Bases de Datos con SQL

Ernesto Martínez Gómez

Índice general

1 Creación de bases de datos

La creación de bases de datos y tablas es un proceso crítico que afecta al rendimiento, la organización y la escalabilidad de las apliaciones. En este capítulo se explican los conceptos básicos de la creación de bases de datos y tablas en MySQL.

1.1 Crear una base de datos

La base de datos es el contenedor lógico que almacena tablas, vistas, procedimientos almacenados, funciones y otros objetos.

1.1.1 Sintaxis

```
CREATE DATABASE nombre_base_datos;
[CHARACTER SET nombre_juego_caracteres]
[COLLATE nombre_colacion];
```

1.1.2 Mejores prácticas

- **Utilizar nombres descriptivos.** Evitar nombres genéricos como `db`, `test`, etc.
- **Utilizar minúsculas.** Los nombres de bases de datos y tablas son case insensitive en Windows, pero case sensitive en Linux por defecto.
- **Utilizar guiones bajos o guiones.** Evitar espacios y otros caracteres especiales.
- **Utilizar un juego de caracteres y colación adecuados.** Se recomienda utilizar `utf8mb4` y `utf8mb4_unicode_ci` respectivamente.
- **Utilizar `IF NOT EXISTS`.** Evitar errores si la base de datos ya existe.

  ```
  CREATE DATABASE IF NOT EXISTS tienda
  CHARACTER SET utf8mb4
  COLLATE utf8mb4_unicode_ci;
  ```

- **Versionamiento.** Incluye un esquema de versionamiento en la documentación o prefijos para reflejar cambios.

1.2 Crear una tabla

Las tablas son la estructura principal donde se almacenan los datos.

1.2.1 Sintaxis

```
CREATE TABLE nombre_tabla (
    nombre_columna1 tipo_dato1 [restricciones],
    nombre_columna2 tipo_dato2 [restricciones],
    ...
    [restricciones_tabla]
);
```

1.2.2 Buenas prácticas al diseñar tablas

Definir la clave primaria

Asegúrate de que cada tabla tenga una clave primaria (`PRIMARY KEY`) que identifique de forma única cada registro.

Las claves primarias suelen ser una columna auto incremental (`INT AUTO_INCREMENT`) o un identificador único (UUID `CHAR(36)`).

```
CREATE TABLE employees (
    employee_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    department_id INT,
    hire_date DATE
);
```

Elige el tipo de datos correcto

Usa los tipos de datos adecuados para cada columna. Algunos tipos de datos comunes son:

- `INT` para números enteros.
- `DECIMAL` para números decimales.
- `VARCHAR` para cadenas de texto.

- `DATE` para fechas.
- `ENUM` para listas de valores.
- `TEXT` para texto largo.
- `BLOB` para datos binarios.

Algunos tipos de datos específicos pueden ayudar a ahorrar espacio:

- `TINYINT` para números pequeños. (0 a 255, 1B)
- `SMALLINT` para números medianos. (-32,768 a 32,767, 2B)
- `VARCHAR` para cadenas de texto cortas en lugar de `TEXT`.
- `DECIMAL` para números decimales exactos en lugar de `FLOAT` o `DOUBLE`. [1]

Normaliza la base de datos

La normalización es un proceso que se utiliza para organizar una base de datos en tablas y columnas para reducir la redundancia y mejorar la integridad de los datos. Véase 3.

Se resume en dividir los datos en varias tablas para eliminar redundancia y usar claves ajenas (`FOREIGN KEY`) para vincular las tablas.

Utiliza restricciones

Las restricciones (`CONSTRAINTS`) son reglas que se aplican a las columnas de una tabla. Algunas restricciones comunes son:

- `NOT NULL` para evitar valores nulos.
- `UNIQUE` para valores únicos.
- `DEFAULT` para valores por defecto.
- `CHECK` para validar valores.
- `FOREIGN KEY` para referencias a otras tablas.

```
1  CREATE TABLE departments (
2      department_id INT PRIMARY KEY,
3      name VARCHAR(100) NOT NULL UNIQUE,
4      budget DECIMAL(10, 2) DEFAULT 0.00
```

```
);
```

Usa índices para mejorar el rendimiento

Define índices en columnas que se usen frecuentemente en consultas o en cláusulas `JOIN`.

```
CREATE INDEX idx_department_name ON departments(name);
```

Nombres claros y consistentes

Usa nombres descriptivos y en minúsculas, separados por guiones bajos (*snake case*). Por ejemplo: `employee_details`, `order_items`.

1.3 Relaciones entre tablas

Las relaciones entre tablas se establecen mediante claves ajenas (`FOREIGN KEY`). Estas aseguran la integridad referencial y permiten realizar consultas que unen datos de varias tablas.

```
CREATE TABLE orders (
    order_id INT AUTO INCREMENT PRIMARY KEY,
    customer_id INT NOT NULL,
    order_date DATE,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
    ON DELETE CASCADE
    ON UPDATE CASCADE
);
```

1.3.1 Tipos de relaciones

- **Uno a uno (1:1).** Cada fila de una tabla se relaciona con una fila de otra tabla.
- **Uno a muchos (1:N).** Cada fila de una tabla se relaciona con varias filas de otra tabla.
- **Muchos a muchos (N:M).** Varias filas de una tabla se relacionan con varias filas de otra tabla.

1.3.2 Buenas prácticas

- Usa `ON DELETE CASCADE` y `ON DELETE SET NULL` según sea necesario.
- Define índices automáticamente en columnas con claves ajenas para acelerar consultas `JOIN`.

1.4 Indexación

Los índices son estructuras de datos que mejoran la velocidad de las consultas al permitir a la base de datos buscar rápidamente los registros.

1.4.1 Tipos de índices

- **Índice primario.** Se crea automáticamente al definir una clave primaria.
- **Índice único.** Garantiza que no haya valroes duplicados en una columna. En MySQL, se crea automáticamente al definir una restricción `UNIQUE`.

```
1  CREATE UNIQUE INDEX idx_email on users(email);
```

- **Índice múltiple.** Se crea en varias columnas para acelerar consultas que usan esas columnas.

```
1  CREATE INDEX idx_id_date on orders(order_id, order_date);
```

1.4.2 ¿Cuándo crear índices?

- En columnas que aparecen frecuentemente en cláusulas `WHERE` o `JOIN`.
- En columnas usadas para ordenar (`ORDER BY`) o agrupar (`GROUP BY`).

Nota: No abuses de los índices, ya que pueden ralentizar las operaciones de escritura y ocupar espacio en disco.

1.5 Partición de tablas

La partición de tablas es una técnica que divide una tabla en fragmentos más pequeños para mejorar el rendimiento y la administración de la base de datos.

Véase 4.

1.6 Ejemplo completo

Ejemplo de creación de base de datos

Crea una base de datos para almacenar empleados y departamentos.

```sql
CREATE DATABASE IF NOT EXISTS company
CHARACTER SET utf8mb4
COLLATE utf8mb4_unicode_ci;

USE company;

CREATE TABLE employees (
    employee_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    department_id INT,
    hire_date DATE,
    salary DECIMAL(10, 2) DEFAULT 0.00,
    FOREIGN KEY (department_id) REFERENCES departments(department_id
    )
    ON DELETE SET NULL
    ON UPDATE CASCADE
)

CREATE TABLE departments (
    department_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL UNIQUE,
    budget DECIMAL(10, 2) DEFAULT 0.00
);

CREATE INDEX idx_department_name ON departments(name);
```

2 CTEs: Common Table Expressions

2.1 ¿Qué es un CTE?

Un Common Table Expression (CTE) es una forma de crear una consulta temporal nombrada que puedes usar en la misma consulta SQL. Los CTEs hacen que las consultas sean más legibles y modulares, especialmente en escenarios complejos.

2.2 Ventajas de los CTEs

- **Legibilidad:** Los CTEs hacen que las consultas sean más fáciles de leer y mantener.
- **Reutilización:** Puedes reutilizar un CTE en la misma consulta.
- **Modularidad:** Puedes dividir una consulta compleja en partes más pequeñas y manejables.
- **Alternativa a las subconsultas:** Ayudan a evitar consultas anidadas difíciles de leer.

2.3 Sintaxis de un CTE

```sql
WITH cte_name AS (
    SELECT column1, column2
    FROM table_name
    WHERE condition
)
SELECT column1
FROM cte_name
WHERE another_condition;
```

Ejemplo de CTE

Crea una consulta que muestre los empleados cuyo salario es mayor que el salario promedio de su departamento.

```sql
WITH DepartmentAverage AS (
    SELECT department_id, AVG(salary) AS avg_salary
    FROM employees
    GROUP BY department_id
)
SELECT e.emplyee_id, e.name, e.salary, e.department_id
FROM employees e
JOIN DepartmentAverage d
ON e.department_id = d.department_id
WHERE e.salary > d.avg_salary;
```

En este ejemplo, la consulta `DepartmentAverage` calcula el salario promedio por departamento. Luego, la consulta principal se une con el CTE `DepartmentAverage` para filtrar los empleados.

De esta forma, los CTEs hacen que la consulta sea más legible y modular.

2.4 Cuándo usar CTEs

- **Dividir consultas complejas:** Cuando una consulta tiene múltiples niveles de anidación o subconsultas difíciles de leer
- **Evitar redundancia:** Si necesitas reutilizar una subconsulta en la misma consulta.
- **Mejorar la legibilidad:** Para hacer que la consulta sea más fácil de entender y mantener.
- **Recursión:** Cuando trabajas con datos jerárquicos o necesitas realizar cálculos iterativos (recursividad).

2.5 CTEs Recursivos

Un CTE recursivo es un CTE que se referencia a sí mismo en la definición. Es útil para trabajar con datos jerárquicos, como estructuras de árbol (ej. una tabla de empleados y sus jefes)

2.5.1 Sintaxis de un CTE Recursivo

```sql
WITH RECURSIVE cte_name AS (
    -- Anchor Member: Define la base de la recursión
    SELECT column1, column2
```

```
    FROM table_name
    WHERE condition

    UNION ALL

    -- Recursive Member: Define cómo la recursión avanza
    SELECT column1, column2
    FROM table_name
    JOIN cte_name
    ON table_name.column = cte_name.column
)
SELECT *
FROM cte_name;
```

Ejemplo de CTE Recursivo

Crea una consulta que muestre la jerarquía de empleados en una tabla de empleados.

```
WITH RECURSIVE EmployeeHierarchy AS (
    -- Anchor Member: el CEO no tiene jefe
    SELECT employee_id, name, manager_id, 1 AS level
    FROM employees e
    WHERE manager_id IS NULL

    UNION ALL

    -- Recursive Member: Empleados con jefe
    SELECT e.employee_id, e.name, e.manager_id, eh.level + 1
    FROM employees e
    JOIN EmployeeHierarchy eh
    ON e.manager_id = eh.employee_id
)
SELECT *
FROM EmployeeHierarchy
ORDER BY level;
```

2.6 Buenas Prácticas con CTEs

- **Usa nombres descriptivos:** Elige nombres significativos para los CTEs.
- **Divide consultas complejas:** Usa un CTE por cada «bloque lógico» de la consulta.

- **Evita CTEs innecesarios:** Si una consulta se puede escribir sin CTEs, no los uses por estética.
- **Cuidado con el rendimiento:** Un CTE no se almacena físicamente y se evalúa cada vez que se referencia. Para reutilización intensiva, considera una tabla temporal.
- **Limita la recursividad:** Asegúrate de establecer una condición de terminación en los CTEs recursivos para evitar bucles infinitos.

3 Normalización

4 Particiones de tablas

Bibliografía

[1] L. Daily. «MySQL: Decimal vs Double vs Float?» Consultado el 19 de noviembre de 2024. (sep. de 2023), dirección: https://laraveldaily.com/post/mysql-decimal-vs-double-vs-float.