Bases de datos

Libro de texto para los módulos
"Gestión de Base de Datos (0372)"
y
"Bases de datos (0484)"
de Formación Profesional

José J. Durán

Para acceder a los contenidos de este libro, y otros títulos, visita:
`https://github.com/cavefish-dev/libros`

Edición: 31 de agosto de 2025

Índice general

C Contenidos exclusivos del módulo Gestión de Base de Datos (0372)

D Contenidos exclusivos del módulo Bases de datos (0484)

A

Introducción

1

Presentación del libro

Este libro está dirigido a estudiantes de Formación Profesional que desean adquirir una comprensión sólida sobre bases de datos y su gestión. El objetivo principal es proporcionar una guía práctica y accesible, combinando teoría con ejemplos y ejercicios que faciliten el aprendizaje activo.

A lo largo de los capítulos, se abordarán los conceptos fundamentales de las bases de datos, el diseño y la manipulación de datos, así como el uso de sistemas gestores como MySQL y PostgreSQL. Además, se incluyen contenidos específicos para los módulos de Formación Profesional, adaptados a los requisitos de los ciclos formativos.

El enfoque del libro es progresivo: se parte de los conceptos básicos y se avanza hacia temas más complejos, permitiendo al lector construir sus conocimientos de manera estructurada. Se fomenta la participación activa mediante ejercicios prácticos y recomendaciones para el desarrollo de habilidades en la gestión y programación de bases de datos.

Al finalizar el libro, el lector será capaz de comprender el funcionamiento de las bases de datos, diseñar esquemas eficientes, realizar consultas y operaciones avanzadas, y aplicar buenas prácticas en el desarrollo de proyectos relacionados con la gestión de datos.

2

Cómo usar este libro

A lo largo de este libro encontrarás la información organizada en capítulos y secciones, cada una dedicada a un tema específico. Dentro de cada capítulo, se presentarán ejemplos de código usando el siguiente formato:

```sql
UPDATE ejemplo
SET columna1 = 'nuevo_valor'
WHERE columna2 = 'condicion';
```

También se incluyen cajas de información destacada, como consejos, advertencias y ejercicios, para resaltar puntos importantes y facilitar la comprensión de los conceptos. Estas cajas están diseñadas para ser visualmente atractivas y fáciles de identificar.

Consejo

A lo largo del libro, encontrarás consejos útiles para mejorar tu comprensión y habilidades. Estos consejos están diseñados para ayudarte a evitar errores comunes y optimizar tu proceso de aprendizaje.

Importante

Es fundamental prestar atención a las advertencias y notas importantes que se presentan en el libro. Estas secciones destacan aspectos críticos que pueden afectar tu comprensión o implementación de los conceptos, o que incluyan información adicional relevante para el tema tratado, pero que sean demasiado avanzados para el nivel del lector.

Ejercicio 2.1

Se incluirán ejercicios prácticos al final de cada capítulo para que puedas aplicar lo aprendido. Estos ejercicios están diseñados para ser desafiantes y fomentar la práctica activa. Los ejercicios se presentan en un formato claro y conciso, permitiendo al lector enfocarse en la resolución de problemas

específicos.

2.1. Estructura del libro

Este libro se divide en varias partes, cada una de las cuales aborda un aspecto diferente de los contenidos asociados a cada módulo. Se recomienda seguir el orden de los capítulos para construir una comprensión progresiva de los conceptos.

La parte A (*Introducción*) proporciona una visión general de los contenidos de este libro, y como poder ejecutar los ejemplos y ejercicios propuestos.

La parte B (*Contenidos*) aborda los conceptos fundamentales de las bases de datos, el diseño y la manipulación de datos, así como el uso de sistemas gestores como MySQL y PostgreSQL. Además, se incluyen contenidos específicos para los módulos de Formación Profesional, adaptados a los requisitos de los ciclos formativos.

La parte C (*Contenidos exclusivos del módulo Gestión de Base de Datos (0372)*) incluye temas avanzados como la administración de bases de datos, la optimización de consultas y la seguridad en el manejo de datos.

La parte D (*Contenidos exclusivos del módulo Bases de datos (0484)*) se centra en temas más avanzados de programación en bases de datos, así como hace una introducción a las bases de datos no relacionales.

2.2. Cómo ejecutar los ejemplos

Para el seguimiento de los ejemplos mostrados en este libro, se recomienda usar los gestores de bases de datos MySQL y PostgreSQL. Ambos sistemas son ampliamente utilizados en la industria y ofrecen una amplia gama de características y funcionalidades.

Para facilitar su ejecución, se recomienda usar el siguiente fichero de *Docker compose*, el cual despliega adicionalmente una instancia de PHPMyAdmin y PGAdmin, lo cual facilitará al usuario poder ver la estructura de las bases de datos creadas.

```
name: ejemplos_sql
services:
 mysql:
  image: mysql:8.0
  restart: always
  environment:
   MYSQL_ROOT_PASSWORD: rootpassword
```

```
    MYSQL_DATABASE: exampledb
    MYSQL_USER: exampleuser
    MYSQL_PASSWORD: examplepass
  ports:
    - "3306:3306"
  volumes:
    - ./init_db_mysql:/docker-entrypoint-initdb.d
  healthcheck:
    test: ["CMD", "mysqladmin" ,"ping", "-h", "localhost"]
    timeout: 20s
    retries: 10

postgres:
  image: postgres:15
  restart: always
  environment:
    POSTGRES_DB: exampledb
    POSTGRES_USER: exampleuser
    POSTGRES_PASSWORD: examplepass
  ports:
    - "5432:5432"
  volumes:
    - ./init_db_psql:/docker-entrypoint-initdb.d
  healthcheck:
    test: ["CMD-SHELL", "pg_isready", "-d", "db_prod"]
    timeout: 20s
    retries: 10

phpmyadmin:
  image: phpmyadmin/phpmyadmin
  restart: always
  environment:
    PMA_HOST: mysql
    PMA_USER: exampleuser
    PMA_PASSWORD: examplepass
  ports:
    - "8080:80"
  depends_on:
    - mysql

pgadmin:
  image: dpage/pgadmin4
  restart: always
  environment:
    PGADMIN_DEFAULT_EMAIL: admin@example.com
    PGADMIN_DEFAULT_PASSWORD: admin
    PGADMIN_CONFIG_SERVER_MODE: "False"
    PGADMIN_CONFIG_MASTER_PASSWORD_REQUIRED: "False"
  ports:
```

```
57      - "8081:80"
58    depends_on:
59      - postgres
60    volumes:
61      - ./pgadmin_servers.json:/pgadmin4/servers.json
```

Importante

Aunque las bases de datos estén accesibles desde tu ip local, los gestores visuales usarán el nombre del contenedor para conectarse a ellas. P.e.: `mysql:3306` y `postgres:5432` desde los gestores, y `localhost:3306` y `localhost:5432` desde el host.

Nota: Si quieres que las bases de datos estén inicializadas, te recomendamos que clones el repositorio del libro en GitHub. De esta manera, tendrás acceso a todos los scripts de inicialización necesarios para cargar los datos de ejemplo en las bases de datos.

B

Contenidos

3

Almacenamiento de la información

En este capítulo se exploran los conceptos fundamentales relacionados con el almacenamiento de la información en sistemas informáticos. Se analizan las distintas formas de persistencia, desde los ficheros tradicionales hasta los sistemas avanzados de bases de datos, así como los mecanismos y herramientas que permiten gestionar grandes volúmenes de datos de manera eficiente y segura. Además, se abordan aspectos legales sobre la protección de datos y se introduce el impacto de Big Data en la gestión y análisis de la información. El objetivo es proporcionar una visión global de las tecnologías y normativas que sustentan el almacenamiento y la explotación de datos en la actualidad.

3.1. Ficheros

Los ficheros son una de las formas más básicas de persistencia de la información en sistemas informáticos. Existen diferentes tipos de ficheros según su estructura y modo de acceso:

- **Ficheros planos**: almacenan los datos de manera secuencial, sin ningún tipo de estructura adicional. Son fáciles de implementar y utilizar, pero su acceso suele ser lento cuando se requiere buscar información específica.
- **Ficheros indexados**: incorporan estructuras de índices que permiten localizar rápidamente los registros dentro del fichero. Esto mejora el rendimiento en operaciones de búsqueda y actualización.
- **Ficheros de acceso directo**: permiten acceder a los registros de manera inmediata mediante una dirección calculada, sin necesidad de recorrer el fichero secuencialmente. Son útiles cuando se conoce la posición exacta del dato.
- **Otros tipos**: existen variantes como ficheros binarios, ficheros de texto, ficheros organizados por bloques, entre otros, que se adaptan a diferentes necesidades de almacenamiento y acceso.

El uso de ficheros como medio de persistencia es adecuado para aplicaciones sencillas o cuando no se requiere una gestión compleja de los datos. Sin embargo, presentan limitaciones en cuanto a concurrencia, integridad y escalabilidad, lo que ha llevado al desarrollo de sistemas más avanzados como las bases de datos.

3.2. Bases de datos

Las bases de datos son sistemas organizados para almacenar, gestionar y recuperar grandes volúmenes de información de manera eficiente y segura. A diferencia de los ficheros tradicionales, las bases de datos permiten una gestión avanzada de los datos, facilitando la concurrencia, la integridad y la escalabilidad.

El concepto de base de datos implica la existencia de un conjunto de datos estructurados, organizados según un modelo de datos. Los modelos más comunes son:

- **Modelo relacional**: organiza la información en tablas relacionadas entre sí mediante claves. Es el modelo más utilizado en la actualidad por su flexibilidad y potencia.
- **Modelo jerárquico**: estructura los datos en forma de árbol, donde cada registro tiene un único padre.
- **Modelo de red**: permite relaciones más complejas entre registros, formando grafos.
- **Modelo orientado a objetos**: almacena información en forma de objetos, integrando datos y comportamiento.
- **Bases de datos NoSQL**: diseñadas para manejar grandes volúmenes de datos no estructurados o semiestructurados, como documentos, grafos o pares clave-valor.

Según la ubicación de la información, las bases de datos pueden ser:

- **Centralizadas**: toda la información se almacena en un único servidor o ubicación física.
- **Distribuidas**: los datos se reparten entre varios servidores o ubicaciones, permitiendo mayor disponibilidad y escalabilidad.

El uso de bases de datos como medio de persistencia es fundamental en aplicaciones modernas, ya que ofrecen mecanismos para garantizar la integridad, la seguridad y el acceso eficiente a la información, además de facilitar la gestión de grandes volúmenes de datos y el trabajo colaborativo entre múltiples usuarios.

3.3. Sistemas gestores de bases de datos

Los sistemas gestores de bases de datos (SGBD) son programas que permiten crear, administrar y manipular bases de datos de manera eficiente y segura. Sus funciones principales incluyen:

- **Definición de datos**: permiten definir la estructura de la base de datos, como tablas, relaciones, índices y restricciones.
- **Manipulación de datos**: facilitan la inserción, modificación, eliminación y consulta de los datos almacenados.
- **Control de acceso**: gestionan los permisos de usuarios y roles, garantizando la seguridad y privacidad de la información.
- **Integridad de datos**: aseguran que los datos cumplan con las reglas y restricciones definidas, evitando inconsistencias.
- **Gestión de concurrencia**: permiten el acceso simultáneo de varios usuarios, manteniendo la coherencia de los datos.
- **Recuperación ante fallos**: proporcionan mecanismos para restaurar la base de datos en caso de errores o pérdidas de información.

Los componentes principales de un SGBD son:

- **Motor de almacenamiento**: gestiona la organización física de los datos en disco.
- **Procesador de consultas**: interpreta y ejecuta las instrucciones de consulta y manipulación de datos.
- **Gestor de transacciones**: controla las operaciones que deben ejecutarse de forma atómica y segura.
- **Sistema de seguridad**: administra usuarios, roles y permisos.
- **Interfaz de usuario**: permite la interacción con el sistema, ya sea mediante comandos o herramientas gráficas.

Existen diferentes tipos de SGBD según el modelo de datos que utilizan:

- **SGBD relacionales**: basados en el modelo de tablas y relaciones (por ejemplo, MySQL, PostgreSQL, Oracle).

- **SGBD orientados a objetos**: gestionan datos como objetos (por ejemplo, db4o, ObjectDB).
- **SGBD NoSQL**: diseñados para datos no estructurados o semiestructurados (por ejemplo, MongoDB, Cassandra).
- **SGBD jerárquicos y de red**: utilizan modelos menos comunes, como IMS o IDMS.

3.3.1. Herramientas gráficas para la realización de consultas

Los SGBD suelen proporcionar herramientas gráficas que facilitan la administración y consulta de las bases de datos. Estas herramientas permiten a los usuarios:

- Crear y modificar tablas, relaciones e índices de forma visual.
- Realizar consultas mediante asistentes o editores visuales de SQL.
- Visualizar resultados de consultas en formato de tablas o gráficos.
- Administrar usuarios, permisos y copias de seguridad sin necesidad de comandos complejos.

Algunos ejemplos de herramientas gráficas son MySQL Workbench, pgAdmin, Oracle SQL Developer y Microsoft SQL Server Management Studio. Estas aplicaciones mejoran la productividad y facilitan el trabajo tanto a administradores como a usuarios finales.

3.4. Bases de datos centralizadas y distribuidas

Las bases de datos centralizadas almacenan toda la información en un único servidor o ubicación física. Este enfoque facilita la administración y el control de los datos, pero puede presentar problemas de disponibilidad, escalabilidad y tolerancia a fallos, ya que la caída del servidor central afecta a todo el sistema.

Por otro lado, las bases de datos distribuidas reparten los datos entre varios servidores o ubicaciones geográficas. Esto permite mejorar la disponibilidad, el rendimiento y la escalabilidad, ya que los usuarios pueden acceder a los datos desde diferentes puntos y el sistema puede continuar funcionando aunque falle alguno de los nodos. Sin embargo, la gestión de la coherencia y la integridad de los datos se vuelve más compleja.

Una de las técnicas fundamentales en bases de datos distribuidas es la fragmentación, que consiste en dividir la base de datos en partes más pequeñas

llamadas fragmentos. Los fragmentos pueden distribuirse entre distintos nodos del sistema. Existen tres tipos principales de fragmentación:

- **Fragmentación horizontal**: cada fragmento contiene un subconjunto de las filas de una tabla, según ciertos criterios (por ejemplo, por región geográfica).
- **Fragmentación vertical**: cada fragmento contiene un subconjunto de las columnas de una tabla, agrupando atributos que suelen consultarse juntos.
- **Fragmentación mixta**: combina la fragmentación horizontal y vertical para adaptar la distribución de los datos a las necesidades específicas de la aplicación.

La fragmentación permite optimizar el acceso a los datos, reducir el tráfico de red y mejorar la seguridad, ya que cada nodo almacena solo la información necesaria. Sin embargo, requiere mecanismos para mantener la coherencia y la integridad entre los fragmentos distribuidos.

3.5. Legislación sobre protección de datos

La protección de datos personales es un aspecto fundamental en el almacenamiento y gestión de la información. Diversos países y regiones han desarrollado leyes específicas para garantizar la privacidad y los derechos de los ciudadanos frente al tratamiento de sus datos.

En España, la Ley Orgánica de Protección de Datos (LOPD) establece el marco legal para la recogida, almacenamiento y uso de datos personales. La LOPD regula los derechos de los titulares de los datos, las obligaciones de las entidades que los gestionan y las medidas de seguridad que deben implementarse para evitar accesos no autorizados, pérdidas o alteraciones. Además, contempla sanciones para el incumplimiento de la normativa.

A nivel europeo, el Reglamento General de Protección de Datos (RGPD o GDPR, por sus siglas en inglés) armoniza la legislación de los países miembros de la Unión Europea. El RGPD refuerza los derechos de los ciudadanos, como el derecho al acceso, rectificación, cancelación, oposición y portabilidad de los datos. También exige el consentimiento explícito para el tratamiento de datos, la notificación de brechas de seguridad y la designación de un delegado de protección de datos en determinadas organizaciones.

En otros países existen normativas similares, como la Ley de Privacidad del Consumidor de California (CCPA) en Estados Unidos, la Ley de Protección de Datos Personales en México, la Ley de Protección de Información Personal

(POPIA) en Sudáfrica, y la Ley General de Protección de Datos (LGPD) en Brasil. Estas leyes comparten principios comunes: transparencia, consentimiento, seguridad y derechos de los titulares de los datos.

El cumplimiento de la legislación sobre protección de datos es esencial en el diseño y operación de sistemas de almacenamiento y bases de datos, especialmente cuando se gestionan datos personales o sensibles. Las organizaciones deben implementar políticas y tecnologías que garanticen la privacidad y la seguridad, adaptándose a las normativas vigentes en cada jurisdicción.

3.5.1. Puntos críticos de la LOPD

La Ley Orgánica de Protección de Datos (LOPD) establece una serie de principios y obligaciones clave para el tratamiento de datos personales en España. Los puntos más críticos de la LOPD son:

- **Consentimiento**: Es obligatorio obtener el consentimiento explícito del titular antes de recopilar o tratar sus datos personales.
- **Finalidad**: Los datos deben ser recogidos para fines determinados, explícitos y legítimos, y no pueden ser tratados posteriormente de manera incompatible con dichos fines.
- **Calidad de los datos**: Los datos personales deben ser exactos, pertinentes y no excesivos en relación con la finalidad para la que se recogen.
- **Derechos de los titulares**: Los ciudadanos tienen derecho a acceder, rectificar, cancelar y oponerse al tratamiento de sus datos personales.
- **Seguridad**: Las entidades responsables deben implementar medidas técnicas y organizativas para garantizar la seguridad de los datos y evitar accesos no autorizados, pérdidas o alteraciones.
- **Deber de confidencialidad**: Quienes intervienen en el tratamiento de datos personales están obligados a mantener la confidencialidad incluso después de finalizar su relación con la entidad responsable.
- **Comunicación y cesión de datos**: La cesión de datos a terceros requiere el consentimiento del titular, salvo excepciones previstas por la ley.
- **Registro de actividades**: Las organizaciones deben mantener un registro actualizado de las actividades de tratamiento de datos personales.
- **Sanciones**: El incumplimiento de la LOPD puede conllevar sanciones económicas y administrativas importantes.

3.5.2. Puntos críticos del RGPD

El Reglamento General de Protección de Datos (RGPD) establece principios y obligaciones fundamentales para el tratamiento de datos personales en la Unión Europea. Los puntos más críticos del RGPD son:

- **Licitud, lealtad y transparencia**: El tratamiento de datos debe ser legal, justo y transparente para el titular.
- **Limitación de la finalidad**: Los datos deben recogerse con fines específicos, explícitos y legítimos, y no tratarse posteriormente de manera incompatible con dichos fines.
- **Minimización de datos**: Solo deben tratarse los datos personales estrictamente necesarios para la finalidad perseguida.
- **Exactitud**: Los datos deben ser exactos y mantenerse actualizados; deben adoptarse medidas para suprimir o rectificar datos inexactos.
- **Limitación del plazo de conservación**: Los datos deben conservarse únicamente durante el tiempo necesario para los fines del tratamiento.
- **Integridad y confidencialidad**: Deben aplicarse medidas técnicas y organizativas adecuadas para proteger los datos contra el tratamiento no autorizado o ilícito y contra su pérdida, destrucción o daño accidental.
- **Responsabilidad proactiva**: El responsable del tratamiento debe poder demostrar el cumplimiento de los principios del RGPD.
- **Derechos de los interesados**: El RGPD refuerza derechos como el acceso, rectificación, supresión, limitación del tratamiento, portabilidad y oposición.
- **Consentimiento**: El consentimiento debe ser libre, específico, informado e inequívoco, y puede ser retirado en cualquier momento.
- **Notificación de brechas de seguridad**: Es obligatorio informar a la autoridad de control y, en ciertos casos, a los afectados, en caso de violaciones de seguridad de los datos personales.
- **Delegado de protección de datos**: En determinadas organizaciones, es obligatorio designar un DPD para supervisar el cumplimiento del RGPD.
- **Sanciones**: El RGPD contempla sanciones económicas muy elevadas en caso de incumplimiento, que pueden alcanzar hasta el 4 % de la facturación anual global.

3.6. Big Data

Big Data se refiere al manejo y procesamiento de grandes volúmenes de datos que superan la capacidad de las herramientas tradicionales de gestión y análisis. Estos datos pueden ser estructurados, semiestructurados o no estructurados, y provienen de diversas fuentes como redes sociales, sensores, transacciones comerciales, dispositivos móviles, entre otros.

La introducción de Big Data ha transformado la forma en que las organizaciones gestionan la información, permitiendo almacenar, procesar y analizar datos a gran escala en tiempo real. Las tecnologías asociadas incluyen sistemas distribuidos, bases de datos NoSQL, procesamiento paralelo y plataformas como Hadoop y Spark.

El análisis de datos en entornos Big Data implica extraer conocimiento útil a partir de grandes conjuntos de información. Se utilizan técnicas de minería de datos, aprendizaje automático, estadística avanzada y visualización para identificar patrones, tendencias y relaciones ocultas. El análisis puede ser descriptivo, predictivo o prescriptivo, según el objetivo de negocio.

La inteligencia de negocios (Business Intelligence, BI) aprovecha el potencial de Big Data para apoyar la toma de decisiones estratégicas. Mediante el uso de herramientas de BI, las organizaciones pueden transformar datos en información relevante, generar informes, cuadros de mando y realizar análisis en profundidad. Esto permite optimizar procesos, identificar oportunidades de mercado, anticipar riesgos y mejorar la competitividad.

En resumen, Big Data es clave para el desarrollo de soluciones innovadoras en diversos sectores, facilitando el análisis avanzado de datos y la generación de inteligencia de negocios que impulsa el crecimiento y la eficiencia organizacional.

Resumen

En este capítulo se han revisado los conceptos esenciales sobre el almacenamiento de la información en sistemas informáticos. Se han comparado los ficheros tradicionales y las bases de datos, destacando las ventajas de estas últimas en cuanto a gestión, integridad y escalabilidad. Se han analizado los sistemas gestores de bases de datos y sus herramientas, así como las diferencias entre bases de datos centralizadas y distribuidas, incluyendo la fragmentación. Además, se ha abordado la legislación sobre protección de datos, especialmente la LOPD y el RGPD, y se ha introducido el impacto de Big Data en la gestión y análisis de grandes volúmenes de información. Todo ello proporciona una visión global de las tecnologías y normativas que sustentan el almacenamiento y explotación de datos en la actualidad.

4

Bases de datos relacionales

Las bases de datos relacionales constituyen el pilar fundamental de la gestión moderna de la información en organizaciones, empresas y sistemas informáticos. Este modelo, propuesto por Edgar F. Codd en la década de 1970, se basa en la organización de los datos en tablas relacionadas entre sí, lo que permite estructurar, consultar y mantener grandes volúmenes de información de manera eficiente y segura.

En este capítulo se exploran los conceptos esenciales del modelo relacional, el funcionamiento de las tablas, las relaciones entre datos y el uso del lenguaje SQL para definir, manipular y proteger la información. Se abordan las principales operaciones y restricciones que garantizan la integridad y coherencia de los datos, así como las herramientas para optimizar el rendimiento y controlar el acceso a la base de datos.

El objetivo es proporcionar una base sólida para comprender cómo funcionan las bases de datos relacionales y cómo aplicarlas en el diseño y administración de sistemas de información robustos y escalables.

4.1. Modelo de datos

El modelo de datos es una representación abstracta que define cómo se estructuran, almacenan y manipulan los datos en una base de datos. En el contexto de bases de datos relacionales, el modelo de datos se basa en la organización de la información en tablas, donde cada tabla representa una entidad y cada fila corresponde a un registro individual. Las columnas de la tabla definen los atributos de la entidad.

Este modelo permite establecer relaciones entre diferentes tablas mediante claves, facilitando la integridad y consistencia de los datos. Además, proporciona mecanismos para definir restricciones, tipos de datos y operaciones que pueden realizarse sobre la información almacenada. El modelo relacional es ampliamente utilizado por su simplicidad, flexibilidad y capacidad para manejar grandes volúmenes de datos de manera eficiente.

4.2. El lenguaje SQL

El lenguaje SQL (Structured Query Language) es el estándar utilizado para interactuar con bases de datos relacionales. Permite definir, manipular y consultar los datos almacenados en las tablas. SQL se compone de varios subconjuntos de instrucciones, entre los que destacan:

DDL (Data Definition Language) Permite crear, modificar y eliminar estructuras de la base de datos, como tablas, índices y vistas. Ejemplo: `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`.

DML (Data Manipulation Language) Se utiliza para insertar, actualizar, eliminar y consultar datos. Ejemplo: `INSERT`, `UPDATE`, `DELETE`, `SELECT`.

DCL (Data Control Language) Gestiona permisos y privilegios de acceso. Ejemplo: `GRANT`, `REVOKE`.

TCL (Transaction Control Language) Controla transacciones y su integridad. Ejemplo: `COMMIT`, `ROLLBACK`.

A continuación se muestra un ejemplo básico de consulta en SQL para obtener los nombres de los estudiantes inscritos en el curso de Matemáticas:

```
1 SELECT Estudiantes.Nombre
2 FROM Estudiantes
3 JOIN Inscripciones ON Estudiantes.ID = Inscripciones.↪
    ID_Estudiante
4 JOIN Cursos ON Inscripciones.ID_Curso = Cursos.ID
5 WHERE Cursos.Nombre = 'Matemáticas';
```

SQL es un lenguaje declarativo, lo que significa que el usuario indica qué información desea obtener, sin especificar cómo debe realizarse la búsqueda. Esto facilita la gestión y el análisis de grandes volúmenes de datos en sistemas relacionales.

4.2.1. Lenguaje de descripción de datos (DDL)

El Lenguaje de Descripción de Datos (DDL, por sus siglas en inglés) es el conjunto de instrucciones SQL que permite definir y modificar la estructura de las bases de datos. A través del DDL, es posible crear, alterar y eliminar tablas, así como establecer restricciones y relaciones entre ellas. Estas operaciones son fundamentales para el diseño y la organización de la información en sistemas relacionales.

En este capítulo, se profundizará en las principales instrucciones del DDL, mostrando ejemplos prácticos de cómo se utilizan para crear y gestionar las estructuras de una base de datos.

4.2.2. Lenguaje de manipulación de datos (DML)

El Lenguaje de Manipulación de Datos (DML, por sus siglas en inglés) es el conjunto de instrucciones SQL que permite realizar operaciones sobre los datos almacenados en las bases de datos. A través del DML, los usuarios pueden insertar, actualizar, eliminar y consultar registros en las tablas.

Entraremos en detalle sobre las principales instrucciones del DML en los capítulos 5 y 6, mostrando ejemplos prácticos de cómo se utilizan para gestionar la información de manera eficiente y segura.

4.2.3. Lenguaje de control de datos (DCL)

El Lenguaje de Control de Datos (DCL, por sus siglas en inglés) es el conjunto de instrucciones SQL que permite gestionar los permisos y privilegios de acceso a los objetos de la base de datos. Mediante el DCL, los administradores pueden otorgar o revocar derechos sobre tablas, vistas y otros elementos, asegurando la seguridad y el control de la información.

En este capítulo se abordarán con mayor detalle los distintos aspectos relacionados con la gestión de usuarios y privilegios en bases de datos relacionales.

4.2.4. Lenguaje de control de transacciones (TCL)

El Lenguaje de Control de Transacciones (TCL, por sus siglas en inglés) es el conjunto de instrucciones SQL que permite gestionar las transacciones en una base de datos. Las transacciones son unidades de trabajo que agrupan una o más operaciones DML (como INSERT, UPDATE o DELETE) y garantizan la integridad de los datos.

Entraremos en detalle sobre las principales instrucciones del TCL en el capítulo 6, mostrando ejemplos prácticos de cómo se utilizan.

4.3. Terminología del modelo relacional

En el modelo relacional, existen varios conceptos fundamentales que permiten comprender cómo se estructuran y relacionan los datos:

Relación Es una tabla que contiene datos organizados en filas y columnas. Cada relación representa una entidad del mundo real.

Tupla Es una fila dentro de una relación. Cada tupla corresponde a un registro único.

Atributo Es una columna de la relación. Cada atributo describe una característica de la entidad representada.

Dominio Es el conjunto de valores permitidos para un atributo.

Clave primaria Es un atributo (o conjunto de atributos) que identifica de manera única cada tupla en una relación.

Clave ajena Es un atributo que establece una relación entre dos tablas, permitiendo vincular registros de diferentes entidades.

Esquema Es la estructura que define las relaciones, atributos y restricciones de la base de datos.

Instancia Es el conjunto de datos almacenados en la base de datos en un momento determinado.

Estos términos son esenciales para comprender el funcionamiento y diseño de bases de datos relacionales.

Ejemplo

Consideremos una base de datos para gestionar información de estudiantes y cursos en una universidad. El modelo relacional permite organizar estos datos en tablas relacionadas.

Estudiantes		
ID	**Nombre**	**Edad**
1	Ana López	20
2	Juan Pérez	22
3	María Ruiz	21

Cursos		
ID	**Nombre**	**Créditos**
101	Matemáticas	6
102	Historia	4
103	Informática	5

Inscripciones	
ID_Estudiante	**ID_Curso**
1	101
2	103
3	102
1	103

En este ejemplo:

- Cada tabla representa una **relación**.
- Cada fila es una **tupla** con información específica.
- Las columnas son **atributos** de las entidades.
- Los campos `ID` funcionan como **clave primaria** en cada tabla.
- Los campos `ID_Estudiante` y `ID_Curso` en la tabla `Inscripciones` son **claves ajenas** que vinculan estudiantes y cursos.

Ejercicio 4.1

Define al menos tres tablas que permitan almacenar la información relacionada con un garaje. Debe representar información sobre los vehículos, los propietarios y las plazas de estacionamiento. Especifica los atributos principales de cada tabla e indica las claves primarias y las claves ajenas necesarias para relacionarlas. Presenta un ejemplo con algunos datos de muestra en cada tabla.

4.4. Tipos de datos

En las bases de datos relacionales, los tipos de datos determinan la naturaleza de los valores que pueden almacenarse en los atributos de una tabla. Los principales tipos de datos incluyen:

- **Enteros (`INT`, `SMALLINT`, `BIGINT`):** Permiten almacenar números enteros, útiles para identificadores, cantidades, edades, etc.
- **Números decimales (`FLOAT`, `DOUBLE`, `DECIMAL`):** Se emplean para valores numéricos con decimales, como precios, medidas o porcentajes.
- **Cadenas de caracteres (`CHAR`, `VARCHAR`, `TEXT`):** Permiten almacenar texto, como nombres, direcciones o descripciones. `CHAR` tiene longitud fija, mientras que `VARCHAR` es de longitud variable.
- **Fechas y horas (`DATE`, `TIME`, `DATETIME`, `TIMESTAMP`):** Se utilizan para almacenar información temporal, como fechas de nacimiento, registros de eventos, etc.
- **Booleanos (`BOOLEAN`):** Representan valores de verdad, típicamente `TRUE` o `FALSE`.

- **Otros tipos:** Algunos sistemas permiten almacenar datos binarios (`BLOB`), enumeraciones (`ENUM`), o conjuntos (`SET`).

La elección adecuada del tipo de dato para cada atributo es fundamental para garantizar la integridad, eficiencia y precisión en el almacenamiento y manipulación de la información.

4.4.1. El valor NULL

El valor `NULL` en bases de datos relacionales representa la ausencia de un valor o dato desconocido en un atributo. No significa cero, cadena vacía ni valor por defecto, sino que indica que no se ha especificado ningún valor para ese campo.

El uso de `NULL` permite modelar situaciones en las que cierta información no está disponible, no es aplicable o aún no se ha registrado. Por ejemplo, el campo `Email` de un estudiante podría ser `NULL` si no se ha proporcionado una dirección de correo electrónico.

Al trabajar con `NULL`, es importante tener en cuenta que las operaciones y comparaciones pueden comportarse de manera diferente. Por ejemplo, una comparación directa con `NULL` usando el operador igual (`=`) no devuelve verdadero; en su lugar, se debe utilizar `IS NULL` o `IS NOT NULL` para comprobar si un atributo tiene o no el valor nulo.

Ejemplo de consulta para seleccionar estudiantes sin correo electrónico registrado:

```sql
SELECT Nombre
FROM Estudiantes
WHERE Email IS NULL;
```

Ejemplo

Supongamos que queremos definir la tabla `Estudiantes` utilizando distintos tipos de datos para sus atributos principales. El siguiente ejemplo muestra una instrucción SQL para crear la tabla y algunos registros de muestra:

```sql
CREATE TABLE Estudiantes (
  ID INT PRIMARY KEY,
  Nombre VARCHAR(50),
  FechaNacimiento DATE,
  Email VARCHAR(100),
  Activo BOOLEAN
);

-- Inserción de datos de ejemplo. Se explicará más adelante.
INSERT INTO Estudiantes (ID, Nombre, FechaNacimiento, Email, ↪
  Activo) VALUES
```

```
(1, 'Ana López', '2003-05-12', 'ana.lopez@universidad.edu', TRUE),
(2, 'Juan Pérez', '2001-11-23', 'juan.perez@universidad.edu', TRUE),
(3, 'María Ruiz', '2002-08-30', 'maria.ruiz@universidad.edu', FALSE);

-- Muestra todas las tuplas en la tabla Estudiantes.
SELECT * FROM Estudiantes;
```

En este ejemplo:

- `ID` utiliza el tipo `INT` para almacenar un identificador numérico.
- `Nombre` y `Email` emplean `VARCHAR` para cadenas de texto de longitud variable.
- `FechaNacimiento` usa el tipo `DATE` para almacenar fechas.
- `Activo` utiliza el tipo `BOOLEAN` para indicar si el estudiante está activo.

4.5. Claves primarias

Una **clave primaria** es un atributo o conjunto de atributos que identifica de manera única cada tupla (fila) dentro de una relación (tabla). Su principal función es garantizar que no existan dos registros con el mismo valor en la clave primaria, asegurando la unicidad y permitiendo acceder rápidamente a los datos.

Las claves primarias cumplen las siguientes características:

- No pueden contener valores `NULL`.
- Deben ser únicas en toda la tabla.
- Pueden estar formadas por uno o varios atributos (clave compuesta).

En SQL, la clave primaria se define al crear la tabla utilizando la restricción `PRIMARY KEY`. Por ejemplo:

```
CREATE TABLE Vehiculos (
  Matricula VARCHAR(10) PRIMARY KEY,
  Marca VARCHAR(30),
  Modelo VARCHAR(30),
  Año INT
);
```

En este ejemplo, el atributo `Matricula` es la clave primaria de la tabla `Vehiculos`, lo que significa que cada vehículo debe tener una matrícula única.

Si se requiere una clave compuesta, se puede definir así:

```sql
CREATE TABLE Inscripciones (
  ID_Estudiante INT,
  ID_Curso INT,
  Fecha DATE,
  PRIMARY KEY (ID_Estudiante, ID_Curso)
);
```

Aquí, la combinación de `ID_Estudiante` e `ID_Curso` identifica de forma única cada inscripción.

El uso adecuado de claves primarias es fundamental para mantener la integridad de los datos y facilitar la relación entre diferentes tablas mediante claves ajenas.

Consejo

En aquellos casos que la clave primaria sea un número entero, es recomendable utilizar un campo autoincremental. Esto facilita la gestión de los identificadores únicos y evita errores al insertar nuevos registros.
Cada SGBD dispone de sus propios mecanismos para claves autoincrementales. En PostgreSQL, se utiliza el tipo de dato `SERIAL`, mientras que en MySQL se emplea `AUTO_INCREMENT`.

```sql
CREATE TABLE Estudiantes (
  ID SERIAL PRIMARY KEY,
  Nombre VARCHAR(100),
  FechaNacimiento DATE,
  Email VARCHAR(100),
  Activo BOOLEAN
);
```

Ejemplo de tabla con campo autoincremental en PostgreSQL

```sql
CREATE TABLE Estudiantes (
  ID INT AUTO_INCREMENT PRIMARY KEY,
  Nombre VARCHAR(100),
  FechaNacimiento DATE,
  Email VARCHAR(100),
  Activo BOOLEAN
);
```

Ejemplo de tabla con campo autoincremental en MySQL

Ejercicio 4.2

Define las claves primarias y ajenas para las siguientes tablas relacionadas con un garaje:

- **Vehiculos**: `Matricula` (`VARCHAR(10)`) como clave primaria, `Marca` (`VARCHAR(30)`), `Modelo` (`VARCHAR(30)`), `PropietarioID` (`INT`) como clave ajena.
- **Propietarios**: `ID` (`INT`) como clave primaria, `Nombre` (`VARCHAR(50)`), `Telefono` (`VARCHAR(15)`).
- **Plazas**: `Numero` (`INT`) como clave primaria, `MatriculaVehiculo` (`VARCHAR(10)`) como clave ajena.

Ejemplo de datos de muestra:

Vehiculos		
Matricula	**Marca**	**PropietarioID**
1234ABC	Toyota	1
5678DEF	Ford	2
9012GHI	Honda	1

Propietarios		
ID	**Nombre**	**Telefono**
1	Laura Gómez	600123456
2	Pedro Martínez	600654321

Plazas	
Numero	**MatriculaVehiculo**
1	1234ABC
2	5678DEF
3	9012GHI

Indica las claves primarias y ajenas en cada tabla y explica cómo se relacionan entre sí.
Además, crea la consulta SQL para crear esas tablas.

4.6. Restricciones de validación

Las restricciones de validación son reglas que se aplican a los datos almacenados en una base de datos para garantizar su integridad, coherencia y calidad. Estas restricciones se definen en el esquema de la base de datos y se aplican automáticamente cada vez que se realizan operaciones de inserción, actualización o eliminación de datos.

Los principales tipos de restricciones en el modelo relacional son:

- **NOT NULL:** Impide que un atributo tome el valor `NULL`, asegurando que siempre se proporcione un dato.
- **UNIQUE:** Garantiza que los valores de un atributo (o conjunto de atributos) sean únicos en toda la tabla, evitando duplicados.
- **PRIMARY KEY:** Combina las restricciones `NOT NULL` y `UNIQUE` para identificar de forma única cada tupla.
- **FOREIGN KEY:** Asegura que los valores de un atributo coincidan con los valores existentes en la clave primaria de otra tabla, manteniendo la integridad referencial.
- **CHECK:** Permite definir condiciones lógicas que deben cumplir los valores de un atributo. Por ejemplo, limitar el rango de edades o asegurar que una fecha sea posterior a otra.
- **DEFAULT:** Establece un valor por defecto para un atributo si no se especifica uno al insertar una nueva tupla.

Estas restricciones son fundamentales para prevenir errores, inconsistencias y datos inválidos en la base de datos, contribuyendo a la fiabilidad y precisión de la información almacenada.

4.6.1. Claves ajenas

Las **claves ajenas** (`FOREIGN KEY`) son atributos en una tabla que establecen una relación con la clave primaria de otra tabla. Su propósito principal es mantener la **integridad referencial**, asegurando que los valores de la clave ajena correspondan a registros válidos en la tabla referenciada.

Ejemplo

Por ejemplo, si en la tabla `Inscripciones` existe el atributo `ID_Estudiante` como clave ajena, este debe coincidir con un valor existente en la columna `ID` de la tabla `Estudiantes`. De este modo, se evita que se creen inscripciones para estudiantes que no existen.

- **Uno a muchos:** Un propietario puede tener varios vehículos, pero cada vehículo pertenece a un solo propietario.

- **Muchos a muchos:** Un estudiante puede inscribirse en varios cursos y cada curso puede tener varios estudiantes, lo que se representa mediante una tabla intermedia con claves ajenas.

En SQL, las claves ajenas se definen con la restricción `FOREIGN KEY`, especificando la columna referenciada y la tabla destino. Además, se pueden establecer acciones ante la eliminación o actualización de registros en la tabla referenciada, como `ON DELETE CASCADE` o `ON UPDATE SET NULL`, para controlar el comportamiento de la relación.

El uso correcto de claves ajenas es fundamental para evitar inconsistencias y garantizar que las relaciones entre tablas sean válidas y seguras.

4.6.2. Efectos ON DELETE y ON UPDATE

Los efectos `ON DELETE` y `ON UPDATE` permiten definir el comportamiento de las claves ajenas cuando se elimina o actualiza un registro en la tabla referenciada. Estas opciones son fundamentales para mantener la integridad referencial y controlar cómo se propagan los cambios entre tablas relacionadas.

Las acciones más comunes son:

- `CASCADE`: Si se elimina o actualiza un registro en la tabla principal, la acción se propaga automáticamente a las filas relacionadas en la tabla secundaria. Por ejemplo, al borrar un propietario, se eliminan todos sus vehículos asociados.
- `SET NULL`: Al eliminar o actualizar el registro principal, los valores de la clave ajena en la tabla secundaria se establecen a `NULL`.
- `SET DEFAULT`: Los valores de la clave ajena se reemplazan por el valor por defecto definido para esa columna.
- `RESTRICT`: Impide la eliminación o actualización si existen registros relacionados en la tabla secundaria.
- `NO ACTION`: Similar a `RESTRICT`, no permite la acción si hay dependencias, pero la comprobación puede diferirse según el sistema.

Estas opciones se especifican al definir la restricción de clave ajena en la sentencia `CREATE TABLE` o `ALTER TABLE`. Por ejemplo:

```
1 CREATE TABLE Vehiculos (
2   Matricula VARCHAR(10) PRIMARY KEY,
3   Marca VARCHAR(30),
4   PropietarioID INT,
```

```
  FOREIGN KEY (PropietarioID) REFERENCES Propietarios(ID)
    ON DELETE RESTRICT
    ON UPDATE CASCADE
);
```

En este ejemplo, si se actualiza un propietario, los cambios se aplican automáticamente a los vehículos asociados, pero prohíbe la eliminación si existen vehículos relacionados. El uso adecuado de estas acciones ayuda a evitar datos huérfanos y mantiene la coherencia entre las tablas relacionadas.

Ejemplo

Supongamos que queremos definir restricciones para la tabla `Estudiantes`:

```
CREATE TABLE Estudiantes (
  ID INT PRIMARY KEY,
  Nombre VARCHAR(50) NOT NULL,
  FechaNacimiento DATE CHECK (FechaNacimiento <= CURRENT_DATE),
  Email VARCHAR(100) UNIQUE,
  Activo BOOLEAN DEFAULT TRUE
);
```

En este ejemplo:

- `ID` es la clave primaria, por lo que es `NOT NULL` y `UNIQUE`.
- `Nombre` no puede ser nulo (`NOT NULL`).
- `FechaNacimiento` debe ser una fecha anterior o igual a la actual (`CHECK`).
- `Email` debe ser único en la tabla (`UNIQUE`).
- `Activo` tendrá el valor `TRUE` por defecto si no se especifica (`DEFAULT`).

Ahora supongamos que queremos asegurar que cada inscripción en la tabla `Inscripciones` corresponda a un estudiante y a un curso válidos. Para ello, definimos claves ajenas (`FOREIGN KEY`) en los atributos `ID_Estudiante` y `ID_Curso` de la tabla `Inscripciones`, que referencian las claves primarias de las tablas `Estudiantes` y `Cursos` respectivamente:

```
CREATE TABLE Estudiantes (
  ID INT PRIMARY KEY,
  Nombre VARCHAR(50) NOT NULL,
  FechaNacimiento DATE CHECK (FechaNacimiento <= CURRENT_DATE),
  Email VARCHAR(100) UNIQUE,
  Activo BOOLEAN DEFAULT TRUE
);

CREATE TABLE Cursos (
```

```
  ID INT PRIMARY KEY,
  Nombre VARCHAR(50) NOT NULL,
  Creditos INT CHECK (Creditos > 0)
);

CREATE TABLE Inscripciones (
  ID_Estudiante INT,
  ID_Curso INT,
  Fecha DATE,
  PRIMARY KEY (ID_Estudiante, ID_Curso),
  FOREIGN KEY (ID_Estudiante) REFERENCES Estudiantes(ID),
  FOREIGN KEY (ID_Curso) REFERENCES Cursos(ID)
);
```

En este ejemplo, las restricciones `FOREIGN KEY` garantizan que no se pueda insertar una inscripción con un `ID_Estudiante` o `ID_Curso` que no existan en sus respectivas tablas, manteniendo la integridad referencial entre ellas.

Ejercicio 4.3

Define restricciones de validación adecuadas para las tablas del garaje (`Vehiculos`, `Propietarios`, `Plazas`). Especifica al menos una restricción `NOT NULL`, una `UNIQUE`, una `CHECK` y una `FOREIGN KEY` en alguna tabla. Explica la función de cada restricción y cómo contribuye a la integridad de los datos.

A continuación, escribe las sentencias SQL para crear las tablas con las restricciones indicadas.

4.7. Modificación de tablas

En SQL, la modificación de tablas se realiza principalmente mediante la instrucción `ALTER TABLE`. Esta instrucción permite agregar, eliminar o modificar columnas, así como cambiar restricciones y relaciones.

Las operaciones más comunes incluyen:

- **Agregar una columna:** Permite añadir nuevos atributos a la tabla.
- **Eliminar una columna:** Quita atributos que ya no son necesarios.
- **Modificar una columna:** Cambia el tipo de dato, el nombre o las restricciones de un atributo existente.
- **Agregar o eliminar restricciones:** Permite definir o quitar restricciones como `NOT NULL`, `UNIQUE`, `CHECK`, `FOREIGN KEY`, etc.

La sintaxis básica para modificar una tabla es:

```
ALTER TABLE nombre_tabla
ADD COLUMN nuevo_atributo tipo_de_dato [restricciones];

ALTER TABLE nombre_tabla
DROP COLUMN nombre_atributo;

ALTER TABLE nombre_tabla
MODIFY COLUMN nombre_atributo nuevo_tipo_de_dato [restricciones];

ALTER TABLE nombre_tabla
ADD CONSTRAINT nombre_restriccion restriccion;
```

Ejemplo

Supongamos que queremos agregar el atributo `Telefono` a la tabla `Estudiantes`:

```
ALTER TABLE Estudiantes ADD COLUMN Telefono VARCHAR(15) UNIQUE;
```

Si necesitamos eliminar la columna `Email` de la tabla `Estudiantes`:

```
ALTER TABLE Estudiantes DROP COLUMN Email;
```

Para modificar el tipo de dato de la columna `Edad` en la tabla `Estudiantes`:

```
ALTER TABLE Estudiantes MODIFY COLUMN Edad SMALLINT;
```

Estas operaciones permiten adaptar la estructura de las tablas a nuevas necesidades sin perder los datos existentes.

Ejercicio 4.4

Realiza las siguientes modificaciones en las tablas del garaje:

- Agrega una columna `Color` (`VARCHAR(20)`) a la tabla `Vehiculos`.
- Elimina la columna `Modelo` de la tabla `Vehiculos`.
- Modifica el tipo de dato de la columna `Telefono` en la tabla `Propietarios` para que sea `VARCHAR(20)`.

Escribe las sentencias SQL correspondientes y explica el efecto de cada una.

4.8. Eliminación de tablas

La eliminación de tablas en una base de datos relacional se realiza mediante la instrucción `DROP TABLE`. Esta operación elimina por completo la estructura de la tabla y todos los datos almacenados en ella. Es importante tener precaución al utilizar esta instrucción, ya que la información borrada no se puede recuperar fácilmente.

La sintaxis básica es:

```
DROP TABLE nombre_tabla;
```

Si la tabla tiene restricciones de claves ajenas, es posible que sea necesario eliminar primero las relaciones o utilizar opciones específicas del sistema de gestión de bases de datos para forzar la eliminación.

Ejemplo

Supongamos que queremos eliminar la tabla `Estudiantes`:

```
DROP TABLE Estudiantes;
```

Esto eliminará la tabla `Estudiantes` y todos sus datos. Si existen restricciones de claves ajenas en otras tablas que dependen de `Estudiantes`, el sistema puede impedir la eliminación hasta que se resuelvan dichas dependencias.

Ejercicio 4.5

Escribe las sentencias SQL para eliminar las tablas `Vehiculos`, `Propietarios` y `Plazas` del garaje. Explica las implicaciones de esta operación y qué precauciones se deben tomar antes de ejecutarla.

4.9. Índices

Los **índices** en bases de datos relacionales son estructuras auxiliares que permiten acelerar la búsqueda y el acceso a los datos en las tablas. Funcionan de manera similar a los índices de un libro, facilitando la localización rápida de registros sin necesidad de recorrer toda la tabla.

Las principales características de los índices son:

- **Velocidad de consulta:** Mejoran significativamente el rendimiento de las operaciones de búsqueda, filtrado y ordenación.
- **Tipos de índices:** Los más comunes son los índices simples (sobre una sola columna) y los índices compuestos (sobre varias columnas). Existen

también índices únicos, que garantizan que los valores indexados no se repitan.

- **Impacto en las modificaciones:** Aunque aceleran las consultas, los índices pueden ralentizar las operaciones de inserción, actualización y eliminación, ya que la estructura del índice debe mantenerse actualizada.
- **Transparencia:** Los índices son gestionados por el sistema de base de datos y no afectan la lógica de las consultas SQL, aunque pueden influir en el plan de ejecución.
- **Espacio adicional:** Requieren espacio extra en disco para almacenar la estructura del índice.

En SQL, los índices se crean con la instrucción `CREATE INDEX`. Esta instrucción permite definir un índice sobre una o varias columnas de una tabla, especificando el nombre del índice y las columnas involucradas. También es posible crear índices únicos utilizando `CREATE UNIQUE INDEX`, lo que garantiza que los valores indexados no se repitan en la columna o combinación de columnas seleccionadas.

Ejemplo

Para crear un índice sobre la columna `Nombre` de la tabla `Estudiantes`:

```
CREATE INDEX idx_nombre_estudiantes ON Estudiantes(Nombre);
```

Para garantizar la unicidad de los valores, se puede crear un índice único:

```
CREATE UNIQUE INDEX idx_email_estudiantes ON Estudiantes(Email);
```

El uso adecuado de índices es fundamental para optimizar el rendimiento de las bases de datos, especialmente en tablas con grandes volúmenes de información y consultas frecuentes.

Ejercicio 4.6

Crea un índice para optimizar la consulta de vehículos por marca y otro índice único para asegurar que no haya dos propietarios con el mismo teléfono. Explica la función de cada índice y cómo mejoran el rendimiento o la integridad de la base de datos.

4.10. Vistas

Las **vistas** en bases de datos relacionales son tablas virtuales que se definen a partir de una consulta SQL sobre una o varias tablas reales. Una vista no

almacena datos por sí misma, sino que muestra los resultados de la consulta cada vez que se accede a ella. Esto permite simplificar el acceso a la información, ocultar detalles complejos y mejorar la seguridad al restringir el acceso a ciertos datos.

Las principales características y usos de las vistas son:

- **Simplicidad:** Permiten presentar datos de forma más sencilla y comprensible, agrupando o filtrando información relevante.
- **Seguridad:** Facilitan el control de acceso, ya que los usuarios pueden consultar solo los datos definidos en la vista, sin acceder directamente a las tablas originales.
- **Reutilización:** Se pueden reutilizar consultas complejas mediante vistas, evitando repetir el mismo código en diferentes partes de la aplicación.
- **Actualización:** Algunas vistas permiten modificar los datos subyacentes si cumplen ciertos requisitos, aunque no todas son actualizables.

La instrucción `CREATE VIEW` permite definir una vista asignándole un nombre y especificando la consulta SQL que determina los datos que mostrará. La sintaxis básica es:

```
CREATE VIEW nombre_vista AS
SELECT columnas
FROM tablas
WHERE condiciones;
```

Las vistas pueden incluir filtrado, agrupamiento, combinaciones de tablas (`JOIN`) y cualquier operación permitida en una consulta SQL. Una vez creada, la vista puede utilizarse en consultas como si fuera una tabla, facilitando el acceso y la gestión de la información.

Ejemplo

Supongamos que queremos crear una vista para mostrar únicamente los nombres y correos electrónicos de los estudiantes activos:

```
CREATE VIEW VistaEstudiantesActivos AS
SELECT Nombre, Email
FROM Estudiantes
WHERE Activo = TRUE;
```

Esta vista permite consultar fácilmente los estudiantes que están activos, ocultando otros detalles de la tabla original. Por ejemplo, para obtener la lista de correos electrónicos de estudiantes activos, basta con ejecutar:

```
SELECT Email FROM VistaEstudiantesActivos;
```

Ejercicio 4.7

Crea una vista que muestre la información de los vehículos junto con el nombre y teléfono de su propietario. Explica cómo esta vista puede facilitar la consulta de datos en el garaje y mejorar la seguridad al restringir el acceso a información sensible.

4.11. Usuarios y privilegios

En los sistemas de bases de datos relacionales, la gestión de usuarios y privilegios es fundamental para garantizar la seguridad y el control de acceso a la información. Cada usuario puede tener diferentes niveles de permisos, que determinan las operaciones que puede realizar sobre la base de datos.

4.11.1. Usuarios

Un **usuario** es una entidad (persona, aplicación o proceso) que accede a la base de datos. Los sistemas de gestión de bases de datos permiten crear, modificar y eliminar usuarios, asignando credenciales de acceso (como nombre de usuario y contraseña).

Ejemplo de creación de usuario en SQL (sintaxis puede variar según el sistema):

```
1 CREATE USER 'usuario_estudiantes' IDENTIFIED BY '↪
    contraseña_segura';
```

4.11.2. Privilegios

Los **privilegios** son permisos que controlan las acciones que los usuarios pueden realizar sobre los objetos de la base de datos (tablas, vistas, procedimientos, etc.). Los principales privilegios incluyen:

- `SELECT`: Permite consultar datos.
- `INSERT`: Permite agregar nuevos registros.
- `UPDATE`: Permite modificar registros existentes.
- `DELETE`: Permite eliminar registros.
- `CREATE`, `ALTER`, `DROP`: Permiten crear, modificar y eliminar objetos de la base de datos.

- `GRANT`, `REVOKE`: Permiten otorgar o revocar privilegios a otros usuarios.

Ejemplo de otorgar privilegios a un usuario:

```
GRANT SELECT, INSERT, UPDATE ON EstudiantesDB.* TO 'usuario_estudiantes';
```

Para revocar privilegios:

```
REVOKE UPDATE ON EstudiantesDB.cursos FROM 'usuario_estudiantes';
```

4.11.3. Gestión de roles

En sistemas avanzados, se pueden definir **roles**, que agrupan varios privilegios y pueden ser asignados a uno o más usuarios, facilitando la administración de permisos.

```
CREATE ROLE gestor_estudiantes;
GRANT SELECT, INSERT, UPDATE ON EstudiantesDB.* TO gestor_estudiantes;
GRANT gestor_estudiantes TO 'usuario_estudiantes';
```

Importancia

La correcta gestión de usuarios y privilegios es esencial para proteger los datos, evitar accesos no autorizados y cumplir con políticas de seguridad y normativas legales. Permite definir quién puede ver, modificar o administrar la información, reduciendo el riesgo de errores o acciones malintencionadas.

Ejercicio 4.8

Crea un usuario para el garaje y asígnale privilegios para consultar y modificar los datos de los vehículos, pero no para eliminar registros. Explica cómo estos privilegios contribuyen a la seguridad de la base de datos.

Resumen

En este capítulo se han presentado los conceptos fundamentales del modelo relacional, el lenguaje SQL y las principales operaciones para definir, manipular y proteger los datos en una base de datos. Se han explicado las diferencias entre los subconjuntos de SQL (DDL, DML, DCL, TCL), la importancia de las claves primarias y ajenas, los tipos de datos y las restricciones de validación que garantizan la integridad de la información.

También se ha abordado la gestión de la estructura de las tablas mediante la instrucción `ALTER TABLE`, la eliminación de tablas con `DROP TABLE`, el uso de índices para optimizar el rendimiento de las consultas y la creación de vistas para simplificar el acceso y mejorar la seguridad. Finalmente, se ha destacado la relevancia de la gestión de usuarios y privilegios para controlar el acceso y proteger los datos almacenados.

El dominio de estos conceptos y herramientas es esencial para diseñar, implementar y administrar bases de datos relacionales de manera eficiente, segura y escalable.

A continuación se presenta un resumen de las principales instrucciones DDL (Data Definition Language) y DCL (Data Control Language) vistas en este capítulo:

```sql
-- Ejemplo DDL: Crear una tabla de estudiantes
CREATE TABLE Estudiantes (
  ID INT PRIMARY KEY,
  Nombre VARCHAR(50) NOT NULL,
  FechaNacimiento DATE,
  Email VARCHAR(100) UNIQUE,
  Activo BOOLEAN DEFAULT TRUE
);

-- Ejemplo DDL: Modificar la tabla para añadir un nuevo atributo
ALTER TABLE Estudiantes ADD COLUMN Telefono VARCHAR(15);

-- Ejemplo DDL: Eliminar la tabla de estudiantes
DROP TABLE Estudiantes;

-- Ejemplo DCL: Otorgar privilegios de consulta e inserción sobre la tabla Estudiantes
GRANT SELECT, INSERT ON Estudiantes TO 'usuario_estudiantes';

-- Ejemplo DCL: Revocar el privilegio de actualización sobre la tabla Estudiantes
REVOKE UPDATE ON Estudiantes FROM 'usuario_estudiantes';
```

5

Realización de consultas

En este capítulo aprenderás cómo realizar consultas en bases de datos utilizando el lenguaje SQL. Las consultas son fundamentales para extraer, analizar y manipular la información almacenada, permitiendo responder preguntas y tomar decisiones basadas en los datos. Se presentarán los conceptos básicos de la sentencia `SELECT`, el uso de filtros, ordenación, operadores lógicos y de comparación, así como técnicas avanzadas como funciones de agregación, composiciones (`JOIN`), subconsultas y optimización de consultas. Al finalizar el capítulo, contarás con las herramientas necesarias para construir consultas eficientes y resolver problemas prácticos en el manejo de bases de datos relacionales.

5.1. La sentencia `SELECT`

La sentencia `SELECT` es la instrucción principal utilizada en SQL para consultar y recuperar datos almacenados en una base de datos. Permite especificar qué columnas y filas se desean obtener de una o varias tablas, así como aplicar filtros, ordenar resultados y realizar operaciones sobre los datos.

La sintaxis básica de una consulta `SELECT` es la siguiente:

```
1 SELECT columnas
2 FROM tabla;
```

Si se desea obtener todas las columnas de una tabla, se puede utilizar el asterisco (`*`) como comodín. Por ejemplo, para recuperar todos los datos de la tabla `Estudiantes`, se utiliza la siguiente consulta:

```
1 SELECT * FROM Estudiantes;
```

Esta instrucción devuelve todas las filas y columnas de la tabla `Estudiantes`.

5.2. Selección y ordenación de resultados

Para seleccionar filas específicas de una tabla, se utiliza la cláusula `WHERE`, que permite establecer condiciones sobre los valores de las columnas. Por ejemplo,

para obtener los estudiantes cuya edad sea mayor a 20 años:

```
SELECT * FROM Estudiantes
WHERE edad > 20;
```

Además, es posible ordenar los resultados utilizando la cláusula `ORDER BY`. Por ejemplo, para mostrar los estudiantes ordenados por apellido de forma ascendente:

```
SELECT * FROM Estudiantes
ORDER BY apellido ASC;
```

Si se desea ordenar de forma descendente, se utiliza `DESC`:

```
SELECT * FROM Estudiantes
ORDER BY apellido DESC;
```

También se pueden combinar ambas cláusulas para filtrar y ordenar simultáneamente:

```
SELECT nombre, apellido, edad FROM Estudiantes
WHERE edad >= 18
ORDER BY edad DESC;
```

De esta manera, se pueden obtener conjuntos de datos más específicos y organizados según las necesidades de la consulta.

5.3. Operadores lógicos y de comparación

Los operadores lógicos y de comparación permiten construir condiciones complejas en las consultas SQL, especialmente en la cláusula `WHERE`. A continuación se describen los principales operadores y su uso.

5.3.1. Operadores de comparación

Estos operadores se utilizan para comparar valores entre columnas y constantes:

Operador	Descripción y ejemplo
=	Igualdad. Ejemplo: `WHERE edad = 21`
<>/ !=	Diferente. Ejemplo: `WHERE nombre <>'Juan'`
>	Mayor que. Ejemplo: `WHERE promedio >8.5`
<	Menor que. Ejemplo: `WHERE edad <18`
>=	Mayor o igual que. Ejemplo: `WHERE edad >= 18`
<=	Menor o igual que. Ejemplo: `WHERE edad <= 25`

5.3.2. Operadores lógicos

Permiten combinar varias condiciones en una consulta:

Operador	Descripción y ejemplo
`AND`	Todas las condiciones deben cumplirse. Ejemplo: `WHERE edad >18 AND promedio >7`
`OR`	Al menos una condición debe cumplirse. Ejemplo: `WHERE ciudad = 'Lima' OR ciudad = 'Cusco'`
`NOT`	Niega una condición. Ejemplo: `WHERE NOT (estado = 'Inactivo')`

Las condiciones pueden agruparse usando paréntesis para definir el orden de evaluación:

```
SELECT * FROM Estudiantes
WHERE (edad > 18 AND promedio > 7) OR ciudad = 'Lima';
```

5.3.3. Operadores especiales

Existen operadores adicionales para realizar comparaciones más avanzadas:

Operador	Descripción y ejemplo
`BETWEEN`	Selecciona valores dentro de un rango. Ejemplo: `WHERE edad BETWEEN 18 AND 25`
`IN`	Verifica si un valor está en una lista. Ejemplo: `WHERE carrera IN ('Ingeniería', 'Medicina', 'Derecho')`
`LIKE`	Busca coincidencias de patrones en cadenas de texto. Ejemplo: `WHERE nombre LIKE 'A%'` (nombres que empiezan con 'A')
`IS NULL` / `IS NOT NULL`	Verifica valores nulos. Ejemplo: `WHERE telefono IS NULL`

Ejemplos prácticos

```
-- Estudiantes mayores de 20 años y con promedio mayor a 8
SELECT nombre, promedio FROM Estudiantes
WHERE edad > 20 AND promedio > 8;

-- Estudiantes cuyo apellido empieza con 'G'
SELECT * FROM Estudiantes
WHERE apellido LIKE 'G%';
```

```sql

-- Estudiantes que no tienen teléfono registrado
SELECT nombre FROM Estudiantes
WHERE telefono IS NULL;

-- Estudiantes de Ingeniería o Medicina
SELECT nombre FROM Estudiantes
WHERE carrera IN ('Ingeniería', 'Medicina');
```

El uso adecuado de estos operadores permite construir consultas precisas y eficientes, adaptadas a las necesidades de análisis de datos.

Ejercicio 5.1

Redacta una consulta SQL que muestre el nombre y la edad de los estudiantes que viven en la ciudad de 'Lima' y tienen un promedio mayor a 8. Ordena los resultados por edad de forma descendente.
Datos de ejemplo:

nombre	apellido	edad	ciudad	promedio
Ana	García	21	Madrid	9.2
Juan	Torres	19	Barcelona	7.8
María	Gómez	22	Madrid	8.5
Pedro	Ruiz	20	Bilbao	8.9
Lucía	Gutiérrez	23	Madrid	9.0

5.4. Funciones de agregación

Las funciones de agregación en SQL permiten realizar cálculos sobre un conjunto de filas y devolver un solo valor como resultado. Son especialmente útiles para obtener estadísticas, resúmenes y análisis de datos en las consultas. Las funciones de agregación más comunes son `COUNT`, `SUM`, `AVG`, `MIN` y `MAX`.

5.4.1. Función `COUNT`

La función `COUNT` se utiliza para contar el número de filas que cumplen una condición específica o el total de filas en una tabla.

```sql
-- Número total de estudiantes
SELECT COUNT(*) FROM Estudiantes;

-- Número de estudiantes con promedio mayor a 8
SELECT COUNT(*) FROM Estudiantes WHERE promedio > 8;
```

5.4.2. Función SUM

La función SUM calcula la suma total de los valores de una columna numérica.

```sql
-- Suma de todos los promedios
SELECT SUM(promedio) FROM Estudiantes;

-- Suma de edades de estudiantes de Ingeniería
SELECT SUM(edad) FROM Estudiantes WHERE carrera = 'Ingeniería';
```

5.4.3. Función AVG

La función AVG devuelve el valor promedio de una columna numérica.

```sql
-- Promedio de edad de todos los estudiantes
SELECT AVG(edad) FROM Estudiantes;

-- Promedio de promedio de estudiantes de Medicina
SELECT AVG(promedio) FROM Estudiantes WHERE carrera = 'Medicina'
  ;
```

5.4.4. Función MIN y MAX

Las funciones MIN y MAX permiten obtener el valor mínimo y máximo de una columna, respectivamente.

```sql
-- Edad mínima y máxima de los estudiantes
SELECT MIN(edad) AS EdadMinima, MAX(edad) AS EdadMaxima FROM
  Estudiantes;

-- Promedio más alto entre los estudiantes
SELECT MAX(promedio) FROM Estudiantes;
```

5.4.5. Uso de GROUP BY

Para aplicar funciones de agregación por grupos, se utiliza la cláusula GROUP BY. Esta permite agrupar filas que tienen el mismo valor en una o varias columnas y calcular agregados para cada grupo.

```sql
-- Número de estudiantes por carrera
SELECT carrera, COUNT(*) AS TotalEstudiantes
FROM Estudiantes
GROUP BY carrera;

-- Promedio de edad por ciudad
SELECT ciudad, AVG(edad) AS PromedioEdad
FROM Estudiantes
GROUP BY ciudad;
```

5.4.6. Uso de HAVING

La cláusula HAVING se emplea junto con GROUP BY para establecer condiciones sobre los resultados agregados de cada grupo. A diferencia de WHERE, que filtra filas antes de la agrupación, HAVING filtra los grupos después de aplicar las funciones de agregación.

Por ejemplo, para mostrar únicamente las carreras que tienen más de 10 estudiantes:

```
SELECT carrera, COUNT(*) AS TotalEstudiantes
FROM Estudiantes
GROUP BY carrera
HAVING COUNT(*) > 10;
```

En este caso, primero se agrupan los estudiantes por carrera y luego se filtran los grupos cuyo conteo sea mayor a 10.

También es posible combinar condiciones en HAVING utilizando operadores lógicos y funciones agregadas:

```
SELECT ciudad, AVG(edad) AS PromedioEdad, COUNT(*) AS Total
FROM Estudiantes
GROUP BY ciudad
HAVING AVG(edad) > 22 AND COUNT(*) >= 5;
```

Así, se obtienen las ciudades donde el promedio de edad supera los 22 años y hay al menos 5 estudiantes registrados.

Importante

Es fundamental comprender cómo funcionan las funciones de agregación y las cláusulas GROUP BY y HAVING para realizar análisis de datos efectivos en SQL.
Siempre que se utilicen funciones de agregación, las columnas que no forman parte de una función deben incluirse en la cláusula GROUP BY.

Ejercicio 5.2

Redacta una consulta SQL que muestre la ciudad y el promedio de edad de los estudiantes en cada ciudad, pero solo para aquellas ciudades donde el promedio de edad sea mayor a 21 años y haya al menos 2 estudiantes registrados.

5.5. Unión de consultas

La unión de consultas en SQL permite combinar los resultados de dos o más sentencias SELECT en un solo conjunto de resultados. Esto es útil cuando se desea

obtener datos de diferentes tablas o consultas que tienen una estructura similar.

5.5.1. Operador `UNION`

El operador `UNION` se utiliza para unir los resultados de dos o más consultas `SELECT`. Las consultas deben tener el mismo número de columnas y tipos de datos compatibles en cada columna correspondiente.

```
SELECT nombre, ciudad FROM Estudiantes
WHERE ciudad = 'Madrid'
UNION
SELECT nombre, ciudad FROM Estudiantes
WHERE ciudad = 'Barcelona';
```

Esta consulta devuelve una lista de nombres y ciudades de estudiantes que viven en Madrid, eliminando duplicados.

5.5.2. Eliminación de duplicados y uso de `UNION ALL`

Por defecto, `UNION` elimina las filas duplicadas en el resultado combinado. Si se desea incluir todos los resultados, incluso los duplicados, se utiliza `UNION ALL`:

```
SELECT nombre, ciudad FROM Estudiantes
WHERE ciudad = 'Madrid'
UNION ALL
SELECT nombre, ciudad FROM Estudiantes
WHERE ciudad = 'Barcelona';
```

En este caso, si una persona aparece en ambas consultas, se mostrará dos veces en el resultado.

5.5.3. Requisitos para la unión de consultas

Para que la unión funcione correctamente, se deben cumplir los siguientes requisitos:

- Cada consulta debe tener el mismo número de columnas en el `SELECT`.
- Los tipos de datos de las columnas correspondientes deben ser compatibles.
- Los nombres de las columnas en el resultado final se toman de la primera consulta.

5.5.4. Ordenación de resultados en la unión

Se puede ordenar el resultado combinado utilizando la cláusula `ORDER BY` al final de la unión. Por ejemplo:

```
SELECT nombre, ciudad FROM Estudiantes
WHERE ciudad = 'Madrid'
UNION
SELECT nombre, ciudad FROM Estudiantes
WHERE ciudad = 'Barcelona'
ORDER BY nombre ASC;
```

5.5.5. Unión de consultas con diferentes tablas

También es posible unir resultados de diferentes tablas, siempre que las columnas seleccionadas sean compatibles. Por ejemplo, si existe una tabla `Profesores` con columnas `nombre` y `ciudad`:

```
SELECT nombre, ciudad FROM Estudiantes
UNION
SELECT nombre, ciudad FROM Profesores;
```

Esto permite obtener una lista combinada de nombres y ciudades de estudiantes y profesores.

Ejemplo práctico

Supongamos que se desea obtener una lista de todas las personas (estudiantes y profesores) que viven en Madrid:

```
SELECT nombre FROM Estudiantes WHERE ciudad = 'Madrid'
UNION
SELECT nombre FROM Profesores WHERE ciudad = 'Madrid';
```

El resultado incluirá los nombres de estudiantes y profesores residentes en Madrid, sin duplicados.

Consejo

Es recomendable utilizar alias para las columnas en las consultas SQL, especialmente cuando se emplean funciones de agregación, subconsultas o cuando se desea mostrar nombres más descriptivos en los resultados. Los alias se definen con la palabra clave `AS` y permiten renombrar temporalmente una columna en el resultado de la consulta.

Por ejemplo:

```
SELECT AVG(edad) AS PromedioEdad
FROM Estudiantes;
```

En este caso, la columna resultante se llamará `PromedioEdad` en vez de `AVG(edad)`. Los alias también son útiles para mejorar la legibilidad de los resultados y facilitar el trabajo con aplicaciones que procesan los datos obtenidos de la base de datos.
Podrás usar el alias `AS` para renombrar columnas en otras partes de tus consultas, como en la cláusula `ORDER BY`, en subconsultas, o cuando quieras unir dos tablas cuyas columnas tengan nombres distintos.

Ejercicio 5.3

Redacta una consulta SQL que muestre los nombres y ciudades de todos los estudiantes y profesores que viven en Madrid o Barcelona, incluyendo duplicados si existen.

5.6. Composiciones internas y externas

Las composiciones (o combinaciones) internas y externas en SQL permiten relacionar datos de dos o más tablas mediante la cláusula `JOIN`. Estas operaciones son fundamentales para consultar información distribuida en diferentes tablas, aprovechando las relaciones entre ellas.

5.6.1. Composición interna (`INNER JOIN`)

La composición interna, conocida como `INNER JOIN`, devuelve únicamente las filas que tienen coincidencias en ambas tablas según la condición especificada. Es el tipo de unión más común y se utiliza para obtener datos relacionados.

Por ejemplo, supongamos que tenemos dos tablas: `Estudiantes` y `Carreras`.

Estudiantes	Carreras
id	id
nombre	nombre
carrera_id	

Para obtener el nombre del estudiante y el nombre de su carrera:

```
1 SELECT e.nombre, c.nombre AS carrera
2 FROM Estudiantes e
3 INNER JOIN Carreras c ON e.carrera_id = c.id;
```

En este ejemplo, solo se mostrarán los estudiantes que tienen una carrera asignada y que existe en la tabla `Carreras`.

Es posible realizar esta unión mediante el uso de más de dos tablas:

```
SELECT e.nombre, e.ciudad, c.nombre AS curso, m.nombre AS
  materia
FROM Estudiantes e
INNER JOIN Carreras c ON e.carrera_id = c.id
INNER JOIN Materias m ON m.carrera_id = c.id;
```

5.6.2. Composición externa (`LEFT JOIN`, `RIGHT JOIN`, `FULL OUTER JOIN`)

Las composiciones externas permiten incluir filas que no tienen coincidencias en una de las tablas, mostrando valores nulos en las columnas de la tabla que no tiene correspondencia.

`LEFT JOIN` (Composición externa izquierda)

Devuelve todas las filas de la tabla de la izquierda y las coincidencias de la tabla de la derecha. Si no hay coincidencia, las columnas de la tabla derecha tendrán valores nulos.

```
SELECT e.nombre, c.nombre AS carrera
FROM Estudiantes e
LEFT JOIN Carreras c ON e.carrera_id = c.id;
```

Aquí se mostrarán todos los estudiantes, incluso aquellos que no tienen una carrera asignada.

`RIGHT JOIN` (Composición externa derecha)

Devuelve todas las filas de la tabla de la derecha y las coincidencias de la tabla de la izquierda. Si no hay coincidencia, las columnas de la tabla izquierda tendrán valores nulos.

```
SELECT e.nombre, c.nombre AS carrera
FROM Estudiantes e
RIGHT JOIN Carreras c ON e.carrera_id = c.id;
```

Se mostrarán todas las carreras, incluso aquellas que no tienen estudiantes asignados.

`FULL OUTER JOIN` (Composición externa completa)

Devuelve todas las filas de ambas tablas, combinando las coincidencias y mostrando valores nulos donde no haya correspondencia.

```
SELECT e.nombre, c.nombre AS carrera
FROM Estudiantes e
FULL OUTER JOIN Carreras c ON e.carrera_id = c.id;
```

Este tipo de unión muestra todos los estudiantes y todas las carreras, incluyendo los casos donde no hay coincidencia en la otra tabla.

5.6.3. Composición cruzada (`CROSS JOIN`)

La composición cruzada, o `CROSS JOIN`, genera el producto cartesiano de ambas tablas, es decir, todas las combinaciones posibles de filas entre las dos tablas.

```
SELECT e.nombre, c.nombre AS carrera
FROM Estudiantes e
CROSS JOIN Carreras c;
```

Se recomienda usar `CROSS JOIN` con precaución, ya que puede generar grandes volúmenes de datos.

5.6.4. Uniones con múltiples condiciones

Las composiciones pueden incluir varias condiciones en la cláusula `ON`, utilizando operadores lógicos para definir relaciones más complejas.

```
SELECT e.nombre, c.nombre AS carrera
FROM Estudiantes e
INNER JOIN Carreras c
ON e.carrera_id = c.id AND c.activa = 1;
```

Importante

Es recomendable utilizar alias para las tablas, especialmente en consultas con múltiples `JOIN`, para mejorar la legibilidad y evitar ambigüedades.

Ejemplo práctico

Supongamos que queremos obtener una lista de estudiantes junto con el nombre de su carrera y la ciudad donde estudian, mostrando también los estudiantes sin carrera asignada:

```
SELECT e.nombre, c.nombre AS carrera, e.ciudad
FROM Estudiantes e
LEFT JOIN Carreras c ON e.carrera_id = c.id
ORDER BY e.nombre;
```

Resumen de tipos de composiciones

Tipo de composición	Descripción
INNER JOIN	Solo filas con coincidencia en ambas tablas.
LEFT JOIN	Todas las filas de la tabla izquierda y coincidencias de la derecha.
RIGHT JOIN	Todas las filas de la tabla derecha y coincidencias de la izquierda.
FULL OUTER JOIN	Todas las filas de ambas tablas, con coincidencias y valores nulos donde no las haya.
CROSS JOIN	Producto cartesiano de ambas tablas.

Ejercicio 5.4

Usando la base de datos de ejemplo que acompaña a este libro, crea las siguientes consultas:

1. Lista de todos los estudiantes junto con el nombre de su curso, incluyendo solo aquellos que están inscritos en algún curso.
2. Lista de todos los profesores, asociado con qué cursos están impartiendo, así como el nombre de cada alumno. Incluye aquellos profesores que no estén impartiendo ningún curso.

5.7. Subconsultas

Las subconsultas, también conocidas como consultas anidadas, son sentencias `SELECT` que se incluyen dentro de otra consulta SQL. Permiten realizar operaciones complejas, como filtrar resultados en función de valores calculados dinámicamente, comparar con agregados, o construir listas de valores para operadores como `IN`. Las subconsultas pueden aparecer en las cláusulas `WHERE`, `FROM`, `SELECT` y `HAVING`.

5.7.1. Subconsultas en la cláusula `WHERE`

La forma más común de utilizar subconsultas es en la cláusula `WHERE`, para comparar valores de una columna con el resultado de otra consulta. Por ejemplo, para obtener los estudiantes cuyo promedio es mayor al promedio general de todos los estudiantes:

```
1 SELECT nombre, promedio
```

```
FROM Estudiantes
WHERE promedio > (
  SELECT AVG(promedio) FROM Estudiantes
);
```

En este ejemplo, la subconsulta calcula el promedio general y la consulta principal selecciona los estudiantes que superan ese valor.

5.7.2. Subconsultas con operadores `IN`, `ANY`, `ALL`

Las subconsultas pueden devolver listas de valores, que se utilizan con operadores como `IN` para comparar si un valor pertenece al conjunto devuelto:

```
SELECT nombre
FROM Estudiantes
WHERE carrera_id IN (
  SELECT id FROM Carreras WHERE activa = 1
);
```

También pueden emplearse los operadores `ANY` y `ALL` para comparar con todos o alguno de los valores devueltos por la subconsulta:

```
SELECT nombre, promedio
FROM Estudiantes
WHERE promedio > ALL (
  SELECT promedio FROM Estudiantes WHERE ciudad = 'Madrid'
);
```

Esta consulta selecciona estudiantes cuyo promedio es mayor que el de cualquier estudiante de Madrid.

5.7.3. Subconsultas en la cláusula `FROM` (subconsultas derivadas)

Las subconsultas pueden actuar como tablas temporales en la cláusula `FROM`, permitiendo realizar operaciones adicionales sobre sus resultados:

```
SELECT ciudad, PromedioEdad
FROM (
  SELECT ciudad, AVG(edad) AS PromedioEdad
  FROM Estudiantes
  GROUP BY ciudad
) AS Subconsulta
WHERE PromedioEdad > 21;
```

Aquí, la subconsulta calcula el promedio de edad por ciudad y la consulta principal filtra las ciudades con promedio mayor a 21.

5.7.4. Subconsultas en la cláusula SELECT

Es posible incluir subconsultas en la lista de columnas seleccionadas, para calcular valores adicionales por cada fila:

```
SELECT nombre,
  (SELECT nombre FROM Carreras WHERE id = e.carrera_id) AS ↪
  carrera
FROM Estudiantes e;
```

En este caso, se muestra el nombre del estudiante y el nombre de su carrera, obtenida mediante una subconsulta.

5.7.5. Subconsultas correlacionadas

Las subconsultas correlacionadas hacen referencia a columnas de la consulta principal. Se evalúan para cada fila de la consulta externa, permitiendo comparaciones dinámicas:

```
SELECT nombre, edad
FROM Estudiantes e
WHERE edad > (
  SELECT AVG(edad)
  FROM Estudiantes
  WHERE ciudad = e.ciudad
);
```

Esta consulta selecciona estudiantes cuya edad es mayor al promedio de edad de su propia ciudad.

5.7.6. Limitaciones y consideraciones

- Las subconsultas pueden afectar el rendimiento si no se optimizan adecuadamente, especialmente las correlacionadas.
- No todas las subconsultas pueden devolver múltiples columnas; la mayoría de los operadores esperan un solo valor o una lista de valores de una sola columna.
- Es recomendable utilizar alias descriptivos para las subconsultas en la cláusula FROM.
- En algunos sistemas de bases de datos, existen restricciones sobre el uso de subconsultas en ciertas cláusulas.

Ejemplo práctico

Supongamos que queremos mostrar el nombre de los estudiantes que tienen el promedio más alto de su ciudad:

```
SELECT nombre, ciudad, promedio
FROM Estudiantes e
WHERE promedio = (
  SELECT MAX(promedio)
  FROM Estudiantes
  WHERE ciudad = e.ciudad
);
```

Ejercicio 5.5

Redacta una consulta SQL que muestre el nombre y la ciudad de los estudiantes cuyo promedio es mayor que la media del promedio de todos los estudiantes de su ciudad.

Las subconsultas son herramientas poderosas para construir consultas avanzadas y obtener información precisa a partir de datos relacionados.

5.8. Combinación de consultas

La combinación de consultas en SQL permite unir, comparar y manipular los resultados de varias consultas para obtener información más compleja y útil. Además de la unión de resultados con `UNION`, existen otras técnicas y operadores que permiten combinar datos de distintas formas, como las composiciones (`JOIN`), las subconsultas y las operaciones de conjuntos.

5.8.1. Operaciones de conjuntos

SQL proporciona operadores para realizar operaciones de conjuntos sobre los resultados de dos consultas. Los principales operadores son `UNION`, `INTERSECT` y `EXCEPT` (o `MINUS` en algunos sistemas).

`UNION`

Ya explicado anteriormente, `UNION` combina los resultados de dos consultas, eliminando duplicados.

`UNION ALL`

Incluye todos los resultados, incluso duplicados.

INTERSECT

Devuelve solo las filas que aparecen en ambas consultas.

```
SELECT nombre FROM Estudiantes
INTERSECT
SELECT nombre FROM Profesores;
```

Esta consulta muestra los nombres que están tanto en la tabla `Estudiantes` como en `Profesores`.

EXCEPT / MINUS

Devuelve las filas de la primera consulta que no aparecen en la segunda.

```
SELECT nombre FROM Estudiantes
EXCEPT
SELECT nombre FROM Profesores;
```

En algunos sistemas, se utiliza `MINUS` en lugar de `EXCEPT`.

5.8.2. Requisitos para operaciones de conjuntos

Para utilizar estos operadores, ambas consultas deben tener:

- El mismo número de columnas.
- Tipos de datos compatibles en cada columna correspondiente.

Los nombres de las columnas en el resultado final se toman de la primera consulta.

5.8.3. Combinación de resultados con `JOIN` y subconsultas

Las composiciones (`JOIN`) permiten combinar filas de dos o más tablas relacionadas. Las subconsultas pueden usarse para filtrar, calcular o generar conjuntos de datos que luego se combinan con otras consultas.

Por ejemplo, para obtener los estudiantes que están inscritos en cursos impartidos por un profesor específico:

```
SELECT e.nombre, c.nombre AS curso
FROM Estudiantes e
INNER JOIN Inscripciones i ON e.id = i.estudiante_id
INNER JOIN Cursos c ON i.curso_id = c.id
WHERE c.profesor_id = (
  SELECT id FROM Profesores WHERE nombre = 'Juan'
);
```

5.8.4. Combinación de agregados y composiciones

Es posible combinar funciones de agregación con composiciones para obtener información resumida de varias tablas. Por ejemplo, para mostrar el número de estudiantes por curso:

```
SELECT c.nombre AS curso, COUNT(i.estudiante_id) AS
  TotalEstudiantes
FROM Cursos c
LEFT JOIN Inscripciones i ON c.id = i.curso_id
GROUP BY c.nombre;
```

5.8.5. Combinación de consultas con alias y subconsultas derivadas

Las subconsultas derivadas permiten crear tablas temporales que pueden combinarse con otras consultas. Por ejemplo, para obtener el promedio de edad por curso:

```
SELECT c.nombre AS curso, sub.PromedioEdad
FROM Cursos c
LEFT JOIN (
  SELECT curso_id, AVG(edad) AS PromedioEdad
  FROM Estudiantes e
  INNER JOIN Inscripciones i ON e.id = i.estudiante_id
  GROUP BY curso_id
) sub ON c.id = sub.curso_id;
```

Ejemplo avanzado: combinación de múltiples técnicas

Supongamos que queremos mostrar los nombres de los estudiantes que están inscritos en cursos activos y cuyo promedio es superior al promedio general de todos los estudiantes:

```
SELECT e.nombre, c.nombre AS curso
FROM Estudiantes e
INNER JOIN Inscripciones i ON e.id = i.estudiante_id
INNER JOIN Cursos c ON i.curso_id = c.id
WHERE c.activo = 1
AND e.promedio > (
  SELECT AVG(promedio) FROM Estudiantes
);
```

Ejercicio 5.6

Redacta una consulta SQL que muestre los nombres de los estudiantes que están inscritos en cursos impartidos por profesores que tienen más de 10 años de experiencia. Incluye el nombre del curso y el nombre del profesor.

5.9. Optimización de consultas

La optimización de consultas es el proceso de mejorar el rendimiento de las sentencias SQL para reducir el tiempo de ejecución y el consumo de recursos. Una consulta optimizada permite obtener resultados más rápidamente, especialmente cuando se trabaja con grandes volúmenes de datos o bases de datos complejas.

5.9.1. Uso de índices

Los índices son estructuras que aceleran la búsqueda y recuperación de datos en las tablas. Al crear índices sobre columnas utilizadas frecuentemente en cláusulas `WHERE`, `ORDER BY` o `JOIN`, el motor de la base de datos puede localizar filas de manera más eficiente.

- Utiliza índices en columnas que se emplean para filtrar (`WHERE`), ordenar (`ORDER BY`) o unir tablas (`JOIN`).
- Evita crear índices innecesarios, ya que pueden ralentizar las operaciones de inserción, actualización y eliminación.
- Prefiere índices compuestos cuando se filtra por varias columnas simultáneamente.

Ejemplo de creación de índice:

```
1 CREATE INDEX idx_estudiantes_ciudad ON Estudiantes(ciudad);
```

5.9.2. Selección de columnas necesarias

Evita seleccionar todas las columnas con `SELECT *`. Especifica únicamente las columnas requeridas para reducir el volumen de datos transferidos y procesados.

```
1 -- Ineficiente
2 SELECT * FROM Estudiantes;
3
4 -- Optimizado
5 SELECT nombre, promedio FROM Estudiantes WHERE promedio > 8;
```

5.9.3. Filtrado temprano de datos

Aplica filtros lo antes posible en la consulta, utilizando la cláusula `WHERE` para limitar el número de filas procesadas por el motor de la base de datos.

```
SELECT nombre FROM Estudiantes WHERE ciudad = 'Madrid' AND ↪
  promedio > 8;
```

5.9.4. Uso eficiente de funciones de agregación

Cuando utilices funciones de agregación (`COUNT`, `SUM`, `AVG`, etc.), asegúrate de agrupar solo lo necesario y filtrar los datos antes de la agregación para evitar cálculos innecesarios.

```
SELECT ciudad, AVG(promedio) FROM Estudiantes
WHERE promedio > 7
GROUP BY ciudad;
```

5.9.5. Optimización de composiciones (`JOIN`)

Las composiciones pueden afectar el rendimiento si involucran grandes tablas o múltiples condiciones. Para optimizarlas:

- Usa índices en las columnas de unión.
- Prefiere `INNER JOIN` cuando sea posible, ya que procesa menos filas que `OUTER JOIN`.
- Filtra las filas antes de unir tablas, si es posible.
- Evita composiciones cruzadas (`CROSS JOIN`) salvo que sean estrictamente necesarias.

5.9.6. Evitar subconsultas innecesarias

Las subconsultas, especialmente las correlacionadas, pueden ser costosas. Considera reemplazarlas por composiciones (`JOIN`) cuando sea posible.

```
-- Subconsulta correlacionada (menos eficiente)
SELECT nombre FROM Estudiantes
WHERE promedio > (SELECT AVG(promedio) FROM Estudiantes);

-- Composición con agregación (más eficiente)
WITH PromedioGeneral AS (
  SELECT AVG(promedio) AS promedio FROM Estudiantes
)
```

```sql
SELECT e.nombre
FROM Estudiantes e, PromedioGeneral pg
WHERE e.promedio > pg.promedio;
```

5.9.7. Uso de EXPLAIN y planes de ejecución

La mayoría de los sistemas de bases de datos ofrecen la instrucción EXPLAIN para analizar cómo se ejecutará una consulta. Utiliza esta herramienta para identificar cuellos de botella y mejorar la consulta.

```sql
EXPLAIN SELECT nombre FROM Estudiantes WHERE ciudad = 'Madrid';
```

El plan de ejecución muestra si se utilizan índices, el orden de las operaciones y el costo estimado.

5.9.8. Limitación de resultados

Cuando solo necesitas una parte de los resultados, utiliza la cláusula LIMIT (o TOP en algunos sistemas) para reducir el número de filas devueltas.

```sql
SELECT nombre FROM Estudiantes WHERE promedio > 8 LIMIT 10;
```

5.9.9. Evitar funciones en columnas indexadas

Evita aplicar funciones a columnas indexadas en la cláusula WHERE, ya que esto puede impedir el uso del índice.

```sql
-- No se usa el índice
SELECT * FROM Estudiantes WHERE UPPER(nombre) = 'JUAN';

-- Se usa el índice
SELECT * FROM Estudiantes WHERE nombre = 'Juan';
```

5.9.10. Desnormalización y tablas resumen

En sistemas con grandes volúmenes de datos y consultas frecuentes de agregados, considera crear tablas resumen o desnormalizadas para acelerar el acceso a la información.

- Las tablas resumen almacenan resultados pre-calculados de agregaciones.
- La desnormalización consiste en duplicar datos para reducir la necesidad de composiciones complejas.

5.9.11. Mantenimiento y actualización de estadísticas

Las bases de datos mantienen estadísticas internas para optimizar las consultas. Es importante actualizar estas estadísticas periódicamente, especialmente después de grandes cambios en los datos.

```
-- Ejemplo en PostgreSQL
ANALYZE Estudiantes;
```

5.9.12. Revisión periódica de consultas

Revisa y ajusta las consultas conforme crece la base de datos y cambian los patrones de uso. El rendimiento puede variar con el tiempo y requerir nuevas optimizaciones.

Importante

La optimización de consultas es un proceso iterativo. Utiliza herramientas de monitoreo, analiza los planes de ejecución y realiza pruebas de rendimiento para identificar oportunidades de mejora.

Resumen

En este capítulo se han presentado los conceptos fundamentales para la realización de consultas en SQL. Se ha explicado la sintaxis básica de la sentencia `SELECT`, el uso de cláusulas para filtrar y ordenar resultados, y la aplicación de operadores lógicos y de comparación. Se abordaron las funciones de agregación y la agrupación de datos con `GROUP BY` y `HAVING`, así como la unión de consultas mediante `UNION`, `INTERSECT` y `EXCEPT`. Se detallaron los distintos tipos de composiciones (`JOIN`) para combinar información de varias tablas y el uso de subconsultas para construir consultas avanzadas. Finalmente, se ofrecieron recomendaciones para la optimización de consultas, con el objetivo de mejorar el rendimiento y la eficiencia en el manejo de bases de datos.

Dominar estas técnicas es esencial para extraer, analizar y manipular datos de manera efectiva, permitiendo resolver problemas complejos y obtener información valiosa a partir de los datos almacenados.

A continuación se muestra un resumen de las principales instrucciones vistas en este capítulo:

```
-- Seleccionar nombre y promedio de estudiantes con promedio mayor a 8
SELECT nombre, promedio
FROM Estudiantes
```

```
4  WHERE promedio > 8;
5
6  -- Contar el número de estudiantes por ciudad
7  SELECT ciudad, COUNT(*) AS TotalEstudiantes
8  FROM Estudiantes
9  GROUP BY ciudad;
10
11 -- Obtener nombres y ciudades de estudiantes y profesores que viven en Madrid
12 SELECT nombre, ciudad FROM Estudiantes WHERE ciudad = 'Madrid'
13 UNION
14 SELECT nombre, ciudad FROM Profesores WHERE ciudad = 'Madrid';
15
16 -- Listar estudiantes junto con el nombre de su carrera usando INNER JOIN
17 SELECT e.nombre, c.nombre AS carrera
18 FROM Estudiantes e
19 INNER JOIN Carreras c ON e.carrera_id = c.id;
20
21 -- Ejemplo de subconsulta: estudiantes con promedio superior al promedio general
22 SELECT nombre, promedio
23 FROM Estudiantes
24 WHERE promedio > (
25   SELECT AVG(promedio) FROM Estudiantes
26 );
27
28 -- Ejemplo de LEFT JOIN: mostrar todos los estudiantes y su carrera (incluyendo los que no tienen carrera)
29 SELECT e.nombre, c.nombre AS carrera
30 FROM Estudiantes e
31 LEFT JOIN Carreras c ON e.carrera_id = c.id;
32
33 -- Ejemplo de función de agregación con HAVING: ciudades con más de 5 estudiantes
34 SELECT ciudad, COUNT(*) AS TotalEstudiantes
35 FROM Estudiantes
36 GROUP BY ciudad
37 HAVING COUNT(*) > 5;
38
39 -- Ejemplo de uso de IN: estudiantes que estudian en ciudades específicas
40 SELECT nombre, ciudad
41 FROM Estudiantes
42 WHERE ciudad IN ('Madrid', 'Barcelona');
43
44 -- Ejemplo de combinación de JOIN y agregación: número de estudiantes por curso
45 SELECT c.nombre AS curso, COUNT(i.estudiante_id) AS
```

```
    TotalEstudiantes
46  FROM Cursos c
47  LEFT JOIN Inscripciones i ON c.id = i.curso_id
48  GROUP BY c.nombre;
49
50  -- Ejemplo de LIMIT: mostrar los 3 estudiantes con mayor promedio
51  SELECT nombre, promedio
52  FROM Estudiantes
53  ORDER BY promedio DESC
54  LIMIT 3;
```

6

Tratamiento de datos

En este capítulo se estudian las principales operaciones de edición de datos en bases de datos relacionales: cómo crear, modificar y eliminar registros, y cómo garantizar la coherencia de la información mediante restricciones y transacciones. Se explican las sentencias SQL fundamentales (`INSERT`, `DELETE`, `UPDATE`), el concepto de integridad referencial y el uso de subconsultas para manipular datos de forma eficiente. Además, se abordan las políticas de bloqueo y concurrencia, esenciales en entornos multiusuario, y se presentan ejemplos prácticos para ilustrar cada tema. El objetivo es proporcionar los conocimientos necesarios para gestionar datos de manera segura y eficaz en sistemas de bases de datos.

6.1. Creación de registros. La sentencia `INSERT`.

La creación de registros en una base de datos relacional se realiza mediante la sentencia `INSERT`. Esta operación permite añadir nuevas filas a una tabla, especificando los valores que se asignarán a cada columna. Es fundamental conocer la estructura de la tabla y las restricciones definidas para garantizar la correcta inserción de los datos. A continuación se presentan las distintas formas de utilizar `INSERT`, así como consideraciones importantes sobre valores por defecto, columnas autoincrementales y posibles errores.

6.1.1. Sintaxis básica de `INSERT`

La sentencia `INSERT` permite añadir nuevos registros a una tabla. La sintaxis general es:

```
1 INSERT INTO nombre_tabla (columna1, columna2, ...) VALUES (↪
    valor1, valor2, ...);
```

Si se omite la lista de columnas, se deben proporcionar valores para todas las columnas en el orden definido en la tabla:

```
1 INSERT INTO nombre_tabla VALUES (valor1, valor2, ...);
```

6.1.2. Inserción de múltiples registros

Es posible insertar varios registros en una sola sentencia:

```
INSERT INTO nombre_tabla (columna1, columna2) VALUES
  (valorA1, valorA2),
  (valorB1, valorB2),
  (valorC1, valorC2);
```

Esto mejora la eficiencia al reducir el número de transacciones.

6.1.3. Inserción de datos desde otra tabla

Se pueden insertar registros obtenidos de una consulta:

```
INSERT INTO nombre_tabla_destino (columna1, columna2)
SELECT columna1, columna2 FROM nombre_tabla_origen WHERE ↪
  condición;
```

Esta técnica es útil para migraciones o copias de datos.

6.1.4. Valores por defecto y columnas autoincrementales

Si una columna tiene un valor por defecto o es autoincremental, puede omitirse en la sentencia `INSERT`. El sistema asignará el valor automáticamente.

```
CREATE TABLE empleados (
  id INT AUTO_INCREMENT PRIMARY KEY,
  puesto VARCHAR(50) NOT NULL DEFAULT 'Empleado',
  nombre VARCHAR(100) NOT NULL,
  email VARCHAR(100) NOT NULL
);

INSERT INTO empleados (nombre, email) VALUES ('Ana', '↪
  ana@example.com');
```

Consejo

Cuando utilices columnas autogenerativas como `AUTO_INCREMENT` (o equivalentes), no incluyas explícitamente el valor de la columna en la sentencia `INSERT`. El sistema asignará automáticamente un identificador único, evitando conflictos y facilitando la gestión de claves primarias.
En PostgreSQL, podrás recuperar el valor generado mediante la sentencia `RETURNING`. Ejemplo:

```
INSERT INTO empleados (nombre, email) VALUES ('Ana', '↪
  ana@example.com') RETURNING id;
```

En MySQL, puedes utilizar la función `LAST_INSERT_ID()` para obtener el

último valor generado. Ejemplo:

```
INSERT INTO empleados (nombre, email) VALUES ('Ana', '↪
  ana@example.com');
SELECT LAST_INSERT_ID();
```

6.1.5. Restricciones y errores comunes

Al insertar datos, el sistema verifica restricciones como claves primarias, unicidad, tipos de datos y no nulos. Los errores más frecuentes incluyen:

- Violación de clave primaria (duplicados).
- Inserción de valores nulos en columnas que no lo permiten.
- Tipos de datos incompatibles.

6.1.6. Ejemplos prácticos

```
-- Insertar un registro completo
INSERT INTO productos (id, nombre, precio) VALUES (1, 'Teclado',↪
   25.99);

-- Insertar varios registros
INSERT INTO productos (id, nombre, precio) VALUES
  (2, 'Ratón', 15.50),
  (3, 'Monitor', 120.00);

-- Insertar usando una subconsulta
INSERT INTO ventas (producto_id, cantidad)
SELECT id, 10 FROM productos WHERE precio < 20;
```

Ejercicio 6.1

En la base de datos de ejemplo de este libro, crea 1 curso llamado Informática. Asígnale un profesor, e inscribe a todos los alumnos que estén inscritos en la materia de Programación.

6.2. Borrado de registros. La sentencia DELETE.

La eliminación de registros en una base de datos relacional se realiza mediante la sentencia DELETE. Esta operación permite borrar filas específicas de una tabla,

según los criterios definidos en una condición. Es fundamental comprender el alcance de la sentencia, las restricciones asociadas y las implicaciones sobre la integridad de los datos.

6.2.1. Sintaxis básica de DELETE

La forma más común de la sentencia DELETE es:

```
DELETE FROM nombre_tabla WHERE condición;
```

La cláusula WHERE especifica qué registros serán eliminados. Si se omite, se eliminarán todos los registros de la tabla:

```
DELETE FROM nombre_tabla;
```

Importante

Omitir la condición WHERE borra todos los datos de la tabla, lo que puede ser irreversible si no existe una copia de seguridad.

6.2.2. Eliminación condicional

La utilidad principal de DELETE radica en la eliminación selectiva de registros. Por ejemplo:

```
DELETE FROM alumnos WHERE curso_id = 3;
```

Este comando elimina todos los alumnos inscritos en el curso con id igual a 3.

6.2.3. Eliminación de múltiples registros

La condición WHERE puede seleccionar varios registros a la vez:

```
DELETE FROM productos WHERE precio < 10;
```

Aquí se eliminan todos los productos cuyo precio sea menor a 10.

6.2.4. Restricciones y errores comunes

Al ejecutar DELETE, el sistema verifica restricciones como claves foráneas y reglas de integridad referencial. Los errores más frecuentes incluyen:

- Violación de integridad referencial: intentar borrar un registro referenciado por otra tabla.
- Omisión de la condición WHERE, provocando el borrado total de la tabla.

- Restricciones de `ON DELETE RESTRICT` o `ON DELETE NO ACTION` en claves foráneas.

6.2.5. Eliminación en cascada

Si una tabla tiene claves foráneas con la opción `ON DELETE CASCADE`, al eliminar un registro, se eliminarán automáticamente los registros relacionados en otras tablas:

```
DELETE FROM cursos WHERE id = 5;
-- Si alumnos.curso_id tiene ON DELETE CASCADE, los alumnos del curso 5 también se eliminarán.
```

6.2.6. Eliminación usando subconsultas

Es posible eliminar registros basados en el resultado de una subconsulta:

```
DELETE FROM ventas WHERE producto_id IN (
  SELECT id FROM productos WHERE precio > 100
);
```

Esto elimina todas las ventas de productos cuyo precio supera los 100.

6.2.7. Consideraciones de rendimiento

La eliminación masiva de registros puede afectar el rendimiento y bloquear la tabla durante la operación. Es recomendable realizar borrados en lotes cuando se trata de grandes volúmenes de datos.

6.2.8. Ejemplos prácticos

```
-- Eliminar un registro específico
DELETE FROM empleados WHERE id = 10;

-- Eliminar todos los registros de una tabla
DELETE FROM logs;

-- Eliminar registros relacionados mediante subconsulta
DELETE FROM inscripciones WHERE alumno_id IN (
  SELECT id FROM alumnos WHERE baja = TRUE
);
```

Ejercicio 6.2

En la base de datos de ejemplo, elimina todos los cursos que no tienen alumnos inscritos. ¿Qué sucede si existen restricciones de integridad referencial?

6.3. Actualización de registros. La sentencia `UPDATE`.

La actualización de registros en una base de datos relacional se realiza mediante la sentencia `UPDATE`. Esta operación permite modificar los valores de una o varias columnas en filas específicas de una tabla, según los criterios definidos en una condición. Es esencial comprender la sintaxis, el alcance de la operación, las restricciones asociadas y las implicaciones sobre la integridad de los datos.

6.3.1. Sintaxis básica de `UPDATE`

La forma general de la sentencia `UPDATE` es:

```
UPDATE nombre_tabla
SET columna1 = valor1, columna2 = valor2, ...
WHERE condición;
```

La cláusula `SET` indica las columnas a modificar y sus nuevos valores. La cláusula `WHERE` especifica qué registros serán actualizados. Si se omite la condición, se actualizarán todos los registros de la tabla.

Importante

Omitir la condición `WHERE` en una sentencia `UPDATE` modificará todos los registros de la tabla, lo que puede provocar cambios masivos e irreversibles.

6.3.2. Actualización de múltiples columnas

Es posible modificar varias columnas en una sola sentencia:

```
UPDATE empleados
SET puesto = 'Gerente', email = 'gerente@example.com'
WHERE id = 5;
```

Esto actualiza el puesto y el correo electrónico del empleado con `id` igual a 5.

6.3.3. Actualización de múltiples registros

La condición `WHERE` puede seleccionar varios registros a la vez:

```
UPDATE productos
SET precio = precio * 1.10
WHERE categoria = 'Electrónica';
```

Aquí se incrementa el precio de todos los productos de la categoría `Electrónica` en un 10 %.

6.3.4. Actualización basada en subconsultas

Es posible asignar valores obtenidos de una subconsulta:

```
UPDATE empleados
SET salario = (
  SELECT AVG(salario) FROM empleados WHERE departamento_id = 2
)
WHERE departamento_id = 2;
```

Este ejemplo asigna el salario promedio del departamento 2 a todos sus empleados.

6.3.5. Restricciones y errores comunes

Al ejecutar `UPDATE`, el sistema verifica restricciones como claves primarias, unicidad, tipos de datos y no nulos. Los errores más frecuentes incluyen:

- Violación de clave única: intentar asignar un valor duplicado en una columna con restricción de unicidad.
- Tipos de datos incompatibles: asignar valores que no corresponden al tipo de la columna.
- Violación de restricciones de integridad referencial: modificar claves foráneas a valores inexistentes en la tabla referenciada.

6.3.6. Actualización condicional y uso de operadores

Se pueden utilizar operadores y funciones para modificar valores de forma dinámica:

```
UPDATE alumnos
SET promedio = promedio + 0.5
WHERE promedio < 7.0;
```

Este comando incrementa el promedio de los alumnos que tienen menos de 7.

6.3.7. Actualización en cascada

Si existen claves foráneas con la opción `ON UPDATE CASCADE`, al modificar el valor de la clave primaria en la tabla principal, el cambio se propagará automáticamente a las tablas relacionadas.

```
UPDATE cursos
SET id = 10
WHERE id = 5;
-- Si inscripciones.curso_id tiene ON UPDATE CASCADE, los registros relacionados se actualizan automáticamente.
```

6.3.8. Consideraciones de rendimiento

La actualización masiva de registros puede afectar el rendimiento y bloquear la tabla durante la operación. Es recomendable realizar actualizaciones en lotes cuando se trata de grandes volúmenes de datos y asegurarse de que existan índices adecuados para las columnas utilizadas en la condición.

6.3.9. Ejemplos prácticos

```
-- Actualizar un registro específico
UPDATE productos SET precio = 99.99 WHERE id = 3;

-- Actualizar varios registros
UPDATE empleados SET puesto = 'Supervisor' WHERE puesto = 'Empleado';

-- Actualizar usando una subconsulta
UPDATE ventas
SET cantidad = cantidad + 5
WHERE producto_id IN (
  SELECT id FROM productos WHERE categoria = 'Accesorios'
);
```

Ejercicio 6.3

En la base de datos de ejemplo, aumenta en un 20 % el salario de todos los profesores que imparten más de 3 cursos. ¿Cómo puedes asegurarte de que ningún salario exceda el límite máximo permitido por la empresa?

6.4. Integridad referencial

La integridad referencial es un principio fundamental en las bases de datos relacionales que garantiza la coherencia de las relaciones entre tablas. Se basa en el uso de claves primarias y foráneas para asegurar que los datos relacionados permanezcan sincronizados y válidos.

6.4.1. Concepto de integridad referencial

La integridad referencial asegura que los valores de una columna (o conjunto de columnas) que actúan como clave foránea en una tabla coincidan con valores existentes en la clave primaria de la tabla referenciada. Esto evita la existencia de referencias «rotas» o huérfanas.

Por ejemplo, si la tabla `inscripciones` tiene una columna `curso_id` que referencia la columna `id` de la tabla `cursos`, no se podrá insertar un valor en `curso_id` que no exista en `cursos.id`.

6.4.2. Definición de claves foráneas

Las claves foráneas se definen al crear o modificar una tabla. Ejemplo en SQL:

```
CREATE TABLE inscripciones (
  id INT PRIMARY KEY,
  alumno_id INT,
  curso_id INT,
  FOREIGN KEY (alumno_id) REFERENCES alumnos(id),
  FOREIGN KEY (curso_id) REFERENCES cursos(id)
);
```

También pueden añadirse posteriormente:

```
ALTER TABLE inscripciones
ADD CONSTRAINT fk_curso
FOREIGN KEY (curso_id) REFERENCES cursos(id);
```

6.4.3. Acciones sobre claves foráneas

Al definir una clave foránea, se pueden especificar acciones automáticas que ocurren cuando el registro referenciado se elimina o actualiza:

- `ON DELETE CASCADE`: Elimina automáticamente los registros relacionados.
- `ON DELETE SET NULL`: Asigna `NULL` a la clave foránea si el registro referenciado se elimina.

- `ON DELETE RESTRICT` o `NO ACTION`: Impide la eliminación si existen referencias.
- `ON UPDATE CASCADE`: Actualiza automáticamente los valores de la clave foránea si la clave primaria cambia.

Ejemplo:

```
CREATE TABLE inscripciones (
  id INT PRIMARY KEY,
  alumno_id INT,
  curso_id INT,
  FOREIGN KEY (curso_id) REFERENCES cursos(id)
    ON DELETE CASCADE
    ON UPDATE CASCADE
);
```

6.4.4. Errores y violaciones de integridad referencial

Las violaciones de integridad referencial ocurren cuando se intenta:

- Insertar un valor en la clave foránea que no existe en la tabla referenciada.
- Eliminar o modificar un registro referenciado sin respetar las reglas definidas.

El sistema de gestión de bases de datos (SGBD) impide estas operaciones y genera errores, protegiendo la coherencia de los datos.

6.4.5. Ejemplo práctico

Supongamos que se elimina un curso:

```
DELETE FROM cursos WHERE id = 5;
```

Si `inscripciones.curso_id` tiene `ON DELETE CASCADE`, todas las inscripciones al curso 5 se eliminarán automáticamente. Si tiene `ON DELETE RESTRICT`, la operación fallará si existen inscripciones asociadas.

6.4.6. Consideraciones de diseño

Al diseñar la integridad referencial, es importante:

- Analizar las relaciones entre entidades y definir las claves foráneas adecuadas.

- Elegir las acciones (`CASCADE`, `SET NULL`, `RESTRICT`) según las necesidades del negocio.
- Documentar las restricciones para facilitar el mantenimiento y la evolución del modelo de datos.

6.4.7. Comprobación y mantenimiento

Los SGBD permiten consultar las restricciones existentes y comprobar la integridad referencial mediante comandos de administración o vistas del sistema. Es recomendable revisar periódicamente la coherencia de los datos, especialmente tras operaciones masivas de edición.

Ejercicio 6.4

En la base de datos de ejemplo, modifica la relación entre `cursos` e `inscripciones` para que, al eliminar un curso, todas sus inscripciones se eliminen automáticamente. ¿Qué sucede si intentas eliminar un curso que tiene inscripciones sin definir la acción `ON DELETE CASCADE`?

6.5. Subconsultas y composición en órdenes de edición

Las subconsultas permiten realizar operaciones de edición (INSERT, UPDATE, DELETE) basadas en el resultado de otras consultas. Esta técnica es fundamental para manipular datos de forma dinámica y eficiente, especialmente cuando se requiere modificar registros en función de condiciones complejas o relaciones entre tablas.

6.5.1. Concepto de subconsulta

Una subconsulta es una consulta anidada dentro de otra sentencia SQL. Puede aparecer en las cláusulas `WHERE`, `FROM`, `SELECT` o directamente en las órdenes de edición. El resultado de la subconsulta se utiliza como criterio o fuente de datos para la operación principal.

Ejemplo básico en una cláusula `WHERE`:

```
1 DELETE FROM alumnos
2 WHERE id IN (SELECT alumno_id FROM inscripciones WHERE curso_id ↪
    = 5);
```

6.5.2. Subconsultas en `INSERT`

Las subconsultas en `INSERT` permiten copiar o transformar datos de una tabla a otra. Es posible seleccionar columnas específicas y aplicar filtros:

```
INSERT INTO historial_precios (producto_id, precio, fecha)
SELECT id, precio, CURRENT_DATE
FROM productos
WHERE precio > 100;
```

Esto inserta en `historial_precios` los productos cuyo precio supera 100, registrando la fecha actual.

6.5.3. Subconsultas en `UPDATE`

En una sentencia `UPDATE`, las subconsultas pueden asignar valores calculados o extraídos de otras tablas:

```
UPDATE empleados
SET salario = (
  SELECT MAX(salario) FROM empleados WHERE departamento_id = 3
)
WHERE departamento_id = 3;
```

Aquí, todos los empleados del departamento 3 reciben el salario máximo de ese departamento.

También es posible utilizar subconsultas en la condición:

```
UPDATE productos
SET descuento = 0.15
WHERE id IN (
  SELECT producto_id FROM ventas WHERE cantidad > 50
);
```

6.5.4. Subconsultas en `DELETE`

Las subconsultas en `DELETE` permiten eliminar registros en función de datos relacionados:

```
DELETE FROM inscripciones
WHERE curso_id IN (
  SELECT id FROM cursos WHERE fecha_fin < CURRENT_DATE
);
```

Esto elimina todas las inscripciones a cursos finalizados.

6.5.5. Subconsultas correlacionadas

Una subconsulta correlacionada depende de cada fila procesada por la consulta principal. Se utiliza para comparar o calcular valores dinámicamente:

```
UPDATE productos p
SET precio = precio * 0.95
WHERE EXISTS (
  SELECT 1 FROM ventas v
  WHERE v.producto_id = p.id AND v.cantidad > 100
);
```

Aquí, el precio de los productos con ventas superiores a 100 unidades se reduce en un 5 %.

6.5.6. Composición de órdenes de edición

La composición consiste en encadenar varias operaciones de edición, a menudo utilizando subconsultas para garantizar la coherencia y eficiencia. Ejemplo de inserción y actualización combinadas:

```
-- Insertar nuevos alumnos y actualizar su estado
INSERT INTO alumnos (nombre, email)
SELECT nombre, email FROM preinscripciones WHERE validado = TRUE
  ;

UPDATE preinscripciones
SET procesado = TRUE
WHERE validado = TRUE;
```

6.5.7. Consideraciones de rendimiento y buenas prácticas

El uso intensivo de subconsultas puede afectar el rendimiento, especialmente si las tablas son grandes o las subconsultas no están optimizadas. Es recomendable:

- Utilizar índices en las columnas involucradas en subconsultas.
- Preferir subconsultas que devuelvan pocos resultados o estén bien filtradas.
- Considerar el uso de `JOIN` en lugar de subconsultas cuando sea posible.
- Verificar el plan de ejecución para identificar cuellos de botella.

6.5.8. Ejemplos prácticos

```sql
-- Eliminar productos sin ventas
DELETE FROM productos
WHERE id NOT IN (SELECT producto_id FROM ventas);

-- Actualizar el estado de cursos con inscripciones activas
UPDATE cursos
SET estado = 'Activo'
WHERE id IN (
  SELECT curso_id FROM inscripciones WHERE fecha_baja IS NULL
);

-- Insertar registros en una tabla de auditoría tras una actualización
INSERT INTO auditoria (tabla, operacion, fecha)
SELECT 'empleados', 'UPDATE', CURRENT_TIMESTAMP
FROM empleados
WHERE salario > 5000;
```

Ejercicio 6.5

En la base de datos de ejemplo, elimina todos los cursos que no tengan alumnos inscritos.
Después, añade un campo a la tabla de cursos que registre la fecha de la última inscripción. Actualiza este campo con el valor de la última inscripción realizada para cada curso.

6.6. Transacciones

Las transacciones son un mecanismo fundamental en los sistemas de gestión de bases de datos (SGBD) para garantizar la integridad y coherencia de los datos ante operaciones complejas, concurrencia y posibles fallos. Una transacción agrupa varias operaciones en una unidad lógica de trabajo que se ejecuta de forma atómica: o todas las operaciones se completan correctamente, o ninguna tiene efecto.

6.6.1. Concepto de transacción

Una transacción es una secuencia de operaciones SQL (INSERT, UPDATE, DELETE, etc.) que se ejecutan como un bloque indivisible. El SGBD asegura que los datos permanezcan consistentes incluso si ocurre un error, una caída del sistema o una interferencia de otros usuarios.

6.6.2. Propiedades ACID

Las transacciones cumplen con las propiedades ACID, esenciales para la fiabilidad de las bases de datos:

Atomicidad Todas las operaciones de la transacción se completan o ninguna se realiza. Si ocurre un error, se revierte todo el bloque.

Consistencia La transacción lleva la base de datos de un estado válido a otro, respetando todas las reglas y restricciones.

Aislamiento Las operaciones de una transacción no son visibles para otras hasta que se confirma (commit), evitando interferencias.

Durabilidad Una vez confirmada, los cambios de la transacción son permanentes, incluso ante fallos del sistema.

6.6.3. Control de transacciones en SQL

Los SGBD proporcionan comandos para gestionar transacciones:

```
START TRANSACTION; -- o BEGIN;
-- Operaciones SQL
COMMIT; -- Confirma los cambios
ROLLBACK; -- Revierte todos los cambios realizados desde el inicio de la transacción
```

Ejemplo de uso:

```
START TRANSACTION;
UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;
UPDATE cuentas SET saldo = saldo + 100 WHERE id = 2;
COMMIT;
```

Si ocurre un error en alguna operación, se puede ejecutar `ROLLBACK` para deshacer todos los cambios.

6.6.4. Puntos de salvaguarda (`SAVEPOINT`)

Los puntos de salvaguarda permiten definir posiciones intermedias dentro de una transacción, facilitando la reversión parcial de operaciones:

```
START TRANSACTION;
UPDATE productos SET precio = precio * 1.10 WHERE categoria = 'Electrónica';
SAVEPOINT antes_descuento;
UPDATE productos SET descuento = 0.15 WHERE precio > 100;
ROLLBACK TO antes_descuento; -- Revierte solo los cambios posteriores al SAVEPOINT
COMMIT;
```

6.6.5. Ejemplo práctico de transacción

Supongamos que se desea inscribir a un alumno en un curso y registrar el pago correspondiente. Ambas operaciones deben realizarse juntas para evitar inconsistencias:

```
START TRANSACTION;
INSERT INTO inscripciones (alumno_id, curso_id, fecha) VALUES
  (5, 2, CURRENT_DATE);
INSERT INTO pagos (alumno_id, curso_id, monto, fecha) VALUES (5,
   2, 200, CURRENT_DATE);
COMMIT;
```

Si alguna inserción falla, se ejecuta `ROLLBACK` para evitar que solo una de las operaciones quede registrada.

6.6.6. Errores y manejo de excepciones

Durante una transacción pueden ocurrir errores por violación de restricciones, bloqueos o problemas de concurrencia. Es fundamental capturar estos errores y decidir si se debe confirmar (`COMMIT`) o revertir (`ROLLBACK`) la transacción.

En aplicaciones, el control de transacciones suele implementarse mediante bloques de manejo de excepciones (try/catch) en el lenguaje de programación utilizado.

6.6.7. Transacciones anidadas

Algunos SGBD permiten transacciones anidadas mediante `SAVEPOINT`, pero no todos soportan transacciones completas dentro de otras. Es importante conocer las capacidades del sistema utilizado.

6.6.8. Consideraciones de rendimiento

El uso excesivo de transacciones largas puede provocar bloqueos y afectar el rendimiento. Se recomienda:

- Mantener las transacciones lo más cortas posible.
- Evitar operaciones interactivas dentro de una transacción.
- Confirmar o revertir la transacción tan pronto como sea posible.

6.6.9. Ejercicio

Ejercicio 6.6

En la base de datos de ejemplo, dentro de una transacción, crea un curso, asigna un profesor y matricula a varios alumnos. Asegúrate de que si alguna de estas operaciones falla, ninguna de las demás se aplique.
Consejo: aprovecha el uso de valores por defecto y columnas autoincrementales para simplificar las inserciones.

6.7. Políticas de bloqueo y concurrencia

Las políticas de bloqueo y concurrencia son esenciales para garantizar la integridad y el rendimiento en sistemas multiusuario, donde varias transacciones pueden acceder y modificar los mismos datos simultáneamente. El control de concurrencia evita conflictos, inconsistencias y pérdidas de datos, asegurando que las operaciones se ejecuten de forma segura y eficiente.

6.7.1. Concepto de bloqueo

El bloqueo es un mecanismo mediante el cual el SGBD restringe el acceso a ciertos recursos (filas, páginas, tablas) mientras una transacción los modifica. Esto previene que otras transacciones lean o escriban datos que están siendo alterados, evitando problemas como lecturas sucias, actualizaciones perdidas o inconsistencias.

6.7.2. Tipos de bloqueo

Existen varios tipos de bloqueo, según el nivel de granularidad y el tipo de acceso:

Bloqueo de fila Restringe el acceso a una fila específica. Permite alta concurrencia, ya que otras filas pueden ser modificadas simultáneamente.

Bloqueo de página Afecta a un conjunto de filas almacenadas en una misma página física.

Bloqueo de tabla Restringe el acceso a toda la tabla. Es menos eficiente, pero puede ser necesario en operaciones masivas.

Según el tipo de operación, los bloqueos pueden ser:

Bloqueo compartido (S) Permite que varias transacciones lean el mismo recurso, pero impide modificaciones hasta que se libere el bloqueo.

Bloqueo exclusivo (X) Solo una transacción puede modificar el recurso, bloqueando tanto lecturas como escrituras por otras transacciones.

6.7.3. Niveles de aislamiento de transacciones

Read Uncommitted Permite leer datos no confirmados por otras transacciones (lecturas sucias).

Read Committed Solo permite leer datos confirmados, evitando lecturas sucias.

Repeatable Read Garantiza que los datos leídos por una transacción no cambien durante su ejecución, evitando lecturas no repetibles.

Serializable El nivel más estricto, simula la ejecución secuencial de transacciones, evitando todas las anomalías de concurrencia.

Ejemplo de configuración en SQL:

```sql
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;
-- Operaciones SQL
COMMIT;
```

6.7.4. Problemas de concurrencia

Sin un control adecuado, pueden surgir varios problemas:

Lectura sucia Una transacción lee datos modificados por otra que aún no ha sido confirmada.

Lectura no repetible Una transacción lee el mismo dato dos veces y obtiene valores diferentes porque otra transacción lo modificó entre lecturas.

Lectura fantasma Una transacción obtiene diferentes conjuntos de filas en consultas repetidas porque otras transacciones insertan o eliminan registros.

Actualización perdida Dos transacciones modifican el mismo dato y una sobrescribe el cambio de la otra.

6.7.5. Gestión de bloqueos en los SGBD

Los SGBD implementan algoritmos para gestionar bloqueos y evitar conflictos:

Bloqueo automático El sistema asigna y libera bloqueos según las operaciones realizadas.

Bloqueo explícito El usuario puede solicitar bloqueos manualmente, por ejemplo:

```
-- Bloqueo a nivel tabla
LOCK TABLE empleados IN EXCLUSIVE MODE;

-- Bloqueo a nivel fila
SELECT * FROM empleados WHERE id = 5 FOR UPDATE;
```

Escalado de bloqueos El sistema puede aumentar el nivel de bloqueo (de fila a tabla) si detecta muchas operaciones concurrentes.

6.7.6. Deadlocks (Interbloqueos)

Un deadlock ocurre cuando dos o más transacciones esperan indefinidamente por recursos bloqueados por las otras. Los SGBD detectan y resuelven deadlocks abortando una de las transacciones implicadas.

Ejemplo de deadlock:

- Transacción A bloquea la fila 1 y espera la fila 2.
- Transacción B bloquea la fila 2 y espera la fila 1.

El sistema aborta una transacción y libera los recursos para evitar el bloqueo permanente.

6.7.7. Buenas prácticas para evitar problemas de concurrencia

- Mantener las transacciones lo más cortas posible.
- Acceder a los recursos en el mismo orden en todas las transacciones.
- Utilizar el nivel de aislamiento adecuado según las necesidades de la aplicación.
- Revisar y optimizar el uso de índices para reducir el tiempo de bloqueo.
- Monitorizar y analizar los bloqueos y deadlocks mediante herramientas del SGBD.

6.7.8. Ejemplo práctico

Supongamos que dos usuarios intentan modificar el saldo de la misma cuenta simultáneamente. Si no se gestiona correctamente el bloqueo, uno de los cambios puede perderse o los datos quedar inconsistentes.

```
START TRANSACTION;
SELECT saldo FROM cuentas WHERE id = 1 FOR UPDATE;
UPDATE cuentas SET saldo = saldo - 50 WHERE id = 1;
COMMIT;
```

El uso de `FOR UPDATE` bloquea la fila hasta que la transacción se confirma, evitando actualizaciones perdidas.

6.7.9. Consideraciones de rendimiento

El exceso de bloqueos puede reducir la concurrencia y el rendimiento. Es importante equilibrar la seguridad de los datos con la eficiencia, eligiendo el nivel de aislamiento y la granularidad de bloqueo más adecuada para cada caso.

Ejercicio 6.7

Investiga cómo tu SGBD gestiona los bloqueos y los deadlocks. Realiza una prueba donde dos transacciones intenten modificar el mismo registro simultáneamente y observa el comportamiento. ¿Qué ocurre si ambas utilizan el nivel de aislamiento `SERIALIZABLE`?
Para poder ejecutar esta prueba, abre dos conexiones al mismo gestor de base de datos y ejecuta las transacciones en paralelo, haciendo uso de una espera artificial (por ejemplo, utilizando `select pg_sleep(1)` en PostgreSQL, o `select sleep(1)` en MySQL) para simular la concurrencia.

Resumen

En este capítulo se han abordado las principales operaciones de edición de datos en bases de datos relacionales: inserción (`INSERT`), borrado (`DELETE`) y actualización (`UPDATE`), incluyendo su sintaxis, variantes y errores comunes. Se ha explicado la importancia de la integridad referencial y cómo las claves foráneas y sus acciones asociadas garantizan la coherencia entre tablas. Se han presentado ejemplos prácticos de subconsultas y composición de órdenes de edición, mostrando cómo manipular datos de forma eficiente y segura. Además, se ha introducido el concepto de transacciones y sus propiedades ACID, así como las políticas de bloqueo y concurrencia necesarias para mantener la integridad y el rendimiento en entornos multiusuario. El capítulo proporciona una base

sólida para comprender y aplicar las operaciones fundamentales de tratamiento de datos en sistemas de bases de datos relacionales.

A continuación, se muestran algunas de las sentencias SQL vistas:

```sql
-- Ejemplo de inserción de un registro
INSERT INTO alumnos (nombre, email) VALUES ('Juan Pérez', 'juan@example.com');

-- Ejemplo de inserción múltiple
INSERT INTO cursos (nombre, profesor_id) VALUES
  ('Matemáticas', 2),
  ('Historia', 3);

-- Ejemplo de inserción usando una subconsulta
INSERT INTO inscripciones (alumno_id, curso_id)
SELECT id, 1 FROM alumnos WHERE promedio > 8;

-- Ejemplo de borrado condicional
DELETE FROM productos WHERE precio < 5;

-- Ejemplo de borrado usando subconsulta
DELETE FROM inscripciones WHERE curso_id IN (
  SELECT id FROM cursos WHERE estado = 'Cancelado'
);

-- Ejemplo de actualización de un registro
UPDATE empleados SET puesto = 'Jefe de área' WHERE id = 7;

-- Ejemplo de actualización masiva
UPDATE productos SET precio = precio * 1.05 WHERE categoria = 'Libros';

-- Ejemplo de actualización usando subconsulta
UPDATE cursos
SET estado = 'Finalizado'
WHERE id IN (
  SELECT curso_id FROM inscripciones WHERE fecha_baja IS NOT NULL
);

-- Ejemplo de definición de clave foránea con acción en cascada
CREATE TABLE pagos (
  id INT PRIMARY KEY,
  alumno_id INT,
  FOREIGN KEY (alumno_id) REFERENCES alumnos(id) ON DELETE CASCADE
);

-- Ejemplo de transacción
```

```
START TRANSACTION;
INSERT INTO cursos (nombre) VALUES ('Informática');
INSERT INTO inscripciones (alumno_id, curso_id) VALUES (5, ↪
  LAST_INSERT_ID());
COMMIT;

-- Ejemplo de bloqueo explícito
LOCK TABLE empleados IN EXCLUSIVE MODE;
```

7

Diagramas Entidad-Relación

Los diagramas Entidad-Relación (ER) son una herramienta fundamental para el diseño lógico de bases de datos. Permiten representar de manera gráfica las entidades relevantes de un sistema, sus atributos y las relaciones entre ellas. El uso de diagramas ER facilita la comprensión y comunicación entre los diseñadores y usuarios, asegurando que la estructura de la base de datos refleje fielmente los requisitos del negocio.

7.1. Elementos de un diagrama ER

7.1.1. Entidades

Las entidades representan objetos o conceptos del mundo real que tienen existencia independiente, como «Estudiante» o «Curso». Se representan mediante rectángulos en el diagrama ER. Cada entidad suele tener uno o más atributos que la describen.

Ejemplo: Una entidad «Paciente» con atributos como «nombre», «número de expediente» y «fecha de nacimiento».

7.1.2. Atributos

Los atributos son propiedades que describen a las entidades, por ejemplo, «nombre», «dirección» o «fecha de nacimiento». Se representan con elipses conectadas a la entidad correspondiente. Los atributos pueden ser simples, compuestos, multivaluados o derivados.

Ejemplo: El atributo «dirección» de «Paciente» puede ser compuesto por «calle», «número» y «ciudad».

7.1.3. Relaciones

Las relaciones son asociaciones entre dos o más entidades, como la relación «Consulta» entre «Paciente» y «Médico». Se representan mediante rombos y

pueden tener atributos propios. Las relaciones pueden ser binarias, ternarias o de mayor grado.

Ejemplo: La relación «Consulta» conecta las entidades «Paciente» y «Médico», indicando qué pacientes han sido atendidos por qué médicos.

7.1.4. Cardinalidades

Las cardinalidades indican el número de ocurrencias de una entidad que pueden estar asociadas a otra entidad a través de una relación. Los tipos principales son: uno a uno, uno a muchos y muchos a muchos. Las cardinalidades se especifican junto a las líneas que conectan entidades y relaciones.

Ejemplo: Un paciente puede tener varias consultas (uno a muchos), y un médico puede atender a varios pacientes (muchos a muchos).

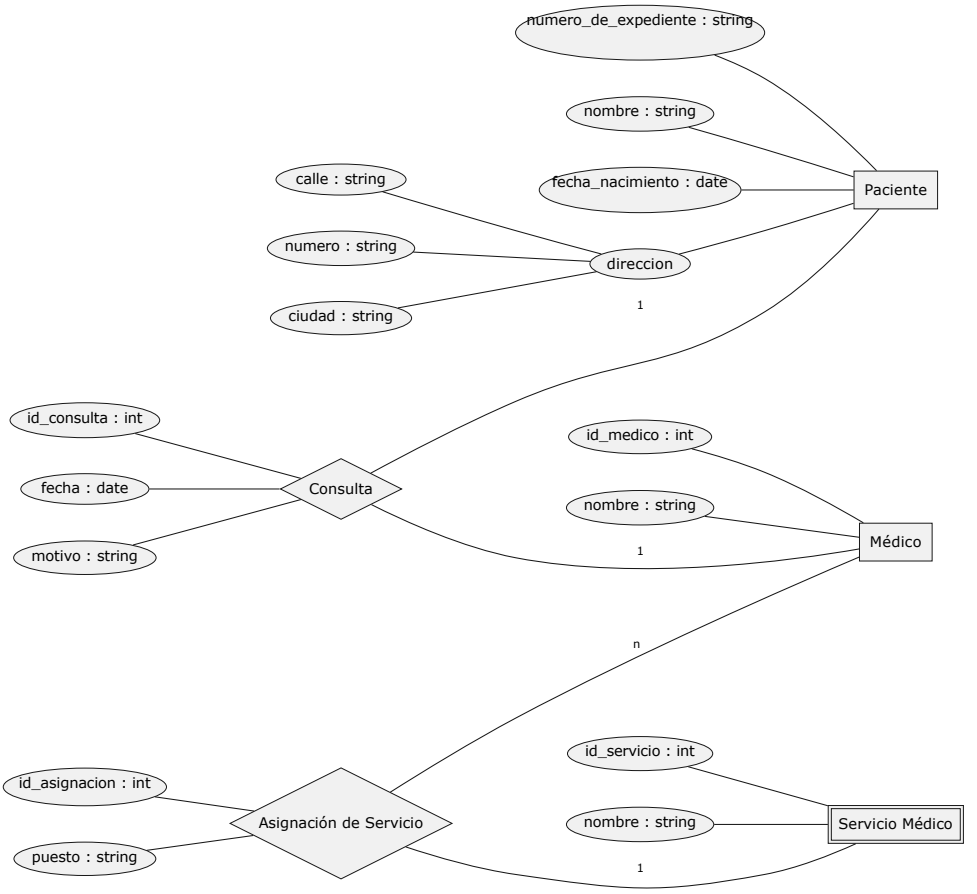
7.1.5. Entidades débiles

Las entidades débiles dependen de otra entidad para su existencia y no tienen clave primaria propia. Se representan con doble rectángulo. Su identificación depende de una entidad fuerte y de una relación identificadora.

Ejemplo: La entidad «Receta» puede ser débil si depende de la entidad «Consulta» para su identificación.

Ejemplo de diagrama Entidad-Relación

A continuación, se muestra un ejemplo de diagrama Entidad-Relación para el sistema de gestión de consultas médicas.



Este diagrama ha sido generado usando PlantUML, aunque en el mercado existen múltiples herramientas para crear diagramas Entidad-Relación, tales como Lucidchart, Draw.io o Microsoft Visio. También es posible crear estos diagramas en algunos sistemas de gestión de bases de datos, tal como ocurre con PGAdmin.

Existen distintas normas de estilo sobre como representar estos diagramas. La usada en el diagrama anterior se basa en la notación Chen. Puedes ver una lista de las notaciones más comunes en `https://en.wikipedia.org/wiki/Entity-relationship_model`.

7.2. Generalización y especialización

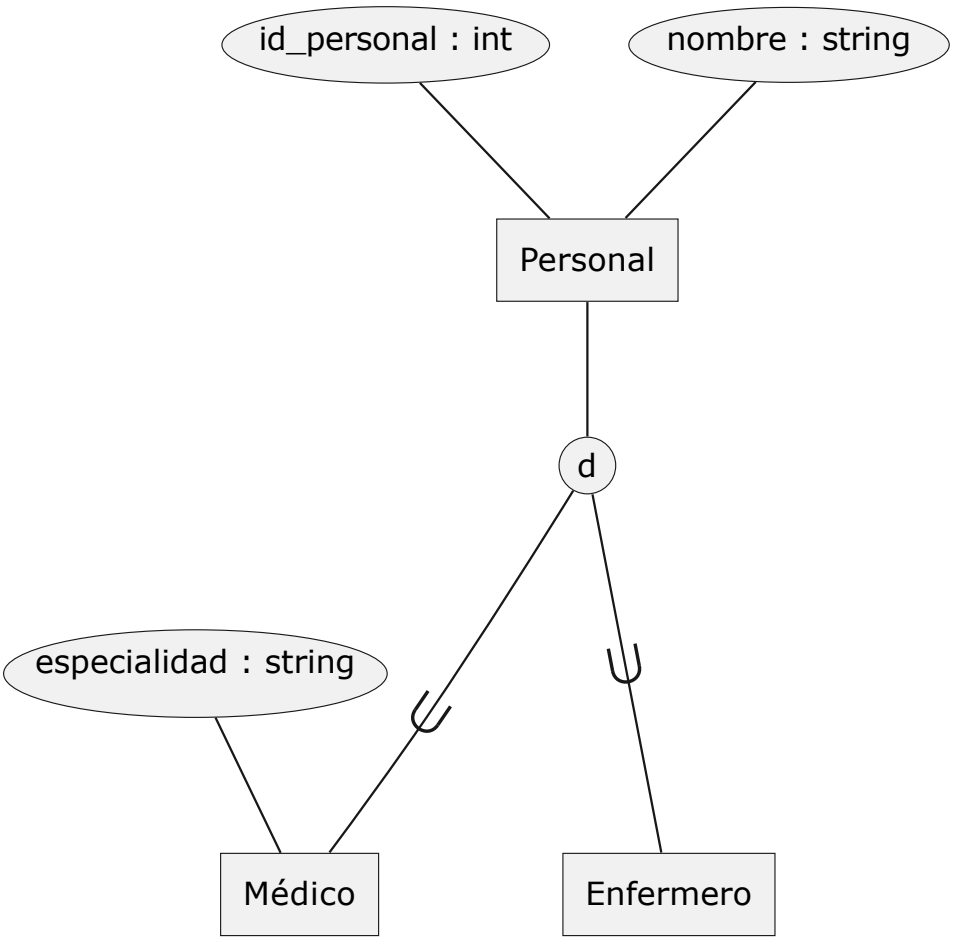
La generalización y especialización son mecanismos para organizar entidades en jerarquías. La **generalización** agrupa entidades similares en una entidad más general (por ejemplo, «Persona» como generalización de «Estudiante» y «Profesor»). La **especialización** divide una entidad en subconjuntos más específicos

con atributos adicionales.

7.2.1. Notación de generalización y especialización

En los diagramas ER, la generalización y especialización se representan mediante jerarquías, donde una entidad general se conecta con entidades especializadas mediante líneas que indican la relación de herencia. La entidad general contiene los atributos comunes, mientras que las entidades especializadas pueden tener atributos adicionales o relaciones propias.

Ejemplo: Supongamos una entidad general «Empleado» con atributos como «nombre» y «fecha de ingreso». Esta entidad puede especializarse en «Médico» y «Administrativo», donde «Médico» tiene el atributo «especialidad» y «Administrativo» el atributo «departamento».



7.2.2. Ventajas de la generalización y especialización

- Permiten reutilizar atributos y relaciones comunes, evitando redundancia.
- Facilitan la extensión del modelo ante nuevos requisitos.
- Mejoran la claridad y organización del diagrama ER.

Estos mecanismos son especialmente útiles en sistemas complejos donde existen entidades con características compartidas y diferencias específicas.

7.3. Agregación

La agregación permite tratar una relación como una entidad para establecer nuevas relaciones. Es útil cuando una relación entre entidades participa en otra relación. Por ejemplo, la relación «Inscripción» puede agregarse para relacionarla con «Pago».

7.3.1. Ventajas de la agregación

- Permite modelar situaciones complejas donde una relación participa en otra relación.
- Facilita la representación de dependencias entre procesos o eventos.
- Mejora la claridad del modelo al evitar ambigüedades en la interpretación de relaciones múltiples.

La agregación es especialmente útil en sistemas donde las relaciones tienen significado propio y requieren ser referenciadas por otras entidades o relaciones.

Ejercicio 7.1

Diseña un diagrama Entidad-Relación para un sistema de gestión de biblioteca. Identifica al menos tres entidades principales, sus atributos y las relaciones entre ellas, incluyendo cardinalidades y posibles entidades débiles.

7.4. El modelo relacional

El modelo relacional es el paradigma más utilizado en bases de datos. Organiza la información en tablas (relaciones), donde cada fila es un registro y cada columna un atributo. Las características principales son:

- **Clave primaria**: Identificador único de cada registro en una tabla.
- **Clave ajena**: Campo que referencia la clave primaria de otra tabla, estableciendo relaciones.
- **Integridad referencial**: Garantiza la coherencia de los datos entre tablas relacionadas.

Ejemplo: Consideremos las tablas `estudiantes` e `inscripciones`. La tabla `estudiantes` tiene una clave primaria `id_estudiante`, y la tabla `inscripciones` tiene una clave ajena `id_estudiante` que referencia a `estudiantes`.

```
CREATE TABLE estudiantes (
  id_estudiante INT PRIMARY KEY,
  nombre VARCHAR(100),
  ciudad VARCHAR(100)
);
CREATE TABLE inscripciones (
  id_inscripcion INT PRIMARY KEY,
  id_estudiante INT,
  FOREIGN KEY (id_estudiante) REFERENCES estudiantes(↪
    id_estudiante)
);
```

El modelo relacional es importante porque proporciona una estructura lógica clara y flexible para organizar los datos, facilitando su manipulación y consulta mediante el lenguaje SQL. Permite garantizar la integridad y coherencia de la información, simplifica el mantenimiento y la evolución de la base de datos, y es ampliamente soportado por los principales sistemas de gestión de bases de datos. Además, su enfoque en relaciones y restricciones asegura que los datos reflejen fielmente las reglas del negocio y facilita la interoperabilidad entre aplicaciones.

7.5. Paso del diagrama Entidad-Relación al modelo relacional

El proceso de transformación consiste en convertir entidades y relaciones del diagrama ER en tablas del modelo relacional. Cada entidad se convierte en una tabla, los atributos en columnas y las relaciones en tablas adicionales o claves ajenas. Las entidades débiles se transforman en tablas que incluyen la clave primaria de la entidad fuerte.

En los siguientes puntos, revisaremos el diagrama Entidad-Relación que hemos visto en este capítulo, y ejecutaremos la transformación al modelo relacional paso a paso.

7.5.1. Conversión de entidades en tablas

En este caso, las tablas son: `paciente`, `personal`, `medico`, `enfermero` y `servicio_medico`.

7.5.2. Conversión de atributos en columnas

Los atributos de la entidad se convierten en columnas de la tabla. Junto con el paso anterior, obtenemos la siguiente sentencia SQL:

```sql
CREATE TABLE paciente (
  id_paciente INT PRIMARY KEY,
  nombre VARCHAR(100),
  numero_expediente VARCHAR(50),
  fecha_nacimiento DATE
);

CREATE TABLE personal (
  id_personal INT PRIMARY KEY,
  nombre VARCHAR(100)
);

CREATE TABLE medico (
  id_medico INT PRIMARY KEY,
  especialidad VARCHAR(100)
);

CREATE TABLE enfermero (
  id_enfermero INT PRIMARY KEY
);

CREATE TABLE servicio_medico (
  id_servicio INT PRIMARY KEY,
  nombre VARCHAR(100)
);
```

7.5.3. Representación de relaciones uno a muchos

Las relaciones uno a muchos se representan mediante claves ajenas. Por ejemplo, la relación entre `servicio_medico` y `medico` se representa añadiendo una clave ajena en la tabla `medico` que referencia a `servicio_medico`.

```sql
ALTER TABLE medico
ADD CONSTRAINT fk_servicio
FOREIGN KEY (id_servicio) REFERENCES servicio_medico(id_servicio
  );
```

7.5.4. Conversión de relaciones muchos a muchos

Las relaciones muchos a muchos se convierten en tablas intermedias. En este caso, crearemos la tabla `consulta`, que representa la relación entre `paciente` y `medico`, y añadiremos las claves foráneas correspondientes.

```
CREATE TABLE consulta (
  id_consulta INT PRIMARY KEY,
  fecha DATE,
  motivo VARCHAR(255),
  id_paciente INT,
  id_medico INT,
  FOREIGN KEY (id_paciente) REFERENCES paciente(id_paciente),
  FOREIGN KEY (id_medico) REFERENCES medico(id_medico)
);
```

7.5.5. Transformación de entidades débiles

Las entidades débiles incluyen la clave primaria de la entidad fuerte. Supongamos que `receta` es una entidad débil dependiente de `consulta`. Crearemos la tabla, la cual incluirá la clave primaria de `consulta`.

```
CREATE TABLE receta (
  id_receta INT PRIMARY KEY,
  fecha DATE,
  id_consulta INT,
  FOREIGN KEY (id_consulta) REFERENCES consulta(id_consulta)
);
```

Ejercicio 7.2

Transforma el diagrama Entidad-Relación del ejercicio anterior, en un esquema relacional en SQL.

7.6. Restricciones semánticas del modelo relacional

Las restricciones semánticas definen reglas adicionales sobre los datos, como unicidad, obligatoriedad, rangos de valores y dependencias. Ejemplos:

- Un estudiante no puede inscribirse dos veces en el mismo curso.
- El email debe ser único en la tabla de profesores.
- La fecha de inscripción debe ser posterior a la fecha de nacimiento.

Estas restricciones pueden implementarse mediante claves únicas, restricciones CHECK y triggers.

7.6.1. Ejemplo de restricciones semánticas en SQL

A continuación, se muestran ejemplos de cómo implementar restricciones semánticas en el modelo relacional utilizando sentencias SQL:

- **Restricción de unicidad**: Para asegurar que el campo `email` en la tabla `profesor` sea único.

```sql
CREATE TABLE profesor (
  id_profesor INT PRIMARY KEY,
  nombre VARCHAR(100),
  email VARCHAR(100) UNIQUE
);
```

- **Restricción CHECK**: Para garantizar que la fecha de inscripción sea posterior a la fecha de nacimiento.

```sql
CREATE TABLE estudiante (
  id_estudiante INT PRIMARY KEY,
  nombre VARCHAR(100),
  fecha_nacimiento DATE,
  fecha_inscripcion DATE,
  CHECK (fecha_inscripcion > fecha_nacimiento)
);
```

- **Restricción de no duplicidad**: Para evitar que un estudiante se inscriba dos veces en el mismo curso, se puede usar una restricción de clave única compuesta.

```sql
CREATE TABLE inscripcion (
  id_inscripcion INT PRIMARY KEY,
  id_estudiante INT,
  id_curso INT,
  fecha DATE,
  UNIQUE (id_estudiante, id_curso),
  FOREIGN KEY (id_estudiante) REFERENCES estudiante(↪
  id_estudiante),
  FOREIGN KEY (id_curso) REFERENCES curso(id_curso)
);
```

En casos más complejos, como validar rangos de valores o dependencias entre columnas, pueden utilizarse `triggers` para definir reglas personalizadas que se ejecutan automáticamente ante operaciones de inserción o actualización. Por

ejemplo, un trigger podría impedir que se registre una consulta médica en una fecha anterior al nacimiento del paciente.

Estas restricciones ayudan a mantener la calidad y coherencia de los datos, reflejando las reglas de negocio directamente en la base de datos.

7.7. Normalización

La normalización es el proceso de organizar los datos para reducir la redundancia y mejorar la integridad. Se realiza mediante la aplicación de formas normales.

7.7.1. Primera Forma Normal (1NF)

La Primera Forma Normal exige que cada columna de una tabla contenga valores atómicos, es decir, sin grupos repetitivos ni listas de valores. Esto significa que cada celda debe tener un solo valor y no una colección de valores. Para cumplir con 1NF, se deben eliminar los atributos multivaluados y compuestos, creando nuevas filas o tablas si es necesario.

Ejemplo: Supongamos una tabla `paciente` con un atributo `telefonos` que almacena varios números en una sola celda. Para normalizar, se crea una tabla `telefono_paciente` donde cada número de teléfono ocupa una fila distinta.

```
CREATE TABLE telefono_paciente (
  id_paciente INT,
  telefono VARCHAR(20),
  FOREIGN KEY (id_paciente) REFERENCES paciente(id_paciente)
);
```

7.7.2. Segunda Forma Normal (2NF)

La Segunda Forma Normal se aplica a tablas que ya cumplen 1NF y tienen una clave primaria compuesta. 2NF elimina las dependencias parciales, es decir, los atributos que dependen solo de una parte de la clave primaria y no de toda ella. Para lograrlo, se separan los atributos en nuevas tablas, asegurando que cada atributo dependa de la clave completa.

Ejemplo: En una tabla `inscripcion` con clave primaria compuesta (`id_estudiante`, `id_curso`), si el atributo `nombre_curso` depende solo de `id_curso`, debe moverse a una tabla `curso`.

```
CREATE TABLE curso (
  id_curso INT PRIMARY KEY,
  nombre_curso VARCHAR(100)
);
```

7.7.3. Tercera Forma Normal (3NF)

La Tercera Forma Normal requiere que la tabla esté en 2NF y que no existan dependencias transitivas, es decir, que los atributos no clave dependan de otros atributos no clave. Todos los atributos deben depender únicamente de la clave primaria. Para cumplir 3NF, se separan los atributos en nuevas tablas si dependen de otros atributos no clave.

Ejemplo: Si en la tabla `paciente` existe el atributo `ciudad` y otro atributo `codigo_postal`, y `codigo_postal` depende de `ciudad`, se debe crear una tabla aparte para la relación entre ciudad y código postal.

```
CREATE TABLE ciudad (
  id_ciudad INT PRIMARY KEY,
  nombre_ciudad VARCHAR(100),
  codigo_postal VARCHAR(10)
);
```

7.7.4. Cuarta Forma Normal (4NF)

La Cuarta Forma Normal (4NF) se enfoca en eliminar las dependencias multivaluadas en una tabla. Una dependencia multivaluada ocurre cuando una entidad tiene dos o más conjuntos de atributos independientes entre sí, pero ambos dependen de la clave primaria. Para cumplir con 4NF, cada tabla debe contener solo dependencias funcionales simples, evitando que una fila represente múltiples valores independientes para diferentes atributos.

Ejemplo: Supongamos una tabla `paciente` con los atributos `telefono` y `alergia`, donde un paciente puede tener varios teléfonos y varias alergias, y ambos conjuntos son independientes. Si se almacenan en la misma tabla, se generan combinaciones redundantes:

id_paciente	telefono	alergia
1	555-1234	Penicilina
1	555-1234	Polvo
1	555-5678	Penicilina
1	555-5678	Polvo

Para normalizar a 4NF, se deben separar las dependencias multivaluadas en tablas distintas:

```
CREATE TABLE telefono_paciente (
  id_paciente INT,
  telefono VARCHAR(20),
  FOREIGN KEY (id_paciente) REFERENCES paciente(id_paciente)
);

```

```sql
CREATE TABLE alergia_paciente (
  id_paciente INT,
  alergia VARCHAR(100),
  FOREIGN KEY (id_paciente) REFERENCES paciente(id_paciente)
);
```

De este modo, se elimina la redundancia y se asegura que cada tabla represente una sola dependencia multivaluada, mejorando la integridad y eficiencia del modelo de datos.

La 4NF es especialmente relevante en sistemas donde los datos presentan múltiples conjuntos independientes de valores asociados a una misma entidad, como teléfonos, alergias, direcciones, etc. Aplicar esta forma normal ayuda a evitar inconsistencias y facilita el mantenimiento de la base de datos.

7.7.5. Quinta Forma Normal (5NF)

La Quinta Forma Normal (5NF), también conocida como Forma Normal de Proyección-Conjunción, se ocupa de eliminar las llamadas «dependencias de unión» en una tabla. Una dependencia de unión ocurre cuando una tabla no puede ser reconstruida correctamente a partir de sus proyecciones (subconjuntos de columnas) debido a la presencia de datos redundantes o inconsistentes que sólo pueden resolverse mediante la descomposición en varias tablas.

En 5NF, una tabla está completamente descompuesta en el menor número posible de tablas, de modo que todas las dependencias de unión se eliminan y los datos pueden ser reconstruidos sin pérdida ni ambigüedad mediante operaciones de unión (`JOIN`). Esta forma normal es relevante en situaciones donde existen relaciones complejas entre tres o más conjuntos de atributos, y la descomposición en tablas menores evita la redundancia y las anomalías de actualización.

Ejemplo: Supongamos una tabla que registra qué productos pueden ser fabricados por qué fábricas, usando qué materiales. Si la relación entre producto, fábrica y material es compleja, puede ser necesario dividir la tabla en tres relaciones binarias (`producto-fábrica`, `producto-material`, `fábrica-material`) para evitar redundancias y asegurar la integridad de los datos.

Producto	Fábrica	Material
A	X	M1
A	X	M2
A	Y	M1
B	Y	M2

Si la combinación de producto, fábrica y material no puede representarse correctamente sólo con las relaciones binarias, es necesario mantener la tabla original o descomponerla en tablas que permitan reconstruir la información mediante `JOIN` sin pérdida.

La 5NF es poco común en aplicaciones prácticas, pero es importante en sistemas donde existen relaciones complejas y múltiples dependencias entre conjuntos de atributos. Aplicar la Quinta Forma Normal garantiza la máxima integridad y flexibilidad en el diseño de la base de datos, evitando redundancias y facilitando el mantenimiento.

En resumen, la 5NF asegura que los datos estén completamente descompuestos y que todas las dependencias de unión sean eliminadas, permitiendo reconstruir la información original mediante operaciones de unión sin pérdida ni ambigüedad.

7.7.6. Sexta Forma Normal (6NF)

La Sexta Forma Normal (6NF) es una extensión de la 5NF que se centra en la descomposición de relaciones temporales. En sistemas donde los datos cambian con el tiempo, es crucial mantener un historial de cambios y permitir consultas eficientes sobre datos temporales.

La 6NF se logra descomponiendo las tablas en relaciones más pequeñas que capturan la evolución de los datos a lo largo del tiempo. Esto implica crear tablas adicionales para almacenar versiones históricas de los datos, junto con marcas de tiempo que indiquen cuándo se realizaron los cambios.

Ejemplo: Supongamos que en la tabla `paciente` se desea almacenar el historial de direcciones y teléfonos, registrando cuándo cada dato estuvo vigente. En 6NF, cada atributo temporal se almacena en una tabla separada, junto con los campos de validez temporal (`fecha_inicio`, `fecha_fin`):

```
CREATE TABLE direccion_paciente (
  id_paciente INT,
  direccion VARCHAR(255),
  fecha_inicio DATE,
  fecha_fin DATE,
  FOREIGN KEY (id_paciente) REFERENCES paciente(id_paciente)
);

CREATE TABLE telefono_paciente (
  id_paciente INT,
  telefono VARCHAR(20),
  fecha_inicio DATE,
  fecha_fin DATE,
  FOREIGN KEY (id_paciente) REFERENCES paciente(id_paciente)
);
```

De este modo, es posible consultar la dirección o el teléfono de un paciente en una fecha específica, reconstruyendo el historial completo de cambios. La 6NF facilita la gestión de datos temporales, mejora la trazabilidad y permite cumplir requisitos legales o de auditoría relacionados con la evolución de la información.

En resumen, la Sexta Forma Normal es esencial para sistemas que requieren un control preciso sobre la validez temporal de los datos, permitiendo una descomposición máxima y consultas eficientes sobre el historial de cambios.

7.7.7. La importancia de la normalización

La normalización es fundamental en el diseño de bases de datos porque permite organizar los datos de manera eficiente, evitando redundancias y anomalías de actualización. Al aplicar las formas normales, se garantiza que cada dato se almacene una sola vez, lo que facilita el mantenimiento y la evolución del sistema. Además, la normalización mejora la integridad de los datos, ya que las dependencias y restricciones se representan explícitamente en la estructura de la base de datos.

Sin embargo, es importante encontrar un equilibrio entre normalización y rendimiento. En algunos casos, una normalización excesiva puede llevar a un gran número de tablas y consultas complejas, lo que puede afectar la eficiencia de las operaciones. Por ello, en sistemas de producción, a veces se recurre a la desnormalización controlada para optimizar el acceso a los datos, especialmente en aplicaciones de alto volumen de lectura.

En resumen, la normalización ayuda a:

- Eliminar la redundancia y evitar inconsistencias en los datos.
- Facilitar la actualización y el mantenimiento de la base de datos.
- Mejorar la integridad y calidad de la información.
- Adaptar el modelo de datos a cambios futuros en los requisitos del sistema.

La correcta aplicación de la normalización es clave para construir bases de datos robustas, escalables y fáciles de gestionar, asegurando que la información refleje fielmente las reglas y necesidades del negocio.

Ejercicio 7.3

Diseña un sistema de base de datos para una clínica veterinaria. Realiza los siguientes pasos:

1. Identifica las entidades principales, sus atributos y las relaciones entre ellas. Dibuja el diagrama Entidad-Relación (ER) correspondiente, indicando las cardinalidades y posibles entidades débiles.
2. Transforma el diagrama ER al modelo relacional, especificando las tablas, claves primarias y foráneas, y las restricciones semánticas

necesarias (unicidad, obligatoriedad, etc.).

3. Aplica el proceso de normalización hasta la Tercera Forma Normal (3NF), justificando cada paso.

4. Escribe las sentencias SQL para crear las tablas principales y las restricciones.

Nota: Incluye al menos las entidades `Mascota`, `Propietario`, `Veterinario`, `Consulta` y `Tratamiento`. Considera atributos relevantes y relaciones como consultas realizadas, tratamientos aplicados y asignación de veterinarios.

Resumen

Los diagramas Entidad-Relación son esenciales para el diseño de bases de datos, permitiendo modelar entidades, relaciones y restricciones de manera gráfica y comprensible. Estos diagramas facilitan la comunicación entre usuarios y diseñadores, asegurando que la estructura de la base de datos refleje los requisitos del negocio.

La transformación del diagrama Entidad-Relación al modelo relacional implica convertir entidades y relaciones en tablas y claves foráneas, lo que permite implementar la base de datos en sistemas reales. El uso de restricciones semánticas garantiza la integridad y coherencia de los datos, reflejando las reglas de negocio directamente en la base de datos.

La normalización, mediante la aplicación de formas normales, ayuda a reducir la redundancia y mejorar la integridad de los datos, asegurando que la información esté organizada de manera eficiente. Este proceso facilita el mantenimiento, la evolución y la consulta de la base de datos, permitiendo adaptarse a nuevos requisitos y cambios en el sistema.

En conjunto, el uso de diagramas Entidad-Relación, el modelo relacional y la normalización constituyen la base para el diseño lógico y físico de bases de datos robustas, escalables y fáciles de gestionar.

8

Construcción de guiones: *scripts* SQL

Los scripts SQL son conjuntos de instrucciones que permiten automatizar tareas en bases de datos, como la creación de estructuras, la manipulación de datos y la administración de usuarios y permisos. El uso de scripts facilita la reproducibilidad, el mantenimiento y la migración de bases de datos entre entornos.

Importante

Cada SGBD tiene sus propias particularidades y extensiones para los scripts SQL. Es fundamental consultar la documentación específica del SGBD que se esté utilizando.
A continuación se presentan enlaces a los manuales oficiales de algunos SGBD populares:

MySQL `https://dev.mysql.com/doc/`

PostgreSQL `https://www.postgresql.org/docs/`

Oracle Database `https://docs.oracle.com/en/database/oracle/oracle-database/`

Debido a que PostgreSQL facilita el proceso de programación de scripts SQL con características avanzadas como procedimientos almacenados, funciones y triggers, en este capítulo se utilizarán ejemplos basados en PostgreSQL. No obstante, los conceptos presentados son aplicables a otros SGBD con las adaptaciones necesarias.

8.1. Estructura de un script SQL

La estructura de un script SQL suele incluir:

- Comentarios para documentar el propósito y los pasos del script.
- Sentencias de definición de datos (DDL): CREATE, ALTER, DROP.
- Sentencias de manipulación de datos (DML): INSERT, UPDATE, DELETE, SELECT.
- Sentencias de control de transacciones: BEGIN, COMMIT, ROLLBACK.
- Control de errores y condiciones.

```
-- Crear tabla
CREATE TABLE cursos (
  id_curso INT PRIMARY KEY AUTO_INCREMENT,
  nombre_curso VARCHAR(100) NOT NULL,
  descripcion TEXT
);

-- Insertar datos
INSERT INTO cursos VALUES (1, 'Matemáticas', 'Curso de matemáticas básicas');
INSERT INTO cursos VALUES (2, 'Historia', 'Curso de historia mundial');

-- Actualizar datos
UPDATE cursos SET nombre_curso = 'Historia Universal' WHERE id_curso = 2;

-- Eliminar datos
DELETE FROM cursos WHERE id_curso = 1;

-- Matricular a cada alumno sin curso asignado en el curso de 'Historia Universal'
DO $$
DECLARE
  fila RECORD;
BEGIN
  FOR fila IN
    SELECT e.id_estudiante
    FROM estudiantes e
    LEFT JOIN inscripciones i ON e.id_estudiante = i.id_estudiante
    GROUP BY e.id_estudiante
    HAVING COUNT(i.id_estudiante) = 0
  LOOP
    INSERT INTO inscripciones (id_estudiante, id_curso, fecha_inscripcion) VALUES (fila.id_estudiante, 2, CURRENT_DATE);
  END LOOP;
```

```
END $$;
```

Ejemplo básico de un script SQL

Consejo

Es recomendable utilizar scripts SQL para realizar cambios en la base de datos en lugar de ejecutar comandos manualmente. Esto permite llevar un registro de los cambios y facilita la reversión en caso de errores.

8.2. Comentarios

Los comentarios en SQL son fundamentales para documentar scripts y facilitar su comprensión y mantenimiento. Permiten explicar el propósito de las instrucciones, advertir sobre posibles riesgos y dejar notas para otros desarrolladores.

Existen dos formas principales de agregar comentarios en scripts SQL:

- **Comentario de una línea:** Se utiliza el doble guion `--`. Todo lo que sigue en la línea es ignorado por el intérprete SQL.
- **Comentario de varias líneas:** Se encierra el texto entre `/*` y `*/`. Todo el contenido entre estos delimitadores es ignorado, incluso si abarca varias líneas.

Ejemplo de comentarios:

```
-- Este es un comentario de una línea

/*
  Este es un comentario
  de varias líneas.
  Se puede usar para explicar bloques de código complejos.
*/

CREATE TABLE estudiantes (
  id_estudiante SERIAL PRIMARY KEY, -- Identificador único
  nombre VARCHAR(100) NOT NULL,    -- Nombre del estudiante
  ciudad VARCHAR(50)          -- Ciudad de residencia
);
```

Buenas prácticas:

- Comenta el propósito general del script al inicio.
- Explica secciones complejas o poco evidentes.

- Usa comentarios para advertir sobre posibles efectos secundarios.
- Mantén los comentarios actualizados cuando modifiques el código.

Los comentarios no afectan la ejecución del script, pero son clave para la colaboración y el mantenimiento a largo plazo.

8.3. Tipos de datos, identificadores y variables

Los tipos de datos definen la naturaleza de la información almacenada en cada columna. Los más comunes son:

- **INTEGER**: Números enteros.
- **VARCHAR(n)**: Cadenas de texto de longitud variable.
- **DATE**: Fechas.
- **BOOLEAN**: Valores lógicos (TRUE/FALSE).
- **SERIAL**: Enteros autoincrementales (PostgreSQL).
- **TEXT**: Cadenas de texto de longitud ilimitada.
- **TIMESTAMP**: Fecha y hora.

Los identificadores son nombres de tablas, columnas y variables. Deben ser únicos y descriptivos. En algunos sistemas, los scripts pueden incluir variables para almacenar resultados temporales o parámetros. En PostgreSQL, las variables se declaran en bloques `DO` o en funciones:

```
DO $$
DECLARE
  contador INTEGER;
  ciudad_con_mas_estudiantes VARCHAR(100);
BEGIN
  SELECT count(*), ciudad INTO contador, ↪
  ciudad_con_mas_estudiantes FROM estudiantes GROUP BY ciudad ↪
  ORDER BY COUNT(*) DESC LIMIT 1;

  -- Imprime el resultado
  RAISE NOTICE 'Ciudad con más estudiantes: %, Total de ↪
  estudiantes: %', ciudad_con_mas_estudiantes, contador;
END $$;
```

Ejercicio 8.1

Crea un script que muestre el nombre del curso con más estudiantes, así como el número total de estudiantes inscritos.

8.4. Operadores en SQL

SQL dispone de operadores para realizar comparaciones, cálculos y operaciones lógicas. Estos operadores son esenciales para construir consultas y manipular datos de manera efectiva.

8.4.1. Operadores aritméticos

Los operadores aritméticos permiten realizar cálculos matemáticos sobre los datos numéricos de las tablas. Son fundamentales para sumar, restar, multiplicar, dividir y obtener el residuo de una división.

- **Suma (+)**: Añade dos valores.
- **Resta (-)**: Sustrae un valor de otro.
- **Multiplicación (*)**: Multiplica dos valores.
- **División (/)**: Divide un valor entre otro.
- **Módulo (%)**: Devuelve el residuo de la división.

Ejemplo:

```sql
SELECT 10 + 5 AS suma, 10 - 5 AS resta, 10 * 5 AS multiplicacion
  , 10 / 5 AS division, 10 % 3 AS modulo;
-- Resultado: suma=15, resta=5, multiplicacion=50, division=2,
  modulo=1
```

También pueden aplicarse sobre columnas:

```sql
SELECT nombre_curso, precio, precio * 0.9 AS precio_descuento
FROM cursos;
-- Calcula el precio con un 10% de descuento
```

8.4.2. Operadores de comparación

Estos operadores se utilizan para comparar valores y establecer condiciones en las consultas. Son esenciales en cláusulas `WHERE` y en expresiones condicionales.

- **Igual (=)**: Verifica si dos valores son iguales.
- **Distinto (<>)**: Verifica si dos valores son diferentes.
- **Mayor que (>)**, **Menor que (<)**: Comparan magnitudes.
- **Mayor o igual (>=)**, **Menor o igual (<=)**: Comparan incluyendo igualdad.

Ejemplo

```
SELECT * FROM cursos WHERE precio >= 100;
-- Selecciona cursos con precio mayor o igual a 100

SELECT nombre_curso FROM cursos WHERE nombre_curso <> 'Matemáticas';
-- Selecciona cursos cuyo nombre no es 'Matemáticas'
```

8.4.3. Operadores lógicos

Permiten combinar múltiples condiciones en una consulta. Son fundamentales para construir filtros complejos.

- **AND**: Todas las condiciones deben cumplirse.
- **OR**: Al menos una condición debe cumplirse.
- **NOT**: Niega una condición.

Ejemplo

```
SELECT * FROM cursos WHERE precio > 50 AND nombre_curso LIKE 'H%';
-- Cursos con precio mayor a 50 y cuyo nombre comienza con 'H'

SELECT * FROM cursos WHERE NOT (precio < 100);
-- Cursos con precio mayor o igual a 100
```

8.4.4. Operadores de conjunto

Estos operadores permiten combinar resultados de varias consultas, facilitando la manipulación de conjuntos de datos.

- **UNION**: Une los resultados de dos consultas, eliminando duplicados.
- **UNION ALL**: Une los resultados de dos consultas, incluyendo duplicados.

- **INTERSECT**: Devuelve los registros que aparecen en ambas consultas.
- **EXCEPT**: Devuelve los registros de la primera consulta que no están en la segunda.

Ejemplo en psql:

```
SELECT nombre_curso FROM cursos WHERE precio > 100
UNION
SELECT nombre_curso FROM cursos WHERE descripcion LIKE '%avanzado%';
-- Cursos con precio mayor a 100 o con descripción que contiene 'avanzado'

SELECT nombre_curso FROM cursos
INTERSECT
SELECT nombre_curso FROM inscripciones;
-- Cursos que tienen inscripciones

SELECT nombre_curso FROM cursos
EXCEPT
SELECT nombre_curso FROM inscripciones;
-- Cursos que no tienen inscripciones
```

8.5. Funciones y procedimientos almacenados

Las funciones y procedimientos almacenados son bloques de código que se guardan en el servidor de bases de datos y pueden ser ejecutados repetidamente para realizar tareas específicas. Permiten encapsular lógica compleja, reutilizar código y mejorar la seguridad y el rendimiento de las operaciones en la base de datos.

8.5.1. Funciones integradas en SQL

Además de poder definir funciones propias, los SGBD ofrecen una amplia variedad de funciones integradas para manipular datos de manera eficiente. Estas funciones abarcan desde operaciones aritméticas hasta manipulación de cadenas, fechas y otros tipos de datos.

Funciones aritméticas

Las funciones aritméticas permiten realizar cálculos matemáticos sobre los datos numéricos. Algunos ejemplos comunes son:

- `ABS(x)`: Devuelve el valor absoluto de `x`.

- `ROUND(x, n)`: Redondea `x` a `n` decimales.
- `CEIL(x)` o `CEILING(x)`: Devuelve el menor entero mayor o igual que `x`.
- `FLOOR(x)`: Devuelve el mayor entero menor o igual que `x`.
- `POWER(x, y)`: Calcula `x` elevado a la potencia `y`.
- `SQRT(x)`: Devuelve la raíz cuadrada de `x`.
- `RANDOM()`: Devuelve un número aleatorio entre 0 y 1.

Ejemplo:

```
SELECT ABS(-15) AS absoluto, ROUND(3.14159, 2) AS redondeado, CEIL(2.3) AS techo, FLOOR(2.7) AS piso, POWER(2, 3) AS potencia, SQRT(16) AS raiz;
-- Resultado: absoluto=15, redondeado=3.14, techo=3, piso=2, potencia=8, raiz=4
```

Funciones de manipulación de cadenas

Las funciones de cadenas permiten modificar, analizar y extraer información de valores de texto.

- `LENGTH(texto)`: Devuelve la longitud de la cadena.
- `LOWER(texto)`: Convierte la cadena a minúsculas.
- `UPPER(texto)`: Convierte la cadena a mayúsculas.
- `SUBSTRING(texto FROM inicio FOR longitud)`: Extrae una subcadena.
- `CONCAT(texto1, texto2, ...)`: Une varias cadenas.
- `TRIM(texto)`: Elimina espacios en blanco al inicio y final.
- `REPLACE(texto, 'buscar', 'reemplazar')`: Reemplaza partes de la cadena.

Ejemplo:

```
SELECT LENGTH('Bases de datos') AS longitud,
    LOWER('Bases de Datos') AS minusculas,
    UPPER('Bases de Datos') AS mayusculas,
    SUBSTRING('Bases de datos' FROM 1 FOR 5) AS subcadena,
    CONCAT('Curso: ', 'SQL') AS concatenado,
    TRIM('  texto  ') AS sin_espacios,
```

```
    REPLACE('Bases de datos', 'datos', 'información') AS reemplazo;
-- Resultado: longitud=14, minusculas='bases de datos', mayusculas='BASES DE DATOS', subcadena='Bases', concatenado='Curso: SQL', sin_espacios='texto', reemplazo='Bases de información'
```

Funciones de fecha y hora

Las funciones de fecha y hora permiten trabajar con valores temporales.

- `CURRENT_DATE`: Devuelve la fecha actual.
- `CURRENT_TIME`: Devuelve la hora actual.
- `NOW()`: Devuelve la fecha y hora actual.
- `AGE(fecha)`: Calcula la diferencia entre la fecha dada y la actual.
- `EXTRACT(field FROM fecha)`: Extrae partes específicas de una fecha (año, mes, día, etc.).

Ejemplo:

```
SELECT CURRENT_DATE AS fecha, CURRENT_TIME AS hora, NOW() AS fecha_hora, AGE('2020-01-01') AS antiguedad, EXTRACT(YEAR FROM NOW()) AS año;
-- Resultado: fecha='2024-06-07', hora='14:23:00', fecha_hora='2024-06-07 14:23:00', antiguedad='4 years ...', año=2024
```

Consejo

Antes de crear funciones personalizadas, revisa la documentación del SGBD para aprovechar las funciones integradas, que suelen estar optimizadas y cubren la mayoría de necesidades comunes.

8.6. Control de flujo en scripts SQL

El control de flujo permite ejecutar instrucciones condicionalmente o de forma repetitiva. Se utiliza principalmente en procedimientos almacenados y funciones en los que se requiere lógica más compleja.

8.6.1. IF...ELSE

Permite ejecutar instrucciones dependiendo de si una condición se cumple o no. Es útil para tomar decisiones dentro de scripts y procedimientos.

```
DO $$
BEGIN
  IF (SELECT COUNT(*) FROM cursos) > 5 THEN
    RAISE NOTICE 'Hay más de 5 cursos';
  ELSE
    RAISE NOTICE 'Hay 5 o menos cursos';
  END IF;
END $$;
```

Ejercicio 8.2

Crea un script que verifique si existe al menos un curso con más de 50 estudiantes inscritos. Si es así, muestra un mensaje indicando que hay cursos populares; si no, muestra un mensaje indicando que ningún curso supera los 50 estudiantes.

8.6.2. CASE

Permite seleccionar entre múltiples alternativas según el valor de una expresión. Es comúnmente usado en consultas para categorizar resultados.

```
SELECT nombre_curso,
  CASE
    WHEN descripcion IS NULL THEN 'Sin descripción'
    WHEN LENGTH(descripcion) < 20 THEN 'Descripción corta'
    ELSE 'Descripción larga'
  END AS tipo_descripcion
FROM cursos;
```

Ejercicio 8.3

Escribe un script que clasifique los cursos según el número de estudiantes inscritos: si tiene más de 30 inscritos, debe mostrar «Curso grande»; si tiene entre 10 y 30, «Curso mediano»; y si tiene menos de 10, «Curso pequeño». Muestra el nombre del curso y su clasificación.

8.6.3. LOOP, WHILE, FOR

Permiten repetir instrucciones mientras se cumpla una condición o para recorrer conjuntos de datos.

LOOP: es una estructura de control en PL/pgSQL utilizada para ejecutar repetidamente un bloque de instrucciones hasta que se encuentre una sentencia de salida explícita, como EXIT o RETURN. Es útil para realizar tareas iterativas cuando no se conoce de antemano el número de repeticiones. En el contexto de psql, permite implementar bucles personalizados dentro de funciones o procedimientos almacenados.

Ejemplo de LOOP:

```
DO $$
DECLARE
  i INTEGER := 1;
BEGIN
  LOOP
    EXIT WHEN i > 3;
    RAISE NOTICE 'Iteración %', i;
    i := i + 1;
  END LOOP;
END $$;
```

Ejercicio 8.4

Escribe un script que utilice un bucle LOOP para sumar los números del 1 al 10 y mostrar el resultado final con un mensaje.

FOR: es una estructura de control en PL/pgSQL que permite iterar sobre un conjunto de valores, como los resultados de una consulta. Es útil para procesar filas de una tabla o ejecutar un bloque de código un número específico de veces.

Ejemplo de FOR:

```
DO $$
DECLARE
  fila RECORD;
BEGIN
  FOR fila IN SELECT nombre FROM cursos LOOP
    RAISE NOTICE 'Curso: %', fila.nombre;
  END LOOP;
END $$;
```

Ejercicio 8.5

Escribe un script que recorra todos los cursos y, para cada uno, cuente cuántos estudiantes están inscritos. Muestra el nombre del curso y el número de estudiantes inscritos utilizando un bucle FOR.

WHILE: es una estructura de control en PL/pgSQL que permite ejecutar un bloque de instrucciones repetidamente mientras se cumpla una condición. Es

útil para realizar tareas iterativas cuando no se conoce de antemano el número de repeticiones.

Ejemplo de WHILE:

```
DO $$
DECLARE
  contador INTEGER := 0;
BEGIN
  WHILE contador < 5 LOOP
    RAISE NOTICE 'Contador: %', contador;
    contador := contador + 1;
  END LOOP;
END $$;
```

Ejercicio 8.6

Escribe un script que utilice un bucle WHILE para generar los números pares del 2 al 20 y mostrar cada número con un mensaje.

Resumen

Los scripts SQL son esenciales para la gestión eficiente de bases de datos. Permiten automatizar tareas, garantizar la integridad de los datos y facilitar la colaboración entre desarrolladores y administradores. Un buen diseño de scripts incluye comentarios, control de errores y el uso adecuado de tipos de datos y operadores.

C

Contenidos exclusivos del módulo Gestión de Base de Datos (0372)

9

Administración de Bases de Datos

La administración de bases de datos es el conjunto de tareas y procedimientos necesarios para garantizar la seguridad, integridad, disponibilidad y rendimiento de la información almacenada. En PostgreSQL, el administrador debe conocer las herramientas y comandos específicos para gestionar usuarios, realizar copias de seguridad, recuperar datos ante fallos y optimizar el sistema.

Importante

Cada SGBD tiene sus propias herramientas y procedimientos para la administración de bases de datos. En este capítulo, nos centraremos en las particularidades de PostgreSQL.

9.1. Gestión de bases de datos

La gestión de bases de datos en PostgreSQL abarca la creación, modificación, eliminación y mantenimiento de las bases de datos dentro de un clúster. El administrador debe conocer los comandos y herramientas que permiten realizar estas tareas de forma segura y eficiente.

9.1.1. Creación de bases de datos

Para crear una nueva base de datos, se utiliza el comando `CREATE DATABASE` en SQL, o la herramienta de línea de comandos `createdb`. Es importante definir el propietario, la codificación y otros parámetros relevantes.

```
-- Crear una base de datos llamada universidad
CREATE DATABASE universidad
  WITH OWNER = exampleuser
  ENCODING = 'UTF8'
  LC_COLLATE = 'es_ES.UTF-8'
  LC_CTYPE = 'es_ES.UTF-8'
  TEMPLATE = template0;
```

```
# Crear una base de datos desde la terminal
createdb -U exampleuser -E UTF8 universidad
```

9.1.2. Listado y conexión a bases de datos

Para listar las bases de datos existentes, se puede usar el comando `-l` en `psql` o la consulta `SELECT` sobre la vista `pg_database`.

```
-- Listar bases de datos
SELECT datname, datdba, encoding FROM pg_database;
```

```
# Listar bases de datos en psql
psql -U exampleuser -l
```

Para conectarse a una base de datos específica:

```
psql -U exampleuser -d universidad
```

9.1.3. Modificación de bases de datos

Algunas propiedades de la base de datos pueden modificarse tras su creación, como el propietario o la configuración de parámetros.

```
-- Cambiar el propietario de la base de datos
ALTER DATABASE universidad OWNER TO nuevo_usuario;

-- Cambiar la configuración de la base de datos
ALTER DATABASE universidad SET timezone TO 'Europe/Madrid';
```

9.1.4. Eliminación de bases de datos

La eliminación de una base de datos es irreversible y debe realizarse con precaución. Se puede usar el comando SQL o la herramienta `dropdb`.

```
-- Eliminar una base de datos
DROP DATABASE universidad;
```

```
# Eliminar una base de datos desde la terminal
dropdb -U exampleuser universidad
```

9.1.5. Mantenimiento y tareas administrativas

El mantenimiento incluye tareas como la reorganización de datos, actualización de estadísticas y limpieza de espacio no utilizado. PostgreSQL proporciona el comando `VACUUM` para estas tareas.

```
-- Limpiar y analizar la base de datos
VACUUM FULL;
ANALYZE;
```

Estas operaciones ayudan a mantener el rendimiento y la integridad de la base de datos.

9.1.6. Renombrar bases de datos

Si es necesario cambiar el nombre de una base de datos, se puede utilizar el siguiente comando:

```
ALTER DATABASE universidad RENAME TO universidad_nueva;
```

9.1.7. Consideraciones de seguridad

Es recomendable restringir el acceso a la creación y eliminación de bases de datos solo a usuarios con privilegios de superusuario o roles específicos, para evitar modificaciones accidentales o maliciosas.

En resumen, la gestión de bases de datos implica conocer los comandos y herramientas adecuados, aplicar buenas prácticas de seguridad y realizar tareas de mantenimiento periódicas para garantizar el correcto funcionamiento del sistema.

9.2. Gestión de usuarios y permisos

La gestión de usuarios y permisos es fundamental para controlar el acceso y las acciones que pueden realizar los distintos usuarios en una base de datos PostgreSQL. El administrador debe crear roles, asignar privilegios y garantizar que cada usuario tenga únicamente los permisos necesarios para sus tareas.

9.2.1. Creación de usuarios y roles

En PostgreSQL, los usuarios se gestionan como roles. Un rol puede tener permisos de inicio de sesión (`LOGIN`) y otros privilegios. Para crear un usuario:

```
-- Crear un usuario con contraseña
CREATE ROLE exampleuser WITH LOGIN PASSWORD 'contraseña_segura';
-- Crear un rol sin permiso de inicio de sesión (por ejemplo, ↪
  para agrupar permisos)
CREATE ROLE solo_lectura;
```

También se puede crear usuarios desde la terminal:

```
createuser -U postgres -P exampleuser
```

9.2.2. Asignación de permisos

Los permisos se asignan mediante los comandos `GRANT` y `REVOKE`. Se pueden conceder privilegios sobre bases de datos, esquemas, tablas, vistas y otros objetos.

```
-- Conceder acceso de conexión a una base de datos
GRANT CONNECT ON DATABASE universidad TO exampleuser;

-- Conceder permisos de lectura sobre una tabla
GRANT SELECT ON TABLE estudiantes TO exampleuser;

-- Conceder permisos de escritura
GRANT INSERT, UPDATE, DELETE ON TABLE estudiantes TO exampleuser
  ;

-- Revocar permisos
REVOKE DELETE ON TABLE estudiantes FROM exampleuser;
```

9.2.3. Roles y herencia

Los roles pueden agruparse y configurarse para heredar permisos. Por ejemplo, se puede crear un rol de solo lectura y asignarlo a varios usuarios:

```
-- Crear rol de solo lectura
CREATE ROLE solo_lectura;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO solo_lectura;

-- Asignar el rol a un usuario
GRANT solo_lectura TO exampleuser;
```

9.2.4. Cambio de contraseña y eliminación de usuarios

Para cambiar la contraseña de un usuario:

```
ALTER ROLE exampleuser WITH PASSWORD 'nueva_contraseña';
```

Para eliminar un usuario, primero asegúrate de que no tenga objetos propios ni conexiones activas:

```
DROP ROLE exampleuser;
```

9.2.5. Consideraciones de seguridad

Es recomendable:

- Utilizar contraseñas seguras y cambiarlas periódicamente.

- Asignar el mínimo privilegio necesario a cada usuario (principio de menor privilegio).
- Auditar los permisos y roles regularmente.
- Restringir el acceso de superusuario solo a administradores.

Una gestión adecuada de usuarios y permisos ayuda a proteger la base de datos frente a accesos no autorizados y acciones maliciosas, garantizando la integridad y confidencialidad de la información.

Ejercicio 9.1

Crea un rol llamado `estudiante` que tenga solo permiso de lectura en las siguientes tablas: `cursos`, `profesores` y `cursos_profesores`. Crea un usuario llamado `usuario_estudiante` y asígnale el rol.
Consejo: Al crear una base de datos, es importante definir claramente los roles y permisos desde el principio para evitar problemas de seguridad más adelante. Puedes mejorar la seguridad de tu base de datos si creas roles específicos para distintas aplicaciones.

9.3. Recuperación de fallos

La recuperación de fallos implica restaurar el funcionamiento de la base de datos tras un error, pérdida de datos o corrupción. PostgreSQL utiliza el sistema de archivos WAL (Write-Ahead Logging) para garantizar la durabilidad y permitir la recuperación ante fallos.

- **Restauración desde backup**: Utiliza copias de seguridad previas para recuperar el estado.
- **Point-in-Time Recovery (PITR)**: Permite restaurar la base de datos hasta un momento específico usando archivos WAL.

En este capítulo nos centraremos al uso de copias de seguridad, al ser una técnica común a la mayoría de SGBD.

9.4. Copias de seguridad

Las copias de seguridad son esenciales para proteger los datos ante pérdidas accidentales o fallos. Una buena política de copias de seguridad incluye la frecuencia, el tipo (completa, incremental, diferencial) y el almacenamiento seguro.
PostgreSQL ofrece varias herramientas:

pg_dump Realiza copias de seguridad lógicas de una base de datos.

pg_dumpall Copia todas las bases de datos del clúster.

pg_basebackup Realiza copias físicas para replicación y recuperación.

pg_restore Restaura bases de datos desde copias de seguridad.

A continuación se muestra un ejemplo de como ejecutar cada comando.

```
# Copia lógica de una base de datos
pg_dump -U exampleuser -F c -b -v -f backup.dump exampledb
# Copia de todas las bases de datos
pg_dumpall -U exampleuser > backup_global.sql
# Copia física para replicación
pg_basebackup -h localhost -D /ruta/backup -U replicador -P --wal-method=stream
# Restaurar una base de datos desde un backup
pg_restore -U anotheruser -d anotherdb backup.dump
```

Ejercicio 9.2

Realiza una copia de seguridad lógica de la base de datos llamada `universidad` utilizando el usuario `exampleuser`. Guarda el archivo de respaldo en la ruta `/backups/universidad.dump`. Luego, indica el comando para restaurar dicha copia en una base de datos vacía llamada `universidad_restaurada`.
Nota: tendrás que crear primero la base de datos, y su usuario asociado.

9.5. Importación y exportación de datos

La importación y exportación de datos es una tarea habitual en la administración de bases de datos, ya sea para migrar información, realizar respaldos parciales, cargar datos desde sistemas externos o compartir resultados. PostgreSQL proporciona varias herramientas y comandos para facilitar estos procesos:

9.5.1. Comando COPY

El comando `COPY` permite importar y exportar datos directamente entre una tabla y un archivo en el servidor. Es eficiente y soporta varios formatos, como texto, CSV y binario.

- Para exportar datos de una tabla a un archivo CSV:

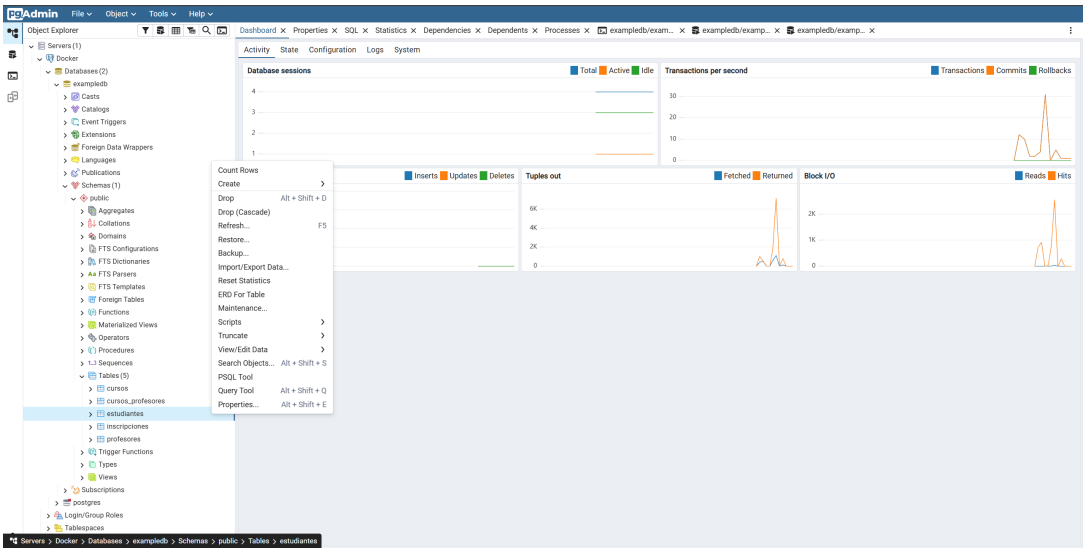
```
COPY estudiantes TO '/ruta/estudiantes.csv' WITH (FORMAT csv, HEADER);
```

- Para importar datos desde un archivo CSV a una tabla:

```
COPY estudiantes FROM '/ruta/estudiantes.csv' WITH (FORMAT csv, HEADER);
```

- Es posible usar la variante `COPY ... TO/FROM STDOUT/STDIN` para trabajar desde clientes remotos.

También podrás crear copias de seguridad, e importar y exportar datos, desde la interfaz gráfica de tu gestor.



Consideraciones

- Verifica que los formatos de archivo sean compatibles y que las rutas sean accesibles por el servidor.
- Para importar datos, la estructura de la tabla debe coincidir con el formato del archivo.
- Es recomendable validar los datos antes y después de la importación para evitar inconsistencias.
- Utiliza transacciones para asegurar la integridad durante procesos de importación masiva.

Ejercicio 9.3

Exporta los datos de la tabla `cursos` a un archivo CSV llamado `/tmp/cursos.csv`, incluyendo los encabezados. Edita el fichero csv, usado los datos para crear copias de los cursos para el año siguiente, cambiando el nombre y el código del curso. Finalmente, importa los datos modificados a la tabla `cursos`.
Nota: tendrás que eliminar la columna id, si es autonumérica, antes de importar los datos.

9.6. Transferencia entre sistemas gestores

La transferencia de datos entre diferentes SGBD requiere convertir el formato y adaptar la estructura. En PostgreSQL, se recomienda:

- Exportar datos en formato estándar (CSV, SQL).
- Adaptar tipos de datos y restricciones.
- Utilizar herramientas como `pgloader` para migraciones automáticas.

```
# Migrar datos desde MySQL a PostgreSQL
pgloader mysql://usuario:clave@localhost/basedatos postgresql://↪
  usuario@localhost/basedatos
```

9.7. Monitorización y optimización

La monitorización permite detectar problemas de rendimiento y anticipar fallos. PostgreSQL ofrece vistas y herramientas para analizar el estado del sistema:

- **pg_stat_activity**: Muestra las conexiones y consultas activas.
- **EXPLAIN**: Analiza el plan de ejecución de una consulta.
- **pgAdmin, Prometheus, Grafana**: Herramientas gráficas para monitorización avanzada.

```
-- Consultar actividad
SELECT * FROM pg_stat_activity;
-- Analizar una consulta
EXPLAIN ANALYZE SELECT * FROM estudiantes WHERE ciudad = 'Madrid↪
  ';
```

Ejercicio 9.4

Ejecuta la siguiente consulta en PgAdmin, y usa la opción de *Explain* para analizar el plan de ejecución y optimizar la consulta si es necesario. Observa como representa el plan de ejecución en la interfaz gráfica.

```
SELECT
  E.CIUDAD,
  C.NOMBRE_CURSO,
  COUNT(E.ID_ESTUDIANTE)
FROM
  ESTUDIANTES AS E
  LEFT JOIN INSCRIPCIONES AS I ON E.ID_ESTUDIANTE = I.
    ID_ESTUDIANTE
  RIGHT JOIN CURSOS AS C ON I.ID_CURSO = C.ID_CURSO
GROUP BY
  E.CIUDAD,
  C.NOMBRE_CURSO
ORDER BY
  E.CIUDAD,
  C.NOMBRE_CURSO;
```

9.8. Redundancia y alta disponibilidad

La redundancia y alta disponibilidad son fundamentales para sistemas críticos donde la pérdida de acceso a la base de datos puede tener consecuencias graves. En PostgreSQL, existen varias estrategias y herramientas para implementar estos conceptos:

9.8.1. Replicación

La replicación permite mantener una o más copias sincronizadas de la base de datos en diferentes servidores. Los principales tipos de replicación en PostgreSQL son:

- **Replicación física**: Copia los archivos de datos y WAL a servidores secundarios. Es ideal para alta disponibilidad y recuperación ante desastres. Se configura mediante parámetros en `postgresql.conf` (`wal_level`, `max_wal_senders`, `hot_standby`) y ajustes en `pg_hba.conf`.
- **Replicación lógica**: Permite replicar datos seleccionados (tablas, esquemas) entre servidores, útil para migraciones y sincronización parcial. Se gestiona con publicaciones y suscripciones (`CREATE PUBLICATION`, `CREATE SUBSCRIPTION`).

9.8.2. Failover y recuperación automática

El *failover* consiste en cambiar automáticamente a un servidor secundario si el principal falla. Herramientas como `repmgr`, `Patroni` o `pgpool-II` facilitan la detección de fallos y la conmutación automática, minimizando el tiempo de inactividad.

9.8.3. Clustering y balanceo de carga

El clustering implica el uso de varios nodos para distribuir la carga de trabajo y mejorar la escalabilidad. Soluciones como `pgpool-II` y `Citus` permiten balancear consultas entre réplicas y particionar datos para grandes volúmenes.

9.8.4. Buenas prácticas

- Realiza pruebas periódicas de conmutación y recuperación.
- Monitorea el estado de los nodos y la sincronización de la replicación.
- Mantén copias de seguridad independientes de la replicación.
- Documenta y automatiza los procedimientos de recuperación.

La correcta implementación de redundancia y alta disponibilidad garantiza la continuidad del servicio y la protección de los datos ante incidentes.

Resumen

En este capítulo hemos revisado los aspectos fundamentales de la administración de bases de datos en PostgreSQL, incluyendo la gestión de bases de datos y usuarios, la asignación de permisos, la realización de copias de seguridad y restauración, la importación y exportación de datos, la transferencia entre sistemas gestores, la monitorización y optimización, y las estrategias de redundancia y alta disponibilidad. Una administración adecuada garantiza la seguridad, integridad y disponibilidad de la información, así como el rendimiento y la escalabilidad del sistema. Aplicar buenas prácticas y utilizar las herramientas apropiadas es esencial para el correcto funcionamiento y protección de los datos en cualquier entorno profesional.

D

Contenidos exclusivos del módulo Bases de datos (0484)

10

Programación avanzada en bases de datos

La programación avanzada en bases de datos permite automatizar procesos, garantizar la integridad de los datos y mejorar el rendimiento mediante el uso de funciones, procedimientos, triggers y otros mecanismos internos. PostgreSQL ofrece un potente lenguaje de procedimientos (PL/pgSQL) para implementar lógica compleja directamente en el servidor.

10.1. Funciones y procedimientos almacenados

Las funciones y procedimientos almacenados son bloques de código que se ejecutan en el servidor de la base de datos. Permiten encapsular lógica, reutilizar código y mejorar la seguridad.

- **Funciones**: Devuelven un valor y pueden ser usadas en consultas.
- **Procedimientos**: Ejecutan acciones, pero no devuelven valores directamente.

10.1.1. Funciones

Las funciones en PL/pgSQL se definen utilizando la instrucción `CREATE FUNCTION`. Pueden recibir parámetros de entrada y devolver un valor. La sintaxis básica es la siguiente:

```
1 -- Función que devuelve el número de inscripciones de un ↪
    estudiante
2 CREATE OR REPLACE FUNCTION contar_inscripciones(est_id INT)
3 RETURNS INT AS $$
4 DECLARE
5  total INT;
6 BEGIN
```

```
 SELECT COUNT(*) INTO total FROM inscripciones WHERE id_estudiante = est_id;
 RETURN total;
END;
$$ LANGUAGE plpgsql;
```

Parámetros y tipos de retorno

Las funciones pueden aceptar múltiples parámetros de diferentes tipos de datos, y el tipo de retorno puede ser un valor escalar, un registro o incluso un conjunto de filas (setof). Por ejemplo:

```
-- Función que devuelve los cursos de un estudiante
CREATE OR REPLACE FUNCTION obtener_cursos(est_id INT)
RETURNS TABLE(id_curso INT, nombre_curso TEXT) AS $$
BEGIN
  RETURN QUERY
    SELECT c.id_curso, c.nombre_curso
    FROM cursos c
    JOIN inscripciones i ON c.id_curso = i.id_curso
    WHERE i.id_estudiante = est_id;
END;
$$ LANGUAGE plpgsql;
```

Funciones con lógica condicional

Las funciones pueden incluir lógica condicional, bucles y manejo de excepciones para realizar operaciones más complejas. Por ejemplo:

```
-- Función que actualiza la ciudad de un estudiante si existe
CREATE OR REPLACE FUNCTION actualizar_ciudad(est_id INT, nueva_ciudad TEXT)
RETURNS BOOLEAN AS $$
BEGIN
  UPDATE estudiantes SET ciudad = nueva_ciudad WHERE id_estudiante = est_id;
  IF FOUND THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

Ejercicio 10.1

Escribe una función en PL/pgSQL llamada `contar_cursos_ciudad` que reciba como parámetro el nombre de una ciudad y devuelva el número total de cursos en los que están inscritos los estudiantes de esa ciudad. Muestra la sintaxis y explica brevemente cómo funciona la función.

10.1.2. Procedimientos

Los procedimientos en PL/pgSQL se definen utilizando la instrucción `CREATE PROCEDURE`. A diferencia de las funciones, no devuelven un valor directamente. La sintaxis básica es la siguiente:

```
-- Procedimiento que inserta un nuevo curso
CREATE OR REPLACE PROCEDURE insertar_curso(nombre TEXT)
LANGUAGE plpgsql AS $$
BEGIN
 INSERT INTO cursos(nombre_curso) VALUES (nombre);
END;
$$;
```

Procedimientos con parámetros y transacciones

Los procedimientos pueden aceptar parámetros y ejecutar múltiples sentencias dentro de una transacción, lo que permite agrupar operaciones que deben realizarse de manera atómica. Por ejemplo, un procedimiento que inscribe a un estudiante en varios cursos:

```
-- Procedimiento para inscribir a un estudiante en varios cursos
CREATE OR REPLACE PROCEDURE inscribir_en_cursos(est_id INT, ↪
  cursos INT[])
LANGUAGE plpgsql AS $$
BEGIN
  FOREACH curso_id IN ARRAY cursos LOOP
    INSERT INTO inscripciones(id_estudiante, id_curso) VALUES (↪
  est_id, curso_id);
  END LOOP;
END;
$$;
```

Este procedimiento recibe el identificador del estudiante y un arreglo de identificadores de cursos, e inserta una inscripción para cada curso. El uso de bucles y parámetros permite automatizar tareas repetitivas y reducir errores.

Además, los procedimientos pueden utilizar el control de transacciones explícito para asegurar la consistencia de los datos:

```
-- Procedimiento que realiza varias operaciones en una transacción
CREATE OR REPLACE PROCEDURE actualizar_datos_estudiante(est_id INT, nueva_ciudad TEXT, cursos INT[])
LANGUAGE plpgsql AS $$
BEGIN
  BEGIN
    UPDATE estudiantes SET ciudad = nueva_ciudad WHERE id_estudiante = est_id;
    FOREACH curso_id IN ARRAY cursos LOOP
      INSERT INTO inscripciones(id_estudiante, id_curso) VALUES (est_id, curso_id);
    END LOOP;
    -- Si ocurre un error, se revierte toda la transacción
  EXCEPTION WHEN OTHERS THEN
    RAISE NOTICE 'Error al actualizar datos del estudiante.';
    ROLLBACK;
  END;
END;
$$;
```

De esta forma, los procedimientos almacenados permiten realizar operaciones complejas y seguras directamente en el servidor de la base de datos.

Ejercicio 10.2

Escribe un procedimiento en PL/pgSQL llamado `eliminar_inscripciones_estudiante` que reciba el identificador de un estudiante y elimine todas sus inscripciones en la tabla `inscripciones`. Muestra la sintaxis y explica brevemente cómo funciona el procedimiento.

10.2. Triggers y eventos

Los triggers (disparadores) son procedimientos que se ejecutan automáticamente ante ciertos eventos en la base de datos, como inserciones, actualizaciones o eliminaciones. Permiten mantener la integridad y realizar acciones automáticas.

Los triggers se definen mediante la instrucción `CREATE TRIGGER` y requieren una función especial que se ejecuta cuando ocurre el evento especificado. Los eventos pueden ser `INSERT`, `UPDATE` o `DELETE`, y el trigger puede ejecutarse antes (`BEFORE`) o después (`AFTER`) de la operación.

Por ejemplo, un trigger puede usarse para validar datos antes de insertar una fila, mantener registros históricos, o actualizar automáticamente otras tablas relacionadas. La función asociada al trigger debe tener el tipo de retorno `TRIGGER`

y puede acceder a los datos de la fila afectada mediante las variables `NEW` (para inserciones y actualizaciones) y `OLD` (para eliminaciones y actualizaciones).

La sintaxis básica para crear un trigger es:

```
CREATE OR REPLACE FUNCTION nombre_funcion_trigger()
RETURNS TRIGGER AS $$
BEGIN
  -- Lógica del trigger
  RETURN NEW; -- o RETURN OLD, según el caso
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER nombre_trigger
{BEFORE | AFTER} {INSERT | UPDATE | DELETE} ON nombre_tabla
FOR EACH ROW EXECUTE FUNCTION nombre_funcion_trigger();
```

Un ejemplo de trigger es el siguiente, en el cual se mantiene un registro de las inscripciones de los estudiantes:

```
-- Trigger que registra la fecha de inscripción

CREATE TABLE log_inscripciones (
 id SERIAL PRIMARY KEY,
 id_estudiante INT,
 fecha TIMESTAMP
);

CREATE OR REPLACE FUNCTION registrar_inscripcion()
RETURNS TRIGGER AS $$
BEGIN
 INSERT INTO log_inscripciones(id_estudiante, fecha)
 VALUES (NEW.id_estudiante, NOW());
 RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER inscripcion_trigger
AFTER INSERT ON inscripciones
FOR EACH ROW EXECUTE FUNCTION registrar_inscripcion();
```

En el siguiente ejemplo, podemos ver un trigger que evita la inserción de inscripciones duplicadas, realizando sus checkeos antes de que se inserte cualquier dato:

```
-- Trigger que evita inscripciones duplicadas usando BEFORE ↪
  INSERT

CREATE OR REPLACE FUNCTION evitar_inscripcion_duplicada()
RETURNS TRIGGER AS $$
BEGIN
```

```
  IF EXISTS (SELECT 1 FROM inscripciones WHERE id_estudiante = NEW.id_estudiante AND id_curso = NEW.id_curso) THEN
    RAISE EXCEPTION 'La inscripción ya existe.';
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER inscripcion_duplicada_trigger
BEFORE INSERT ON inscripciones
FOR EACH ROW EXECUTE FUNCTION evitar_inscripcion_duplicada();
```

Los triggers son herramientas poderosas para automatizar tareas y garantizar reglas de negocio directamente en la base de datos, evitando errores y mejorando la consistencia de los datos.

Ejercicio 10.3

Escribe un trigger en PL/pgSQL que, después de eliminar una inscripción de la tabla `inscripciones`, registre en una tabla llamada `log_bajas` el identificador del estudiante y la fecha de la eliminación. Muestra la sintaxis y explica brevemente cómo funciona el trigger.

Ejercicio 10.4

Escribe un trigger en PL/pgSQL que, evite inscripciones en un curso, cuando no haya ningún profesor asociado.

10.3. Excepciones

El manejo de excepciones permite controlar errores y situaciones inesperadas durante la ejecución de funciones y procedimientos. En PL/pgSQL se utiliza el bloque `EXCEPTION` para capturar y manejar estos errores, evitando que la ejecución falle abruptamente y permitiendo dar mensajes informativos o realizar acciones alternativas.

10.3.1. Sintaxis básica

El bloque `EXCEPTION` se coloca al final del bloque `BEGIN ... END` de una función o procedimiento. Dentro de este bloque se pueden especificar diferentes tipos de errores que se desean capturar, usando la cláusula `WHEN`.

```
-- Función que maneja excepciones al insertar un estudiante
```

```
CREATE OR REPLACE FUNCTION insertar_estudiante(nombre TEXT,
  ciudad TEXT)
RETURNS VOID AS $$
BEGIN
 INSERT INTO estudiantes(nombre, ciudad) VALUES (nombre, ciudad)
  ;
EXCEPTION WHEN unique_violation THEN
 RAISE NOTICE 'El estudiante ya existe.';
END;
$$ LANGUAGE plpgsql;
```

10.3.2. Captura de diferentes excepciones

Se pueden capturar distintos tipos de errores, como violaciones de clave única, errores de sintaxis, o cualquier otro error genérico.

```
-- Ejemplo capturando varios tipos de excepciones
CREATE OR REPLACE FUNCTION ejemplo_excepciones()
RETURNS VOID AS $$
BEGIN
 -- Intento de división por cero
 PERFORM 1 / 0;
EXCEPTION
 WHEN division_by_zero THEN
  RAISE NOTICE 'No se puede dividir por cero.';
 WHEN others THEN
  RAISE NOTICE 'Ocurrió un error inesperado.';
END;
$$ LANGUAGE plpgsql;
```

10.3.3. Buenas prácticas

- Utiliza excepciones para manejar errores previsibles, como violaciones de restricciones.
- Evita capturar todos los errores con `WHEN others` sin informar adecuadamente, ya que puede ocultar problemas importantes.
- Es recomendable registrar los errores o notificar al usuario cuando ocurre una excepción.

Ejemplo práctico

Supón que quieres insertar un estudiante y, si ya existe, actualizar su ciudad:

```
CREATE OR REPLACE FUNCTION upsert_estudiante(nombre TEXT, ciudad TEXT)
RETURNS VOID AS $$
BEGIN
 INSERT INTO estudiantes(nombre, ciudad) VALUES (nombre, ciudad);
EXCEPTION WHEN unique_violation THEN
 UPDATE estudiantes SET ciudad = ciudad WHERE nombre = nombre;
 RAISE NOTICE 'Estudiante actualizado.';
END;
$$ LANGUAGE plpgsql;
```

10.4. Cursores

Los cursores en PL/pgSQL son objetos que permiten recorrer los resultados de una consulta de manera secuencial, procesando cada fila individualmente. Son especialmente útiles cuando se necesita realizar operaciones complejas sobre cada registro, como cálculos, actualizaciones o validaciones, que no pueden resolverse fácilmente con una sola sentencia SQL.

10.4.1. Sintaxis básica de cursores

Para utilizar un cursor, se debe declararlo en la sección `DECLARE` de la función o procedimiento, abrirlo con `OPEN`, extraer filas con `FETCH`, y finalmente cerrarlo con `CLOSE`. El ciclo típico de uso es el siguiente:

```
DECLARE
  mi_cursor CURSOR FOR SELECT ...;
  mi_record RECORD;
BEGIN
  OPEN mi_cursor;
  LOOP
    FETCH mi_cursor INTO mi_record;
    EXIT WHEN NOT FOUND;
    -- Procesar la fila actual
  END LOOP;
  CLOSE mi_cursor;
END;
```

10.4.2. Cursores explícitos e implícitos

Cursores explícitos Se declaran y gestionan manualmente, como en el ejemplo anterior.

Cursores implícitos PL/pgSQL permite recorrer resultados directamente con la sentencia `FOR record IN SELECT ... LOOP`, sin necesidad de declarar un cursor explícito.

```
-- Cursor implícito usando FOR-IN-SELECT
FOR est_record IN SELECT id_estudiante, nombre FROM estudiantes
  LOOP
  RAISE NOTICE 'Estudiante: %', est_record.nombre;
END LOOP;
```

Ejemplo práctico

Supón que quieres actualizar la ciudad de todos los estudiantes que cumplen cierta condición, usando un cursor:

```
CREATE OR REPLACE FUNCTION actualizar_ciudad_estudiantes(
  ciudad_antigua TEXT, ciudad_nueva TEXT)
RETURNS VOID AS $$
DECLARE
  est_cursor CURSOR FOR SELECT id_estudiante FROM estudiantes
  WHERE ciudad = ciudad_antigua;
  est_record RECORD;
BEGIN
  OPEN est_cursor;
  LOOP
    FETCH est_cursor INTO est_record;
    EXIT WHEN NOT FOUND;
    UPDATE estudiantes SET ciudad = ciudad_nueva WHERE
  id_estudiante = est_record.id_estudiante;
  END LOOP;
  CLOSE est_cursor;
END;
$$ LANGUAGE plpgsql;
```

Esta función recorre todos los estudiantes de una ciudad específica y actualiza su ciudad a un nuevo valor.

10.4.3. Ventajas y consideraciones

- Los cursores permiten procesar grandes volúmenes de datos sin cargar todo el resultado en memoria.
- Son útiles para operaciones fila a fila, pero pueden ser menos eficientes que operaciones en bloque.
- Es importante cerrar los cursores para liberar recursos.

Ejercicio 10.5

Crea una función que utilice un cursor para recorrer todos los estudiantes y mostrar su nombre y ciudad.

Resumen

La programación avanzada en PostgreSQL permite implementar lógica compleja, automatizar tareas y mejorar la integridad y el rendimiento de la base de datos. El uso de funciones, procedimientos, triggers, manejo de excepciones y cursores es fundamental para el desarrollo profesional de aplicaciones basadas en bases de datos.

11

Bases de datos no relacionales: NoSQL

Las bases de datos no relacionales, conocidas como NoSQL, surgieron como respuesta a las limitaciones de los sistemas de bases de datos relacionales tradicionales frente a los nuevos desafíos de escalabilidad, flexibilidad y manejo de grandes volúmenes de datos no estructurados. NoSQL abarca una variedad de tecnologías y modelos de datos que permiten almacenar y procesar información de manera eficiente en aplicaciones modernas, como redes sociales, sistemas de recomendación y análisis de big data. Estas bases de datos se caracterizan por su capacidad para gestionar datos distribuidos, su alta disponibilidad y su facilidad para adaptarse a cambios en los requisitos de las aplicaciones.

11.1. Características de las bases de datos NoSQL

Las bases de datos NoSQL presentan una serie de características distintivas que las diferencian de los sistemas relacionales tradicionales:

Escalabilidad horizontal Permiten distribuir los datos y la carga de trabajo entre múltiples servidores, facilitando el crecimiento del sistema sin necesidad de adquirir hardware más potente.

Flexibilidad en el modelo de datos No requieren esquemas fijos, lo que posibilita almacenar datos semiestructurados o no estructurados, adaptándose fácilmente a cambios en los requisitos de la aplicación.

Alto rendimiento Están optimizadas para operaciones rápidas de lectura y escritura, incluso con grandes volúmenes de datos y alta concurrencia de usuarios.

Disponibilidad y tolerancia a fallos Utilizan mecanismos de replicación y particionamiento para asegurar la disponibilidad de los datos y la continuidad del servicio ante fallos de hardware o red.

Consistencia eventual Muchas bases de datos NoSQL priorizan la disponibilidad y la partición sobre la consistencia estricta, siguiendo el teorema CAP. Esto significa que los datos pueden no estar sincronizados en todo momento, pero eventualmente alcanzarán un estado consistente.

Soporte para grandes volúmenes de datos Son capaces de gestionar conjuntos de datos masivos, como los generados por aplicaciones web, dispositivos IoT o sistemas de análisis de datos.

Facilidad de integración Ofrecen APIs y drivers para diversos lenguajes de programación, facilitando su integración en aplicaciones modernas y arquitecturas basadas en microservicios.

Optimización para casos de uso específicos Existen diferentes tipos de bases de datos NoSQL (clave-valor, documento, columna, grafo), cada una optimizada para necesidades particulares de almacenamiento y consulta.

Estas características hacen que las bases de datos NoSQL sean una opción atractiva para aplicaciones que requieren alta escalabilidad, flexibilidad y rendimiento, especialmente en entornos distribuidos y de big data.

11.2. Tipos de bases de datos NoSQL

Las bases de datos NoSQL se clasifican en varios tipos, cada uno diseñado para resolver diferentes necesidades de almacenamiento y consulta de datos. Los principales tipos son:

11.2.1. Bases de datos clave-valor

Este tipo almacena los datos como pares de clave y valor. La clave es única y se utiliza para recuperar el valor asociado. Son ideales para aplicaciones que requieren búsquedas rápidas y almacenamiento sencillo, como cachés y sesiones de usuario. Ejemplos: Redis, Amazon DynamoDB.

11.2.2. Bases de datos de documentos

Almacenan datos en documentos semiestructurados, generalmente en formatos JSON, BSON o XML. Cada documento puede tener una estructura diferente, lo que proporciona gran flexibilidad. Son adecuadas para aplicaciones que manejan datos heterogéneos y requieren consultas complejas sobre los documentos. Ejemplos: MongoDB, CouchDB.

11.2.3. Bases de datos de columnas

Organizan los datos en columnas en lugar de filas, lo que permite almacenar grandes volúmenes de información y realizar consultas analíticas eficientes. Son utilizadas en sistemas de análisis de datos y big data, donde se requiere procesar rápidamente grandes conjuntos de datos. Ejemplos: Apache Cassandra, HBase.

11.2.4. Bases de datos de grafos

Están diseñadas para almacenar relaciones complejas entre entidades, representadas como nodos y aristas. Son ideales para aplicaciones que requieren modelar redes, como redes sociales, sistemas de recomendación y análisis de rutas. Ejemplos: Neo4j, Amazon Neptune.

11.2.5. Otros tipos

Existen otros enfoques especializados, como bases de datos orientadas a objetos, bases de datos de series temporales y bases de datos multivalentes, que resuelven necesidades específicas en ciertos dominios.

Cada tipo de base de datos NoSQL está optimizado para casos de uso particulares, por lo que la elección depende de los requisitos de la aplicación, el volumen y la naturaleza de los datos, y las operaciones que se deben realizar.

11.3. Elementos de las bases de datos NoSQL

Las bases de datos NoSQL, aunque varían según el tipo y el sistema gestor, comparten ciertos elementos fundamentales que permiten su funcionamiento y gestión eficiente. A continuación se describen los principales componentes y conceptos presentes en la mayoría de las bases de datos NoSQL:

Modelo de datos Cada tipo de base de datos NoSQL utiliza un modelo de datos específico:

- **Clave-valor:** Los datos se almacenan como pares clave-valor, donde la clave es única y el valor puede ser cualquier tipo de dato.
- **Documento:** Los datos se representan como documentos (por ejemplo, JSON o BSON), que pueden contener estructuras anidadas y campos flexibles.
- **Columna:** Los datos se organizan en familias de columnas, permitiendo almacenar grandes volúmenes y realizar consultas analíticas eficientes.

- **Grafo:** Los datos se modelan como nodos y aristas, facilitando la representación de relaciones complejas.

Identificadores únicos La mayoría de los sistemas NoSQL utilizan identificadores únicos para acceder y manipular los datos, ya sea claves primarias, IDs de documentos o identificadores de nodos.

Particionamiento (Sharding) Para lograr escalabilidad horizontal, los datos se dividen en fragmentos (shards) que se distribuyen entre varios servidores o nodos. Esto permite manejar grandes volúmenes de datos y mejorar el rendimiento.

Replicación Los sistemas NoSQL suelen replicar los datos en múltiples nodos para garantizar la disponibilidad y tolerancia a fallos. La replicación puede ser síncrona o asíncrona, dependiendo del sistema y los requisitos de consistencia.

Consistencia El nivel de consistencia puede variar según el sistema NoSQL. Algunos ofrecen consistencia eventual, mientras que otros permiten configuraciones de consistencia fuerte o personalizada, según las necesidades de la aplicación.

APIs y lenguajes de consulta Las bases de datos NoSQL proporcionan interfaces de programación (APIs) y lenguajes de consulta específicos para interactuar con los datos. Por ejemplo, MongoDB utiliza consultas basadas en JSON, mientras que Cassandra emplea CQL (Cassandra Query Language).

Índices Para mejorar el rendimiento de las consultas, muchos sistemas NoSQL permiten la creación de índices sobre ciertos campos o atributos, facilitando búsquedas rápidas y eficientes.

Gestión de transacciones Aunque tradicionalmente las bases de datos NoSQL no ofrecen soporte completo para transacciones ACID, algunos sistemas han incorporado mecanismos para garantizar la atomicidad y la integridad en operaciones críticas.

Seguridad Incluye mecanismos de autenticación, autorización y cifrado para proteger los datos y controlar el acceso de los usuarios.

Monitorización y administración Herramientas para supervisar el estado del sistema, gestionar nodos, realizar copias de seguridad y restauraciones, y ajustar parámetros de rendimiento.

Estos elementos permiten que las bases de datos NoSQL sean altamente adaptables y escalables, respondiendo a las demandas de aplicaciones modernas que requieren flexibilidad, rendimiento y disponibilidad en entornos distribuidos.

11.4. Sistemas gestores de bases de datos NoSQL

Los sistemas gestores de bases de datos NoSQL (NoSQL DBMS) son plataformas especializadas que implementan los distintos modelos de datos NoSQL y proporcionan las funcionalidades necesarias para almacenar, consultar y administrar grandes volúmenes de información de manera eficiente. A continuación se describen algunos de los sistemas gestores más representativos, sus características principales y casos de uso típicos:

11.4.1. Redis

Redis es un sistema gestor de base de datos clave-valor en memoria, conocido por su alta velocidad y eficiencia. Soporta estructuras de datos como cadenas, listas, conjuntos, hashes y más. Redis es ampliamente utilizado como sistema de caché, gestor de sesiones y cola de mensajes en aplicaciones web y sistemas distribuidos. Ofrece replicación, persistencia opcional y soporte para operaciones atómicas.

11.4.2. MongoDB

MongoDB es uno de los sistemas gestores de bases de datos de documentos más populares. Almacena los datos en documentos BSON (una extensión de JSON), permitiendo estructuras flexibles y consultas complejas. MongoDB soporta replicación, particionamiento automático (sharding), índices avanzados y agregaciones. Es ideal para aplicaciones que requieren flexibilidad en el esquema y manejo de datos heterogéneos, como sistemas de gestión de contenido y plataformas de comercio electrónico.

11.4.3. Apache Cassandra

Cassandra es un sistema gestor de base de datos orientado a columnas, diseñado para alta escalabilidad y disponibilidad en entornos distribuidos. Utiliza un modelo de consistencia eventual y permite la replicación de datos entre múltiples centros de datos. Cassandra es adecuado para aplicaciones que requieren procesamiento de grandes volúmenes de datos y alta tolerancia a fallos, como sistemas de análisis de logs, IoT y big data.

11.4.4. Neo4j

Neo4j es un sistema gestor de base de datos de grafos, optimizado para almacenar y consultar relaciones complejas entre entidades. Utiliza el lenguaje de consulta Cypher para explorar y analizar redes de nodos y aristas. Neo4j se emplea en aplicaciones como redes sociales, motores de recomendación, análisis de fraude y gestión de rutas.

11.4.5. Amazon DynamoDB

DynamoDB es un servicio de base de datos NoSQL gestionado por Amazon Web Services, basado en el modelo clave-valor y documento. Ofrece escalabilidad automática, replicación global, seguridad integrada y baja latencia. DynamoDB es utilizado en aplicaciones que requieren alta disponibilidad y rendimiento, como plataformas de comercio electrónico, juegos en línea y sistemas de seguimiento en tiempo real.

11.4.6. CouchDB

CouchDB es un sistema gestor de base de datos de documentos que utiliza JSON para almacenar datos y HTTP para la comunicación. Soporta replicación entre servidores y manejo de conflictos, lo que lo hace adecuado para aplicaciones distribuidas y móviles.

11.4.7. HBase

HBase es una base de datos orientada a columnas construida sobre el sistema de archivos distribuido Hadoop (HDFS). Está diseñada para manejar grandes volúmenes de datos y consultas analíticas en tiempo real. HBase es utilizada en aplicaciones de big data, análisis de series temporales y almacenamiento de datos históricos.

11.4.8. Características comunes de los sistemas gestores NoSQL

- **Escalabilidad horizontal:** Permiten añadir nodos para aumentar la capacidad y el rendimiento.
- **Alta disponibilidad:** Implementan replicación y tolerancia a fallos para garantizar el acceso continuo a los datos.

- **Modelos de consistencia flexibles:** Ofrecen opciones de consistencia eventual, fuerte o personalizada según las necesidades de la aplicación.
- **Interfaces de programación:** Proporcionan APIs y drivers para múltiples lenguajes, facilitando la integración en diferentes entornos.
- **Gestión distribuida:** Permiten la administración de datos y nodos en clústeres distribuidos.

La elección del sistema gestor de base de datos NoSQL depende de los requisitos específicos de la aplicación, el tipo de datos, el volumen de información y las necesidades de escalabilidad y rendimiento.

11.5. Herramientas de los sistemas gestores de bases de datos NoSQL

Los sistemas gestores de bases de datos NoSQL ofrecen una variedad de herramientas y utilidades que facilitan la administración, monitoreo, desarrollo y operación de los datos en entornos distribuidos y escalables. Estas herramientas pueden ser proporcionadas por el propio sistema gestor o por terceros, y suelen cubrir los siguientes aspectos:

11.5.1. Herramientas de administración y configuración

Permiten gestionar la instalación, configuración y mantenimiento de la base de datos. Incluyen utilidades para crear y modificar clústeres, gestionar nodos, ajustar parámetros de rendimiento, y realizar tareas de mantenimiento como compactación y limpieza de datos. Ejemplos:

MongoDB Compass Interfaz gráfica para administrar bases de datos MongoDB, visualizar documentos, crear índices y analizar el rendimiento.

RedisInsight Herramienta visual para monitorear y administrar instancias de Redis, explorar datos y analizar comandos.

Cassandra OpsCenter Plataforma para la administración y monitoreo de clústeres Cassandra, gestión de nodos y visualización de métricas.

11.5.2. Herramientas de monitoreo y diagnóstico

Estas herramientas permiten supervisar el estado del sistema, el rendimiento, la utilización de recursos y la salud de los nodos. Ofrecen alertas, gráficos y reportes para detectar cuellos de botella, fallos o anomalías. Ejemplos:

Prometheus y Grafana Integración común para recolectar métricas y visualizar el estado de bases de datos NoSQL en paneles personalizados.

Elasticsearch Kibana Para bases de datos orientadas a búsqueda y análisis, permite visualizar logs y métricas en tiempo real.

Herramientas nativas Muchos sistemas incluyen comandos o APIs para obtener estadísticas y reportes de uso.

11.5.3. Herramientas de respaldo y recuperación

Facilitan la creación de copias de seguridad (backups) y la restauración de datos en caso de pérdida o corrupción. Pueden ser manuales o automatizadas, y soportar diferentes niveles de granularidad (por base de datos, colección, tabla, etc.). Ejemplos:

mongodump/mongorestore Utilidades de línea de comandos para realizar backups y restauraciones en MongoDB.

Redis RDB/AOF Mecanismos de persistencia y recuperación de datos en Redis.

Cassandra nodetool snapshot Comando para crear instantáneas de datos en Cassandra.

11.5.4. Herramientas de migración y sincronización

Permiten transferir datos entre diferentes instancias, versiones o sistemas gestores, así como sincronizar datos en entornos distribuidos. Son útiles para actualizaciones, cambios de infraestructura o integración con otros sistemas. Ejemplos:

MongoDB Atlas Data Lake Permite integrar y migrar datos entre MongoDB y otros servicios en la nube.

Cassandra DataStax Bulk Loader Herramienta para importar y exportar grandes volúmenes de datos en Cassandra.

Herramientas de replicación Utilidades para configurar y gestionar la replicación entre nodos o centros de datos en sistemas NoSQL.

11.5.5. Herramientas de desarrollo y prueba

Incluyen clientes, bibliotecas y entornos de prueba para interactuar con la base de datos desde diferentes lenguajes de programación, así como simuladores y entornos de desarrollo local. Ejemplos:

Drivers oficiales APIs para Python, Java, Node.js, Go, etc., que permiten conectar y operar con la base de datos.

Shells interactivos Consolas como `mongo shell`, `cqlsh` (Cassandra), o `redis-cli` para ejecutar comandos y consultas.

Frameworks de prueba Herramientas para simular cargas, realizar pruebas de estrés y validar la integridad de los datos.

11.5.6. Herramientas de seguridad

Proporcionan mecanismos para gestionar usuarios, roles, permisos y cifrado de datos, así como auditoría de accesos y actividades. Ejemplos:

Gestión de roles y usuarios Interfaces para definir permisos y políticas de acceso.

Cifrado en reposo y en tránsito Herramientas para habilitar y administrar el cifrado de datos.

Auditoría Registro de operaciones y accesos para cumplir con normativas y detectar actividades sospechosas.

11.5.7. Integración con otras plataformas

Muchos sistemas gestores NoSQL ofrecen conectores y herramientas para integrarse con servicios en la nube, sistemas de análisis, motores de búsqueda y plataformas de big data, facilitando la interoperabilidad y el procesamiento avanzado de datos.

En resumen, las herramientas de los sistemas gestores de bases de datos NoSQL son fundamentales para garantizar la administración eficiente, la seguridad, el monitoreo y el desarrollo ágil de aplicaciones que requieren alta disponibilidad, escalabilidad y flexibilidad en el manejo de datos.

11.6. Caso práctico: Implementación de una base de datos NoSQL con MongoDB

En este caso práctico, se describen los pasos para implementar una base de datos NoSQL utilizando MongoDB, uno de los sistemas gestores de bases de datos de documentos más populares.

Para poder realizar este caso práctico, es necesario ejecutar el siguiente fichero de Docker Compose, el cual está disponible en el repositorio de este libro:

```yaml
name: ejemplos_nosql
services:
 mongo:
  image: mongo:latest
  restart: always
  ports:
   - "27017:27017"
  environment:
   MONGO_INITDB_ROOT_USERNAME: admin
   MONGO_INITDB_ROOT_PASSWORD: admin123

 mongo-express:
  image: mongo-express:latest
  ports:
   - "8082:8081"
  environment:
   ME_CONFIG_BASICAUTH_ENABLED: true
   ME_CONFIG_BASICAUTH_USERNAME: admin
   ME_CONFIG_BASICAUTH_PASSWORD: admin
   ME_CONFIG_MONGODB_URL: mongodb://admin:admin123@mongo:27017/
  depends_on:
   - mongo
```

Una vez ejecutado, accede a Mongo Express en la URL `http://localhost:8082`, y usa las credenciales definidas en el archivo de Docker Compose para iniciar sesión. Después, realiza los siguientes pasos, fijándote en las particularidades de MongoDB:

1. Crea una base de datos llamada `testdb` y una colección llamada `testcollection` utilizando la interfaz de Mongo Express.

2. Crea un documento en la colección `testcollection` con los siguientes campos:

```
{
  "_id": ObjectId(),
  "name": "John Doe",
  "age": 30,
  "email": "john.doe@example.com",
}
```

3. Inserta varios documentos adicionales en la colección con diferentes valores para los campos `name`, `age` y `email`.

4. Realiza una consulta para encontrar todos los documentos donde la edad (`age`) sea mayor a 25 años. Utiliza la siguiente query: `{ age: { $gt: 25 } }`

5. Actualiza el documento de `John Doe` para cambiar su correo electrónico a `john.new@example.com`.

6. Inserta un nuevo documento, pero en vez del campo `email`, utiliza el campo `web` con un valor de «`https://john.doe`».

7. Inserta el siguiente documento:

```
{
  "_id": ObjectId(),
  "name": "Jane Doe",
  "age": 27,
  "email": "jane.doe@example.com",
  "web": "https://jane.doe",
  "vehicles": [
    {
      "make": "Toyota",
      "model": "Camry",
      "year": 2020
    },
    {
      "make": "Honda",
      "model": "Civic",
      "year": 2019,
      "state": {
        "registered": true,
        "inspection": {
          "status": "valid",
          "expiry": "2023-12-31"
        }
      }
    }
  ]
}
```

En este ejercicio práctico, habrás podido observar cómo MongoDB maneja documentos con estructuras flexibles, permitiendo la inclusión de campos adicionales y anidados sin necesidad de definir un esquema rígido. Además, habrás experimentado con operaciones básicas como inserción, consulta y actualización de documentos en una base de datos NoSQL.

Resumen

En este capítulo se ha explorado el concepto de bases de datos NoSQL, sus características principales y los diferentes tipos existentes, como clave-valor, documento, columna y grafo. Se han detallado los elementos fundamentales que comparten estos sistemas, así como los sistemas gestores más representativos y las herramientas que facilitan su administración y operación. Finalmente, se ha

presentado un caso práctico con MongoDB para ilustrar la flexibilidad y potencia de las bases de datos NoSQL en aplicaciones modernas. La elección de una base de datos NoSQL adecuada depende de los requisitos específicos de cada proyecto, destacando su utilidad en escenarios que demandan escalabilidad, disponibilidad y manejo eficiente de datos no estructurados.