Paper #: 3012

Shell Script Data Base Modeling

F. John Kluth
National Machine Company
4880 Hudson Drive
Stow OH 44224-1799
330-688-6494

In spite of it's somewhat cryptic nature, the Bourne shell script of UNIX presents a number of tools and opportunities for modeling data structures that will be quite useful in a number of common situations. I can list three situations from my own experience. The first situation involves the porting of data. When programs and data are ported from one platform to another, some data bases typically will not move cleanly and must be modified. This is especially true where more than two platforms are involved and data must be combined. The second situation involves a disaster were the computer is destroyed and the system must be brought back from tape. The programs written in the shell script come back first, and are usable as soon as the system is restored. Encryption key problems may prevent the use of vended software for days. After the disaster, users will want to know the state of the computer before the disaster. Though this information is on tape, it cannot be loaded because the old data would need to replace active data, at a time when active data is vitally needed by the organization. A good practice would be to maintain "prints to file" outputs of key programs that would be manipulated with the shell.

A third situation arises when data is being generated but no information about the nature of the data is available. The shell can record this data in whatever form seems suitable, then later this data can be restructured with shell tools as more information about the structure becomes available. The shell script is not used as a full development tool, but rather as an intermediate tool that allows a better understanding of data structures and paths. The particular value of the shell script is that it allows the development of flexible structures that can be easily modified with feedback loops that involve actual manipulation of the data. This third situation will be examined in some detail using the Bourne shell because of its wide availability.

A top down approach to data base design is much to be preferred, but it is not always possible. A top down approach has a number of advantages. It provides a unified goal to which the various parts of the project can be fit. It takes advantage of the better organizational visibility from the top. The commitment of top managers to a project is helpful mainly for this reason. A project which fits major goals of an organization are more likely to be supportive of the organization.

In spite of the truth of these statements it is sometimes not possible to take a top down approach. A crisis may prevent the involvement of top management so that lower level personnel must fend for themselves. In this case one needs a more hypothetical approach. Top level goals are hypothesized, and lower level activities are designed to support these goals. The effectiveness of the activities are then observed. If the activities lack effectiveness, then new goals are hypothesized and the activities must be changed accordingly. This process can work even if there is little or no organizational visibility. What should be done in this case is to record all relevant data coming in as well as requests for data going out. Typically the data that is recorded should be the data that has been of interest in the past, plus whatever will satisfy known requests for data. It is desirable at this point to be creative as to what might be needed. There is strong resistance for storing all data simply because of the amount of data associated with every conversation, phone call, and piece of junk mail. It is better to store too much, than too little, because without the data, patterns will be lost.

This process of focusing only on what is coming in and going out is referred to as the black box approach. Ideally the structure of a data base is determined by the nature of the questions that are asked of it. In the black box approach both the data and the questions should be stored in some way. The questions asked must be analyzed for their value of predicting future questions. The data structure may then need to be modified to accommodate new types of questions that are being asked. Typically, each question has its effect, and the data base may be redesigned to accommodate the new questions. A question may not be literally recorded, but its record remains as the changed structure of the data base.

The UNIX shell script supports this black box approach to data management because applications can be rapidly mounted and rapidly modified. When the data base becomes large and established, then other tools should be used as their appropriateness is determined by the data structures that become evident. Unix shell script is not going to be efficient with data analysis in situations where applications exist to deal with the data. Shell script will be useful where applications do not exist, or where data structures cannot be identified. Shell script allows immediate data entry, and then many shell tools are provided that allow manipulation of the data structure to a more convenient form. When the desirable form for the data is identified, a program can be produced to increase the data base size and increase the convenience of the data and speed of access. The UNIX shell script allows this to take place by substitution by function, or as a whole. Since the purpose of the shell is to run programs, a more suitable program can replace all, or part of the shell script.

A model is a structure which contains the relationships of a real structure, but in a more convenient form. In the data base area modeling or more specifically, data modeling refers to the use of diagrams or charts to graphically represent data flow and structure. CASE tools purport to automatically convert these diagrams into executable code. Flow charts were once widely touted as an easy graphical way to understand software programs. Such modeling is, no doubt useful, but it has its limitations. Such diagrams work well when the project is small, but as the project grows in complexity, The diagrams become more difficult to manage. Multi-dimensional projects are impossible to model, because the two dimensional aspect of the diagram fails to provide more than a cross-section of what is going on. To verify the model a parallel structure of code must be constructed. Only when the data is pumped through the code can the model be verified.

The UNIX shell script provides modeling, but not by diagrams. What the shell provides is a logical structure to represent data flow and structure. By applying the logical structure of the shell to the data flow of the computer, a situation can be set up to move data in the same way as a real world situation. The shell works as a modeling tool because it has a full set of flow control structures, complex expression matching capability, and the ability to provide code that is modular at several levels. The logic is well structured so that modularity is enhanced and documentation is minimized. The quick access to files and their manipulation adds to the development power.

The purpose of the shell is to run application programs. Shell scripts provide a logical ordering of these programs and their execution. This is a key to shell script modularity. The shell script is composed of programs that execute, and it is composed of such programs. A shell script can be composed of other scripts that isolate functionality of a program. These sub-program scripts can be replaced by compiled or interpreted programs in other languages such as C or PASCAL. The shell script can then define the functionality of the program that will allow other programs to be written that are more efficient. The shell script has the advantage that typically programs are written and executed much faster, but they are neither as precise nor as efficient as programs written in standard data base management languages. The efficient approach seems to be to define the basic structure of the data base application with a process that uses shell scripts, then replace these scripts with data base language or a complied program once the structure is known.

Typical data bases are fixed length fields so that records are of equal length. This allows the location of individual records to be calculated. Indices point to the results of these calculations. Shell scripts lack the ability to reference records by indexed location. Typically files must be searched from beginning to end to locate records. Formerly, the access times of such data bases caused severe limitations. Contemporary machines can search hundreds of thousands of bites in less than a second, so this is less of a handicap now. Since the whole file is searched, some indexing on particular fields is not necessary. Standard shell data bases are character delimited with either the tab character or the space character being the default character delimiter. The delimiting character can be changed by modifying the IFS variable (internal field separator), however programming is somewhat simpler if it is not. For small data bases fixed field files are useful for their simplicity. If the record can fit on the screen then formatting is not required and printouts are particularly simple. Care must be taken with shell script commands used with these data bases because typically extra white space will be removed and the formatting lost.

Typically, the difficulty with data base projects is not the type, speed, or efficiency of programs; it is the availability of good data. Data that the computer delivers in response to questions is not going to be any more accurate than the data that is entered. The shell program awk provides rather elaborate tools for analyzing data and formatting output. The challenge is not providing answers from data that is available; the challenge is getting good data in the first place. Many people only enter data that is requested, and then only when it is required. The shell has the disadvantage of requesting very little. And yet there are a variety of ways to enter data, and one of the advantages is that it need have little structure. The text editors such as vi and ed allow you to just start typing. Scanners can also be used to copy in text. Once the data is in it can be studied for structure. Descriptions, instructions, and articles with subjects fit better into a file structure with a name. Lists that can be divided into rows and columns can be formed into the fields of traditional data bases. The tree structure of the UNIX file path can be used if the data has a parent child relationship. The shell provides tools for other structures as well. What is required is a little creativity for seeing the data structure.

Perhaps the most challenging aspect of data entry for UNIX shell data bases are the restrictions on characters that can be used. Some printing characters are command characters for the shell. In some cases shell commands can enter characters which other programs cannot handle, such as control characters. In other cases the shell interprets characters as commands and will not allow them to be used. Some command characters such as # can be entered if quoted, but others are more complex. Either the program needs to be so constructed that the necessary characters can be entered as data, or a suitable substitution is made so that later the proper character is inserted via the tr command just before the data is printed.

The simplest data bases for the beginning user are fixed field data bases entered with a text editor such as vi or with a variation of the cat command (cat >> temp waits for input from the keyboard), or perhaps the mail command. Printing such a data base is easily accomplished with a print command such as pr and/or lp since the data does not need to be formatted. Such data bases are especially convenient if the fields of one record fits on one line of the display terminal. Searching capabilities are typically available in the editor used. Other commands such as grep can be used, but others, such as read, delete excess white space, thus destroying the formatting. The cut command can be used to convert such a data base to the character delimited form. The printif command of awk can be used to convert character delimited data bases to fixed field ones.

Another form of data base that can be entered fairly easily behaves similarly to hypertext. This mechanism can be incorporated into the shell using the following script:

```
#! /bin/sh
# File .hyper created 19960409 by F. John Kluth
file=l
while test "$file" != x
   do
   clear
   if expr substr "$file" 1 2 = ls > /dev/null
   then
      $file
   fi
   if expr substr "$file" 1 2 = pg > /dev/null
   then
      $file
   fi
    echo "enter x to exit, ls to list keywords, or pg (keyword) to
display"
   echo "enter keyword > \c"
   read file
done
```

The file .hyper is a program that displays definitions. The text is entered as a file with the word defined (keyword) as the name of the file. The file .hyper resides in the same directory as the data files. The period as the first name of the file prevents it from being listed as a keyword. A dictionary of several thousand words can be handled in this way. The ls command can be used to limit the search to the first few characters so more than a whole screen is not displayed. The grep command can search the files for a string that is used in a definition. The shell command nroff can be used to get a formatted printout.

Character delimited files can also be entered with vi, but this can be quite confusing. The best solution is a program that lists the field names and allows entry of data for each field. What follows is a description of such a program written entirely in shell script.

```
#! /bin/sh
# file add
# Section 1
prog=$0
fldnam1=NAME
fldnam2=ADDRESS
fldnam3=PHONE
rcrd="_     _     _"
ndc=n
# Section 2
until test $ndc = y
do
      clear
      echo "Enter letter <space> value to change."
      echo "Enter S to save or X to exit."
      echo
      # Section 3
      ival=0
      for field in $rcrd
      do
            ival=`expr $ival + 1`
            eval echo "$ival. \$fldnam$ival = $field"
      done
      # Section 4
      echo
      echo $msg
      echo "$prog > \c"
      read input sval
      # Section 5
      n=0
      for i in $sval
      do
            if test $n = 0
            then
                  val=$i
            else
                  val="${val}_$i"
            fi
            n=`expr $n + 1`
      done
      sval="$val"
      if test $n = 0
      then
            sval=_
      fi
      # Section 6
      case $input in
            [0-9]* )    clear
                        ival=0
                        rcrda=
                        for field in $rcrd
                        do
                              ival=`expr $ival + 1`
                              if test $ival = $input
                              then
                                    if test $ival -gt 1
                                    then
                                          rcrda="$rcrda     $sval"
                                    else
                                          rcrda="$sval"
                                    fi
                              else
                                    if test $ival -gt 1
                                    then
                                          rcrda="$rcrda     $field"
                                    else
                                          rcrda="$field"
                                    fi
                              fi
                        done
                        rcrd="$rcrda"
                        echo "rcrd=$rcrd"
                        msg= ;;
            S )         echo "$rcrd" >> add.dat;msg="Data saved!" ;;
            X )         clear; ndc=y ;;
            * )         clear; msg="Bad input!" ;;
      esac
done
```

Section 1 defines the basic variables for the program. The prog variable provides the program name for debugging purposes. The variables fldnam* form an array that lists the field names for the record. The variable rcrd stores the record itself and must always contain as many field values as there are fields. The variable ndc provides an exit for the main program loop. Section 2 begins the main loop and displays instruction information. Section 3 interprets the array and displays the field names and the current field values. Section 4 displays messages and inputs data from the user. Two parameters are input. The first is an instruction parameters that indicates either that an instruction is to be executed or the number of the field to be changed. The second parameter is the new field value. Section 5 tests for a null entry and replaces it with an underscore so that no fields are lost. If a space is found it is replaced with an underscore. Section 6 is a case statement that either modifies field values or executes commands. This routine forms the basic structure of a longer program mdat that includes a number of other commands including a modify command, a search command, a list command, and a thread command. The full program listing is available but will not be presented in any further detail here.

The program listed above, named add, has a number of advantages. The most important is that fields are entered in an unambiguous manner. The field lengths are dynamic. The menu approach can be easily built upon to add many complex functions. The disadvantages include the fact that full screen editing is not available. More frustrating is the fact that the entry of certain characters will cause the program to fail. The most significant of these is the apostrophe, which cannot be entered at all. The # sign can be entered if it is quoted. The * causes the curious result that the contents of the current directory are incorporated into the data. The one consolation of these problems is that you learn quickly about the type of characters needed for your data.

A listing of the full program mdat with numerous enhancements is available as a separate listing or file on disk from the present author. Though the code of the enhancements to this program are not included here, the inspection of the general concepts involved is quite instructional. The modify feature uses the shell sed command. The search feature uses the shell grep command. It is quite useful when designing a search command to allow a further search to reduce the size of the found group if it is too big to display. The thread command is the most significant enhancement, because it allows relating of two or more data bases. The thread command is built upon the observation that a data base can be constructed that has as its records information about a number of data bases. Each record consists of the following fields: The name of the data base, the path and file of the data base, and a list of every field in the data base. The field names have the same order as the field data in the records so the field names can be read from this data base as an array. Now so arrange the field names so that two records in separate data base that have the same value in the same field are referencing the same fact. Two data bases that are connected by having such an equality found by a program are said to be threaded.

As an example of threading I give the following example. Companies in a vendor data base are given a unique company number upon data entry. Another data base is set up with company phone numbers. In this second data base, the company number is stored in each record with a company phone number. The company phone numbers are found by locating the company by name in the vendor data base, and then threading to the phone number data base to find all the phone numbers associated with that company number. This process can be used to reduce duplication of data. It can also be used as an aid in searching for data.

Threading is heavily dependent upon the various search routines. The grep command can be used easily if the thread involves the first or last field. Threads involving other fields are dependent upon more complex looping structures combining read and grep. The awk command is quite suitable for this because of its ability to reference fields by position. One feature that must be considered is the uniqueness of the data used for searching. A thread routine can easily be written which only displays the first match found. This may be a mistake that causes the other values to be lost. The thread must be designed to handle all possible matches, whether one to one, many to one, or one to many.

The mdat program has a data base which stores information about other data bases including their location and field names. This feature compares field names and determines how many instances of a field name occur. This comparison is helpful for the threading feature. Data base names can be searched and by including a common prefix for related data bases the search can list data bases by topic. This feature allows a particular data base to be the target to be displayed.

The mdat program also stores the data from previous entries. When new data copies previous data, the new data does not need to be entered. In addition to reducing keystrokes, this feature also displays regularities in the data. When a new data base is connected the program reads the last record and displays this data as the previous data.

The following routine provides another method of entering data. This script has the advantage of allowing many more types of characters to be entered as data including those that bothered the previous script. In addition, space characters are converted to underscores, which makes the fields easier to handle by shell programs. The horizontal labeling of fields will be easier for some. Full screen editing is not available here either, to the frustration of many.

```
#! /bin/sh
# file addc
echo "^D saves; ^C quits without saving"
echo NAME.............ADDRESS...........PHONE
cat | tr ' ' '_' > addc.dat
```

Some consideration needs to be given to multi-user situations. If a data base exists as a file that is readable to shell scripts, then multiple users can read that file simultaneously. Problems occur when users are attempting to write to the file simultaneously. The HP-UX system seems capable of managing simple appends in the multi-user environment, so shell script programs incorporate some multi-user capability. The major problems come with attempts to modify a file. Suppose user A reads a file xx in state 1. Five minutes later user B reads the same file still in state 1. Now user A modifies 3 records and saves the changes to xx now in state 2. User B now has possession of a file that is no longer current. Now user B modifies 2 different records and saves them to xx now in state 3. What user B actually does is restore the original file plus what changes were made by B. This effectively destroys the changes made by user A.

The simplest way to deal with the problems raised with file modification is to put one user in charge of modifying files and give other users the ability to read the files but not write to them. This can be handled by changing the permissions on the file. If more than one user needs to modify the file, then some type of file locking must be set up. It appears that commands relating to permissions will handle this as well, but no information is available on the overhead that this might require. Record locking schemes are also possible, but this would require an additional field in the data base that must be read and interpreted by the managing program. Overhead may also be a problem here.

Overhead becomes a concern because of the shell approach to reading data from a data base. Typically every byte of the file is searched. On an Intel 386-20 running Interactive UNIX the grep command can search a 250K file in 5 seconds. On a HP9000 model E35 running HP-UX the grep command can search about 1 MEG in 5 seconds. These figures represent the largest practical data files for these systems using the grep command for the search. 250K represents about 500 pages of data and 17 to 35 hours of data entry time, or $500 to $1000 of business investment just for data entry. In spite of these limitations, it is surprising how much data can be accumulated by these techniques. A separate paper by the author "Hazardous Chemical Management in a Manufacturing Environment" is available which describes an application using the bourne shell techniques described here. This application involves the use of 5 data bases, the largest of which has about 160k bytes and 1500 records. Two separate programs access these data bases. One allows data entry which is controlled by one person. The other allows read only access to the data and is available to every user. The only characters to provide problems with data entry have been # which is uniformly replaced by NO and apostrophe which is simply omitted.

Unix commands of use in this context:
1. For data entry - Parameters can be entered on the command line. The var="value" structure can enter values into variables. The echo command can append whole lines including control characters to files. The read command allows values to be assigned to several variables. The editors ed, ex and vi allow entry of various types of text into a file. The cat command in the form cat >> filename is a simple method. The mail command can be used for data entry if you mail the file to yourself.
2. For data search - Grep and its variations search the entire file without regard to fields. The search for specific fields is more efficient in awk. sed and vi provide text search as well.
3. For data base manipulation the commands cut and paste allow the rearranging of fields, sort rearranges record, and allows sorting by fields. The command tr allows characters to be modified in records and strings. This is useful mainly for changing field separators, and porting data. The command vi allows moving of records and some field manipulations.
4. For output there are a number of options - The printif command of awk allows fixed-field output. The awk command can be considered a generalized report writer. Nroff, in combination with pr serves a similar function. These programs can output both to the screen and to the printer.
5. The shell variable IFS contains the "Internal Field Separators" which are normally space, and tab.
6. If dates are entered in the form yyyymmdd then they can be sorted using the sort command.

Normally, as the shell script data base becomes slow to access, a project should be prepared to port the data to a more efficient data base management environment, such as a 4GL. If this is not practical there are techniques for extending the usefulness of a common UNIX data base. The best supported approach is to use the path structure of the UNIX. If one field is used to enter data then data files can be named for the first letter of the data to be stored. The data base will then consist of a number of files named after the letter used to find them. A still larger data base will name paths after the first letter and files after the second letter. In this way the file location of the record can be computed from the data that is to be filed.

Data entry persons need to be protected from the power of the shell. For these persons shell looping structures can be used to simplify entry possibilities. Backup files can be created to assist if the program crashes. Writing each line to a file helps to minimize data loss.

Inclusion of an index record as a field is redundant but helpful with debugging data bases. The simplest form of index has the field value equal to the record number. For sorting purposes it is desirable to include leading zeros in this field so that every value has a fixed string length. Uniqueness can be checked by comparing the field value against the record number. Uniqueness is important when the index is used to substitute values from one data base into another. One structural observation that can be made is that certain data hangs together. The address of a company, for example, is unique to the company and can be said to define it. Often a separate data base is set up with an index called company number, with a unique number and the name and address of that company. The particular value in the index is often referred to as a pointer to that record. A purchase order data base containing a record of a purchase from a company contains only the index in a field called company number. The purchase order is said to be related to the company data base by the company number field. A shell script program can be written which displays both the purchase order information and the company information by using the value of the company number field in the purchase order record to search the company information data base. One of the advantages of entering the data into a shell script data base is that the clumping quality of the data can be studied. As the structure becomes obvious then ways can be tried to symbolize and index it to enhance the performance of the data base application.

Search strings need to be short strings that find data. A good search string is as short and unique as possible. For example to search for instances of where the two words "GRAND CENTRAL" occur in a document, it may be sufficient to search for the string "ND C". Often a search must be repeated on a new string if no data is found because of the possibility of spelling errors in the data or search string. Some characters used in abbreviations such as & for 'and' cause difficulty because of spelling and spacing ambiguity for example B and D Supply could be written as B&D Supply, B. & D. Supply, B & D Supply. A search could be constructed for all instances of B and D, but it would be easier to search for the word Supply.

Shell script programming provides an efficient way to get started on a data base application. As the data is accumulated, insights into the nature of the data are provided. While these insights are being gained, access to the data can be provided. When the data is well formed and a suitable application program is identified, the data can be manipulated to input that the application can accept. These features make the shell script quite useful of working with data base information.

Bibliography:

1. Arthur, Lowell Jay, Ted Burns, "UNIX Shell Programming", Third Edition, John Wiley & Sons, Inc., New York, 1994.
2. Blinn, Bruce, "Portable Shell Programming: An Extensive Collection of Bourne Shell Examples", Prentice Hall PTR, Upper Saddle River, New Jersey, 1996
3. Hewlett Packard, "HP-UX Reference", Volume 1: Sections 1 and 9, HP9000 Computers, HP-UX Release 9.0, Third Edition, Hewlett Packard, USA, August 1992
4. Hewlett-Packard Company, "The Ultimate Guide to the vi and ex Text Editors", Benjamin/Cummings Publishing Company, Inc.; Redwood City, 1990.
5. Hewlett Packard, "Shells: User's Guide", HP 9000 Computers, Hewlett Packard, USA, Second Edition, August 1992.
6. Kluth, F. John, "Hazardous Chemical Management in a Manufacturing Environment", paper presented at the Ohio Academy of Science 105th Annual meeting, Malone College, Canton, Ohio, May 4, 1996
7. Kluth, F. John, "mdat, Shell Script Data Base Management Program", unpublished, Last edit 03/19/1996
8. Leach, Ronald J., "Advanced Topics in UNIX, Processes, Files, and Systems", John Wiley & Sons, Inc., New York, 1994.

9. Peek, Jerry, Tim O'Reilly, and Mike Loukides, "UNIX Power Tools", Bantam Books, New York, 1993.