LINUX JOURNAL

Strictly On-Line: NoSQL Tutorial

Date: Monday, November 01, 1999
Topic: Miscellaneous

Giuseppe Paterno

A comprehensive look at the NoSQL database.

Some months ago I had a discussion with NoSQL creator, Carlo Strozzi, regarding the databases.

I should admit, I am an SQL fan! It's hot having the same language, no matter which platform or database engine is used. He underlined the fact that most SQL engines lack of flexibility and waste system resources (memory and disk space) because of their multi-platform environment (such as Oracle, DB2, Informix, etc.).

He suggested I have a look at the white paper that inspired him: ``The UNIX Shell As a Fourth Generation Language" by Evan Schaffer (evan@rsw.com) and Mike Wolf (wolf@hyperion.com).

Quoting from the above paper:

> ... almost all [database systems] are software prisons that you must get into and leave the power of UNIX behind. [...] 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.

The UNIX file structure is the fastest and most readily-available database engine ever built: directories may be viewed as catalogs and tables as plain ASCII files. Commands are common UNIX utilities, such as **grep**, **sed** and **awk**. Nothing should be reinvented.

NoSQL was born with these ideas in mind: getting the most from the UNIX system, using some commands that glue together various standard tools. Although NoSQL is a good database system, this is not a panacea for all you problems. If you have to deal with a 10 gigabytes table that must be updated each second from various clients, NoSQL doesn't work for you since it lacks of performance on very big tables, and on frequent updates you must be in real time. For this case, I suggest you use a stronger solution based on Oracle, DB2 or such packages on a Linux cluster, AS/400 or mainframes.

However, if you have a web site containing much information and more reading occurs than writing, you will be surprised how fast is it. NoSQL (pronounced noseequel, as the author suggests) derives most of its code from the RDB database developed at RAND Organization, but more commands have been built in order to accomplish more tasks.

Installing NoSQL

The latest NoSQL source code can be found at ftp://ftp.linux.it/pub/database/NoSQL, but RPM and Debian packages are also available. At the time of writing, latest stable version is 2.1.3.

Just unpack the source using the command `tar -xzvvf nosql-2.1.3.tar.gz` in a convenient directory (such as $HOME/src), and you will get all the code in the nosql-2.1.3 subdirectory. Enter the above subdirectory and do the following:

```
./configure
make
make install
```

The software will put its engine into /usr/local/lib/nosql, its documentation in /usr/local/doc/nosql and a symlink /usr/local/bin/nosql that points to the real executable (/usr/local/lib/nosql/sh/nosql). You can change the directory prefix (e.g., /usr instead of /usr/local) invoking `./configure --prefix=/usr`.

You should copy the sample configuration file to the $NSQLIB directory (i.e., /usr/local/lib/nosql). This is not required but it's useful for changing some parameters via configuration file instead of variables. The commands

```
cp nosql.conf.sample /usr/local/lib/nosql/nosql.conf
chmod 0664 /usr/local/lib/nosql/nosql.conf
```

will copy it with the right permissions. You can optionally have a personal NoSQL configuration file creating an $HOME/.nosql.conf with 0664 permission applied.

Although NoSQL installation is quite simple, I suggest you to read the INSTALL file: the author gives some good tips.

Getting Some Steps Around

Now that the package has been installed, let's start getting acquainted with NoSQL commands through an example.

We are the usual Acme Tools Inc. which supplies stuffs to the Toonies land. We would like to track our customers, so we should create a first table in which we will list some customers details (such as code, phone, fax, e-mail, etc...). The best way to create tables from scratch is via template files.

A template file contains the column names of the table and associated optional comments separated by tabs and/or spaces. Comments may be also specified with the usual hash sign (#) as the first character of the line. The file below, customer.tpl, is our template for the customer table.

```
# Acme Tools, Inc.
# Customers table
#######################
CODE    Code number
NAME    Name/Surname
PHONE   Phone no
EMAIL   E-mail
```

Most of the NoSQL commands read from STDIN and write to STDOUT. The **maketable** command, which builds tables from templates, is one of these. Issuing the command

```
nosql maketable < customer.tpl
```

we'll get the table header on STDOUT:

```
CODE    NAME    PHONE   EMAIL
----    ----    -----   -----
```

Great, but we should keep it in a file. We could simply redirect the command output to a file, e.g.,

```
nosql maketable < customer.tpl > customer.rdb
```

but this wouldn't be the right way. The **write** command may be helpful in this case, as it reads a table from STDIN (a simple header in this case) and writes a file, checking data integrity.

The resulting command would be

```
nosql maketable < customer.tpl |
  nosql write -s customer.rdb.
```

The `-s` switch in the write operator suppress STDOUT, e.g., `nosql write`, similar to the **tee** UNIX utility, writes both to file and STDOUT, unless `-s` is specified.

Pay attention, because the write command does not do any locking on the output table: `nosql lock table` and `nosql unlock table` must be used for this purpose.

First Editing: Using edit Command

Now let's add our first customer with the command

```
nosql edit customer.rdb
```

The default editor is the **vi** command, but you can use your favorite editor changing the EDITOR environment variable. The screen below is presented to the user:

```
CODE
NAME
PHONE
EMAIL
```

Just fill the fields with some information, remembering to separate values from field names with a tab. *Do not delete* first and last blank lines, this is not a bug: it's the way NoSQL handle lists. But I prefer to let you discover this little feature later in this article.

```
CODE    ACM001
NAME    Bugs Bunny
PHONE   1
EMAIL   bugs.bunny@looneys.com
```

Now that we have filled in the form, just write it (`ESC` then `:wq!`) and the command will check if the format is correct and write it to disk. Wow, we have a real table and real data!

NoSQL Table Format

Since we are curious, we will taking a look to the real file on disk.

```
CODE    NAME    PHONE   EMAIL

ACM001  Bugs Bunny      1
```

First of all, is important to note the fact that all columns are tab-separated: please keep this in mind when you want some external program to update the table, otherwise you will break the table integrity.

The first line is called the *headline* and contains column names; the second is the *dashline* and separates the headline from the body: both are named the *table header*. The rest is called the *table body* and contains the actual data.

A number of commands have been build to displays these parts, and they are simply calls to ordinary UNIX utilities:

- `nosql body`: displays the table body (same as: `tail +3 > table`)
- `nosql dashline`: displays the table dash line (same as: `sed -n 2p < table`)
- `nosql header`: displays the full table header (same as: `head -2 < table`)
- `nosql headline`: displays the table headline (same as: `head -1 < table`)
- `nosql see`: displays the `TAB` character as `^I` and `newline` as `$`, making much easier to see what's wrong on a broken table (same as: `cat -vte < table`)

Once again, this shows how powerful the UNIX OS is on its own, and how handy it can be for add-on packages such as NoSQL to tap into this power without having to re-invent the wheel.

Simple Data Insertion: Using Shell Variables

A fun way to fill the table is using the environment variables. You can export variables in any way, e.g., using UNCGI in a CGI environment or named as column names with the desirable values as follows:

```
export CODE="ACM002"
export NAME="Daffy Duck"
export PHONE="1-800-COOK-ME"
export EMAIL="dduck@looneys.com"
```

Then issue the command:

```
nosql lock customer.rdb; env | nosql shelltotable |
nosql column CODE NAME PHONE EMAIL |
nosql merge CODE NAME PHONE EMAIL customer.rdb |
nosql write -s customer.rdb; nosql unlock customer.rdb
```

and the work is done--a bit cryptic? Yes, but that's the power of NoSQL: all can be done in a single shell command. Let's explain it:

- `nosql lock customer.rdb`: this locks the table and ensures noone else can write in the table at the same time we do.
- `env`: prints the environment variable.
- `nosql shelltotable`: reads all variables from the pipe and writes a single record table containing all values to STDOUT.
- `nosql column CODE NAME PHONE EMAIL`: reads the NoSQL table containing the environment variables from the pipe and selects column CODE, NAME, PHONE and EMAIL in that order and

writes STDOUT.

- `nosql merge CODE NAME PHONE EMAIL customer.rdb`: reads the two merging tables, one from pipe (STDIN) and other from file, writing the merged table to stdout. The resulting table has two records: the existing one and the new one extracted from the above process.
- `nosql write -s customer.rdb`: reads the resulting table (merged from the above command) and writes it to disk as customer.rdb. We already explained what switch `-s` means.
- `nosql unlock customer.rdb`: unlocks the table.

Now have a look to the resulting table using the command that reads the table: `nosql cat customer.rdb`.

```
CODE    NAME              PHONE           EMAIL
------  ----------------  --------------- ----------------------
ACM001  Bugs Bunny        1-800-CATCH-ME  bugs.bunny@looneys.com
ACM002  Daffy Duck        1-800-COOK-ME   dduck@looneys.com
```

More Insertion: The Lists

NoSQL can handle data in different way, called list format. A sample table may be:

```
CODE    ACM003
NAME    Bart Simpson
PHONE   1-555-5432-321
EMAIL   bart@springfield.org

CODE    ACM004
NAME    Wiley The Coyote"
PHONE   1-800-ILLGETIT
EMAIL   wiley@looneys.com
```

Yes, you're right! This is the same way the edit command displays data. Although list tables aren't performing at all, in my opinion they are a good way to insert new data into tables. It's handy creating a program that can output this fashion. An example is shown in Listing 1.

Okay, this is not a ``state of the art'' shell program, but this example may show that the complete operation is easy with *every* language, even the shell.

There are a couple of things I would like to emphasize about the above code. How can a list be merged with a real table? The command **listtotable** does the job for you, as you probably guessed looking at the pipe, converting the list format to the table one. A reverse command, **tabletolist**, exists as well.

Please notice the beginning and ending newlines, as well as the tabs between field name and value, which appear in the print statement: these are required in order to create the correct list structure.

The list structure, as well as table structure, are well documented in the Chapter 2 of the NoSQL reference you will find in /usr/local/doc/nosql/doc.

Getting More from NoSQL

Now, let's have a business example? A complete catalog was created using a NoSQL table (see the catalog.rdb file below) and published on the Web dynamically. Every two days, we receive orders from our customers that they have usually created with Excel and exported, at our request, in a coma-separated file.

```
PRID    DESC                             PRICE
------  ------------------------       ------
PRD001  Acme glue for RoadRunners        30.00
PRD002  Acme TNT                        150.00
PRD003  Carrots                           5.00
PRD004  Acme toolbox                     75.00
```

The file (sample_order.txt below) we receive has the following format: *requester's unique code*, *Product ID*, *Quantity*.

```
ACM004,PRD001,5
ACM004,PRD002,30
ACM004,PRD004,1
```

Now from a shell or command line we run:

```
export TMPFILE=`mktemp -q /tmp/$0.XXXXXX` ; cat sample_order.txt |
perl -e 'print "CODE    PROD    QTY
"; print "----  ----    ---
";
while(
catalog.rdb |
nosql addcol SUBTOTAL | nosql compute 'SUBTOTAL = QTY*PRICE' > $TMPFILE ;
echo "Please bill to:" ; echo "---------------" ; echo ""; cat
$TMPFILE |
nosql join -j CODE - customer.rdb | nosql column NAME PHONE EMAIL |
nosql body | head -1 ; echo "";echo "" ; cat $TMPFILE | nosql rmcol CODE |
nosql print -w; echo ""; echo -n "Total due: "; cat $TMPFILE |
nosql subtotal -T SUBTOTAL | nosql body ; rm $TMPFILE
```

and our output is:

```
Please bill to:
---------------
Wiley The Coyote        1-800-ILLGETIT  wiley@looneys.com
PROD   QTY DESC                       PRICE SUBTOTAL
------ --- ------------------------- ------ --------
PRD001   5 Acme glue for RoadRunners  30.00      150
PRD002  30 Acme TNT                  150.00     4500
PRD004   1 Acme toolbox               75.00       75
Total due: 4725
```

This result may be sent via e-mail to our logistic people. Not so bad for a *five-minute single shell command*, is it?

I know it's a bit hermetic, so let's have a closer look: the explanation is divided in four sections to be easily read. I will keep out the **echo** commands which are quite obvious.

Section 1: Extracting Useful Data From the Received File

- `export TMPFILE=`mktemp -q /tmp/$0.XXXXXX``: exporting environment variable `TMPFILE` that contains a runtime generated tempfile.
- `cat sample_order.txt`: get as input the file we received.

- `perl -e 'print "CODE PROD QTY "; print "----- ---- --- "; while(<STDIN>) { s/,/ /g; print };``: prints a NoSQL compliant header, composed of CODE, PROD and QTY. Then, it prints the table received from STDIN, substituting a coma (,) with a tab ( ), in order to get the correct table structure.
- `nosql join -j PROD - catalog.rdb`: joins the table read from STDIN (the `-` character) and catalog.rdb using the PROD (product ID) column.
- `nosql addcol SUBTOTAL`: now a column SUBTOTAL is added to the resulting table.
- `nosql compute 'SUBTOTAL = QTY*PRICE' > $TMPFILE`: calculates the SUBTOTAL column by multiplying quantity and price columns. It then redirects the resulting table to the previous calculated temporary file.

Section 2: Getting and Printing Billing Name

`cat $TMPFILE`: reads the table written from previously stored temp file.

- `nosql join -j CODE - customer.rdb`: joins the STDIN table (minus sign) and the customer.rdb table on the CODE column.
- `nosql column NAME PHONE EMAIL`: selecting NAME, PHONE and EMAIL columns.
- `nosql body`: gets only table content, without printing the header lines.
- `head -1`: prints only one line; all the lines are identical since who sent the file (or order) is the same customer.

Section 3: Printing the Order Content

- `cat $TMPFILE`: reads the table written from previously stored temp file. "`nosql rmcol CODE`: removes CODE column from the STDIN table (it's not useful for us at this moment).
- `nosql print -w`: prints the results in a simple but useful form. Columns containing only numbers are right-justified with blanks, while anything else is left-justified. The `-w` switch forces the **print** command to fit in a terminal window.

Section 4: Getting Total Amount Due

- `cat $TMPFILE`: reads the table written from previously stored temp file.
- `nosql subtotal -T SUBTOTAL`: calculates the sum of the SUBTOTAL column. The result of this command is a NoSQL compliant table.
- `nosql body`: gets only table content, without printing the header lines.
- `rm $TMPFILE`: removes the temporary file created before. The commands used at this moment have, of course, more options. For a complete overview, please check out the documentation included with the NoSQL package.

Indexing the Tables

The best way to search information on a large amount of data is, of course, indices, and NoSQL obviously has its own operators to interact with them.

Suppose our customers are increasing greatly since Acme Inc. started, so we need to speed up our searches. Most of the time, we look up on the CODE column, so it would be better to build the index on it last.

The command `nosql index customer.rdb CODE` will create an index on customer.rdb table against the CODE column. Index files are named by appending an x and the column name(s) (separated by a dot) to the base name of the table it refers to. In our case, the index file name is customer.x.CODE. If we want to update the index without rebuilding it, we should run

```
nosql index -update customer.x.CODE
```

A crontab job on our system will ensure a periodic index update, by running the command:

```
cd /var/tables/acmeinc; nosql index -update
```

Not specifying any index file name on `nosql index -update` will update all indices in the current working directory.

Now that indices are built, let's try a search. In order to search against indices, you should create a small key table as input to the `nosql search` command, for example:

```
echo -e "CODE
----
ACM004" | nosql search -ind customer.x.CODE
```

This command will extract the `ACM004` value in the CODE column using the customer.x.CODE index. Creating this key table may not seem handy, but this ensures the maximum reuse of commands. Suppose you extracted some data from table1, you can search this last result on table2 easily with a single pipe. Try to think a bit about it, this is not a bad idea at all.

Your Lifebelt: The RCS

What if you discovered that last write caused a disaster and you do not have a backup? You will never have the latest copy of the table before the disaster happened.

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. The command `nosql edit`, for example, will automatically check out a table for editing, and then check the new version back into RCS. Other commands can utilize tables that are under RCS control by using explicit commands like:

```
co  -p  table | nosql row  ... | nosql column  ... | nosql print
```

or relying upon the **cat** command to handle interactions with RCS automatically:

```
nosql cat table | nosql row  ... | nosql column  ... | nosql print
```

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

Now that you have RCS keeping watch for you, if the disaster happens in real life, you can easily **rollback**: the command

```
nosql rollback [-d datestring] tablename
```

will extract the table updated at the desired time.

Security

NoSQL works with UNIX, not in addition to it. Set correct permissions to files using groups: create a group for those users who can access these files. This is a great security wall.

Performance Tips

No matter what program you have to deal with, sooner or later you will have to deal with performance. I'm not a developer of the NoSQL Database System, but I can give you some useful advice.

First of all, keep your tables small. Don't keep all the data in a single table, this is a waste of performance. Using the method that best suits your environment, try splitting tables into several files and organizing them into directories. In our examples, we can keep track of our customers creating a single "phone directory" (i.e., customer.rdb), then creating a directory for each order status (received, waiting_for_bill, archive), and last a table for each customer (the file name will be the customer code). For example:

```
/var/tables/acmeinc/
customer.rdb
                    catalog.rdb
                    received/
                             ACM001.rdb
                             ACM003.rdb
                    w4bill/
ACM002.rdb
                             ACM003.rdb
                    archive/
                             ACM001.rdb
                             ACM002.rdb
                             ACM003.rdb
```

If you must do everything in a big table and you have to update it frequently, there's a trick for you if it's an indexed file: journaling.

Create a journaling table, say customer.j, with exactly the same header as customer.rdb, but containing only those records which we want to insert into, remove from or append to customer.rdb. The entries in customer.j must be in a format suitable for the `nosql merge` command.

Whenever we fetch data from customer.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 be merged into customer.j, and the final output will undergo the original query statement again.

Any updates to customer.rdb will be done to customer.j with the syntax described in the documentation of the **merge** command (there you'll find how to use the operator to the optimum level). You will also have to make sure that customer.j is kept sorted on its primary key field after each update. For example, if you have an index on the CODE column in the customer.rdb table, you should use:

```
echo -e "CODE
----
ACM004" | nosql search -ind customer.x.CODE |
nosql merge CODE NAME PHONE EMAIL customer.j |
nosql row 'CODE=="ACM004"'
```

As you can see, the trick is:

1. Perform an indexed search on customer.rdb to quickly obtain a much smaller (possibly empty) subset of the table.
2. Merge the first output with customer.j on the fly during the query.
3. Do a sequential post-query on the final output.

If that's not enough, another trick to improve speed is to make your AWK use the **ash** (by Kenneth Almquis) for **system()**'s and pipes, since this shell is small and fast at startup. Suppose your AWK is /usr/bin/mawk and your shell is /bin/ash (a fairly common case, especially on Debian GNU/Linux), then you can do something like this:

1. Create the following hard link: `ln /bin/ash /bin/ah`
2. Modify AWK to make it use /bin/ah as opposed to /bin/sh, and write the modified AWK to /usr/local/bin/nsq-mawk: sed 's//bin/sh//bin/ah/g' /usr/bin/mawk
   > /usr/local/bin/nsq-mawk
   chmod 755 /usr/local/bin/nsq-mawk
3. Modify the NSQAWK value in the config file (/usr/local/lib/nosql/nosql.conf or $HOME/.nosql.conf) to /usr/local/bin/nsq-mawk. This will speed up your queries !

Going to the Web

Using NoSQL on the Web is a matter of seconds. Let's suppose that Acme Tools Inc. now has a web site, and you want your customer to search on database for their pending orders. We first create a small input form, the file getname show in Listing 2, on which we ask for customer name. I did not use any security at all in this example, but at least password should be asked in a production environment.

Since we are not good at graphics, we create a small template (result.html) that can be modified easily. This template, result.html, is shown in Listing 3.

In this template, some keywords are substituted by the CGI: our standard keywords start and end with a double hash (#) sign, for example ##KEYWORD##.

A special section of the template, named stanza, will be repeated as many times as the number of query rows. The stanza starts and ends with special comments that will be recognized by the CGI.

Now is the time for writing the CGI: Listing 4, result.pl, is a perl script that should perform the queries based on the input name and then creates a resulting page based on the previous template. Most of the queries used in this CGI are those used in previous examples, so we won't repeat them. Just have a look to the main query:

```
@cusdata = `nosql cat $datafile | nosql join -j PROD - $ctlgfile |
nosql addcol SUBTOTAL | nosql compute 'SUBTOTAL = QTY*PRICE' | nosql
column PROD DESC QTY PRICE SUBTOTAL | nosql body`;
```

The thing to notice is the array that will contain query rows: each row contains a tab separated list of fields, as the NoSQL table row specification. In fact, in the stanza keyword substitution, a **split** function against tab is used:

```
my ($prod, $desc, $qty, $price, $subtotal) = split(/	/, $data);
```

All queries run with back ticks that, instead of the **exec()** and **system()**, return the STDOUT of the program. This output may be reused in the program using variables. A negative fact of this is security: you cannot run this program in tainted mode (`-T` switch in the perl command line), but this can be avoided with some tricks such as the ones I used. First of all, you should avoid buffer overruns by using a substring function

`($cusname = substr($cusname, 0, 50))`, then keeping out some escape characters (such as >
Once the queries have been executed and we have all the necessary values, we load the template file and associate it with the default input and pattern-searching space. The template file, in which we transformed new lines and multiple spaces into a single space, is now divided into three parts using pattern matching: the header, the body (aka stanza) and the footer.

```
/(.+)<s*!--s*heres+startss+nosqls+stanzas*--->(.+)<s*!--s*here
s+endss+nosqls+stanzas*--s*>(.+)/i ;
```

This search will identify the stanza into the template, using the keyword `<!-- here starts nosql stanza -->` as the beginning and `<--! here ends nosql stanza -->` as the end. As you can notice, these are simple HTML comments, so can be introduced easily by our graphic experts. All items before the beginning comment is considered the header, while the rest is the footer.

Before entering the keyword processing in the body, we will do it in the header and footer in order to set proper customer name and total amount due:

```
$header =~ s/##NAME##/$cusname/;
$footer =~ s/##TOTAL##/$total/;
```

The final part is the keyword substitution in the stanza. Here we will swap the original variable (*$body*) with a temporary one (*$tmpbody*), in order to leave the first unchanged for next loop. Here the fields are split using the method I described earlier, then substituted for the keywords in the template file. Of course, there are thousands of way of writing down this kind of CGI, be it in Perl or other languages. Write one in your favorite language and let your imagination be your guide: databases are plain ASCII files, so you can process them as you like, and you will get great results.

For a REAL example of NoSQL usage on the web, check out http://www.whoswho-sutter.com/, http://annunci-auto.repubblica.it/ and http://www.secondamano.it/ (the first is in English, while the others are in Italian).

NoSQL in the Near Future

Talking to Mr. Strozzi, the main developer of NoSQL, he revealed to me some news on ongoing development of the RDBMS.

The first minor modification you will see at a glance is the version number: kernel release schema has been introduced, so even numbers are the stable ones, while odd ones are unstable (current unstable is 2.3.1).

The major modification is the **rel** command. It checks table reference integrity before an update/insert/delete, but won't take any action: it only advises you if something will be broken , so you should use it in your program before you do any table operation.

Other minors enhancements are some commands such as **insert**, **delete**, **tabletoperl**, **perlencode** and **tabletom4** that are quite useful in a programming environment as well as on the command line. At present, no official reference for those commands, but comments in the source code will easily let you understand how to use them.

Mr. Carlo Strozzi told me that the next stable release, 2.4.0, will be available around November of 1999.

Conclusion

NoSQL is a great database system for web-based applications, in which reading occurs much more than writing. I recommend it also for full-text searching and in those applications where ASCII tables may be handy.

For more information have a look at the official web page http://www.mi.linux.it/People/carlos/nosql/ or subscribe to the mailing list by sending a message to noseequel@mi.linux.it with the words `subscribe noseequel` in the `Subject:` line of the message.

I would like to thank the NoSQL creator, Carlo Strozzi, for being supportive of me in writing this article; Maurizio Sartori, who gave me some hints; Giovanni Granata, Andrea Bortolan and all the people who have encouraged me to go on researching.

Giuseppe Paternó has recently changed jobs from System Engineer for IBM Italy, where he worked on firewalls, Lotus Domino, AS/400 and mail systems, to Internet System Administrator for Infostrada, a local Telco/ISP. He likes music, cooking, science, and of course, beer. He can be reached at gpaterno@spacelab.gpaterno.com.

All listings referred to in this article are available by anonymous download in the file ftp://ftp.ssc.com/pub/lj/listings/issue67/3294.tgz.