Centro de Investigación en Matemáticas, A.C.

Control de versiones sobre bases de datos relacionales: análisis de herramientas y sus características

REPORTE TÉCNICO

Que para obtener el grado de

Maestro en Ingeniería de Software

P r e s e n t a

Dante Ramos Orozco

Director(a) de Reporte Técnico

Dr. Jorge Manjarrez Sánchez

Zacatecas, Zacatecas., 10 de octubre de 2012

Agradecimientos

Durante el transcurso de los estudios de maestría ha habido muchas situaciones por las cuales estoy agradecido, desde la aceptación para el ingreso al CIMAT hasta el examen de grado. Por lo tanto agradezco a toda la gente que tuvo influencia en esos momentos, mis padres, mis amigos, mis compañeros, los maestros; pero especialmente a mi novia Elizabeth que se convirtió en mi esposa en un punto de estos dos años y que ha sido mi apoyo desde hace tiempo.

Así mismo, agradezco:

- Al Centro de Investigación en Matemáticas y al personal involucrado en los trámites que se llevaron a cabo para la beca de manutención que me fue otorgada. Igualmente al Consejo Zacatecano de Ciencia, Tecnología e Innovación encargados de los fondos de la beca.

- Al personal docente y directivo del Centro de Investigación en Matemáticas por sus enseñanzas, empeño y preocupación por que la educación que se imparta en la institución sea de alta calidad.

- A mis compañeros de la cuarta generación y quinta generación tiempo completo porque juntos forjamos un ambiente sano de trabajo y aprendizaje.

Centro de Investigación en Matemáticas, Unidad Zacatecas. Av. Universidad No. 222. Fracc. La Loma. C.P. 98068. Zacatecas, Zacatecas, México.

www.ingsoft.mx

Dante Ramos

Ingeniero en Sistemas Computacionales

CIMAT Zacatecas

dante.ro86@gmail.com

Resumen

En la mayoría de los sistemas de software, el componente de bases de datos es fundamental, pues contiene toda la información importante para el negocio y el cómputo científico. Sin embargo, dentro del ciclo del desarrollo su relevancia se minimiza. Ahora bien, la base de datos no solamente es el esquema, sino también todo el código para consultas precompiladas, procedimientos almacenados, y demás código necesario para agilizar los procesos y mantener operacional toda la aplicación, incluida la documentación. Mantener el control de todo esto en su desarrollo, evolución y mantenimiento es una tarea compleja para hacerla manualmente, mucho más difícil si el desarrollo está a cargo de un equipo geográficamente distribuido. En este documento estudiamos la administración del desarrollo de bases de datos mediante sistemas de control de versiones (**SCV**) como una solución al control de cambios y desarrollo colaborativo. Primero abordamos los sistemas de control de versiones conocidos, sus categorías y las características de las mismas; además ejemplificamos las maneras de configurar un sistema de control de versiones con dos SCVs más representativos proponiendo algunas recomendaciones para el buen manejo de versiones sobre bases de datos. Enseguida, nos enfocamos a SCVs más específicos para el desarrollo de base de datos, se ejemplifica su uso y se hace un análisis comparativo con los SCVs genéricos.

Palabras y frases clave: control de versiones, SCVs, bases de datos relacionales, Subversion, SVN, Mercurial, Liquibase.

ABSTRACT

For most software systems, the database component is critical because it contains all important information for business, scientific computation, and so on. However, during the development cycle, its relevance is minimized. Now well, the database is not only the scheme, but all the code for precompiled queries, stored procedures, and other code to keep operational and maintain the entire application, including documentation. The management of all these during its development, evolution and maintenance is a complex task to do it manually, even harder if the development is done by a geographically distributed team. In this document, we study the management of database development using version control systems (VCS) as a solution to change tracking and collaborative development. First, we discuss some known version control systems, their categories and their characteristics; also we exemplify the configuration process using two of the most representative VCS and provide some advice to their use on database version control. Next, we focus on more specific VCS for database development, exemplifying their use and making a comparative analysis with the generic VCSs. This allows us to conclude on specific advantages of specific VCSs over generic ones during the development of databases.
Key words and phrases: revision control, versioning software, version control, VCS, relational databases, Subversion, SVN, Mercurial, Liquibase.

Contenido

1 Introducción 5
2 Sistemas de Control de Versiones Genéricos 7
2.1 Estado del Arte 7
2.1.1 Clasificación 7
2.1.2 Centralizado vs distribuido 10
2.1.3 Historia de CSVs 10
2.1.4 Acciones básicas sobre repositorios 11
2.1.5 Flujo evolutivo de un repositorio 12
2.1.6 Términos usados comúnmente en un SCV 13
2.2 Control de Versiones de Bases de Datos con un SCV genérico 14
2.2.1 Ciclo de desarrollo de una base de datos 17
2.2.2 Introducción a Mercurial y Subversion 19
2.2.3 Ejemplo Mercurial 20
2.2.4 Ejemplo SVN 26
2.2.5 Conclusiones parciales del control de versiones sobre bases de datos utilizando un SCV genérico 30
3 Sistemas de control de versiones especializados 31
3.1 Introducción a Red Gate 32
3.2 Introducción a DBForge 33
3.3 Liquibase 35
4 Conclusiones y trabajo futuro 43
5 Bibliografía 45

Índice de Figuras

Figura 2.1. Proceso de producción sin mecanismo de control [12]. 8
Figura 2.2. Proceso de producción con control exclusivo [12]. 9
Figura 2.3. Proceso de producción con control colaborativo [12]. 9
Figura 2.4. Ejemplo de flujo evolutivo de un proyecto [18]. 13
Figura 2.5. Jerarquía de bases de datos propuesta para el control de versiones con Delta Scripts. 15
Figura 2.6. Actividades de establecer el entorno del proyecto [29]. 17
Figura 2.7. Actividad de realizar un trabajo de desarrollo iterativo aislado [29]. 18
Figura 2.8. Actividad de generar el proyecto [29]. 18
Figura 2.9. Actividades de realizar la implementación desde el entorno del proyecto [29]. 19
Figura 2.10. Ventana de clonación de repositorio TortoiseHG. 21
Figura 2.11. Carpetas de los repositorios TortoiseHg. 22
Figura 2.12. Ventana de commit 1 TortoiseHg. 23
Figura 2.13. Ventana de sincronización TortoiseHg. 23
Figura 2.14. Ventana de exploración de repositorio TortoiseHg. 24
Figura 2.15. Ventana de commit 2 TortoiseHg. 25
Figura 2.16. Creación de repositorio con TortoiseSVN. 27
Figura 2.17. Ventana de Checkout con TortoiseSVN. 27
Figura 2.18. Carpetas copia local TortoiseSVN actualizadas. 28
Figura 2.19. Carpetas copia local TortoiseSVN desactualizadas. 28
Figura 2.20. Ventana de commit 1 TortoiseSVN. 29
Figura 2.21. Ventana de commit 2 TortoiseSVN. 30
Figura 3.1. Ventana de ligue de base de datos con SVN en Red Gate. 32
Figura 3.2. Ventana de SQL Source Control en Red Gate. 33
Figura 3.3. Ventana de creación de alias en DBForge. 34
Figura 3.4. Ventanas de comparación y sincronización en DBForge. 34
Figura 3.5. Código de creación de la tabla DATABASECHANGELOG de Liquibase [32]. 36
Figura 3.6. Tablas agregadas automáticamente por Liquibase. 37
Figura 3.7. Ejemplo de ChangeLog de Liquibase. 38
Figura 3.8. Comandos para la ejecución de acciones en Liquibase [35]. 38
Figura 3.9. ChangeLog de actualización de columna con comandos de Liquibase. 39
Figura 3.10. Ejecución en línea de comandos de Liquibase. 40
Figura 3.11. Estado de las columnas luego del primer cambio con Liquibase. 40
Figura 3.12. ChangeLog de actualización de columna con comandos SQL en Liquibase. 41
Figura 3.13. Registro de cambios en la tabla DATABASECHANGELOG de Liquibase. 41

Índice de tablas

Tabla 2.1. Clasificación de SCVs y año del primer release estable [7][16][17]. 11
Tabla 2.2. Sistemas de comparación de bases de datos [24] [25] [26] [27] [28]. 16
Tabla 3.1. Sistemas especializados en el control de versiones sobre bases de datos. [31] [32] 31
Tabla 3.2. Fases del ciclo de desarrollo de bases de datos que cubre Liquibase. 42

1 Introducción

Las bases de datos son el **elemento central** de la mayoría de los sistemas actuales de software, porque contienen la información que incumbe al negocio, sin embargo no se les ha dado la importancia requerida. Las bases de datos son tan **complejas y cambiantes** como los sistemas mismos; las bases de datos de sistemas "reales" **son más que solo un esquema** con pocas tablas y no más de 100 registros cada una (como las de puntos de venta de negocios pequeños, de controles escolares, etc.), las bases de datos reales contiene cientos de tablas, con miles de registros cada una, además de procedimientos almacenados, y demás código necesario para agilizar la operación y mantener funcional el sistema; por otro lado, **no importa** qué metodología de desarrollo se utilice, bases de datos como la que mencionamos no quedan listas para la implementación después del primer intento, y generalmente cuando se requiere un cambio significativo en el código, dicho cambio afecta a la base de datos [1] [2]. Administrar los cambios de una base de datos así no es una tarea que se pueda hacer manualmente ni mucho menos por un solo individuo. La problemática con el control de versiones de las bases de datos es muy amplia, simplemente consultando con excompañeros de estudios profesionales y conocidos de la universidad que se dedican al desarrollo de software, se obtiene respuestas vagas sobre **cómo trabajan en conjunto** los *product masters* (que por lo general son los *testers* porque conocen las necesidades del sistema detalladamente) y los desarrolladores para poder crear "versiones" copia de las bases de datos adecuadas para cada una de los *deploys,* o simplemente se puede ver la reacción en sus miradas, creando tareas ficticias en sus mentes para argumentar el control sobre las versiones de las bases de datos mientras mencionan la palabra "*scripts*" repetidamente, y entre los más sinceros simplemente dicen "pobremente se hace control de versiones" sabiendo que dicho problema **cuesta mucho [** 3]**. Hasta cierto punto** la idea de que las modificaciones a la base de datos la hagan en conjunto los *testers* y los desarrolladores es buena porque por lo general los principales cambios que se hace son durante el diseño y desarrollo temprano, por lo que defectos en esas fases implican mucho costo en la fase de pruebas. Por experiencia profesional sabemos que existen proyectos muy grandes, donde durante el transcurso de los años, un sistema base evoluciona y se modifica para satisfacer las necesidades de un cliente específico, ahí es donde resalta la necesidad de un sistema de control de versiones para bases de datos que facilite administrar las diferentes versiones y evite problemas irreversibles; por otro lado, también puede prevenir a las empresas de hacer respaldos enormes de bases de datos y luego obligar a servidores la tarea titánica de recuperarlas.

El **control de versiones** es la administración de los cambios que ocurren sobre un producto, donde una versión es un estado del producto en un momentos específico, y se usa más comúnmente en la industria de la informática siendo el **código fuente** el objeto que más se administra además de otros componentes como documentos o imágenes que formen parte del sistema. El control se puede llevar manualmente aunque existen **herramientas para automatizar este proceso** y es, desde nuestro punto de vista, la vía más económica en cuanto a tiempo y recursos de almacenamiento. Un control manual implicaría guardar una copia de un documento cada vez que haga un cambio, y por otro lado tener un registro manual descriptivo de todos los cambios que se hagan y cuando se necesite regresar a una versión anterior, aplicar el inverso de cada uno de los cambios que tenga en dicho registro [4]. La **motivación** principal de utilizar un sistema de control de versiones (SCV), independientemente del tamaño del proyecto, es que **ahorra tiempo y problemas** en caso de ocurrir un error (accidental o no) y

esos sistemas te permiten volver fácilmente a una versión anterior del trabajo en caso de necesitarlo [5]. Hablando de proyectos grandes y evolutivos a largo plazo el uso de un SCV para administrar los cambios dentro de un sistema tiene otras ventajas como facilitar el **trabajo distribuido** permitiendo a individuos trabajar sobre diferentes módulos a la vez y no manejar versiones desactualizadas en el proceso; otra ventaja en ese tipo de proyectos es la **administración de ramificaciones**, una ramificación dentro de un historial de versiones es cuando a partir de un punto hay dos caminos evolutivos diferentes de algún producto (código, imagen o documento) ya sea por versiones separadas o para trabajar sobre una línea de cambios sin afectar los productos originales del proyecto directamente [4]. La evidencia de la necesidad actual del control de versiones circula por internet mediante opiniones personales sobre **experiencias e historias de éxito**, tal como los múltiples comentarios de la eficiencia y utilidad de Git con desarrollo paralelo aun cuando se utiliza el Visual Studio .net, o tal como la historia que narra Joseph Scott [6] acerca de un equipo de trabajo dedicado a la creación de libros con Adobe FrameMaker, un grupo de personas no especializadas en desarrollo que fácilmente implementaron el Subversion para llevar el control de versiones efectivo sobre los libros, **reduciendo el tamaño del repositorio** de aproximadamente 35 a 6 Gb.

Los SCVs comenzaron a surgir desde los años 70s, administrando código principalmente (por esa razón los llamaremos de aquí en adelante “SCVs genéricos”) aunque también administran otros tipos de archivos concernientes a los proyectos de software, por lo que han tenido más de 30 años para madurar lo suficiente y ofrecer un proceso definido para la correcta administración de versiones, pero la naturaleza de las bases de datos **es diferente** a la de cualquier archivo como para pensar que un SCV genérico pueda acoplarse con el desarrollo de una base de datos completamente, por lo cual, es importante primero conocer el ciclo de desarrollo de una base de datos (abarcando creación, evolución y mantenimiento) para poder ver la influencia de los SCVs dentro del ciclo.

Actualmente hay varias formas de administrar los cambios en las bases de datos utilizando herramientas especializadas (que llamaremos de aquí en adelante “SCVs específicos”), algunas de ellas incluyen la utilización de **SCVs genéricos** como auxiliar para el control de versiones y otros generando su **propio método** para mantener el control de cambios.

En los siguientes capítulos abordaremos los siguientes temas:

- En el capítulo 2, se ilustran los **SCVs genéricos**, su definición, clasificaciones, diferencias entre las principales clasificaciones, un resumen histórico de la aparición de cada uno de ellos, se ilustrará un flujo evolutivo de ejemplo de un SCV genérico. Con respecto a su uso sobre bases de datos, primeramente se dará una introducción al **ciclo de desarrollo de bases de datos** y luego **se ejemplificará** el uso de un SCV genérico sobre los scripts de una base de datos.
- El capítulo 3 es acerca de los **SCVs especializados** en el control de cambios de bases de datos, utilizando tres de las más representativas en la industria: DBForge, Red Gate y Liquibase, **ejemplificando** el uso de éste último.
- En el capítulo final se exponen las **conclusiones** identificadas a raíz de la investigación, además se aborda el **trabajo futuro** describiendo la evolución propuesta de la investigación.

2 Sistemas de Control de Versiones Genéricos

La industria del software lleva de existencia lo que las primeras computadoras (desde los 60's), y la urgencia del control de versiones sobre código fuente nació relativamente a la creación del software; cuando a principios de los 80s salían a la luz las primeras computadoras personales de escritorio**, el primer SCV** ya había sido liberado a finales de los 70s.

Existen muchos sistemas para administrar el control de versiones de productos informáticos, algunos basados en otros, siendo los más mencionados en los foros de desarrolladores: el **CVS** (Concurrent Versions System) que para muchos no ha perdido vigencia, el **SVN** (Subversion) que para muchos es el sucesor de CVS y ampliamente usado los últimos años; **Git** y **Mercurial** famosos entre usuarios de GNU Linux principalmente; por decir algunos más: **SourceSafe**, **ClearCase**, **Darcs**, **Bazaar** y **Plastic SCM** (se listan todos en la Tabla 2.1 más abajo) [7].

Tomando en cuenta que la opinión importante es la de los usuarios, es decir, la aceptación del factor industria es el principal parámetro para calificar los SCVs, los nombres de sistemas más sonados al momento de la redacción (según una consulta de blogs, foros y *posts*) son el ya considerado obsoleto *SourceSafe* de Microsoft, su sucesor *Team Foundation Server* igualmente de Microsoft que además del control de versiones cuenta con otras herramientas (como recolección de datos, reporteo y seguimiento de proyectos), el *Subversion* usado por mucho tiempo y por último pero no menos importante *Git* que obtuvo su fama por ser creado y utilizado por **Linus Torvalds** para facilitar el desarrollo del Kernel de Linux además de que desde su comienzo fue enfocado a ser veloz [8].

Un punto importante acerca de los SCVs genéricos, es que hay varias formas de trabajar con ellos, es decir, las ideologías sobre el trabajo **influyen** en el proceso, por ejemplo, en Git hay varias formas de obtener actualizaciones del *trunk* (este y otros términos serán desarrollados mas adelante) durante el desarrollo de una *branch* local, el comando *rebase* es para incluir los cambios que se han hecho en otra rama o el *trunk* en tu historial local, por otro lado, para tener los cambios hechos como tales en tu copia local hay dos formas principales, la más práctica y que es útil generalmente es usando el comando *pull* que hace internamente lo que dos comandos: *fetch* y *merge*, la **ventaja** de usar los comandos separadamente es que puedas tener una visión de lo que estas trayendo a tu copia local y luego decidir si haces el merge de dichos cambios. Otra forma de diferenciar entre dos ideologías diferentes durante el trabajo es juzgando la claridad del árbol evolutivo, es decir, si el *trunk* tiene pocos puntos (*commits*) independientes de ramificación, si los puntos dentro de las ramificaciones son concisos con cambios; para mejorar la limpieza del historial de cambios hay varias herramientas como por ejemplo el **modo interactivo** del comando *rebase* que permite unir *commits*, permitiendo tener una lista más clara de los puntos y lo que resolvieron [9] [5] [10].

2.1 Estado del Arte

A continuación abordamos la clasificación de los SCVs genéricos, hacemos un pequeño repaso sobre la historia de los mismos, y listamos terminología y flujo evolutivo de un repositorio de un SCV genérico.

2.1.1 Clasificación

Cada una de los SCVs se clasifica dentro de las siguientes familias:

Sistemas distribuidos: cada uno de los usuarios tiene su propia copia del repositorio y el historial de los cambios, en cualquier momento los usuarios sincronizan los repositorios y las versiones modificadas son puestas como las más actuales y en caso de que hubiera modificado un mismo documento, el sistema avisa para que los usuarios implicados decidan qué cambios son relevantes de cada uno. Ejemplos de estos sistemas son Bazaar, Git y Mercurial [11].

Sistemas centralizados: hay un solo repositorio central donde se encuentra todo el código y sus versiones, los usuarios descargan una copia de trabajo local, al realizar cambios se envía el documento al repositorio para que lo tome con una versión nueva y pueda estar disponible para los demás usuarios [11].

Para permitir el trabajo colaborativo sobre repositorios y evitar conflictos como el de sobrescribir archivos eliminando los cambios que alguien mas realizó (como en la Figura 2.1), los SCV usan **mecanismos de control [** 12**]**.

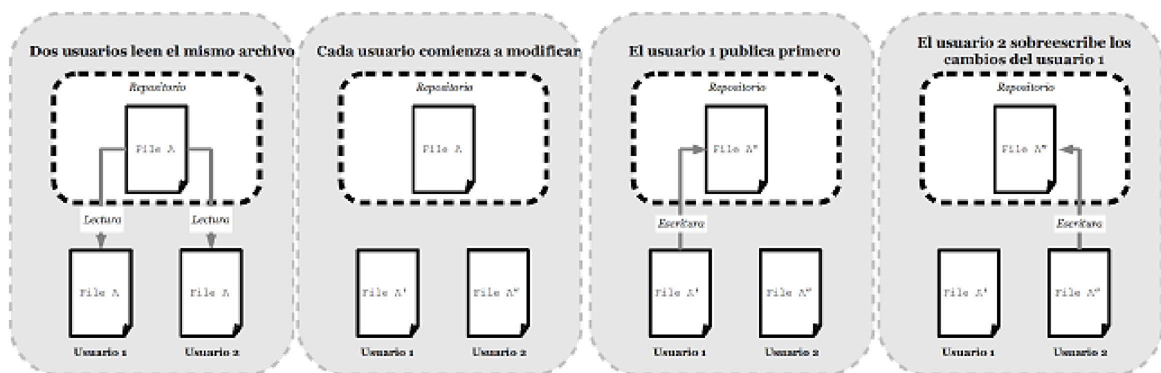

Figura 2.1. Proceso de producción sin mecanismo de control [12]**.**

La Figura 2.1 muestra como los usuarios 1 y 2 acceden a un mismo archivo, hacen cambios respectivamente, el usuario 1 produce una versión nueva del archivo y el usuario 2 sobrescribe el archivo subido por el otro usuario solo conservando los cambios que él mismo hizo.

Existen dos **mecanismos de control**:

- **Exclusivo**. Es cuando el sistema, al hacer el usuario el *checkout (petición de extracción) por parte del usuario,* no permite que nadie más haga modificaciones sobre ese mismo archivo.

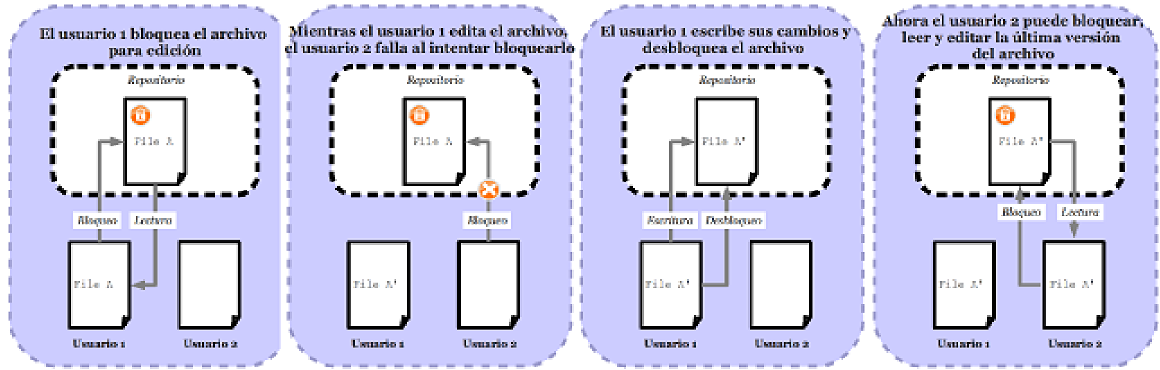

Figura 2.2. Proceso de producción con control exclusivo [12].

La Figura 2.2 muestra cómo es el proceso de modificación de archivos exclusiva, el funcionamiento básico es por el uso de **"candados"** que permiten que solo uno de los usuarios edite un archivo evitando conflicto de edición como el mostrado arriba.

- **Colaborativo**. Es cuando varios usuarios hacen copias locales y las modifican, luego cuando se envían los cambios, el sistema hace un *merge (fusión)* de todos los cambios para crear un solo documento, si existen errores en dicha acción el sistema notifica a los usuarios para que los solucionen mediante la función 'resolver'.

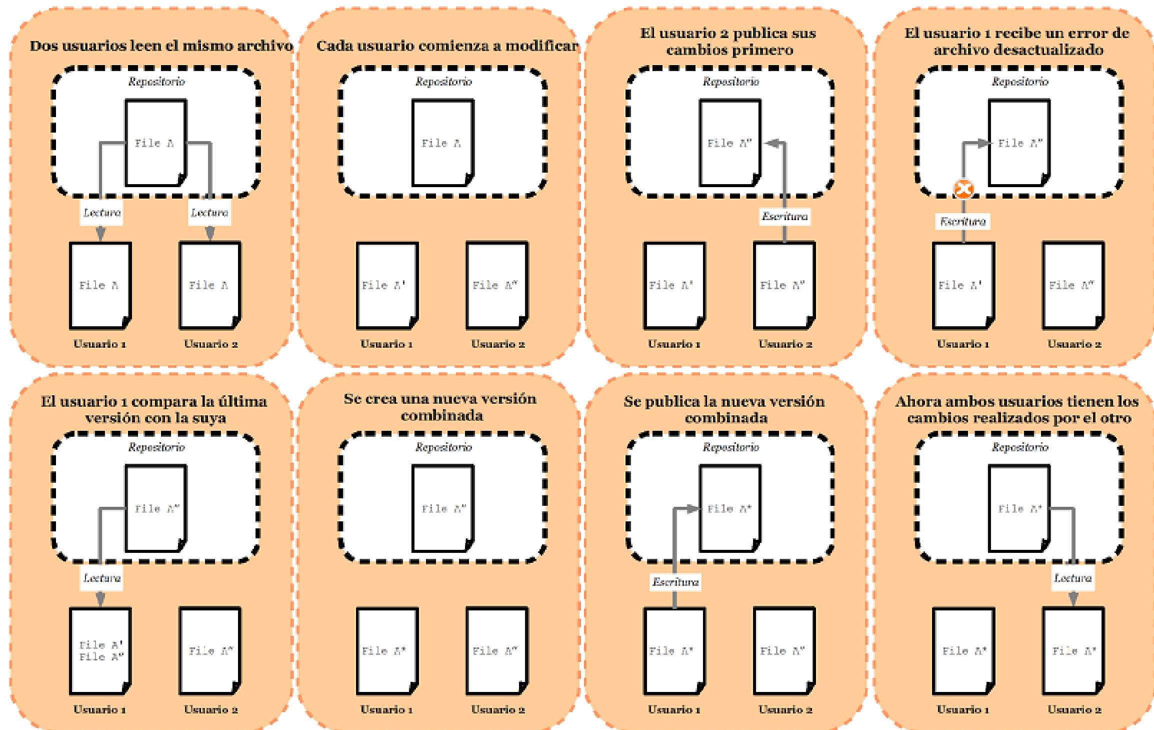

Figura 2.3. Proceso de producción con control colaborativo [12].

La Figura 2.3 muestra cómo se lleva a cabo el proceso de edición de archivos colaborativamente, como se puede apreciar, es más complicado a la manera exclusiva pero tiene **ventajas** que abordaremos mas adelante. En vez de sobrescribir el archivo que otro usuario haya modificado, el sistema manda un mensaje de 'archivo desactualizado' debiendo el usuario

resolver los conflictos para poder generar un documento que incluya los cambios que hizo el otro usuario y él mismo.

2.1.2 Centralizado vs distribuido

Los sistemas de control de versiones centralizados son fáciles de administrar y todos los usuarios saben hasta cierto punto en qué trabaja cada uno de los usuarios aunque **dependen de un servidor central** es una limitante muy fuerte dada la posibilidad de deficiencias en la comunicación a él como la pérdida de datos por mal funcionamiento, por ejemplo, o la corrupción en las unidades de almacenamiento [13][14]. Por otro lado en los sistemas distribuidos se reduce tal peligro ya que cada uno de los usuarios tiene una copia local del repositorio, además que varios de las instrucciones principales se hacen de manera más ágil porque no se necesita establecer comunicación con un servidor, dicha comunicación solo se establece cuando se hace el envío y obtención de datos de otro usuario [8].

Como ejemplo, Subversion es el más representativo de los sistemas genéricos centralizados, por el tipo de modificación de archivos (por medio de *check out* y *check in*) puede causar **problemas** como tiempos muertos y esperas injustificadas, porque archivos no pueden ser modificados simultáneamente y el proceso es susceptible a descuidos del usuario por ejemplo que haga *check out* sobre archivos innecesarios o que olvide devolver el control de algún archivo [15].

Pero por otro lado, los sistemas descentralizados no son solo ventajas, para las empresas de desarrollo con grandes proyectos y equipos de trabajo rotativos, los sistemas descentralizados generan un **problema de seguridad** de datos (pero no como los de pérdida de datos que mencionamos para los centralizados) dado que cada uno de los que trabajan sobre el proyecto tienen acceso a todos los componentes del mismo permitiendo consecuentemente mayor riesgo de que se dé mal uso de esa propiedad intelectual de la empresa [13].

En el caso de los descentralizados dado sus mecanismos de control, se puede decir que sus características no pueden ser aprovechadas cuando se trabaja sobre archivos binarios ya que como se puede ver en la Figura 2.3 el mecanismo de control se fundamenta en el uso de los comandos *solve y merge* que no se puede usar sobre **archivos binarios [** 9]**[** 14**]**. Éste punto es importante porque indica que no se pueden versionar archivos binarios de bases de datos utilizando esos comandos; por lo tanto, ya sea un SCV centralizado o uno distribuido, antes de decidir, es importante tener en mente si uno u otro son apropiados para el tipo de proyectos que se trabajarán.

2.1.3 Historia de CSVs

A partir de los años 70s comenzaron a existir sistemas para el control de versiones, el primero de ellos fue el *Software Change Manager* siendo de distribución privativa, el primer sistema de distribución gratuita que surgió fue el **CVS** en 1990. La Tabla 2.1 muestra el listado completo de CSVs.

Tabla 2.1. Clasificación de SCVs y año del primer *release* estable [7][16][17].

	Gratuitos	Propietarios
Centralizados	CVS (1990) ♦ CVSNT (1998) ♦ Vesta (2001)♦ Subversion (2004)	Software Change Manager (1970s) ♦ PVCS (1985) ♦ CMVC (1990s) ♦ QVCS (1991) ♦ ClearCase (1992) ♦ ~~Visual SourceSafe (1994) ♦~~ Perforce (1995) ♦ StarTeam (1995) ♦ MKS (2001) ♦ AccuRev (2002) ♦ Autodesk Vault (2003) ♦ Vault (2003) ♦ Team Foundation Server (2005) ♦ IC Manage (2007) ♦ Rational Team Concert (2008) ♦ Visual Studio Application Lifecycle Management (en Visual Studio 2005)
Descentralizados	Aegis (1999) ♦ ~~GNU arch (2001) ♦~~ Darcs (2002) ♦ ~~DCVS (2002) ♦~~ ~~SVK (2003) ♦~~ Monotone (2003) ♦ ~~Codeville (2005) ♦~~ Git (2005) ♦ Mercurial (2005) ♦ Bazaar (2005) ♦ ArX (2005) ♦ Fossil (2007) ♦ LibreSource (2008)	~~TeamWare (90s) ♦~~ Code Co-op (1997) ♦ BitKeeper (1998) ♦ Plastic SCM (2006)

En la Tabla 2.1Figura 2.1 se listan los sistemas de control de versiones clasificados por su manera de distribución, y junto a ellos el año de liberación de su primera versión estable. Los sistemas tachados son los que han quedado descontinuados o tienen fecha de cierre.

NOTA: la información en los SCVs centralizados es más segura a robo y ayuda a proteger la propiedad intelectual ante los propios desarrolladores aunque aumenta el riesgo de pérdida de datos o corrupción para lo cual es conveniente adoptar acciones para prevenir o corregir según sea el caso.

2.1.4 Acciones básicas sobre repositorios

Dentro de las acciones que se realizan sobre repositorios, las **más importantes** o las más comúnmente usadas son [15][9][18]:

- **Despliegue**: obtener o *check out* es cuando el usuario requiere una copia local del repositorio.
- **Enviar**: *commit, check in o submit* es cuando se envía un documento nuevo o modificado para integrarlo al repositorio.
- **Sincronizar**: integra los cambios hechos en el repositorio en la copia local.
- **Fusión**: o *merge,* se utiliza para la solución de problemas aplicando dos conjuntos de cambios en un mismo archivo. Puede usarse cuando dos usuarios modifican un mismo

archivo creando dos ramificaciones y para unir dichas ramificaciones se resuelven los cambios de ambas mediante el *merge*.

- **Añadir:** o *add,* es un comando para agregar un archivo o carpeta a la copia local de trabajo, para que se agregue al repositorio se debe hacer un envío (*commit*).
- **Diferencia**: o *dif,f*, muestra las diferencias entre dos versiones diferentes.
- **Eliminar**: o *delete,* es la contraparte del *add*, elimina un archivo de la copia local de trabajo y se elimina del repositorio cuando se hace el envío (*commit*).
- **Resolver**: o *solve*, se utiliza cuando hay algún conflicto (se define mas detalladamente posteriormente) para resolver los problemas a la hora de hacer una fusión.
- **Revertir**: o *revert,* es cuando se decide deshacer los cambios hechos localmente volviendo a la copia prístina.
- **Actualizar**: o *update,* se utiliza para actualizar la copia local con el estado actual del tronco del repositorio fusionando cualquier cambio que alguien más ha hecho.

2.1.5 Flujo evolutivo de un repositorio

Cuando un repositorio es modificado varias veces el flujo evolutivo o historial de evolución puede parecer complicado ya que desde ese historial se puede recuperar a una versión anterior por lo cual en la Figura 2.4 se plasma un ejemplo de la evolución de un producto desde el punto de vista del control de versiones resaltando las principales características de éste tipo de gráficos.

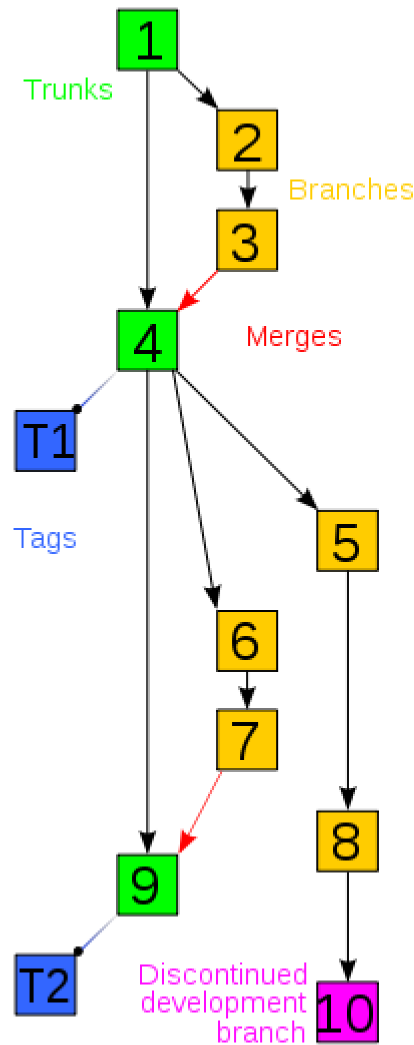

Figura 2.4. Ejemplo de flujo evolutivo de un proyecto [18].

En la Figura 2.4, los nodos verdes representan el tronco evolutivo o *trunks* (se podría decir que es la rama principal de la evolución) donde cada uno de los nodos representa adición de funcionalidades, en éste caso los nodos amarillos representan versiones en ramificaciones o *branches* que se encargan de la corrección de bugs a los cuales se les hace *merge* con el tronco principal para que las versiones del mismo no sean afectadas por bugs y la legibilidad del árbol sea mas congruente, los nodos azules son rótulos o *tags,* sirven para asegurar la localización de las versiones marcadas (en la mayoría de los sistemas de gestión de versiones usan ya sea *tags, labels o baselines* para este propósito), y por último el nodo rosa es una ramificación descontinuada.

2.1.6 Términos usados comúnmente en un SCV

Además de los términos de las actividades que mencionamos antes, los siguientes son algunos de los términos con los que se debe estar familiarizado cuando se refiera a SCVs genéricos [15] [9]:

- **Repositorio**: lugar central donde se almacenan y mantienen los datos.

- **Rama:** cuando el desarrollo se parte en un punto en concreto y sigue dos caminos separados, se puede crear una rama a partir de la línea principal de desarrollo para desarrollar una nueva funcionalidad sin hacer que la línea principal quede inestable.
- **Conflicto**: es cuando al hacer un envío (*commit*) los cambios ocurren en las mismas líneas. En este caso el SCV no puede decidir automáticamente qué versión utilizar, y se dice que el archivo está en conflicto. El usuario debe editar el archivo manualmente y resolver el conflicto antes que pueda confirmar más cambios.
- **Registro o historial**: muestra la historia de las revisiones de un archivo o carpeta.
- **Parche**: es un archivo generado con un *diff* que resume los cambios en un código fuente.

2.2 Control de Versiones de Bases de Datos con un SCV genérico

Un método tentador para trabajar con bases de datos sin pensar en el control de versiones es tener una **base de datos centralizada** y con permisos de los desarrolladores para hacer modificaciones concurrentes, lo cual es un gran **error** que ha conducido a fracasos de proyectos completos, esto según K. Scott Allen en una publicación en su blog personal con el nombre de '*Three rules for database work*' [2] y con quien estamos de acuerdo, pero en vez de responsabilizar la forma en la que se hacen los cambios, nosotros tenemos la hipótesis de que la razón por la que esa metodología es susceptible al fracaso es porque no se tiene un correcto control de los cambios, por ello debemos buscar una herramienta que nos ayude en el registro de los cambios [19].

El control de versiones de una base de datos se puede apoyar utilizando un SCV genérico, pero al no ser herramientas especializadas en bases de datos, es obvio que el control de versiones no se automatiza por completo, es por eso que en ésta sección explicamos la manera de hacer éste trabajo utilizando un SCV genérico.

En el caso de las bases de datos, lo que versionaremos principalmente son los scripts con los que se crean o hacen modificaciones a las bases de datos, aunque también es posible versionar respaldos de los archivos binarios que las bases de datos generan (dependiendo del tipo de base de datos que se esté trabajando), para ello es posible usar un SCV genérico.

Un término importante y que sirve de apoyo para el control de versiones, son los ***delta scripts***, que en éste caso son script en lenguaje SQL (*structured query language*) que resultan de la comparación de dos estados de una base de datos permitiendo resumir los cambios que ocurrieron entre los dos estados y de ser necesarios aplicarlos para igualar ambas bases de datos [**20**]. Los delta scripts, pueden ser de índole **estructural** (creación de bases de datos, tablas y vistas), **funcional** (creación de procedimiento almacenados, consultas precompiladas, etc.) y de **datos** (datos precargados para pruebas o que incluyan datos sobre la configuración del sistema). Pero antes de comenzar con este método, se debe tomar una decisión: un script o varios scripts; existen herramientas de creación de delta scripts que generan un solo script para la sincronización, pero si se desea tener un control más específico de los cambios la técnica más recomendada sería elaborar los scripts manualmente agrupando o separando los cambios según se desee [21] [22].

Como un SCV genérico solo influye en el control de versiones de los scripts de una base de datos, es necesario tener una metodología para trabajar sobre la base de datos, para lo cual proponemos que se tenga las siguientes pautas [22]:

1. Que se tengan los scripts de las bases de datos dentro de un repositorio administrado por un SCV genérico.
2. Que cada desarrollador de la base de datos haga una copia local de la versión estable de la base de datos oficial (final).
3. Que haya un responsable de aplicar los scripts de los cambios cuando éstos sean liberados

Por lo tanto la jerarquía propuesta de las bases de datos sería como la siguiente:

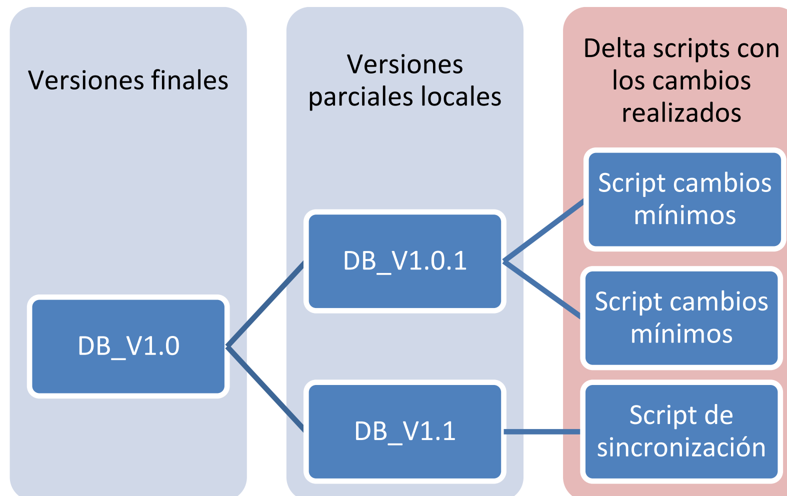

Figura 2.5. Jerarquía de bases de datos propuesta para el control de versiones con Delta Scripts.

La Figura 2.5 muestra la jerarquía de las bases de datos, la idea es que el desarrollador trabaje sobre una copia local de la base de datos, genere versiones **parciales** administrando scripts de cambios mínimos y cuando se mande a pruebas o se comience un nuevo ciclo de desarrollo, se genere un script de **sincronización** que esté accesible al resto del equipo de trabajo y un responsable haga la sincronización de los cambios con la base de datos final.

Según lo propuesto arriba, el uso de los scripts puede variar entre scripts de cambios mínimos (pocas operaciones sobre la base de datos) o scripts de sincronización que sirvan para aplicar los cambios necesarios para que las bases de datos queden uniformes. Una propuesta para la forma de utilizar los scripts seria que localmente el desarrollador genere los scripts de cambios mínimos para controlar lo que hace y poder **revertir** las acciones en caso de algún inconveniente, además de que cuando el producto tenga un nivel aceptable para llevarlo a la fase de pruebas se cree un script de sincronización para que los responsables de pruebas tengan **un solo script** para llevar la base de datos al estado deseado para la prueba, además que usar ese tipo de scripts representa mayor **economicidad** a la hora de realizar la documentación del proyecto. Es decir, los scripts de cambios mínimos serían para uso del desarrollador y los scripts de sincronización para el proyecto [20].

Si se deseara gestionar las versiones finales sin necesidad de mantener los scripts de sincronización, ni tener un respaldo por cada una de las versiones liberadas, una **solución** sería mantener un solo respaldo **versionado** con un SCV (puede ser el mismo que se use para versionar los scripts) , dicha idea esta fundamentada en la historia compartida por Joseph Scott [6] y que ya abordamos antes donde demuestra que el uso del SVN mejoró el repositorio de trabajo comparado con un control de versiones manual; hay muchos comentarios en la *web* que resaltan que para manejo de archivos binarios han usado **SVN** durante años y que lo seguirán

usando, pero un estudio hecho por Josh Carter [23] que comparó el rendimiento de SVN, Mercurial, Git y Bazaar, demuestra que tanto en velocidad como en tamaño del repositorio el menos eficiente fue SVN y el mejor calificado fue Git.

Pero antes de comenzar de lleno con los SCVs, es importante analizar las herramientas que permiten la diferenciación de bases de datos, dichas herramientas **son parecidas** a las herramientas *diff* de los SCVs ya que muestran la comparativa entre dos diferentes versiones de bases de datos delineando los cambios que se hicieron, pero las herramientas que nos interesan son las que además de mostrar las modificaciones, permiten crear un delta script que sirva para sincronizar las bases de datos, es decir, si tuvieran una base de datos A y una base de datos B siendo ésta segunda una versión modificada de la primera, un delta script de la comparativa entre las bases de datos debería permitir igualar A con B mediante su aplicación [21].

Tabla 2.2. Sistemas de comparación de bases de datos [24] [25] [26] [27] [28].

Herramienta	Plataforma (DBMS)	Delta Scripts	Compara Estructura/Datos	Distribución	Comentarios
DBComparer	MSSQL	× (Solo ambos scripts)	✓/✓ (filtrado de comparativa)	Gratuita	Permite ver a detalle las diferencias gráficamente y en script
DBForge Schema/Data Comparer	MSSQL, MySQL	✓ (incluye sincronización automática)	✓/✓ (dos herramientas separadas)	Privativa	Parte de una suite más completa
Redgate's MySQL Schema/Data Comparer	MySQL	✓ (incluye sincronización automática)	✓/✓ (dos herramientas separadas)	Privativa	Parte de una suite más completa
AdeptSQL	MSSQL	✓ (opción de detalle)	✓/✓	Privativa	
Tablediff Utility	MSSQL	✓	✓/✓	Gratuita (en SQL Server)	Solo línea de comandos
OpenDiff	MSSQL	✓	✓/×	Gratuita	Aún en versión Beta
SQL Admin Studio	MSSQL, Access, MySQL	✓	✓/✓	Gratuita	Muy completo
SQL Delta	MSSQL	✓	✓/✓	Privativa	
Mysql-diff	MySQL	✓	✓/×	Gratuita	Línea de comandos
MySQL Workbench for Database Change Management	MySQL	✓	✓/✓	Gratuita (hay una edición comunidad)	La versión gratuita no ofrece tanta versatilidad
Sqlt-diff	MySQL	✓	✓/×	Gratuita	Línea de comandos, versión BETA
Mysqldiff	MySQL	✓	✓/×	Gratuita	Línea de comandos, versión ALPHA
Diffkit	Oracle, MySQL, DB2, cualquier JDBC	✓	✓/×	Gratuita	Línea de comandos, multiplataforma, código abierto

SQLyog	MySQL	✓	✓/✓	Privativa	Muchas herramientas pero no control de versiones como tal

La Tabla 2.2 muestra un listado amplio de herramientas con las características principales como, plataforma o Sistema Administrador de Bases de Datos (DBMS), si genera *delta scritps* (scripts de sincronización), si compara estructura y datos, la distribución (privativa o gratuita) y una sección de comentarios.

Hay muchas herramientas para crear los delta scripts, muchos son de paga, por lo que **no son tan atractivas** hacia los desarrolladores a la hora de elegir una herramienta para trabajar aunque tengan muchas otras características y la mayoría de los gratuitos solo se enfocan en la comparación de los **esquemas** (en ocasiones no hacen una comparación completa de ellos). Además de los listados hay muchas aplicaciones que las empresas desarrollan para satisfacer sus necesidades a la hora de crear delta scripts [21].

Para poder ubicar el control de versión dentro del ciclo de desarrollo de una base de datos es necesario conocer cuál es ése ciclo. A continuación ilustraremos el ciclo de desarrollo de bases de datos que incluye el control de versiones de las mismas.

2.2.1 Ciclo de desarrollo de una base de datos

Es importante conocer la cabida del control de versiones en el **ciclo de administración de bases de datos**, seguramente a este punto podemos fabricar una idea de dicha relación pero es necesario desarrollarla más formalmente.

Tomaremos como ejemplo el ciclo de vida de bases de datos usado por una de las suites más completas para desarrollo de software, el Visual Studio .net de Microsoft. Según la documentación oficial del sitio de ayuda de Microsoft, el **ciclo de vida** de desarrollo de una base de datos contiene las siguientes fases:

1. **Establecer el entorno del proyecto.** Se establece la base del proyecto de bases de datos importando un esquema ya existente de una base de datos sobre un servidor de pruebas.

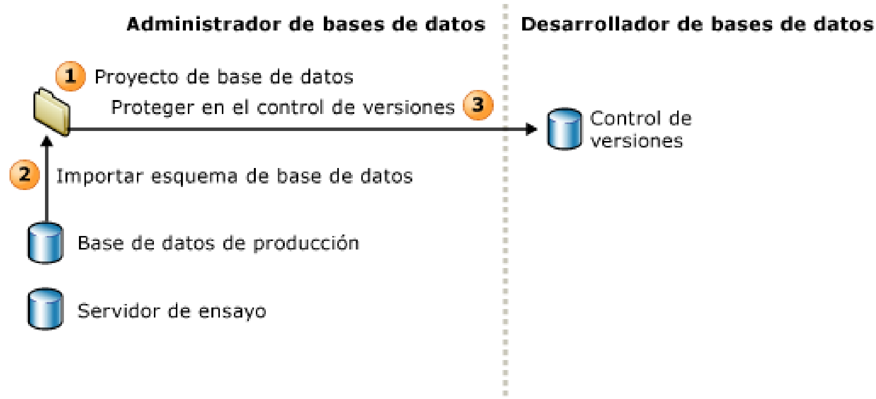

Figura 2.6. Actividades de establecer el entorno del proyecto [29].

2. **Realizar un trabajo de desarrollo iterativo aislado.** Se lleva a cabo el desarrollo con control de versiones.

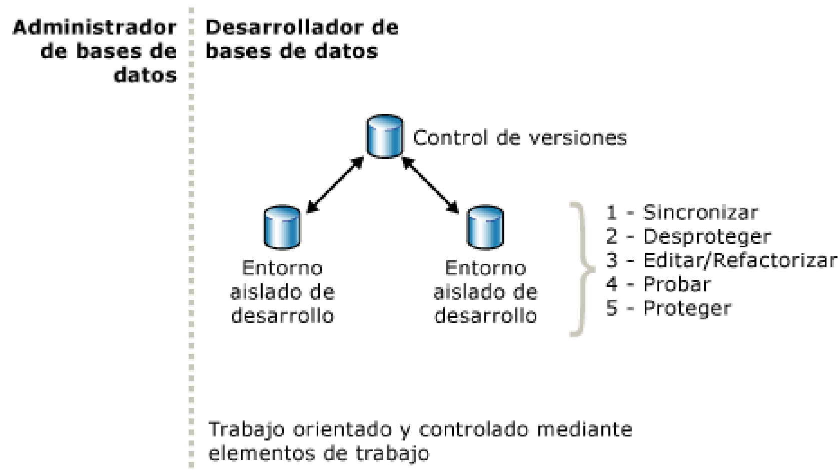

Figura 2.7. Actividad de realizar un trabajo de desarrollo iterativo aislado [29].

3. **Generar el proyecto.** Durante el desarrollo del proyecto de base de datos se hace una sincronización del proyecto de la versión más reciente aprobada.

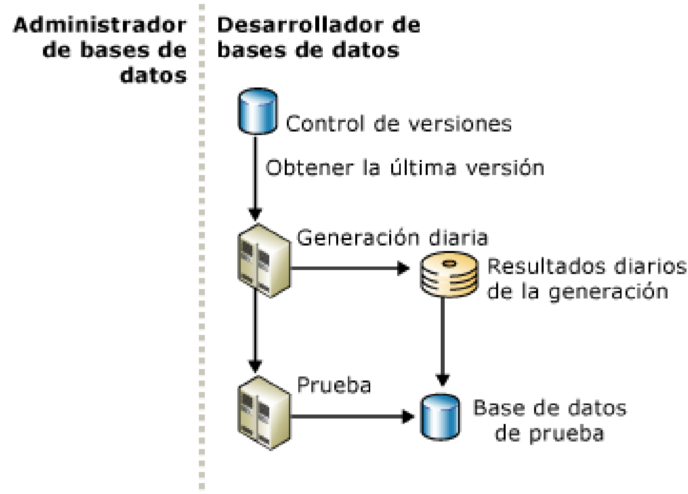

Figura 2.8. Actividad de generar el proyecto [29].

4. **Realizar la implementación desde el entorno del proyecto.** A base de la generación del proyecto anterior, se genera la implementación del proyecto recuperando los archivos del proyecto (esquema y pruebas), y se hacen implementaciones en los servidores de pruebas y de producción.

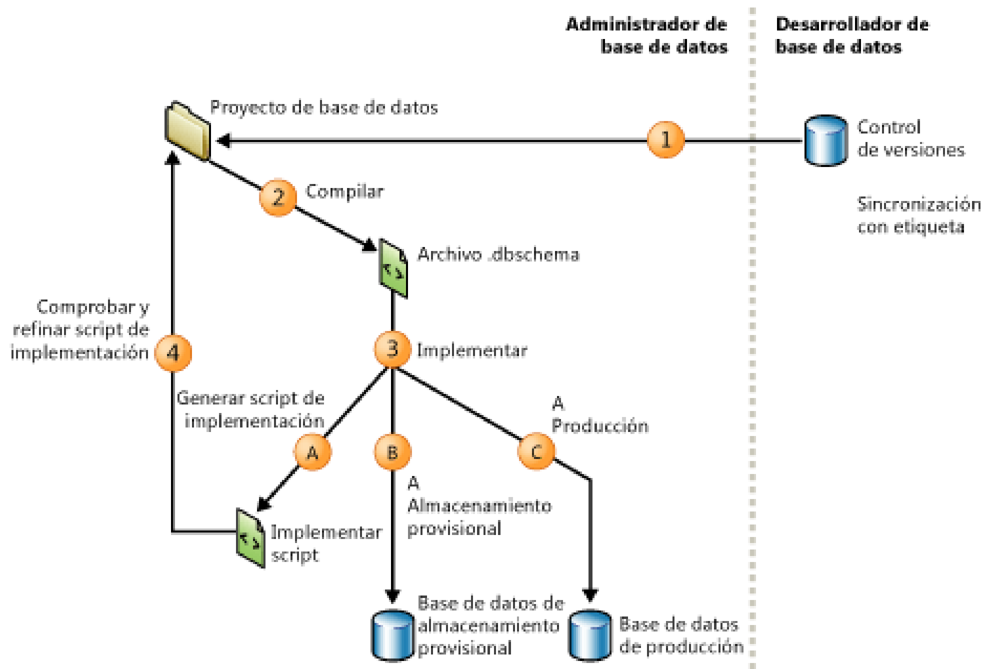

Figura 2.9. Actividades de realizar la implementación desde el entorno del proyecto [29].

Según el ciclo de desarrollo que se puede ver en las imágenes, del equipo de trabajo se designa a un individuo como el **administrador de base de datos** que realiza tareas específicas y que sería complicado que varios individuos realizaran a la vez, un punto importante de éste modelo y que marca la pauta para su uso es que es **centralizado** y por ello tiene los pasos que definimos arriba. Y debemos recordar que ésta suite **no es la única** que controla las versiones de bases de datos, es por eso que en la sección 4 analizamos los SCVs especializados en bases de datos incluyendo éste.

A continuación se da un pequeña introducción a dos SCVs: Mercurial y SVN, Mercurial porque se desglosará un ejemplo sobre él, y SVN porque es el perteneciente a los centralizados y del cual recientemente hablamos.

2.2.2 Introducción a Mercurial y Subversion

Ya teniendo una idea de cómo debe ser el ciclo de desarrollo de una base de datos, comenzaremos con un ejemplo **sencillo pero básico** del control de versiones de los **scripts** de una base de datos, pero **sin olvidar** que éste tipo de control no **involucra** ni la creación de la base de datos local ni la implementación de los cambios en las bases de datos.

Los SCVs genéricos a usar son el Mercurial y SVN, que son dos de los más representativos y aunque se habló de que no son los más efectivos, se decidió que se utilizarían los mismos para una ejemplificación dado que en empresas locales se ha visto que se utilizan mas [8] [4].

2.2.2.1 Mercurial

Es un sistema de control de versiones distribuido, y como lo mencionamos arriba, los sistemas distribuidos de control de versiones hacen que cada uno de los usuarios tenga una copia del repositorio debiendo sincronizar entre repositorios para tener una versión actualizada [**9**].

En **bases de datos** esto sería que cada persona tuviera a la mano todos los scripts necesarios para la creación de la base de datos junto con todo el historial de versiones de los mismos.

En nuestra opinión los SCVs **distribuidos** serían de utilidad cuando el equipo de trabajo se encuentra distribuido permitiendo no limitar los recursos para el correcto trabajo, por otro lado los sistemas centralizados en un panorama de trabajo distribuido implicaría que los usuarios cada vez que necesitaran modificar, enviar o regresar de versión un documento hicieran comunicación con el servidor y esto tendría las desventajas que acarrea el medio de comunicación [8].

2.2.2.2 SVN

Recientemente SVN es el más popular SCV centralizado y por lo tanto el claro sucesor de CVS. Los sistemas centralizados para control de versiones usan una **comunicación cliente-servidor** lo que, como lo describimos en el párrafo anterior necesita el usuario hacer peticiones al servidor para modificaciones sobre el repositorio e igualmente para poder revisar el historial de cambios [15] [4].

El uso de un sistema **centralizado** sería para darle seguridad a la **propiedad intelectual** principalmente cuando se trabajas con *outsourcing* y aumenta ese peligro.

2.2.3 Ejemplo Mercurial

Siguiendo con la política de proporcionar ejemplos para usuarios principiantes, hemos decidido usar solo herramientas gráficas y explicar paso a paso cada una de las acciones realizadas. Por ello, para este ejemplo usamos el sistema operativo GNU Linux distribución Ubuntu 12.04 montado sobre una máquina virtual VirtualBox, como el título dice, el SCV que se utilizó fue **Mercurial**, siendo la herramienta gráfica **TortoiseHg** la seleccionada para el ejemplo. La razón de utilizar una **herramienta gráfica** es para que los usuarios primerizos se familiaricen más rápidamente con las ideas que mencionamos aquí, además que los usuarios que prefieran usar línea de comandos pueden hacerlo, utilizando las instrucciones necesarias y haciendo uso de la ayuda que viene con Mercurial en caso de necesitarla.

Aunque el TortoiseHg también tiene versiones para otros sistemas operativos, se eligió hacer el ejemplo con Linux ya que en muchas ocasiones este es preferido para realizar trabajos de desarrollo.

Siguiendo con la propuesta de la Figura 2.5, es decir, usando versiones locales de bases de datos para trabajar sobre ellas, es requisito previo para el ejemplo, haber **creado o importado** una base de datos, en el caso de MySQL la creación se hace a base de un archivo .sql, de preferencia los archivos .sql de estructuras de bases de datos también deben ser **versionados** para facilitar su administración y posiblemente sobre ellos realizar cambios, también se debe haber instalado el TortoiseHg y su integración con el navegador de archivos de Ubuntu (Nautilus).

Para ejemplificar el control de versiones decidimos dejar de lado las grandes bases de datos y scripts de funciones complicados porque como se dijo antes, éste ejemplo es sencillo pero básico, por lo tanto, el ejemplo es sobre una base de datos que se llama “Escuela” que cuenta con una **tabla** que se llama “**Alumno”** con los espacios: ID alumno, “nombre, apellidos, edad, dirección, número de teléfono 1, número de teléfono 2 y créditos (adquiridos por asignaturas aprobadas). El script de creación de la tabla sería:

```
CREATE TABLE  Alumno (ID_Alumno VARCHAR( 9 ) NOT NULL ,
      Nombre VARCHAR( 30 ) NOT NULL ,
      Apellidos VARCHAR( 30 ) NOT NULL ,
      Edad INT NOT NULL ,
      Direccion VARCHAR( 50 ) NOT NULL ,
      Tel_Casa VARCHAR( 7 ) NOT NULL ,
      Tel_Contacto VARCHAR( 10 ),
      Creditos INT NOT NULL ,
      PRIMARY KEY (  ID_Alumno ));
```

La razón hipotética de que se necesite el control de versiones es que el **requerimiento** sobre la base de datos cambia, de tener el campo 'número de teléfono 2' (llamado en la declaración 'Tel_Contacto') cambiándolo a no nulo y luego quitando esta misma opción.

Los pasos para crear el repositorio y luego hacer éstos cambios son los siguientes:

2.2.3.1 Creando el repositorio.

Como la herramienta que elegimos es una extensión para el escritorio la creación del repositorio es muy **sencilla**. Solo se debe crear una carpeta en el equipo que llamaremos repositorio, se dará clic derecho sobre la misma y en el menú de TortoiseHg se elegirá "Crear repositorio aquí", en la ventana emergente solo se dará aceptar.

2.2.3.2 Clonando repositorio.

Para representar la forma en que dos personas trabajarían sobre un mismo repositorio vamos a crear una segunda carpeta donde vamos a clonar el repositorio. La metodología es muy parecida, solo se crea una nueva carpeta que llamaremos **repositorio clonado**, daremos clic derecho sobre ella y en el menú de TortoiseHg damos clic en "Clonar…", se mostrará una ventana como la que ilustra la Figura 2.10:

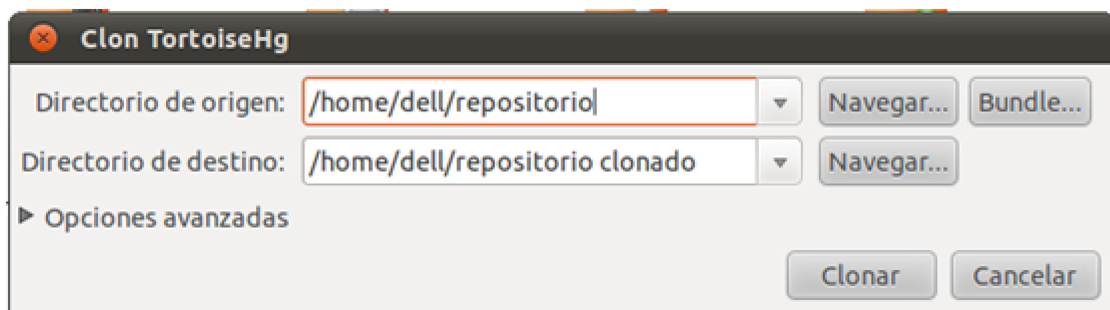

Figura 2.10. Ventana de clonación de repositorio TortoiseHG.

En la Figura 2.10 muestra dónde se especificará la carpeta **origen** del repositorio y la carpeta **destino** que es donde se clonará (se pude ayudar del botón 'Navegar' para introducir dichas direcciones).

NOTA: *dell* es el usuario personal de la figura que creé.

2.2.3.3 Creando el script.

Hasta este punto sus carpetas deben verse como se ilustra en la Figura 2.11, en caso de no serlo actualice la ventana y si aún no se ven así verifique los pasos anteriores.

Figura 2.11. Carpetas de los repositorios TortoiseHg.

En la Figura 2.11 se muestran las carpetas creadas junto con el indicador en **verde** señalando que el estado de ambas es **correcto y actualizado**.

El script que usamos tiene el siguiente texto guardado sobre un archivo llamado “Alter_Alumno.sql” que colocaremos en la **carpeta clonada**.

```
ALTER TABLE 'Escuela'.'Alumno' ALTER COLUMN 'Tel_Contacto'
VARCHAR(10) NOT NULL;
```

Ese archivo ya está en la carpeta, pero no pertenece al repositorio, así que para agregarlo debemos hacer clic derecho sobre él y en el menú de TortoiseHg elige “Añadir archivos” y sobre la ventana emergente da clic en añadir.

Hasta éste punto se ha agregado un archivo al repositorio pero solo **localmente**, lo siguiente es hacerlo disponible para que otros usuarios puedan actualizarlo en su repositorio.

2.2.3.4 Enviando archivo y actualizando el repositorio.

En este paso vamos a usar **las dos** carpetas.

En la carpeta clonada se deberá dar clic derecho y elegir “Realizar …” que sería lo equivalente a un *Commit* que mencionamos antes, la Figura 2.12 ilustra la ventana que se muestra cuando se efectúa esta acción:

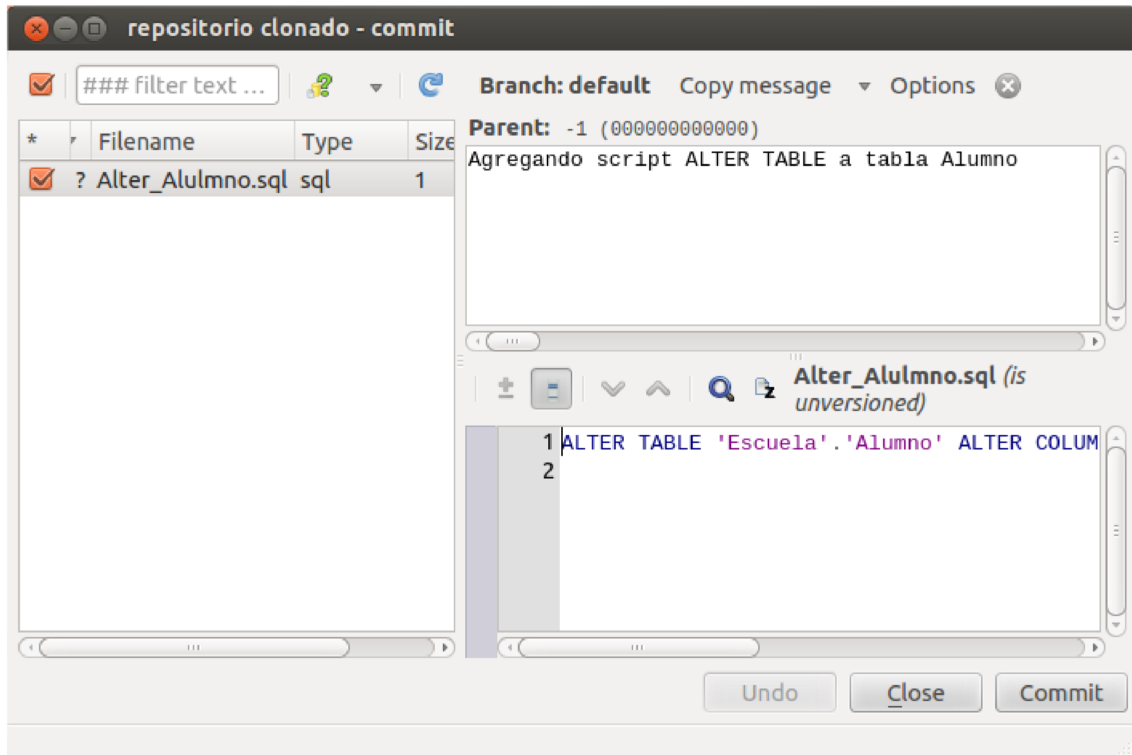

Figura 2.12. Ventana de commit 1 TortoiseHg.

En la Figura 2.12 se pueden ven en las diferentes secciones los archivos que se han modificado, las **líneas modificadas** y el archivo en el que se han hecho cambios, como éste es un archivo nuevo solo marca las líneas que contiene el archivo.

Se debe agregar un comentario para el envío como "Agregando script tabla" o alguna **frase representativa** para que sirva de referencia a la hora de revisar el historial de cambios, luego se da clic en el primer botón de izquierda a derecha para terminar la acción.

Para completar la tarea se debe dar nuevamente clic derecho sobre la **carpeta clonada**, en el menú de TortoriseHg elegir "Sincronizar", la ventana de sincronización se muestra en la Figura 2.13, ahí se debe hacer clic sobre el botón de "enviar cambios locales a repositorio seleccionado" por supuesto antes verificando que se haya seleccionado el repositorio correcto.

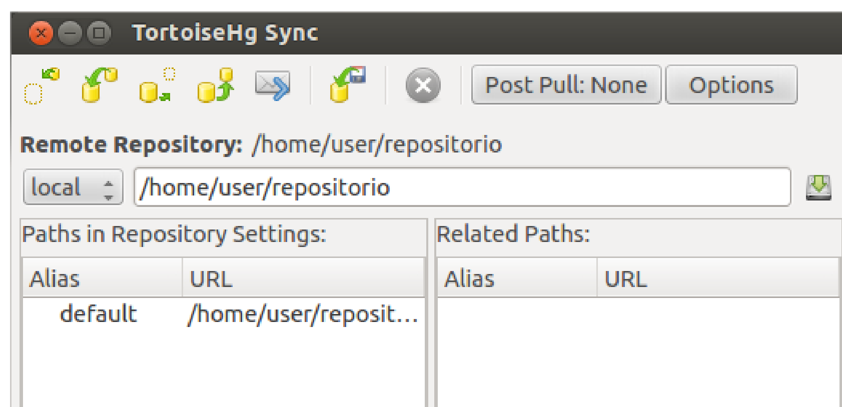

Figura 2.13. Ventana de sincronización TortoiseHg.

Cada uno de los íconos superiores de la ventana tienen una utilidad, los 2 primeros (de izquierda a derecha) son para visualizar y jalar los cambios con respecto a otro repositorio, el tercero es para visualizar los cambios que van a ser enviados a otro repositorio y el que esta resaltado es para hacer el envío.

Como se ha elegido que se envíe al repositorio que se encuentra en /home/dell/repositorio los siguientes pasos son para que los cambios se **visualicen** en dicha carpeta: ir de nuevo a la ventana que se ilustra en la Figura 2.13 pero sobre la carpeta repositorio y dar clic sobre el segundo botón de izquierda a derecha, luego clic derecho sobre la misma carpeta y elegir la opción "Explorar repositorio", lo cual mostrará una ventana como la de la Figura 2.14 en la que se debe dar clic derecho sobre la línea histórica que se desee (generalmente la primera de arriba abajo) y elegir "Update".

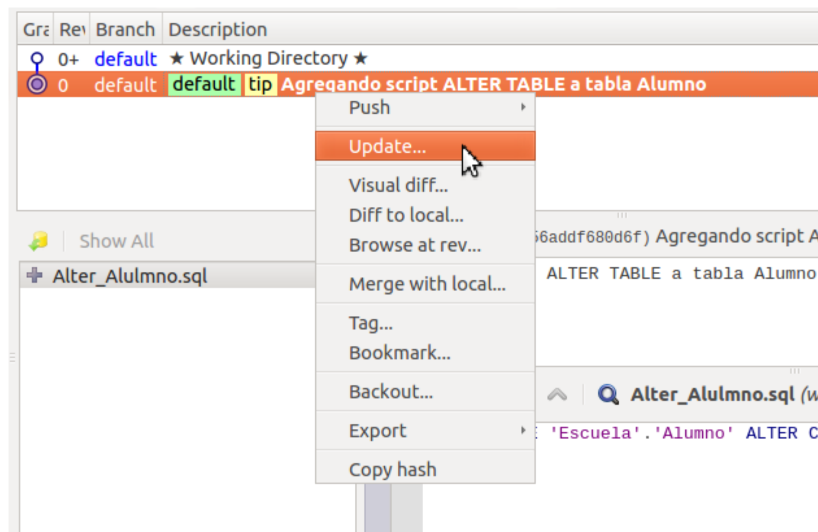

Figura 2.14. Ventana de exploración de repositorio TortoiseHg.

En la Figura 2.14 se muestra la ventana donde está el estado y el historial del repositorio, cuando el historial es más amplio se puede notar que el estado histórico actual del repositorio esta **subrayado**, en mi caso ya tenía actualizado el repositorio cuando hice la captura de pantalla.

NOTA: la acción actualizar puede variar entre herramientas, en algunas podría estar directamente al dar clic derecho sobre la carpeta del repositorio.

2.2.3.5 Modificando archivo y actualizando el repositorio.

Este paso puede parecer redundante ya que lo que se debe de hacer es casi lo mismo que se hizo en los pasos 2.2.3.3 y 2.2.3.4.

El archivo .sql debe ser modificado y guardado de modo que quede de la siguiente forma:

```
ALTER TABLE 'Escuela'.'Alumno' ALTER COLUMN 'Tel_Contacto'
VARCHAR(10) NULL;
```

Como en el paso 2.2.3.4 ya se realizaron estos pasos, solo recordamos de manera general el orden:

Elegir "Realizar…" recuerde agregar un comentario para el envío como "Quitando NOT NULL" y **revise** que los cambios sobre el archivo que se muestran sean correctos.

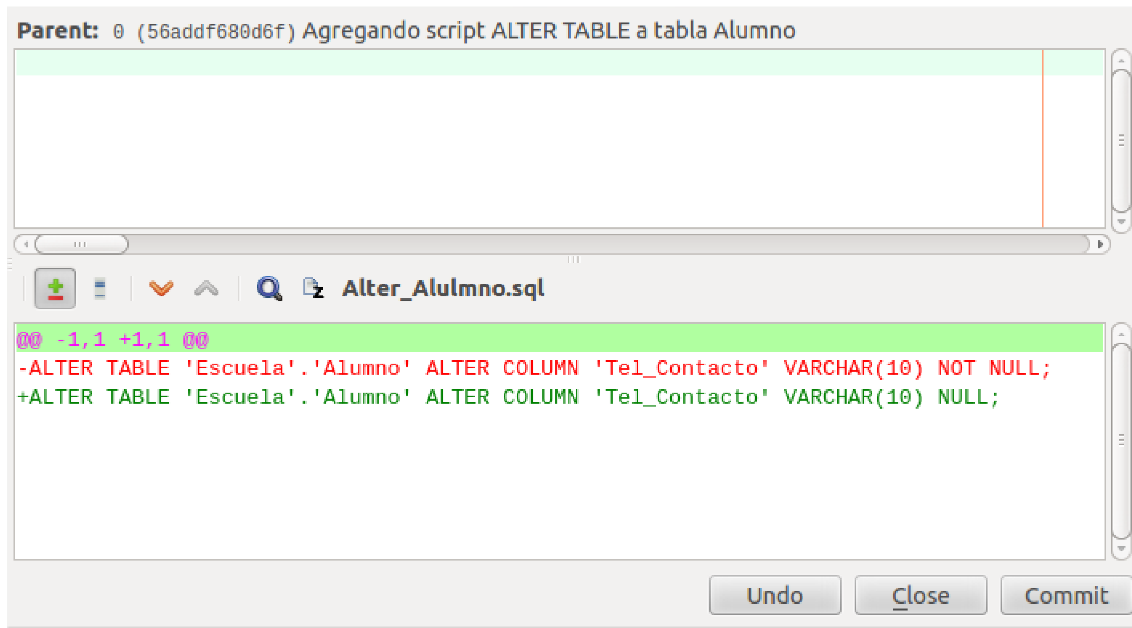

Figura 2.15. Ventana de commit 2 TortoiseHg.

En la Figura 2.15 las líneas marcadas con rojo y verde son en las que ha habido cambios, como explicamos antes las líneas donde hay **cambios** están marcadas con un + o un – al comienzo, y es el caso en el que la línea fue modificada por lo que el sistema de diferencias reconoce como una línea eliminada y una línea agregada.

Después de esto ejecuta la acción de "enviar cambios locales a repositorio seleccionado" en la ventana de **sincronización**. Y se actualiza la información en el repositorio destino como en la Figura 2.13.

Se puede constatar el cambio abriendo el archivo en el repositorio luego de hacer *update* sobre la carpeta.

Recordemos que el control de versiones ejemplifica el control de versiones sobre scripts, que es solo una pequeña parte del control de versiones, es decir, que además de lo que se hizo en ejemplo se deberían realizar las siguientes acciones:

1. Aplicar los scripts sobre la base de datos.
2. Hacer pruebas locales de los cambios.
3. Liberar para que los *testers* realicen las pruebas pertinentes.
4. Aplicar los cambios ya liberados sobre la base de datos final.

Eso sin mencionar que las acciones de *rollback* entre otros implica también acciones manuales, Al final de estos dos ejemplos listamos más conclusiones parciales sobre el control de versiones sobre bases de datos usando un SCV genérico.

A continuación ilustramos el ejemplo del control de versiones sobre scripts de bases de datos usando el SVN.

2.2.4 Ejemplo SVN

Continuando con la política de proporcionar ejemplos para novatos, hemos decidido usar herramientas gráficas también para SVN y explicar paso a paso cada una de las acciones realizadas. Por ello, para este ejemplo usamos el sistema operativo Windows y como el título dice, el SCV que se utilizó fue **Subversion**, siendo la herramienta gráfica **TortoiseSVN** la seleccionada para el ejemplo. La razón de utilizar una **herramienta gráfica** es para que los usuarios primerizos se familiaricen más rápidamente con las ideas de mencionamos aquí, además que los usuarios que prefieran usar línea de comandos pueden hacerlo con las instrucciones necesarias.

Los requisitos previos para el ejemplo es tener instalado el TortoiseSVN y tener ya creada la copia local de la base de datos.

Para ejemplificar el control de versiones usamos el mismo ejemplo que con Mercurial, dejando de lado las grandes bases de datos y scripts de funciones complicados, el ejemplo es sobre una base de datos que se llame escuela, el script es sobre la creación de una **tabla** que se llame **alumno** con los espacios: ID alumno, nombre, apellidos, edad, dirección, número de teléfono 1, número de teléfono 2 y créditos (adquiridos por asignaturas aprobadas). La estructura de la tabla sería:

```
CREATE TABLE  Alumno (ID_Alumno VARCHAR( 9 ) NOT NULL ,
      Nombre VARCHAR( 30 ) NOT NULL ,
      Apellidos VARCHAR( 30 ) NOT NULL ,
      Edad INT NOT NULL ,
      Direccion VARCHAR( 50 ) NOT NULL ,
      Tel_Casa VARCHAR( 7 ) NOT NULL ,
      Tel_Contacto VARCHAR( 10 ),
      Creditos INT NOT NULL ,
      PRIMARY KEY (  ID_Alumno ));
```

La razón hipotética de que se necesite el control de versiones es que el **requerimiento** sobre la base de datos cambia, de tener el campo 'número de teléfono 2' (llamado en la declaración 'Tel_Contacto') cambiarlo a nulo y quitar opción de nuevo. Los siguientes son los pasos para realizar el control de versiones sobre el script.

2.2.4.1 Creando el repositorio.

Como la herramienta que elegimos se integra al escritorio la creación del repositorio es muy **sencilla**. Solo se debe crear una carpeta en el equipo que llamaremos repositorio, se dará clic derecho sobre la misma y en el menú de TortoiseSVN se dará clic en "Crear repositorio aquí", en la ventana emergente mostrará la opción para crear la estructura de archivos y deberá elegirla, el programa creará las carpetas de trunk, tags y branches que estarán disponibles desde las copias locales o el explorador del repositorio.

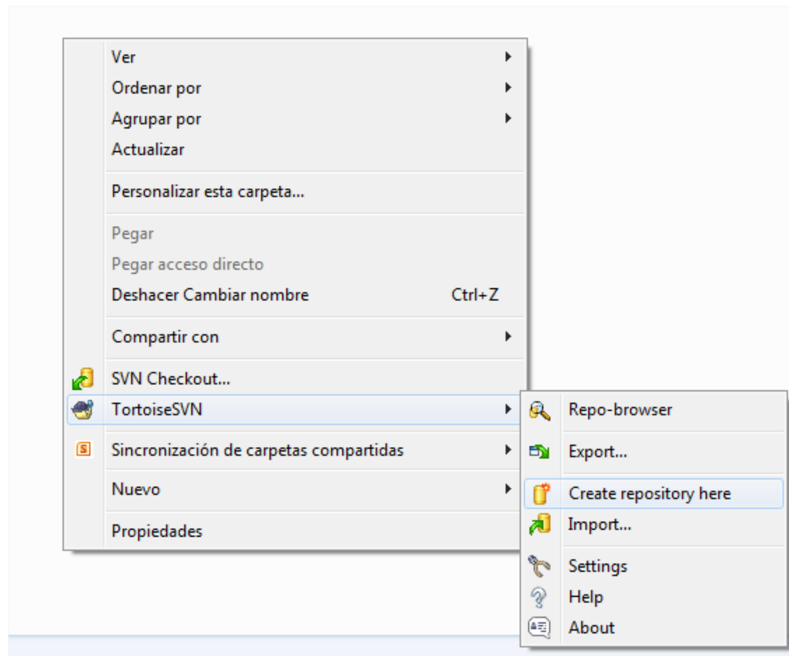

Figura 2.16. Creación de repositorio con TortoiseSVN.

La Figura 2.16 muestra el menú donde se elige la creación del repositorio.

2.2.4.2 Creando el script.

Como SVN es centralizado, no es necesario hacer un clon del repositorio como en el caso de Mercurial, a cambio de eso se debe **importar** el repositorio, para ello se debe hacer clic derecho sobre la carpeta que usted desee hacer el *checkout* y junto al menú del TortoiseSVN elegir checkout, un diálogo como el siguiente aparecerá donde ingresará el nombre del repositorio.

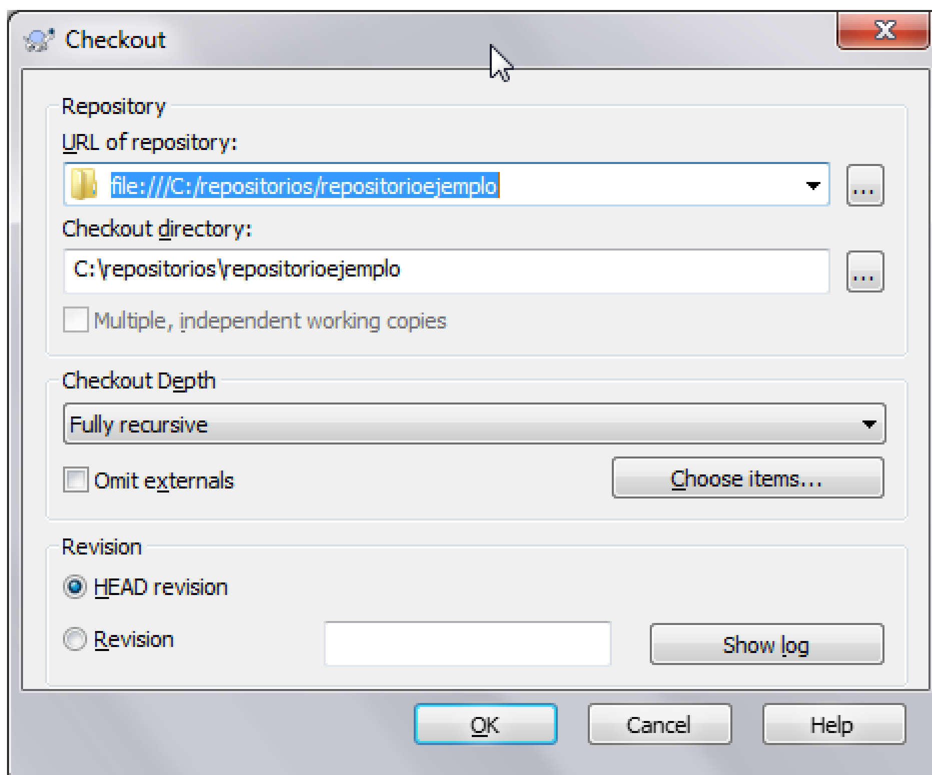

Figura 2.17. Ventana de Checkout con TortoiseSVN.

La URL del repositorio en este caso es un repositorio local por lo cual se escribe como muestra la Figura 2.17, además que es la misma dirección que **se mostró** a la hora de crear el repositorio.

Hasta este punto las carpetas importadas deben verse como se ilustra en la figura siguiente, en caso de no serlo actualice la ventana y si aún no se ven así verifique los pasos anteriores, puede que no haya aceptado la creación de la estructura de archivos.

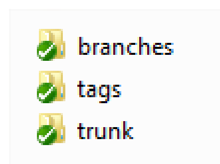

Figura 2.18. Carpetas copia local TortoiseSVN actualizadas.

En la Figura 2.18 se muestran las carpetas creadas junto con el indicador en **verde** señalando que el estado de ambas es **correcto y actualizado**.

El script que usamos tiene el siguiente texto guardado sobre un archivo llamado "Alter_Alumno.sql" que colocaremos en la **copia local** de preferencia en la carpeta trunk.

```
ALTER TABLE 'Escuela'.'Alumno' ALTER COLUMN 'Tel_Contacto'
VARCHAR(10) NOT NULL;
```

Hasta este punto, el archivo ya está en la carpeta, pero no pertenece al repositorio, así que para agregarlo debemos hacer clic derecho sobre él y en el menú de TortoiseSVN elige "Añadir archivos" y sobre la ventana emergente da clic en añadir, con eso se ha agregado un archivo al repositorio pero solo **localmente**, lo sabremos porque aparecerá un ícono con un signo de adición sobre el archivo y la carpeta del trunk aparecerá con un signo de exclamación junto a ella; lo siguiente es hacerlo disponible para que otros usuarios puedan actualizarlo en su repositorio.

Figura 2.19. Carpetas copia local TortoiseSVN desactualizadas.

La Figura 2.19 nuestra el estado en el que las carpetas están. La carpeta "trunk" aparece con un signo de exclamación indicando que no esta actualizada, dentro de ella se encuentra el archivo "tabla.sql" con el signo de adición que simboliza que **no ha sido enviado dicho cambio** al repositorio.

2.2.4.3 Enviando archivo y actualizando el repositorio.

En la carpeta clonada se deberá dar clic derecho y elegir *Commit,* comando que mencionamos antes, la Figura 2.20 ilustra la ventana que se abre cuando se efectúa esta acción:

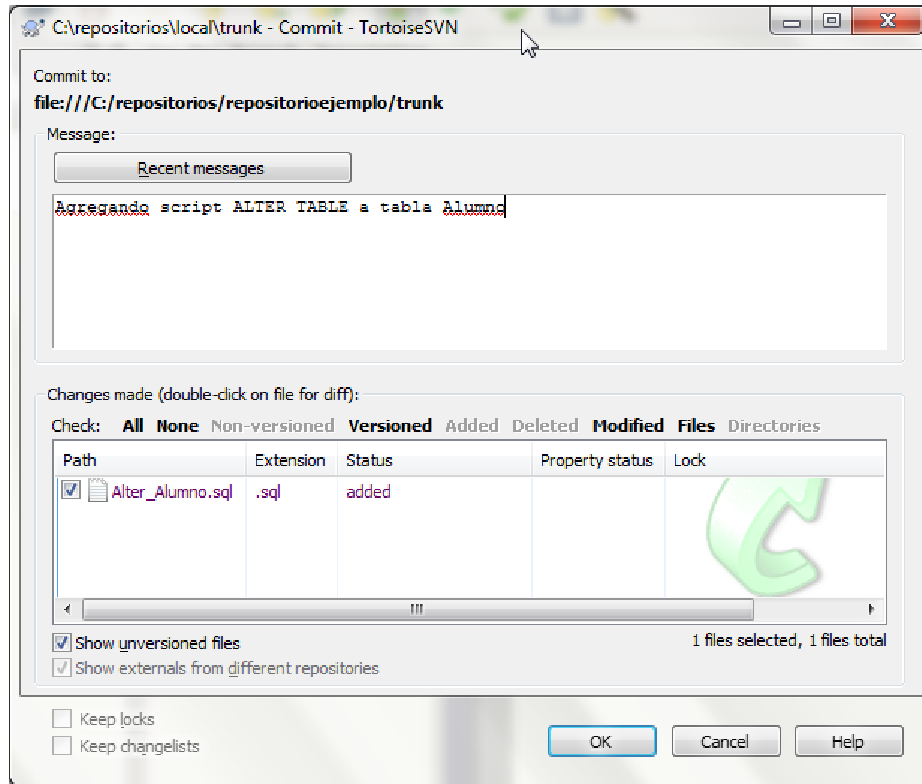

Figura 2.20. Ventana de commit 1 TortoiseSVN.

En la Figura 2.20 se muestran los cambios sobre los archivos, en este caso, solo el archivo tabla.sql con el status *added*.

Se debe agregar un comentario para el envío como “Se agregó el script de tabla” o alguna **frase representativa** para que sirva de referencia a la hora de revisar el historial de cambios, luego clic en OK para terminar la acción.

2.2.4.4 Modificando archivo y actualizando el repositorio.

El archivo .sql debe ser modificado y guardado de modo que quede de la siguiente forma:

```
ALTER TABLE 'Escuela'.'Alumno' ALTER COLUMN 'Tel_Contacto' VARCHAR(10) NULL;
```

Se hará el *commit*, y se mostrará una ventana parecida al del anterior *commit* pero ahora con un status de *modified*, si se hace doble clic sobre esa línea, se mostrará una ventana con las modificaciones hechas sobre el archivo como la siguiente:

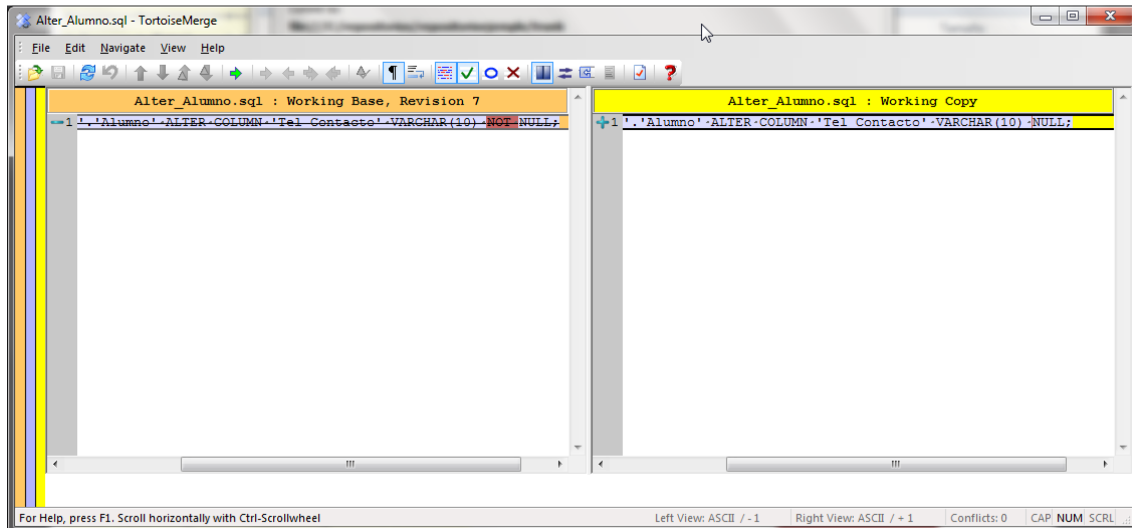

Figura 2.21. Ventana de *commit* 2 TortoiseSVN.

En la Figura 2.21 , la parte que resaltamos son las líneas en las que ha habido cambios, las líneas donde hay cambios están marcadas con un + o un – al comienzo, y en este caso, la línea fue modificada por lo que el sistema de diferencias reconoce como si se hubiera eliminado una línea completa y se hubiera agregado otra. Luego de elegir “OK” se termina la acción.

Como ya se mencionó al final del ejemplo de Mercurial, ésta es una pequeña parte del ciclo de desarrollo de software y que ayuda solamente a automatizar algunas de las acciones del control de versiones, dejando como trabajo manual el aplicar los cambios en las respectivas bases de datos a sus respectivos tiempos.

2.2.5 Conclusiones parciales del control de versiones sobre bases de datos utilizando un SCV genérico

Los SCV genéricos, al no ser creados para las necesidades ni la forma de trabajo de las bases de datos, dejan muchas actividades para realizarlas manualmente, por ejemplo, luego de controlar los cambios en los scripts, las siguientes acciones serían:

1. Aplicar los scripts sobre la base de datos.
2. Hacer pruebas locales de los cambios.
3. Liberar para que los *testers* realicen las pruebas pertinentes.
4. Aplicar los cambios ya liberados sobre la base de datos final.

Sin mencionar que otras acciones que los SCVs genéricos automatizan para el código, es poco posible hacerlo para las bases de datos, como los *rollbacks* entre otros.

3 Sistemas de control de versiones especializados

Ya describimos las clases de SCVs, analizamos las herramientas que permiten la comparación de bases de datos y la creación de delta scripts, pero ahora veremos varias herramientas **especializadas** en la administración de cambios en bases de datos y qué es lo que hacen diferente a las SCVs genéricos [30].

Los puntos a tomar en cuenta cuando hablamos de SCVs especializados, es que pueden variar dependiendo de las necesidades que se quieran satisfacer, pero de manera general, las características tomadas en cuenta son:

1. La **clase de bases de datos** para los que están hechos, que se define con sistema administrador de bases de datos (DBMS por sus siglas en inglés), tales como MySQL, SQL Server, Oracle, etc.
2. Facilidad de uso y configuración (cantidad de pasos para realizar determinada acción, configuraciones que hay que satisfacer, etc.).
3. Modo de distribución (gratuita o privativa).

Recopilando información en sitios de opinión, se obtuvo un listado de las herramientas que más se usan para administrar los cambios en bases de datos, de la que se pueden resaltar los siguientes nombres: Red Gate y DBForge por el lado de los sistemas **propietarios**, y Liquibase por el lado de los **gratuitos**. No son los únicos pero son los más representativos, de igual manera la siguiente tabla muestra una comparativa de estos.

Tabla 3.1. Sistemas especializados en el control de versiones sobre bases de datos. [31] [32]

Herramienta	Plataforma (DBMS)	Distribución	Facilidad de uso / configuración
DBForge	SQL Server, MySQL	De paga (de 50 a 200)	Depende de un repositorio de un SCV genérico, se debe crear un alias y la configuración de la base de datos.
Red Gate's tools	SQL Server, Oracle	De paga (1500 dólares)	Depende de un repositorio de un SCV genérico, se debe configurar la integración con la herramienta de admón. de base de datos de Microsoft
Liquibase	Múltiples (se especifican mas abajo)	Gratuita	Se deben crear scripts para los cambios y los comandos son por medio de línea de comandos, no se necesita configurar nada.

La Tabla 3.1 muestra un resumen de las 3 herramientas elegidas resaltando la plataforma que maneja las bases de datos y la manera de distribución. Solo se analizaron estas tres porque como ya se dijo son las **más representativas**, además que **los demás** que se analizaron tienen **características parecidas** a las del Red Gate y el DBForge (son compatibles con un solo tipo de bases de datos y necesitan un SCV genérico para trabajar además de otras configuraciones). La razón por la que no se ahondó en la herramienta de Microsoft aunque se tomara como ejemplo su ciclo de desarrollo de bases de datos, es que además de las características que mencionamos para herramientas como DBForge o Red Gate, siempre los sistemas son muy limitados a la interacción con elementos de la misma suite, por ejemplo, el SCV que utiliza para versionar es el Team Server Foundation desarrollado por ellos mismos e integrado en la suite.

A continuación se hizo una breve introducción a Red Gate y DBForge para demostrar su dependencia de un SCV genérico y las configuraciones necesarias, además se realizó un ejemplo con Liquibase igual a los presentados con Mercurial y SVN para demostrar su sencillez de uso y la nula configuración necesaria.

3.1 Introducción a Red Gate

Red Gate ofrece un conjunto de herramientas **aisladas** o agrupadas en *suites*, las que se pueden usar para el control de versiones son de 3 tipos: las que generan delta scripts, las de sincronización y las de control de código. La forma más **cómoda** y que ofrece mayores ventajas dado el contenido de la suite es usar las suites de Development con su control de código, aunque las herramientas incluidas automatizan mayor parte del control de versiones, éste método **necesita de un SCV** para realizar su trabajo, ya sea SVN o TSF (Team Server Foundation), éste último fue desarrollado por Microsoft.

Los pasos para el control de versiones sobre una base de datos con las herramientas de Red Gate son bastante simples a las que hacen utilizando solo un SCV, antes de nada, se debe crear un repositorio de Subversion apropiadamente y tener instalada la suite de desarrollo de SQL Server, se debe ligar la base de datos al SCV, esto se hace con el **SQL Server Management Studio** que se deberá instalar por separado y se debe configurar para que reconozca a Red Gate como SCV, el primer paso luego de configurar la integración, es ligar la base de datos como se muestra en la siguiente figura [33] [30].

Figura 3.1. Ventana de ligue de base de datos con SVN en Red Gate.

Como se puede ver en la Figura 3.1, se utilizó el Subversion como SCV para el control de versiones, la URL del repositorio debe referenciar a un **archivo** si es que hicimos un repositorio localmente, tal y como se hizo en la sección 2.2.4.1, dicha URL se muestra luego de que creamos el repositorio.

Luego de agregar la URL del repositorio y dar clic en Link, la base de datos debe encontrarse bajo control de versiones; al hacer cambios en la base de datos, la herramienta **automáticamente** registra cada una de las modificaciones en un listado de *commit* parecido al que manejábamos en los ejemplos con los SCVs, la siguiente figura muestra un ejemplo.

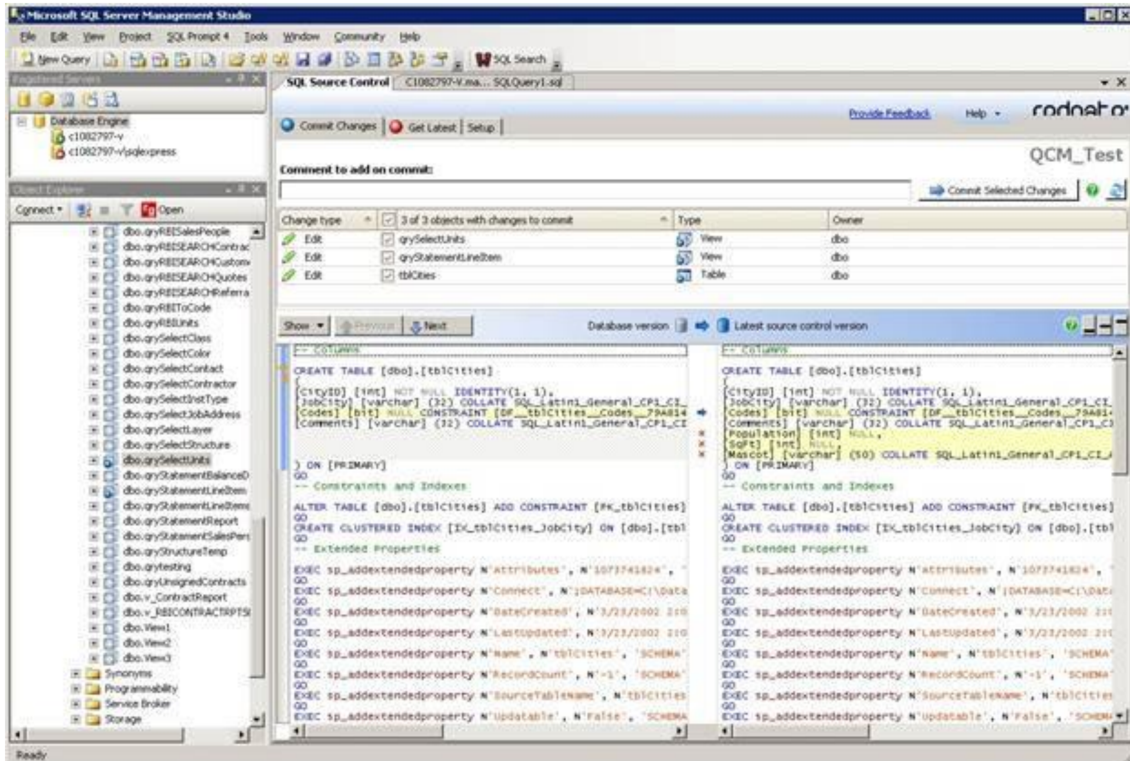

Figura 3.2. Ventana de SQL Source Control en Red Gate.

La Figura 3.2. muestra la pestaña del “SQL Source Control” donde se pueden identificar listadas las **ediciones** que se hicieron y bajo ellas los cambios que implicaron.

3.2 Introducción a DBForge

La **suite** de DBForge consta de varias herramientas, pero las principales son las de *Schema Compare* y *Data Compare* que tienen sus versiones para SQL Server y para MySQL [31]. Al igual que las de Red Gate, el DBForge **utiliza como base un SCV** para llevar a cabo el control de versiones, en este caso es el SVN.

Lo primero que hay que hacer es crear un alias para la conexión con SVN, la siguiente figura muestra la ventana para ello.

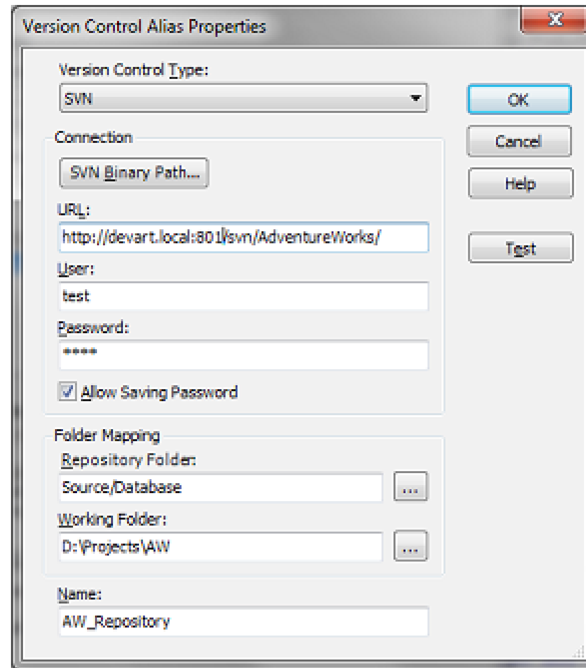

Figura 3.3. Ventana de creación de alias en DBForge.

La Figura 3.3. muestra la ventana donde se deben llenar los datos para la conexión con el SCV.

Ya teniendo el **alias** para la conexión, se debe crear una versión base de la base de datos (*snapshot*), los datos necesarios son el alias y los datos de la base de datos, algo importante que hay que remarcar es que el control de versiones lo hace con archivos **binarios** que crea el programa (con extensión .snap).

Luego de realizar cambios en la base de datos, se utiliza el *schema compare tool* o el *data compare tool* utilizando el *snapshot* obtenido y el estado actual de la base de datos para determinar los cambios que se hicieron en la base de datos, para hacer el envío de los cambios hay dos formas: mediante **delta script** o sincronizando con el **sistema de control de versiones** [31]**.** La siguiente figura muestra ambas ventanas, a la izquierda el *compare tool* y a la derecha la sincronización.

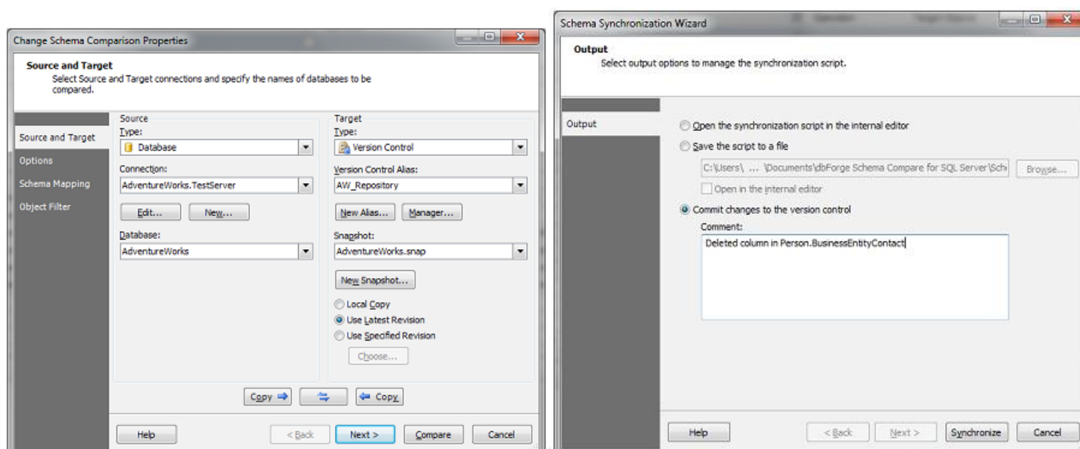

Figura 3.4. Ventanas de comparación y sincronización en DBForge.

En la ventana de comparación se puede delimitar la información necesaria en dos partes, la parte izquierda que contiene la fuente de la comparación (el estado actual de la base de datos) y del lado derecho el objetivo de la comparación (en este caso el repositorio de SVN que creamos).

3.3 Liquibase

Es una de las herramientas más representativas del ámbito y que ha tenido mucho revuelo últimamente, desarrollada por Nathan Voxland. En el **2006** se liberó la aplicación y el código fuente al público, antes de eso sólo se usaba dentro de la empresa de su creador y fue en el 2007 cuando se llegó a la primera versión estable (la versión 1.0), siendo la versión más actual al momento de la redacción la **1.3**.

Liquibase es una herramienta que ataca varias **necesidades esenciales** en el desarrollo de bases de datos, antes que nada ya sabemos que los proyectos de software son cambiantes, si las bases de datos son una parte esencial de los proyectos se software por lo tanto también son cambiantes; los sistemas han crecido exponencialmente durante los últimos años, para ello, bases de datos más extensas han sido utilizadas y un solo desarrollador de bases de datos no es la opción más eficiente [32]. Liquibase se encarga del control de cambios sobre bases de datos, permitiendo el **involucramiento de varios desarrolladores**, administrando ramificaciones y el *merging* además que da la facilidad de que se apliquen los cambios directamente en la base de datos o se creen **scripts** para la sincronización futura.

Como ya resolvimos durante el análisis de las herramientas *diff* o basadas en él, las ventajas al usarlas es que se hacen los cambios y te preocupas del control de versiones hasta que lo consideras, pero un problema es que las herramientas *diff* no entienden la semántica de las bases de datos. Liquibase **no es una herramienta diff** porque aunque incluye una como herramienta de **apoyo**, se encarga de administrar los cambios en **semántica** de base de datos sin la necesidad de ser apoyado por un SCV [32].

Las características de Liquibase son:

- Gratuito (código abierto).
- Ajeno a plataformas específicas. Ya sea de sistema operativo (utiliza Java) o tipo de base de datos (utiliza conexiones JDBC).
- Hace control de versiones solo sobre la base de datos y no de los scripts en si.
- Se ejecuta sobre línea de comandos.
- Cuenta con complementos instalables.
- Integrable mediante plugins a IDEs (Entornos de desarrollo integrado)

Liquibase tiene muchas **ventajas**, pero una muy importante es su **compatibilidad** con distintas bases de datos, la siguiente lista muestra las bases de datos con que es compatible [32].

- MySQL
- PostgreSQL
- Oracle
- MS-SQL
- Sybase Enterprise
- Sybase Anywhere
- DB2

- Apache Derby
- HSQL
- H2
- Informix
- InterSystems Caché
- Firebird
- SAPDB
- SQLite

Aunque si se desea trabajar con una base de datos no soportada, es posible porque esta desarrollado sobre la **Conectividad a Bases de Datos de Java** (JDBC), pero aunque para Liquibase sea posible, no significa que lo vaya a hacer sin problemas, para eso ayuda mucho su adaptabilidad permitiendo que además de los parámetros por defecto se puedan adecuar de manera que la base de datos lo entienda, por ejemplo, un problema común con su uso en bases de datos no compatibles es que las etiquetas de refactorización/cambio que tiene por defecto no son compatibles, pero para eso se pueden modificar los comandos de **etiquetado** con la operación "<sql>" para crear una etiqueta que la base de datos entienda [32] [34].

La forma en la que Liquibase mantiene los cambios es con una tabla **DATABASECHANGELOG** dentro de la base de datos que guarda la información de los cambios, esta tabla se crea automáticamente pero si por alguna razón ocurre algún problema (por ejemplo, por el tipo de base de datos) se puede **crear manualmente**, su declaración debería quedar de la siguiente forma:

```
CREATE TABLE DATABASECHANGELOG (id varchar(150) NOT NULL,
author varchar(150) NOT NULL,
filename varchar(255) NOT NULL,
dateExecuted datetime NOT NULL,
md5sum varchar(32),
description varchar(255),
comments varchar(255),
tag varchar(255),
liquibase varchar(10),
PRIMARY KEY(id, author, filename))
```

Figura 3.5. Código de creación de la tabla DATABASECHANGELOG de Liquibase [32].

Los campos que contiene la tabla son los necesarios para mantener un control de los cambios almacenando datos **precisos e irrepetibles** que ayuden al seguimiento de las modificaciones.

Otra tabla que se crea con el Liquibase es la tabla DATABASECHANGELOGLOCK cuyo propósito es evitar que dos usuarios modifiquen la base de datos a la vez, si no se hace automáticamente ésta puede ser creada con un comando específico de Liquibase.

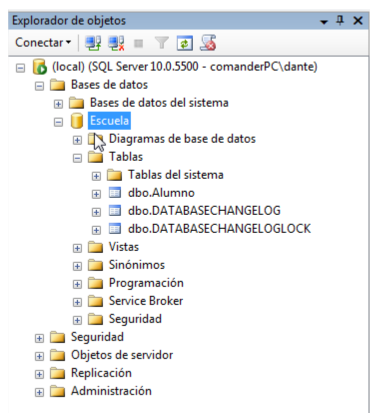

Figura 3.6. Tablas agregadas automáticamente por Liquibase.

La Figura 3.6. muestra las dos nuevas tablas que creó Liquibase a la hora de la primera ejecución, dbo.DATABASECHANGELOG y dbo.DATABASECHANGELOGLOCK.

En cuanto a la forma en que trabaja Liquibase, las **ventajas** son que los cambios son fáciles de entender, soporta las ramificaciones y el *merging,* y que conserva la **semántica** de las bases de datos, además tiene una **desventaja** principal que es que la **forma de trabajar** con las bases de datos es diferente, lo que implica una curva de aprendizaje y experimentación al comenzar a aplicar la herramienta en proyectos; y aunque existen *plugins* para IDEs como Netbeans o IntelliJ, su **integración** depende de otro *plugin* llamado "DBHelper", lo cual puede causar algunos problemas y además éste tipo de *plugins* de Liquibase, no ofrecen todas las funcionalidades que se obtienen utilizando la línea de comandos. Además de las integraciones con IDEs existe la **posibilidad** de implementaciones sobre Ant, Maven, Spring, Grails y Server Listener para apoyar el desarrollo, por ejemplo éste último se podría utilizar para utilizar Liquibase sobre una base de datos de una aplicación ya en la nube. La **más efectiva** es la utilización mediante línea de comandos ya que ofrece la funcionalidad completa de Liquibase y no presenta problemas de integración porque no necesita nada más para ejecutarse, más adelante mostraremos la forma de utilizar Liquibase mediante este método.

Liquibase trabaja usando **archivos .xml** que manejan los cambios, dicha administración ofrece un **rastreo independiente**, los cambios en la bases de datos se crean en archivos llamados ***ChangeLogs*** en formato xml y que dentro de ellos los cambios o refactorizaciones (una o varias por *ChangeSet*) se delimitan por ***ChangeSets***, hay que recordar que Liquibase incluye 34 refactorizaciones **precargadas**; el versionamiento de Liquibase es parecido al que se hace con esquemas con otras herramientas, con la diferencia que no hay una versión de esquema particular. Una característica interesante acerca de los *ChangeLogs* es que aceptan inclusiones, es decir, es posible dividir los *ChangeSets* en varios archivos mejorando la **mantenibilidad** de la base de datos. Los *ChangeSets* tienen varios atributos, como el Sistema Administrador de Bases de Datos (DBMS), el nombre del desarrollador, además llevan un identificador único que no debe repetirse para un mismo usuario, etc.

Un ejemplo básico de un archivo *ChangeLog* sería como el siguiente.

```
ChangeLog.xml
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <databaseChangeLog
4    xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
5    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6    xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
7           http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-2.0.xsd">
8      <changeSet id="1" author="dante">
9          <createTable tableName="Alumno">
10             <column name="Nombre" type="varchar (30)">
11                 <constraints primaryKey="true" nullable="false"/>
12             </column>
13             <column name="Apellidos" type="varchar(30)">
14                 <constraints nullable="false"/>
15             </column>
16             <column name="Edad" type="int">
17                 <constraints nullable="false"/>
18             </column>
19             <column name="Direccion" type="varchar(50)">
20                 <constraints nullable="false"/>
21             </column>
22             <column name="Tel_Casa" type="varchar(7)">
23                 <constraints nullable="false"/>
24             </column>
25             <column name="Tel_Contacto" type="varchar(10)">
26             </column>
27             <column name="Creditos" type="varchar(10)">
28                 <constraints nullable="false"/>
29             </column>
30         </createTable>
31     </changeSet>
32 </databaseChangeLog>
33
```

Figura 3.7. Ejemplo de ChangeLog de Liquibase.

La Figura 3.7. contiene el código necesario para agregar la tabla ‘Alumno’ con la que trabajamos en los ejemplos de Mercurial y SVN. Las etiquetas que delimitan el *ChangeLog* son las de databaseChangeLog y las que delimitan los *ChangeSets* llevan ese nombre. En el ejemplo se crea un *ChangeSet* con un ID “1” y con un autor “Desarrollador1”, el cambio es la adición de una tabla ‘Alumno’ igual al que se usó en los ejemplos pasados sobre la base de datos ‘Escuela’, las declaraciones de las columnas van delimitadas por las etiquetas “column”, una posible **buena practica** sería usar las etiquetas “comment” para agregar comentarios a cada uno de los *ChangeSets*, un ejemplo de la adición de comentarios sería:

<comment>
Adición de una table Departamente
</comment>

Para **aplicar los cambios** hay dos formas, una es mediante el comando "*Migrate*" para aplicar los cambios **directamente** en la base de datos y otra manera es con el comando MigrateSQL que exporta los cambios a un archivo con extensión SQL que sirva de manera parecida a los **delta scripts** que abordamos anteriormente.

La forma mediante **línea de comandos** de ejecutar el Liquibase es mediante un conjunto de parámetros que incluyen el **jdbc driver**, url de la base de datos y datos de autentificación, entre otros, por ejemplo para el comando Migrate que mencionamos:

```
liquibase --driver=com.mysql.jdbc.Driver \
     --classpath=/path/to/classes \
     --changeLogFile=com/example/db.changelog.xml \
     --url="jdbc:mysql://localhost/example" \
     --username=user \
     --password=asdf \
     migrate
```

Figura 3.8. Comandos para la ejecución de acciones en Liquibase [35].

Se comienza con el comando Liquibase, se pasan los parámetros que mencionamos separados por diagonales inversas (\), nombre y localización del driver, la dirección del archivo *ChangeLog*, dirección y autenticación de la base de datos, y el último parámetro es el **comando** que deseamos ejecutar. Por razón de legibilidad la Figura 3.8. no fue obtenida de una ejecución local del programa, más adelante se mostrarán tomas de pantalla de éste tipo.

Retomando el ejemplo utilizado en las secciones 2.2.3 y 2.2.4 que son de los ejemplos con los SCVs genéricos, donde teníamos una tabla 'Alumno' creada con el siguiente script.

```
CREATE TABLE  Alumno (ID_Alumno VARCHAR( 9 ) NOT NULL ,
      Nombre VARCHAR( 30 ) NOT NULL ,
      Apellidos VARCHAR( 30 ) NOT NULL ,
      Edad INT NOT NULL ,
      Direccion VARCHAR( 50 ) NOT NULL ,
      Tel_Casa VARCHAR( 7 ) NOT NULL ,
      Tel_Contacto VARCHAR( 10 ),
      Creditos INT NOT NULL ,
      PRIMARY KEY (  ID_Alumno ));
```

Para modificar el campo 'Tel_Contacto' con Liquibase no se necesita más que crear un archivo ChangeLog.xml y lo vamos a hacer de **dos formas**: la primera es con **comandos exclusivos de Liquibase** donde eliminaremos la columna y luego la crearemos con la forma deseada, y la segunda es con la **etiqueta <sql>** donde usaremos comandos SQL para hacer la modificación.

El *ChangeLog* para la primera forma quedaría como sigue:

```
ChangeLog3.xml
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <databaseChangeLog
4    xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
5    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6    xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
7           http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-2.0.xsd">
8      <changeSet id="2" author="Desarrollador1">
9          <comment>
10             Cambiando Tel_Contacto a Not Null
11         </comment>
12         <dropColumn tableName="Alumno" columnName="Tel_Contacto"/>
13         <addColumn tableName="Alumno">
14             <column name="Tel_Contacto" type="varchar(10)">
15                 <constraints nullable="false"/>
16             </column>
17         </addColumn>
18     </changeSet>
19 </databaseChangeLog>
```

Figura 3.9. ChangeLog de actualización de columna con comandos de Liquibase.

Como se puede ver en la Figura 3.9., utilizamos las etiquetas de comentario, **eliminación** de columna y **adición** de columna, como dice en la sección de comentario, estamos cambiando el campo 'Tel_Contacto' de *null* a no *null* especificando esto último con la etiqueta **<constraints>.**

La ejecución en línea de comandos se hizo en **una sola línea** (a diferencia del ejemplo de arriba) y en la siguiente figura se muestra una impresión de pantalla con los comandos y los mensajes que retorna Liquibase.

```
C:\Windows\system32\cmd.exe
        at com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser.parse(Un
known Source)
        at com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParser.p
arse(Unknown Source)
        at liquibase.parser.core.xml.XMLChangeLogSAXParser.parse(XMLChangeLogSAX
Parser.java:98)
        ... 3 more

For more information, use the --logLevel flag)

C:\repositorios\liquibase\liquibase-2.0.5-bin>liquibase --driver=com.microsoft.s
qlserver.jdbc.SQLServerDriver --classpath=/repositorios/liquibase/liquibase-2.0.
5-bin/drivers/sqlserver/sqljdbc4.jar --changeLogFile= /repositorios/liquibase/Ch
angeLog3.xml --url="jdbc:sqlserver://localhost;databaseName=Escuela" --username=
Usuario --password=pass migrate
INFO 12/08/12 02:34 PM:liquibase: Successfully acquired change log lock
INFO 12/08/12 02:34 PM:liquibase: Reading from [dbo].[DATABASECHANGELOG]
INFO 12/08/12 02:34 PM:liquibase: Reading from [dbo].[DATABASECHANGELOG]
INFO 12/08/12 02:34 PM:liquibase: ChangeSet /repositorios/liquibase/ChangeLog3.x
ml::2::Desarrollador1 ran successfully in 91ms
INFO 12/08/12 02:34 PM:liquibase: Successfully released change log lock
Liquibase Update Successful

C:\repositorios\liquibase\liquibase-2.0.5-bin>_
```

Figura 3.10. Ejecución en línea de comandos de Liquibase.

En éste caso, la base de datos a la que se ligó el cambio tiene como DBMS el SQL SERVER de Microsoft, por lo que se necesitó un **conector jdbc específico**, el nombre del driver que se utilizó fue ‘com.microsoft.sqlserver.jdbc.SQLServerDriver’ y se direccionó al archivo .jar: sqljdbc4.jar.

Las líneas utilizadas **exactas** (separadas para facilitar su lectura) fueron:

liquibase --driver=com.microsoft.sqlserver.jdbc.SQLServerDriver
--classpath=/repositorios/liquibase/liquibase-2.0.5-bin/drivers/sqlserver/sqljdbc4.jar
--changeLogFile= /repositorios/liquibase/ChangeLog4.xml
--url="jdbc:sqlserver://localhost;databaseName=Escuela"
--username=Usuario
--password=pass
migrate

Luego de la ejecución, las columnas de la tabla quedaron de la siguiente forma.

- dbo.Alumno
  - Columnas
    - Nombre (PK, varchar(30), No NULL)
    - Apellidos (varchar(30), No NULL)
    - Edad (int, No NULL)
    - Direccion (varchar(50), No NULL)
    - Tel_Casa (varchar(7), No NULL)
    - Creditos (varchar(10), No NULL)
    - Tel_Contacto (varchar(10), No NULL)

Figura 3.11. Estado de las columnas luego del primer cambio con Liquibase.

Como se puede notar, luego de aplicar el *changelog* de la Figura 3.11Figura 3.9., el campo ‘Tel_Contacto’ es ahora “no nulo”.

La segunda forma con la que se puede editar la tabla es, como ya se dijo, mediante **comandos SQL** y utilizando la etiqueta con el mismo nombre, de modo que la *ChangeLog* quedaría de la siguiente manera.

changeLog4.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>

<databaseChangeLog
  xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
         http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-2.0.xsd">
    <changeSet id="2" author="Desarrollador1">
        <comment>
            Cambiando Tel_Contacto a Null de nuevo
        </comment>
        <sql>ALTER TABLE Alumno ALTER COLUMN Tel_Contacto VARCHAR(10) NULL;</sql>
    </changeSet>
</databaseChangeLog>
```

Figura 3.12. ChangeLog de actualización de columna con comandos SQL en Liquibase.

Los comandos propios de SQL que se utilizaron fueron el ALTER TABLE y el ALTER COLUMN y **se aplicaron con éxito**, lo que indica que cualquier modificación puede ser soportada por Liquibase aun cuando no tenga instrucciones específicas que él entienda, solo se necesita utilizar la etiqueta <sql>.

El seguimiento de los cambios quedó **guardado** en una de las tablas que creó el Liquibase en la primera ejecución y la siguiente figura muestra algunos de los datos de la tabla.

SQLQuery2.sql - (...derPC\dante (53))* | Detalles del Explorador de objetos

```sql
/****** Script para el comando SelectTopNRows de SSMS ******/
SELECT ID, AUTHOR, DATEEXECUTED,COMMENTS, DESCRIPTION
  FROM [Escuela].[dbo].[DATABASECHANGELOG]
```

Resultados | Mensajes

	ID	AUTHOR	DATEEXECUTED	COMMENTS	DESCRIPTION
1	1	Desarrollador1	2012-08-12 14:13:47.233	Creacion de la tabla 'Alumno'	Create Table
2	2	Desarrollador1	2012-08-12 14:34:26.620	Cambiando Tel_Contacto a Not Null	Drop Column, Add Column
3	2	Desarrollador1	2012-08-12 14:40:46.667	Cambiando Tel_Contacto a Null de nuevo	Custom SQL

Figura 3.13. Registro de cambios en la tabla DATABASECHANGELOG de Liquibase.

En la tabla, quedaron plasmados los cambios hechos, desde la creación de la tabla (que hicimos con Liquibase pero no presentamos en el documento), hasta la última acción donde usamos comandos SQL, en todas ellas ofrece además una columna de descripción que Liquibase crea a base de los comandos que se utilizaron para hacer el cambio.

Liquibase cuenta con más características dignas de mencionarse, una es la habilidad de aplicar **retrocesos** (rollbacks) en las bases de datos hacia 3 puntos en el pasado: a un *tag* específico, cierta cantidad de pasos atrás o una fecha en específico; Liquibase incorpora una herramienta Diff que aunque no la utiliza en el funcionamiento principal del programa, se utiliza como **respaldo** para el control de cambios permitiendo obtener una comparativa para mejorar la abstracción de los cambios en caso de necesitar realizar un análisis; la última de las herramientas remarcables es la llamada **DBDoc** que es muy parecida a la javaDoc para Java, ésta guarda información de la base de datos a modo de recopilado.

El trabajo de desarrollo para la **nueva versión** de Liquibase (1.4) promete mejoras importantes como la compatibilidad con plataformas .net de Microsoft y la adición de más refactorizaciones.

Por otro lado, la **idea** de desarrollar una herramienta gráfica que implemente Liquibase a la vez que no pierda su dinamicidad sería importante, desarrollada para cualquier sistema operativo ya sea mediante **Java** o alguna plataforma web como **PHP** y permitiendo la fácil

interacción con el usuario mediante una interfaz gráfica que ofrezca la funcionalidad completa de la refactorización con Liquibase.

Como conclusiones parciales sobre la herramienta de Liquibase se puede deducir que **es suficiente para cumplir con el ciclo completo de desarrollo de las bases de datos**; si retomamos lo pasos que describieron y que tomamos del proceso de desarrollo que se utiliza en la suite de Microsoft, Liquibase cumple con cada una de las partes, para ejemplificarlo presentamos la siguiente tabla.

Tabla 3.4. Fases del ciclo de desarrollo de bases de datos que cubre Liquibase.

<table>
<tr><th>Ciclo de desarrollo de bases de datos</th><th>Liquibase</th></tr>
<tr><td>Establecer el entorno del proyecto</td><td>No se necesita establecer un entorno del proyecto porque lo único que se necesita es tener la autentificación para trabajar sobre la base de datos, las tablas de registro de cambios son creadas automáticamente</td></tr>
<tr><td>Realizar un trabajo de desarrollo iterativo aislado</td><td>Permite el trabajo colaborativo y aislado mediante un mecanismo de control exclusivos.</td></tr>
<tr><td>Generar el proyecto</td><td rowspan="2">No se necesita porque los cambios son aplicados directamente a la base de datos y en caso de hacerse cambios incorrectos permite “retrocesos” a versiones más estables.</td></tr>
<tr><td>Realizar la implementación desde el entorno del proyecto</td></tr>
</table>

Como se puede notar en la tabla, Liquibase cubre por completo el ciclo de desarrollo de bases de datos, esto es principalmente porque aplica los cambios directamente sobre la base de datos, es decir, no necesita generar un entorno de trabajo colaborativo, no se necesita generar el proyecto para aplicar los cambios ni realizar la implementación.

4 Conclusiones y trabajo futuro

El utilizar un Sistemas de Control de Versiones (SCV) genéricos (sea del tipo que sea como se comprobó en las secciones de ejemplo con SVN y Mercurial) como herramienta para el control de versiones de bases de datos no satisfacen las necesidades del desarrollo de bases de datos, dado que éstos sistemas no fueron creados para ello, lo cual provoca que se deba utilizar un proceso complementario, y por lo general manual, que ligue los cambios con las bases de datos, además de involucrar el factor "comunicación" entre el equipo de desarrollo para conservar la estabilidad del proceso.

La mayoría de las herramientas especializadas en el control de versiones sobre bases de datos, como Red Gate y DBForge, usan SCVs genéricos para llevar a cabo el control de versiones, solo automatizando algunas tareas, lo cual disminuye la versatilidad de los sistemas y que pudieran ser sustituidas por prácticas personales y herramientas más sencillas. Además éstas herramientas especializadas que utilizan un SCV como apoyo, tienen otro obstáculo muy marcado, que es la compatibilidad con tipos de bases de datos, es decir, se especializan en un tipo de bases de datos en específico dejando fuera a los demás tipos de bases de datos, aunque por otro lado, su especialización en un tipo de base de datos les permite ofrecer al usuario cierta comodidad porque compensan esa deficiencia con herramientas gráficas que facilitan la aplicación de los cambios.

El SCV especializado Liquibase, propone un modo de trabajo diferente y prometedor, además de ser gratuito y altamente compatible con las bases de datos relacionales, ofrece una forma segura, administrada y cooperativa de trabajar sobre bases de datos evolutivamente, además su facilidad de uso y fiabilidad son muy altos, puesto que la configuración necesaria es casi nula y no depende de otras herramientas para llevar a cabo su trabajo. El único problema visible de la herramienta es que implica una fuerte inversión de capacitación por ser relativamente nueva, proponer una forma diferente de trabajo y no lo suficiente madura para ofrecer entornos en los que cualquier desarrollador aunque fuese novato pueda familiarizarse rápidamente, es decir, los cambios deben codificarse en vez de crearse mediante una herramienta gráfica. En cuanto a la cobertura del ciclo de desarrollo de bases de datos por parte de Liquibase es total, ya que, al trabajar directamente sobre la base de datos, permitir el trabajo colaborativo, ofrecer un historial de cambios completo y contener mecanismos de control de retroceso (*rollbacks*) entre otras herramientas, abarca por completo las necesidades comunes que pudiera exigir el desarrollo de una base de datos.

El trabajo futuro de la investigación sería enfocado a la herramienta Liquibase, e iría dirigido a dos rubros:

- Mejoras en la forma de trabajo. Analizar la manera de crear los comandos necesarios para trabajar de manera más aislada con el control de versiones, o dicho de otra manera, buscar la forma en la que Liquibase pueda crear, trabajar e implementar copias locales de bases de datos para eliminar la necesidad de mantener conexión directa con la base de datos; aunque esto es posible, no es un proceso que actualmente automatice el Liquibase. La versión actual de la herramienta solo considera, como trabajo distribuido, la creación de los scripts.
- Analizar la manera y el procedimiento necesario para crear una herramienta gráfica que sin disminuir la autonomía (sin necesitar la integración o intervención de otras

herramientas como ocurre en el caso de DBForge y Red Gate) y funcionalidad de Liquibase (permitiendo la conexión con diferentes tipos de bases de datos gracias a drivers jdbc), administre los cambios de las bases de datos, sin olvidar que la herramienta debe ser multiplataforma (java o PHP por ejemplo).

5 Bibliografía

1. Allen, S. OdeToCode, 2008. Versioning Databases. http://odetocode.com/Blogs/scott/archive/2008/02/04/versioning-databases-branching-and-merging.aspx (accessed Julio 18, 2012).

2. Allen, K. S. Five part series on the philosophy and practice of database version control, 2008. http://odetocode.com. http://odetocode.com/blogs/scott/archive/2008/01/30/three-rules-for-database-work.aspx.

3. Mooney, M. Simple Talk, 2010. when database source control goes bad. http://www.simple-talk.com/sql/t-sql-programming/when-database-source-control-goes-bad/ (accessed Julio 3, 2012).

4. Ruiz-Berton, F. J.; Zaragoza-Soria, F. J. Universidad de Zaragoza, 2010. El Control de Versiones en el aprendizaje de la Ingeniería. http://bioinfo.uib.es/~joemiro/aenui/procJenui/Jen2007/ruelco.pdf (accessed Junio 18, 2012).

5. Mason, M. *Pragmatic Version Control with Subversion,* 3rd ed.; The Pragmatic Programmers, 2006; Vol. 1.

6. Scott, J. Joseph Scott. http://joseph.randomnetworks.com/2007/01/03/subversion-success-story/ (accessed Julio 2012).

7. en.wikipedia.org, 2011. List of revision control software. http://en.wikipedia.org/wiki/List_of_revision_control_software (accessed Junio 22, 2011).

8. Kunene, G.; Dhandhania, A. www.developer.com, 2010. Subversion vs. Git: Choosing the Right Open Source Version Control System. http://www.developer.com/open/article.php/3902371/Subversion-vs-Git-Choosing-the-Right-Open-Source-Version-Control-System.htm (accessed Julio 13, 2011).

9. Chacon, S. *Pro Git,* 1st ed.; Apress, 2009; Vol. 1.

10. sandofsky. Understanding the Git workflow. http://sandofsky.com/ (accessed Julio 10, 2012).

11. www.chuidiang.com, 2009. Sistema de control de versiones. http://chuwiki.chuidiang.org/index.php?title=Sistema_de_control_de_versiones (accessed Julio 27, 2011).

12. Oviedo, J. juanoviedo. wordpress, 2010. Juan Oviedo. http://juanoviedo.files.wordpress.com/2010/09/control-de-versiones.pdf (accessed Junio 20, 2012).

13. De Alwis, B.; Sillito, J. Why are software projects moving from centralized to decentralized version control systems? *CSE Workshop on Cooperative and Human Aspects on Software Engineering*, 2009; pp 36-39.

14. Ploski, J. e. a. Introducing Version Control to Database-Centric Applications in a Small Enterprise. *IEEE Software* **2007,** 38-44.

15. Collins-Sussman, B.; Fitzpatrick, B. W.; Pilato, M. Version Control with Subversion. *O'Reilly Media* **2008,** 432.

16. en.wikipedia.org, 2011. Software versioning. http://en.wikipedia.org/wiki/Software_versioning (accessed Junio 22, 2011).

17. en.wikipedia.org. http://en.wikipedia.org/ (accessed Junio 22, 2011).

18. es.wikipedia.ogr, 2011. Control de versiones. http://es.wikipedia.org/wiki/Control_de_versiones (accessed Junio 23, 2011).

19. Pérez, J.; Ferrer, J. AgileSpain, 2012. http://www.willydev.net/descargas/WillyDev_Refactorizacion.pdf (accessed Julio 20, 2012).

20. Hayase, Y.; Matsushita, M.; Inoue, K. Revision control system using delta script of syntax tree. *Proceedings of the 12th international workshop on Software configuration management* **2005,** *1* (1), 133-149.

21. Downs, K. The Database Programmer en Blogspot, 2008. Database Development: Table Structure Changes. http://database-programmer.blogspot.mx/2008/02/database-development-table-structure.html (accessed Junio 5, 2012).

22. Hex, E. Google Code - Tarantino, 2010. DatabaseChangeManagement. http://code.google.com/p/tarantino/wiki/DatabaseChangeManagement (accessed Junio 13, 2010).

23. Carter, J. Multipart mixed, 2008. SVN vs. Mercurial vs. Git For Managing Your Home Directory. http://joshcarter.com/productivity/svn_hg_git_for_home_directory (accessed Julio 16, 2012).

24. Oracle. Oracle, 2011. http://www.oracle.com/technetwork/oem/pdf/512065.pdf (accessed Julio 16, 2012).

25. Oracle. Oracle, 2011. http://www.oracle.com/technetwork/oem/pdf/511949.pdf (accessed Julio 16, 2012).

26. DBComparer. DBComparer. http://dbcomparer.com/ (accessed Julio 19, 2012).

27. MySQL, D. MySQL. http://dev.mysql.com/ (accessed Julio 19, 2012).

28. Simego. Simego. http://www.simego.com/ (accessed Julio 19, 2012).

29. Microsoft. MSDN. http://msdn.microsoft.com (accessed Julio 24, 2012).

30. Lessandrini, T. DatabaseJournal. http://www.databasejournal.com/features/mssql/article.php/3890406/Red-Gate-SQL-Source-Control-a-Simple-and-Elegant-Solution-That-Works.htm (accessed Julio 25, 2012).

31. DevArt. DevArt tools, 2010. http://www.devart.com/dbforge/sql/datacompare/demostutorials/tutorial.html (accessed Julio 26, 2012).

32. Voxland, N. Liquibase. http://www.liquibase.org/ (accessed Julio 12, 2012).

33. Hunt, T. Troy Hunt. http://www.troyhunt.com/2011/02/automated-database-releases-with.html (accessed Julio 25, 2012).

34. Ambler, S. W.; Sdalage, P. J. *Refactoring Databases: Evolutionary Dabatase Design;* Addison Wesley Professional: Boston, 2006.

35. Abbate, J. Taming Serpents and Pachyderms, 2011. SQL Database Version Control – Liquibase. http://pyrseas.wordpress.com/2011/03/01/sql-database-version-control-%E2%80%93-liquibase/ (accessed Julio 14, 2012).

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS, A.C.

BIBLIOTECA

AUTORIZACION
PUBLICACION EN FORMATO ELECTRONICO DE TESIS

El que suscribe

Autor(s) de la tesis: Dante Ramos Orozco

Título de la tesis: Título del reporte técnico: Control de versiones sobre bases de datos relacionales: análisis de herramientas y sus características

Institución y Lugar: Centro de Investigación en Matemáticas, A. C., Zacatecas, Zacatecas

Grado Académico: Licenciatura ( ) Maestría (X) Doctorado ( ) Otro ( )

Año de presentación: 2012

Área de Especialidad: Ingeniería de Software

Director(es) de Tesis: Dr. Jorge Roberto Manjarrez Sánchez

Correo electrónico: dramos@cimat.mx, dramos@kerneltechnologies.com, dante.ro86@gmail.com

Domicilio: Zaragoza #2 int. 1, Colonia Centro, Tlaltenango, Zacatecas, C. P. 99700

Palabra(s) Clave(s): Control de versiones, SCVs, bases de datos relacionales, Subversion, SVN, Mercurial, Liquibase

Por medio del presente documento autorizo en forma gratuita a que la Tesis arriba citada sea divulgada y reproducida para publicarla mediante almacenamiento electrónico que permita acceso al público a leerla y conocerla visualmente, así como a comunicarla públicamente en la Página WEB del CIMAT.

La vigencia de la presente autorización es por un periodo de 3 años a partir de la firma de presente instrumento, quedando en el entendido de que dicho plazo podrá prorrogar automáticamente por periodos iguales, si durante dicho tiempo no se revoca la autorización por escrito con acuse de recibo de parte de alguna autoridad del CIMAT

La única contraprestación que condiciona la presente autorización es la del reconocimiento del nombre del autor en la publicación que se haga de la misma.

Atentamente

Dante Ramos Orozco