NoSQL

A Relational Database Management System

Carlo Strozzi

Italian Linux Society.

v3.1 - August 8th, 2001

NoSQL is a shell-level relational database system for UNIX. Unlike most DBMS'es, NoSQL is not a single large program, rather it is a set of small programs that are run from the shell. This allows the user to utilize the full power of the Unix shell in coordination with NoSQL. It uses the Operator/Stream paradigm described in "Unix Review", March, 1991, page 24, entitled "A 4GL Language". Actually, calling NoSQL a DBMS is some of a misnomer, and it should rather be called a Database Management Toolkit. It's purpose is simply to offer a "Data Dictionary" view over structured ASCII files, allowing an application program to refer to each piece of data by name rather than by its physical position in a file. There are a number of "operators" that each perform a unique function on the data. The "stream" is supplied by the UNIX Input/Output redirection mechanism. Therefore each operator processes some data and then passes it along to the next operator via the UNIX pipe function. This tends to be rather efficient as UNIX pipes are implemented in memory. NoSQL is compliant with the "Relational Model".

Contents

Chapter 1

Foreword and Introduction

1.1 Copyright

1.2 Preface

This working draft describes, and provides instructions for the use of, NoSQL (I personally like to pronounce it *noseequel*), a derivative of the RDB DataBase system. The RDB system was (and still is) developed at RAND Organization by Walter W. Hobbs. Part of the NoSQL code, as well as large parts of the text of this document, have been taken directly from RDB, so a good share of the credit goes to the original author.

Other major contributors to the original RDB system, besides W. Hobbs, were:

Chuck Bush

Don Emerson

Judy Lender

Roy Gates Rae Starr

People who helped with turning RDB into NoSQL:

David Frey

Maurizio (Masar) Sartori

Vincenzo (Vicky) Belloli

Giuseppe Paternò

Paul Lussier

Seth LaForge

The NoSQL.png logo has been kindly provided by Kyle Hart.

NoSQL tends to be biased in favour of *Linux*. This means that, wherever it matters, NoSQL makes use of the GNU versions of the various UNIX commands, as those are the ones normally found on this UNIX workalike. NoSQL is *Free Software*, released under the terms of the *GNU General Public License*. As such, it fully qualifies as *Open Source* Software.

1.3 Introduction

A good question one could ask is "With all the relational database management systems available today, why do we need another one ?". The main reasons are:

1. Several times I have found myself writing applications that needed to rely upon *simple* database management tasks. Most commercial database products are often too costly and too feature-packed to encourage casual use. There is also plenty of good free databases around, but they too tend to provide far more than I need most of the times, and they too lack the shell-level approach of NoSQL. By contrast, NoSQL takes a very simple approach (even simplistic, as some may argue :-), but that is exactly its distinguishing feature. Admittedly, having been written mostly with interpretive languages (Shell, Perl, AWK), NoSQL is not the fastest DBMS of all, at least not always (a lot depends on the application).

2. NoSQL is easy to use by non-computer people. The concept is straightforward and logical. To select rows of data, the 'row' operator is used; to select columns of data, the 'column' operator is used.

3. The data is highly portable to and from other types of machines, like Macintoshes or DOS computers.

4. The system should run on any UNIX machine that has perl(1) and mawk(1) installed.

5. NoSQL essentially has no arbitrary limits, and can work where other products can't. For example there is no limit on data field size, the number of columns, or file size (the number of columns in a table may actually be limited to 32.768 by some implementations of the AWK interpreter, including `mawk` I think).

Note: NoSQL has only been tested with mawk(1), that is Mike Brennan's implementation of the AWK programming language, and will most likely *not* work out-of-the-box with any other AWK, including gawk(1). While getting NoSQL to work also with gawk(1) should not be difficult, making it work with other AWKs may prove hard, if at all possible.

As its name implies, NoSQL is *not* an SQL database. The rationale behind this is well explained in the accompanying paper *4gl.ps* (Postscript), or *4gl.txt* (ASCII).

The data is contained in regular UNIX ASCII files, and so can be manipulated by regular UNIX utilities, e.g. ls, wc, mv, cp, cat, more, less, editors like 'vi', head, RCS, etc.

The form of each file of data is that of a relation, or table, with rows and columns of information.

To extract information, a file of data is fed to one or more "operators" via the UNIX Input/Output redirection mechanism.

There are also programs to generate, modify, and validate the data. A thorough discussion on why this type of relational database structure makes sense is found in the book, "UNIX Relational Database Management", Reference #2.

It is assumed that the reader has at least a minimum knowledge of the UNIX Operating System, including knowledge of Input/Outout redirection (e.g., STDIN, STDOUT, pipes).

Again, the key feature of NoSQL (and other similar packages mentioned in this manual), is its **close integration with UNIX**. Real-world problems are tipically more complex than the data models provided by many DBMS'es. Actual applications, and Web-based ones make no exception, are complex puzzles made up by many small pieces, several of which are data-related. Unlike other fourth generation systems, NoSQL is an extension of the UNIX environment, making available the full power of UNIX during application development and usage. NoSQL was designed with the UNIX shell language as its user interface. This level of integration removes the need to learn yet another set of commands to use and administer the database system. A database is just a file, and can be maintained like all other files that the user owns or has access to. Because NoSQL commands are executable programs, the UNIX shell is inherited as the primary command language of the database; no other proprietary database scripting language, to my knowledge, is as powerful and flexible as the UNIX shell. The shell-level nature of NoSQL encourages casual use of the system, and successful casual use leads to familiarity and successful formal use. This concept is much more thoroughly explained in the paper "The UNIX Shell As a Fourth Generation Language", included in the NoSQL documentation tree with the file name *4gl.ps* (Postscript) or *4gl.txt* (ASCII), that shows why the UNIX shell is an excellent tool for scripting database access.

1.4 Perl and the Operator/Stream Paradigm

As stated in the Abstract, NoSQL uses the Operator/Stream DBMS Paradigm. The main reason why I decided to turn the original RDB system into NoSQL is that the former is entirely written in Perl. Perl is a good programming language for writing *self-contained* programs, but Perl's pre-compilation phase and long start-up time are worth paying only if once the program has loaded it can do everything in one go. This contrasts sharply with the Operator/Stream model, where operators are chained together in pipelines of two, three or more programs. The overhead associated with initializing Perl at every stage of the pipeline makes pipelining somewhat inefficient. A better way of manipulating structured ASCII files is using the AWK programming language, which is *much* smaller than Perl, is more specialized for this task and very fast at startup (on my Pentium II Linux /usr/bin/mawk (POSIX AWK) is just 99K. Perl 5 is almost 500K. You get the point).

1.5 Bug reports

There is a mailing list for discussions related to NoSQL. The address is *noseequel@texne.com*. To subscribe simply send a message to *minimalist@texne.com* with the word "subscribe" (without the quotes) in the message subject.

Please send bug reports (fixes are most welcome :-) to the same list *noseequel@texne.com*. Always include as much information as possible, especially the content of file `nosql_version`, which is created in the NoSQL installation directory during install. By 'bug reports' I mean not just errors in the code, but also grammatical mistakes, typos and bad English constructions in the documentation, as English isn't my native language :-)

Chapter 2

Data formats

2.1 NoSQL table (relation) structure.

A table (or *relation*) is an ordinary ASCII file, with some additional rules that make it possible to use it as a database table. The file has records (rows) and fields (columns). The relation, or table structure is achieved by separating the columns with ASCII TAB characters, and terminating the rows with ASCII NEWLINE characters. That is, each row of data in a file contains the data values (a data field) separated by TAB characters and terminated with a NEWLINE character. Therefore a fundamental rule is that data values must NOT contain TAB characters.

The first section of the file, called the header, contains the file structure information used by the operators. The rest of the file, called the body, contains the actual data values. A file of data, so structured, is said to be a 'table'.

The header consists of exactly two lines that contain the structure information: the column name row and the *dashline*. The fields in the column name row contain the names of each column, and are separated from each other by a single TAB character. The dashline is a set of dashed lines, one set for each column, separated by single TAB characters. The dashline signals the start of the actual data rows and its sole purpose is to make the header visually easy to find.

The column names are case sensitive, i.e. 'COUNT' is different from 'Count'. The guideline for characters that may be used in column names is that alphabetic, numeric, and the underscore (_) are good choices. Numeric-only column names are not allowed. No rows, except the dashline, should contain only dashes and TABs.

The TAB character must never be used in column names, nor should spaces or UNIX I/O redirection characters (<,>,|) be used. To be on the safe side, column names should always start with a letter and contain only upper and lower case letters, numbers and the underscore (_). The following names are reserved to the awk programming language, and should not be used to indicate column names:

BEGIN, END, break, continue, else, exit, exp, for, getline, gsub, if, in, index, int, length, log, next, print, printf, split, sprintf, sqrt, sub, substr, while, and possibly others. Refer to the mawk(1) man page. Furthermore, the '_nosql_' prefix is reserved for NoSQL internal use, and should never be used at the beginning of column names.

For instance, suppose you have a table that maps names to nicknames, then its two columns could be called *Name* and *Nickname*. Some NoSQL operators create new columns that have the same name as pre-existing table columns, with lower-case letters prepended to them. This is why you *really* should stick to these rules.

Not abiding by these naming rules may still work, but there may be unexpected results.

A sample table (named SAMPLE) that will be used in later examples is shown in Table 1. The picture in Table 1 is for illustrative purposes; what the file would actually look like is shown in Table 2, where a TAB character is represented by '<T>' and a NEWLINE character is represented by '<N>'.

```
                Table 1

             table (SAMPLE)

        NAME    COUNT   TYP     AMT
        ----    -----   ---     ---
        Bush    44      A       133
        Hansen  44      A       23
        Jones   77      X       77
        Perry   77      B       244
        Hart    77      D       1111
        Holmes  65      D       1111
```

```
               Table 2

        table (SAMPLE) actual content

        NAME<T>COUNT<T>TYP<T>AMT<N>
        ----<T>-----<T>---<T>---<N>
        Bush<T>44<T>A<T>133<N>
        Hansen<T>44<T>A<T>23<N>
        Jones<T>77<T>X<T>77<N>
        Perry<T>77<T>B<T>244<N>
        Hart<T>77<T>D<T>1111<N>
        Holmes<T>65<T>D<T>1111<N>
```

It is important to note that only actual data is stored in the data fields, with no leading or trailing space characters. This fact can (and usually does) have a major effect on the size of the resulting datafiles (tables) compared to data stored in "fixed field width" systems. The datafiles in NoSQL are almost always smaller, sometimes dramatically smaller.

A table can also be represented in a different format, called *'list format'*. The *list* format of the above SAMPLE table is:

```
NAME  Bush
COUNT 44
TYP   A
AMT   133

NAME  Hansen
COUNT 44
TYP   A
AMT   23

NAME  Jones
```

```
COUNT 77
TYP   X
AMT   77

NAME  Perry
COUNT 77
TYP   B
AMT   244

NAME  Hart
COUNT 77
TYP   D
AMT   1111

NAME  Holmes
COUNT 65
TYP   D
AMT   1111
```

The actual contents of a table in 'list' format, showing newlines and TABs is:

```
<N>
NAME<T>Bush<N>
COUNT<T>44<N>
TYP<T>A<N>
AMT<T>133<N>
<N>
NAME<T>Hansen<N>
COUNT<T>44<N>
TYP<T>A<N>
AMT<T>23<N>
<N>
NAME<T>Jones<N>
COUNT<T>77<N>
TYP<T>X<N>
AMT<T>77<N>
<N>
NAME<T>Perry<N>
COUNT<T>77<N>
TYP<T>B<N>
AMT<T>244<N>
<N>
NAME<T>Hart<N>
COUNT<T>77<N>
TYP<T>D<N>
AMT<T>1111<N>
<N>
NAME<T>Holmes<N>
COUNT<T>65<N>
TYP<T>D<N>
AMT<T>1111<N>
<N>
```

Long lines, i.e. lines that are too long to fit in the width of the screen, may be folded over multiple rows in the 'list' format, provided that each continuation row starts with one or more spaces (blanks, not TABs). Field (column) names need to be separated by the associated data by exactly one TAB characters. The data part may contain physical TABs and newlines, which will be turned into '\t' and '\n' escapes respectively by the 'listtotable' operator when the list is turned into a table.

```
COMMENTS  This is a very looong comment, that I want to fold over
  multiple lines.
```

and the actual content is :

```
<N>
COMMENTS<T>This is a very looong comment, that I want to fold over<N>
<T>multiple lines.<N>
<N>
```

As we will see, there are NoSQL operators that convert back and forth between 'table' and 'list' formats.

It is suggested, though not required, that table file names be given the filename extension '.rdb', to make them recognizable right away.

2.2 NoSQL and Data-Types.

Unlike most other database systems, NoSQL knows nothing about data-types. Everything is just a string, that occurs between one TAB character and the next one. This was done on purpose, of course, as NoSQL tables can be accessed in a number of ways, even directly with a text editor. NoSQL has no way of enforcing any data-typing that we may possibly establish, so why bother about types at all. This model goes well with the plethora of text utilities that come with most Unices, and with Linux in particular, and is a very natural way of representing data, more on the human-level than other conventions. The drawback is that it is up to the application to enforce datatypes if necessary.

As I have already pointed-out, NoSQL should be seen just as a simple *data-dictionary toolkit*. Its main purpose is to attach names to slices of an otherwise flat-data file. Having a dictionary means that you can reference individual pieces of data by name rather than by their physical position in the file, thus attaining a basic level of information abstraction.

A table column can contain anything except physical tabs and newlines. The data itself can be anything that is considered to be text according to the local character-set (mine is iso-8859-1, or Latin1). A field can even contain an entire text-encoded file (a *BLOB*). Common encodings are *uuencode*, *base64* and *quoted-printable*. Large fields may of course break AWK or the other utilities, but that must be seen as a limitation in those programs or in the operating system, not something pertaining to the paradigm.

A valid NoSQL table needs always to contain the header. Keeping the latter in a separate file is possible but strongly deprecated.

Table editing/writing/locking/unlocking/versioning should not be seen as core NoSQL features, but simply add-on facilities. In real applications the locking policy may become quite complicated and should be provided by the application program itself, according to its needs. The same is true for modifying/versioning a table and ensuring overall database consistency.

The structure of a NoSQL table is record-oriented, so that it can easily be acted upon with the wealth of existing Unix utilities, which are mostly record-oriented. This does not mean that a table cannot map a more complicated structure, like an XML document or any other hierarchical tree-like structure. Such *higher-order* dictionaries will not pertain to the paradigm though, but rather to the application that uses the table.

2.3 Notes on similar database packages.

Besides NoSQL and RDB there are other UNIX DBMS's, both commercial and free, that are based on ASCII tables. A commercial implementation is /rdb, by *Revolutionary Software*, while among the free ones there are *Starbase*, developed at the Harvard Smithsonian Astrophysical Observatory, and Gunnar Stefansson's *reldb*, a collection of interesting tools available at sites that carry archives of the *comp.sources.unix* Usenet newsgroup.

The ASCII table format of those database engines is very close to that of NoSQL, therefore data can easily be converted back and forth between them and NoSQL.

Chapter 3

Using NoSQL

3.1 Operators

The NoSQL system comprises a set of programs called Operators.

Each operator is a separate program module that performs a unique function on the data.

At one time NoSQL used to have extra operators, like *body*, *dashline*, etc. At some point I realized how useless they were, so I dropped them. After all they were just calls to ordinary UNIX utilities. There was really no point in providing special operators just for them, as their function could be done by applying those system commands directly. For instance:

body

Same as : `tail +3 < table`

dashline

Same as : `sed -n 2p < table`

headline

Same as : `head -1 < table`

fieldsof

Same as : `head -1 < table`

header

Same as : `head -2 < table`

see

Same as : `cat -vte < table`

Once again, this shows how powerful the UNIX operating system already is of its own, and how handy it can be for an add-on package like NoSQL to be able to tap into this power without having to re-invent the wheel.

Invoking NoSQL programs and operators is straightforward :

```
command [options] [arguments]
```

where `command` is the name of the desired NoSQL operator or utility. Make sure the directory containing the NoSQL executables, usually /usr/local/nosql/bin/, is in your PATH.

Most operators take a `-help` switch, that will show the list of available command-line options and arguments, plus some usage notes.

Here follows a list of NoSQL operators, with a short description of the function of each:

addcolumn

Create new empty columns in the rightmost position of a table.

addrow

Append a new empty record to a table.

column

Pick columns by name, output columns in listed order.

compute

Compute an arbitrary expression using column names.

depend

Test for "functional dependency" betwen two columns.

edittable

Use an editor to allow modifications to a table.

envtolist

Convert environment variables into a one-record list.

envtotable

Convert environment variables into a one-record table.

filter

Run standard utilities against a table.

filetolist

Convert a (possibly binary) file into a one-record list.

gregorian

Translate selected date columns from Julian to calendar format.

index

Generate table index files to be used by 'search'.

islist

Check whether an input file has a valid 'list' format.

istable

Check whether an input file has a valid table format.

jointable

Natural or "Master/Detail" join of two tables on a common field.

julian

Translate selected date columns from calendar to Julian format.

justify

An alias for prtable.

listtoform

Take a file in 'list' format and use the values from the last record to replace special tags on a template file.

listtotable

Take a file in 'list' format and makes it into a table.

maketable

Build a valid table header from a template file.

number

Insert a unique record identifier into a table.

project

An alias for column.

prtable

Quick and easy table formatter for character displays.

random

pick one or more table records at random.

rename

Rename a column.

repair

Append empty data fields to rows to make them match the table header.

rmcolumn

Remove one or more columns.

row

Select rows based on arbitrary AWK expressions.

search

Select rows based on a multi-column key of a sorted or indexed table.

select

An alias for row.

sorttable

Sort a table by one or more columns.

soundex

Compute Knuth's soundex codes for selected columns.

subtotal

List subtotals of specified columns.

tabletoform

Replace special tags in a template file.

tabletolist

Convert a table into the corresponding 'list' format.

tabletorc

Convert a table into rc(1) variable assignments.

tabletosh

Convert a single-record table into sh(1) variable assignments.

template

Build a table template file for the input table.

union

Concatenate multiple union-compatible tables.

unique

Make a table unique on the primary key field.

update

Insert/update/delete table rows based on the contents of an edit table.

viewtable

Visualize a table in a nice list-like format.

Chapter 4

Extracting information from tables

4.1 General

The following shows some examples of how the system is usually used, which involves a combinations of operators. Using the table named 'sample' the command:

```
column NAME AMT TYP < sample | sorttable TYP | justify
```

gives the output:

```
NAME     AMT  TYP
------  ----  ---
Bush     133  A
Hansen    23  A
Perry    244  B
Hart    1111  D
Holmes  1111  D
Jones     77  X
```

Note that the column COUNT was excluded by the 'column' operator, and that the order of the selected columns was changed from that in the input table. Of course to save the output in a file, (redirection of STDOUT into a file) something like the following is used:

```
column ... < sample | sorttable TYP | justify > file.out
```

An example using the operator 'row' on the table 'sample' is:

```
row 'AMT < 900' < sample | column NAME AMT TYP |
      sorttable NAME | justify
```

Note how the row selection expression is surrounded by single quotes, to prevent its interpretation by the shell. Here we select some rows using 'row', select some columns using 'column', sort what we have with 'sorttable', and print with 'justify'. The output is:

```
NAME    AMT  TYP
------  ---  ---
Bush    133  A
Hansen   23  A
Jones    77  X
Perry   244  B
```

Chapter 5

Generating or modifing NoSQL tables

5.1 Generating new tables

Any editor may be used to construct or modify a table, since it is a regular UNIX file, and this 'direct editing' method is occasionally used, especially for small amounts of data. However, avoid using an editor that destroys TAB characters.

To generate a new table the best plan (and usually the safest one) is to first generate a template file, then convert it to table format and add the rows of data. Any convenient editor may be used to generate a template file. To convert it to a table the command 'maketable' may be used, which will produce an empty table. Next use the operator 'edittable' to enter rows of data.

A typical template file is shown below:

```
# These are lines of table documentation. They can be of any length,
# and any number of such lines may exist.
# Each line must start correctly, e.g with "#", surrounded by any
# number of spaces and/or TABs.

Name      Name of item
Type      Type: 1,2,3,7,9,A,B,X
Count     Number of items
K         Constant modifier
SS7       Special status for type 7
Size      In Kilobytes
```

The above template file contains the necessary elements to describe a table of six columns: table documentation (the comment lines that each start with a sharp sign '#'), column name ("Name", "Type", "Count", ...), and column documentation for each column (the text at the end of each column line).

To build the final table header, use the command:

```
maketable < table.tpl
```

where `table.tpl` is the template file described above. The command will produce the correct table header to STDOUT, (that can be redirected to a file as usual) :

```
Name    Type    Count   K   SS7 Size
----    ----    -----   -   --- ----
```

5.2 Modifying existing tables

I will now explain how NoSQL tables can be modified, both manually and with automated programs.

Basically there are two ways to manually modify an existing table: either direct editing with a TAB-conscious text editor, like *vi(1)*, or by using the 'edittable' command. The latter method is recommended, especially in a multi-user environment where multiple concurrent editing sessions may occur against the same table file.

The 'edittable' utility can be used to add new rows, change or delete existing rows of data in a table. The optional '–list' switch tells 'edittable' to convert the table to 'list' format for editing, which makes working with wide tables much more comfortable. The editor called by 'edittable' defaults to *vi*, or to whatever is set in the environment variable EDITOR.

After editing, the table is automatically checked for validity by 'edittable', using 'istable'. If structure errors are detected, the program prompts you for what to do (re-edit, quit, etc.).

5.3 Big tables

One thing to point out here is that the only way to update a table, both manually and with a program, is to capture the SDTOUT stream of an operator and write the results back to a table file, paying attention to not overwriting the original input file while it is being read (familiarity with the Unix shell is expected from the user). For instance, you have better not do:

```
operator < table > table
```

or you will end up with 'table' being zero'ed! The correct procedure is:

```
operator < table > table.tmp
mv table.tmp table
```

At the moment NoSQL does not provide a record-level update facility. This may be inefficient if you need to do frequent changes to a large table, for instance with a CGI program behind a Web server, as the whole table needs to be rewitten upon each change. But don't be disappointed by this. There are ways to circumvent this apparently major limitation of the NoSQL paradigm.

The obvious "trick" is to try and keep your tables small. Silly a suggestion as it may sound, before laughing at it please have a look at section 2.9 of the included *4gl.ps* or *4gl.txt* paper. Remember that NoSQL works *with* UNIX, not *in addition* to it. The underlying UNIX file system, that most commercial databases tend to disregard, if used creatively can provide an extremely powerful way of pre-organizing your tables (relations) efficiently and keep them small (where small means roughly within a few hundred kilobytes in size).

If you really *must* do frequent modifications to a big indexed table, then you can still do it efficiently by applying your changes to a separate file rather than to the actual table, and merging the changes back into the big table (as well as re-build its indices) only every now and again, with a batch job that can be run in the background, overnight or when the system activity is low. The following example will try to explain this better.

Suppose we have the large indexed table `bigtable.rdb` and we need to insert/change/delete one or more records. What we can do is:

- Create a *table journaling buffer*, say `bigtable.tjb`, with exactly the same header as `bigtable`, but containing only those records that we want to insert/update into or remove from `bigtable.rdb`. The entries in `bigtable.tjb` will have to be in a format suitable for the 'update' operator.
- Whenever we fetch data from `bigtable.rdb` we will have to do it in three logical steps. The first step is to use 'search' on `bigtable` to take advantage of the indices. This will produce an intermediate output that will then have to be merged into `bigtable.tjb`, and the final output will undergo the original query statement again.
- Any updates to `bigtable.rdb` will rather be done to `bigtable.tjb`, with the sintax described in the documentation of the 'update' operator. Unlike `bigtable.rdb`, where records must be kept sorted on the primary key field (i.e. on the leftmost table column), the records in `bigtable.rdb` do not need to be sorted in any particular order.

This may seem complicated, but it isn't much, really. Say `bigtable.rdb` contains two columns, SSN (for Social Security No.) and SURNAME, where the unique table key is on SSN, and we have built a secondary index on the SURNAME field. If we want to extract all the rows which SURNAME field is equal to "Smith", the query necessary to take advantage of a fast search method on `bigtable.rdb` and to account for the presence of `bigtable.tjb` is:

```
printf 'SURNAME\n-------\nSmith\n' |
  search --index bigtable.rdb.x.SURNAME | update bigtable.tjb |
  row 'SURNAME=="Smith"'
```

As you can see, the trick is:

- Perform an indexed 'search' on `bigtable.rdb` to quickly obtain a much smaller (possibly empty) subset of its data.
- Merge the first output with `bigtable.tjb` *on-the-fly* during the query.
- Do a sequential post-query on the final output.

As shown, these logical steps can be performed in one single line of UNIX shell code!

5.4 Concatenating tables

The need to concatenate tables comes up every so often and although it is simple to do it may not be obvious. The UNIX 'cat' command can not be used as it would result in duplicating the header and thus make an invalid table. And of course, only tables with the same header, i.e. tables that are said to be "union-compatible", should be concatenated, otherwise an invalid table would result (in this case it could be a gross inconsistency if the number of columns were different). If we have two tables, TABA and TABB, then to concatenate TABB onto the end of TABA we use the command:

```
union TABA TABB
```

Chapter 6

Conversion of external data into NoSQL tables

The easiest and probably most efficient way to convert data in files that were received from an external source is to use the AWK Programming Language. If the data file to be imported has a fixed field structure then GAWK (GNU-AWK) may be preferable over standard AWK, as the former can cope easily with fixed-field records. Note that the table header must be inferred manually from the structure of the input file.

The best way to accomplish the above is to first construct a template file from known or observed information in the external data file, then generate the header using 'maketable'. Then run the AWK, script appending the output to the header file. Then run 'istable' to make sure everything went allright.

Another possibility is using the PERL Programming Language. A typical PERL script for such a conversion from a file of fixed column data is shown in Table 7. The last two lines do most of the work. Note that this process converts the data into a table body only; the table header must still be generated manually.

Table 7

PERL SCRIPT FOR DATA CONVERSION

```
#! /usr/local/bin/perl
$templ = "A6 A12 x A5 x6 A18 A17 x A38 x4 A2 x A x2 A15 x3 A6 A4 x2" .
         " A2 x4 A5 x A3 A3 A4 x26 A12 A12 A12 A29 x7 A3 A3 x6 A6 A6 A8" .
         " A4 A3 x3 A3 x3 A8 x10 A2 x4 A12 x138 A6 x66 A24 A36 A12 x6" .
         " A6 A6 x6 A6 A24 A12 x50 A42 A42 A42 x34 A12 A12 x186 A6" ;
$0 =~ s-.*/-- ;
$HelpInfo = <<EOH ;

    Strip out and reform an 'external' data file into a table.

Usage:  $0  [options]  file
```

```
Options:

    -help    Print this help information.

Strips out the first 46 fields from an 'external' data file and
reformats it into 'table' format (TAB delimited with NEWLINE at end).
Output is on STDOUT.
EOH
while ( $ARGV[0] =~ /^-/ ) {                            # Get args
    $_ = shift ;
    if( /-h.*/ ){ die $HelpInfo ; }
    die  "Bad arg: $_\n", $HelpInfo ;
}
while(<>){
    @a = unpack( $templ, $_ );
    print join( "\t", @a), "\n" ;
}
```

Chapter 7

Data access control

7.1 General

Since tables are regular UNIX files, I have so far found no need to implement, and have the associated overhead of, general data access controls in NoSQL. Setting the UNIX permissions on files or directory has proven very useful and effective. This is another example of how the NoSQL system works "with" UNIX, not in addition to it, e.g. not duplicating UNIX functions.

The Revision Control System (RCS) is one of the best configuration management tools available, it can be used for version control of many types of files, including tables. Most NoSQL operators can utilize tables that are under RCS control by using a command like:

```
co -p table | row  ... | column  ... | justify
```

This checks out a table, sends it to 'row', then to 'column', and finally prints the data with 'justify'. In general, any series of commands necessary can be constructed to do a given task even if the tables are checked into RCS.

7.2 Write concurrency control

When the 'edittable' operator is used (which modifies a table in place) there could be a possibility of silmultaneous writing of a table by multiple users. That is, if two or more users, on the same computer or perhaps on different computers on a network, attempted to modify a given table with 'edittable' at the same time, the table could become corrupted. To prevent this, write concurrency control is provided by the use of a lockfile, and is in effect whenever a user runs 'edittable' against the table.

The name of the lockfile is the name of the table being modified, with a suffix of ".lock". For example a table named "main.rdb" would have a lockfile named "main.rdb.lock". The lockfile is placed in the same directory as the table itself and is removed after the modification process is complete, even if the operation is aborted with an INTERRUPT signal (CONTROL-C or <DEL>). When an attempt to use this utility is made and there is already a valid lockfile associated with the referenced table an error message is produced and the utility exits.

Chapter 8

Fast access methods

The NoSQL operator 'search' may be used to execute one of two fast access methods: binary or hashtable search.

These methods are useful when the key field values of a number of rows in an existing, large table is known in advance, a common situation.

One example of updating a table using these methods is as follows. First, 'search' is used to quickly obtain a new, small table consisting of the desired rows from the existing, large table. Then 'edittable' is used to update the data in the new table, including generating new rows, changing some rows, and marking certain rows for deletion, if desired. The next stage would be to use 'update' to combine the new and old tables into a new, large table, which will still be in sort order (both the old table and the new, small one will still exist for backup and/or journaling purposes). Note that the last step could be done in the background.

More explanations on handling big tables can be found in section 5.3 (Big tables).

Chapter 9

Limits, a few minor ones

The following limits apply.

There must not be any ASCII TAB characters in the data. This is the primary limit as the ASCII TAB character is the delimiter in tables. The following names are reserved to the awk language, and should not be used to indicate column names:

BEGIN, END, break, continue, else, exit, exp, for, getline, if, in, index, int, length, log, next, print, printf, split, sprintf, sqrt, substr, while, and possibly others, depending on the implementation of your awk (i.e. mawk, gawk, etc.). Refer to the man page and the documentation of you awk interpeter.

Horizontal TABs and newlines, although forbidden as such in table data, can be conveniently represented by means of the ASCII strings '\t' and '\n' respectively. This rule applies to tables only. Files in 'list' format can contain any characters literally, including physical TABs and newlines.

The number of columns in a table may be limited to 32.768 by some AWK implementations (I think `mawk` is one of those). It should not be a problem though, as it is a very high number anyway. In spite of this, mawk is very fast and I recommend it over other AWK implementations.

A more serious drawback of the operator-stream paradigm is that it is process-based. This means that an average pipeline will open several processes at once, one or more for each operator. On complex queries this can lead to exceed the max. No. of child processes allowed by your operating system. This limit is O.S. specific and it can usually be overcome by getting the system administrator to increase this value as needed.

Chapter 10

More on speed

I have done some benchmarking, though not at all exhaustive, to assess the actual NoSQL speed with respect to other similar packages, and here are the results :

```
Input table:

  No. of fields  = 18
  No. of records = 21000
  Total size     = 3.4 MB

  Column names:

    seq
    NOME
    PREF
    INTERNO
    REP
    POSTO
    CITTA
    NODO
    USERID
    DEPTT
    CAPIN
    EMPTYPE
    EMPNUM
    ADDITIONAL
    MGRCC
    TELEFONO
    MGR
    MGRNUM


Machine: P200, 64MB RAM, 1GB Wide-SCSI HD

OS: Debian GNU/Linux 2.0 (kernel 2.0.35)


RDB (Perl):
```

```
  time col POSTO NOME PREF USERID < table.rdb |
    col POSTO NOME PREF | col NOME >/dev/null

  real    0m13.937s
  user    0m10.340s
  sys     0m0.440s


NoSQL (awk):

  time column POSTO NOME PREF USERID < table.rdb |
      column POSTO NOME PREF | column NOME >/dev/null

  real    0m3.910s
  user    0m3.250s
  sys     0m0.280s


Starbase (C):

  time column POSTO NOME PREF USERID < table.rdb |
    column POSTO NOME PREF | column NOME >/dev/null

  real    0m1.230s
  user    0m0.720s
  sys     0m0.170s


                    -------------------

Input tables:

  join1.rdb:

    No. of fields  = 4
    No. of records = 21000
    Total size     = 660 KB

    Column names:

      NOME
      PREF
      INTERNO
      REP


  join2.rdb:

    No. of fields  = 4
    No. of records = 21000
    Total size     = 700 KB

    Column names:

      NOME
```

```
      POSTO
      CITTA
      NODO


Machine: P-II 233, 32MB RAM, 3GB EIDE U-DMA HD

OS: Debian GNU/Linux 2.0 (kernel 2.0.35)


RDB (Perl):

  time jointbl -md < join1.rdb NOME join2.rdb >/dev/null

  real    0m10.630s
  user    0m10.540s
  sys     0m0.020s


NoSQL (awk):

  time jointable -a 1 -j NOME - join2.rdb < join1.rdb >/dev/null

  real    0m1.402s
  user    0m1.340s
  sys     0m0.060s


Starbase (C):

  time jointable -A1 -j NOME join1.rdb join2.rdb >/dev/null

  real    0m0.881s
  user    0m0.860s
  sys     0m0.020s
```

Chapter 11

Notes for contributors

I will welcome contributions to NoSQL, both in the form of new operators, improvements to the current code and to the documentation, porting to other operating systems, and so on. Contributed code, however, will have to abide by the general philosophy of NoSQL. Operators will have to be lightweight ones, written in C, mawk(1) or any other language that has a small memory footprint and allows for fast startup of the command. While Perl is welcome for utilities that are meant to be used interactively by users at the shell prompt, and for seldom-used commands in general, it is fairly bad (i.e. unefficient) for operators that may occur several times along the same pipeline. See section 1.4 (Perl and the Operator/Stream Paradigm). Common Unix shells, like *bash(1)* and *zsh(1)*, which are great for interactive use, have similar drawbacks, but the availability of lightweight ones, like *rc(1)*, *ash(1)*, or even *ksh(1)*, makes it viable, while Perl doesn't come in any *lighter* versions (to my knowledge). NoSQL does have some Perl operators, but they are either leftovers of the original RDB system, that will eventually be rewritten in C or AWK, or they are utilities to be used once in a while, which appare at most once in a single pipeline (usually at the end).

Unless your code has been accepted for inclusion in the main NoSQL distribution, you may not use the official NoSQL source code banners and Copyright statements in your own operators.

Chapter 12

Frequently Asked Questions (FAQ)

- **Q1:** How do I modify existing tables ? **A1:** There are several ways:
  - Do it manually with the 'edittable' operator.
  - Simply append the new record(s) (table2) to the old table (table1):

    ```
    tail +3 < table2 >> table1
    ```

  - On large, indexed tables you can apply the changes to a smaller journalling table (edit buffer) and then use the 'update' operator. See section "Big tables" for more details on this method.
  - Use the old table header to build a valid new record, modify the fields of the latter and write the modified table back to disk:

    ```
    addrow < table |
      compute 'Column1="value1"; Column2="value2"; ...' > table.tmp
    mv table.tmp table
    ```

  - Use the 'row' operator to select only those records that you want to retain:

    ```
    row 'selection expression' < table > table.tmp
    mv table.tmp table
    ```

  - Use the 'update' operator. The two tables are assumed to have the respective primary key fields in their leftmost columns.
- **Q2:** Will NoSQL ever provide record-level locking and update facilities ? **A2:** This is currently 'not on the cards'. NoSQL assumes that the tables are collections of variable-length records. Record-level update requires fixed-length records. It can be implemented by right-padding the records with blanks to 'make room' for updates, but that is a bit of a kludge, IMHO. If you need such facilities you should rather use */rdb*, or resort to a more usual SQL DBMS, either free or commercial. I prefer the solution outlined in section "Big tables" .

Chapter 13

References

1. "Unix Review" magazine, March, 1991, page 24, "A 4GL Language".
2. "UNIX Relational Database Management", R. Manis, E. Schaffer, R. Jorgensen, 1988, Prentice Hall.
3. "The AWK Programming Language", Alfred V. Aho, Brian W. Kernighan and Peter J. Weinberger, 1988, Addison-Wesley.
4. "Programming PERL", L. Wall, R. Schwartz, 1991, O'Reilly & Associates.
5. "UNIX Shell Programming", L.J. Arthur, T. Burns, 1997, John Wiley & Sons.
6. "Consultix Shell Quoting Guidelines", *http://www.consultix-inc.com/quoting1_2.txt*