Building Tag Clouds in Perl and PHP

Tag clouds are everywhere on the Web these days. First popularized by the websites Flickr, Technorati, and del.icio.us, these amorphous clumps of words now appear on a slew of websites as visual evidence of their membership in the elite corps of "Web 2.0."

This article analyzes what is and isn't a tag cloud, offers design tips for using them effectively, and then goes on to show how to collect tags and display them in the tag cloud format. Scripts are provided in Perl and PHP.

Yes, some have said tag clouds are a fad. But as you will see, tag clouds, when used properly, have real merits. More importantly, the skills you learn in making your own tag clouds enable you to make other interesting kinds of interfaces that will outlast the mercurial fads of this year or the next.

Tag Clouds: Ephemeral or Enduring?

If you're reading this, you've probably seen a tag cloud (Figure 1)—they are suddenly everywhere on the Web these days. In this article, I'm going to provide a little analysis and history of tag clouds, and then get on to more important matters: I'll demonstrate how to make your own tag clouds in Perl and PHP.

Tag clouds are a current fashion. But in April of 1995, web design guru Jeffrey Zeldman decried their faddishness in his headline, "Tag Clouds Are the New Mullets," comparing them to the once popular haircut that has become a fashion joke. And this was before they *really* started to catch on.

But jaded criticism is a common side effect of sudden ubiquity, and Zeldman also praised the brilliance of the idea. And as I will show, tag clouds, when used properly, have real, lasting merits.

Note

All of the scripts in this article can be downloaded from O'Reilly's website at the following URL: *www.oreilly.com/catalog/tagclouds/scripts/index.html*

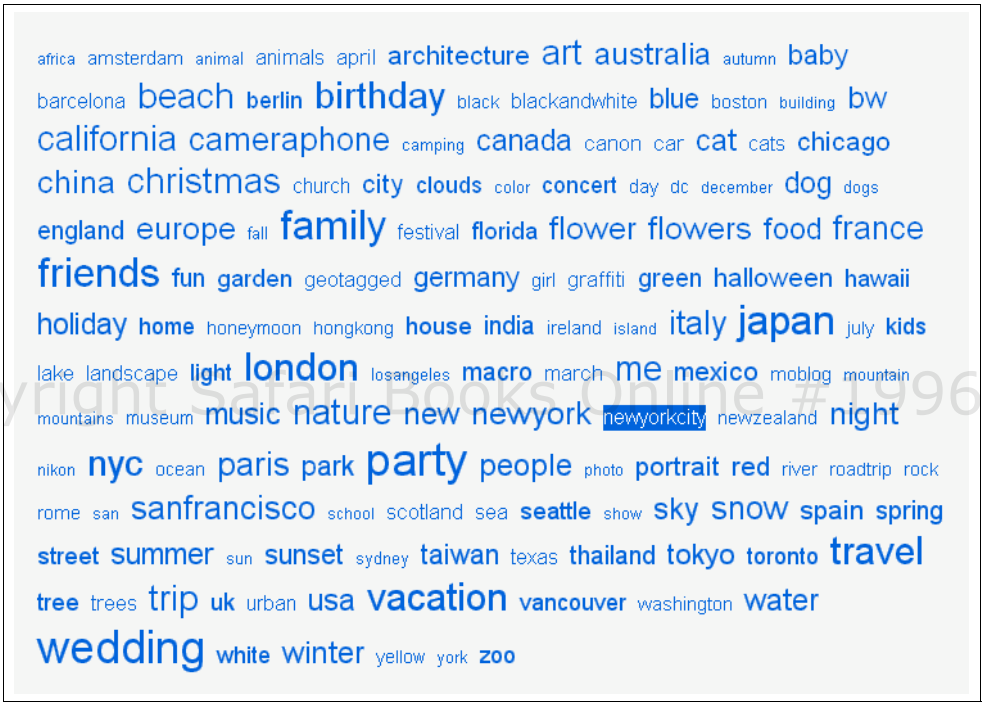


Figure 1. A tag cloud from Flickr

Weighted Lists

So what is a tag cloud? A tag cloud is a specific kind of *weighted list*. For lack of a standard working definition of weighted list, I'm going to make one up.

Weighted list

n. A list of words or phrases, in which one or more visual features in the list (such as font size) are correlated to some underlying data.

While tag clouds are a specific type of weighted list, not all weighted lists are tag clouds. For example, the list of cities at the popular craigslist website (Figure 2) is a weighted list because font size is correlated with popularity, but it lacks the random appearance of a tag cloud, due to the arrangement of the cities in a matrix.

Building Tag Clouds in Perl and PHP
Building Tag Clouds in Perl and PHP By Jim Bumgardner ISBN: 0596527942 Publisher: O'Reilly
Print Publication Date: 5/1/2006

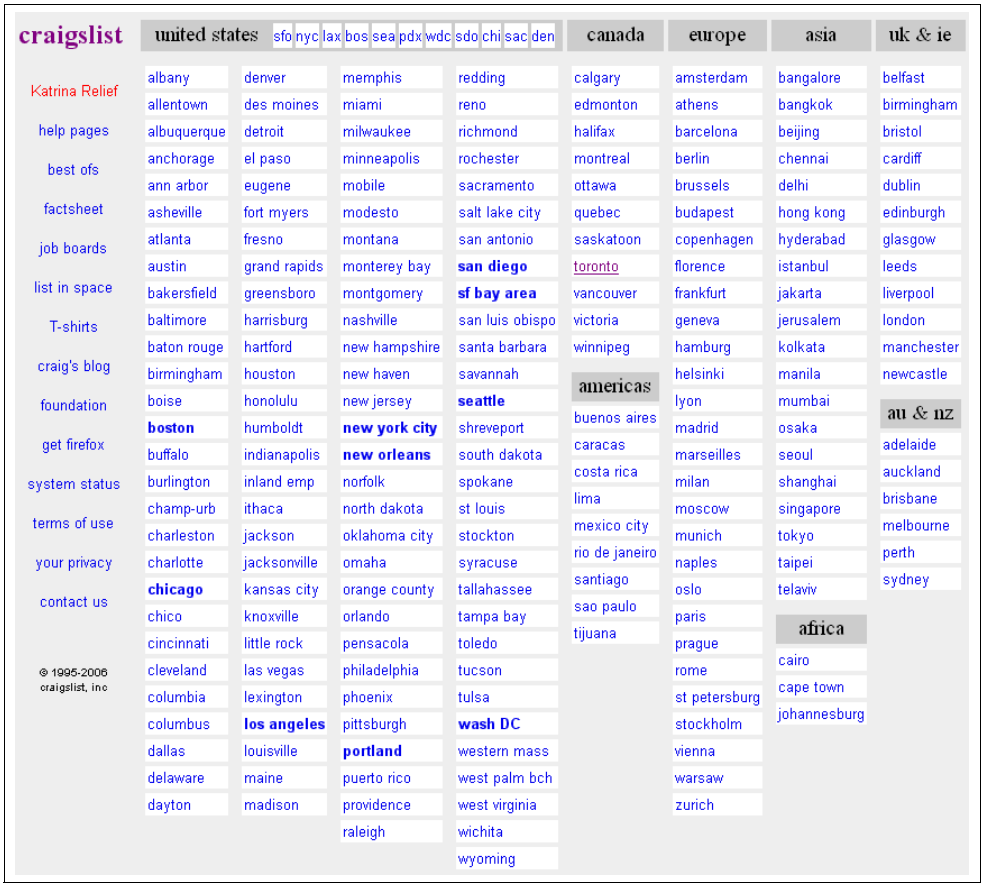


Figure 2. Weighted cities list from craigslist

Another kind of weighted list, one that's even more distant from tag clouds, is that of the statistically improbable phrases (SIPs) and capitalized phrases (CAPs) lists provided by Amazon.com (Figure 3). In the SIP list, word order correlates to the improbability of the phrase, and in the CAP list, to the frequency with which the phrase appears in the book.

Figure 3. Weighted phrase lists from Amazon.com

Creating Weighted Lists

There are lots of ways to make weighted lists. Given any list of words or phrases, there are a handful of visual features that you can choose to correlate with underlying data:

A: Visual Features

- Font size
- Word order
- Word color
- Word shape (typeface and style)

The kinds of underlying data you might correlate or map these features to is a much larger list:

B: Underlying Data

- Quantity
- Lexical order
- Subject
- Location
- Time

To make a weighted list, take one of the items from column A and correlate it to one of the items in column B (and repeat, if you like, with different items).

Tag clouds are just one kind of weighted list. There are many different implementations of tag clouds, and they do not all share the same mappings, but almost all of them tend to associate font size with quantity. For example, the weighted lists at Flickr have the following mappings:

A. Visual Features	**B. Underlying Data**
Font size	Quantity
Word order	Lexical order
Word color	Blue
Word shape	Sans serif

Weighted lists on other websites differ in varying degrees from Flickr's basic design, but the more closely they follow it, the more likely they will be described as "tag clouds" rather than as "weighted lists" or "lists." The tag cloud on the website 43 Things (Figure 4) has the following mappings:

A. Visual Features	**B. Underlying Data**
Font size	Quantity
Word order	Random
Word color	Black with beige background
Type face	Sans serif

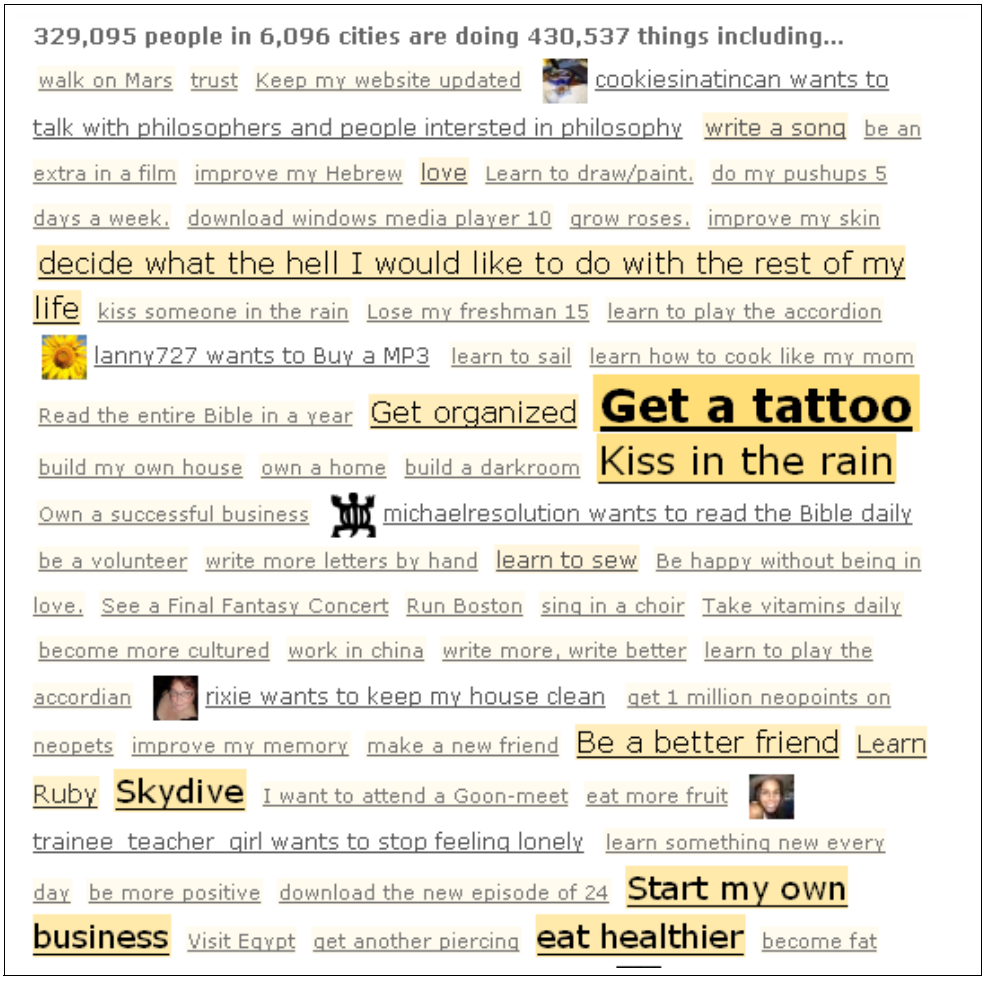


Figure 4. Tag cloud for 43 Things

Tag Cloud Properties

Tag clouds generally have the following additional properties:

- The words are arranged in a continuous list, rather than a table. The order of the words is uncorrelated to tag frequency; for example, they might be listed alphabetically or randomly.
- The words represent tags, or community-created metadata. This metadata often follows power laws—there are few popular items, and many more unpopular items.

- The tags are links navigable to the tagged content.

The first two properties give tag clouds their cloudy or amorphous appearance. They have a simple beauty that is more attractive than a grid.

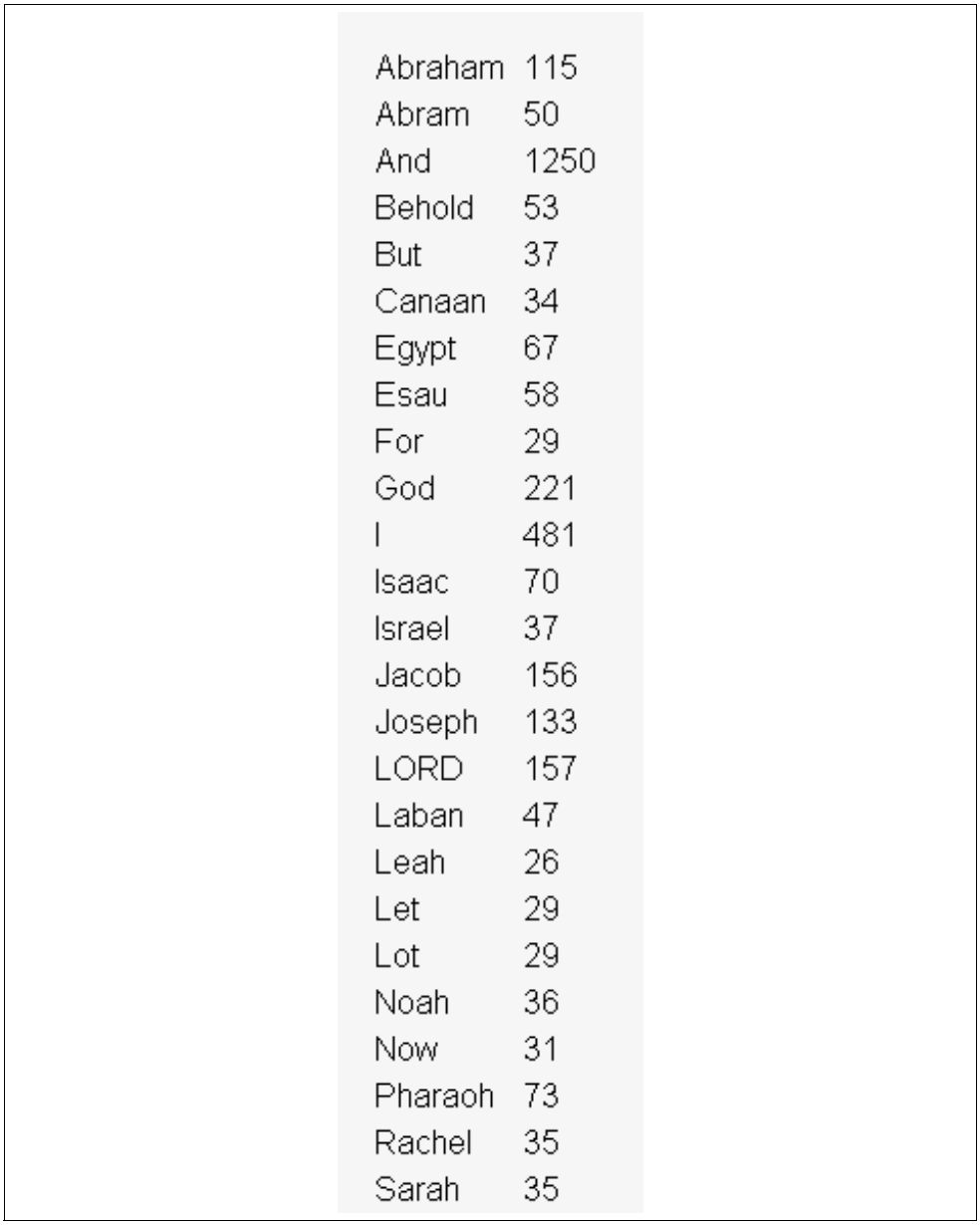
The second two properties give tag clouds a dual function. They function not only as a graph of interesting data, but are a navigation interface to user-generated content (or what Derek Powazek calls “authentic media”). In other words, tag clouds are both something to look at and something to click on.

While you can click on tag clouds, you can also just look at them to get a quick reading of a website’s zeitgeist. Looking at the Flickr tag cloud in Figure 1, you can see that wedding photos are to be found in large quantities, and that they have a lot of photos taken in London and Japan (perhaps at weddings?). Looking at 43 Things (Figure 4), you can see that a lot of people want to get a tattoo. The list at 43 Things is a randomized selection from a much larger list, so if you refresh the page you’ll get different winners such as “buy a house,” “write a book,” and “be happy.”

Tag Clouds as Metadat

The dual nature of tag clouds comes at the expense of a design trade-off. There are more effective ways to navigate. In general, “browsing” interfaces are not as efficient for finding stuff as searching (and tag clouds are usually accompanied by a standard issue search box, which sees more use). But browsing and searching are two different activities that serve different needs. The dynamic way that tag clouds show popular lists is a remarkably effective way to browse.

There are also more accurate ways to graph tag popularity. Consider the following lists, which show the most common words in the book of Genesis. You could provide tags in a table with actual numbers (Figure 5a), or in a bar graph (Figure 5b).



Abraham	115
Abram	50
And	1250
Behold	53
But	37
Canaan	34
Egypt	67
Esau	58
For	29
God	221
I	481
Isaac	70
Israel	37
Jacob	156
Joseph	133
LORD	157
Laban	47
Leah	26
Let	29
Lot	29
Noah	36
Now	31
Pharaoh	73
Rachel	35
Sarah	35

Figure 5a. Word frequency list

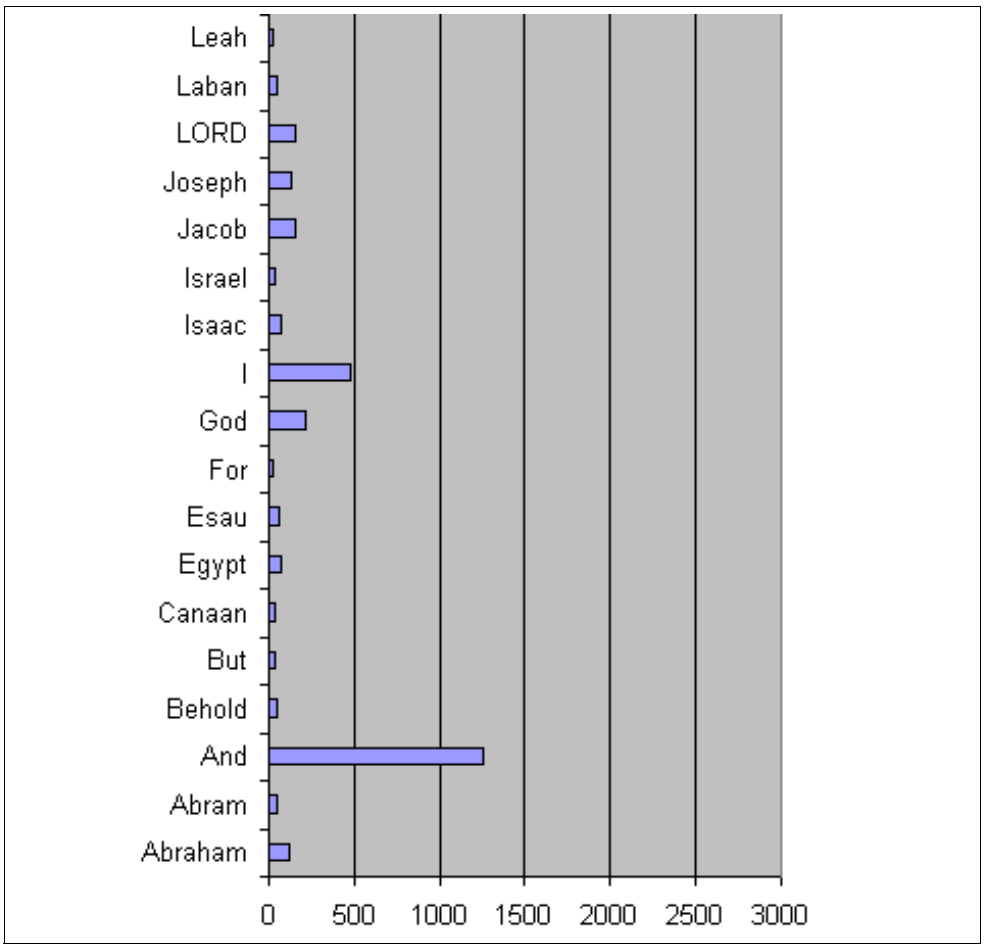


Figure 5b. Word frequency bar graph

These methods both provide an unnecessary increase in accuracy at the expense of a great loss in visual real estate (especially the bar graph!). Unless you're into Biblical numerology, you don't really need to know that the name "Esau" is mentioned exactly 58 times. You just want to get a general sense of what is popular or frequent. Because tag clouds use the words themselves to describe the data (Figure 5c), they can provide the essential information for a larger number of words in a much smaller space.

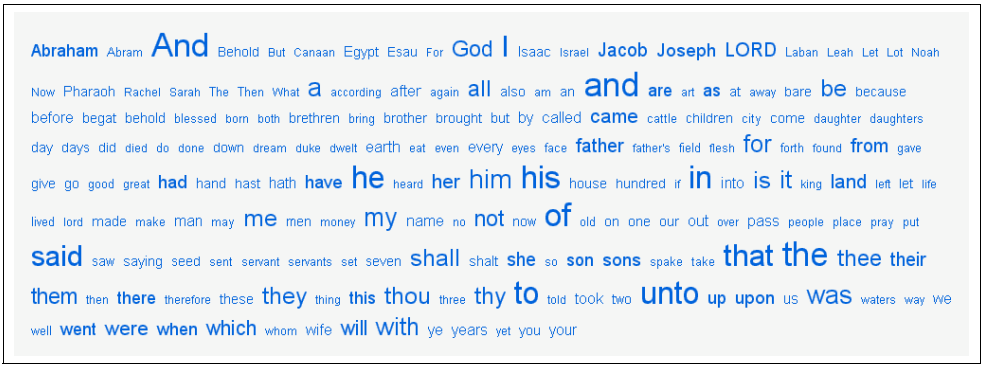


Figure 5c. Word frequency tag cloud

If my own experience is typical, tag clouds are more frequently read than clicked on. Generally, I only click on tag clouds when they correspond very closely to my actual interests at the time. However, their function as a measurement of zeitgeist is quite useful by itself.

Tag clouds have another, less obvious function, along with being something to look at and something to click on: they effectively describe the nature of a website to search engines like Google. In static websites, people use the `<meta description>` and `<meta keywords>` tags to describe the content of the website to search engines. But in sites like Flickr, which consist primarily of user-generated content, you can't predict what the principal themes will be tomorrow or next month. Tag clouds solve this problem by providing a running meter of the important items on a site. Thus, they can dynamically boost search-engine rankings for those tags. And if the search engine pays attention to font size (and some of them do), so much the better!

Some History

Although Flickr was the first website to use something called a tag cloud, the idea was likely inspired (directly or indirectly) from an older blog plugin called Zeitgeist (Figure 6), by Jim Flanagan. Jim provided this story when I asked him about it:

In 1997, when I was working at Brookhaven National Lab in Long Island NY, the Web was becoming popular enough so that everybody had to have a web page, and I wanted somehow to rebel against the canonical, hierarchical bulleted list of links. So I wrote a Perl CGI that would take a small database of links and present them on the page in varying colors and sizes. The color and size were selected randomly so different things would cycle into your attention each time you loaded the page.

Much later, when I got into blogging, I fell into the narcissistic practice of checking my blog referral logs to see what was linking to me. I developed several personal "narcissurfing" tools, and noticed that the Google and Yahoo searches that led to my site were often very amusing. In an attempt to build a page to share the search information with my readers, I fell back to the random-colored links approach, except that this time, the number of hits from a certain search term controlled the size.

After a while, several bloggers asked for the code, and I cleaned it up a bit and sent it along. It's still available at http://jimfl.tensegrity.net/zeitcode.

Many bloggers use the word "zeitgeist" to mean a weighted word list in the style of Jim's plugin, as in Figure 6.

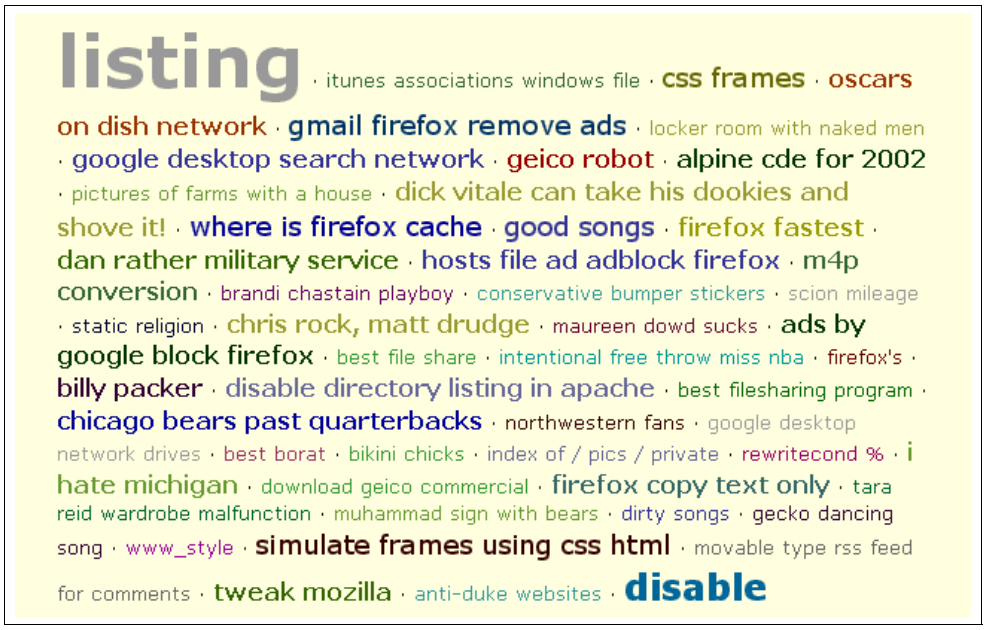


Figure 6. Jim Flanagan's Zeitgeist plugin in action

If you look at Jim's code (or Figure 6), you'll see that it has the following mappings:

A. Visual Features	B. Underlying Data
Font size	Quantity
Word order	Random
Word color	Random
Type face	Default

The team at Flickr (Stewart Butterfield, Cal Henderson, and George Oates) implemented the first tag clouds at Flickr using Zeitgeist-like weighted lists as inspiration. The Flickr tag clouds have a few fundamental differences from the word lists produced by Zeitgeist:

- They represent tags rather than search engine phrases, so the data being shown is actively generated by the site's community within the site, rather than gathered from the site's server logs.
- They do not use random word order (although many other tag clouds do). The alphabetical word order provides an additional way to browse the list, while still giving the list a random appearance.

Flickr also gave their tag clouds a more polished design that many other sites have emulated. They chose an attractive font, a single color (rather than a random assortment of colors, which adds visual complexity but no additional information), and they kept the lists of words relatively short, rather than allowing them to go on for pages and pages, as many Zeitgeist-based pages do.

Clearly, one of the reasons for the success of tag clouds is their origin in the blogging community. Bloggers have both a need to organize the large amounts of material they constantly churn out and an excellent communications medium to propagate new and interesting methods.

Design Tips for Building Tag Clouds

Here are some design suggestions for creating tag clouds.

Choose the Right Language

I like to write code in lots of different languages, and I believe in choosing the right language for a particular job (rather than using any one language for all jobs). I think higher-level scripting languages like Perl, PHP, Python, and Ruby are all good choices for making tag clouds. They tend to be supported on servers and they

have associative lists (which make counting tags much easier). Lower-level languages that don't support associative arrays (such as C++ or Java) are not as good for implementing tag clouds, because you will end up writing considerably more code.

Make Your Tag Clouds Visible to Search Engines

You can make tag clouds fairly easily in Flash/ActionScript and JavaScript, and you can make them look much snazzier—flashier, even. However, I don't think these client-side languages are as good a choice as the server-based scripting languages. Why? Because you want search engines to see your tag clouds. Both of these technologies would effectively blind most search engines to the content of your tag clouds. If you do pursue a Flash or JavaScript solution for the interface, consider including the actual tags themselves in a comment block in the HTML.

Frequency Sorting

You can, if you like, sort tag clouds by word frequency (Figure 17). Personally, I don't think it's a good idea. Not only does it reduce the "cloudy" nature of the word list, but it denies your users an additional organizational axis. Most tag clouds use either random or alphabetic sorts. I prefer the alphabetic sort because it provides a quick way to eliminate or identify a particular tag.

Make Tag Clouds Relevant to the User

Tag clouds are best when they are relevant to the user's particular interests. For example, a tag cloud that shows popular tags of the last few days is likely to be more interesting (to the breathing) than a tag cloud that shows popular tags "of all time." Also, if you filter for recent activity, the content of your tag clouds will change every day, rather than remaining static.

Tag clouds can also be used to accompany and annotate search results. When a user searches for a particular tag, you can display a cloud showing related tags. This result will be much more interesting to the user than a tag cloud showing only the most popular tags on the server.

Try Different Mappings

Tag clouds are only one, specific kind of weighted list. There are many kinds of mappings from visual features to underlying data that have not yet been exploited. How about trying some weighted lists that don't look like common tag clouds? For example, you could map font size to time, showing more recent tags in large sizes. Or, in a historical database, you could map font to decade or century, using progressively older-fashioned fonts for older data.

Making Tag Clouds in Perl

Now I'll show you how to make a Flickr-style tag cloud in Perl. In order to run these scripts, you'll need the following four CPAN modules installed on your system:

- LWP::Simple, which provides a `get()` function that retrieves the contents of a web page and stores it in a text string.
- HTTP::Cache::Transparent, which provides a caching mechanism that speeds up your scripts (the second time you run them) and reduces the load on servers that you are querying for tags.
- XML::RSSLite, which is an RSS parser—one of many such parsers on CPAN. We'll use it to parse the RSS feed at del.icio.us. I chose RSSLite because code that uses it is relatively easy to read, compared to some other parsers. However, if you already prefer another parser, then by all means use it.
- Data::Dumper, which produces a Perl listing of any Perl data structure. I use it all the time to save data to files for later use. It is also incredibly helpful for examining and understanding the contents of complex data structures (such as XML trees and the data returned by RSS parsers).

There are three key steps to making a tag cloud:

1. Make tags
2. Collect tags
3. Display tags

Most of this article is concerned with the last two tasks: collecting tags and displaying them in the form of a tag cloud. I will assume that you have a source of tags or phrases. But to begin with, we do need some raw data, so the scripts that count the tags will use websites that provide data we can use to build tag clouds.

Collecting Tags

When developing a general-purpose script, it's a good idea to work with at least two very different sets of data so you can get a better idea of what kinds of challenges you might encounter. I am providing scripts that retrieve data from two very different sources—one very old and one very new. Both scripts collect the data and then use Data::Dumper to save the data to a file. The file contains a data structure called `$tags` that contains the set of tags, the tags' associated counts, and associated URLs. The contents of this file are intended to be read by another Perl script, and it is formatted as valid Perl. Here's a sample:

```
$tags = {
  'RSS' => {
    'count' => 1,
    'url' => 'http://magpierss.sourceforge.net/',
    'tag' => 'RSS'
    },
  'NYQUIST' => {
    'count' => 1,
    'url' => 'http://audacity.sourceforge.net/help/nyquist3',
    'tag' => 'Nyquist'
    },
  'GENERATORS' => {
    'count' => 1,
    'url' => 'http://generatorblog.blogspot.com/',
    'tag' => 'generators'
    },
 # etc…
               };
             1;
```

Notice that each tag has both a key (uppercase) and a tag value (mixed case). The uppercase key is used to insure that all case-spellings of the same word are stored in a single record, and to simplify the sort order. The tag contained within each record is the spelling of the tag that we will use in the tag cloud (and generally corresponds to the first use of the tag in the data).

Collecting Genesis Words

Our first script, *makeGenesisTags.pl*, produces a list of the words that appear in the book of Genesis, in the Bible. The data is retrieved from the copy of the book of Genesis at the Project Gutenberg website. To run the script, enter this command:

```
makeGenesisTags.pl
```

It will produce a file called *genesis.pl*. This script uses LWP::Simple to screen-scrape the Project Gutenberg website. Let's see how it works by examining the script:

```
#!/usr/bin/perl

use HTTP::Cache::Transparent;
use LWP::Simple;
use Data::Dumper;

use strict;
use warnings;
```

These lines insure that the HTTP::Cache::Transparent, LWP::Simple, and Data::Dumper modules are available. If they aren't, you'll see an error message when you run the script that says something like "Can't locate Data/Dumper.pm in @INC."

```
use strict;
```

```
use warnings;
```

The above lines turn on strict warnings that help you avoid misspelled variable names and other common problems in your script.

```
$Data::Dumper::Terse= 1;   # avoids $VAR1 = * ; in dumper output
```

This line prevents Data::Dumper from prefixing its output with the boilerplate text "$VAR1 = ". This allows us to save the data to different variable names.

```
HTTP::Cache::Transparent::init( {
    BasePath => './cache',
    NoUpdate  => 30*60
  } );
```

The HTTP::Cache::Transparent module provides a simple way to make screen-scraping scripts more efficient. When you read data from a website, a copy of the data is kept in a cached file. Subsequent reads will use the cached data rather than pulling the data from the website, if appropriate. The only additional code that needs to be added are the above lines, which specify where to keep cached data and how frequently to poll the website. We will retrieve data no more than every 30 minutes.

In this particular script, we are accessing a text that is unlikely to change (the Bible), so we can use a much larger number of `NoUpdate`.

```
# specify where to get the bible, and the desired verses
my $url = 'http://www.gutenberg.org/dirs/etext05/bib0110.txt';
```

This line specifies the URL of the web page we are going to screen-scrape. This particular page contains the text of the book of Genesis. If you'd like to use some other text, go to the Project Gutenberg website and find what you want at *www.gutenberg.org*.

To see what this text looks like in its raw form, check out the web page we're grabbing in your browser:

`http://www.gutenberg.org/dirs/etext05/bib0110.txt`

```
my $ofilename = "genesis.pl";
```

This line specifies the name of the output file where we are going to save our tags.

```
# get the text
my $txt = get($url);
```

This line retrieves the actual text of the page using the `get()` function that is provided by LWP::Simple.

```
# Remove Project Gutenberg Header
$txt =~ s/^.*\*\*\* START OF THE PROJECT GUTENBERG[^\n]*\n//s;

# skip the preface (this line is needed for the book of genesis only)
$txt =~ s/^.*(\nBook 01)/\1/s;
```

```
# Remove Project Gutenberg Trailer
$txt =~ s/\*\*\* END OF THE PROJECT GUTENBERG.*$//s;
```

These lines extract the portions of the text we are interested in. Project Gutenberg text contain some standard-issue boilerplate above and below the text, so we extract everything above and below those sections. The book of Genesis contains a preface, so we also remove that.

```
# remove some punctuation
$txt =~ s/[\.,]/ /gs;
```

This line removes commas and periods from the text, so that we have just a list of words, separated by spaces.

```
# convert text into individual words and count 'em
my $tags;

foreach my $w (split /\s+/, $txt)
{
  next if $w =~ /[0-9]/;  # skip paragraph numbers and other numbers
  next if $w eq '';
  my $uw = uc($w);
  $tags->{$uw} = {url=>'http://dictionary.reference.com/search?q='.$w, count=>0,
tag=>$w} if !(defined $tags->{$uw});
  $tags->{$uw}->{count}++;
}
```

This section examines each word individually and builds up the `tags` data structure. For each word, it builds a URL to the *http://dictionary.reference.com* website (keep in mind that this link may not work for all words), and it maintains a count of how many times the word has occurred. The associative array feature in languages like Perl makes it very simple to count words or tags.

Notice that we are using an uppercase version of the word (`$uw`) for the key. This ensures that if we encounter the same word with different capitalization, it will be counted as the same word.

Notice that we also store a copy of the word as it first appears. This will be output to the tag cloud.

```
open (OFILE, ">$ofilename") or die ("Can't open $ofilename file for $ofilename ");
print OFILE "package mytags;\n\n\$tags = " . Dumper($tags) . ";\n1;\n";
close OFILE;

printf "Wrote %d tags to $ofilename \n", scalar(keys %{$tags});
```

The above section uses the `Dumper()` function provided by Data::Dumper to output our `$tags` data structure to a file. Here are the first few lines of that file:

```
package mytags;

$tags = {
  'PASSED' => {
      'count' => 8,
      'url' => 'http://dictionary.reference.com/search?q=passed',
      'tag' => 'passed'
    },
  'SORE' => {
```

```
      'count' => 10,
      'url' => 'http://dictionary.reference.com/search?q=sore',
      'tag' => 'sore'
    },
  'AT' => {
      'count' => 54,
      'url' => 'http://dictionary.reference.com/search?q=at',
      'tag' => 'at'
    },
// etc…
```

Later in this article, we'll be using this data to build a tag cloud, but first let's make another script that gathers data from a different source.

Collecting del.icio.us Tags

Our second script, *makeDeliciousTags.pl*, produces a list of tags from the most recent entries in your del.icio.us account. If you don't have a del.icio.us account, you can either get one (it's free) or use my username (*jbum*) as I'm doing in these examples.

To run the script, enter its name on the command line, followed by the del.icio.us username that you want to collect tags for, like so:

```
makeDeliciousTags.pl jbum
```

In this example, the script will produce a file called *deliciousTags_jbum.pl*.

This script is a little different, since it is parsing an RSS file. Let's examine it in more detail.

```
#!/usr/bin/perl

use HTTP::Cache::Transparent;
use LWP::Simple;
use Data::Dumper;
use XML::RSSLite;
```

These lines load the CPAN modules we are going to use. In addition to the modules used in the previous script, we add the XML::RSSLite module, which enables us to parse the XML data in an RSS feed.

```
use strict;
use warnings;
```

As in the previous script, the above lines turn on strict warning messages, and initialize the caching mechanism.

```
HTTP::Cache::Transparent::init( {
    BasePath => './cache',
    NoUpdate  => 30*60
  } );
```

The above lines specify a local caching directory, as before, and insure that we don't access the website more than once every 30 minutes. This precaution not only makes your script more efficient, it is the recommended access policy at

del.icio.us. If you attempt to read the del.icio.us RSS feeds more frequently, you can get blacklisted at that site.

```
$Data::Dumper::Terse= 1;   # avoids $VAR1 = * ; in dumper output
```

This line prevents Data::Dumper from prefixing its output with the boilerplate text "$VAR1 = " and allows us to save the data to different variable names.

```
my $who = shift;
$who = 'jbum' if !$who;
my $delURL = "http://del.icio.us/rss/$who";
```

These lines load the username from the command-line argument. If a username isn't specified, the script uses my username (*jbum*). The username forms the URL that contains the RSS feed we are going to parse, and later will be incorporated into the name of the output file we are going to write.

```
print "loading tags...\n";
my $xml = get($delURL);
```

In this line, we load the RSS feed data into a variable, using the `get()` function provided by LWP::Simple.

```
my %result = ();
parseRSS(\%result, \$xml);
```

In these lines, we parse the data in the RSS feed (which is in XML format). This particular parsing function, `parsers()`, is provided by the XML::RSSLite module. On CPAN, there are other parsers you can use, but they will have different calling conventions. If you wish to use a different parser, check the documentation to make sure you are using it properly.

```
my $tags = {};
foreach my $item (@{$result{'item'}})
{
	my $url = $item->{link};
	foreach my $tag (split / /,$item->{'dc:subject'})
	{
		my $utag = uc($tag);
		if (!$tags->{$utag}) {
			$tags->{$utag} = {url=>$url, count=>0, tag=>$tag};
		}
		$tags->{$utag}->{count}++;
	}
}
```

These lines walk through the parsed RSS data and extract the tags. Each bookmark on del.icio.us is stored in an `item` record. The item record contains a link URL and a set of tags that are space-delimited.

This code splits the tags up into an array and then walks through the array, incorporating each tag into our data structure and maintaining a tally of all of the tags.

```
my $ofilename = "deliciousTags_$who.pl";
```

This line sets the output filename, incorporating the name of the user whose tags we are interested in. From here on out, the script is essentially the same as the script that counted the words in Genesis.

```
my $ofilename = "deliciousTags_$who.pl";
open (OFILE, ">$ofilename") or die ("Can't open $ofilename file for $ofilename ");
print OFILE "package mytags;\n\n\$tags = " . Dumper($tags) . ";\n1;\n";
close OFILE;

printf "Wrote %d tags to $ofilename \n", scalar(keys %{$tags});
```

These final lines use the `Dumper()` function provided by Data::Dumper to write the parsed tag data to the output file. Now we can use this data to build a tag cloud!

Displaying Tags Using HTML::TagCloud

CPAN contains a module called HTML::TagCloud that displays tag clouds. It is faster (but not as flexible) to use this module than to write your own code to display tag clouds. We'll develop our own code for displaying tag clouds in a moment, but first let's take a look at the easier method. Here is a sample script that uses HTML::TagCloud:

```
#!/usr/bin/perl

use HTML::TagCloud;

use strict;
use warnings;

require "genesis.pl";

my $cloud = HTML::TagCloud->new;

foreach my $tag (keys %{$tags})
{
       $cloud->add($tag, $tags->{$tag}->{url}, $tags->{$tag}->{count});
}

print $cloud->html_and_css();
```

This code produces a tag cloud with centered words using font sizes of 12 to 36 points. If you wish to customize the look of the tag clouds produced by this method, you'll need to modify the CSS code. You can do this by providing a custom CSS file and using this alternate function to produce the tag cloud:

```
print $cloud->html();
```

Displaying Tags Using Your Own Code

There are various ways to display a tag cloud. I've chosen a style that closely resembles the tag clouds on Flickr. Here is the HTML for a very small tag cloud, so you can see how it is structured:

```
<div class="cdiv">
<p class="cbox">
<a href="link1" style="font-size:23px;">tag1</a>
<a href="link2" style="font-size:18px;">tag3</a>
<a href="link3" style="font-size:13px;">tag3</a></p>
</div>
```

Each word or tag is associated with the style division class `cdiv` (which is defined in the CSS style file), and the font size is given explicitly for each tag.

These days, it is fashionable to separate style from structure, and keep all stylistic information in the CSS file. The tag clouds produced by HTML::TagCloud accomplish this goal by eliminating the explicit font-size references and using a set of individual styles (`tagcloud1`, `tagcloud2` ,`tagcloud3`, etc.), one for each font size. While the basic idea of separating style from structure is desirable, this particular use strikes me as silly, since the separate classes are functioning as implicit font-size directives. It reduces clarity in the HTML code and makes the CSS code needlessly complex.

Since we have full control over the code that generates the tag cloud, there is little need to use CSS to modify the range of font sizes—instead, we will control this detail through scripting.

For the tag clouds in this article, I am putting the `font-size` directive in the tag-cloud code itself, and using a shorter CSS file called *mystyle.css* that mimics the Flickr look:

```
body { padding-bottom: 10px; padding-top: 0px; margin: 0px; background: #fff; }
p { font: 12px Arial, Helvetica, sans-serif; }
.cbox { padding: 12px; background: #f8f8f8; }
.cdiv {margin-top: 0; padding-left: 7px; padding-right: 7px; }
.cdiv a { text-decoration: none;  padding: 2px; }
.cdiv a:visited { color: #07e; }
.cdiv a:hover { color: #fff; background: #07e; }
.cdiv a:active { color: #fff; background: #F08; }
```

Our challenge is to build some HTML code that looks like the sample above, giving each tag an appropriate font size and the appropriate link from the database.

We'll start with a very simple way to accomplish this task and then work up to a more sophisticated method. If you'd like to cut directly to the chase, you'll find the code in *makeTagCloud.pl*.

The first script simply uses the count associated with each tag for the `font-size`. Here is the script, called *makeTagCloud1.pl*:

```
#!/usr/bin/perl

use strict;
use warnings;

# load in tag file
```

```
my $tagfile = shift;
$tagfile = 'genesis.pl' if !$tagfile;

# output beginning of tag cloud
print <<EOT;
<html>
<head>
  <link href="mystyle.css" rel="stylesheet" type="text/css">
</head>
<body>
<div class="cdiv">
<p class="cbox">
EOT

# output individual tags
foreach my $k (sort keys %{$tags})
{
  my $fsize = $tags->{$k}->{count};
  my $url = $tags->{$k}->{url};
  my $tag = $tags->{$k}->{tag};
  printf "<a href=\"%s\" style=\"font-size:%dpx;\">%s</a>\n",
       $url, int($fsize), $tag;
}

# output end of tag file
print <<EOT;
</p>
</div>
</body></html>
EOT
```

To use this script, supply the name of the tag file as the first parameter and redirect the output to a temporary HTML file to test it. The result is shown in Figure 7.

```
$ makeTagCloud1.pl deliciousTags_jbum.pl >test.html
```

Figure 7. *makeTagClouds1.pl*

As you can see, the words are far too small. We probably don't want to see a font size smaller than about ten points, so let's add ten to the count. We'll change the line that converts tag count to font size from this:

```
my $fsize = $tags->{$k}->{count};
```

to this:

```
my $fsize = 10+$tags->{$k}->{count};
```

This change produces the tag cloud shown in Figure 8.

Figure 8. Minimum font size of ten points

This looks okay, but there are a few problems. The word "music" is really big, but all of the other words are quite small. I'd like to see a little more variety in the font sizes. Another problem becomes apparent when I run the script on my Genesis words. I get the tag cloud shown in Figure 9.

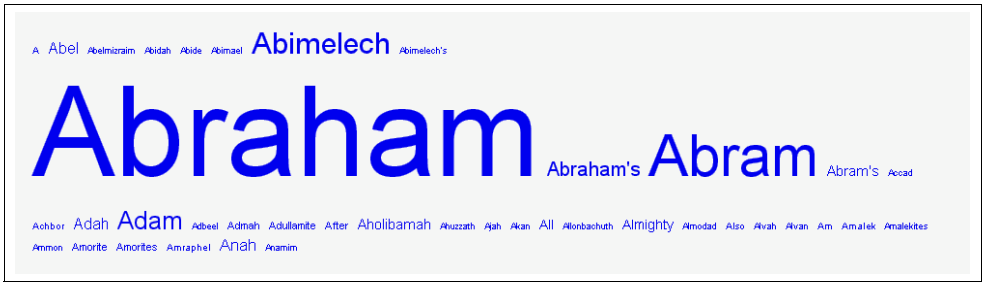


Figure 9. [Needs Caption]

The fonts in this tag cloud are much too large! What would happen if I had a tag with a count of 2,000? I'd get a font taller than the resolution of most monitors. Clearly, I need to do something a bit more sophisticated. What I want to do is map the tag counts, which are going to go from some minimum value to some maximum value (minimum tag count → maximum tag count) to a range of desired font sizes (minimum font size → maximum font size).

To do this, I first need to determine what those numbers are. The following code sets the minimum and maximum font sizes to constant values:

```
my $minFontSize = 10;
my $maxFontSize = 36;
my $fontRange = $maxFontSize - $minFontSize;
```

As you can see, I'm using the range 10 to 36—a little wider than the range used by HTML::TagCloud. I think this is more elegant than using a stylesheet that contains a bunch of individual font directives.

To determine the minimum and maximum tag counts, we'll let the script loop through the data:

```
# determine counts
my $maxTagCnt = 0;
my $minTagCnt = 10000000;

foreach my $k (@sortkeys)
{
  $maxTagCnt = $tags->{$k}->{count}
      if $tags->{$k}->{count} > $maxTagCnt;
  $minTagCnt = $tags->{$k}->{count}
```

```
      if $tags->{$k}->{count} < $minTagCnt;
}
```

We'll add the following function to the bottom of the script. It converts a tag count to a font size. This function does a straight linear mapping. It first converts the tag count to a ratio (which goes from 0 to 1) and then maps it to the desired range of font sizes:

```
sub DetermineFontSize($)
{
  my ($tagCnt) = @_;
  my $cntRatio = ($tagCnt-$minTagCnt)/($maxTagCnt-$minTagCnt);
  my $fsize = $minFontSize + $fontRange * $cntRatio;
  return $fsize;
}
```

To use the above function, we'll replace this line:

```
my $fsize = 10+$tags->{$k}->{count};
```

with this, which calls the function:

```
my $fsize = DetermineFontSize($tags->{$k}->{count});
```

The resultant tag cloud, for Genesis, looks like Figure 10:

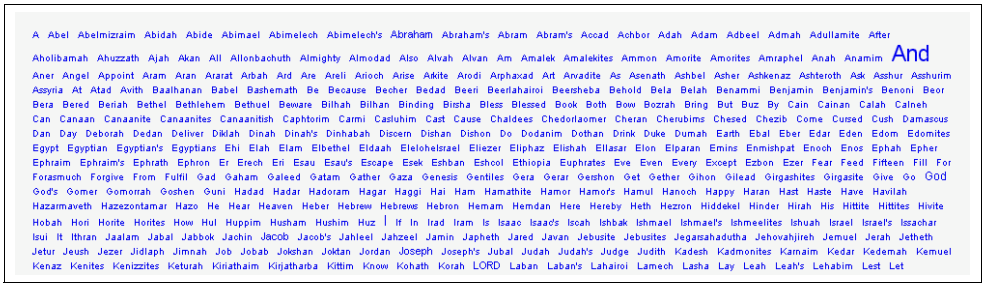


Figure 10. Linear mapping

Magnifying the Long Tail (Inverse Power Mapping)

The uniformity of the font sizes I noted earlier is still a problem. The reason for this is that the tag counts are arranged in a power curve (Figure 11). Power curves are a very common phenomenon found in popularity or frequency data collected from human activity.

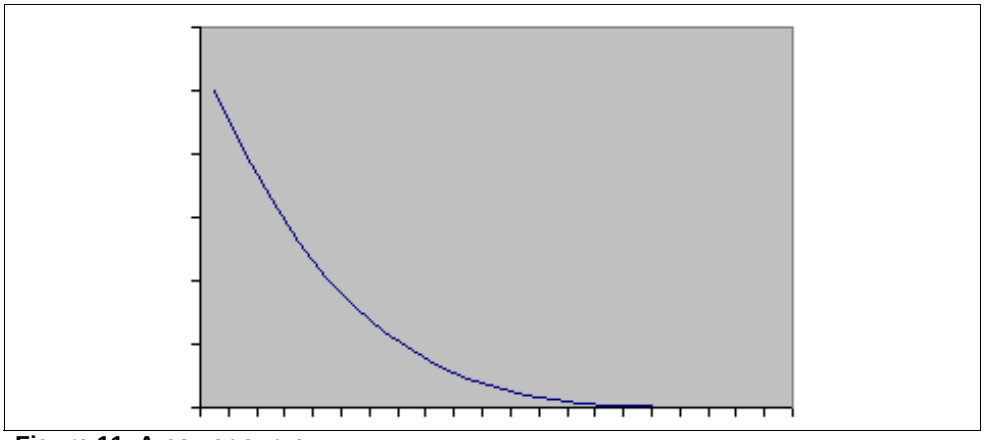


Figure 11. A power curve

There tends to be a very few large values in the data, and lots and lots of small values. The problem with mapping a power curve to a limited set of font sizes is that the "long tail" of the power curve ends up getting represented by just one or two font sizes. Many of the intermediate font sizes won't get used at all because of the larger gaps between the counts of the most popular words.

The way to make this tag cloud look better is to use a logarithmic function to reverse the power curve's effects. Essentially, we will map the linear range of font values to the logarithmic range of tag counts, magnifying the differences between smaller counts and making the "long tail" of the power curve more visible (Figures 12 and 13).

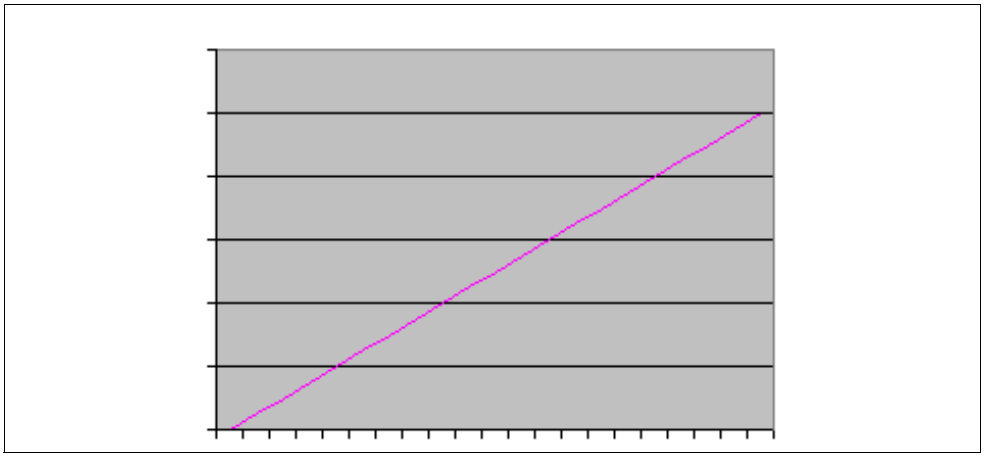


Figure 12. Linear mapping of x to y

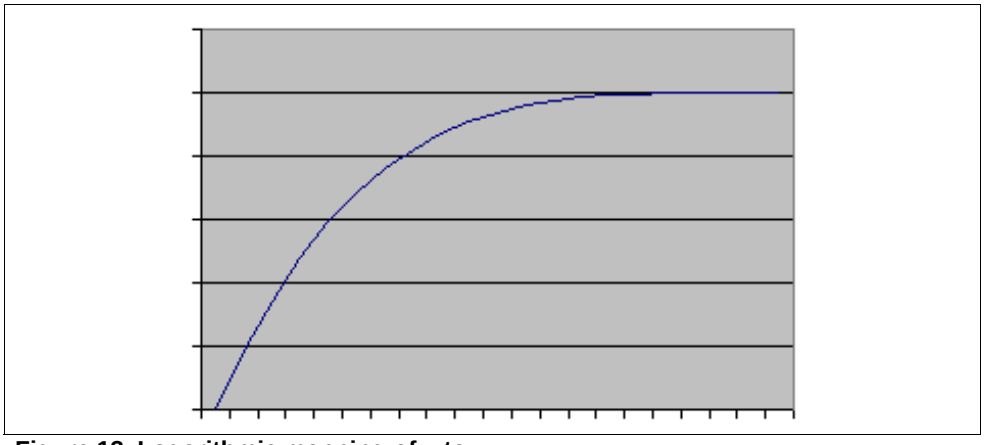


Figure 13. Logarithmic mapping of x to y

To do this, we'll add a logarithmic measure of the tag counts:

```
my $minLog = log($minTagCnt);
my $maxLog = log($maxTagCnt);
my $logRange = $maxLog - $minLog;
$logRange = 1 if ($maxLog == $minLog);
```

And we'll modify the function that determines font size.

```
sub DetermineFontSize($)
{
  my ($tagCnt) = @_;
  my $cntRatio;

  if ($useLogCurve) {
    $cntRatio = (log($tagCnt)-$minLog)/$logRange;
  }
  else {
```

```
    $cntRatio = ($tagCnt-$minTagCnt)/($maxTagCnt-$minTagCnt);
  }
  my $fsize = $minFontSize + $fontRange * $cntRatio;
  return $fsize;
}
```

The variable `$useLogCurve` will be used to provide logarithmic mapping. I suggest setting it to `1` (or true) by default.

Note that if `$useLogCurve` is set to `0`, we get the straight linear mapping we had before.

The logarithmic mapping is shown in Figure 14.

Figure 14. Logarithmic mapping of Genesis tags (compare to Figure 8)

The tags are looking a little better; however, there are still far too many small words. Let's filter the tags down to the top 200 so we can see just the most common words. This step produces a tag cloud that fits on a single page and displays a wider variety of font sizes.

To do this, we'll add the following code to collect the 200 most common tags into an array:

```
$maxtags = 200;
mmy @sortkeys = sort {$tags->{$b}->{count} <=> $tags->{$a}->{count}} keys
%{$mytags::tags};
@sortkeys = splice @sortkeys, 0, $maxtags;
```

We use this array anywhere we were previously using `keys %{$tags}`.

The final Perl script, called *makeTagCloud.pl*, reads as follows:

```
#!/usr/bin/perl

use strict;
use warnings;

# load in tag file
my $tagfile = shift;
$tagfile = 'genesis.pl' if !$tagfile;

require "$tagfile";
die ("No tags loaded\n") if (!$mytags::tags);
my $tags = $mytags::tags;

my $useLogCurve = 1;
my $minFontSize = 10;
my $maxFontSize = 36;
my $fontRange = $maxFontSize - $minFontSize;
```

```
my $maxtags = 200;
my @sortkeys = sort {$tags->{$b}->{count} <=> $tags->{$a}->{count}} keys
%{$mytags::tags};
@sortkeys = splice @sortkeys, 0, $maxtags;

# determine counts
my $maxTagCnt = 0;
my $minTagCnt = 10000000;

foreach my $k (@sortkeys)
{
  $maxTagCnt = $tags->{$k}->{count}
      if $tags->{$k}->{count} > $maxTagCnt;
  $minTagCnt = $tags->{$k}->{count}
      if $tags->{$k}->{count} < $minTagCnt;
}

my $minLog = log($minTagCnt);
my $maxLog = log($maxTagCnt);
my $logRange = $maxLog - $minLog;
$logRange = 1 if ($maxLog == $minLog);

sub DetermineFontSize($)
{
  my ($tagCnt) = @_;
  my $cntRatio;

  if ($useLogCurve) {
    $cntRatio = (log($tagCnt)-$minLog)/$logRange;
  }
  else {
    $cntRatio = ($tagCnt-$minTagCnt)/($maxTagCnt-$minTagCnt);
  }
  my $fsize = $minFontSize + $fontRange * $cntRatio;
  return $fsize;
}

# output beginning of tag cloud
print <<EOT;
<html>
<head>
  <link href="mystyle.css" rel="stylesheet" type="text/css">
</head>
<body>
<div class="cdiv">
<p class="cbox">
EOT

# output individual tags
foreach my $k (sort @sortkeys)
{
  my $fsize = DetermineFontSize($tags->{$k}->{count});
  my $url = $tags->{$k}->{url};
  my $tag = $tags->{$k}->{tag};
  printf "<a href=\"%s\" style=\"font-size:%dpx;\">%s</a>\n",
    $url, int($fsize), $tag;
}
```

```
# output end of tag file
print <<EOT;
</p>
</div>
</body></html>
EOT
```

The Genesis tag cloud produced by this script is shown in Figure 15.

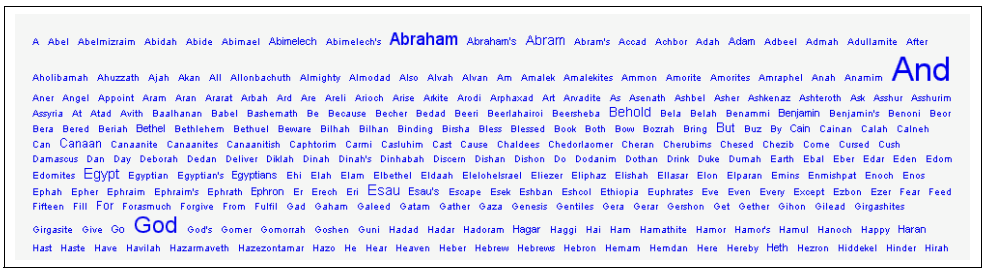


Figure 15. Final Genesis tag cloud: top 200 terms and logarithmic mapping

As I mentioned earlier, you can use a frequency sort (Figure 16) instead of an alphabetical sort. Just change this line in the display loop:

```
foreach my $k (sort @sortkeys)
```

To this:

```
foreach my $k (sort
   {$tags->{$b}->{count} <=> $tags->{$a}->{count}}
   @sortkeys)
```

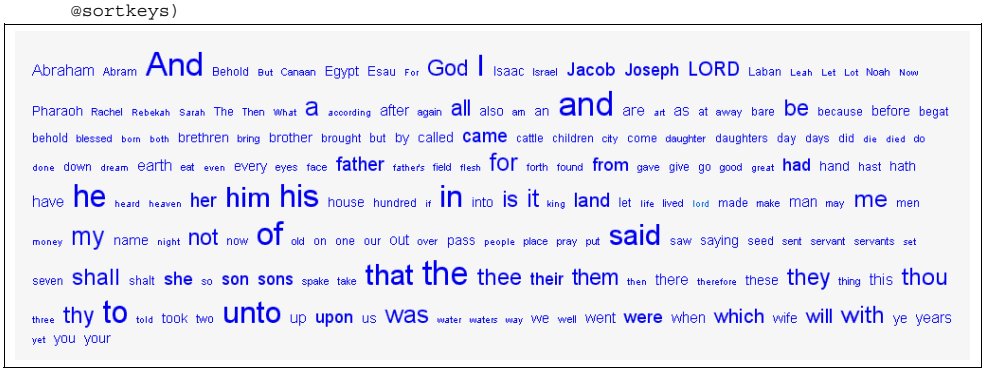


Figure 16. Tag cloud with word frequency sorting

However, as I also mentioned earlier, I prefer the alpha sort. It's more "cloudy" and it provides additional information.

Making Tag Clouds in PHP

In the Perl scripts in this article, I have saved the tag information to a file and passed that file (via a command-line argument) to another script, *makeTagCloud.pl*, which generates the tag cloud. The PHP versions will not use this temporary file, but instead collect the tags directly into an array. Otherwise, these scripts mimic the functionality of the scripts we developed in Perl.

Collecting Genesis Words in PHP

Here is a PHP function that collects tags by counting the words that appear in Genesis. Save this as *getGenesisTags.php*:

```
<?
//
// Collect text from genesis

function getTags()
{
  global $tags;

  $url = 'http://www.gutenberg.org/dirs/etext05/bib0110.txt';

  // $txt = file_get_contents($url);
  $ch = curl_init();
  $timeout = 30; // set to zero for no timeout
  curl_setopt ($ch, CURLOPT_URL, $url);
  curl_setopt ($ch, CURLOPT_RETURNTRANSFER, 1);
  curl_setopt ($ch, CURLOPT_CONNECTTIMEOUT, $timeout);
  $txt = curl_exec($ch);
  curl_close($ch);

  $searches = array('/^.*\*\*\* START OF THE PROJECT GUTENBERG[^\n]*\n/s',
                    '/^.*(\nBook 01)/s',
                    '/\*\*\* END OF THE PROJECT GUTENBERG.*$/s',
                    '/[^\w\'\-]/s');
  $replaces = array('',
                    '$1',
                    '',
                    ' ');


  $txt = preg_replace($searches,$replaces,$txt);

  $words = preg_split('/\s+/', $txt);
  foreach ($words as $w)
  {
    if ($w == '' || preg_match('/[0-9]/',$w))
        continue;
    addTag($w, 'http://dictionary.reference.com/search?q='.$w);
  }

}
?>
```

If the PHP implementation you are using supports the `allow_url_fopen` option, you can shorten the script by replacing these lines:

```
$ch = curl_init();
$timeout = 30; // set to zero for no timeout
curl_setopt ($ch, CURLOPT_URL, $url);
curl_setopt ($ch, CURLOPT_RETURNTRANSFER, 1);
curl_setopt ($ch, CURLOPT_CONNECTTIMEOUT, $timeout);
$txt = curl_exec($ch);
curl_close($ch);
```

with this:

```
$txt = file_get_contents($url);
```

However, many servers leave this option turned off, because it is insecure.

This script uses a function called `addTag()` to count the tags. We'll take a look at the code for this function later, when we discuss displaying tag clouds in PHP.

Collecting del.icio.us Tags in PHP

To collect tags from an RSS feed, we'll use the library MagpieRSS, a freely available RSS parser for PHP. You can download Magpie from http://magpierss.sourceforge.net/.

Here is the script for collecting tags from del.icio.us. Save this as *getDeliciousTags.php*.

```
<?
//
// Collect delicious tags (or load them from a cache)

function getTags()
{
  global $tags;

  // use the parameter 'who' to determine which account to poll
  // it defaults to 'jbum'

  $who = 'jbum';
  if (isset($_GET['who']))
    $who = $_GET['who'];

  // remove troublesome characters from name
  $who = preg_replace('/\W/', '',$who);

  if ($who == '')
    return;

  $delURL = "http://del.icio.us/rss/$who";

  require('magpierss/rss_fetch.inc');

  // fetch and parse RSS data
  $rss = fetch_rss($delURL);
```

```
  // collect tags
  foreach ($rss->items as $item ) {
    $tagstrings = preg_split('/ /',$item[dc][subject]);
    foreach ($tagstrings as $tagstring)
    {
      addTag($tagstring, "http://del.icio.us/$who/$tagstring");
    }
  }
}
?>
```

One of the nice things about MagpieRSS is that it has a built-in caching function, so we don't have to worry about overtaxing the del.icio.us servers.

Display Tags in PHP

Here is our final script for displaying tags in PHP:

```
<html>
<head>
  <link href="mystyle.css" rel="stylesheet" type="text/css">
</head>
<body>
<div class="cdiv">
<p class="cbox">

<?

$tags = array();

function addTag($tag,$url)
{
       global $tags;
       $utag = strtoupper($tag);
       // echo "Adding tag $tag<br>";
       if (!$tags[$utag])
              $tags[$utag] = array('cnt' => 0,
                                   'firstUrl' => $url,
                                   'tag' => $tag);
       $tags[$utag]['cnt']++;
       $tags[$utag]['lastUrl']= $url;
}

// use either 'getGenesisTags.php' or 'getDeliciousTags.php'
//
include "getDeliciousTags.php";

getTags();

// grab top LIMIT here, if arguments specify a limit
// and reduce to top N tags.
if (isset($_GET['limit']))
{
  arsort($tags);
  $tags = array_slice($tags, 0, (int)($_GET['limit']));
}

// Build Log Cloud from Tags
```

```
//
$useLogCurve = 1;
if (isset($_GET['linear']))
       $useLogCurve = 0;

$minFontSize = 10;
$maxFontSize = 36;
$fontRange = $maxFontSize - $minFontSize;
$maxTagCnt = 0;
$minTagCnt = 10000000;

foreach ($tags as $tag => $trec)
{
       $cnt = $trec['cnt'];

  if ($cnt > $maxTagCnt)
       $maxTagCnt = $cnt;
  if ($cnt < $minTagCnt)
       $minTagCnt = $cnt;
}
$tagCntRange = $maxTagCnt+1 - $minTagCnt;

$minLog = log($minTagCnt);
$maxLog = log($maxTagCnt);
$logRange = $maxLog - $minLog;
if ($maxLog == $minLog) $logRange = 1;


ksort($tags); # use arsort($tags) to sort by descending count

foreach ($tags as $utag => $trec)
{
  $cnt = $trec['cnt'];
  $url = $trec['lastUrl'];
  $tag = $trec['tag'];
  if ($useLogCurve)
    $fsize = $minFontSize + $fontRange * (log($cnt) - $minLog)/$logRange;
  else
    $fsize = $minFontSize + $fontRange * ($cnt - $minTagCnt)/$tagCntRange;
  printf("<a target=xxx href=%s style=\"font-size:%dpx;\">%s</a>\n", $url,(int)$fsize,
$tag);
}

?>

</p>
</div>
</body></html>
```

To use these scripts, upload the PHP files to your web server. Then invoke the *makeTagCloud.php* script from your web browser by typing in a URL like the following:

http://ww.yourdomain.com/makeTagCloud.php?limit=200&who=jbum

The scripts accept three parameters.

`limit`

Limits the maximum number of tags (particularly useful when you have a lot of them, as with the Genesis tags).

`who`

Specifies an account to collect tags for from del.icio.us.

`linear`

Turns off the logarithmic font mapping.

Figure 17 shows a sample result.

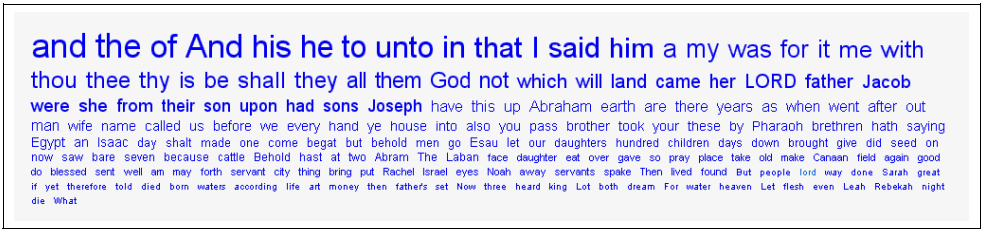


Figure 17. Tag cloud using tags from del.icio.us

[Jim: add brief closing para about PHP section here?]

Summary

[Jim: pls add brief conclusion to wrap up PDF here]

Copyright

ISBN: 0596527942