|  |
| --- |
| **Bases de Datos Relacionales (RDBMS)** |

**Introducción**

En la actualidad, los avances tecnológicos y la actividad diaria generan una enorme cantidad de datos. Esta producción masiva de información requiere de un enfoque estructurado para su organización y almacenamiento en entornos seguros, lo que permite su utilización y procesamiento posterior. Para llevar a cabo esta tarea compleja, se han desarrollado grandes almacenes de datos, comúnmente conocidos como bases de datos.

El **Sistema de Gestión de Bases de Datos (SGBD) o en ingles Data Base Management System (DBMS)** son softwares que se utilizan para definir, manipular, gestionar y controlar el acceso a las bases de datos. Su función principal es facilitar la interacción entre los usuarios y la base de datos para que puedan realizar operaciones como consultar, insertar, actualizar y eliminar datos, así como garantizar la integridad y seguridad de la información.

|  |
| --- |
|  |

,

|  |
| --- |
| **Primeras tareas que se pueden realizar con SQL** |
| **Definir esquemas de base de datos**  SQL permite definir la estructura de la base de datos, incluyendo tablas, columnas, tipos de datos, claves primarias y restricciones de integridad referencial. |
| **Inserción de datos**  Puedes utilizar SQL para agregar datos a las tablas de la base de datos utilizando la sentencia `INSERT`. |
| **Actualización y eliminación de datos**  SQL proporciona las sentencias `UPDATE` y `DELETE` para modificar y eliminar datos existentes en la base de datos. |
| **Recuperación de datos**  La sentencia `SELECT` se utiliza para recuperar datos de la base de datos. Puedes especificar condiciones, filtrar datos y ordenar resultados según tus necesidades. |
| **Creación y gestión de índices**  SQL permite crear índices en columnas específicas para mejorar el rendimiento de las consultas. |

**﻿Lenguaje SQL (Structured Query Language):**

Las bases de datos relacionales utilizan SQL para definir la estructura de las tablas, realizar consultas, modificar datos y administrar la base de datos.

SQL es un lenguaje poderoso que permite realizar operaciones como SELECT (para recuperar datos), INSERT (para agregar nuevos datos), UPDATE (para modificar datos existentes) y DELETE (para eliminar datos) de manera eficiente.

**Transacciones ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad):**

**Atomicidad:** Las transacciones se realizan en su totalidad o no se realizan en absoluto.

**Consistencia:** La base de datos debe estar en un estado consistente antes y después de una transacción.

**Aislamiento:** Las transacciones deben ser independientes entre sí y no deben interferir.

**Durabilidad:** Los cambios realizados en una transacción deben ser permanentes incluso después de un fallo del sistema.

**Normalización**

La normalización es un proceso utilizado en las bases de datos relacionales para organizar los datos de manera eficiente y reducir la redundancia.

Se logra dividiendo las tablas en varias tablas más pequeñas y relacionándolas entre sí mediante claves primarias y externas. Esto evita la duplicación innecesaria de datos y mantiene la integridad de los datos.

**Integridad Referencial**

Las bases de datos relacionales mantienen la integridad referencial mediante el uso de claves foráneas. Esto garantiza que no se puedan insertar datos que violen las relaciones definidas entre las tablas.

**Escalabilidad Vertical y Horizontal**

Las bases de datos relacionales pueden escalar verticalmente (añadiendo más recursos a una sola máquina) o horizontalmente (dividiendo los datos entre múltiples servidores) para satisfacer las demandas de aplicaciones de diferentes tamaños.

Las bases de datos relacionales se utilizan comúnmente en aplicaciones empresariales, sistemas de gestión de contenido, sistemas de seguimiento de pedidos, sistemas de contabilidad y cualquier aplicación donde se requiere un alto grado de estructura y relaciones complejas entre los datos.

Las bases de datos relacionales ofrecen una estructura sólida y coherente para almacenar y gestionar datos. Su capacidad para garantizar la integridad de los datos y admitir consultas SQL complejas las convierte en una elección sólida para una amplia variedad de aplicaciones comerciales y de misión crítica. Sin embargo, también es importante considerar las necesidades de escalabilidad y el rendimiento al seleccionar una base de datos para un proyecto específico.

|  |  |
| --- | --- |
| Creación de una Base de Datos | Para empezar, necesitas tener una base de datos que puedas usar. Puedes crear una nueva base de datos en tu sistema de gestión de bases de datos elegido utilizando la instrucción "CREATE DATABASE".(En sqlite3 es un archivo, si existe lo abre, sino al API lo crea automáticamente sin necesidad de create) |
| Creación de Tablas y Columnas | Luego, debes crear tablas para almacenar información en la base de datos. Esto se logra mediante la instrucción "CREATE TABLE". Debes definir los campos (columnas) de la tabla y especificar su tipo de datos (texto, número, fecha, etc.). |
| Inserción de Datos | Una vez que hayas creado las tablas y las columnas, puedes introducir datos y registros en la base de datos. Utiliza la instrucción "INSERT INTO", especificando la tabla y las columnas en las que deseas insertar los datos. |
| Consulta de Datos | La parte central de SQL es la consulta de datos. Para ello, emplea la instrucción "SELECT". Indica las tablas y las columnas que deseas consultar. También puedes aplicar condiciones utilizando "WHERE" para filtrar los datos que necesitas. |
| Actualización y Eliminación de Datos | Por último, puedes modificar y eliminar datos utilizando las instrucciones "UPDATE" y "DELETE". Estas operaciones te permiten mantener la integridad y precisión de la información almacenada en la base de datos. |

### ·

|  |
| --- |
| **Sintaxis de SQL** |
| SELECT: Se utiliza para recuperar datos de una tabla.  Por ejemplo:SELECT nombre, edad FROM clientes WHERE ciudad = 'Nueva York'; |
| INSERT: Agrega nuevos registros a una tabla.  Por ejemplo:INSERT INTO productos (nombre, precio) VALUES ('Portátil', 1000); |
| UPDATE: Actualiza registros existentes en una tabla.  Por ejemplo:UPDATE empleados SET salario = salario + 500 WHERE departamento = 'Ventas'; |
| DELETE: Elimina registros de una tabla.  Por ejemplo: DELETE FROM pedidos WHERE fecha < '2023-01-01'; |
| CREATE TABLE: Crea una nueva tabla en la base de datos.  Por ejemplo:CREATE TABLE libros (id INT PRIMARY KEY, titulo TEXT, autor TEXT); |
| ALTER TABLE: Modifica la estructura de una tabla existente.  Por ejemplo:ALTER TABLE clientes ADD COLUMN telefono VARCHAR(20); |
| CREATE DATABASE: Crea una nueva base de datos.  Por ejemplo:CREATE DATABASE mi\_nueva\_db; |

La seguridad es una consideración importante en SQL. Para prevenir la inyección SQL y otros ataques, se deben utilizar consultas parametrizadas y procedimientos almacenados en lugar de concatenar valores directamente en las consultas SQL.

**Algunos de los DBMS populares**

SQL es un lenguaje estándar, pero diferentes sistemas de gestión de bases de datos (DBMS) lo implementan de manera ligeramente diferente.

Cada uno de estos DBMS puede tener características y extensiones específicas, pero todos siguen el estándar SQL en su núcleo.

|  |  |
| --- | --- |
| **Bases de datos** | **Descripción** |
| **SQLite** | Fue lanzado en 2000 como una alternativa a MySQL. Es una base de datos SQL ligera y fácil de utilizar. Es muy popular en el desarrollo de aplicaciones móviles y para navegadores web. |
| ventajas | Es muy ligero y fácil de utilizar. No requiere instalación. También cuenta con soporte para un gran número de lenguajes de programación. |
| desventajas | SQLite no está diseñado para manejar grandes cantidades de datos. Puede ser limitado en cuanto a capacidades de escalabilidad y alta disponibilidad. |
| **MariaDB** | Creada en 2009 por Michael "Monty" Widenius, uno de los creadores originales de MySQL, en MySQL AB, una empresa sueca que fue adquirida por Sun Microsystems en 2008. Luego, en 2010, Sun Microsystems fue adquirida por Oracle Corporation. Para mantenerla fuera de las fauces de Oracle, se genero este folk en código abierto |
| ventajas | Es un proyecto de código abierto  Altamente compatibilidad con MySQL  Alto Rendimiento como el uso del motor de almacenamiento InnoDB, la gestión de subconsultas y la optimización de consultas.  Es escalable y puede gestionar grandes volúmenes de datos y cargas de trabajo de alto rendimiento.  Ofrece una variedad de motores de almacenamiento, InnoDB (default), Aria, TokuDB y más, para las diferentes necesidades. |
| desventajas | MariaDB es compatible en gran medida con MySQL, puede haber algunas diferencias y características adicionales que aun no están documentadas.  Menos Herramientas y Compatibilidad de Terceros(en relacion a MySQL).  Baja cuota de mercado en grandes empresas |
| **MySQL** | Fue lanzada en 1995, es una base de datos SQL de código abierto que ganó popularidad por su facilidad de uso y rendimiento. |
| ventajas | Es muy fácil de instalar y utilizar. Ofrece opciones de alta disponibilidad y escalabilidad. También cuenta con una gran comunidad de desarrolladores y una gran cantidad de recursos en línea. |
| desventajas | Las actualizaciones a veces pueden ser complicadas, y a pesar de ser de código abierto, algunas funciones avanzadas pueden requerir una licencia paga. |
| **Oracle** | Oracle Database es una base de datos SQL desarrollada por la empresa Oracle Corporation en 1977. Es especialista en grandes cantidades de datos y altas cargas de trabajo. |
| ventajas | Es muy escalable y ofrece grandes capacidades de seguridad, incluyendo opciones de copia de seguridad y recuperación ante desastres. También cuenta con soporte técnico de nivel empresarial. |
| desventajas | Es uno de los sistemas de base de datos más costosos del mercado. Muchas de las funciones avanzadas son difíciles de manejar para los principiantes. |
| **SQL Server** | Fue lanzado por Microsoft en 1989 y es el sistema de base de datos de la oferta de Microsoft. Está disponible en ediciones Express, Standard y Enterprise. |
| ventajas | Ofrece una integración nativa con otras herramientas de Microsoft como Excel, SharePoint, y Windows Server. También cuenta con opciones de alta disponibilidad y escalabilidad. |
| desventajas | La mayoría de las funciones avanzadas requieren una licencia de pago. No es nativa para otras plataformas que no sean Microsoft, lo que puede ser una limitación si se desea portabilidad. |
| **PostgreSQL** | Fue lanzado en 1989 como un proyecto universitario y se ha ido desarrollando desde entonces. Es una base de datos SQL de código abierto.  Es una ORDBMS orientado a objetos complejos |
| ventajas | Es muy segura y cuenta con opciones de escalabilidad y alta disponibilidad. También es conocida por su alta compatibilidad con los estándares SQL. |
| desventajas | La instalación y configuración inicial pueden ser complicadas. Algunas funciones avanzadas pueden requerir de trabajo adicional por parte del desarrollador. |

·

**SQL: Lenguaje de Consulta Estructurada**

SQL (Structured Query Language) o Lenguaje de Consulta Estructurada es uno de los lenguajes más importantes que deben dominarse en el ámbito de la programación. Es particularmente relevante para aquellos que se encuentran en proceso de formación en esta área. SQL ha sido diseñado específicamente para administrar y recuperar información de sistemas de gestión de bases de datos relacionales, tales como SQL Server, MySQL, Oracle, PostgreSQL e incluso Microsoft Access. Estos sistemas comparten un lenguaje estándar que facilita la realización de consultas, permitiendo así la recuperación eficiente de información y la modificación de los datos almacenados.

El lenguaje SQL es fundamental en el desarrollo de prácticamente cualquier programa que maneje datos, por lo que su aprendizaje se convierte en un componente esencial para quienes buscan profundizar en el campo de la programación y el manejo de información.

**Aprendizaje de SQL**

Aprender a crear bases de datos no es una tarea tan compleja, aunque sí requiere práctica y el desarrollo de habilidades lógicas para diseñar la estructura de una base de datos. Posteriormente, se debe implementar esta estructura en el gestor de base de datos elegido. En el contexto actual, no es imprescindible dominar todos los comandos de SQL a la perfección, ya que las herramientas de administración de bases de datos modernas ofrecen interfaces gráficas que permiten crear bases de datos sin la necesidad de escribir grandes volúmenes de código.

Sin embargo, a pesar de que podemos crear bases de datos sin recurrir al código, es altamente recomendable tener un conocimiento básico de SQL. Este conocimiento no solo facilitará la gestión de las bases de datos, sino que también permitirá resolver problemas de forma más eficaz y optimizar procesos de administración.

Ademas permite el poder hacer todas estas tareas desde Python.

**Justificaciones para Aprender SQL**

1. **Universalidad**: SQL es un lenguaje universal que se utiliza en una amplia variedad de sistemas. Las bases de datos se emplean en diversos sectores, desde finanzas hasta comercio electrónico, lo que convierte a SQL en un tema de gran relevancia que todos deberían conocer.
2. **Facilidad de Aprendizaje**: Adquirir los conocimientos básicos de SQL no es complicado. Es posible crear una base de datos con sus tablas en pocas horas. Si bien existen conceptos avanzados que requieren mayor estudio y dedicación, con la práctica constante se pueden dominar.
3. **Estabilidad y Evolución**: A lo largo de los años, aunque SQL ha evolucionado, sus fundamentos lógicos se han mantenido constantes. Esto lo convierte en un área de estudio valiosa, ya que su relevancia se ha sostenido en el tiempo.
4. **Valor en el Mercado Laboral**: Actualmente, las empresas valoran enormemente a los expertos en SQL, debido a que manejan grandes volúmenes de datos y requieren servicios que aseguren una adecuada administración y optimización de sus bases de datos. Los profesionales con habilidades en SQL a menudo disfrutan de buenas oportunidades laborales y salarios competitivos.
5. **Conocimiento como Poder**: Cuanto más sepa una persona sobre bases de datos y SQL, más oportunidades tendrá para destacarse en el ámbito profesional y contribuir de manera significativa a su organización.

**Modelo Relacional en Bases de Datos RDBMS- Relational Database Management Systems**

El modelo de bases de datos relacional es una herramienta poderosa y ampliamente utilizada para organizar y administrar datos. Este modelo es la base de muchas de las bases de datos más populares en el mundo, y su aplicación abarca desde sistemas de contabilidad hasta tiendas en línea. En esta sección, exploraremos los fundamentos de las bases de datos relacionales, incluyendo la arquitectura de sus estructuras de datos, los diferentes tipos de consultas que se pueden realizar y las formas en que este modelo contribuye a mejorar la eficiencia de las aplicaciones.

**Conceptos Fundamentales del Modelo Relacional**

El modelo relacional se basa en el concepto matemático de relación, que define cómo se estructuran los datos en una base de datos. Desarrollado por el Dr. E.F. Codd en 1970, este modelo se ha consolidado como el más popular y utilizado en la actualidad. Las bases de datos relacionales son cruciales para el funcionamiento de numerosas empresas y organizaciones en todo el mundo.

En términos básicos, una base de datos relacional es una colección de datos organizados en tablas, donde cada tabla contiene un conjunto de datos relacionados. Estas tablas están vinculadas entre sí a través de relaciones, lo que permite compartir datos de manera eficiente. Esta estructura simplifica la búsqueda, el acceso y la gestión de datos, favoreciendo un almacenamiento y recuperación de información más eficientes.

**Desarrollo y Reglas del Modelo Relacional**

El modelo de bases de datos relacional fue innovador porque introdujo un sistema de reglas que especifican cómo deben almacenarse y manipularse los datos. Este sistema es la base de la potencia y eficiencia del modelo relacional. La normalización es uno de los principios clave en el diseño de bases de datos relacionales, ya que busca reducir la redundancia y mejorar la integridad de los datos mediante la organización en tablas y la definición de relaciones.

**Características y Beneficios del Modelo Relacional**

Uno de los aspectos más destacados del modelo relacional es su capacidad para acceder a los datos de manera rápida y sencilla. Al estar organizados en tablas, los datos son fáciles de encontrar y manipular. La posibilidad de establecer relaciones entre tablas permite realizar operaciones complejas de manera eficiente.

Además, el modelo relacional ofrece ventajas significativas en términos de seguridad de los datos. Almacenar información en tablas separadas ayuda a proteger los datos contra amenazas externas y facilita la auditoría y el seguimiento de cambios, ya que se puede rastrear cada dato hasta su origen.

Otro beneficio notable del modelo relacional es su escalabilidad. A medida que la cantidad de datos aumenta, el sistema puede adaptarse para manejar grandes volúmenes de información. Esto resulta ideal para aplicaciones que requieren un alto rendimiento en el almacenamiento y la recuperación de datos.

Finalmente, el modelo relacional es particularmente adecuado para aplicaciones que demandan la manipulación de datos complejos. Gracias a sus capacidades avanzadas de consulta y su estructura relacional, es posible desarrollar consultas sofisticadas que pueden responder a preguntas complejas de manera rápida y precisa.

**Modelo Entidad-Relación (ER) en Bases de Datos**

El **modelo entidad-relación (ER)** es una de las herramientas más cruciales y ampliamente utilizadas en el diseño de bases de datos. Sirve para modelar y estructurar cómo los datos se organizan y se relacionan dentro de un sistema de información. Este modelo no solo ayuda a los diseñadores a visualizar de manera clara la arquitectura de la base de datos, sino que también mejora la comprensión del sistema, lo que permite crear soluciones más eficientes y escalables.

**Definición de Entidades y Relaciones**

Una Entidad-Relación (ER) es un concepto fundamental en el diseño de bases de datos, utilizado para representar de manera gráfica y estructurada las relaciones entre diferentes entidades dentro de un sistema. Este modelo fue propuesto por Peter Chen en 1976 y ha sido ampliamente utilizado para el desarrollo de bases de datos relacionales.

**ACID**

Las propiedades **ACID** aseguran que las transacciones en una base de datos sean fiables y que los datos no se corrompan o pierdan debido a errores en las operaciones o problemas del sistema. Las bases de datos que implementan ACID correctamente son más seguras y confiables para manejar datos críticos. 1. 1. **Atomicidad (Atomicity)**:

La atomicidad asegura que una transacción es **indivisible**; es decir, todas las operaciones que forman parte de la transacción se deben ejecutar por completo o, en caso de fallo, ninguna de ellas tendrá efecto. El principio detrás de la atomicidad es "todo o nada".

* **Ejemplo**: Si estás transfiriendo dinero entre dos cuentas bancarias, la operación debe incluir el retiro del dinero de la primera cuenta y el depósito en la segunda. Si ocurre un error antes de que se complete el depósito (como una pérdida de conexión), la operación de retiro también se deshace para que la base de datos no quede en un estado inconsistente.
* **Rollback**: Si ocurre un fallo en alguna operación dentro de una transacción, se ejecuta un "rollback", lo que deshace todos los cambios realizados por la transacción hasta ese punto, devolviendo la base de datos a su estado inicial. Esto evita que la base de datos quede en un estado incompleto o corrupto.

2. **Consistencia (Consistency)**:

La consistencia garantiza que la base de datos pase de un **estado válido** a otro estado válido después de una transacción. En otras palabras, después de que se ejecuta una transacción, todas las reglas y restricciones de la base de datos deben seguir siendo válidas (integridad referencial, restricciones de unicidad, triggers, etc.).

* **Ejemplo**: Si existe una regla que impide que una cuenta bancaria tenga un saldo negativo, al finalizar una transacción que involucre una transferencia, la consistencia asegura que ninguna cuenta tenga un saldo por debajo de cero. Si una operación viola esta regla, la transacción será rechazada y no se aplicará ningún cambio.
* **Restricciones**: Las bases de datos tienen reglas como claves primarias únicas, claves foráneas, valores no nulos, restricciones CHECK, entre otras. La consistencia asegura que todas estas reglas se cumplan incluso después de aplicar cambios mediante una transacción.

3. **Aislamiento (Isolation)**:

El aislamiento garantiza que **las transacciones concurrentes** no interfieran entre sí. Las transacciones en ejecución de manera simultánea deben dar la impresión de que se ejecutan en serie, una después de otra, aunque realmente se ejecuten en paralelo.

El grado de aislamiento puede variar según el nivel de concurrencia permitido en el sistema. Hay cuatro niveles comunes de aislamiento:

* **Read Uncommitted**: Permite que una transacción lea datos que aún no han sido confirmados por otra transacción. Esto puede llevar a **lecturas sucias**, es decir, lecturas de datos que podrían deshacerse si la transacción falla.
* **Read Committed**: Una transacción solo puede leer datos que ya han sido confirmados por otras transacciones. Esto evita las lecturas sucias.
* **Repeatable Read**: Una vez que una transacción lee un valor, ninguna otra transacción puede modificar ese valor hasta que la transacción original termine, evitando **lecturas no repetibles**.
* **Serializable**: El nivel más alto de aislamiento, donde todas las transacciones se ejecutan de manera que parecen completamente secuenciales. Esto garantiza la mayor integridad, pero también es el nivel que más puede afectar el rendimiento.
* **Ejemplo**: Si una transacción está actualizando los datos de un cliente, otra transacción concurrente no debe poder leer o modificar esos datos hasta que la primera transacción haya terminado. Esto evita que dos transacciones entren en conflicto y generen resultados incoherentes.

4. **Durabilidad (Durability)**:

La durabilidad garantiza que, una vez que una transacción se ha confirmado (commit), los cambios realizados son **permanentes** y se mantendrán incluso si ocurre una falla del sistema o un apagón eléctrico.

* **Ejemplo**: Si has hecho un depósito en una cuenta bancaria y la transacción ha sido confirmada, esos fondos deben estar disponibles en la cuenta incluso si hay un fallo en el sistema inmediatamente después. La durabilidad asegura que los datos confirmados no se perderán.
* **Persistencia**: Esta propiedad se garantiza al escribir los cambios en un medio de almacenamiento persistente, como un disco duro o un sistema de almacenamiento redundante, de modo que los datos no dependan únicamente de la memoria volátil (RAM).

Los niveles de aislamiento comunes son:

1. **Read Uncommitted**: Las transacciones pueden leer cambios no confirmados de otras transacciones, lo que puede llevar a lecturas sucias.
2. **Read Committed**: Las transacciones solo pueden leer cambios que han sido confirmados.
3. **Repeatable Read**: Asegura que si una transacción lee un valor, otras transacciones no pueden cambiar ese valor hasta que la transacción original termine.
4. **Serializable**: El nivel más alto de aislamiento, donde las transacciones se ejecutan de forma secuencial para evitar cualquier interferencia.

El nivel de aislamiento que se elija puede influir en el rendimiento y en el riesgo de problemas como lecturas sucias, lecturas no repetibles y fenómenos como fantasmas.

**Componentes Clave del Modelo Entidad-Relación  
Entidad:**

* **Una entidad es un objeto o concepto del mundo real que tiene existencia independiente y puede ser identificado de forma única.** Las entidades pueden ser físicas (como un empleado o un producto) o abstractas (como un proyecto o una reserva).
* Cada entidad se representa como un rectángulo en el diagrama ER.
* Ejemplos de entidades: Empleado, Cliente, Producto, Pedido.

**Atributo:**

* **Un atributo es una característica o propiedad de una entidad u objeto**. Los atributos describen las cualidades de la entidad.
* Los atributos se representan como elipses conectadas a sus respectivas entidades.
* Ejemplos de atributos: Para la entidad Empleado, los atributos podrían ser Nombre, ID de empleado, Fecha de contratación.

**Relación:**

* **Una relación describe cómo las entidades interactúan entre sí.** Representa una asociación entre dos o más entidades.
* Las relaciones se representan como rombos en el diagrama ER.
* Ejemplos de relaciones: Emplea (entre Empleado y Departamento), Compra (entre Cliente y Producto).

**Tipos de Relaciones**  
Las relaciones entre entidades pueden clasificarse en varias categorías:

1. **Uno a Uno (1:1):** Una entidad de un tipo está relacionada con una única entidad de otro tipo. Por ejemplo, un Empleado puede tener un Coche asignado.
2. **Uno a Muchos (1):** Una entidad de un tipo puede estar relacionada con múltiples entidades de otro tipo, pero no al revés. Por ejemplo, un Cliente puede realizar múltiples Pedidos.
3. **Muchos a Muchos (M):** Una entidad de un tipo puede estar relacionada con múltiples entidades de otro tipo y viceversa. Por ejemplo, un Estudiante puede inscribirse en múltiples Cursos, y un Curso puede tener múltiples Estudiantes.

|  |
| --- |
| **Ejemplo de Diagrama Entidad-Relación**  Imagina un sistema de gestión de una tienda.  **Un diagrama ER podría incluir:** **Entidades:**  Cliente Pedido Producto  **Atributos:**  Cliente: ID de cliente, Nombre, Email Pedido: ID de pedido, Fecha, Monto total Producto: ID de producto, Nombre del producto, Precio  **Relaciones:**  Un Cliente realiza Pedidos (1). Un Pedido contiene Productos (M). |

**Importancia del Modelo Entidad-Relación**

El modelo entidad-relación es crucial en el proceso de diseño de bases de datos porque:

* **Visualiza la Estructura:** Permite a los diseñadores visualizar cómo se organizan los datos y cómo se relacionan entre sí.
* **Facilita la Comunicación:** Ayuda a comunicar las necesidades del sistema a los desarrolladores y a otros interesados.
* **Base para la Implementación:** Sirve como base para la creación de un esquema de base de datos, que se implementará en un sistema de gestión de bases de datos relacional.

**Recordar:**

En el contexto del **modelo ER**, una **entidad** es cualquier objeto o concepto que tiene una existencia independiente dentro del sistema que se va a modelar. Las entidades pueden ser tangibles, como "Clientes", "Productos" o "Empleados", o más abstractas, como "Transacciones" o "Eventos". Cada entidad se describe mediante un conjunto de **atributos**, que son las características o propiedades de esa entidad. Por ejemplo, un "Cliente" podría tener atributos como su "Nombre", "Dirección", "Teléfono" y "Fecha de Registro".

Las **relaciones**, por otro lado, representan las interacciones entre las entidades. Estas pueden ser de varios tipos, como **uno a uno (1:1)**, **uno a muchos (1)** o **muchos a muchos (N)**. Un ejemplo típico es la relación entre "Clientes" y "Pedidos", donde un cliente puede realizar varios pedidos (relación 1). Las relaciones permiten capturar la forma en que las entidades interactúan en el sistema, facilitando la creación de modelos que reflejen con precisión la realidad que se está intentando replicar en la base de datos.

**Atributos y Claves Primarias**

Cada entidad en un modelo ER está compuesta por **atributos** que describen sus características. Los atributos pueden ser simples, compuestos, multivaluados o derivados. Un atributo simple puede ser el "Nombre" de un cliente, mientras que un atributo compuesto podría ser la "Dirección", que a su vez puede desglosarse en "Calle", "Número" y "Código Postal". Los **atributos derivados** son aquellos que no se almacenan directamente en la base de datos, sino que se calculan a partir de otros atributos, como la "Edad", que se puede derivar de la "Fecha de Nacimiento".

Una **clave primaria** (Primary Key) es un atributo o un conjunto de atributos que identifican de manera única cada registro dentro de una entidad. Por ejemplo, en una tabla de "Clientes", el atributo "ID Cliente" podría actuar como la clave primaria, asegurando que no haya dos clientes con el mismo identificador.

**Diagramas - Representación Gráfica del Modelo ER**

Una de las grandes ventajas del modelo ER es su capacidad para ser representado gráficamente mediante **diagramas entidad-relación (ERD)**.

**En estos diagramas:**

* Las **entidades** se representan como rectángulos.
* Los **atributos** de las entidades se representan como óvalos conectados a las entidades.
* Las **relaciones** entre entidades se muestran mediante rombos conectados a los rectángulos de las entidades.

Estos diagramas son una poderosa herramienta visual que facilita la comprensión de la estructura de la base de datos, tanto para los diseñadores técnicos como para los usuarios no técnicos. Al observar el diagrama, se puede ver rápidamente cómo se organizan los datos y cómo se relacionan las distintas entidades.

**Beneficios del Modelo ER**

El uso del modelo ER presenta varios beneficios clave:

1. **Claridad en la representación de datos**: Su estructura gráfica permite una representación clara de las entidades y sus relaciones, facilitando la visualización del diseño de la base de datos.
2. **Facilita la detección de problemas**: Al visualizar las relaciones entre entidades, es más fácil identificar posibles problemas de diseño, como redundancias, dependencias cíclicas o relaciones mal definidas, antes de implementar el sistema.
3. **Mejora la comunicación con usuarios no técnicos**: Dado que los diagramas ER son fáciles de entender, incluso para personas sin experiencia técnica, este modelo actúa como un puente entre los diseñadores de bases de datos y los usuarios finales.
4. **Optimización del rendimiento**: Al permitir un diseño estructurado desde el inicio, el modelo ER contribuye a la creación de bases de datos más eficientes en términos de espacio de almacenamiento y rendimiento, optimizando las consultas y las operaciones sobre los datos.

**Proceso de Diseño de un Modelo ER**

El diseño de un modelo ER sigue un proceso iterativo y estructurado que incluye los siguientes pasos:

1. **Identificación de Entidades**: Este es el primer paso en el que se definen los objetos o conceptos que deben ser representados en la base de datos. Es importante capturar todas las entidades necesarias para reflejar el sistema que se está modelando.
2. **Definición de Atributos**: En este paso se especifican los atributos que describen cada entidad, asegurando que se almacene toda la información relevante para cada entidad.
3. **Establecimiento de Relaciones**: Se determinan las interacciones entre las entidades, lo que permite modelar de manera adecuada cómo estas entidades interactúan en el sistema.
4. **Creación del Diagrama ER**: Se utiliza una representación gráfica para ilustrar cómo las entidades y sus relaciones se organizan dentro de la base de datos. Este diagrama servirá como una guía valiosa durante la fase de implementación.

**Tipos de Bases de Datos**

Una vez que se ha diseñado el modelo ER, es crucial comprender los tipos de bases de datos en los que se puede implementar este diseño. Algunas de las bases de datos más comunes incluyen:

* **Relacionales** (como MySQL, PostgreSQL y SQL Server), que organizan los datos en tablas y utilizan SQL para gestionar y consultar la información.
* **NoSQL** (como MongoDB o Cassandra), que son más flexibles en la estructura de los datos y se utilizan en escenarios que requieren alta escalabilidad y rendimiento.
* **Orientadas a objetos**, que permiten almacenar datos en formas más complejas, como objetos con métodos y propiedades.

**Ventajas de las Bases de Datos Relacionales (no es mi opinión )**

El modelo relacional, basado en el modelo ER, es uno de los más utilizados en la actualidad debido a sus múltiples ventajas:

* **Reducción de redundancia**: Al centralizar la información en tablas y normalizar los datos, se evita la duplicación innecesaria.
* **Mejora de la seguridad**: Los sistemas de gestión de bases de datos (DBMS- Data Base Manager System) implementan niveles de acceso y autenticación que permiten proteger los datos de accesos no autorizados.
* **Consistencia de datos**: Las bases de datos relacionales aseguran que las actualizaciones se reflejen en todas las vistas de los datos, minimizando el riesgo de errores.
* **Acceso eficiente**: Los lenguajes de consulta como SQL permiten acceder a los datos de manera rápida y eficiente, optimizando las consultas complejas.

**Conexión a DBMS - Data Base Manager System**

Para trabajar con bases de datos relacionales, es esencial conocer cómo conectarse a los sistemas de gestión de bases de datos más utilizados.

El modelo entidad-relación es una herramienta fundamental en el diseño de bases de datos, no solo por su capacidad de representar de manera clara la estructura de los datos, sino también por su facilidad para adaptarse a diferentes contextos y necesidades empresariales. Su implementación correcta garantiza la creación de sistemas de bases de datos robustos, escalables y fáciles de mantener.

Ejemplos de cómo crear tablas en cuatro sistemas de gestión de bases de datos diferentes (SQLite, MySQL, SQL Server y PostgreSQL). Aunque las sintaxis pueden variar ligeramente, la funcionalidad subyacente es similar en todos ellos.  
Crear una Conexión y una Base de datos ‘mi\_base\_de\_datos’ en SQL.

|  |
| --- |
| **SQLite**  import sqlite3 # Conectar o crear la base de datos si no existe conexion = sqlite3.connect ('mi\_base\_de\_datos.db')# Nos conectamos a la base de datos si existe, sino la crea en el momento # Crear un cursor  cursor = conexion.cursor()  #-------------Desarrollo-------------------- # (Aquí se pueden realizar operaciones en la bbdd)  # Cerrar el cursor y la conexión cursor.close() conexion.close() |
| **Nota:** SQLite no requiere servidor, ya que la base de datos se almacena en un archivo. La conexión se realiza directamente a este archivo. No es necesario crear una base de datos como en otros DBMS, porque el archivo de la base de datos se crea al conectarse.  El archivo de base de datos se crea automáticamente al conectarse si no existe. |

.

|  |
| --- |
| **MySQL**  import mysql.connector # Conectarse al servidor MySQL  conexion = mysql.connector.connect(  host='localhost', user='tu\_usuario', password='tu\_contraseña' )  # Crear un cursor  cursor = conexion.cursor() # Crear una base de datos cursor.execute("CREATE DATABASE mi\_base\_de\_datos") # Cerrar el cursor y la conexión cursor.close() conexion.close()  #-------------Reconexión a la nueva base de datos------------------- conexion = mysql.connector.connect(  host='localhost', user='tu\_usuario', password='tu\_contraseña',  database='mi\_base\_de\_datos' Nos conectamos a la bbdd recién creada  )  # Crear un cursor  cursor = conexion.cursor()  #-------------Desarrollo-------------------- # (Aquí se pueden realizar operaciones en la bbdd)  # Cerrar el cursor y la conexión cursor.close() conexion.close() |
| **Nota:** Cuando se crea una nueva base de datos en **MySQL**, uno se conecta al DBMS sin seleccionar base de datos. Luego se cierra la conexión y recrea una nueva directo a la base de datos |

.

|  |
| --- |
| **SQL Server**  import pyodbc# Conectarse al servidor SQL Server conexion = pyodbc.connect(  'DRIVER={ODBC Driver 17 for SQL Server};' 'SERVER=localhost;' 'DATABASE=master;' # La base de datos "master" por defecto 'UID=usuario;' 'PWD=contraseña' )  # Crear un cursor cursor = conexion.cursor() # Crear una nueva base de datos cursor.execute("CREATE DATABASE mi\_base\_de\_datos") # Cerrar el cursor y la conexión cursor.close() conexion.close()  #-------------Reconexión a la nueva base de datos-------------------  conexion = pyodbc.connect(  'DRIVER={ODBC Driver 17 for SQL Server};' 'SERVER=localhost;' 'DATABASE=mi\_base\_de\_datos;'# Nos conectamos a la bbdd recién creada  'UID=usuario;' 'PWD=contraseña' )  # Crear un cursor cursor = conexion.cursor() #-------------Desarrollo-------------------- # (Aquí se pueden realizar operaciones en la bbdd)  # Cerrar el cursor y la conexión cursor.close() conexion.close() disponible y contiene metadatos críticos. |
| **Nota:** Cuando se crea una nueva base de datos en SQL Server, la base de datos master es el lugar donde se almacena la información del sistema. master contiene todos los detalles sobre bases de datos, configuraciones y recursos del servidor SQL Server. Aunque podrías conectarte a otra base de datos existente, master es una opción común, ya que siempre está. |

.

|  |
| --- |
| **PostgreSQL**  import psycopg2# Conectarse al servidor PostgreSQL from psycopg2 import sql conexion = psycopg2.connect(  dbname='postgres', # Conectar a la base de datos por defecto user='tu\_usuario', password='tu\_contraseña', host='localhost' # O la dirección IP del servidor )  conexion.autocommit = True # Permitir crear la base de datos fuera de una transacción  # Crear un cursor cursor = conexion.cursor() # Crear una base de datos nueva cursor.execute(sql.SQL("CREATE DATABASE mi\_base\_de\_datos")) # Cerrar el cursor y la conexión cursor.close() conexion.close()  #-------------Reconexión a la nueva base de datos-------------------  # Conectar ahora a la nueva base de datos conexion = psycopg2.connect(  dbname='mi\_base\_de\_datos', # Nos conectamos a la bbdd recién creada user='tu\_usuario', password='tu\_contraseña', host='localhost' # O la dirección IP del servidor )  # Crear un cursor cursor = conexion.cursor()  #-------------Desarrollo-------------------- # (Aquí se pueden realizar operaciones en la bbdd)  cursor.close() conexion.close() |
| **Nota:** PostgreSQL requiere que te conectes a alguna base de datos para ejecutar cualquier comando SQL. No puedes conectarte al servidor sin especificar una base de datos, por lo que te conectas a una base existente. postgres es una base de datos por defecto que siempre está disponible para este propósito. En PostgreSQL, no existe el comando USE para cambiar el contexto de la base de datos. |

.

|  |
| --- |
| **IF NOT EXISTS** |
| La cláusula IF NOT EXISTS es una extensión de SQL que se usa al crear bases de datos o tablas para evitar errores si ya existe una base de datos o tabla con el mismo nombre. Esta cláusula permite que la operación de creación sea opcional y no cause una excepción si el objeto ya existe. |
| * Etapas   1. **Conexión inicial.**   2. Crear la base de datos.   3. **Desconexión (inicial)**   4. **Conexión a la base de datos recién creada.** * Reconectar a la base de datos es una **buena práctica** para garantizar que todas las operaciones se ejecuten en el contexto correcto y para evitar errores si intentas usar una base de datos que la conexión original no tenía configurada. * Reconectar garantiza que las **nuevas consultas** se ejecutan en la base de datos correcta, ya que la librería asocia todas las operaciones de SQL a la base de datos definida en el momento de la conexión. |

**.**

### **Tipos de datos mas comunes**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Tipo de dato** | **Descripción** | **SQLite3** | **MySQL** | **SQL**  **Server** | **Postgre**  **SQL** |
| VARCHAR(n) | Cadena de texto de longitud variable | si | si | si | si |
| TEXT | Cadena de texto de longitud variable | si | si | si | si |
| INTEGER | Número entero | si | si | si | si |
| FLOAT | Número decimal de precisión simple | si | si | si | si |
| DOUBLE | Número decimal de doble precisión | no | si | sí(3) | si |
| DECIMAL(p,s) | Número decimal de precisión variable | no | si | sí | si |
| BOOLEAN | Valor booleano verdadero o falso | si | si | sí(4) | si |
| DATE | Fecha en formato 'AAAA-MM-DD' | si | si | sí | si |
| TIME | Hora en formato 'HH:MM:SS' | si | si | sí | si |
| DATETIME | Fecha y hora en formato 'AAAA-MM-DD HH:MM:SS | si | si | sí | si |
| BLOB | Datos binarios de longitud variable | si | si | sí | si |
| JSON | Almacena datos estructurados en formato JSON | si | si | no(5) | si |
| ARRAY | Almacena datos de una columna en formato de arreglo | no | si(1) | no | si |
| ENUM | Almacena un conjunto de valores predeterminados | no | si(2) | no(6) | si |

### Notas:

1. En MySQL el almacenamiento de datos tipo ARRAY se logra utilizando las columnas de tipo TEXT o VARCHAR y separando los valores por comas o algún otro carácter. Se utiliza principalmente en el contexto de almacenamiento y recuperación de valores JSON.
2. En MySQL, el almacenamiento de datos tipo ENUM se logra utilizando una lista de valores predeterminados separados por comas y utilizando una columna de tipo VARCHAR.
3. **DOUBLE** en SQL Server está representado por FLOAT con una mayor precisión de bits.
4. En **SQL Server**, el tipo **BOOLEAN** no existe directamente, pero se puede simular con el tipo **BIT** (donde 0 es falso y 1 es verdadero).
5. **JSON** en SQL Server no es un tipo de dato nativo, pero admite funciones para almacenar y manipular datos JSON en columnas tipo **NVARCHAR**.
6. SQL Server no admite **ENUM** como un tipo de dato nativo. En su lugar, se suele utilizar una tabla de referencia para almacenar valores predefinidos o se emplea el tipo **CHECK** para validar valores.

Es importante destacar que, aunque los tipos de datos básicos son similares en la mayoría de las bases de datos, pueden haber diferencias en la precisión, el almacenamiento, la longitud y otras propiedades. Además, cada motor de búsqueda puede tener diferentes tipos de datos avanzados y personalizados.

|  |
| --- |
| **SQLite**:  CREATE TABLE Clientes (  id INTEGER PRIMARY KEY AUTOINCREMENT,  nombre VARCHAR(255) NOT NULL,  direccion VARCHAR(255),  telefono VARCHAR(20)  );  **Notas:**   * AUTOINCREMENT se usa para generar valores únicos en la columna id. * El tipo INTEGER se usa para la clave primaria, y VARCHAR para las cadenas de texto. |
| **MySQL**:  CREATE TABLE clientes (  id INT AUTO\_INCREMENT PRIMARY KEY,  nombre VARCHAR(255) NOT NULL,  direccion VARCHAR(255),  telefono VARCHAR(20)  );  **Notas:**   * AUTO\_INCREMENT define la columna id como autoincremental. * MySQL usa INT en lugar de INTEGER. |
| **SQL Server**:  CREATE TABLE Clientes (  Id INT IDENTITY(1,1) PRIMARY KEY,  Nombre NVARCHAR(255) NOT NULL,  Direccion NVARCHAR(255),  Telefono NVARCHAR(20)  );  **Notas:**   * IDENTITY(1,1) define un valor inicial de 1 y un incremento de 1 para la columna Id. * NVARCHAR se usa para cadenas de texto, lo que permite almacenar caracteres Unicode. |
| **PostgreSQL**:  CREATE TABLE Clientes (  id SERIAL PRIMARY KEY,  nombre VARCHAR(255) NOT NULL,  direccion VARCHAR(255),  telefono VARCHAR(20)  );  **Notas:**   * SERIAL es un tipo especial que define una columna autoincremental. |

**Llaves Primarias y Foráneas: Integridad y Manipulación de Datos**

**Ejemplos de creación de Claves Primarias en cada DBMS:**

|  |
| --- |
| **Claves Primarias:**  Una clave primaria es un identificador único para cada registro en una tabla. Tiene las siguientes propiedades:   * **Unicidad:** Cada registro debe tener un valor único en la columna de la clave primaria. * **No Nulo:** No puede contener valores nulos. |
| **SQLite**  CREATE TABLE Clientes (id INTEGER PRIMARY KEY AUTOINCREMENT,...); |
| **MySQL**  CREATE TABLE clientes ( id INT AUTO\_INCREMENT PRIMARY KEY,...); |
| **SQL Server**  CREATE TABLE Clientes ( Id INT IDENTITY(1,1) PRIMARY KEY,...); |
| **PostgreSQL**  CREATE TABLE Clientes ( id SERIAL PRIMARY KEY,...); |

**.**

**Ejemplos de creación de Claves Foráneas en cada DBMS:**

|  |
| --- |
| **Claves Foráneas:**  Una clave foránea establece una relación entre dos tablas y asegura la integridad referencial al garantizar que los valores de una tabla secundaria (foránea) correspondan a la clave primaria de otra tabla. |
| **SQLite**  CREATE TABLE Pedidos (  id INTEGER PRIMARY KEY AUTOINCREMENT,  id\_cliente INTEGER,  FOREIGN KEY (id\_cliente) REFERENCES Clientes(id)  ); |
| **MySQL**  CREATE TABLE pedidos (  id INT AUTO\_INCREMENT PRIMARY KEY,  id\_cliente INT,  FOREIGN KEY (id\_cliente) REFERENCES clientes(id)  ); |
| **SQL Server**  CREATE TABLE Pedidos (  Id INT IDENTITY(1,1) PRIMARY KEY,  Id\_Cliente INT,  FOREIGN KEY (Id\_Cliente) REFERENCES Clientes(Id)  ); |
| **PostgreSQL**  CREATE TABLE Pedidos (  id SERIAL PRIMARY KEY,  id\_cliente INT,  FOREIGN KEY (id\_cliente) REFERENCES Clientes(id)  ); |

**Importancia de las Claves Foráneas**

Las claves foráneas aseguran la **integridad referencial**, un aspecto crucial en el diseño de bases de datos.

1. **Relaciones Lógicas Claras:** Facilitan la organización de los datos mediante la creación de relaciones entre tablas.
2. **Integridad de los Datos:** Impiden la inserción de datos inconsistentes.
3. **Optimización del Rendimiento:** Al reducir la duplicación de datos, contribuyen a un diseño más eficiente y mejor rendimiento.

**Tipos de Integridad Referencial**

1. **Integridad Referencial Declarativa:** Se implementa mediante restricciones como PRIMARY KEY y FOREIGN KEY. El DBMS valida automáticamente la coherencia de los datos.
2. **Integridad Referencial Procedural:** Usa **triggers** y **procedimientos almacenados** para verificar la integridad cuando ocurren eventos específicos como inserciones o actualizaciones.

A pesar de las diferencias en la sintaxis, los DBMS como SQLite, MySQL, SQL Server y PostgreSQL ofrecen capacidades similares para garantizar la integridad de los datos a través de la definición de claves primarias y foráneas. Estas herramientas son fundamentales para crear bases de datos coherentes y optimizadas, manteniendo la consistencia de la información almacenada.

**Importancia de la Integridad Referencial**

La integridad referencial es crucial en el diseño de bases de datos relacionales porque asegura que las relaciones entre las tablas se mantengan coherentes y válidas a lo largo del tiempo. Algunos de los principales beneficios de la integridad referencial son:

1. **Garantía de Consistencia**: Al definir relaciones lógicas entre tablas, la integridad referencial impide la inserción de datos que no tengan sentido o que estén incompletos. Por ejemplo, al asegurarse de que cada valor en una clave foránea corresponda a un valor existente en una clave primaria, se evita que existan registros "huérfanos".
2. **Reducción de Redundancia de Datos**: Las claves foráneas ayudan a evitar la duplicación de información, ya que los datos relacionados se almacenan en tablas separadas, referenciadas entre sí. Esto optimiza el uso del espacio y reduce la posibilidad de errores.
3. **Protección Contra Errores de Integridad**: La implementación de restricciones de integridad referencial asegura que los datos permanezcan coherentes durante inserciones, actualizaciones o eliminaciones. Si un dato clave es eliminado o modificado en una tabla principal, la integridad referencial evitará que la base de datos quede en un estado inconsistente.
4. **Facilita la Escalabilidad**: En sistemas que manejan grandes volúmenes de datos, la integridad referencial ayuda a mantener la calidad y precisión de la información, independientemente de la cantidad de datos que se manejen.

El diseño de una base de datos robusta va más allá del simple almacenamiento de información. Debe garantizar la integridad y coherencia de los datos a lo largo del tiempo mediante el uso de herramientas como claves primarias y foráneas. La **integridad referencial** asegura que las relaciones entre las tablas sean válidas y coherentes, manteniendo la calidad de los datos y facilitando la manipulación correcta de grandes volúmenes de información.

|  |
| --- |
|  |

**Concepto de Consulta de Base de Datos**

Una **consulta de base de datos** es un conjunto de instrucciones diseñadas para interactuar con la base de datos y obtener información específica basada en condiciones definidas. Este mecanismo permite a los usuarios o aplicaciones recuperar, modificar o analizar los datos almacenados de manera eficiente. Las consultas se ejecutan utilizando el **Lenguaje de Consulta Estructurado (SQL)**, el cual es un estándar en la gestión de bases de datos relacionales.

|  |
| --- |
|  |

**Operaciones Básicas en SQL**

Los DBMS como **SQLite**, **MySQL**, **SQL Server**, y **PostgreSQL** ofrecen mecanismos similares para implementar y garantizar la integridad referencial, lo que permite una gestión eficaz y confiable de los datos en cualquier plataforma.

|  |
| --- |
| Las consultas SQL permiten realizar diversas operaciones con los datos almacenados en las tablas de una base de datos. |
| **SELECT**: Recupera datos específicos de una o varias tablas.  En este caso, la consulta selecciona los campos nombre y telefono del cliente con id = 1.  SELECT nombre, telefono FROM clientes WHERE id = 1; |
| **INSERT**: Inserta nuevos registros en una tabla.  INSERT INTO clientes (nombre, direccion, telefono) VALUES ('Ariel', 'Sin calle 123', '4321-1234'); |
| **UPDATE**: Actualiza registros existentes en una tabla.  UPDATE clientes SET telefono = '4321-5678' WHERE id = 1; |
| **DELETE**: Elimina registros específicos de una tabla.  DELETE FROM clientes WHERE id = 1; |
| **TRUNCATE**: Elimina todos los registros de una tabla de manera rápida.  TRUNCATE TABLE clientes; |

**Cláusulas Comunes en las Consultas SQL**

|  |
| --- |
| Las consultas SQL suelen usar **cláusulas** para restringir o modificar el conjunto de resultados que devuelven. |
| **FROM**: Especifica la tabla desde la cual se extraerán los datos.  SELECT \* FROM clientes; |
| **WHERE**: Define las condiciones que deben cumplir los registros seleccionados.  SELECT nombre FROM clientes WHERE telefono = '4321-1234'; |
| **ORDER BY**: Ordena los resultados según los campos seleccionados, ya sea de manera ascendente (ASC) o descendente (DESC).  SELECT nombre FROM clientes ORDER BY nombre ASC; |
| **GROUP BY**: Agrupa los registros con valores iguales en las columnas especificadas.  SELECT COUNT(id), direccion FROM clientes GROUP BY direccion; |
| **HAVING**: Filtra los resultados agrupados después de aplicar la cláusula GROUP BY.  SELECT direccion, COUNT(id) FROM clientes GROUP BY direccion HAVING COUNT(id) > 1; |

.

|  |
| --- |
| SQL ofrece una serie de funciones que permiten realizar cálculos sobre un conjunto de registros. |
| **AVG**: Calcula el promedio de un conjunto de valores.  SELECT AVG(edad) FROM clientes; |
| **COUNT**: Devuelve el número total de registros en una consulta.  SELECT COUNT(\*) FROM clientes; |
| **SUM**: Calcula la suma de los valores de una columna.  SELECT SUM(saldo) FROM cuentas; |
| **MAX**: Devuelve el valor máximo en una columna.  SELECT MAX(saldo) FROM cuentas; |
| **MIN**: Devuelve el valor mínimo en una columna.  SELECT MIN(saldo) FROM cuentas; |

Las consultas de bases de datos son una herramienta esencial para la manipulación y recuperación de datos. Utilizando SQL, se pueden realizar diversas operaciones que van desde la selección y modificación de registros hasta la agregación de datos. Las cláusulas como WHERE, ORDER BY, y GROUP BY permiten crear consultas dinámicas y específicas, mientras que las funciones de agregación facilitan el análisis de grandes volúmenes de información. Esto convierte a las consultas SQL en un pilar fundamental para el manejo eficiente de bases de datos relacionales.

**Tipos de Operadores Lógicos:**

Es importante utilizar correctamente los operadores lógicos para obtener los resultados esperados en las consultas, ya que un uso inadecuado de ellos puede generar resultados inesperados. Cada opción de operadores lógicos tiene su propia sintaxis y, dependiendo de la base de datos utilizada, pueden haber diferencias en la forma en que se usan

|  |  |
| --- | --- |
| **Operadores**  **lógicos** | **Descripción** |
| AND | Devuelve verdadero si ambos operadores son verdaderos. |
| SELECT \* FROM clientes WHERE edad > 18 AND edad < 30; | |
| **OR** | Devuelve verdadero si alguno de los operadores es verdadero. |
| SELECT \* FROM clientes WHERE apellido = 'González' OR apellido = 'García'; | |
| **NOT** | Invierte el resultado de la expresión que le sigue. |
| SELECT \* FROM clientes WHERE NOT edad > 18; | |
| **IN** | Devuelve verdadero si una expresión coincide con cualquiera de los valores de una lista. |
| SELECT \* FROM clientes WHERE apellido IN ('González', 'García'); | |
| **BETWEEN** | Devuelve verdadero si una expresión se encuentra dentro de un rango determinado. |
| SELECT \* FROM clientes WHERE edad BETWEEN 18 AND 30; | |
| **LIKE** | Devuelve verdadero si una expresión coincide con una cadena de caracteres determinada. |
| SELECT \* FROM clientes WHERE apellido LIKE 'G%'; | |

|  |
| --- |
|  |

**Comandos Básicos de SQL**

|  |
| --- |
| Cuando se trabaja con bases de datos relacionales utilizando SQL, es esencial dominar una serie de comandos que permiten crear, modificar, y manipular los datos de manera eficiente. |
| **SELECT**: Recupera datos de una tabla o vista.  SELECT nombre, edad FROM clientes; |
| **INSERT**: Agrega nuevos registros a una tabla.  INSERT INTO clientes (nombre, edad) VALUES ('Ariel', 30); |
| **UPDATE**: Modifica registros existentes en una tabla.  UPDATE clientes SET edad = 31 WHERE nombre = 'Ariel'; |
| **DELETE**: Elimina registros de una tabla que cumplan ciertas condiciones.  DELETE FROM clientes WHERE edad < 20; |
| **CREATE TABLE**: Crea una nueva tabla especificando columnas y tipos de datos.  CREATE TABLE clientes (  id INT PRIMARY KEY AUTO\_INCREMENT,  nombre VARCHAR(100),  edad INT  ); |
| **ALTER TABLE**: Modifica una tabla existente agregando, modificando o eliminando columnas.  ALTER TABLE clientes ADD telefono VARCHAR(15); |

.

|  |
| --- |
| **DROP TABLE**: Elimina una tabla y todos sus datos de forma permanente.  DROP TABLE clientes; |
| **CREATE DATABASE**: Crea una nueva base de datos.  CREATE DATABASE tienda; |
| **USE DATABASE**: Selecciona una base de datos específica para trabajar en ella.  USE tienda; |
| **ORDER BY**: Ordena los resultados de una consulta.  SELECT nombre FROM clientes ORDER BY edad DESC; |
| **GROUP BY**: Agrupa filas basadas en una columna, usado con funciones de agregación.  SELECT COUNT(\*), edad FROM clientes GROUP BY edad; |

.

|  |
| --- |
| **JOIN**: Combina filas de dos o más tablas basadas en una columna común.  SELECT clientes.nombre, pedidos.fecha  FROM clientes  JOIN pedidos ON clientes.id = pedidos.cliente\_id; |
| **WHERE**: Aplica condiciones para filtrar los registros seleccionados.  SELECT \* FROM clientes WHERE edad > 30; |
| **AND, OR**: Operadores lógicos para combinar condiciones en una consulta.  SELECT \* FROM clientes WHERE edad > 30 AND nombre = 'Ariel'; |
| **LIMIT**: Limita el número de filas devueltas por una consulta.  SELECT \* FROM clientes LIMIT 5; |
| **DISTINCT**: Elimina duplicados en los resultados de una consulta.  SELECT DISTINCT edad FROM clientes; |
| **HAVING**: Filtra grupos creados con GROUP BY en función de una condición.  SELECT COUNT(\*), edad FROM clientes GROUP BY edad HAVING COUNT(\*) > 1; |

.

|  |
| --- |
| **UNION**: Combina resultados de dos o más consultas.  SELECT nombre FROM clientes  UNION  SELECT nombre FROM empleados; |
| **CREATE INDEX**: Crea un índice en una o más columnas para mejorar la velocidad de las consultas.  CREATE INDEX idx\_nombre ON clientes (nombre); |
| **DROP INDEX**: Elimina un índice de una tabla.  DROP INDEX idx\_nombre ON clientes; |
| **GRANT**: Otorga permisos a usuarios en la base de datos.  GRANT SELECT, INSERT ON tienda.\* TO 'usuario'@'localhost'; |
| **REVOKE**: Revoca permisos otorgados previamente a usuarios.  REVOKE INSERT ON tienda.\* FROM 'usuario'@'localhost'; |
| **TRANSACTION**: Inicia una transacción para agrupar una serie de operaciones.  START TRANSACTION; |
| **COMMIT**: Confirma una transacción, guardando los cambios realizados.  COMMIT; |
| **ROLLBACK**: Deshace una transacción, revirtiendo los cambios no confirmados.  ROLLBACK; |

Los comandos SQL proporcionan una base sólida para administrar datos en sistemas de bases de datos relacionales. Con ellos, se pueden realizar operaciones fundamentales como crear tablas, insertar y actualizar registros, controlar los permisos de usuarios, y realizar consultas avanzadas con operaciones de unión, agregación y filtrado. Con un buen dominio de estos comandos, es posible gestionar eficientemente bases de datos robustas y optimizadas.

|  |
| --- |
| commit |

|  |
| --- |
| 1. ****SQLite****  * **Transacciones: En SQLite, cada instrucción que modifica la base de datos (como** CREATE**,** DROP**,** INSERT**,** UPDATE**,** DELETE**) se ejecuta en una transacción implícita. No necesitas hacer un** COMMIT **explícito a menos que estés utilizando transacciones manualmente.** |
| 2. ****MySQL****  * **Transacciones**: MySQL utiliza transacciones en tablas que tienen soporte para transacciones (como InnoDB). Para operaciones como CREATE, DROP, y ALTER, estas son implícitas y no requieren un COMMIT. |
| 3. ****SQL Server****  * **Transacciones**: SQL Server también requiere un COMMIT para transacciones explícitas. Las operaciones como CREATE y DROP son automáticas y no necesitan un COMMIT. |
| 4. ****PostgreSQL****  * **Transacciones**: Al igual que en SQL Server, en PostgreSQL las operaciones DDL (Data Definition Language) como CREATE y DROP son automáticas y no requieren COMMIT. Sin embargo, puedes usar transacciones explícitas si agrupas múltiples operaciones. |

### .

**Comandos SQL para Manipular Datos**

SQL es un lenguaje fundamental para la manipulación de bases de datos relacionales. En esta sección, profundizaremos en la inserción de datos en tablas y las mejores prácticas para hacerlo de manera eficiente y manteniendo la integridad referencial en bases de datos con múltiples relaciones.

**Inserción de Datos en una Tabla SQL**

El comando INSERT INTO es ampliamente utilizado para agregar datos a una tabla. SQL permite insertar tanto un solo registro como varios a la vez en una única instrucción.

Sintaxis de la Instrucción INSERT

**Inserción de un registro:**

INSERT INTO Nombre\_Tabla (columna1, columna2, ...)

VALUES (valor1, valor2, ...);

|  |
| --- |
| INSERT INTO clientes (nombre, telefono, email)  VALUES ('Ariel', '123456789', 'Ariel .Garcia@email.com'); |

**Inserción de Múltiples Registros**

Para insertar varios registros a la vez, la sintaxis es la siguiente:

INSERT INTO Nombre\_Tabla (columna1, columna2, ...)

VALUES

(valor1a, valor2a, ...),

(valor1b, valor2b, ...),

(valor1c, valor2c, ...);

|  |
| --- |
| INSERT INTO clientes (nombre, telefono, email)  VALUES  ('Ariel', '123456789', 'Ariel.Garcia@email.com'),  ('María López', '987654321', 'maria.lopez@email.com'); |

**Confirmar los Cambios**

En sistemas es necesario confirmar los cambios con commit después de Si

**Se puede haber excepciones en casos como:**

**SQLite**: Si no defines explícitamente una transacción (con BEGIN TRANSACTION), cada consulta es tratada como una transacción independiente y se confirma automáticamente (autocommit). Sin embargo, si trabajas con transacciones explícitas, también necesitas un COMMIT.

**PostgreSQL**: Al igual que en otros sistemas, si trabajas con transacciones explícitas, necesitas confirmar los cambios con COMMIT.

|  |
| --- |
| connection.commit()  **Nota**: En PostgreSQL, si conexion.autocommit = True, no es necesario usar commit. |

**Inserción de Valores en Columnas Específicas**

Es posible insertar valores únicamente en columnas específicas. Sin embargo, las columnas omitidas deben permitir valores nulos (NULL). Si una columna está definida como NOT NULL, es obligatorio proporcionar un valor para ella durante la inserción.

|  |
| --- |
| INSERT INTO clientes (nombre, telefono, campo\_no\_nulo, capo\_nulo)  VALUES ('Pedro Gómez', '654321987', 'Valor para campo\_no\_nulo', NULL); |

**Inserción de Múltiples Registros con executemany**

La función executemany permite insertar múltiples registros de manera eficiente en distintos motores de bases de datos como SQLite, MySQL, SQL Server y PostgreSQL.

Datos a Insertar:

|  |
| --- |
| data = [  ('Ariel1', 'avenida 01', '123456789'),  ('Ariel2', 'avenida 02', '123456789'),  ('Ariel3', 'avenida 03', '123456789'),  ('Ariel4', 'avenida 04', '123456789')  ] |
| **SQLite o SQL Server::**  cursor.executemany(''' INSERT INTO Alumnos (Nombre, Direccion, Telefono)  VALUES ( ? , ? , ? )''', data) |
| **MySQL o PostgreSQL::**  cursor.executemany(''' INSERT INTO Alumnos (Nombre, Direccion, Telefono)  VALUES (%s, %s, %s)''', data) |
| Al finalizar la inserción, es necesario confirmar los cambios:  connection.commit() |

**Inserción de Datos en Tablas Relacionadas**

Cuando trabajamos con bases de datos con múltiples tablas relacionadas, es crucial seguir una secuencia lógica para la inserción de datos, manteniendo la **integridad referencial**. Este enfoque asegura que las llaves foráneas apunten correctamente a los registros existentes.

Ejemplo: Tablas Relacionadas

En una base de datos escolar, tenemos las siguientes tablas:

1. **Asignaturas**
2. **Profesores**
3. **Alumnos**
4. **Inscripciones** (recibe llaves foráneas de las tablas anteriores)

Flujo de Inserción:

|  |
| --- |
| **Agregar Asignaturas**: Inserta primero los registros en la tabla Asignaturas.  INSERT INTO **Asignaturas** (nombre) VALUES ('Matemáticas'); |
| **Agregar Profesores**: Inserta los registros en la tabla Profesores.  INSERT INTO **Profesores** (nombre) VALUES ('Ariel'); |
| **Agregar Alumnos**: Inserta los registros en la tabla Alumnos.  INSERT INTO **Alumnos** (nombre, direccion) VALUES ('María López', 'Calle 123'); |
| **Agregar Inscripciones**: Una vez que las tablas anteriores tienen los registros necesarios, inserta los datos en la tabla Inscripciones, asegurando que las llaves foráneas apunten a los registros correctos.  INSERT INTO **Inscripciones** (IdAlumno, IdProfesor, IdAsignatura)  VALUES (1, 1, 1); |

Este flujo garantiza que las relaciones entre las tablas se mantengan intactas y que la base de datos mantenga su **integridad referencial**.

El comando INSERT INTO es esencial en la manipulación de datos en SQL, permitiendo agregar registros de manera flexible tanto en tablas simples como en aquellas con múltiples relaciones. Conocer cómo insertar datos correctamente y manejar las relaciones entre tablas es clave para mantener bases de datos consistentes y eficientes.

|  |
| --- |
|  |

Para insertar datos en dos o más tablas relacionadas usando SQLite, MySQL, SQL Server y PostgreSQL, y trabajar con claves foráneas para garantizar la integridad referencial,

1. crear las tres tablas relacionadas: **Alumnos**, **Asignaturas**, e **Inscripciones**.
2. insertar los datos correspondientes.

|  |
| --- |
| **SQLite**  -- Crear tabla Alumnos CREATE TABLE Alumnos (  IdAlumno INTEGER PRIMARY KEY AUTOINCREMENT,  Nombre VARCHAR(255) NOT NULL,  Direccion VARCHAR(255),  Telefono VARCHAR(15)  );  -- Crear tabla Asignaturas CREATE TABLE Asignaturas (  IdAsignatura INTEGER PRIMARY KEY AUTOINCREMENT,  NombreAsignatura VARCHAR(100)  );  -- Crear tabla Inscripciones CREATE TABLE Inscripciones (  IdInscripcion INTEGER PRIMARY KEY AUTOINCREMENT, IdAlumno INTEGER, IdAsignatura INTEGER, FechaInscripcion VARCHAR(15), FOREIGN KEY (IdAlumno) REFERENCES Alumnos(IdAlumno), FOREIGN KEY (IdAsignatura) REFERENCES Asignaturas(IdAsignatura)  ); |

.

|  |
| --- |
| **MySQL**  -- Crear tabla Alumnos  CREATE TABLE Alumnos (  IdAlumno INT AUTO\_INCREMENT PRIMARY KEY,  Nombre VARCHAR(255) NOT NULL,  Direccion VARCHAR(255),  Telefono VARCHAR(15)  );  -- Crear tabla Asignaturas  CREATE TABLE Asignaturas (  IdAsignatura INT AUTO\_INCREMENT PRIMARY KEY,  NombreAsignatura VARCHAR(100)  );  -- Crear tabla Inscripciones  CREATE TABLE Inscripciones (  IdInscripcion INT AUTO\_INCREMENT PRIMARY KEY,  IdAlumno INT,  IdAsignatura INT,  FechaInscripcion DATE,  FOREIGN KEY (IdAlumno) REFERENCES Alumnos(IdAlumno),  FOREIGN KEY (IdAsignatura) REFERENCES Asignaturas(IdAsignatura)  ); |

.

|  |
| --- |
| **SQL Server**  -- Crear tabla Alumnos  CREATE TABLE Alumnos (  IdAlumno INT IDENTITY(1,1) PRIMARY KEY,  Nombre NVARCHAR(255) NOT NULL,  Direccion NVARCHAR(255),  Telefono NVARCHAR(15)  );  -- Crear tabla Asignaturas  CREATE TABLE Asignaturas (  IdAsignatura INT IDENTITY(1,1) PRIMARY KEY,  NombreAsignatura NVARCHAR(100)  );  -- Crear tabla Inscripciones  CREATE TABLE Inscripciones (  IdInscripcion INT IDENTITY(1,1) PRIMARY KEY,  IdAlumno INT,  IdAsignatura INT,  FechaInscripcion DATE,  FOREIGN KEY (IdAlumno) REFERENCES Alumnos(IdAlumno),  FOREIGN KEY (IdAsignatura) REFERENCES Asignaturas(IdAsignatura)  ); |

.

|  |
| --- |
| **PostgreSQL**  -- Crear tabla Alumnos  CREATE TABLE Alumnos (  IdAlumno SERIAL PRIMARY KEY,  Nombre TEXT NOT NULL,  Direccion TEXT,  Telefono VARCHAR(15)  );  -- Crear tabla Asignaturas  CREATE TABLE Asignaturas (  IdAsignatura SERIAL PRIMARY KEY,  NombreAsignatura TEXT  );  -- Crear tabla Inscripciones  CREATE TABLE Inscripciones (  IdInscripcion SERIAL PRIMARY KEY,  IdAlumno INT,  IdAsignatura INT,  FechaInscripcion DATE,  FOREIGN KEY (IdAlumno) REFERENCES Alumnos(IdAlumno) ON DELETE CASCADE,  FOREIGN KEY (IdAsignatura) REFERENCES Asignaturas(IdAsignatura) ON DELETE CASCADE  ); |

.

|  |
| --- |
| En SQLite, **MySQL,** **SQL Server** ,**PostgreSQL** |
| -- Insertar datos en Alumnos  INSERT INTO Alumnos (IdAlumno, Nombre, Direccion, Telefono)  VALUES (1, 'Ariel1', 'avenida 01', '123456789'),  (2, 'Ariel2', 'avenida 02', '987654321');  -- Insertar datos en Asignaturas  INSERT INTO Asignaturas (IdAsignatura, NombreAsignatura)  VALUES (1, 'Matemáticas'), (2, 'Historia');  -- Insertar datos en Inscripciones  INSERT INTO Inscripciones (IdInscripcion, IdAlumno, IdAsignatura, FechaInscripcion)  VALUES (1, 1, 1, '2024-09-28'),(2, 2, 2, '2024-09-29'); |

.

Inserción de datos en todas las bases de datos

Los siguientes ejemplos insertan datos en las tablas **Alumnos**, **Asignaturas** e **Inscripciones** en los cuatro sistemas de bases de datos.

|  |
| --- |
| -- Insertar datos en la tabla Alumnos  INSERT INTO Alumnos (Nombre, Direccion, Telefono)  VALUES  ('Ariel1', 'avenida 01', '123456789'),  ('Ariel2', 'avenida 02', '987654321');  -- Insertar datos en la tabla Asignaturas  INSERT INTO Asignaturas (NombreAsignatura)  VALUES  ('Matemáticas'),  ('Historia');  -- Insertar datos en la tabla Inscripciones (relacionando alumnos con asignaturas)  INSERT INTO Inscripciones (IdAlumno, IdAsignatura, FechaInscripcion)  VALUES  (1, 1, '2024-09-28'),  (2, 2, '2024-09-29'); |

Detalles:

* **IdAlumno** y **IdAsignatura** son claves foráneas que aseguran la relación entre las tablas.
* En **Inscripciones**, estamos asociando alumnos con asignaturas y registrando la fecha de inscripción.

Este enfoque permite mantener la integridad de los datos y facilita la ejecución de consultas posteriores que involucren **JOINs** entre las tres tablas para obtener información como "qué alumnos están inscritos en qué asignaturas".

Ejemplos de consultas con JOINs en **SQLite**, **MySQL**, **SQL Server** y **PostgreSQL**, junto con una breve explicación de los tipos de JOIN y cómo se comportan en cada DBMS:

|  |
| --- |
| **JOIN SQLite, **MySQL,** **SQL Server y** **PostgreSQL**** |
| -- INNER JOIN (muestra solo los alumnos con inscripciones)  SELECT A.Nombre, I.IdAsignatura, I.FechaInscripcion  FROM Alumnos A  INNER JOIN Inscripciones I ON A.IdAlumno = I.IdAlumno;  **Resultado esperado**: Solo se mostrarán los alumnos que están inscritos en alguna asignatura. |
| -- LEFT JOIN o LEFT OUTER JOIN (muestra todos los alumnos aunque no tengan inscripciones)  SELECT A.Nombre, I.IdAsignatura, I.FechaInscripcion  FROM Alumnos A  LEFT JOIN Inscripciones I ON A.IdAlumno = I.IdAlumno;  **Resultado esperado**: Se mostrarán todos los alumnos. Si un alumno no tiene inscripción, se verá como NULL en las columnas correspondientes a la inscripción. |
| -- RIGHT JOIN o **RIGHT OUTER JOIN** (muestra todas las inscripciones aunque no haya alumnos asignados)  SELECT A.Nombre, I.IdAsignatura, I.FechaInscripcion  FROM Alumnos A  RIGHT JOIN Inscripciones I ON A.IdAlumno = I.IdAlumno;  **Resultado esperado**: Se mostrarán todas las inscripciones. Si no hay un alumno asignado a una inscripción (caso raro debido a la integridad referencial), el nombre del alumno aparecerá como NULL. |
| -- FULL JOIN o FULL OUTER JOIN (muestra todos los alumnos y todas las inscripciones aparecerán en el resultado.si no hay asociados sera NULL)  SELECT alumnos.nombre, inscripciones.asignatura FROM alumnos FULL JOIN inscripciones ON alumnos.id = inscripciones.alumno\_id;  **Resultado esperado**: Se mostrarán todas las filas de ambas tablas, tanto las que tienen coincidencias como las que no. Si una fila no tiene coincidencia en la otra tabla, los valores de las columnas correspondientes a esa tabla serán NULL. En otras palabras, es una combinación de un LEFT JOIN y un RIGHT JOIN. Si una fila no tiene coincidencia en la otra tabla, los valores de las columnas correspondientes a esa tabla serán NULL. |
| -- CROSS JOIN (muestra todos lo)  SELECT alumnos.nombre, inscripciones.asignatura FROM alumnos CROSS JOIN inscripciones;  **Resultado esperado**: Se mostrarán todas las combinaciones posibles de filas entre las dos tablas. |
| **Nota:**  **MySQL,** **SQL Server** ,**PostgreSQL** soporta INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN y CROSS JOIN .  **SQLite** soporta INNER JOIN, LEFT JOIN y CROSS JOIN. **No soporta** RIGHT JOIN ni FULL JOIN . |

.

Explicación de los tipos de JOIN:

1. **INNER JOIN**: Devuelve solo las filas que tienen coincidencias en ambas tablas. Si un alumno no está inscrito en ninguna asignatura, no aparecerá en el resultado.
   * Ejemplo: Si Ariel1 está inscrito en Matemáticas, aparecerá en el resultado, pero si Ariel2 no tiene inscripciones, no aparecerá.
2. **LEFT JOIN o LEFT OUTER JOIN** : Devuelve todas las filas de la tabla de la izquierda (Alumnos) y las coincidencias de la tabla de la derecha (Inscripciones). Si no hay coincidencias, los valores de las columnas de la tabla de la derecha serán NULL.
   * Ejemplo: Todos los alumnos aparecerán en el resultado, y si algún alumno no tiene inscripciones, las columnas de inscripción mostrarán NULL.
3. **RIGHT JOIN **o RIGHT OUTER JOIN****: Devuelve todas las filas de la tabla de la derecha (Inscripciones) y las coincidencias de la tabla de la izquierda (Alumnos). Si no hay coincidencias, los valores de las columnas de la tabla de la izquierda serán NULL.
   * Ejemplo: Todas las inscripciones aparecerán en el resultado. Si una inscripción no tiene un alumno asociado (caso raro), el nombre del alumno será NULL.
4. **FULL JOIN o FULL OUTER JOIN:** Devuelve **todas** las filas de **ambas** tablas, tanto las que tienen coincidencias como las que no. Si una fila no tiene coincidencia en la otra tabla, los valores de las columnas correspondientes a esa tabla serán NULL. En otras palabras, es una combinación de un **LEFT JOIN** y un **RIGHT JOIN**.
   * **Ejemplo**: Todos los alumnos y todas las inscripciones aparecerán en el resultado. Si un alumno no está inscrito en ninguna asignatura, las columnas de inscripción serán NULL. De igual manera, si hay inscripciones sin un alumno asociado, el nombre del alumno será NULL
5. **CROSS JOIN**: Devuelve el producto cartesiano de ambas tablas, es decir, combina cada fila de la primera tabla con cada fila de la segunda tabla. No se requiere que haya coincidencias entre las tablas. El número de filas resultantes es igual al número de filas en la primera tabla multiplicado por el número de filas en la segunda tabla.
   * **Ejemplo**: Si hay 3 alumnos y 2 asignaturas, el resultado de un CROSS JOIN incluirá 6 combinaciones (cada alumno asociado con cada asignatura).

Este tipo de consultas son útiles para relacionar la información de los alumnos con sus inscripciones y asignaturas, mostrando tanto las coincidencias como los casos en que no existen registros relacionados.

|  |
| --- |
| **Vistas** |

**Vistas en Bases de Datos: Una Herramienta Esencial para la Gestión de Datos**

Las vistas son una de las herramientas más poderosas y versátiles que ofrecen los sistemas de gestión de bases de datos. Estas permiten crear tablas virtuales a partir de consultas SQL y ofrecen una representación dinámica de los datos, que se actualiza conforme a los cambios en las tablas subyacentes. A diferencia de las tablas físicas, las vistas no almacenan los datos, sino que actúan como un espejo que refleja el resultado de la consulta que las define. Esta cualidad las convierte en una herramienta invaluable para la simplificación de consultas, la optimización del acceso a datos, el control de seguridad y la mejora del análisis de información.

**¿Qué es una Vista?**

Una vista es esencialmente una consulta SQL almacenada en la base de datos que se presenta y se trata como si fuera una tabla. Su propósito principal es simplificar el acceso a conjuntos de datos complejos y proporcionar a los usuarios una manera más clara e intuitiva de interactuar con la información. Las vistas no solo ocultan la complejidad de las consultas, sino que también permiten un mayor control sobre los datos que se exponen a diferentes tipos de usuarios.

Para crear una vista, se utiliza la instrucción CREATE VIEW. Esta puede incluir no solo consultas básicas, sino también combinaciones de múltiples tablas (mediante JOINs), operaciones de filtrado (WHERE), agregaciones (GROUP BY), entre otras. Por lo tanto, las vistas pueden adaptarse a necesidades específicas de consulta, proporcionando una abstracción flexible y eficiente.

**Ejemplo de Creación de Vista**

Consideremos el siguiente ejemplo: queremos combinar información de las tablas **Alumnos**, **Inscripciones** y **Asignaturas** para crear una vista que muestre un listado de los alumnos junto con las asignaturas en las que están inscritos y la fecha de inscripción. La instrucción SQL para crear esta vista sería la siguiente:

|  |
| --- |
| **VIEW en SQLite, **MySQL,** **SQL Server y** **PostgreSQL**** |
| CREATE VIEW VistaInscripcionesAlumnos AS  SELECT A.Nombre AS NombreAlumno, S.NombreAsignatura, I.FechaInscripcion  FROM Alumnos A  JOIN Inscripciones I ON A.IdAlumno = I.IdAlumno  JOIN Asignaturas S ON I.IdAsignatura = S.IdAsignatura; |

Esta vista nos ofrece una manera simple y directa de consultar la información que, de otro modo, requeriría la redacción de una consulta más compleja cada vez que necesitemos estos datos.

**Ventajas y Utilidades de las Vistas**

1. **Simplificación de Consultas Complejas**

Las vistas son extremadamente útiles para encapsular consultas SQL complicadas, que incluyen múltiples combinaciones de tablas (JOINs), filtrados o cálculos agregados. En lugar de repetir estas consultas cada vez que se necesiten, los desarrolladores pueden crear una vista que simplifique el acceso a los datos.

|  |
| --- |
| La consulta de la vista VistaInscripcionesAlumnos se puede realizar de la siguiente manera: |
| SELECT \* FROM VistaInscripcionesAlumnos; |

Esta consulta es mucho más clara y legible en comparación con volver a escribir la combinación de tablas y las relaciones cada vez que se necesiten estos datos. De esta manera, se optimiza el código SQL, lo que es especialmente útil en entornos de desarrollo complejos o aplicaciones grandes.

2. **Reutilización**

La reutilización de código es uno de los principales beneficios de las vistas. Si un conjunto de datos debe consultarse con frecuencia, encapsular la lógica en una vista permite que los desarrolladores no tengan que escribir repetidamente las mismas consultas. Esto no solo ahorra tiempo, sino que también reduce las probabilidades de cometer errores.

Además, al modificar la consulta subyacente de una vista, automáticamente se refleja en todas las consultas que la utilicen, lo que facilita el mantenimiento del código y asegura la consistencia en toda la aplicación.

3. **Seguridad y Control de Acceso**

Las vistas ofrecen un control granular sobre el acceso a los datos. Mediante la creación de vistas que expongan solo las columnas y filas necesarias, los administradores de la base de datos pueden limitar el acceso a información sensible.

|  |
| --- |
| Aquí queremos exponer únicamente el nombre y la dirección de los alumnos, ocultando otros detalles confidenciales. |
| CREATE VIEW VistaAlumnosPublico AS  SELECT Nombre, Direccion  FROM Alumnos; |

Con esta vista, podemos otorgar permisos para que ciertos usuarios accedan solo a la vista sin exponer información confidencial como el número de identificación o los detalles financieros de los alumnos. Este enfoque es común en entornos donde diferentes roles de usuario requieren distintos niveles de acceso.

4. **Facilitar el Análisis de Datos**

Las vistas son especialmente útiles para preparar datos para el análisis y la generación de informes. Al crear una vista que filtre y estructure los datos de manera adecuada, se elimina la necesidad de escribir consultas complejas repetidamente. Herramientas de análisis de datos o de Business Intelligence (BI) pueden aprovechar estas vistas, proporcionando acceso a conjuntos de datos predefinidos que pueden ser interpretados y visualizados sin complicaciones técnicas adicionales.

5. **Mejor del Rendimiento**

Aunque las vistas no almacenan datos por defecto, su uso puede mejorar el rendimiento de las consultas en ciertas situaciones. Esto ocurre cuando la vista permite al motor de la base de datos optimizar la ejecución de la consulta subyacente. Además, en algunos sistemas de gestión de bases de datos, como Oracle o PostgreSQL, existen **vistas materializadas**, que almacenan físicamente el resultado de la consulta. Las vistas materializadas ofrecen tiempos de respuesta más rápidos en consultas complejas, a cambio de ocupar espacio en disco y requerir actualizaciones periódicas para reflejar cambios en los datos subyacentes.

**Consideraciones sobre el Rendimiento y los Recursos**

El impacto en el rendimiento y el uso de recursos depende en gran medida de la complejidad de la vista:

* **Vistas Simples**: Si una vista es sencilla, basada en consultas directas sin muchas uniones o agregaciones, el impacto en los recursos es mínimo. Las vistas simples generalmente no requieren grandes cantidades de memoria o procesamiento, y su ejecución es rápida.
* **Vistas Complejas**: Las vistas que involucran múltiples JOINs, agregaciones o subconsultas pueden requerir un uso intensivo de CPU y memoria. Cada vez que se consulta una vista compleja, el sistema de la base de datos debe ejecutar la consulta subyacente, lo que puede aumentar el tiempo de respuesta, especialmente si las tablas involucradas contienen grandes volúmenes de datos.

**Vistas Materializadas**

En sistemas como PostgreSQL y Oracle, las **vistas materializadas** permiten almacenar los resultados de una consulta físicamente en disco. Estas vistas son útiles cuando se trabaja con consultas muy complejas o grandes volúmenes de datos que no cambian con frecuencia. Sin embargo, deben ser actualizadas manual o automáticamente para reflejar cambios en los datos subyacentes. A pesar de esto, mejoran notablemente el rendimiento en situaciones donde se necesita consultar la misma información repetidamente.

Las vistas materializadas almacenan físicamente el resultado de la consulta y pueden mejorar significativamente el rendimiento en sistemas de análisis o de consulta intensiva. Sin embargo, deben ser actualizadas para reflejar los cambios en las tablas subyacentes.

**Ventajas de las Vistas Materializadas:**

* **Rendimiento**: Acceso rápido a datos pre-calculados, útil en entornos donde el rendimiento es crucial.
* **Almacenamiento de Datos**: Los resultados de la consulta se almacenan físicamente, evitando el cálculo continuo.

**Desventajas:**

* **Actualización Manual**: Es necesario actualizar las vistas materializadas periódicamente para que los datos reflejen los cambios recientes.

|  |
| --- |
| En sistemas como PostgreSQL u Oracle, existen vistas materializadas que **sí** almacenan físicamente los resultados de la consulta. Estas son útiles para mejorar el rendimiento en consultas complejas. |
| REFRESH MATERIALIZED VIEW VistaMaterializadaEjemplo; |
| Este comando actualiza la vista materializada para reflejar los cambios en las tablas subyacentes. |

**Casos en los que las Vistas son Necesarias**

1. **Modularización del Acceso a Datos**

En entornos donde varios equipos de desarrollo o aplicaciones acceden a la misma base de datos, las vistas permiten modularizar y abstraer la lógica de las consultas complejas, haciendo el acceso a los datos más manejable y controlable. Por ejemplo, en un sistema educativo, una vista puede centralizar la consulta de las inscripciones, evitando duplicación de esfuerzos.

2. **Optimización del Mantenimiento de Consultas**

Las vistas permiten ajustar la forma en que se consulta la base de datos sin necesidad de modificar todas las consultas individuales. Esto es especialmente útil en entornos de desarrollo que cambian con rapidez o cuando los requisitos del sistema evolucionan.

3. **Mejora de la Seguridad y Privacidad**

Las vistas permiten definir qué datos están disponibles para cada tipo de usuario. Esto mejora la seguridad al limitar la exposición de información confidencial solo a aquellos que tienen permisos específicos para acceder a ella.

|  |
| --- |
| Ejemplos de Vistas en Diferentes Sistemas de Gestión de Bases de Datos |
| CREATE VIEW VistaInscripcionesAlumnos AS  SELECT A.Nombre AS NombreAlumno, S.NombreAsignatura, I.FechaInscripcion  FROM Alumnos A  JOIN Inscripciones I ON A.IdAlumno = I.IdAlumno  JOIN Asignaturas S ON I.IdAsignatura = S.IdAsignatura; |

Las vistas son una herramienta fundamental en la gestión de bases de datos. Al proporcionar una manera eficiente de simplificar consultas complejas, mejorar la reutilización de código, controlar el acceso a los datos y optimizar el rendimiento, las vistas son una pieza clave en la arquitectura de cualquier sistema basado en datos. Cuando se utilizan de manera estratégica, permiten construir soluciones más robustas y escalables, facilitando la gestión, el análisis y la seguridad de los datos.

Es crucial para los desarrolladores entender cómo y cuándo utilizar vistas para maximizar su valor en aplicaciones reales, asegurando un equilibrio entre simplicidad, rendimiento y seguridad.

**Exportación de Vistas**

Las vistas, al no almacenar datos, permiten que la exportación se centre en la definición de la consulta en lugar de los datos en sí. Dependiendo del sistema de base de datos, se pueden exportar las definiciones de las vistas de diferentes maneras:

Cuando exportamos una vista en una base de datos, no estamos exportando los datos que resultan de la consulta, sino más bien la definición de la consulta misma. Esto es útil para compartir la lógica detrás de una vista sin exponer los datos subyacentes.

**Métodos de Exportación por un sistemas de gestión de bases de datos (SGBD - DBMS** "Data Base Management System"**):**

|  |
| --- |
| MySQL y PostgreSQL: Se puede exportar la definición de la vista con la instrucción SHOW CREATE VIEW. Esto genera la consulta SQL usada para crear la vista. |
| SHOW CREATE VIEW VistaInscripcionesAlumnos; |

* **SQL Server**: A través de herramientas como SQL Server Management Studio (SSMS), se pueden generar scripts que contienen la definición de la vista para exportarla. SSMS tiene una opción para "Generar scripts" que facilita este proceso.

**Otras Consideraciones**

* **Rendimiento**: Si bien las vistas son herramientas útiles para simplificar consultas y mejorar la modularidad, es crucial considerar su rendimiento al trabajar con grandes volúmenes de datos o consultas complejas. Evaluar el rendimiento de las vistas es esencial para asegurar que no se conviertan en un cuello de botella en el sistema.
* **Necesidad**: Las vistas son esenciales para reutilizar consultas, simplificar el acceso a datos y controlar qué información se expone a los usuarios. Al utilizarlas estratégicamente, los desarrolladores pueden mejorar la eficiencia del sistema y facilitar la gestión de datos.
* **Exportación**: Exportar una vista implica compartir la consulta subyacente, lo que permite preservar la estructura lógica sin comprometer la seguridad de los datos sensibles.

Las vistas son una herramienta fundamental en la gestión de bases de datos modernas, proporcionando beneficios significativos en términos de simplificación de consultas, reutilización, seguridad y rendimiento. Al comprender y aplicar el uso adecuado de las vistas, los desarrolladores pueden optimizar el acceso a los datos y mejorar la eficiencia en la gestión de bases de datos, convirtiéndose en una práctica estándar en el desarrollo de aplicaciones basadas en datos.

Además, es crucial fomentar una comprensión más profunda de cómo las vistas pueden transformar la forma en que interactuamos con los datos. Al implementar vistas de manera estratégica, se puede lograr una arquitectura de base de datos más robusta y flexible que no solo cumpla con los requisitos actuales, sino que también se adapte a futuras necesidades de negocio.

**Vistas Actualizables**

No todas las vistas permiten operaciones de actualización, inserción o eliminación.

|  |
| --- |
| Condiciones para que una Vista sea Actualizable:   1. **Selección Simple**: La consulta no debe contener agregaciones (como SUM o COUNT) ni subconsultas. 2. **No JOINs Complejos**: Las vistas que incluyen muchas uniones complicadas o subconsultas anidadas pueden no ser actualizables. 3. **Columnas Necesarias**: Todas las columnas requeridas para la actualización (por ejemplo, las claves primarias) deben estar presentes. 4. **Permisos**: Los usuarios que trabajan con la vista deben tener permisos sobre las tablas subyacentes. |
| -- Vista Actualizable  CREATE VIEW VistaAlumnosActivos AS  SELECT IdAlumno, Nombre, Direccion  FROM Alumnos  WHERE Activo = 1; |
| -- Vista No Actualizable (debido a la función COUNT)  CREATE VIEW VistaInscripcionesConContador AS  SELECT I.IdInscripcion, A.Nombre, COUNT(I.IdAsignatura) AS TotalAsignaturas  FROM Inscripciones I  JOIN Alumnos A ON I.IdAlumno = A.IdAlumno  GROUP BY I.IdInscripcion, A.Nombre; |

**Limitaciones de las Vistas**

* **Cláusulas Restringidas**

Algunas bases de datos no permiten usar ciertas cláusulas en las vistas.

Por ejemplo, en SQL Server, no se puede usar ORDER BY sin TOP en una vista.

* **Complejidad en JOINs**

Las vistas que usan múltiples uniones o subconsultas complejas pueden no ser actualizables y también pueden disminuir el rendimiento.

* **Rendimiento**

El uso excesivo de vistas anidadas o vistas con varias capas puede degradar el rendimiento del sistema, ya que cada capa añade más complejidad a la consulta.

* **Anidación de Vistas**

Es posible crear vistas dentro de otras vistas para modularizar las consultas y mejorar la reutilización del código.

|  |
| --- |
| CREATE VIEW VistaAlumnosActivos AS  SELECT IdAlumno, Nombre FROM Alumnos WHERE Activo = 1; |
| CREATE VIEW VistaInscripcionesActivas AS  SELECT I.IdInscripcion, A.Nombre  FROM Inscripciones I  JOIN VistaAlumnosActivos A ON I.IdAlumno = A.IdAlumno; |

**Beneficios:**

* **Modularización**: Facilita la gestión y comprensión de consultas complejas.
* **Reutilización**: Permite la reutilización de lógica SQL en múltiples consultas.

**Limitaciones:**

* **Rendimiento**: Cada vista anidada añade una capa de complejidad que puede degradar el rendimiento si no se gestiona adecuadamente.

**Indexación de Vistas**

Algunas bases de datos permiten crear índices sobre vistas (particularmente sobre vistas materializadas) para mejorar el rendimiento.

**Ventajas de la Indexación:**

* **Consultas más rápidas**: Las consultas sobre vistas indexadas pueden ser significativamente más rápidas, ya que el índice acelera la búsqueda de los datos almacenados en la vista.

**Mantenimiento de Vistas**

Mantener las vistas actualizadas es esencial, especialmente cuando cambian los datos o la estructura de las tablas subyacentes. Algunas prácticas recomendadas incluyen:

* **Actualización periódica de vistas materializadas**.
* **Documentación clara** de las vistas para facilitar el mantenimiento y actualización.

**Buenas Prácticas:**

* **Evaluar las vistas regularmente**: Para evitar cuellos de botella en el rendimiento, sobre todo con consultas complejas.
* **Uso estratégico de vistas materializadas**: Mejora consultas costosas, pero debe considerarse el costo en espacio y tiempo de actualización.
* **Modularización**: Usar vistas para dividir consultas complejas y reutilizarlas en diferentes partes del sistema.

|  |
| --- |
| **Mas temas** |

### 1. ****Índices****

Los índices son estructuras que mejoran la eficiencia de las consultas en bases de datos al permitir un acceso más rápido a los registros sin necesidad de leer toda la tabla. Funcionan como un índice en un libro, permitiendo saltar directamente a la página correspondiente. Los tipos comunes de índices incluyen índices únicos (evitan duplicados), compuestos (que abarcan más de una columna), y parciales (aplicados solo a ciertas filas). Si bien los índices mejoran la velocidad de lectura, pueden ralentizar las operaciones de escritura como inserciones o actualizaciones debido a la necesidad de mantener la estructura del índice actualizada.

### 2. ****Transacciones y manejo de concurrencia****

Las transacciones en bases de datos son un conjunto de operaciones que se ejecutan como una unidad indivisible, garantizando que todas las operaciones se completen o ninguna lo haga. Los principios **ACID** (Atomicidad, Consistencia, Aislamiento y Durabilidad) aseguran que las transacciones mantengan la integridad de la base de datos. El manejo de la concurrencia permite que múltiples usuarios accedan a la base de datos de manera simultánea sin causar inconsistencias. Para esto, se utilizan bloqueos (locks) y diferentes niveles de aislamiento, como Read Committed o Serializable, para controlar cómo las transacciones ven y afectan los datos de otras transacciones en curso.

### 3. ****Triggers****

Un trigger es un procedimiento almacenado que se ejecuta automáticamente en respuesta a ciertos eventos como inserciones, actualizaciones o eliminaciones de registros. Los triggers pueden usarse para tareas como validar datos antes de la inserción, mantener la integridad de datos en tablas relacionadas, o generar automáticamente registros de auditoría. Por ejemplo, un trigger podría registrar en una tabla de auditoría cada vez que un registro es eliminado de una tabla sensible. Aunque los triggers son potentes, es importante usarlos con moderación, ya que pueden afectar el rendimiento si se ejecutan frecuentemente.

### 4. ****Optimización de consultas****

La optimización de consultas consiste en mejorar el rendimiento de las consultas SQL para reducir los tiempos de respuesta. Un plan de ejecución es un mapa que muestra cómo el motor de la base de datos ejecuta una consulta, permitiendo identificar cuellos de botella. Algunas técnicas incluyen el uso adecuado de índices, evitar subconsultas innecesarias, reducir el uso de SELECT \*, y preferir joins sobre consultas anidadas. Los optimizadores de consultas también ayudan a reorganizar y ejecutar de manera eficiente las instrucciones SQL.

### 5. ****Particionamiento de tablas****

El particionamiento divide una tabla grande en partes más pequeñas y manejables llamadas particiones. Cada partición puede almacenarse y gestionarse de manera independiente. Existen varios tipos de particionamiento, como particionamiento por rango, lista o hash. Esta técnica mejora el rendimiento de consultas que se centran en una parte específica de los datos, como una partición de fechas recientes en una tabla de registros históricos. También facilita la administración, permitiendo eliminar o archivar particiones enteras sin afectar otras.

### 6. ****Replicación y alta disponibilidad****

La replicación es el proceso de copiar datos de una base de datos a otra para garantizar la disponibilidad de la información y distribuir la carga de trabajo. Existen varios tipos de replicación, como la replicación síncrona (donde los datos se replican instantáneamente) y la replicación asíncrona (donde puede haber un pequeño retraso). La replicación es clave en configuraciones de alta disponibilidad, asegurando que si un servidor falla, otro pueda tomar el control sin pérdida de datos. Los clústeres de alta disponibilidad permiten manejar fallas con mínima interrupción del servicio.

### 7. ****Backup y recuperación****

El respaldo (backup) es una copia de seguridad de los datos de la base de datos que se utiliza para restaurarla en caso de fallo o pérdida. Los tipos de respaldo incluyen completo (todos los datos), diferencial (solo los cambios desde el último respaldo completo), e incremental (solo los cambios desde el último respaldo, sea completo o incremental). Los planes de recuperación de desastres deben incluir estrategias de backup y pruebas regulares de recuperación para garantizar que los datos pueden restaurarse de manera confiable en situaciones críticas.

### 8. ****Seguridad en bases de datos****

La seguridad en bases de datos es crucial para proteger datos sensibles y evitar accesos no autorizados. El control de acceso se implementa mediante roles y permisos, restringiendo qué usuarios pueden acceder o modificar qué partes de la base de datos. La encriptación protege los datos tanto en tránsito como en reposo, asegurando que solo usuarios autorizados puedan leer la información. La auditoría es otro componente clave, registrando cada acción que los usuarios realizan en la base de datos, lo cual permite monitorear actividades sospechosas.

### 9. ****Integridad referencial****

La integridad referencial asegura que las relaciones entre tablas se mantengan coherentes. Por ejemplo, una clave foránea en una tabla de inscripciones debe referenciar un registro válido en la tabla de alumnos. Si se elimina un alumno, también deberían eliminarse o actualizarse las inscripciones asociadas para mantener la consistencia de los datos. Esto se logra mediante restricciones como FOREIGN KEY, ON DELETE CASCADE, y ON UPDATE CASCADE, las cuales automatizan el mantenimiento de estas relaciones.

### 10. ****Bases de datos NoSQL****

Las bases de datos NoSQL son una alternativa a las bases de datos relacionales tradicionales, diseñadas para manejar grandes volúmenes de datos no estructurados o semiestructurados. A diferencia de las bases de datos SQL, que utilizan tablas y relaciones, NoSQL puede trabajar con formatos de datos más flexibles, como documentos (MongoDB), clave-valor (Redis), grafos (Neo4j), o columnas anchas (Cassandra). NoSQL es ideal para aplicaciones donde los datos cambian rápidamente o son difíciles de estructurar en un formato rígido.

### 11. ****Sharding y escalabilidad horizontal****

El **sharding** es una técnica de particionamiento que distribuye los datos de una base de datos entre múltiples servidores para manejar grandes volúmenes de datos y tráfico. Cada "shard" contiene un subconjunto de datos, lo que reduce la carga en un solo servidor y mejora la capacidad de la base de datos para manejar un mayor número de solicitudes. La escalabilidad horizontal se refiere a añadir más servidores para dividir la carga de trabajo, en lugar de aumentar la capacidad de un solo servidor (escalabilidad vertical).

### 12. ****Data Warehousing y OLAP****

Un **data warehouse** es una base de datos optimizada para consultas y análisis de grandes volúmenes de datos históricos, en lugar de para transacciones rápidas. OLAP (Online Analytical Processing) es una tecnología que permite realizar análisis multidimensionales sobre datos almacenados en un data warehouse. Esto es útil en contextos como análisis de tendencias, previsión de ventas o minería de datos, donde las consultas requieren procesar grandes cantidades de datos agrupados por múltiples dimensiones como tiempo, ubicación o producto.

|  |
| --- |
| OLAP (Online Analytical Processing) es una tecnología utilizada para realizar consultas y análisis multidimensionales en grandes volúmenes de datos. Las herramientas OLAP permiten analizar los datos desde diferentes perspectivas, generalmente a través de la creación de "cubos" OLAP, que son estructuras de datos multidimensionales. Aquí tienes algunos ejemplos de cómo se puede utilizar OLAP en diferentes sectores: 1. ****Ejemplo en el Sector de Ventas y Marketing:****  * **Caso:** Una cadena de supermercados desea analizar las ventas. * **Dimensiones:** Producto, tiempo (día, mes, año), ubicación (tienda, región). * **Medidas:** Ingresos, cantidad de productos vendidos. * **Consultas OLAP:**   + **"¿Cuáles fueron las ventas totales por producto en la región norte durante el último trimestre?"**   + **"¿Cómo han cambiado las ventas de productos electrónicos en diferentes tiendas a lo largo de los últimos tres años?"**   + **"¿Cuál fue el rendimiento de ventas en cada trimestre para una categoría de productos específica?"**   El sistema OLAP permite navegar en los datos (drill-down) para analizar las ventas en detalle o resumirlas en un nivel más alto (roll-up). 2. ****Ejemplo en el Sector Financiero:****  * **Caso:** Un banco quiere realizar un análisis de riesgos y rentabilidad. * **Dimensiones:** Tipo de cliente (corporativo, personal), productos financieros (préstamos, hipotecas, tarjetas), tiempo (día, mes, trimestre). * **Medidas:** Ganancias, pérdidas, riesgo crediticio. * **Consultas OLAP:**   + **"¿Cuál fue la ganancia total por producto financiero en el primer trimestre del año?"**   + **"¿Cuál es el riesgo crediticio promedio para clientes corporativos en la región oeste?"**   + **"¿Cómo se han comportado las pérdidas por impago de préstamos en diferentes segmentos de clientes en los últimos cinco años?"**   El análisis OLAP ayuda a identificar patrones de comportamiento financiero y a segmentar riesgos o rentabilidades por diferentes atributos. 3. ****Ejemplo en el Sector de la Salud:****  * **Caso:** Un hospital necesita analizar el rendimiento de sus médicos y tratamientos. * **Dimensiones:** Tiempo, especialidad médica, ubicación (hospitales o clínicas), tipo de paciente. * **Medidas:** Número de pacientes tratados, costo del tratamiento, tasa de éxito. * **Consultas OLAP:**   + **"¿Cuál es la tasa de éxito de cirugías ortopédicas en el último año?"**   + **"¿Cuántos pacientes han sido tratados por enfermedad X en todas las clínicas en el último trimestre?"**   + **"¿Cuál es el costo promedio de tratamiento en diferentes departamentos durante los últimos tres meses?"**   Los datos pueden analizarse para mejorar el rendimiento de los tratamientos o detectar áreas donde los costos son elevados. 4. ****Ejemplo en el Sector Logístico:****  * **Caso:** Una empresa de logística desea optimizar su cadena de suministro. * **Dimensiones:** Proveedor, producto, tiempo, región de entrega. * **Medidas:** Tiempo de entrega, costo de transporte, cantidad entregada. * **Consultas OLAP:**   + **"¿Cuál es el costo promedio de entrega por región durante el último mes?"**   + **"¿Qué proveedores han entregado productos con retraso más de un 5% del tiempo en el último año?"**   + **"¿Cómo han variado los costos de transporte por tipo de producto en diferentes trimestres?"**   OLAP permite analizar las operaciones logísticas para optimizar costos y tiempos de entrega. 5. ****Ejemplo en el Comercio Electrónico:****  * **Caso:** Un sitio web de e-commerce analiza el comportamiento de los clientes. * **Dimensiones:** Cliente, tiempo, categoría de producto, ubicación geográfica. * **Medidas:** Ingresos, número de visitas, tasa de conversión. * **Consultas OLAP:**   + **"¿Qué categorías de productos tienen la mayor tasa de conversión en cada región?"**   + **"¿Cuál es el ingreso promedio por cliente en el último año, desglosado por categoría de producto?"**   + **"¿Cómo ha cambiado el comportamiento de compra de los clientes en el último trimestre en comparación con el trimestre anterior?"**   El análisis OLAP aquí permite personalizar estrategias de marketing y mejorar la experiencia del cliente. |
| Características Clave de OLAP:  * **Drill-down**: Permite analizar los datos en un nivel de detalle más granular. * **Roll-up**: Resumen de los datos a un nivel más general o agregado. * **Slice and dice**: Extraer y analizar un subconjunto de datos en base a ciertas dimensiones. * **Pivoting**: Reorganizar las dimensiones para ver los datos desde otra perspectiva.   Los sistemas OLAP se utilizan para obtener información de manera rápida y flexible, permitiendo a las organizaciones tomar decisiones más informadas. |
| Para explorar y ver en acción una herramienta OLAP (Online Analytical Processing), tienes varias opciones dependiendo de tus necesidades y el nivel de interactividad que busques. Aquí te menciono algunas opciones para acceder a sistemas OLAP y hacer análisis multidimensional: 1. ****Power BI (Microsoft)****  * **Descripción:** Power BI es una herramienta de visualización de datos y análisis que ofrece funcionalidades OLAP. Puedes conectarte a varias fuentes de datos, crear cubos OLAP y realizar análisis multidimensionales mediante gráficos y tablas dinámicas. * **Cómo acceder:**   + Puedes descargar la versión gratuita de Power BI Desktop desde [aquí](https://powerbi.microsoft.com/es-es/desktop/).   + Power BI también tiene una versión online (en la nube) que permite compartir y ver informes interactivos. * **Funciones OLAP:** Puedes usar funcionalidades como Drill-Down, Roll-Up y filtros dinámicos en tablas y gráficos.  2. ****Tableau****  * **Descripción:** Tableau es una herramienta de análisis visual que también soporta análisis OLAP. Permite crear dashboards y visualizaciones interactivas de datos a partir de cubos OLAP y otras fuentes de datos. * **Cómo acceder:**   + Puedes obtener una versión de prueba gratuita de 14 días en Tableau.   + Tableau se conecta fácilmente a bases de datos multidimensionales y cubos OLAP para realizar análisis visuales. * **Funciones OLAP:** Al igual que Power BI, Tableau permite realizar drill-down y roll-up, y ofrece una visualización fluida para datos complejos.  3. ****Pentaho (Community Edition)****  * **Descripción:** Pentaho es una plataforma de BI de código abierto que ofrece funcionalidades OLAP mediante su componente Mondrian OLAP. Pentaho es ideal si deseas una solución gratuita y de código abierto. * **Cómo acceder:**   + Descarga la edición comunitaria (gratuita) desde Pentaho Community Edition.   + Mondrian OLAP se integra en Pentaho para crear y visualizar cubos OLAP. * **Funciones OLAP:** Pentaho permite explorar cubos OLAP, realizar consultas multidimensionales (MDX), y ofrece visualizaciones básicas y complejas.  4. ****Apache Superset****  * **Descripción:** Superset es una plataforma de visualización de datos y análisis de código abierto que se puede configurar para consultas OLAP con varias fuentes de datos, como bases de datos multidimensionales. * **Cómo acceder:**   + Puedes instalar Apache Superset en tu sistema siguiendo las instrucciones en su [repositorio de GitHub](https://github.com/apache/superset) o utilizar versiones alojadas en la nube. * **Funciones OLAP:** Aunque no es un sistema OLAP puro, puedes conectarlo a bases de datos OLAP y realizar consultas, visualizaciones, y análisis de datos en tiempo real.  5. ****Excel (con Analysis Services)****  * **Descripción:** Excel tiene soporte para análisis OLAP a través de su funcionalidad de **tablas dinámicas** conectadas a cubos OLAP, especialmente cuando se usa junto con **SQL Server Analysis Services (SSAS)**. * **Cómo acceder:**   + Si ya tienes Excel, puedes conectarlo a un cubo OLAP alojado en SQL Server u otros proveedores compatibles con OLAP.   + Para usar OLAP en Excel, crea una tabla dinámica y selecciona una fuente de datos que sea compatible con OLAP (por ejemplo, un cubo de Analysis Services). * **Funciones OLAP:** Excel permite realizar análisis con drill-down, slice and dice, y es una solución común para los usuarios de Microsoft que necesitan OLAP de forma sencilla.  6. ****Qlik Sense****  * **Descripción:** Qlik Sense es otra poderosa herramienta de análisis y visualización de datos que puede trabajar con fuentes OLAP para proporcionar un análisis multidimensional avanzado. * **Cómo acceder:**   + Puedes descargar la versión gratuita desde Qlik Sense. * **Funciones OLAP:** Qlik Sense es conocido por su capacidad de combinar múltiples fuentes de datos, incluidos cubos OLAP, y ofrece análisis avanzados con dashboards interactivos.  7. ****Microsoft SQL Server Analysis Services (SSAS)****  * **Descripción:** SSAS es una solución OLAP de Microsoft que permite crear y administrar cubos OLAP directamente desde SQL Server. Es una herramienta potente para empresas que ya trabajan con el ecosistema de SQL Server. * **Cómo acceder:**   + SQL Server Analysis Services se instala junto con SQL Server, por lo que si tu organización ya tiene SQL Server, puedes acceder a SSAS.   + Una vez configurado, puedes conectarte a SSAS desde herramientas como Excel, Power BI o Tableau para explorar los cubos OLAP. * **Funciones OLAP:** SSAS permite crear cubos OLAP desde bases de datos relacionales y ofrece un entorno rico para consultas MDX y análisis multidimensional. |

### 13. ****Big Data y bases de datos distribuidas****

El **Big Data** se refiere a la gestión de conjuntos de datos extremadamente grandes y complejos que no pueden ser tratados eficientemente con bases de datos tradicionales. Las bases de datos distribuidas, como **Hadoop** y **Cassandra**, dividen los datos y el procesamiento entre múltiples nodos, permitiendo una escalabilidad masiva y procesamiento paralelo. Estas tecnologías son clave en aplicaciones como análisis de redes sociales, sistemas de recomendación, o cualquier aplicación que genere grandes volúmenes de datos rápidamente.

|  |
| --- |
| El concepto de **Big Data** y las **bases de datos distribuidas** están estrechamente relacionados, ya que ambos son fundamentales en la gestión, almacenamiento y procesamiento eficiente de grandes volúmenes de datos en entornos modernos. A continuación, te explico ambos conceptos y cómo interactúan: |
| **1. ¿Qué es Big Data?**  **Big Data** se refiere al manejo de conjuntos de datos extremadamente grandes y complejos que no pueden ser tratados o analizados de manera eficiente mediante las técnicas tradicionales de procesamiento de datos. Big Data es caracterizado comúnmente por las **5 V's**:   1. **Volumen:** La cantidad masiva de datos generados por diversas fuentes (redes sociales, dispositivos IoT, transacciones, sensores, etc.). 2. **Velocidad:** La rapidez con la que los datos son generados, almacenados y procesados. Esto es crucial en aplicaciones en tiempo real. 3. **Variedad:** La diversidad en el tipo de datos (estructurados, semiestructurados y no estructurados), como imágenes, textos, audio, vídeo y datos de sensores. 4. **Veracidad:** La calidad y fiabilidad de los datos, ya que pueden estar incompletos o inexactos. 5. **Valor:** El valor que se puede extraer de los datos mediante análisis avanzados, que es el objetivo principal de Big Data. |
| ****2. ¿Qué son las Bases de Datos Distribuidas?**** Una **base de datos distribuida** es un sistema en el que los datos no se almacenan en un único lugar central, sino que están distribuidos en varios nodos o servidores ubicados en diferentes lugares geográficos o dentro de un clúster de servidores. Los datos pueden estar distribuidos en diferentes ubicaciones físicas, pero los usuarios pueden acceder a ellos como si estuvieran almacenados en un solo lugar. Características clave:  * **Escalabilidad horizontal:** Permiten agregar más servidores para manejar mayores volúmenes de datos o tráfico sin necesidad de reemplazar hardware existente. * **Tolerancia a fallos:** Al distribuir los datos en varios nodos, el sistema puede continuar funcionando incluso si algunos de los nodos fallan. * **Alta disponibilidad:** Los datos están replicados en múltiples nodos para garantizar que estén disponibles incluso si una parte del sistema se cae. * **Consistencia, disponibilidad y partición (Teorema CAP):** Las bases de datos distribuidas deben equilibrar estos tres factores, pero no pueden maximizar los tres a la vez. Esto se resume en el **Teorema CAP**, que indica que un sistema puede ser consistente, estar siempre disponible o ser tolerante a la partición, pero no todas a la vez en el mismo grado. |
| ****Relación entre Big Data y Bases de Datos Distribuidas**** El procesamiento de **Big Data** requiere plataformas capaces de manejar el almacenamiento y el procesamiento de cantidades masivas de datos que a menudo están distribuidos geográficamente y generados por diversas fuentes. Las bases de datos distribuidas son fundamentales para implementar soluciones de Big Data debido a las siguientes razones:   1. **Almacenamiento Escalable:**    * El volumen de datos en aplicaciones de Big Data puede crecer rápidamente. Las bases de datos distribuidas permiten escalar de manera horizontal, añadiendo más nodos o servidores en lugar de actualizar el hardware existente. 2. **Procesamiento Distribuido:**    * Los sistemas de Big Data como **Hadoop**, **Spark** y **MapReduce** se basan en la idea de dividir los datos en partes más pequeñas y procesarlas en paralelo en diferentes nodos dentro de un clúster. 3. **Alta Disponibilidad y Tolerancia a Fallos:**    * Las bases de datos distribuidas como **Cassandra** o **HBase** garantizan que los datos estén disponibles incluso en caso de fallo de uno o más nodos. Esto es esencial para aplicaciones que no pueden permitirse tiempo de inactividad, como los sistemas de recomendación, análisis financiero o aplicaciones en tiempo real. 4. **Variedad de Datos:**    * Los sistemas de Big Data deben manejar datos estructurados (por ejemplo, bases de datos relacionales), semiestructurados (como archivos JSON, XML) y no estructurados (como imágenes, vídeos). Las bases de datos distribuidas permiten almacenar diferentes tipos de datos mediante arquitecturas flexibles como **NoSQL**. |
| ****Tecnologías Populares en Big Data y Bases de Datos Distribuidas********Bases de Datos Distribuidas:****  1. **Apache Cassandra:**    * Base de datos NoSQL distribuida diseñada para manejar grandes volúmenes de datos distribuidos en múltiples servidores sin un único punto de fallo.    * Popular en empresas como Netflix y Facebook. 2. **HBase:**    * Sistema de base de datos distribuida y no relacional construido sobre Hadoop HDFS. Está diseñado para manejar grandes tablas de datos y provee acceso aleatorio y en tiempo real a los mismos. 3. **MongoDB:**    * Base de datos NoSQL basada en documentos, distribuida por diseño. MongoDB permite la escalabilidad horizontal y es utilizada en aplicaciones que requieren almacenar y consultar grandes volúmenes de datos no estructurados. 4. **Amazon DynamoDB:**    * Base de datos NoSQL totalmente gestionada, distribuida y escalable que ofrece alta disponibilidad y rendimiento constante. Es utilizada por empresas que necesitan baja latencia y escalabilidad. 5. **Google Bigtable:**    * Base de datos distribuida diseñada para manejar grandes cantidades de datos a baja latencia. Utilizada por Google para servicios como Google Maps y Gmail. |
| ****Plataformas de Procesamiento de Big Data:****  1. **Apache Hadoop:**    * Framework de procesamiento distribuido que permite el almacenamiento y procesamiento de grandes conjuntos de datos a través de múltiples nodos en un clúster. Utiliza HDFS (Hadoop Distributed File System) para almacenar datos distribuidos. 2. **Apache Spark:**    * Sistema de procesamiento distribuido en memoria que ofrece una alternativa más rápida que Hadoop MapReduce para el análisis en tiempo real de grandes volúmenes de datos. 3. **Apache Kafka:**    * Plataforma distribuida de streaming que permite el manejo de grandes flujos de datos en tiempo real. Se utiliza en aplicaciones como análisis de eventos en tiempo real y procesamiento de datos IoT. 4. **Elasticsearch:**    * Motor de búsqueda y análisis distribuido, basado en documentos JSON. Aunque comenzó como una herramienta de búsqueda, es ampliamente utilizada para análisis en tiempo real de grandes volúmenes de datos. |
| ****Casos de Uso de Big Data y Bases de Datos Distribuidas****  1. **Análisis en Tiempo Real (IoT):**    * Los dispositivos IoT generan grandes cantidades de datos que deben procesarse en tiempo real. Las bases de datos distribuidas y plataformas de procesamiento de Big Data, como Apache Kafka y Spark, son fundamentales para capturar, almacenar y procesar estos datos rápidamente. 2. **E-commerce y Recomendaciones Personalizadas:**    * Las plataformas de e-commerce utilizan bases de datos distribuidas (por ejemplo, Cassandra o MongoDB) para almacenar el comportamiento de los usuarios y ofrecer recomendaciones de productos en tiempo real mediante el análisis de grandes volúmenes de datos históricos y actuales. 3. **Monitorización y Análisis de Redes Sociales:**    * Las empresas que analizan redes sociales utilizan tecnologías de Big Data como Hadoop o Elasticsearch junto con bases de datos distribuidas para capturar y analizar millones de publicaciones en tiempo real para obtener información sobre la opinión de los usuarios. 4. **Servicios Financieros y Banca:**    * En la banca y las finanzas, se utilizan sistemas de Big Data y bases de datos distribuidas para realizar análisis de riesgo, detección de fraudes y monitoreo en tiempo real de transacciones. |

### Conclusión

Cada uno de estos temas cubre un aspecto crucial en el manejo profesional de bases de datos. Abarcan desde la optimización del rendimiento y la seguridad, hasta la preparación para grandes volúmenes de datos y alta disponibilidad, ofreciendo un conjunto de herramientas y conocimientos que permiten a los administradores de bases de datos manejar de manera eficiente y segura cualquier entorno de datos, ya sea en pequeñas aplicaciones o en sistemas d

|  |
| --- |
| Procedimientos |

**Creación de Procedimientos Almacenados en SQL**

Un **procedimiento almacenado** (o Stored Procedure) es un conjunto de instrucciones SQL que se almacena en la base de datos y se puede ejecutar de manera repetitiva. Este concepto es fundamental en el diseño de bases de datos y en la optimización del rendimiento de aplicaciones, ya que permite encapsular lógica de negocio compleja, mejorar la seguridad, y facilitar el mantenimiento del código.

Los **Procedimientos Almacenados** son validos en MySQL, SQL Server, PostgreSQL y, aunque SQLite no los soporta de manera directa, se puede ilustrar el uso de funciones.

**Importancia de los Procedimientos Almacenados**

1. **Reutilización de Código**: Almacenar un conjunto de instrucciones SQL permite reutilizar este código en diferentes partes de una aplicación sin necesidad de reescribirlo.
2. **Rendimiento**: Los procedimientos almacenados son compilados y optimizados por el servidor de bases de datos, lo que significa que su ejecución suele ser más rápida que la ejecución de múltiples sentencias SQL enviadas desde una aplicación.
3. **Seguridad**: Permiten implementar controles de acceso, ya que se puede conceder permiso para ejecutar el procedimiento sin otorgar acceso directo a las tablas subyacentes. Esto es crucial en entornos donde la seguridad de los datos es prioritaria.
4. **Mantenimiento**: Al encapsular la lógica de negocio en procedimientos almacenados, es más sencillo realizar cambios. Los cambios se efectúan en el procedimiento sin afectar a la aplicación que lo utiliza.
5. **Reducción de Tráfico en la Red**: Al ejecutar la lógica directamente en el servidor de bases de datos, se reduce la cantidad de datos que deben ser enviados entre el servidor y la aplicación, mejorando así la eficiencia.

**Sintaxis General**

La sintaxis para crear un procedimiento almacenado puede variar ligeramente dependiendo del sistema de gestión de bases de datos (SGBD) utilizado.

|  |
| --- |
| CREATE PROCEDURE NombreDelProcedimiento  [Parámetros]  AS  BEGIN  -- Instrucciones SQL  END; |

* **NombreDelProcedimiento**: El nombre que se le asigna al procedimiento.
* **Parámetros**: Opcionalmente, se pueden definir parámetros de entrada (IN), de salida (OUT) o de entrada/salida (INOUT).
* **Instrucciones SQL**: Aquí se escribe la lógica que se ejecutará cuando se llame al procedimiento.

**Ejemplos de Procedimientos Almacenados**

Ejemplos prácticos de cómo crear procedimientos almacenados en diferentes sistemas de gestión de bases de datos: MySQL, SQL Server, PostgreSQL y, aunque SQLite no los soporta de manera directa, se puede ilustrar el uso de funciones.

|  |
| --- |
| **SQLite**  SQLite no tiene soporte directo para procedimientos almacenados, pero puedes crear funciones definidas por el usuario (UDF) que pueden servir un propósito similar. |
| CREATE FUNCTION ObtenerNombreAlumno(IdAlumno INTEGER)  RETURNS TEXT  BEGIN  DECLARE Nombre TEXT;  SELECT Nombre INTO Nombre FROM Alumnos WHERE IdAlumno = IdAlumno;  RETURN Nombre;  END; |

.

|  |
| --- |
| **MySQL**  En MySQL se puede almacenado un procedimiento de la siguiente manera. |
| DELIMITER //  CREATE PROCEDURE ObtenerInscripcionesAlumno(IN IdAlumno INT)  BEGIN  SELECT A.Nombre AS NombreAlumno, S.NombreAsignatura, I.FechaInscripcion  FROM Alumnos A  JOIN Inscripciones I ON A.IdAlumno = I.IdAlumno  JOIN Asignaturas S ON I.IdAsignatura = S.IdAsignatura  WHERE A.IdAlumno = IdAlumno;  END // |
| DELIMITER ;   * **DELIMITER**: Cambia el delimitador para permitir múltiples sentencias dentro del procedimiento. * **IN IdAlumno INT**: Especifica que el procedimiento acepta un parámetro de entrada, que en este caso es el identificador del alumno. |

.

|  |
| --- |
| **SQL Server**  La sintaxis en SQL Server utilizan algunas palabras clave específicas. |
| CREATE PROCEDURE ObtenerInscripcionesAlumno  @IdAlumno INT  AS  BEGIN  SELECT A.Nombre AS NombreAlumno, S.NombreAsignatura, I.FechaInscripcion  FROM Alumnos A  JOIN Inscripciones I ON A.IdAlumno = I.IdAlumno  JOIN Asignaturas S ON I.IdAsignatura = S.IdAsignatura  WHERE A.IdAlumno = @IdAlumno;  END; |
| * **@IdAlumno**: Indica un parámetro de entrada que se utiliza en la consulta. |

.

|  |
| --- |
| **PostgreSQL**  En PostgreSQL, puedes crear un procedimiento o una función que devuelva resultados. |
| CREATE OR REPLACE FUNCTION ObtenerInscripcionesAlumno(IdAlumno INT)  RETURNS TABLE(NombreAlumno TEXT, NombreAsignatura TEXT, FechaInscripcion DATE) AS $$  BEGIN  RETURN QUERY  SELECT A.Nombre AS NombreAlumno, S.NombreAsignatura, I.FechaInscripcion  FROM Alumnos A  JOIN Inscripciones I ON A.IdAlumno = I.IdAlumno  JOIN Asignaturas S ON I.IdAsignatura = S.IdAsignatura  WHERE A.IdAlumno = IdAlumno;  END;  $$ LANGUAGE plpgsql; |
| * **RETURNS TABLE(...)**: Define la estructura de los resultados devueltos por la función. * **RETURN QUERY**: Utiliza esta declaración para devolver un conjunto de resultados. |

Ejecución de Procedimientos Almacenados

Una vez creado un procedimiento almacenado, se puede ejecutar utilizando la siguiente sintaxis:

|  |
| --- |
| **MySQL**:  CALL ObtenerInscripcionesAlumno(1); |
| **SQL Server**:  EXEC ObtenerInscripcionesAlumno @IdAlumno = 1; |
| **PostgreSQL**:  SELECT \* FROM ObtenerInscripcionesAlumno(1); |

**Consideraciones Adicionales**

1. **Gestión de Errores**: Muchos SGBD permiten manejar errores dentro de procedimientos almacenados, lo que es útil para mantener la integridad de la base de datos y gestionar situaciones excepcionales.
2. **Transacciones**: Los procedimientos pueden incluir transacciones, lo que permite agrupar varias operaciones en una única unidad de trabajo. Esto asegura que todas las operaciones se completen correctamente o que ninguna de ellas se aplique en caso de error.
3. **Optimización**: Es fundamental optimizar las consultas dentro de los procedimientos almacenados para mejorar el rendimiento general. Esto puede incluir el uso de índices, la revisión de planes de ejecución y la eliminación de consultas innecesarias.
4. **Documentación**: Al crear procedimientos almacenados, es importante documentar su funcionalidad y parámetros para que otros desarrolladores puedan entender su propósito y uso en el futuro.

La creación de procedimientos almacenados es una habilidad esencial para cualquier desarrollador de bases de datos. Al permitir la encapsulación de lógica de negocio, la mejora del rendimiento y la seguridad de los datos, los procedimientos almacenados se convierten en herramientas invaluables en el desarrollo de aplicaciones. A través de ejemplos prácticos en varios SGBD, este artículo proporciona una base sólida para que los estudiantes y profesionales comprendan y apliquen esta técnica en sus proyectos. A medida que continúes aprendiendo, explorarás características más avanzadas y optimizaciones que te ayudarán a convertirte en un experto en el manejo de bases de datos.

|  |
| --- |
| SELECT FROM sales-rep ORDER BY commission DESC;  SELECT FROM sales-rep ORDER BY commission DESC, surname ASC ,firs t-name ASC ;  SELECT first-name,surname,commission FROM sales-rep ORDER BY commission DESC LIMIT 1;  SELECT first name,surname,commission FROM sales-rep ORDER BY commission DESC LIMIT 2,3; |

.

|  |
| --- |
| def crear\_procedimiento\_obtener\_stock():  try:  # Conexión a la base de datos MySQL  cursor = conectar\_base\_de\_datos()# sqlite // mysql  if base\_de\_datos == "S":  # Seleccionar productos con stock bajo  cursor.execute("""  SELECT codigo\_producto, cantidad  FROM stock  WHERE cantidad <= 20  """)# -- Puedes ajustar este valor a lo que consideres "stock bajo"  # Obtener los resultados  stock\_bajo = cursor.fetchall()  # Mostrar el listado de productos con stock bajo  if stock\_bajo:  print("Listado de productos con stock bajo:")  for codigo, cantidad in stock\_bajo:  print(f"Código: {codigo}, Cantidad: {cantidad}")  else:  print("No hay productos con stock bajo.")  else:  # Crear el procedimiento almacenado en MySQL  procedimiento = """  CREATE PROCEDURE ObtenerStockBajo()  BEGIN  SELECT codigo\_producto, cantidad  FROM stock  WHERE cantidad <= 20; -- Puedes ajustar este valor a lo que consideres "stock bajo"  END  """  cursor.execute(procedimiento)  print("Procedimiento almacenado 'ObtenerStockBajo' creado exitosamente.")  # Commit para confirmar los cambios en la base de datos  connection.commit()  except mysql.connector.Error as err:  print(f"Error: {err}")  except sqlite3.Error as err:  print(f"Error: {err}")  finally:  cursor.close()  connection.close()  pausa()  #---------------------------------------------------------------------------------------  limpiar() |
| def obtener\_stock\_bajo():  try:  # Conexión a la base de datos MySQL  cursor = conectar\_base\_de\_datos()# sqlite // mysql  # Ejecutar el procedimiento almacenado  cursor.callproc('ObtenerStockBajo')  # Obtener los resultados  for result in cursor.stored\_results():  stock\_bajo = result.fetchall()  # Mostrar el listado de productos con stock bajo  if stock\_bajo:  print("Listado de productos con stock bajo:")  for codigo, cantidad in stock\_bajo:  print(f"Código: {codigo}, Cantidad: {cantidad}")  else:  print("No hay productos con stock bajo.")  except Exception as err:  print(f"Error: {err}")  finally:  cursor.close()  connection.close()  pausa()  #---------------------------------------------------------------------------------------  limpiar() |

.

|  |
| --- |
| def crear\_procedimiento\_obtener\_stock\_variable():  # Conexión a la base de datos MySQL  cursor = conectar\_base\_de\_datos()# sqlite // mysql  # Crear el procedimiento almacenado  procedimiento = """  CREATE PROCEDURE ObtenerStockBajoCantidad(IN desde\_python INT)  BEGIN  -- Crear una tabla temporal con los productos que cumplen la condición  CREATE TEMPORARY TABLE IF NOT EXISTS TempStockBajo AS  SELECT codigo\_producto, cantidad  FROM stock  WHERE cantidad <= desde\_python;  -- Seleccionar y devolver la cantidad de registros  SELECT COUNT(\*) AS cantidad\_registros  FROM TempStockBajo;  -- Seleccionar los datos para mostrarlos  SELECT codigo\_producto, cantidad  FROM TempStockBajo;  END  """  try:  cursor.execute("DROP PROCEDURE IF EXISTS ObtenerStockBajoCantidad") # Eliminar el procedimiento si ya existe  cursor.execute(procedimiento)  print("Procedimiento almacenado 'ObtenerStockBajoCantidad' creado exitosamente.")  except mysql.connector.Error as err:  print(f"Error creando el procedimiento: {err}")  cursor.close()  connection.close()  # Crear el procedimiento almacenado  pausa()  limpiar() |
| def obtener\_stock\_bajo\_variable():  # Conexión a la base de datos MySQL  cursor = conectar\_base\_de\_datos()# sqlite // mysql  # Llamar al procedimiento almacenado  try:  cantidad\_parametro = ""  while not cantidad\_parametro.isdigit():  cantidad\_parametro = input ("ingrese la cantidad minima de productos:")  cantidad\_parametro = int(cantidad\_parametro )  cursor.callproc('ObtenerStockBajoCantidad', [cantidad\_parametro])  # Obtener el resultado  for resultado in cursor.stored\_results():  filas = resultado.fetchall()  for fila in filas:  print(f"Cantidad de registros con stock <= {cantidad\_parametro}: {fila[0]}")  except mysql.connector.Error as err:  print(f"Error llamando al procedimiento: {err}")  finally:  cursor.close()  connection.close()  pausa()  limpiar() |
| La diferencia principal entre una tabla temporal y una vista en bases de datos como MySQL radica en su naturaleza, uso y almacenamiento. Aquí te explico las características y diferencias más importantes:  **Tabla Temporal**   1. **Naturaleza**:    * Es una tabla real, pero **temporal**, lo que significa que existe solo durante la sesión del usuario o mientras se ejecuta una transacción.    * Una vez que la sesión termina o la tabla temporal se elimina explícitamente, los datos también se eliminan. 2. **Almacenamiento**:    * Se almacena físicamente en el disco o en memoria, dependiendo de la configuración del servidor.    * Puede tener índices, claves primarias, y otras características similares a una tabla permanente. 3. **Contenido**:    * Contiene datos específicos que pueden ser manipulados (insertados, actualizados o eliminados) durante su existencia.    * Los datos son independientes y no se actualizan automáticamente si cambian los datos de las tablas subyacentes. 4. **Usos comunes**:    * Para almacenar resultados intermedios o datos temporales que se necesiten procesar durante una sesión.    * Útil para operaciones complejas que requieren cálculos o transformaciones de datos intermedios. 5. **Limitaciones**:    * Solo es accesible durante la sesión en la que se creó.    * No está disponible para otros usuarios o sesiones.   **Vista**   1. **Naturaleza**:    * Es una **consulta guardada**, no almacena datos directamente. Es un objeto que presenta los datos desde una o varias tablas subyacentes según una consulta predefinida.    * Actúa como una "ventana" hacia los datos subyacentes. 2. **Almacenamiento**:    * No almacena físicamente los datos; siempre que se consulta, la base de datos ejecuta la consulta subyacente para generar el resultado.    * No se pueden definir índices directamente en una vista. 3. **Contenido**:    * Muestra los datos actuales de las tablas subyacentes. Si los datos en las tablas cambian, el contenido de la vista también cambia automáticamente.    * Es de solo lectura si la consulta incluye combinaciones complejas o funciones agregadas. 4. **Usos comunes**:    * Simplificar consultas complejas, ya que encapsulan la lógica de la consulta.    * Restringir el acceso a ciertas columnas o filas para usuarios específicos (control de seguridad).    * Proporcionar una abstracción lógica de los datos. 5. **Limitaciones**:    * No es adecuada para operaciones complejas de escritura o para almacenar datos temporales.    * El rendimiento puede ser más lento si la consulta subyacente es compleja y se ejecuta frecuentemente.  |  |  |  | | --- | --- | --- | | **Comparación Tabla Temporal vs Vista** | | | | **Característica** | **Tabla Temporal** | **Vista** | | **Persistencia** | Existe solo durante la sesión. | Persiste hasta que se elimina. | | **Almacenamiento de datos** | Almacena datos físicamente. | No almacena datos, solo consulta. | | **Contenido** | No cambia a menos que se modifique. | Refleja siempre los datos actuales. | | **Uso para escritura** | Puede ser escrita (INSERT, UPDATE, DELETE). | Generalmente de solo lectura. | | **Uso compartido** | No puede ser compartida entre sesiones. | Puede ser usada por múltiples usuarios. | | **Actualización automática** | No se actualiza automáticamente. | Se actualiza automáticamente. | | **Velocidad** | Puede ser más rápida para datos pequeños. | Puede ser más lenta si la consulta es compleja. |   . |

.

|  |
| --- |
| Inserción de grandes cantidades de datos |

.

Inserción desde un archivo de texto con LOAD DATA

Una ultima forma de insertar datos (y la mejor en caso de que se necesiten introducir grandes cantidades de datos a la vez) consiste en utilizar la instruccion LOAD DATA, de la siguiente forma: mys ql> LOAD DATA LOCAL INE'ILE " sales-rep . sql 'I INTO TABLE sales-rep ; El formato del archivo de datos debe ser correcto, sin excepciones.

En este caso, en el que estamos utilizando 10s valores predeterminados, el archivo de texto incluye cada registro en una nueva linea y cada campo se separa por medio de un tabulador.

Si asumimos que el caracter \ t representa un tabulador y que cada linea termina en un caracter de nueva linea, el archivo presentaria este aspecto: Se utiliza una variacion de esta secuencia para restaurar copias de seguridad (comentadas en un capitulo posterior).

Esta instruccion resulta incluso mas eficaz que una instruccion INSERT para introducir varios registros.

La palabra clave LOCAL indica a1 servidor que el archivo se encuentra incluido en el equipo cliente (el equipo desde el que se establece la conexion). Si se omite, MySQL buscara el archivo en el servidor de la base de datos.

De manera predeterminada, LOAD DATA asume que 10s valores se encuentran en el mismo orden que en la definicion de la tabla, que cada campo esta separado por un tabulador y que cada registro se incluye en una linea.

|  |
| --- |
| LOAD DATA |
| 1. **SQLite**  En SQLite, puedes usar el módulo sqlite3 en Python o importar desde un archivo CSV directamente en la interfaz de línea de comandos. |
| import sqlite3  # Conectar a la base de datos conexion = sqlite3.connect("mi\_base\_de\_datos.db") cursor = conexion.cursor()  # Crear tabla de ejemplo cursor.execute('''CREATE TABLE IF NOT EXISTS sales\_rep ( id INTEGER PRIMARY KEY, nombre TEXT, ventas INTEGER  )''') # Cargar datos desde un archivo CSV with open("sales\_rep.csv", "r") as archivo: for linea in archivo: datos = linea.strip().split("\t") cursor.execute("INSERT INTO sales\_rep (nombre, ventas) VALUES (?, ?)", datos)  # Confirmar cambios y cerrar conexion.commit() conexion.close() |
| 2. **MySQL**  MySQL usa la instrucción LOAD DATA, lo que permite importar datos desde un archivo. Asegúrate de que los permisos de LOCAL estén habilitados y que el archivo esté correctamente formateado. |
| LOAD DATA LOCAL INFILE 'sales\_rep.csv'  INTO TABLE sales\_rep  FIELDS TERMINATED BY '\t'  LINES TERMINATED BY '\n'  (nombre, ventas); |
| 3. **SQL Server**  En SQL Server, puedes usar BULK INSERT, que es similar a LOAD DATA. Este comando asume que el archivo está en el servidor, y el delimitador por defecto es la coma, pero puedes especificar el tabulador como delimitador. |
| BULK INSERT sales\_rep  FROM 'C:\ruta\al\archivo\sales\_rep.csv'  WITH (  FIELDTERMINATOR = '\t',  ROWTERMINATOR = '\n'  ); |
| 4. **PostgreSQL**  PostgreSQL permite importar archivos CSV o delimitados por tabuladores usando COPY. Aquí te muestro cómo hacerlo desde un archivo delimitado por tabuladores en el sistema de archivos: |
| COPY sales\_rep (nombre, ventas)  FROM '/ruta/al/archivo/sales\_rep.csv'  DELIMITER '\t'  CSV HEADER; |

.

|  |
| --- |
| Funciones de Agregación |

.

### Funciones de agregación (COUNT, AVG, MIN, MAX, SUM)

Las **funciones de agregación** en SQL permiten realizar cálculos sobre un conjunto de filas y devolver un único resultado. Son especialmente útiles para obtener información como conteos, promedios, valores mínimos y máximos de los datos en una tabla.

1. **Contar (**COUNT**)**: Útil para obtener el número de filas en un conjunto de datos.
   * COUNT(\*): Cuenta todas las filas, incluidas las duplicadas y valores nulos.
   * COUNT(columna): Cuenta solo los valores no nulos de una columna específica.
   * COUNT(DISTINCT columna): Cuenta el número de valores únicos en una columna.
2. **Promedio (**AVG**)**: Calcula el valor promedio de una columna numérica.
3. **Mínimo (**MIN**) y Máximo (**MAX**)**: Recupera el valor mínimo o máximo en una columna específica.
4. **Suma (**SUM**)**: Suma todos los valores en una columna numérica.

Estas funciones también permiten hacer cálculos directos en consultas SQL, como sumar valores o modificar campos.

|  |
| --- |
| Cada función realiza una operación específica en los datos de la tabla sales\_rep:   * **COUNT**: Muestra el número de filas o valores distintos. * **AVG, MIN, MAX y SUM**: Calculan el promedio, el valor mínimo, el máximo y el total de la columna commission, respectivamente. * **Operación aritmética**: La expresión commission \* 1.01 incrementa cada comisión en un 1% sobre el valor original. |
| 1. **SQLite** |
| -- Contar el número total de filas en la tabla:  SELECT COUNT(\*) FROM sales\_rep;  -- Contar el número de apellidos no nulos:  SELECT COUNT(surname) FROM sales\_rep;  -- Contar el número de apellidos únicos:  SELECT COUNT(DISTINCT surname) FROM sales\_rep;  -- Calcular el promedio de comisiones:  SELECT AVG(commission) FROM sales\_rep;  -- Encontrar el valor mínimo de las comisiones:  SELECT MIN(commission) FROM sales\_rep;  -- Calcular la suma total de comisiones:  SELECT SUM(commission) FROM sales\_rep;  -- Incrementar la comisión de cada comercial en un 1%:  SELECT nombre, surname, commission \* 1.01 AS comision\_incrementada FROM sales\_rep; |
| 2. **MySQL** |
| -- Contar el número total de filas en la tabla:  SELECT COUNT(\*) FROM sales\_rep;  -- Contar el número de apellidos no nulos:  SELECT COUNT(surname) FROM sales\_rep;  -- Contar el número de apellidos únicos:  SELECT COUNT(DISTINCT surname) FROM sales\_rep;  -- Calcular el promedio de comisiones:  SELECT AVG(commission) FROM sales\_rep;  -- Encontrar el valor mínimo de las comisiones:  SELECT MIN(commission) FROM sales\_rep;  -- Calcular la suma total de comisiones:  SELECT SUM(commission) FROM sales\_rep;  -- Incrementar la comisión de cada comercial en un 1%:  SELECT nombre, surname, commission \* 1.01 AS comision\_incrementada FROM sales\_rep; |
| 3. **SQL Server** |
| -- Contar el número total de filas en la tabla:  SELECT COUNT(\*) FROM sales\_rep;  -- Contar el número de apellidos no nulos:  SELECT COUNT(surname) FROM sales\_rep;  -- Contar el número de apellidos únicos:  SELECT COUNT(DISTINCT surname) FROM sales\_rep;  -- Calcular el promedio de comisiones:  SELECT AVG(commission) FROM sales\_rep;  -- Encontrar el valor mínimo de las comisiones:  SELECT MIN(commission) FROM sales\_rep;  -- Calcular la suma total de comisiones:  SELECT SUM(commission) FROM sales\_rep;  -- Incrementar la comisión de cada comercial en un 1%:  SELECT nombre, surname, commission \* 1.01 AS comision\_incrementada FROM sales\_rep; |
| 4. **PostgreSQL** |
| -- Contar el número total de filas en la tabla:  SELECT COUNT(\*) FROM sales\_rep;  -- Contar el número de apellidos no nulos:  SELECT COUNT(surname) FROM sales\_rep;  -- Contar el número de apellidos únicos:  SELECT COUNT(DISTINCT surname) FROM sales\_rep;  -- Calcular el promedio de comisiones:  SELECT AVG(commission) FROM sales\_rep;  -- Encontrar el valor mínimo de las comisiones:  SELECT MIN(commission) FROM sales\_rep;  -- Calcular la suma total de comisiones:  SELECT SUM(commission) FROM sales\_rep;  -- Incrementar la comisión de cada comercial en un 1%:  SELECT nombre, surname, commission \* 1.01 AS comision\_incrementada FROM sales\_rep; |

.

|  |
| --- |
| Modificación de la Estructura de una Tabla |

.

### Modificación de la Estructura de una Tabla con ALTER TABLE

La instrucción DDL (Lenguaje de Definición de Datos) ALTER TABLE en SQL permite modificar la estructura de una tabla existente, incluyendo las siguientes operaciones comunes:

1. **Agregar columnas (**ADD**)**: Permite añadir una columna nueva a una tabla existente.
2. **Modificar definiciones de columna (**MODIFY **o** CHANGE**)**: Permite cambiar el tipo de datos de una columna o su nombre.
   * CHANGE: Cambia tanto el nombre como el tipo de datos de una columna.
   * MODIFY: Cambia solo el tipo de datos, manteniendo el nombre.
3. **Renombrar tablas (**RENAME**)**: Permite cambiar el nombre de una tabla.
4. **Eliminar columnas (**DROP**)**: Permite eliminar una columna de la tabla.

Estas operaciones son útiles para ajustar la estructura de la base de datos a nuevas necesidades. Sin embargo, es importante definir la estructura correctamente desde el inicio para minimizar cambios que podrían afectar la integridad de los datos y la estabilidad del sistema.

|  |
| --- |
| ALTER TABLE Se utilizará una tabla ficticia llamada sales\_rep con las columnas iniciales id, nombre, surname y commission. |
| 1. SQLite Nota: SQLite es más limitado en cuanto a las modificaciones posibles con ALTER TABLE. Solo permite agregar nuevas columnas y renombrar tablas. |
| -- Agregar una columna de fecha de contratación:  ALTER TABLE sales\_rep ADD COLUMN date\_joined DATE;  -- Renombrar la tabla:  ALTER TABLE sales\_rep RENAME TO cash\_flow\_specialist;  -- Eliminar una columna en SQLite requiere recrear la tabla sin esa columna,  -- copiando los datos relevantes. |
| 2. MySQL |
| -- Agregar una columna para la fecha de contratación:  ALTER TABLE sales\_rep ADD COLUMN date\_joined DATE;  -- Agregar una columna para el año de nacimiento:  ALTER TABLE sales\_rep ADD COLUMN year\_born YEAR;  -- Cambiar el año de nacimiento por una fecha completa de nacimiento:  ALTER TABLE sales\_rep CHANGE year\_born birthday DATE;  -- Modificar el tipo de datos de `commission` sin cambiar su nombre:  ALTER TABLE sales\_rep MODIFY commission DECIMAL(10,2);  -- Renombrar la tabla:  ALTER TABLE sales\_rep RENAME TO cash\_flow\_specialist;  -- Agregar una columna de valor adicional:  ALTER TABLE cash\_flow\_specialist ADD COLUMN enhancement\_value INT;  -- Cambiar el nombre de la tabla de nuevo:  ALTER TABLE cash\_flow\_specialist RENAME TO sales\_rep;  -- Eliminar la columna de valor adicional:  ALTER TABLE sales\_rep DROP COLUMN enhancement\_value; |
| 3. SQL Server |
| -- Agregar una columna para la fecha de contratación:  ALTER TABLE sales\_rep ADD date\_joined DATE;  -- Agregar una columna para el año de nacimiento:  ALTER TABLE sales\_rep ADD year\_born INT;  -- Cambiar el año de nacimiento por una fecha completa de nacimiento:  ALTER TABLE sales\_rep ALTER COLUMN year\_born DATE;  -- Modificar el tipo de datos de `commission` sin cambiar su nombre:  ALTER TABLE sales\_rep ALTER COLUMN commission DECIMAL(10,2);  -- Renombrar la tabla:  EXEC sp\_rename 'sales\_rep', 'cash\_flow\_specialist';  -- Agregar una columna de valor adicional:  ALTER TABLE cash\_flow\_specialist ADD enhancement\_value INT;  -- Cambiar el nombre de la tabla de nuevo:  EXEC sp\_rename 'cash\_flow\_specialist', 'sales\_rep';  -- Eliminar la columna de valor adicional:  ALTER TABLE sales\_rep DROP COLUMN enhancement\_value; |
| 4. PostgreSQL |
| -- Agregar una columna para la fecha de contratación:  ALTER TABLE sales\_rep ADD COLUMN date\_joined DATE;  -- Agregar una columna para el año de nacimiento:  ALTER TABLE sales\_rep ADD COLUMN year\_born INT;  -- Cambiar el año de nacimiento por una fecha completa de nacimiento:  ALTER TABLE sales\_rep RENAME COLUMN year\_born TO birthday;  ALTER TABLE sales\_rep ALTER COLUMN birthday TYPE DATE;  -- Modificar el tipo de datos de `commission` sin cambiar su nombre:  ALTER TABLE sales\_rep ALTER COLUMN commission TYPE NUMERIC(10,2);  -- Renombrar la tabla:  ALTER TABLE sales\_rep RENAME TO cash\_flow\_specialist;  -- Agregar una columna de valor adicional:  ALTER TABLE cash\_flow\_specialist ADD COLUMN enhancement\_value INT;  -- Cambiar el nombre de la tabla de nuevo:  ALTER TABLE cash\_flow\_specialist RENAME TO sales\_rep;  -- Eliminar la columna de valor adicional:  ALTER TABLE sales\_rep DROP COLUMN enhancement\_value; |

.

|  |
| --- |
| Modificación de Registros y Estructura de Tablas en SQL |

.

Para administrar datos en bases de datos, SQL nos permite modificar, insertar, eliminar, y consultar registros. Además, SQL facilita el cambio de estructura de las tablas usando distintas instrucciones. A continuación, detallamos cómo utilizar algunas de estas instrucciones, junto con ejemplos en MySQL, SQLite, SQL Server y PostgreSQL.

**Modificación de Registros con UPDATE**

La instrucción UPDATE permite actualizar registros específicos en una tabla. Es importante añadir una condición para evitar modificar todos los registros de forma accidental.

**Ejemplo:**

Supongamos que el empleado con employee\_number igual a 1 ha incrementado su comisión al 12%.

|  |
| --- |
| UPDATE sales\_rep  SET commission = 12  WHERE employee\_number = 1; |

#### Consideraciones:

* **Advertencia**: Si no se incluye la cláusula WHERE, UPDATE aplicará la modificación a todos los registros de la tabla.
* **Instrucciones complementarias**: Las operaciones de INSERT, SELECT, UPDATE, y DELETE son las más comunes para la manipulación de datos y forman el Lenguaje de Manipulación de Datos (DML).

### Eliminación de Tablas y Bases de Datos

El Lenguaje de Definición de Datos (DDL) incluye instrucciones para crear y eliminar tablas o bases de datos. Estas instrucciones deben usarse con precaución debido a sus efectos permanentes.

|  |
| --- |
| Ejemplo para eliminar una tabla de una base de datos: |
| SQLite, MySQL, SQL Server y PostgreSQL |
| DROP TABLE IF EXISTS mi\_tabla; |

.

|  |
| --- |
| Ejemplo para eliminar una base de datos: |
| SQLite no soporta la instrucción DROP DATABASE porque se maneja sobre un solo archivo de base de datos.  Ver la librería os para eliminar el archivo |
| MySQL, SQL Server y PostgreSQL |
| DROP DATABASE IF EXISTS mi\_base\_de\_datos; |
| Nota: La eliminación de una tabla o base de datos no requiere confirmación; cualquier dato almacenado se perderá permanentemente. |

### Modificación de la Estructura de Tablas con ALTER

#### Agregar una Columna

La instrucción ALTER permite agregar nuevas columnas a una tabla existente. Supongamos que necesitamos una columna para almacenar la fecha de contratación de los empleados.

* **Nota**: El tipo DATE almacena valores en el formato AAAA-MM-DD. Otros formatos requieren ajustes adicionales para la compatibilidad.

#### Cambiar el Nombre de una Columna

* Cambiar el nombre de una columna se realiza mediante ALTER junto con CHANGE o MODIFY. Por ejemplo, si deseamos renombrar sales\_rep a cash\_flow\_specialist.

#### Modificar una Definición de Columna

* Para cambiar la definición de una columna existente sin cambiar su nombre, usamos MODIFY. Si necesitamos modificar el nombre y la definición, utilizamos CHANGE.

#### Ejemplo General de Modificación en Distintos Sistemas

* Cada sistema de gestión de bases de datos tiene ligeras variaciones en la sintaxis para modificar estructuras de tablas. A continuación, se presentan ejemplos básicos en los sistemas más comunes.

|  |
| --- |
| ALTER |
| 1. ****SQLite**** En SQLite, algunas opciones como ALTER COLUMN pueden tener limitaciones y requerir soluciones alternativas. |
| -- Agregar una columna  **ALTER TABLE** sales\_rep A**DD COLUMN** date\_joined DATE; |
| 2. **MySQL** |
| -- Agregar una columna  ALTER TABLE sales\_rep **ADD** date\_joined DATE;  -- Cambiar el nombre de una columna  **ALTER TABLE** sales\_rep **CHANGE** year\_born birthday DATE; |
| 3. **SQL Server** |
| -- Agregar una columna  **ALTER TABLE** sales\_rep **ADD** date\_joined DATE;  -- Cambiar el nombre de una columna  **EXEC** sp\_rename 'sales\_rep.year\_born', 'birthday', 'COLUMN'; |
| 4. **PostgreSQL**  PostgreSQL es muy flexible con ALTER TABLE. |
| -- Agregar una columna  **ALTER TABLE** sales\_rep ADD **COLUMN** date\_joined DATE;  -- Cambiar el nombre de una columna  **ALTER TABLE** sales\_rep **RENAME COLUMN** year\_born **TO** birthday; |

.Actualización, eliminación y funciones de fecha en SQL

En esta sección exploraremos cómo modificar registros existentes, eliminar columnas, y trabajar con funciones de fecha en SQL. Abarcaremos cada acción utilizando ejemplos en varios sistemas de gestión de bases de datos: SQLite, MySQL, SQL Server y PostgreSQL.

## Modificar registros existentes

Para actualizar valores de un registro, utilizamos la instrucción UPDATE. Esta instrucción es parte del Lenguaje de Manipulación de Datos (DML) en SQL, el cual permite realizar operaciones de modificación de datos, como INSERT, DELETE, SELECT, y UPDATE.

|  |
| --- |
| UPDATE |
| ****SQLite, MySQL, SQL Server y PostgreSQL**** |
| -- Modificar datos de una fila específica  UPDATE sales\_rep SET commission = 12 WHERE employee\_number = 1; |

.

En este caso, se actualiza el valor de la comisión de un empleado específico en la tabla sales\_rep. Es fundamental incluir la cláusula WHERE para evitar modificar todos los registros.

## Eliminación de una columna

Para eliminar una columna de una tabla, se utiliza la instrucción ALTER TABLE ... DROP COLUMN. En este ejemplo, eliminaremos la columna enhancement\_value de la tabla sales\_rep.

### Ejemplo en MySQL:

ALTER TABLE sales\_rep DROP COLUMN enhancement\_value;

Este comando elimina la columna de manera permanente, así que debe ejecutarse con precaución.

### Ejemplo en otros sistemas:

|  |
| --- |
| eliminar columnas |
| 1. **SQLite**  SQLite no permite eliminar columnas directamente. Para hacerlo, necesitas crear una nueva tabla sin la columna que deseas eliminar, copiar los datos de la tabla original a la nueva y luego renombrar la nueva tabla. |
| -- Crear una nueva tabla sin la columna enhancement\_value  CREATE TABLE new\_sales\_rep AS  SELECT id, name, other\_columns -- lista de columnas, excluyendo enhancement\_value  FROM sales\_rep;  -- Eliminar la tabla original  DROP TABLE sales\_rep;  -- Renombrar la nueva tabla  ALTER TABLE new\_sales\_rep RENAME TO sales\_rep; |
| **MySQL**, **SQL Server** y **PostgreSQL** |
| ALTER TABLE sales\_rep DROP COLUMN enhancement\_value; |

## Uso de funciones de fecha en SQL

### Consulta de fechas en la tabla

Para visualizar los campos date\_joined y birthday, basta con realizar una consulta SELECT:

|  |
| --- |
| SELECT date\_joined, birthday FROM sales\_rep; |

### Especificar formato de fecha

Se permite especificar formatos especiales al recuperar fechas mediante la función DATE\_FORMAT(). En el siguiente ejemplo, se muestra el formato MM/DD/YYYY para el campo date\_joined:

|  |
| --- |
| Equivalente a DATE\_FORMAT para obtener el formato de fecha MM/DD/YYYY. |
| 1. SQLite En SQLite, puedes usar la función strftime para formatear fechas: |
| SELECT strftime('%m/%d/%Y', date\_joined) FROM sales\_rep WHERE employee\_number = 1; |
| 2. MySQL En MySQL, puedes utilizar DATE\_FORMAT como en tu ejemplo: |
| SELECT DATE\_FORMAT(date\_joined, '%m/%d/%Y') FROM sales\_rep WHERE employee\_number = 1; |
| 3. SQL Server En SQL Server, puedes usar la función FORMAT (disponible en SQL Server 2012 y versiones posteriores) o CONVERT: |
| -- Usando FORMAT  SELECT FORMAT(date\_joined, 'MM/dd/yyyy') FROM sales\_rep WHERE employee\_number = 1;  -- Usando CONVERT  SELECT CONVERT(VARCHAR(10), date\_joined, 101) FROM sales\_rep WHERE employee\_number = 1; |
| 4. PostgreSQL En PostgreSQL, puedes usar TO\_CHAR para formatear fechas: |
| SELECT TO\_CHAR(date\_joined, 'MM/DD/YYYY') FROM sales\_rep WHERE employee\_number = 1; |

En esta consulta:

* %m devuelve el mes (01-12).
* %d devuelve el día (01-31).
* %Y devuelve el año en formato de cuatro dígitos.

|  |
| --- |
| Recuperación de la fecha y hora actual Permiten recuperar tanto la fecha como la hora en formato AAAA-MM-DD HH:MM:SS. |
| **SQLite**: |
| SELECT datetime('now'), date('now'); |
| **MySQL**: |
| SELECT NOW(), CURRENT\_DATE(); |
| **SQL Server**: |
| SELECT GETDATE(), CAST(GETDATE() AS DATE); |
| **PostgreSQL**: |
| SELECT NOW(), CURRENT\_DATE; |

## .

|  |
| --- |
| Funciones adicionales para manejo de fechas Funciones para recuperar partes específicas de una fecha, como el año, mes y día.  Estas funciones permiten obtener el año (YEAR()), el mes (MONTH()) y el día (DAYOFMONTH()) de una fecha almacenada. |
| **SQLite**: |
| SELECT strftime('%Y', birthday) AS year, strftime('%m', birthday) AS month, strftime('%d', birthday) AS day FROM sales\_rep; |
| **MySQL**: |
| SELECT YEAR(birthday) FROM sales\_rep;  SELECT MONTH(birthday), DAYOFMONTH(birthday) FROM sales\_rep; |
| **SQL Server**: |
| SELECT YEAR(birthday) AS year, MONTH(birthday) AS month, DAY(birthday) AS day FROM sales\_rep; |
| **PostgreSQL**: |
| SELECT EXTRACT(YEAR FROM birthday) AS year, EXTRACT(MONTH FROM birthday) AS month, EXTRACT(DAY FROM birthday) AS day FROM sales\_rep; |

### Creación de Consultas Más Avanzadas

#### 1. ****Encabezados de Columnas con Alias (****AS****)****

* Para mejorar la legibilidad de las consultas, se pueden asignar alias a las columnas usando AS.
* **Ejemplo**:

|  |
| --- |
| SELECT surname, first\_name, MONTH(birthday) AS month, DAYOFMONTH(birthday) AS day  FROM sales\_rep  ORDER BY month; |

**Resultado**:

| surname | first\_name | month | day |
| --- | --- | --- | --- |
| Rive | Mongane | 11 | 4 |
| Rive | Sol | 3 | 18 |
| Serote | Mike | 6 | 18 |
| Gordimer | Charlene | 11 | 30 |

#### 2. ****Concatenación de Columnas con**** CONCAT

* Combina los resultados de varias columnas en un solo campo de salida.
* **Ejemplo**:

|  |
| --- |
| SELECT CONCAT(first\_name, ' ', surname) AS name, MONTH(birthday) AS month, DAYOFMONTH(birthday) AS day  FROM sales\_rep  ORDER BY month; |

**Resultado**:

| name | month | day |
| --- | --- | --- |
| Mongane Rive | 11 | 4 |
| Sol Rive | 3 | 18 |
| Mike Serote | 6 | 18 |
| Charlene Gordimer | 11 | 30 |

#### 3. ****Uso de Funciones de Fecha****

* Se pueden realizar cálculos con fechas, como el cálculo de la edad a partir de la fecha de nacimiento.
* Para obtener una edad aproximada:

|  |
| --- |
| SELECT YEAR(NOW()) - YEAR(birthday) AS age FROM sales\_rep; |

* Para calcular la edad exacta considerando el mes y día:

|  |
| --- |
| SELECT  YEAR(NOW()) - YEAR(birthday) -  (DATE\_FORMAT(NOW(), '%m-%d') < DATE\_FORMAT(birthday, '%m-%d')) AS age  FROM sales\_rep; |

#### 4. ****Combinaciones de Tablas (Uniones)****

* Las bases de datos relacionales permiten relacionar y combinar datos de diferentes tablas usando claves compartidas.
* **Ejemplo de combinación**:

|  |
| --- |
| SELECT sales\_rep.first\_name, sales\_rep.surname, sales.value  FROM sales  JOIN sales\_rep ON sales.sales\_rep = sales\_rep.employee\_number; |

#### 5. ****Ejemplo Completo de Combinación para Filtrar por Vendedor Específico****

* Consulta para recuperar todas las ventas de un vendedor específico:

|  |
| --- |
| SELECT sales.code, sales.customer, sales.value  FROM sales  JOIN sales\_rep ON sales.sales\_rep = sales\_rep.employee\_number  WHERE sales\_rep.first\_name = 'Sol' AND sales\_rep.surname = 'Rive'; |

#### 6. ****Evitar Ambigüedades en los Nombres de Columnas****

* En casos donde el nombre de columna se repite en varias tablas, especifica el nombre de la tabla antes de la columna (ejemplo: sales.employee\_number).
* **Ejemplo de uso en consultas**:

|  |
| --- |
| SELECT sales.code, sales.customer, sales.value  FROM sales  JOIN sales\_rep ON sales.sales\_rep = sales\_rep.employee\_number; |

### 1. Obtener el Componente MMDD de la Fecha

En SQL, podemos extraer los últimos caracteres de una fecha para obtener el mes y día usando la función RIGHT(). Esto es útil cuando queremos manipular o comparar partes de una fecha.

|  |
| --- |
| SELECT RIGHT(CURRENT\_DATE, 5) AS "Fecha Actual (MM-DD)", RIGHT(birthday, 5) AS "Fecha de Cumpleaños (MM-DD)"  FROM sales\_rep; |

* RIGHT(CURRENT\_DATE, 5): Toma los últimos cinco caracteres de CURRENT\_DATE, que representan el mes y día en formato MM-DD.
* RIGHT(birthday, 5): Toma los últimos cinco caracteres del campo birthday para obtener el mes y día de cumpleaños en el mismo formato.

### 2. Calcular la Suma de las Ventas Totales

Para obtener la suma total de ventas, utilizamos la función SUM() en el campo value de la tabla sales.

|  |
| --- |
| SELECT SUM(value) AS "Ventas Totales"  FROM sales; |

Este cálculo nos proporciona una sola fila con el total de todas las ventas realizadas.

### 3. Agrupar Ventas por Comercial

Para obtener las ventas agrupadas por cada comercial, usamos la cláusula GROUP BY. Esto permite sumar todas las ventas asociadas a cada sales\_rep.

|  |
| --- |
| SELECT sales\_rep, SUM(value) AS "Ventas por Comercial"  FROM sales  GROUP BY sales\_rep; |

En esta consulta:

* GROUP BY sales\_rep agrupa las filas según el comercial (sales\_rep).
* SUM(value) calcula el total de ventas para cada comercial.

### 4. Error al Omitir GROUP BY con una Función de Agregado

Si intentamos utilizar una función de agregación, como SUM(), sin incluir GROUP BY, se genera un error, ya que SQL no sabe cómo manejar la combinación de columnas normales con columnas agregadas. Esto nos recuerda que siempre se debe especificar cómo agrupar los datos al usar funciones de agregación.

### 5. Ordenar las Ventas por Comercial de Mayor a Menor

Para ordenar las ventas totales de cada comercial en orden descendente, podemos añadir la cláusula ORDER BY:

|  |
| --- |
| SELECT sales\_rep, SUM(value) AS "Ventas por Comercial"  FROM sales  GROUP BY sales\_rep  ORDER BY SUM(value) DESC; |
| * ORDER BY SUM(value) DESC ordena las ventas de mayor a menor. |

### 6. Encontrar al Comercial con Menos Ventas

Si queremos identificar al comercial con el menor número de ventas, podemos utilizar la función COUNT() junto con LIMIT para devolver solo el comercial con menos registros en la tabla de ventas.

|  |
| --- |
| SELECT sales\_rep, COUNT(\*) AS "Cantidad de Ventas"  FROM sales  GROUP BY sales\_rep  ORDER BY COUNT(\*) ASC  LIMIT 1; |

* COUNT(\*): Cuenta la cantidad de registros de ventas para cada comercial.
* ORDER BY COUNT(\*) ASC: Ordena los comerciales de menor a mayor cantidad de ventas.
* LIMIT 1: Devuelve solo el comercial con la menor cantidad de ventas.

### 7. Vincular el Nombre del Comercial con la Venta de Menor Cantidad

Para enlazar el nombre del comercial con sus ventas, utilizamos una consulta que une la tabla sales con sales\_rep, de manera que podamos mostrar el nombre del comercial junto con el conteo de ventas.

|  |
| --- |
| SELECT first\_name, surname, sales\_rep, COUNT(\*) AS "Cantidad de Ventas"  FROM sales  JOIN sales\_rep ON sales.sales\_rep = sales\_rep.employee\_number  GROUP BY sales\_rep, first\_name, surname  ORDER BY COUNT(\*) ASC  LIMIT 1; |
| En esta consulta:   * JOIN sales\_rep ON sales.sales\_rep = sales\_rep.employee\_number: Une la tabla sales con sales\_rep usando la columna employee\_number como referencia. * ORDER BY COUNT(\*) ASC LIMIT 1: Devuelve el comercial con el menor número de ventas. |

Este proceso es una excelente práctica para afianzar habilidades en SQL y en el manejo de datos a través de consultas cada vez más avanzadas.