SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 15(10), 1011–1020 (OCTOBER 1985)

DDA — A Data Definition Facility for UNIX* using Awk

PAUL A. BAILES
Computing and Information Studies Unit, School of Social and Industrial Administration, Griffith University, Nathan QLD 4111, Australia

SUMMARY

We discuss how the Awk language is an effective tool for interrogating sequential data files whose records consist of characters grouped into fields. This facility is used to implement a system to manage an 'address book' database. We then generalize the design, describing a system which allows one to interactively specify a DBMS according to one's own requirements within our general framework. We choose a wine-tasting DBMS as an example specification for further consideration and evaluation.

KEY WORDS Awk Database UNIX Wine-tasting

INTRODUCTION — AWK

We consider a *database* to consist of a set of *entities*, each with various *attributes*. If the database is implemented by a sequential file of records, each of which represents an entity, and if each record comprises a set of fields (of characters), each of which represents an attribute of the entity, then the identification of sets of entities on the basis of properties of relevant attributes may be implemented by extracting records whose fields possess certain values.

The Awk language[1] available under the UNIX system[2, 3] supports the suggested scenario. It regards a file as a series of records (delimited by 'newline' characters), each of which consists of numerically-indexed fields separated by 'white space' characters (this default is subject to amendment by the Awk programmer). An Awk program is a series of statements of the general form.

```
pattern action
```

For each record of the input to which the pattern is applicable, the associated action is executed. For example, the pattern

```
$3 ~ / regular expression /
```

* UNIX is a Trademark of A. T. & T. Bell Laboratories.

0038–0644/85/101011–10$01.00
Received 12 July 1984

is applicable to a record if its third field ($3) is matched (~) by the regular expression. The regular expression (r.e.), as in the UNIX **ed** editor,[4] is the most powerful form of pattern definition available in Awk. An r.e. matches a string of characters s1 if there exists as a member of the set of strings defined by the r.e. a string s2 such that s2 is a substring of s1. Patterns may be viewed as boolean-valued expressions, and the applicability of a pattern as evaluating to true. Thus, patterns may be logically-composed, e.g.

```
$1 ~ /abc/ && $2 ~ /cde/
```

is true if abc matches field 1 **and** cde matches field 2.

The action is a sequence of commands enclosed in braces. They resemble the statements of C.[5] For example, the Awk statement

```
$1 ~ /abc/ {print $3; print $4}
```

causes the printing on the standard output file, on separate lines, of the third and fourth fields of each input record whose first field is matched by abc. An empty pattern is always applicable, and an empty action means to print the complete record. For example,

```
{print $1, $3}
```

prints on standard output a line containing the first and third fields of each input record. Similarly,

```
$3 ~ /^pqr$/
```

prints on standard output each input record whose third field is precisely matched by pqr.

GENERAL ORGANIZATION

Following the approach of data abstraction methodology,[6] access to the database is via a set of interface procedures. We identify the general requirements:

(a) add new information
(b) make enquiries
(c) list all information
(d) edit database
(e) dump database in raw form

Items (a), (b) and (c) are envisaged as the normal user interface, and (d) and (e) as for maintenance. Hidden in the implementation will be the data file itself and a general formatting routine. Figure 1 shows the primary information flows in such a system.

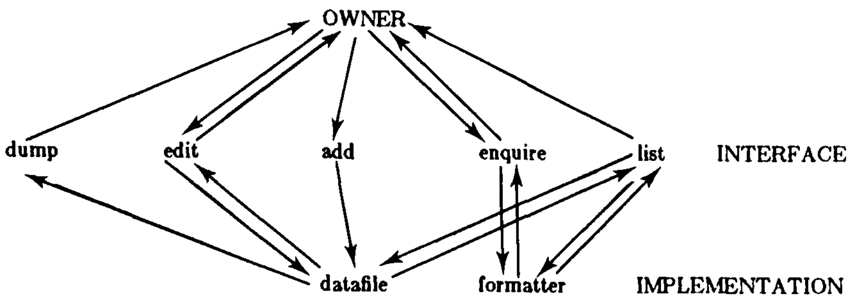

Figure 1.

EXAMPLE I — AN 'ADDRESS BOOK' DATABASE

Information on an individual will comprise an entity with the following attributes:
1. name (e.g. 'john brown')
2. address (e.g. '42 surf st seatown qld 4999')
3. phone no (e.g.'(070)8673882').

For simplicity of access, a single character case has been adopted.

We will use a directory store to hide the datafile and the formatter. The interface programs will be identified as aaddr (add), qaddr (enquire), laddr (list), eaddr (edit), and daddr (dump).

We require that the aaddr program prompt the owner of the database for each attribute (field) of an entity (record) to be added. An effective UNIX shell[7] script is as follows:

```
echo -n "enter name:"
read F1
while test "$F1"; do
echo -n "enter address:"
read F2
echo -n "enter phone no.:"
read F3
echo
echo $F1'|'$F2'|'$F3 >> store/datafile
echo -n "enter name:"
read F1
done
```

Noting that computer output is distinguished by **bold** type, an example is as follows:

```
$ aaddr
enter name: john brown
enter address: 42 surf st seatown qld 4999
enter phone no.: (070)8673882
enter name: tom jones
enter address: 88 hilly rd bushville qld 4998
enter phone no.: (078)8888881

enter name: ⟨null response — just type RETURN — to terminate⟩
```

Assuming that the database was initially empty, the contents of store/datafile after the above would be as follows:

```
john brown|42 surf st seatown qld 4999|(070)8673882
tom jones|88 hilly rd bushville qld 4998|(078)8888881
```

The laddr command is implemented simply by applying the formatter to the entire database:

```
store/formatter < store/datafile
```

The formatter itself is

```
awk -F\|'
{
print "name:",$1;
print "address:",$2;
print "phone no.:",$3;
print ""
}'
```

Thus, we may have as follows:

```
$ laddr
name: john brown
address: 42 surf st seatown qld 4999
phone no: (070)8673882

name: tom jones
address: 82 hilly rd bushville qld 4998
phone no.: (078)8888881
```

When making enquiries, it may be that some attributes do not interest us, either by design or by lack of knowledge. For example, one might be interested in finding all one's acquaintances living in 'seatown', or one might be interested in finding the address of a friend whose name is known. The following shell script is the qaddr program:

```
while true; do
echo -n "enter name:"
read F1
if test -z "$F1"
then F1= '.*'
```

```
fi
echo -n "enter address:"
read F2
if test -z "$F2"
then F2= '.*'
fi
echo -n "enter phone no.:"
read F3
if test -z "$F3"
then F3= '.*'
fi
echo
awk -F\|"
~ /$F1/ && ~ /$F2/ && ~ /$F3/
 " < store/datafile | store/formatter | pg
done
```

The form

```
if test -z "$Fi"
then Fi= '.*'
fi
```

for some field i ensures that if a null response is given to a prompt for an attribute (field), that all values for that attribute will be matched by the r.e.'.*'. After all fields have been sought, Awk is invoked. Because permitting white space to appear in attribute values requires that some other field separator be used, we specify '|' as such by the -F argument. Input to Awk comes from the data file, is piped through the formatter, and as a final touch is paginated (by a program known as pg). Note that in general a non-null response to a prompt for information about a field may be a regular expression.

For example, consider the following dialogue

```
$ qaddr
name: john
address: seatown
phone no.:

⟨now appears output pertaining to the above⟩

name:
address: 82 hilly rd bushville qld
phone no.:

⟨now appears output pertaining to the above⟩
```

The first enquiry will produce output for each entity for which the string john appears in the name attribute, seatown in the address attribute, and with any telephone number. The second enquiry will produce output for any person whose address contains the text

82 hilly rd bushville qld. The format of the output is similar to that of the list program.

Recall that responses to enquiry prompts are regular expressions. For example, the dialogue

```
name: ^j
address:
phone no.: (.[678]
```

causes the display of all entities, the name attribute of which begins with the letter 'j' and the phone number of which has as its second area code digit '6', '7' or '8'. Typing RUBOUT serves to terminate enquiry dialogues.

The daddr program is simply

```
cat store/datafile
```

The eaddr program is likewise

```
ed store/datafile
```

GENERALIZATION

The simplicity of the above programs and their lack of any great dependence upon the properties of attributes in the database means that development of a program to generate such programs should be easy. An experimental system verifies this hypothesis.

Our system, called dda, goes through the following dialogue with a user to construct the address book DBMS described above:

```
$ dda
define database

enter name for field 1: name
enter name for field 2: address
enter name for field 3: phone no.
enter name for field 4:

3 fields in data base

enter storage directory name: store
enter "add" command name: aaddr
enter "enquire" command name: qaddr
enter "list" command name: laddr
enter "dump" command name: daddr
enter "edit" command name: eaddr
enter name of editor to use in "edit" command:
uses default editor "ed"(1)

DBMS generated successfully
$
```

The first part of the dialogue defines the names of the fields (and by the way, their number) to be used when adding, enquiring and listing. A null response signifies that all fields have been defined. A requirement dictated by the implementation of dda is that some characters with special meaning to the shell or to Awk (e.g.') are removed from field names.

The second part of the dialogue names the required files. The creator of the DBMS may specify what particular editor to use for the edit function — here a null response means that ed is the default. If a supplied file name contains blank characters, only the first blank-free sequence is recognized. If no file name is given, or if the file already exists, dda aborts with an appropriate indication of error.

DISCUSSION

Although these comments refer to the address book DBMS described above, they are generally applicable to any systems generated by dda.

A limitation is that each entity must have the same set of attributes. Because our DBMS is embedded in the UNIX environment, it is easy to simulate an heterogeneous database by using a number of different homogeneous databases. They may be unified by high-level shell scripts which invoke each of the particular interface programs for each individual database.

A related limitation is that only a single-level hierarchy of attribution is supported. For example, having chosen to treat an address as a single attribute, there is no all-embracing way to recognize the distinction between street, suburb and state. However, if we have an idea of the form of the stored data for an attribute, we may make enquiries using r.e.s which simulate the existence of subattributes. Similarly, enquiries based on numeric ranges are not directly available. For example, if we had an age attribute, then consider an enquiry to yield all persons between the ages of 18 and 65. Once again, our problems may be mitigated by defining appropriate r.e.s on which to enquire.

More generally, it is wise when adding information to use a single character case to permit easy access. However, if the owner of a set of information so desires, both upper- and lower-case information may be stored. It is often also wise to input lengthy descriptions of attributes, so that enquiries by substring abbreviation will have a greater chance of success.

In the above, we have stated that regular expressions give us the capability to perform interesting enquiries. We now consider a more involved example to support these claims.

EXAMPLE II — A WINE-TASTING DBMS

An entity will be a tasting record for a single wine. Thus, multiple tastings of the one wine would require multiple entries in the database. An appropriate set of attributes/field names is as follows:

(1) brand name (e.g. 'sichel')
(2) vintage year (e.g. '1980'; or 'n.v.' for non-vintage)
(3) district (e.g. 'rhone')
(4) grape variety (e.g. 'pinot noir')
(5) any other description (e.g. 'liebfraumilch'; or 'white burgundy')
(6) information source (e.g. 'september 1980 decanter')

(7) assessment (e.g. '16/20'; or 'very good')
(8) cellaring potential (e.g. '1985')

Blends of either vintage years, districts or varieties can be specified by separating the elements of the blend with some standard character e.g. '/'. For example, 'pinot noir/chardonnay' describes a typical Champagne blend.

Using the dda program to define this scheme is easy. For example:

```
$ dda
define database

enter name for field 1: brand name
...
enter name for field 8: cellaring potential
enter name for field 9

8 fields in database

enter "add" command name: avin
enter "enquire" command name: qvin
enter "list" command name: lvin
enter "dump" command name: dvin
enter "edit" command name: evin
enter name of editor to use in "edit" command: med

DBMS generated successfully
$
```

Here we have nominated an editor called med.

Let us consider some of the conventions an owner of such a database should adopt for more effective use. First, we always use lower-case characters, so that enquiries will match when they are 'supposed' to. Secondly, label information should be standard, typically a rationalization of what is given by the maker. Our categories of brand, year, district, grape variety and other description provide a range which seems adequate. An interesting case in point is Lindemans 1979 Hunter River Riesling Bin 5615, a reputable Australian wine. For the district 'hunter' alone suffices, as 'river' adds no worthwhile information (in the case of a wine from the Barossa district in South Australia however, the qualifier 'valley' or 'hills' is meaningful). The grape variety we know to be 'semillon'. It is in the category of any other description that the colloquial name ('white burgundy') and bin number would appear.

Thirdly, when referrring to an issue of a journal, such as the September 1980 issue of *Decanter* above, we express the specification in a standard order, so that natural enquiries of the form 'any month, particular year, particular magazine' or 'particular year, any magazine' are supported. Note that if it is an abbreviation by which the journal is commonly known, it should be used.

Finally, cellaring potential should be expressed in absolute rather than relative terms, as there is no guarantee that a record will be kept of when such a recommendation was made. We could redesign the DBMS to enforce this, but the expression of a particular year or list of years seems more relevant for purposes of enquiry.

As an example, consider the following dialogue:

```
$ qvin
enter brand name:
enter vintage year: 19[67].
enter district:
enter grape variety: pinot noir
enter any other description:
enter information source: july .*wsbg
enter assessment:
enter cellaring potential: 198[4–9]
```

It will yield information on all wines attributed to pinot noir grapes vintaged during the nineteen-sixties or seventies, and which according to July issues of the *Wine and Spirit Buying Guide* magazine will mature during the remaining years of the eighties. Note how we are able to treat the information source attribute as consisting of three subattributes (month, year, magazine) and make our enquiry on only the first and the third.

In fact, the given r.e.s are more precise than really necessary. For example, although the dialogue fragment

```
enter cellaring potential: 8[4–9]
```

would have covered the following additional years — 87, 877, 1862, 2862, 87654 etc. — it is unlikely, given the nature of what is stored in the database, that problems would have arisen.

Thus we have seen how r.e.s might be used to effect subattributes and numeric range enquiries.

CONCLUSIONS

When we take a simple view of what constitutes a database, and have relatively small numbers of entries, or are not particularly concerned with speed of access, Awk is a suitable implementation language. The simplicity of the scheme means that dda, a generalized data definition facility, is easy to produce. Because databases are accessed via regular expressions covering substrings of fields, unlike the UNIX refer system[8] which requires exact keys, sophisticated enquiries may be attempted successfully.

REFERENCES

1. A. V. Aho, B. W. Kernighan and P. J. Weinberger, 'Awk — a pattern scanning and processing language', *Software—Practice and Experience*, **9**, 267–280 (1979).
2. D. M. Ritchie and K. Thompson, 'The UNIX timesharing system', *CACM* **17** (7), 365–375 (1974).
3. S. R. Bourne, *The UNIX system*, Addison-Wesley, 1982.
4. B. W. Kernighan, 'A tutorial introduction to the UNIX text editor', *UNIX Programmer's Manual Volume 2*, 1979.
5. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.
6. R. T. Yeh, (ed.), *Current Trends in Programming Methodology Volume IV — Data Abstraction*, Prentice-Hall, 1978.

7. S. R. Bourne, 'An introduction the the UNIX shell', *Unix Programmer's Manual Volume 2*, 1979.
8. M. E. Lesk, 'Some applications of inverted indexes on the UNIX system', *UNIX Programmer's Manual Volume 2*, 1979.