DATABASE VERSION CONTROL
A Software Configuration Management Approach to Database Version Control

Stephen Mc Kearney
Datatrust Software, Bournemouth, U.K.

Konstantina Lepinioti
School of Design, Engineering and Computing, Bournemouth University, Talbot Campus, Bournemouth, U.K.

Keywords: Databases, database schema, version control, source code control, change management.

Abstract: This paper introduces a database configuration management tool, called *DBVersion*, that provides database developers with many of the benefits of source code control systems and integrates with software configuration tools such as Subversion. While there has been a lot of research in software configuration management and source code control, little of the work has investigated database configuration issues. *DBVersion*'s main contribution is to allow database developers to use working practices such as version control, branching and concurrent working that have proved successful in traditional software development.

1 INTRODUCTION

Database refactoring has been proposed as one approach to making database design more agile (Ambler and Sadalage, 2006). An important part of software refactoring is tracking changes and being able to undo actions easily. However, while there has been a lot of research in software configuration management and source code control (Conradi and Westfechtel, 1998), little of the work has investigated database configuration issues. This paper introduces a database configuration management tool, called *DBVersion*, that provides database developers with many of the benefits of source code control systems and integrates with software configuration tools such as Subversion (Pilato et al., 2004).

1.1 Current Approaches

Two approaches to database configuration management are common. In the first approach, database comparison software, such as DBGhost (Innovartis, 2008) or Oracle Change Management (Oracle, 2007), compares two database instances and generates a list of differences as an update script that can be applied to the older database. This approach has been used to introduce version control into database-centric application development (Ploski et al., 2007). However, not all changes can be identified by comparing schemas, for example, moving an attribute may be recorded as more than one change rather than as a single logical change. In the second approach, the developer writes change scripts to update the database and may keep different change scripts for each modification made. This is a popular but more informal approach that is simple to implement and can be integrated with existing software configuration tools. However, when working with the database, developers do not benefit from many aspects of version control such as reverting changes or concurrent working. Also, change scripts tend to be database specific, which limits their use to one database management system (DBMS).

There are a small number of software tools that take a more structured approach to the problem. LiquiBase (LiquiBase, 2008), for example, manages change scripts as XML files. Each change is implemented using a database independent language and so can be applied to different DBMSs. This is important as there are many different database systems and developers can often be working on one system but targeting another. LiquiBase supports basic version control processes such as reverting changes and, because it uses text files, it can be easily integrated into existing source control systems. We believe our approach is more closely integrated with standard de-

Mc Kearney S. and Lepinioti K. (2008).
DATABASE VERSION CONTROL - A Software Configuration Management Approach to Database Version Control.
In *Proceedings of the Third International Conference on Software and Data Technologies - ISDM/ABF*, pages 81-87
DOI: 10.5220/0001889000810087

velopment approaches as a DBVersion repository is not only stored in the source code repository but DBVersion uses the source code system to help manage versioning, branching and conflicts. We have also implemented a sandbox style of development that allows the developer to checkout and manage a version of the database in a similar manner to traditional source code sandboxes (Vesperman, 2006).

1.2 Benefits of Configuration Management

Software configuration management has been part of software development best practice for many years. Version control tools such as CVS (Vesperman, 2006), Subversion (Pilato et al., 2004) and Perforce (Wingerd, 2005) provide sophisticated support for managing changes, versions and branches. These tools support concurrent working and conflict resolution using sandboxed workspaces and automated merging. Version control systems store a complete history of the software which is invaluable for change control and bug tracking. In addition, they are well integrated with issue tracking and requirements collection systems.

In application development, changes to the source code can have corresponding changes that must be applied to the database schema. When a new software version is released, it is important to apply the correct sequence of changes to the live database. This can often involve writing complex data scripts that must update the schema and the data. A database configuration management system would automatically generate upgrade and downgrade scripts from the history of changes recorded during the development process.

1.3 Software Configuration Management Concepts

In the following sections, we shall review the components of a standard software configuration model upon which *DBVersion* is designed (Conradi and Westfechtel, 1998).

1.3.1 Repository and Workspace

The repository is a complete history of all objects under version control (for example, source code files). When a developer wishes to make changes to the software, they check-out a version of the source code into a local workspace containing all version controlled objects. Workspaces allow developers to work independently of each other. When the developer has finished a set of changes, they commit the workspace back to the repository.

Conflicts occur when an object that is modified in the workspace has also been modified and committed by another developer. In this case, the version control system tries to merge the second developer's changes into the workspace. When the workspace contains conflicts, it is the responsibility of the developer to resolve the problems before trying to commit their changes to the repository. The repository should contain an unconflicted version of the system and the developer can revert to a clean version of the software at any time.

1.3.2 Versions, Revisions and Change Sets

How a version control system manages changes between versions depends on the version control model used. Subversion assigns a single revision number to each set of changes committed to the repository. At each commit point the system calculates the set of changes that have occurred since the last commit. Each revision consists of a set of changes that can be applied to the previous revision to produce the current revision. For source code that is stored in text files a set of changes is represented as a series of additions and deletions to the previous version of the file. For efficiency reasons, the latest version of a file is stored in the repository with previous versions stored as reverse deltas (Tichy, 1985).

1.3.3 Branching and Merging

An important feature of version control systems is the ability to create separate development branches. For example, developers are often called upon to fix problems with a previous version of the software and to do so without changing the current version. Projects start with a single line of development called the *trunk* but the developer can create a branch of the trunk that sits alongside the main development (see figure 1). Changes can be made to a branch without affecting the trunk. This model allows different versions of the software to be developed without interfering with each other.

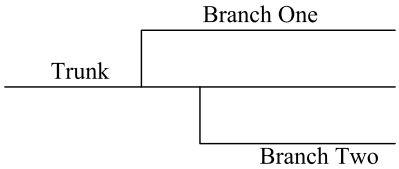

Figure 1: Trunk and Branches.

An important part of branching is the ability to merge changes from one branch into another (see figure 2). For example, after fixing a bug in version 1 of an application it may be necessary to merge the fix into version 2. Ideally, this process should be performed automatically or with the minimum of user intervention. Merging is an important part of concurrent development processes and is particularly important for large teams.

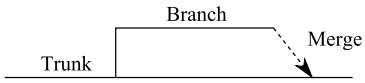

Figure 2: Merging Branches.

2 DBVERSION

In this section, we explain how we applied software configuration concepts to database configuration management in the tool *DBVersion*. The benefits of this approach include better concurrent working, improved history tracking and the ability to use advanced development techniques such as branching and merging. The current implementation uses Subversion but the approach can be applied to similar systems.

2.1 Tracking Changes

There are two approaches to tracking change in a database schema. The first approach is to track changes between versions by comparing a previous version of the schema with a later version. This is a state-based approach (Conradi and Westfechtel, 1998) that builds a list of differences and can be used to generate a script which converts one database schema to another. The state-based approach is used by systems such as DBGhost (Innovartis, 2008) and Oracle Change Management (Oracle, 2007). However, although differences can be identified, it is not always possible to correctly describe the exact change that occurred. For instance, renaming a table attribute will not necessarily be tracked as a `rename` operation but as two operations: `drop` and `create`.

Instead of comparing schema versions, *DBVersion* tracks individual schema modification commands. Changes to the current version are written in a script file and applied to the database using the `apply` command. This approach is similar to the `alter` command in standard SQL and has been used in Ruby on Rails Database Migrations (Thomas et al., 2005). Tracking individual changes has the advantage that more complex database refactorings can be implemented. For example, the `move-attribute` command moves an attribute from one table to another based on matching primary/foreign key values.

```
dbv = DBVersion.new('test.dbv')
DBI.connect('test', 'u', 'p') do | dbh |
  dbv.apply(dbh) do
    create_table :t4, :a1, 'varchar', 10
    add_attribute :t4, :a2, 'integer'
    add_attribute :t4, :a3, 'varchar', 20
  end
end
```

Figure 3: Database Change Script.

The `apply` command reverts any changes made to the database by a previous `apply` so that the change script is always run on a clean version of the database. Figure 3 shows a simple database change script.

2.2 Database Workspace

A database workspace is contained in the Subversion workspace and provides a sandbox environment for making schema changes. It consists of three components:

Schema Descriptor File. A schema descriptor file contains the commands to create one version or branch of the database schema from an empty database instance. The schema descriptor contains a sequence of change sets each of which contains a sequence of schema modification (or refactoring) commands.

Database Instance. The database instance contains the current version of the database schema and is used to test the database. The *DBVersion* `synchronize` command modifies the database instance to match the version of the database contained in the schema descriptor file. If necessary, the `synchronize` command will (i) rollback a newer version of the database, (ii) rollforward an older version of the database, or (iii) switch one schema branch to another.

Change Script. The change script is the set of schema modification commands to be applied to the current database version as described in section 2.1. The database instance is modified using the *DBVersion* `apply` command that also updates the workspace schema descriptor file if the changes are successful (see section 2.1).

Changes to the database schema can be committed to the Subversion repository using the standard Subversion `commit` command or reverted using the

`revert` command.

2.3 Database Repository

The database repository consists of one or more schema descriptor files with each descriptor representing one branch of the database. The database repository is stored in a version control system such as Subversion where schema descriptors can be branched and merged alongside the source code. *DBVersion* understands the Subversion command interface and manages the database repository as part of the Subversion repository. *DBVersion* uses the Subversion `status` and `info` commands to track the status of the database repository. When *DBVersion* is asked to perform any actions on the repository, it can identify whether the repository is unchanged, has been modified or is in conflict.

2.4 Concurrent Changes

An important function of a version control system is the ability to merge other users' changes into the current workspace and resolve any conflicts that might occur. In *DBVersion*, we have adopted a simple method of merging that outputs the set of changes that have to be merged and allows the developer to resolve any conflicts the merge may cause.

There are two circumstances under which different development paths have to be combined: (i) merging changes on the current workspace branch that have been made by another developer, called *updating* and (ii) merging changes on another branch into the current workspace branch, called *merging*.

2.4.1 Updating a Branch

DBVersion uses Subversion to identify merges. The *DBVersion* repository is stored as a single binary file in the Subversion repository. When the developer tries to commit a set of changes to the repository, a check is made to see if the file has been changed. The alternatives are:

No Changes in the Source Code Repository. If no-one has committed changes in the current branch then the *DBVersion* repository can be committed without conflict.

Changes in the Source Code Repository. If someone has committed changes in the current branch then the Subversion will produce a conflict. Because the *DBVersion* repository is a binary file, Subversion will not attempt to resolve the conflict but will generate two additional files for the conflicted file `test.dbv` labeled `test.dbv.merge-left.rXXX` and `test.dbv.merge-right.rYYY` where XXX is the revision of the database before the latest changes and YYY is the revision of the database after the latest changes.

The next step depends on the developer but the normal process would be:

1. Execute the *DBVersion* `resolve` command on the conflicted *DBVersion* repository. This will rollback the developer's current changes and rollforward the database so that it matches the latest database revision.
2. Apply the developer's change script to the newly updated version of the database instance and fix any problems or issues that occur. For example, the update may have deleted an attribute that the developer assumes still exists. This process can be done by editing the change script and reapplying it.
3. Commit the current workspace as a new revision.

2.4.2 Merging From a Different Branch

In this scenario, the main development branch, b_1, has been branched at some stage in the past and another developer has been making changes on the alternative branch, b_2. At some point the changes on branch b_2 must be pulled into the branch b_1 and merged into the final version of the database.

Merging changes made on branch b_2 into another branch b_1 requires taking each of the change sets from branch b_2 and adding them to branch b_1. *DBVersion* does not attempt to identify conflicts that might occur when changes from b_2 are added to b_1. Instead, the DBMS itself is used to identify problems when the merged changes are applied to the database instance. *DBVersion* writes a *change script* for branch b_1 that contains all the changes from b_2. This new *change script* can be applied to the database instance and any errors identified and fixed. When the developer is happy that the new *change script* is valid, they commit the changes to branch b_1.

As with updating, merging uses the source code system to process the merge. The developer requests Subversion to merge a different branch into the current one and, because the *DBVersion* repository is a binary file, two files are created, `test.dbv.merge-left` and `test.dbv.merge-right` representing the original version of the database and the version of the database that is to be merged into the current version.

The merge process is as follows:

1. Execute the Subversion merge command to merge one Subversion branch into another. In the case

of a *DBVersion* repository, this will produce two additional files. The files represent the two sides of the merge process, the right hand side and the left hand side (Pilato et al., 2004).

2. Execute the *DBVersion* `merge` command. This takes the right-hand side of the merge and compares it to the base and then compares the left-hand side to the base. It extracts the set of changes that are in either the right-hand side of the merge or the left-hand side of the merge. These changes are applied to the current branch to bring it up-to-date with the merged branch. *DBVersion* writes out a merge script that describes the sequence of changes to be applied to the target branch.
3. Execute the *DBVersion* `apply` command to apply the new merged change script to the current database instance. This rolls back the current set of changes and applies the merged set of changes.
4. Fix any conflicts that occurred trying to run the merged script. As with the update, the merge may conflict with the developer's script. For example, the developer may have changed an attribute that the merged branch deleted.
5. Commit the new database workspace to the *DBVersion* repository and then commit the Subversion repository.

Note that, as with source code versioning, the merge happens in the database workspace and the *DBVersion* schema descriptor is only updated if the `apply` command is successful. This means that the repository cannot contain an invalid version of the database.

3 DATA MIGRATION

One of the issues raised by applying version control principles to database schema configuration is how to migrate the data between versions. This section discusses some of these issues and introduces the approach used in *DBVersion*.

3.1 Migration Problems

For development and test databases, data migration is a convenience that can make working with the database easier. For production databases data migration is a necessary component of any schema version control solution. The most common method of data migration is to write dedicated software to perform the upgrade. These programs often use a form of "dump and load" between versions. For example, each version of the Subversion software can read dump files produced by any previous version of the software (Pilato et al., 2004). This is an interesting example of a source control system trying to manage a data versioning problem.

A recent development is the Ruby on Rails Framework's Database Migrations that formalise the process of migrating one version of a database to another (Thomas et al., 2005). Ruby on Rails Database Migrations is a simple but powerful form of schema version control and, because it allows almost any algorithm to be executed during the up and down migration phases, it can support data instance migration as well.

The approach used in *DBVersion* is based on database refactoring (Ambler and Sadalage, 2006). Database refactorings describe specific database changes and provide developers with a library of changes they can perform. The main advantage of refactoring is that the results are well known and can be easily tested. The LiquiBase (LiquiBase, 2008) database version control system uses simple refactorings to describe schema changes. It is our proposal to use refactoring to control changes to both the schema and data.

3.2 Schema Modifications and Refactorings

Schema modification refers to the process of adding, changing or removing elements in the database schema. Simple modifications include:

- Create relation
- Drop relation
- Add attribute
- Delete attribute

These modifications have a direct correspondence to standard relational database modification commands such as the SQL ALTER commands.

During the software development process, a single change in the software may require a number of modifications to be made to the database, for example, moving an attribute from one entity to another requires creating a new attribute in a relation, moving the data from the original attribute to the new attribute and, finally, dropping the original attribute. Many of these changes can be expressed as higher level refactorings.

Refactoring is the process of changing the structure of a program while maintaining the same result from the algorithm. In database refactoring, the structure of the data is changed to improve the database design. Refactorings can be expressed as sequences

of schema modification commands and in many cases the data can be migrated appropriately. However, more complex schema modifications can cause data migration problems.

In the current work, we have limited our interest to the set of refactorings necessary to normalise a relational database. For example, some of the refactorings that we currently offer include:

- Convert an attribute to a table
- Convert a table to an attribute
- Extract dependent attributes to a new table
- Move attributes to an existing table
- Merge two tables
- Split a table

Each refactoring is defined in terms of two operations: (i) *rollforward*, which applies the refactoring to an existing version, and (ii) *rollback*, which reverses the refactoring process. Of course, some refactorings lose data when they are applied and reversing them does not reconstruct the data.

DBVersion is implemented in Ruby and refactoring operations are simple Ruby plugins, which makes it easy to add new refactorings. A refactoring can use any sequence of SQL statements but it is important that each *rollforward* operation has a corresponding *rollback* operation. As figure 3 shows, the change scripts are also written in Ruby which isolates the developer from database-specific SQL code embedded in the plugins.

Database researchers have investigated similar problems while studying how to map a given database to an alternative schema, called Schema Mapping (Yan et al., 2001). We are currently investigating the best combination of schema modifications and refactorings for different development scenarios.

3.3 Database Releases

The final part of *DBVersion* is the release management process. Unlike source control systems, in which extracting and compiling a version of the source code produces a version of the software, database versioning is more akin to patching source code. A database version control system must generate a patch to an existing database.

Our current solution to this problem is to distribute *DBVersion* with an application upgrade. When the `synchronise` command is executed, *DBVersion* will identify the current version of the database instance and generate the correct database script to upgrade the database to the current version.

4 RELATED WORK

As discussed, there are a small number of software products that help to manage database changes, for example, LiquiBase (LiquiBase, 2008) and DBGhost (Innovartis, 2008). None of these systems is widely used. Compared to these systems, *DBVersion*'s main contribution is to provide a closer integration with source control systems and to support database development practices that are similar to traditional software development good practice.

There are three database research areas that deal with database configuration issues (Roddick, 1995): (i) schema modification, which allows changes to existing database instances, (ii) schema evolution, which supports changes the schema without losing any data, and (iii) schema versioning, which allows access to previous versions of the database schema and data.

Although these techniques deal with changing the database schema, they focus on managing change within live databases rather than tracking change during the development process. For example, the Microsoft Repository supports versioning in an SQL Server database (Bergstraesser et al., 1999) with some performance and storage costs. Similar approaches have been applied in object database systems that must persist data objects and also manage different versions of those objects as the code changes (Sciore, 1994; db4objects Inc., 2008). However, most object databases do not provide the developer with a history of changes.

Temporal databases record a time dimension with data and track (i) the real world time of an event and (ii) the transaction time in the database (Conradi and Westfechtel, 1998). Temporal databases do not handle schema versioning or evolution and so do not provide support for database developers.

5 CONCLUSIONS

In this paper, we have described the problem of database version control. We have distinguished database version control from other database-oriented methods such as schema modification, schema evolution and schema versioning by focusing on developing a method that integrates well with standard software development processes and tools. The system we have described, *DBVersion*, is an experimental prototype that integrates with the Subversion source control system. Close integration with existing source control systems is one of the characteristics of *DBVersion*.

We plan to implement a wider range of database refactorings and formalise the refactoring process. We are currently testing the software in a small software development team.

REFERENCES

Ambler, S. W. and Sadalage, P. J. (2006). *Refactorings Databases: Evolutionary Database Design*. The Addison-Wesley Signature Series. Addison-Wesley.

Bergstraesser, T., Bernstein, P., Pal, S., and Shutt, D. (1999). Versions and workspaces in microsoft repository. In *Proceedings of the ACM SIGMOD*, pages 532–533, Philadelphia. Microsoft Corporation.

Conradi, R. and Westfechtel, B. (1998). Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282.

db4objects Inc. (2008). *db4o*. db4objects Inc.

Innovartis (2008). *DB Ghost*. Innovartis.

LiquiBase (2008). *LiquiBase*. LiquiBase.

Oracle (2007). *Oracle Change Management Pack for Oracle Database 11g*. Oracle.

Pilato, C. M., Collins-Sussman, B., and Fitzpatrick, B. W. (2004). *Version Control with Subversion*. O'Reilly Media, Inc.

Ploski, J., Hasselbring, W., Rehwinkel, J., and Schwierz, S. (2007). Introducing version control to database-centric applications in a small enterprise. *IEEE Software*, 24(1):38–44.

Roddick, J. F. (1995). A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393.

Sciore, E. (1994). Versioning and configuration management in an object-oriented data model. *VLDB J.*, 3(1):77–106.

Thomas, D., Heinemeier, D., and Breedt, L. (2005). *Agile Web Development with Rails: A Pragmatic Guide*. Pragmatic Bookshelf.

Tichy, W. F. (1985). RCS — a system for version control. *Software Practice and Experience*, 15(7):637–654.

Vesperman, J. (2006). *Essential CVS*. O'Reilly Media, Inc.

Wingerd, L. (2005). *Practical Perforce*. O'Reilly Media, Inc.

Yan, L. L., Miller, R. J., Haas, L. M., and Fagin, R. (2001). Data-driven understanding and refinement of schema mappings. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 485–496, New York, NY, USA. ACM.