URLs, hashes

Sevki
s@sevki.org

ABSTRACT

This article is briefly describes how to setup a sample application that works with OPC clients.

Problem as it turns out the image on the left is you can see you can't make out blondie, the URL **http://fistfulofbytes.com/how-to-bypass-ssl-validation-for-exchange-webservices-managed-api** is 91 characters long. On right hand-side **http://lea.cx/MQb2dg==** is encoded in QR because the URL is considerably shorter, you can sort of make out blondie's outline.

Straight forward way

Have a sql db, stick urls one by one, look up if a url has been been inserted, return the id if it has and so on. Problem with this approach is, every time you want to shorten a url you have to query a server. Can we do without the querying bit?

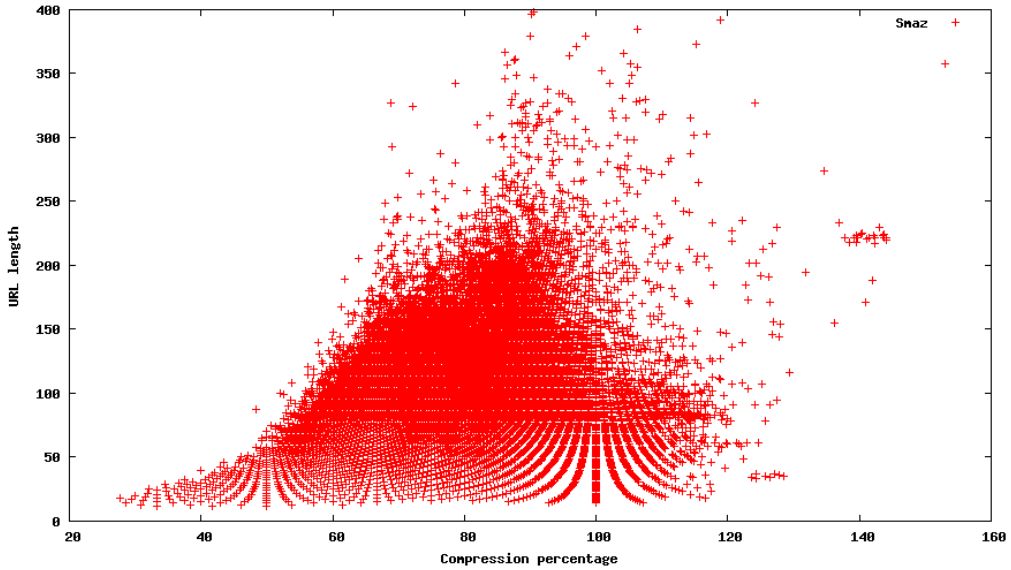
If something that needs to be less of that same something, we compress it right?

Plan B: Compression

Traditional compression methods such as **gzip**

bzip

lzw are not effective on short text. However there is a better solution by the great mind of antirezcalled **smaz** which does just that, make long urls shorter.



It works fairly well if you are storing gajillion urls. However if you are trying to make a url shortener it'll only compresses it to about 75%, which doesn't work 100of the time, but even when it does, it assumes that you have about 100-250 characters to begin with so getting the size to 60is an act in futility.

Plan A: Hashing

Ok, so compression was never going to work but hashes are probably going to work.

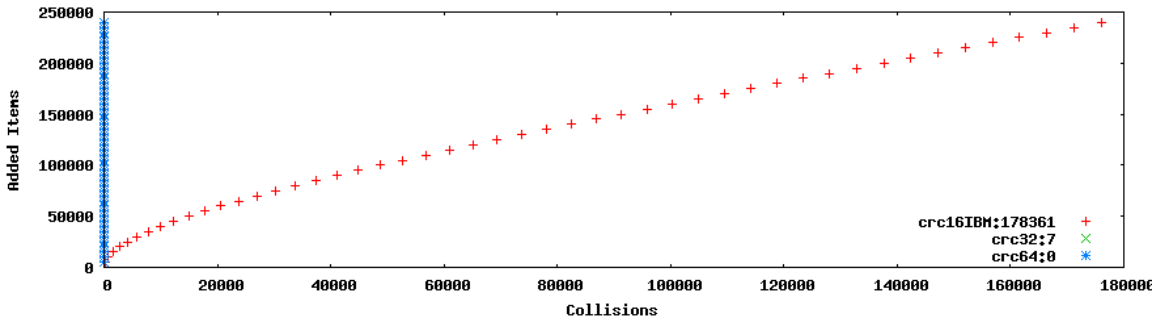
Why Hash?

Hashes are deterministic functions that always return **a** for any given **f(x)**

When building a URL shortener they come in handy; for instance given that you know that a **id:url** tuple exists, you don't have to query your service to serve a shortened version of the url, you can just serve the hash of the url. This saves you a roundtrip to your service.

Does it work?

YES. Here is comparison of 242 286 unique urls that are gathered from HNhashed with **crc16** , **crc32** and **crc64**



As you can see the 4 byte and 8 byte hashes nearly and don't collide at all respectively. And 2byte hashes behave in an extremely predictable way when you are hashing 17mb of urls. Clear winner in this race is crc32 seeing that in 242286 we got 7 collisions out of only 4 bytes. Which is negligible, or sheer bad luck. When talking about hashes, bitwise length is raised as 2's power to calculate possible combinations as such:

```
margin-left: 0;
  margin-right: 0;
  background-color: #fff;
  border-width: 1px;
  border-color: #ddd;
  border-radius: 4px 4px 0 0;
  box-shadow: none;
  border-collapse: collapse;
  border-spacing: 0; }
  border-spacing: 2px;
  border-color: gray;
  border-bottom: 2px gray solid;
  display: table-header-group;
  vertical-align: middle; }
  border-bottom: 1px solid #c0c0c0; }
  padding-left:10px;
  padding-right:10px; }
```

bits
bytes

possible combinations

16
2
65,536

32
4
4,294,967,296

64
8
18,446,744,073,709,551,616

Seeing that there are a possible 4 294 967 296 combinations, 7 collisions seem like a happy conincidince.

Encoding, UTF-8, base64

Unfortunately there is a caveat, our 4 bytes as is, are not url safe nor are they type-able on a qwerty keyboard. Because type-able ascii characters are 7 bits, when you divide 32bits into 4 bytes and use as text each one thats greater than 127 should look like gibberish, but in reality it's even less than that because we want to use the stuff that doesn't need modifier keys to be typed. Enter http://en.wikipedia.org/wiki/Base64. Encoding something in base64 is the process of taking a bitmap and slicing it into 6 bit characters and mapping those bits to base64 chars. But a 4 byte array such as this **49,6,246,118** encoded in base64 looks like this **MQb2dg==** , and the length jumped to 8.

Can we go from 8 to 4?

Would halving the bytes encoded also half the encoded string length, it probably will. But we really don't need to go that far, for instance if the hash is just 2 bytes **49,6** it will be **MQY=** encoded in base64. Just like before there are **=** signs at the end which is padding. Because neither 32 nor 16 are exact multiples of 6 we need to pad them, but 24 is, which is 3 bytes.

```
//  width:100%;
  text-align:center;
  border-spacing: 0px;
  font-family:Monospace;
  size:8px;
  border-collapse: collapse; }
  padding:0px; }
  float:left;
  border:solid red 2px;
  padding:2px; }
  margin-right: 20px; }
```

```
//  width: 256px;

} //  width: 192px; }
  border-right:dashed white 1px;
  border-left:dashed white 1px;

} //  width: 128px;

}
  width: 64px; }
  width:48px; }
  border-right:dashed gray 1px;
  border-left:dashed gray 1px; }
  border-right:dashed white 1px; }
  border-left:dashed white 1px; }
```

```
    49
    6
    246
    118


    00110001
    00000110
    11110110
    01110110
```

```
     M
     Q
     b
     2
     d
     g
     =
     =
```

```
    49
    6
    246


    00110001
```

00000110
11110110

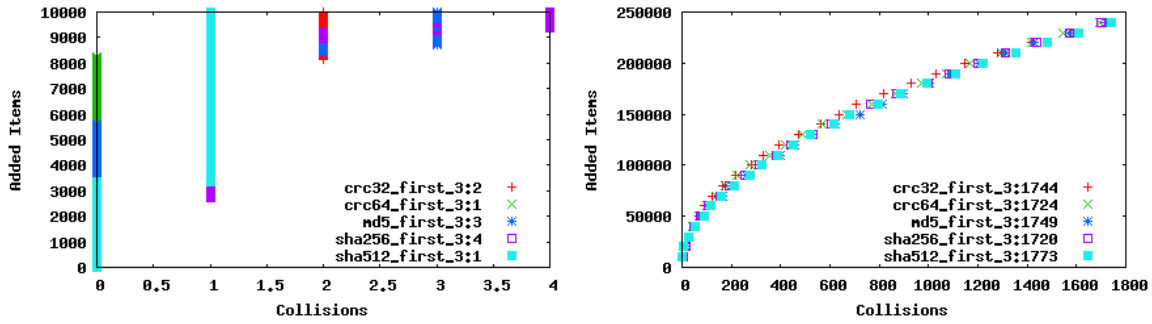
M
Q
b
2

49
6

00110001
00000110

M
Q
Y
=

So how well does 24bit hashes work? Great, the thing about hashes are, if you have good enough entropy they work in an extremely predictable way.



Comparing first 3 bytes different hashes, they all perform very similarly when tested over a huge set, **md5** works very similarly to **crc32**

In a scenario where there is sufficient randomy goodness in the original hash, if you take a smaller slice of the original hash it will preform very similar to other hash slices of the same size over a very large dataset. So toss a coin or choose the least computationally expensive hashing function, and stick with it because it's highly unlikely to get a collision until there are 4000˜ urls, so this is a rather simpler solution to this particular problem.