Anonymizing Your Data

Whil Hentzen

Even if you've been living under a rock for the last five years, you're aware that data privacy is one of The Big Big Things these days. You can't go buy a soda at Walgreens without having to sign a half dozen forms acknowledging what they'll do with your data and how they'll keep it safe and only distribute it to a third party in return for a suitcase full of small bills and the express written consent of the National Football League.

You're going to need to keep your customer's data safe, but it's difficult to test an application without data that fairly represents the real world. One solution is to mask a live data set. Doing so has more complexities than may be initially evident, so this article first discusses the business logic behind such an anonymizing program, and second, describes a simple program to perform the actual work.

So it's likely that you've had to work with some sort of confidential data for your customers or users, and you've had to sign a bunch of NDAs and data confidentiality agreements over the last few years. Even then, you don't really want to be messing with live data that contains real social security numbers, names and addresses, and so on.

At the same time, it's unusual for a customer to provide a cleansed data set with made up information that still makes sense, and illustrates the variety of relations and conditions that their live data provides. For any data model past the trivial, it's almost impossible to create a robust enough data set that's consistent and still completely anonymous.

Back in the olden days, I had a default set of test data for common tables like customers, orders, machines, dependents, and payments. I could mock up a simple data set that represented their needs with an hour or two of work, and that was sufficient. The days where systems are that simple are long gone, though, and I've found myself exclusively relying on the customer's data set. Yet I don't want to have their confidential data on my system anymore than they do, so I've taken to anonymizing it and then sending the results to them with beta versions of the application, so that we can see the same results, yet with safe data.

So the problem is we need a data set that accurately mirrors the customer's real data, yet we don't want real values in there. The goal is to wave a magic wand over their live data, turning, for example, live SSNs into fake ones, actual addresses into make-believe, that sort of thing.

Problems, Problems

At first blush, it would seem trivial to do so. Just do a search and destroy on any field that needs to be anonymized. Numbers can be randomized, strings like names can be substituted from a table of source replacements. Not so fast.

The trouble we're going to face is consistency over a non-normalized data set. In a perfect world, your data would be perfectly normalized, 'all the data, just the data, and only the data'. But the world isn't perfect. Just look at the Packers a few years ago, 16-0 and then they lose to New York.

So it's very likely you're going to have a table that is at least in part denormalized. It might looks something like this:

SSN	Name	Code
345-77-1800	Al Anxious	YZ
345-77-1800	Al Anxious	N5
502-16-3451	Barbara Boisterous	01
502-16-3451	Barbara Boisterous	77
502-16-3451	Barbara Boisterous	03
691-35-7012	Carl Calamity	YZ
822-80-0062	Dave Dashing	01
822-80-0062	Dave Dashing	04

where the third column is a product code or plan number of something. For whatever reason, the application wasn't set up to pull the SSN and Name into a separate table and provide a foreign key to that table.

At first blush, one would think one could just rip through each field like so:

```
replace SSN with
padr(alltrim(str(int(rand()*1000))),3,'0')+'-'
padl(alltrim(str(int(rand()*100))),2,'0')+'-'
padl(alltrim(str(int(rand()*10000))),4,'0')
```

However, if you did so, you'd get different SSNs for each instance of the same person. Similarly, one would be tempted to handle a name or address with a lookup replacement, like this:

```
replace fullname with nameLookup(fullname)
```

where the nameLookup() function grabbed a random row from a table of dummy names, perhaps using the original name as a seed, or as a check to make sure the same name wasn't accidentally returned. If done randomly, and the dummy name table was large enough, each row representing Al Anxious could be filled with a different name. Again, the problem is that every instance of Al should be replaced with the same value.

Clearly, using random replacements is just as useful as having those million monkeys at a million typewriters do your data entry for you.

Not All Is Lost

Fortunately, I've found that it's generally just one or a couple of tables that need to be anonymized, as there aren't that many types of sensitive data to be dealt with. For example, if you've got a table listing payments and credits, with foreign keys pointing towards the person or organization attached to those transactions, the transactions themselves can usually be kept whole. After all, what's anyone going to do with a table that looks like this?

PK	Date	Amount
9e8973a414803b84	1/1/2005	197.16
b998ae11577daa37	1/1/2005	202.00
8bsh4-530146394z	1/1/2005	88.18
06538f49834e1ae6	1/1/2005	155.46

So that means we'll likely only have to deal with a few tables at most, and likely only a few columns per table.

And the job gets even easier. I've alluded to the use of the rand() function as a potential tool for randomizing data, and I know there are some of you who are arguing that rand() isn't truly random. Yes, you're right, but that's ok, we don't need a truly random generator, or even a good one.

When you use VFP's rand() repeatedly, you'll get the same results each time. (I've started up VFP and typed

```
? rand()
```

in the Command Window every day for the last week, and always get '0.85' as the first result.

As a result, given the results, you could theoretically work backwards and get the original values. However, you can provide rand() a seed value that changes the results. By using an ever-changing seed, it becomes impossible to reverse engineer the results unless you know how the seed was generated.

Now, on top of this, by using the random value generated to pick a value out of a large table filled with random values, the work needed to produce the original values becomes impractical for data less valuable than military secrets or the recipe for fast food chicken.

Finally, the purpose of this anonymizer is to obfuscate the data such that if a miscreant gets ahold of our data, they can't do anything useful with it. That's the ONLY purpose. In other words, we don't ever have to turn the garbled data back into their real values. The only thing to worry about is that another user shouldn't be able to look at data and reverse engineer

When you need to be able to get back to where you came from, such as decrypting a password, the algorithm has to go both ways, and so you need to be able to track how the randomness was handled. When you encrypt data with the intention of decrypting it later, either you use a mechanism that contains the decnryption method as well, or you include info in the encrypted data itself that allows itself to be unwound. Here, we don't have to do either, since we're never going to decrypt the data. So we can use our everchanging seed and random values in a table mechanisms, because, unlike home, we'll never want to go back.

Types of Data To Anonymize

So let's take a look at the types of data that need to be anonymized. We can group them into three types.

ID Numbers

The first type of data is ID numbers, like SSNs, account numbers, and the like. They may be numeric, alphanumeric, or even include special characters like hyphens and periods. I ran into one company where somebody decided that their

account numbers had to be randomly generated, like passwords, and thus looked like this:

```
33x-z5LTQ$7teM@@
```

Well, whatever floats your boat. But some people are just too clever for their own good.

Anyway, I digress. ID numbers can be anonymized via a one-off cipher, replacing either a character or a group of characters with a replacement that's generated via VFP's rand() function.

Dates

Dates, particularly birth dates, often need to be anonymized. However, they can't simply be randomized, because often the resulting date still needs to make sense in the system. For example, suppose you changed a birth date from 12/14/1954 to 6/17/1999. If the date of death associated in that record was 6/12/1999, that new random DOB is going to gum up all sorts of business logic.

So we'll need to add some intelligence to the anonymizing of a date, so that it get converted to a date that still makes contextual sense. The easiest way to to do is to pass the date plus two more parameters, serving as the minimum and maximum allowable values, to the date anonymizing function. This could be done via date literals, like so:

```
ldDateNew = anonDate(ldDateOrig, ;
  ldDateMin, ldDateMax)
```

where you'd define the dates like so:

```
ldDateMin = {^2000/1/1}
ldDateMax = {^2000/12/31}
```

or with numeric extents, like so:

```
ldDateNew = anonDate(ldDateOrig, ;
  lnDateNeg, lnDatePos)
```

where you'd initialize the extents like so:

```
lnDateNeg = 90
lnDatePos = 90
```

Names

Next to ID numbers, names are the next most important entity to deal with.

For sure, those named John Smith, Juan Carlos, Zhang Wei, or Muhammed Ahmed face a fair amount anonymity already, but what about those with obscure names due to family heritage or unusual nicknames? Let's protect them, too.

Since we're dealing with non-random text strings, we can't just make up a random string of alphabetical characters to replace the original names. Instead, we'll use a table of thousands of first names and last names (separate lists) and pick a random name out of the table. We'll use rand() to determine which row out of those thousands to pull out of the table.

To keep things simple, we'll put both the first and last names in the same table, and use a second column to identify what type of name the value is, like so:

cType	cData
FN	Al
FN	Barb
FN	Carl
FN	Donna
LN	Chang
LN	Johnson
LN	Wilson
LN	Young

Then the anonymizing function call will look something like this:

```
lcFNnew = anonString(lcFNorig, 'FN')
lcLNnew = anonString(lcLNorig, 'LN')
```

The original name is passed to the function so that the function can ensure the same value isn't passed back. The type of data is passed as the second parameter, so that the lookup table can be filtered for the appropriate rows.

Street Addresses

Almost as important as names are addresses. By themselves, addresses don't always provide much personally identifiable information, unless the address is extremely unique. 3701 North 44th Street (without a city) is pretty vague. I just made that up as I was typing the first draft of this article, then did a Web search on it, and got about 312,000 hits, in cities from Albequerque to Zagreb. 1 Knightsbridge Church Drive Northwest, on the other hand, even without a city included, points very specifically to one location.

Additionally, the occupants at an address change from time to time, so, again, are not necessarily as personally identifiable as numbers or names.

Still, if one has the means, why not anonymize them as well? Except for applications that use street addresses for specific uses (such as package delivery), 100 Elm Street could be

replaced by 200 Main Place without any adverse consequences.

Addresses, then, will be anonymized using the same table and function as people's names:

```
lcAddressNew = anonString(lcAddressOrig, ;
  'Addr')
```

Cities

Cities, again, can be handled in much the same way as people's names and street addresses.

```
lcCityNew = anonString(lcCityOrig, 'City')
```

A sample table with a hundred sample first names, last names, street addresses and cities, anonstrings.dbf, is provided in the subscriber downloads for this article.

Additional City Restrictions

In some datasets, there might be restrictions for the City/State/Zip range. For example, I wrote one of those 'calculate the closest store' programs back in the 80s (back when longitudes and latitudes were hard to figure out.) The test data needed to be relatively appropriate; if the user was in Atlanta, they weren't likely going to be searching for a store in Montana.

Another example of location data needing to be geographically relevant is when doing queries based on location, such as population counts per MMSA or metro regions. (My SQLite articles a couple of years ago did just that.)

If these or similar constraints exist in your system, you could pass an additional parameter to the anonString() function that will serve to limit how far afield the anonymizer will be allowed to wander.

Putting the Data Types Together

So we've identified the types of data to anonymize, and we've got functions to call for each of them. We'll create a list of the fields in the table that we want to process, and map them to one of those data types. Then we'll simply pass the real value to the appropriate function and get an anonymized value in return. These three functions are anonID(), anonString(), and anonDate().

The mapping would look something like this:

field	type
ssn	id
fname	fn
lname	ln
addr1	addr
city	city
dob	date
dod	date

Not. Done. Yet.

In the excitement of seeing, you may have forgotten that there's more to this problem than simply doing a

```
replace cCity with anonString(lcCityOrig, ;
  'City') all
```

call. We can't treat each individual row in the table as independent from every other row. Multiple rows may contain the same data, and we have to anonymize all of those rows in the same way, else the data threatens to become meaningless.

Sample Data

A table of sample data to anonymize, zData.dbf, is included in the subscriber downloads. It has a variety of columns, some to be anonymized and some to be ignored. For demonstration sake, I've included mirror columns for the fields to be anonymized, so that running the anonymizing routines on the fields won't alter the original values. The mirror columns bear the same names, but with a trailing underscore. Thus, the sample programs won't

```
replace cCity with anonString(lcCityOrig, ;
  'City')
```

but, rather

```
replace cCity_ with anonString(lcCityOrig, ;
  'City')
```

so that the routines can be rerun without concern.

Handling Data In Groups

I've mentioned earlier the need to map the fields in the table to be anonymized and the type of data. We'll need to include one more file in the mapping, the purpose of which is to identify a unique key for groups of rows that should be anonymized together.

The identification of this key varies in complexity, depending on which field we're processing. For example, let's look at date of birth. We can't just throw a random date into every row, because multiple rows may apply to the same entity, like so:

Name	DOB	Benefit
Al Anxious	1/23/1978	EO7
Al Anxious	1/23/1978	ELA
Al Anxious	1/23/1978	GF12

Simply replacing every row with a random date value would produce results like this:

Name	DOB	Benefit
Al Anxious	12/07/1977	EO7
Al Anxious	10/10/1977	ELA
Al Anxious	2/29/1980	GF12

which is completely wrong. We need to substitute the same anonymized value in each of Al Anxious's rows. However, we can't blindly substitute the same anonymized date value for every row that contains 1/23/1978, since Al may share his birthday with other people. So how do we determine just those DOBs that belong to Al? We could use his name, but there may be others with his name, so it's not a unique key. Furthermore, even if the combination of first/middle and last were guaranteed to be unique, why use three fields if there's a simpler way. In this example, the SSN would be unique and simpler to use.

There are other cases, however, where we may not be able to use a single field as the unique key. For example, the SSN itself can't be used as the PK when doing updates on it; we'll need to use a different field (or set of fields) to identify all records for a single SSN. Those of you who work with medical data likely know that full name plus DOB is often used as a unique identifier.

Thus, the mapping for fields to anonymize looks like this:

Field	Type	PK
ssn	id	firstname+lastname+dtos(dob)
firstname	fn	ssn
lastname	ln	ssn
address	addr	firstname+lastname+ssn
city	city	ssn
dob	date	ssn

Processing the Data Table

We now have all of the pieces needed to begin processing our table of data to be anonymized. The we we're going to do so is spin through our array of fields to anonymize. For each field, we'll grab the unique keys to create the groups of records to process. For each group (such as all of Al Anxious's records), we'll anonymize the values in that field by calling the appropriate function and stuffing the return value in the table.

Here's the start of creating the field mapping array:

```
dimension laFieldTypeKey[9,3]
laFieldTypeKey[1,1] = 'ssn'
laFieldTypeKey[1,2] = 'id'
laFieldTypeKey[1,3] = ;
  'firstname+lastname+dtos(dob)'
laFieldTypeKey[2,1] = 'lastname'
laFieldTypeKey[2,2] = 'ln'
laFieldTypeKey[2,3] = 'ssn'
```

Next, spin through this array of fields.

```
for li = 1 to alen(laFieldTypeKey,1)
```

Find all unique combinations of ssn, using the name+dob expression as the unique key. Since we're going to be passing a variety of field expressions through this routine, we'll need to provide an alias for the key expression in the SELECT.

```
lcCmd = "select distinct " ;
  + laFieldTypeKey[li,3] + " as pk, " ;
  + laFieldTypeKey[li,1] + " from " ;
  + lcNaDBFtoProcess + " order by " ;
  + laFieldTypeKey[li,1] ;
  + " into cursor csrX"
&lcCmd
```

This cursor contains the PK and the value that will be anonymized for that PK. Then spin through the cursor. For each row, get anonymized values for each value by calling the function for that type of data. First, we'll test to see if there is a value, as things tend to blow up when trying to anonymize a blank value.

```
if empty(&laFieldTypeKey[li,1])
loop
endif
```

We'll need to evaluate the field expression contained in the array. Along the way, we'll create variables for the other parts of the expression to update for ease of use.

```
luValToAnon = &laFieldTypeKey[li,1]
luValToKey = pk
luValType = laFieldTypeKey[li,2]
```

In this example, we'll determine which function to call by the field name for simplicity's sake, but this could be generalized easily enough.

```
case inlist(upper(laFieldTypeKey[li,1]), ;
  'SSN')
```

Now that we know to call the anonID() function, we execute the first part of the magic happens here, passing the value to anonymize and the type of data we're passing.

```
luNewVal = l_anonID(luValToAnon, ;
  laFieldTypeKey[li,2])
```

(The function has a 'l_' prefix, since each of the functions is contained in the main anonymizing program.) Now equipped with the anonymized value, it's time to update the appropriate rows in the data table.

```
lcCmd = "update " + lcNaDBFtoProcess ;
 + " set " + (laFieldTypeKey[li,1]) + '_'  ;
 + " = [" + allt(transform(luNewVal)) + "]" ;
 + " where " + laFieldTypeKey[li,1] + " = [" ;
 + alltrim(transform(luValToAnon)) + "]" ;
 + " and " + laFieldTypeKey[li,3] + " = [" ;
 + transform(luValToKey) + "]"
&lcCmd
```

Note the inclusion of an underscore to the name of the field to update (second line of the preceeding code snippet), and the use of brackets as delimiters, since there are likely going to be values with apostrophes in them.

This anonymizing program, anonAtable.prg, is included in the subscriber downloads for this article.

A Closer Look At the Anonymizer Functions

I've talked about the various anonymizer functions, anonID(), anonString() and anonDate() in general terms, but each uses specific algorithms to produce an anonymized value. Let's take a closer look at each.

anonID()

The purpose behind anonID() is to return a random string of characters that match the size and format of the string passed in. The most common example would be a social security number, where '123-45-6789' passed in would result in a random return value such as '738-49-2361'.

In this version, the input string is examined for the number of hyphen characters, and is then broken into chunks. So the sample SSN would be broken into three chunks, 123, 45, and 6789. Each chunk would then be randomized, using a system clock value as the seed, and the three random chunks assembled and returned as a single string.

```
* determine how many chunks
liNumChunks = occurs('-',lcStrOrig)+1

* add a hyphen to the end so that every chunk
* is of the form NNNN- (a trailing hyphen)
lcStrToProcess = alltrim(lcStrOrig) + '-'

for li = 1 to liNumChunks
* break out a chunk by grabbing the leftmost
* characters until reaching a hyphen
lcThisChunk = iif('-'$lcStrToProcess, ;
  left(lcStrToProcess, ;
  at('-',lcStrToProcess)-1),;
  lcStrToProcess)
* remove this chunk from the string to process
lcStrToProcess = strtran(lcStrToProcess, ;
  lcThisChunk+'-', '')
* create a random string out of this chunk
lcNewChunk = ;
  l_rand(lcThisChunk,len(lcThisChunk))
* add the new chunk to the existing string to
* be returned
lcStrNew = lcStrNew + ;
  iif(!empty(lcStrNew),'-','') + lcNewChunk
```

anonString()

The purpose behind anonString() is to pass in a string, such as a first name or a city, and get back a different string. This function doesn't pass back a random character string, for after all, who wants to see test data littered with gobbledegook like this:

First Name	Last Name
KaFJd8sRlsk	UjflszIlefu

Instead, we want random but valid strings. Send in 'Steve' and get back 'Gabriel' in return.

This function takes both the original string and the type of string (first name, last name, address, or city) as parameters, and returns a value from the ANONSTRINGS lookup table. ANONSTRINGS has the following structure.

iRecno	cType	cData
1	FN	Allan
2	FN	Alvin
3	...	
3404	LN	Armstrong
3405	..	
7726	ADDR	All Saints Drive
7727	..	
10558	CITY	Albuquerque
10559	..	

The first thing to do, then, is to determine how many potential values we're going to have to pick from, since that number will determine how we calculate the random number value. (In the data set included with this article, I've supplied just 100 values of each type, but out in the wild, my table includes over between 5,000 and 15,000 values of each type.)

I've actually done this in the calling program, like so:

```
dimension paHowMany[4,1]
select count(*) from ANONSTRINGS where ;
  upper(allt(cType)) == 'FN' into array laX
paHowMany[1,1] = laX[1]
select count(*) from ANONSTRINGS where ;
  upper(allt(cType)) == 'LN' into array laX
paHowMany[2,1] = laX[1]
select count(*) from ANONSTRINGS where ;
  upper(allt(cType)) == 'ADDR' into array laX
paHowMany[3,1] = laX[1]
select count(*) from ANONSTRINGS where ;
  upper(allt(cType)) == 'CITY' into array laX
paHowMany[4,1] = laX[1]
```

So then in the function, we just grab the appropriate value based on what type of string was passed in.

```
case upper(alltrim(lcType)) = 'FN'
  liHowMany = paHowMany[1,1]
case upper(alltrim(lcType)) = 'LN'
  liHowMany = paHowMany[2,1]
case upper(alltrim(lcType)) = 'ADDR'
  liHowMany = paHowMany[3,1]
case upper(alltrim(lcType)) = 'CITY'
  liHowMany = paHowMany[4,1]
```

Now that we know how many potential values are in the lookup table, we can calculate a random record number and frame it to lie between 1 and the number of values.

```
liRecnoToPull = mod(int(rand()*10000), ;
  liHowMany)
```

So, for example, if there were 7820 first names in the ANONSTRINGS table, this would produce a random value between 1 and 7820.

Armed with this number, we'll grab the value from the lookup table.

```
select cData from ANONSTRINGS ;
where uppe(allt(cType))==uppe(allt(lcType)) ;
and uppe(allt(cdata))<>uppe(allt(lcStrOrig)) ;
and irecno = liRecnoToPull ;
into array laStrNew
```

A couple of notes about the SELECT. First, the lookup table actually has a column, iRecno, that contains a fake record number so that we don't have to later rework this logic if we decided to move to a non-record number-aware data source.

Second, the SELECT makes sure that we're not grabbing the same value that we're passing in. Some theorists may argue that doing actually reduces the randomness of the function, but, frankly, out of a set of five or ten thousand values, that's probably not going to help someone break the code. Meanwhile, it's handy to know in testing that the return value will NOT be the same value that we sent in.

anonDate()

The purpose behind this function is to obscure dates that could be used for nefarious purposes, such as birth and anniversary dates. (Indeed, some folks suggest guarding your online privacy by providing a fake date of birth so that in the event of a breach, this piece of critical information isn't left out there, swinging in the wind.)

However, as mentioned, making up a completely random birth date could well impede the testing and use of the system the data is being used in. Birth dates that are set into the future, or after the date of death of the individual in question, will likely screw up something eventually.

The function call,

```
anonDate(luValToAnon)
```

can pass two optional parameters that indicate the earliest and latest values that the new random date must lie between. Without those parameters provided, the function uses hard-coded values of 6 months in both directions.

```
* provide hard-coded bounds
if pcount() < 2
luMin = 180
luMax = 180
endif
```

Conveniently, the parameters can either be actual dates or can be a number of days – the data

type of the parameters will be tested in the function.

```
* luMin and luMax might be dates or # of dates
if vartype(luMin)='D' and vartype(luMax)='D'
  ldDateMin = luMin
  ldDateMax = luMax
else
  ldDateMin = ldDateOrig - luMin
  ldDateMax = ldDateOrig + luMax
endif
```

Now that we have the bounds established, generating new values is simple.

```
ldDateNew = ldDateMin ;
  + int(mod(int(rand()*10000000), ;
  ldDateMax-ldDateMin))
```

As a catch, sort of a belt and suspenders device, if the new date happens to be the same as the original date, just run the function again.

```
if ldDateOrig = ldDateNew
  ldDateNew = ldDateMin ;
    + int(mod(int(rand()*10000000), ;
    ldDateMax-ldDateMin))
endif
```

Purists may choose to turn this into a full scale 'do while not' construct.

Enhancements

As with any program that you've worked on for a while, this one has the potential for several enhancements.

anonID()

An original design parameter was to handle alphabetic strings as well as numerics, so that an ID of the form

```
123ABC
```

could be anonymized. As I've worked with this function, I never found the need to anonymize an alphanumeric string, so that piece never got written. It wouldn't be difficult, and could conceivably be useful in other environments, so it's the first ER on the list.

The current function has extremely limited formatting, expecting just numbers from 0 to 9 and possibly hyphens. It's conceivable that other separating characters could be passed in, such as periods, underscores, or octothorpes. On the other hand, those characters might be part of the data I the string, so it wouldn't be wise to simply assume they're separators, and then hard-code traps for them.

A better way would be to send a second string into the function that defined a character map, much like Fox's format strings.

'9' would represent a number to be randomized, 'A' would represent an alphabetic character to be randomized, an X would represent an alphanumeric character to be randomized, and a hyphen represents a separator that is not to be randomized. Here are some sample character mappings:

Sample Map	Meaning
999-99-9999	3 numeric, a separator, 2 numeric, a separator, 4 numeric
AAA999	3 alphabetic, 3 numeric
XXXX-99	4 alphanumeric, a separator, 2 numeric.

The third possible enhancement might be to pass a seed to the randomizing function, to further add to the entropy of the data.

anonString()

For addresses, add a number and direction to the address.

For cities, make the city closer to the target, if needed to make subsequent calculations with the data relevant. If the system is doing that 'closest store to you' process, it'd be a pain to have your carefully assembled data set of local stores to be suddenly scattered across 20 time zones.

anonDate()

Could fine tune the range of dates, say, to be within the same year. An insurance policy that was taken out in one year and terminated in another might need new dates that had the same year as the originals.

Source Code Notes

anonStrings.dbf – master lookup table that contains a hundred random first names, last names, addresses, and cities.

zdata.dbf – sample table of names, addresses, IDs, dates, and non-anonymizing data to practice on.

anonAtable.prg – the anonymizing program and subroutines.

Author Profile

Whil Hentzen is an independent software developer based in Milwaukee, Wisconsin (as opposed to Milwaukee, Minnesota, as many people think.) His writing has killed many trees over the years, but none since 2007. He has realized he really sort of misses it. You can reach him at whil@whilhentzen.com