



Building Tag Clouds in Perl and PHP

By Jim Bumgardner

.....
Publisher: O'Reilly
Pub Date: May 2006
Print ISBN-10: 0-596-52794-2
Print ISBN-13: 978-0-59-652794-5
Pages: 48

Table of Contents

Overview

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



Building Tag Clouds in Perl and PHP

By Jim Bumgardner

.....

Publisher: O'Reilly
Pub Date: May 2006
Print ISBN-10: 0-596-52794-2
Print ISBN-13: 978-0-59-652794-5
Pages: 48

Table of Contents

- [Copyright](#)
- [Building Tag Clouds in Perl and PHP](#)
- [Tag Clouds: Ephemeral or Enduring?](#)
- [Weighted Lists](#)
 - [Section 1.1. Creating Weighted Lists](#)
 - [Section 1.2. Tag Cloud Properties](#)
 - [Section 1.3. The Utility of Tag Clouds](#)
- [Some History](#)
- [Design Tips for Building Tag Clouds](#)
 - [Section 4.1. Choose the Right Language](#)
 - [Section 4.2. Make Your Tag Clouds Visible to Search Engines](#)
 - [Section 4.3. Frequency Sorting](#)
 - [Section 4.4. Avoid Random Mappings](#)
 - [Section 4.5. Make Tag Clouds Relevant to Your Users](#)
 - [Section 4.6. Try Different Mappings](#)
- [Making Tag Clouds in Perl](#)
 - [Section 5.1. Collecting Tags](#)
 - [Section 5.2. Collecting Genesis Words in Perl](#)
 - [Section 5.3. Collecting del.icio.us Tags in Perl](#)
 - [Section 5.4. Displaying Tags In Perl Using HTML::TagCloud](#)
 - [Section 5.5. Displaying Tags In Perl Using Your Own Code](#)
 - [Section 5.6. Magnifying the Long Tail \(Inverse Power Mapping in Perl](#)
- [Making Tag Clouds in PHP](#)
 - [Section 6.1. Collecting Tags](#)
 - [Section 6.2. Collecting Genesis Words in PHP](#)
 - [Section 6.3. Collecting del.icio.us Tags in PHP](#)
 - [Section 6.4. Display Tags in PHP](#)
 - [Section 6.5. Magnifying the Long Tail \(Inverse Power Mapping in PHF](#)
- [Conclusion](#)



Copyright

Building Tag Clouds with Perl and PHP, by Jim Bumgardner

Copyright © 2006 O'Reilly Media, Inc. All rights reserved.

Not for redistribution without permission from O'Reilly Media, Inc.

ISBN: 0596527942





Building Tag Clouds in Perl and PHP

By Jim Bumgardner

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

This PDF 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, 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 constructing 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

Contents

Tag Clouds: Ephemeral or Enduring?	2
Weighted Lists	3
Some History	11
Design Tips for Building Tag Clouds	13
Making Tag Clouds in Perl	15
Making Tag Clouds in PHP	31
Conclusion	46

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

This PDF 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 constructing 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?

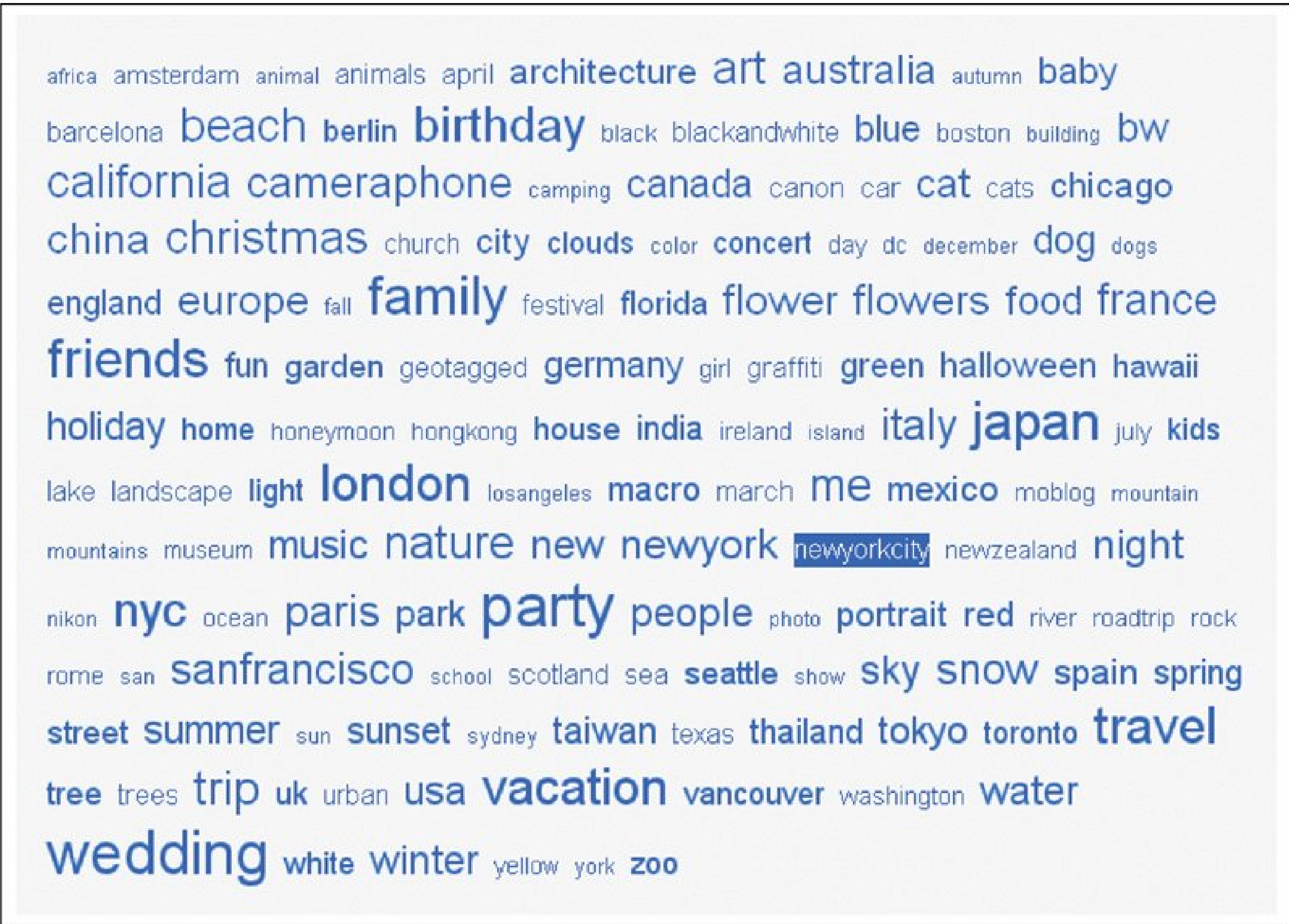
If you're reading this, you've probably seen a tag cloud (Figure 1) as you've browsed the Web. In this article, I'm going to provide a little analysis and history of tag clouds, and then get on to more important matters: I'll demonstrate how to create your own tag clouds in Perl and PHP.

Tag clouds are a current fashion. But in April of 2005, web design guru Jeffrey Zeldman decried their faddishness in his headline, "Tag Clouds Are the New Mullet," 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 have said, I will show how tag clouds, when used properly, have real, and lasting merits.

Note: All of the scripts in this article can be downloaded from O'Reilly's web site at the following URL:
<http://examples.oreilly.com/tagclouds/> .

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 web site (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.

Figure 2. Weighted cities list from craigslist

er kind of weighted list, one that's even more distant from tag clouds, is that of the statistically probable 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

First Sentence:

THUS, WITH UNABASHED conceits, wrote Roberto della Griva presumably in July or August of 1643. [Read the first page](#)

Statistically Improbable Phrases (SIPs): ([learn more](#))

antipodal meridian, orange dove, first meridian, relief army, aqua vitae, deserted ship, infinite worlds

Capitalized Phrases (CAPs): ([learn more](#))

[Father Caspar](#), [Padre Emanuele](#), [Islands of Solomon](#), [Powder of Sympathy](#), [San Patrizio](#), [Specula Melitensis](#), [Stone Fish](#), [Charles Emmanuel](#), [Isla de Hierro](#), [Land of Romances](#), [Tweede Daphne](#), [Monsieur de la Grive](#), [Monsieur de Saint-Savin](#), [Prime Meridian](#), [Solomon Islands](#), [Canon of Digne](#), [Terra Incoqnita](#), [Island of Solomon](#), [Punto Fijo](#), [Roberto de la Grive](#), [San Giorgio](#), [Knight of Malta](#), [Monsieur de Toiras](#), [Persona Vitrea](#), [Captain Gambero](#)

New!

[Books on Related Topics](#) | [Concordance](#) | [Text Stats](#)

Browse Sample Pages:

[Front Cover](#) | [Copyright](#) | [Table of Contents](#) | [Excerpt](#) | [Back Cover](#) | [Surprise Me!](#)

Search Inside This Book:

GO!

1.1. Creating Weighted Lists

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

1.1.1. 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, but here are a few possible things you might want to map:

1.1.2. 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

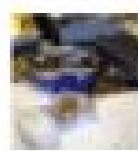



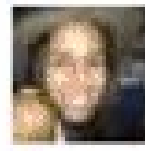
A. Visual Features	B. Underlying Data
Word shape	Sans serif

Weighted lists on other web sites 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 web site 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	San serif

Figure 4. Tag cloud for 43 Things

329,095 people in 6,096 cities are doing 430,537 things including...

walk on Mars trust Keep my website updated  cookiesinatincan wants to
talk with philosophers and people intersted in philosophy write a song be an
extra in a film improve my Hebrew love Learn to draw/paint. do my pushups 5
days a week. download windows media player 10 grow roses. improve my skin
decide what the hell I would like to do with the rest of my
life kiss someone in the rain Lose my freshman 15 learn to play the accordion
 lanny727 wants to Buy a MP3 learn to sail learn how to cook like my mom
Read the entire Bible in a year Get organized **Get a tattoo**
build my own house own a home build a darkroom **Kiss in the rain**
Own a successful business  michaelresolution wants to read the Bible daily
be a volunteer write more letters by hand learn to sew Be happy without being in
love. See a Final Fantasy Concert Run Boston sing in a choir Take vitamins daily
become more cultured work in china write more, write better learn to play the
accordion  rixie wants to keep my house clean get 1 million neopoints on
neopets improve my memory make a new friend **Be a better friend** **Learn**
Ruby **Skydive** I want to attend a Goon-meet eat more fruit 
trainee teacher girl wants to stop feeling lonely learn something new every
day be more positive download the new episode of 24 **Start my own**
business Visit Egvpt get another piercing **eat healthier** become fat



1.2. 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 lawsthere are few popular items, and many more unpopular items.
- The tags are links navigable to the tagged content.

The first property gives 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.



1.3. The Utility of Tag Clouds

While you can click on tag clouds, you can also just look at them to get a quick reading of a web site' 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 get different winners such as "buy a house," "write a book," and "be happy."

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 5), or in a bar graph (Figure 6).

Figure 5. Word frequency list

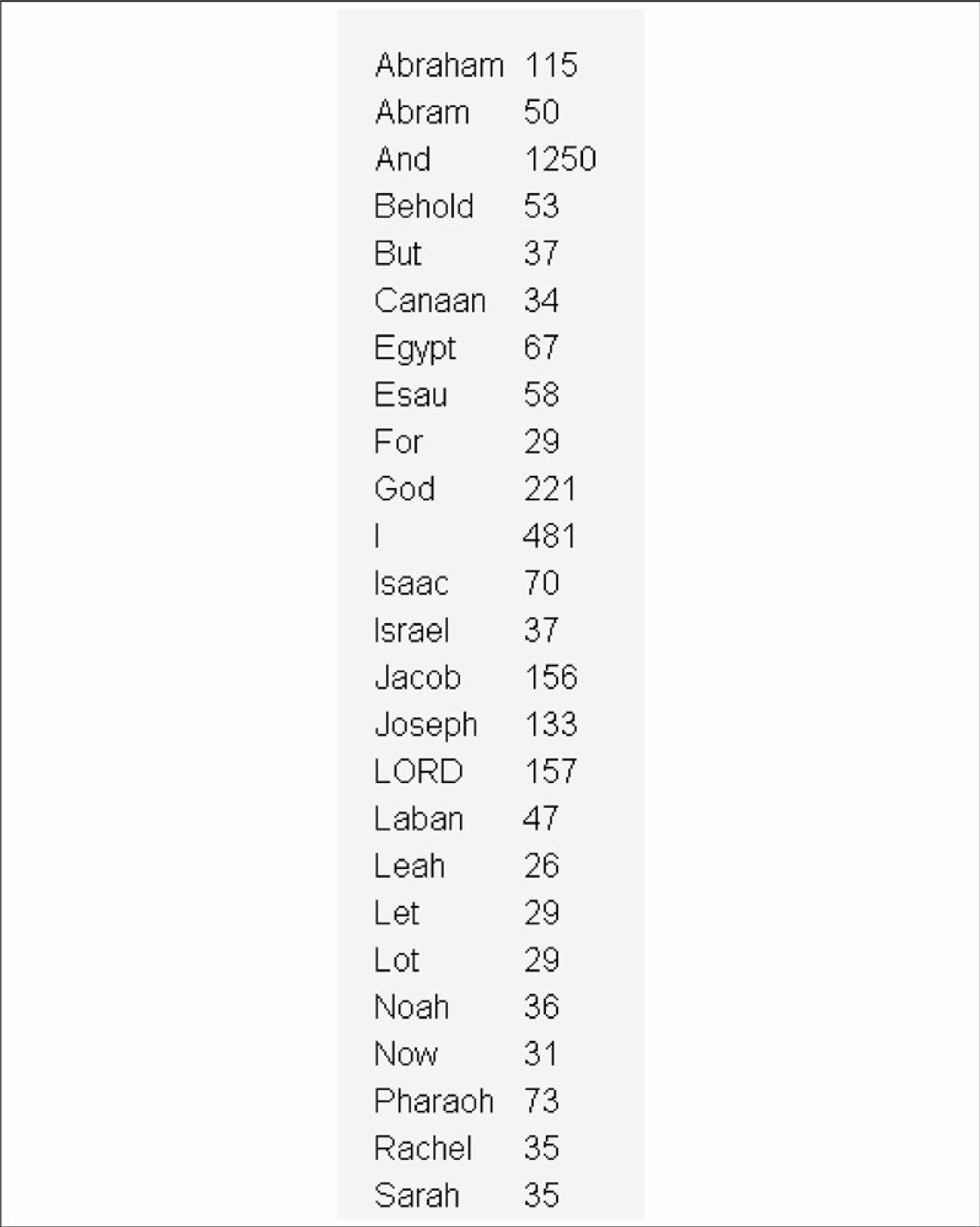
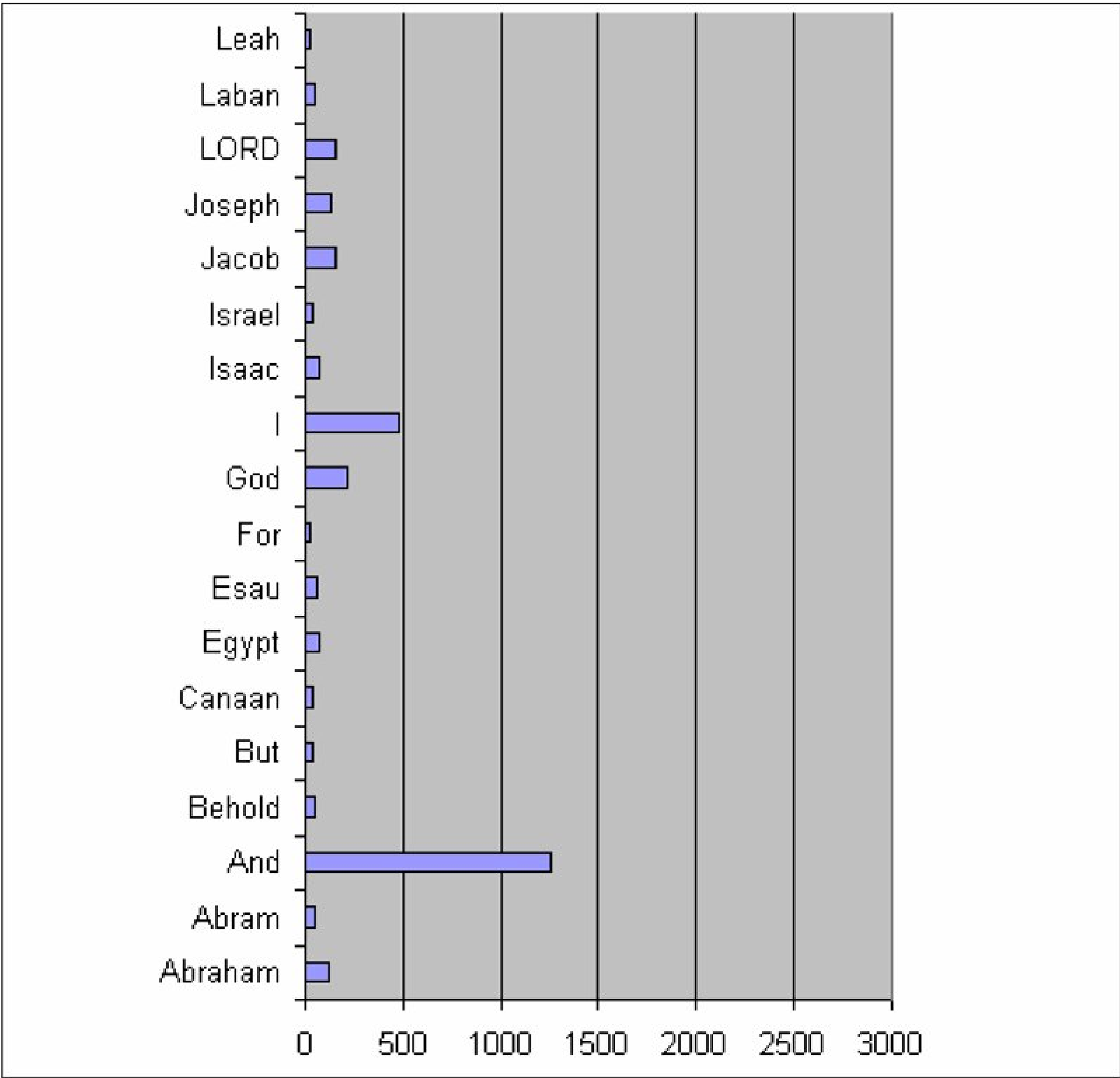


Figure 6. Word frequency bar graph



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 7), they can provide the essential information for a larger number of words in a much smaller space.

Figure 7. Word frequency tag cloud

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 web site to search engines like Google. In static web sites, people use the <meta description> and <meta keywords> tags to describe the content of the web site 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

Flickr, a photography-sharing web site that caters to bloggers, was the first web site to use something called a tag cloud. However, tag clouds really have their roots in the blogging community. Bloggers have a need to organize the large amounts of material they constantly churn out, and an excellent communications medium to propagate new and interesting methods.

Flickr's tag cloud idea was likely inspired (directly or indirectly) from an older blog plugin called Zeitgeist (Figure 8), 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 8 .

Figure 8. Jim Flanagan's Zeitgeist plugin in action



If you look at Jim's code (or Figure 8), 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 visua

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.



[← PREV](#)

Design Tips for Building Tag Clouds

Tag clouds can be used effectively, and provide real value to a web site, or they can be tacked on as an afterthought, simply because they look cool, or to make the site appear similar to other, better web sites that offer them. Ultimately, you need to keep in mind their dual function, both as a graph of current activity, and as a navigation aid. Here are some design and implementation tips:

[← PREV](#)

 [PREV](#)

4.1. 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.

 [PREV](#)

[← PREV](#)

4.2. 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 snazzierflashier, 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 in a comment block in the HTML.

[← PREV](#)



4.3. Frequency Sorting

You can, if you like, sort tag clouds by word frequency ([Figure 19](#)). Personally, I don't think it's a good idea. Not only does it reduce the "cloudy" nature of the word list, but it also 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.



 [PREV](#)

4.4. Avoid Random Mappings

In Jim Flanagan's Zeitgeist plugin, the words are colored randomly, and the colors have no significance. While some people like this, I believe that if you desire clarity in the interface, you should try to make each mapping meaningful, and eliminate random information that adds no value. For example, you could associate color with time, or omit the color mapping entirely.

 [PREV](#)

 [PREV](#)

4.5. Make Tag Clouds Relevant to Your Users

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.

 [PREV](#)

[← PREV](#)

4.6. 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.

[← PREV](#)

Making Tag Clouds in Perl

Note: This section, which shows how to make tag clouds in Perl, is followed by a section that covers the same material, but uses PHP. If you are more familiar with PHP, I suggest you skip ahead to the PHP section.

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 parserone 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 web sites that provide data we can use to build tag clouds.



5.1. 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 sourcesone 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).



5.2. Collecting Genesis Words in Perl

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 web site. 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 web site. 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 web site, 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 web site, 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 web site. 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 web site (<http://www.gutenberg.org/>) to find what you want.

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 $filename = "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> web site (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, ">$filename") or die ("Can't open $filename file for $filename");
print OFILE "package mytags;\n\n$tags = " . Dumper($tags) . ";\n\n";
close OFILE;
printf "Wrote %d tags to $filename \n", scalar(keys %{$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 construct the HTML for the tag cloud, but now let's make another script that gathers actual tags from a different source.



5.3. Collecting del.icio.us Tags in Perl

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. You can view my del.icio.us bookmarks at the following URL: <http://del.icio.us/jbum>.

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 web site 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, `parseRSS()`, 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!



5.4. Displaying Tags In Perl 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();
```


5.5. Displaying Tags In Perl 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`, and so on), 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 tagcloud 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: Let's call it *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;
}

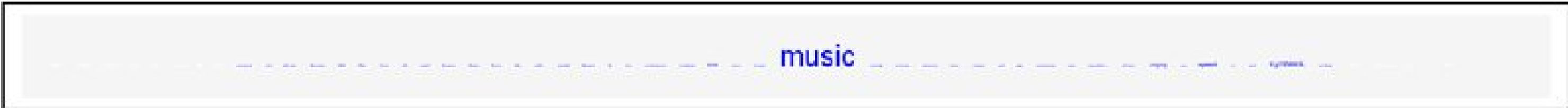
# output end of tag file
print <<EOT;
</p>
</div>
</body></html>
EOT
```

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:

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

The result is shown in Figure 9.

Figure 9. *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 10.

Figure 10. Minimum font size of ten points

This looks OK, 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 11

Figure 11. Genesis Tags, without scaled mapping

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). In other words, I need to scale the mapping.

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 style sheet 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 12:

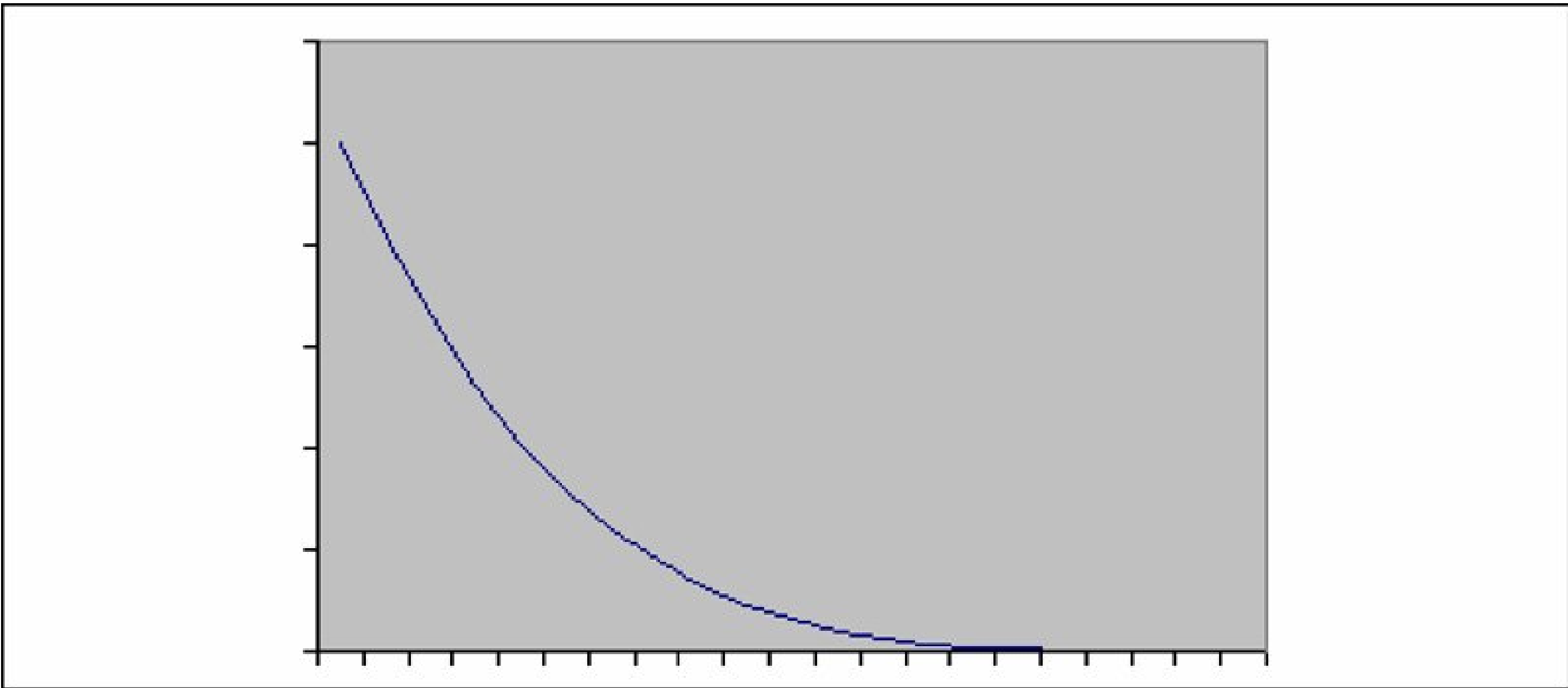
Figure 12. Linear mapping



5.6. Magnifying the Long Tail (Inverse Power Mapping in Perl)

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 13). Power curves are a very common phenomenon found in popularity or frequency data collected from human activity.

Figure 13. 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 14 and 15).

Figure 14. Linear mapping of x to y

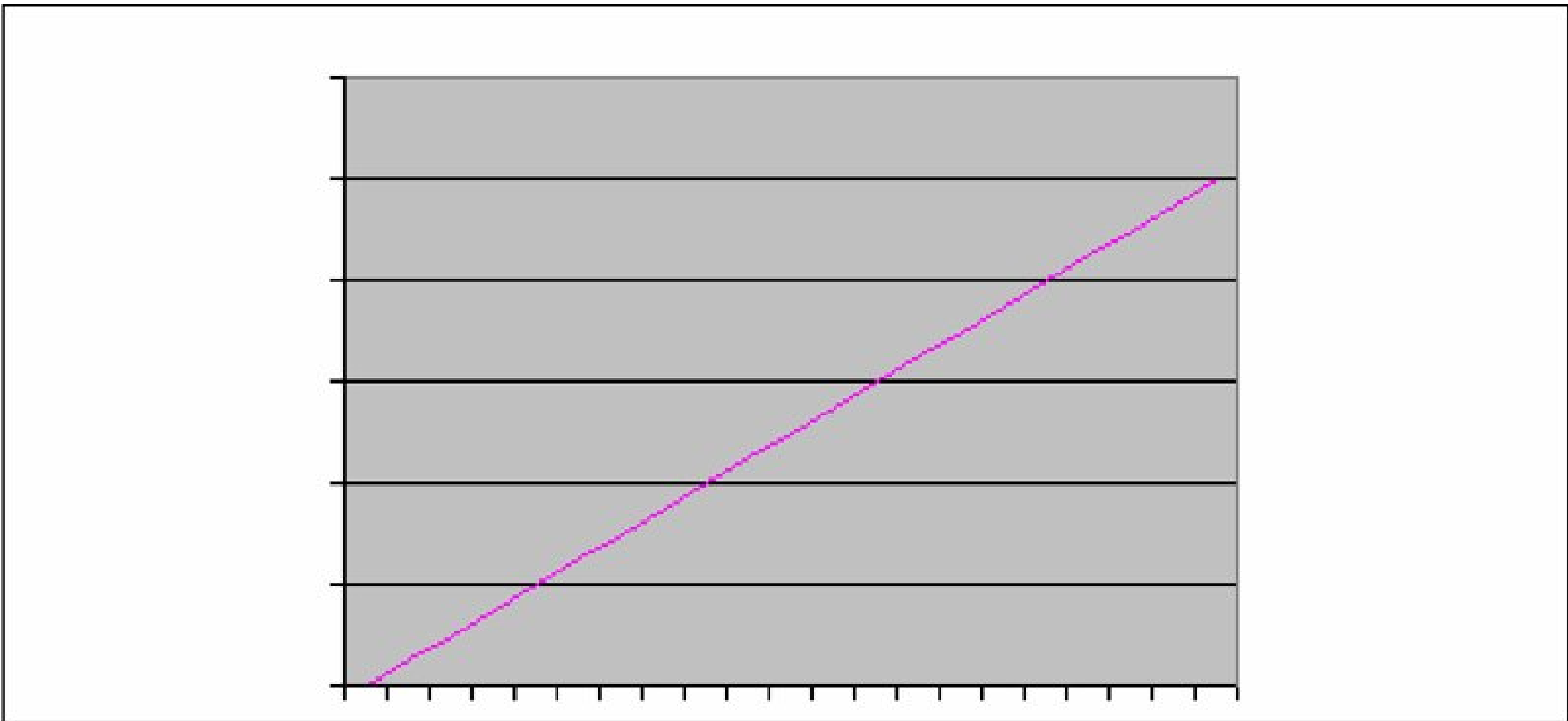


Figure 15. Logarithmic mapping of x to y

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 16.

Figure 16. Logarithmic mapping of del.icio.us tags (compare to Figure 10)

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:

```
$maxtags = 200;  
my @sortkeys = sort {$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;
}

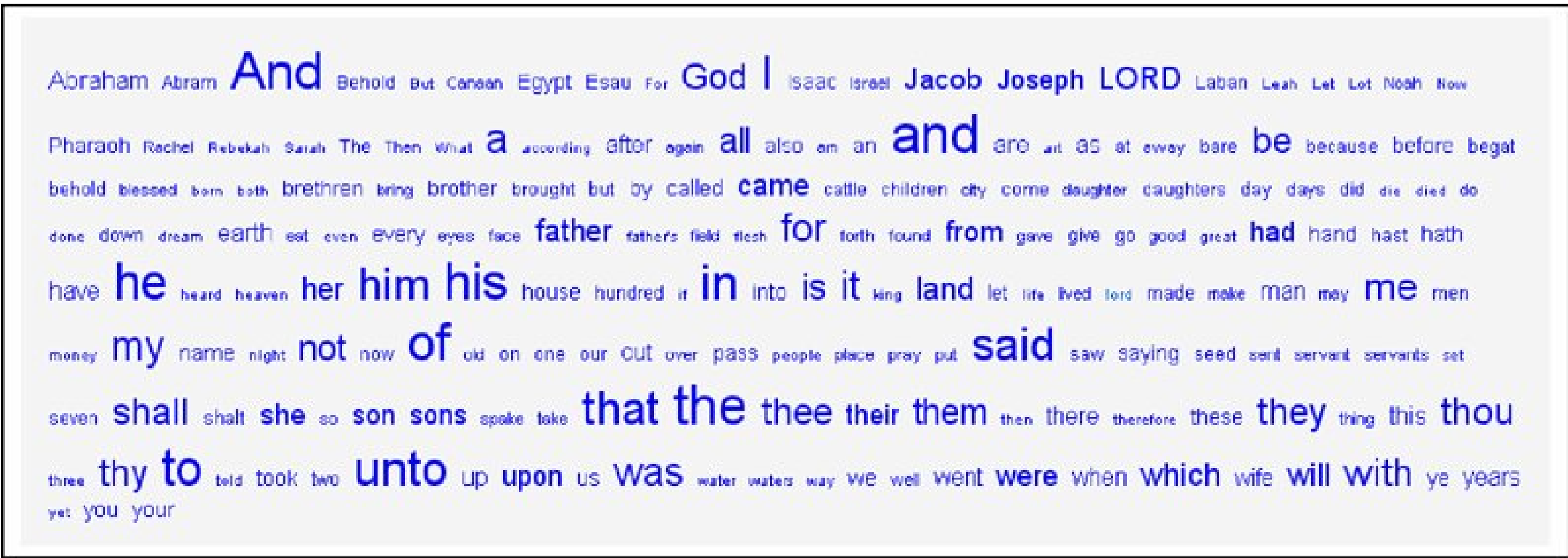
# output beginning of tag cloud
print <<EOT;
<html>
<head>
    <link href="mystyle.css" rel="stylesheet" type="text/css">
</head>
<body>
<div class="cdiv">
<p class="cbox">
EOT

# 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 17.

Figure 17. Final Genesis tag cloud: top 200 terms and logarithmic mappin



As I mentioned earlier, you can use a frequency sort (Figure 18) instead of 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 18. Tag cloud with 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

Note: This section, which shows how to make tag clouds in PHP, is preceded by a section that covers the same material, but uses Perl. If you are more familiar with Perl, and haven't read that section, I suggest you read it first.

Now I'll show you how to make a Flickr-style tag cloud in PHP. In order to run these scripts, you'll need some familiarity with PHP and access to a web server that supports PHP. PHP scripts are typically uploaded to the server using FTP, and then tested by invoking them in a local web browser that can access the server. You can learn more about PHP by visiting the PHP web site:

<http://www.php.net/>.

There is one minor difference in the following scripts from the Perl scripts described in the previous section: In the Perl scripts, we 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 counterpart scripts 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.

Most of this article is concerned with 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 web sites that provide data we can use to build tag clouds.

6.1. 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 save the data to a global associative array called `$tags`. The `$tags` array contains the set of tags, the tags' associated counts, and associated URLs. If I were to initialize the array from scratch, it might look like this:

```
$tags = array(
    'RSS' => array(
        'count' => 1,
        'url' => 'http://magprier.ss.sourceforge.net/',
        'tag' => 'RSS'
    ),
    'NYQUIST' => array(
        'count' => 1,
        'url' => 'http://audacity.sourceforge.net/help/nyquist3',
        'url' => 'http://audacity.sourceforge.net/help/nyquist3',
        'tag' => 'Nyquist'
    ),
    'GENERATORS' => array(
        'count' => 1,
        'url' => 'http://generatorblog.blogspot.com/',
        'tag' => 'generators'
    ),
    // etc...
);
```

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' element 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).

Here is a function that can be used to build up such an array, using successive calls. This function is included in our script *addTag.php*.

```
function addTag($tag, $url)
{
    global $tags;
    $utag = strtoupper($tag);
    if (!$tags[$utag])
        $tags[$utag] = array('count' => 0, 'url' => $url, 'tag' => $tag);
}
```

```
    $tags[$utag][ 'cnt' ]++;  
}
```

To build up a set of tags, we call `addTag()` once for each occurrence of a tag, providing the name of the tag, and the URL we want the tag to link to, like so:

```
addTag( 'weddings', 'http://www.mywebsite.com/tags/weddings' );  
addTag( 'sunsets', 'http://www.mywebsite.com/tags/sunsets' );  
addTag( 'puppies', 'http://www.mywebsite.com/tags/puppies' );
```

If a tag occurs more than once, we call the function once for each occurrence, inorder to increment the count.

 **PREV**

6.2. Collecting Genesis Words in PHP

Here is a PHP script, *getGenesisTags.php*, which collects tags by counting 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 web site. (This script is available at <http://examples.oreilly.com/tagclouds/>.) Let's see what it does.

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

function getTags()
{
    global $tags;
```

The script contains a single function, called `getTags()`. This function will be invoked from another script, *makeTagCloud.php*, which we will invoke later. The purpose of the `getTags()` function is to populate the global associative array called `$tags`.

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

The previous 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 web site (<http://www.gutenberg.org/>) to find what you want.

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

The previous lines retrieve the bible text from the Project Gutenberg web site. I am accomplishing

this using the curl library. There is a simpler way to retrieve this text, but it only works if your web host supports the PHP

`allow_url_fopen` option. Mine doesn't, because it is believed to be a security risk, so I am providing this alternate method, which uses curl. If the PHP implementation you are using supports the `allow_url_fopen` option, you can shorten the script by replacing the previous lines with the single line which is commented out at the top of the block:

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

Moving right along, we do a little processing on the text by performing a series of text replacements. These next few lines describe the search and replacement patterns and perform the replacement.

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

The patterns passed to `preg_replace` are used to 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. We also remove commas and periods from the text, so that we have just a list of words, separated by spaces.

Next we convert the words into an array, by using the `split` function:

```
$words = preg_split('/\s+/', $txt);
```

And then we walk thru the array and call the `addTag()` function on each word.

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


For each word, we provide a URL to the <http://dictionary.reference.com> web site(keep in mind that this link may not work for all words).

This script uses the `addTag()` function, which will count how many times each word occurs. This function will be provided by the invoking script, *makeTagCloud.php*.

Now let's look at a similar script, which collects actual tags, rather than words, from a different source.





6.3. Collecting del.icio.us Tags in PHP

Our second PHP script, *getDeliciousTags.php*, 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. You can view my del.icio.us bookmarks at the following URL: <http://del.icio.us/jbum>.

Del.icio.us provides tags in an RSS feed. To collect and parse the RSS data, we'll use the library *magpie RSS*, a freely available RSS parser for PHP. You can download *magpie* at: <http://magpierss.sourceforge.net/>

You will need to follow the instructions that come with *magpie RSS* to install it on your server.

Because this script deals with RSS data, it looks a bit different than the previous script. Let's examine it in detail:

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

function getTags()
{
    global $tags;
```

Again, the script contains a single function, `getTags()`, which is responsible for building up the global associative array, `$tags`.

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

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

The previous lines determine which del.icio.us account you are collecting tags for. If one is specified in the URL parameters, it will be used, otherwise, my account, 'jbum' will be used.

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

Since this script is intended to be used on a web site, we have to be careful that user don't enter invalid data that causes unintentional side effects, so we remove all alphanumeric characters from the name, to reduce the security risks that might come from accessing unfiltered URLs.

```
if ($who == '')
return;
```

And in the previous line, we return early if a blank name is provided, or we get a blank after filtering.

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

In the previous line, the username is used to form the URL that retrieves the del.icio.us feed we are interested in.

```
require('magpie/rss_fetch.inc');
```

Since we are using magpie RSS, we have to let PHP know that we require it. Otherwise, the script won't work.

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

Finally, we fetch the RSS data. Magpie RSS collects and parses the data. One of the nice things about magpie RSS is that it has a built-in caching function, so we don't have to worry about overtaxing the del.icio.us servers.

```
// 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");
    }
}
?>
```

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 the tags, by calling our `addTag()` function. Each tag's URL is set to point to the appropriate del.icio.us page, which contains the tagged links.



6.4. Display Tags in PHP

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 some scripts accomplish this goal by eliminating the explicit font-size references and using a set of individual styles (`tagcloud1`, `tagcloud2`, `tagcloud3`, and so on), one for each font size. While the basic idea of separating style from structure is desirable, this particular method 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 tagcloud 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.php*.

Out first version of the script simply uses the count associated with each tag for thefont size. Here is the script, called *makeTagCloud1.php*:

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

<?

$tags = array();

include "addTags.php";

include "getDeliciousTags.php";
// include "getGenesisTags.php";

getTags();

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

foreach ($tags as $utag => $trec)
{
    $cnt = $trec['count'];
    $url = $trec['url'];
    $tag = $trec['tag'];
    $fsize = $cnt;
    printf("<a target=xxx href=%s style=\"font-size:%dpx;\">%s</a>\n", $url,(int)$fsize,
$tag);
}

?>

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

To use this script, upload the four scripts, *addTags.php*, *getDeliciousTags.php*, *getGenesisTags.php*, and *makeTagCloud1.php* to your server. Invoke the script in your web browser by typing in the URL to the script:

<http://www.yourdomain.com/makeTagCloud1.php>

You can optionally add a 'who' parameter to indicate the del.icio.us account you wish to access.

The result is shown in Figure 19.

Figure 19. The output of *makeTagClouds1.php*



Note: Figures 19 thru 28 (in the PHP section) exactly duplicate Figures 9thru 18 (in the Perl section). I did this to keep the illustrations inline with the text, so that PHP programmers don't have to keep flipping back to the Perl section.

As you can see, the words in Figure 19 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:

```
$fsize = $cnt;
```

to this:

```
$fsize = $cnt+10;
```

This change produces the tag cloud shown in Figure 20.

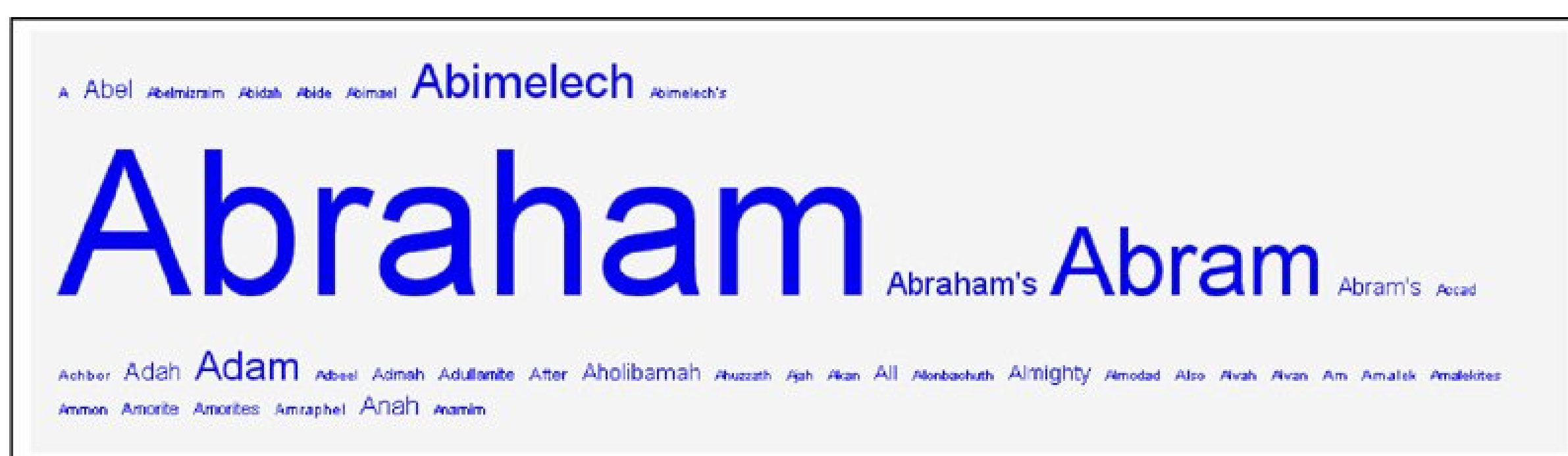
Figure 20. Minimum font size of ten points

This looks OK, but there are a few problems. The word "music" is really big, but all the other words are quite small. I'd like to see a little more variety in the font sizes. Another problem becomes apparent when we use the Genesis words instead of the del.icio.us tags. You can use the Genesis words by changing which include file is commented out.

```
// include "getDeliciousTags.php";  
include "getGenesisTags.php";
```

If you try this, you'll see the tag cloud shown in Figure 21

Figure 21. Genesis Tags, without scaled mapping.



The fonts in this tag cloud are much too large! What would happen if you had a tag with a count of 2,000? You'd get a font taller than the resolution of most monitors. Clearly, we need to do something a bit more sophisticated. What we 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 \longrightarrow maximum font size). In other words, we need to scale the mapping.

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

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

As you can see, I'm using the range 10 to 36. I think specifying the font this way is more elegant than using a style sheet 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:

```
$maxTagCnt = 0;
$minTagCnt = 10000000;

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

    if ($cnt > $maxTagCnt)
        $maxTagCnt = $cnt;
    if ($cnt < $minTagCnt)
        $minTagCnt = $cnt;
}

$tagCntRange = $maxTagCnt+1 - $minTagCnt;
```

We'll then modify our loop, which renders the tags to use this information to map the range of tag

counts to the desired range of font sizes.

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

The resultant tag cloud, for Genesis, looks like Figure 22:

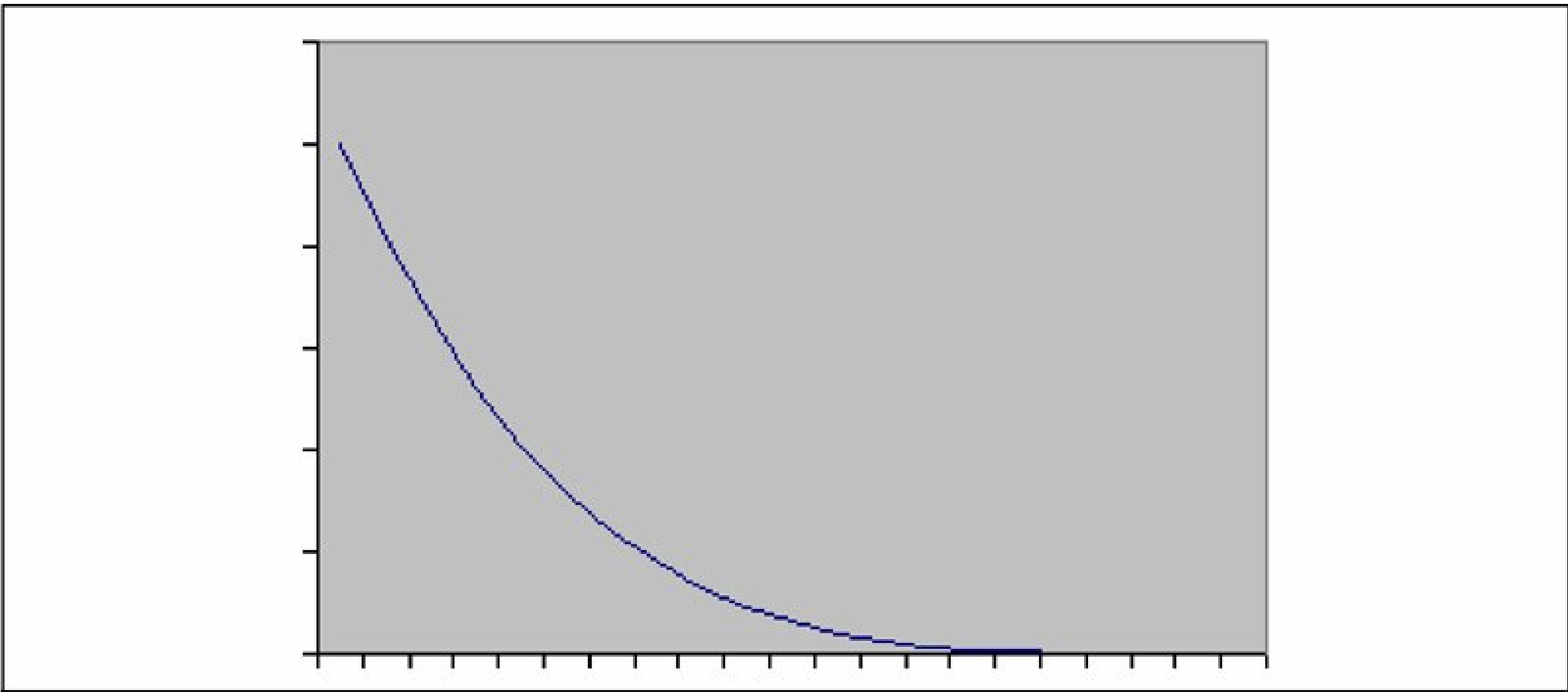
Figure 22. Linear mapping



6.5. Magnifying the Long Tail (Inverse Power Mapping in PHP)

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 23). Power curves are a very common phenomenon found in popularity or frequency data collected from human activity.

Figure 23. 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 24 and 25).

Figure 24. Linear mapping of x to y

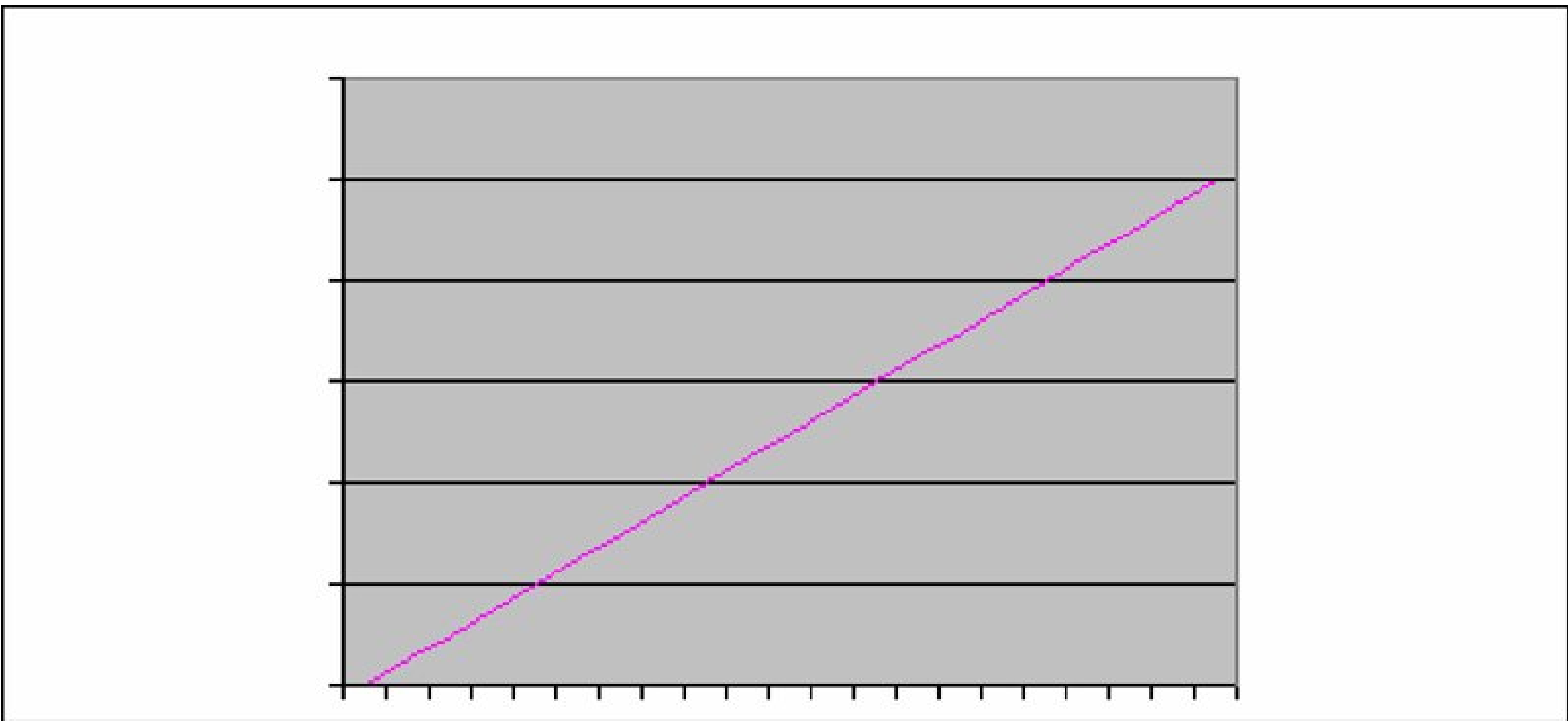
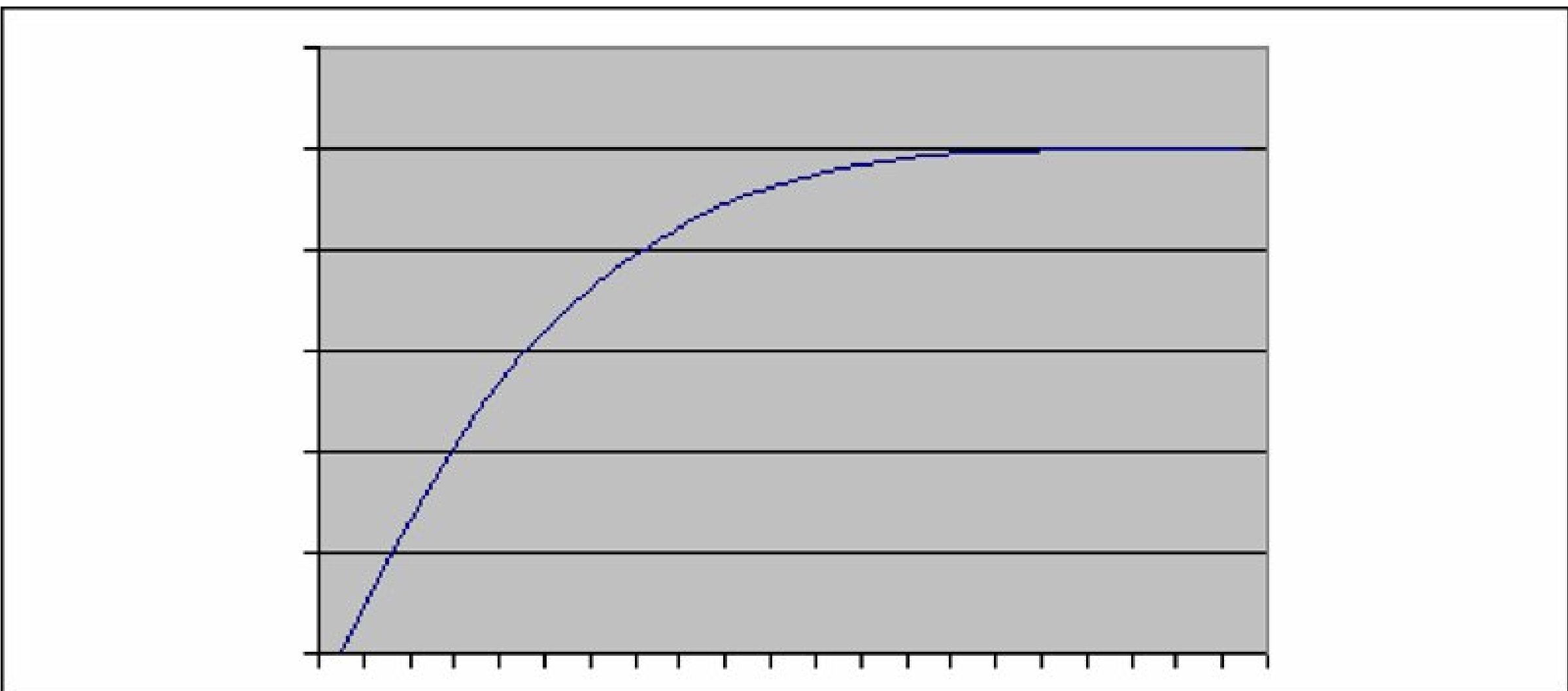


Figure 25. Logarithmic mapping of x to y



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

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

And we'll modify the line that determines the font size, to allow for a logarithmiccurve option:

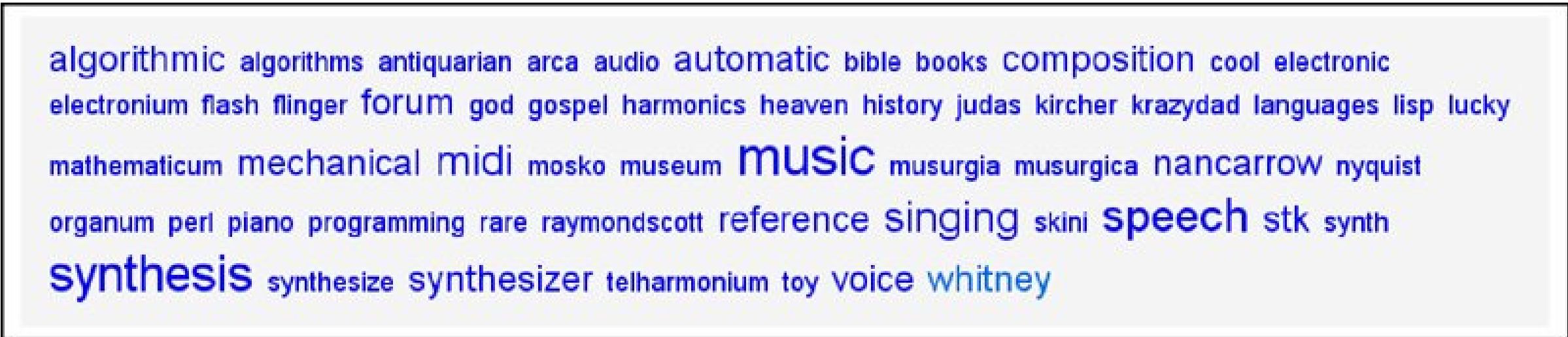
```
if ($useLogCurve)
    $fsize = $minFontSize + $fontRange * (log($cnt) - $minLog)/$logRange;
else
    $fsize = $minFontSize + $fontRange * ($cnt - $minTagCnt)/$tagCntRange;
```

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 26.

Figure 26. Logarithmic mapping of del.icio.us tags (compare to Figure 10)



The tags are looking a little better; however, there are still far too many small words. Let's provide an option to filter the tags down to some user-provided limit (such as 200) so we can see just the most common words. This will produce a tagcloud that fits on a single page and displays a wider variety of font sizes.

To do this, we'll add the following code to pay attention to the 'limit' parameter in the URL.

```
if (isset($_GET['limit']))
{
    arsort($tags);
    $tags = array_slice($tags, 0, (int)($_GET['limit']));
}
```

The final PHP script, called *makeTagCloud.php*, reads as follows:

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

<?

$tags = array();

include "addTags.php";

include 'getDeliciousTags.php';
```



```

// include "getGenesisTags.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['count'];

    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['count'];
    $url = $trec['url'];
    $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:%dp;\">%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 make *TagCloud.php* script from your web browser by typing in a URL like the following:

<http://www.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.

The Genesis tag cloud produced by this script is shown in Figure 27.

Figure 27. Final Genesis tag cloud: top 200 terms and logarithmic mapping

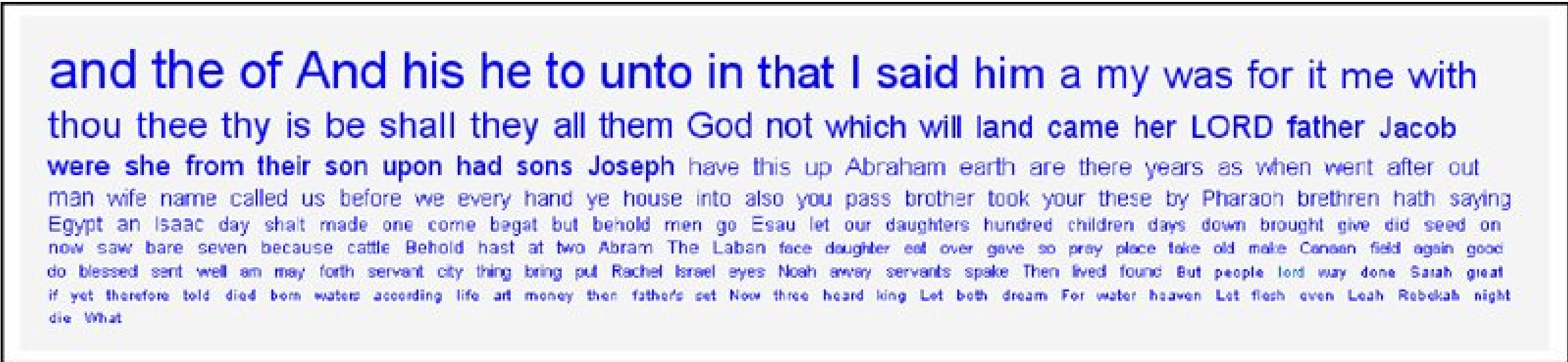
As I mentioned earlier, you can use a frequency sort (Figure 28) instead of alphabetical sort. Just change this line above the display loop:

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

To this:

```
arsort($tags);
```

Figure 28. Tag cloud with frequency sorting



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



Conclusion

Over the past few decades, digital technologies have dramatically increased our ability to store, organize, and access information. Today, I can instantly answer all kinds of questions that would have stumped me 20 years ago, and I have access to a wealth of words, sounds, and images far more than I have the intellectual capacity to consume. You could say we're in the midst of an information explosion, but I like to think we're being served an information cornucopia.

The abundance of this information is ever increasing, and the user interfaces we built ten years ago to access and organize it are starting to show signs of strain and wear, like a rickety folding table supporting the weight of a thousand pies.

Tag clouds are just one of a new crop of interfaces that aim to ease this strain. There are others, which succeed to greater and lesser degrees, and there will be better ones to come. I hope to have a part in making some of them, and I hope you do too.