Notas de SQL

Lucho Cervantes Jorge Luis

20 de septiembre de 2024

1. Comentario inicial

No hay mejores notas que la documentación original [1], de este modo, el presente trabajo no busca sustituir a dicha documentación. Por el mismo motivo, la estructura y los conceptos no son tan rigurosos. Este trabajo es más un catálogo, con una muy breve y probablemente incompleta descripción, de las herramientas que utilizo con más frecuencia. El presente trabajo tiene como base inicial el curso *Curso de SQL desde cero (completo)* [2] y posteriormente se va complementando con otros cursos que se van mencionando en las referencias.

Índice

2. Conceptos previos

2.1. SQL (structured Query Language):

lenguaje de programación estandarizado que permite el manejo de bases de datos relacionales. Está basado en el álgebra relacional. Permite crear y administrar bases de datos (crear, modificar y eliminar, etc. con diferentes bases de datos, tablas y otros objetos). Permite hacer consultas de las bases de datos para obtener una información específica. También permite modificar, actualizar, insertar, eliminar, restringir, etc. Los datos de una base de datos. Permite generar informes y hacer análisis de datos. Permite administrar usuarios y permisos, entre muchas otras cosas.

Nota: como curiosidad, SQL no es sensible a mayúsculas o minúsculas. Generalmente, se trabaja con mayúsculas por convención, exceptuando los nombres y las funciones.

2.2. Base de datos:

consta de una colección de tablas en las que se almacena un conjunto específico de datos estructurados. Una tabla contiene una colección de filas, también denominadas tuplas o registros, y columnas, también denominadas atributos [1]. Las bases de datos cuentan con diferentes objetos:

- **Entidad:** es la representación de cualquier objeto o concepto de la vida real (En este contexto a una entidad es descrita por una tabla). En la notación de Chen (nCH) se representan con su nombre encerrado en un rectángulo.
- **Atributos:** todas las entidades cuentan con ellos y son básicamente sus propiedades. Estos pueden ser
  - **Simples:** conformados por datos únicos.
  - **Compuestos:** se ramifican por atributos más pequeños. En nCH se representan como ramificaciónes del atributo principal.

- **Multivalor:** adquieren más de un valor. En nCH se representan con el nombre encerrado por dos óvalos.
- **Derivados:** se pueden obtener a partir de operar otros atributos ya existentes. En nCH se representan con un óvalo punteado.

- **Clave (Key):** Es un atributo asignado a una entidad de forma única para identificarla. En nCH se representa con el nombre, subrayado, encerrado en un óvalo.

2.3. Gestor de bases de datos:

es la interfaz gráfica que facilita el uso del código de SQL. Es como VScode pero enfocado a SQL. También se puede trabajar con SQL desde VS code[1].

Hint: **cuando se tiene un código en el cual se trabaja con una palabra recurrente (por ejemplo una variable), esta se puede sustituir, sin reescribir de una por una, mediante la ventana de "Find and Replase" (presionar ctrl + h).**

2.4. Consultas (querys):

En SQL son las líneas de código equivalentes a los comandos. Por ejemplo, la consulta:

```
CREATE DATABASE "NOMBRE_BASE_DE_DATOS"
```

Al usar el gestor DB Browser for SQLite el comando anterior es equivalente al botón "new database". La función del gestor es la de crear código a partir de las acciones que hagamos en la interfaz gráfica. Por ejemplo, después de crear una base de datos aparece una pestaña con la opción de crear una tabla con el nombre deseado. Al computar el nombre y aceptar, lo que en realidad está sucediendo es que el gestor produjo la parte de la consulta

```
CREATE TABLE "NOMBRE_TABLA" ();
```

En SQL y bases de datos en general, a las columnas de una tabla se les llama CAMPOS y a las filas REGISTROS. A cada recuadro de una tabla se le llama VALOR DE CAMPO. Los valores de campo pueden ser de diferentes tipos:

- **INTEGER:** números enteros.
- **TEXT:** cadenas de texto.
- **BLOB:** archivo que almacena imágenes, fotos, videos etc. (almacena archivos en binario).
- **REAL:** números reales (como el float en python) con una precisión limitada.
- **NUMERIC:** flotantes de alta precisión.

[1]Para la elaboración de la mayoría de estas notas se uso SQLite

Podemos configurar la tabla de manera que tenga un valor de campo por defecto. Por ejemplo, una tabla con tres campos, con valores enteros y con 0 por defecto, puede crearse en el gestor o con el siguiente código:

```
CREATE TABLE "NOMBRE_TABLA" (

    "NOMBRE_CAMPO_1" INTEGER DEFAULT 0,
    "NOMBRE_CAMPO_2" INTEGER DEFAULT 0,
    "NOMBRE_CAMPO_3" INTEGER DEFAULT 0

    );
```

Para ejecutar código SQL en el gestor, tenemos que ir a la opción "Execute SQL". Una vez creada o descargada una base de datos podremos trabajar con ella ejecutando consultas en dicha opción.

3. Primeras consultas

La consulta más básica es:

```
SELECT * FROM NOMBRE_TABLA
```

que selecciona todo (*) de la tabla NOMBRE_TABLA (a notar que en este caso el nombre no está entre comillas).

Para insertar datos en una tabla se usa:

```
INSERT INTO NOMBRE_TABLA (CAMPO_1,..., CAMPO_n)
VALUES (VAL_CAMPO_1, ..., VAL_CAMPO_n)
```

Si son los mismos campos, pero es más de un registro:

```
INSERT INTO TABLA (CAMPO_1,..., CAMPO_n)
VALUES (VAL_CAMPO_1,..., VAL_CAMPOn),
       (VAL_CAMPO_1,..., VAL_CAMPOn),
                        .
                        .
                        .
       (VAL_CAMPO_1,..., VAL_CAMPOn),
```

Es habitual que existan dos registros idénticos en todo, por lo que es conveniente diferenciarlos asignándoles una clave única (**key**) o identificador. Si se quiere usar un entero que se vaya incrementando automáticamente al agregar registros, debe agregarse la key desde el inicio.

Para borrar todos los registros de una tabla se usa:

```
DELETE * FROM NOMBRE_TABLA
```

Nota: Curiosamente no funciona si la tabla solo tiene un registro. En tal caso basta quitar * en el código anterior.

Si solo se quieren eliminar registros específicos, puede expresarse la condición que englobe a dichos registros:

```
DELETE FROM NOMBRE_TABLA
WHERE CONDICION
```

Nota muy importante: Siempre, siempre, siempre, debe colocarse la condición "WHERE", a menos que se desee borrar la base de datos por completo, como en el meme.

Para agregar la keys se agregan las líneas

```
"KEY" INTEGER,
PRIMARY KEY("KEY" AUTOINCREMENT)
```

después de la especificación de los campos, antes de cerrar con '):' en el código con el que se crea la tabla.

Nota: para hacer más de una consulta, se escribe cada una de estas se separándolas con ;

Cuando las keys tienen la propiedad AUTOINCREMENT automáticamente se les toma como "PRIMARY KEY" (o PK), ya que aseguran una identificación única de los registros de la tabla. Sin embargo, cuando estas claves son colocadas en otra tabla para referenciar los registros de la tabla en donde son PK, se les conoce como FOREIGN KEYS y en ese caso sí pueden referenciarse más de dos veces en la misma tabla.

Para modificar temporalmente (solo durante una consulta) el nombre de algún campo se usa "AS" o []:

```
SELECT NOMBRE AS NOMBRE_TEMPORAL FROM NOMBRE_TABLA
SELECT [NOMBRE] NOMBRE_TEMPORAL FROM NOMBRE_TABLA
```

Si se quieren seleccionar más de dos campos:

```
SELECT CAMPO1, CAMPO2  FROM NOMBRE_TABLA
```

Nota: la tabla desplegada muestra los campos en el orden en que están en el código.

Se puede también realizar operaciones básicas sobre los campos seleccionados, como +, -, * / etc. incluso funciones, f(CAMPO). Para esto se usa:

```
SELECT f(CAMPO) FROM NOMBRE_TABLA
```

Para que la tabla seleccionada se ordene según alguno de los campos y no por la "PRIMARY KEY" auto incrementable por defecto, se usa:

```
SELECT CAMPO_1,..., CAMPO_N FROM NOMBRE_TABLA
ORDER BY CAMPO
```

Por defecto, el ordenamiento es de manera descendente, es decir, la línea ORDER BY CAMPO es equivalente a ORDER BY CAMPO ASC. Además de ASC, se puede ordenar de manera descendente con DESC. La jerarquía de ordenamiento, de menos a más, es la siguiente:

1. Valores nulos (NULL)
2. Números
3. Carácteres especiales (coma, punto, acento, etc.)
4. Letras

En el caso de contar con registros con valores nulos no deseados puede usarse:

```
SELECT CAMPO_1,..., CAMPO_n FROM NOMBRE_TABLA
ORDER BY CAMPO ASC NULLS LAST
```

de este modo los registros con valores nulos son enviados al final de la tabla. Si se quieren ordenar de manera descendente debe usarse DESC NULLS FIRST.

Si el campo a partir del cual se establece el orden cuenta con elementos repetidos, puede indicarse un segundo campo, para ordenar dichos elementos. Esto se hace simplemente colocando "ORDER BY CAMPO1, CAMPO2". Naturalmente, esto puede generalizarse a más de dos campos.

Si desean ordenarse aleatoriamente, se utiliza la función "random()":

```
SELECT CAMPO_1,..., CAMPO_n FROM NOMBRE_TABLA
ORDER BY random()
```

Nota: en este caso, aunque se coloquen, son ignoradas las instrucciones "NULL LAST" y "NULL FIRST".

Si se quieren seleccionar solo los valores distintos de un campo se usa:

```
SELECT DISTINCT CAMPO FROM NOMBRE_TABLA
```

3.1. Condiciones

Sirven para especificar los registros sobre los que se quiere aplicar una consulta. Estas se especifican después de la consulta con WHERE + CONDICIÓN, la condición puede ser de la forma CAMPO = VALOR, donde la igualdad puede reemplazarse por !=, <, >, <= y >=.

Hasta ahora, se han visto las consultas SELECT y DELETE, con las cuales puede usarse WHERE, sin embargo, hay una consulta que sirve para modificar (o actualizar) los valores de campo sin tener que agregar un registro nuevo (con INSERT INTO). A saber, es la consulta UPDATE y se aplica junto a WHERE, como sigue:

```
UPDATE NOMBRE_TABLA
SET CAMPO_1 = VALOR_1,..., CAMPO_n = VALOR_n
WHERE CONDICION
```

Una condición puede ser la negación de otra condición, para expresar esto se utiliza NOT CONDICION. También puede ocurrir que una condición se exprese como que suceda al menos una de un conjunto de condiciones o todas ellas a la vez. Para esto se utilizan OR Y AND. Las condiciones son de la forma:

```
CONDICIÓN\_1 OR ... OR CONDICION\_n

CONDICION\_1 AND ... AND CONDICION\_n
```

Una vez establecida una condición para una consulta, puede ocurrir que solo queramos que se aplique a los primeros n registros que cumplan dicha condición. Para esto se usa LIMIT n, después de WHERE CONDICION. Si deseamos que se aplique a cualesquiera n registros, debe colocarse ORDER BY random() antes de LIMIT. En la actualización anterior esto quedaría como:

```
UPDATE NOMBRE_TABLA
SET CAMPO_1 = VALOR_1,..., CAMPO_n = VALOR_n
WHERE CONDICION
ORDER BY random()
LIMIT n
```

Para especificar un rango cerrado, pueden usarse, a manera de condición, cualquiera de las siguientes dos líneas:

```
CAMPO >= VALOR_INICIAL AND CAMPO >= VALOR_FINAL

CAMPO BETWEEN VALOR_INICIAL AND VALOR_FINAL
```

Para trabajar con el complemento de dicho rango se usa:

```
CAMPO NOT BETWEEN VALOR_INICIAL AND VALOR_FINAL
```

Nota: en este caso también es quivalente poner el NOT antes de CAMPO.

Para encontrar registros específicos mediante coincidencia de caracteres, se usa LIKE. Para esto se usa una condición de la forma:

```
CAMPO LIKE "SECUENCIA_CARACTERES"
```

Si la secuencia de caracteres (incluyendo espacios) se coloca simplemente entre comillas como en el ejemplo anterior, LIKE buscará los registro cuyo valor de campo sea exactamente "SECUENCIA_CARACTERES". En este sentido, LIKE funciona como una igualdad. Sin embargo, se puede colocar el símbolo % para indicar "cualquier secuencia de caracteres". Así, para indicar una secuencia que empiece con r se coloca **r %**, para indicar que termina con r se coloca **%r**, para indicar que contiene al menos una r se coloca **%r %**, etc.

Para indicar "cualquier caractér" se usa _. Así, para indicar que una palabra empieza con r seguida de cualesquiera 3 caracteres, se coloca **"r___"**, etc. Naturalmente % y _ se pueden combinar.

Para indicar la condición de que los valores de campo de los registros seleccionados sean nulos o no, se usa IS NULL y IS NOT NULL respectivamente.

Para indicar la condición de que un campo tome valores dentro de un cierto conjunto, se usa IN de la forma siguiente:

```
WHERE CAMPO IN (VALOR_1,...,VALOR_n)
```

Si se quiere que el campo tome valores en el complemento se usa NOT IN.

3.2. Funciones de agregación

Son aquellas que ayudan a agrupar, resumir o hacer directamente estadísticas con los datos de un campo. Estas son de la forma "funcion()". Algunas de las funciones de agregación más comunes son:

- **count():** cuenta el número de registros en un campo.
- **sum():** suma los valores de campo.
- **avg():** promedia los valores de campo.
- **round(VALOR,n):** redondea el argumento "VALOR" a un número de decimales n.
- **min(), max():** obtienen el valor mínimo y el valor máximo de un campo, respectivamente.

Entre muchas otras que pueden encontrarse en la documentación.

Otra cláusula útil es GROUP BY CAMPO, que agrupa, en un solo registro, todos los valores repetidos en un CAMPO. Si el registro cuenta con otros campos, sus valores se sumarán por defecto si estos se tienen valores numéricos y se conservará el valor del primer elemento repetido, si es que estos no pueden sumarse (por ejemplo, si se tienen nombres).

En lugar de sumarse, puede aplicarse alguna otra función de agregación a los valores numéricos de los valores repetidos, Esto se hace desde la selección como en el siguiente ejemplo:

```
SELECT CAMPO_1, funcion(CAMPO_2) AS FCAMPO_2 ...
                        ...FROM NOMBRE_TABLA
WHERE CONDICIÓN
GROUP BY CAMPO_1
ORDER BY FCAMPO_2
```

A pesar de ir primero en el código, la función de agregación no se aplica a cada uno de los valores de CAMPO_2 como se esperaría, en su lugar, se aplica al cada grupo de elementos de CAMPO_2 que tienen el mismo valor en CAMPO_1.

Esto es debido a que los registros de la tabla original primero se filtran y luego se agrupan, por tal motivo WHERE se indica antes que ORDER BY en el código[2]. Debido a esto, si se desean filtrar los registros de la tabla que se obtiene al agrupar, se tiene que hacer con una cláusula distinta a WHERE. Para esto se usa HAVING seguido de la condición. En el ejemplo anterior esto sería:

```
SELECT CAMPO_1, funcion(CAMPO_2) AS FCAMPO_2 ...
                        ...FROM NOMBRE_TABLA
WHERE CONDICIÓN
GROUP BY CAMPO_1
HAVING FCAMPO_2 >= 5
ORDER BY FCAMPO_2
```

Respecto al orden en que se ejecuta el código, el orden general es el siguiente, primero se ejecuta SELECT, seguido de WHERE, GROUP BY, se ejecutan las funciones de agregación y se sigue con HAVING, ORDER BY y finalmente LIMIT. Cabe resaltar que para ocupar HAVING es necesario haber usado antes GROUP BY y que una función de agregación no se puede aplicar a otra función de agregación (i.e. a los elementos de campo de la tabla obtenida al agrupar). Para esto último se requiere hacer una sub consulta.

4. Relaciones entre tablas (subconsultas)

Cuando realizamos una consulta, ya sea con WHERE, DELETE, INSERT INTO y UPDATE, tenemos que especificar los campos (o valores) "CAMPO_1, ... , CAMPO_n" (O * si queremos seleccionar todo) a los que queremos que se aplique la consulta. Sin embargo, dichos campos o valores pueden ser resultado de otra consulta. En tal caso, se dice que se hace una sub consulta.

Una sub consulta se hace colocando otra consulta entre paréntesis en el lugar donde se colocaría un campo, esta también puede renombrarse con "AS". Una sub consulta es de la forma:

```
SELECT (SELECT ...), AS NOMBRE_SUBCONSULTA
FROM NOMBRE_TABLA
```

Las sub consultas son útiles cuando se quieren relacionar los datos de dos tablas distintas. Esto se hace mediante las KEYS. Si por ejemplo un objeto está identificado con una KEY a lo largo de varias tablas que contienen distintas propiedades, para obtener una nueva tabla conformada por propiedades provenientes de tablas distintas, se realiza una sub consulta. Esta sería más o menos de la forma:

```
SELECT CAMPO_1 AS C1,
              .
              .
              .
       CAMPO_n AS Cn,
      (SELECT CAMPO_a1 FROM NOMBRE_TABLA_a1 WHERE NOMBRE_TABLA_a1.KEY = KEY) AS CONSULTA_a1,
              .
              .
              .
      (SELECT CAMPO_an FROM NOMBRE_TABLA_an WHERE NOMBRE_TABLA_an.KEY = KEY) AS CONSULTA_an
FROM NOMBRE_ TABLA_1
```

Donde se seleccionan n campos de la TABLA_1 Y a_n campos provenientes de otras tablas distintas. Para que las propiedades de las distintas tablas correspondan al objeto en cuestión, las sub consultas se hacen con la condición de que las KEY de los registros en las tablas "externas" coincidan con la de la TABLA_1. Es decir, con la condición:

```
WHERE TABLA_1.KEY = KEY
```

[2]A notar, que en el código la función se indica antes que la instrucción de agrupar, lo cual pareciera contra intuitivo, pues para realizar la instrucción primero debe agruparse y luego debe aplicarse la función a cada grupo.

En este contexto, la sub consulta se realiza en la tabla externa, de modo que para "traer" la KEY de la TABLA_1 para compararla, se tiene que hacer referencia a dicha tabla, por tal motivo se coloca "TABLA_1." antes de KEY en la condición.

Dado que una sub consulta entrega un campo, estas también pueden colocarse en las condiciones.

Otra forma de utilizar las sub consultas es después de FROM. Esto ocurre cuando queremos seleccionar registros de una tabla generada a partir de una consulta (que puede tener también sub consultas) y no de una tabla que está en la base de datos. Esto es más o menos de la forma:

```
SELECT C1,...,Cn FROM (CONSULTA_TABLA)
```

5. Joins

Operación para combinar la información de dos o más tablas de una base de datos en una sola tabla. Los tipos más comunes son los INNER, LEFT, RIGHT, FULL Y CROSS joins. Usualmente, se pueden obtener los mismos resultados con sub consultas, sin embargo, los joins son más eficientes y más rápidos.

5.1. CROSS JOIN

Entrega el producto cartesiano de los registros. Por ejemplo, dadas dos tablas A y B, con n y m registros respectivamente. Esta operación entrega una tabla con $n \times m$ registros de la forma:

$camposA$	$camposB$
$1A$	$1B$
	$\vdots$
$1A$	nB
$\vdots$	
mA	$1B$
	$\vdots$
mA	nB

Esto se hace de manera implícita con:

```
SELEC * FROM TABLA_1, TABLA_2
```

y de manera explícit con:

5.2. INNER JOIN

Considerando el ejemplo de las tablas A, B y suponiendo que estas tienen un campo en común (Usualmente dicho campo es una KEY con la que se relacionan las tablas). El INNER JOIN es, en esencia, un CROSS JOIN en el que se seleccionan los registros en los cuales el valor de campo en cuestión es el mismo para ambas tablas. Nuevamente, esto se puede hacer implícitamente:

```
SELECT * FROM TABLA_1 REF_1, TABLA_2 _REF
WHERE REF_1.CAMPO_EN_COMUN = REF_2.CAMPO_COMUN
```

donde REF_1 y REF_2 son nombres provisionales de las tablas (Es como usar AS ó [ ] implícitamente.). Así como explícitamente:

```
SELECT * FROM TABLA_1 REF_1
INNER JOIN TABLA_2 REF_2
ON REF_1.CAMPO_COMUN =REF_2.CAMPO_COMUN
```

5.3. LEFT JOIN

En este caso, la tabla producida muestra los n registros de a tabla A y aquellos que cumplen la condición de tener el mismo valor de campo son completados con el registro de la tabla B. Aquellos que no cumplen dicha condición son completados con valores nulos. En este caso solo hay una manera explícita de implementarlo:

```
SELECT * FROM TABLA_1 REF_1
LEFT JOIN TABLA_2 REF_2
ON REF_1.CAMPO_COMUN =REF_2.CAMPO_COMUN
```

5.4. RIGHT JOIN

Hace lo mismo que LEFT JOIN pero aplicado a la tabla B.

Nota: en el gestor SQLite solo se puede aplicar LEFT JOIN, de manera que para aplicar un RIGHT JOIN, basta cambiar el orden de A y B y aplicar un LEFT JOIN. A esta operación se le conoce como simulación de un RIGHT JOIN.

5.5. FULL JOIN

Se utiliza para unir dos o más tablas distintas que tengan exactamente los mismos campos. Existen dos opciones, UNION ALL y UNION.

5.5.1. UNION ALL

Suponiendo que A y B tienen los mismos campos y que estas son generadas por las consultas CONSULTA_A y CONSULTA_B respectivamente. Entonces el código:

```
CONSULTA_A
UNION ALL
CONSULTA_B
```

Entrega una tabla con $n+m$ registros, siendo los primeros n, los correspondientes a la tabla A y los m posteriores a la tabla B.

5.5.2. UNION

Ya que A y B tienen los mismos campos, puede ocurrir que haya registros repetidos en ambas tablas. Si se desea que se muestre solo uno de estos registros repetidos, simplemente se quita ALL en el código anterior. Así se obtienen una tabla con todos los registros distintos de la unión de A y B.

6. Cardinalidad

Especifica la relación entre dos entidades (tablas). Hay cuatro tipos de relaciones o cardinalidades:

- **1:1 (uno a uno):** Un registro de una tabla se relaciona con uno y solo uno de los registros de otra tabla.
- **1:n (uno a muchos):** Un registro de una tabla se relaciona con más de uno de los registros de otra tabla.
- **n:1 (muchos a uno):** Más de un registro de una tabla se relaciona con solo uno de los registros de otra tabla.
- **$n : m$ (muchos a muchos):** Más de un registro de una tabla se relaciona con más de uno de los registros de otra tabla. En este último caso, debe crearse una tabla intermedia que tenga una relación n:1 con la primera tabla y una relación 1:m con la segunda tabla.

7. Normalización de bases de datos

La normalización sirve para eliminar anomalías en las bases de datos, hacerla más eficiente y hacer consultas más rápidas.

Existen 5 niveles o formas normales:

1. **Primera forma normal (1NF):** Cuando cada atributo (columna) tenga un único, dato. Por ejemplo, es preferible

Día	Mes	Año
06	02	1998

   que simplemente

Fecha
06/02/1998

2. **Segunda forma normal (2NF):** Cuando solo hay una clave primaria por tabla. Así se evita la redundancia en la información. De este modo, los atributos solo pueden depender de otro atributo de la tabla o de una sola clave primaria.
3. **forma normal:** Cuando los atributos de una tabla dependen solamente de una única clave primaria y no de oro atributo.
4. **forma normal** Cuando las categorías y subcategorías se separan en tablas distintas.
5. **forma normal** Cuando no hay dependencias de unión entre los atributos.

8. Índices

Tiene como objetivo mejorar el rendimiento de las consultas en una base de datos. Existen dos tipos, los índices únicos y los no únicos. Un ejemplo de índice único es el PRIMARY KEY.
Para indexar los valores de un campo específico se usa:

```
CREATE INDEX NOMBRE_INDICE ON TABLA (CAMPO)
```

La utilidad de esto es que en una consulta ordinaria, con una condición que depende del campo, primero se va un registro, de ahí al campo de la condición donde se corrobora si dicha condición se cumple y luego se repite el proceso en el siguiente registro. Cuando se indexa el campo de la condición, la consulta se sitúa directamente en dicho campo, corrobora si la condición se cumple y pasa al siguiente valor de campo. Esto acelera la búsqueda y ahorra recursos.

Cuando se usa solo INDEX los valores del campo pueden repetirse y tener valores nulos. Si se usa UNIQUE INDEX los valores de campo ya no pueden repetirse ni tener valores nulos.

Se pueden crear índices compuestos de manera que lo que se indexa es la combinación de los valores de campo en cuestión. En el caso de usar UNIQUE, LA COMBINACIÓN SE VUELVE ÚNICA EN LA TABLA. Esto se hace de la siguiente forma:

```
CREATE INDEX NOMBRE ON TABLA (CAMPO1,...,CAMPOn )
```

La desventaja de los índices es que su creación requiere de espacio en disco, de manera que si se establecen muchos, o se establecen en campos que no aparecen frecuentemente en las condiciones de las consultas, puede ser contraproducente.

Nota: una buena práctica es nombrar los índices con un formato de la forma *idx_ tabla_ nombre.*

Para eliminar un índice se usa:

```
DROP INDEX NOMBRE_INDICE
```

9. Vista

Una manera más limpia de trabajar sobre el código de una tabla obtenida con sub consultas, es mediante VIEW. De esta manera se separa el código que genera dicha tabla, del que manipula sus registros. Esto se hace como sigue:

```
CREATE VIEW NOMBRE_VISTA AS
CONSULTA O SUBCONSULTAS
```

Ejecutado lo anterior ya se pueden realizar consultas a la nueva tabla. Nota: en particular, si una vista es nombrada de la misma manera que una tabla de la base de datos, al realizar la consulta se dará prioridad a la vista.

Para eliminar una vista se usa:

```
DROP VIEW IF EXIST NOMBRE_VISTA
```

En este caso se agrega IF EXIST de manera que primero se corrobore si dicha vista aún sigue en la base de datos y no arroje un error

10. Transacciones

Básicamente es cuando se asientan permanentemente los cambios en una base de datos. Para iniciar una transacción se utiliza

```
BEGIN
```

seguido de los cambios a realizar. En este punto los cambios se pueden deshacer con

```
ROLLBACK
```

Una vez hechos los cambios necesarios, estos se asientan de forma permanente con

```
COMMIT
```

Cabe recalcar que una vez hecho un COMMIT, los cambios ya no se pueden deshacer con ROLLBACK.

11. SQLite con Python

Para trabajar con una base de datos de SQLite desde Python se usa el módulo **sqlite3**. Para acceder a dicha base, se requiere la ruta del archivo .db que la contiene. Para acceder se usa:

```
import sqlite3
b_datos = sqlite3.connect('archivo.db')
```

Para poder aplicar una función escrita en python, a los elementos de la base de datos, esta debe definirse primero en la base de datos. Para esto se usa:

```
b_datos.create_function('nombre_bd', n, funcion)
```

donde 'nombre_bd' es el nombre con el que se llamará la función en la base de datos, n es el número de argumentos y funcion es la función en cuestión.

Para realizar una subconsulta se hace lo siguiente:

```
# Se crea un objeto donde se almacene la
# informacion:
cursor = b_datos.cursor()

# Se instruye que consulta quiere hacerse:
cursor.execute('''
    --- Aqui se aplica la funcion
    SELECT * FROM products...
    ''')

# Se guardan los resultados de la consulta:
resultados = cursor.fetchall()

# Si se usa pandas, se hace un data frame:
resultados_df = pd.DataFrame(resultados)

# Una vez almacenada la informacion deseada
# se libera espacio del disco:
cursor.close()
b_datos.close()
```

En este punto los cambios realizados en la base de datos no se han asentado, i.e. aún puede aplicarse ROLLBACK (el BEGIN se aplicó automáticamente al cargar la base de datos). Para hacer el commit se usa:

```
b_datos.commit()
```

Al igual que para abrir archivos, se puede usar **with** para abrir una base de datos de manera más óptima (sin tener que estár cerrando la base de datos cada que se llame). En este caso sería:

```
with sqlite3.connect('archivo.db') as b_datos:

    b_datos.create_function('nom_bd', n, func)
    cursor = b_datos.cursor()
    cursor.execute('''
    --- Aqui se aplica la funcion
    SELECT * FROM products...
    ''')
    resultados = cursor.fetchall()
    resultados_df = pd.DataFrame(resultados)
```

Referencias

[1] M. Learn, "Introducción a la modelación epidémica 2024-1documentación técnica de sql server." https://learn.microsoft.com/es-es/sql/sql-server/?view=sql-server-ver16, * 2024. Accedido en junio de 2024.

[2] S. Dalto, "Documentación técnica de sql server." https://www.youtube.com/watch?v=DFg1V-rO6Pg, abril 2023. Accedido en junio de 2024.