ChatGPT

Manipulación de Datos (DML) y Transaccionalidad en Bases de Datos

Lenguaje de Manipulación de Datos (DML)

El **Data Manipulation Language (DML)** es un subconjunto de SQL utilizado para gestionar los datos almacenados en las tablas de una base de datos [1]. A través de comandos DML podemos **recuperar, insertar, modificar y eliminar** datos de las tablas. En particular, para la **modificación** de datos existentes nos enfocamos en las operaciones de inserción, actualización y borrado de registros (INSERT, UPDATE, DELETE). Estos comandos permiten agregar nuevos registros, cambiar valores de registros existentes o eliminar registros, respectivamente, siempre respetando las reglas y restricciones de la base de datos.

Inserción de datos (INSERT)

La sentencia `INSERT` se utiliza para **añadir nuevos registros** a una tabla [2]. Su sintaxis básica consiste en especificar la tabla destino, las columnas a poblar (opcionalmente) y los valores para cada columna. Por ejemplo:

```
INSERT INTO Empleados (nombre, apellido, salario)
VALUES ('Ana', 'Pérez', 3000);
```

En este ejemplo se inserta un nuevo empleado con nombre "Ana Pérez" y salario 3000 en la tabla **Empleados**. Es posible insertar múltiples filas en una sola sentencia (proporcionando varios grupos de valores) o incluso insertar resultados de una consulta en otra tabla (`INSERT ... SELECT ...`). Al realizar inserciones, se debe tener en cuenta que si existen restricciones como **NOT NULL** o **UNIQUE** en ciertas columnas, se deben proveer valores válidos que no violen dichas restricciones, de lo contrario la base de datos rechazará la inserción.

Cuando la tabla posee una columna que actúa como **clave primaria** numérica con generación automática (por ejemplo, un campo autoincremental o una secuencia), a menudo no es necesario especificar manualmente el identificador en la inserción. Muchas bases de datos relacionales soportan mecanismos para generar identificadores únicos automáticamente. Por ejemplo, Oracle y PostgreSQL utilizan **secuencias**, mientras que MySQL o SQL Server permiten definir la columna con la propiedad AUTO INCREMENT (o IDENTITY). Estos mecanismos garantizan que cada nuevo registro reciba un identificador único sin colisiones.

Actualización de datos (UPDATE)

La sentencia `UPDATE` se emplea para **modificar datos existentes** en una tabla [3]. Con `UPDATE` se puede cambiar el valor de una o varias columnas de uno o varios registros que ya existen. Su sintaxis general es:

```
UPDATE <Tabla>
SET <columna1> = <nuevo_valor1>, <columna2> = <nuevo_valor2>, ...
WHERE <condición>;
```

La cláusula `WHERE` indica **qué registros** se van a actualizar; si se omite, la actualización se aplicará a *todas* las filas de la tabla, por lo que es muy importante especificar la condición correcta para no modificar datos inadvertidamente. Por ejemplo, para incrementar en un 10% el salario de todos los empleados del departamento 5, se puede hacer:

```
UPDATE Empleados
SET salario = salario * 1.10
WHERE depto_id = 5;
```

Esta sentencia ajusta el salario de aquellos registros que cumplen la condición (`depto_id = 5`). Si la tabla tiene restricciones, las nuevas asignaciones deben respetar dichas reglas; por ejemplo, no podemos actualizar una columna clave primaria a un valor duplicado que ya exista en otra fila, porque violaría la restricción de unicidad.

Eliminación de datos (DELETE)

La sentencia `DELETE` se utiliza para **eliminar registros** de una tabla [4]. Su sintaxis básica es:

```
DELETE FROM <Tabla>
WHERE <condición>;
```

Al igual que con `UPDATE`, la cláusula `WHERE` determina qué registros serán eliminados. Si se omite `WHERE`, la operación `DELETE` borrará **todas** las filas de la tabla (operación que debe usarse con precaución). Por ejemplo, para borrar de la tabla *Empleados* al empleado con ID 101:

```
DELETE FROM Empleados
WHERE id = 101;
```

Eliminar registros afecta a los datos almacenados, y es irreversible una vez confirmada la transacción (ver sección de Transacciones más adelante). Siempre que se borra información, la base de datos verifica las restricciones existentes; por ejemplo, si se intenta borrar un registro que es referenciado por otra tabla mediante una clave foránea, la operación será rechazada a menos que se maneje adecuadamente la integridad referencial (como se explica en la siguiente sección).

Secuencias para asignar identificadores únicos

Las **secuencias** son objetos de la base de datos diseñados para generar valores numéricos secuenciales únicos, generalmente utilizados para asignar identificadores (IDs) a registros nuevos [5]. Emplear una secuencia garantiza que cada vez que necesitemos un nuevo identificador (por ejemplo, para la clave primaria de una tabla), obtengamos un número único que no colisionará con los ya existentes.

En sistemas como Oracle o PostgreSQL, primero se **crea una secuencia** mediante `CREATE SEQUENCE`, definiendo parámetros como valor inicial (*start with*) e incremento (*increment by*). Luego, en la sentencia `INSERT`, se utiliza la secuencia con una pseudocolumna especial (por ejemplo, `mi_secuencia.NEXTVAL`) para generar el siguiente número de la secuencia y asignarlo al campo clave. De este modo, múltiples usuarios pueden insertar simultáneamente sin riesgo de duplicar la clave primaria. En otras plataformas, esta funcionalidad puede lograrse marcando la columna como autoincremental (p.ej., `AUTO_INCREMENT` en MySQL o la propiedad `IDENTITY` en SQL Server). El objetivo es el mismo: **asignar automáticamente un identificador único** a cada nuevo registro sin intervención manual.

Por ejemplo, si tenemos una secuencia `sec_cliente` en Oracle, podríamos insertar un nuevo cliente así:

```
INSERT INTO Clientes(id, nombre, email)
VALUES (sec_cliente.NEXTVAL, 'Juan Pérez', 'jperez@example.com');
```

Aquí `sec_cliente.NEXTVAL` provee un ID único para el nuevo cliente de forma automática. El uso de secuencias y valores autoincrementales facilita la manipulación de datos al no tener que preocuparnos por generar manualmente identificadores únicos.

Integridad referencial en operaciones DML

La **integridad referencial** es una propiedad que garantiza la consistencia de las relaciones entre tablas mediante claves foráneas (foreign keys). Cuando dos tablas están relacionadas (por ejemplo, *Pedidos* referencia a *Clientes* a través de una clave foránea cliente_id), el gestor de la base de datos aplicará reglas para evitar inconsistencias o datos "huérfanos". En términos prácticos, esto implica varias consideraciones al insertar, actualizar o eliminar datos:

- **Inserción con clave foránea**: No se puede insertar un registro en una tabla hija (tabla *relacionada*) si en su columna de clave foránea se especifica un valor que no existe en la tabla padre (tabla *principal*) [6]. En caso contrario, estaríamos creando un registro huérfano (que referencia a un padre inexistente) y la base de datos rechazará la operación. Por ejemplo, no podríamos insertar un *Pedido* con `cliente_id = 50` si no existe un cliente con ID 50 en la tabla *Clientes*.

- **Eliminación de registros padre**: No se puede eliminar un registro de la tabla principal si existen registros hijos relacionados que dependan de él [7]. Por ejemplo, si intentamos borrar un cliente que tiene pedidos asociados, la base de datos impedirá la operación para no dejar pedidos huérfanos sin cliente [7]. Existen formas de manejar esto automáticamente, como habilitar la eliminación en *cascada* (cascade), de modo que al borrar el registro padre, el sistema borre también todos sus registros hijos relacionados. Si la opción **ON DELETE CASCADE** está definida en la clave foránea, entonces la acción de borrado del padre propagará la eliminación a los hijos automáticamente. De lo contrario, será necesario primero eliminar o re-asignar los registros hijos manualmente antes de borrar el padre.

- **Actualización de claves primarias**: Similar al caso anterior, no se puede cambiar (actualizar) el valor de una clave primaria en la tabla principal si con ello se rompe la referencia de los registros hijos [8]. Por ejemplo, no podríamos simplemente cambiar el código de un pedido en la tabla *Pedidos* si existen filas en *DetallePedidos* que lo referencian, ya que esos detalles quedarían

apuntando a un pedido que ya no existe [8]. Para estos casos también existe la opción **ON UPDATE CASCADE** en las claves foráneas, que si está habilitada permite que un cambio en la clave primaria del padre se refleje automáticamente en todos los hijos. Sin dicha opción, la actualización de una clave primaria relacionada requeriría actualizar o eliminar primero los registros dependientes.

En resumen, la integridad referencial **evita registros huérfanos** y mantiene la coherencia entre tablas relacionadas [9]. Cada operación DML que involucre tablas relacionadas debe considerar estas reglas. Si se intenta una operación que las infrinja, el sistema lanzará un error para proteger la consistencia de los datos. Los desarrolladores deben planificar las operaciones en el orden adecuado (por ejemplo, primero insertar en la tabla padre y luego en la hija; o primero borrar los hijos y luego el padre) o usar las opciones de cascada según corresponda para cumplir con la integridad referencial.

Restricciones de integridad en las tablas

Además de las claves foráneas, existen otras **restricciones (constraints)** definidas en las tablas que las operaciones DML deben respetar [10]. Estas restricciones garantizan la **integridad de los datos** dentro de una tabla individual. Algunas de las más comunes son:

- **Clave primaria (PRIMARY KEY)**: Una columna (o combinación de columnas) marcada como clave primaria debe ser única y no nula. Esto significa que al insertar registros, ningún valor duplicado puede existir en la columna de clave primaria, y tampoco se puede insertar un registro sin valor en esa columna. Intentar duplicar una clave primaria o dejarla nula resultará en error.

- **Claves únicas (UNIQUE)**: Garantizan que ciertos campos no repitan valores ya existentes en otras filas. Por ejemplo, si una tabla *Usuarios* tiene un campo UNIQUE para *email*, no podremos insertar dos usuarios con el mismo correo electrónico, ni actualizar el email de un usuario a uno que ya esté registrado por otro usuario.

- **Restricción NOT NULL**: Obliga a que una columna tenga siempre un valor (no se permiten valores NULL en esa columna). Por tanto, al insertar un nuevo registro o actualizar uno existente, debemos proporcionar un valor válido en las columnas marcadas como NOT NULL. Si omitimos una columna NOT NULL en un INSERT (y no tiene valor por defecto), o intentamos ponerla a NULL en un UPDATE, la operación será rechazada.

- **Restricciones CHECK**: Son condiciones lógicas que deben cumplirse en los datos de una columna o conjunto de columnas. Por ejemplo, una restricción CHECK puede exigir que el valor de *edad* esté entre 0 y 120. Si intentamos insertar o actualizar un dato que viole esa condición, la base de datos no lo permitirá.

- **Valores por defecto (DEFAULT)**: Si bien no es una restricción de integridad en el sentido estricto (es más bien una propiedad de la columna), vale la pena mencionar que las columnas pueden tener un valor *default*. Esto significa que si en una inserción no proporcionamos valor para esa columna, la base de datos automáticamente asignará el valor por defecto especificado. Esto ayuda a simplificar inserciones y a asegurar que ciertos campos siempre tengan al menos un valor predefinido.

En todos los casos, el **motor de la base de datos valida las restricciones** en cada operación DML. Si una sentencia `INSERT` o `UPDATE` intenta violar una restricción (ya sea clave primaria, foránea,

unique, not null, check, etc.), la operación se abortará y se informará un error. Por eso es importante conocer el esquema y las reglas de integridad de la base de datos al manipular los datos, para cumplirlas adecuadamente y mantener la **consistencia**. En suma, los comandos DML deben respetar las restricciones definidas en las tablas para asegurar que la base de datos permanezca en un estado válido [10].

Transaccionalidad en Bases de Datos

Una vez que comprendemos cómo insertar, actualizar y eliminar datos, es crucial entender cómo mantener la **consistencia** y **fiabilidad** de esas operaciones mediante el uso de **transacciones**. La *transaccionalidad* se refiere a la capacidad de agrupar operaciones de base de datos de manera que se ejecuten como una unidad lógica, cumpliendo ciertas propiedades que garantizan la integridad de la información incluso en casos de fallos o accesos concurrentes.

¿Qué es una transacción y por qué es importante?

En bases de datos, una **transacción** es una secuencia de operaciones que se ejecutan como una unidad indivisible de trabajo. Esto significa que todas las operaciones de la transacción deben completarse con éxito para que los cambios surtan efecto permanente; si alguna de las operaciones falla, **toda la transacción se revierte**, dejando la base de datos como estaba antes de iniciarla [11]. En otras palabras, una transacción sigue el principio de "todo o nada": o se realizan *todas* las modificaciones propuestas, o **ninguna**.

La importancia de las transacciones radica en que garantizan la **integridad y consistencia de los datos** durante operaciones múltiples o críticas. Algunas razones clave por las que son cruciales las transacciones son:

- **Integridad de los datos**: Las transacciones aseguran que la base de datos no quede en un estado inconsistente. Sin este mecanismo, si ocurriese un fallo a mitad de una serie de operaciones, podríamos terminar con datos incompletos o corruptos [12]. Por ejemplo, en un sistema bancario, si se transfiere dinero de una cuenta a otra, se debita una cuenta y se acredita la otra; ambas operaciones deben ocurrir, o ninguna, para que el dinero no “desaparezca” ni se duplique.

- **Operaciones complejas y seguras**: Permiten ejecutar conjuntos de pasos lógicos como una sola unidad segura. Por ejemplo, una compra en línea puede involucrar verificar inventario, restar stock, registrar el pago y generar una orden; gracias a las transacciones, podemos asegurarnos de que *todos* esos pasos se completen correctamente antes de consolidar la venta, lo que hace posible implementar procesos de negocio multietapa de forma fiable [13].

- **Concurrencia controlada y rendimiento**: Bien gestionadas, las transacciones mejoran el rendimiento y la consistencia en entornos multiusuario. Permiten que múltiples usuarios trabajen sobre la base de datos al mismo tiempo sin interferir unos con otros, aislando sus operaciones hasta que se confirmen. Además, evitan condiciones de carrera o lecturas de datos intermedios. Los sistemas de bases de datos utilizan transacciones para manejar eficientemente muchas operaciones simultáneas, manteniendo la integridad sin sacrificar rendimiento [14].

En resumen, las transacciones son fundamentales para asegurar que los datos permanezcan confiables. Una operación compleja que implique varios cambios dispersos (por ejemplo, mover dinero entre cuentas, o registrar un pedido con todos sus detalles) debe ejecutarse bajo una transacción para

garantizar que la base de datos no quede en estado parcial ante eventuales errores. Si algo falla, la transacción **"da marcha atrás"** y deja todo como estaba, protegiendo la integridad de la información de la aplicación.

Propiedades ACID de las transacciones

Para describir el comportamiento correcto de las transacciones, se utiliza el acrónimo **ACID**, que corresponde a cuatro propiedades fundamentales que debe cumplir cualquier sistema transaccional confiable [15] :

- **Atomicidad**: La transacción es **atómica**, es decir, **indivisible**. Todas las operaciones enmarcadas en la transacción se consideran una sola unidad; si **cualquier** parte de la transacción falla, *toda* la transacción se cancela y se deshacen los cambios parciales realizados [16] . Esto garantiza el *todo o nada*: o se aplican todos los cambios, o ninguno.

- **Consistencia**: Cada transacción lleva la base de datos de un **estado válido a otro estado válido**, respetando todas las reglas de integridad definidas (restricciones, disparadores, etc.) [17] . Si los datos eran consistentes al inicio de la transacción, al finalizar (tras un commit) también lo serán, porque la transacción no viola ninguna de las reglas de negocio ni de integridad. Cualquier fallo en cumplir una restricción provocará la anulación de la transacción, manteniendo la base de datos consistente.

- **Aislamiento**: Las transacciones concurrentes se ejecutan como si estuvieran **aisladas** unas de otras. Esto significa que los cambios en progreso en una transacción **no serán visibles** para otras transacciones hasta que se confirmen (commit). Cada transacción "cree" que es la única operando en la base de datos en un momento dado [18] . El aislamiento evita interferencias entre operaciones simultáneas; en la práctica se logran diferentes niveles de aislamiento (READ COMMITTED, REPEATABLE READ, SERIALIZABLE, etc.), pero el principio general es prevenir que transacciones paralelas corrompan datos o lean información intermedia de otra transacción incompleta.

- **Durabilidad**: Una vez que una transacción ha sido **confirmada** (se ha hecho commit), sus efectos persisten de forma permanente en la base de datos [19] . Los cambios no se perderán incluso si ocurre una caída del sistema, fallo de energía u otro desastre. El sistema de almacenamiento (y los mecanismos como el registro de transacciones o *logs*) se encarga de asegurar que, después de confirmar una transacción, los datos quedan guardados de forma fiable (por ejemplo, escribiéndolos en disco, replicándolos, etc.). La durabilidad brinda confianza de que los datos comprometidos sobrevivirán ante eventualidades.

Estas cuatro propiedades juntas garantizan que las transacciones se ejecuten de manera **confiable** [20] . Los sistemas gestores de bases de datos (DBMS) implementan internamente estas propiedades para cada transacción. Como desarrolladores o administradores, generalmente no tenemos que manejar ACID manualmente, sino aprovechar que el DBMS lo hace por nosotros al delimitar transacciones con comandos (BEGIN, COMMIT, ROLLBACK). No obstante, conocer estas propiedades nos ayuda a entender el comportamiento de nuestras operaciones en escenarios concurrentes o ante fallos.

Confirmación de una transacción (COMMIT)

Una transacción comienza típicamente con una instrucción de inicio (explícita o implícita) y agrupa varias operaciones DML. Para **finalizar** la transacción con éxito, haciendo **permanentes** todos los

cambios realizados, se utiliza la instrucción `COMMIT`. El comando **COMMIT** le indica a la base de datos que *confirme* (confirme/valide) las operaciones de la transacción, volcando definitivamente todos los cambios al almacenamiento persistente [21]. En otras palabras, después de un COMMIT, las modificaciones quedan guardadas y ya no es posible deshacerlas mediante ROLLBACK (la transacción se considera concluida).

Por ejemplo, en SQL estándar podríamos hacer:

```
BEGIN;
   UPDATE Cuentas SET saldo = saldo - 100 WHERE cuenta_id = 1;
   UPDATE Cuentas SET saldo = saldo + 100 WHERE cuenta_id = 2;
COMMIT;
```

Aquí se inicia una transacción, se realizan dos actualizaciones (transferir 100 de la cuenta 1 a la 2) y finalmente el `COMMIT` confirma la transacción. Si ambas operaciones se ejecutaron sin problemas, el commit consolidará los nuevos saldos en la base de datos de forma permanente. Si hubiese ocurrido un error antes del COMMIT (por ejemplo, que la cuenta 2 no exista), lo normal es que la transacción se aborte y ningún cambio quede aplicado.

Es importante mencionar que ciertas operaciones de definición de datos (DDL), como crear o eliminar tablas, pueden implicar *commits implícitos*. Por ejemplo, en muchos sistemas, al ejecutar un `CREATE TABLE` o `DROP TABLE`, el gestor de base de datos realiza automáticamente un commit antes y después de la operación DDL [21]. Esto significa que cualquier transacción en curso se confirmará alrededor de la instrucción DDL. Por ello, se suele recomendar no mezclar operaciones DDL con DML dentro de la misma transacción si se quiere tener control explícito.

En resumen, **COMMIT confirma** la transacción actual, haciendo irreversibles los cambios efectuados durante ella. Después de un commit exitoso, la única forma de "deshacer" sería iniciando nuevas operaciones compensatorias en otra transacción; pero la transacción ya confirmada no se puede revertir directamente.

Reversión de una transacción (ROLLBACK)

Si durante una transacción ocurre un problema, o decidimos descartar los cambios por alguna razón, debemos **revertir** la transacción. Para ello se utiliza la instrucción `ROLLBACK`. El comando **ROLLBACK** deshace todas las modificaciones de datos realizadas desde el inicio de la transacción actual (o desde un punto de guardado específico, si se hubiera definido alguno) [22]. Esto restaura la base de datos al estado consistente que tenía antes de comenzar la transacción, liberando además los recursos y bloqueos que estuviesen en uso por dicha transacción.

Continuando con el ejemplo anterior de transferencia bancaria, si después de debitar los 100 de la cuenta 1 ocurriese un error al acreditar la cuenta 2 (imaginemos que la cuenta 2 no existe), se podría ejecutar un ROLLBACK para anular la transacción entera. Así, el débito en la cuenta 1 también se revertiría, regresando su saldo al valor original, como si nada hubiera pasado. Esto evita que $100 "desaparezcan" de la cuenta 1 sin haberse acreditado en ninguna otra; la operación incompleta no tendrá efecto permanente.

La sintaxis básica es simplemente:

```
ROLLBACK;
```

(lo cual revierte la transacción activa completa). Si se usan **savepoints** (puntos de guardado dentro de una transacción), es posible hacer rollback parcial a un punto intermedio, pero en contextos básicos, ROLLBACK sin parámetros siempre deshará todo lo hecho en la transacción en curso.

Una vez ejecutado un ROLLBACK, la transacción se termina (abortada) y todos los cambios quedan descartados. Cabe señalar que si la transacción ya había hecho COMMIT, ya no es posible hacer rollback de esa transacción – el rollback solo se aplica a transacciones activas no confirmadas. Por eso, es importante decidir realizar COMMIT solo cuando estemos seguros de los cambios; hasta antes del commit, tenemos la opción de rollback para deshacer.

En resumen, **ROLLBACK revierte** una transacción en caso de error o cancelación, garantizando que la base de datos no incorpore cambios parciales indeseados [22]. Esto es esencial para mantener la *consistencia* ante fallos: ante cualquier problema, hacemos rollback y los datos quedan tal como estaban al inicio de la transacción, sin efectos colaterales.

Modo autocommit (confirmación automática)

Muchos sistemas de gestión de bases de datos operan por defecto en un modo llamado **autocommit** (confirmación automática). En **modo autocommit**, cada sentencia SQL individual se trata implícitamente como una transacción completa: la base de datos realiza un **commit automático al final de cada sentencia** exitosa [23]. Es decir, **cada instrucción DML se confirma de inmediato** una vez ejecutada, sin necesidad de un COMMIT explícito por parte del usuario. Si la instrucción se ejecuta correctamente, sus cambios quedan permanentes; si ocurre un error durante la ejecución, el SGBD revierte los cambios de esa instrucción en particular (efectuando un rollback automático de esa única sentencia) [24].

El modo autocommit facilita las interacciones simples con la base de datos, ya que no requiere gestionar manualmente las transacciones en operaciones triviales (por ejemplo, un simple `UPDATE` se confirma solo). Por ejemplo, en MySQL el autocommit suele estar activado por defecto, de modo que uno tras ejecutar `INSERT ...` puede asumir que esa inserción ya está guardada permanentemente sin hacer nada más. Sin embargo, el autocommit no es deseable cuando queremos agrupar varias operaciones en una misma transacción. En esos casos, debemos **desactivar el autocommit** temporalmente o iniciar una transacción explícita.

Cuando el autocommit está activado (`AUTOCOMMIT = ON`), *no* es posible deshacer varias operaciones juntas, porque cada una ya quedó confirmada por separado inmediatamente después de ejecutarse. Por eso, si deseamos usar `ROLLBACK` para tener la posibilidad de revertir una serie de acciones, es necesario trabajar con autocommit deshabilitado (`AUTOCOMMIT = OFF`), iniciar una transacción manualmente, y luego decidir hacer COMMIT o ROLLBACK según corresponda [25]. En entornos de programación, esto suele implicar comandos como `BEGIN TRANSACTION` (o simplemente `START TRANSACTION` en SQL) para empezar, seguido de múltiples operaciones DML, y finalmente un `COMMIT` o `ROLLBACK` explícito.

En sistemas donde el autocommit está *deshabilitado* por defecto (por ejemplo, Oracle), el desarrollador/DBA tiene el control total: ninguna sentencia DML se confirma hasta que se ejecute un COMMIT. Mientras tanto, en sistemas con autocommit *habilitado* por defecto (como MySQL), si queremos agrupar sentencias debemos iniciar una transacción (lo que automáticamente suspende el autocommit para ese

bloque) o, dependiendo de la interfaz, establecer la sesión en modo “manual commit”. Muchos clientes o *drivers* de bases de datos permiten configurar esto. Por ejemplo, en JDBC (Java) uno puede llamar a `setAutoCommit(false)` sobre la conexión para manejar las confirmaciones manualmente [26].

En resumen, **autocommit ON** implica simplicidad pero menos control: cada sentencia se confirma inmediatamente [23]. **Autocommit OFF** nos permite manejar explícitamente grupos de sentencias bajo transacciones (usando COMMIT/ROLLBACK). Es importante que los desarrolladores conozcan el comportamiento por defecto de la base de datos que utilizan, para evitar sorpresas – por ejemplo, pensando que varias operaciones estarán en la misma transacción cuando en realidad cada una se confirmó por separado. Ajustando el autocommit según la necesidad, podemos aprovechar la transaccionalidad adecuada en nuestras operaciones.

Fuentes: Las definiciones y ejemplos anteriores se basan en documentación y recursos actualizados sobre SQL y sistemas de bases de datos, incluyendo tutoriales y manuales técnicos [1] [2] [27] [11], entre otros. Estas referencias respaldan las prácticas recomendadas al usar sentencias DML y manejar transacciones para mantener la integridad y consistencia de la información en una base de datos relacional. Se enfatiza la importancia de seguir las reglas de integridad referencial [28] y las propiedades ACID [29] para asegurar que la manipulación de datos produzca resultados correctos y confiables.

---

[1] [2] [3] [4] [10] Comandos DML en SQL | HDP115
https://hdp115.caeher.com/courses/sql/comandos-dml-en-sql/

[5] Descripción : Secuencias (create sequence - currval - nextval - drop sequence) (Oracle)
https://www.tutorialesprogramacionya.com/oracleya/temarios/descripcion.php?cod=193

[6] [7] [8] [9] [27] [28] Crear, modificar o eliminar una relación - Soporte técnico de Microsoft
https://support.microsoft.com/es-es/topic/crear-modificar-o-eliminar-una-relaci%C3%B3n-dfa453a7-0b6d-4c34-a128-fdebc7e686af

[11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [29] ¿Qué es una transacción en base de datos?
https://consultoriainformatica.net/que-es-una-transaccion-en-base-de-datos/

[21] [25] 6.1.- Hacer cambios permanentes. | DAM_BD05_Contenido_CIDEAD
https://sarreplec.caib.es/pluginfile.php/9863/mod_resource/content/2/61_hacer_cambios_permanentes.html

[22] TRANSACCIÓN DE REVERSIÓN (Transact-SQL) - SQL Server | Microsoft Learn
https://learn.microsoft.com/es-es/sql/t-sql/language-elements/rollback-transaction-transact-sql?view=sql-server-ver17

[23] [24] [26] AUTOCOMMIT - Oracle VS Postgres - JGS Endeavors
https://jgsendeavors.com/autocommit-oracle-vs-postgres-configuracion/