OpenVMS Technical Journal V8

Bringing Vegan Recipes to the Web with OpenVMS

Bernd Ulmann (ulmann@vaxman.de)

Introduction

In 2001 my girl friend, Rikka, who was studying computer science, had a lab concerning databases. Being a vegan, she decided to develop a demonstration database for vegan recipes with a simple web front end. Since she loves to cook the database grew from a couple of recipes to several dozen very quickly. It turned out that there were people in Germany quite interested in vegan cooking so we decided to make the database publicly available via the Internet.

This was the time when area-wide DSL infrastructure became available in Germany and we decided to use her LINUX-based PC as a small MySQL and Apache server using some CGI scripts written in Perl. After a couple of years of growing interest it became clear that this interim solution was not adequate any longer. The PC encountered minor hardware problems and LINUX just isn't for someone like me running a VAX-7820 at home. So we decided to migrate the recipe database -- which has grown to the largest German database of vegan recipes and sees thousands of requests per day -- from the small LINUX system to FAFNER, the VAX.

We need to talk a bit about the details of the database model and the CGI scripts to understand the decisions described in the following, so these points will be outlined below.

Structure of the database and CGI scripts

The database model of the recipe database features twelve tables, some of which are described in the following:

Recipes	The recipe itself. This table contains the name and a descriptive text for each recipe as well as some foreign keys to other tables.
Units	All necessary units for measuring ingredients.
Ingredients	All ingredients used to create a meal.
recipe_ingredients	Since recipes and ingredients form an n-times-m relationship, this table

	is necessary to link a particular recipe with all of its ingredients.
categories	Categories for the recipes like pasta, salad, and so on.
recipe_categories	n-times-m link between categories and recipes.

Access to the recipe database is performed by three CGI scripts written in Perl. The first of these, RZ1.PL, creates an HTML form containing several selection lists: one for the categories, one for the ingredients, and so on. These lists can be used to narrow the search for a particular recipe. If you are looking for recipes using “chanterelle” you could select "Chanterelle" in the ingredients selection list and then click the search button on the bottom of the form created by RZ1.PL.

This will trigger the next stage of recipe selection, namely the Perl script RZ2.PL. This CGI script now reads all selection parameters and creates a proper SQL statement to select all recipes from the database satisfying the conditions selected in the previous step.

Using this list of available recipes, RZ2.PL creates another HTML form with only one selection list containing all titles of the matching recipes. In the current example this list will contain seven entries, one of them being "Mushroom Carpaccio." Selecting a recipe from this list and clicking the search button at the bottom of this second form will trigger RZ3.PL, the last CGI script.

RZ3.PL now reads the title of the recipe selected before and starts gathering all necessary information to display the recipe, a picture of the resulting meal, and so forth. This is by far the most complicated step since all twelve tables are involved in this operation.

Bringing all this to OpenVMS

The first step in bringing all this to OpenVMS was to decide which database to use. Since there is a port of MySQL for OpenVMS Alpha, it seemed like a good idea to try porting it to OpenVMS VAX. Unfortunately this turned out to be very complicated. The first obstacle -- compiler qualifiers not applicable for a VAX -- was resolved easily using some small Perl scripts to automatically modify the build procedures.

Later it turned out that porting MySQL to OpenVMS VAX was not easy at all. In fact, I had to give up after some time. One of the problems was that I was unable to change all occurrences of 64-bit integers (long long) by setting appropriate #define clauses. Some floating-point routines proved to be quite tricky, too, so I eventually decided to abandon the idea of running MySQL on a VAX. For a short moment I thought about replacing FAFNER with a more modern Alpha-based system like an AlphaServer 4100 but I couldn’t do it – living without a large VAX in the basement seemed horrible to me.

Having ruled out MySQL, I took a look at other systems like Postgres and the like. After digging into many lines of source and thinking about databases over and over again it became clear that using RDB was the way to go. No more tinkering with products not being designed for OpenVMS and no more concessions.

How to migrate from MySQL to RDB

After installing RDB version 7.0 on FAFNER it was time to set up an empty database to hold all of the recipe data. This was done with a DCL procedure as shown below (ellipses denote omitted code):

```
$ sql
create database alias recipes filename disk$rdb_data:[000000]recipes
number of buffers is 1000
number of users is 100
row cache is enabled
!
grant all on database alias recipes to [ulmann];
grant select,update on database alias recipes to [http$nobody];
!
set dialect 'sql92';
...
create table recipes.recipes (
        rnr int not null,
        name varchar (80) not null,
        enr int not null,
        anr int not null,
        regnr int not null,
        description long varchar,
        pnr int,
        photo varchar (80),
        source int,
        counter int,
        story long varchar,
        monthcounter int,
        tim char(14),
        instim char(14),
        cookbook int,
        primary key (rnr));
...
commit;
...
create unique index recipes.irecipes on recipes.recipes (rnr);
...
commit;
...
grant all on table recipes.recipes to [ulmann];
grant select,update on table recipes.recipes to [http$nobody];
...
commit;
exit
$exit
```

Having done this resulted in an empty database with all necessary tables (the fact that the user HTTP$NOBODY needs update privilege for the recipe table stems from the fact that two counters will be incremented to keep track of recipe requests). The next problem was to move all of the data from the MySQL database running on LINUX to the new RDB database running on VMS.

The first idea was to create one text file for each table on the MySQL system and extract the table data using standard SQL. Using Perl for necessary format changes and standard RDB methods it should be possible to load the raw data from these files into the corresponding RDB tables.

After a quick experiment it turned out to be a clumsy and inelegant solution so it was abandoned. Then the idea of using a Perl script to connect to the MySQL system running on LINUX and the RDB system running on OpenVMS emerged. This script, MYSQL2RDB.PL, makes use of three Perl modules -- Net::MySQL, DBI, and finally DBD::RDB.

Net::MySQL, a pure Perl implementation of the MySQL interface, was chosen to connect to the MySQL system due to the fact that this script did not need low overhead since it was to be used only once. This seemed to justify not compiling a native MySQL interface on the VAX.

After connecting to MySQL and RDB, the script copies one table after the other using an array containing the names of all tables to be copied. The main loop looks like this:

```
for my $table (@tables)
```

```
{
  print "Deleting from $table...\n";
  $rdb -> do ("delete from recipes.$table");

  $mysql -> query ("select * from $table");
  my $record_set = $mysql -> create_record_iterator;
  my @fields = $record_set -> get_field_names;

  my $statement = "insert into recipes.$table (" .
                join (',', @fields) . ') values (' .
                join (',', map {'?'} 0..$#fields) . ')';

  my $rdb_sth = $rdb -> prepare ($statement);
  my $counter = 0;

  print "Inserting into recipes.$table: ";
  while (my $record = $record_set -> each)
  {
    $rdb_sth -> execute (@$record);
    $rdb -> commit unless $counter++ % 50;
  }
  $rdb -> commit if $counter % 50;
  print "\nInserted $counter lines into recipes.$table\n";
}
```

First all data is read from the MySQL source table using the statement select * from $table. Applying the method create_record_iterator results in an iterator which can be used to loop over the result set of the select operation.

To be able to insert the data just read into the RDB destination table a proper insert statement has to be created. First, a list of all column names will be needed which can be generated by applying the get_field_names method. The statement resulting from the line my $statement = will look like this:

```
insert into recipes.<table_name>
(<column_name>, <column_name>, ...)
values
(?, ?, ...)
```

After preparing this statement the data is copied in a loop performing an execute call for each row of raw data.

In the first version of this migration script the connection to RDB was made with default values implying autocommit. This made the script run incredibly slow since some tables contain tens of thousands of rows of data. Turning autocommit off and performing an explicit commit every 50 rows increased the performance by nearly a factor of 100.

Another lesson learned was that writing a db-trace file (which was enabled during debugging) takes more time than writing to the database, thus slowing things down considerably.

After modifying these procedures, the script copied the whole database flawlessly from MySQL to RDB in less then two minutes. The next step concerned the CGI scripts.

Selecting a web server and tuning Perl-based CGI scripts

Selecting a web server for this application was simple: WASD. There is just nothing that can compare to WASD when it comes to web servers for OpenVMS. It is true that, as a VAX, FAFNER can't use the Apache-based CSWS. But, the fact is, WASD has far less overhead than CSWS.

With the web server set, the next thing to do was configure WASD in a way that allowed the use of Perl-based CGI scripts. This turned out to be a simple task after having a look at the documentation.

After adapting the three Perl programs RZ1.PL, RZ2.PL and RZ3.PL in a way that allowed the use of RDB, the time for a first test had come. It worked out of the box! But it was much slower than the original system running under LINUX.

To be honest, this was exactly what I expected. The LINUX system was running on a recent PC with a several-GHz-CPU and lots of memory while FAFNER, a VAX-7820, is quite slow in comparison. It was clear that some tuning would be necessary.

The first thing to do was eliminate the overhead resulting from starting the rather large Perl interpreter every time a Perl-based CGI script was to be run. Fortunately, WASD offers everything necessary for this -- namely CGIplus and RTE, the Run-Time Environment. Using these techniques it is possible to have the Perl interpreter start only once (more or less) and the scripts parsed only once, too.

After installing and compiling the necessary software for this, the response time of the system skyrocketed. The invocation of CGI scripts became three to five times faster -- as fast as on the LINUX system!

How to tune RDB using Perl

The next problem to tackle involved accessing the RDB database system on FAFNER. It turned out to be a lot slower than using MySQL on the LINUX system. It was clear that something had to be done. It quickly became obvious that a crucial index was missing on the RDB database. Creating this index made things a lot faster but still not as fast as on the LINUX system. I decided to ask my friend, Thomas Kratz (a Perl guru), if he thought that a caching database proxy written in Perl might be a solution.

The idea behind this is as follows: A dedicated server process, the proxy will connect to the RDB database and listen to a particular port for connections from client processes. If some process wants to perform a database operation it will wrap up its request in a hash data structure and send this to the proxy. The proxy will unwrap the request and have a look into its local cache. If no appropriate data has been cached previously it will route the request to the RDB system, cache the resulting data set and send it back to the client program. If the data was found in the proxy's cache the RDB request will be omitted and the cached data sent immediately.

After only two evenings of programming, Thomas came up with a small yet powerful Perl program which did everything necessary to serve as an RDB proxy. It expects a request in the form of a hash like this:

```
{
  SELECT => 'SELECT DESCRIPTION FROM RECIPES',
  ALIAS  => 'RECIPES_PRODUCTION',
}
```

The first key may be either SELECT or DO distinguishing between requests that do not alter data and those that do. (The latter will normally result in clearing all affected cache entries. There is a way to circumvent this in case it is clear that only minor changes like incrementing a counter in the database will be performed. But this leads into too much detail.) The second key, ALIAS, contains the name of the RDB database on which the statement is to be performed, so a single proxy server can be used to process requests from the production system as well as requests from a test system.

The problem of sending a complete Perl data structure through a socket interface was solved like this: The calling program, the client (RZ1.PL, etc.), first opens a socket to write to:

```
my $sock = IO::Socket::INET->new
```

```
(
  PeerAddr  => $host,
  PeerPort  => $port,
  ReuseAddr => 1,
) or return("Can't connect to $host:$port: $!\n");
```

Then it sets up the data structure to be sent using $statement and writes it to the socket as shown below:

```
my $stmt_type = $stmt =~ /^\s*select/i
              ? 'SELECT'
              : 'DO'
              ;
print $sock encode_base64(nfreeze(
  {
   $stmt_type => $stmt,
   ALIAS      => 'RECIPES_PRODUCTION',
  }
), '', ), "\n";
```

The scalar $stmt_type contains SELECT or DO depending on the type of the SQL statement to be performed. This will be used as a part of the structure

```
{
  $stmt_type => $stmt,
  ALIAS      => 'RECIPES_PRODUCTION',
}
```

which is packed using the nfreeze function of the module Storable and base64 encoded before being written to the socket.

The proxy server then reads $raw from the socket and restores the data structure using a statement like this:

```
my $request_ref = eval {thaw(decode_base64($raw))};
```

This will result in a reference to a hash having the exactly same structure as the hash sent by the client. This hash will then be parsed by the proxy server to extract all necessary information to perform the action being requested.

Sending the resulting data set from the server to the client works quite the same but with a tiny difference: While the request from the client will always be short enough to be written to the socket in a single step, the resulting data set may be several hundred kB in size, exceeding the maximum amount of data which can be written to a socket in one operation. This requires the server to send the results in chunks using a sequence like this:

```
print $sock $_, "\n"
  for unpack("A$chunk_size" x (int(length($response) / $chunk_size) + 1),
    $response);
```

This will split the contents of the scalar $response into several smaller chunks of data which will be sent to the client through the socket $sock. The client then concatenates these chunks and works with the data thus returned.

Advantage of using a proxy server

Using this proxy server results in a dramatic reduction in response time as the following example shows:

The first request from RZ1.PL to build the category selection list takes 3.52 seconds using the proxy since the cache is empty and the request has to be actually performed on the RDB system. All following requests for building this list require about 0.34 seconds – a factor of ten faster than the direct access method!

Conclusion -- or sheer CPU speed is not everything

Looking back I have to admit that I underestimated the efforts necessary to port this application from a LINUX system to a VAX running OpenVMS. If the destination was a more modern system like an Alpha or an Itanium things would have been much easier since it would have been possible to use MySQL. This, and the sheer computational and IO speed of such a system, would have been more than sufficient to get faster response times than the LINUX system was able to offer.

Nevertheless I am glad that I decided to stay with my VAX. I learned a lot of things I would have missed otherwise and using all of the techniques sketched above it was possible to get performance from a VAX that was equal to if not better than that delivered by the original LINUX system.

Choosing RDB as the heart of this strictly non-commercial application was a good decision. It is quite easy to use and integrates perfectly with OpenVMS running on the VAX. Backing up databases and restoring them is a matter of seconds and performance is quite impressive for a VAX.

There was never any doubt that Perl is the ideal language for nearly everything -- not just CGI scripts as could be seen above. The minor drawback of being an interpreted language is more than compensated for by the expressive power of Perl as well as the possibility of using a variety of programming styles – functional, object-oriented, and purely imperative. No other language would have allowed writing the CGI scripts and the proxy server in such a short amount of time.

Of course, all of the techniques described above can also be used on Alpha and Itanium systems with equal reductions in response time. And the MySQL/RDB-conversion script may be applied in other areas as well.

The overall system has been running flawlessly for several months now with an increasing load of recipe requests. The VAX is rock solid and is down only when we encounter a power failure -- about twice a year. The LINUX system has been shut down and the last non-OpenVMS system has vanished from the house.

For more information

You may contact author at ulmann@vaxman.de, to get to the latest issue of the OpenVMS Technical Journal, go to: http://www.hp.com/go/openvms/journal.