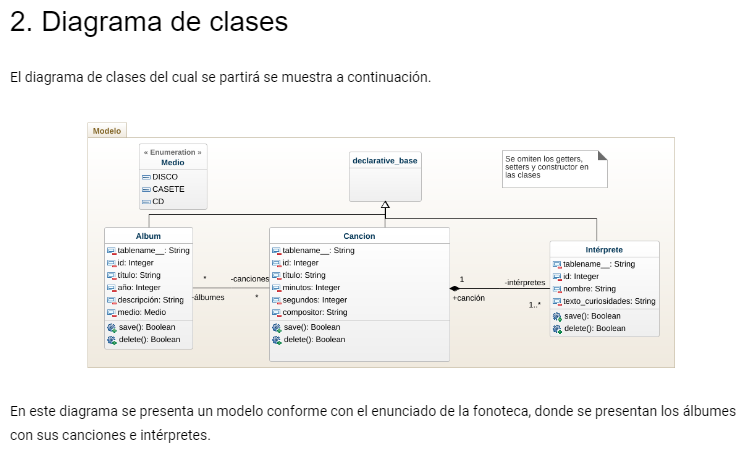
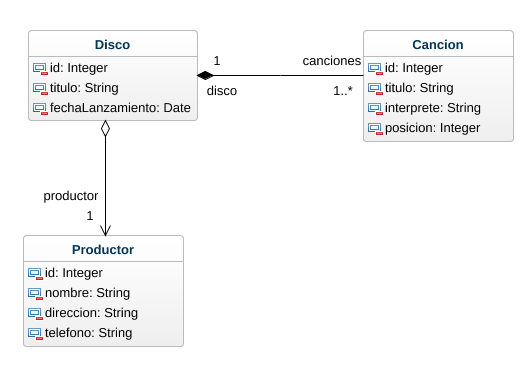
<https://misovirtual.virtual.uniandes.edu.co/codelabs/tutorial-SQLAlchemy-Python-ejemplo/index.html?index=..%2F..index#1>

todos los cursos: https://misovirtual.virtual.uniandes.edu.co/codelabs/





3. Instalar SQLAlchemy y crear proyecto

Para comenzar a utilizar SQLAlchemy, primero se debe instalar la librería correspondiente, utilizando la siguiente instrucción.

pipinstallsqlalchemy

Una vez instalada, ingrese al prompt de python con la siguiente instrucción:

python

Verifique la versión actual de sqiAchemy en Python revisando la siguiente variable luego de importar la librería:

>>>importsqlalchemy

>>>sqlalchemy.\_\_version\_\_

Debe obtener una salida similar a la siguiente:

'1.2.11'

Para salirse, recuerde que puede utilizar ^Z

>>> ^Z

Para comenzar este tutorial, primero cree una carpeta localmente, con el nombre del proyecto que trabajará, por ejemplo, tutorial\_sql\_alchemy. Una vez creada la carpeta, cree al interior otra carpeta llamada src, con el propósito de alojar allí el código que se va a desarrollar.

Si revisamos la estructura de carpetas, tendremos algo similar a lo siguiente

/user/path/to/tutorial\_sql\_alchemy/src

Y a continuación, puede crear un archivo vacío llamado \_\_init\_\_.py dentro de la carpeta src p**ara indicarle a python que esta carpeta puede ser un módulo**, y continuar almacenando los demás archivos de este tutorial también dentro de la carpeta src.

## 4. Definición de motor, sesión y base

Cree un archivo llamado declarative\_base.py, para adicionar las instrucciones que utilizaremos a continuación, e inicie el archivo con la importación de las siguientes librerías:

fromsqlalchemyimportcreate\_engine

fromsqlalchemy.ext.declarativeimportdeclarative\_base

fromsqlalchemy.ormimportsessionmaker

Para interactuar con una base de datos, primero se debe contar con un motor (Engine), el cual se encarga de administrar los llamados que se realizan para el almacenamiento y recuperación de información, y sobre el que se especifican los datos para el uso de la base de datos. Para utilizar este motor con SQL Lite, primero se importó la librería create\_engine, y luego se debe crear en Python la base de datos con la siguiente instrucción:

engine=create\_engine('sqlite:///baseDeDatos.sqlite')

Donde baseDeDatos representa el nombre del archivo con la base de datos en SQL Lite, la cual, al no tener una ruta de almacenamiento, se crea en la raíz del proyecto. Para cambiar el sitio de almacenamiento de la base de datos se puede especificar una ruta antes del nombre del archivo.

Además del motor, se debe contar con una sesión, la cual implementa el patrón de diseño Unit of Work, un objeto que tendrá las transacciones realizadas en un conjunto de instrucciones para actualizar los cambios en la base de datos. Para definir la sesión se crea una clase con la siguiente instrucción:

Session=sessionmaker(bind=engine)

Esta clase permitirá crear los objetos para cada sesión y realizar el almacenamiento y recuperación de los datos.

Finalmente, se debe crear una clase Base, de la cual heredarán las clases del diagrama presentado con el propósito de definir las propiedades a almacenar sobre la base de datos, y sobre la que se definirá el dialecto con el cual se traducirán las consultas en la base de datos, ya que este puede ser eventualmente personalizado. Para crear la clase base se utiliza la siguiente instrucción

Base=declarative\_base()

Estos tres elementos se definen dentro de un archivo que se llamará declarative\_base.py, de la siguiente manera:

fromsqlalchemyimportcreate\_engine

fromsqlalchemy.ext.declarativeimportdeclarative\_base

fromsqlalchemy.ormimportsessionmaker

engine=create\_engine('sqlite:///aplicacion.sqlite')

Session=sessionmaker(bind=engine)

Base=declarative\_base()

5. Definición de las clases

SQL Alchemy provee sus tipos de datos para modelar la información en la base de datos. Esos tipos de datos permiten hacer un mapa de los atributos o propiedades de la clase en el almacenamiento, y permiten especificar en cuál entidad se van a guardar los datos. Para definir la clase se debe:

1. Definir la tabla sobre la que se va a almacenar la información utilizando la propiedad \_\_tablename\_\_ con la asignación del nombre de la tabla como una cadena de caracteres.
2. Definir cada propiedad de la clase asignando un valor de la clase Column y la especificación del tipo de dato que va a almacenar.
3. Definir las relaciones de la clase con las demás a través de propiedades a las que se les va a asignar una relación.

La primera parte es definir las clases con sus propiedades. Cada clase debe conservar las siguientes características:

1. Heredar de la clase Base, con el propósito de poder definir sus propiedades como columnas en una tabla de la base de datos.
2. Incluir una propiedad llamada \_\_tablename\_\_ que hará referencia al nombre de la entidad en la base de datos a mapear
3. Definir una propiedad llamada id que se utilizará como llave principal en la base de datos y que será una columna de tipo entero
4. Definir las demás propiedades declaradas como columnas de diferentes tipos.

A continuación se define la clase Canción (en este momento sin relaciones) en el archivo cancion.py según estas especificaciones:

fromsqlalchemyimportColumn, Integer, String

from .declarative\_baseimportBase

classCancion(Base):

\_\_tablename\_\_ = 'cancion'

id = Column(Integer, primary\_key=True)

titulo = Column(String)

minutos = Column(Integer)

segundos = Column(Integer)

compositor = Column(String)

De acuerdo a la definición, se identifican dos tipos de columnas para SQLAlchemy, las cuales se definen como enteros y cadenas de caracteres, sin embargo, la especificación permite definir otros tipos como Boolean, Numeric o Date.

De esta manera se pueden definir también las clases Album e Intérprete.

6.-Definición de las relaciones

https://misovirtual.virtual.uniandes.edu.co/codelabs/tutorial-SQLAlchemy-Python-ejemplo/index.html?index=..%2F..index#5

Las clases pueden definir relaciones de tipo uno a uno, uno a muchos, muchos a uno y muchos a muchos. Para el ejemplo podemos identificar las siguientes relaciones:

1. Entre Cancion e Interprete, se tiene una relación de uno a muchos, donde la canción conoce a sus intérpretes y el intérprete conoce a su canción. Adicionalmente esta es una asociación compuesta, por lo que un intérprete sólo puede relacionarse con una canción, y si la canción deja de existir, los intérpretes asociados también.
2. Entre Album y Cancion, se tiene una relación donde el álbum conoce a sus canciones, y las canciones conocen a los álbumes a los que pertenecen, por lo que se puede tomar como una relación muchos a muchos.

Existe otro tipo de relación llamada uno a uno, la cual no se aborda directamente en este diagrama, sin embargo, en el caso de la relación de muchos a muchos, se deberá abordar este tipo de relación para explicar el caso.

Primer caso: Relación uno a muchos

Para la primera relación, Cancion tiene una relación de uno a muchos la clase Interprete. Para esta relación se utiliza una propiedad llamada interpretes (Se conserva el nombre de la relación en el diagrama de clases), la cual se crea como una relación con la palabra relationship, y a continuación se especifica la clase con la cual se relaciona, como se muestra a continuación:

interpretes = relationship('Interprete')

Adicionalmente, en las librerías que se traen desde el ORM de SQL Alchemy se debe importar relationship de la siguiente manera:

fromsqlalchemy.ormimportrelationship

En la relación también se puede definir qué sucede con los elementos relacionados o cómo se propagan las operaciones sobre un objeto respecto a los relacionados, en este caso, lo que sucede con los intérpretes, cuando se edita o borra una canción. Al tener una asociación compuesta de Canción a Intérprete, los intérpretes asociados a una canción deberían ser borrados cuando se elimina la canción, o un intérprete no debería existir si está huérfano, o sea, no está asociado a una canción. Este comportamiento puede ser definido desde la misma relación con un segundo parámetro llamado cascade en la relación, y con la especificación de que ese parámetro puede ser "all, delete, delete-orphan", donde:

* all se aplica para hacer propagación de operaciones cuando se guarda o actualiza información. Por ejemplo, cuando se crea una canción con dos intérpretes, al almacenar la canción se deben almacenar los intérpretes también.
* delete se aplica al momento de borrar un objeto, de manera que los objetos relacionados también se borren. Por ejemplo, al borrar una canción se deben borrar sus intérpretes.
* delete-orphan se aplica al momento de desasociar un objeto relacionado, por ejemplo, cuando un intérprete deja de hacer parte de una canción, al guardar los cambios el intérprete debe ser borrado.

En el caso de la relación entre Canción e intérprete, se deben aplicar los tres elementos al parámetro cascade, por lo que la relación anterior se actualiza para dejarla como se aprecia en el siguiente ejemplo:

interpretes = relationship('Interprete', cascade='all, delete, delete-orphan')

La clase Cancion queda definida de la siguiente manera:

fromsqlalchemyimportColumn, Integer, String, ForeignKey

fromsqlalchemy.ormimportrelationship

from .declarative\_baseimportBase

classCancion(Base):

\_\_tablename\_\_ = 'cancion'

id = Column(Integer, primary\_key=True)

titulo = Column(String)

minutos = Column(Integer)

segundos = Column(Integer)

compositor = Column(String)

interpretes = relationship('Interprete', cascade='all, delete, delete-orphan')

SQLAlchemy establece que en la relación uno a muchos, en el lado de los muchos se especifique una llave foránea a quien lo referencia, por lo cual se adiciona una propiedad llamada cancion con un ForeignKey el lado del intérprete, como se muestra a continuación:

cancion = Column(Integer, ForeignKey('cancion.id'))

La clase queda de la siguiente manera:

fromsqlalchemyimportColumn, Integer, String, ForeignKey

from .declarative\_baseimportBase

classInterprete(Base):

\_\_tablename\_\_ = 'interprete'

id = Column(Integer, primary\_key=True)

nombre = Column(String)

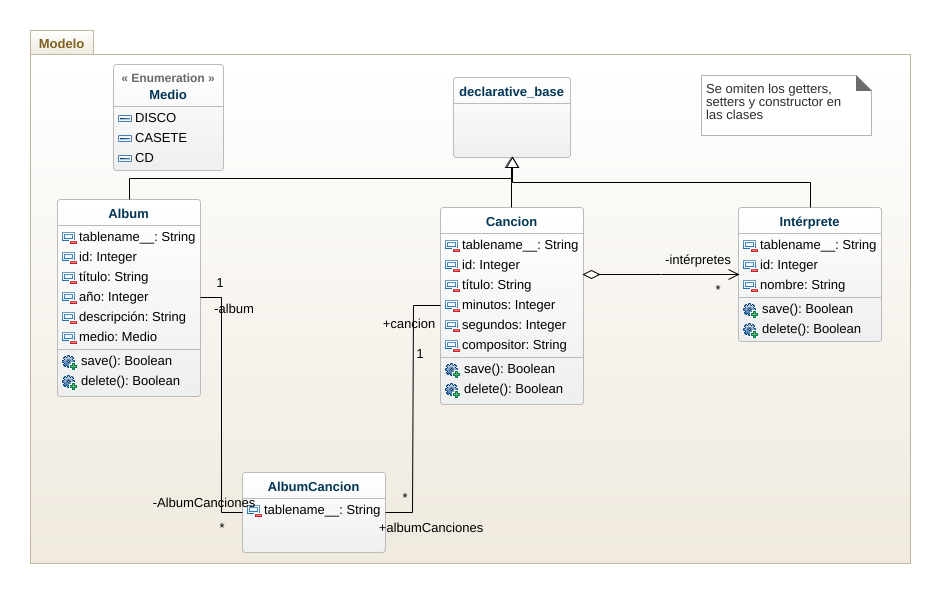
cancion = Column(Integer, ForeignKey('cancion.id'))

Para realizar relaciones con asociaciones compuestas, se pueden revisar las opciones de cascade que se encuentran en el siguiente enlace:

<https://docs.sqlalchemy.org/en/13/orm/cascades.html>

Segundo caso: Relación muchos a muchos

Para este caso, las clases Album y Canción tienen una relación de muchos a muchos entre ellas. Para esta relación se hace necesario modificar nuestro diagrama de clases e introducir una nueva clase que se llamará AlbumCancion, la cual tendrá la propiedad tablename, pero no contará con un identificador, de manera que el diagrama queda de la siguiente manera:



En el diagrama se puede observar que se cambia la relación muchos a muchos entre Album y Cancion por las siguientes relaciones:

* Una relación uno a uno entre AlbumCancion y Album y una relación uno a muchos entre Album y AlbumCancion
* Una relación uno a uno entre AlbumCancion y Cancion y una relación uno a muchos entre Cancion y AlbumCancion

Para modelar la relación uno a uno entre AlbumCancion y Album, AlbumCancion debe tener una referencia a Album, la cual se modela con una propiedad llamada album, que se crea como una columna que hace referencia al identificador de Album. El nombre de la propiedad debe ser el mismo que el de la relación en el diagrama de clases. El identificador del álbum es de tipo entero, por lo que la columna se crea como Integer, y, a continuación, se establece una llave foránea hacia dicho identificador con la instruccion ForeignKey(**'album.id'**), lo que indica que los valores asignados a Album están restringidos por las canciones existentes. La creación de la propiedad album queda definida de la siguiente manera:

album = Column(Integer, ForeignKey('album.id'))

Adicionalmente, en las librerías que se traen desde SQL Alchemy se debe importar ForeignKey de la siguiente manera:

fromsqlalchemyimportForeignKey

La clase AlbumCancion queda inicialmente definida de la siguiente manera:

fromsqlalchemyimportColumn, Integer, ForeignKey

from .declarative\_baseimportBase

classAlbumCancion(Base):

\_\_tablename\_\_ = 'album\_cancion'

album = Column(Integer, ForeignKey('album.id'))

Para modelar la relación uno a uno entre AlbumCancion y Cancion, se repiten los mismos pasos para crear la propiedad cancion. La clase AlbumCancion queda de la siguiente manera:

fromsqlalchemyimportColumn, Integer, ForeignKey

from .declarative\_baseimportBase

classAlbumCancion(Base):

\_\_tablename\_\_ = 'album\_cancion'

album = Column(Integer, ForeignKey('album.id'))

cancion = Column(Integer, ForeignKey('cancion.id'))

Sin embargo, SQL Alchemy requiere que por cada clase a "mapear" en la base de datos se defina una llave primaria, por lo cual, se establece que la llave primaria para esta clase son los atributos album y canción, de manera que esta clase queda definida de la siuguiente manera:

fromsqlalchemyimportColumn, Integer, ForeignKey

from .declarative\_baseimportBase

classAlbumCancion(Base):

\_\_tablename\_\_ = 'album\_cancion'

album = Column(Integer, ForeignKey('album.id'),primary\_key=True)

cancion = Column(Integer, ForeignKey('cancion.id'), primary\_key=True)

Dado que ya existen las relaciones desde AlbumCancion hacia Album y Cancion, se pueden modelar las relaciones uno a muchos desde Canción a AlbumCancion y desde Album a AlbumCancion.

Para la relación uno a muchos de Album a AlbumCancion, se utiliza una propiedad llamada canciones (Se conserva el nombre de la relación en el diagrama de clases), la cual se crea como una relación con la palabra relationship, y a continuación se especifica la clase con la cual se relaciona, y como enlace secundario el nombre de la propiedad tablename de AlbumCancion, como se muestra a continuación:

Canciones = relationship('cancion', secondary='album\_cancion')

Adicionalmente, en las librerías que se traen desde el ORM de SQL Alchemy se debe importar relationship de la siguiente manera:

fromsqlalchemy.ormimportrelationship

La clase Album queda de la siguiente manera:

importenum

fromsqlalchemyimportColumn, Integer, String, Enum

fromsqlalchemy.ormimportrelationship

from .declarative\_baseimportBase

classMedio(enum.Enum):

DISCO = 1

CASETE = 2

CD = 3

classAlbum(Base):

\_\_tablename\_\_ = 'album'

id = Column(Integer, primary\_key=True)

titulo = Column(String)

ano = Column(Integer)

descripcion = Column(String)

medio = Column(Enum(Medio))

canciones = relationship('Cancion', secondary='album\_cancion')

Para la relación uno a muchos de Cancion a AlbumCancion, se utiliza una propiedad llamada albumes, y se repite el mismo procedimiento que en el caso de la relación uno a muchos entre Album y AlbumCancion. La clase Cancion queda de la siguiente manera:

fromsqlalchemyimportColumn, Integer, String, ForeignKey

fromsqlalchemy.ormimportrelationship

from .declarative\_baseimportBase

classCancion(Base):

\_\_tablename\_\_ = 'cancion'

id = Column(Integer, primary\_key=True)

titulo = Column(String)

minutos = Column(Integer)

segundos = Column(Integer)

compositor = Column(String)

albumes = relationship('Album', secondary='album\_cancion')

interpretes = relationship('Interprete')

Para profundizar sobre cómo definir relaciones muchos a muchos se puede consultar la documentación del ORM de SQLAlchemy en el siguiente enlace:

<https://docs.sqlalchemy.org/en/13/orm/basic_relationships.html>

## 7. Ejercicios de práctica

En las secciones a continuación se propone realizar ejercicios prácticos sobre el modelo que se presentó y construyó anteriormente. Estos ejercicios suponen la implementación del diseño realizado en secciones anteriores, donde, la implementación del diseño se encuentra disponible en paquete Modelo del código en <https://github.com/MISW-4101-Practicas/TutorialCancionesSQLAlchemy>

Los ejercicios se proponen a manera de ejemplos y no se encuentran en el repositorio, con el propósito que se puedan utilizar a manera de práctica. Se propone que estos ejercicios sean creados como archivos dentro del mismo paquete Modelo y se ejecuten para ver su correcto funcionamiento. Los ejercicios de ejemplo son los siguientes:

* Ejemplo de almacenamiento
* Ejemplo de consultas (recuperación) de datos
* Ejemplo de actualización de datos
* Ejemplo de borrado de datos

8. Ejemplo de almacenamiento

Para almacenar la información en la base de datos se debe crear primero la base de datos, y luego se deben crear y adicionar los objetos a un objeto de la clase Session definida. Esta sesión será la encargada de transferir la información entre la base de datos y Python, y de mantener los objetos sin almacenar, actualizar o borrar hasta que se ejecute la operación commit().

En el siguiente archivo, llamado almacenar.py, se muestra un ejemplo de un programa ejecutable en Python donde se guardan datos para intérpretes y canciones de un álbum:

fromsrc.modelo.cancionimportCancion

fromsrc.modelo.interpreteimportInterprete

fromsrc.modelo.albumimportAlbum, Medio

fromsrc.modelo.declarative\_baseimportSession, engine, Base

if \_\_name\_\_ == '\_\_main\_\_':

*#Crea la BD*

Base.metadata.create\_all(engine)

*#Abre la sesión*

session = Session()

*#crear intérpretes*

interprete1 = Interprete(nombre = "Samuel Torres", texto\_curiosidades = "Es colombiano y vive en NY")

interprete2 = Interprete(nombre = "Aldo Gavilan", texto\_curiosidades = "Cantó a Cuba")

interprete3 = Interprete(nombre = "Buena Vista Social club")

interprete4 = Interprete(nombre = "Arturo Sandoval", texto\_curiosidades = "No sabía quien era")

session.add(interprete1)

session.add(interprete2)

session.add(interprete3)

session.add(interprete4)

session.commit()

*# Crear álbumes*

album1 = Album(titulo = "Latin Jazz Compilation", ano = 2021, descripcion = "Album original", medio = Medio.DISCO)

album2 = Album(titulo = "Bandas sonoras famosas", ano = 2021, descripcion = "Compilación", medio = Medio.DISCO)

session.add(album1)

session.add(album2)

*# Crear canciones*

cancion1 = Cancion(titulo = "Ajiaco", minutos = 3, segundos = 1, compositor = "Samuel Torres")

cancion2 = Cancion(titulo = "ForcedDisplacement", minutos = 3, segundos = 12, compositor = "Desconocido")

cancion3 = Cancion(titulo = "Alegría", minutos = 4, segundos = 27, compositor = "AU")

session.add(cancion1)

session.add(cancion2)

session.add(cancion3)

*# Relacionar albumes con canciones*

album1.canciones = [cancion1, cancion2]

album2.canciones = [cancion1, cancion3]

*# Relacionar canciones con intérpretes*

cancion1.interpretes = [interprete1]

cancion2.interpretes = [interprete2]

cancion3.interpretes = [interprete3, interprete4]

session.commit()

session.commit()

session.close()

Las primeras líneas permiten traer las definiciones de las clases establecidas, posteriormente se crea la base de datos y se abre una sesión con las siguientes líneas:

Base.metadata.create\_all(engine)

session = Session()

Posteriormente, en la sección "#Crear intérpretes" se crean los intérpretes, y se almacenan de inmediato. Este almacenamiento se hace en este punto para mostrar que la operación se puede realizar en cualquier momento. Se debe anotar que no se asignaron identificadores a la propiedad id de la clase, ya que al almacenarlos se generan los identificadores de los intérpretes. Para crear los intérpretes se crean los objetos con las siguientes instrucciones:

interprete1 = Interprete(nombre = "Samuel Torres", texto\_curiosidades = "Es colombiano y vive en NY")

interprete2 = Interprete(nombre = "Aldo Gavilan", texto\_curiosidades = "Cantó a Cuba")

interprete3 = Interprete(nombre = "Buena Vista Social club")

interprete4 = Interprete(nombre = "Arturo Sandoval", texto\_curiosidades = "No sabía quien era")

Para adicionarlos a la sesión de utilizan las siguientes instrucciones:

session.add(interprete1)

session.add(interprete2)

session.add(interprete3)

session.add(interprete4)

Para realizar el almacenamiento, se hace commit:

session.commit()

Al ejecutar la instrucción session.add, los intérpretes fueron provistos de un valor en su propiedad id, por lo que se puede utilizar este identificador para crear las relaciones con las canciones.

Luego, en la sección "#Crear álbumes" se crean los álbumes y se adicionan a la sesión para generar los identificadores que serán usados en las canciones. En este caso, se crearon los álbumes con sus propiedades, como se muestra a continuación.

album1 = Album(titulo = "Latin Jazz Compilation", ano = 2021, descripcion = "Album original", medio = Medio.DISCO)

album2 = Album(titulo = "Bandas sonoras famosas", ano = 2021, descripcion = "Compilación", medio = Medio.DISCO)

session.add(album1)

session.add(album2)

Por último, en la sección "*# crear canciones*", se crean las canciones con sus propiedades y se adicionan a la sesión, como se muestra a continuación:

cancion1 = Cancion(titulo = "Ajiaco", minutos = 3, segundos = 1, compositor = "Desconocido")

cancion2 = Cancion(titulo = "Ajiaco", minutos = 3, segundos = 1, compositor = "Desconocido")

cancion3 = Cancion(titulo = "Ajiaco", minutos = 3, segundos = 1, compositor = "Desconocido")

session.add(cancion1)

session.add(cancion2)

session.add(cancion3)

En este punto solamente se crearon los objetos, sin embargo, falta relacionar álbumes, canciones e intérpretes. Para estas relaciones se asignan los objetos en las propiedades relacionadas y estas relaciones se mantienen al ejecutar el almacenamiento.

En este ejemplo, las canciones se adicionan a los álbumes y los álbumes a las canciones, como si las relaciones entre álbum y canción recibieran una lista de objetos. A continuación se muestra la manera de relacionar álbumes y canciones:

album1.canciones=[cancion1, cancion2]

album2.canciones=[cancion1, cancion3]

cancion1.albumes=[album1, album2]

cancion2.albumes=[album1]

cancion3.albumes=[album2]

De igual forma se adicionan los intérpretes a las canciones:

Cancion1.interpretes = [interprete1]

Cancion2.interpretes = [interprete2]

Cancion3.interpretes = [interprete3, interprete4]

Para el almacenamiento se hace commit.

session.commit()

9. Ejemplo de recuperación de datos

Para recuperar la información en los objetos, se pueden hacer consultas sobre la base de datos, utilizando las funcionalidades de query, find y all sobre la sesión. En el siguiente archivo, llamado recuperar.py, muestra un ejemplo de recuperación de datos:

from src.modelo.cancion import Cancion

from src.modelo.interprete import Interprete

from src.modelo.album import Album, Medio

from src.modelo.declarative\_base import Session, engine, Base

if \_\_name\_\_ == '\_\_main\_\_':

session = Session()

canciones = session.query(Cancion).all()

print('Las canciones almacenadas son:')

for cancion in canciones:

print("Titulo: " + cancion.titulo + " (00:" +

str(cancion.minutos) + ":" +

str(cancion.segundos) + ")")

print("Intérpretes")

for interprete in cancion.interpretes:

print(" - " + interprete.nombre)

for album in cancion.albumes:

print(" -- Presente en el album: " + album.titulo)

print("")

print('Los álbumes almacenados en discos son:')

albumes = session.query(Album).filter(Album.medio == Medio.DISCO).all()

for album in albumes:

print("Album: " + album.titulo)

session.close()

En este ejemplo lo primero que se debe hacer es crear la sesión con la siguiente instrucción:

session = Session()

Posteriormente se consultan todas las canciones con la sentencia query y la opción all. En query se especifica la clase del objeto que se va a llenar con la consulta, y a continuación se adicionan las opciones de consulta, que en este caso, fue la de traer todos los registros. La instrucción quedó así:

canciones = session.query(Cancion).all()

A continuación se recorren todas las canciones y se muestra por pantalla su título, intérpretes y álbumes a los que pertenecen. Dado que la clase Canción tiene una relación de uno a muchos con la clase Intérprete, y muchos a muchos con álbumes, al recuperar las canciones, se recuperan también los intérpretes y los álbumes relacionados, por eso se pueden recorrer sin hacer llamados a la sesión, como se recorre en el ciclo:

print('Las canciones almacenadas son:')

for cancion in canciones:

print("Titulo: " + cancion.titulo + " (00:" +

str(cancion.minutos) + ":" +

str(cancion.segundos) + ")")

print("Intérpretes")

for interprete in cancion.interpretes:

print(" - " + interprete.nombre)

for album in cancion.albumes:

print(" -- Presente en el album: " + album.titulo)

print("")

Sin embargo, esta no es una búsqueda, esta es una recuperación de todos los datos de una clase. Para limitar las búsquedas, por ejemplo, para ver a cuáles álbumes vienen en el medio Disco, se puede hacer una búsqueda a partir del id del álbum. Para esto se ejecuta una consulta sobre Album, donde se limita con la sentencia filter aquel que cumpla con las condiciones de búsqueda, que en este caso, que el medio coincida con el elemento DISCO de la enumeración Medio. La consulta y el recorrido de los datos se ve de la siguiente manera:

print('Los álbumes almacenados en discos son:')

albumes = session.query(Album).filter(Album.medio == Medio.DISCO).all()

for album in albumes:

print("Album: " + album.titulo)

Al final, se cierra la sesión.

Ojoooooo: practicar con esto:

https://docs.sqlalchemy.org/en/13/orm/tutorial.html#building-a-relationship

query.filter(User.name.in\_(['ed', 'wendy', 'jack']))

# works with query objects too:

query.filter(User.name.in\_(

session.query(User.name).filter(User.name.like('%ed%'))

))

# use tuple\_() for composite (multi-column) queries

from sqlalchemy import tuple\_

query.filter(

tuple\_(User.name, User.nickname).\

in\_([('ed', 'edsnickname'), ('wendy', 'windy')])

)

[notin\_()](https://docs.sqlalchemy.org/en/13/core/sqlelement.html#sqlalchemy.sql.operators.ColumnOperators.notin_):

query.filter(~User.name.in\_(['ed', 'wendy', 'jack']))

* [ColumnOperators.is\_()](https://docs.sqlalchemy.org/en/13/core/sqlelement.html#sqlalchemy.sql.operators.ColumnOperators.is_):
* query.filter(User.name == **None**)

*# alternatively, if pep8/linters are a concern*

* query.filter(User.name.is\_(**None**))
* [ColumnOperators.isnot()](https://docs.sqlalchemy.org/en/13/core/sqlelement.html#sqlalchemy.sql.operators.ColumnOperators.isnot):
* query.filter(User.name != **None**)

*# alternatively, if pep8/linters are a concern*

query.filter(User.name.isnot(**None**))

* [AND](https://docs.sqlalchemy.org/en/13/core/sqlelement.html#sqlalchemy.sql.expression.and_):

*# use and\_()*

**from** **sqlalchemy** **import** and\_

query.filter(and\_(User.name == 'ed', User.fullname == 'Ed Jones'))

*# or send multiple expressions to .filter()*

query.filter(User.name == 'ed', User.fullname == 'Ed Jones')

*# or chain multiple filter()/filter\_by() calls*

query.filter(User.name == 'ed').filter(User.fullname == 'Ed Jones')

[Query.all()](https://docs.sqlalchemy.org/en/13/orm/query.html#sqlalchemy.orm.query.Query.all) returns a list:

>>> query = session.query(User).filter(User.name.like('*%e*d')).order\_by(User.id)

>>> query.all()

**SELECT** users.id **AS** users\_id,

users.name **AS** users\_name,

users.fullname **AS** users\_fullname,

users.nickname **AS** users\_nickname

**FROM** users

**WHERE** users.name **LIKE** ? **ORDER** **BY** users.id

('%ed',)

[<User(name='ed', fullname='Ed Jones', nickname='eddie')>,

<User(name='fred', fullname='Fred Flintstone', nickname='freddy')>]

### Counting

[Query](https://docs.sqlalchemy.org/en/13/orm/query.html#sqlalchemy.orm.query.Query) includes a convenience method for counting called [Query.count()](https://docs.sqlalchemy.org/en/13/orm/query.html" \l "sqlalchemy.orm.query.Query.count" \o "sqlalchemy.orm.query.Query.count):

>>> session.query(User).filter(User.name.like('*%e*d')).count()

**SELECT** **count**(\*) **AS** count\_1

**FROM** (**SELECT** users.id **AS** users\_id,

users.name **AS** users\_name,

users.fullname **AS** users\_fullname,

users.nickname **AS** users\_nickname

**FROM** users

**WHERE** users.name **LIKE** ?) **AS** anon\_1

('%ed',)

2

>>> **from** **sqlalchemy** **import** func

>>> session.query(func.count(User.name), User.name).group\_by(User.name).all()

**SELECT** **count**(users.name) **AS** count\_1, users.name **AS** users\_name

**FROM** users **GROUP** **BY** users.name

()

[(1, u'ed'), (1, u'fred'), (1, u'mary'), (1, u'wendy')]

To achieve our simple SELECT count(\*) FROM table, we can apply it as:

>>> session.query(func.count('\*')).select\_from(User).scalar()

**SELECT** **count**(?) **AS** count\_1

**FROM** users

('\*',)

4

The usage of [Query.select\_from()](https://docs.sqlalchemy.org/en/13/orm/query.html" \l "sqlalchemy.orm.query.Query.select_from" \o "sqlalchemy.orm.query.Query.select_from) can be removed if we express the count in terms of the User primary key directly:

>>> session.query(func.count(User.id)).scalar()

**SELECT** **count**(users.id) **AS** count\_1

**FROM** users

()

4

## Querying with Joins

>>> **for** u, a **in** session.query(User, Address).\

... filter(User.id==Address.user\_id).\

... filter(Address.email\_address=='jack@google.com').\

... all():

... print(u)

... print(a)

**SELECT** users.id **AS** users\_id,

users.name **AS** users\_name,

users.fullname **AS** users\_fullname,

users.nickname **AS** users\_nickname,

addresses.id **AS** addresses\_id,

addresses.email\_address **AS** addresses\_email\_address,

addresses.user\_id **AS** addresses\_user\_id

**FROM** users, addresses

**WHERE** users.id = addresses.user\_id

**AND** addresses.email\_address = ?

('jack@google.com',)

<User(name='jack', fullname='Jack Bean', nickname='gjffdd')>

<Address(email\_address='jack@google.com')>

O

The actual SQL JOIN syntax, on the other hand, is most easily achieved using the [Query.join()](https://docs.sqlalchemy.org/en/13/orm/query.html" \l "sqlalchemy.orm.query.Query.join" \o "sqlalchemy.orm.query.Query.join) method:

>>> session.query(User).join(Address).\

... filter(Address.email\_address=='jack@google.com').\

... all()

**SELECT** users.id **AS** users\_id,

users.name **AS** users\_name,

users.fullname **AS** users\_fullname,

users.nickname **AS** users\_nickname

**FROM** users **JOIN** addresses **ON** users.id = addresses.user\_id

**WHERE** addresses.email\_address = ?

('jack@google.com',)

[<User(name='jack', fullname='Jack Bean', nickname='gjffdd')>]

query = session.query(User, Address).select\_from(Address).join(User)

10. Ejemplo de actualización de datos

Los datos pueden ser actualizados a partir de la actualización de los objetos en Python. Para esto se pueden recuperar los objetos almacenados o los objetos que se tienen en edición y cambiar su información y sus relaciones. En el siguiente archivo, llamado actualizar.py, se muestra el cambio de datos de una canción y de la relación con un intérprete:

from src.modelo.cancion import Cancion

from src.modelo.interprete import Interprete

from src.modelo.declarative\_base import Session, engine, Base

if \_\_name\_\_ == '\_\_main\_\_':

session = Session()

cancion = session.query(Cancion).get(2)

interprete = session.query(Interprete).get(4)

cancion.minutos = 5

cancion.segundos = 30

cancion.compositor = "Pedro Pérez"

cancion.interpretes.append(interprete)

session.add(cancion)

session.commit()

session.close()

En este ejemplo, nuevamente el primer paso es crear la sesión:

session = Session()

Posteriormente se consultan los objetos que se necesitan para la modificación. En este caso se cargan desde la base de datos la canción con el id 2 utilizando el método get(2) sobre la consulta, y el intérprete creado con el id 4, utilizando el método get(4) sobre la consulta, como se muestra a continuación:

cancion = session.query(Cancion).get(3)

interprete = session.query(Interprete).get(4)

A continuación, se actualizan las propiedades de los objetos o las relaciones entre los objetos. Para este caso se cambian minutos, segundos y compositor de la canción, y se adiciona el intérprete de la canción (Se debe tener en cuenta que este intérprete antes hacía parte de la canción 3, por lo que, automáticamente, se desasocia de esta canción y pasa a relacionarse con esta canción), así:

cancion.minutos = 5

cancion.segundos = 30

cancion.compositor = "Pedro Pérez"

cancion.interpretes.append(interprete3)

Para que los cambios se puedan tener en cuenta en la transacción, se adiciona el objeto cancion3 en la sesión, de la misma manera en que se hizo cuando se iba a crear, como se muestra a continuación:

session.add(cancion)

Y al final, para fijar los cambios en la base de datos, se ejecuta la transacción, como se ve a continuación:

session.commit()

Luego se pueden ejecutar otras transacciones o se puede cerrar la sesión. Para verificar los cambios, se puede ejecutar de nuevo el ejemplo de recuperación en el apartado anterior.

## 11. Ejemplo de borrado de datos

Para borrar información almacenada, se deben enviar los objetos a borrar y luego ejecutar la transacción. El borrado es una operación que se debe ejecutar con cuidado, debido a que al eliminar objetos, se pueden borrar o no los objetos relacionados según la configuración de las relaciones, y pueden quedar objetos con relaciones inexistentes o se puedan presentar errores al momento del borrado.

En el siguiente ejemplo se muestra el borrado de una canción:

from src.modelo.cancion import Cancion

from src.modelo.interprete import Interprete

from src.modelo.album import Album, Medio

from src.modelo.declarative\_base import Session, engine, Base

if \_\_name\_\_ == '\_\_main\_\_':

session = Session()

cancion2 = session.query(Cancion).get(2)

session.delete(cancion2)

session.commit()

session.close()

En este ejemplo, nuevamente el primer paso es crear la sesión. Posteriormente se consultan los objetos que se necesitan para la modificación. En este caso se carga desde la base de datos la canción con el id 2 con el método get() que llama a los objetos con su identificador, como se muestra a continuación:

cancion2 = session.query(Cancion).get(2)

A continuación, se realiza el borrado de la cancion con el llamado que se muestra a continuación:

session.delete(cancion2)

Debido a que la canción 2 estaba relacionada con con el album 1, esta asociación desaparece de AlbumCancion. Adicionalmente la canción 2 estaba relacionada con dos intérpretes, al ser borrada, por las condiciones del cascade, los intérpretes fueron borrados automáticamente también.

Y al final, para fijar los cambios en la base de datos, se ejecuta la transacción, como se ve a continuación:

session.commit()

Luego se pueden ejecutar otras transacciones o se puede cerrar la sesión. Para verificar los cambios, se puede ejecutar de nuevo el ejemplo de recuperación en el apartado de ejemplo de recuperación de datos.

En el ejemplo, y según la definición de las clases, se debe tener cuidado con el borrado de intérpretes, dado que, al no estar relacionados a través de la instrucción relationship, es posible borrarlos y dejar las canciones con una relación a un intérprete inexistente. Para profundizar en las relaciones se recomienda revisar esta referencia.

<https://docs.sqlalchemy.org/en/13/orm/tutorial.html#building-a-relationship>