1   Web Content Compression FAQ

1.1 Basics of Content Compression

Compression of outgoing traffic from web servers is beneficial for clients who get quicker responses, as well as for providers who experience less consumption of bandwidth.

Recently content compression for web servers has been provided mainly through use of the gzip format. Other (non perl) modules are available that provide so-called `deflate` compression. Both approaches are currently very similar and use the LZ77 algorithm combined with Huffman coding. Luckily for us, there is no real need to understand all the details of the obscure underlying mathematics in order to compress outbound content. Apache handlers available from CPAN can usually do the dirty work for us. Content compression is addressed through the proper configuration of appropriate handlers in the httpd.conf file.

Compression by its nature is a content filter: It always takes its input as plain ASCII data that it converts to another `binary` form and outputs the result to some destination. That's why every content compression handler usually belongs to a particular chain of handlers within the content generation phase of the request-processing flow.

A chain of handlers is one more common term that is good to know about when you plan to compress data. There are two of them recently developed for Apache 1.3.X: `Apache::OutputChain` and `Apache::Filter`. We have to keep in mind that the compression handler developed for one chain usually fails inside another.

Another important point deals with the order of execution of handlers in a particular chain. It's pretty straightforward in `Apache::Filter`. For example, when you configure

```
PerlModule Apache::Filter
<Files ~ "*\.blah">
  SetHandler perl-script
  PerlSetVar Filter On
  PerlHandler Filter1 Filter2 Filter3
</Files>
```

the content will go through `Filter1` first, then the result will be filtered by `Filter2`, and finally `Filter3` will be invoked to make the final changes in outgoing data.

However, when you configure

```
PerlModule Apache::OutputChain
PerlModule Apache::GzipChain
PerlModule Apache::SSIChain
PerlModule Apache::PassHtml
<Files *.html>
SetHandler perl-script
  PerlHandler Apache::OutputChain Apache::GzipChain Apache::SSIChain Apache::PassHtml
</Files>
```

execution begins with `Apache::PassHtml`. Then the content will be processed with `Apache::SSIChain` and finally with `Apache::GzipChain`. `Apache::OutputChain` will not be involved in content processing at all. It is there only for the purpose of joining other handlers within the chain.

It is important to remember that the content compression handler should always be the last executable handler in any chain.

Another important problem of practical implementation of web content compression deals with the fact that some buggy web clients declare the ability to receive and decompress gzipped data in their HTTP requests, but fail to keep their promises when an actual compressed response arrives. This problem is addressed through the implementation of the `Apache::CompressClientFixup` handler. This handler serves the `fixup` phase of the request-processing flow. It is compatible with all known compression handlers and is available from CPAN.

1.2 Q: Why it is important to compress web content?

1.2.1 A: Reduced equipment costs and the competitive advantage of dramatically faster page loads.

Web content compression noticeably increases delivery speed to clients and may allow providers to serve higher content volumes without increasing hardware expenditures. It visibly reduces actual content download time, a benefit most apparent to users of dialup and high-traffic connections.

1.3 Q: How much improvement can I expect?

1.3.1 A: Effective compression can achieve increases in transmission efficiency from 3 to 20 times.

The compression ratio is highly content-dependent. For example, if the compression algorithm is able to detect repeated patterns of characters, compression will be greater than if no such patterns exist. You can usually expect to realize an improvement between of 3 to 20 times on regular HTML, JavaScript, and other ASCII content. I have seen peak HTML file compression improvements in excess of more than 200 times, but such occurrences are infrequent. On the other hand I have never seen ratios of less than 2.5 times on text/HTML files. Image files normally employ their own compression techniques that reduce the advantage of further compression.

On May 21, 2002 Peter J. Cranstone wrote to the mod_gzip@lists.over.net mailing list:

"...With 98% of the world on a dial up modem, all they care about is how long it takes to download a page. It doesn't matter if it consumes a few more CPU cycles if the customer is happy. It's cheaper to buy a newer faster box, than it is to acquire new customers."

1.4 Q: How hard is it to implement content compression on an existing site?

1.4.1 A: Implementing content compression on an existing site typically involves no more than installing and configuring an appropriate Apache handler on the web server.

This approach works in most of the cases I have seen. In some special cases you will need to take extra care with respect to the global architecture of your web application, but such cases may generally be readily addressed through various techniques. To date I have found no fundamental barriers to practical implementation of web content compression.

1.5 Q: Does compression work with standard web browsers?

1.5.1 A: Yes. No client side changes or settings are required.

All modern browser makers claim to be able to handle compressed content and are able to decompress it on the fly, transparent to the user. There are some known bugs in some old browsers, but these can be taken into account through appropriate configuration of the web server.

I strongly recommend use of the `Apache::CompressClientFixup` handler in your server configuration in order to prevent compression for known buggy clients.

1.6 Q: What software is required on the server side?

1.6.1 A: There are four known mod_perl modules/packages for the web content compression available to date for Apache 1.3.X (in alphabetical order):

- **Apache::Compress**

  a mod_perl handler developed by Ken Williams (U.S.). `Apache::Compress` is capable to gzip output through `Apache::Filter`. This module accumulates all incoming data and then compresses the whole content body at once.

- **Apache::Dynagzip**

  a mod_perl handler, developed by Slava Bizyayev -- a Russian programmer residing in the U.S. `Apache::Dynagzip` uses the gzip format to compress output through the `Apache::Filter` or through the internal Unix pipe.

  `Apache::Dynagzip` is most useful when one needs to compress dynamic outbound web content (generated on the fly from databases, XML, etc.) when content length is not known at the time of the request.

`Apache::Dynagzip`'s features include:

- ○ **Support for both HTTP/1.0 and HTTP/1.1.**
- ○ **Control over the chunk size on HTTP/1.1 for on-the-fly content compression.**
- ○ **Support for Perl, Java, or C/C++ CGI applications.**
- ○ **Advanced control over the proxy cache with the configurable `Vary` HTTP header.**
- ○ **Optional control over content lifetime in the client's local cache with the configurable `Expires` HTTP header.**
- ○ **Optional support for server-side caching of the dynamically generated (and compressed) content.**
- ○ **Optional extra-light compression**

  removal of leading blank spaces and/or blank lines, which works for all browsers, including older ones that cannot uncompress gzip format.

- • **Apache::Gzip**

  an example of mod_perl filter developed by Lincoln Stein and Doug MacEachern for their book *Writing Apache Modules with Perl and C* (U.S.), which like `Apache::Compress` works through `Apache::Filter`. `Apache::Gzip` is not available from CPAN. The source code may be found on the book's companion web site at http://www.modperl.com/

- • **Apache::GzipChain**

  a mod_perl handler developed by Andreas Koenig (Germany), which compresses output through `Apache::OutputChain` using the gzip format.

  `Apache::GzipChain` currently provides in-memory compression only. Using this module under `perl-5.8` or higher is appropriate for Unicode data. UTF-8 data passed to `Compress::Zlib::memGzip()` are converted to raw UTF-8 before compression takes place. Other data are simply passed through.

1.7 Q: Is it possible to compress the output from `Apache::Registry` with `Apache::Dynagzip`?

1.7.1 A: Yes, it is supposed to be pretty easy:

If your page/application is initially configured like

```
<Directory /path/to/subdirectory>
  SetHandler perl-script
  PerlHandler Apache::Registry
  PerlSendHeader On
  Options +ExecCGI
</Directory>
```

you might want just to replace it with the following:

```
PerlModule Apache::Filter
PerlModule Apache::Dynagzip
PerlModule Apache::CompressClientFixup
<Directory /path/to/subdirectory>
  SetHandler perl-script
  PerlHandler Apache::RegistryFilter Apache::Dynagzip
  PerlSendHeader On
  Options +ExecCGI
  PerlSetVar Filter On
  PerlFixupHandler Apache::CompressClientFixup
  PerlSetVar LightCompression On
</Directory>
```

You should be all set usually after that.

In more common cases you need to replace the line

```
    PerlHandler Apache::Registry
```

in your initial configuration file with the set of the following lines:

```
    PerlHandler Apache::RegistryFilter Apache::Dynagzip
    PerlSetVar Filter On
    PerlFixupHandler Apache::CompressClientFixup
```

You might want to add optionally

```
    PerlSetVar LightCompression On
```

to reduce the size of the stream even for clients incapable to speak gzip (like *Microsoft Internet Explorer* over HTTP/1.0).

Finally, make sure you have somewhere declared

```
  PerlModule Apache::Filter
  PerlModule Apache::Dynagzip
  PerlModule Apache::CompressClientFixup
```

This basic configuration uses many defaults. See `Apache::Dynagzip` POD for further thin tuning if required.

1.8 Q: Is it possible to compress the output from Mason-driven application with `Apache::Dynagzip`?

1.8.1 A: Yes. `HTML::Mason::ApacheHandler` is compatible with `Apache::Filter` chain.

If your application is initially configured like

```
PerlModule HTML::Mason::ApacheHandler
<Directory /path/to/subdirectory>
  <FilesMatch "\.html$">
    SetHandler perl-script
    PerlHandler HTML::Mason::ApacheHandler
  </FilesMatch>
</Directory>
```

you might want just to replace it with the following:

```
PerlModule HTML::Mason::ApacheHandler
PerlModule Apache::Dynagzip
PerlModule Apache::CompressClientFixup
<Directory /path/to/subdirectory>
  <FilesMatch "\.html$">
    SetHandler perl-script
    PerlHandler HTML::Mason::ApacheHandler Apache::Dynagzip
    PerlSetVar Filter On
    PerlFixupHandler Apache::CompressClientFixup
    PerlSetVar LightCompression On
  </FilesMatch>
</Directory>
```

You should be all set safely after that.

In more common cases you need to replace the line

```
    PerlHandler HTML::Mason::ApacheHandler
```

in your initial configuration file with the set of the following lines:

```
    PerlHandler HTML::Mason::ApacheHandler Apache::Dynagzip
    PerlSetVar Filter On
    PerlFixupHandler Apache::CompressClientFixup
```

You might want to add optionally

```
    PerlSetVar LightCompression On
```

to reduce the size of the stream even for clients incapable to speak gzip (like *Microsoft Internet Explorer* over HTTP/1.0).

Finally, make sure you have somewhere declared

```
  PerlModule Apache::Dynagzip
  PerlModule Apache::CompressClientFixup
```

This basic configuration uses many defaults. See `Apache::Dynagzip` POD for further thin tuning.

1.9 Q: Why is it important to keep control over chunk size?

1.9.1 A: It helps to reduce the latency of the response.

`Apache::Dynagzip` is the only handler to date that begins transmission of compressed data as soon as the initial uncompressed pieces of data arrive from their source, at a time when the source process may not even have completed generating the full document it is sending. Transmission can therefore be taking place concurrent with creation of later document content.

This feature is mainly beneficial for HTTP/1.1 requests, because HTTP/1.0 does not support chunks.

I would also mention that the internal buffer in `Apache::Dynagzip` always prevents Apache from the creating too short chunks over HTTP/1.1, or from transmitting too short pieces of data over HTTP/1.0.

1.10 Q: Are there any content compression solutions for vanilla Apache 1.3.X?

1.10.1 A: Yes, There are two compression modules written in C that are available for vanilla Apache 1.3.X:

- **mod_deflate**

  an Apache handler written in C by Igor Sysoev (Russia).

- **mod_gzip**

  an Apache handler written in C. Original author: Kevin Kiley, *Remote Communications, Inc.* (U.S.)

Both of these modules support HTTP/1.0 only.

1.11 Q: Can I compress the output of my site at the application level?

1.11.1 A: Yes, if your web server is CGI/1.1 compatible and allows you to create specific HTTP headers from your application, or when you use an application framework that carries its own handler capable of compressing outbound data.

For example, vanilla Apache 1.3.X is CGI/1.1 compatible. It allows development of CGI scripts/programs that might be generating compressed outgoing streams accomplished with specific HTTP headers.

Alternatively, on mod_perl enabled Apache some application environments carry their own compression code that could be activated through the appropriate configurations:

`Apache::ASP` does this with the `CompressGzip` setting;

`Apache::AxKit` uses the `AxGzipOutput` setting to do this.

See particular package documentation for details.

1.12 Q: Are there any content compression solutions for Apache-2?

1.12.1 A: Yes, a core compression module written in C, *mod_deflate*, has recently become available for Apache-2.

`mod_deflate` for Apache-2 is written by Ian Holsman (USA).

This module supports HTTP/1.1 and is filters compatible.

Despite its name `mod_deflate` for Apache-2 provides `gzip`-encoded content. It contains a set of configuration options sufficient to keep control over all recently known buggy web clients.

1.13 Q: When `Apache::Dynagzip` is supposed to be ported to Apache-2?

1.13.1 A: There no recent plans to port `Apache::Dynagzip` to Apache-2:

`mod_deflate` for Apache-2 seems to be capable to provide all basic functionality required for dynamic content compression:

- **This module supports flushing over HTTP/1.1**
- **It is filters compatible.**
- **It has a set of configuration options to keep control over the buggy clients.**

The rest of the main `Apache::Dynagzip` options could be easily addressed through the implementation of pretty tiny and specific accomplishing filters.

1.14 Q: Where can I read the original descriptions of `gzip` and `deflate` formats?

1.14.1 A: `gzip` format is published as rfc1952, and `deflate` format is published as rfc1951.

You can find many mirrors of RFC archives on the Internet. Try, for instance, my favorite at http://www.ietf.org/rfc.html

1.15 Q: Are there any known compression problems with specific browsers?

1.15.1 A: Yes, Netscape 4 has problems with compressed cascading style sheets and JavaScript files.

You can use `Apache::CompressClientFixup` to disable compression for these files dynamically. `Apache::Dynagzip` is capable of providing so-called `light compression` for these files.

1.16 Q: Where can I find more information about the compression features of modern browsers?

1.16.1 A: Michael Schroepl maintains a highly valuable site

Try it at http://www.schroepl.net/projekte/mod_gzip/browser.htm

1.17 Acknowledgments

I highly appreciate efforts of Dan Hansen done in order to make this text better English...

1.18 Maintainers

The maintainer is the person you should contact with updates, corrections and patches.

- Slava Bizyayev <slava (at) cpan.org>

1.19 Authors

- Slava Bizyayev <slava (at) cpan.org>

Only the major authors are listed above. For contributors see the Changes file.

Table of Contents:

Table of Contents: