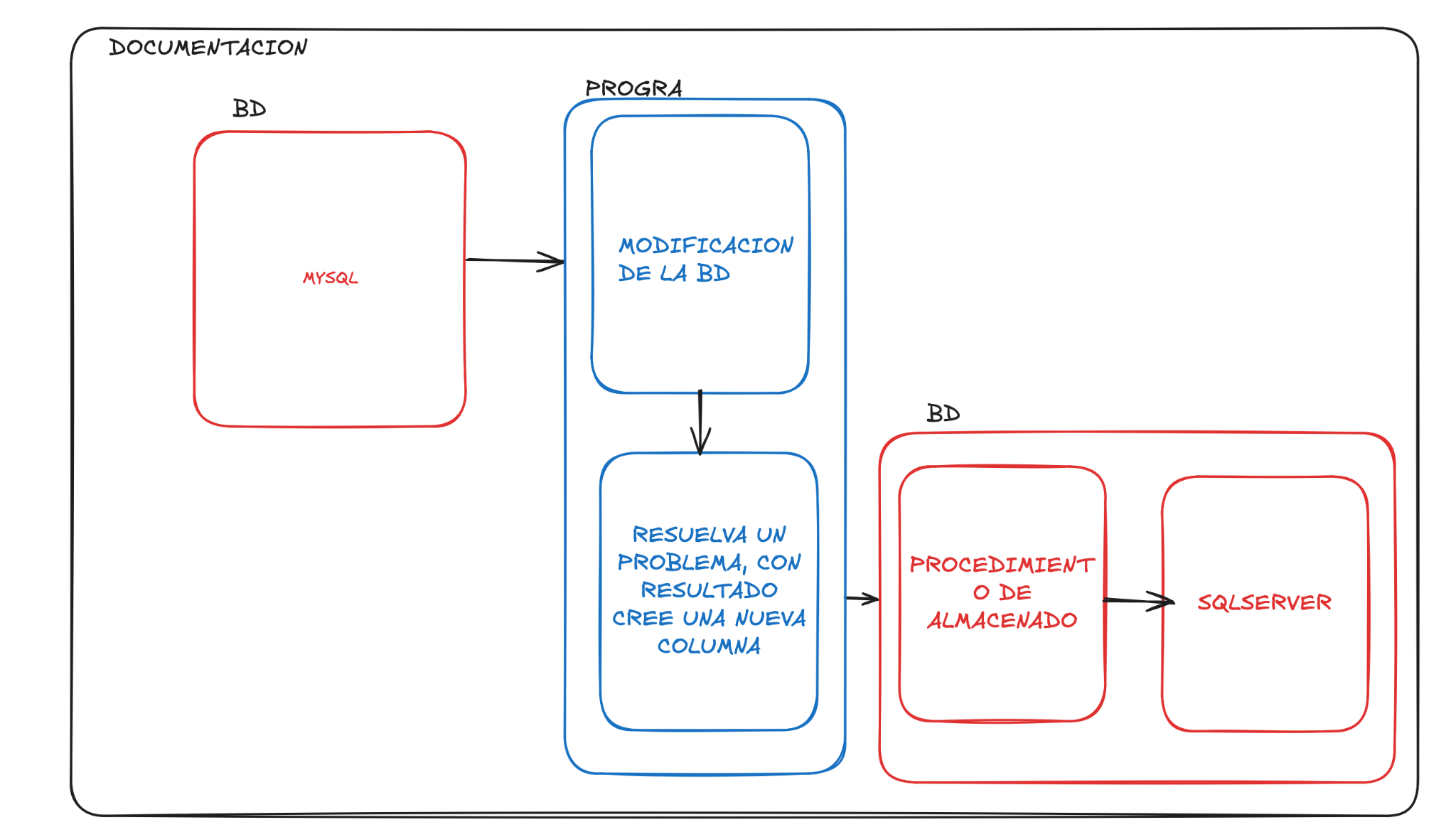
**Descripción del problema**

Como ingeniero de software recién contratado en una empresa, se le ha asignado la tarea de modernizar el sistema heredado de la empresa para gestionar los datos. El sistema actual, basado en tecnología obsoleta, es lento, poco confiable y propenso a inconsistencias en los datos. El proyecto consiste en diseñar e implementar un nuevo sistema utilizando una base de datos relacional y una aplicación en Python. Este sistema debe ser eficiente, escalable y proporcionar una interfaz fácil de usar para gestionar los datos. El desafío no solo radica en construir un software, sino también en integrarlo sin problemas con la infraestructura existente de la empresa y abordar las posibles complejidades de la migración de datos. Se tendrá que desarrollar e implementar procedimientos almacenados para realizar tareas comunes de modificación de datos, garantizando la integridad de la información. Como guía puede referenciarse en el diagrama de abajo.



**Figura 1.** Descripción del problema

**Tarea de proyecto: Migración de datos**

Ha llegado el momento de escribir código. Utiliza las bibliotecas para la conexión con las bases de datos y crea un script, un programa con python, que migre los datos a una nueva base de datos. Si es necesario modificar el esquema original, utiliza las sentencias en SQL para ello. Recuerde que su esquema inicial se encuentra en MySQL, se debe migrar de gestor según requerimientos, por ejemplo a SQL Server, PostgreSQL, etc. Para las siguientes indicaciones se utiliza PostgreSQL como ejemplo, pero recuerde que todo se puede aplicar a otros manejadores de base de datos.

**Criterios de finalización:**

1. Tras ejecutar el script, todas las tablas deben aparecer en el nuevo manejador de BD, PostgreSQL.
2. Todas las relaciones entre registros deben mantenerse intactas.
3. El código debe utilizar *dataclass*. Revisar, ejemplo adjunto.
4. La ejecución repetida del script no debe generar registros duplicados en el nuevo manejador de base de datos, por ejemplo en PostgreSQL.
5. El código debe incluir manejo de errores de lectura y escritura.
6. Para establecer y cerrar conexiones, deben emplearse try, except, finally.
7. Las conexiones deben establecerse una sola vez al inicio del programa y se deben cerrar cuando corresponda.

Sube la solución en una carpeta llamada *mysql\_to\_postgres* en un repositorio github. Note que el postgres puede ser otro manejador, dependiendo de su selección.

**Pruebas de la solución**

Asegúrate de que el código de carga de datos de MySQL a PostgreSQL funcione correctamente. Para ello, escribe una prueba sencilla que compare el contenido de las tablas en ambas bases de datos.

**Requisitos de las pruebas**

1. Verifica la integridad de los datos entre cada par de tablas en MySQL y PostgreSQL. Es suficiente comprobar el número de registros en cada tabla.
2. Verifica el contenido de los registros dentro de cada tabla. Asegúrate de que todos los registros en PostgreSQL tengan los mismos valores de campos que en MySQL.
3. Realiza pruebas en las tablas que estén presentes en su BD.

Utiliza *assert* para comparar los datos esperados con los reales.

**Tarea de proyecto: Interfaz con Python Tkinter**

Ha llegado el momento de desarrollar una interfaz gráfica para interactuar con la base de datos migrada. Utilizando la biblioteca *Tkinter* de *Python*, deberás diseñar una aplicación que facilite a los usuarios gestionar los datos en el nuevo sistema.

**Criterios de finalización**

1. **Diseño de la interfaz**

* La aplicación debe contar con un diseño claro y fácil de usar.
* Incluir menús o botones para las principales acciones (visualizar, agregar, modificar y eliminar registros).

1. **Funcionalidades principales**

* Visualizar registros: Mostrar los datos almacenados en las tablas del nuevo manejador de BD (ejemplo: PostgreSQL) en una tabla o lista dentro de la interfaz.
* Agregar registros: Proporcionar un formulario para ingresar nuevos datos en las tablas.
* Modificar registros: Incluir un mecanismo para seleccionar un registro existente, editarlo y guardar los cambios.
* Eliminar registros: Permitir la eliminación segura de registros seleccionados.

1. **Conexión a la base de datos**

* Establecer una conexión con la base de datos al iniciar la aplicación y cerrarla al salir.
* Manejar errores de conexión y mostrar mensajes de error claros para el usuario.

1. **Validación**

* Validar los datos ingresados por los usuarios antes de enviarlos a la base de datos.
* Mostrar mensajes de confirmación o error según el caso.

Sube la solución en una carpeta llamada *tkinter\_app* en un repositorio de GitHub.

**Pruebas de la solución**

Asegúrate de que todas las funcionalidades de la interfaz funcionen correctamente mediante pruebas manuales o unitarias.

**Requisitos de las pruebas**

1. **Pruebas de las acciones principales**

* Verificar que los datos se muestren correctamente en la interfaz.
* Confirmar que la adición, modificación y eliminación de registros actualicen la base de datos.
* Validar que la búsqueda devuelva los resultados esperados.

1. **Validación de entradas**

* Comprobar que se manejan correctamente los errores de entrada (campos vacíos, formatos incorrectos, etc.).
* Confirmar que se muestren mensajes claros en caso de errores.

1. **Pruebas de integridad de datos**

* Asegúrate de que los cambios realizados desde la interfaz (adición, modificación, eliminación) se reflejen correctamente en la base de datos.
* Verificar que los datos en la base de datos permanezcan consistentes después de realizar varias operaciones.

1. **Pruebas de cierre de conexión**

* Verifica que la conexión a la base de datos se cierre correctamente al cerrar la aplicación.

**Ejemplo de interfaz**

1. **Una ventana principal con**

* Un menú o botones para las acciones principales.
* Una tabla que muestre los registros cargados desde la base de datos.

1. **Ventanas secundarias** (o cuadros de diálogo) para agregar, modificar y eliminar registros.

**Herramientas adicionales**

Puedes utilizar bibliotecas como *Pandas* para manipular los datos o *ttk* (complemento de Tkinter) para una interfaz más moderna.

**Consideraciones**

* Considera que la aplicación debe de funcionar antes y después de la migración de base de datos. Es decir, con el maneador antiguo y nuevo.
* El código debe ser modular, con funciones y clases cuando vea correspondiente.
* Deben tener un repositorio github con lo solicitado.
* Se debe presentar documentación, diagramas de clases, diagramas de relación entidad, etc.

**Rúbrica**

| ***Criterio*** | **Excelente** | **Bueno** | **Regular** | **Deficiente** | **SIS 112** | **SIS 122** |
| --- | --- | --- | --- | --- | --- | --- |
| *Documentación \*README* | Definición del problema clara y concisa. Documentación completa que incluye todos los requisitos, casos de uso y supuestos. Bien organizada y fácil de entender. Se tiene un repositorio Github en donde se muestra una estructura clara y concisa del proyecto | La definición del problema es comprensible. La documentación está casi completa y cubre la mayoría de los requisitos. Algunos problemas menores de organización. Se tiene un repositorio Github en donde se muestra una estructura deficiente del proyecto | La definición del problema es vaga. La documentación está incompleta o es difícil de seguir. Falta información clave. Se tiene un repositorio Github pero no se tiene una estructura del proyecto. | La definición del problema no está clara o falta. La documentación es mínima o está ausente. Repositorio github ausente. | 10 | 10 |
| *Diseño Relacional* | Diseño cumple hasta la 3ra forma normal, las entidades, relaciones y atributos están descritos de manera adecuada al sistema. | Diseño cumple hasta la 3ra forma normal pero con ligeras inconsistencias, la mayoría de entidades, relaciones y atributos están correctas | Diseño parcialmente normalizado, entidades, relaciones y atributos con errores o inconsistencias. | Diseño no normalizado, no tiene definidas correctamente las llaves primarias y/o foráneas. |  | 20 |
| *Scripts SQL* | Comandos DDL y DML estructurados de manera correcta, tipos de datos bien definidos. | Comandos DDL y DML con ligeros errores en tipos de datos. | Comandos DDL y DML con errores que impiden una completa funcionalidad. | Comandos sin uso de restricciones esenciales (null en llaves primarias). |  | 20 |
| *Migración de datos* | Migración completa y precisa, sin pérdida de datos. Se crea correctamente la nueva columna solicitada. No se han duplicado los datos. | Migración funcional con mínimas pérdidas de datos. | Migración con pérdidas significativas de datos. | Migración fallida. | 15 | 20 |
| *Lógica de Programación e Implementación* | El código está bien estructurado, es eficiente, legible y bien comentado. El manejo de errores es robusto. El diseño del algoritmo es apropiado y efectivo. Se utilizan las mejores prácticas de programación. La interfaz es coherente se trabaja modularmente. | El código es funcional, pero podría mejorarse en estructura, eficiencia o legibilidad. El manejo de errores es adecuado. | El código es funcional pero está mal estructurado, es ineficiente y/o tiene pocos comentarios. El manejo de errores es débil. | El código está mal estructurado, es ineficiente y difícil de entender. Falta manejo de errores. Puede no funcionar correctamente. | 25 |  |
| Interacción con la Base de Datos | Las interacciones con la base de datos son eficientes y manejan eficazmente la recuperación, modificación y almacenamiento de datos. Se utilizan consultas SQL adecuadas. Se implementan correctamente procedimientos o procedimientos almacenados (según el diagrama). La interfaz cumple con los requerimientos solicitados con las operaciones CRUD. | Las interacciones con la base de datos son funcionales, pero podrían mejorarse en términos de eficiencia o efectividad. Algunos problemas menores con las consultas SQL. | Las interacciones con la base de datos son ineficientes o contienen errores. Las consultas SQL están mal escritas o son incorrectas. | Las interacciones con la base de datos están mayormente rotas o ausentes. Errores significativos en las consultas SQL. | 10 | 15 |
| *Pruebas y Validación* | Las pruebas exhaustivas demuestran la funcionalidad de todas las características. Los casos de prueba cubren varios escenarios y casos límite. Los resultados están documentados. | Las pruebas son adecuadas y cubren la mayoría de las funcionalidades. Pueden existir algunos problemas menores. | Las pruebas son mínimas e incompletas. No cubren adecuadamente todas las características. | Se realizaron pocas o ninguna prueba. | 10 |  |
| *Adherencia al Diseño* | El proyecto refleja con precisión el diseño inicial, con desviaciones mínimas. El proyecto es defendido correctamente, el estudiante logra modificar según lo requerido sin ninguna dificultad. | El proyecto refleja en gran parte el diseño inicial, pero tiene algunas desviaciones menores. El proyecto es defendido correctamente, el estudiante logra modificar según lo requerido con poca dificultad. | El proyecto se desvía significativamente del diseño inicial. El proyecto es defendido correctamente, el estudiante logra modificar según lo requerido varias dificultades. | El proyecto tiene poca o ninguna relación con el diseño inicial. El proyecto no se logró defender | 30 | 15 |
|  |  |  |  |  |  | 100% |