Base de Datos Avanzadas

Preguntas de examen confirmadas

¿Cuál fue el principal objetivo del modelo relacional propuesto por Edgar Codd?
Organizar datos en tablas normalizadas

¿Qué característica diferenciaba a los sistemas de base de de datos a objetos de los relacionales en la década de 1980?
Tratamiento de datos como objetos

¿Qué innovaciones introdujeron los sistemas SQL en la gestión de bases de datos en los años 70?
Estandarización de un lenguaje de consulta estructurado

¿Cuál fue una de las principales críticas al modelo de navegación CODASYL?
La ineficiencia en la búsqueda de registros

¿Qué técnica se introdujo en 1989 para mejorar el rendimiento de las bases de datos?
Indización de datos replicados en bases de datos temporales.

1. Evolución de los sistemas de gestión de bases de datos relacionales

- Las bases de datos han estado en uso desde los primeros días de las computadoras electrónicas.
- A diferencia de los sistemas modernos, que se pueden aplicar a datos y necesidades muy diferentes, la mayor parte de los sistemas originales estaban enfocados a bases de datos específicas y pensados para ganar velocidad a costa de perder flexibilidad

1.1 Sistemas de navegación (1960)

- Según las computadoras fueron ganando velocidad y capacidad, aparecieron sistemas de bases de datos de propósito general.
- A mediados de 1960 ya había algunos sistemas en uso. Apareció el interés en obtener un estándar y *Charles Bachman* autor de uno de los primeros SGBD, el **Integrated Data Store (IDS)** — fundó el Database Task Group dentro de CODASYL, el grupo responsable de la creación y estandarización de COBOL. En 1971 publicaron su estándar, que pasó a ser conocido como la «aproximación CODASYL»
- La estrategia de CODASYL estaba basada en la navegación manual por un conjunto de datos enlazados en red. Cuando se arrancaba la base de datos, el programa devolvía un enlace al primer registro de la base de datos, el cual a su vez contenía punteros a otros datos. Para encontrar un registro concreto el programador debía ir siguiendo punteros hasta llegar al registro buscado.

1.2 Sistemas relacionales (1970)

Edgar Codd trabajaba en IBM, en una de esas oficinas periféricas que estaba dedicada principalmente al desarrollo de discos duros. Estaba descontento con el modelo de navegación CODASYL, principalmente con la falta de operación de búsqueda.

En 1970 escribió algunos artículos en los que perfilaba una nueva aproximación que culminó en el documento «A Relational Model of Data for Large Shared Data Banks»

Sistema para almacenar y trabajar con grandes bases de datos.
En vez de almacenar registros de tipo arbitrario en una lista encadenada como en CODASYL, la idea de Codd era usar una «tabla» de registros de tamaño fijo.

Una lista encadenada tiene muy poca eficiencia al almacenar datos dispersos donde algunos de los datos de un registro pueden dejarse en blanco.

El modelo relacional resuelve esto dividiendo los datos en una serie de tablas o relaciones normalizadas, en las que los elementos optativos han sido extraídos de la tabla principal para que ocupen espacio sólo si lo necesitan.

En este modelo relacional los registros relacionados se enlazan con una «clave»

1.3 Sistemas SQL (finales década de los 70)

- IBM comenzó a trabajar a principios de 1970 en un prototipo lejanamente basado en los conceptos de Codd llamándolo System R.
- La primera versión estuvo lista en 1974 o 1975, y comenzó así el trabajo en sistemas multitabla, en los que los datos podían disgregarse de modo que toda la información de un registro (alguna de la cual es opcional) no tiene que estar almacenada en un único trozo grande.
- Las versiones multiusuario siguientes fueron probadas por los usuarios en 1978 y 1979, tiempo por el que un lenguaje SQL había sido estandarizado.

1.4 Sistemas orientados a objetos (1980)

- Durante la década de 1980 el auge de la programación orientada a objetos influyó en el modo de manejar la información de las bases de datos. Programadores y diseñadores comenzaron a tratar los datos en las bases de datos como objetos.
- Esto quiere decir que si los datos de una persona están en la base de datos, los atributos de la persona como dirección, teléfono y edad se consideran que pertenecen a la persona, no son datos extraños. Esto permite establecer relaciones entre objetos y atributos, más que entre campos individuales
- En 1989, dos profesores de la Universidad de Wisconsin publicaron un artículo en una conferencia ACM en el que exponían sus métodos para mejorar las prestaciones de las bases de datos.
- La idea consistía en replicar la información importante y más solicitada en una base de datos temporal de pequeño tamaño con enlaces a la base de datos principal. Esto implicaba que se podía buscar mucho más rápido en la base de datos pequeña que en la grande. Su mejora de prestaciones llevó a la introducción de la indización, incorporado en la totalidad de los SGBD.

1.5 Sistemas NoSQL (2000)

- Esta tendencia introducía una línea no relacional significativamente diferentes de las clásicas.

- No requieren por lo general esquemas fijos, evitan las operaciones join almacenando datos desnormalizados y están diseñadas para escalar horizontalmente.
- La mayor parte de ellas pueden clasificarse como almacenes clave-valor o bases de datos orientadas a documentos
- Entre las aplicaciones más populares encontramos MongoDB, MemcacheDB, Redis, CouchDB, Hazelcast, Apache Cassandra y HBase, todas ellas de código abierto.

1.6 Sistemas XML (2010)

- Las Bases de Datos XML forman un subconjunto de las Bases de Datos NoSQL. Todas ellas usan el formato de almacenamiento XML, que está abierto, legible por humanos y máquinas y ampliamente usado para interoperabilidad.
- En esta categoría encontramos: BaseX, eXist, MarkLogic Server, MonetDB/XQuery, Sedna.

1.7 Conceptos importantes de Base de Datos

Base de Datos

Una colección organizada de datos interrelacionados que se almacenan y gestionan de manera estructurada para permitir el acceso, la recuperación y la manipulación eficiente de la información.

Sistemas de Gestión de Base de Datos (DBMS)

Software que permite a los usuarios crear, acceder, gestionar y manipular bases de datos. Ejemplos incluyen MySQL, PostgreSQL, Oracle, Microsoft SQL Server, etc.

Modelo de Datos

Una descripción lógica y conceptual de cómo se estructuran los datos en una base de datos. Incluye modelos como el modelo relacional, el modelo jerárquico, el

modelo de red, etc.

Tabla

En un modelo relacional, una tabla es una estructura que almacena datos en filas y columnas. Cada fila representa un registro y cada columna representa un atributo.

Campo o Columna

Una unidad de datos en una tabla que representa una característica específica de los registros almacenados.

Registro o Fila

Una entrada en una tabla que contiene valores específicos para cada columna, representando una instancia individual de datos.

Clave Primaria

Una columna o conjunto de columnas en una tabla que identifica de manera única cada registro en la tabla. Garantiza la unicidad y se utiliza para establecer relaciones entre tablas.

Clave Externa

Una columna o conjunto de columnas en una tabla que establece una relación con la clave primaria de otra tabla. Se utiliza para mantener la integridad referencial entre las tablas.

Consulta

Una solicitud para acceder o manipular datos en una base de datos. Se realiza utilizando lenguajes de consulta como SQL (Structured Query Language).

Normalización

El proceso de diseñar una base de datos de manera que los datos estén organizados de manera eficiente y libre de redundancias. Se divide en varias formas normales para evitar problemas de actualización anómala y redundancia.

Índice

Una estructura de datos utilizada para acelerar la búsqueda y recuperación de datos en una tabla. Mejora la velocidad de las consultas, pero también puede aumentar el espacio de almacenamiento.

Transacción

Una unidad de trabajo en una base de datos que consta de una o más operaciones. Deben cumplir con las propiedades ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad) para garantizar la integridad de los datos.

Integridad de Datos

La garantía de que los datos en una base de datos sean precisos, consistentes y se ajusten a las reglas y restricciones definidas.

Vista

Una representación virtual de datos seleccionados de una o varias tablas. Permite a los usuarios acceder a una parte específica de los datos sin acceder a la tabla subyacente directamente.

Respaldo y Recuperación

Procesos y técnicas para crear copias de seguridad de los datos y restaurar la base de datos en caso de fallos, errores o pérdida de datos.

1.8 Reglas para asegurar la integridad y confiabilidad de los datos

Regla de Integridad de Entidad

Cada fila en una tabla debe ser única y estar identificada por una clave primaria. Esto evita duplicados y asegura la unicidad de los registros.

Regla de Integridad Referencial

Las relaciones entre tablas deben estar respaldadas por claves primarias y claves externas. Cualquier valor en una columna de clave externa debe coincidir con un valor existente en la columna de clave primaria relacionada.

Regla de Integridad de Dominio

Los valores en una columna deben cumplir con el dominio de datos definido para esa columna, lo que significa que deben ser del tipo y rango correctos.

Regla de Integridad de Valor NULL

Los valores NULL (ausencia de datos) deben manejarse de manera adecuada y consistente en la base de datos, siguiendo reglas específicas para cada columna.

Regla de Integridad de Llave Primaria

Las claves primarias no deben contener valores NULL y deben ser únicas para cada fila. Esto garantiza la identificación única de los registros.

Regla de Integridad de Dependencia Funcional

Si una columna A determina funcionalmente una columna B, entonces cualquier valor en la columna B debe depender de manera única del valor en la columna A.

Regla de Actualización Anómala

Las actualizaciones en los datos deben realizarse de manera coherente para evitar situaciones en las que los datos se actualicen de forma incompleta o inconsistente.

Regla de Eliminación Anómala

La eliminación de datos no debe afectar inadvertidamente otros datos que dependen de ellos. Las claves externas y las relaciones deben gestionarse para evitar eliminaciones anómalas.

Regla de Consistencia de Datos

Los datos deben mantenerse en un estado coherente y lógico en todas las operaciones, evitando discrepancias o contradicciones.

Regla de Acceso Controlado

Los sistemas de bases de datos deben proporcionar controles de acceso adecuados para garantizar que solo los usuarios autorizados puedan acceder, modificar o eliminar datos según sus permisos.

SGBD (Sistemas de gestión de base de datos)

Es un software que permite crear, gestionar y manipular bases de datos de manera eficiente. Facilita la organización, almacenamiento, recuperación y administración de grandes cantidades de información de forma estructurada. Además, proporciona herramientas para garantizar la seguridad, integridad y consistencia de los datos, permitiendo que múltiples usuarios puedan acceder y modificar la información simultáneamente. Ejemplos de SGBD incluyen MySQL, PostgreSQL, y Oracle.

2. Tipos de comandos

Existen los comandos DDL, DQL, DML, DCL, TCL, te los voy a explicar de la manera más simple.

DDL (Data Definition Language)

Instrucciones que se utilizan para definir, modificar y eliminar la estructura de la base de datos, como crear tablas, modificar su estructura o eliminarlas. Los comandos son: CREATE, ALTER, TRUNCATE y DROP.

DQL (Data Query Language)

Nos permiten consultar dentro de nuestra estructura de BD. Un ejemplo es SELECT

DML (Data Manipulation Language)

Instrucciones que se utilizan para manipular los datos dentro de la base de datos, como insertar, actualizar, eliminar o recuperar datos de las tablas. Los comandos son: INSERT, SELECT, UPDATE y DELETE.

DCL

Son los que nos permite la gestión de permisos de acceso a nuestra BD. Podemos encontrar “Grant (Privilegios)” o “Revoke” (Quitar acceso).

TCL

Nos permita la transacción en nuestra BD. Por ejemplo commit nos permite efectuarla, y Rollback revertir

3. SQL Helper

¿Qué hace SQL?

Organiza, busca y analiza datos en bases de datos relacionales eficientemente.

1. Consulta

SQL nos beneficia al permitirnos realizar consultas precisas y eficientes en grandes conjuntos de datos, lo que facilita la extracción de información relevante para la toma de decisiones estratégicas.

2. Filtrado

Podemos aplicar filtros avanzados para seleccionar y analizar datos específicos según criterios definidos, lo que nos ayuda a identificar tendencias, patrones y anomalías de manera rápida y precisa.

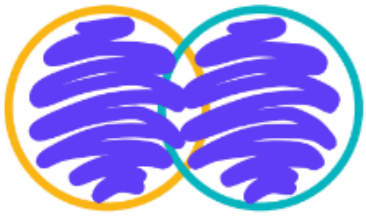
3. Análisis

SQL nos proporciona herramientas poderosas para realizar análisis complejos de datos, como cálculos estadísticos, agregaciones y comparaciones, lo que nos permite obtener perspectivas detalladas y significativas para respaldar la toma de decisiones informadas.

Join SQL

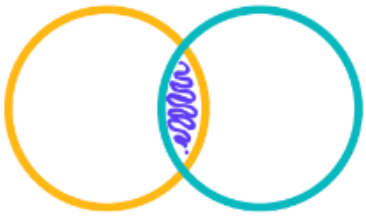
Los joins en SQL son utilizados para combinar datos de dos o más tablas con base en una condición especificada, permitiendo así relacionar la información entre ellas para consultas más completas y precisas.

FULL JOIN



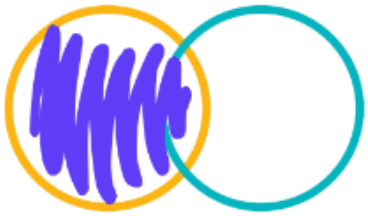
Nombre	Tipo
Pikachu	Eléctrico
Bulbasaur	Planta
Charmander	**null**
Squirtle	**null**
null	Luz

INNER JOIN



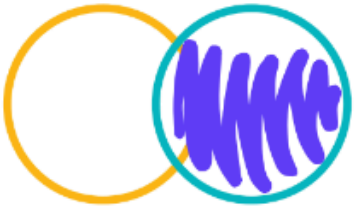
Nombre	Tipo
Pikachu	Eléctrico
Bulbasaur	Planta

LEFT JOIN



Nombre	Tipo
Pikachu	Eléctrico
Bulbasaur	Planta
Charmander	NULL
Squirtle	NULL

RIGHT JOIN



Nombre	Tipo
Pikachu	Eléctrico
Bulbasaur	Planta
NULL	Luz

Filtros SQL

```
SELECT Nombre, Tipo, Poder
FROM Pokemon
WHERE Nombre = 'Squirtle';
```

Nombre	Tipo	Poder
Squirtle	Agua	48

Filtra datos según condición.

```
SELECT  Nombre,  Poder
FROM Pokemon
WHERE Nombre
LIKE 'P%';
```

Nombre	Poder
Pikachu	55

Filtra datos con patrón **REGEX**.

```
SELECT Nombre, Tipo, Poder
FROM Pokemon
WHERE Poder
BETWEEN 50 AND 54
```

Nombre	Tipo	Poder
Charmander	Fuego	**52**

Filtra datos en rango.

```
SELECT Nombre, Tipo, Poder
FROM Pokemon
WHERE Tipo
IN  ( 'Planta', 'Luz');
```

Nombre	Tipo	Poder
Bulbasaur	**Planta**	49

Filtra datos en lista.

```
SELECT Nombre, Tipo, Poder
FROM Pokemon
WHERE Nombre IS NULL;
```

Nombre	Tipo	Poder

Filtra datos nulos.

```
SELECT  Nombre,  Poder
FROM Pokemon
WHERE Nombre = Pikachu
AND  Tipo = 'Eléctrico' ;
```

Nombre	Poder
Pikachu	55

Filtra datos con múltiples condiciones.

```
SELECT Nombre, Tipo, Poder
FROM Pokemon
WHERE Poder = 50
OR Poder = 52 ;
```

Nombre	Tipo	Poder
Charmander	Fuego	**52**

Filtra datos con alguna condición.

```
SELECT Nombre, Tipo, Poder
FROM Pokemon
WHERE NOT Poder > 49 ;
```

Nombre	Tipo	Poder
Squirtle	Agua	48

Invierte una condición.

```
SELECT Nombre, Tipo, Poder
FROM Pokemon
ORDER BY Poder ;
```

Ordena resultados. Para ordenar ASCENDENTE: "ORDER BY columna **ASC**" Para ordenar DESCENDENTE: "ORDER BY columna **DESC**"

Ejemplo de uso:

```
SELECT Nombre, Tipo, Poder
FROM Pokemon
GROUP BY Gen ;
```

Agrupa resultados. Si hay varios Pokémon con la misma generación, la consulta devolverá solo uno de esos Pokémon, generalmente el primero encontrado.

```
SELECT Tipo, AVG(Poder) AS
PoderPromedio
FROM Pokemon
GROUP BY Tipo
HAVING AVG(Poder) > 50;
```

Filtra después de agrupar. sólo podemos **utilizar** WHERE en los datos "brutos" y no **en los valores agregados.**

```
SELECT DISTINCT Nombre,
Tipo, Poder
FROM Pokemon;
```

Filtra duplicados.

```
SELECT Nombre, Tipo, Poder
FROM Pokemon
LIMIT 1;
```

Nombre	Tipo	Poder
Pikachu	Eléctrico	55

Limita cantidad de resultados.

```
SELECT Nombre, Tipo, Poder
FROM Pokemon
OFFSET 3;
```

Nombre	Tipo	Poder
Squirtle	Agua	48

Desplaza resultados.

Funciones SQL

Agregación

COUNT, SUM, AVG, MIN, MAX

Numérico

ABS, CEIL, FLOOR, ROUND, RAND

Cadena

CONCAT, SUBSTRING,UPPERT, LOWER, LENGTH, TRIM, REPLACE, REVERSE, SPLIT

Fecha

NOW, DATE, YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, DATE_ADD, DATE_FORMAT, DATE_SUB, DATEDIFF

4. JOINS, HAVING, UNIONS AND GROUPBY

JOINS

Los joins en bases de datos son operaciones fundamentales que permiten combinar filas de dos o más tablas basándose en una relación común entre ellas. Son esenciales para consultar datos distribuidos en varias tablas. Existen varios tipos de joins, cada uno con un propósito específico y que genera un conjunto de resultados diferentes.

Inner Join

- Descripción: Combina filas de dos tablas cuando existe una coincidencia en las columnas especificadas en ambas tablas.
- Uso: Se utiliza cuando se necesitan solo las filas que tienen datos relacionados en ambas tablas.

Ejemplo:

```
SELECT * FROM empleados INNER JOIN departamentos ON
empleados.departamento_id = departamentos.id;
```

- Aquí se obtendrán solo los empleados que tienen un departamento asociado.

Left Join

- Descripción: Devuelve todas las filas de la tabla de la izquierda, junto con las filas coincidentes de la tabla de la derecha. Si no hay coincidencia, las filas de la tabla de la derecha serán NULL.
- Uso: Se utiliza cuando se desean obtener todas las filas de la tabla principal (izquierda) y los datos relacionados de la tabla secundaria (derecha), si existen.

Ejemplo:

```
SELECT * FROM empleados LEFT JOIN departamentos ON
empleados.departamento_id = departamentos.id;
```

- Aquí se obtendrán todos los empleados, incluso aquellos que no tienen un departamento asignado.

Right Join

- Descripción: Es similar al LEFT JOIN, pero devuelve todas las filas de la tabla de la derecha y las filas coincidentes de la tabla de la izquierda. Si no hay coincidencia, las filas de la tabla de la izquierda serán NULL.

- Uso: Se utiliza cuando se desean obtener todas las filas de la tabla secundaria (derecha) y los datos relacionados de la tabla principal (izquierda), si existen.
- Ejemplo:

```
SELECT * FROM empleados RIGHT JOIN departamentos ON
empleados.departamento_id = departamentos.id;
```

- Aquí se obtendrán todos los departamentos, incluso aquellos sin empleados asignados.

Full Join

- Descripción: Devuelve todas las filas cuando hay una coincidencia en una de las tablas. Si no hay coincidencia, se devolverán NULL en las filas donde no haya datos coincidentes en ambas tablas.
- Uso: Se utiliza cuando se quieren obtener todas las filas de ambas tablas, mostrando NULL donde no haya coincidencias.

Ejemplo:

```
SELECT * FROM empleados FULL JOIN departamentos ON
empleados.departamento_id = departamentos.id;
```

- Aquí se obtendrán todos los empleados y todos los departamentos, incluso aquellos que no tienen coincidencias.

Cross Join

- Descripción: Devuelve el producto cartesiano de las dos tablas. Es decir, cada fila de la primera tabla se combina con todas las filas de la segunda tabla.
- Uso: Es útil en situaciones donde se necesita generar combinaciones entre todas las filas de ambas tablas.
- Ejemplo:

```
SELECT * FROM empleados CROSS JOIN departamentos;
```

- Aquí se obtendrán combinaciones de todos los empleados con todos los departamentos.

Self Join

- Descripción: Es una join de una tabla consigo misma. Se usa cuando se necesita comparar filas de la misma tabla.
- Uso: Se utiliza comúnmente para comparar registros dentro de una misma tabla, como en una jerarquía.

Ejemplo:

```
SELECT A.*, B.* FROM empleados A, empleados B WHERE A.supervisor_id =
B.empleado_id;
```

- Aquí se obtienen los empleados junto con sus supervisores.

Natural Join

- Descripción: Es un tipo de join donde las columnas que se utilizan para hacer la unión se determinan automáticamente basándose en las columnas con el mismo nombre y tipo en ambas tablas.
- Uso: Se utiliza cuando las tablas tienen columnas con el mismo nombre y tipo que deben coincidir.

Ejemplo:

```
SELECT * FROM empleados NATURAL JOIN departamentos;
```

- Aquí se hace la unión automáticamente sobre las columnas que tienen el mismo nombre.

HAVING

El comando HAVING en bases de datos se utiliza para filtrar los resultados de una consulta después de aplicar una función de agregación como SUM(), COUNT(), AVG(), MAX(), MIN(), etc. Es similar a la cláusula WHERE, pero mientras WHERE se usa para filtrar filas antes de la agregación, HAVING filtra grupos de resultados después de la agregación.

Conceptos Clave

1. Función de agregación

Una función que realiza un cálculo sobre un conjunto de valores y devuelve un solo valor, como SUM(), COUNT(), AVG(), MAX(), MIN(), etc.

2. Cláusula HAVING

Se utiliza para establecer condiciones en los grupos de datos resultantes después de aplicar funciones de agregación. Esto permite filtrar los grupos que cumplen con ciertas condiciones.

3. Sintaxis Básica

```
SELECT columna1, función_agregada(columna2) FROM tabla GROUP BY
columna1 HAVING condición_de_agregación;
```

Diferencia entre WHERE y HAVING

- WHERE se usa para filtrar filas antes de que se aplique cualquier función de agregación. No puede usarse con funciones agregadas directamente.
- HAVING se usa después de que las funciones de agregación hayan procesado los datos y se usa para filtrar esos resultados agregados.

UNION

El concepto de UNION en bases de datos se refiere a la operación que permite combinar los resultados de dos o más consultas SQL en un único conjunto de resultados. Esto es útil cuando se desea obtener datos de varias tablas o consultas que tienen la misma estructura (mismo número y tipo de columnas), y se quieren presentar esos resultados juntos.

Tipos de UNION en SQL

UNION

- Descripción: Combina los resultados de dos o más consultas en un solo conjunto de resultados, eliminando automáticamente las filas duplicadas.
- Uso: Se utiliza cuando se necesita un conjunto de resultados donde cada fila sea única.

Ejemplo:

```
SELECT nombre FROM empleados UNION SELECT nombre FROM clientes;
```

- Aquí se obtienen todos los nombres de empleados y clientes, sin duplicados.

UNION ALL

- Descripción: Similar al UNION, pero incluye todas las filas, incluso las duplicadas.
- Uso: Se utiliza cuando se necesitan ver todas las filas, incluidas las duplicadas.

Ejemplo:

```
SELECT nombre FROM empleados UNION ALL SELECT nombre FROM clientes;
```

- Aquí se obtienen todos los nombres de empleados y clientes, incluyendo posibles duplicados.

Requisitos y Reglas de UNION

Mismo número de columnas

Las consultas que se combinan deben devolver el mismo número de columnas.

Tipos de datos compatibles

Las columnas correspondientes en cada consulta deben tener tipos de datos compatibles (aunque no tienen que ser exactamente iguales, deben ser convertibles entre sí).

Orden de las columnas

Las columnas de las consultas deben estar en el mismo orden.

Comparación con JOIN

- UNION combina conjuntos de resultados de consultas distintas y los une verticalmente (una consulta después de otra).
- JOIN combina tablas basándose en una condición específica, uniendo filas relacionadas horizontalmente.

GROUP BY

El comando GROUP BY en SQL se utiliza para agrupar filas que tienen valores idénticos en una o más columnas. Luego, se pueden aplicar funciones de agregación (como SUM(), COUNT(), AVG(), MAX(), MIN()) a estas agrupaciones para calcular valores agregados para cada grupo.

Conceptos Clave

1. Agrupación de Datos

El comando GROUP BY agrupa los resultados de una consulta en conjuntos de filas que comparten el mismo valor en una o más columnas.

2. Funciones de Agregación

Estas funciones operan en los grupos creados por GROUP BY y devuelven un solo valor por grupo.

Ejemplos de funciones de agregación son:

1. SUM(): Suma de valores.
2. COUNT(): Cuenta el número de filas.
3. AVG(): Calcula el promedio.
4. MAX(): Encuentra el valor máximo.
5. MIN(): Encuentra el valor mínimo.

Sintaxis Básica

```
SELECT columna1, función_agregada(columna2) FROM tabla GROUP BY
columna1;
```

Uso Combinados con Otras Cláusulas

GROUP BY con WHERE: Puedes usar WHERE para filtrar filas antes de agruparlas.

```
SELECT id_producto, SUM(cantidad) AS total_vendido FROM ventas WHERE
precio_unitario > 50 GROUP BY id_producto;
```

Este ejemplo suma las cantidades vendidas solo para productos con un precio unitario mayor a 50.

GROUP BY con HAVING

- HAVING se usa para filtrar los resultados después de que se han agrupado y agregado.

```
SELECT id_producto, SUM(cantidad) AS total_vendido FROM ventas GROUP BY id_producto HAVING SUM(cantidad) > 100;
```

Este ejemplo devuelve solo los productos que han vendido más de 100 unidades.

GROUP BY con ORDER BY

- Puedes usar ORDER BY para ordenar los resultados de los grupos.

```
SELECT id_producto, SUM(cantidad) AS total_vendido FROM ventas GROUP BY id_producto ORDER BY total_vendido DESC;
```

Aquí se ordenan los productos por la cantidad total vendida en orden descendente.

Reglas Importantes

- Todas las columnas en la cláusula SELECT que no sean parte de una función de agregación deben aparecer en la cláusula GROUP BY.
- Si se omite una columna en el GROUP BY que está en la cláusula SELECT, SQL generará un error o, en algunos casos, se comportará de manera impredecible.

Casos de Uso de GROUP BY

1. Resúmenes y Reportes

Es común en reportes de ventas, análisis de datos, estadísticas, etc., donde necesitas ver datos agregados como totales por categoría, promedios por región, máximos y

mínimos por departamento, etc.

2. Análisis de Datos

Ayuda a analizar datos agrupándolos por criterios como fechas, categorías de productos, departamentos, etc.

3. Contar Elementos Distintos

Por ejemplo, contar cuántos clientes únicos hicieron compras en un mes:

```
SELECT COUNT(DISTINCT id_cliente) AS clientes_unicos FROM ventas
GROUP BY MONTH(fecha);
```

5. Reglas de negocio en SQL

Las reglas de negocio en SQL se refieren a las restricciones y condiciones que se aplican a los datos almacenados en una base de datos para garantizar la integridad y consistencia de los mismos.

Estas reglas se utilizan para garantizar que los datos cumplan con ciertas restricciones y requisitos específicos que son importantes para una aplicación o un negocio en particular.

Restricciones de clave primaria

(Primary Key Constraints)

- Una de las reglas más básicas es garantizar que cada fila en una tabla tenga un valor único en la columna que se define como clave primaria. Esto evita duplicados y asegura que cada fila se pueda identificar de manera única.

```
CREATE TABLE Empleados (
    empleado_id INT PRIMARY KEY,
```

```
    nombre VARCHAR(50),
    salario DECIMAL(10, 2)
);
```

Restricciones de clave foránea

(Foreign Key Constraints)

- Estas reglas aseguran que los valores en una columna coincidan con los valores de otra columna en otra tabla. Esto se utiliza para establecer relaciones entre tablas y mantener la integridad referencial.

```
CREATE TABLE Pedidos (
    pedido_id INT PRIMARY KEY,
    cliente_id INT,
    fecha_pedido DATE,
    FOREIGN KEY (cliente_id) REFERENCES Clientes(cliente_id)
);
```

Restricciones de unicidad

(Unique Constraints)

- Garantizan que los valores en una columna (o combinación de columnas) sean únicos en toda la tabla, pero no necesariamente una clave primaria.

```
CREATE TABLE Productos (
    producto_id INT PRIMARY KEY,
    nombre VARCHAR(50) UNIQUE,
    precio DECIMAL(10, 2)
);
```

Restricciones de verificación

(Check Constraints)

- Estas reglas permiten especificar condiciones que deben cumplirse para que los datos se inserten o actualicen en una tabla. Pueden utilizarse para imponer restricciones específicas en los valores de las columnas.

```
CREATE TABLE Clientes (
    cliente_id INT PRIMARY KEY,
    nombre VARCHAR(50),
    edad INT,
    estado_civil VARCHAR(10) CHECK (estado_civil IN ('Soltero',
    'Casado', 'Divorciado', 'Viudo'))
);
```

Implementación de vistas en SQL

En SQL, una vista es una representación virtual de una o más tablas en una base de datos.

Una vista no es una tabla física en sí misma, sino que es una consulta predefinida que se almacena en la base de datos y se puede utilizar como si fuera una tabla real.

Razones para usar las vistas

Simplificar consultas complejas

Las vistas permiten definir consultas complejas y luego utilizar esas consultas como si fueran tablas simples. Esto facilita la escritura y lectura de consultas, especialmente cuando se necesita combinar datos de varias tablas o realizar cálculos complicados.

Ocultar detalles de implementación

Las vistas pueden ocultar la estructura subyacente de la base de datos a los usuarios o aplicaciones. Esto significa que los usuarios pueden interactuar con los datos a través de la vista sin necesidad de conocer la complejidad de las tablas subyacentes.

Seguridad

Las vistas se pueden utilizar para controlar el acceso a los datos. Puedes otorgar permisos de acceso a una vista sin permitir acceso directo a las tablas subyacentes. Esto es útil para proteger datos sensibles.

Reutilización de consultas

Si tienes consultas que se utilizan con frecuencia en diferentes partes de una aplicación, puedes crear una vista para esa consulta y, luego, reutilizar la vista en lugar de volver a escribir la consulta en múltiples lugares

Create View

- Para crear una vista en SQL, puedes usar la sentencia CREATE VIEW. Aquí tienes un ejemplo sencillo de cómo crear una vista que muestra la lista de empleados que ganan más de $50,000 al año.

```
CREATE VIEW EmpleadosBienRemunerados AS
SELECT nombre, salario
FROM Empleados
WHERE salario > 50000;
```

- Una vez que se crea la vista, puedes consultarla como si fuera una tabla real:

```
SELECT * FROM EmpleadosBienRemunerados;
```

- Ten en cuenta que las vistas son virtuales y no almacenan datos por sí mismas. Cuando consultas una vista, en realidad estás ejecutando la consulta definida en la vista en ese momento.

6. Implementación de Funciones

¿Qué son las funciones?

- Las funciones son bloques de código que realizan una tarea específica y devuelven un valor.
- Estas funciones son útiles para realizar operaciones en los datos almacenados en una base de datos, transformar datos o realizar cálculos.

Sintaxis básica de una función

```
NOMBRE_DE_LA_FUNCION(argumento1, argumento2, ...)

NOMBRE_DE_LA_FUNCION: El nombre de la función que deseas utilizar.

argumento1, argumento2, ...: Los argumentos que la función necesita
para realizar su tarea.
```

Funciones de agregación

COUNT( )

Cuenta el número de filas en un conjunto de datos.

SUM( )

Calcula la suma de valores en una columna.

AVG( )

Calcula el promedio de valores en una columna.

MIN( )

Encuentra el valor mínimo en una columna.

MAX( )

Encuentra el valor máximo en una columna.

Funciones de cadena

CONCAT( )

Combina dos o más cadenas.

LENGTH( )

Devuelve la longitud de una cadena.

SUBSTRING( )

Obtiene una subcadena de una cadena.

UPPER( )

Convierte una cadena en mayúsculas.

LOWER( )

Convierte una cadena en minúsculas.

Funciones de Fecha y Hora

CURRENT_DATE()

Devuelve la fecha actual.

CURRENT_TIME()

Devuelve la hora actual.

CURRENT_TIMESTAMP()

Devuelve la fecha y hora actual.

DATE_FORMAT()

Formatea una fecha en un formato específico.

DATE_ADD()

Suma una cantidad de tiempo a una fecha.

DATE_SUB()

Resta una cantidad de tiempo a una fecha.

Funciones Matemáticas

ABS( )

Devuelve el valor absoluto de un número.

ROUND( )

Redondea un número.

CEIL( )

Redondea hacia arriba un número decimal.

FLOOR( )

Redondea hacia abajo un número decimal.

POWER( )

Calcula una potencia.

SQRT( )

Calcula la raíz cuadrada.

Funciones condicionales

CASE WHEN THEN ELSE END

Realiza una evaluación condicional en una consulta SQL.

Funciones de conversión de tipos

CAST( )

Convierte un valor de un tipo de dato a otro.

Funciones de agregación avanzadas

GROUP_CONCAT( )

Concatena valores en un grupo en una sola cadena.

HAVING( )

Filtra grupos en función de una condición de agregación.

Funciones de ventana (Window Functions)

Estas funciones permiten realizar cálculos a lo largo de un conjunto de filas relacionadas en una consulta, como ROW_NUMBER(), RANK(), DENSE_RANK(), LEAD(), LAG(), entre otras.

Funciones de manipulación de conjuntos

UNION( )

Combina el resultado de dos o más consultas.

INTERSECT( )

Devuelve la intersección de dos o más conjuntos de resultados.

EXCEPT( )

Devuelve las filas que están en el primer conjunto pero no en el segundo.

Funciones de agregación JSON (En sistemas que admiten JSON)

JSON_ARRAYAGG( )

Agrega valores en un arreglo JSON.

JSON_OBJECTAGG( ):

Agrega pares clave-valor en un objeto JSON.

Procedimientos almacenados

- Son una colección de instrucciones SQL que se almacenan en el servidor de la base de datos y se pueden invocar o ejecutar de manera repetida desde una aplicación o desde el propio servidor de la base de datos.
- Estos procedimientos son similares a las funciones en lenguajes de programación, ya que permiten encapsular un conjunto de acciones en una sola unidad lógica y reutilizable.

Creación de un procedimiento almacenado que selecciona todos los registros de una tabla llamada usuarios

```
DELIMITER //

CREATE PROCEDURE ObtenerUsuarios()

BEGIN

    SELECT * FROM usuarios;

END //

DELIMITER ;
```

- En este ejemplo, DELIMITER // se utiliza para cambiar el delimitador de sentencias SQL temporalmente para permitir definir el procedimiento.

```
CREATE PROCEDURE crea el procedimiento llamado ObtenerUsuarios.
```

Innovación de procedimientos almacenados

Una vez que hayas creado un procedimiento almacenado, puedes invocarlo desde una aplicación o desde el cliente MySQL utilizando la sentencia CALL. Por ejemplo:

```
CALL ObtenerUsuarios();
```

Procedimientos almacenados con parametros

- Los procedimientos almacenados pueden aceptar parámetros, lo que les permite recibir valores desde la aplicación o el cliente MySQL. Puedes definir parámetros dentro del procedimiento y usarlos en las instrucciones SQL.

```
DELIMITER //

CREATE PROCEDURE ObtenerUsuarioPorID(IN userID INT)

BEGIN

SELECT * FROM usuarios WHERE id = userID;

END //

DELIMITER ;
```

Invocar el procedimiento con parámetros

```
CALL ObtenerUsuarioPorID(1);
```

Esto buscará y devolverá el usuario con el ID 1.

Disparadores (Triggers)

¿Qué son los disparadores (Trigggers)

Son un tipo de objeto de la base de datos que se utilizan para automatizar ciertas acciones o tareas cuando ocurren eventos específicos en una tabla, como la inserción, actualización o eliminación de datos. Estos eventos pueden ser desencadenados por las sentencias SQL que modifican los datos en la tabla correspondiente

¿Cómo funcionan?

Evento desencadenante

Los disparadores se ejecutan en respuesta a eventos específicos en una tabla, como INSERT, UPDATE o DELETE. Estos eventos se asocian a una tabla y se definen como parte del disparador.

Acción del disparador

Cuando se produce el evento desencadenante, el disparador ejecuta una serie de acciones o instrucciones SQL predefinidas. Estas acciones pueden ser cualquier sentencia SQL válida, como INSERT, UPDATE, DELETE, SELECT o incluso llamadas a funciones almacenadas.

Momento de ejecución

Los disparadores pueden ser activados antes o después de que se produzca el evento desencadenante. Esto se especifica al crear el disparador:

- Disparadores "BEFORE"
  Se ejecutan antes de que se realice la acción que desencadena el evento.
  Pueden utilizarse para validar datos antes de su inserción o actualización.
- Disparadores "AFTER"(o "FOR EACH ROW")
  Se ejecutan después de que se completa la acción que desencadena el evento.
  Suelen utilizarse para realizar acciones de auditoría, registrar cambios o actualizar otras tablas relacionadas.

Condición (opcional)

Puedes especificar una condición en un disparador para controlar cuándo debe ejecutarse. Esto te permite crear disparadores que se activen solo cuando se cumplen ciertas condiciones específicas.

Creación y gestión

Los disparadores se crean y gestionan utilizando sentencias SQL o herramientas de administración de bases de datos.

Sintaxis de un Disparador (Trigger)

```
CREATE TRIGGER nombre_del_disparador

BEFORE INSERT ON nombre_de_la_tabla

FOR EACH ROW

BEGIN

    -- Acciones o instrucciones SQL a ejecutar antes de la inserción

    -- Puedes acceder a los valores de la nueva fila con NEW.columna

END;
```