Firebird Conference
Prague 2006

Managing Metadata Changes

Milan Babuškov
http://www.flamerobin.org

About the author

Education:

2001 - B.Sc. In Business Information System Engineering

2003 - M.Sc. In Internet Technology at University of Belgrade

Started to program as a 14 year old, making simple games in BASIC and later assembler on Motorola's 680x0 series CPUs. Programmed in C, Perl and Java. Now writing the code in C++ and PHP. Started to work with Informix at University, after few experiments with Paradox, MySQL and MSSQL, finally switched to Firebird. Starting from 2002, developing various software using Firebird, PHP and Apache.

Developer of open source FBExport and FBCopy tools, for manipulation of data in Firebird databases. In 2003, started a project to build a lightweight cross-platform graphical administration tool for Firebird. The project was later named FlameRobin, and is built entirely with open source tools and libraries.

Hobbies include playing basketball and writing cross-platform computer games, some of them very popular (Njam has over 60000 downloads on sf.net):

http://njam.sourceforge.net

http://abrick.sourceforge.net

http://scalar.sourceforge.net

Born in 1977. Married this year.

Live and work in Subotica, Serbia. Currently employed at small ISV company doing software for retail systems.

Metadata management problem

Development of database applications involves three distinct activities: changing the database schema, updating the data in the database and changing the code that uses the database. The last one is easy to manage: we have version control software for source code, and many ways to distribute executables to customers.

However, during development we all face problem of managing the database metadata in development and productions systems. In development systems many developers can work on their own (sandbox) versions of database, and apply their changes to the master database. When new version is ready for release, changes have to be applied to the customer's database.

There are also developers that work in the field. For example, they take notebook computer and go to the customer's site to fix some problems. During that process, they might need to change database structure. When they get back to the office they need to merge their changes with others. However, in the meantime, some of the developers in the office are making their own changes.

Some of your clients might have more than one sites where software and database is installed. Some may even have dozens of branch offices all around the world. Sometimes it's much easier to have an automated system that would propagate the changes you make at the head office.

All these situations call for some kind of metadata management system.

Possible solutions

Developers have been facing these situations for years, and developed various methods to handle them. Here are some of the many solutions currently in use:

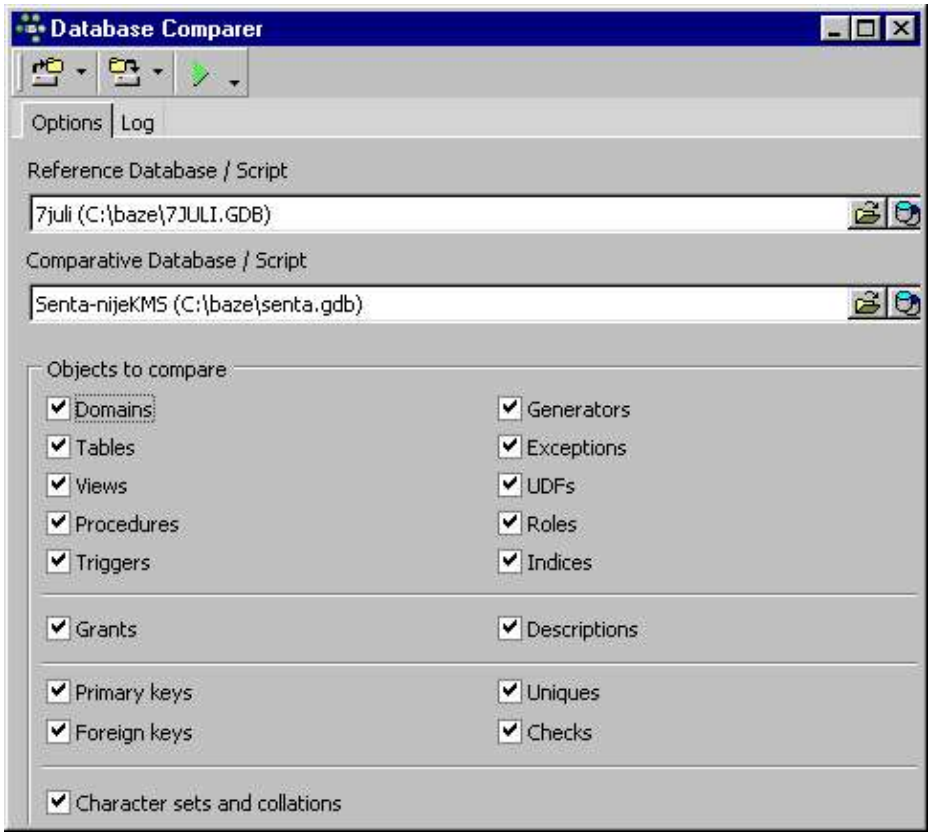
Compare target and source database and do the changes

There are specialized tools like DBComparer that can compare the structure of two databases and dump the script with ALTER, DROP and CREATE statements that change one of the databases to be same as another. Some general administration tools like IBExpert also have this feature. There are two problems with this approach. The main

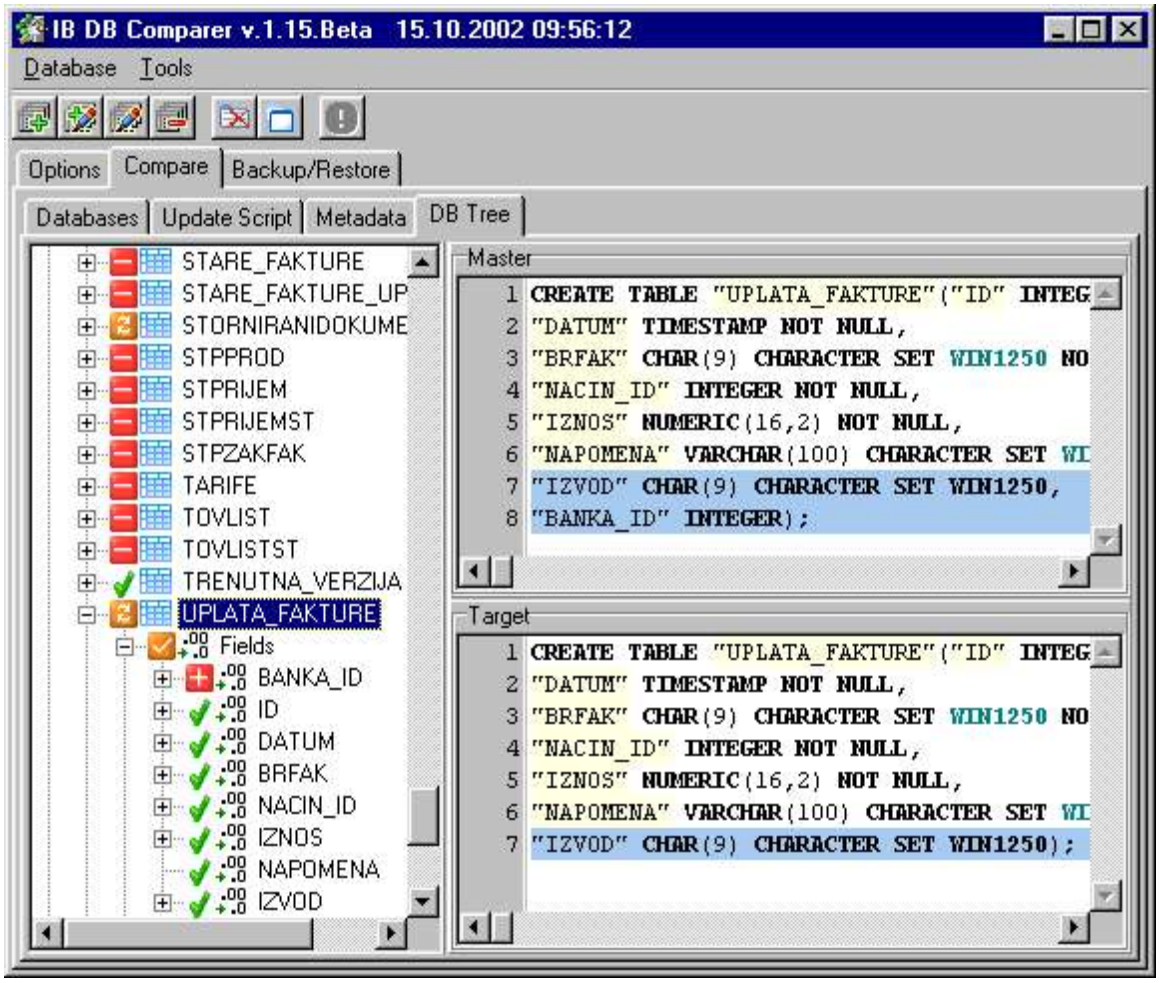
problem is that beside metadata, there is also support-data which is not inserted by the final user, but rather by development team. Those are various helper tables that contain data needed by applications to run properly. When changing the structure, some of those data need to be changed as well, so developers have to remember it. Also, some of the user's data needs to be validated and/or transformed. The second problem is that this process is semi-automatic and requires manual work and concentration.

Create blank database with desired structure and pump the data

There are various approaches to create a blank database. One is to do metadata-only backup (gbak -m) of database, and then restore it. Other is to DELETE FROM all tables and then backup and restore to reduce the size. Third is to re-create database from .sql script each time. The latter one requires that you maintain database creation script while developing, or create it from database using isql -x or some other tool. There are various tools that can pump the data, like IBDBPump on Windows or FBCopy on Linux, Mac or FreeBSD. The problem with this approach is that some columns names might have changed so you need to map them to appropriate ones.



Screenshot 1: Metadata comparer in IBExpert



Screenshot 2: IBDBComparer

However, none of these approaches is fully automatic, and all are error-prone. The main problem is that they neglect the required changes in "supporting" data.

The solution

The most natural way of tracking the changes in metadata is to record them. Later, you can replay them on any other database. In order to do this, we must add a version number to each database structure. To handle it easily, we add a simple table in the database, calling it DATABASE_VERSION. Here's the DDL:

```
CREATE TABLE DATABASE_VERSION
(
      CURRENT_VERSION INTEGER NOT NULL,
      LAST_CHANGE TIMESTAMP NOT NULL
);
```

You can add other fields as well, or name the table differently. For each change you make, increase the version number. For example, suppose you have an empty database, only with DATABASE_VERSION table. To start, set the version to one:

```
INSERT INTO DATABASE_VERSION VALUES (1, current_timestamp);
```

Let's create some table to play with:

```
CREATE TABLE TEST_TABLE
(
      ID INTEGER NOT NULL PRIMARY KEY
);
```

Now, backup and restore the database to a different name. You'll have two identical databases with version one. Let's change one of them:

```
ALTER TABLE TEST_TABLE ADD X INTEGER;
```

Save that SQL statement to some .sql file and call it *version_2.sql* or something like that. After the change is made, increment the version number:

```
UPDATE DATABASE_VERSION SET CURRENT_VERSION = 2;
```

Now, when you wish to upgrade the other database, you would first check for database version:

```
SELECT CURRENT_VERSION FROM DATABASE_VERSION;
```

It returns 1. Now, you see that you have *version_2.sql* and you run it. Thus the other database is also updated to version 2.

You can put multiple changes into a single .sql file, or keep "one statement by file" rule. The later is a better idea as situations may happen when upgrading fails, and database is left between versions and it has to be debugged manually.

With this approach you can easily track the needed changes in data as well. Suppose you add something like this:

```
INSERT INTO INVOICE_TYPES VALUES ('Export');
```

You simply add it to the list of *version_XXX.sql* files and increase the CURRENT_VERSION by one. This way the changes would be propagated to anyone. Of course, you shouldn't log everything, only the changes you wish to be replicated to any other database. You may run some statements or create some test data that is meant only for your sandbox database.

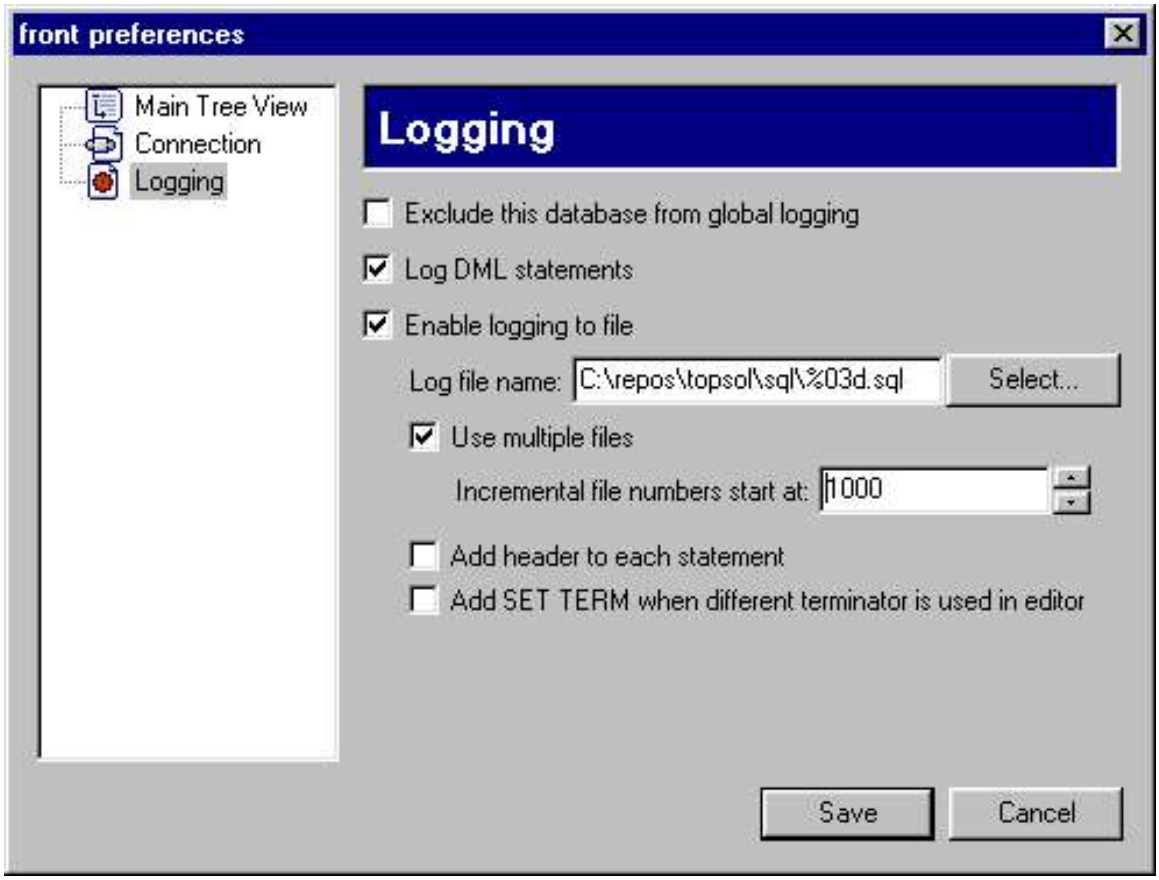
Beside logging changes to .sql files, you could alternatively put them in some database table as well. However, in practice, the solution with files has been proven easier to maintain.

To implement all this, you don't need to start from scratch. You can easily apply to any of your current databases. Just add the DATABASE_VERSION table and set the version number to one. This will be the initial version of the database structure.

Logging changes to sql files

You could do it manually, for example IBExpert is quite usable that way as it has *Copy* button for each script it executes. However, it is much easier to use the administration tool that has built in logging like FlameRobin. To access it from the menu, go to

Database -> Advanced -> Database preferences -> Logging



Screenshot 3: Statement logging in FlameRobin

The example in the screenshot shows *%03d.sql* used as a filename. This means that files would be named *000.sql, 001.sql, 002.sql*, etc. You can set it to be *version_%d.sql*, or whatever you like. The *%d* placeholder follows a standard printf format placeholder syntax from C programming language. This means that you can write *%d* for numbers 1, 2, 3, etc. Write *%0xd* to force output to have at least x digits and be prefixed with zeros if necessary. The path should contain exactly one % format specifier. As you can see, it has option to log DML changes together with DDL changes, so changes you made in data are

also recorded. If you run a lot of DML that isn't meant for production databases, then it is better to turn this option off and add those kind of statements manually.

The main problem is updating the version of current development database after commit. While we could add a feature to run custom update statement supplying version as a parameter, it is not necessary, and perhaps even not a good idea at all. For reasons why, read the section about using repository to track the changes.

As it happens, there are changes you make to both data and metadata during development, that you don't want to propagate. For example, you might add some testing data, or you might create some object and drop it afterwards as you change your mind. FlameRobin (and other tools as well) would log those changes nevertheless. Later, when you remove those scripts, you're left with gaps in number sequence. There are various ways to handle that situation. One is to write the update script in such way to skip the missing files. However, I wouldn't recommend it as it is better to have all the files in sequence, and missing files would mean that something is wrong. It is much better to take and renumber the files to fill in the gaps. For that purpose, here is a small PHP script that does the renumbering.

```
<? //----------------- movesql.php source code -------------------------
if ($argc != 6)
{
  echo "Usage:   movesql.php in_pattern out_pattern start end moveto\n";
  echo "Example: movesql.php log%5d.sql %03d.sql    1     145 346\n";
  echo "         log00001.sql -> 346.sql\n";
  echo "         log00002.sql -> 347.sql\n";
  exit();
}

$inp = $argv[1];
$outp = $argv[2];
$start = $argv[3];
$end = $argv[4];
$moveto = $argv[5];
echo "MOVE: (". sprintf($inp, $start) . " - " . sprintf($inp, $end)
  . ") TO: (". sprintf($outp, $moveto) . " - "
  . sprintf($outp, $moveto + $end - $start) . ")\n";

for ($i = $start; $i <= $end; $i++)
{
  $from = sprintf($inp, $i);
  if (!file_exists($from))
  {
    echo "FILE: $from does not exists, skipping.\n";
    continue;
  }
  $to = sprintf($outp, $moveto++);
  if ($from == $to)
  {
    echo "SAME FILES: $from, skipping\n";
    continue;
  }
  echo "MOVE: $from -> $to\n";
  if (!rename($from, $to))
  {
    echo "ERROR!!!";
    break;
  }
} //----------------- movesql.php source end -------------------------
?>
```

Beside filling the gaps, it can be also used to move a bunch of scripts few numbers up or down in case some other developer has already made changes to the database - see section about using repository to track the changes. For example, let's say to have files *003.sql, 005.sql* and *006.sql* and you wish to fill the gap. Run:

```
php -f movesql.php %03d.sql %03d.sql 3 6 3
```

The result would be renaming *005.sql* to *004.sql* and *006.sql* to *005.sql*. Other example: suppose you have the file *003.sql, 004.sql* and *005.sql* and wish to rename them to *version_1.sql, version_2.sql and version_3.sql.* Run:

```
php -f movesql.php %03d.sql version_%d.sql 3 5 1
```

When you run the *movesql.php* script without any arguments, it displays the usage instructions. Of course, you need to have PHP installed on the system. As you can see, script is quite trivial, so you can easily write your own version in preferred programming language.

Applying changes

There are various techniques to apply the changes to other databases. You could simply open the SQL editor and run scripts one by one, but that is highly inefficient. One of the solutions is to write a program that reads all the *version_XXX.sql* files in a directory and update the database one by one. Here is a simple PHP program that does it, but it is trivial to write your own:

```
<? //------------------ updatedb.php source code ---------------------
error_reporting(0); // we'll do our own error handling
if ($argc == 3)
{
  echo "Connect...";
  if (!ibase_connect($argv[1], 'SYSDBA', $argv[2], 'WIN1250'))
  {
    echo "ERROR: ".ibase_errmsg();
    exit("\n");
  }
  $res = ibase_query('select current_version from database_version');
  if (!$res)
  {
    echo "ERROR: ".ibase_errmsg();
    exit("\n");
  }
  while ($row = ibase_fetch_row($res))
    echo "Version = ".$row[0]."\n";
  exit();
}
if ($argc != 4)
{
  echo "Usage: updatedb.php ";
  echo "database_path sysdba_password [start|SET=version]\n";
  exit();
}
echo "Updating database: ".$argv[1]."\n";

if (strpos($argv[3], '='))
{
  $p = explode('=', $argv[3]);
  if ($p[0] != "SET")
    exit("Error. Unknown option\n");

  echo "Connect...";
  if (!ibase_connect($argv[1], 'SYSDBA', $argv[2], 'WIN1250'))
  {
    echo "ERROR: ".ibase_errmsg();
    exit("\n");
  }
```

```
  if (!ibase_query("update database_version set current_version = "
    .$p[1].", last_change = current_timestamp") || !ibase_commit())
  {
    echo "\nVersion number not written, error:\n\n".ibase_errmsg;
    exit("\n");
  }
}

$i = $argv[3];
while (true)
{
  $fn = sprintf("%04d.sql", $i);
  if (!file_exists($fn))
    break;

  echo "Connect...";
  if (!ibase_connect($argv[1], 'SYSDBA', $argv[2], 'WIN1250'))
  {
    echo "ERROR: ".ibase_errmsg();
    exit("\n");
  }

  echo "Loading script: $fn.";
  $fp = fopen($fn, 'r');
  if (!$fp)
    exit("cannot open file");
  $sql = fread ($fp, filesize ($fn));
  fclose($fp);
  if (trim($sql) == "")
    exit("ERROR: empty script\n");

  echo "...RUNNING...";
  if (!ibase_query($sql))
  {
    echo "ERROR.\n\n".ibase_errmsg();
    exit("\n");
  }

  echo "Commit...";
  if (!ibase_commit())
  {
    echo "ERROR.\n\n".ibase_errmsg();
    exit("\n");
  }

  if (!ibase_query("update database_version set current_version = $i"
    ."., last_change = current_timestamp") || !ibase_commit())
  {
    echo "\nVersion number not written, error:\n\n".ibase_errmsg;
    exit("\n");
  }

  echo "OK.\n";
  ibase_close();
  $i++;
}
echo "Done.\n";   //--------------- update.php source end -----------
?>
```

The *updatedb.php* script has three functions. To check database version, don't supply the last argument:

```
php -f updatedb.php /path/to/database.fdb masterkey
```

Second function is to update database. Suppose your database is at version 3 and you wish to upgrade it from three to whichever is the last version.

```
php -f updatedb.php /path/to/database.fdb masterkey 4
```

The script would stop at first .sql file that isn't available. So if you wish to update to some arbitrary version, temporarily remove the "wanted version+1" file. The third function is to set database version to some desire value. That is usually done on your sandbox databases. For example, to set database to version five, without running any update scripts, run:

```
php -f updatedb.php /path/to/database.fdb masterkey SET=5
```

This just runs: UPDATE DATABASE_VERSION SET CURRENT_VERSION = 5.

As you can see when you run the upgrade, after each statement it commits and disconnects from database. This is important, especially for older versions of Firebird, as some statements can run without errors, but when you try to commit, the error shows up. So, if multiple statements would be stacked up, it would be hard to tell which one is problematic. One of such usual problems is creation of foreign keys when there are other connections to the database. Some of the statements might also report the famous OBJECT IS IN USE error, so the best way is to disconnect and connect again between each statement.

Beside using an external program like *updatedb.php*, you can always write an equivalent piece of code in your own application and let the end-user run it. Or, integrate is into installer of your applications. It is also useful to add a check in your applications: what is the minimal version of database structure it needs to run properly. When application is started it can select the CURRENT_VERSION from DATABASE_VERSION table and not allow user to work if either application or database are not up to date. One of the approaches is also to integrate update scripts with each application, so first user that runs it would trigger the upgrade of database structure. The main problem with such approaches is that they break if you try to add a foreign key or similar task that requires exclusive access.

If your customer has multiple sites and you're doing some kind of replication, it is useful to integrate metadata upgrade scripts into the system. They should happen before replication starts in order to make sure that replicated databases have the same structure.

Things to be careful of

If you're going to use any kind of automated system, there are some things to watch for in order to make the system robust and less error prone. One of the main problems are the unnamed constraints. For example, in one of the examples, I ran a statement:

```
CREATE TABLE TEST_TABLE
(
      ID INTEGER NOT NULL PRIMARY KEY
);
```

The primary key constraint would get an auto-generated name by Firebird, something like INTEG_88. Everything would work fine for a while. Imagine one day your customer has a hard-disk crash and has to restore a backup. While backup is restoring, there is no guarantee that that primary key would get the same constraint name. The problem is now created: if you wish to drop that primary key, you would run:

```
ALTER TABLE TEST_TABLE DROP CONSTRAINT INTEG_88;
```

However, it would fail on customers database. Things like this are improved with each Firebird version. In the meantime, make sure you name your constraints. Also, a good idea is to defer creation of primary and foreign keys to separate SQL statements.

Using version control software

Using the version control systems for database structure makes it easy to resolve conflicting changes. It shows who made the change, when (s)he did it and why. Since changes are plain text files, they can be committed together with application source code within same revision, making the entire change atomic. Therefore, it is recommended that change scripts are kept close to source code files, possibly in one of the project's subdirectories. This makes it easier to track changes of both application code and database structure at the same time.

There are a lot of version control software available, both commercial and open source. Of open source ones, I would highly recommend Subversion, which seems to be the right successor for CVS. Version control systems enable you to track changes that were made and also prevent developers from stepping on each other's toes. Suppose one developer has made changes to the database and moved it from version 2 to version 5. He commits scripts: *version_3, version_4* and *version_5*. Now, another developer did his development in the meantime, and made one change himself, moving from version 2 to version 3 in his sandbox. When he tries to commit the changes, commit would fail, as the file

version_3.sql already exists in repository. At this time, developers have to resolve the conflict manually. If changes are compatible, the other developer would simply move his script to version 6 and then commit it. In case there are a lot of files that need moving, *movesql.php* script mentioned before is quite useful.

Using version control system also helps you remember to update the database version. You should simply keep in mind that each time you successfully commit *version_XXX.sql* scripts, you should run UPDATE DATABASE_VERSION SET CURRENT_VERSION = *[last script number]*; Updating version number when you fetch work of others from repository is not an issue, as *updatedb.php* script increments it automatically after each successful statement. Updating the version only after you commit is a way to be sure what is the version of your local database. If versions would be incremented each time you change the database (but didn't commit to repository), it would create problems if someone else already committed those version numbers with different SQL statements.

Master databases

One of the common dilemmas it whether a master database is needed. The bottom line is that is isn't absolutely required, but it doesn't hurt to have it. Master database is the one that all developers can use to test the applications, but never develop on it. This means that only DDL that is committed to the repository is applied to the master database. Any other DDL changes are not allowed. You can imagine master database as a copy of client's production database installed that your office.

Master databases are good to check that your changes are really going to be applied at customer's server. Developers can mess up their sandbox databases on their computers, so some statements seem valid. Applying those to the master database usually breaks. At that point, tools that compare metadata (mentioned at the beginning) are really useful to track the incompatibilities and fix them.

Going back a version

One of the not so uncommon operations is going back a version. You simply make too many changes and finally see it was a wrong direction and want to revert it. It's easy on your local system, but if you already committed some changes to repository and others have picked it up, then it isn't always easy to go back. Also, sometimes you already have database version X, while customer still has X minus 42, so the problem he's reporting doesn't show up. You need to go back to X minus 42 and see the problem there.

With source code, it is really easy to go back. Simply fetch the revision number X minus 42, and you have all the code, exactly in the state it was back then. With databases, it is not that easy as you only have forward log of the changes. There are three different approaches to this problem:

Store database file in repository

This approach is the simplest to use and maintain. There are two drawbacks though. One is that even development databases tend to grow very big. By big, I mean their size on disk. If you do frequent changes, it means that your repository is going to grow a lot. The second, even more important problem is that this means that developers have to share a common sandbox database, which is really a bad idea. One solution to that problem is to have a master database. The master database would be kept under version control. After changes have been successfully committed to repository and applied to master database, the changed master database file itself would be committed. Main problem with this is that changes are not atomic, but it is probably worth the trouble (in each master database commit, add a comment showing which version of database it is).

Write "reverse" statements

This means that, for each statement, you would write a statement that reverses the change. For example, if statement is:

```
ALTER TABLE TEST_TABLE ADD X INTEGER;
```

the reverse statement would be:

```
ALTER TABLE TEST_TABLE DROP X;
```

For ALTER PROCEDURE, the reverse would be ALTER PROCEDURE that alters to previous procedure code, for DROP VIEW + CREATE VIEW, the opposite would be DROP VIEW + CREATE VIEW with old definition.

There are many problems with this approach as there are many statements that don't have a reverse one. For example, if you DROP a column it isn't enough to recreate it: you need to have all the data, especially if column is defined as NOT NULL. Beside that, these kind of system is hard to maintain as there isn't any tool available that would automatically build reverse statements, i.e. it has to be done manually.

Roll forward from initial version

This is the most bulletproof approach and even if you use one of the above, this should be available as a backup. At the time you start versioning your database, simply backup that initial version and keep it in the repository - but don't ever change it. When you need to get a database version Y, simply copy the initial database to new database file and run update program. It will run all the statements from version one to version Y, and you'll effectively have database version Y at that point.

Conclusion

In order to control such dynamic system such as database structure, developers need a toolset that is fit for the task. Since we're dealing with versioning, the best idea is to use some kind of version control system. In order to do that, we are breaking the changes into smallest possible atomic units: a single SQL statement. Since there exist a lot of version controls systems that work with files, and files are one way to store SQL statements, the solution imposes itself. There might be other, not file-oriented, systems that could be built, but this is currently available and ready for immediate use.

Table of Contents

About the author..... 2
Metadata management problem..... 3
Possible solutions 3
    Compare target and source database and do the changes..... 3
    Create blank database with desired structure and pump the data..... 4
The solution..... 6
    Logging changes to sql files..... 8
    Applying changes..... 11
    Things to be careful of..... 14
Using version control software..... 14
Master databases..... 15
Going back a version..... 15
    Store database file in repository..... 16
    Write "reverse" statements..... 16
    Roll forward from initial version..... 17
Conclusion..... 17