Propuesta de arquitectura de control de versiones en bases de datos evolutivas.

Gerardo Adolfo Salas Montoya
Maestría en ingeniería de software con énfasis en arquitectura y diseño de software.
Universidad Cenfotec
San Jose, Costa Rica
Email: gsalasm@ucenfotec.ac.cr

***Abstract*—Este artículo de maestría se centra en el estudio y la mejora de los procesos de control de versiones para bases de datos evolutivas, un área que ha recibido relativamente poca atención en la disciplina de ingeniería de software, sin embargo, es una práctica cada vez más demandada en los grupos de desarrollo. Las bases de datos están diseñadas para adaptarse y cambiar con el tiempo sin embargo el no tener un registro completo de los cambios que se aplican sobre cada uno de los objetos que la componen pueden llegar a causar diferentes problemas en los grupos de desarrollo y por su puesto en las bases de datos.**

***Index Terms*—Control de versiones de bases de datos, Bases de datos evolutivas, Integración/Despliegue continuo (CI/CD), Gestión de cambios en bases de datos, Arquitectura de bases de datos, Liquibase, Flyway, Desarrollo ágil de software**

I. Introducción

En el desarrollo de software moderno, las bases de datos desempeñan un rol fundamental para asegurar la adaptabilidad y flexibilidad de las aplicaciones. El concepto de bases de datos evolutivas surge como respuesta a la necesidad de las organizaciones de adaptarse rápidamente a cambios en los requisitos y necesidades del negocio, integrándose efectivamente en los procesos de Integración Continua y Despliegue Continuo (CI/CD). Sin embargo, la gestión de versiones en este tipo de bases de datos plantea desafíos significativos, ya que las herramientas tradicionales de control de versiones no están diseñadas para manejar la complejidad y estructura de las bases de datos, lo cual con frecuencia resulta en inconsistencias y problemas de implementación.

Este proyecto aborda la problemática de la falta de una arquitectura robusta de control de versiones para bases de datos evolutivas, lo cual afecta la calidad y eficiencia de los procesos de CI/CD en las organizaciones. La propuesta tiene como objetivo desarrollar una arquitectura de control de versiones específica para bases de datos que evolucionan constantemente, buscando reducir riesgos de errores y conflictos en los cambios, mejorar la consistencia de los datos y facilitar la integración rápida y frecuente de nuevas características. Además, esta solución permitirá mantener un registro exhaustivo de todos los cambios, apoyando tanto la implementación de mejoras como la corrección de errores.

Desde el punto de vista técnico, herramientas como Liquibase y Flyway ofrecen funcionalidades clave que pueden adaptarse y optimizarse para satisfacer las necesidades de control de versiones en bases de datos. Operativamente, el proyecto es viable, ya que muchas organizaciones ya emplean prácticas de control de versiones en sus procesos de CI/CD, y la integración de una arquitectura específica para bases de datos no requerirá cambios disruptivos, sino mejoras y adaptaciones efectivas.

Conforme al paradigma Objetivo-Pregunta-Métrica (GQM) [1], el objetivo de esta investigación se puede enunciar de la siguiente manera:

> ***Aplicar:*** Una arquitectura para el control de versiones en bases de datos que cambian con frecuencia.
> ***Con el propósito de:*** mejorar la trazabilidad, consistencia y facilidad de integración de los cambios en la base de datos en entornos de CI/CD.
> ***Con respecto a:*** los desafíos en el rastreo y gestión de cambios en bases de datos.
> ***Desde el punto de vista de:*** investigadores y profesionales de ingeniería de software enfocados en control de versiones y prácticas de integración y despliegue continuo.
> ***En el contexto de:*** mejorar las prácticas de control de versiones en organizaciones que dependen de actualizaciones frecuentes de bases de datos como parte de sus flujos de trabajo.

La estructura del resto del artículo es la siguiente. La Sección II presenta las preguntas de investigación. La Sección III discute los trabajos relacionados y el estado del arte. Los antecedentes se describen en la Sección IV. La metodología de investigación se explica en la Sección V, y los objetivos se presentan en las Secciones VI y VII. La Sección VIII describe las herramientas utilizadas para esta propuesta, ”Liquibase” y ”Flyway”. La Sección IX aborda la descripción de la arquitectura propuesta para el control de cambios en bases de datos. La Sección X describe las pautas de la solucion. Finalmente, las Secciones XI y XII discuten los resultados, conclusiones y presentan el trabajo futuro.

II. Preguntas de Investigación.

Esta sección enumera las principales preguntas de investigación que se pretende responder.

- ¿Cómo mantener la integridad del esquema de base de datos a lo largo del tiempo?
- ¿Cómo colaborar efectivamente entre equipos de desarrollo y administradores de bases de datos?

- ¿Cuál es el impacto de la implementación de un sistema de control de versiones en la integridad y consistencia de los datos?

III. Trabajos Relacionados

La siguiente investigación se ha realizado con un enfoque en el control de versiones dentro del contexto de la arquitectura de bases de datos:

En el 2006, Ambler y Sadalage [2] presentan en su obra una guía exhaustiva sobre la refactorización de bases de datos en entornos de desarrollo ágil. El libro se enfoca en cómo aplicar técnicas de refactorización para mejorar el diseño, la mantenibilidad, la extensibilidad y el rendimiento de las bases de datos. Los autores explican cómo realizar pequeños cambios en las estructuras de tablas, datos, procedimientos almacenados y disparadores sin modificar la semántica de la base de datos. Además, se detalla el proceso de evolucionar los esquemas de bases de datos al mismo ritmo que el código fuente en proyectos ágiles. La obra proporciona ejemplos prácticos y casos aplicados con Oracle y Java, adaptables a otras plataformas como CSharp, C++ y SQL Server. Su principal objetivo es ofrecer estrategias para reducir el desperdicio, el retrabajo, los riesgos y los costos en el desarrollo de bases de datos, haciendo posible su evolución en entornos productivos complejos.

Luego en el 2007, Sadalage [3] aborda la integración continua de bases de datos dentro de equipos ágiles, un desafío común debido a la dificultad de integrar continuamente la base de datos con el código de la aplicación. El autor defiende que el Desarrollo Evolutivo de Bases de Datos puede ser perfectamente compatible con los métodos ágiles, como el Desarrollo Guiado por Pruebas (TDD) y la Programación en Pareja. Este enfoque facilita a los equipos de desarrollo realizar modificaciones constantes en la base de datos sin comprometer la eficacia de las prácticas ágiles. A través de este trabajo, Sadalage demuestra que la integración continua de bases de datos no solo es posible, sino beneficiosa para mantener la calidad y confiabilidad del software a lo largo de todo el ciclo de desarrollo.

Spinellis [4] discute en su artículo la importancia de los sistemas de control de versiones (VCS) en el desarrollo de software. El autor subraya que, aunque muchos programadores no escribirían código de producción sin la ayuda de un editor o un compilador, muchos proyectos de software no utilizan un VCS, lo cual puede ser perjudicial a largo plazo. Spinellis argumenta que la adopción de un VCS puede mejorar la eficiencia y reducir el riesgo de errores en los proyectos, aunque su implementación requiere un esfuerzo inicial que muchas veces es percibido como un obstáculo. A través de este artículo, se destaca cómo un buen sistema de control de versiones puede ser una de las mejoras más significativas para los equipos de desarrollo.

Fluri, Fornari y Pustulka [5] abordan los beneficios de integrar prácticas de CI/CD (Integración Continua / Entrega Continua) en el desarrollo de aplicaciones de bases de datos. A través de estudios de caso industriales, los autores demuestran que la automatización de la integración y el despliegue de bases de datos mediante CI/CD reduce los fallos de despliegue, mejora la estabilidad del sistema y aumenta la frecuencia de los despliegues. Además, desde una perspectiva cualitativa, los desarrolladores informan una carga cognitiva reducida y mejoras en la calidad del software. Los resultados cuestionan las prácticas actuales de lanzamiento de bases de datos, que a menudo están impulsadas por expectativas comerciales y ventanas de lanzamiento fijas.

Cao [6] presenta un estudio empírico sobre la estimación del esfuerzo en el desarrollo ágil de software, evaluando diversas actividades como el desarrollo de características, la corrección de errores y la refactorización. Los resultados del estudio indican que, en contraste con la creencia popular, la precisión de las estimaciones de esfuerzo no mejora con el tiempo en el desarrollo ágil. Además, el estudio revela que las tareas de corrección de errores y refactorización tienden a ser sobreestimadas, mientras que las tareas de desarrollo de características no muestran tal patrón. Esta investigación aporta nueva información sobre las dificultades de estimación en el contexto de metodologías ágiles.

Roddick [7] explora en su artículo la evolución de esquemas en los sistemas de bases de datos, un tema crucial para garantizar que los sistemas se adapten a los cambios del mundo real mediante modificaciones en la estructura del esquema. Además, destaca la importancia de retener estados anteriores del esquema para asegurar la confiabilidad y la integridad de los datos. El autor aborda los desafíos que enfrentan los administradores de bases de datos al tomar decisiones sobre la validez de los datos existentes cuando se implementa un nuevo esquema. La evolución de esquemas se presenta como una propiedad esencial para la adaptación y la gestión de los sistemas de bases de datos a lo largo del tiempo.

Las fuentes de información que se mencionan, como IEEE, ACM, Research Gate entre otras, proporcionan una base sólida para entender las prácticas actuales en el desarrollo de software y la gestión de bases de datos. Los trabajos revisados destacan la importancia del uso de sistemas de control de versiones, como se observa en el estudio de Spinellis [4], que subraya cómo estos sistemas mejoran la productividad y reducen los riesgos asociados al desarrollo de software. Por otro lado, el trabajo de Fluri et al. [5] aborda la integración continua y entrega continua (CI/CD) en el contexto de bases de datos, demostrando cómo la automatización de estos procesos puede mejorar la estabilidad y eficiencia en los proyectos de desarrollo. Además, la evolución de esquemas en sistemas de bases de datos, un tema tratado por Roddick [7], es fundamental para asegurar que las bases de datos puedan adaptarse a los cambios sin comprometer la integridad de los datos. Finalmente, la investigación de Cao [6] sobre la estimación de esfuerzos en el desarrollo ágil ofrece perspectivas valiosas sobre cómo las metodologías ágiles impactan en la precisión de las estimaciones de trabajo, un aspecto crítico en proyectos complejos. Estos trabajos no solo enriquecen la comprensión de los procesos ágiles y su integración con la gestión de bases de datos, sino que también forman la base para las soluciones propuestas en esta investigación.

IV. Antecedentes

Dada la investigacion realizada se aprecia como muchos autores consideran que el campo de las bases de datos ha experimentado una evolución significativa, impulsada por la necesidad de adaptarse a los cambios rápidos del entorno de desarrollo ágil y las metodologías de integración continua (CI) y entrega continua (CD). La gestión de versiones y la refactorización de bases de datos, temas clave en este estudio, han sido objeto de múltiples investigaciones. Uno de los trabajos más relevantes en este campo es el de Ambler y Sadalage (2006) [2], quienes introdujeron las técnicas de refactorización de bases de datos como un medio para mejorar la estructura y la calidad de los sistemas de bases de datos sin comprometer su semántica. Este enfoque permite que las bases de datos evolucionen de manera paralela al código fuente, lo que facilita la adaptación continua en proyectos ágiles.

En paralelo, la investigación sobre la integración de bases de datos en entornos CI/CD ha ganado relevancia, como lo demuestra el trabajo de Fluri et al. (2023) [5], que mostró cómo la automatización de la integración y despliegue de bases de datos contribuye a la estabilidad y eficiencia de los proyectos de software. Sin embargo, a pesar de estos avances, sigue siendo un desafío integrar las bases de datos en los ciclos ágiles de manera eficiente y con control adecuado de versiones. Así, la evolución de esquemas en sistemas de bases de datos, descrita por Roddick (1992) [7], sigue siendo una cuestión crucial, ya que permite que las bases de datos se adapten a los cambios del mundo real mientras mantienen la integridad de los datos. Estos antecedentes destacan la importancia de crear una arquitectura sólida de control de versiones para bases de datos.

V. Metodología

La metodología de investigación de esta propuesta adopta un enfoque mixto, integrando tanto técnicas cualitativas como cuantitativas. Según Monje (2011) [8], el enfoque mixto permite abordar de manera integral los fenómenos bajo estudio, ofreciendo una comprensión más profunda al combinar la recolección de datos descriptivos y numéricos. Este enfoque se utilizará para explorar las percepciones y prácticas relacionadas con el control de versiones en bases de datos evolutivas, utilizando técnicas como entrevistas semiestructuradas y análisis documental en el caso cualitativo, y encuestas y experimentos controlados en el caso cuantitativo.

La metodología cualitativa permite obtener una comprensión profunda de las experiencias y desafíos de los desarrolladores y administradores de bases de datos en relación con el control de versiones. Para ello, se llevaron a cabo entrevistas semiestructuradas con expertos en el área de desarrollo de software y administración de bases de datos, así como un análisis documental de los registros de cambios y manuales de proyectos reales basado en la adapcion de la ciencia de diseño [9]. La selección de los participantes se realiza mediante un muestreo intencional, y los datos obtenidos se analizan a través de un enfoque de análisis temático, permitiendo identificar patrones recurrentes en las prácticas de control de versiones.

Por otro lado, el enfoque cuantitativo se aplica para medir y analizar los datos numéricos relacionados con la implementación y el rendimiento de las prácticas de control de versiones en bases de datos. A través de encuestas estructuradas, se recolectaron datos sobre la frecuencia, efectividad y desafíos de las prácticas de control de versiones, mientras que los experimentos controlados permitirán evaluar el rendimiento y costo técnico de herramientas como Liquibase y Flyway. Los datos recolectados se analizan utilizando técnicas estadísticas descriptivas e inferenciales, permitiendo obtener conclusiones más objetivas y generalizables sobre las mejores prácticas en este campo.

La integración de ambos enfoques cualitativos y cuantitativos permite obtener una visión holística del fenómeno investigado. La triangulación de los datos cualitativos y cuantitativos se llevo a cabo mediante la comparación de los resultados de las entrevistas y las encuestas, lo que permite identificar convergencias y divergencias en los hallazgos. Además, se realizaron análisis comparativos entre los resultados obtenidos mediante ambos métodos para validar y complementar los hallazgos, ofreciendo así una comprensión más robusta y completa del problema de investigación.

El uso del enfoque mixto se justifica por la complejidad del tema investigado. Esta metodología permite abordar tanto los aspectos técnicos como los humanos del control de versiones en bases de datos evolutivas, lo que proporciona una base sólida de datos empíricos. Estos datos no solo permitirán obtener conclusiones sobre el estado actual de las prácticas en este campo, sino también ofrecer recomendaciones prácticas basadas en evidencia para mejorar las prácticas de control de versiones en la gestión de bases de datos.

VI. Objetivo

El objetivo general de este proyecto es proponer una arquitectura de control de versiones para bases de datos relacionales evolutivas que facilite la gestión de cambios, mejore la eficiencia y la seguridad de los procesos de CI/CD. La elección de la taxonomía de Bloom (1956) [10] se debe a su amplia documentación y aplicabilidad, ya que su estructuración jerárquica permite definir los objetivos de manera clara y ordenada, desde los más generales hasta los más específicos. Esto facilita una alineación adecuada con los principios del control de versiones en bases de datos dentro de entornos ágiles de desarrollo.

VII. Objetivos Específicos

1) Analizar las herramientas actuales de control de versiones para bases de datos relacionales, tales como Liquibase y Flyway, con el fin de comprender sus características, ventajas y limitaciones en entornos de CI/CD.
2) Evaluar el impacto técnico de implementar cambios en bases de datos sin un control de versiones adecuado, con énfasis en los efectos sobre la integridad, escalabilidad y confiabilidad de las bases de datos en entornos de desarrollo ágiles.
3) Desarrollar una arquitectura que aborde los desafíos específicos de las bases de datos evolutivas, adaptando y

mejorando las prácticas actuales de control de versiones para permitir un manejo efectivo de los cambios.

4) Probar la efectividad de la arquitectura propuesta en entornos reales de desarrollo de software, asegurando que los beneficios descritos se materialicen en la práctica y mejoren la productividad y seguridad de los procesos de desarrollo.

VIII. Marco Propuesto

La propuesta de solución para el control de versiones en bases de datos evolutivas se diseñada para abordar los desafíos en la gestión de cambios en bases de datos, particularmente dentro del contexto de metodologías ágiles y prácticas de CI/CD. La arquitectura planteada permite a los equipos de desarrollo rastrear, revertir y desplegar cambios de manera eficiente y segura, minimizando los riesgos de errores y conflictos en los esquemas de bases de datos, mejorando la colaboración y asegurando la integridad de los datos a lo largo de todo el ciclo de vida del desarrollo.

La metodología propuesta se apoya en herramientas como Liquibase y Flyway, enfocándose en su capacidad para gestionar cambios y automatizar migraciones en bases de datos. A través de un enfoque iterativo e incremental, la arquitectura fue refinada y adaptada a las necesidades específicas del proyecto, resultando en una solución robusta y flexible que puede integrarse en flujos de trabajo ágiles de CI/CD. Esta solución incluye una guía detallada para la implementación de control de versiones en bases de datos, ofreciendo recomendaciones basadas en la evaluación de las herramientas mencionadas y contribuyendo a mejorar los procesos de desarrollo y despliegue en entornos de bases de datos.

A continuacion se describen las dos herramientas antes mencionadas para esta propuesta:

A. Liquibase

Liquibase es una herramienta de código abierto ampliamente utilizada para el control de versiones en bases de datos. Ofrece funcionalidades clave, como la compatibilidad con bases de datos SQL y NoSQL, la reversión de cambios (rollback) y la integración con herramientas de CI/CD como Jenkins y GitHub Actions. Estas características permiten una gestión eficaz de las modificaciones en el modelo de datos, asegurando la trazabilidad y minimizando el riesgo de errores en los entornos de desarrollo, prueba y producción.
Además, Liquibase permite definir cambios en formatos como XML, YAML, JSON o SQL, lo que brinda flexibilidad a los equipos de desarrollo y facilita su integración en proyectos de diferentes escalas.

Para organizar los cambios, Liquibase utiliza un archivo principal denominado *changelog*, que contiene una lista cronológica de cambios estructurada en *changesets* como podemos verlo en la Figura 1. Cada *changeset* es una unidad atómica de cambio que incluye un identificador único, el autor y las instrucciones para realizar el cambio, como sentencias SQL o referencias a archivos SQL externos. En este proyecto, se utiliza el formato YAML para el *changelog* por su simplicidad y claridad. A través del *changelog*, Liquibase asegura que los cambios se apliquen de manera ordenada, controlada y consistente, particularmente en entornos de CI/CD, reduciendo el riesgo de conflictos y facilitando la implementación de actualizaciones.

```yaml
databaseChangeLog:
  - changeSet:
      id: create-clients-table
      author: gesalas
      labels: squad-1
      context: development
      changes:
        - sqlFile:
            path:  squad1/tables/clients_tb_creation.sql
            relativeToChangelogFile: true
        - tagDatabase:
            tag: phase15-sprint1

  - changeSet:
      id: create-insert-user-fn
      author: johnsmith
      labels: squad-2
      context: development
      runOnChange: true
      changes:
        - sqlFile:
            path: ../../mutables/functions/fn_insert_or_update_client.sql
            relativeToChangelogFile: true
            splitStatements: false
            stripComments: true
        - tagDatabase:
            tag: phase15-sprint1

  - changeSet:
      id: alter-roles-tb-adding-description-col
      author: gesalas
      labels: squad-1
      changes:
        - sqlFile:
            path:  squad1/tables/alter_roles_tb_adding_description_col.sql
            relativeToChangelogFile: true
```

Fig. 1. Ejemplo de Changelog de Liquibase

La configuración del proyecto con Liquibase requiere un archivo `liquibase.properties` que especifica la conexión a la base de datos y el *changelog* principal en el cual se iran agregando los cambios que queramos ejecutar en el proyecto, como e puede ver en la Figura 2. Este enfoque también permite que los desarrolladores y DBAs trabajen directamente sobre archivos `.sql` para realizar modificaciones, aprovechando la trazabilidad de Liquibase. En algunos casos, como en los scripts de datos que necesitan ejecutarse múltiples veces, es posible configurar el *changeset* con la opción `runOnChange`, lo que permite ejecutar el script cada vez que se despliegue.

```
changelogFile: db.changelog-master.yaml
driver: org.postgresql.Driver
url: jdbc:postgresql://localhost:5432/postgres?schema=lb
username: postgres
password: tu_contraseña
liquibase.hub.mode=off
```

Fig. 2. Ejemplo del archivo de configuración ”liquibase.properties” de Liquibase

Para la ejecución de los cambios, Liquibase emplea el comando `liquibase update`, que aplica únicamente los *changesets* aún no registrados en la tabla **DATABASECHANGELOG**. Esta tabla, creada en la base de datos, mantiene un registro detallado de cada *changeset* ejecutado, incluyendo el ID, autor, fecha y checksum, lo que verifica la integridad de los cambios. Este sistema asegura que los cambios no se dupliquen, y permite que Liquibase

mantenga una secuencia controlada, respetando dependencias entre objetos para evitar errores. Así, Liquibase optimiza el control de versiones en bases de datos relacionales, integrándose eficazmente en entornos de desarrollo ágiles y facilitando la gestión de cambios.

La organización efectiva de los cambios en la base de datos de un proyecto de control de versiones con Liquibase requiere una jerarquía de carpetas clara y estandarizada, como se ilustra en la Figura 3. Esta estructura permite gestionar de manera ordenada los archivos modificables e inmutables, y facilita la trazabilidad de los cambios al dividir el proyecto en fases, sprints y equipos (squads). A continuación, se presenta la propuesta de jerarquía de carpetas para un proyecto bajo metodología ágil, estructurada en fases y sprints, donde cada fase representa una iteración de desarrollo.

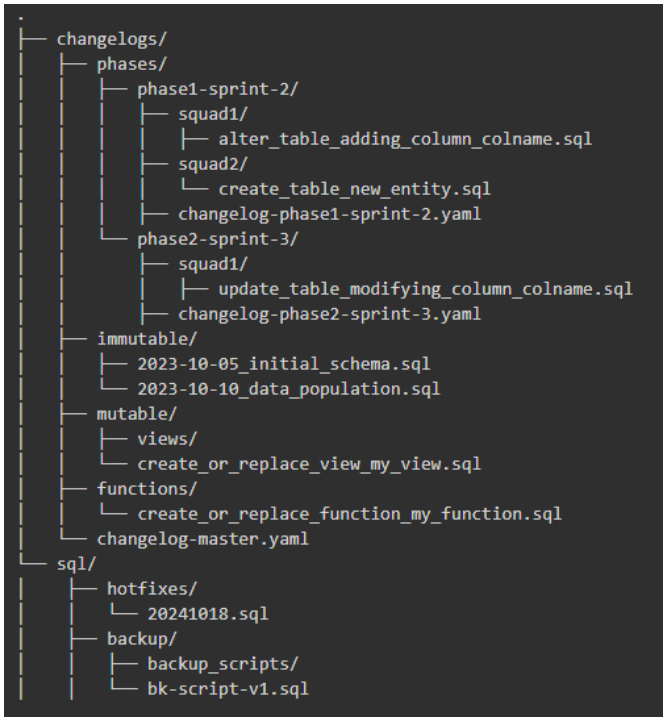

Fig. 3. Ejemplo de jerarquía de carpetas propuesta para Liquibase

La siguiente descripción detalla las carpetas principales y su función en el control de versiones utilizando Liquibase:

- **changelogs/**: Carpeta principal para todos los archivos de cambios. Contiene todos los changelogs y los archivos SQL que están directamente relacionados con los cambios aplicados a la base de datos. Es el corazón de la estructura de cambios en tu base de datos.
- **phases/**: Dentro de esta carpeta se organizan los cambios en subcarpetas que representan las fases y los sprints del proyecto. El formato utilizado para nombrar cada subcarpeta es `phaseX-sprint-Y`, donde `X` representa el número de la fase del proyecto y `Y` representa el número del sprint.
- **squadX/**: Cada fase y sprint tendrá su propio subdirectorio para los cambios realizados por los squads. En estas carpetas se almacenarán los archivos SQL correspondientes a los cambios específicos que realiza cada equipo.
- **immutable/**: Carpeta destinada a los archivos no modificables. Aquí se almacenan los cambios iniciales y los scripts que, una vez ejecutados, no deben alterarse. Estos archivos representan una parte crítica de la historia del proyecto y son inmutables para asegurar que el historial de la base de datos se mantenga íntegro.
- **mutable/**: Aquí se almacenan los objetos que pueden modificarse y recrearse a lo largo del tiempo, como vistas y funciones. Estos objetos son esenciales para la evolución de la base de datos. Su naturaleza mutable implica que pueden ser recreados en múltiples sprints dependiendo de las necesidades del proyecto. Las vistas se almacenan en el subdirectorio `mutable/views/`, mientras que las funciones de la base de datos se encuentran en `mutable/functions/`.

Características principales de Liquibase:

- **Compatibilidad con múltiples bases de datos**: Soporta bases de datos SQL y NoSQL, permitiendo su uso en entornos con diferentes sistemas de gestión de bases de datos.
- **Rollback o reversión de cambios**: Permite deshacer cambios aplicados previamente, asegurando una recuperación controlada en caso de errores.
- **Integración con herramientas CI/CD**: Se integra con plataformas como Jenkins y GitHub Actions, facilitando la automatización del despliegue y la sincronización de cambios.
- **Formatos de definición de cambios flexibles**: Ofrece opciones para definir cambios en varios formatos como XML, YAML, JSON o SQL, proporcionando flexibilidad para distintos equipos de desarrollo.
- **Changelog y changesets**: Organiza los cambios en un archivo de *changelog* que estructura los cambios en *changesets* atómicos, los cuales incluyen un identificador único, autor y detalles del cambio.
- **Registro de cambios en la base de datos**: Utiliza la tabla **DATABASECHANGELOG** para registrar el historial de cambios, garantizando la trazabilidad y la integridad de cada actualización.
- **Comando de actualización controlada**: Ejecuta el comando `liquibase update` para aplicar solo los cambios nuevos, evitando duplicados y controlando la secuencia de ejecución.
- **Configuración personalizada**: Admite configuraciones específicas en el archivo `liquibase.properties`, permitiendo conexiones personalizadas.

B. Flyway

Flyway es una herramienta de migración de base de datos que utiliza archivos SQL o código Java para aplicar cambios secuenciales y versionados a la base de datos. A diferencia de Liquibase, que emplea archivos de control llamados changelogs, Flyway sigue una convención de nombres simple para versionar los archivos SQL y asegura que se ejecuten en el orden correcto. Los cambios en la base de datos se aplican a

través de archivos de migración, los cuales tienen un formato de nombres específico que permite a Flyway controlar el orden en que se aplican.

Cada archivo de migración comienza con un prefijo *V* seguido de un número de versión por ejemplo, `V1__`, `V2__`, `etc.`, lo que asegura que los cambios se ejecuten secuencialmente en función del número de versión. Flyway también soporta correcciones a través de archivos especiales que comienzan con `R__` en lugar de `V__`, lo que permite aplicar cambios repetidos como en el caso de vistas y funciones.

La configuración de Flyway requiere el archivo llamado `flyway.conf` Figura 4, donde se especifican parámetros clave como la conexión a la base de datos, el directorio donde se encuentran los scripts de migración, y otras configuraciones relevantes. Estos parámetros incluyen la URL de la base de datos, las credenciales de acceso, el esquema de destino y la ubicación de los archivos de migración. Además, el parámetro `flyway.baselineOnMigrate` permite establecer un punto de partida para bases de datos que ya contienen datos, facilitando la integración de Flyway en proyectos existentes.

```
flyway.driver=org.postgresql.Driver
flyway.url=jdbc:postgresql://localhost:5432/postgres
flyway.user=postgres
flyway.pass=tu_contraseña
flyway.schemas=public
flyway.locations=filesystem:./sql
flyway.baselineOnMigrate=true
```

Fig. 4. Ejemplo de archivo de configuración de Flyway

La jerarquía de carpetas en Flyway es sencilla como se muestra en la Figura 5, siguiendo una convención en la que los archivos de migración versionados comienzan con `V__` y los archivos de migración repetibles con `R__`. Los archivos `V__` se aplican secuencialmente según el número de versión, mientras que los archivos `R__` son ejecutados cada vez que se detectan cambios. Esta estructura organizada facilita la gestión de las migraciones y asegura que se apliquen en el orden correcto.

```
sql/
├── V1__initial_schema.sql
├── V2__add_column_to_users.sql
├── V3__modify_orders_table.sql
└── R__refresh_views.sql
```

Fig. 5. Ejemplo de jerarquía de carpetas propuesta para Flyway

Los archivos SQL son fundamentales en Flyway para gestionar las migraciones. Estos archivos contienen los scripts que Flyway ejecuta secuencialmente para realizar los cambios en la base de datos. La convención de nombres de los archivos garantiza que se sigan las secuencias adecuadas para las migraciones versionadas y repetibles. Cada archivo SQL es ejecutado una sola vez, en el caso de las migraciones versionadas, y repetidamente, en el caso de las migraciones repetibles.

Flyway proporciona una serie de comandos útiles para la gestión de migraciones. El comando `flyway migrate` se utiliza para aplicar todas las migraciones pendientes, mientras que `flyway info` permite obtener información sobre el estado de las migraciones aplicadas. Además, el comando `flyway repair` se utiliza para reparar la tabla de historial en caso de inconsistencias. Los cambios realizados se registran en una tabla llamada `flyway_schema_history`, que mantiene un historial detallado de las migraciones aplicadas, incluyendo el número de versión, la descripción, la fecha de ejecución y el resultado de la migración. Esta tabla es crucial para evitar la duplicación de migraciones y permite el seguimiento de los errores en el proceso de migración.

Las principales características de Flyway son las siguientes:

- **Convención de nombres para archivos de migración**: Los archivos de migración versionados comienzan con `V__` seguido de un número de versión (ej. `V1__`), mientras que los archivos de migración repetibles comienzan con `R__`.
- **Ejecución secuencial de migraciones**: Los archivos de migración son ejecutados en orden según su número de versión.
- **Soporte para correcciones**: Los archivos de corrección (`R__`) se ejecutan repetidamente, lo cual es útil para cambios frecuentes como en vistas y funciones.
- **Registro de historial de migraciones**: Flyway mantiene una tabla llamada `flyway_schema_history` donde guarda información sobre todas las migraciones aplicadas, incluyendo la versión, descripción, fecha de ejecución y resultado.
- **Configuración mediante flyway.conf**: En el archivo de configuración `flyway.conf` se especifican parámetros clave como la URL de la base de datos, las credenciales de acceso, el esquema de destino y la ubicación de los archivos de migración.
- **Migraciones aplicadas automáticamente**: Flyway aplica todas las migraciones pendientes al ejecutar el comando "`flyway migrate`".
- **Soporte para bases de datos existentes**: Flyway permite establecer un punto de partida para bases de datos que ya contienen datos mediante el parámetro "`flyway.baselineOnMigrate`".
- **Integración con CI/CD**: Flyway se puede integrar fácilmente en pipelines de CI/CD para automatizar el proceso de migración de bases de datos.

C. GitOps para bases de datos relacionales

En el caso de bases de datos, aplicar GitOps significa gestionar los cambios de esquema y migraciones a través de pipelines de CI/CD, asegurando que los despliegues de la base de datos sean consistentes, seguros y completamente automatizados. Este enfoque permite que las migraciones de

bases de datos sean tratadas como código, lo que mejora la trazabilidad, la consistencia y la eficiencia en los despliegues.GitOps es un enfoque que automatiza la gestión de infraestructura y aplicaciones mediante la sincronización de estados declarativos almacenados en un repositorio Git con los entornos de producción.
El flujo GitOps propuesto para la base de datos sigue los siguientes pasos:

- **Control de cambios con Git:** Todas las migraciones de la base de datos se almacenan en un repositorio Git. Los archivos de migración de Liquibase o Flyway (changelogs, scripts SQL) se versionan y son revisados mediante pull requests.
- **Integración Continua (CI):** Cada vez que se sube un cambio (una nueva migración de base de datos) al repositorio Git, un pipeline CI valida la integridad del código SQL. Un SQL linter como SQLFluff puede ser utilizado en este proceso para verificar que los scripts sigan las mejores prácticas de estilo y calidad. Además, se ejecutan pruebas automáticas en un entorno de base de datos de prueba para verificar que las migraciones no rompan la integridad del esquema.
- **Despliegue Continuo (CD):** Si las pruebas CI son exitosas, el pipeline de CD es activado y los cambios de la base de datos se despliegan de forma automatizada en entornos como staging y producción. Dependiendo de la herramienta utilizada (Liquibase o Flyway), se ejecutarán las migraciones correspondientes para garantizar que la base de datos esté sincronizada con el estado actual del código en el repositorio Git.

D. SQL Linter

Un SQL linter es una herramienta diseñada para analizar el código SQL y detectar posibles errores, inconsistencias o malas prácticas antes de que se ejecuten en la base de datos. Al igual que los linters utilizados en otros lenguajes de programación, un linter de SQL ayuda a mejorar la calidad del código, asegurando que se adhiera a los estándares de codificación establecidos y que sea fácil de mantener y entender. El uso de un linter en los proyectos de bases de datos es crucial, ya que garantiza que el código SQL esté optimizado, siga convenciones predefinidas, y evite errores comunes como errores de sintaxis, inconsistencias y malas prácticas.

En esta solución se propone utilizar `SQLFluff`, un linter extensible y configurable para SQL, ideal para proyectos que involucren diversas plataformas de bases de datos como PostgreSQL, MySQL y SQL Server. `SQLFluff` es una herramienta de código abierto que soporta múltiples dialectos SQL, lo que lo convierte en una opción versátil para entornos con diferentes bases de datos. Su integración en los procesos de desarrollo e implementación asegura que las consultas SQL sigan las mejores prácticas antes de ser enviadas a producción, lo que ayuda a mantener la calidad del código en todo momento. Además, `SQLFluff` es altamente adaptable, permitiendo la creación de reglas personalizadas que se ajustan a los estándares de codificación específicos de cada equipo o proyecto. Su capacidad para integrarse en pipelines de CI/CD facilita la verificación del código SQL antes del despliegue, garantizando el cumplimiento de las convenciones establecidas. También puede analizar archivos `.sql`, detectando errores como inconsistencias en mayúsculas, problemas de formato o uso incorrecto de columnas, y ofrece autocorrección para adoptar un estilo uniforme en los proyectos de manera más eficiente.

E. Configuración de SQLFluff

El primer paso para incorporar `SQLFluff` es su instalación. Dado que se basa en Python, es recomendable contar con un entorno de desarrollo en Python bien configurado. Una vez asegurado esto, se puede instalar `SQLFluff` utilizando el gestor de paquetes de Python "`pip`", mediante el comando `pip install sqlfluff`. Esto facilita su integración en cualquier flujo de trabajo automatizado. Es importante aclarar que Python no es obligatorio para el uso de `SQLFluff`, pero es la opción más directa y sencilla.

Para que `SQLFluff` se comporte de acuerdo con las reglas del proyecto, es necesario un archivo de configuración específico, el cual debe ser llamado `.sqlfluff` y debe de estar en la raíz del proyecto. Este archivo define aspectos clave como el dialecto de base de datos, la longitud máxima de líneas, el estilo de indentación, entre otros. En la Figura 6, se muestra un ejemplo de la configuración de SQLFluff para un proyecto que usa PostgreSQL como base de datos:

```
[sqlfluff]
dialect = postgres
max_line_length = 120
exclude_rules = L009
tab_space_size = 4
indent_unit = space
```

Fig. 6. Ejemplo de archivo de configuración para sqlfluff

F. Configuración de Pre-Commit

Para asegurar que ningún script SQL llegue al repositorio con errores de estilo o formato, podemos integrar `SQLFluff` con `pre-commit`, de manera que cada vez que un desarrollador intente hacer un commit, el código SQL sea analizado automáticamente. Esto añade una capa adicional de control, asegurando que el código SQL que se integra cumpla con las reglas del proyecto. Para integrar `pre-commit`, debemos instalarlo utilizando el manejador de paquetes de Python, `pip`, mediante el comando `pip install pre-commit`. Luego, se configura un archivo llamado `.pre-commit-config.yaml`, Figura 7 que define los hooks que `SQLFluff` ejecutará antes de cada commit que se quiera agregar al repositorio.

Esto permite que cada vez que se haga un commit, `SQLFluff` valide los archivos `.sql` y si es posible, corrija automáticamente algunos errores de estilo sin intervención

manual. Además, cabe aclarar que SQLFluff puede ejecutarse manualmente sobre los scripts SQL del proyecto utilizando la terminal del sistema. Esto es útil cuando se quiere revisar el código de manera proactiva antes de hacer cualquier cambio significativo. En este caso, el comando para ejecutar la revisión del código sería sqlfluff lint <nombre_del_archivo.sql>, y en caso de querer corregir los errores encontrados, se ejecutaría sqlfluff fix <nombre_del_archivo.sql>. Si se desea ejecutar la revisión sobre todo el proyecto, solo basta con reemplazar el nombre del archivo por un punto, un ejemplo de esto sería: "sqlfluff fix .".

```
repos:
  - repo: https://github.com/sqlfluff/sqlfluff
    rev: v1.0.0  # Versión de SQLFluff
    hooks:
      - id: sqlfluff-lint
        args: ["--dialect", "postgres"]
      - id: sqlfluff-fix
        args: ["--dialect", "postgres"]
```

Fig. 7. Ejemplo de archivo de configuración para pre-commit en proyecto con sqlfluff

G. Consideraciones de la implementación

Ambas herramientas, Liquibase y Flyway, se integran perfectamente en un entorno GitOps con PostgreSQL, proporcionando trazabilidad completa de los cambios en la base de datos. Al utilizar un repositorio Git como la fuente de verdad y un linter para SQL que nos ayude a mantener algunas reglas entre los miembros del equipo de desarrollo, se asegura que cualquier modificación en la base de datos esté bien documentada y pueda ser revisada, probada y desplegada de forma controlada. Además, tanto Liquibase como Flyway mantienen una tabla interna (databasechangelog en Liquibase y flyway_schema_history en Flyway) en la base de datos para registrar las migraciones aplicadas. Estas tablas son esenciales para el correcto funcionamiento de los despliegues automatizados, y es importante considerar las capacidades de rollback que ofrecen las herramientas, siendo más flexible en Liquibase que en Flyway, ya que este último depende más de scripts SQL orientados a versiones.

IX. Conclusiones y Trabajos Futuros

Este proyecto ha explorado la implementación de un sistema de control de versiones en bases de datos altamente cambiantes, evaluando herramientas como Liquibase y Flyway en el contexto de metodologías ágiles y prácticas de CI/CD. A continuación, se presentan las conclusiones obtenidas y algunas recomendaciones para futuras investigaciones.

A. Conclusiones

La implementación de control de versiones en bases de datos es fundamental en entornos ágiles y CI/CD, ya que mejora la trazabilidad de cambios y la colaboración en el equipo de desarrollo. La comparación entre Liquibase y Flyway mostró que ambas herramientas son efectivas para gestionar cambios en bases de datos, cada una con sus ventajas. Liquibase es robusto en soporte para múltiples bases de datos y cambios complejos, mientras que Flyway ofrece una curva de aprendizaje más sencilla, siendo ideal para equipos menos experimentados. Este proyecto concluye que la adopción de cualquiera de estas herramientas, en combinación con prácticas ágiles, incrementa la eficiencia en la implementación de cambios en bases de datos.

B. Trabajos Futuros

Este estudio abre varias áreas de investigación futura. Una línea interesante es la automatización avanzada de la integración de bases de datos en entornos CI/CD, explorando cómo tecnologías emergentes, como inteligencia artificial y machine learning, pueden mejorar la predicción de conflictos en cambios. Además, sería valioso estudiar la adaptación del control de versiones a bases de datos no relacionales y distribuidas, como MongoDB o Cassandra, dada la creciente popularidad de estos sistemas. Finalmente, es importante investigar el impacto cultural y organizacional de adoptar control de versiones en bases de datos, considerando los ajustes en mentalidad y procesos que podrían ser necesarios para una implementación efectiva en entornos reales.

Referencias

[1] V. R. Basili, "Software modeling and measurement: The goal/question/metric paradigm," *IEEE*, 1992.

[2] S. W. Ambler and P. J. Sadalage, *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional, 2006.

[3] P. J. Sadalage, *Recipes for Continuous Database Integration: Evolutionary Database Development*. Addison-Wesley Professional, 2007.

[4] D. Spinellis, "Version control systems," *IEEE Software*, vol. 22, no. 5, pp. 108–109, 2005.

[5] J. Fluri, F. Fornari, and E. Pustulka, "Measuring the benefits of ci/cd practices for database application development," in *IEEE/ACM International Conference on Software and System Processes (ICSSP)*, 2023, pp. 46–57.

[6] L. Cao, "Estimating efforts for various activities in agile software development: An empirical study," *IEEE Access*, vol. 10, pp. 83 311–83 321, 2022.

[7] J. F. Roddick, "Schema evolution in database systems: An annotated bibliography," *ACM SIGMOD Record*, vol. 21, no. 4, pp. 35–40, 1992.

[8] C. Monje, "Metodología de la investigación cuantitativa y cualitativa," 2011. [Online]. Available: https://www.uv.mx/rmipe/files/2017/02/Guia-didactica-metodologia-de-la-investigacion.pdf

[9] S. V.-C. H. . N.-Z. L. Robles-Sandoval, "Adaptación de la metodología de ciencia de diseño en el desarrollo de luminarias," *Revista de la Facultad de Ingenierías y Tecnologías de la Información y Comunicación*, vol. 3, no. 2, pp. 74–79, 2019.

[10] B. S. Bloom and collaborators, *Taxonomy of Educational Objectives: The Classification of Educational Goals, Handbook I: Cognitive Domain*. New York: David McKay Company, 1956.