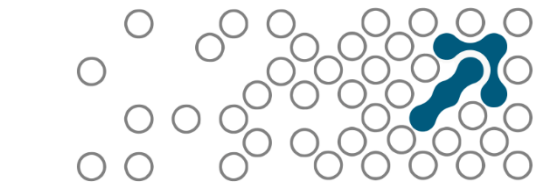
HOW LONG SHOULD YOUR URIs BE?

By: Dave McComb



semantic arts

This applies to URIs that a system needs to generate when it finds it needs to mint a new resource.

I've been thinking a lot about automated URI assignment lately. In particular the scheme we've been using (relying on the database to maintain a "next available number" and incrementing that), is fraught with potential problems. However I really don't like the guid style with their large unwieldy and mostly unnecessarily long strings.

I did some back of the envelop thinking and came up with the following recommendations. After the fact I decided to search the web and see what I could find. I found some excellent stuff, but not this in particular, nor anything that seemed to rule it out. Of note, Phil Archer has some excellent guidelines here: http://philarcher.org/diary/2013/uripersistence/. This is much broader than what I'm doing here, but it is very good. He even has "avoid auto increment" as one of his top 10 recommendations.

The points in this paper don't apply to hand crafted URIs (as you would typically have for your classes and properties, and even some of your hand curated special instances). This applies to URIs that a system needs to generate when it finds it needs to mint a new resource. A quick survey of the approaches and who uses them:

- Hand curate all—dbpedia essentially has the author create a URI when they create a new topic.
- Modest-sized number—dbpedia page IDs and page revision IDs look like next available number types.
- Type+longish number—yago has URIs like yago:Horseman110185793 (class plus up to a billion numbers; not sure if there is a next available number behind this, but it kind of looks like there is).
- Guids—cyc identifies everything with a long string like Mx4rvkS9GZwpEbGdrcN5Y29ycA.
- Guids—Microsoft uses 128 bit guids for identifying system components, such as {21EC2020-3AEA-4069-A2DD-08002B30309D]. The random version uses 6 bits to indicate random and therefore has a namespace of 1036, thought to be large enough that the probability of generating the same number is negligible.

Being pragmatist, I wanted to figure out if there is an optimal size and way to generate URIs.

First, we have a huge advantage, the domain name system as part of the URI means we don't have to worry about the number being unique by itself, just within the domain or subdomain you are assigning.

Auto increment (next available number strategy) works but it has two drawbacks one that Phil Archer pointed out, is that if you need to rerun a process you could easily get duplicates (we've done this on many occasions, when we backed up databases and then reapplied triples that we minted after the restore and saved off line). The other problem is that it creates a choke point for updates, everything has to go through the same routine and lock the next available number while it's writing.

So, how to assign numbers. The airline industry seems to do a pretty good job with randomness (six alpha characters give you 300 million unique confirmation codes), but they may not be relying on randomness alone, if my calculations are right (see below) they'd get a duplicate about every 25K confirmations if that was all they were relying on.

How good does the randomness have to be? I think it needs to be better, say an order of magnitude or two better than the identity problems we are going to introduce in our systems through other means, that we need to handle anyway.

What are these other types of identity problems? Let's say you have a data base of people, and you need to find out if a particular person is already in the database. You do some sort of match spec and either find or don't find the person. Let's say you find the person and then start adding other triples to their record. You may find out later that this person you found in your system is not in fact the same one that presented to you. We have to have systems that can detect this type of anomaly and systems that can correct them.

Let's say it's a health care system. Someone presents and identifies themselves and we begin work on them. At some point we might draw blood send it to the lab and ... discover that it is of a different type that that of the patients record we were updating. The detection part will be partly automated (the system should be scanning for these types of inconsistencies) and partly manual (it may be a human notices in reviewing the chart that this just can't be the same person).

At that point we need to a) assign a new patient ID (the easy part) and b) determine which of the triples that were assigned incorrectly need to be moved over (this is harder, and is clearly an exception process but not impossible). My guess is that this type of misidentification occurs in healthcare somewhere between one in every 10K to 1 Million encounters. My guess is it is much more frequent in areas where identification is less important, such as more traditional commerce.

The point is we have to have procedures to handle a situation that comes up say once every 100K transactions. Once this is in place if we also have to deal with another 1% exceptions that were purely system generated it won't be that big a burden. In other words if our URI assigning system creates a duplicate every 10 million or so records that should be acceptable, as it will represent far less than 1% of the identity errors we encounter.

I wrote this little script to estimate how many URIs I could likely mint before the odds were even that I'd have minted a duplicate:

```
Field = 7.23e13# number to draw from
maxDraws = 100000000 # number of draws
odds = 0
for mintedURI in range (0,maxDraws):

      thisDRAW = (mintedURI) / field
      odds = odds + thisDRAW
      if odds >1 : break

print "the odds are that you will get a collision if you {0:,d}
URIs in a field of {1:,f} ".format(mintedURI, field)
```

…which gave this for the above values:

> *The odds are that you will get a collision if you do 12,024,974 URIs in a field of 72,300,000,000,000.000000.*

Explanation mintedURI is the number of the new URI you are minting (it is 10 if you've minted 10 URIs). If you randomly assign it out of a large "field" in this example 72 trillion possible values, you will get a collision about every 12 million URIs. The logic is as follows:

On the first draw you can't have a duplicate. The second draw has a 1/field chance (on in 72 trillion in this example). The second draw has 2/field (it could collide with either of the two previous), and so on. I'm sure there is a mathematical short cut for this, but I don't know it. I checked this against the famous, what are the odds that two people in a

classroom have the same birthday, and this algorithm suggested 27 as the break even, which I think is right.

So, back to the example, where did I get 72 trillion from?

I like relatively short URIs (there are lots of times you have to look at and deal with URIs). The best way to get a large field with short URI is to include letters in the number. I decided to exclude the ambiguous letters (I in some fonts looks like 1, same with l, o and 0, and z and 2) that gives me 22 letters, 44 if we allow upper and lower case (sure) and 54 if we include digits. I decided to stay away from special characters, all kinds of problems there.

So with an 8 digit character string composed like this you get 72 trillion possibilities. So with 8 characters randomly assigned I should expect a duplicate every 12 million URIs. This number is further reduced as we will be assigning them by major primitive type, so I'd need to have 12 million people, or 12 million events, before I'm likely to get a duplicate from this. My guess is I will already have several hundred from other sources, so I'm willing to do that.

If one in 12 million seems excessive, you could do an optimistic mint collision detection: generate the new URI, and then check to see if it exists. If it did, then you had a collision, create a new one.

So here's a routine to give random URIs:

```
import random
def newURI():

      myChars =
      ['a','b','c','d','e','f','g','h','j','k','m','n','p','q','
      r','s','t','u','v','w','x','y','A','B','C','D','E','F','G'
      ,'H','J',
      'K','M','N','P','Q','R','S','T','U','V','W','X','Y','0','1
      ','2','3','4','5','6','7','8','9']

return ''.join(myChars[random.randrange(53)] for x in range(8))
for a in range(10):

      print newURI()
```

I started to wonder if my calculation were ok, so I build a test harness:

```
import random
uris = set()
def newURI():

      myChars =
      ['a','b','c','d','e','f','g','h','j','k','m','n','p','q','
      r','s','t','u','v','w','x','y','A','B','C','D','E','F','G'
      ,'H','J',
      'K','M','N','P','Q','R','S','T','U','V','W','X','Y','0','1
      ','2','3','4','5','6','7','8','9']

      return ''.join(myChars[random.randrange(54)] for x in
      range(4))

experiments = 100
experimentArray =[]
for trial in range(experiments):

      for x in range(100000):

            y = newURI()

            if y in uris:

                  break

            uris.add(y)

      uris=set()

      experimentArray.append(x+1) # when x is 1 its on the
      second item

print “average”, sum(experimentArray)/len(experimentArray)
print "min", min(experimentArray)
print "max", max(experimentArray)print
```

This routine uses the newURI routine to build 4 digit URIs (don't have enough memory or time to test the millions needed for the 8 character). It runs a set of experiments where I keep adding URIs to a set until it gets a duplicate. My prediction from the above routine was that I should expect a duplicate at the 4,124th URI. When I run the experiments with randomness, I typically averages around 3400-3600, close but puzzling discrepancy.

We're going to adopt the 8 character URI, + a primitive prefix, and do collision detection. We should get a collision about every 100 million writes (depending mostly on how evenly they spread to the different primitives).

11 Old Town Square
Suite 200
Fort Collins, CO 80524

970-490-2224
305-425-2224
info@semanticarts.com