The UNIX Shell As a Fourth Generation Language

Evan Schaffer and Mike Wolf
Revolutionary Software, Inc.
131 Rathburn Way, Santa Cruz CA 95062, USA

evan@rsw.com — wolf@hyperion.com

ABSTRACT

There are many database systems available for UNIX. But almost all are software prisons that you must get into and leave the power of UNIX behind. Most were developed on operating systems other than UNIX. Consequently their developers had very few software features to build upon, and wrote the functionality they needed directly, without regard for the features provided by the operating system. The resulting database systems are large, complex programs which degrade total system performance, especially when they are run in a multi-user environment.

UNIX provides hundreds of programs that can be piped together to easily perform almost any function imaginable. Nothing comes close to providing the functions that come standard with UNIX. Programs and philosophies carried over from other systems put walls between the user and UNIX, and the power of UNIX is thrown away.

The shell, extended with a few relational operators, is the fourth generation language most appropriate to the UNIX environment.

1. Fourth Generation Systems

In recent years, a variety of developments in programming language design have emerged. Object-oriented languages are becoming common, and languages explicitly supporting multiple tasks and inter-task communication are also gaining popularity. Unfortunately, these efforts have resulted in productivity increases too small to offset the growth in the size and complexity of software systems. A response to this has been the development of fourth generation programming languages. Although not commonly thought of as such, the UNIX shell is one of the most powerful and flexible fourth generation languages available.

1.1 Attempts at a Definition

There is no consensus on the definition of what constitutes a third or fourth generation language. Mainstream third generation languages are typed, procedural languages. They are standardized and largely hardware independent. Operations in the language must be specified in a detailed, step-by-step algorithmic fashion. Third generation languages do very little implicit processing. Third generation languages are general purpose, even most of those which were ostensibly designed as special purpose languages.

Fourth generation languages are usually intended as design tools for a particular application domains. They are usually free form in their use of variables, often not requiring type definitions and allowing dynamic typing of variables. They don't emphasize a modular, procedure-based coding style. Instead, they contain a number of predefined procedures for performing various high-level operations. The high-level operations involve large amounts of implied processing. For example, a "sort" operator is usually available. The facilities of a fourth generation language are usually both more powerful and less flexible than the facilities available in a third generation language.

A fourth generation programming language (4GL) should make possible the simple statement of what you want, rather than a detailed procedure of how to produce it. Although there are many products calling themselves 4GL today, they are mostly rewrites of COBOL and report writers. They are too low level and tedious. This is definitely not what a 4GL should be.

1.2 Previous Generations

The first generation of computer languages was the sequence of zeroes and ones that were the machine instructions. In the beginning people had to code in this way.

The second generation was "assembly language", which has a one-to-one correspondence with machine instructions. Humans could write names words to be converted into machine language. For example this assembler code adds register 1 to register 2.

Figure 1. Second Generation Program

```
add r1,r2
```

One line of code produced one computer instruction. Then, in about 1956, FORTRAN was written to do *formula translation*, and it became much easier to write programs. Each line of code produced several computer instructions. The third generation has come to encompass sophisticated macro assembly languages, and other so called "high level" languages like C, COBOL, Pascal, LISP, PL/I and many others. There are advanced constructs close to English like "if", "then", and "else", but the types of statements are constrained to mostly arithmetical operations, with limited string capabilities. A typical third generation program is:

Figure 2. Third Generation Program

```
for i=1 to 10
      print i, sqrt(i)
next i
```

The next step comes in describing what you want and letting the computer figure how to give it to you. The fourth generation has English-like words, but statements typically deal with more than numbers, and are "non-procedural". A program to sort all the lines in a file, for example, is reduced to "sort file" in a fourth generation environment. Fourth generation language primitives often include relational operators, while third generation languages generally do not. And, when you need to mix procedural with non-procedural instructions, that is easy to do.

Figure 3. Fourth Generation Program

```
for file in file1 file2 file3
do
      sort $file
done
```

At the UNIX shell level you can, in many cases, say what you want without saying how (non-procedural), and you will get it:

$ sort file

and you get a sorted file.

$ spell file

and you get a list of words in your file that are not in the dictionary. One line of commands produces calls to one or more programs, each of which may have thousands of instructions.

With the shell you can put together application in minutes or hours, instead of the weeks or months required with 3GL code. In a 4GL you should be able to write most applications in a line or two. With the shell you can say things like:

```
$ cut some columns < file |
grep 'string' |
sort |
lpr
```

This short program gets some of the columns in a file, pipes them through grep to get just the lines with a certain string, sorts them and sends them to a line printer.

This same report would take tens to hundreds of lines in COBOL, PL/I, C, and most commercial 4GLs. In those languages you write instructions one at a time, to process records from the file one at a time. This is very tedious compared to writing one instruction to operate on the entire file.

1.3 Data Structures in the Data

In an ideal environment, the structure of data is in the data. Newline separators for records, and column separators for fields can tell any program where the fields and records are. 3GL languages have the data structure hard coded into them, so that one program reads only one kind of file.

In a traditional third generation environment, the structure of the file must be hard coded into the program. In a fourth generation environment files have their structure embedded with newline and tab record and field separators. Any program can find a record by just looking at the stream of characters. Add a single character to the data file read by a COBOL program and all will be changed or lost, so you have to do file conversions in the COBOL environment all of the time. And these are done in COBOL. Any changes require editing and recompiling all of the programs that read that file.

In addition, there are no file operators in 3GLs, only field at a time instructions. Therefore you have to write loops to process each record. This takes a lot more code than just processing a whole file at a time.

Most commercial 4GLs are very similar to COBOL. You still have to do record at a time processing. If the COBOL program takes 100 lines, the 4GL will take anywhere from 50 to 100 lines, to do what we did above in one pipeline.

1.4 A Revolution in Computing

If you write C programs on UNIX, you miss most of the advantages of shell level programming. It's been suggested that since C and other languages on UNIX, give you the **system** command this converts them into 4GLs. This means that assembler is a 4GL if it has a "system" command. But on non-UNIX operating systems like DOS and VMS there is not as rich a variety of tools available as in UNIX, except to the extent that UNIX has influenced these systems.

The UNIX system itself offers an integral tree index approach to data organization: the hierarchical file system itself. Many utilities traverse these trees, search them, add and delete nodes, and in general provide procedural tools with which to deal with files. The same is true of DOS and Macintosh systems. The opportunity is afforded to avoid re-inventing the wheel.

This really is a revolution in computing. Working with great tools will spoil you, but most of the computing world is still writing COBOL. To have to go back and forth between such environments is painful.

A good 4GL should be written in C ... once. It should be written so general purpose and easy to use that its functionality can be used from then on, rather than recode in each application. Then these good programs can be used to put together applications, not coding each entire application in C, unless there is some critical need.

As users become more familiar with their environment they are more able to use the power of these advanced systems, if only to shorten repetitive command sequences, another key feature of 4GLs. In every computing environment there are facilities to collapse a sequence of keystrokes like aliasing, scripting, and macro construction.

Marketing people got wind of 4GL and turned it into a big marketing hype. Most database management systems wrote their own **procedural** language like COBOL or RPG and called it a 4GL. They are usually worse than COBOL, because you have to learn their new language, rather than use a classic. Few 4GL designers put as much time and energy into designing their language as was put into COBOL.

The driving force behind fourth generation languages comes from several needs. Programming projects commonly involve man years of work. The shortage of experienced software engineers and the need to increase productivity pushes us towards tools allowing faster development cycles. The increased use of computers by users who do not have formal computer science education requires very high-level tools which let novice programmers concentrate on algorithms rather then implementation details. As more work is done on computers, there is more demand for single use programs to perform a specific task. The relatively high expense of coding a software tool with a one time use encourages the use of any method available for simplifying the development process.

As third generation languages are becoming increasingly less able to meet the diverse needs of computer users, several principles of software design are gaining great popularity, especially within the UNIX community:

Data should be kept in flat ASCII files, not binary, so that we can always see what we are doing, and do not have to depend upon some special program to decode our data for us.

Programming should be done in fourth generation languages, except when the expected heavy use and/or resource consumption of the program justify the expense of a more efficient coding in a third generation language.

Programs should be small and should pass data on to other programs. Software prisons, or large programs with self-contained environments, must be avoided because they require learning and they make extracting data difficult.

We should build software and systems to meet interface standards so that we can share software and stop dreaming that any individual or company can do it all from scratch.

Approaching software engineering with principles like these does have some drawbacks. The major drawback is that fourth generation languages almost always produce slower code then third generation languages do. As computers increase in speed and power this drawback becomes less and less of a consideration. As improved compiler optimization techniques spread, the difference between code produced by 3GLs and 4GLs will become smaller.

1.5 A Paradigm

A programming paradigm is important for ensuring a robust language which has a consistent style to its syntax and semantics. Paradigms for fourth generation languages must meet requirements more stringent then those for third generation languages. To start with, a fourth generation language should provide a consistent interface to high-level facilities working with a variety of complex data types, while simultaneously providing fundamental low-level language constructs for coding any functionality missing from the predefined facilities. Too many 4GL's are good only for projects within a narrow application area. It's difficult to allow for high-level constructs from a variety of fields without the programmer having to specify the level of detail required in a 3GL.

The paradigm we choose for fourth generation languages is the operator/stream paradigm. In this model, data flows in unidirectional 'streams' on which operators are placed. Each operator transforms the data as it passes by. The set of streams in a program form a directed graph, possibly with cycles. This paradigm concentrates on what needs to be done to the data, and deemphasizes the techniques used in the transformation.

Fourth generation languages which attempt to use only the procedural paradigm of mainstream third generation languages usually end up being limited to a specific application domain. The procedural model doesn't describe data in an abstract enough way. Different types of operations require too much detailed code to work with, and the languages don't have the simple relation between all data and operators the way an operator/stream paradigm does.

A side benefit of using the operator/stream paradigm comes up in the design of graphical programming tools. Traditional third generation languages haven't been well adapted to a graphical programming interface. The problem stems, in part, from the difficulty of expressing the numerous possibilities in an intuitive pictorial way. With operator/stream as the basis for a language, a graphical programming aid can easily convey the process of placing an operator on a stream.

The operator/stream paradigm has proven effective in more domains than just language design. Some UNIX kernels make use of the paradigm to reduce the complexity of the operating system code. Rather than having one large, complex piece of code handling all the functionality of a particular aspect of the operating system (such as a device driver), data in the kernel is run through a linked set of operators, each operator performing one small, well-defined function. This allows users to modify the system by introducing new operators, without having to understand the innards of other operators on the stream.

2. The Shell

The shell and the set of UNIX utilities form a fourth generation language (4GL) based on the operator/stream paradigm. The critical feature of the shell which puts it in the class of 4GL's is the UNIX pipe, which allows a shell to start a sequence of processes, each reading its input from its predecessor process, and writing its output to a successor process. The UNIX pipe is one of the major reasons leading to the adoption of UNIX as the standard multi-user operating system. Unfortunately, few people fully understand the philosophy behind it; most software developers are still producing large, self-contained applications using data formats incompatible with anything else.

For the shell, the UNIX pipe provides the data streams, and the hundreds of standard UNIX utilities provide the core set of operators. The power of this approach is tremendous. Since the data streams are flat ASCII, all the operators can read each other's data. UNIX includes a few standard utilities which are capable of most data formatting needed to transform one program's output to the form required by another. In addition, using stand-alone programs as operators allows easy use of custom or commercial packages of operators, such as statistics or database packages. This modularity encourages code reuse, and the flat ASCII stream format makes it easy to get operators from a variety of sources talking to each other. Finally, since the operators can only transform the data stream running through them, side effects can't surprise the software engineer by giving unexpected results.

The UNIX filesystem also provides a hierarchal storage medium for data. Since UNIX files are flat ASCII data files, and UNIX makes a deliberate attempt to make all data sources look the same, most utilities can't distinguish between data coming from a stream and data coming from a file. This gives great flexibility, allowing the shell to store the results of a pipeline into a file, and then feed that data back into a stream at some future point.

There are two frequent criticisms of fourth generation languages. It is often noted that 4GL's tend to be suited for a particular application area, and that their low-level facilities are not up to the task of providing complex functions which don't already exist in the high-level library. The shell escapes this problem; UNIX utilities can be written in any language, from shell scripts to assembly language. If a tool is needed which isn't currently available, the developer is free to pick the language most suited to solving the problem, whether it's a CASE tool or standard C. This ability to combine the shell with products of all other existing development tools results in a uniquely general 4GL.

Many also complain that fourth generation languages sacrifice too much efficiency for the sake of short source code and high-speed development. The ability to use operators from any source is an answer to this complaint as well. It allows a shell programmer to code speed-critical routines in a language more suited to efficiency considerations. If an application requires floating point number crunching, one codes the appropriate routines in Fortran, and the non time-critical sections of the code can still be done in the high level shell code.

With the shell, development is quite easy for even the novice programmer. The interpreted environment allows easy access to the internals of the script as it runs, as well as a fast test-change-test cycle. The flat ASCII data format and lack of operator side-effects make it easy to examine the effect different operators have on the stream of data.

The shell relies heavily on its operators. For example, it has essentially no expression evaluation capability. Instead, it uses the 'test' utility to provide expressions.

The 'string' data type is the only one the shell supports. The shell assumes that there are operators which will do any more advanced data type a programmer might need. Operators exist to perform numerical functions. For multi-field records, operators commonly use the space or tab character as a field separator and the newline character as a record terminator. This allows great flexibility, despite the overhead incurred of converting data into and out of ASCII for non-string operations.

One of the greatest strengths of the shell is the ability to process an entire file with a single command. The shell does allow for defining procedures, as well as execution control constructs like if, while, and case. However, these flow control constructs are often not needed. In the example presented in the following section, no looping is done explicitly by the shell script, because the operators implicitly loop, acting on each line of the program.

2.1 Compatibility with DOS

DOS shares key underlying features with UNIX, enough so that the operator/stream paradigm can be utilized identically in both environments. Except for minor limitations on file name syntax, the DOS hierarchically structured file system appears to the user to be functionally identical to the UNIX file system. The multi-tasking capabilities of UNIX, while missing from DOS, are not essential elements of the paradigm. While DOS shells use intermediate temporary files to implement pipes, the interface presented to users, even using COMMAND.COM, can be described with the operator/stream terminology we use here. The UNIX shell and awk are available as DOS shell replacements and enhancements, notably in the MKS Toolkit for DOS.

With this foundation in mind, let's examine a 4gl that uses the shell as its development environment: **/rdb**.

2.2 How /rdb Defines a Relational Database

A relational database is a collection of relations or *tables* that may be related on one or more common columns. Relational data bases implemented like this are easily transportable from one environment to another.

Relational databases have a solid mathematical base in relational set theory, relational algebra, and relational calculus. There are theorems in this relational math that prove that any data put into a relational database can be extracted. The mathematical base also assures that manipulations performed will have correct results, just as arithmetic assures us that the math functions we perform on the computer have correct results.

2.3 What is a Relation?

A **/rdb** relation, or table, is an ordinary ASCII file. But some rules must be followed to use an ordinary file as a database table. A **/rdb** table has rows, or records, separated by newlines. It has fields, or columns, separated by a tab character. Every row must have the same number of columns. The first row of a table contains the names of the columns; the second row contains columns of dashes. Any kind of information can be represented in such a table: numbers, words, file names, etc: **/rdb** commands and relational set theory doesn't care about the content of the table -- just as long as these rules are followed for the form of tables. Another important rule to remember when designing a database is: *If many columns are used in a single row to describe the same type of information, it's time to make a new table.* For example, consider a table of family members:

```
id   mom     dad    kid1    kid2
--   ----    ---    ----    ----
1    mary    jack   billy   bobby
2    nancy   joe    terry   susie
3    sally   john   adam
```

In this example there are two *kid* columns in each row. The right way to express this relationship is with two tables: one for parents and one for kids. They are *related* or linked by a common column, *id*.

```
% cat folks
id   mom     dad
--   ---     ---
1    mary    jack
2    nancy   joe
3    sally   john
```

```
% cat kids
id   kid
--   ---
1    billy
1    bobby
2    terry
2    susie
3    adam
```

2.4 How Is Information Accessed?

Tables are accessed through **/rdb** and shell commands issued at the UNIX prompt or from within shell or C programs. Here is a list of some common **/rdb** commands which are used or mentioned in these examples.

Figure 4. Selected /rdb Commands

/rdb command	description
column, project	select only certain columns
row, select	select only certain rows
mean	compute the mean of selected columns
jointable	join two tables
sorttable	sort a table
compute	do calculations on columns
subtotal	subtotal selected columns
total	total selected columns
rename	change the name of a column
justify	make a table line up properly
headoff	remove the first two header rows
report	report writer
ve	vi-like table editor

/rdb commands are programs that read tables from the standard input and write tables to the standard output. Suppose there's a table that looks like this:

```
% cat inventory
Item   Amount   Cost   Value   Description
----   ------   ----   -----   --------------
1      3        50     150     rubber gloves
2      100      5      500     test tubes
3      5        80     400     clamps
4      23       19     437     plates
5      99       24     2376    cleaning cloth
6      89       147    13083   bunsen burners
7      5        175    875     scales
```

Then a sample query might be:

```
% column Item Cost Amount < inventory
Item   Cost   Amount
----   ----   ------
1      50     3
2      5      100
3      80     5
4      19     23
5      24     99
6      147    89
7      175    5
```

This is read aloud as: ‘‘*select the Item, Cost, and Amount columns from the inventory table.*’’ It’s important to voice queries because people often type stuff in that they would never say out loud.

```
% row 'Cost > 50' < inventory
Item   Amount   Cost   Value     Description
----   ------   ----   -----     --------------
3      5        80     500       clamps
6      89       147    16353.8   bunsen burners
7      5        175    1093.75   scales
```

This is, ‘‘*select rows where the Cost column is greater than 50 from the inventory table.*’’ To put commands together:

```
% column Item Cost Value < inventory |
row 'Cost > 50'
Item   Cost   Value
----   ----   -----
3      80     400
6      147    13083
7      175    875
```

This is pronounced, ‘‘*select the Item, Cost and Value columns from the inventory file and select those rows where Cost is greater than 50.*’’ Inside the single quotes the < and > symbols are pronounced *less than* and *greater than* respectively, while outside single quotes they are pronounced *from* and *to.* The | (pipe) symbol is pronounced *and.*

To take the mean of the result while listing each line:

```
% column Item Cost Value < inventory |
row 'Cost > 50' | mean -l Value
Item   Cost   Value
----   ----   -----
3      80     400
6      147    13083
7      175    875
----   ----   -----
               4786
```

2.5 Creating Tables and Entering Data

There are many different ways to create tables; editors, programs, shell commands such as **sed** or **awk**, etc. Most often, however, when **/rdb** tables are entered from scratch, **ve**, the **/rdb** table editor is used. **ve** allows the creation of tables quickly and in a familiar and easy way. It’s a lot like **vi**. The first step in the creation of a table with **ve** is to create a screen definition file with any editor. This can be accomplished with any editor. Here is a ‘screen’ file for the *states* file:

```
% cat states-s
          The States File
st      < st >
state   <  state  >
```

ve uses this screen file to create the table. The rules for screen files are simple: column names go inside the angle brackets. anything outside of angle brackets is just text that appears on the screen.

The space between angle brackets is the viewable window over the field, and isn’t a restriction on how wide the field can really be. After creating a screen file like this:

```
% ve states
```

and the states file will be created. Let’s say **ve** has been used to add new records to our *states* table so that it looks like this:

```
% cat states
st   state
--   -----
CA   California
NV   Nevada
NY   New York
```

A mailing list can be created the same way, by making a ‘screen’ file and then using **ve** to add a few rows:

```
% cat mlist-s
  Yet Another Mailing List

Name           <name>
Street         <address>
City           <city>
State          <st>
```

```
% justify < mlist
name   address    city          st
----   -------    ----          --
Evan   Main St.   Santa Cruz    CA
Rod    Broadway   Ithaca        NY
```

To ‘‘*select the st and name columns from mlist and join it with the states table.*’’

```
% column st name < mlist |
sorttable |
jointable - states
st   name
--   ----
CA   Evan
NY   Rod
```

The **sorttable** command was silent. But it has to be there. Both files to be joined must be sorted. The states file is already sorted. The dash in the **jointable** command means use the standard input, just like the UNIX **join** command.

2.6 Reports

For numeric information, **/rdb**'s standard table output adjusted with a **justify** or **trim** command is often sufficient, especially when combined with **tabletotbl** and the UNIX *tbl* and *nroff/troff* formatters. In addition to these methods, **/rdb** has a **report** command that uses a prototype report form and has built-in command processing capabilities.

Let's look at a sample report form. It's like the screen file for **ve**: text is outside brackets, and column names are inside brackets. Other commands can also go inside the brackets. Here's a report form for the *mlist* file:

```
% cat mlist.form
<name>
<address>
< city >, <st>
         <! date +%D !>
Dear < name >,
This is a computer chain letter.
I am also sending it to:

<!  column name city < mlist |
row 'name != "<name>"' | justify  !>
Bye, <! echo <name> !>.
```

```
% row 'name ˜ /Evan/' < mlist |
report mlist.form
```

```
Evan
Main St.
Santa Cruz, CA
         09/03/89
Dear Evan:
This is a computer chain letter.
I am also sending it to:
name   city
----   ------
Rod    Ithaca
Bye, Evan.
```

Arbitrary text goes outside the angle brackets; column names go inside angle brackets, and *any* arbitrary command or shell program or shell command(s) can go between exclamation marks within angle brackets, and you can *still* specify columns from the current record therein. You can even have reports within reports ...

2.7 The Big Text Field Problem

The 'bug report', 'long text column', and 'every word indexed' problems are all facets of the same situation. Let's say a file has some relatively short columns, and one or more long text columns on which you'd like to use **vi**.

Take the case of a bug report database with associated arbitrarily long narrative descriptions: a solution is to keep the descriptions in a sub-directory called, for example, *bugreports*, one file per record, with the file name being *bugreports/recid* where *recid* is the record identifier from the current record. Then, a CTRL-key is mapped in the .verc file (analogous to the **vi**'s .exrc file) to the command ''*vi bugreports/<recid>*''. This grabs an identifying column from the record, constructs the name of the associated file(s), and pops the user into vi on the named file(s). This is quite flexible even if there is more than one file associated with each record, switching between **ve** files with a keystroke, thus effecting multiple screens: map a CTRL-key to write the record and switch the files, and another to switch back.

A simple report makes a two column table with record id and word for each word in each narrative, allowing for queries like *give me all the bugs mentioning word 'xyzzy':*

```
% cat wordy
#!/bin/sh
(echo "word     id"
echo  "----      --"
for i in [0-9]*
do
    word < $i | awk '{print $1,"'$i'"}'
done) |
sorttable -u
```

Now records having a particular word can be found easily. If speed is a consideration, build an(other) inverted index on the word/id concordance list just created:

```
% cd bugreports
% wordy > bugwords
% index -mi -x bugwords word
% echo xyzzy | search -mi -x bugwords word
```

This produces a list of record id numbers on the standard output. Once you have the record id numbers, one more search is necessary to find the original record in the 'bugs' table. Of course, with the record id numbers, NO search is necessary to find the narrative, because the file name IS the record id.

2.8 Non-Text Data Structures

Suppose a field is a picture, or a sound, or some other non-textual object. The **/rdb** approach is to identify an object resource, with text, within a field in a table, describing the type and location of the object. Fields from the current record can be referenced in the .verc file by the same <*column_name*> specification used in the report program and customized **ve** screen files. This allows a clear, user-defined way of tying ve into X based or other graphical user interfaces. For example, suppose a field contains the name of a file containing an image. A CTRL-key can be mapped in the .verc file to generate the appropriate commands to pop up a new window, call a picture display program, and display the file named in the field into that window. The image file can be in any format, and may reside anywhere on the network. Additional functionality ties the X-window mouse into this system, so that when the mouse is positioned over a field and pressed, the appropriate commands are executed. This approach is the UNIX-like way of integrating all our previous UNIX experience and software expertise into the X user interface, and it's easy to show how it can also be used with the existing report generation features of **/rdb**.

2.9 Large Tables

Large tables are often as easily handled as small tables. When working with very large tables some form of indexing is desirable: hash, inverted sequential secondary, binary (sorted relations), or some form of tree (linked list).

The shell approach is to use the UNIX directory structure as the first (few) levels of tree index. One financial application using **/rdb** involves the 80 megabyte file of World Bank time series from the International Monetary Fund. As distributed, it takes several large machine CPU minutes to peruse this big file and extract a single time series. The file was divided into a directory for each country and within each country directory, a file for each time series, with the file name being the time series code as given by the IMF. Each of those time series files is a **/rdb** file, with columns `YR ANN Q1 Q2 Q3 Q4` and so forth. A separate ‘‘description’’ file in each country directory has a line for each file in the directory, giving `CODE DESCRIPTION UNITS`.

Thus, the time to retrieve any time series (if the country and time series code are known) is *independent of the size of the database.* Queries like ‘‘*which countries have this time series in common?*’’ are answered with the *ls* command. More than one level of index can of course be implemented just by adding directory hierarchies.

UNIX has many commands to traverse directory trees, and to add, delete, and otherwise manipulate nodes. With this approach, nodes are tables, and the plethora of UNIX directory and file handling commands are all relational database manipulation commands.

2.10 Architectural Performance Enhancements

Because of **/rdb**'s shell level approach, enhancements and advantages resulting from multi-processor architectures are immediately available. In a loosely coupled architecture with tcp/ip protocols connecting a number of processors, the following code fragment performs searches in parallel on a number of processors:

```
#!/bin/sh
cat head
(for i in a b c ...
do
    rsh $i "cat keys  |
    search portion.$i |
    headoff" &
done) | continue ...
```

It is is the work already done by the implementors of the shell that collects individual rows from the parallel search processes spun off on each of the processors and arbitrates the output so that only one row at a time is presented to the "continue" process at the end of the parenthesized command. Of course, there is some overhead involved in splitting the data themselves into the portions to be made available for each parallel search, so this technique is appropriate when the speed advantages gained by parallelism overcomes the overhead necessary to split the files.

ON massively parallel SIMD and vector machines such as FPS systems, IBM 3090 vector processors, and MasPar MP platforms, the straightforward method of taking advantage of the architecture to is implement matrix capabilities at the shell level. Matrices are tables, and enclosing the already optimized subroutines in shell callable programs is not problematic.

There is also renewed interest in medium granularity. For example, Cogent Research provides a transputer based LINDA system, automatically distributing multiple processors over the available computing power. The most general MIMD approach is the most difficult to implement at the shell level, and database capabilities that take full advantage of the HyperCube approach, for example, will take more time to fully implement.

2.11 An Example From Trade Literature

UNIX/World magazine printed two articles about fourth generation programming languages (July 1986 and April 1987), and invited several competing companies to produce a sample report using their 4GL systems. Their languages seemed more like COBOL or RPG than a real 4GL. If they represent the standard by which to measure which generation a language is, the shell fits easily into the fourth generation category.

To demonstrate the capabilities of the shell, here are two scripts for producing the sample report called for in the UNIX/World article. These shell scripts use the standard UNIX utilities extended only by the **/rdb** relational database management tools. The first example below produces the data required by the UNIX/World test, but leaves it in a default format. The second report uses the formatting commands necessary to conform exactly to the articles' example.

```
% pay
number  fname    lname     code  hours  rate  total
------  -------  --------  ----  -----  ----  -----
     1  Evan     Schaffer     2      3    75    225
     1  Evan     Schaffer     2      4    75    300
------  -------  --------  ----  -----  ----  -----
     1                               7          525

     2  Mike     Wolf         1      4    85    340
     2  Mike     Wolf         2      5    85    425
------  -------  --------  ----  -----  ----  -----
     2                               9          765

     3  Barbara  Wright       2      5    75    375
     3  Barbara  Wright       1      6    75    450
------  -------  --------  ----  -----  ----  -----
     3                              11          825

                                               2115
```

Here's the shell script that produces this report. Note it only takes only 9 lines of simple, readable code. There's no counting columns or characters. There's no "line-at-a-time" processing, as with the other so-called 4GLs. The shell really shows its power here. Note that although the commands in the shell script appear to consist almost entirely of **/rdb** commands, **/rdb** makes use of UNIX utilities to do its work. Most of the **/rdb** commands are shell scripts or C programs which make extensive use of the UNIX utilities. The 'compute' program, for example, is merely (?) a front end to 'awk'.

```
% cat pay
jointable hours employee |
sorttable code |
jointable -j1 code -j2 number - task |
sorttable number |
project number fname lname code hours rate total |
compute 'total = hours * rate' |
justify > tmp
subtotal -l number hours total < tmp
total total < tmp | justify | tail -1
```

Not shown in the UNIX/World articles are the data tables themselves. The main reason for that is that each of the other 4GLs demonstrated has a special binary format for their files that was not easy to print and that is accessible only through their interface. When programming with the shell, the data is in ASCII files. That means the data are accessible by humans, by UNIX, or by any program you choose to write. Here are the files mentioned in the script:

```
% cat hours
number  hours  code
------  -----  ----
     1      3     2
     1      4     2
     2      4     1
     2      5     2
     3      5     2
     3      6     1
```

```
% cat employee
number  fname    lname     rate
------  -----    -----     ----
     1  Evan     Schaffer    75
     2  Mike     Wolf        85
     3  Barbara  Wright      75
```

```
% cat task
number  name
------  ----
     1  unix/world
     2  Lawrence Livermore
```

The shell program that produces the exact format required, appended as Exhibit 2, is still only 28 lines. A C program to perform the same task would take pages of code. The July 1986 UNIX/World contained example code from nine 4GL's solving the problem. Note that only one language took fewer lines of code to solve the problem, and the **Progress** solution doesn't become shorter when allowed to use another report format, as **/rdb**'s does.

Language	Lines of Code
Progress	20
Rubix	32
Empress/32	34
Unify DBMS	34
filePro 16 Plus	42
Informix-4GL	48
SHAR->IX	48
C/Base	64
Plain English	84

Learning to use the UNIX utilities has a much greater value than learning yet another special programming language. Once a small critical mass of UNIX familiarity is achieved, application development becomes little more than writing simple yet powerful scripts to perform tasks which used to be laboriously performed by hand, or just not done at all.

All these techniques comprise a marriage of the facilities that come with the UNIX system itself, and relational capabilities provided by **/rdb.**

This attitude of not reinventing the wheel is the basis of the shell and **/rdb** approach. All UNIX knowledge is knowledge about databases, and experiences with databases teach more about UNIX. That's why the combination of the **/rdb** extensions to UNIX and the shell command language is a 4GL most appropriate to the UNIX environment.

3. SQL

SQL is another language for querying a database. It's used as the foundation for many contemporary 4GL's. It does not use the stream/operator paradigm, but "nests" queries to pass data from one operator to another. When SQL was developed, UNIX was non-existent, so an entire environment had to be developed to express queries. SQL is *another system* to learn, with little use outside of itself, and typically no relation to the operating environment surrounding it.

SQL does not specify any particular file format. While there is an ANSI standard SQL for expressing queries, implementors are free to store data however they want. In a way, this is a contradiction, because getting away from these walls that stand between data is very important, and was the principal reason that the concept of a database came about. The idea goes under the name of *integrated* and *modeless* software, and most recently, *interoperability.*

There are reasons why SQL based systems are popular, even desirable. SQL based systems are widely available and there is a large body of expertise also readily available. Many U.S. government agencies require access to corporate databases via SQL, especially in the defense industry. SQL is valuable in non-UNIX environments. Partly because SQL is difficult for novices to understand and use, SQL providers typically field a large, helpful support organization. Of course, this drives the price up, and doesn't adequately address the needs that prompted the development of these tools in the first place: making non-experts proficient and productive in the construction of basic database applications.

SQL queries can be easily converted to shell scripts by the **sql2rdb** filter available with **/rdb**. Appended as Exhibit 1 is a sample conversion table.

4. Fifth Generation Systems

There is a distinction to be made between fourth generation languages and CASE tools. The programmer of a fourth generation system must still specify the fundamental algorithms for completing a task, perhaps at a higher level of abstraction, especially of data types. CASE tools, on the other hand, require only a specification of the task, and generate not only the code, but also the algorithm. CASE tools tend to be of very limited domain. An example would be a screen layout tool. The developer draws the positions of the windows wanted, and the tool generates the code to create the windows, manage the text and graphics inside them, and deal with icons and menus.

Some graphical CASE tools (X-rdb, X-Builder and NeXT STEP) are examples of what we might call 5GL's. Using a graphical user interface, these tools allow applications to be built by example. Some force the user to specify actions algorithmically, and some do not. There's even less agreement about what constitutes a 5GL than there is about 4GL's.

5. Discussion

The shell appears to be quite a powerful tool indeed. It is not without limitations, however. First, only a few companies are currently producing tools oriented towards use in shell scripts. **/rdb** remedies the shell's weakness of not being able to store complex data types, and there are many additional tools for for correcting some of the other major limitations, such as numerical computation, statistical analysis, and business graphics output. Although UNIX has applications dedicated to mathematics and numerical analysis, most are themselves large self-contained programs. A programmer needing matrix inversions, for example must adapt existing tools, like System S or SAS, to work within shell scripts, or write a special purpose tool.

The shell needs improvement in the ability to connect multiple pipes together more freely. The original designers didn't anticipate the need for more than linear pipelines. While more complex, non-linear pipes can be created by the use of temporary files, this method is only barely adequate for constructing complex structures such as cyclic streams. Finally, more operators are needed which allow incoming data to be split between or duplicated on multiple output pipes.

The shell shares another problem with weakly typed languages: errors in the format of that data stream can lead to unexpected output. Since there is no method of type or format checking, the programmer must write code which avoids the problem. The interpreted environment does allow careful examination of the stream data as it passes through each operator, which reduces the difficulty of writing error free code. The shell won't lend itself to the sort of correctness proofs offered by the newer CASE tools unless a formal definition is proffered, specifying not only syntax but all the operators as well.

Now that the operator/stream paradigm is being recognized as an extremely powerful model for language design, we expect to see several new tools based upon the principle in the next year. More tools for graphical program design should also start to appear, now that **rdb** has adopted the X window standard and has provided a tool for designing shell pipelines graphically. As more such graphical interfaces become available, less programming experience will be needed to create shell scripts, drastically increasing productivity.

6. Conclusion

The operator/stream paradigm has produced a simple, powerful, general purpose tool. It allows one to prototype or generate a proof of concept in hours or days, when it might have required weeks with C. Although the shell produces slower code than a third generation language, the increasing power of modern computers makes this a minor concern for many tasks. This framework provides an easily visualizable way of manipulating large (or small) amounts of structured data.

While there is currently a shortage of utilities designed for general purpose use within shell scripts, awareness of the potential of shell programming is spreading, and more packages are being written outside of the traditional monolithic program tradition. In this way, computing is coming full circle, returning to the original concepts of Von Neumann, whose computing paradigm embodies the stream of sequential memory passing by the operator of the central processing unit.

Exhibit 1. SQL Conversion Table

SQL	UNIX and /rdb
select col1 col2 from filename	column col1 col2 < filename
where column - expression	row 'column == expression'
compute column = expression	compute 'column = expression'
group by	subtotal
having	row
order by column	sorttable column
unique	uniq
count	wc -l
outer join	jointable -a1
update	delete, replace
nesting	pipes

Exhibit 2. Exact Format Shell Program

Here is the modified shell program that produces the exact article format:

```
% cat payexact
echo "Employee                             Charge"
jointable hours employee |
sorttable code |
jointable -j1 code -j2 number - task |
sorttable number |
project number hours code fname lname rate name total |
compute 'total = hours * rate; name = sprintf("%s %s",fname,lname)' |
project number name code hours rate total > tmp
compute 'if (name == prev) name = "";\
     prev = name;\
     hours = sprintf("%4.2f",hours);\
     rate = sprintf("%6.2f",rate);\
     total = sprintf("%7.2f",total)' < tmp |
subtotal -l number hours total |
compute 'if (code ˜ / / && code !˜ /-/) code = "* Employee Total";\
     if (code ˜ / / && code !˜ /-/) number = ""' > tmp1
rename name "Employee Name" < tmp1 |
justify -r number hours rate total -l "Employee Name" -c code |
sed '/---/d
     s/ˆ/  /
     s/rate/  rate/
     s/total/  total/'
TOTAL=`project total < tmp |
total |
compute 'total = sprintf("%10.2f",total)'|
headoff`
echo "                              \
** Report Total                                    $TOTAL"
```

7. References

1. B. Kernighan and R. Pike, *The UNIX Programming Environment*, Prentice Hall, Englewood Cliffs, NJ, 1985.

2. S. Kochan and P. Wood, *UNIX Shell Programming*, Hayden Book Company, 1985.

3. S. Prata, *Advanced UNIX - A Programmers Guide*, Howard W. Sams and Co., Inc., 1985.

4. A. Winston, "4GL Faceoff: A look at Fourth-Generation Languages," *UNIX/World*, July 1986, pp. 34-41.

5. S. Misra and P. Jalics, "Third-Generation versus Fourth-Generation Software Development," *IEEE Software*, July 1988, pp. 8-14.

6. R. Manis, E. Schaffer and R. Jorgensen, *UNIX Relational Database Management*, Prentice Hall, Englewood Cliffs, NJ, 1988.

7. J. Verner and G. Tate, "Estimating Size and Effort in Fourth-Generation Development," *IEEE Software*, July 1988, pp 15-22.

8. V. Matos and P Jalics, "An Experimental Analysis Of The Performance Of Fourth Generation Tools On PCs," *Communications of the ACM*, November 1989, pp. 1340-1351.

9. R. Manis, M. Meyer, *UNIX Shell Programming*, Howard Sams, 1987