# WEB APPLICATION MANAGEMENT: IMPLEMENTING A DYNAMIC DATABASE UPGRADE MODEL

Richard Wilson [1] & Daniel Lowes [2]

[1] Dept. of Computer Science and Software Engineering, University of Melbourne (Australia)
[2] Dept. of Computer Science, University of Pretoria (South Africa)

ABSTRACT: Web-based applications play a major role in the development and deployment of online services. Most web applications use databases as their primary data storage mechanism, which not only provide well-developed routines for data management, but also can be accessed via common interfaces such as Microsoft's OLEDB and Sun's JDBC. Furthermore, most web applications are not static; they go through numerous generations of code. During that period of incremental updating, it is common for the supporting database(s) to be altered repeatedly to support new features or to improve application performance. This paper will propose a kind of "version control" system for online databases, using a .NET-based implementation that is database-independent, allows up- or down-grading to and from any version in an efficient manner, and is logically unbounded in the number of operations it can perform and of which it can keep track. The system will also allow the construction of a complete database for any given version of an application.

## INTRODUCTION

Web-based applications play a major role in the development and deployment of online services. Due to their unique medium of operation, they also have unique requirements when it comes to data storage. Most web applications use databases as their primary data storage mechanism, which not only provide well-developed routines for data management, but also can be accessed via common interfaces such as Microsoft's OLEDB and Sun's JDBC.

Furthermore, most web applications are not static; they go through numerous generations of code, until finally the program reaches the end of its viable lifetime and is retired in favour of a newer and better architecture. During that period of incremental updating, it is common for the supporting database(s) to be altered repeatedly to support new features or to improve application performance. Despite being a vital part of the development process, this editing of the database can be extremely time-consuming.

Scripts (usually employing some implementation of SQL, the standard language for accessing databases) must be created to perform the editing remotely; the scripts must then be uploaded and run. If there are errors, they must be trapped at run-time, the code edited, and the entire procedure run once again. If multiple database types are supported, then multiple scripts must be created. This is a cumbersome process, and one that is highly error-prone, especially in the later phases of program development.

For instance: assume a theoretical program *P*. *P* is currently on version 0.10, after ten updates. In every other update, the database was changed in some way. Thus, there have been five database alterations. *P* also supports four different database engines, from different vendors. These vendors each implement the ANSI-92 SQL standard in slightly differing ways, requiring that *P* have four different SQL strings for every database operation performed. Thus, since five different changes have been made, at least 20 different SQL strings will be required. This is assuming that each change only requires the execution of a single SQL statement, of course; it is more than likely that two, three or even more strings will be needed for each change. Keeping track of such changes is difficult, even using conventional version control tools, since most are not designed to handle such situations. Add to this the complexity of supporting upgrading from ANY prior version of a database, and it can be seen

that such database upgrade scripts are certainly a less than optimal method for keeping the program database structure up-to-date with the program code.

This paper will propose a kind of “version control” system for online databases, using a Microsoft .NET-based implementation that is database-independent, allows up- or down-grading to and from any version in an efficient manner, and is logically unbounded in the number of operations it can perform and of which it can keep track. The system will also allow the construction of a complete database for any given version of an application.

## DETAILS OF THE MODEL

I will first examine the criteria that such a system must possess, before moving on to the details of a proposed implementation, namely using Microsoft .NET and XML. While there are a number of theoretical solutions to this problem, the ideal implementation of this "Web Upgrade Model" should satisfy a number of criteria:

Firstly, the implementation should be database-independent. This implies not only that the program will be able to upgrade to any one of a number of database types, but that it can do so using a single "upgrade script" – that is, a file provided as input to the upgrading system. How it achieves this is will be dealt with later; it will most likely involve parsing a general keyset into provider-specific SQL. The program can specify a supported database set (e.g. “Engine A from company X and engine B from company Y”), and/or a supported database access type set (e.g. "Any database that can be accessed via OleDB or ODBC").

Secondly, the upgrade program should provide services for upgrading from any prior version of the program to the current version. This, coupled with structure maintenance, is the primary goal of the implementation. The requirements in this area are similar to those in any version control system: for example, it should be possible for a user, currently running any version of the program, to run the latest upgrade script and be sure that their database is now valid for the latest version of the program.

Thirdly, the system should sacrifice data in order to achieve the desired database structure. If a field in a database is to be modified, but modification will lead to data loss, then the program should continue. Correct database format maintenance is the primary goal of the system; anything else is secondary. To overcome the annoying problem of possible data loss, the implementation can create a subsystem which allows database data to be backed up prior to a database change, in case of loss, and then restored (where possible) after the change has been made. This would likely be implemented as a series of post-operation notices to the user, with which they are told what data was lost, and the user must then choose an appropriate course of action (this is similar to revision conflict resolution in other version control systems). While this backup sub-system will necessarily consume additional processing time and memory, its importance is such that it should form a part of the Web Upgrade Model implementation.

Finally, the implementation should be logically unbounded. That is, there should be no theoretical limit on the number of operations that can be performed by the system. For performance reasons, of course, as few operations as possible should be performed. The only type of operation whose support is mandatory is one that allows the creation (and subsequent alteration, should the fields be modified) of tables. Optional operation types include, but are not limited to, data insertion, data backup, data exporting and data swapping. Ideally, all database operations should be supported, but this would only take place in a highly generalised implementation, or in a context in which such routines are regularly used.

These four principles are by no means all of the features that such a system can have: they are merely the basic requirements that are necessary to define the boundaries of the system in such a way that it can be viewed as a holistic entity. Before moving onto our own implementation, let us first briefly deal with the limited previous work in this field.

## PREVIOUS/RELATED WORK

The concept being described here is a unique one. While there have been attempts to create version control systems for databases in the past, they usually take one of two forms. Either the company exports their database structure to SQL file, and then uses those in a conventional version control

system (such as CVS or SourceSafe), or the company writes their own version control wrappers and embeds them in whatever program they are producing.

The first of these solutions is cumbersome and limited: conventional version controls systems pay attention only to the script as a whole, and this makes certain functionality such as downgrading, rebuilding and repairing very difficult to implement efficiently. The second solution is more effective, but obviously non-portable: having to re-write code when creating a new application, to perform a common task, violates one of the primary principles of software engineering.

There are also version control systems available for purchase for various individual database types, but none provide the overwhelming flexibility of the mechanism described here. Furthermore, these implementations are commercial software, requiring users to purchase licenses should they wish to use the software. More importantly, these other implementations are not specifically geared towards online interaction, and are often themselves non-portable, being written with a particular combination of code and database in mind, eg version control for an MS Access database using Visual Basic 6.

IMPLEMENTATION

With this in mind, let us now move on to describing our implementation of the system, with the working title *Structurer*, which is to be implemented as the upgrade and maintenance mechanism for some ASP.NET web application suite.

*Structurer* makes use of the .NET Framework's powerful and extensive XML integration. The scripts for the system are written in XML (or DBML, which stands for Database Markup Language, and is simply a way of describing XML scripts written specifically for programs such as *Structurer*), with the system parser currently in C#. The code is compiled as a .NET assembly, which can then be referenced in any .NET project, no matter what code language is being used. One of the biggest advantages to the system is that it can be ported literally directly to any .NET application (along with a very few methods for database support) and will function precisely the same. This allows the creation of a client interface for the system, perhaps even a design GUI for simple management. Such extensibility and compatibility is highly desirable, especially in the context of a web-based application, since it makes it far easier to convert the application to another language if needed. The scripts can remain unchanged, and can even be developed for use in one language, and used on another. XML is a worldwide standard, which means that you can be sure the schema you define once, will be valid for any platform and any language.

How does *Structurer* fulfil the implementation criteria mentioned earlier? In terms of being database independent, *Structurer* is able to construct relevant SQL statements for any provider for which it has rules. These rules take the form of XML files that specify limitations and particular requirements for any given provider, for any given SQL statement. For example, a MySQL rule-set would specify the use of the "LIMIT" keyword in the "SELECT" statement, whereas a Transact-SQL rule-set would instead describe the "TOP" keyword (among other things) to achieve the same result. This obviates the need for complex conditional code, and allows the system to operate more efficiently, as rule-sets can inherit from each other. For example, a particular type of MySQL database may have different requirements when creating fields in a table. It is also particularly simple to update the rule-sets should any changes to the language specification be made. In the event that *Structurer* is asked to handle a database about which it knows nothing, an error will naturally be reported.

*Structurer* has implemented the actual upgrade mechanism by means of version numbers. Every field in the database was originally created, or last modified, at a specific version number. This version number is associated with the field as an attribute in the XML file. The parser, when processing the file, searches for the version numbers. Each script is assigned a specific version number, and the database version number is also known. If these two numbers differ, then the script needs to be run for the current database. The parser is initialised, and processing begins. The following is a slightly simplified version of the algorithm performed (actual code omitted for size reasons):

```
foreach valid table in DBMLFile {
      Process the list of fields;
      if (!tableExists) createTable;
      Delete all fields marked for deletion;
      foreach remaining field {
```

```
            if (!fieldExists) createField;
            else modifyField;
      }
      update the database version field;
}
```

Processing the list of fields involves determining the fields that are marked for removal, and checking the validity of the entries for the fields to be created or modified. If the table needs to be created, the appropriate SQL for each field is determined, and the table is written to the database. Deleting fields marked as such is a simple process, as all the fields were determined in the processing stage. When it comes to modifying existing fields, or creating new ones, the latest field in the set of those present in the DBML file is selected; this is because a history of previous fields is maintained in the DBML file, for the purposes of efficient downgrading and repairing.

Extensive use is made of sophisticated XPath syntax for avoid unnecessary iterations and loops through node lists. Since this implementation is being developed under the GNU GPL, the actual code is available: the URL will be provided at the end of the paper.

While table creation and modification is one of the main purposes of *Structurer*, there are a number of other operations that are important. Trivial operations such as INSERT and DELETE can be specified in a single entry in the DBML file, including such qualifiers as only deleting rows which fulfil certain criteria. Useful non-trivial operations include data importing and exporting, data moving, and stored procedure execution.

While it is all very well to modify existing fields in the database, the question of lost data can pose a problem. It was mentioned earlier that, given that the primary aim of the Web Upgrade Model is to maintain database structure, actual data can be sacrificed in order to ensure a field conforms to its required type and/or size. However, it may not always be desirable to discard “extraneous” data. In these cases, *Structurer* can be asked to first make a backup of the data in a table to be modified: this can be done simply by using the provider’s mechanism to dump the recordset either to disk (an XML file) or to an in-memory structure (such as a DataSet in .NET). Then, once the modifications are completed (any lost data is marked as such while being parsed), and the user is presented with a list of conflicts (for unattended execution, these would be written to a log file for later action). The user can then either rollback the entire process, or make decisions on what to do with the data based on its context.

The system as described here is indeed logically unbounded: given that the parser will usually use only one connection, and at most two other temporary connections in highly complex operations, memory usage is unlikely to be a problem. Even when backing up data before processing, the time taken will be acceptable (except perhaps for very large databases – there it would probably be best to have a dedicated backup process before attempting any upgrade that is likely to affect a significant amount of the data).

## CONCLUSION

It thus becomes clear that, as in so many other situations, automation is the most efficient way of handling database upgrading, whether online or offline. An implementation of the Web Upgrade Model described here can considerably simplify the process of “patching” or upgrading a web application. Our particular implementation, *Structurer*, is currently in the advanced stages of development, and will most likely we available (under an open source license) in the next couple of months. We believe that there is immense scope for advancement in the realm of web applications, not only in terms of user interfaces and interactivity, but even more so in terms of support code and functionality. The advent of concepts such as .NET and the Intermediate Language, and all they entail, can go a long way towards achieving these advances - advances which include technologies such as the Web Upgrade Model discussed here.

## REFERENCES

*Access Version Control* [cited 8 September 2003]. Available from http://www.neptune400.co.uk

*Database Version Control* [cited 8 September 2003]. Available from http://dbforums.com/arch/230/2002/10/622830

*The Database Version Control Utility* [cited 8 September 2003]. Available from http://www.crescentbloom.com/I/M/B/239.htm