

Bases de Datos con SQL



Ernesto Martínez Gómez

Índice general

1. Creación de bases de datos	1
1.1. Crear una base de datos	1
1.1.1. Sintaxis	1
1.1.2. Mejores prácticas	1
1.2. Crear una tabla	2
1.2.1. Sintaxis	2
1.2.2. Buenas prácticas al diseñar tablas	2
1.3. Relaciones entre tablas	4
1.3.1. Tipos de relaciones	4
1.3.2. Buenas prácticas	4
1.4. Indexación	5
1.4.1. Tipos de índices	5
1.4.2. ¿Cuándo crear índices?	5
1.5. Partición de tablas	5
1.6. Ejemplo completo	6
2. Normalización	7
3. Particiones de tablas	8

4. CTEs: Common Table Expressions	9
4.1. ¿Qué es un CTE?	9
4.2. Ventajas de los CTEs	9
4.3. Sintaxis de un CTE	9
4.4. Cuándo usar CTEs	10
4.5. CTEs Recursivos	10
4.5.1. Sintaxis de un CTE Recursivo	10
4.6. Buenas Prácticas con CTEs	11
5. Window Functions	13
5.1. Qué son las window functions	13
5.1.1. Comparación con funciones de agregación	13
5.2. ¿Cuándo se usan las window functions?	14
5.3. Sintaxis de las window functions	14
5.4. Ejemplos de window functions	15
5.4.1. Ranking y análisis relativo	15
5.4.2. Cálculos acumulativos	15
5.4.3. Comparaciones entre filas	15
5.4.4. Agregados no destructivos	16
5.4.5. Series temporales	16
5.5. Mejorar el rendimiento con window functions	16

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 aplicaciones. 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

```
1 CREATE DATABASE nombre_base_datos;  
2 [CHARACTER SET nombre_juego_caracteres]  
3 [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.

```
1 CREATE DATABASE IF NOT EXISTS tienda  
2 CHARACTER SET utf8mb4  
3 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

```
1 CREATE TABLE nombre_tabla (  
2     nombre_columna1 tipo_dato1 [restricciones],  
3     nombre_columna2 tipo_dato2 [restricciones],  
4     ...  
5     [restricciones_tabla]  
6 );
```

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)).

```
1 CREATE TABLE employees (  
2     employee_id INT AUTO_INCREMENT PRIMARY KEY,  
3     name VARCHAR(100) NOT NULL,  
4     department_id INT,  
5     hire_date DATE  
6 );
```

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 2.

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  
5 );
```

Usa índices para mejorar el rendimiento

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

```
1 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.

```
1 CREATE TABLE orders (  
2     order_id INT AUTO INCREMENT PRIMARY KEY,  
3     customer_id INT NOT NULL,  
4     order_date DATE,  
5     FOREIGN KEY (customer_id) REFERENCES customers(customer_id)  
6     ON DELETE CASCADE  
7     ON UPDATE CASCADE  
8 );
```

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 valores 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 3.

1.6 Ejemplo completo

Ejemplo de creación de base de datos

Crea una base de datos para almacenar empleados y departamentos.

```
1 CREATE DATABASE IF NOT EXISTS company
2 CHARACTER SET utf8mb4
3 COLLATE utf8mb4_unicode_ci;
4
5 USE company;
6
7 CREATE TABLE employees (
8     employee_id INT AUTO_INCREMENT PRIMARY KEY,
9     name VARCHAR(100) NOT NULL,
10    department_id INT,
11    hire_date DATE,
12    salary DECIMAL(10, 2) DEFAULT 0.00,
13    FOREIGN KEY (department_id) REFERENCES departments(department_id)
14    ON DELETE SET NULL
15    ON UPDATE CASCADE
16 )
17
18 CREATE TABLE departments (
19     department_id INT PRIMARY KEY,
20     name VARCHAR(100) NOT NULL UNIQUE,
21     budget DECIMAL(10, 2) DEFAULT 0.00
22 );
23
24 CREATE INDEX idx_department_name ON departments(name);
```

2 Normalización

3 Particiones de tablas

4 CTEs: Common Table Expressions

4.1 ¿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.

4.2 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.

4.3 Sintaxis de un CTE

```
1 WITH cte_name AS (  
2     SELECT column1, column2  
3     FROM table_name  
4     WHERE condition  
5 )  
6 SELECT column1  
7 FROM cte_name  
8 WHERE another_condition;
```

Ejemplo de CTE

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

```
1 WITH DepartmentAverage AS (  
2     SELECT department_id, AVG(salary) AS avg_salary  
3     FROM employees  
4     GROUP BY department_id  
5 )  
6 SELECT e.employee_id, e.name, e.salary, e.department_id  
7 FROM employees e  
8 JOIN DepartmentAverage d  
9 ON e.department_id = d.department_id  
10 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.

4.4 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).

4.5 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)

4.5.1 Sintaxis de un CTE Recursivo

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

```
4      FROM table_name
5      WHERE condition
6
7      UNION ALL
8
9      -- Recursive Member: Define cómo la recursión avanza
10     SELECT column1, column2
11     FROM table_name
12     JOIN cte_name
13     ON table_name.column = cte_name.column
14 )
15 SELECT *
16 FROM cte_name;
```

Ejemplo de CTE Recursivo

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

```
1  WITH RECURSIVE EmployeeHierarchy AS (
2      -- Anchor Member: el CEO no tiene jefe
3      SELECT employee_id, name, manager_id, 1 AS level
4      FROM employees e
5      WHERE manager_id IS NULL
6
7      UNION ALL
8
9      -- Recursive Member: Empleados con jefe
10     SELECT e.employee_id, e.name, e.manager_id, eh.level + 1
11     FROM employees e
12     JOIN EmployeeHierarchy eh
13     ON e.manager_id = eh.employee_id
14 )
15 SELECT *
16 FROM EmployeeHierarchy
17 ORDER BY level;
```

4.6 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.

5 Window Functions

Las window functions son una herramienta para realizar cálculos que dependen de un conjunto de filas relacionadas, conocidas como una ventana.

A diferencia de las funciones de agregación como SUM() o AVG(), las window functions no agrupan los resultados en una sola fila. En su lugar, devuelven un valor para cada fila.

5.1 Qué son las window functions

Una **window function** es una función que opera sobre un conjunto de filas definidas por una cláusula OVER. A este conjunto de filas se le conoce como «ventana».

5.1.1 Comparación con funciones de agregación

Ejemplo con función de agregación

Media de salarios por departamento con GROUP BY

Devuelve una fila por departamento y se pierde el detalle de cada empleado.

```
1 SELECT department_id, AVG(salary) as avg_salary
2 FROM employees
3 GROUP BY department_id;
```

Ejemplo con window function

Rango de salarios por departamento con OVER

Devuelve todas las filas, pero incluye la media de salarios por departamento junto a cada empleado.

```
1 SELECT employee_id, department_id, salary,
2        AVG(salary) OVER (PARTITION BY department_id) as avg_salary
3 FROM employees;
```

5.2 ¿Cuándo se usan las window functions?

- **Ranking y análisis relativo:**
 - Determinar el puesto de un empleado dentro de su departamento.
 - Comparar ventas de un mes con el promedio anual.
- **Cálculos acumulativos:**
 - Sumar ingresos mes a mes de un cliente.
 - Obtener totales acumulativos en un periodo de tiempo.
- **Agregados no destructivos:**
 - Mostrar el promedio o el máximo sin agrupar los resultados.
- **Comparaciones entre filas:**
 - Calcular diferencias entre el salario de un empleado y el promedio de su departamento.
- **Análisis de series temporales:**
 - Calcular la variación de ventas de un producto en el tiempo.

5.3 Sintaxis de las window functions

Una función agregada tiene la siguiente estructura:

```
1 <function>(<column>) OVER (  
2     [PARTITION BY <column>]  
3     [ORDER BY <column>]  
4     [frame_spec]  
5 )
```

1. **Función agregada:**
 - Por ejemplo: SUM, AVG, ROW_NUMBER, RANK, LEAD, LAG, etc.
2. **Cláusula OVER:**
 - Define el conjunto de filas sobre el que se aplica la función.
3. **PARTITION BY:**
 - Divide las filas en subconjuntos independientes.
 - Similar a GROUP BY, pero sin colapsar los datos.
4. **ORDER BY:**
 - Ordena las filas dentro de cada partición.
 - Necesario para cálculos como acumulativos o rankings.

5. frame_spec:

- Define el rango de filas sobre el que se aplica la función.
- Opcional y avanzado.

5.4 Ejemplos de window functions

5.4.1 Ranking y análisis relativo

Determinar el ranking de empleados por salario dentro de cada departamento.

```
1 SELECT employee_id, department_id, salary,  
2     RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) as  
   rank  
3 FROM employees;
```

Funciones de ranking:

- RANK() devuelve el puesto de cada fila con posibles «saltos» en caso de empate.
- DENSE_RANK() devuelve el puesto sin saltos.
- ROW_NUMBER() devuelve un número de fila único, por lo que no hay empates.

5.4.2 Cálculos acumulativos

Calcular el total acumulado de ventas por cliente, ordenado por fecha.

```
1 SELECT customer_id, sale_date, amount,  
2     SUM(amount) OVER (PARTITION BY customer_id ORDER BY sale_date)  
   as cumulative_total  
3 FROM sales;
```

5.4.3 Comparaciones entre filas

Calcular la diferencia entre el salario actual y el del empleado anterior en el departamento.

```
1 SELECT employee_id, department_id, salary,  
2     salary - LAG(salary) OVER (PARTITION BY department_id ORDER BY  
   salary) as salary_diff  
3 FROM employees;
```

Funciones de comparación:

- LEAD() devuelve el valor de la siguiente fila.
- LAG() devuelve el valor de la fila anterior.

5.4.4 Agregados no destructivos

Mostrar el salario promedio del departamento junto a cada empleado.

```
1 SELECT employee_id, department_id, salary,  
2     AVG(salary) OVER (PARTITION BY department_id) as avg_salary  
3 FROM employees;
```

5.4.5 Series temporales

Calcular el cambio porcentual de ventas mes a mes.

```
1 SELECT sale_date, amount,  
2     (amount - LAG(amount) OVER (ORDER BY sale_date)) / LAG(amount)  
3     OVER (ORDER BY sale_date) * 100 AS pct_change  
4 FROM sales;
```

LAG(amount) recupera el valor del mes anterior para calcular la diferencia y el cambio porcentual.

5.5 Mejorar el rendimiento con window functions

- **Usa índices:**
 - Las columnas en las cláusulas PARTITION BY y ORDER BY pueden beneficiarse de índices.
- **Evita recalcular datos repetidamente:**
 - Las funciones de ventana calculan resultados basados en un solo conjunto de datos, mejorando la eficiencia comparado con subconsultas.

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>.