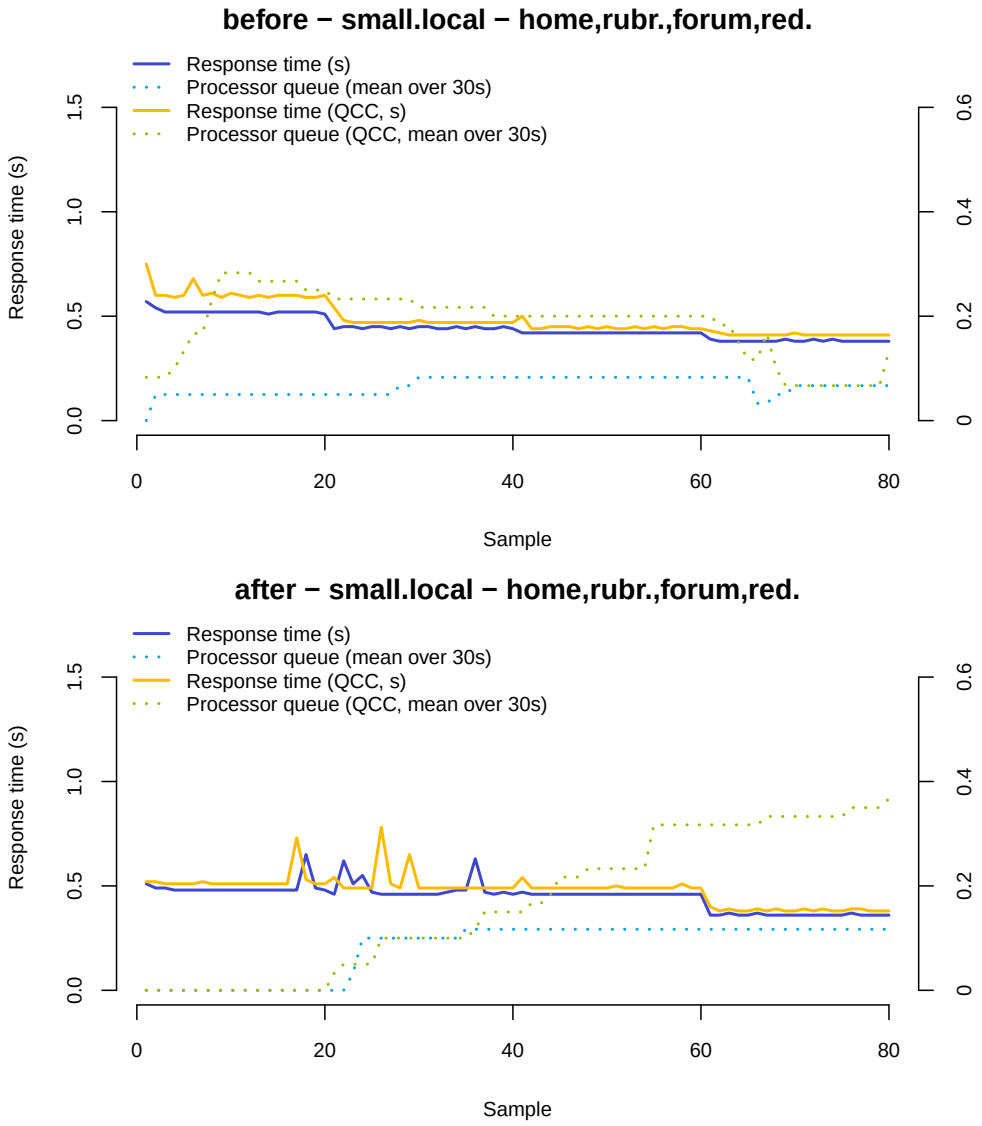


Figure C.10: Viewing of several pages from *small.local*, before and after optimisation.

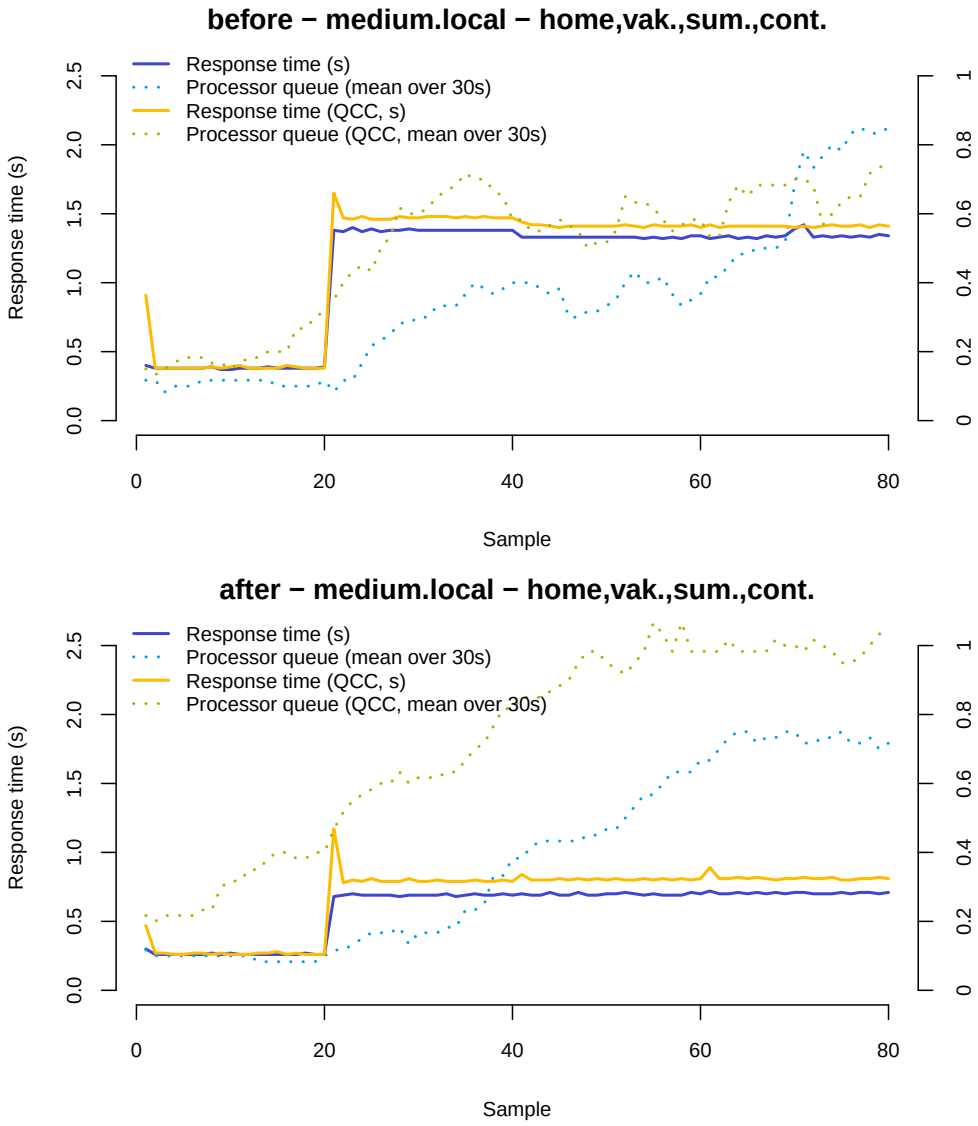


Figure C.11: Viewing of several pages from *medium.local*, before and after optimisation.

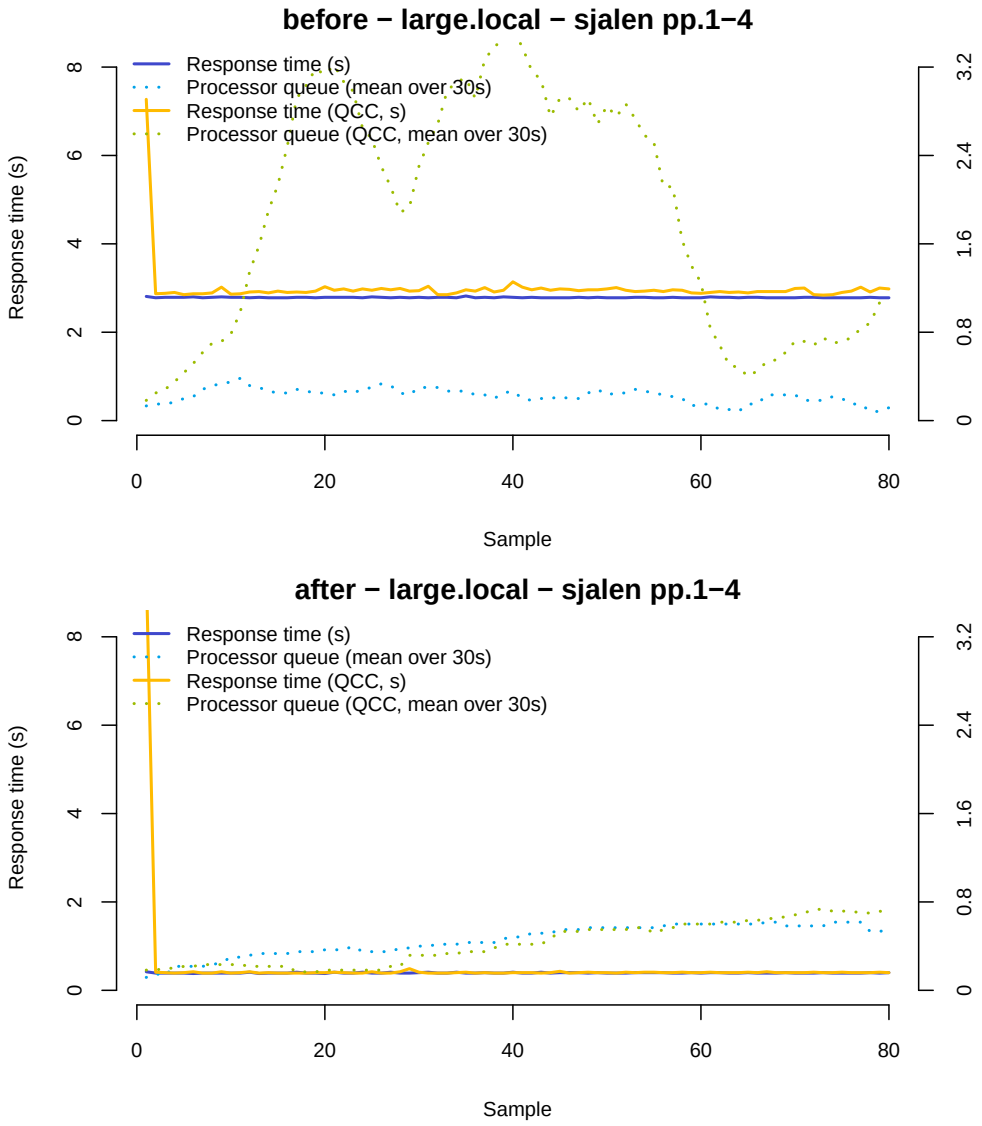


Figure C.12: Viewing of several pages from *large.local*, before and after optimisation.