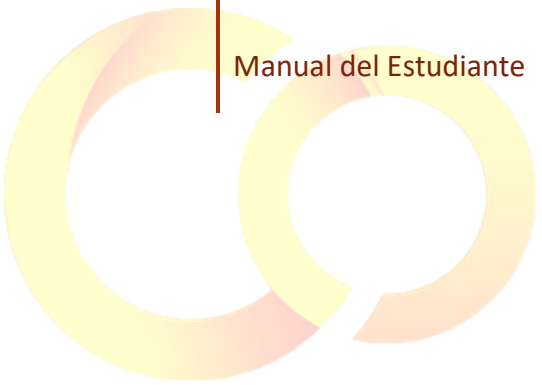


Unidad 9

Base de Datos Relacionales

Manual del Estudiante



CLOUD
FORMACIÓN

Maria del Carmen Sierra Fernández

Contenido

| | |
|--|----|
| Nivel III: Aplicaciones Web | 4 |
| Unidad 9: Base de Datos Relacionales | 4 |
| Introducción a BBDD Relacionales | 4 |
| Qué es una base de datos relacional | 4 |
| Beneficios de las bases de datos relacionales..... | 5 |
| Consistencia de los datos | 5 |
| Atomicidad | 5 |
| Procedimientos almacenados | 5 |
| Bloqueo de bases de datos y concurrencia..... | 6 |
| Descripción de las relaciones de tablas | 7 |
| Tipos de relaciones de tablas | 8 |
| Relaciones de uno a varios..... | 8 |
| Relaciones de varios a varios | 9 |
| Relaciones de uno a uno | 9 |
| Cómo funcionan las bases de datos relacionales..... | 10 |
| Diferencias entre una base de datos SQL y una NOSQL | 12 |
| API Database Python para acceso a bases de datos relacionales..... | 13 |
| SQLite..... | 14 |
| ¿Por qué utilizar SQLite? | 14 |
| Instalación Windows | 14 |
| Crear una BBDD desde el Shell de SQLite | 15 |
| Utilizar SQLite en Python..... | 15 |
| APIs del módulo Python sqlite3 | 16 |
| Conectarse a la base de datos..... | 17 |
| Crear una tabla (CREATE TABLE) | 18 |
| Insertar registros (INSERT INTO) | 18 |
| Mostrar registros (SELECT)..... | 19 |
| Personalizar los resultados de una Consulta | 23 |
| Mostrar las tablas de una BBDD..... | 24 |
| Actualizar registro (UPDATE)..... | 24 |
| Eliminar registros (DELETE) | 25 |
| Gestión de transacciones | 25 |
| Eliminar una Tabla de una BBDD (DROP)..... | 26 |
| DB Browser para SQLite (DB4S) | 26 |

| | |
|--|----|
| Ejemplo de una Base de Datos relacional de Gestión de Proyectos..... | 29 |
| Conexiones ODBC..... | 35 |
| Controlador Python para SQL: pyodbc..... | 35 |
| Otras Conexiones | 36 |
| Controlador Python para SQL: pymssql | 36 |
| Controlador Python para Bases de datos de Oracle DB..... | 37 |
| Controlador Python para Bases de datos MySQL | 37 |
| SQLObject..... | 37 |
| Usando SQLObject..... | 37 |
| Declaración de las clases (tablas)..... | 39 |
| Métodos de la clase SQLObject..... | 41 |
| Insertar registros | 41 |
| Eliminar registros..... | 43 |
| Configuración de metadatos..... | 43 |
| Consultas con SQLObject | 44 |
| Funciones de comparación avanzadas..... | 47 |
| Funciones de agrupación. | 48 |
| Métodos: | 48 |
| Ordenar Consultas..... | 49 |
| Relaciones entre tablas | 50 |
| Relaciones uno-a-uno..... | 51 |
| Consultas entre clases relacionadas..... | 52 |
| Forzando las consultas de JOIN..... | 53 |
| Subconsultas..... | 54 |
| Técnicas avanzadas con SQLObject..... | 55 |
| Añadiendo propiedades..... | 55 |
| Modificando los atributos de columnas..... | 57 |
| Gestión de eventos..... | 58 |
| Herencia | 59 |
| Diseño de herencia entre clases. | 59 |
| Consultas con herencia | 61 |
| Relaciones con otras tablas..... | 61 |
| Limitaciones de la herencia..... | 62 |

Nivel III: Aplicaciones Web

Unidad 9: Base de Datos Relacionales

Introducción a BBDD Relacionales

Una base de datos la podemos definir como:

- Una colección de datos o registros. Es decir, una recopilación de información que se organiza para que se pueda acceder, administrar y actualizar fácilmente.
- Un conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior uso (Wikipedia)

Las bases de datos son parte esencial de cualquier sistema de información, los programas necesitan obtener datos mientras se ejecutan o generan datos que necesitamos almacenar de forma permanente y fiable. En una base de datos podemos almacenar todo tipo de datos como videos, imágenes, textos, etc. De allí que estén presentes en prácticamente todas las aplicaciones web y móviles como Facebook, WhatsApp, etc.

Existen diferentes modelos de bases de datos: transaccionales, documentales, relacionales, jerárquicas, NoSQL entre muchos otros. En esta Unidad conoceremos las bases de datos relacionales.

Qué es una base de datos relacional

Una base de datos relacional es un tipo de base de datos que almacena y proporciona acceso a puntos de datos relacionados entre sí.

- Las bases de datos relacionales se basan en el modelo relacional, una forma intuitiva y directa de representar datos en tablas.
- En una base de datos relacional, cada fila de la tabla es un registro con un ID único llamado clave.
- Las columnas de la tabla contienen atributos de los datos, y cada registro generalmente tiene un valor para cada atributo, lo que facilita el establecimiento de las relaciones entre los puntos de datos.
- La relación entre una tabla padre y un hijo se lleva a cabo por medio de las claves primarias y claves foráneas.
- Las claves primarias son la clave principal de un registro dentro de una tabla y estas deben cumplir con la integridad de datos.
- Las claves foráneas se colocan en la tabla hija, contienen el mismo valor que la clave primaria del registro padre; por medio de estas se hacen las formas relacionales.

Nivel III: Aplicaciones Web

- Dentro de la base de datos, las tablas se pueden normalizar, es decir, hacer que cumplan las reglas de normalización que hacen a la base de datos flexible, adaptable y escalable.
- Las bases relacionales generalmente se escriben en Lenguaje de Consulta Estructurado (SQL, por sus siglas en inglés). El modelo fue introducido por E. F. Codd en 1970.

Beneficios de las bases de datos relacionales

Se puede considerar una base de datos relacional para cualquier necesidad de información en la que los puntos de datos se relacionen entre sí y se deban administrar de una manera segura, consistente y basada en reglas. Las bases de datos relacionales se utilizan, por ejemplo, para hacer seguimiento de los inventarios, procesar transacciones de comercio electrónico, administrar grandes cantidades de información de clientes de misión crítica.

Consistencia de los datos

El modelo relacional es el mejor para mantener la consistencia de los datos en todas las aplicaciones. Por ejemplo, cuando un cliente deposita dinero en un cajero automático y, luego, mira el saldo de la cuenta en un teléfono móvil, el cliente espera ver que ese depósito se refleje inmediatamente en un saldo de cuenta actualizado. Las bases de datos relacionales se destacan en este tipo de consistencia de datos, lo que garantiza que múltiples instancias de una base de datos tengan los mismos datos todo el tiempo.

Es difícil para otros tipos de bases de datos mantener este nivel de coherencia oportuna con grandes cantidades de datos. Algunas bases de datos recientes, como NoSQL, solo pueden proveer “consistencia eventual.” Bajo este principio, cuando la base de datos se escala o cuando varios usuarios acceden a los mismos datos al mismo tiempo, los datos necesitan algo de tiempo para “ponerse al día”. La consistencia eventual es aceptable para algunos usos, como para mantener listados en un catálogo de productos, pero para operaciones comerciales críticas como transacciones de un carrito de compras, la base de datos relacional sigue siendo la mejor alternativa.

Atomicidad

Las bases de datos relacionales manejan las reglas y políticas comerciales en un nivel muy detallado, con políticas estrictas para hacer cambios permanentes en la base de datos. La atomicidad es la clave para mantener la precisión de los datos en la base de datos y garantizar que cumpla con las reglas, regulaciones y políticas de la empresa.

Procedimientos almacenados

El acceso a los datos implica muchas acciones repetitivas. Por ejemplo, una consulta simple para obtener información de una tabla de datos puede necesitar repetirse cientos o miles de veces para producir el resultado deseado. Las bases de datos relacionales permiten

Nivel III: Aplicaciones Web

procedimientos almacenados, que son bloques de código a los que se puede acceder con una simple llamada de aplicación..

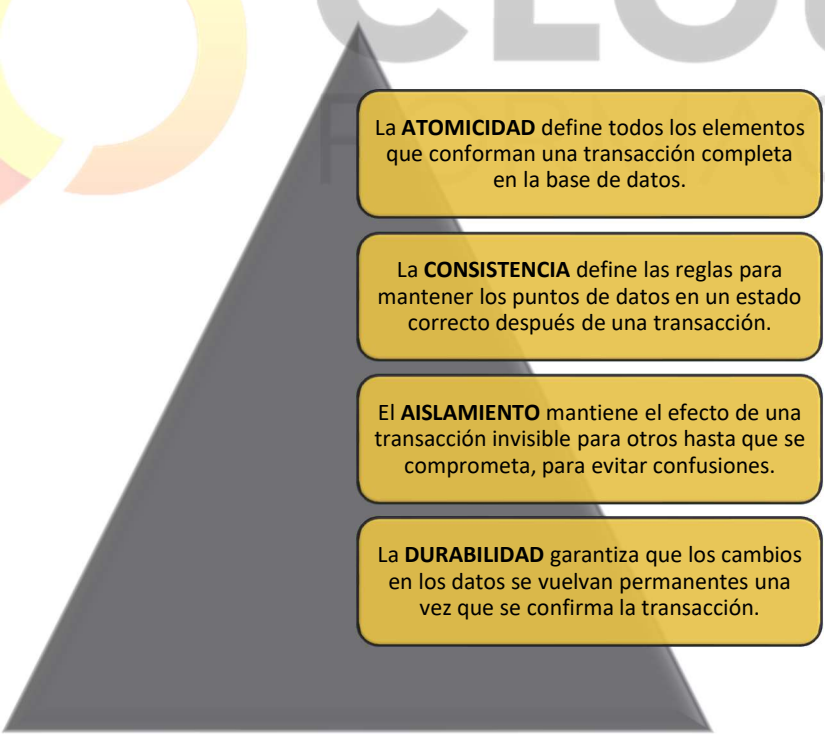
Bloqueo de bases de datos y concurrencia

Pueden surgir conflictos en una base de datos cuando varios usuarios o aplicaciones intentan cambiar los mismos datos al mismo tiempo. Las técnicas de bloqueo y concurrencia reducen la posibilidad de conflictos mientras mantienen la integridad de los datos.

El bloqueo evita que otros usuarios y aplicaciones accedan a los datos mientras se actualizan. En algunas bases de datos, el bloqueo se aplica a toda la tabla, lo que crea un impacto negativo en el rendimiento de la aplicación. Otras bases de datos, como las bases de datos relacionales de Oracle, aplican bloqueos a nivel de registro, lo que deja disponibles los otros registros dentro de la tabla, lo que ayuda a garantizar un mejor rendimiento de la aplicación.

La concurrencia gestiona la actividad cuando varios usuarios o aplicaciones realizan consultas al mismo tiempo en la misma base de datos. Esta capacidad proporciona el acceso correcto a los usuarios y las aplicaciones de acuerdo con las políticas definidas para el control de datos.

En otras palabras, las transacciones de bases de datos relacionales se pueden definir mediante cuatro propiedades: atomicidad, consistencia, aislamiento y durabilidad, los cuales conocemos como **ACID (Atomicity, Consistency, Isolation, Durability)**.



La **ATOMICIDAD** define todos los elementos que conforman una transacción completa en la base de datos.

La **CONSISTENCIA** define las reglas para mantener los puntos de datos en un estado correcto después de una transacción.

El **AISLAMIENTO** mantiene el efecto de una transacción invisible para otros hasta que se comprometa, para evitar confusiones.

La **DURABILIDAD** garantiza que los cambios en los datos se vuelvan permanentes una vez que se confirma la transacción.

El software que se utiliza para almacenar, administrar, consultar y recuperar datos almacenados en una base de datos relacional se denomina sistema de gestión de bases de datos relacionales (RDBMS). El RDBMS proporciona una interfaz entre usuarios y aplicaciones

Nivel III: Aplicaciones Web

y la base de datos, así como funciones administrativas para administrar el almacenamiento, el acceso y el rendimiento de los datos.

Entre los gestores de bases de datos relacionales más utilizados tenemos:

- **Db2:** Es un sistema de gestión de BBDD relacional propietario de IBM.
- **Microsoft SQL Server:** la aplicación de Microsoft para gestionar bases de datos relacionales está disponible con una licencia Microsoft de pago.
- **MySQL:** MySQL un sistema de gestión de BBDD muy utilizado. MySQL se distribuye con una licencia dual.
- **PostgreSQL:** un sistema de gestión de BBDD libre y orientado a objetos de cuyo continuo desarrollo se ocupa su comunidad open source.
- **Oracle Database:** el programa de Oracle se distribuye como software propietario.
- **SQLite:** es una biblioteca de programas con licencia de dominio público que contiene un gestor de bases de datos relacionales.

Descripción de las relaciones de tablas

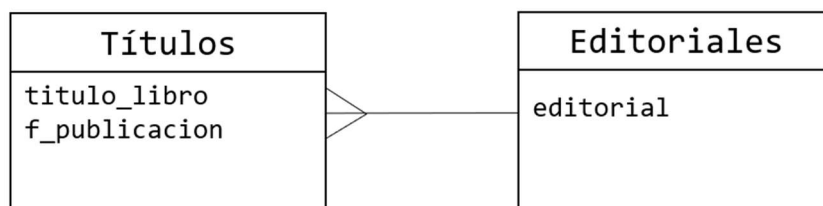
En una base de datos relacional, las relaciones permiten evitar datos redundantes. Por ejemplo, si estamos diseñando una base de datos que realizará un seguimiento de la información sobre libros, es posible que tengamos una tabla llamada "Títulos" que almacene información sobre cada libro, como el título del libro, la fecha de publicación y la editorial. También necesitamos almacenar información sobre la editorial, como el número de teléfono, la dirección y el código postal. Si almacenara toda esta información en la tabla "Títulos", el número de teléfono de la editorial se duplicaría para cada título que imprima la editorial.

| Títulos |
|--|
| titulo_libro f_publicacion editorial |

| titulo_libro | f_publicacion | editorial |
|---|----------------|-----------|
| Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition | September 2019 | O'Reilly |
| Machine Learning with Python Cookbook | March 2018 | O'Reilly |
| Introduction to Machine Learning with Python | October 2016 | O'Reilly |

En este caso, lo mejor es almacenar la información de la editorial una sola vez, en una tabla separada "Editoriales". Para asegurarnos que los datos permanecen sincronizados, podemos forzar la integridad de referencias entre las tablas. Las relaciones de integridad de referencias ayudan a asegurarse de que la información de una tabla coincide con la información de otra. Por ejemplo, cada libro de la tabla "Títulos" debe estar asociado a una editorial específica de

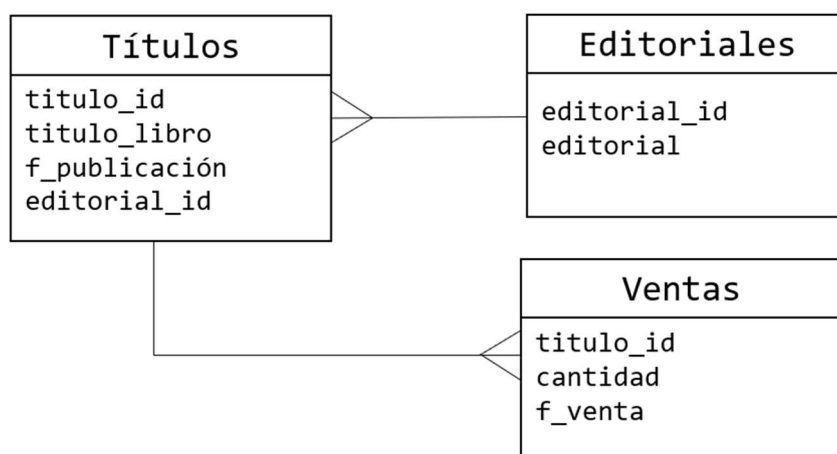
la tabla "Editoriales". No se puede agregar un título a la base de datos de una editorial que no existe en la base de datos.



Las relaciones lógicas de una base de datos nos permiten consultar datos de forma eficaz y crear informes.

Tipos de relaciones de tablas

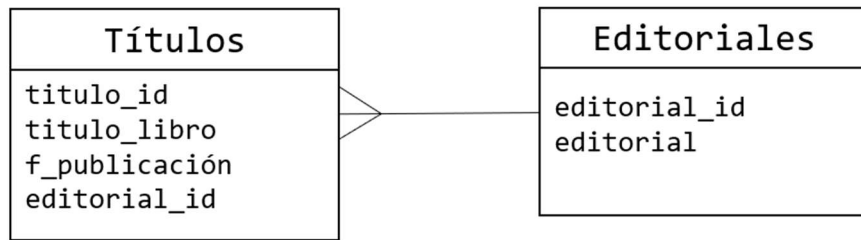
Una relación funciona haciendo coincidir datos en columnas de clave, normalmente columnas (o campos) que tienen el mismo nombre en ambas tablas. En la mayoría de los casos, la relación conecta la clave principal, o la columna de identificador único para cada fila, de una tabla a un campo de otra tabla. La columna de la otra tabla se conoce como la "clave foránea". Por ejemplo, si deseamos realizar un seguimiento de las ventas de cada libro, creamos una relación entre la columna de clave principal **titulo_id** en la tabla **Títulos** y una columna de la tabla **Ventas** denominada **titulo_id**. La columna **titulo_id** de la tabla **Ventas** es la clave foránea.



Hay tres tipos de relaciones entre tablas. El tipo de relación que se crea depende de cómo se definan las columnas relacionadas.

Relaciones de uno a varios

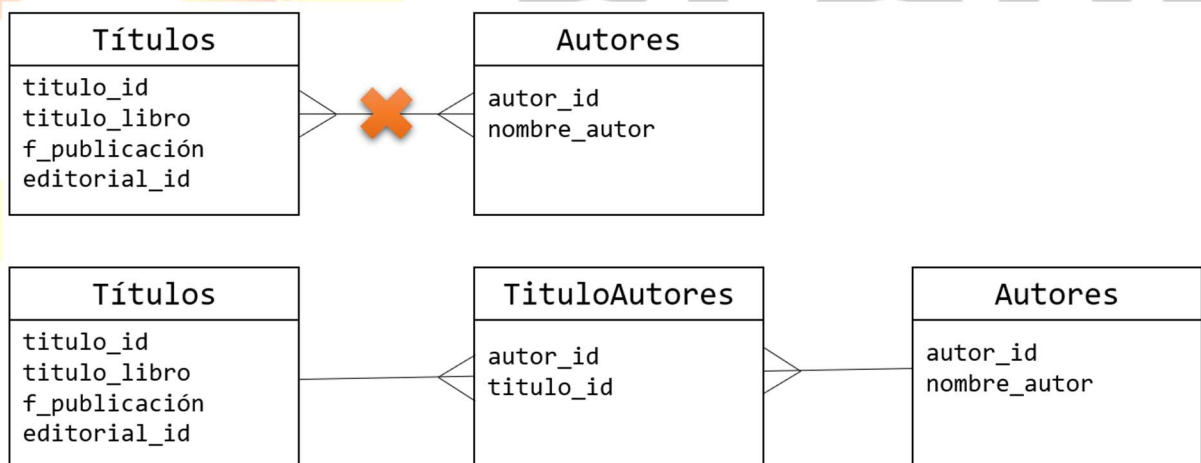
Una relación de uno a varios es el tipo de relación más común. En este tipo de relación, una fila de la tabla A puede tener muchas filas coincidentes en la tabla B. Pero una fila de la tabla B solo puede tener una fila coincidente en la tabla A. Por ejemplo, las tablas **Editoriales** y **Títulos** tienen una relación de uno a varios. Es decir, cada editorial produce muchos títulos. Pero cada título proviene de una sola editorial.



Se crea una relación de uno a varios si solo una de las columnas relacionadas es una clave principal o tiene una restricción única.

Relaciones de varios a varios

En una relación de varios a varios, una fila de la tabla A puede tener muchas filas coincidentes en la tabla B y viceversa. Esta relación se crea definiendo una tercera tabla denominada tabla de conexión o intermedia. La clave principal de la tabla de conexión consta de las claves extranjeras de la tabla A y de la B. Por ejemplo, la tabla **Autores** y la tabla **Títulos** tienen una relación de varios a varios que se define mediante una relación de uno a varios de cada una de estas tablas a la tabla **TítuloAutores**. La clave principal de la tabla **TítuloAutores** es la combinación de la columna **autor_id** (la clave principal de la tabla Autores) y la columna **titulo_id** (la clave principal de la tabla Títulos).



Relaciones de uno a uno

En una relación uno a uno, una fila de la tabla A no puede tener más de una fila coincidente en la tabla B y viceversa. Se crea una relación uno a uno si ambas columnas relacionadas son claves principales o tienen restricciones únicas.

Este tipo de relación no es común porque la mayoría de la información que está relacionada de esta manera estaría en una tabla. Podemos usar una relación uno a uno para realizar las siguientes acciones:

- Dividir una tabla con muchas columnas.
- Aislar parte de una tabla por motivos de seguridad.

Nivel III: Aplicaciones Web

- Almacenar datos de corta duración y que podrían eliminarse fácilmente borrando la tabla.
- Almacenar información que solo se aplique a un subconjunto de la tabla principal.

Cómo funcionan las bases de datos relacionales

Los datos estructurados en tablas constituyen la BD de un sistema relacional. El RDBMS define su estructura y gestiona también los permisos de escritura y lectura y para interactuar con él, los usuarios utilizan un lenguaje de bases de datos. Todo gestor de bases de datos relacionales soporta al menos un lenguaje formal que permite ejecutar las siguientes operaciones:

1. **Definir la estructura de datos:** en la definición de los datos se guarda una descripción con metadatos de la estructura de datos en el diccionario del sistema (esquema).
2. **Definir derechos:** todos los lenguajes de bases de datos proporcionan una sintaxis que permite otorgar o retirar permisos.
3. **Definir condiciones de integridad:** por condiciones de integridad se entienden los requisitos de estado que se exigen a un banco de datos. Si se definen condiciones para su integridad, la BD garantiza que se cumplan en todo momento. Una condición básica de integridad en una base de datos relacional es, por ejemplo, que cada registro (tupla) pueda identificarse de forma inequívoca.
4. **Definir transacciones:** Las transacciones contienen una serie de instrucciones que deben ejecutarse siempre de forma íntegra. Si una se interrumpe, la BD vuelve a su estado original (Rollback). Cada transacción debe seguir los siguientes pasos:
 1. Crear una conexión con la BD
 2. Iniciar las operaciones de datos
 3. Comprobar la operación (Commit) que asegura la integridad de la BD. Las operaciones que pongan en peligro la integridad de la tabla, no se escriben en la base de datos de forma permanente.
 4. Cerrar la conexión con la BD.
5. **Definir vistas:** las llamadas views son vistas virtuales de un subconjunto de los datos de una tabla. Para crear una vista, el sistema de gestión de BBDD genera una tabla virtual (relación lógica) sobre la base de las tablas físicas. En estas vistas pueden emplearse las mismas operaciones que se utilizarían en tablas físicas.

Un modelo relacional utiliza de forma estándar para sus operaciones el lenguaje de bases de datos **SQL (Structured Query Language)** o lenguaje de consulta estructurado.

Las operaciones típicas de las BBDD como consultar, crear, actualizar o borrar datos se realizan por medio de las llamadas sentencias SQL (SQL statements), una combinación de órdenes SQL, semánticamente vinculadas al inglés.

SQL nos permite:



Para crear una base de datos necesitamos comandos básicos de SQL como: SELECT, INSERT, UPDATE, DELETE, CREATE, entre otros. Es importante destacar, que cada RDBMS puede tener pequeñas variaciones en su SQL con la cual puede agregarle funcionalidades, haciendo que el sistema de base de datos sea único.

Los comandos del lenguaje SQL se dividen según su función en estos 5 tipos:

DDL (Data Definition Language): definen el esquema o estructura de la base de datos. Ejemplos: CREATE (crear), ALTER (alterar), DROP (eliminar objetos), RENAME (renombrar), TRUNCATE (quita todos los registros de una tabla, incluidos los espacios de los registros eliminados), COMMENT (comentar); entre otros.

DQL (Data Query Language): sirven para hacer consultas sobre los datos en el esquema de objetos. Un objeto puede ser desde un resultado de búsqueda a una tabla. Ejemplo: SELECT (recuperar registros de la base de datos).

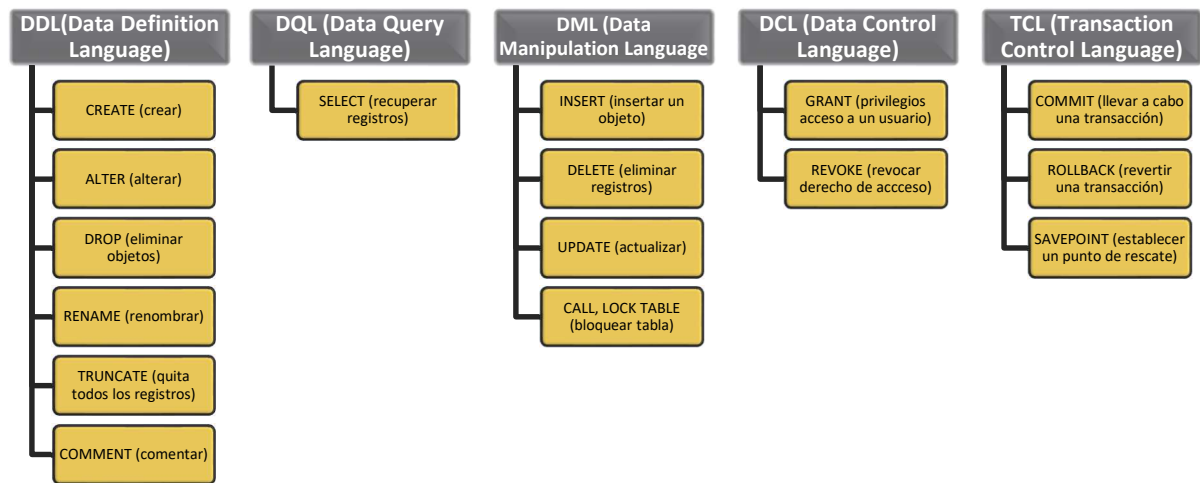
DML (Data Manipulation Language): tratan la manipulación de los datos presentes en la base de datos. La mayoría de los comandos pertenecen a este tipo. Ejemplo: INSERT (insertar un objeto), DELETE (eliminar registros), UPDATE (actualizar), CALL, LOCK TABLE (bloquear tabla), entre otros.

DCL (Data Control Language): se encargan de los derechos, los permisos y otros controles del sistema de la base de datos. Ejemplos: GRANT (proporcionar privilegios acceso a un usuario), REVOKE (revocar el derecho de acceso dado a un usuario), entre otros.

TCL (Transaction Control Language): sirve para las transacciones con la base de datos. Es decir, con estos comandos se puede llevar un control sobre otros comandos y cómo afectan a la

Nivel III: Aplicaciones Web

base de datos. Ejemplos: COMMIT (llevar a cabo una transacción), ROLLBACK (revertir una transacción en caso de que ocurra algún error), SAVEPOINT (establecer un punto de rescate dentro de una transacción).



Diferencias entre una base de datos SQL y una NOSQL

A la base de datos tradicional, la relacional SQL, le ha salido una competidora, la NoSQL (Not Only SQL, no solo SQL). La popularidad de la base de datos NoSQL se debe a grandes empresas como Facebook, Twitter, Google o Amazon. Estas empresas la usan para proporcionar grandes volúmenes de información en el menor tiempo posible, aunque para ellos sacrifican la consistencia de los datos. Las redes sociales utilizan bases de datos NoSQL porque estas le permiten procesar grandes cantidades de información creando la estructura de grafos (tipo abstracto de datos) con el que se establecen relaciones entre las personas.

En cuanto a la rapidez, una base de datos será más rápida que la otra dependiendo del contexto. Por ejemplo, las bases de datos SQL serán más rápidas que las NoSQL cuando se trata de uniones, consultas, actualizaciones, etc. En cambio, las bases de datos NoSQL son más rápidas que las SQL cuando se trata de realizar operaciones de lectura o escritura en una sola entidad de datos. Esto se debe a que las bases de datos NoSQL están diseñadas especialmente para datos desestructurados, los cuales pueden estar orientados a: documentos, columnas, gráficos, etc. Así, una entidad particular de datos está almacenada de forma conjunta y no está en particiones.

Como se ha mencionado, la base de datos NoSQL no está estructurada, al igual que tampoco está centralizada, permitiendo una mayor escalabilidad. Además, este tipo de base de datos no necesita tantos recursos para ejecutarse ya que es abierta y, por lo tanto, flexible.

La base de datos NoSQL son útiles cuando el volumen de datos crece rápidamente; cuando necesitamos una gran escalabilidad; o, cuando el esquema de la base de datos no es homogéneo. Algunos ejemplos de DBMS de NoSQL son: MongoDB, Redis y Apache CouchDB.

Diferencias entre BBDD Relacionales (SQL) y NO SQL

| Base de datos relacional | NoSQL |
|---|---|
| Almacena datos en forma tabular (filas y columnas) | Almacenan datos de varias formas: key-value, documento, basado en columnas o en grafos |
| Esquema predefinido | Esquema dinámico |
| Escalable verticalmente | Escalable horizontalmente |
| SQL soporta consultas complejas y anidadas | Soportan las operaciones CRUD básicas: Trasladan la complejidad a la aplicación. No soportan joins. |
| Son ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad) | Son BASE (Basically Available Soft-state Eventually consistency). |
| Se utilizan en sistemas transaccionales (banca...) | Se utilizan en servicios Web2.0 (redes sociales, blogs, etc.), aplicaciones IoT, almacenamiento de perfiles sociales... |

API Database de Python para acceso a bases de datos relacionales.

La API de base de datos de Python (DB-API) define una interfaz estándar para los módulos de acceso a la base de datos de Python. Está documentado en la PEP 249.

Como la especificación Database API (DB-API) estandariza el acceso en Python a bases de datos relacionales, los pasos a seguir serán siempre los siguientes:

1. Importar un módulo conector.
2. Obtener una conexión con el método `connect()` del conector.
3. Abrir un cursor usando el método `cursor()` de la conexión.
4. Ejecutar comando SQL con el método `execute()` del cursor.
5. Recuperar datos de una consulta con los métodos `fetchall()`, `fetchmany()` o `fetchone()` del cursor.
6. Cerrar el cursor con su método `close()`.

El módulo conector es específico para cada base de datos. La siguiente tabla muestra módulos conectores para las bases de datos más populares:

| Base de datos | Módulo conector |
|-----------------------------|-----------------|
| SQLite | sqlite3 |
| Microsoft SQL Server | pymssql |
| MySQL | PyMySQL |
| Conexiones ODBC | pyodbc |
| Oracle Database 11g | cx_Oracle |

Podemos descargar e instalar estos módulos conectores con la herramienta pip:

```
pip install módulo_conector
```

SQLite

SQLite es una librería en lenguaje C que implementa un motor de base de datos SQL pequeño, rápido, autónomo, de alta confiabilidad y con todas las funciones. SQLite es el motor de base de datos más utilizado del mundo. SQLite está integrado en todos los teléfonos móviles y en la mayoría de las computadoras y viene incluido dentro de innumerables otras aplicaciones que la gente usa todos los días.

¿Por qué utilizar SQLite?

- No hay necesidad de un servidor independiente o sistema operativo (sin servidor).
- SQLite no requiere configuración, es decir ninguna instalación o gestión.
- SQLite es una librería compacta.
- SQLite es autosuficiente, lo que significa que no hay dependencias externas.
- Las transacciones SQLite son totalmente compatible con ACID.
- SQLite escrito usando ANSI-C, es simple y fácil de usar.
- SQLite puede ejecutarse UNIX (Linux, Mac OS-X, Android, iOS) y Windows.

Instalación Windows

Para instalar SQLite en Windows debemos seguir, los siguientes pasos:

1. Descargamos el SQLite, desde la página **SQLite Download Page** <https://sqlite.org/download.html>, en la sección **Precompiled Binaries for Windows**, seleccionamos el fichero **sqlite-tools-win32-x86-3330000.zip**. Este contiene las herramientas para la línea de comandos, el archivo **sqlite3.exe**, **sqldiff.exe** y **sqlite3_analyzer.exe**
2. Una vez completada la descarga del fichero **sqlite-tools-win32-x86-3330000.zip**. Creamos una carpeta por ejemplo: **C:\Program Files (x86)\SQLite**, donde descomprimiremos el fichero.
3. Renombramos la carpeta creada **sqlite-tools-win32-x86-3330000** por **sqlite**. Dentro de esa carpeta debe estar tus tres archivos .exe
4. Agregar la variable de entorno al PATH.
 - a. En propiedades del sistema, seleccionamos la opción **Variables de entorno...**
 - b. Seleccionamos Path, haciendo doble clic.
 - c. Desde la ventana variable de entorno, seleccionamos **Nuevo** e introducimos **C:\Program Files (x86)\SQLite\sqlite**, seleccionamos **Aceptar**.
5. Para validar la instalación, ejecutamos el **Simbolo del sistemas (cmd)** y escribimos **sqlite3**

```
Microsoft Windows [Version 10.0.19041.630]  
(c) 2020 Microsoft Corporation. Todos los derechos reservados.
```

```
C:\Users\msierra>sqlite3
SQLite version 3.33.0 2020-08-14 13:23:32
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

6. Desde el prompt `sqlite>` escribimos `.help`

```
sqlite> .help
.archive ...           Manage SQL archives
.auth ON|OFF           Show authorizer callbacks
.backup ?DB? FILE      Backup DB (default "main") to FILE
.bail on|off           Stop after hitting an error. Default OFF
.binary on|off         Turn binary output on or off. Default OFF
.cd DIRECTORY          Change the working directory to DIRECTORY
.changes on|off        Show number of rows changed by SQL
.check GLOB            Fail if output since .testcase does not
...
sqlite>
```

Crear una BBDD desde el Shell de SQLite

1. Ejecutamos el **Símbolo del sistemas (cmd)**
2. Nos ubicamos en el directorio donde queremos crear la BBDD, por ejemplo: `cd C:\Users\mssss\Desktop\SQLDatabase`
3. Desde el prompt escribimos `...>sqlite3 MiBasedeDatos.db`

```
C:\Users\msierra\Desktop\SQLDatabase>sqlite3 MiBasedeDatos.db
SQLite version 3.33.0 2020-08-14 13:23:32
Enter ".help" for usage hints.
sqlite>
```

4. Desde el prompt escribimos `...>.databases`

```
sqlite> .databases
main: C:\Users\msierra\Desktop\SQLDatabase\MiBasedeDatos.db
sqlite>
```

5. Para salir CTRL-Z

Utilizar SQLite en Python

SQLite3 se puede integrar con Python usando el módulo `sqlite3`. No es necesario instalar este módulo por separado porque viene integrado a partir de la versión 2.5.x de Python.

Para usar el módulo `sqlite3`, debemos crear un objeto de conexión que represente la base de datos y luego, opcionalmente, podemos crear un objeto cursor, que nos ayudará a ejecutar todas las sentencias SQL.

APIs del módulo Python sqlite3

Las siguientes son rutinas del módulo sqlite3, que nos permitirán trabajar con la base de datos SQLite desde su programa Python.

| API | Descripción |
|---|---|
| sqlite3.connect(database[,timeout,other optional arguments]) | <p>Esta API abre una conexión al archivo de base de datos SQLite. Puede utilizar ":memory:" para abrir una conexión de base de datos a una base de datos que reside en la RAM en lugar de en el disco. Si la base de datos se abre correctamente, devuelve un objeto de conexión.</p> <p>Cuando se accede a una base de datos mediante varias conexiones y uno de los procesos modifica la base de datos, la base de datos SQLite se bloquea hasta que se confirma la transacción. El parámetro de tiempo de espera especifica cuánto tiempo debe esperar la conexión para que desaparezca el bloqueo hasta generar una excepción. El valor predeterminado para el parámetro de tiempo de espera es 5,0 (cinco segundos). Si el nombre de base de datos proporcionado no existe, esta llamada creará la base de datos. También puede especificar el nombre de archivo con la ruta requerida si desea crear una base de datos en cualquier otro lugar excepto en el directorio actual.</p> |
| connection.cursor([cursorClass]) | <p>Esta rutina crea un cursor que se utilizará en toda la programación de su base de datos con Python. Este método acepta un único parámetro opcional cursorClass. Si se proporciona, debe ser una clase de cursor personalizada que amplíe sqlite3.Cursor.</p> |
| cursor.execute(sql[,optional parameters]) | <p>Esta rutina ejecuta una instrucción SQL. La instrucción SQL puede parametrizarse (es decir, marcadores de posición en lugar de literales SQL). El módulo sqlite3 admite dos tipos de marcadores de posición: signos de interrogación y marcadores de posición con nombre (estilo con nombre). Por ejemplo: cursor.execute("insert into people values (?, ?)", (who, age))</p> |
| connection.execute(sql[,optional parameters]) | <p>Esta rutina es un atajo del método de ejecución anterior proporcionado por el objeto cursor y crea un objeto cursor intermedio llamando al método del cursor, luego llama al método de ejecución del cursor con los parámetros dados.</p> |
| cursor.executemany(sql, seq_of_parameters) | <p>Esta rutina ejecuta un comando SQL contra todas las secuencias de parámetros o asignaciones que se encuentran en la secuencia sql.</p> |

| | |
|---|---|
| connection.executemany(sql[, parameters]) | Esta rutina es un atajo que crea un objeto de cursor intermedio llamando al método cursor, luego llama al método <code>cursor.s executemany</code> con los parámetros dados. |
| cursor.executescript(sql_script) | Esta rutina ejecuta varias sentencias SQL a la vez proporcionadas en forma de script. Primero emite una declaración COMMIT, luego ejecuta el script SQL que obtiene como parámetro. Todas las sentencias SQL deben estar separadas por punto y coma (;). |
| connection.executescript (sql_script) | Esta rutina es un atajo que crea un objeto de cursor intermedio llamando al método del cursor, luego llama al método de ejecución del cursor con los parámetros dados. |
| connection.total_changes | Esta rutina devuelve el número total de filas de la base de datos que se han modificado, insertado o eliminado desde que se abrió la conexión a la base de datos. |
| connection.commit() | Este método confirma la transacción actual. Si no llama a este método, cualquier cosa que haya hecho desde la última llamada a <code>commit()</code> no es visible desde otras conexiones de base de datos. |
| connection.rollback () | Este método revierte cualquier cambio en la base de datos desde la última llamada a <code>commit()</code> . |
| connection.close () | Este método cierra la conexión a la base de datos. Tenga en cuenta que esto no llama automáticamente a <code>commit()</code> . Si simplemente cierra la conexión de su base de datos sin llamar a <code>commit()</code> primero, los cambios se perderán. |
| cursor.fetchone () | Este método obtiene la siguiente fila de un conjunto de resultados de consulta y devuelve una sola secuencia, o <code>None</code> cuando no hay más datos disponibles. |
| cursor.fetchmany([size= cursor.arraysize]) | Esta rutina recupera el siguiente conjunto de filas del resultado de una consulta y devuelve una lista. Se devuelve una lista vacía cuando no hay más filas disponibles. El método intenta obtener tantas filas como indique el parámetro de tamaño. |
| cursor.fetchall() | Esta rutina recupera todas las filas (restantes) del resultado de una consulta y devuelve una lista. Se devuelve una lista vacía cuando no hay filas disponibles. |

Conectarse a la base de datos

El siguiente código de Python nos muestra cómo conectarse a una base de datos existente. Si la base de datos no existe, se creará y finalmente se devolverá un objeto de base de datos.

```
import sqlite3
```

Conectarse a la base de datos

```
conn = sqlite3.connect( PATH_DATA + '\\test.db')
print("Abierta la conexión a la Base de Datos ");
```

Aquí, también podemos proporcionar el nombre de la base de datos como el nombre especial :memory: para crear una base de datos en la RAM.

Crear una tabla (CREATE TABLE)

El siguiente programa de Python se utilizará para crear una tabla en la base de datos test.db creada anteriormente.

Crear una tabla

```
conn.execute('''CREATE TABLE COMPANY
              (ID INT PRIMARY KEY     NOT NULL,
              NAME           TEXT      NOT NULL,
              AGE            INT       NOT NULL,
              ADDRESS        CHAR(50),
              SALARY         REAL);''')
print("Tabla creada...");
```

Tipos de datos.

SQLite sólo soporta los siguientes tipos de datos para definir columnas:

- NULL, valor es un NULL.
- INTEGER, entero con signo, almacenado automáticamente en 1, 2, 3, 4, 6 u 8 bytes dependiendo de la magnitud del valor.
- REAL, números con coma flotante, almacenados en 8 bytes en formato IEEE.
- TEXT, texto en codificación UTF-8, UTF-16BE o UTF-16LE.
- BLOB, datos binarios almacenados exactamente con el tamaño de entrada.

No hay un tipo de datos específico para almacenar fechas. Una fecha podrá almacenarse en formato de texto, o como un número que represente los días julianos de la fecha.

Insertar registros (INSERT INTO)

El siguiente programa de Python muestra cómo crear registros en la tabla COMPANY creada en el ejemplo anterior.

Insertar registros (INSERT INTO)

```
conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)\
VALUES (1, 'Paul', 32, 'California', 20000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)\
VALUES (2, 'Allen', 25, 'Texas', 15000.00 )");
```

```
conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)\nVALUES (3, 'Teddy', 23, 'Norway', 20000.00 )");\n\nconn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)\nVALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 )");\n\nconn.commit() # Este método confirma la transacción actual\n\nprint("Registros creados...");
```

Mostrar registros (SELECT)

Una vez creadas las tablas e insertados registros, podemos realizar consultas para recuperar todos los registros o una parte de los datos. Para ello debemos ejecutar consultas con el comando SELECT. Este se ejecuta con el método `execute()` de un cursor y los resultados se pueden recuperar con varios métodos del objeto cursor:

- `fetchall()`: recupera todos los registros generados por la consulta.
- `fetchmany(n)`: recupera n registros generados por la consulta.
- `fetchone()`: se puede ir invocando sucesivamente para recuperar los registros generados por la consulta uno a uno.

Los métodos `fetchmany()` y `fetchone()` retornan registros en forma de tupla, con los valores en cada columna especificada en la consulta. El método `fetchall()` retorna una lista de registros.

El siguiente programa de Python muestra como buscar y mostrar registros de la tabla COMPANY creada en el ejemplo anterior.

```
# Muestra todos los registros de la tabla COMPANY\n\ncursor = conn.execute('SELECT * FROM COMPANY')\n\n# fetchall() recupera las filas, retorna una lista\nrows = cursor.fetchall()\n\nfor row in rows:\n    print(row)\n\n(1, 'Paul', 32, 'California', 20000.0)\n(2, 'Allen', 25, 'Texas', 15000.0)\n(3, 'Teddy', 23, 'Norway', 20000.0)\n(4, 'Mark', 25, 'Rich-Mond ', 65000.0)
```

```
# Muestra todos los registros, únicamente las columnas name,  
address y salary
```

```
cursor = conn.execute("SELECT name, address, salary from  
COMPANY")  
print("-----")  
for row in cursor:  
    print("NAME = ", row[1])  
    print("ADDRESS = ", row[2])  
    print("SALARY = ", row[3])  
    print("-----")
```

```
-----  
NAME =  Paul  
ADDRESS =  California  
SALARY =  20000.0  
-----
```

```
NAME =  Allen  
ADDRESS =  Texas  
SALARY =  15000.0  
-----
```

```
NAME =  Teddy  
ADDRESS =  Norway  
SALARY =  20000.0  
-----
```

```
NAME =  Mark  
ADDRESS =  Rich-Mond  
SALARY =  65000.0  
-----
```

```
# Muestra los registros donde dirección sea igual a California
```

```
cursor = conn.execute("SELECT name, address, salary  
                        from COMPANY WHERE address = 'California'")  
print("-----")  
for row in cursor:  
    print("NAME = ", row[0])  
    print("ADDRESS = ", row[1])  
    print("SALARY = ", row[2])  
    print("-----")
```

```
-----  
NAME =  Paul  
ADDRESS =  California  
SALARY =  20000.0  
-----
```

```
# Muestra los registros donde el salario sea menor o igual a 20000
```

```
cursor = conn.execute("SELECT name, address, salary
                        from COMPANY WHERE salary <= 20000")
print("-----")
for row in cursor:
    print("NAME = ", row[0])
    print("ADDRESS = ", row[1])
    print("SALARY = ", row[2])
    print("-----")
```

```
-----
NAME = Paul
ADDRESS = California
SALARY = 20000.0
-----
```

```
NAME = Allen
ADDRESS = Texas
SALARY = 15000.0
-----
```

```
NAME = Teddy
ADDRESS = Norway
SALARY = 20000.0
-----
```

La cláusula **LIMIT** se usa para limitar la cantidad de datos retornados por la sentencia **SELECT**.

```
cursor = conn.execute("SELECT * from COMPANY LIMIT 2")
print("-----")
for row in cursor:
    print("NAME = ", row[0])
    print("ADDRESS = ", row[1])
    print("SALARY = ", row[2])
    print("-----")
```

La cláusula **ORDER BY** se utiliza para ordenar los datos en orden ascendente o descendente, según una o más columnas.

Sintaxis:

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

Mostrar los registros de **COMPANY** ordenados por Salario

```
cursor = conn.execute('SELECT * FROM COMPANY ORDER BY SALARY
ASC')
```

Nivel III: Aplicaciones Web

```
rows = cursor.fetchall() # recupera todas las filas del
resultado de una consulta y devuelve una lista
for row in rows:
    print(row)

(2, 'Allen', 25, 'Texas', 15000.0)
(1, 'Paul', 32, 'California', 20000.0)
(3, 'Teddy', 23, 'Norway', 20000.0)
(4, 'Mark', 25, 'Rich-Mond ', 65000.0)
```

Mostrar los registros de COMPANY ordenados por Nombre y Salario

```
cursor = conn.execute('SELECT * FROM COMPANY ORDER BY NAME,
SALARY ASC')
rows = cursor.fetchall() # recupera todas las filas del
resultado de una consulta y devuelve una lista
for row in rows:
    print(row)

(2, 'Allen', 25, 'Texas', 15000.0)
(4, 'Mark', 25, 'Rich-Mond ', 65000.0)
(1, 'Paul', 32, 'California', 20000.0)
(3, 'Teddy', 23, 'Norway', 20000.0)
```

La cláusula **GROUP BY** se utiliza en colaboración con la instrucción SELECT para organizar datos idénticos en grupos.

Sintaxis:

```
SELECT column-list
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2....columnN
```

Muestra la suma de los salarios agrupados por nombre

```
cursor = conn.execute('SELECT NAME, SUM(SALARY) FROM COMPANY
GROUP BY NAME') # Todos los registros ordenados por Nombre y
Salario
rows = cursor.fetchall()
for row in rows:
    print(row)

('Allen', 15000.0)
('Mark', 65000.0)
('Paul', 20000.0)
('Peter', 20000.0)
('Teddy', 20000.0)
```

Muestra la cantidad personas por Address

```
cursor = conn.execute('SELECT ADDRESS, COUNT(NAME) FROM COMPANY
GROUP BY ADDRESS ORDER BY NAME') # Todos los registros
ordenados por Nombre y Salario
rows = cursor.fetchall()
for row in rows:
    print(row)

('Texas', 1)
('Rich-Mond ', 1)
('California', 3)
('Norway', 1)
```

La cláusula **HAVING** permite especificar condiciones para los resultados del GROUP BY. La cláusula WHERE coloca condiciones en las columnas seleccionadas, mientras que la cláusula HAVING coloca condiciones en los grupos creados por la cláusula GROUP BY.

Muestra las direcciones con más de 2 empleados

```
cursor = conn.execute('SELECT address, count(name) FROM COMPANY
GROUP BY address HAVING count(address) > 2 ')
rows = cursor.fetchall()
for row in rows:
    print(row)

('California', 3)
```

La palabra clave **DISTINCT** se usa junto con la instrucción SELECT para obtener solo los registros únicos.

Muestra los registros únicos en dirección

```
cursor = conn.execute('SELECT DISTINCT address FROM COMPANY')
rows = cursor.fetchall()
for row in rows:
    print(row)

('California',)
('Texas',)
('Norway',)
('Rich-Mond ',)
```

Personalizar los resultados de una Consulta

Como hemos visto, al realizar una consulta los registros son devueltos en una tupla. Nosotros podemos modificar ese formato asignando al atributo `row_factory` del cursor una función de transformación.

Veamos un ejemplo utilizando la función predefinida `sqlite3.Row`:

Muestra todos los registros de la tabla COMPANY

```
conn.row_factory = sqlite3.Row
cursor = conn.execute('SELECT * FROM COMPANY')
rows = cursor.fetchall()
for row in rows:
    print(f'Nombre {row["name"]}, Dirección: {row["address"]},
    Salario {row["salary"]}')
```

La función `sqlite.Row()` devuelve objetos de tipo `sqlite3.Row` que permiten indexar los campos por su nombre. Es decir, podemos acceder a los valores utilizando el nombre de las columnas.

Muestra todos los registros de la tabla COMPANY

```
cursor = conn.execute('SELECT * FROM COMPANY')
cursor.row_factory = lambda cur, reg: 'Nombre {} con Salario
{}'.format(reg[1], reg[4])
rows = cursor.fetchall()
for row in rows:
    print(row)
```

Mostrar las tablas de una BBDD

Para mostrar todas las tablas en una base de datos SQLite3, debemos consultar la tabla **sqlite_master** y luego usar **fetchall()** para obtener los resultados de la sentencia SELECT. El **sqlite_master** es la tabla maestra en SQLite3 que almacena todas las tablas.

Conectarse a la base de datos

```
conn = sqlite3.connect( PATH_DATA + '\\test.db') # Si no
existe, creará la base de datos.
print("Abierta la conexión a la Base de Datos ");
```

Mostrar todas las tablas

```
cursorObj = conn.cursor()
cursorObj.execute('SELECT name from sqlite_master where type=
"table"')
print(cursorObj.fetchall())
```

Actualizar registro (UPDATE)

El siguiente código de Python muestra cómo usar la instrucción UPDATE para actualizar cualquier registro y luego buscar y mostrar los registros actualizados de la tabla COMPANY.

```
# Actualiza el salario de los empleados con ID = 1
conn.execute("UPDATE COMPANY set SALARY = 25000.00
             where ID = 1")
conn.commit()
```



```
print("Total de registros actualizados: ", conn.total_changes)
```

```
cursor = conn.execute('SELECT * FROM COMPANY')
rows = cursor.fetchall()
for row in rows:
    print(row)
```

Total de registros actualizados: 1

```
(1, 'Paul', 32, 'California', 25000.0)
(2, 'Allen', 25, 'Texas', 15000.0)
(3, 'Teddy', 23, 'Norway', 20000.0)
(4, 'Mark', 25, 'Rich-Mond ', 65000.0)
```

Eliminar registros (DELETE)

El siguiente código de Python muestra cómo usar la instrucción DELETE para eliminar cualquier registro y luego buscar y mostrar los registros restantes de la tabla COMPANY.

```
# Elimina el empleado con ID = 2
```

```
conn.execute("DELETE from COMPANY where ID = 2;")
conn.commit()
print("Total registros eliminados: ", conn.total_changes)
```

```
cursor = conn.execute('SELECT * FROM COMPANY')
rows = cursor.fetchall() # recupera todas las filas del
resultado de una consulta y devuelve una lista
for row in rows:
    print(row)
```

Total registros eliminados: 1

```
(1, 'Paul', 32, 'California', 25000.0)
(3, 'Teddy', 23, 'Norway', 20000.0)
(4, 'Mark', 25, 'Rich-Mond ', 65000.0)
```

Gestión de transacciones

Debemos invocar el método `commit()`, para confirmar los cambios en la BBDD, tanto cuando un INSERT, UPDATE o DELETE. Esto sucede porque las operaciones que se realizan a través de una conexión aplican automáticamente a las transacciones.

Una transacción determina un contexto de cambios sobre una base de datos. Las actualizaciones de registros se realizan sobre este contexto, y no sobre las tablas físicas. Cuando la transacción se confirma los cambios se trasladan a las tablas físicas y se inicia una

nueva transacción. También podemos desechar los cambios realizados sobre una transacción con **rollback()**:

```
with obtenerConexion() as conexion:
    try:
        cursor = conexion.cursor()
        ## Ejecutamos operaciones de actualización ... ##
        conexion.commit() # confirmamos las operaciones y se
        inicia otra transacción
    except Exception:
        conexion.rollback() # se cancelan todas las operaciones
    finally:
        cursor.close()
```

Es recomendable, realizar las operaciones de actualización dentro de un bloque **try/except**. Si no ocurre un error se confirman las operaciones con **commit()**, si ocurre un error se podemos anular todas las operaciones con **rollback()**.

Es importante destacar, que el método **close()** de una conexión, antes de cerrarla, confirma todos los cambios de la transacción actual.

Eliminar una Tabla de una BBDD (DROP)

DROP se utiliza para eliminar una definición de tabla y todos los datos, índices, constraints, restricciones y especificaciones de permisos asociados para esa tabla.

Sintaxis

```
DROP TABLE table_name;
```

Ejemplo:

```
conn.execute("DROP TABLE COMPANY;")
```

Con la sentencia **DROP TABLE**, retorna un error si la tabla no existe. Para evitar podemos utilizar:

```
conn.execute("DROP TABLE if exists COMPANY;")
```

DB Browser para SQLite (DB4S)

DB Browser for SQLite (DB4S) es una herramienta de código abierto, visual y de alta calidad para crear, diseñar y editar archivos de base de datos compatibles con SQLite. Utiliza una interfaz familiar similar a una hoja de cálculo y no es necesario aprender los comandos SQL.

La herramientas nos permite:

- Crear y compacta archivos de base de datos
- Crear, definir, modificar y eliminar tablas
- Crear, definir y eliminar índices

Nivel III: Aplicaciones Web

- Examinar, editar, agregar y eliminar registros
- Buscar registros
- Importar y exportar registros como texto
- Importar y exportar tablas desde/hacia archivos CSV
- Importar y exportar bases de datos desde/hacia archivos SQL
- Emitir consultas SQL e inspeccionar los resultados
- Examinar el log de todos los comandos SQL emitidos por la aplicación.
- Trazar gráficos simples basados en datos de consulta o tabla

Instalación

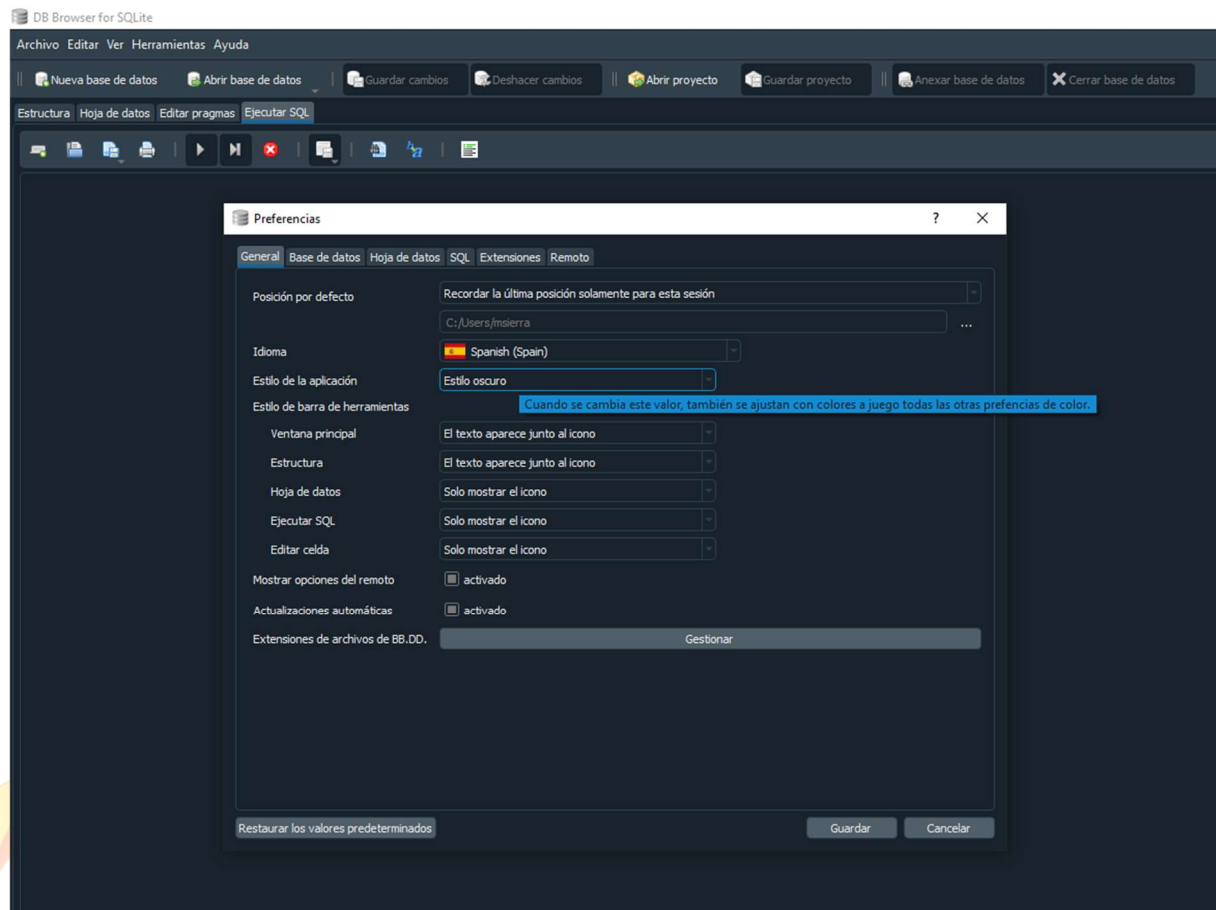
Opción Standard installer

1. Obtener la última versión del instalador en [Downloads - DB Browser for SQLite \(sqlitebrowser.org\)](https://sqlitebrowser.org/downloads). Descargar el **DB Browser for SQLite – Standard installer for XX – bit Windows**.
2. Seguir las instrucciones del instalador
3. Ejecutar la aplicación **DB Browser (SQLite)**

Opción .zip no installer

1. Desde la página [Downloads - DB Browser for SQLite \(sqlitebrowser.org\)](https://sqlitebrowser.org/downloads) nos podemos descargar el **DB Browser for SQLite – .zip (no installer) for XX – bit Windows**.
2. Una vez descargamos el fichero lo descomprimos en un directorio. Por ejemplo: **C:\Program Files (x86)**
3. Este crear el directorio **DB Browser for SQLite**
4. Ejecutamos el fichero **DB Browser for SQLite.exe**

Una vez instalada la aplicación, la ejecutamos y podemos cambiar la configuración de la misma desde el menú **Editar – Preferencias – Estilo de la aplicación: Estilo Oscuro**

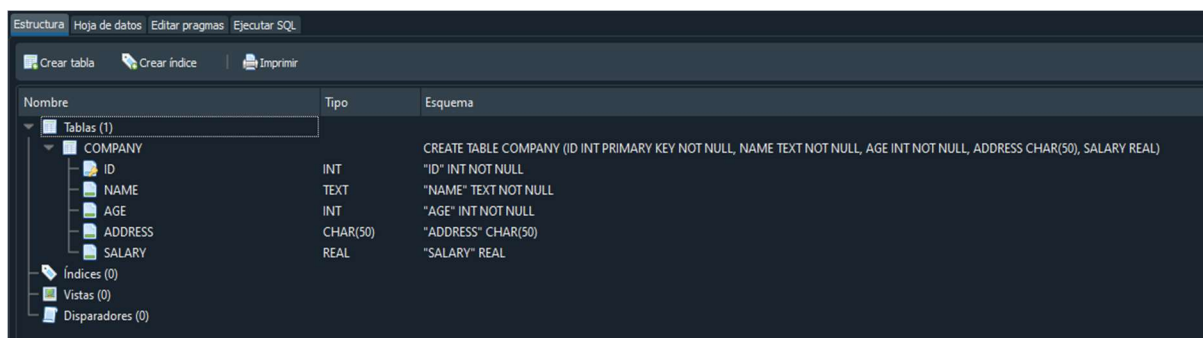


Desde el DB Browser podemos ver las base de datos que hemos creado, visualizar los datos, insertar registros, así como otras operaciones con BBDD SQLite.

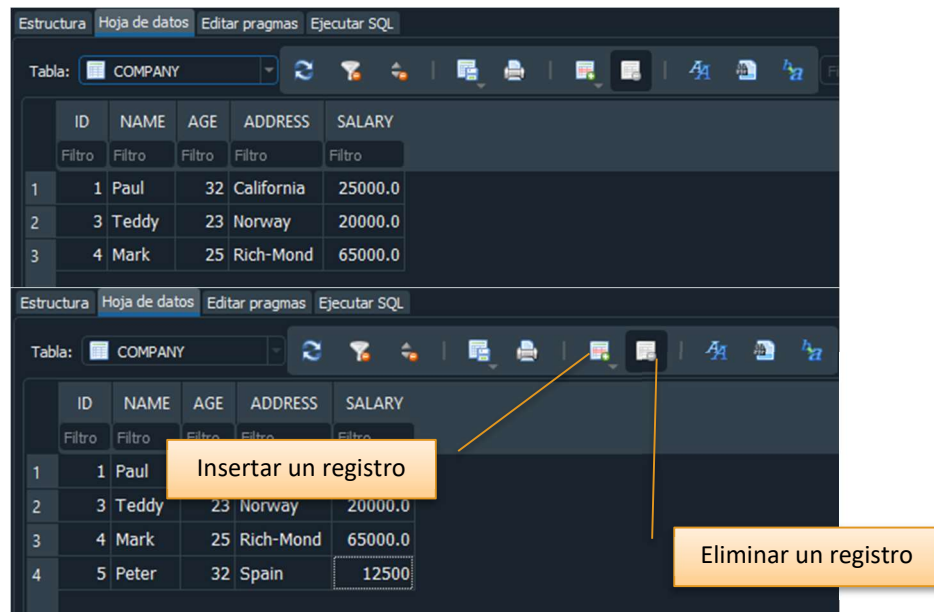
Vamos a ver nuestra BBDD **test.db**:

1. Desde el menú **Archivo**, seleccionamos la opción **Abrir base de datos...**, ubicamos la ruta donde creamos nuestra base de datos y seleccionamos el fichero **test.db**.
2. El programa **DB Browser (SQLite)** nos muestra la estructura de nuestra BBDD: Tablas, índices, vistas, etc.

Estructura



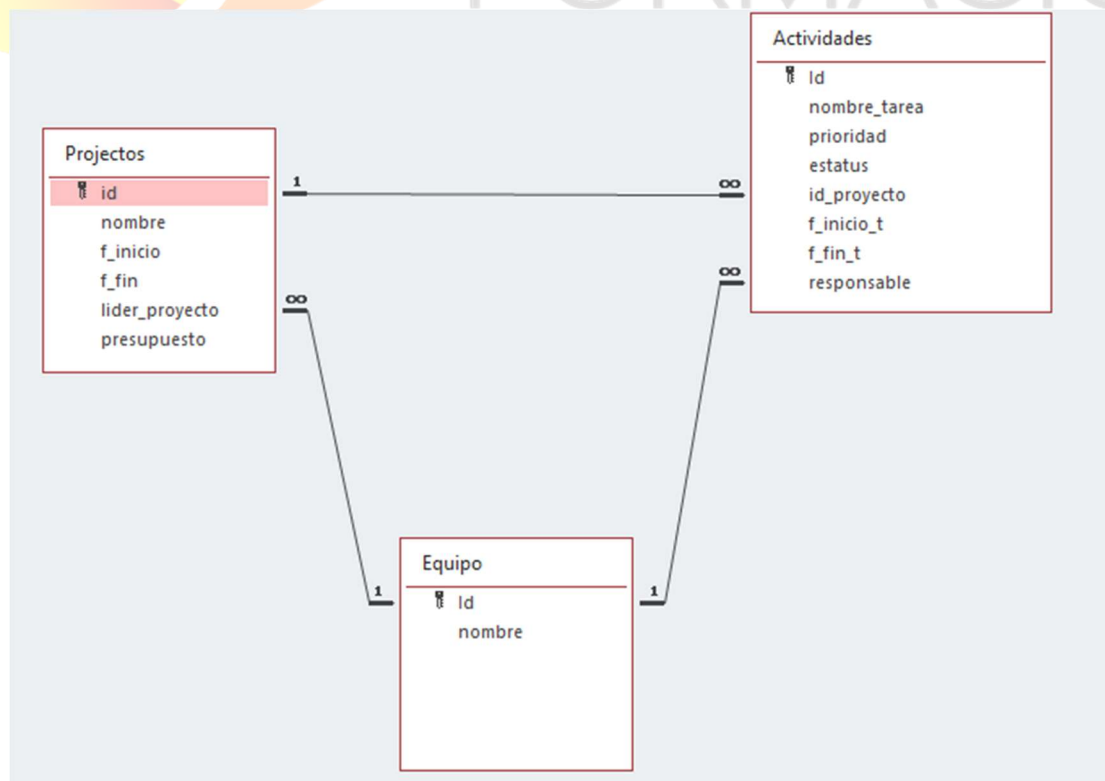
3. En la pestaña **Hoja de datos**, podemos ver los registros que contiene nuestra tabla COMPANY. Desde aquí podemos modificar, insertar y/o eliminar registros.



4. En la pestaña **Ejecutar SQL**, podemos ver los registros que contiene nuestra tabla COMPANY. Desde aquí podemos modificar, insertar y/o eliminar registros.

Ejemplo de una Base de Datos relacional de Gestión de Proyectos

Utilizaremos como ejemplo una simple BBDD relacional de Gestión de Proyectos, donde los proyectos pueden tener una o más actividades asociadas, a su vez cada actividad tiene un responsable asignado y los proyectos tiene un líder de proyectos asignado. Veamos el modelo:



Nivel III: Aplicaciones Web

El modelo consta de tres tablas: Proyectos, Actividades y Equipo, esta ultima contiene todas las personas que trabajan en los diferentes proyectos, como responsables de una actividad o como líder de proyectos.

Tabla Proyectos

| id | nombre | f_inicio | f_fin | lider_proyecto | presupuesto |
|------|------------------------|------------|------------|----------------|-------------|
| 100 | New Computers | 01/01/2020 | 31/12/2020 | 444444 | 98500 |
| 101 | PC Refresh | 01/01/2020 | 31/12/2020 | 444444 | 250000 |
| 102 | Project 2019 | 01/01/2019 | 01/03/2020 | 444444 | 40250 |
| 103 | Office Moves | 01/01/2020 | 31/12/2020 | 444444 | 10800 |
| 104 | Video Conference | 01/06/2020 | 01/12/2020 | 444444 | 42000 |
| 105 | Projects Other CeCo | 01/01/2020 | 31/12/2020 | 555555 | 25000 |
| 106 | Phone & Cordless Phone | 01/01/2020 | 31/12/2020 | 666666 | 1500 |
| 107 | Easy Color | 01/03/2020 | 31/05/2020 | 555555 | 38500 |
| 108 | Infraestructura | 01/01/2020 | 31/12/2020 | 666666 | 105000 |
| 109 | Data Center | 01/03/2020 | 31/08/2020 | 444444 | 110000 |
| 110 | Mac Refresh | 01/03/2020 | 31/05/2020 | 555555 | 4500 |
| 9999 | Expenses | 01/01/2020 | 31/12/2020 | 444444 | 850000 |

Tabla Actividades

| Id | nombre_tarea | prioridad | estatus | id_proyecto | f_inicio_t | f_fin_t | responsable |
|-------|---|-----------|---------|-------------|------------|------------|-------------|
| 10001 | Realizar inventarios de los equipo disponibles | 1 | 1 | 100 | 08/01/2020 | 10/01/2020 | 444444 |
| 10002 | Obtener listado de nuevos ingresos | 1 | 1 | 100 | 01/01/2020 | 10/01/2020 | 555555 |
| 10101 | Realizar inventarios de los equipo con 3 o mas años | 1 | 1 | 100 | 08/01/2020 | 10/01/2020 | 444444 |
| 10102 | Definir la cantidad de equipos a comprar por Modelo | 1 | 1 | 100 | 08/01/2020 | 10/01/2020 | 444444 |

Tabla Equipo

| Id | nombre |
|--------|------------------|
| 444444 | Maria Sierra |
| 555555 | Pedro Perez |
| 666666 | Manuel Rodriguez |

BBDD Relacionales - Gestión de Proyectos

Importar librerías

```
import sqlite3
from sqlite3 import Error

from datetime import datetime
```

Funciones

```
def create_connection(db_file):
    """ Crea una conexión a una Base de Datos SQLite Local
        :parametros db_file: base de datos
        :return: el objeto de la conexión a la BBDD o None
    """
```

Nivel III: Aplicaciones Web

```
conn = None
try:
    conn = sqlite3.connect(db_file)
    print(sqlite3.version)
except Error as e:
    print(e)
    if conn:
        conn.close()
return conn

def create_connection_memory():
    """ Crea una conexión a una Base de Datos SQLite en Memoria"""
    conn = None;
    try:
        conn = sqlite3.connect(':memory:')
        print(sqlite3.version)
    except Error as e:
        print(e)
        if conn:
            conn.close()
    return conn

def create_table(conn, create_table_sql):
    """ Crear una Table en una BBDD SQLite
    :parametro conn: Conexión a la base de datos SQLite
    :parametro create_table_sql: La instrucción CREATE TABLE
    :return:
    """
    try:
        c = conn.cursor()
        c.execute(create_table_sql)
    except Error as e:
        print(e)

def create_project(conn, project):
    """
    Inserta un nuevo poryecto en la Tabla Proyectos
    :param conn: Conexión a la base de datos SQLite
    :param project:
    :return: project id
    """
    sql = ''' INSERT INTO proyectos(nombre, f_inicio, f_fin,
        lider_proyecto, presupuesto)
        VALUES(?,?,?,?,:) '''

    cur = conn.cursor()
    cur.execute(sql, project)
    conn.commit()
    return cur.lastrowid
```



```
def create_task(conn, task):  
    """  
    Inserta una actividad o tarea en la Tabla actividades  
    :param conn: Conexión a la base de datos SQLite  
    :param task:  
    :return:  
    """  
  
    sql = ''' INSERT INTO actividades(nombre_tarea,  
                                     prioridad, estatus, id_proyecto ,f_inicio_t, f_fin_t)  
               VALUES(?,?,?,?,?,?) '''  
    cur = conn.cursor()  
    cur.execute(sql, task)  
    conn.commit()  
  
    return cur.lastrowid  
  
def update_task(conn, task):  
    """  
    Actualiza prioridad, fecha de inicio, fecha fin de una actividad  
    :param conn: Conexión a la base de datos SQLite  
    :param task:  
    :return: project id  
    """  
    sql = ''' UPDATE actividades  
               SET prioridad = ? ,  
                 f_inicio_t = ? ,  
                 f_fin_t = ?  
               WHERE id = ?'''  
    cur = conn.cursor()  
    cur.execute(sql, task)  
    conn.commit()  
  
def select_all_tasks(conn):  
    """  
    Muestra todas las actividades de la tabla actividades  
    :param conn: Conexión a la base de datos SQLite  
    :return:  
    """  
  
    cur = conn.cursor()  
    cur.execute("SELECT * FROM actividades")  
  
    rows = cur.fetchall()  
  
    for row in rows:  
        print(row)
```


Nivel III: Aplicaciones Web

```
def select_task_by_priority(conn, priority):
    """
    Mostrar todas las actividades por prioridad
    :param conn: Conexión a la base de datos SQLite
    :param priority:
    :return:
    """
    cur = conn.cursor()
    cur.execute("SELECT * FROM actividades WHERE prioridad=?",
    (priority,))

    rows = cur.fetchall()

    for row in rows:
        print(row)

def delete_task(conn, id):
    """
    Borra una actividad por id
    :param conn: Conexión a la base de datos SQLite
    :param id: id de la tarea
    :return:
    """
    sql = 'DELETE FROM actividades WHERE id=?'
    cur = conn.cursor()
    cur.execute(sql, (id,))
    conn.commit()

def delete_all_tasks(conn):
    """
    Borra todas las actividades
    :param conn: Conexión a la base de datos SQLite
    :return:
    """
    sql = 'DELETE FROM actividades'
    cur = conn.cursor()
    cur.execute(sql)
    conn.commit()

if __name__ == '__main__':

    database =
        r'C:\Users\msierra\Desktop\SQLDatabase\ProyectosIT4.db'

    # crear la conexión a la base de datos local
    conn = create_connection(database)

    # crear la conexión a la base de datos en memoria
    # create_connection_memory()
```

```
# crear las tablas
```

```
sql_create_projects_table = """ CREATE TABLE IF NOT EXISTS
                                proyectos (
                                id integer PRIMARY KEY,
                                nombre text NOT NULL,
                                f_inicio date,
                                f_fin date,
                                lider_proyecto text,
                                presupuesto float
                                ); """
```

```
sql_create_tasks_table = """CREATE TABLE IF NOT EXISTS
                                actividades (
                                id integer PRIMARY KEY,
                                nombre_tarea text NOT NULL,
                                prioridad integer,
                                estatus texto NOT NULL,
                                id_proyecto integer NOT NULL,
                                f_inicio_t text NOT NULL,
                                f_fin_t text NOT NULL,
                                FOREIGN KEY (id_proyecto)
                                REFERENCES proyectos (id)
                                ); """
```

```
if conn is not None:
    # create projects table
    create_table(conn, sql_create_projects_table)

    # create tasks table
    create_table(conn, sql_create_tasks_table)
else:
    print("Error! La conexión a la BBDD no ha sido creada...")
```

```
# create a new project
```

```
project = ( 'New Computers', datetime.date(2020,1,1),
            datetime.date(2020,12,31),
            "Maria Sierra", 125000.00);
project_id = create_project(conn, project)
```

```
# tasks
```

```
task_1 = ('Realizar inventarios de los equipo disponibles', 1,
          'En proceso', project_id, '2015-01-01', '2015-01-02')
task_2 = ('Obtener listado de nuevos ingresos', 1, 'En proceso',
          project_id, '2015-01-03', '2015-01-05')
```

```
# create tasks
```

Nivel III: Aplicaciones Web

```
create_task(conn, task_1)
create_task(conn, task_2)

# Mostrar todas las actividades
print("Mostrar todas las tareas")
select_all_tasks(conn)

# Mostrar todas las actividades x prioridad
print("Consultar las tareas por prioridad")
select_task_by_priority(conn, 1)

# Eliminar actividades
delete_task(conn, 2)

# Eliminar todas las actividades
delete_all_tasks(conn)

# Cerrar la conexión
conn.close()
```

Conexiones ODBC

Open DataBase Connectivity (ODBC) es una tecnología estandarizada que sirve de capa intermedia entre las aplicaciones y un sistema de gestión de base de datos. Windows soporta esta tecnología, al igual que muchos gestores de base de datos. Cuando instalamos un gestor de base de datos (SQL Server, Oracle, Access, Excel,) si es compatible, podemos tener acceso a través de ODBC.

Para usar ODBC con Python, primero debemos instalar el módulo conector **pyodbc**. Si necesitamos conocer las base de datos soportadas por nuestro sistema podemos consultarlos con la función `pyodbc.drivers()`.

Controlador Python para SQL: pyodbc

Paso 1: Configuración del entorno de desarrollo para Python pyodbc en Windows.

Conexión a SQL Database mediante Python - pyodbc en Windows:

1. Instalar Microsoft ODBC Driver for SQL Server en Windows
2. Instalar pyodbc con el administrador de paquetes de pip:

```
pip install pyodbc
```

Paso 2: Creación de una instancia de SQL Database para el desarrollo de Python pyodbc

Paso 3: Conexión a SQL con pyodbc

```
import pyodbc

server = 'tcp:myserver.database.windows.net'
```

Nivel III: Aplicaciones Web

```
database = 'mydb'
username = 'myusername'
password = 'mypassword'
cnxn = pyodbc.connect('DRIVER={ODBC Driver 17 for SQL
Server};SERVER='+server+';DATABASE='+database+';UID='+username+
';PWD='+ password)

cursor = cnxn.cursor()
```

Ejemplo de conexión a una base de datos via ODBC.

```
import pyodbc
cnxn = pyodbc.connect('DRIVER={Devart ODBC Driver for SQLite};
Direct=True;Database=mydatabase')
```

Ejemplo de conexión a una base de datos Access.

```
import pyodbc

conn = pyodbc.connect(r'Driver={Microsoft Access Driver (*.mdb,
*.accdb)}; DBQ=path where you stored the Access
file\file name.accdb;')
```

Ejemplo de Conexión a Excel

```
conn_str = (
    r'Driver={Microsoft Excel Driver (*.xls,*.xlsx,*.xlsm, *.xlsb)};'
    r'DBQ=file name.xls;'
    r'ReadOnly=0'
)
cnxn = pyodbc.connect(conn_str, autocommit=True)
```

Otras Conexiones

Controlador Python para SQL: pymssql

Paso 1: Configuración del entorno de desarrollo para el desarrollo de Python pymssql

1. Descargar el del módulo pymssql
2. Instalar el módulo pymssql.

```
pip install pymssql-2.1.1-cp27-none-win_amd64.whl
```

Paso 2: Creación de una instancia de SQL Database para el desarrollo de Python pymssql

Paso 3: Prueba de concepto de la conexión a SQL con pymssql

```
import pymssql

conn = pymssql.connect(server='yourserver.database.windows.net',
    user='yourusername@yourserver', password='yourpassword',
```

```
database='AdventureWorks')
```

Controlador Python para Bases de datos de Oracle DB

```
import cx_Oracle

with cx_Oracle.connect('usuario/contraseña@127.0.0.1/nombrebd')
as conexión:
    print(conexion.version)
```

Controlador Python para Bases de datos MySQL

```
import MySQLdb

conexion = MySQLdb.connect( host="localhost", db="nombrebd", \
    user="usuario", passwd="contraseña" )
cursor = conexion.cursor()
```

SQLObject

Un modelo **ORM (Object Relational Model)** permite trabajar con bases de datos (tablas y registros) en el código de Python a través de clases e instancias. Todas las operaciones sobre la base de datos se podrán implementar a través de las clases y objetos del modelo ORM

SQLObject es un modelo **Objeto-Relacional (ORM)** muy popular que proporciona una interfaz donde las tablas de bases de datos se corresponden con clases, los registros con instancias y las columnas con atributos. Para utilizar SQLObject debemos descargar su módulo:

```
pip install -U SQLObject
```

SQLObject se puede usar con varios módulos conectores de bases de datos. Actualmente SQLObject soporta las bases de datos:

- MySQL (vía mysqlclient, MySQL Connector, oursql, PyMySQL, PyODBC y PyPyODBC)
- PostgreSQL (vía psycopg2, PyGreSQL, py-postgresql y pg8000)
- SQLite
- Firebird (vía fdb orkinterbasdb)
- MAX DB o SAP DB (vía sapdb)
- Sybase (vía Sybase), MSSQL Server (vía pymssql, PyODBC y PyPyODBC).

Podemos instalar los módulos conectores por separado, pero también es posible instalarlos junto con **SQLObject**. Por ejemplo:

```
pip install SQLObject[mysql]
```

Usando SQLObject

Con el modelo SQLObject, debemos **crear una clase para cada tabla de la base de datos**. La definición de la clase debe incluir atributos que se correspondan con los campos de la tabla.

Nivel III: Aplicaciones Web

SQLObject se encargará del resto, generando el código necesario para acceder a la base de datos, actualizar registros y realizar consultas.

Para utilizar SQLObject debemos comenzar importando el módulo `sqlobject` y sus clases.

```
from sqlobject import *
```

Y a continuación, declarando una conexión. Las conexiones se declaran mediante una URI (Uniform Resource Identifier) con el siguiente formato:

```
scheme://[user[:password]@]host[:port]/database[?parameters]
```

El esquema debe ser uno de: `sqlite`, `mysql`, `postgres`, `firebird`, `interbase`, `maxdb`, `sapdb`, `mssql`, `sybase`. Ejemplos:

```
mysql://user:password@host/database
mysql://host/database?debug=1
postgres://user@host/database?debug=&cache=
postgres:///full/path/to/socket/database
postgres://host:5432/database
sqlite:///full/path/to/database
sqlite:/C:/full/path/to/database
sqlite://:memory:
```

En parámetros son: `debug` (por defecto `False`), `debugOutput` (por defecto `False`), `cache` (por defecto `True`), `autoCommit` (por defecto `True`), `debugThreading` (por defecto `False`), `logger` (por defecto `None`), `loglevel` (por defecto `None`), `schema` (por defecto `None`).

Para asignar el valor `True` en estos parámetros, en la URI, debemos escribir `yes`, `true`, `on` o `1`; y para asignar el valor `False` debemos dejar un string vacío, o `no`, `false`, `off` o `0`.

La URI de conexión servirá de argumento para el método `sqlobject.connectionForURI()`, el cual retorna un objeto de conexión:

```
from sqlobject import *

cadenaURI = 'sqlite://:memory:'
conexion = connectionForURI(cadenaURI)
sqlhub.processConnection = conexion

import os
db_filename = os.path.abspath('data.db')
connection_string = 'sqlite:' + db_filename
connection = connectionForURI(connection_string)
sqlhub.processConnection = connection
```

El objeto `sqlobject.sqlhub` es de tipo `sqlobject.dbconnection.ConnectionHub`. Esta clase representa un centro de actividad para conexiones con la base de datos. En su

Nivel III: Aplicaciones Web

atributo `processConnection` establecemos el objeto de conexión. Todas las clases de `SQLObject` que mapearán las tablas de la base de datos usarán, por defecto, este objeto para conectarse con ella.

Declaración de las clases (tablas)

Para los ejemplos de uso de `SQLObject` crearemos una tabla `CLIENTE` con los campos `id`, `nombre` y `dirección`. Como hemos dicho, `SQLObject` hace corresponder las tablas de base de datos con clases. Por tanto crearemos una clase `Cliente` heredando de `SQLObject`:

```
class Cliente(SQLObject):
    nombre = StringCol(length=100, default='', alternateID=True)
    direccion = StringCol()
```

Al crear una clase `SQLObject` se hereda el atributo `id`, asociado a la clave primaria de la tabla, y se establece una correspondencia entre el nombre de la clase y una tabla con el mismo nombre. En la clase debemos declarar un atributo por cada columna que tenga la tabla.

La correspondencia entre los atributos y las columnas de la tabla la establecemos instanciando objetos `IntCol`, `StringCol`, `BoolCol`, `DateCol`, `FloatCol`, etc. Todas estas clases heredan de la superclase `Col`:

- `BLOBCol`: columna para datos binarios. Actualmente solo funciona con backends de MySQL, PostgreSQL y SQLite.
- `BoolCol`: creará una columna `BOOLEAN` en Postgres o `INT` en otras bases de datos. También convertirá los valores a "t"/"f" o 0/1 según el backend de la base de datos.
- `CurrencyCol`: Equivalente a `DecimalCol` (tamaño 10, precisión 2).
- `DateTimeCol`: una fecha y hora (normalmente devuelta como un objeto `datetime` o `mxDateTime`).
- `DateCol`: una fecha (normalmente devuelta como un objeto `datetime` o `mxDateTime`).
- `TimeCol`: una hora (normalmente devuelta como un objeto `datetime` o `mxDateTime`).
- `TimestampCol`: soporta el tipo MySQL `TIMESTAMP`.
- `DecimalCol`: Base-10, número preciso.
- `DecimalStringCol`: Similar a `DecimalCol` pero almacena datos como cadenas para solucionar problemas en algunos controladores y problema de afinidad de tipo en SQLite.
- `EnumCol`: Para asignar un rango posible de valores de cadena a una columna. En el parámetro `enumValues` debemos asignar una lista con los valores posibles.
- `SetCol`: Para bases de datos MySQL da soporte al tipo `SET`.
- `FloatCol`: Para columnas numérico con punto flotante de tamaño fijo.
- `IntCol`: Para columnas numéricas enteras de tamaño fijo.
- `JsonbCol`: Para bases de datos de Postgres, da soporte a columnas que contienen objetos en formato `Json`. En Python podemos serializar a formato `Json` cualquier objeto con la función `json.dumps()`.

Nivel III: Aplicaciones Web

- PickleCol: Hereda de BLOBCol. Permite almacenar objetos serializados a cadenas binarias.
- StringCol: Para columnas de texto. Se puede indicar la longitud máxima con el parámetro length. Con el parámetro varchar=False se almacenan los texto con longitud máxima fija (CHAR), sino se almacenan con longitud variable (VARCHAR).
- UnicodeCol: Subclase de StringCol, que codifica los valores con el formato UTF. El parámetro dbEncoding especifica la variante UTF, donde el valor por defecto es codificación "utf-32".
- UuidCol: Para bases de datos Postgres, soporta el tipo UUID.

En las funciones constructoras de estas subclases de Col podemos utilizar los siguientes parámetros:

- dbName: El nombre de la columna en la base de datos. Por defecto se asocia con el nombre fonético del atributo.
- Default: El valor por defecto para la columna cuando se crea un nuevo registro. Con columnas de tipo DateTimeCol se puede utilizar DateTimeCol.now para asignar la fecha actual.
- alternateID: Si es True, se crea una columna NOT NULL y UNIQUE. También genera en la clase un método byAtributo(valor) que permite hacer búsquedas de registros por el valor en el atributo.
- Unique: Si es True, se crea una columna UNIQUE.
- notNone: Si es True, se crea una columna NOT NULL.
- sqlType: Si SQLAlchemy va a crear las tablas, establece el tipo de la columna en la base de datos (como INT, BOOLEAN, etc.).
- validator - validator2: Se asigna a un objeto que contiene métodos to_python() y from_python(), para convertir y validar los valores.

El esquema mapeado por la definición de esta clase, para una tabla de SQLite, se corresponde con:

```
CREATE TABLE cliente (  
    id INTEGER PRIMARY KEY AUTOINCREMENT,  
    nombre VARCHAR(100) NOT NULL UNIQUE,  
    direccion TEXT,  
    telefono TEXT  
)
```

Como SQLite soporta campos autoincrementales se crea un id con la opción AUTOINCREMENT, con otras bases de datos esto puede cambiar. Al asignar alternateID=True, se crea el campo **nombre** con las restricciones NOT NULL y UNIQUE. Además, esta opción genera automáticamente, en la clase, un método llamado byNombre(valor), que permite recuperar un registro de cliente por su nombre:


```
registro = Cliente.byNombre('Juan')
```

Al heredar de SQLAlchemy se utiliza por defecto la conexión establecida en el objeto sqlalchemy, sino tendremos que asignar una conexión a través del método setConnection():

```
Cliente.setConnection( conexión )
```

Si la tabla CLIENTE ya existe en la base de datos no tenemos que hacer nada más. Pero si queremos crear la tabla basta con ejecutar:

```
if not Cliente.tableExists():  
    Cliente.createTable()
```

o abreviado

```
Cliente.createTable(ifNotExists=True)
```

Métodos de la clase SQLAlchemy

Métodos de SQLAlchemy

Descripción

| | |
|---|---|
| clearTable(clearJoinTables=True) | Vacía la tabla en la base de datos. |
| createTable(ifNotExists=False, createJoinTables=True, createIndexes=True, applyConstraints=True) | Crea la tabla en la base de datos. |
| delete(id) | Elimina un registro por id. |
| deleteBy(**kw) | Elimina registros por algún campo. |
| deleteMany(where) | Elimina registros por alguna condición. |
| dropTable(ifExists=False, dropJoinTables=True, cascade=False) | Elimina la tabla en la base de datos. |
| get(id, selectResults=None) | Recupera un registro por id. |
| select(clause=None, clauseTables=None, orderBy, limit=None, lazyColumns=False, reversed=False, distinct=False, join=None, forUpdate=False) | Recupera varios registros mediante una consulta SELECT. |
| selectBy(**kw) | Método alternativo a select(). |
| setConnection(value) | Asigna una conexión. |
| sqlrepr(value) | Retorna un valor entrecomillado. |
| tableExists() | Comprueba si la tabla física existe. |

Insertar registros

Una vez que tenemos configurada la clase del modelo podemos insertar registros a la base de datos. Para ello basta con instanciar la clase:

```
c1 = Cliente(nombre='Maria', direccion="Ronda de Nelle",  
telefono = "444444444")
```

```
<Cliente 2 nombre='Maria' direccion='Ronda de Nelle'  
telefono='444444444'>
```

Podemos comprobar que realmente se ha creado el registro en la tabla física, y no solamente un objeto en memoria, recuperándolo con el método `get(id)`:

```
c1 = Cliente.get(1)  
c1
```

```
<Cliente 1 nombre='Maria' direccion='Ronda de Nelle'  
telefono='444444444'>
```

SQLObject almacena de forma inmediata los registros, de forma que siempre establece una asociación de identidad entre un registro y una instancia de la clase. Podemos comprobarlo con el siguiente código:

```
c1 = Cliente.get(1)  
c2 = Cliente.get(1)
```

```
c1 is c2  
True
```

A través de los atributos del objeto recuperado podemos acceder a los valores del registro:

```
c1.nombre  
'Maria'
```

Y modificarlos:

```
c1.nombre = 'María'
```

Los nuevos datos serán enviados al registro físico de modo inmediato. Si queremos modificar varios datos en una única operación de UPDATE debemos usar el método `set()`:

```
## Modificar varias columnas
```

```
c3.set(nombre='Ana Lucia', direccion='Avenida A')  
c3  
<Cliente 2 nombre='Ana Lucia' direccion='Avenida A'  
telefono='44555554'>
```

Si vamos a realizar varias operaciones de actualización, puede que nos interese no enviarlas directamente a la base de datos de forma inmediata. Podemos deshabilitar este comportamiento de actualización automática asignando `True` al atributo `sqlmeta.lazyUpdate`:

```
Cliente.sqlmeta.lazyUpdate= True
```

Ahora los cambios en un cliente no provocarán comandos UPDATE:

Nivel III: Aplicaciones Web

```
maria = Cliente.get(1)
maria.nombre = 'MariCarmen'
```

Una vez realizados todos los cambios sobre un cliente podemos trasladarlos a la base de datos usando el método `sync()` o `syncUpdate()`:

```
if maria.sqlmeta.dirty: # busca los cambios pendientes
    maria.sync()
```

El método `sync()` provoca una consulta para releer todos los campos, mientras que el método `syncUpdate()` no provoca esta consulta. Con el atributo `sqlmeta.dirty` podemos saber si hay cambios pendientes de confirmar o no.

Eliminar registros

Para eliminar registros podemos utilizar el método `delete()` si necesitamos eliminar un registro por ID:

```
Cliente.delete(1)
```

Para eliminar registros por otros criterios podemos utilizar el método `deleteBy()`, que permite especificar valores de coincidencia de uno o varios campos. Por ejemplo, para eliminar los clientes cuyo nombre sea 'Ana':

```
Cliente.deleteBy(nombre='Ana')
```

Para eliminar por criterios más elaborados podemos usar el método `deleteMany()`. Este método utiliza como argumento un objeto de tipo Query, el cual establece criterios de filtro para generar la cláusula WHERE del comando SQL.

```
Cliente.deleteMany(Cliente.q.id > 1)
```

Configuración de metadatos.

La clase `SQLObject` incluye un atributo `sqlmeta` que permite configurar ciertos comportamientos de las operaciones con la base de datos. La siguiente tabla resume algunos de los atributos que proporciona este objeto:

| Atributos de <code>SQLObject.sqlmeta</code> | Descripción |
|---|---|
| table | Establece el nombre de la tabla física. Este nombre, por defecto, se corresponde con el asignado a la clase. |
| idName | El nombre de la columna de clave primaria en la tabla. Por defecto, este nombre es <code>id</code> . En la clase, la clave primaria siempre se asocia con el atributo <code>id</code> . |
| idType | Una función que moldea los valores para el ID. Por defecto, los IDs se moldean al tipo <code>int</code> . |

| | |
|---------------------|--|
| lazyUpdate | Por defecto es False. Si se asigna a True los cambios que se realizan en los atributos de un registro no se trasladan inmediatamente a la tabla física; es necesaria una invocación posterior de <code>instancia.syncUpdates()</code> o <code>instancia.sync()</code> . |
| defaultOrder | Establece la ordenación por defecto cuando se realizan consultas con <code>select()</code> . |
| cacheValues | Por defecto es True. Provoca que las instancias correspondientes con los registros se almacenen en una caché a medida que se recuperan desde la base de datos. Esto optimiza su recuperación posterior sin tener que consultar la base de datos ejecutando un comando SELECT. Para eliminar una instancia de la caché hay que invocar <code>instancia.expire()</code> . |
| columns | Retorna un diccionario con los nombres y tipos de las columnas. |
| columnList | Retorna una lista con la descripción de las columnas. |
| createSQL | Retorna el comando de creación de la tabla. |
| expired | Retorna True o False, para indicar si una instancia ha expirado en la caché. |

Si queremos establecer el valor de alguno de estos atributos, lo habitual es hacerlo en la declaración de la clase de la siguiente forma:

```
class Cliente(SQLObject):
    class sqlmeta:
        lazyUpdate = True
        cacheValues = False
        nombre = StringCol(length=100, default='',
                           alternateID=True)
        direccion = StringCol()
```

Consultas con SQLObject

Veamos cómo realizar consultas con el método `select()`, con este método podemos especificar condiciones sobre uno o varios atributos.

Mostrar todos los registros de la tabla clientes

```
resultado = Cliente.select()
list(resultado)

[<Cliente 2 nombre='Ana Lucia' direccion='Avenida A' telefono='44555554'>,
 <Cliente 3 nombre='Alejandro' direccion='Ronda' telefono='44555554'>,
 <Cliente 4 nombre='Ana Rosa' direccion='Ronda' telefono='44555554'>,
 <Cliente 5 nombre='Manuel' direccion='Ronda' telefono='44555554'>,
 <Cliente 6 nombre='Pedro' direccion='Avenida' telefono='55555555'>,
 <Cliente 7 nombre='Juan' direccion='Calle 37' telefono='45858595'>,
 <Cliente 8 nombre='Maria' direccion='Ronda de Nelle' telefono='444444444'>,
 <Cliente 9 nombre='Ana' direccion='Primera Ronda' telefono='44555554'>]
```

Mostrar todos los Clientes con nombre Alejandro

```
resultado = Cliente.select(Cliente.q.nombre == 'Alejandro')  
list(resultado)
```

```
[<Cliente 3 nombre='Alejandro' direccion='Ronda' telefono='44555554'>]
```

Mostrar todos los Clientes con nombre Ana Rosa

```
resultado = Cliente.select(Cliente.q.nombre == 'Ana Rosa')  
list(resultado)
```

```
[<Cliente 4 nombre='Ana Rosa' direccion='Ronda' telefono='44555554'>]
```

Cuando se asocia una clase con su tabla, automáticamente se crea un atributo, llamado **q**, de tipo `sqlobject.sqlbuilder.SQLObjectTable`. Este atributo contiene las definiciones de cada campo de la tabla como objetos `sqlobject.sqlbuilder.SQLObjectField`. Estos objetos se utilizan para aplicar condiciones de filtro para el método `select()`, puesto que ha reescrito los operadores de comparación habituales.

En estas consultas cuando convertimos en lista la variable **resultados**, es cuando realmente se realiza la consulta a la base de datos. Este comportamiento se denomina LAZY (o perezoso, o bajo demanda). Este comportamiento minimiza las operaciones sobre la base de datos, si no vamos a hacer uso de todos los resultados de una consulta.

Mostrar todos los Clientes con id >= 1

```
resultado = Cliente.select(Cliente.q.id >= 1)  
list(resultado)
```

```
[<Cliente 2 nombre='Ana Lucia' direccion='Avenida A' telefono='44555554'>,  
<Cliente 3 nombre='Alejandro' direccion='Ronda' telefono='44555554'>,  
<Cliente 4 nombre='Ana Rosa' direccion='Ronda' telefono='44555554'>,  
<Cliente 5 nombre='Manuel' direccion='Ronda' telefono='44555554'>,  
<Cliente 6 nombre='Pedro' direccion='Avenida' telefono='55555555'>,  
<Cliente 7 nombre='Juan' direccion='Calle 37' telefono='45858595'>,  
<Cliente 8 nombre='Maria' direccion='Ronda de Nelle' telefono='44444444'>,  
<Cliente 9 nombre='Ana' direccion='Primera Ronda' telefono='44555554'>]
```

El método `select()` también nos permite especificar condiciones válidas para la cláusula WHERE como un string:

Mostrar todos los Clientes con id >= 1

```
resultado = Cliente.select('Cliente.id >= 1')  
list(resultado)
```

```
[<Cliente 2 nombre='Ana Lucia' direccion='Avenida A' telefono='44555554'>,  
<Cliente 3 nombre='Alejandro' direccion='Ronda' telefono='44555554'>,  
<Cliente 4 nombre='Ana Rosa' direccion='Ronda' telefono='44555554'>,  
<Cliente 5 nombre='Manuel' direccion='Ronda' telefono='44555554'>,  
<Cliente 6 nombre='Pedro' direccion='Avenida' telefono='55555555'>,  
<Cliente 7 nombre='Juan' direccion='Calle 37' telefono='45858595'>,  
<Cliente 8 nombre='Maria' direccion='Ronda de Nelle' telefono='44444444'>,  
<Cliente 9 nombre='Ana' direccion='Primera Ronda' telefono='44555554'>]
```

Alternativamente al método `select()`, el método `selectBy()` permite establecer una consulta filtrada por igualdad en uno o varios campos:

Mostrar el Cliente con id = 2

```
resultado = Cliente.selectBy(id = 2)
list(resultado)
[<Cliente 2 nombre='Ana Lucia' direccion='Avenida A' telefono='44555554'>]
```

Cuando hacemos una consulta que debe retornar un único registro o ninguno, podemos usar el método `getOne()` para establecer un valor por defecto si no hay resultado.

Mostrar el Cliente con id = 19

```
resultado = Cliente.select(Cliente.q.id==19).
                                getOne('no existe')

resultado
'no existe'
```

Para combinar varias condiciones en una consulta debemos hacer uso de las funciones `NOT()`, `AND()`, y `OR()` del módulo `sqlobject`.

Mostrar los Cliente con nombre igual a Alejandro o Ana Rosa

```
resultado = Cliente.select(OR(Cliente.q.nombre=='Ana Rosa',
                              Cliente.q.nombre=='Alejandro')) )
list(resultado)

[<Cliente 3 nombre='Alejandro' direccion='Ronda' telefono='44555554'>,
 <Cliente 4 nombre='Ana Rosa' direccion='Ronda' telefono='44555554'>]
```

Podemos anidar consultas

Mostrar los Cliente con id > 2 y nombre diferente a Manuel

```
resultado = Cliente.select( AND(Cliente.q.id > 2,
                                NOT(Cliente.q.nombre=='Manuel')) )
list(resultado)

[<Cliente 3 nombre='Alejandro' direccion='Ronda' telefono='44555554'>,
 <Cliente 4 nombre='Ana Rosa' direccion='Ronda' telefono='44555554'>,
 <Cliente 6 nombre='Pedro' direccion='Avenida' telefono='55555555'>,
 <Cliente 7 nombre='Juan' direccion='Calle 37' telefono='45858595'>,
 <Cliente 8 nombre='Maria' direccion='Ronda de Nelle' telefono='444444444'>,
 <Cliente 9 nombre='Ana' direccion='Primera Ronda' telefono='44555554'>]
```

También podemos usar el método `filter()` para añadir varias condiciones como un `AND`.

Mostrar los Cliente con id > 0 y teléfono = 44555554

```
resultado = Cliente.select().filter(Cliente.q.id > 0).
                                filter(Cliente.q.telefono == '44555554')
```

```
list(resultado)
```

```
[<Cliente 2 nombre='Ana Lucia' direccion='Avenida A' telefono='44555554'>,
 <Cliente 3 nombre='Alejandro' direccion='Ronda' telefono='44555554'>,
 <Cliente 4 nombre='Ana Rosa' direccion='Ronda' telefono='44555554'>,
 <Cliente 5 nombre='Manuel' direccion='Ronda' telefono='44555554'>,
 <Cliente 9 nombre='Ana' direccion='Primera Ronda' telefono='44555554'>]
```

```
# Mostrar los Cliente con id > 0 y dirección = Ronda
```

```
resultado = Cliente.select().filter(Cliente.q.id > 0).
    filter(Cliente.q.direccion == 'Ronda')
list(resultado)
```

```
[<Cliente 3 nombre='Alejandro' direccion='Ronda' telefono='44555554'>,
 <Cliente 4 nombre='Ana Rosa' direccion='Ronda' telefono='44555554'>,
 <Cliente 5 nombre='Manuel' direccion='Ronda' telefono='44555554'>]
```

Funciones de comparación avanzadas.

La función `sqlobject.LIKE()` permite comparar un atributo de texto con un patrón. El patrón se corresponde con el que se utiliza en SQL con la cláusula `LIKE`: un `%` indica cualquier secuencia y un guion de subrayado un carácter.

```
# Mostrar los Cliente que su nombre comienza por Ana
```

```
resultado = Cliente.select( LIKE(Cliente.q.nombre, 'Ana%') )
list(resultado)
```

```
[<Cliente 2 nombre='Ana Lucia' direccion='Avenida A' telefono='44555554'>,
 <Cliente 4 nombre='Ana Rosa' direccion='Ronda' telefono='44555554'>,
 <Cliente 9 nombre='Ana' direccion='Primera Ronda' telefono='44555554'>]
```

```
# Mostrar los Cliente que su nombre comienza por la letra A
```

```
resultado = Cliente.select( LIKE(Cliente.q.nombre, 'A%') )
list(resultado)
```

```
[<Cliente 2 nombre='Ana Lucia' direccion='Avenida A' telefono='44555554'>,
 <Cliente 3 nombre='Alejandro' direccion='Ronda' telefono='44555554'>,
 <Cliente 4 nombre='Ana Rosa' direccion='Ronda' telefono='44555554'>,
 <Cliente 9 nombre='Ana' direccion='Primera Ronda' telefono='44555554'>]
```

```
# Mostrar los Cliente que su nombre comienza por Ana y tiene un
segundo nombre
```

```
resultado = Cliente.select(LIKE(Cliente.q.nombre, 'Ana _%'))
list(resultado)
```

```
[<Cliente 2 nombre='Ana Lucia' direccion='Avenida A' telefono='44555554'>,
 <Cliente 4 nombre='Ana Rosa' direccion='Ronda' telefono='44555554'>]
```


Nivel III: Aplicaciones Web

Si queremos mostrar resultados únicos debemos asignar el parámetro `distinct` a valor `True`.

```
Cliente.select( distinct = True )
```

También podemos encadenar el método `distinct()` a la consulta.

```
Cliente.select().distinct()
```

Funciones de agrupación.

Podemos obtener el número de registros de una consulta encadenando el método `count()`.

Número de cliente cuyos nombres comiencen por A

```
resultado = Cliente.select( LIKE(Cliente.q.nombre,  
                                                                           'A%')).count()  
  
resultado  
4
```

Métodos:

- **avg()** permite calcular medias aritméticas
- **sum()** permite calcular la suma de los valores en un campo
- **min()** permite obtener el valor mínimo de un campo.
- **max()** permite obtener el valor máximo de un campo.
- **accumulateOne(operación, campo)** podemos aplicar una de las operaciones anteriores ('sum', 'min', 'max', 'avg') sobre el resultado de una consulta.
- **accumulateMany(*expresiones)** podemos aplicar varias operaciones sobre el resultado de una consulta. Cada expresión es un par indicando la operación y el campo.
- **accumulate(*expresiones)** podemos añadir expresiones de acumulación en la parte `select` de la consulta.

Veamos algunos ejemplos:

```
# Promedio o media de los id  
resultado = Cliente.select().avg(Cliente.q.id)  
  
# Suma de los id  
resultado = Cliente.select().sum(Cliente.q.id)  
  
# Valor mínimo en id  
resultado = Cliente.select().min(Cliente.q.id)  
  
# Valor máximo en id  
resultado = Cliente.select().max(Cliente.q.id)  
# Valor suma acumulada de id  
resultado = Cliente.select().accumulateOne('sum', 'id')
```

Nivel III: Aplicaciones Web

```
# Valor suma acumulada de id y minimo id
resultado = Cliente.select().accumulateMany(('sum', 'id'),
('min', Cliente.q.id))

# Valor Cantidad total de registros y máximo id
resultado = Cliente.select().accumulate('count(*)', 'max(id)')
```

Ordenar Consultas

Para ordenar la consulta debemos hacer uso del parámetro `orderBy` del método `select()`.

```
Cliente.select( orderBy=Cliente.q.nombre )
```

También podemos asignar el nombre de la columna como un string:

```
Cliente.select( orderBy='cliente.nombre' )
```

Para aplicar la ordenación descendente se utiliza la función `DESC()` o el string `'-campo'`:

```
Cliente.select( orderBy=DESC(Cliente.q.nombre) )
Cliente.select( orderBy='-cliente.nombre' )
```

Otra forma de ordenar descendentemente es realizar una consulta y encadenar el método `reversed()`:

```
Cliente.select( orderBy='cliente.nombre').reversed()
```

Además, el parámetro `orderBy` admite una lista de columnas. Por ejemplo, si queremos ordenar los clientes por su nombre en orden ascendente, y por su dirección en orden descendente:

```
Cliente.select( orderBy=['cliente.nombre',
                        '-cliente.direccion'] )
Cliente.select( orderBy=[Cliente.q.nombre,
                        DESC(Cliente.q.direccion)] )
```

Si vamos a aplicar el mismo tipo de ordenación sobre todas las consultas que hagamos podemos establecer el criterio en el atributo `sqlmeta.defaultOrder` de la clase:

```
Cliente.sqlmeta.defaultOrder = 'cliente.nombre'
```

A menos que indiquemos lo contrario, las consultas posteriores a esta asignación se ordenarán por el nombre del cliente. Para anular la ordenación por defecto, debemos asignar `None` en el parámetro `orderBy`.

Relaciones entre tablas

Como hemos visto anteriormente existen diferentes tipos de relaciones entre tablas, ahora veremos como representar estas relaciones utilizando SQLAlchemy:

Relación de uno-a-varios

Para reflejar esta relación, en la clase Compra debemos añadir un atributo que haga referencia a su cliente mediante un objeto de tipo ForeignKey:

```
class Compra(SQLObject):
    fecha = DateCol()
    costo = DecimalCol(size=7, precision=2)
    cliente = ForeignKey('Cliente')

if not Compra.tableExists():
    Compra.createTable()

compra1 = Compra(fecha='2020-05-20', costo=125.00, cliente = 5)
```

Mediante ForeignKey('Cliente') referenciamos la clase Cliente y en la tabla física COMPRA se creará una clave foránea con el nombre cliente_id.

Pero, además, en la tabla Cliente debemos añadir un atributo que haga referencia a sus compras. Para ello se utiliza un objeto de tipo MultipleJoin:

```
class Clientes(SQLObject):
    nombre = StringCol(default='')
    direccion = StringCol()
    compras = MultipleJoin('Compra')
```

Relaciones varios-a-varios.

Si un cliente puede tener varias compras y una compra puede ser de varios clientes, nos encontraremos con una relación varios-a-varios.

En este caso, en la base de datos será necesario crear una tabla intermedia que asocie las claves de cliente con las claves de compra. Y en las clases debemos declarar los siguientes atributos de relación:

Relaciones varios a varios

```
class Cliente(SQLObject):
    nombre = StringCol(default='')
    direccion = StringCol()
    compras = RelatedJoin('Compra')

class Compra(SQLObject):
    fecha = DateCol()
```

```
costo = DecimalCol(size=7, precision=2)
clientes = RelatedJoin('Cliente')
```

La función `RelatedJoin()`, además de establecer la relación entre las tablas, provoca la creación de un método `addCompra()` en la clase `Cliente`, y un método `addCliente()` en la clase `Compra`. Usaremos estos métodos para relacionar instancias.

```
Cliente.createTable()
Compra.createTable()
```

Durante el proceso de creación de tablas se crea una tabla intermedia de nombre `cliente_compra`, que define como columnas foráneas las claves principales de `cliente` y `compra`. Esta tabla no se expone a través de ninguna clase, y sus registros tampoco pueden ser recuperados directamente.

En esta situación debemos crear las compras y clientes de forma independiente:

```
Juan = Cliente(nombre='Juan', direccion="C/ Alfonso")
Ana = Cliente(nombre='Ana', direccion="C/ Nelle")
compra1 = Compra(fecha=datetime.now(), costo=100)
compra2 = Compra(fecha=datetime.now(), costo=200)
```

Para asociar `compra1` con los dos clientes basta con usar el método `addCliente()`:

```
compra1.addCliente(Juan)
compra1.addCliente(Ana)
```

Podemos comprobar las compras de 'Juan' y de 'Ana':

```
Juan.compras
[<Compra 1 fecha='datetime.date(202...)' costo=100>]
```

```
Ana.compras
[<Compra 1 fecha='datetime.date(202...)' costo=100>]
```

Asimismo, podemos añadir compras a un cliente:

```
Juan.addCompra(compra2) # Se utiliza addCliente o addCompra
Juan.compras

[<Compra 1 fecha='datetime.date(202...)' costo=100>,
 <Compra 2 fecha='datetime.date(202...)' costo=200>]
```

Basta con usar uno de los métodos `addCompra()` o `addCliente()` para establecer la relación, no es necesario usar los dos.

Relaciones uno-a-uno.

Si un cliente puede tener una única compra y una compra solo puede ser de un cliente, nos encontraremos con una relación uno-a-uno.

Nivel III: Aplicaciones Web

En este caso, en la base de datos el esquema de las tablas es similar al de una relación uno-a-varios, en una de las tablas se pone una clave foránea que hace referencia a la otra tabla. Y en las clases debemos declarar los siguientes atributos de relación:

```
class Cliente(SQLObject):
    nombre = StringCol(default='')
    direccion = StringCol()
    compras = ForeignKey('Compra')

class Compra(SQLObject):
    fecha = DateCol()
    costo = DecimalCol(size=7, precision=2)
    clientes = SingleJoin('Cliente')
```

En una de las clases declaramos un `ForeignKey()`, lo que provocará que se cree en la tabla correspondiente la clave foránea, y en la otra clase declaramos un `SingleJoin()`, que se encargará de generar las consultas apropiadas.

```
Cliente.createTable()
Compra.createTable()
```

Las compras se crearán de forma independiente, y al crear un cliente debemos especificar su compra:

```
compra1 = Compra(fecha=datetime(2019, 1, 3), costo=100)
Cliente(nombre='Juan', direccion="C/", compra=compra1)

# Consultar el cliente de la compra 1
compra1.cliente
```

Consultas entre clases relacionadas.

A través de los atributos de relación podemos crear consultas que filtren ambas clases. Supongamos el caso de la relación uno-a-varios entre clientes y compras.

```
db_filename = os.path.abspath(PATH_DATA + '\ClientesVentas.db')
connection_string = 'sqlite:' + db_filename
connection = connectionForURI(connection_string)
sqlhub.processConnection = connection

class Clientes(SQLObject):
    class sqlmeta:
        fromDatabase = True
    nombre = StringCol(default='')
    direccion = StringCol()
    compras = MultipleJoin('Compra')

class Compras(SQLObject):
    class sqlmeta:
```

```
fromDatabase = True
fecha = DateCol()
costo = DecimalCol(size=7, precision=2)
cliente = ForeignKey('Clientes')
```

Podemos obtener los clientes que tengan una compra con un costo mayor que 100 de la siguiente manera:

```
resultado =
Clientes.select(AND(Compras.q.clienteID==Clientes.q.id,
                    Compras.q.costo > 100))
list(resultado)
```

```
[<Clientes 1 nombre='Manuel' direccion='Ronda'>]
```

Al corresponderse cada compra con un único cliente (relación mediante `ForeignKey`), el objeto incluye un atributo **clienteID** que hace referencia a su clave foránea **cliente_id**. Pero al igual que se crea el objeto **q**, también se crea un objeto **j** para facilitar relaciones. La consulta anterior también podremos expresarla así:

```
resultado = Clientes.select(AND(Compras.j.cliente,
                               Compras.q.costo > 100))
list(resultado)
[<Clientes 1 nombre='Manuel' direccion='Ronda'>]
```

La expresión `Compra.j.cliente` es equivalente a `(Compra.q.clienteID==Cliente.q.id)`. Para las relaciones de tipo `RelatedJoin` y `SingleJoin`, el objeto **j** proporcionará atributos (con el nombre del atributo de relación) que serán equivalentes a lo que pondríamos en la cláusula `WHERE` para relacionar las tablas.

También es posible especificar las condiciones como un string:

```
resultado = Clientes.select('compras.cliente_id = clientes.id
AND compras.costo > 100', clauseTables=['compras'])
list(resultado)
[<Clientes 1 nombre='Manuel' direccion='Ronda'>]
```

Debemos especificar en el parámetro `clauseTables` una lista con los nombres de las tablas que vamos a relacionar.

Forzando las consultas de JOIN

En ocasiones, al realizar una consulta sobre tablas relacionadas nos puede interesar que se combinen registros de una tabla aunque no existan registros asociados en la otra tabla. Por ejemplo, podemos consultar clientes y su número de compras aunque el cliente no tenga compras, y en este caso debemos obtener el número cero. Este tipo de consultas se corresponde con un `LEFT JOIN`.

Nivel III: Aplicaciones Web

Si queremos forzar una consulta LEFT JOIN debemos hacer uso del parámetro join del método select() y de un objeto de tipo LEFTJOINOn.

```
from sqlobject.sqlbuilder import LEFTJOINOn

resultado = Clientes.select(join=LEFTJOINOn(Clientes, Compras,
Compras.j.cliente), distinct=True)
list(resultado)
```

A veces nos puede interesar realizar un join de una tabla consigo misma. Para estos casos podemos crear un alias para una tabla:

```
from sqlobject.sqlbuilder import Alias
aliasCliente = Alias(Clientes, "Cliente_2")
```

Si ahora, por ejemplo, queremos obtener aquellos clientes que tienen un mismo nombre:

```
resultado = Clientes.select(
AND(Clientes.q.id != aliasCliente.q.id,
Clientes.q.nombre == aliasCliente.q.nombre),
distinct = True)
list(resultado)

[<Clientes 1 nombre='Manuel' direccion='Ronda'>,
<Clientes 2 nombre='Manuel' direccion='Ronda'>,
<Clientes 3 nombre='Manuel' direccion='Ronda'>]
```

Subconsultas.

Para crear consultas que incluyan subconsultas debemos hacer uso de las siguientes clases y funciones del espacio de nombres sqlobject.sqlbuilder: Select, Outer, IN, NOTIN, SOME, ANY, ALL, NOTEXISTS y EXISTS.

Por ejemplo, para obtener clientes que tengan alguna compra igual a 20 euros de costo:

```
from sqlobject.sqlbuilder import NOTEXISTS, IN, EXISTS, Select,
Outer

resultado = Clientes.select(
IN(20, Select(Compras.q.costos,
where=Compras.q.id==Outer(Clientes).q.id) ) # Outer hace
referencia a la tabla Clientes
)
list(resultado)
[<Clientes 3 nombre='Manuel' direccion='Ronda'>]
```

Debemos hacer uso de la función Outer() para hacer referencia a la tabla Cliente de la consulta externa.

Nivel III: Aplicaciones Web

Podemos hacer uso de funciones como IN, NOTIN, SOME, ANY y ALL para evaluar los valores devueltos por la subconsulta. Por ejemplo, podemos obtener clientes que incluyan el valor 100 en el costo de alguna de sus compras:

```
resultado = Clientes.select( EXISTS( Select(
    Compras.q.id,
where=AND(Compras.q.clienteID==Outer(Clientes).q.id,
    Compras.q.costo>100)
    ) ) )
list(resultado)
[<Clientes 1 nombre='Manuel' direccion='Ronda'>]
```

Técnicas avanzadas con SQLAlchemy.

Añadiendo propiedades.

En las clases de mapeado con tablas podemos añadir todos los métodos que necesitemos como con cualquier otra clase. Por ejemplo:

```
from sqlalchemy import *
import os

db_filename = os.path.abspath(PATH_DATA +
    '\ClientesVentasTA.db')
connection_string = 'sqlite:' + db_filename
connection = create_engine(connection_string)
sqlhub.processConnection = connection

class Cliente(SQLObject):
    nombre = StringCol(default='d', alternateID=True)
    direccion = StringCol()

    def info(self):
        return "Cliente: {},
            dirección {}".format(self.nombre,
                self.direccion)

Cliente.createTable()

Juan = Cliente(nombre='Juan', direccion="Alfonso")
Ana = Cliente(nombre='Ana', direccion="Nelle")
```

El método info() nos permitirá recuperar la información de una instancia:

```
c1 = Cliente.get(1)
print( c1.info() )
```

Cliente: Juan, dirección Alfonso

Nivel III: Aplicaciones Web

Pero existen unos métodos especiales que nos permitirán definir propiedades. Por ejemplo, supongamos que a la clase `Cliente` debemos añadir una propiedad llamada `telefono` que no se corresponde con una columna en la tabla física. Podemos añadir métodos `_set_telefono()`, `_get_telefono()`, `_del_telefono()`, y/o `_doc_telefono()` en la clase:

```
from sqlobject import *
import os
from datetime import datetime

PATH_DATA = r'C:\Users\msierra\Desktop\SQLDatabase'

db_filename = os.path.abspath(PATH_DATA +
'\ClientesVentasTA.db')
connection_string = 'sqlite:' + db_filename
connection = connectionForURI(connection_string)
sqlhub.processConnection = connection

class Cliente(SQLObject):
    nombre = StringCol(default='d', alternateID=True)
    direccion = StringCol()

    def info(self):
        return "Cliente: {}, dirección {}".format(self.nombre,
self.direccion)

    def __init__(self, telefono=None, **kw):
        self.tel = telefono
        return super().__init__( **kw)

    def _get_telefono(self):
        return self.tel

    def _set_telefono(self, value):
        self.tel = value

    def _del_telefono(self, value):
        del(self.tel)

    def _doc_telefono(self):
        return "Teléfono del cliente"

Cliente.createTable()
```

Todos estos métodos con el sufijo `_telefono` constituyen una propiedad `telefono`, que actúa como un atributo más de una instancia de la clase. No es necesario implementar todos los métodos. Por ejemplo, si sólo definimos `_get_telefono()` estaríamos creando una propiedad de sólo lectura.

Nivel III: Aplicaciones Web

De esta manera podremos realizar las siguientes operaciones:

```
Maria = Cliente(telefono='1111111', nombre='Maria',  
                direccion="Ronda")
```

```
Maria.telefono = "2222222"  
print(Maria.telefono)  
2222222
```

```
Alejandro = Cliente(telefono='55555', nombre='Alejandro',  
                    direccion="Ronda")  
Alejandro.telefono  
55555
```

```
Alejandro.telefono = "4562686" # modificamos el telefono  
Alejandro.telefono  
4562686
```

```
Alejandro._del_telefono("4562686") # Eliminamos el telefono
```

Modificando los atributos de columnas

Hemos visto cómo añadir propiedades a las subclases de SQLAlchemy, también podemos añadir propiedades asociadas a las columnas de la tabla correspondiente.

Es posible porque SQLAlchemy crea automáticamente métodos `_SO_set_atributo(valor)` y `_SO_get_atributo()` para cada atributo asociado a una columna. Podemos crear propiedades que utilicen internamente estos métodos. Por ejemplo, en la clase `Cliente` controlaremos que el nombre comience siempre por una mayúscula:

```
from sqlalchemy import *  
import os  
from datetime import datetime  
  
PATH_DATA = r'C:\Users\msierra\Desktop\SQLDatabase'  
  
db_filename = os.path.abspath(PATH_DATA +  
                                '\ClientesVentasMP.db')  
connection_string = 'sqlite:' + db_filename  
connection = connectionForURI(connection_string)  
sqlhub.processConnection = connection  
  
class Cliente(SQLObject):  
    nombre = StringCol(default='d', alternateID=True)  
    direccion = StringCol()  
  
    def _get_nombre(self):  
        return self._SO_get_nombre().capitalize()  
  
    def _set_nombre(self, value):
```

```
self._SO_set_nombre(value.capitalize())

Cliente.createTable()

Juan = Cliente(nombre='juan', direccion="Calle Alfonso")
Juan.nombre
'Juan'
```

Gestión de eventos

Los eventos o señales son mecanismos que nos notifican de que los datos o esquema de una tabla han cambiado a través de las instancias SQLAlchemy.

Si queremos hacer un seguimiento de cambios en la base de datos podemos registrar funciones, asociándolas a un evento determinado.

Los eventos están asociados a las siguientes clases del espacio de nombres `sqlalchemy.events`:

| Clases de eventos | Descripción |
|---------------------------------|--|
| <code>ClassCreateSignal</code> | Se produce después de crear la clase. |
| <code>CreateTableSignal</code> | Se produce después de crear la tabla. |
| <code>AddColumnSignal</code> | Se produce después de añadir una nueva columna a la tabla. |
| <code>DeleteColumnSignal</code> | Se produce después de eliminar una columna de la tabla. |
| <code>DropTableSignal</code> | Se produce después de eliminar la tabla. |
| <code>CommitSignal</code> | Se produce una vez confirmados cambios. |
| <code>RollbackSignal</code> | Se produce una vez anulados cambios pendientes. |
| <code>RowCreateSignal</code> | Se produce antes de que se añada un nuevo registro a la tabla. |
| <code>RowCreatedSignal</code> | Se produce después de que se añada un nuevo registro a la tabla. |
| <code>RowDestroySignal</code> | Se produce antes de que se elimine un registro de la tabla. |
| <code>RowDestroyedSignal</code> | Se produce después de que se elimine un registro de la tabla. |
| <code>RowUpdateSignal</code> | Se produce antes de que se modifique un registro a la tabla. |
| <code>RowUpdatedSignal</code> | Se produce después de que se modifique un registro a la tabla. |

Se utiliza la función `sqlalchemy.events.listen(func, tabla, evento)` para asociar una clase de evento con una función que gestione el evento sobre una tabla concreta.

Por ejemplo, para gestionar la inserción de nuevos registros sobre la tabla de clientes:

```
from sqlalchemy.events import listen, RowCreateSignal

def create_cliente_listener(self, instance, kwargs):
    """
    instance: un diccionario con los datos del registro.
    kwargs: una lista.
    """
    import datetime
    self.fechaCreacion = datetime.datetime.now()
```

```
listen(create_cliente_listener, Cliente, RowCreateSignal)
```

Herencia

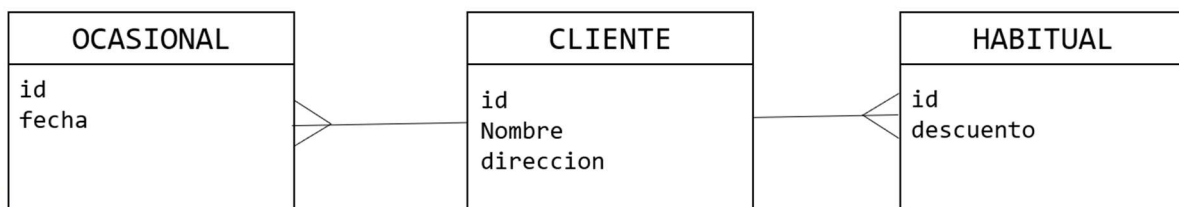
Por último, veremos cómo reflejar herencia de clases en una base de datos a través de SQLAlchemy.

Diseño de herencia entre clases.

Supongamos que en nuestra aplicación de gestión de clientes debemos distinguir entre clientes ocasionales y clientes habituales.

Tendremos una clase `Cliente`, con su nombre y dirección; una subclase `Ocasional`, que además del nombre y dirección tendrán la fecha de la última compra; y una subclase `Habitual`, que además de su nombre y dirección tendrá asignado un valor de descuento especial.

En la base de datos física tendremos tres tablas para cada tipo:



Podrán existir varios clientes ocasionales y varios clientes habituales, pero cada registro de Ocasional y de Habitual se corresponde con un único registro de Cliente (esta tabla contendrá los datos comunes a todos los tipos de cliente). Los id de Ocasional y Habitual serán además claves foráneas de la tabla Cliente.

Para implementar esto en SQLAlchemy crearemos las clases correspondientes a cada tabla, pero estableceremos que Cliente va a ser un SQLAlchemy heredable usando herencia sobre InheritableSQLObject en vez de herencia sobre SQLAlchemy.

```

from sqlalchemy import *
from sqlalchemy.inheritance import InheritableSQLObject

import os

PATH_DATA = r'C:\Users\msierra\Desktop\SQLDatabase'
db_filename = os.path.abspath(PATH_DATA + '\ClientesH.db')
connection_string = 'sqlite:' + db_filename
connection = connectionForURI(connection_string)
sqlhub.processConnection = connection

class Cliente(InheritableSQLObject):
    nombre = StringCol(length=100, default='',
                       alternateID=True)
    direccion = StringCol()
```

Nivel III: Aplicaciones Web

```
class Ocasional(Cliente):
    _inheritable = False
    fecha = DateCol()

class Habitual(Cliente):
    _inheritable = False
    descuento = FloatCol(default= 0.1)

Cliente.createTable()
Ocasional.createTable()
Habitual.createTable()
```

El atributo `_inheritable` inserta una columna de discriminación llamada `child_name` a cada tabla. Esta columna será asignada automáticamente por `SQLObject` al nombre de cada subclase. Este campo puede resultar útil para consultar directamente la tabla `Cliente` por sus distintos tipos de registros, pero no resulta nada útil en las tablas correspondientes a las subclases. Incluso, a efectos de programación orientada a objetos, realmente no es necesaria. Podremos trabajar con objetos `Cliente`, `Ocasional` y `Habitual` de la manera que ya hemos visto sin ningún problema. Para no generar esta columna debemos asignar el atributo `_inheritable`, en las subclases, a valor `False`.

Es importante destacar que cada instancia de la clase secundaria tendrá el mismo ID asociado que la instancia de la clase principal.

Insertemos registros en las tablas `Ocasional` y `Habitual`

```
Ocasional(nombre='Juan', direccion='Madrid', fecha='2017-01-04' )
Habitual(nombre='María', direccion='Jerez', descuento=0.2 )
Ocasional(nombre='Andrés', direccion='Lugo', fecha='2019-02-04' )
```

```
<Ocasional 4 fecha='datetime.date(201...) ' nombre='Andrés' direccion='Lugo'>
```

Por cada nueva instancia se crea un registro en la tabla cliente (con el nombre y dirección) y otro en la tabla ocasional o habitual con el mismo valor de ID y los datos específicos de cada tabla. Podemos recuperar todos los clientes:

```
print(list(Cliente.select()))
[<Ocasional 2 fecha='datetime.date(201...) ' nombre='Juan' direccion='Madrid'>,
<Habitual 3 descuento=0.2 nombre='María' direccion='Jerez'>, <Ocasional 4
fecha='datetime.date(201...) ' nombre='Andrés' direccion='Lugo'>]

print(list(Ocasional.select()))
[<Ocasional 2 fecha='datetime.date(201...) ' nombre='Juan' direccion='Madrid'>,
<Ocasional 4 fecha='datetime.date(201...) ' nombre='Andrés' direccion='Lugo'>]

print(list(Habitual.select()))
[<Habitual 3 descuento=0.2 nombre='María' direccion='Jerez'>]
```

Nivel III: Aplicaciones Web

No habrá ningún problema en actualizar y eliminar instancias de las subclases. Podemos actualizar, por ejemplo, el nombre de un cliente Ocasional y la actualización se trasladará a la tabla base cliente:

```
a = Ocasional.select(Cliente.q.nombre == 'Juan')[0]
a.nombre = 'Juan Antonio'
```

Al eliminar una instancia de cliente se eliminará tanto el registro en la tabla base como en la tabla secundaria:

```
Ocasional.delete(2)
```

Consultas con herencia

El atributo `q` sigue funcionando con la clase base y las clases secundarias. Simplemente se generará en la cláusula `WHERE` la condición de relación adecuada entre los registros asociados. Por ejemplo, seleccionaremos los clientes habituales con un descuento superior a 0.1:

```
resultado = Habitual.select(Habitual.q.descuento > 0.1)
print(list(resultado))
[<Habitual 3 descuento=0.2 nombre='María' direccion='Jerez'>]
```

Relaciones con otras tablas

No habrá problema en seguir estableciendo relaciones uno a varios o varios a varios con una clase base o secundaria cuando aplicamos herencia.

Si añadimos la clase `Compra`, con las compras que puede realizar cualquier cliente, simplemente añadiremos los atributos correspondientes de relación en la clase base `Cliente` y la nueva clase `Compra`:

```
class Cliente(SQLObject):
    nombre = StringCol(default='')
    direccion = StringCol()
    compras = MultipleJoin('Compra')

class Compra(SQLObject):
    fecha = DateCol()
    costo = DecimalCol(size=7, precision=2)
    cliente = ForeignKey('Cliente')
```

Creamos clientes y compras:

```
o1 = Ocasional(nombre='Juan', direccion='Madrid',
               fecha='2017-01-04' )
h1 = Habitual(nombre='María', direccion='Jerez', descuento=0.2 )
o2 = Ocasional(nombre='Andrés', direccion='Lugo',
               fecha='2019-02-04' )
c1 = Compra(fecha='2020-01-02', costo=200, cliente=o1)
o1.fecha = '2010-01-02'
```


Nivel III: Aplicaciones Web

```
c2 = Compra(fecha='2020-01-12', costo=100, cliente=h1)
c2 = Compra(fecha='2020-01-13', costo=150, cliente=h1)
```

Y consultamos las compras del cliente habitual:

```
print(h1.compras)
```

Obteniendo:

```
[<Compra 2 fecha='datetime.date(202...) ' costo=100
clienteID=2>,
<Compra 3 fecha='datetime.date(202...) ' costo=150 clienteID=2>]
```

Es decir, las subclases hereden los atributos de relación declarados en la superclase.

Limitaciones de la herencia

La API SQLAlchemy presenta las siguientes limitaciones con la herencia:

- Sólo funciona la herencia única. No es posible heredar de múltiples clases SQLAlchemy.
- Es posible heredar de una clase heredada y esto funcionará bien. Por ejemplo, podemos crear una subclase Intermittente que herede de Ocasional, y todos los atributos de los padres estarán disponibles.
- No pueden redefinir columnas en una clase heredada (esto generará una excepción).
- La implementación de la herencia es incompatible con las actualizaciones diferidas. No se puede establecer `lazyUpdate` en `True`.
- La herencia funciona en dos etapas: primero dibuja los ID de la tabla principal y luego dibuja las filas de las tablas secundarias. La primera etapa podría fallar si intentamos hacer cosas complejas. Por ejemplo, `Ocasional.select(orderBy = Ocasional.q.fecha, distinct=True)` podría fallar porque en la primera etapa la herencia genera una consulta `SELECT` para la tabla primaria con el `ORDER BY` de la columna de la tabla secundaria.
- Aunque todos los atributos son heredados, lo mismo no se aplica a los datos `sqlmeta`.