Roxen WebServer 5.4

Database Tutorial

Contents

1 Introduction

Welcome to the Roxen CMS Tutorials. This section is dedicated to all users of Roxen CMS. The tutorials are intended for both beginners and experienced users, and we hope that all find some interesting reading and get creative ideas.

It is assumed that the reader is familiar with HTML and have some knowledge of XML.

As always, if you have any suggestions, comments or complaints regarding these tutorials do not hesitate to send an email to manuals@roxen.com and if the issue is an obvious bug do not hesitate to report it to Bug Crunch, our bug tracking system.

2   Tutorial Contents

This section of the manual deals with how Roxen CMS and Pike can connect to SQL databases, retrieve data and modify the data stored there. It doesn't aim at teaching SQL or how to design a database, save for very simple cases, which are not surprisingly the most common in normal Web-related programming tasks. So you won't find references to triggers, stored procedures, referential integrity or complex privileges management here: they *can* be used from Roxen CMS or Pike, but they're more of an SQL matter, which is out of the scope of this manual.

Roxen CMS and Pike offer an uniform layer to access all the supported databases. However such a layer does not cover anything but issuing queries and retrieving data. SQL is unfortunately another matter: it is an ANSI standard, but just about every SQL server has its own dialect, which may be a subset or a superset of the standard. You'll need to check your server of choice's documentation about its version of SQL.

This section of the manual tries to be a reference for both Roxen CMS and Pike programmers. To do so, most examples will be available in two versions, a Pike snippet of code, and RXML code.

Note!

The RXML <sqloutput> and <sqltable> have been deprecated in favour of the <emit source="sql"> container. In this manual the 'old' tags are used, but the new tag is briefly introduced on the The emit and sqlquery Tags page.

2.1   Introduction to MySQL

MySQL by TcX AB is a simple SQL server, very popular among web-designers. It is a relatively simple and lightweight server, which aims at being very fast, but is not fully ANSI-SQL compliant, as it doesn't support features such as triggers or sophisticated access control.

Since MySQL is so popular among web-developers, it was chosen as the reference RDBMS for Roxen CMS. This chapter will introduce you to it, and to some of the pitfalls most easily encountered when using it. The examples shown are however as cross-platform as they could be: they should work with any SQL server which claims at least a partial degree of ANSI-SQL compliance.

- Privileges
- Building a Sample Database

2.2   Querying

Querying a server is by far the most used DB-related functionality. Almost everything (in some cases, plain everything) you'll do when interacting with an SQL server goes through specifying correctly-formed SQL queries, sending them to a server and then interpreting the results the server sends back.

RXML 2 offers two different ways to query a server, Pike too. These are needed to fit all situations; a query may yield results, or it might not, and the only way to tell the difference is by looking at the SQL code being executed by the server.

It would seem that programs (or RXML pages) accessing SQL resources are difficult and cryptic because the results queries can return are inherently dynamic in number

and structure. Fortunately, very few programs need to handle the full range of possible outcomes from a query. In fact, most SQL queries are either non-interactive, or are parametric. This means they have a fixed structure where a few values (or no value if the query is non-interactive) change on each execution. This ensures that the results (or lack thereof) can be predicted accurately; if not in number, at least in structure.

It is best to see SQL statements not as a foreign plug-in into a program's execution flow, but as an integral part of it. Whenever the data storage structure changes, the program must be changed according to it (this is why database design is such an important matter: a bad database design decision might end requiring an application rewrite almost from scratch).

- The query() function

- The big_query() function

- Quoting

2.3   Data Extraction

In this chapter we'll introduce how to perform data-extraction queries. We'll introduce the SQL syntax for data-extraction, and provide a few examples, both in RXML and in Pike.

- SQL Syntax

- Conditions

- Sorting

- Limiting

- Functions

- Features Missing from MySQL

2.4   Data Insertion

In this chapter we'll introduce how to insert data into a database.

Notice that data insertion and modification are two different operations, using two different SQL commands.

- Insertion Syntax

2.5   Using RXML Features with SQL Databases

In this chapter we'll examine how to exploit some RXML features when working with SQL databases.

The examples here contained are geared towards SQL-driven data sources, but it is not of course the only use for them.

- The tablify Container

- The Business Graphics Module

- The emit and sqlquery Tags

2.6   Database Maintenance

Up to this point we have assumed the databases to be already present for us. But this of course isn't the case in some real-world situations.

Designing a database is a very complex task for nontrivial cases. It is also a very delicate operation: when dealing with data-storage-related applications, usually the application is built around the data, and not the other way around. So a bad data storage design will snowball, leading to a bad application design, which is very expensive to fix, going as far as a rewrite from scratch.

So for the umpteenth time we'll remark that if an application uses non-trivially organized data, the best solution is to hire someone to design the database.

In this chapter we'll examine how to build and delete a database, how to set the tables and indices up or remove them. We'll assume that the database structure is so simple to be self-evident (which is often the case for web-related systems), database design won't be taken into account.

Also, the examples will be in pike-only: these activities are meant to be used only once at database-creation, and are really not suited for a web-based application.

- Database Creation

- Creating Tables

- Indices

- Dropping

3 Introduction to MySQL

3.1 Privileges

A fundamental point, and a very common pitfall, in day-to-day MySQL operations is understanding how the MySQL privileges system works. This chapter is meant to provide only an overview of the basic functionalities. You may safely skip this section when you only use the internal database shipped with Roxen CMS, since this hassle is already covered by the internal workings of Roxen CMS. To get more details on the MySQL privileges system, please refer to the MySQL manual.

The first noteworthy aspect is that MySQL does not use the security features of the host system. It has its own authentication schemes, different from the system's.

This section uses the GRANT and REVOKE commands, which have been implemented in MySQL version 3.22.11. If you have an earlier version, you're suggested to upgrade.

MySQL offers four levels of access control: global, database, table and column. We'll only deal with the first two, as they are the most important. If you think your setup would require finer-grained security, you'll probably also need to hire a knowledgeable Database Administrator: data storage and retrieval is a very sensitive matter, performance- and security-wise.

To manage privileges you'll have to use the GRANT and REVOKE SQL commands. Their (simplified) syntaxes are:

```
GRANT <priv_type> [, priv_type ...] ON <*.*|database.*> TO
<user name> [IDENTIFIED BY '<password>']
      [, <user name> [IDENTIFIED BY '<password>'] ,...]
      [WITH GRANT OPTION]

REVOKE <priv_type> ON <*.*|database.*> FROM <user name>
[, <user name>, ...]
```

Where *priv_type* is a type of privilege, chosen among

```
ALL [PRIVILEGES]     FILE           RELOAD
ALTER                INDEX          SELECT
CREATE               INSERT         SHUTDOWN
DELETE               PROCESS        UPDATE
DROP                 USAGE
```

"ALL" or "ALL PRIVILEGES" means (guess what?) everything. "USAGE" is the same as "no privilege".

If you use the "*.*" syntax, the altered privileges will be at the global level. If you use "database.*", you'll touch the database-level privileges.

The user name can have the form 'username@host', and can have wild-cards ('%' or '_', see later) in both the host or username parts.

If you specify the "IDENTIFIED BY..." clause, you'll set a password for the named user. Users without a password are legal in MySQL, but they are a very serious security hazard.

WITH GRANT OPTION means that the user is given the privilege to grant the same privileges he has to other users. It can be revoked with the syntax

```
REVOKE GRANT OPTION ON ... FROM *user name*
```

In the default MySQL setup there is an anonymous user ('%@localhost'), whose existence can cause unexpected results while authenticating other users. It is advised to remove the anonymous user. It can't be done with the GRANT syntax, but you have to do it manually as detailed the examples below.

Also, in the default MySQL setup there's an empty database named 'test', open for anonymous use. We'll use it throughout this tutorial, but it's advised to remove it ('DROP DATABASE test') after you're done, as it can be a source of denial-of-service attacks.

Note!

> In order to maximize the security of your site, it's always best to give each user the minimal privileges allowing him to do his work.

Create a new user named 'kinkie', having basic data access to the 'test' database.

With Pike:

```
$ pike
Pike v0.6 release 116 running Hilfe v2.0 (Incremental Pike Frontend)
object o=Sql.sql("mysql://root:<password>@localhost/mysql");
o->query("grant select,insert,update,delete on test.* to kinkie identified
by
               '<password>');
o->query("flush privileges");
```

Or, from the MySQL monitor:

```
$ mysql -uroot -p<password> mysql
> grant select,insert,update,delete on test.* to kinkie identified by
  '<password>';
```

Create a new user named 'dbmanager' having full SQL access to all databases (but deny him server-related maintainance tasks):

With Pike:

```
object o=Sql.sql("mysql://root:<password>@localhost/mysql");
o->query("grant select,insert,update,delete,create,drop,alter,index on
               *.* to dbmanager identified by '<password>'");
```

Disable the 'nasty' user.

With Pike:

```
object o=Sql.sql("mysql://root:<password>@localhost/mysql");
o->query("revoke all on *.* from nasty");
```

Note!

> This will NOT remove the user from the authentication database, only prevent him from connecting.

To remove the user completely, you'll have to act directly on the "mysql" database; with Pike:

```
object o=Sql.sql("mysql://root:<password>@localhost/mysql");
o->query("delete from user where user='nasty'");
```

Delete the anonymous users and the public-access entries to the test databases:

```
object o=Sql.sql("mysql://root:<password>@localhost/mysql");
o->query("delete from user where user='');
o->query("delete from db where db like 'test%'");
```

You might have noticed there are no RXML examples in this chapter. These tasks can be executed from RXML (provided that you connect with enough access rights), but it's not advised to have RXML code perform such critical tasks: one reload too much could make your database useless. Using the DBs tab in the server Administration Interface could prove handy, though.

3.2   Building a Sample Database

In the previous chapters we introduced how to build and install your database server. In this chapter we'll build the sample database that will be used throughout this manual.

Make sure your MySQL daemon is running and that the MySQL program files are in your PATH, then use this command line

```
$ mysqladmin -u root -p password create sample
```

The database server will create files making a database. A single database server can handle many databases: each is a data repository, completely independent from all the other databases hosted by the same server.

A database can be dumped using the "mysqldump" utility. It will create an SQL script file, that when run will re-create the structure and contents of a database. The sample database was dumped with this utility.

You'll now want to fill in the sample database. To do so, you must use the "mysql" utility, with these command lines:

```
$  mysql -u root -p password sample <sample_db.schema
$  mysql -u root -p password sample <sample_db.data
```

The "mysql" utility is a so-called "interactive monitor", an application whose purpose is to execute arbitrary SQL statements interactively. It is a very powerful and useful tool, and it's advised to get familiar with it.

The two lines are required because I chose to dump the database structure (the so-called 'schema') and the data separately.

The sample database is a simplified excerpt of the CIA World Factbook. It only covers a few nations, and for each nation only a small amount of data.

From this moment on, we will not use the administrative user to develop the examples. Instead, we will create a user named 'user' with password 'password' and use it. Make sure you remove that user once you are done with this tutorial.

To create the user, you will need to issue this query from inside the mysql interactive monitor:

```
$ mysql -u root -p *password* sample
```

...which grants all privileges on sample.* to the user identified by 'password'.

3.2.1 The Sample Database Structure

The sample database consists of four tables. The first one, named 'ids' is used to tie country names to their 2-letter unique codes, which are used everywhere else. The one named 'areas' has the purpose to tie a few world areas to an unique integer identifier.

Although in theory both those tables could be not necessary (they handle a very simple association, the 2-letter country code could be very easily substituted with the country name in every place it appears), they actually serve two purposes: they make the other tables more compact and efficient (a 2-letter unique code is simpler to handle and requires less space than a variable-length name), and they formalize and restrict the domain of possible choices, allowing for a cleaner and more robust design.

The 'countries' table contains a few descriptive fields for each country, possibly in relation with other tables. The 'boundaries' table contains informations about the countries boundaries. It could be considered relationed to the 'countries' table, but it's more practical to see it as a separate entity.

```
$ mysql -u root -p *password* sample
```

4   Querying

4.1   The query() function

The query() method of the Sql.sql object is the "simple" query interface. It is meant to be used for those queries that return little or no data.

It's signature could look frightening:

```
array(mapping(string:string|float|int)) query (string sql)
```

but it isn't that bad, really.

The returned value is an array, one element for every row, of mappings whose indices are the column names, and values the column contents.

So in order to access the "foo" column in the fourth returned row, you'll use

```
mixed datum = db[4]->foo;
```

If there are no results, the method will return an empty array.

Find out the country code for Italy

```
string country_code_for_italy() {
    object db=Sql.sql("mysql://user:password@localhost/sample");
    array result=db->query("select code from ids where name='Italy'");
    if (sizeof(result)>0) { //if there is any result
        return result[0]->code;
    }
    return 0; //no code found
}
```

The reason why this interface is only suited for simple queries is that it will fetch the whole results set and store it locally. It's not that big a deal for small databases, but make a small mistake in specifying the query on an HUGE database, and it will be tens or hundreds of megabytes to fetch. Talk about bloat... If you're going to retrieve potentially huge data-sets, you'll need the big_query interface instead. It's a bit more complex to use, but it will allow you fetch results on demand.

4.2   The big_query() function

The big_query() function allows programmers more control than the simpler query() function on how data is retrieved from the database server, as it allows fetching the data rows on demand. This is especially useful when you wish to do client-side computations on the fly on big datasets, that would require too much memory to be completely fetched and then processed.

The function's signature is `object(Sql.sql_result) big_query(string sql)`

The returned object is a handle to the results dataset. It offers methods allowing you to retrieve rows and get informations on the dataset itself.

int num_rows()

returns the total number of rows in the result object. Some drivers (i.e. Sybase) might not provide this functionality, and thus the only way to know how many rows there are is by explicitly querying the server (see example below).

int num_fields()

returns the number of columns for the result object. This function is usually meant for development purposes only, you shouldn't need it on production systems.

int eof()

returns true if all rows in the result object have been fetched.

array(mapping(string:mixed)) fetch_fields()

retrieves descriptions for the columns in the results set. The mappings in the returned array (one for each column) have some default fields, but they change in different drivers. See the example below to discover what fields your driver of choice provides. This function is usually used for development purposes only. You should rarely need it on production systems. Also, notice that the returned results will correspond to the server's idea of the fields, which might be different from the actual declaration.

void seek(int skip)

This method allows to skip fetching some rows (the skip argument must be a nonnegative integer).

int|array(string|int) fetch_row()

The most important function of all, this one allows you to fetch a row of data. There is one element of the array for each column, and the columns are ordered as returned by fetch_fields() and as specified in the SQL query. If 0 is returned instead, it means that there are no more rows to retrieve. An integer 0 is returned for (SQL) NULL values, while all types of stored data are returned as strings. It's up to the user to do the adequate type casts where appropriate. Type information can usually be retrieved with the fetch_fields() function.

Note!

There are some restrictions on how data are retrieved with some drivers. Please check the drivers-specific section for more detailed information.

Print the name and background for all the countries in Europe.

```
object(Sql.sql) db=Sql.sql("mysql://user:password@localhost/sample");
object(Sql.sql_result) result=db->big_query(
      "select ids.name, countries.background "
      "from ids,countries,areas "
      "where areas.name='Europe' and countries.map_refs=areas.id and "
      "ids.code=countries.country");
array(string) row;
while (row=result->fetch_row())
{
      //row[0] is the country name, row[1] is the background info
      write("---"+row[0]+"\n");
      write(row[1]+"\n");
}
```

Now let's try writing a simple pikescript handling a multi-page table without resorting to the LIMIT SQL clause (see ../data_extract/limiting). The main purpose of this example is showing the usage of num_rows and seek functions, so despite being a complete example, it's a bit stretched (in real-world, this is one of the cases where

the Roxen CMS caching capabilities come handy). Also, it doesn't output formally valid HTML, and it doesn't handle exceptions. We'll show the 'ids' table contents, with ten entries per page and links to the other pages.

```
#define DBHOST "mysql://user:password@localhost/sample"
#define QUERY "select name, code from ids order by name"
#define ENTRIES_PER_PAGE 10
#define SEEK_IS_BROKEN

string parse (object id)
{
  string toreturn;
  object(Sql.sql) db;
  int number_of_entries, number_of_pages, page, j;
  object(Sql.sql_result) result;
  array(string) row;

  page=(int)(id->variables->page);
  toreturn="<table border=1>\n";
  db=Sql.sql(DBHOST);                               //connect
  result=db->big_query(QUERY);                      //query
  number_of_entries=result->num_rows();             //get the number of
rows
#ifdef SEEK_IS_BROKEN
  //it looks like mysql's implementation of seek() is broken, probably at
  //the mysql level in my version (3.22.29). I'll do a loop to emulate
seek
  for (j=0;j<ENTRIES_PER_PAGE*page;j++)
    result->fetch_row();
#else
  result->seek(ENTRIES_PER_PAGE*page);              //skip unneeded
results
#endif

  for(j=0; j<10; j++) {                    //at most 10 results
    row=result->fetch_row();               //fetch the row
    if (!row)                              //no more data?
      break;                              //exit
    toreturn += "<tr><td>"+row[0]+"</td><td>"+row[1]+"</td></tr>\n";
  }
                                          //now the links section
  number_of_pages=number_of_entries/ENTRIES_PER_PAGE;
  if (number_of_entries%ENTRIES_PER_PAGE)
    number_of_pages++;                     //there might be an incomplete page
  toreturn+="<tr><td colspan=2>";
  for (j=0;j<number_of_pages;j++)
  {
    toreturn += "<a href='"+id->not_query+"?page="+j+"'>"+(j+1)+"</a> ";
  }
  toreturn +="</td></tr>";
  toreturn +="</table>";
  return toreturn;
}
```

What happens if the num_rows function is not available? The same results can be obtained via a simple SQL query, obtained modifying the actual query being executed. It is of course less efficient because two queries are issued instead of one. But it's better than nothing.

The query is obtained replacing the list of fields being fetched with the 'COUNT(*)' SQL function. It has slightly different semantics for complex queries, but for all the query types covered in this manual, it works. You might want to alias it for easier manageability (see ../data_extract/syntax).

So the previous example would have been written as:

```
#define DBHOST "mysql://user:password@localhost/sample"
#define COUNT_QUERY "select count(*) as num from ids"
#define QUERY "select name, code from ids order by name"
#define ENTRIES_PER_PAGE 10
```

```
#define SEEK_IS_BROKEN

string parse (object id)
{
  string toreturn;
  object(Sql.sql) db;
  int number_of_entries, number_of_pages, page, j;
  object(Sql.sql_result) result;
  array(string) row;

  page=(int)(id->variables->page);
  toreturn="<table border=1>\n";
  db=Sql.sql(DBHOST);                                       //connect
  number_of_entries=(int)(db->query(COUNT_QUERY)[0]->num); //(1)
  result=db->big_query(QUERY);                             //query
#ifdef SEEK_IS_BROKEN
  //it looks like mysql's implementation of seek() is broken, probably at
  //the mysql level in my version (3.22.29). I'll do a loop to emulate
seek
  for (j=0;j<ENTRIES_PER_PAGE*page;j++)
    result->fetch_row();
#else
  result->seek(ENTRIES_PER_PAGE*page);                     //skip unneeded
results
#endif

  for(j=0; j<10; j++)                         //at most 10 results
  {
    row=result->fetch_row();            //fetch the row
    if (!row)                           //no more data?
      break;                                   //exit
    toreturn += "<tr><td>"+row[0]+"</td><td>"+row[1]+"</td></tr>\n";
  }
                                        //now the links section
  number_of_pages=number_of_entries/ENTRIES_PER_PAGE;
  if (number_of_entries%ENTRIES_PER_PAGE)
    number_of_pages++;                        //there might be an incomplete page
  toreturn+="<tr><td colspan=2>";
  for (j=0;j<number_of_pages;j++)
    toreturn += "<a href='"+id->not_query+"?page="+j+"'>"+(j+1)+"</a> ";
  toreturn +="</td></tr>";
  toreturn +="</table>";
  return toreturn;
}
```

(1): this line is a quick shortcut using the simpler query (see query) interface. It is appropriate in this case, because the results are tiny. We didn't make any checks on the results either, because their structure is very well-known.

The values returned by fetch_fields depend on the server you are connecting to, save for a few ones which should be always there. This is one of the reasons why you shouldn't need to use this function except during development. Let's see an example of it in action:

With Pike:

```
> object db=Sql.sql("mysql://user:password@localhost/sample");
Result: object
> object res=db->big_query("select country, map_refs, flag from
countries");
Result: object
> res->fetch_fields();
Result: ({ /* 3 elements */
    ([ /* 7 elements */
      "decimals":0,
      "flags":(< /* 2 elements */
            "primary_key",
            "not_null"
>),
      "max_length":2,
      "length":2,
      "type":"string",
```

```
        "table":"countries",
        "name":"country"
    ]),
    ([ /* 7 elements */
        "decimals":0,
        "flags":(< /* 1 elements */
            "not_null"
>),
        "max_length":1,
        "length":4,
        "type":"char",
        "table":"countries",
        "name":"map_refs"
    ]),
    ([ /* 7 elements */
        "decimals":0,
        "flags":(< /* 2 elements */
            "not_null",
            "blob"
>),
        "max_length":13127,
        "length":65535,
        "type":"blob",
        "table":"countries",
        "name":"flag"
    ])
})
```

An array of mappings is returned, one mapping for each field. The "name" key is always present, as is the "flags" key. The other fields change depending on the server, and (as you might see) on the data type.

4.3   Quoting

As better explained in the Conditions page, constants (especially string constants) must be quoted in SQL. How the quoting must actually be composed will be explained later, now we'll introduce the facilities Pike and RXML offer to perform the quoting operation. The operation is server-transparent (that is, it adapts to the various servers' quoting schemes.

4.3.1   Pike

The Pike solution is pretty straightforward: quoting is handled via the Sql.sql->quote(string) method. It returns a string, which is the quoted argument.

It is supposed to be used when assembling a query, and is strongly encouraged to use it whenever a query is interactively built from some user's input: a malformed input could break the query by causing an SQL syntax error. It's useless to say that it could also be used maliciously, to completely alter the query structure, thus giving access to the lowlevel database contents.

Let's write a small interactive Pike application which prints the background for user-entered countries.

```
#!/usr/local/bin/pike
#define DATABASE "mysql://user:password@localhost/sample"

//sample program: find out some country's background information
int main() {
  object readline=Stdio.Readline();          //used for interactive
input
  object db=Sql.sql(DATABASE);               //connect to the DB
  readline->set_prompt("Country (q to quit)> ");
  string input;
  array(mapping(string:mixed)) result;
  while (input=readline->read()) {           //while !eof
```

```
    if (input=="q") break;                    //exit on "q"
    //query-building. I like to use sprintf to build parametric queries,
as
    //it shows the query structure in the source (increased readability),
    //as well as allowing easier control over the SQL statement
    string query=sprintf("select background from countries, ids "
                         "where countries.country=ids.code and "
                         "name='%s'",
                         db->quote(input)        //notice the quoting!
                         );
    result=db->query(query);
    if (!sizeof(result)) {
      write("No such country in the database\n");
      continue;
    }
    write(result[0]->background+"\n");
  }
}
```

4.3.2   RXML

There are two occasions in which you'll want to do quoting in RXML when performing SQL-related operations: parametric query building and results quoting (for instance to populate a selection list). In most cases the RXML parser tries to do the "sensible" thing, but sometimes that's just not enough, and you'll need to manually override the parser's "opinion".

On production systems, any degree of freedom is a risk: on such systems it is thus recommended to always specify the encodingq, as it will lessen the probability of errors, failures or security vulnerabilities.

4.3.3   Parametric Queries

You can use the standard entity-syntax to build parametric queries: just use entities in your query strings. Make sure to force the sql encoding, or you might head into trouble.

The example beneath does the same task as the above pike application using RXML. It performs both of the encoding operations: results-encoding to populate a selection list and variable encoding to perform a parametric query:

```
<form method="post" action="&page.url;">
Select a country: <select name="country">
<emit source="sql"
      query="SELECT name,code FROM ids,countries
              WHERE countries.country=ids.code
              ORDER BY name">
<option value="&_.code;">&_.name;</option></sqloutput>
</select>
<input type="submit">
</form>

<if variable='form.country'>
<sqltable host="mysql://user:password@localhost/sample"
  query="SELECT name,background FROM countries,ids
          WHERE countries.country=ids.code
            AND ids.code='&form.country:sql;'"/>
</if>
```

5 Data Extraction

5.1 SQL Syntax

The most basic SQL syntax for a data-extraction query is:

```
SELECT what FROM table name[, table name ...] [WHERE conditions]
```

what defines what you wish to get from the query. It can be a column name (more on column names later), a function to be performed on the retrieve data (more on this in the functions chapter). The special notation '*' means "all columns from all the specified tables".

In order to extract everything from a table, with RXML:

```
<sqltable border="1"
  host="mysql://user:password@localhost/sample"
  query="SELECT * FROM boundaries">
```

with Pike:

```
string parse (object id) {
    object db=Sql.sql("mysql://user:password@localhost/sample");
    array(mapping) results=db->query("select * from boundaries");
    string output="<table border=1>";
    foreach (results,mapping m) {
        output+="<tr><td>"+m->country_1+"<td>"+m->country_2+"<td>"+
        m->length+"</tr>\n";
    }
    output+="</table>";
    return output;'
}
```

If we wanted to get the results only for a column in that table, we would have instead

with RXML:

```
<sqltable border="1"
  host="mysql://user:password@localhost/sample"
  query="SELECT length FROM boundaries">
```

Of course you can select more than one column, simply having what be a comma-separated list of column names.

With RXML:

```
<sqltable border="1"
  host="mysql://user:password@localhost/sample"
  query="SELECT country_1, country_2 FROM boundaries">
```

Using a single table doesn't harness the power of relations. Those are not "physical" entities, but are built when a query is executed if multiple tables are specified together with conditions to explain how the data from the tables should be collated (or "the tables are joined"). Usually an equality test is used to specify those conditions, but it's not a requirement. The result of the join operation is a virtual table merging those records from every involved table that satisfy the specified conditions.

Let's print the name of the known countries and the geographic regions they belong to. The country names are in the 'ids' table, the regions are in the 'areas' table, the two are tied via the 'countries' table. The relations we'll use are two: ids.code must be equal to countries.country, and countries.map_refs must be equal to areas.id

with RXML

```
<sqltable border="1"
  host="mysql://user:password@localhost/sample"
  query="SELECT ids.name AS country, areas.name AS region
         FROM ids, countries, areas
        WHERE ids.code=countries.country
          AND countries.map_refs=areas.id">
```

5.1.1 Column Names

A column can be addressed in two ways: "plain" and "dotted notation". The latter is the more complete form, and is guaranteed not to be ambiguous. The former is allowed for brevity's sake by most servers (including MySQL), but only when no confusion is possible.

5.1.2 Aliases for Columns

It is possible (usually to have a function result with a simpler name) to alias the names of the returned columns, simply extending the what parameter above with the syntax

```
column_name AS alias
```

The values will be then available in the result as "alias" column, rather than "column_name".

With RXML:

```
<sqltable border="1"
  host="mysql://user:password@localhost/sample"
  query="SELECT country_1 AS first_country,
              country_2 AS second_country FROM boundaries">
```

See the functions chapter to see how for an example when using functions.

5.1.3 Aliases for Tables

Table names can be aliased with the "as" syntax, too. This is especially important in one case, and that is when you need to cross-reference a table with itself, or if a table is involved in multiple relations with another. It's illegal in SQL to have two or more tables with the same name mentioned in the tables list of a query.

With our sample database, it's necessary to alias a table if we want to expand the country codes in the boundaries table to their names. In order to accomplish that result, we will need to:

With RXML:

```
<sqltable border="1"
  host="mysql://user:password@localhost/sample"
  query="SELECT ids_1.name AS name_1,
              ids_2.name AS name_2, length
          FROM ids AS ids_1, ids AS ids_2, boundaries
        WHERE ids_1.code=boundaries.country_1
          AND ids_2.code=boundaries.country_2">
```

Here we aliased two times the 'ids' table for clarity's sake, we could have aliased it only once. Also, we aliased the column names for the same reason.

5.1.4 No Tables Involved

It is possible to have queries which don't involve any table, simply by not specifying the "FROM" clause. Such queries are not very useful, except sometimes to perform server-assisted translations.

With Pike:

```
> object db=Sql.sql("mysql://user:password@localhost/sample");
Result: object;
> db->query("select now() as time")[0]->time;
Result: "2000-02-29 12:12:57"
```

The conditional part of a query is explained in the following chapter.

5.2 Conditions

The condition part of the query, as shown in the "syntax" paragraph is a boolean expression, usually arbitrarily complex (old versions of MiniSQL have heavy limitations the syntax of this portion). Only rows that satisfy it will appear in the results set. If none does, the results set will be empty.

When evaluating the condition, column names are substituted with the data they contain, and operators are evaluated according to a well-specified grammar. Constants must be quoted according to their type.

5.2.1 Numeric Constants

Integer and floating-point numbers are not quoted. They can be told apart because floating-point numbers have the decimal separator (.). Usually the server's parser is quite lenient though, fixing types when possible according to the context.

5.2.2 Column Names

These are not quoted. Since they mustn't be ambiguous this poses a bit of limitations on column names. As a general rule, legal C variable names are legal column names (unless they are reserved words of course). SQL is a bit more lenient than C, so you should get a little more leeway.

5.2.3 String Constants and Quoting

Strings are quoted using the apostrophe symbol ('). If a string contains the literal apostrophe character, it must be escaped. Different escaping schemes are specified, the most usual ones being doubling it (i.e. 'It''s a shame') or prepending it with a backslash (i.e. 'It\'s a shame').

Let's obtain from our sample database the total area of Italy.

With RXML:

```
<emit source="sql" host="mysql://user:password@localhost/sample"
  query="select name, area_tot from ids, countries
    where ids.code = countries.country and ids.name='Italy'>
&_.name;'s total surface is &_.area_tot; sq. km.
</emit>
```

5.2.4   Other Data Types

Other data types are usually represented as formatted string, which get interpreted by the server according to the context.

5.2.5   The LIKE Operator

This operator is used to do glob-like matching. It has the syntax value `LIKE PATTERN` where the value is usually a column, and the pattern a string literal, possibly containing two magic characters: '_' and '%', which act like glob characters '?' and '*', that is they match any (single) character, and any arbitrarily long sequence of any character. If what you're matching against contains the literal '_' or '%' characters, you can escape them prepending the backslash character '\'.

Let's try to find out the countries neighbouring Italy. The right way to do so would be looking in the 'boundaries' table. But a summary can be found in the countries.location text, and we'll use that.

With RXML:

```
<sqltable border=1 host="mysql://user:password@localhost/sample"
  query="select name from ids, countries
         where countries.country=ids.code and location like '%italy%'">
```

Notice that the "column like '%something%'" syntax (with leading and tailing globs) is very inefficient, and should be avoided whenever possible.

MySQL offers the more powerful REGEXP operator, with the syntax value `REGEXP expression` where the value is usually a column name or a function result, and expression is a string-quoted regular expression.

5.2.6   NULL Column Values

Some columns can be empty, or (in SQL terms) be NULL. To deal with them when selecting data, you use the 'IS' syntax, which takes the form value `IS [NOT] NULL` where value can be obtained from a a column (thus be a column name) or can be a constant value (of course it would be rather dumb to evaluate a constant expression, but you can of course do that if you wish).

5.3   Sorting

Data in a result is in undefined order. To have it sorted to some other order, the `ORDER BY` clause can be used. It modifies the basic query syntax:

```
SELECT <columns> FROM <table> [, <table> ...] WHERE <condition>
       ORDER BY <column name> [DESC] [, <column name> [DESC] ...]
```

This will sort the returned rows according to the specified columns, depending on the column type (numerically if the column type is numeric, syntactically if the column type is textual, etc.) If the `DESC` modifier is specified, the rows will be sorted in reverse (descending) order.

5.4   Limiting

It is sometimes useful not to retrieve all the rows in a query.

You can do it using SQL or (in Pike) you can do it by simply not using some of the results you fetch.

Doing it in SQL has some advantages, for instance it will reduce the load on your SQL server, your Pike application and your internal network. On the other side, the syntax for performing such an operation is not part of the SQL standard, and so every server adds its own extensions to perform this operation.

We will introduce the MySQL syntax here. For other systems, consult your server's of choice SQL reference manual.

MySQL offers limiting via an extension of the SELECT syntax, which gets changed like this:

```
SELECT <columns> FROM <tables> [WHERE <condition>] [ORDER BY <columns>]
       [LIMIT [offset,]howmany>]
```

`offset` and `howmany` are two numbers. When returning rows, MySQL will skip the first offset, and only return `howmany`.

Fetch the 20th to 30th countries with their associated codes (sorted by country name) with the 'LIMIT' syntax, in RXML:

```
<sqltable border=1 host="mysql://user:password@localhost/sample"
  query="select name,code from ids order by name limit 20,10">
```

...or with Pike, using the query() function and result selection:

```
object db;
array(mapping(string:mixed)) result;
db=Sql.sql("mysql://user:password@localhost/sample");
result=db->query("select name,code from ids order by name");
if (sizeof(result)>20)
    result=result[20..];
else
    result=({});
if (sizeof(result)>10)
    result=result[..10];
foreach(result,mapping m) {
    write(m->name+"\t"+m->code+"\n");
}
```

The two sizeof()-based conditionals are needed because when slicing arrays, we need to make sure that valid indexes are used, and that the required semantics are respected.

With Pike, using the big_query() function and result selection:

```
object(Sql.sql) db;
object(Sql.sql_result) result;
int j;
db=Sql.sql("mysql://user:password@localhost/sample");
result=db->big_query("select name,code from ids order by name");
for(j=0;j<19 && result->fetch_row();j++)
    ; //empty body, it's all done in the condition
for(j=0;j<11;j++) {
    array row;
    if (!(row=result->fetch_row()))
        break;
    write (row[0]+"\t"+row[1]+"\n"); //row[0] is the name, row[1] is the
code
}
```

5.5 Functions

Whenever a column or a constant can be used in a query definition, a function can be used instead. Functions perform operations on the data, the usual quoting rules applying to their arguments.

The available function and their names names vary wildly from server to server, as does their syntax. We'll introduce here the most important MySQL functions. For further information, consult your server's documentation.

5.5.1   Arithmetic and math functions

```
+, -(unary or binary), / (with infix notation), *
ABS(X)
SIGN(X)
MOD(X Y)
```

modulo, like 'X % Y' in C

```
FLOOR(X)
CEILING(X)
ROUND(X)
```

rounding operators

```
LEAST(X, Y,...)
```

returns the smallest of its arguments

```
GREATEST(X, Y,...)
```

returns the greatest of its arguments

5.5.2   Comparison and logic functions

```
=
```

equality

```
!= or <>
```

dis-equality

```
>, >=, <, <=
IS [NOT] NULL
```

true if the compared value is (not) NULL

```
expr IN (value, ...)
```

true if the expression expr appears in the list

```
NOT or !
OR or ||
AND or &&
```

logic operators

5.5.3   String comparison and operations

```
value LIKE pattern
```

see the Conditions page

```
value REGEXP pattern
```

performs a regular-expression match

```
CONCAT(str1, str2,...)
```

concatenates the arguments

`LENGTH(str)`

returns the length of its argument

`LEFT(str,len)`

return the leftmost len characters

`RIGHT(str,len)`

return the rightmost len characters

`SUBSTRING(string,start_at,length)`

returns length characters starting from position start_at

`TRIM([LEADING|TRAILING|BOTH] FROM string)`

trims leading, trailing or both spaces from string

`LOWER(string)`

returns the string in lower case

`UPPER(string)`

returns the string in upper case

`PASSWORD(string)`

returns a Mysql password that checks against string

`ENCRYPT(string[,salt])`

same as the Unix crypt(3) function. If supplied, 'salt' should be 2 characters long. Otherwise it uses a random salt.

5.5.4 Control flow operators

`IFNULL(expr1,expr2)`

if expr1 is not null, returns it, otherwise it returns expr2

`IF(expr1,expr2,expr3)`

if expr1 is true, returns expr2, else expr3

5.5.5 Date-related functions

`DAYOFWEEK(date)`

returns the weekday index for date (Sunday=1...Saturday=7)

`DAYOFMONTH(date)`

returns the day of the month for date (1..31)

`DAYOFYEAR(date)`

returns the day of the year for date (1..366)

`MONTH(date)`

returns the month for date (1..12)

`YEAR(date)`

returns the year from date (1000..9999)

`HOUR(time), MINUTE(time), SECOND(time)`

time extraction functions

`CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP`

'magic' variables, that are treated like functions. They contain the current date, time, and timestamp respectively.

5.5.6 Miscellaneous functions

`LAST_INSERT_ID()`

> returns the last value automatically generated by an 'AUTO_INCREMENT'-type column

5.5.7 Special functions

These functions are somewhat 'special', in that they have different semantics when used in conjunction with the 'GROUP BY' clause (which is not covered in this manual).

`COUNT([DISTINCT] expr)`

> if 'expr' is a column name, it returns the number of non-null rows returned for that column. If it's an asterisk '*', it gives the number of returned rows. If the DISTINCT keyword is specified, duplicate values are not counted.

`AVG(expr)`

> Returns the average of the columns matched.

`MIN(expr)`

> Returns the least of the columns matched.

`MAX(expr)`

> Returns the greatest of the columns matched.

`SUM(expr)`

> Returns the sum of the columns matched.

> Show the current day of the week:

```
$ mysql -u user -p password sample
[MySQL Monitor]
> select dayofweek(now()) as day;
```

> Count the number of rows in a table:

```
> select count(*) from ids;
```

> Count the number of countries whose name begins by 'i':

```
> select count(*) as number from ids where ids.name like 'I%'
```

5.6 Features Missing from MySQL

What's missing?

There are a few very powerful features ANSI SQL provides that we haven't mentioned, and that we won't go into detail on.

The main excu.. ahem, reason for this is that they're quite powerful and complex and thus out of scope for this manual. Furthermore, they're not supported by all SQL servers. MySQL in particular doesn't support them.

The features we're talking about here are views and sub-queries. When your SQL server of choice supports them, Pike and Roxen CMS can use them.

Also, SQL is designed to support transactions via some special keywords. However, not all servers implement this feature. If you need transactions, you also need an

experienced database administrator to optimize your SQL and your application in general, so you won't find any reference to that here. Sometimes used as a simpler scheme in place of transactions, table locking is available for instance on MySQL. Refer to the MySQL manual for further information on the topic.

6 Data Insertion

6.1 Insertion Syntax

While data extraction queries enforce the relationships between the tables in a database, data insertion queries do not. Data is always inserted into a table, never into a relation.

This fact is reflected in the SQL syntax for an insertion query. There are of course a few variations:

```
INSERT INTO table VALUES (value [, value ...])
```

is the basic version. It only allows to specify all the values in a table row. The values' order is of course relevant: it must correspond to the order the columns were defined in when the table was created.

Sometimes it's preferrable not to specify data for all the columns: data may be unknown, automatically completed by the server (unique IDs, timestamps, ...) or the default value might be acceptable for some columns. This can be obtained with the alternate syntax:

```
INSERT INTO table (column [, column ...]) VALUES (value [, value ...])
```

A value must be supplied for each column specified in the columns-list. Unspecified columns will be assigned the default value or NULL. If no default value is specified and NULL is declared invalid for a column, an error will be thrown when trying to insert.

A particular form of subquery can be used to fill in a table (usually temporary tables). This is the only form of subquery supported by the MySQL database. The syntax is:

```
INSERT INTO table [(column [, column ...])] SELECT ...
```

There are a few limitations for the SELECT query, check your server's of choice manual to know more about them.

An insert-type query doesn't return any results, so you should use SQLQUERY in RXML, or not expect any results if you're using Pike. Also, you have to watch out and Quoting quote the values you're inserting. Program errors and possible security breaches are possible of no proper quoting is used.

6.1.1 Insertion Query with Pike

This program was used to build the sample database, and as such it's hackishly raw. It takes the contents of a file named "country-codes.data" in the current directory. That file has one entry per row, with two tab-separated fields (country code and country name). Those same data are dumped into the sample database.

```
int main () {
  object o=Sql.sql("mysql://user:password@localhost/sample");
  array(string) rows=Stdio.read_file("countries-codes.data")/"\n";
  rows-=({""});
```

```
  foreach (rows,string row) {
    array(string)fields=row/"\t";
    o->query("insert into ids(code,name) values ('"+fields[0]+"','"+
      o->quote(fields[1])+"')");
  }
}
```

6.1.2   Insertion Query with RXML

Performing insertion queries with RXML must be considered with extreme caution:
while it is a great system, it is undoubtably less flexible than the Pike programming
language.

This simple RXML page will allow you to insert a new country - country code entry into
the sample database:

```
<form action="&page.url;" method="post">
Country name: <input name="name"><br>
Country code: <input name="code" maxlength="2"><br>
<input type="submit"><input type="reset">
</form>

<if variable="form.name">
<!-- we're inserting data here -->
<sqlquery host="mysql://user:password@localhost/sample"
  query="INSERT INTO ids (code,name)
         VALUES ('&form.code:sql;','&form.name:sql;')"/>
</if>
```

Notice that while this sample works, and can be used in a development/internal
environment, it is not suited to be used in a production environment: events such as
a duplicate entry will cause uncaught exceptions, which could potentially leak
information such as the database's password or the implementation internals.

See <catch> to address these issues.

7 Using RXML Features with SQL Databases

7.1 The <tablify> Container

This page isn't meant to be a reference for tablify, as it can be found on the <tablify> page.

The `<tablify>` container can be used to make (plain or nice) HTML tables out of formatted text, as well as allowing to perform some operations on the data like sorting.

Of course it is not mandatory to use it to build tables, but it can save some work, especially to build "nice" tables. The `<sqltable>` tag can be used for the same purpose too, but it doesn't have the same flexibility, and it is being slowly phased out, so support for it might be dropped in the future.

Build a table with `<emit>` to print the total area of each known nation:

```
<table border="1">
<tr><th>Country</th><th>Total area</th></tr>
<emit source="sql"
      host="mysql://user:password@localhost/sample"
     query="SELECT name,area_tot FROM ids,countries
            WHERE ids.code=countries.country">
<tr><td>&_.name;</td>
<td>&_.area_tot;</td></tr>
</emit>
</table>
```

Doing the same with tablify:

```
<tablify nice="yes" interactive-sort="yes" size="3"
         titlecolor="white" cellseparator="|">
Country|Total Area
<emit source="sql"
      host="mysql://user:password@localhost/sample"
     query="SELECT name,area_tot FROM ids,countries
            WHERE ids.code=countries.country">
&_.name;|&_.area_tot;</emit>
</tablify>
```

Tablify expects to receive its data in a tabular form, with newline-separated rows of tab-delimited entries. In this case I chose to override the default cell separator because some editors try to translate the tab character to a sequence of spaces. Should you choose to do the same, make sure that your delimiter is a character that does not occur in your dataset.

While I shamelessly used the interactive-sort parameter to tablify, it is not recommendable to use the tablify sorting functions in general, but rather using the SQL "order by" clause for performance reasons.

7.2   The Business Graphics Module

The business graphics module (providing the diagram tag) allows Roxen CMS to build different kind of diagrams on-the-fly. A reference chapter on the module's features (ref: the business graphics module chapter) is available.

We won't duplicate the reference specification, but instead focus on how to use a sql data-source to feed a diagram generation tag.

We'll start off with an example:

Show a graphic documenting the total areas for the known countries:

```
<diagram type=barchart horgrid name="Areas"
  namefont="franklin gothic demi" namesize=25>
<data xnames form=column xnamesvert>
<sqltable ascii host="mysql://user:password@localhost/sample"
          query="select name,area_land,area_tot from ids, countries where
                 ids.code=countries.country order by area_tot desc" />
</data>
<legend separator=|>Total area|Land area</legend>
</diagram>
```

In the example, the data are fed by columns rather than by rows (which is the default for the diagram tag) because SQL modules are better suited for that kind of layout.

The `<sqltable>` tag, together with the ascii parameter, is the most suited system to feed data to a `<diagram>` tag.

You always need to watch out for possible field separator misinterpretation problems: the default field separator (the tab character) and line separator (newline) are not usually found in SQL-obtained data-sources, especially the numeric data used to feed the diagram module. But you must not take this for granted, so make sure to check, and possibly use a different separator and the `<sqloutput>` tag to make sure.

7.3   The <emit> and <sqlquery> Tags

The `<emit>` tag is a plugin-based data management system.

Generally speaking, `<emit>` will iterate through all the data in a dataset such as the result of a SQL query, processing the contents of the tag for each item in the dataset. The source of the dataset is specified in the tag's arguments, along with a few source-dependent parameters. See the chapter on emit in the creator manual for more details. The emit tag allows to take full advantage of the Roxen CMS variable scopes.

Applying this to the case of SQL queries, the dataset is a tabular result, and the items are the result's rows. The source to be used is named "sql", and it takes as additional arguments host (the SQL-URL of the host to be contacted) and query (the SQL query to be executed). Additionally, it accepts the same parameters as the sqloutput tag.

The example in the tablify chapter can be rendered with emit as:

```
<table border=1>
<emit source="sql" host="mysql://user:password@localhost/sample"
  query="select name,area_tot from ids, countries where
  ids.code=countries.country">
<tr><td>&_.name;</td><td>&_.area_tot;</td></tr>
</emit>
</table>
```

Remember: _ is the default scope. Should it be unavailable, or should you want to use it for some other tag, you can use another scope, like this:

```
<table border=1>
<emit source="sql" host="mysql://user:password@localhost/sample"
  scope="queryscope"
  query="select name,area_tot from ids, countries where
  ids.code=countries.country">
<tr><td>&queryscope.name;</td><td>&queryscope.area_tot;</td></tr>
</emit>
</table>
```

The sqlquery tag can be rendered with an empty emit tag

```
<sqlquery host="mysql://user:password@localhost/sample"
  query="insert into foo(bar) values ('gazonk')">
```

can thus be translated into

```
<emit source="sql" host="mysql://user:password@localhost/sample"
  query="insert into foo(bar) values ('gazonk')" />
```

There is no builtin way to emulate the sqltable tag, you'll have to follow the syntax described for the `<tablify>` tag previously.

8   Database Maintenance

8.1   Database Creation

Database creation is not a part of the SQL standard, and the details are very much server-specific. The Pike SQL interface, however, offers two functions as part of the Sql.sql object that can serve for this purpose.

Create a "foo" database:

```
void create_new_database(string dbname)
{
  mixed error;
  object db = Sql.sql("mysql://admin:password@localhost/");
  error = catch {
    db->create_db("newdb");
  };
  if(error)
  {
    werror("Error: "+db->error()+"\n");
    return;
  }
}
```

Delete the "foo" database:

```
void delete_database(string dbname)
{
  object db = Sql.sql("mysql://admin:password@localhost/");
  db->drop_db("newdb");
}
```

Of course the catch {} clause in the first example is overkill here, these operations are really REALLY meant to be used interactively, and so a stack backtrace can be very descriptive and useful.

Most servers provide an SQL syntax to perform this operation. In some cases creating a database is so expensive that an external app is used to perform the operation. When your server supports it via SQL, using SQL is advised. This functions are provided mostly for MiniSQL compatibility (MiniSQL doesn't provide an SQL syntax to create a database).

8.2   Creating Tables

Tables are created via a mostly standard SQL syntax. When a table is declared, the names and types of its columns are specified, possibly along with constraints, default values and other options.

Most databases, however, allow changing a table structure at any time. Be warned that doing so without breaking any constraint might be not trivial. We won't go into details on how to modify a table structure here. You can check your server's SQL reference manual, looking for the keywords "ALTER TABLE".

Also, we won't go into details on referential integrity constraints. If you need them, you also need a skilled database administrator, and explaining them here would be out of scope.

Again, the SQL standard is not well-specified here. While the basic syntax to create a table is standardized, column types are not (except a few). Also, some servers allow defining custom types, further complicating the matter. Finally, the syntax to define constraints is heavily dialectized, save for the most basic functions. Check your server's documentation for further informations.

We'll use the MySQL syntax as reference.

The basic syntax is:

```
CREATE TABLE name (declaration[, declaration ...])
```

The declarations can be columns, keys or indices (see the indices chapter) in various flavors. Let's take a look at a column declaration syntax first: it is

```
column_name column_type [NOT NULL] [DEFAULT value] [AUTO_INCREMENT]
[PRIMARY KEY]
```

The column name can be pretty much anything, as long as it doesn't clash with any reserved word. For simplicity's sake, using short, descriptive names is advised. Dots, spaces and other non-alphabetical characters are forbidden.

If the NOT NULL clause is specified, it poses a constraint on the column, namely that it must be specified (or, in other terms, it can't be NULL). An attempt to insert a row without specifying this value will result in an SQL error and a (Pike or RXML) exception.

If the DEFAULT clause is specified, inserting a row without specifying this column will result in inserting the default value instead. If it's not specified, NULL will be inserted instead (possibly clashing with the NOT NULL condition).

AUTO_INCREMENT is only meaningful for numeric types, and useful only for integer types. Its behavior is like a specialized default value: if NULL is specified as data for the column, then the actual inserted value will be the maximum present value + 1. This is useful for creating unique IDs for the rows in the table.

We'll return on the PRIMARY KEY argument later.

8.2.1   SQL Data Types

All servers should support at least the INTEGER, REAL, CHAR and VARCHAR types. Unluckily, that's about as far as it goes, and there is even no wide-accepted agreement on the semantics of CHAR and VARCHAR.

INTEGER

is what it seems, an (usually 32-bits) integer. It is signed, unless the keyword UNSIGNED (e.g. INTEGER UNSIGNED) is used.

CHAR

is a fixed-length character string. Some servers space-pad it at the end (and use the VARCHAR type for unpadded strings), others don't. MySQL doesn't pad it.

VARCHAR

is a variable-length string. Usually it differs from CHAR in terms of how it is stored on disk: while CHAR values allocate the storage space for the entire field length (and if it's shorter leave it unused), VARCHAR values are usually stored as a (length, value) pair and are packed. This means that they use less space on disk,

but are somewhat slower to access. More importantly, usually VARCHAR values can't be used in indices or keys.

8.2.2 MySQL Data Types

Of course servers provide many more data types. Here are some details on MySQL's types:

TINYINT [UNSIGNED], SMALLINT [UNSIGNED], MEDIUMINT [UNSIGNED], INTEGER [UNSIGNED], BIGING [UNSIGNED]

are respectively 8-, 16-, 24-, 32-, 64-bit wide integers (signed, 2's complement unless the UNSIGNED clause is specified). Notice that while performing internal arithmetic all values are transformed into 64-bit signed integers, so even for BIGINT UNSIGNED (which is theoretically 64-bit wide, no more than 63 bits values should be used.

FLOAT and DOUBLE

are what you can expect them to be (single- and double-precision floating-point numbers).

NUMERIC

(length,decimal) is an unpacked floating-point number. It is stored as a string, one char per digit. If DECIMAL is 0, then the numbers are considered integer, and can't have a decimal part. LENGTH is the size, and must be in the 0-255 range.

DATE, DATETIME, TIME

are date-related types. The legal range for them is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. MySQL uses the "yyyy-mm-dd hh:mm:ss" syntax to display dates, but also understands others. It is however recommended to stick to the default.

TIMESTAMP

is a somewhat magic column-type. It stores a date and a time as a 32-bit UNIX datetime value, thus the legal range is from '1970-01-01 00:00:00' to sometime in 2037. It is magic in that when you perform an INSERT or UPDATE operation on a row and don't specify the value for a TIMESTAMP column, MySQL will fill it for you with the date-time of the operation. Useful for time-stamping operations (hence the name).

CHAR (length) [BINARY]

is a fixed-length string as described above. Padding spaces are not added by MySQL. Comparisons are case-insensitive unless the BINARY keyword is specified. length must be in the 1-255 range. Values longer than the specified length are truncated.

VARCHAR (length) [BINARY]

is a variable-length string. Same arguments as the CHAR type apply.

TINYBLOB, BLOB, MEDIUMBLOB, LONGBLOB

are amorphous storage spaces, long at most (2^8-1), $(2^{16}-1)$, $(2^{24}-1)$ or $(2^{32}-1)$ bytes.

TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT

are the same as BLOBS, save that comparisons between values are case-insensitive.

8.3 Indices

Indices are one of the reasons why RDBMSes are fast when retrieving data: they are built from the data in user-specified columns when rows are inserted into the database and are used when data is selected or retrieved, thus avoiding in most cases the necessity to do a full table scan when performing read operations. Indices cause insertion operations to be slightly slower, but can make data extraction operations and joins orders of magnitude faster. Indices can span multiple columns, and could even include all the columns (although such an index would be of limited use). Usually DBMS allow to define more than one index per table (the maximum number might be constrained).

Keys (or unique indices) can be seen as "a stronger kind of index". A key is an index which is also constrained to be unique: having two rows with the same key in a table is forbidden, for any key defined on the table. The interpretation the various databases give to this concept varies, however. For some (including MySQL), a key is merely an alias for "index". For others, indices can be used to enforce constraints but have no impact on data organization while keys do. For further information on your server's concept of keys, consult its manual.

One key is special, and is named "primary key". The data is usually put in storage in such a way that read operations involving only the primary key are even faster than operation involving keys or indices. It is also usually very slow to update, and is not allowed to contain NULL values.

Unique indices are the way provided by SQL to avoid duplicate rows, defining one that spans all the columns you wish to maintain unique, maybe even all of them. There can be multiple constraints, that can be expressed by defining multiple indices.

The syntax to create an index varies from RDBMS to RDBMS. However, there are two main syntaxes we'll explain here. Consult your server's documentation for details on the syntax it supports.

8.3.1 MySQL Syntax

MySQL has indices and keys definitions inside table creation clauses. The basic syntax is:

```
CREATE TABLE name ( declaration [, declaration ...])
```

where a declaration is either a column declaration, a key declaration or an index declaration. For columns declaration, see the Creating Tables page.

For indices, unique indices and primary key the syntax is respectively:

```
PRIMARY KEY (column [,column...])
UNIQUE INDEX index_name (column [,column...])
INDEX index_name (column [,column...])
```

The names for indices (unique or not) must be unique in a table (no pun intended).

This is the defininion for the "areas" table in the sample database:

```
CREATE TABLE areas (
  id tinyint NOT NULL auto_increment,
  name char(20) NOT NULL,
  PRIMARY KEY (id),
```

```
    UNIQUE INDEX name (name)
)
```

There are two constraints: the area id must be unique, as must the area name. Joins are made on the primary key for efficiency purposes.

8.3.2   Postgres Syntax

With PostgreSQL and other databases indices are seen not as part of a table definition, but are "external" entities attached to a table. They are created by a CREATE clause, whose basic syntax is

```
CREATE [UNIQUE] INDEX ON table (column [, column])
```

Primary keys are defined using the same syntax as MySQL.

The definition above would have been with PostgreSQL:

```
CREATE SEQUENCE areas_seq

CREATE TABLE areas (
  id tinyint NOT NULL DEFAULT NEXTVAL('areas_seq'),
  name char(20) NOT NULL,
  PRIMARY KEY (id)
)

CREATE UNIQUE INDEX unique_area ON areas (name)
```

Notice that recent versions of MySQL (3.22 and later) and PostgreSQL support both syntax styles.

8.3.3   Single-Column Primary Keys

If your table has a primary key spanning over a single column, you can declare it simply appending the "PRIMARY KEY" keyword to the column definition:

```
CREATE TABLE areas (
      id tinyint NOT NULL auto_increment PRIMARY KEY,
      ...
)
```

Notice that in most cases the PRIMARY KEY clause implies the NOT NULL clause.

8.4   Dropping

To delete indices, tables or databases, the DROP command is used in its variations:

To delete an index (where the CREATE INDEX syntax is used), the syntax is:

```
DROP INDEX name
```

To drop a table the syntax is:

```
DROP TABLE name
```

The table, its contents and definition will be deleted from the database irrevocably.

To drop a database altogether (where supported), you can use:

```
DROP DATABASE name
```

The pike SQL-interface provides a specific-purpose function to drop a database: this
is mainly for compatibility with MiniSQL where the operation of dropping a database
is demanded to a specific-purpose API function, named drop_db.

Using SQL:

```
object db = Sql.sql("mysql://admin:pass@localhost");
mixed exception;
exception = catch {
  db->query("DROP DATABASE test");
};
if(exception)
{
  werror("Error while dropping the database: "+db->error()+"\n");
  throw(exception);
}
```

Using the API functions:

```
object db=Sql.sql("msql://admin:pass@localhost");
mixed exception;
exception=catch {
  db->drop_db("test");
};
if(exception)
{
  werror("Error while dropping the database: "+db->error()+"\n");
  throw(exception);
}
```

Notice that I haven't either tried to fetch results (there's no result to fetch anyways)
and the exception handling has been very limited, and for diagnostic purposes only:
these operations are really meant to be used only interactively.