Claves SQL básicas

Claves primarias

Una clave primaria es un campo especial de una tabla que actúa como identificador único de los registros, en otras palabras, no se puede repetir un registro con la misma clave primaria. Por ejemplo dos usuarios con el mismo DNI:

```
import sqlite3

conexion = sqlite3.connect('usuarios.db')

cursor = conexion.cursor()

# Creamos un campo dni como clave primaria
cursor.execute('''CREATE TABLE IF NOT EXISTS usuarios (
                    dni VARCHAR(9) PRIMARY KEY,
                    nombre VARCHAR(100),
                    edad INTEGER,
                    email VARCHAR(100))''')

usuarios = [('11111111A', 'Hector', 27, 'hector@ejemplo.com'),
            ('22222222B', 'Mario', 51, 'mario@ejemplo.com'),
            ('33333333C', 'Mercedes', 38, 'mercedes@ejemplo.com'),
            ('44444444D', 'Juan', 19, 'juan@ejemplo.com')]

cursor.executemany("INSERT INTO usuarios VALUES (?,?,?,?)", usuarios)

conexion.commit()
conexion.close()
```

Si ahora intentamos introducir un registro con un DNI duplicado, saltará un error:

Código

```
import sqlite3

conexion = sqlite3.connect('usuarios.db')

cursor = conexion.cursor()

# Añadimos un usuario con el mismo DNI
cursor.execute("INSERT INTO usuarios VALUES " \
    "('11111111A', 'Fernando', 31, 'fernando@ejemplo.com')")

conexion.commit()
conexion.close()
```

Resultado

```
---------------------------------------------------------------------------
---
IntegrityError                            Traceback (most recent call
last)
<ipython-input-88-1f8a69b706db> in <module>()
    6
    7 # Añadimos un usuario con el mismo DNI
----> 8 cursor.execute("INSERT INTO usuarios VALUES " \
        "('11111111A', 'Fernando', 31, 'fernando@ejemplo.com')")
    9
    10 conexion.commit()

IntegrityError: UNIQUE constraint failed: usuarios.dni
```

Claves autoincrementales

No siempre contaremos con claves primarias en nuestras tablas (como el DNI), sin embargo siempre necesitaremos uno para identificar cada registro y poder consultarlo, modificarlo o borrarlo.

Para estas situaciones lo más útil es utilizar campos autoincrementales, campos especiales que asignan automáticamente un número (de uno en uno) al crear un nuevo registro. Es muy útil para identificar de forma única cada registro ya que nunca se repiten.

En SQLite, si indicamos que un campo numérico entero es una clave primaria, automáticamente se tomará como un campo auto incremental. Podemos hacerlo fácilmente así:

```
import sqlite3

conexion = sqlite3.connect('productos.db')

cursor = conexion.cursor()

# Las cláusulas not null indican que no puede ser campos vacíos
cursor.execute('''CREATE TABLE IF NOT EXISTS productos (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    nombre VARCHAR(100) NOT NULL,
                    marca VARCHAR(50) NOT NULL,
                    precio FLOAT NOT NULL)''')

conexion.close()
```

¡Problema al insertar registros con campos autoincrementales!

Al utilizar un nuevo campo autoincremental, la sintaxis sencilla para insertar registros ya no funciona, pues en primer lugar se espera un identificador único, por lo que recibimos un error indicándonos se esperan 4 columnas en lugar de 3:

Código

```
import sqlite3

conexion = sqlite3.connect('productos.db')

cursor = conexion.cursor()

productos = [('Teclado', 'Logitech', 19.95),
            ('Pantalla 19"' 'LG', 89.95),]

cursor.executemany("INSERT INTO productos VALUES (?,?,?)", productos)

conexion.commit()
conexion.close()
```

Resultado

```
---------------------------------------------------------------------------
---
OperationalError                          Traceback (most recent call
last)
<ipython-input-96-7b99f15c4bb5> in <module>()
    8             ('Pantalla 19"' 'LG', 89.95),]
    9
---> 10 cursor.executemany("INSERT INTO productos VALUES (?,?,?)",
productos)
    11
    12 conexion.commit()

OperationalError: table productos has 4 columns but 3 values were
supplied
```

Para arreglarlo cambiaremos la notación durante la inserción, especificando el valor null para el auto incremento:

```
import sqlite3

conexion = sqlite3.connect('productos.db')

cursor = conexion.cursor()

productos = [('Teclado', 'Logitech', 19.95),
            ('Pantalla 19"','LG', 89.95),
            ('Altavoces 2.1','LG', 24.95),]

cursor.executemany("INSERT INTO productos VALUES (null,?,?,?)",
productos)

conexion.commit()
conexion.close()
```

Ahora podemos consultar nuestros productos fácilmente con su identificador único:

Código

```
import sqlite3

conexion = sqlite3.connect('productos.db')
cursor = conexion.cursor()

# Recuperamos los registros de la tabla de usuarios
cursor.execute("SELECT * FROM productos")

# Recorremos todos los registros con fetchall
# y los volcamos en una lista de usuarios
productos = cursor.fetchall()

# Ahora podemos recorrer todos los productos
for producto in productos:
    print(producto)

conexion.close()
```

Resultado

```
(1, 'Teclado', 'Logitech', 19.95)
(2, 'Pantalla 19"', 'LG', 89.95)
(3, 'Altavoces 2.1', 'LG', 24.95)
```

Claves únicas

El problema con las claves primarias es que sólo podemos tener un campo con esta propiedad, y si da la casualidad que utilizamos un campo autoincremental, ya no podemos asignarla a otro campo.

Para estos casos existen las claves únicas, que nos permiten añadir otros campos únicos no repetibles.

```
import sqlite3

conexion = sqlite3.connect('usuarios_autoincremental.db')

cursor = conexion.cursor()

# Creamos un campo dni como clave primaria
cursor.execute('''CREATE TABLE IF NOT EXISTS usuarios (
                    id INTEGER PRIMARY KEY,
                    dni VARCHAR(9) UNIQUE,
                    nombre VARCHAR(100),
                    edad INTEGER(3),
                    email VARCHAR(100))''')
```

```
usuarios = [('11111111A', 'Hector', 27, 'hector@ejemplo.com'),
            ('22222222B', 'Mario', 51, 'mario@ejemplo.com'),
            ('33333333C', 'Mercedes', 38, 'mercedes@ejemplo.com'),
            ('44444444D', 'Juan', 19, 'juan@ejemplo.com')]

cursor.executemany("INSERT INTO usuarios VALUES (null, ?,?,?,?)",
usuarios)

conexion.commit()
conexion.close()
```

Si intentamos añadir un usuario con la misma clave da error de integridad:

Código

```
import sqlite3

conexion = sqlite3.connect('usuarios_autoincremental.db')

cursor = conexion.cursor()

# Añadimos un usuario con el mismo DNI
cursor.execute("INSERT INTO usuarios VALUES " \
    "(null, '11111111A', 'Fernando', 31, 'fernando@ejemplo.com')")

conexion.commit()
conexion.close()
```

Resultado

```
---------------------------------------------------------------------------
---
IntegrityError                            Traceback (most recent call
last)
<ipython-input-100-fdd36d467cfc> in <module>()
    6
    7 # Añadimos un usuario con el mismo DNI
----> 8 cursor.execute("INSERT INTO usuarios VALUES " \
            "(null, '11111111A', 'Fernando', 31,
'fernando@ejemplo.com')")
    9
    10 conexion.commit()

IntegrityError: UNIQUE constraint failed: usuarios.dni
```

Con la ventaja de contar con un identificador automático para cada registro:

Código

```
import sqlite3
```

```
conexion = sqlite3.connect('usuarios_autoincremental.db')
cursor = conexion.cursor()

# Recuperamos los registros de la tabla de usuarios
cursor.execute("SELECT * FROM usuarios")

# Recorremos todos los registros con fetchall
# y los volcamos en una lista de usuarios
usuarios = cursor.fetchall()

# Ahora podemos recorrer todos los usuarios
for usuario in usuarios:
    print(usuario)

conexion.close()
```

Resultado

```
(1, '11111111A', 'Hector', 27, 'hector@ejemplo.com')
(2, '22222222B', 'Mario', 51, 'mario@ejemplo.com')
(3, '33333333C', 'Mercedes', 38, 'mercedes@ejemplo.com')
(4, '44444444D', 'Juan', 19, 'juan@ejemplo.com')
```