Bringing Vegan Recipes to the Web

Bernd Ulmann

07-MAY-2006

Introduction

In 2001 my girl friend Rikka who studied computer science back then, had a lab covering databases.

She decided to develop a demonstration database for vegan recipes with a simple web front end using MySQL and Apache on a LINUX system.

Since she loves to cook the database grew larger and larger and during the following months and years the system became Germany's largest online collection of vegan recipes.

Obviously staying with LINUX, MySQL and Apache is not a real option when there is a large VAX-7000/820 running in the cellar. :-)

So it was necessary to bring the complete database system consisting of the database itself and a web front end from LINUX to OpenVMS which turned out to be not that easy.

The database

The database for these recipes is very well normalized resulting in twelve tables containing data like recipes, categories, ingredients, units, links between recipes and ingredients, etc.

This rules out an easy reimplementation without a relational database system, so the first task was to select a database system for OpenVMS/VAX – using an Alpha was out of the question, it had to run on the VAX.

Since there is a MySQL port for OpenVMS I first tried to port this from Alpha to VAX which turned out to be not too simple:

MySQL on a VAX?

- Nearly all of the build scripts use compiler qualifiers which are not supported on a VAX. This problem could be resolved with some small Perl scripts which automatically modified the build scripts.
- Compile times even on a VAX-7000/820 are quite slow, so finding bugs becomes a nerve wracking task.
- There are several places in the source code using 64 bit integers which were not easily converted back to 32 bit integers.
- Floating point handling turned out to be problematic, too. . .

After struggling for some days I gave up and decided not to use MySQL. Some experiences of damaged databases on Alpha systems running MySQL made this decision even easier.

Which database to use?

There is not too much choice when one is looking for a database system running under OpenVMS on a VAX.

I had a short look at Postgres, then at Ingres, then at RDB and. . .

. . . decided to stay with RDB. Why fiddle with a database which has not been developed with OpenVMS in mind when there is something like RDB available?

So it was clear that using RDB was the way to go. The first thing to do was to create a database containing all necessary tables for the recipe database.

Setting up the database

```
$ 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
```

How to get data into this database?

After creating all necessary tables and indices it was time to think about filling these tables with data. . .

But how? The production database was still running on MySQL on the LINUX system.

The first attempt was to extract the data from the MySQL system using some simple SQL statements and then loading it into the RDB system after some postprocessing using a Perl script.

This turned out to be not that simple! Many VARCHAR fields contained text with apostrophes, with double quotes and other things causing quite a lot of trouble.

A better solution was needed: Using Perl it would be possible to write a program running on the VAX which could connect to MySQL and RDB simultaneously and copy the data from one sysstem to the other by directly reading and writing the databases.

Connecting to the databases

```
use strict;
use warnings;

no warnings qw /uninitialized/;

use Net::MySQL;
use DBI;
use DBD::RDB;

my $rdb = DBI -> connect (
  'dbi:RDB: ATTACH ALIAS RECIPES FILENAME DISK$RDB_DATA:[000000]RECIPES',
  undef, undef,
  { RaiseError => 1, PrintError => 1, AutoCommit => 0, ChopBlanks => 1 }
 );

my @tables = qw/art eigenschaften einheiten glutenfrei kategorien personen
                region rezept_kategorien rezept_zutaten recipes zustand
                zutaten/;

my $mysql = Net::MySQL -> new (hostname => 'klapauzius.pi-research.de',
                               database => 'recipes', user => 'rikka',
                               password => ':-)');
```

Now there are two handles `$rdb` and `$mysql` for the two databases.

Copying the data

Having two database handles it is now easy to copy the tables as soon as you know which columns to copy:

```
my $record_set = $mysql -> create_record_iterator;
my @fields = $record_set -> get_field_names;
```

The next step requires the creating of an appropriate insert statements of the form

```
insert into <table> (<column>, ...) values (<value>, ...)
```

This can be done in the following way:

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

Putting everything together

```
for my $table (@tables)
{
  $rdb -> do ("delete from recipes.$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;

  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";
}

$rdb -> disconnect;
```

Lessons learned

The program fragments shown above havw undergone some incarnations during which several things were learned:

1. It is not a good idea to write a database trace file (specified at the connection stage). Writing a trace file will write more data to the trace than to the database which results in about 0.2 seconds per row being inserted!

2. It is not a good idea using autocommit, either. It is far better to perform an explicit commit every 50 or 100 rows.

Using this script the entire database can be copied from MySQL to RDB in just one or two minutes!

What's next?

After porting the database from MySQL to RDB it was time to have a look at the Perl based CGI scripts which form the user interface to the recipe database.

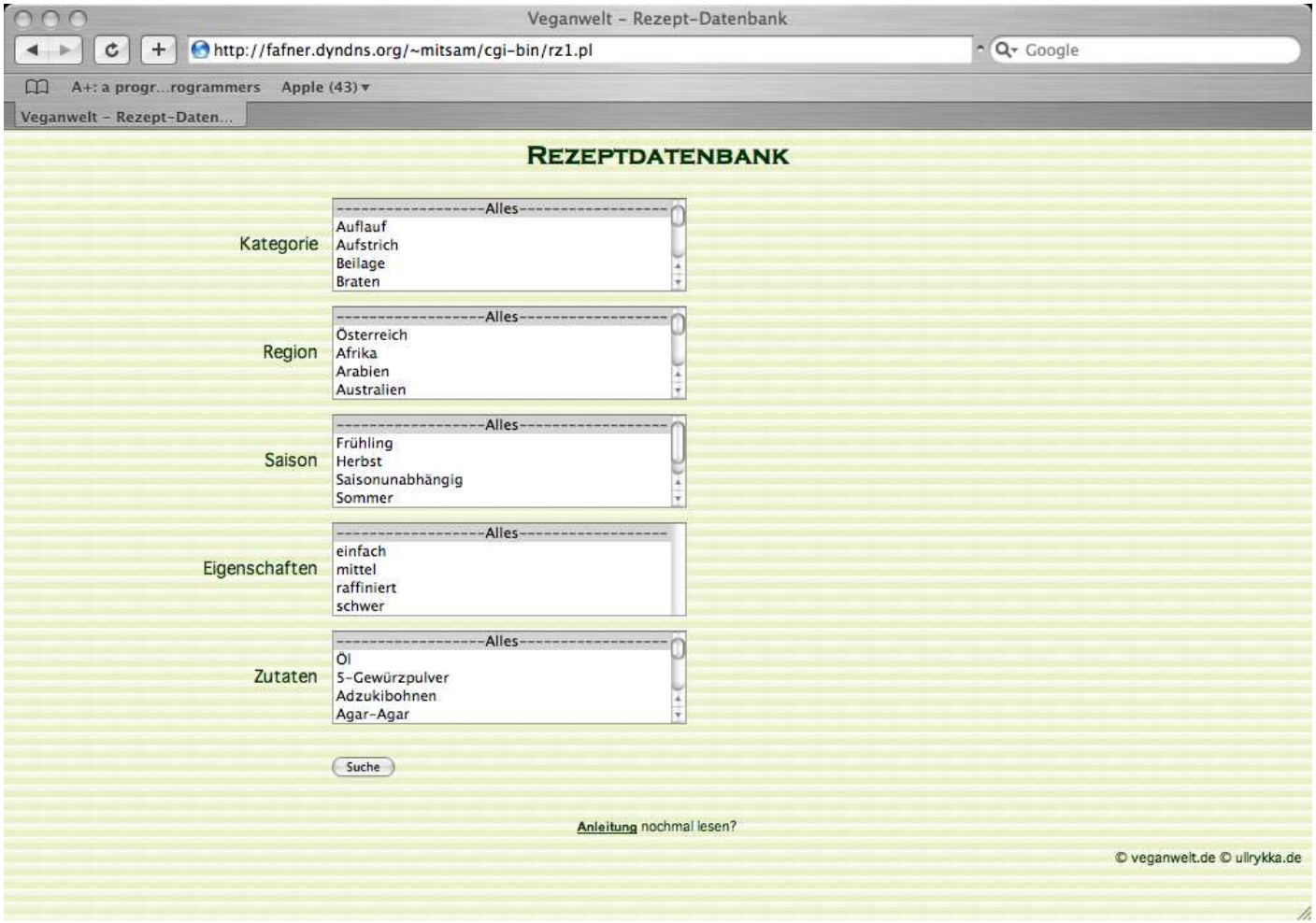
This user interface consists of three parts:

RZ1.PL: Displays a web form with several selection lists to allow the selection of recipes based on some preconditions like special ingredients, special purposes, etc. This script calls

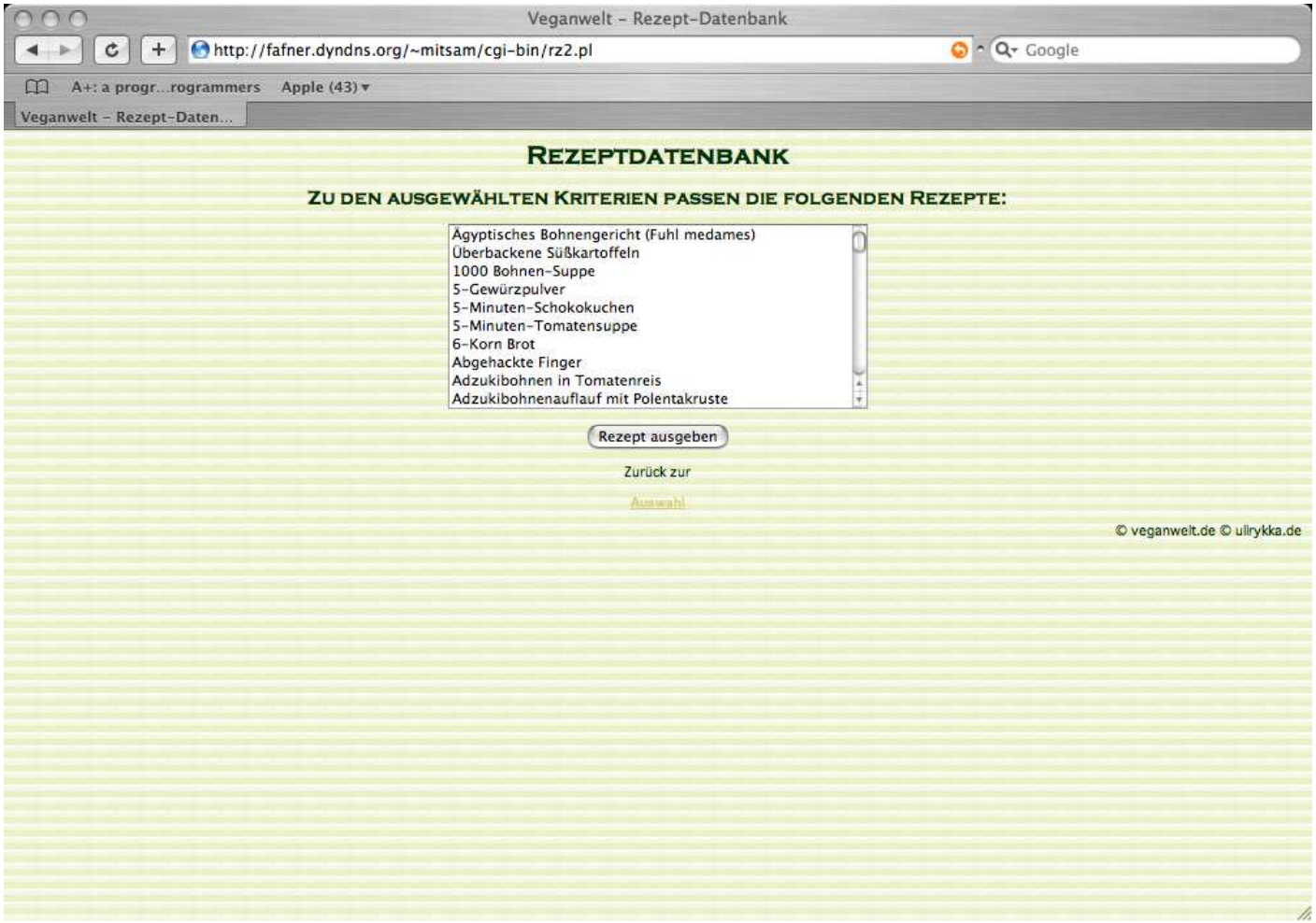
RZ2.PL: This will display a single selection list containing the names of all recipes satisfying the criteria selected in the previous step. Selecting one of these will call

RZ3.PL: This script will finally display the selected recipe in all its beauty.

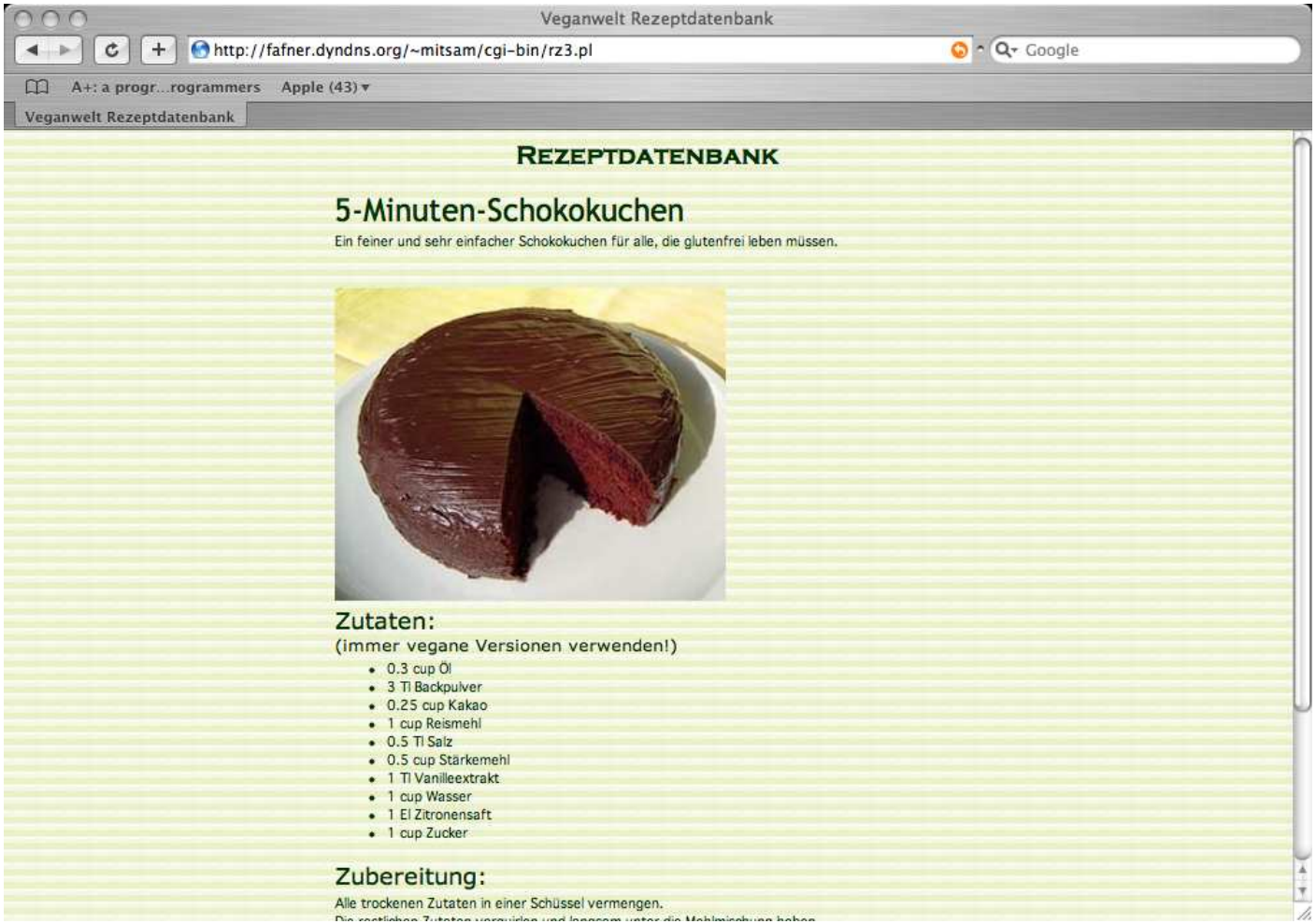
RZ1.PL


Veganwelt – Rezept-Datenbank
http://fafner.dyndns.org/~mitsam/cgi-bin/rz1.pl
Google
A+: a progr...rogrammers
Apple (43)
Veganwelt – Rezept-Daten...
REZEPTDATENBANK
Kategorie
Region
Saison
Eigenschaften
Zutaten
----------Alles----------
Auflauf
Aufstrich
Beilage
Braten
Österreich
Afrika
Arabien
Australien
Frühling
Herbst
Saisonunabhängig
Sommer
einfach
mittel
raffiniert
schwer
Öl
5-Gewürzpulver
Adzukibohnen
Agar-Agar
Suche
Anleitung nochmal lesen?
© veganwelt.de © ullrykka.de

RZ2.PL


Veganwelt – Rezept-Datenbank
http://fafner.dyndns.org/~mitsam/cgi-bin/rz2.pl
Google
A+: a progr...rogrammers
Apple (43)
Veganwelt – Rezept-Daten...
REZEPTDATENBANK
ZU DEN AUSGEWÄHLTEN KRITERIEN PASSEN DIE FOLGENDEN REZEPTE:
Ägyptisches Bohnengericht (Fuhl medames)
Überbackene Süßkartoffeln
1000 Bohnen-Suppe
5-Gewürzpulver
5-Minuten-Schokokuchen
5-Minuten-Tomatensuppe
6-Korn Brot
Abgehackte Finger
Adzukibohnen in Tomatenreis
Adzukibohnenauflauf mit Polentakruste
Rezept ausgeben
Zurück zur
Auswahl
© veganwelt.de © ullrykka.de

RZ3.PL


Veganwelt Rezeptdatenbank
http://fafner.dyndns.org/~mitsam/cgi-bin/rz3.pl
Google
A+: a progr...rogrammers
Apple (43)
Veganwelt Rezeptdatenbank
REZEPTDATENBANK
5-Minuten-Schokokuchen
Ein feiner und sehr einfacher Schokokuchen für alle, die glutenfrei leben müssen.
Zutaten:
(immer vegane Versionen verwenden!)
0.3 cup Öl
3 Tl Backpulver
0.25 cup Kakao
1 cup Reismehl
0.5 Tl Salz
0.5 cup Stärkemehl
1 Tl Vanilleextrakt
1 cup Wasser
1 El Zitronensaft
1 cup Zucker
Zubereitung:
Alle trockenen Zutaten in einer Schüssel vermengen.

Adapting the CGI scripts

Adapting the three CGI scripts was fairly easy – the Perl RDB interface is slightly different to the MySQL interface used on the LINUX system, but changing some lines did the trick.

Testing could begin. . .

. . . and resulted in a mere catastrophe:

- The response time was incredibly slow and
- sometimes the CGI scripts crashed with messages indicating that there were conflicting accesses to the RDB system.

Tuning the system

First I decided to tune the system – the LINUX system usually answered in less than a second while the VAX needed up to 15 seconds for a single request, which was completely inacceptable.

Obviously starting the Perl interpreter for every single request was the main reason for the poor performance of the system.

Using an Apache server it would be possible to use ModPerl, while the web server used on the VAX, WASD, offers two solutions for this problem:

- CGIplus and
- RTE.

CGIplus vs. RTE

CGIplus allows the script called by the webserver to persist. So calling such a script will only once need the complete startup overhead.

This feature requires some (minor) changes to the script to be used in this mode of operation to avoid problems arising from the script's persistence.

Another method, called RTE, allows the (Perl) interpreter called to stay resident in memory while the script will have to be loaded during every call as before. The performance gain is not as dramatic as with CGIplus but using RTE is yet simpler than using CGIplus (which already is quite simple).

The effects of using RTE

I decided to give RTE a try. The observed effects on performance were dramatic! The scripts called now ran up to three to five times faster (the scripts are small so that parsing them has only little effect) and I decided to stay with RTE.

This cured the startup overhead from starting the Perl interpreter over and over again. It did not cure the infrequent crashes which became annoyingly nor did it alone suffice to get a speed as the LINUX system offered.

Please do not forget that we are dealing with a VAX – a fast VAX, but still a VAX. Compared with a modern PC based LINUX system running on a CPU clocked with several GHz even a VAX-7000/820 is a very, very slow system. So it was clear that some loss in performance was to be expected.

Crashes. . .

It was time to have a look at the crashes occuring spuriously.

To force crashes like those observed during the test runs, two small test programs were written using Perl. Both connected to the RDB system and performed select statements involving several joins.

Running one instance of this script did not reveal any problem (apart from the expected fact that RDB on a VAX is a lot slower than MySQL on a multi-GHz x86 based platform).

Running a second instance nearly instantaneously resulted in a crash. Fiddling around a bit it became clear that there was a bug in the RDB driver used by Perl! A bug I could not find and thus not correct. . .

That's it – game over. . . . . or is there a way to circumvent this?

Making database access faster and stable

Indeed, there is a solution to this problem: The RDB driver is very stable then handling only one connection at a time, so it is not that bad.

Taking this and the fact that the RDB system is slower than the MySQL database running on the LINUX server, a new idea emerged:

We need a database proxy!

This proxy would hold a single connection to the database and talk with a plethora of clients via TCPIP sockets thus serializing requests from the clients. Doing this it could readily cache results previously read from the database, thus speeding up everything.

The implementation of this database proxy was done by my friend Thomas Kratz.

Client/server communication

Whenever a client wants to execute an SQL statement on the RDB database (or any other database since the server supports multiple connections to different databases) it will wrap up its request into a hash data structure and send this whole structure to the server:

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

The SELECT-entry contains the select statement to be executed (insert, update, delete statements would be transmitted in a DO-entry which facilitates the cache invalidation logic of the proxy) while the ALIAS-entry contains a symbolic name for the destination database to be used.

Sending a request to the server

Sending such a request to the server is a bit tricky since the server should receive the complete datastructure prepared by the client, not only the plain text request. Thus sending makes use of the `freeze` function from the `Storable` module:

```
print $sock encode_base64(nfreeze(
  {
   $stmt_type => $stmt,
   ALIAS      => 'RECIPES_PRODUCTION',
  }
), '', ), "\n";
```

The frozen data structure will thus be sent in a base-64 encoded format to the server.

Receiving a request

The server reads this base-64 stream from the socket and recreates the datastructure with a statement like this:

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

In the following it will have a look at its local cache if there has been the same request before. If this is the case the server will just send the data from its cache.

Otherwise it will send the SQL statement to the RDB database, cache the resulting data set and send it back to the client.

Sending the result data set

Sending the result data set is a bit more tricky than processing the client request. This is due to the fact that the result data set may very well exceed several 100 kB which can not be sent through a socket in a single step.

Therefore it is necessary to split the result data set into smaller chunks and send these to the client which has to concatenate the chunks in order to restore the data. Creating and sending these chunks is done like this (`$response` contains the data to be sent):

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

Speed up effect

The speedup effect using this database proxy is dramatic:

The first request from `RZ1.PL` to build the category selection list takes 3.52 seconds since the proxy cache is empty and the statement has to be performed by the RDB database system.

Every following request requires about 0.34 seconds. This is about

10 times faster

than the direct access method!

Conclusion

The initial problem, the bug in the RDB driver, turned out to be a big advantage. It triggered the development of this simple database proxy which made the resulting system consisting of

- the WASD web server with the Perl RTE,
- RDB
- several Perl based CGI scripts and, finally,
- the Perl based database proxy

running on a VAX **as fast** as its predecessor system running on a modern LINUX server (and sometimes even **faster**)!

Comparing the 70 VUP/CPU of the VAX-7000/820 with the multi GHz driven Intel processor of the LINUX server makes this even more impressive.

Conclusion

In its current configuration the proxy server needs about 10 MB of cache memory under heavy load which is a small price to pay for a speedup factor of nearly 10.

Even using obsolete hardware – and as wonderful a system a VAX-7000/820 is as obsolete it is – it is possible to compete with so called industry standard systems. It takes a bit more effort to accomplish this but isn't it great to replace a modern LINUX system with a VAX running OpenVMS?

More information

The vegan recipe database is the heart of

`http://www.veganwelt.de`

and may be reached directly at

`http://fafner.dyndns.org/~mitsam/cgi-bin/rz1.pl`

The author may be reached at `ulmann@vaxman.de`.