Paper 132

Linking Medical Records to Medics in Cyberspace

Sigurd W. Hermansen, Westat, Rockville, MD, USA

ABSTRACT

In the brave new world of Web registries, the National Provider Identifier (NPI) supposedly links providers of medical services to a registry of medics. When electronic medical records (eMR's) include accurate NPI, using the NPI to look up a medic's personal data on the NPI registry (http://www.npi-search.com/) works seamlessly. Without an accurate NPI, searches become more complex and less reliable. The downloadable NPI registry database turns out to be an exceedingly complex and difficult target for searches. Fortunately, the SAS System provides all of the tools that we need to uncompress, restructure, index, and search the NPI registry.

KEYWORDS

Web registry, compress, csv, pipe, database, identifier, structure, restructure, view, index, search.

INTRODUCTION

Big data on the Web give researchers an open door to rich and extensive information resources. The Jurassic era of information technology has ended in an explosion of Web servers connected over high speed networks to desktop and mobile browsers. But downloading megabits in a second is one thing, while extracting useful information from really big data is another thing entirely.

If one takes a close look at some of the very large collections of data that government agencies, universities, research consortia, and corporations are uploading to the Web or distributing on DVD's, one discovers a distressing lack of useful metadata and documentation, and a curious mixture of text files of wide mainframe records with repeating groups of fields; xml-tagged records with metadata buried in style sheets; delimited worksheets; spreadsheet workbooks; and zip archives of files all linked at the same level by one key. One has to wonder if the developers have never heard of E. F. Codd and his standards for database development.

Before discussing a specific Web registry, it seems important to mention that the current Tower of Babel standards for databases on the Web have created a cottage industry of service bureaus specializing in guiding others through the process of accessing big data. Should we suspect that the principals in these service bureaus had some role in developing one or another of the architecture of large collections of data on the Web?

NATIONAL PROVIDER INDENTIFIER (NPI)

A provision of the Health Insurance Portability and Accessibility Act (HIPAA) of 1996 requires medical service providers in the US to obtain an NPI. The acronym NPI refers both to the medical provider ID and to the downloadable data that index each provider's attributes to an NPI. The Web page for the monthly National Plan and Provider Enumeration System (NPPES) downloadable NPI data file (http://nppes.viva-it.com/NPI_Files.html) has this caveat:

"Please be sure to read the Facts about the Downloadable File before attempting to download the actual file. The NPPES Downloadable File is very large (exceeds 4 GB) and is intended to be downloaded by individuals with the requisite technical expertise."

Fair warning. The documentation describes the file as a compressed, comma-separated variable (csv) file. From that description, we know that we won't be able to stream data from a compressed data file as we would an ordinary text file. Instead, we'll have to stream data from the file in a SAS DATA step using a FILENAME statement with a device-type of PIPE. An example appears in the inset below. The –c, -a, and

SAS Solution: Read Big Data from a compressed csv file.

FILENAME: PIPE

syntax

```
FILENAME <filename> PIPE ….
```

examples

```
filename NPIPipe PIPE " gunzip -c -a -q /user/path/file.zip *.csv ";
```

caveats

The gunzip command is available as a Linux shell command, but may have to be installed or replaced with another command under MS Windows or other operating systems.

–q "switches" specify streaming of text to standard output (where SAS will read data as if reading a file), ASCII data, and quiet mode respectively. The beauty of the PIPE in a FILENAME statement is that it separates out the details of reading data from a device-type and makes operating system commands independent of a SAS DATA step that reads or writes data from or to one or another device-type.

CONVERTING THE COMPRESSED NPI CSV FILE TO A SAS DATA SET

SAS holds off on invoking the gunzip command until the compiler sees the file reference NPIPipe in a DATA step INFILE statement. The INPUT statement begins streaming data into the SAS program data vector once the INFILE statement tells the SAS compiler what to expect: data records with end of line (EOL) markers, column values delimited by commas (DELIMITER), records that may have missing variable values at the end of a record (MISSOVER), two consecutive delimiters denote a missing value and delimiters contained within quotes don't count as delimiters (DSD), skip records at the beginning of the file (FIRSTOBS=), and assume an EOL if the line length exceeds a certain number of characters (LRECL=). The LENGTH statement declares variables that will be assigned values and their maximum lengths, and the INPUT statement defines the order in which they will be read from comma-separated text strings. The large number of variables in the csv file gives us a repetitive typing exercise that most would prefer to avoid. By reading all of the variables in sequence from a compressed file and writing them to a compressed SAS

SAS Solution: Converting a csv file to a SAS data set .

SAS Data Step: INFILE … INPUT …

syntax

```
DATA <DATASET NAME> INFILE <FILENAME
REFERENCE>
    DELIMITER=',' MISSOVER DSD; …. INPUT ….
```

examples

```
DATA LnxNPI.NPI (compress=yes);
    INFILE NPIPipe
           DELIMITER=','
           MISSOVER
           DSD
           LRECL=32767
           FIRSTOBS=2
       ;
    LENGTH
           npi $ 10
           entity_type_code $ 1
           replcmnt_npi $ 10
           ein $ 9
           prov_org_nm $ 70";
   ....
      ;
    INPUT
           npi $
           entity_type_code $
           replcmnt_npi $
           ein $
           prov_org_nm $
           prov_lnm $
   ....
```

caveats

LRECL option not required on IBM z/OS machines.

data set, we can limit the storage requirements to around a 400 MB source and a 276 MB SAS data set. The SAS data set must, of course, have variables that match SAS requirements. The INPUT process assigns variable names that are free of banned special character and no more than 32 characters long. The NPI-documented variable names look more like labels than names in that they include spaces (sometimes two instead of one) and a variety of special characters. Although legal names for variables as well as files seem a matter of taste, I tend to prefer names that have some descriptive value, but are also easy to recall and free of special characters that make typing them difficult. Whatever the form of the variable name, however, I would see the 329 variables in the NPI as too many.

DESIGN STANDARDS FOR THE NPI

The documentation of the 329 variables that accompanies the cyberspace download of the NPI says volumes about the structure of the database. Each record has values arising from different dimensions of interest. Representing multiple dimensions of reality in a "flatfile" seem akin to artists' illusions of three dimension perspectives on two-dimensional surfaces: good artistry, but not a good way to represent real entities and events in the context of space and time. One group of variables represents at least somewhat permanent attributes of providers, while others contain multiple addresses and other contact information, repeating groups of medical credential and taxonomy variables, and NPI dates and status variables. A group of "other provider ID" variables repeats fifty times to accommodate an arbitrary maximum number of instances. All this increases the extent to which programmers have to spend time managing and documenting variable names.

Database design standards include minimal levels of database complexity and physical footprint that can fully support content and access. Separating out repeating values into a separate table, when keyed with the NPI and a sequencing variable, would reduce the number of variables by almost thirty times in the other provider ID group to six variables. Replacing two hundred variables with five reduces the total number of variables by half. Even more, restructuring the NPI to reduce the number of variables would combine many variables with the same set of possible values (domain) into one variable. Something as basic as standardizing spellings of names of other issuers of ID's (e.g., Blue Cross Blue Shield, BC/BS, BCBS, …) becomes far easier when dealing with one domain instead of fifty.

Proper database design minimizes the difficulties associated with data redundancy (say, failures to update or delete all instances) and missing values (say, distinguishing missing values by definition from omissions). From a broader perspective, we are looking for *data independence*. This means that keys embedded in data link related data items, not somewhat arbitrarily chosen variable names associated with pointers in lookup tables. Linking by matching key values in data makes it possible to transfer data from one platform and application environment to another with no loss of information. This higher standard for databases on the Web may not eliminate the requirement of technical expertise for those interested in using databases such as the NPI, but it would support automated access to data and improve the quality of information derived from them.

The next section demonstrates methods for reducing the complexity of the NPI and working toward data independence. None of these methods require the SAS System, but SAS certainly makes database restructuring easier.

RESTRUCTURING THE NPI

Sensible database design and architecture calls for "normalizing" databases, in part by replacing repeating groups of the same variables with a separate, "child" table of one row per group. A group of rows replaces a group of variables. The key (NPI) and the variable group number in each subsidiary table make this restructuring lossless in that joins or merges of the subsidiary tables back to a main table can reconstruct the original NPI file structure.

In SAS, the first step in normalizing the NPI entails simple partitioning of the NPI into multiple tables (data sets). A sketch of an expanded DATA statement of the earlier INFILE … INPUT program illustrates the method (see inset).

This first step in the restructuring process roughly partitions groupings of columns in the original NPI "flatfile" into data sets representing different dimensions. (The first data set will be output only if the entity_type_code indicates a group NPI record.) When run within a SAS Macro, repeating groups of variables with an additive index distinguishing variables in one group from those in other groups, as in the NPI, a simple Macro loop saves on typing. The specification of the output data set LnxNPI.NPIotherID shows how this works. Other data set options, such as compress=yes or rename=-, may also be included within the parentheses. Even so, simply partitioning the NPI will neither reduce data volume nor normalize the database. The log of the partitioning process does, nonetheless, give us useful hints about the extent to which data volume will decrease as a side-effect of normalization.

```
NOTE: 3717915 records were read from the infile NPIPIPE.
      The minimum record length was 0.
      The maximum record length was 3901.
NOTE: The data set LNXNPI.NPISAMPLE has 14105
      observations and 329 variables.
NOTE: Compressing data set LNXNPI.NPISAMPLE decreased
      size by 95.85 percent┬..
NOTE: The data set LNXNPI.PROVIDERS has 2825229
       observations and 25 variables.
NOTE: Compressing data set LNXNPI.PROVIDERS decreased
      size by 85.56 percent.
      Compressed is 7286 pages; un-compressed would
        require 50451 pages.
NOTE: The data set LNXNPI.BUSINESSCONTACT has 2825229
        observations and 21 variables.
NOTE: Compressing data set LNXNPI.BUSINESSCONTACT
        decreased size by 72.82 percent┬..
NOTE: The data set LNXNPI.NPISTATUS has 2825229
        observations and 6 variables.
NOTE: Compressing data set LNXNPI.NPISTATUS decreased
        size by 13.64 percent┬..
NOTE: The data set LNXNPI.NPIOTHERID has 2825229
        observations and 201 variables.
NOTE: Compressing data set LNXNPI.NPIOTHERID decreased
        size by 99.00 percent┬.
NOTE: DATA statement used (Total process time):
      real time           5:17.08
      CPU TIME            4:22.28
```

The Data step populates the last data set, NPIOTHERID, by looping through a repeating group of variable name "stems" and appending an index value to each name. The repeating variable group structure seems substandard from the onset for a database because it has to be extended to larger index values or it will drop information. An adverse side-effect on data volume seems typical as well. After separating out the fifty groups of variables into NPIOTHERID, SAS compression reduces the space required to hold the dataset by 99 percent. While SAS compression often reduces data sets to fractions of original sizes, this scale of reduction suggests that the vast majority of the variables in the NPI download are empty.

If the NPI follows the usual convention that a missing value in the first variable of a repeating group of variables means

SAS Solution: Restructure NPI (1).

SAS Data Step: DATA options

syntax

```
DATA <DATASET NAME 1> (<OPTIONS>)
     <DATASET NAME 2> (<OPTIONS>)
     ....
```

examples

```
  DATA LnxNPI.NPIGroup
     (compress=yes)
     LnxNPI.Providers
     (compress=yes
      keep=
          NPI
          prov_lnm
          prov_fnm
          prov_mnm
        ....
      )
     LnxNpi.businessContact
     (compress=yes
     keep=
          NPI
          prov_busns_mail_addr1
          prov_busns_mail_addr2
          prov_busns_mail_addrcity
          prov_busns_mail_addrstat
        ....
      )
     ....
     LnxNPI.NPIotherID
     (compress=yes
      keep=
          NPI
      %do __n = 1 %to 50 ;
          othr_prov_ident&__n
          othr_prov_identcode&__n
          othr_prov_identstate&__n

    othr_prov_identissuer&__n
      %end;
     );
     ....
```

Caveats

SAS Macro loops must be run within a SAS Macro.

that all other variables in that group have missing values as well, we can block writing of the row containing a repeating group of missing values. In effect, we are removing rows in which all variables except the NPI have missing values:

SAS Solution: Remove groups of variables with missing values.

SAS Data Step: DATA options

syntax

```
IF <condition> THEN OUTPUT <dataset>
      ....
```

examples

```
if entity_type_code = '1' and NOT missing(hlth_prov_taxono1)
   then output LnxNPI.NPItaxonomy ;
if entity_type_code = '1' and NOT missing(othr_prov_ident1)
   then output LnxNPI.NPIotherID ;
if entity_type_code = '1' and NOT missing(Health_Prov_Tax_Group_1)
   then output LnxNPI.NPIHealthProvTaxGroup ;
```

Caveats

None

The absence of a row of repeating values for an NPI implies that those values are missing for the provider represented by the NPI. Not writing empty rows to a table will reduce the number of rows, sometimes dramatically. Although the 1st table containing repeating groups of variables (NPITAXONOMY) has at least one variable with a non-missing value in each row, less than half of the NPI have at least one value in the NPIOTHERID table, and only 1,190 NPI have at least one value in NPIHEALTHPROVTAXGROUP:

```
NOTE: The data set LNXNPI.NPITAXONOMY has 2825228 observations and 61 variables.
NOTE: Compressing data set LNXNPI.NPITAXONOMY decreased size by 91.07 percent.
      Compressed is 5487 pages; un-compressed would require 61419 pages.
NOTE: The data set LNXNPI.NPIOTHERID has 1133405 observations and 201 variables.
NOTE: Compressing data set LNXNPI.NPIOTHERID decreased size by 98.40 percent.
      Compressed is 1513 pages; un-compressed would require 94451 pages.
NOTE: The data set LNXNPI.NPIHEALTHPROVTAXGROUP has 1900 observations and 16 variables.
NOTE: Compressing data set LNXNPI.NPIHEALTHPROVTAXGROUP decreased size by 73.68 percent.
      Compressed is 10 pages; un-compressed would require 38 pages.
NOTE: DATA statement used (Total process time):
      real time           4:57.43
      CPU TIME            4:12.38
```

The size of the data set NPIOTHER decreased less after compression by a fraction of a percent. Why? Simple, really. The number of rows halved after removing empty rows, leaving a higher proportion of non-missing data

Now, once we have separated out repeating groups of variables into subsidiary data sets, the next step of the database restructuring process entails stacking each group of n variables into n columns keyed by the NPI and a sequence number. So, under the heading VarA, …,VarX, we stack VarA1,…,VarX1; VarA2,…,VarX2; …. To identify and differentiate each row in the stack, we retain the NPI and put the index number of the group into a separate variable (seq). The NPI and seq variables together serve as the composite index key to each row. Each row has a distinct key value. A distinct key lets us find any one group of a repeating group of variables containing non-missing values. If not found in a search for a value of the distinct key, we can infer that all the variables in that instance of the group had missing values:

SAS Solution: Stack Repeating Group Variables from Same Domain

SAS Data Step: DATA options

syntax

```
(SELECT … ) UNION CORR (SELECT …) UNION CORR …
      ....
```

examples

```
%macro stackVarGrps;
    proc sql;
         create table WinNPI.NPIotherIDs (compress=yes) as
         select * from
         (
            (select     NPI,' 1' as seq,othr_prov_ident1 as othr_prov_ident,
                    othr_prov_identcode1 as othr_prov_identcode,
                    othr_prov_identstate1 as othr_prov_identstate,
                    othr_prov_identissuer1 as othr_prov_identissuer
              from WinNPI.NPIotherID where NOT othr_prov_ident1 IS NULL)
       %do __n = 2 %to 50;
                     union corr
            (select      NPI,"&__n" as seq,othr_prov_ident&__n as othr_prov_ident,
                    othr_prov_identcode&__n as othr_prov_identcode,
                    othr_prov_identstate&__n as othr_prov_identstate,
                    othr_prov_identissuer&__n as othr_prov_identissuer
                    from WinNPI.NPIotherID where NOT othr_prov_ident&__n IS NULL)
       %end;
            )
         ;
      quit;
%mend stackVarGrps;
  %stackVarGrps
```

Caveats

SAS assigns the variable labels in the first dataset to the variables in the new "stacked" dataset.

Here, as before, a simple SAS Macro loop takes care of the tedious task of writing almost the same subquery, one for each group in the repeating groups of variables. In this example, the program stacks over three million rows with six variables in each. After the stacking operation, the new NPIOTHERIDs data set actually occupies about one-third more disk space than the original NPIOTHERID data set (both compressed). The key index in the stacked data set repeats the NPI and adds the seq variable, but having the key index makes the extra space worthwhile. We can use the key index to count the number of other ID's any one provider with an NPI may have, or to search for all instances of other identifiers issued by, e.g., the State of Kentucky for a list of provider NPI's. Less obvious but perhaps more important, selecting or summarizing values of a few

SAS Solution: Reconstruct NPI (2).

SAS PROC SQL: JOIN

syntax

```
SELECT … FROM … <dsn> … JOIN <dsn> … ON
```

examples

```
proc sql;
 create view WinNPI.vwJoin as
 select coalesce(r1.NPI,r2.NPI) as NPI,
        seq,othr_prov_identissuer
 from WinNPI.Providers as r1
      inner join
          WinNPI.NPIOtherids as r2
          on r1.NPI=r2.NPI
 where NOT othr_prov_identissuer IS NULL
 order by NPI,seq ;
quit;
```

caveats

Altering database structures or data updates may lead to errors or affect the yield of a stored view.

variables helps us improve data quality. For example, when we stack all values of othr_prov_identissuer (the label that identifiers the issuers of other ID's), a summary of that variable displays obvious discrepancies such as "Johns Hokins" instead of "Johns Hopkins", "Kaier" and "Kais" instead of "Kaiser", and "Antem", "Antehm", "Antham", and "Anthlem" (to name a few) instead of "Anthem". Many of the low-frequency instances of label values look suspicious.

Restructuring a complex file structure or database to achieve data independence often leads to a better understanding of content and structure. Partitioning complex files into simpler tables, stacking multiple instances, and indexing help programmers write more reliable summary and search programs in less time.

JOINS ON KEYS AND VIEWS

When partitioned into tables with linkage keys, complex flatfiles or sections of these files can be reconstructed into their original form, or alternatively into almost any structure required by analysts or clients. Typically programmers extract subsets of big data such as the NPI. The *Reconstruct* inset (above) displays a typical JOIN of two tables derived from the NPI: the Providers table that contains all NPI (the identifiers), and the NPIOtherids that we created to stack other provider ID's and issuers of those ID's. JOINs may also link tables derived from the NPI to lists of provider names, credentials, and other attributes that could constrain the selections from a restructured NPI to providers of interest.

A restructured NPI would not have to be limited to tables linked on the NPI (the identifier). Tables containing values from the same domain in different column variables can be "decomposed" into two or more tables linked on new keys.

The example of a JOIN actually creates a *stored view* of the database; that is, a query that generates a predefined tabular structure (a base table or a report) from data currently in the database. Big database documentation could include a variety of view definitions that would reconstruct repeating groups of variables or other structures that analysts or clients may prefer.

INDEXES AND SEARCHES

Were searches of the NPI and other big data limited as a rule to linkage to one huge file of records on a single key, ordering data by that key would effectively solve the search problem. Searching on different keys, say, names, credentials, or organizations as well as NPI, makes sorting and searching strategies impractical. Pre-defined "standing" indexes speed up searches by "inverting" the order of pointers to records to the order of the key values. For example,

IndexValue	Record	Key
1	95888	Aanseld
2	00086	Abner
.	.	.
.	.	.

The index virtually orders the pointers to records in the sort order of the key values. The pointers give a program direct access to the records in the desired order.

Although database systems support building, storing, and updating of standing indexes, providing these services seems well outside what we might expect in a Web database supplier. Instead, given appropriate key values in data as part of data independence, we can build indexes dynamically to support any number of searches of a Web database. At this stage it should not come as too much of a shock to learn that SAS supports highly efficient dynamic indexing of big databases. As in other SAS Java object programs, building the index begins with a quick look at the header of the data set being indexed (if 0 then set <dsn>;), followed by declaring (dcl) the hash object (hg) and naming the data set. The "multidata" option (new in SAS V9.2) indexes all instances of a key value and related data, not just the first instance. The hg.define… methods specify variables in the key and related data. That's it for the index (see Dynamic Indexing inset below).

SAS Solution: Dynamic Indexing

SAS Data Step: Hash Object

syntax

SAS Hash Object methods dcl hash, .definekey and .definedata

examples

```
proc sql;
 create table Providers as
 select NPI,prov_lnm length=6 as prov_lnm, prov_fnm length=4 as prov_fnm,
        prov_mnm length=1 as prov_mnm,prov_othr_lnm length=6 as prov_othr_lnm
 from LnxNPI.Providers   quit;
 data viewhashMatches/ view=vwhashMatches ;
   if 0 then set Providers ;
   dcl hash hg (dataset:'Providers',multidata:'y',hashexp:16) ;
     hg.definekey   ('prov_lnm','prov_fnm') ;
     hg.definedata  ('NPI','prov_lnm','prov_fnm','prov_mnm','prov_othr_lnm');
     hg.definedone () ;
   dcl hash hh (dataset:'Providers',multidata:'y',hashexp:16) ;
     hh.definekey   ('prov_othr_lnm','prov_fnm') ;
     hh.definedata  ('NPI','prov_lnm','prov_fnm','prov_mnm','prov_othr_lnm');
     hh.definedone () ;
   do until (eof2) ;
        length lastname $ 6 firstname $ 4 midname $ 1 ;
      set LnxNPI.ProviderStationNPIname end = eof2 ;
                Prov_lnm=substr(scan(ProviderName,1,','),1,6);
                Prov_othr_lnm=Prov_lnm;
                Prov_fnm=substr(scan(scan(ProviderName,2,','),1,' '),1,4);
                Prov_mnm=substr(scan(ProviderName,2,' '),1,1);
                lastname=Prov_lnm;
                firstname=Prov_fnm;
                midname=Prov_mnm;
          rc = hg.find ();   if (rc = 0) /* Success. */
              then do; output ;
                       hg.has_next(result: r);
                       do while(r ne 0);
                          hg.find_next();
                          output;
                          hg.has_next(result: r);
                       end;
                   end;
          rc = hh.find () ;  if (rc = 0) /* Success. */
              then do; output ;
                       hh.has_next(result: r);
                       do while(r ne 0);
                          hh.find_next();
                          output;
                          hh.has_next(result: r);
                       end;
                   end;
     end;
 run ;
```

Caveats

Requires substantial memory for hash object index.

The `do until` block that comes next reads another data set and uses the `hg.find ()` and hg.find_next () methods to search for first (if any) and subsequent matches to the key and to output any found.

Extensions of this simple example willbuild multiple indexes and search each index for matches to different keys. A further extension defines data as a pointer to rows in the indexed data set instead of the data values that come from those rows.

CONCLUSION

Big data on the Web offer the promise of fuller and better access to large collections of administrative and factual data. We have observed here that making the promise a reality will require better methods for transporting data from one environment and reconstructing those data into informative databases capable of supporting meaningful reseach. If we don't take reasonable precautions, database transports over the Web may, as in the movie **Spaceballs**, leave us after reconstruction with misfit parts. Instead, if we make good use of available data compression/decompression methods, adhere to good database design standards, and restructure Web databases to facilitate data selection, summary, cleaning, and indexing, we will be able to use data from the Web in ways that will change our lives for the better. The SAS System provides tools in one package that programmers and analysts can use to make better use of big data on the Web.

REFERENCES

Codd, E. The Relational Model for Database Management (Version 2 ed.). Addison Wesley Publishing Company. Boston (1990). ISBN 0-201-14192-2.

P. Dorfman, G. Snell. Hashing: Generations. SUGI 28, Seattle, WA, 2003.

R. DeVenezia. Greetings from the Edge: Using javaobj in DATA Step, Montreal, Canada, 2004 Paper 033-29 SUGI 29

R. Ray, J. Secosky Better Hashing in SAS® 9.2, SGF, San Antonio, TX, 2008 Paper 155-31 SUGI 31

ACKNOWLEDGMENTS

Peter Eberhardt SESUG 2012 academic chair and Producer of the Transporter Room, has stretched the boundaries of the Users' Group conference. My fellow Transporters, Paul Dorfman and Richard DeVenezia, introduced hash indexes and Java objects to SAS programmers, and Robert Ray and Jason Secoskyof SAS Institute extended basic hash objects to multidata searches. Mike Rhoads reviewed and edited the paper to bring it up to Westat standards and beyond. He and Carina Tornow suggested improvements in content and presentation; despite their best efforts, the author has sole responsibility for any errors or oversights.

DISCLAIMERS

The contents of this paper are the work of the author and do not necessarily represent the opinions, recommendations, or practices of Westat.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Sigurd W. Hermansen
Westat
1650 Research Blvd.
Rockville, MD 20850
Work Phone: 301.251.4268
E-mail: hermans1@westat.com