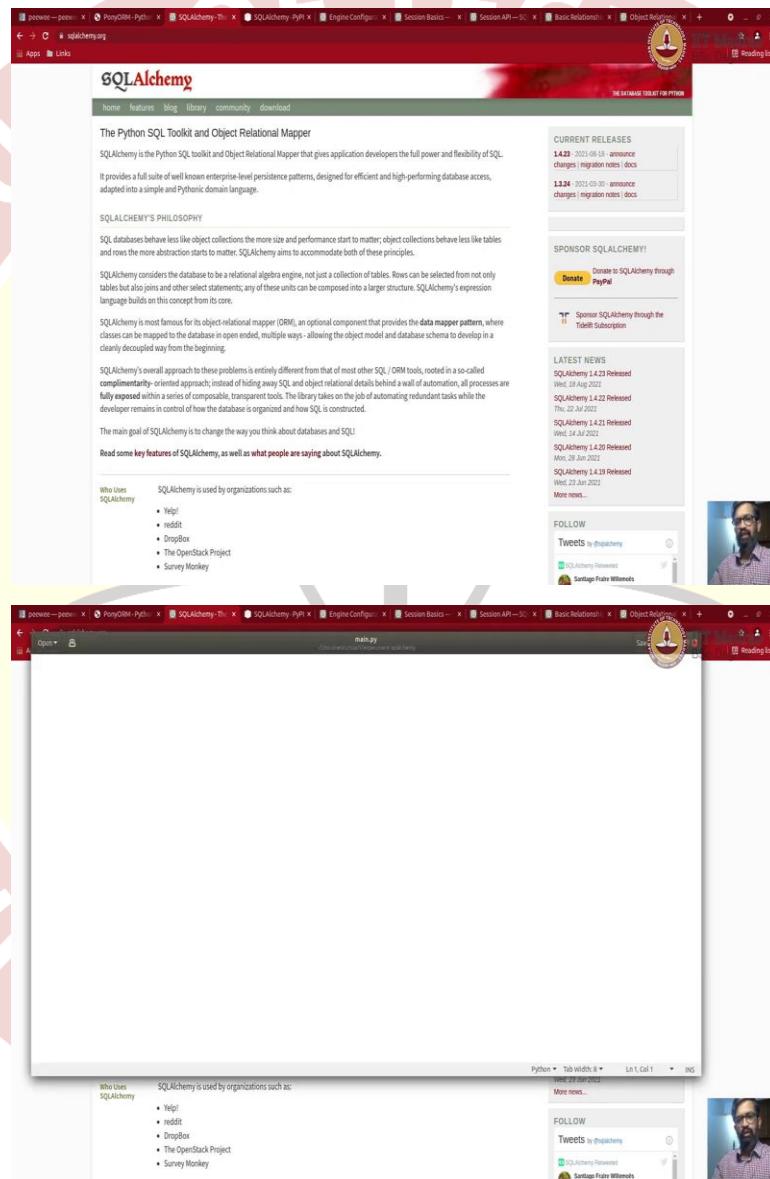


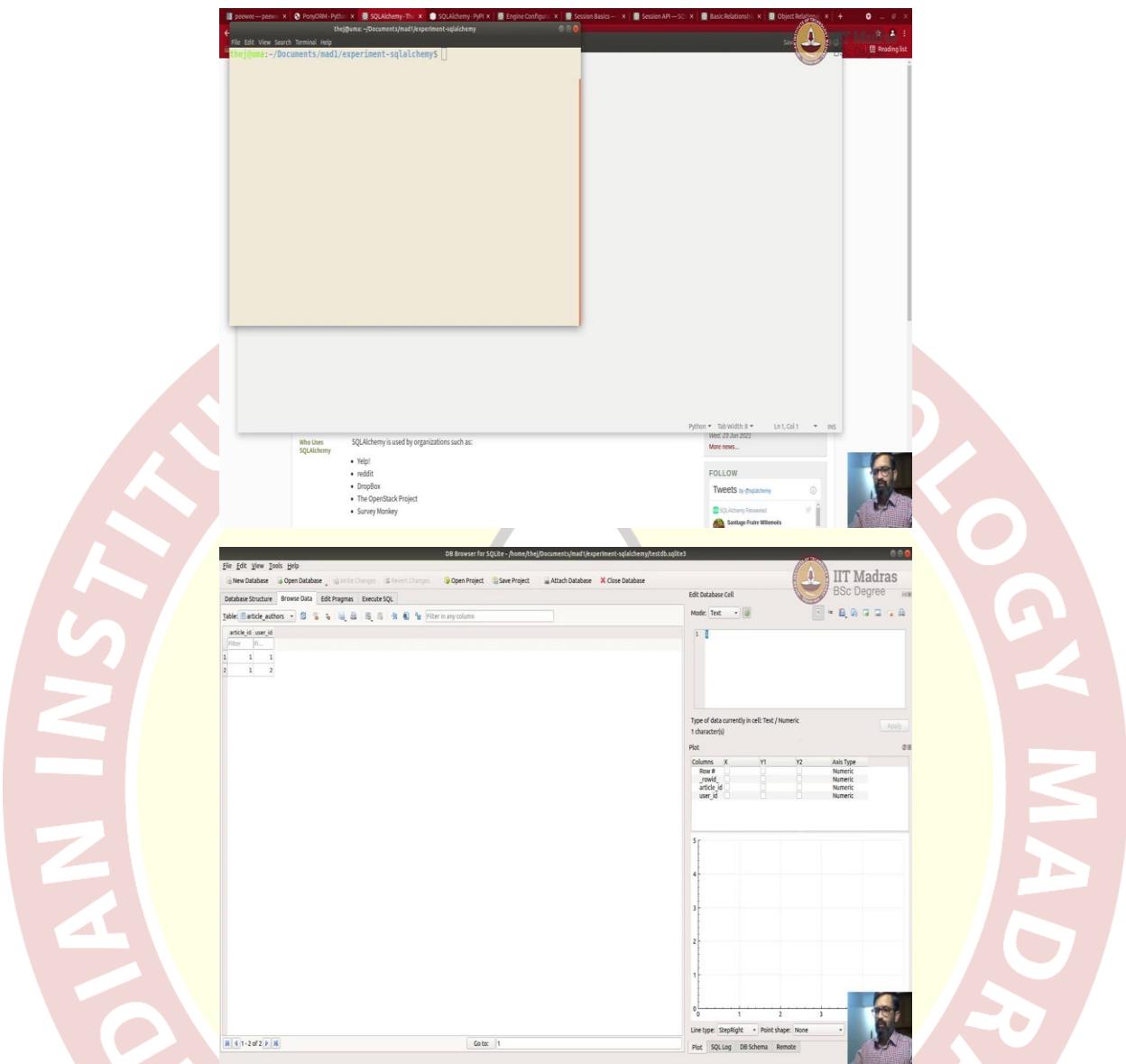
IIT Madras

ONLINE DEGREE

Modern application Development – I
Professor. Thejesh G N
Software Consultant
Indian Institute of Technology, Madras
How to Query Database using SQL Alchemy?

(Refer Slide Time: 0:15)



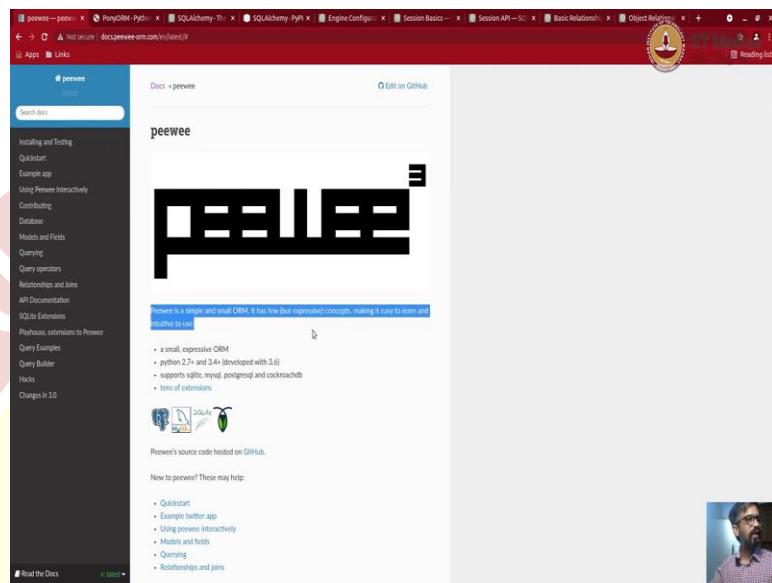


Welcome to the modern application development screencasts. In this short screencast, we will learn how to connect and query a database using SQL alchemy. Before we start, open the following applications on your Ubuntu desktop. So, that you can work along with me. Browser, preferably Firefox, Chromium or Brave is also fine, G edit text editor. Any other text editor would be fine. But G edit comes by default in your Ubuntu. Terminal or default terminal is fine. Please also make sure that you have Python 3 installed on your machine.

We would also need a DB browser, like we saw in the last SQL lite screencast to manipulate the database and work with it. In simple words, object relational mapping is a technique that allows the developer to query and manipulate their databases using object oriented paradigm. So,

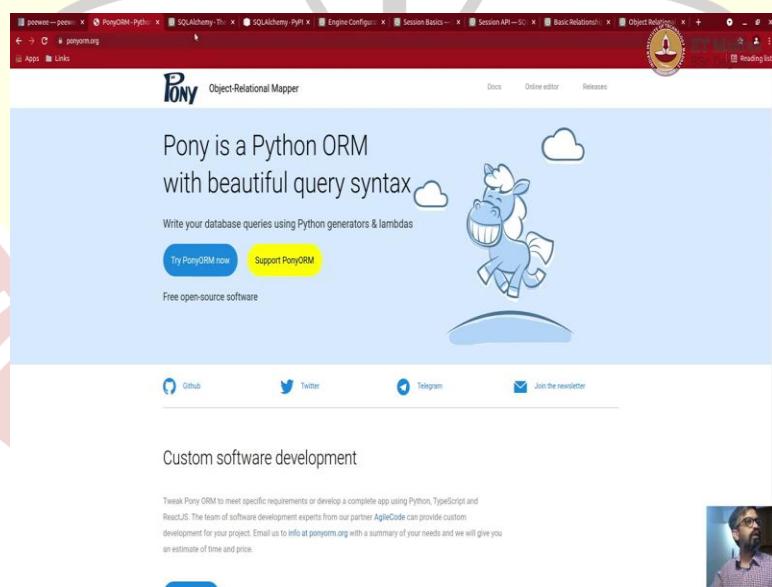
accessing the database becomes as easy as accessing classes and objects in Python. There are many toolkits available in Python world.

(Refer Slide Time: 1:36)



Some of the known ones are, for example PEEWEE, it is quite small and expressive.

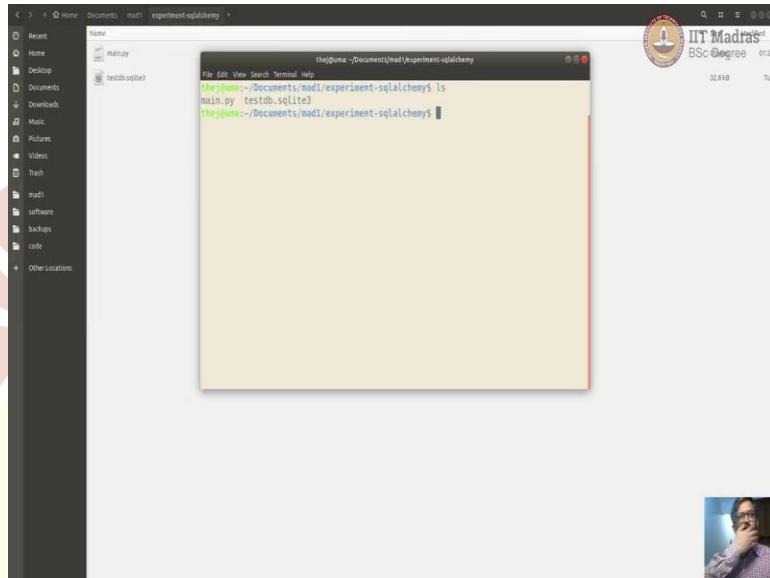
(Refer Slide Time: 1:37)



There is another one called Pony. And of course, there is an SQL alchemy. The most common one is SQL alchemy. SQL alchemy exposes a standard programming interface that is database agnostic. This helps us to write database agnostic code, which means we write code for one

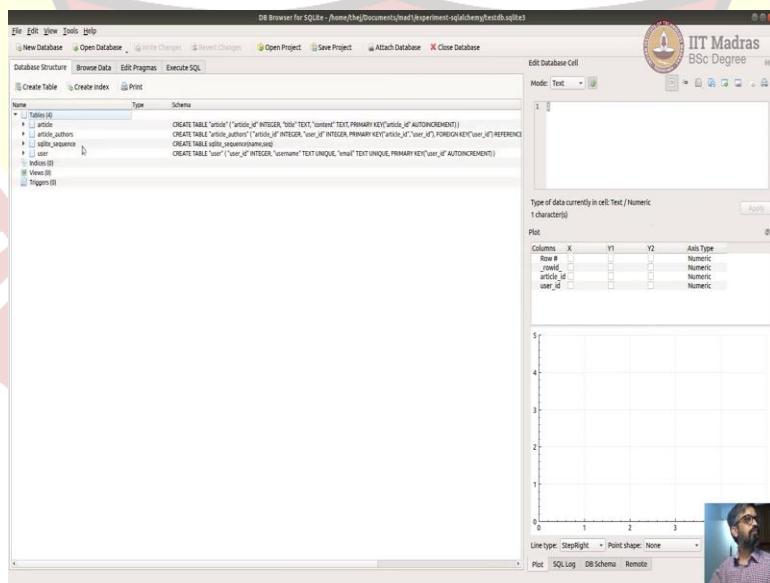
database, you should be able to change the database, and it should still work in most cases. Let us start with our experiment.

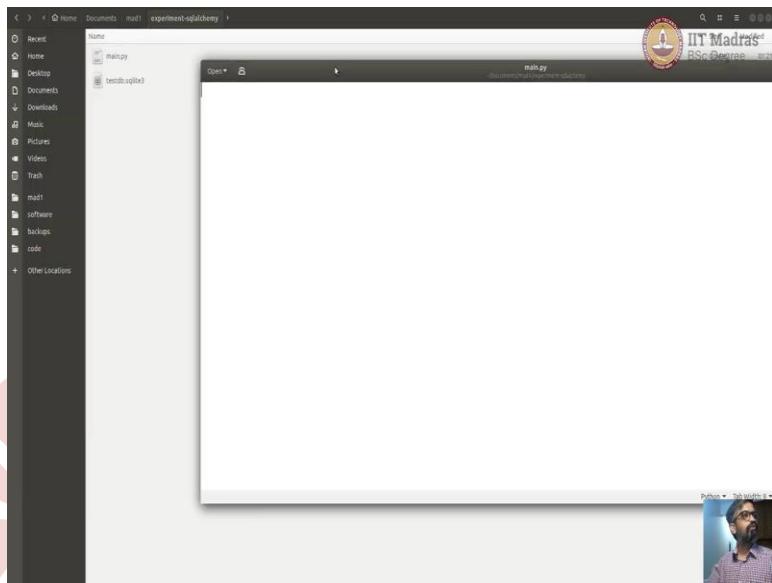
(Refer Slide Time: 2:11)



We have a folder called experiment-sqlalchemy. I have opened the command prompt, I have already seed it into that folder.

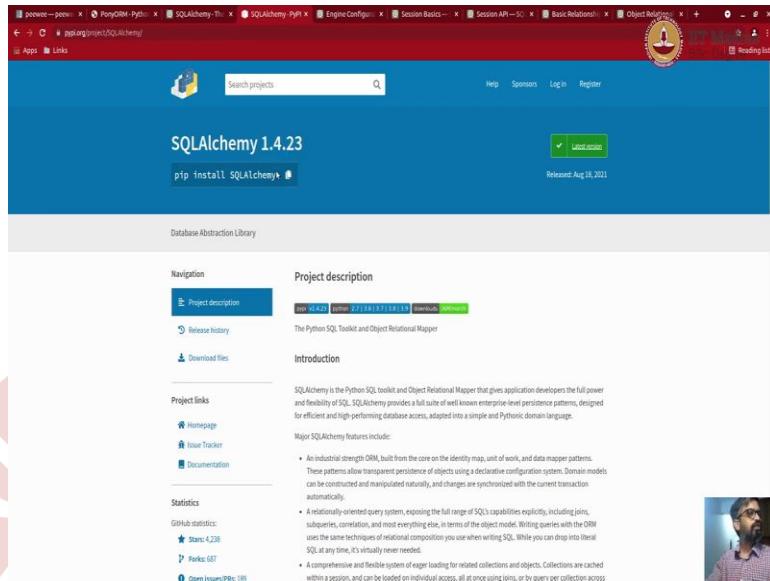
(Refer Slide Time: 2:29)





```
the@home:~/Documents/mad1/experiment-sqlalchemy$ ls
main.py  testdb.sqlite3
the@home:~/Documents/mad1/experiment-sqlalchemy$ python3 -m venv .experiment-sqlalchemy-env
the@home:~/Documents/mad1/experiment-sqlalchemy$
```



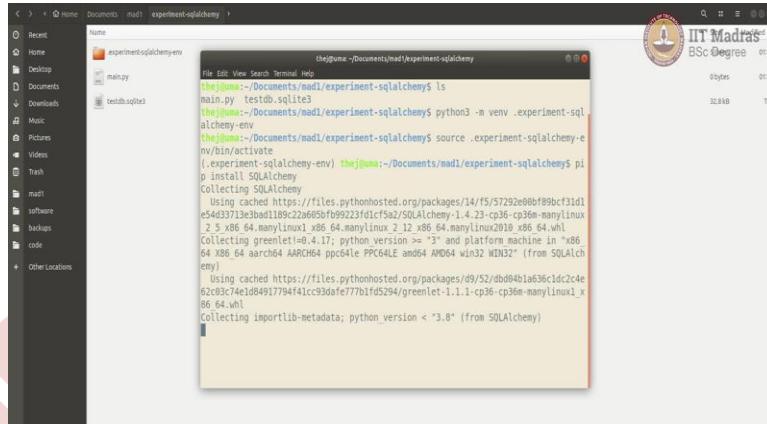


I now got the database from our last experiment opened. And in, in DB browser, you can see the same database and it is the same database here. And I have a main.py, which I have opened in, in a G edit which is empty. We are going to use this database that we created last time to experiment. We are going to use the same tables, that we created last time. If you are not seeing that, please watch it before you start this.

Let us start with the, creating a virtual environment. I am just going to paste the code to create the virtual environment. I am guessing you already know how to do this by now. It is already created. Now we need to install SQL alchemy. You can go to sqlalchemy.org see the latest version.

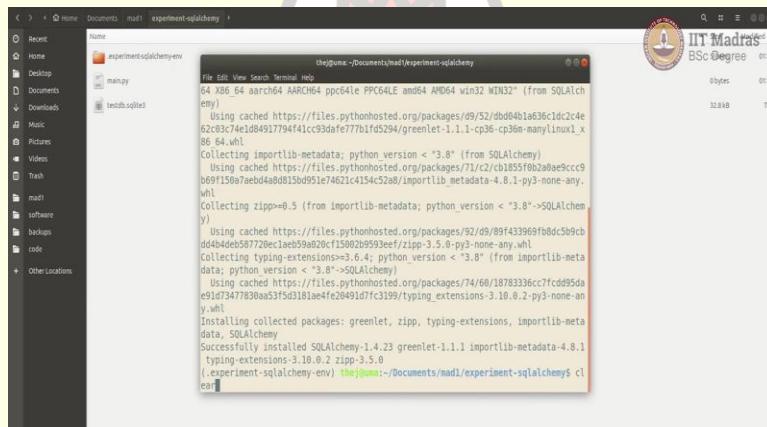
It is the latest version 1.4.23. You can also go to pypi site and see the latest version of the package. And here is the installation command. I think the latest version is 1.4.23, we will use the same thing. I am just going to copy this and install.

(Refer Slide Time: 3:42)



```
thejh@thejh-OptiPlex-5070:~/Documents/mdl/experiment-sqlalchemy$ pip install SQLAlchemy
Collecting SQLAlchemy
  Using cached https://files.pythonhosted.org/packages/14/f4/57292e80bf89bcf31d1e54d33713ebfb1189c226995fb99223fd1cf5a2/SQLAlchemy-1.4.23-py3-cp36-manylinux_2_5_x86_64_manylinux_x86_64_manylinux_2_12_x86_64_manylinux2019_x86_64.whl
Collecting greenlet<=0.4.17; python_version >= "3" and platform_machine in "x86_64_X86_64_aarch64 AARCH64 ppc64le PPC64LE amd64 AMD64 win32 WIN32" (from SQLAlchemy)
  Using cached https://files.pythonhosted.org/packages/d9/52/dbd04b1a636c1dc2c4e62c0374e1c084917794741cc93dafe777bf5294/greenlet-1.1.1-py3-cp36-manylinux1_x86_64.whl
Collecting importlib-metadata; python_version < "3.8" (from SQLAlchemy)

```



```
thejh@thejh-OptiPlex-5070:~/Documents/mdl/experiment-sqlalchemy$ pip install SQLAlchemy
Collecting SQLAlchemy
  Using cached https://files.pythonhosted.org/packages/14/f4/57292e80bf89bcf31d1e54d33713ebfb1189c226995fb99223fd1cf5a2/SQLAlchemy-1.4.23-py3-cp36-manylinux_2_5_x86_64_manylinux_x86_64_manylinux_2_12_x86_64_manylinux2019_x86_64.whl
Collecting greenlet<=0.4.17; python_version >= "3" and platform_machine in "x86_64_X86_64_aarch64 AARCH64 ppc64le PPC64LE amd64 AMD64 win32 WIN32" (from SQLAlchemy)
  Using cached https://files.pythonhosted.org/packages/d9/52/dbd04b1a636c1dc2c4e62c0374e1c084917794741cc93dafe777bf5294/greenlet-1.1.1-py3-cp36-manylinux1_x86_64.whl
Collecting importlib-metadata; python_version < "3.8" (from SQLAlchemy)

```



The image shows a computer screen with two windows open. The top window is a Python code editor showing a file named 'main.py'. The code imports SQLAlchemy and defines a base class 'Base'. The bottom window is 'DB Browser for SQLite' showing the schema of a database named 'testdb.sqlite'. It lists tables like 'article', 'article_authors', and 'article_text', and a sequence 'article_id_seq'. A plot window is also visible at the bottom right.

```
import sqlalchemy
from sqlalchemy import create_engine
from sqlalchemy import Table, Column, Integer, String, ForeignKey
from sqlalchemy import select

from sqlalchemy.orm import Session
from sqlalchemy.orm import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()
```

DB Browser for SQLite - /home/thehg/Documents/mad/experimentsqlalchemy/testdb.sqlite

File Edit View Tools Help

Tables (4)

- article
- article_authors
- article_text
- article_id_seq

Views (0)

Triggers (0)

Execute SQL

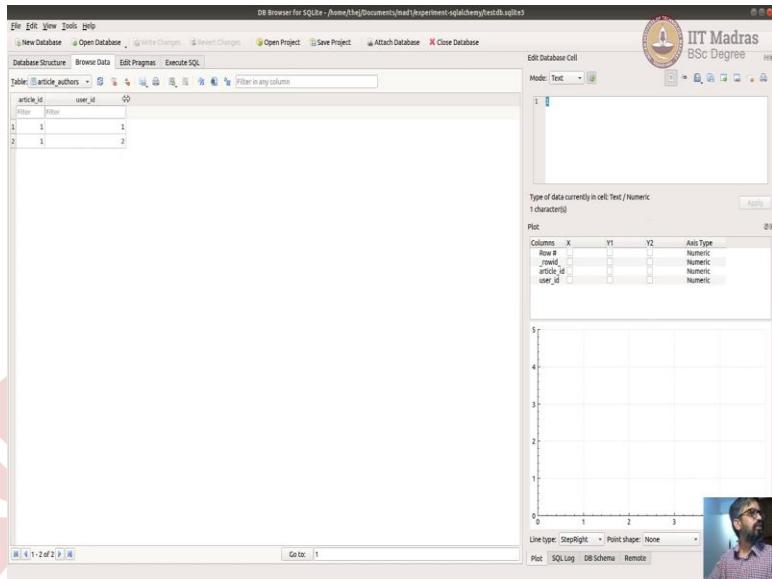
```
CREATE TABLE "article" ("id" INTEGER, "title" TEXT, "content" TEXT, PRIMARY KEY("id"))  
CREATE TABLE "article_authors" ("article_id" INTEGER, "user_id" INTEGER, PRIMARY KEY("article_id", "user_id"), FOREIGN KEY("article_id") REFERENCES article, FOREIGN KEY("user_id") REFERENCES user)  
CREATE TABLE "article_text" ("text" TEXT, "article_id" INTEGER, "user_id" INTEGER, PRIMARY KEY("text", "article_id", "user_id"), FOREIGN KEY("article_id") REFERENCES article, FOREIGN KEY("user_id") REFERENCES user)
```

Plot

Columns	X	Y1	Y2	Axis Type
Row #	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Numeric
rowid	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Numeric
article_id	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Numeric
user_id	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Numeric

Line type: StepRight • Point shape: None

Plot SQL Log DB Schema Remote



Before we install, we have to enable the virtual environment. So, I am just going to source it, bin activate. So, it is source now. Now I am going to install SQL alchemy, might take couple of minutes. Based on your internet connection installed. I am just going to clear, to install in our virtual environment. Now we are just going to edit the main point pi main.py and do our experimentation.

Before we start, I am just going to paste some of the imports. Instead of pasting one by one, I am just going to paste, some of them at once. Here I am just importing some of the classes that are required to do the experiment. Before we begin, actually we need to start writing the models for the tables that we have, that will be the first step. So, I am just going to get the base class for the model.

And write, models for all over tables. Like if you go to a database we have three tables, article, article_authors and user. Article is one which contains article, user contains all the user, article authors contains a link table, which is between the article table and the user table, which has the article_authors. It is a many to many relationship between article and user, that you can see here. article_id and user_id. Now let us create the models. I am just going to paste it and take you through it. Let us write for the user first.

(Refer Slide Time: 5:40)

The image shows two screenshots of DB Browser for SQLite. The top screenshot displays a Python script for SQLAlchemy mapping a 'User' class to a 'user' table. The bottom screenshot shows the database structure with tables 'article' and 'authors', and a data editor window.

DB Browser for SQLite - /home/thej/Documents/mad/experiment sqlalchemy/testdb.sqlite

```
import sqlalchemy
from sqlalchemy import create_engine
from sqlalchemy import Table, Column, Integer, String, ForeignKey
from sqlalchemy import select
from sqlalchemy.orm import Session
from sqlalchemy.orm import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    user_id = Column(Integer, autoincrement=True, primary_key=True)
    username = Column(String, unique=True)
    email = Column(String, unique=True)
```

Database Structure | Browse Data | Edit Pragmas | Execute SQL

Table: article_authors

article_id	user_id
1	1
2	1
2	2

File Edit View Tools Help

Open Database | Open Project | Save Project | Attach Database | Close Database

Edit Database Cell

Mode: Text

Type of data currently in cell: Text / Numeric
1 character(s)

Plot

Columns X Y1 Y2 Axis Type

Row #	x	y1	y2	Axis Type
1	1			Numeric
2	2			Numeric
3	3			Numeric
4	4			Numeric

Line type: StepRight | Point shape: None

Plot | SQL Log | DB Schema | Remote

DB Browser for SQLite - /home/thej/Documents/mad/experiment sqlalchemy/testdb.sqlite

```

File Edit View Tools Help
New Database Open Database
Database Structure Browse Data Edit Projects Execute SQL
Table: user
user_id username email
1 thejshgn @thejshgn.com
2 rag@example.com
from sqlalchemy import create_engine
from sqlalchemy import Table, Column, Integer, String, ForeignKey
from sqlalchemy import select
from sqlalchemy.orm import Session
from sqlalchemy.orm import declarative_base
from sqlalchemy.orm import relationship
Base = declarative_base()
class User(Base):
    tablename = 'user'
    user_id = Column(Integer, autoincrement=True, primary_key=True)
    username = Column(String, unique=True)
    email = Column(String, unique=True)

```

Python Tab width: 8 Ln 16, Col 27

Line type: StepRight Point shape: None

Plot SQL Log DB Schema Remote

1 1-1 of 2 | Go to: 1

Here this is the user model, it extends the base and then the table name is given here. This should be the same table name as in your database. Here this user in lower caps, the same thing here. This could be anything, the class name could be anything. I am just going to use capital or camel case user. And it has three columns.

As you can see user_id, username and email. Just going to use that same name as in the database. And then we are defining the column as an integer, auto increment is true and it is a primary key. Similarly, user name is column, it is a string or a text and the uniqueness is going to true. Similarly email, it is a string and uniqueness is going to true.

(Refer Slide Time: 6:36)

DB Browser for SQLite - /home/thej/Documents/mad/experiment sqlalchemy/testdb.sqlite

File Edit View Tools Help

New Database Open Database Write Changes Insert Changes Open Project Save Project Attach Database Close Database

Database Structure Browse Data Edit Projects Execute SQL

Create Table Create Index Modify Table Delete Table Print

Name Type Schema

Tables (0) Article (0) Article_Authors (0) User (0) Indexes (0) Modify Table (0) Views (0) Delete Table (0) Scripts (0) Copy Create Statement Export as CSV file

```

CREATE TABLE "article" ("article_id" INTEGER, "title" TEXT, "content" TEXT, PRIMARY KEY("article_id") AUTOINCREMENT())
CREATE TABLE "article_authors" ("article_id" INTEGER, "user_id" INTEGER, PRIMARY KEY("article_id", "user_id"), FOREIGN KEY("user_id") REFERENCES "user"("user_id"))
CREATE TABLE "user" ("user_id" INTEGER, "username" TEXT UNIQUE, "email" TEXT UNIQUE, PRIMARY KEY("user_id") AUTOINCREMENT())

```

Edit Database Cell Mode: Text

Type of data currently in cell: Text / Numeric 1 character(s)

Plot

Columns	X	Y1	Y2	Axes Type
rowid				Numeric
user_id				Numeric
username				Label
email				Label

Line type: StepRight Point shape: None

Plot SQL Log DB Schema Remote

DB Browser for SQLite - /home/the/Documents/ned/experiment sqlalchemy/testdb.sqlite

IIT Madras
BSc Degree

Database Structure

- Tables (0)
 - article
 - article_authors
 - article_sessions
 - user
- Views (0)
- Triggers (0)

Mode: Text

Table: user

Fields

Name	Type	NN	PK	AI	U	Default
user_id	INTEGER					
username	TEXT					
email	TEXT					

Constraints

```

CREATE TABLE "user" (
    "user_id" INTEGER,
    "username" TEXT UNIQUE,
    "email" TEXT UNIQUE
) PRIMARY KEY("user_id") AUTOINCREMENT;

```

Text Editor:

```

import sqlalchemy
from sqlalchemy import create_engine
from sqlalchemy import Table, Column, Integer, String, ForeignKey
from sqlalchemy import select
from sqlalchemy.orm import Session
from sqlalchemy.orm import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    user_id = Column(Integer, autoincrement=True, primary_key=True)
    username = Column(String, unique=True)
    email = Column(String, unique=True)

class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True, autoincrement=True)
    title = Column(String)
    content = Column(String)

```

Code Editor:

```

# coding: utf-8
# Create a database engine
engine = create_engine('sqlite:///testdb.sqlite')
# Create a base class for declarative models
Base = declarative_base()
# Create a session
Session = sessionmaker(bind=engine)
# Create a session object
session = Session()
# Create a User object
user = User(username='the', email='the@the.com')
# Add the User object to the session
session.add(user)
# Commit the transaction
session.commit()
# Query the User object from the session
user = session.query(User).filter_by(username='the').first()
# Print the User object
print(user)
# Close the session
session.close()

```

Output:

```

<User> object at 0x7f3e0d05a0>

```

You can double check it here, go to modify it, yeah, yup. This should match, user and user, so table name matches. User_id, auto increment true and primary key correct username is unique and email is unique, username is unique, ((06:56)) is unique, to create it for user. Now let us go ahead and create an article.

(Refer Slide Time: 07:15)

The screenshot shows a database management interface and a code editor side-by-side. On the left, a table definition for 'article' is being created with columns: article_id (integer, primary key, autoincrement), title (text), and content (text). On the right, a Python script named 'main.py' defines a 'User' class and an 'Article' class, both inheriting from 'Base'. The 'User' class has attributes: user_id (integer, primary key, autoincrement), username (string, unique), and email (string, unique). The 'Article' class has attributes: article_id (integer, primary key, autoincrement), title (string), and content (string).

```
sqlplus -f /home/teja/Documents/Insta/Experiment-sqlalchemy/testdb.sqlts
IIT Madras
BSc Degree
main.py
File Edit View Tools Help
New Database Open Database Write Changes Recent Changes Open Project Save Project Attach Database Close Database
Database Structure Create Table Modify Table Delete Table Print
Tables (4)
article
article_authors
article_sequences
user
Views (0)
Triggers (0)
import sqlalchemy
from sqlalchemy import create_engine
from sqlalchemy import Table, Column, Integer, String, ForeignKey
from sqlalchemy import select
from sqlalchemy.orm import Session
from sqlalchemy import declarative_base
from sqlalchemy import relationship
Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    user_id = Column(Integer, autoincrement=True, primary_key=True)
    username = Column(String, unique=True)
    email = Column(String, unique=True)

class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True, autoincrement=True)
    title = Column(String)
    content = Column(String)

DB Browser for SQLite - /home/teja/Documents/Insta/Experiment-sqlalchemy/testdb.sqlite
IIT Madras
BSc Degree
File Edit View Tools Help
New Database Open Database Write Changes Recent Changes Open Project Save Project Attach Database Close Database
Edit Database Cell
Database Structure Create Table Modify Table Delete Table Print
Tables (4)
article
article_authors
article_sequences
user
Views (0)
Triggers (0)
import sqlalchemy
from sqlalchemy import create_engine
from sqlalchemy import Table, Column, Integer, String, ForeignKey
from sqlalchemy import select
from sqlalchemy.orm import Session
from sqlalchemy import declarative_base
from sqlalchemy import relationship
Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    user_id = Column(Integer, autoincrement=True, primary_key=True)
    username = Column(String, unique=True)
    email = Column(String, unique=True)

class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True, autoincrement=True)
    title = Column(String)
    content = Column(String)
```

So, here is article, let us open this table details over here. So, here it is, you can see it has three columns article, title and content article title and content. This is integer and auto increment primary key, you can see it here. It is defined here and you can also see it here. Same thing we should define here. Now, the next we are going to define, this article authors.

(Refer Slide Time: 7:52)

```

  Open   B   mail.py
  import sqlalchemy
  from sqlalchemy import create_engine
  from sqlalchemy import Table, Column, Integer, String, ForeignKey
  from sqlalchemy import select

  from sqlalchemy.orm import Session
  from sqlalchemy.orm import declarative_base
  from sqlalchemy.orm import relationship

  Base = declarative_base()

  class User(Base):
      __tablename__ = 'user'
      user_id = Column(Integer, autoincrement=True, primary_key=True)
      username = Column(String, unique=True)
      email = Column(String, unique=True)

  class Article(Base):
      __tablename__ = 'article'
      article_id = Column(Integer, primary_key=True, autoincrement=True)
      title = Column(String)
      content = Column(String)

  class ArticleAuthors(Base):
      __tablename__ = 'article_authors'
      user_id = Column(Integer, ForeignKey("user.user_id"), primary_key=True, nullable=False)
      article_id = Column(Integer, ForeignKey("article.article_id"), primary_key=True, nullable=False)

```

article_author extends base table in this case, user_id is a foreign key to user.user_id this. And it is a primary key and nullable is going to false. Similarly, article id is a primary key to article table, here. And it is a primary key true, nullable is false. This is a composite key, because both are primary keys. Now this is done.

Now, we have to define many to many relationship between users and articles. Basically, whenever I get article, I want to get users too, so we can define many to many relationship.

(Refer Slide Time: 8:32)

docs.sqlalchemy.org/en/1.4/orm/basic_relationships.html

SQLAlchemy 1.4 Documentation
CURRENT RELEASE
Contents | Index | Download as ZIP file
Search term: search...

SQLAlchemy ORM
Object Relational Tutorial (Rx API)
Mapper Configuration
Relationship Configuration
Basic Relationship Patterns

- One To Many
 - Configuring Delete Behavior for One to Many
 - Many To One
 - One To One
 - Many To Many
 - Deleting Rows from the Many to Many Table
 - Association Object
 - Late Evaluation of Relationship Arguments
 - Adjacency List Relationships
 - Linking Relationships with Backref
 - Configuring how Relationship Joins
 - Collection Configuration and Techniques
 - Special Relationship Persistence Patterns
 - Relationships API
 - Querying Data Loading Objects
 - Using the Session
 - Events and Internals

One To Many

A one to many relationship places a foreign key on the child table referencing the parent. relationship() is then specified on the parent, as referencing a collection of items represented by the child:

```

class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship('Child')

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('parent.id'))

```

To establish a bidirectional relationship in one-to-many, where the "reverse" side is a many to one, specify an additional relationship() and connect the two using the relationship.back_populates parameter:

```

class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship('Child', back_populates='parent')

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary key=True)
    parent_id = Column(Integer, ForeignKey('parent.id'))
    parent = relationship('Parent', back_populates='children')

```

Child will get a parent attribute with many-to-one semantics.

Alternatively, the relationship.backref option may be used on a single relationship() instead of using relationship.back_populates:

```

class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary key=True)
    children = relationship('Child', backref='parent')

```

Configuring Delete Behavior for One to Many





Using foreign key ON DELETE cascade with ORM relationships

delete-orphan

Many To One

Many to one places a foreign key in the parent table referencing the child. `relationship()` is declared on the parent, where a new scalar-holding attribute will be created.

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child = Column(Integer, ForeignKey('child.id'))
    child = relationship('Child')

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
```

Bidirectional behavior is achieved by adding a second `relationship()` and applying the `relationship.back_populates` parameter in both directions:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship('Child', back_populates='parents')

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parents = relationship('Parent', back_populates='child')
```

Alternatively, the `relationship.backref` parameter may be applied to a single `relationship()`, such as `Parent.child`:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship('Child', backref='parents')
```



Many To Many

Many to many adds an association table between two classes. The association table is indicated by the `relationship.secondary` argument to `relationship()`. Usually, the Table uses the `Metadata` object associated with the declarative base class, so that the `ForeignKey` directives can locate the remote tables with which to link:

```
association_table = Table('association', Base.metadata,
    Column('left_id', ForeignKey('left.id')),
    Column('right_id', ForeignKey('right.id'))
)

class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship('Child',
        secondary='association_table')

class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
```

Tip

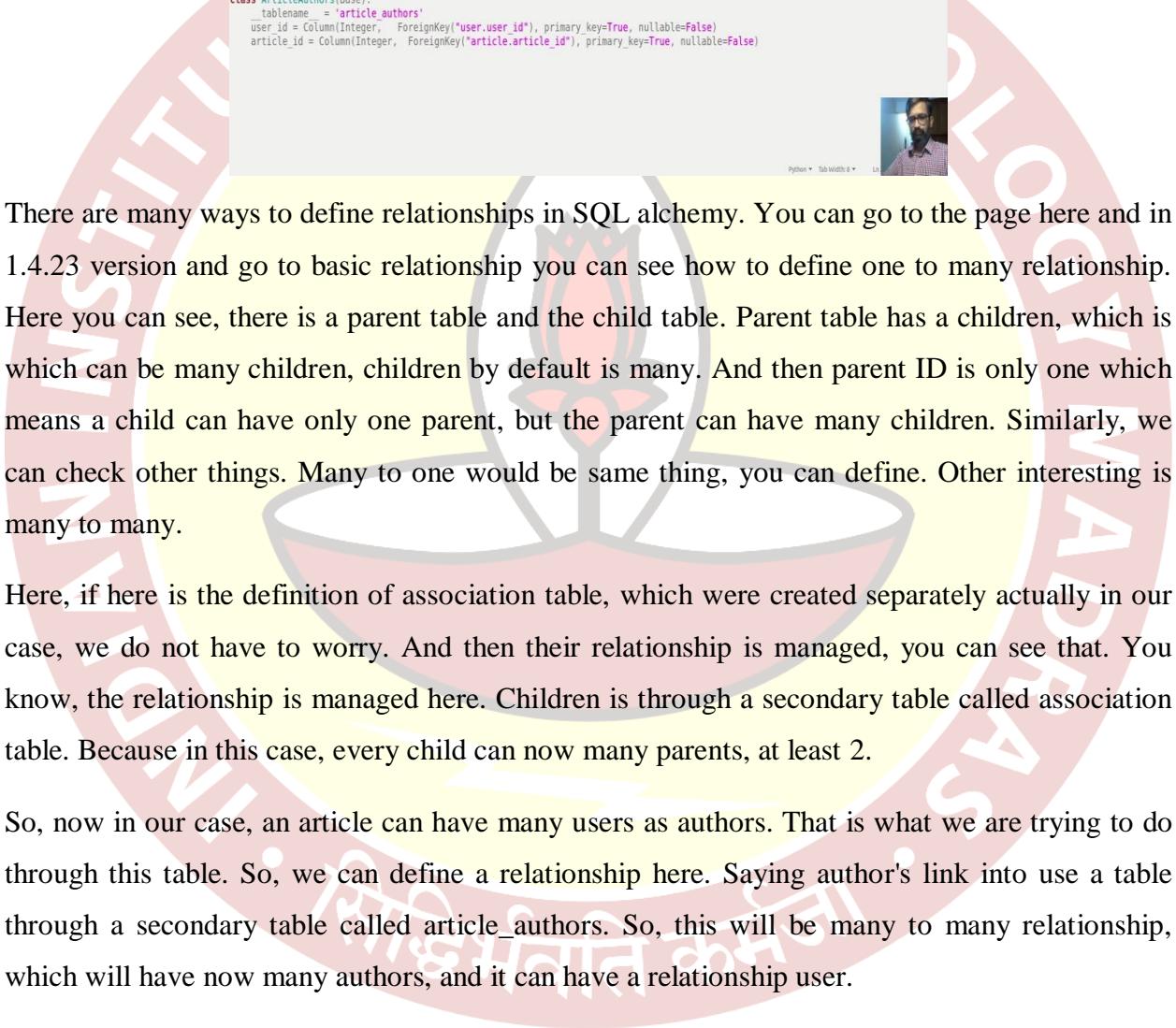
The "association table" above has foreign key constraints that refer to the `left` entity tables on either side of the relationship. The data type of each of `association.left_id` and `association.right_id` is normally inferred from that of the referenced table and may be omitted. It is also recommended, though not in any way required by SQLAlchemy, that the columns which refer to the two entity tables are established within either a `unique constraint` or more commonly as the `primary key constraint`; this ensures that duplicate rows won't be persisted within the table regardless of issues on the application side.

```
association_table = Table('association', Base.metadata,
    Column('left_id', ForeignKey('left.id'), primary_key=True),
    Column('right_id', ForeignKey('right.id'), primary_key=True)
)
```

For a bidirectional relationship, both sides of the relationship contain a collection. Specify using `relationship.back_populates`, and for each `relationship()` specify the common association table:

```
association_table = Table('association', Base.metadata,
```





```
Open 8 main.py - Document with local modifications IIT Madras BSc Degree
import sqlalchemy
from sqlalchemy import create_engine
from sqlalchemy import Table, Column, Integer, String, ForeignKey
from sqlalchemy import select

from sqlalchemy.orm import Session
from sqlalchemy.orm import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    user_id = Column(Integer, primary_key=True, autoincrement=True)
    username = Column(String, unique=True)
    email = Column(String, unique=True)

class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True, autoincrement=True)
    title = Column(String)
    content = Column(String)
    authors = relationship("User", secondary="article_authors")

class ArticleAuthors(Base):
    __tablename__ = 'article_authors'
    user_id = Column(Integer, ForeignKey("user.user_id"), primary_key=True, nullable=False)
    article_id = Column(Integer, ForeignKey("article.article_id"), primary_key=True, nullable=False)
```



There are many ways to define relationships in SQL alchemy. You can go to the page here and in 1.4.23 version and go to basic relationship you can see how to define one to many relationship. Here you can see, there is a parent table and the child table. Parent table has a children, which is which can be many children, children by default is many. And then parent ID is only one which means a child can have only one parent, but the parent can have many children. Similarly, we can check other things. Many to one would be same thing, you can define. Other interesting is many to many.

Here, if here is the definition of association table, which were created separately actually in our case, we do not have to worry. And then their relationship is managed, you can see that. You know, the relationship is managed here. Children is through a secondary table called association table. Because in this case, every child can now many parents, at least 2.

So, now in our case, an article can have many users as authors. That is what we are trying to do through this table. So, we can define a relationship here. Saying author's link into use a table through a secondary table called article_authors. So, this will be many to many relationship, which will have now many authors, and it can have a relationship user.

And through a secondary table, you can also similarly define, articles here. And this can link to this table. Basically, it will, it will link to all the users articles, link to article_authors. If you really want it, whenever you want to get user, you can also get articles. Similarly, now whenever you get article, you will get all the author, acceptor.

So, I am just going to remove this, because I do not want it to get all the articles related user all the time. Only this relationship I wanted to maintain. Because whenever I get an article, I also want to get authors. Now let us start our real experiment by connecting to a DB.

(Refer Slide Time: 11:19)

The screenshot shows the SQLAlchemy 1.4 Documentation page. The main content is the **Engine Configuration** section. It explains that the Engine is the starting point for any SQLAlchemy application, serving as the "base" for the actual database and its DBAPI. The section details the general structure and provides a diagram:

```

graph LR
    Engine[Engine] -- connects to --> Pool[Pool]
    Engine -- connects to --> Dialect[Dialect]
    Dialect -- connects to --> Database[Database]
  
```

Below the diagram, it states: "Where above, an Engine references both a Dialect and a Pool, which together interpret the DBAPI's module functions as well as the behavior of the database."

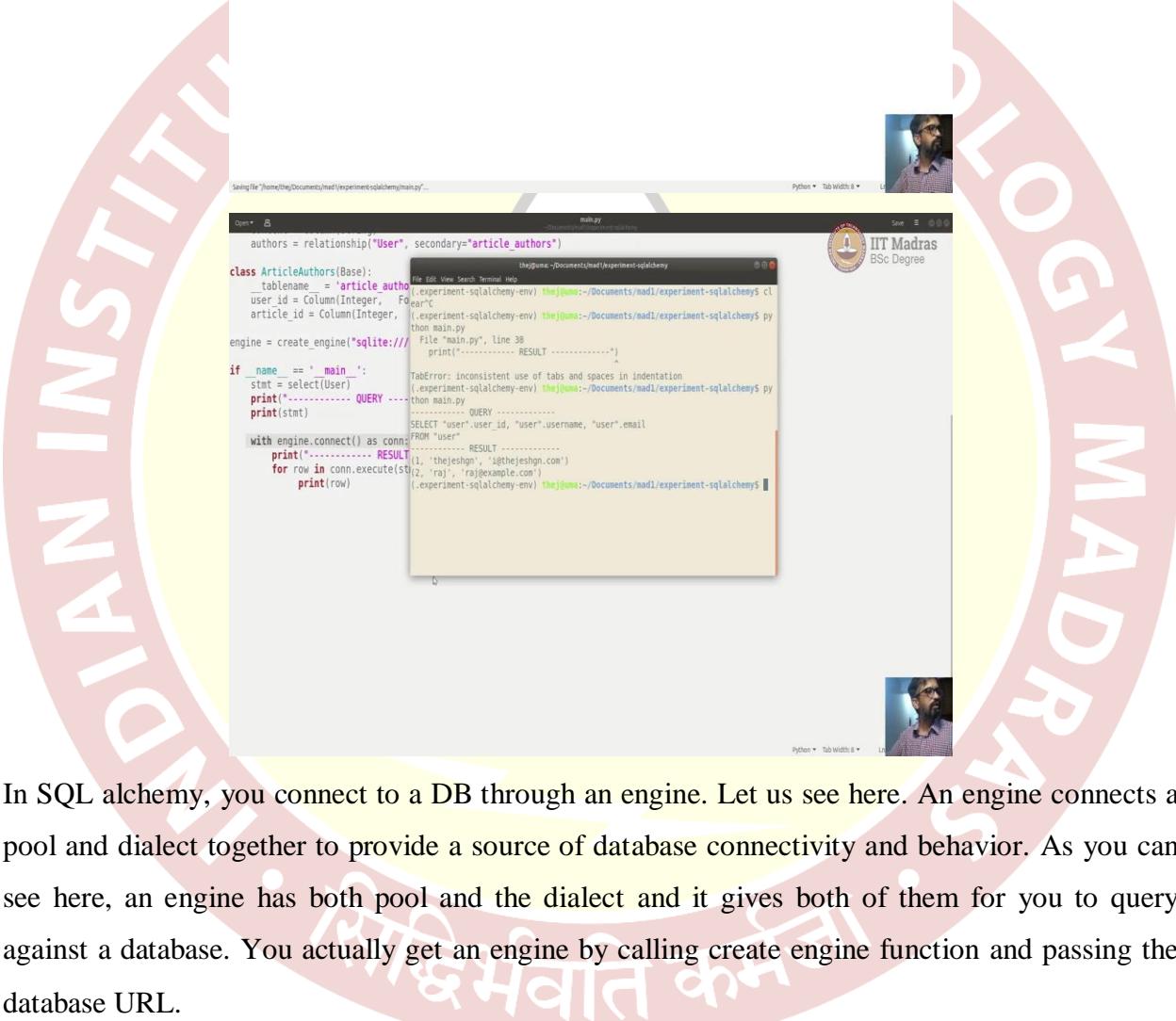
Code example for creating an engine:

```

from sqlalchemy import create_engine
engine = create_engine('postgresql://scott:tiger@localhost:5432/mydatabase')
  
```

The section concludes with a note about connection pooling and the creation of Session objects.

On the right side of the slide, there are two small profile pictures of a man with glasses and a beard.



In SQL alchemy, you connect to a DB through an engine. Let us see here. An engine connects a pool and dialect together to provide a source of database connectivity and behavior. As you can see here, an engine has both pool and the dialect and it gives both of them for you to query against a database. You actually get an engine by calling create engine function and passing the database URL.

Now the database URL will be of the format, this. It will be dialect+driver://username:password@host:port/database. In our case, we are going to use SQL alchemy, SQLite. So, there will not be any host, or port. It will be just part of the database. So, let us try to do that, create an engine, here.

So here, I have not given any driver. So, you do not see any plus pi my sqlite or any of the drivers. It uses, it is going to use the default driver, which is installed. So, I am just going to remove this. It will be sqlite://, this is relative part of the database. Since it is in the same folder, it will be ./ test sqlite3.

Now this engine uses, will give us a connection pool and dialect to talk to this database. Now we can run a connection, we can get a connection, and then a query. Let me show an example. Here, so here, you can see that I am creating a statement to run against a database, which is a select statement on user table. This is how you generally write the select, I am just printing it for our own reference, just want to make it clear. And then I am just going to print the...

So, here, I am just going to create a statement, I am just going to create a select statement from user table. And then we are going to print a statement. Then I am going to get a connection using engine or connect, it will give me a connection using this connection, I am going to execute this statement here. And I am just going to print the rows of the result.

We could, split it into multiple, statements, or you can use with combined statement here. And I am just going to run this, just going to run it again. You can see here, it is printing the statement SQL statement, which is a select * or select each column, from user here, this this statement. And then it is going to get a connection. And then run the query, get all.

(Refer Slide Time: 14:42)

The screenshot shows the DB Browser for SQLite interface. On the left, there is a table named 'user' with three columns: 'user_id', 'username', and 'email'. The data in the table is:

user_id	username	email
1	thejeshgn	@ejtego.com
2	raj	raj@example.com

A cursor is positioned at the end of the second row. On the right, there is a 'Plot' window showing a grid. Below the plot, there is a table titled 'Columns' with the following data:

Column	X	Y1	Y2	Axis Type
rowid				Numeric
user_id				Numeric
username				Label
email				Label

At the bottom right of the plot area, there is a small video camera icon showing a person's face.

Open... main.py

```

authors = relationship("User", secondary="article_authors")

class ArticleAuthors(Base):
    __tablename__ = 'article_authors'
    user_id = Column(Integer, ForeignKey("user.user_id"), primary_key=True, nullable=False)
    article_id = Column(Integer, ForeignKey("article.article_id"), primary_key=True, nullable=False)

engine = create_engine("sqlite:///./testdb.sqlite3")

if __name__ == '__main__':
    stmt = select([User])
    print("..... QUERY .....")
    print(stmt)

    with engine.connect() as conn:
        print("..... RESULT .....")
        for row in conn.execute(stmt):
            print(row)

```

Python Tab Width: 8 Line:

DB Browser for SQLite - /home/thejeeshp/.local/share/sqlitebrowser/testdb.sqlite

File Edit View Tools Help

New Database Open Database View Changes Recent Databases Open Project Save Project Attach Database Close Database

Database Structure Browser Data DB Pragmas Execute SQL

User

user_id	username	email
1	thejeeshp	thejeeshp.com
2	raj	raj@example.com

File Edit View Search Terminal Help (.experiment sqlalchemy-env) [thejeeshp:~/Documents/mad1/experiment-sqlalchemy\$ clea

car C (.experiment sqlalchemy-env) [thejeeshp:~/Documents/mad1/experiment-sqlalchemy\$ py

thon main.py

File "main.py", line 38

print("..... RESULT,")

TabError: inconsistent use of tabs and spaces in indentation (.experiment sqlalchemy-env) [thejeeshp:~/Documents/mad1/experiment-sqlalchemy\$ py

thon main.py

..... QUERY

SELECT "user".user_id, "user".username, "user".email

FROM "user"

..... RESULT

(1, 'thejeeshp', 'thejeeshp.com')

(2, 'raj', 'raj@example.com')

(.experiment sqlalchemy-env) [thejeeshp:~/Documents/mad1/experiment-sqlalchemy\$

Y1 Y2 Axis Type

Plot SQL Log DB Schema Remote

Line type: StepRight Point shape: None

Plot SQL Log DB Schema Remote

Since there are two rows in our database, in the article, which table are we trying to get? User table, in the user table there are two rows, we are actually printing two rows with all the columns. That, so you actually get a connection run a query. Now we could have made it simple, I mean, if you really wanted to separate it out, you can just connect, get a connection separately get all the rows separately and then iterate through it.

So, I just combined all of them, this is like an ideal way of doing it. Now, usually we do not try to get connection every time. We actually create a session and then use a session. The session establishes all the conversations with the database and represents a holding zone for all your objects, that you are loaded or associated with the ((0)(15:37) during the lifetime.

(Refer Slide Time: 15:40)



The image shows a screenshot of a computer desktop with two windows open. The top window is a web browser displaying the SQLAlchemy documentation for version 1.4.23, specifically the 'Session Basics' page. The bottom window is a terminal or code editor showing Python code for defining database models and interacting with a SQLite database.

SQLAlchemy Documentation - Session Basics

Session Basics

What does the Session do?

In the most general sense, the Session establishes all conversations with the database and represents a "holding zone" for all the objects which you've loaded or associated with it during its lifespan. It provides the interface where SELECT and other queries are made that will return and modify ORM-mapped objects. The ORM objects themselves are maintained inside the Session, inside a structure called the identity map - a data structure that maintains unique copies of each object, where "unique" means "only one object with a particular primary key".

The Session begins in a mostly stateless form. Once queries are issued or other objects are persisted with it, it requests a connection resource from an engine that is associated with the Session, and then establishes a transaction on that connection. This transaction remains in effect until the Session is instructed to commit or roll back the transaction.

The ORM objects maintained by a Session are instrumented such that whenever an attribute or a collection is modified in the Python program, a change event is generated which is recorded by the Session. Whenever the database is about to be queried, or when the transaction is about to be committed, the Session first flushes all pending changes stored in memory to the database. This is known as the unit of work pattern.

When using a Session, it's useful to consider the ORM mapped objects that it maintains as proxy objects to database rows, which are local to the transaction being run by the Session. In order to maintain consistency of the objects and their values when they're in the database, there are a variety of events that will cause objects to re-access the database in order to keep synchronized. It is possible to "detach" objects from a Session, and to continue using them; though this practice has its caveats. It's intended that usually, you'd re-associate detached objects with another Session when you want to work with them again, so that they can resume their normal task of representing database stats.

Basics of Using a Session

The most basic Session use patterns are presented here.

Opening and Closing a Session

Opening and Closing a Session

```
mail.py
username = Column(String, unique=True)
email = Column(String, unique=True)

class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True, autoincrement=True)
    title = Column(String)
    content = Column(String)
    authors = relationship("User", secondary="article_authors")

class ArticleAuthors(Base):
    __tablename__ = 'article_authors'
    user_id = Column(Integer, ForeignKey("user.user_id"), primary_key=True, nullable=False)
    article_id = Column(Integer, ForeignKey("article.article_id"), primary_key=True, nullable=False)

engine = create_engine("sqlite:///./testdb.sqlite3")

if name == '__main__':
    with Session(engine) as session:
        articles = session.query(Article).filter(Article.article_id == 1).all()
        for article in articles:
            print(article.title)
            for author in article.authors:
                print(author.username)
```



Let us go to the session, page and see the details here. When you take, take a little bit of time, and then go through it. There are many uses of sessions, we will go through a couple of them now. Let us run a query using a session. First you have to get a session. And you get a session using a session object or session class and passing the engine to it. Let me just paste a piece of code and show it to you. Here, I am getting a session by passing the engine which has connection pool and other things.

And then I am trying to get articles here, session gives a function called query and that you can use to query things here. Let me explain this, what is happening here. I am just querying the

```
Open  B main.py
username = Column(String, unique=True)
email = Column(String, unique=True)

class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True, autoincrement=True)
    title = Column(String)
    content = Column(String)
    authors = relationship("User", secondary="article_authors")

class ArticleAuthors(Base):
    __tablename__ = 'article_authors'
    user_id = Column(Integer, ForeignKey("user.user_id"), primary_key=True, nullable=False)
    article_id = Column(Integer, ForeignKey("article.article_id"), primary_key=True, nullable=False)

engine = create_engine("sqlite:///./testdb.sqlite3")

if __name__ == '__main__':
    with Session(engine) as session:
        articles = session.query(Article).filter(Article.article_id == 1).all()
        for article in articles:
            print(article.title)
            for author in article.authors:
                print("Author : {}".format(author.username))
```

```
Open  B main.py
username = Column(String, unique=True)
email = Column(String, unique=True)

class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True, autoincrement=True)
    title = Column(String)
    content = Column(String)
    authors = relationship("User", secondary="article_authors")

class ArticleAuthors(Base):
    __tablename__ = 'article_authors'
    user_id = Column(Integer, ForeignKey("user.user_id"), primary_key=True, nullable=False)
    article_id = Column(Integer, ForeignKey("article.article_id"), primary_key=True, nullable=False)

engine = create_engine("sqlite:///./testdb.sqlite3")

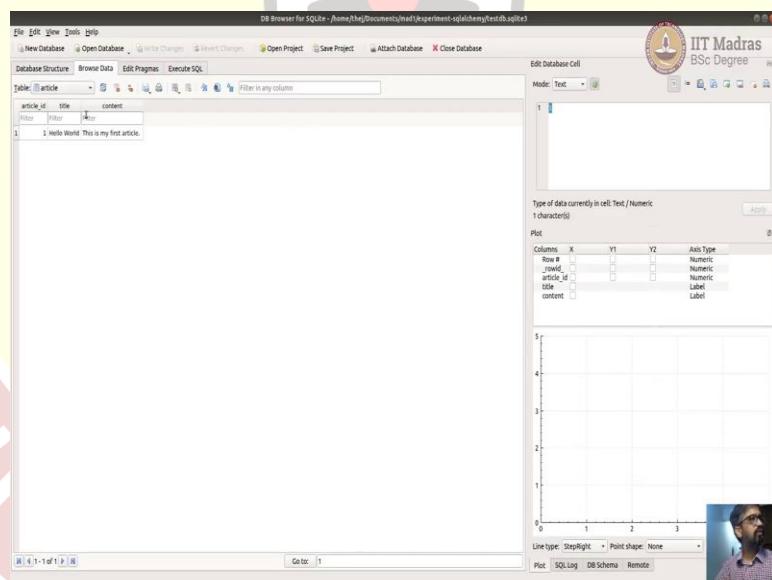
if __name__ == '__main__':
    with Session(engine) as session:
        articles = session.query(Article).filter(Article.article_id == 1).all()
        for article in articles:
            print("Article : {}".format(article.title))
            for author in article.authors:
                print("Author : {}".format(author.username))
```

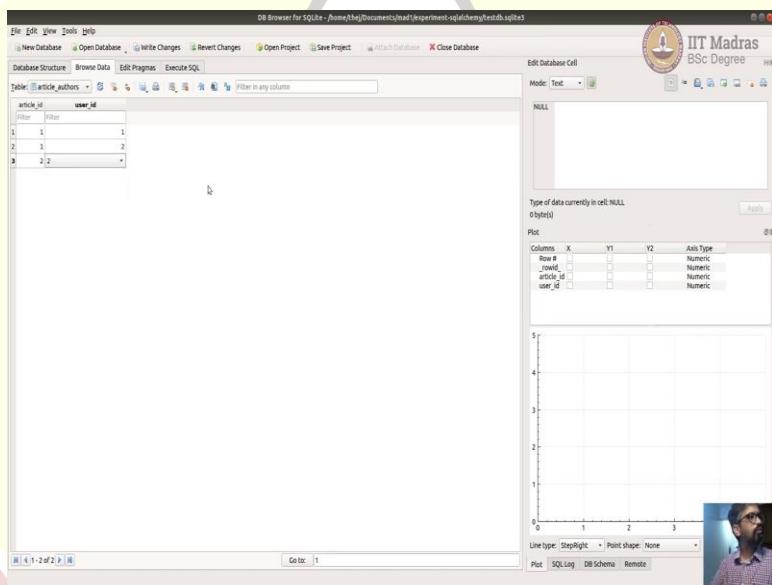
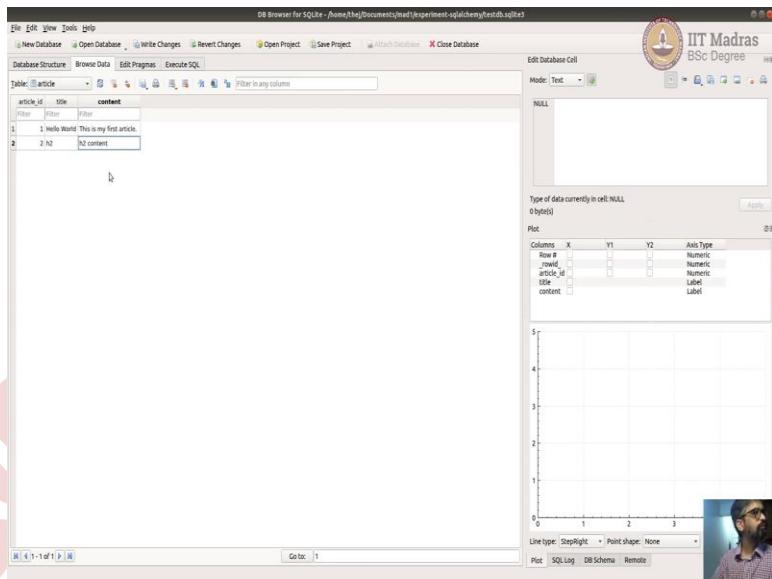
article table, then filtering the rows by article_id=1. And then getting all those filtered articles. And then I am iterating through those articles and printing the article title.

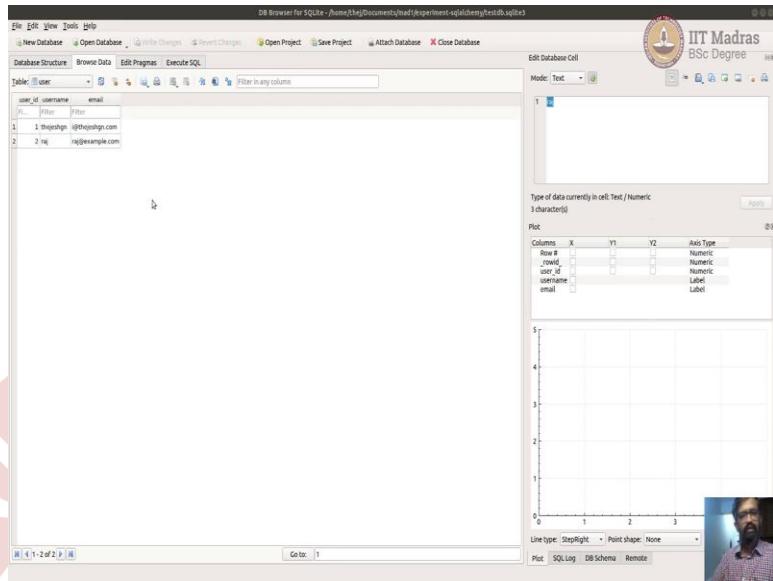
And then, for author, author in article_authors I am printing author name, like I explained authors is a relationship as soon as we get article in this case it will also get all the related articles using this relational table and load it into the array. And then I can directly access like an attribute in this article and print them. The user name is part of author so I am just printing the user name. Let me run this, just going to clear.

It is printing hello world, thejeshgn and raj. Hello world is a article title and you know there two authors. I am just going to make it clear by printing, in a better way. I am just going to make this also. For my 12, so that it is easy to read. I am just going to save and you can see that article is hello world author one is thejeshgn and author two is raj. Now you can also make it to article two.

(Refer Slide Time: 18:21)







Let us check if there is another article, there is no another article. If we insert an article let us add an article to h2, h2 content. I am going to save it and go to article author and I did not throw 2 the author is 2, on that will be Raj.

(Refer Slide Time: 18:56)

```

username = Column(String, unique=True)
email = Column(String, unique=True)

class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True)
    title = Column(String)
    content = Column(String)
    authors = relationship("User", secondary=articles)

class ArticleAuthors(Base):
    __tablename__ = 'article_authors'
    user_id = Column(Integer, ForeignKey(experiment_sqlalchemy.env))
    article_id = Column(Integer, ForeignKey(experiment_sqlalchemy.env))

engine = create_engine("sqlite:///testdb")
Session = sessionmaker(bind=engine)

if __name__ == '__main__':
    with Session() as session:
        articles = session.query(Article)
        for article in articles:
            print("Article : {} ".format(article.title))
            for author in article.authors:
                print("Author : {}".format(author.username))

```

The terminal output shows:

```

thejeshgn:~/Documents/mad1/experiment-sqlalchemy$ python main.py
Hello World
Author : thejeshgn

```



```
multi.py
username = Column(String, unique=True)
email = Column(String, unique=True)

class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True, autoincrement=True)
    title = Column(String)
    content = Column(String)
    authors = relationship("User", secondary="article_authors")

class ArticleAuthors(Base):
    __tablename__ = 'article_authors'
    user_id = Column(Integer, ForeignKey("user.user_id"), primary_key=True, nullable=False)
    article_id = Column(Integer, ForeignKey("article.article_id"), primary_key=True, nullable=False)

engine = create_engine('sqlite:///testdb.sqlite3')

if __name__ == '__main__':
    with Session(engine) as session:
        articles = session.query(Article).filter(Article.article_id == 2).all()
        for article in articles:
            print("Article : {} ".format(article.title))
            for author in article.authors:
                print("Author : {}".format(author.username))

 SQLAlchemy 1.4 Documentation
 CURRENT RELEASE
 Contents (index) Download as ZIP file
 Search terms: search.

# alternatively, if pep8/linters are a concern
query.filter(User.name.is_(None))

• 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')

Note
Make sure you use and_() and not the Python and operator!

• OR:
from sqlalchemy import or_
query.filter(or_(User.name == 'ed', User.name == 'wendy'))

Note
Make sure you use or_() and not the Python or operator!

• ColumnOperators.match():
query.filter(User.name.match('wendy'))

Note
ColumnOperators.match() uses a database-specific match or contains function; its behavior will vary by backend and is not available on some backends such as SQLite.
```

```

>>> session.add(jack)
>>> session.commit()

Querying for Jack, we get just Jack back. No SQL is yet issued for Jack's addresses:

>>> jack = session.query(User).filter_by_name('jack').one()
>>> jack
<User(name='jack', fullname='Jack Bean', nickname='gijffdd')>

Let's look at the addresses collection. Watch the SQL:

>>> jack.addresses
[Address(email_address='jack@google.com'), <Address(email_address='j2@yahoo.com')>]

When we accessed the addresses collection, SQL was suddenly issued. This is an example of a lazy loading relationship. The addresses collection is now loaded and behaves just like an ordinary list. We'll cover ways to optimize the loading of this collection in a bit.

Querying with Joins

Now that we have two tables, we can show some more features of query, specifically how to create queries that deal with both tables at the same time. The Wikipedia page on SQL JOIN offers a good introduction to join techniques, several of which we'll illustrate here.

To construct a simple implicit join between user and Address, we can use query.filter() to equate their related columns together. Below we load the user and Address entities at once using this method:

>>> for u, a in session.query(User, Address).filter(Address.id==Address.user_id).filter(Address.email_address=='jack@google.com').all():
...     print(u)
...     print(a)
...     print(u)
...     print(a)
...     print(u)
...     print(a)

The actual SQL JOIN syntax, on the other hand, is most easily achieved using the query.join() method:

>>> session.query(User).join(Address).filter(Address.email_address=='jack@google.com').all()

```

So, now let us go back here, change it to 2 and then run it again. Now it changed article h2 author is Raj. And you know there are many functions and many way to query. We can go to querying part and see many of the operators available and you know how do you do join in and left join, right join, inner join and outer join etcetera on this thing and it is straightforward.

For example, let us see if there is an example here. So, here is where actually we are filtering by multiple conditions using an and. You are importing an and, then making sure both user name and user full name is this. This of ((19:53) and. And if you want to get only one record you can see dot one, etcetera.

(Refer Slide Time: 20:04)

```

Querying with Joins

Now that we have two tables, we can show some more features of query, specifically how to create queries that deal with both tables at the same time. The Wikipedia page on SQL JOIN offers a good introduction to join techniques, several of which we'll illustrate here.

To construct a simple implicit join between user and Address, we can use query.filter() to equate their related columns together. Below we load the user and Address entities at once using this method:

>>> for u, a in session.query(User, Address).filter(Address.id==Address.user_id).filter(Address.email_address=='jack@google.com').all():
...     print(u)
...     print(a)
...     print(u)
...     print(a)
...     print(u)
...     print(a)

The actual SQL JOIN syntax, on the other hand, is most easily achieved using the query.join() method:

>>> session.query(User).join(Address).filter(Address.email_address=='jack@google.com').all()
[User(name='jack', fullname='Jack Bean', nickname='gijffdd')]

query.join() knows how to join between user and Address because there's only one foreign key between them. If there were no foreign keys, or several, query.join() works better when one of the following forms are used:

query.join(Address, User.id==Address.user_id) # explicit condition
query.join(Address, User.addresses) # implied relationship from left to right
query.join(Address, User.addresses) # same, with explicit target
query.join(User.addresses, and_(Address.name != 'foo')) # use relationship + additional ON criteria

As you would expect, the same idea is used for "outer" joins, using the query.outerjoin() function.

query.outerjoin(User.addresses) # LEFT OUTER JOIN

The reference documentation for query.join() contains detailed information and examples of the calling styles accepted by this method. query.join() is an important method at the center of usage for any SQL-fluent application.

What does query select from if there's multiple entities?

```

And then there are joints. And you can use the joints to query as well. For example, here, they are joining user table and address table, using ID is across them and then filtering by email address of the user. This is how we write joins, there are many other options available, I think you should just explore this page on your own, and learn about various ways of querying.

(Refer Slide Time: 20:28)



The image shows two screenshots of a Python code editor. The top screenshot displays a file named `main.py` containing SQLAlchemy code for creating a database and adding an article. The bottom screenshot shows the same code running in a terminal window, with the output of the script visible.

```

class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True, autoincrement=True)
    title = Column(String)
    content = Column(String)
    authors = relationship("User", secondary="article_authors")

class ArticleAuthors(Base):
    __tablename__ = 'article_authors'
    user_id = Column(Integer, ForeignKey("user.user_id"), primary_key=True, nullable=False)
    article_id = Column(Integer, ForeignKey("article.article_id"), primary_key=True, nullable=False)

engine = create_engine("sqlite:///./testdb.sqlite3")

if __name__ == '__main__':
    with Session(engine, autoflush=False) as session:
        session.begin()
        try:
            article = Article(title="my new article", content="my new article content")
            session.add(article)
            session.flush()
            print("... get article id ....")
            print(article.article_id)
            article_authors = ArticleAuthors(user_id=1, article_id=article.article_id)
            session.add(article_authors)
        except:
            print("Rolling back")
            session.rollback()
            raise
        else:
            print("Commit")
            session.commit()

class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True)
    title = Column(String)
    content = Column(String)
    authors = relationship("User", secondary="article_authors")

class ArticleAuthors(Base):
    __tablename__ = 'article_authors'
    user_id = Column(Integer, ForeignKey("user.user_id"))
    article_id = Column(Integer, ForeignKey("article.article_id"))

engine = create_engine("sqlite:///./testdb")

if __name__ == '__main__':
    with Session(engine, autoflush=False) as session:
        session.begin()
        try:
            article = Article(title="my ne
            session.add(article)
            session.flush()
            print("... get article id ..")
            print(article.article_id)
            article_authors = ArticleAuthors(user_id=1, article_id=article.article_id)
            session.add(article_authors)
        except:
            print("Rolling back")
            session.rollback()
            raise
        else:
            print("Commit")
            session.commit()

```

```

class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True, autoincrement=True)
    title = Column(String)
    content = Column(String)
    authors = relationship("User", secondary="article_authors")

class ArticleAuthors(Base):
    __tablename__ = 'article_authors'
    user_id = Column(Integer, ForeignKey("user.user_id"), primary_key=True, nullable=False)
    article_id = Column(Integer, ForeignKey("article.article_id"), primary_key=True, nullable=False)

engine = create_engine("sqlite:///./testdb.sqlite3")

if __name__ == '__main__':
    with Session(engine, autoflush=False) as session:
        session.begin()
        try:
            article = Article(title="my new article", content="my new article content")
            session.add(article)
            session.flush()
            print(".... get article_id ....")
            print(article.article_id)
            raise Exception("Dummy Error")
            article_authors = ArticleAuthors(user_id=1, article_id=article.article_id)
            session.add(article_authors)
        except:
            print("Rolling back")
            session.rollback()
            raise
        else:
            print("Commit")
            session.commit()

```



Now, there is another use of session that is you can use session you can create transactions. Transactions is a unit of work, you want to either completely be done, or you want to completely roll back, you do not want a piece of work to commit into database. Let us say you have 3 or 4 steps to do. In our case, we have two steps, first we say the article, and then say we say the authors of the article.

If, after saving the article, something happens on article authors does not get saved, then we will have this random article with no authors. We do not want that to happen. So, ideally, we want both queries to and both the inserts to happen, or nothing to happen. Let us see a general case of and then by error case handling of it. I am just going to write a piece of code here. So, here I am creating a session with autoflush=false.

Autoflush in SQL alchemy is a way to send the commands to a database and make it execute. And then here, I am making it false by default, so we can manually call it by default it is auto. I am creating a session, then I am actually creating article and adding that article to the session, which is, we just want to insert an article, and then session, session.flash, I am sending it to the database now.

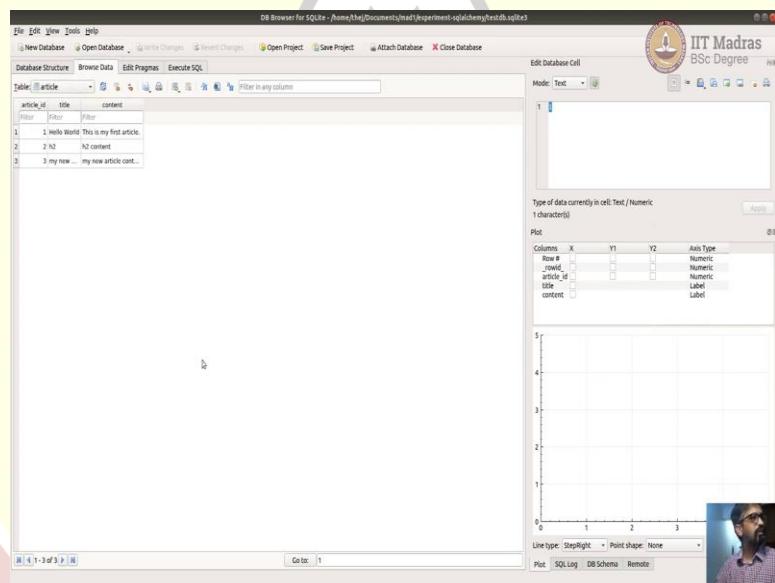
And then it sends me back an article_id, which is an auto incremented id, and then I take this id, and then I am going to insert into article_authors. And then if everything goes well, then it goes to else part and commit, if something happens in between then it goes to roll back section in this

except section, and then it rolls back embraces the thing raises an exception again, and then ends the program.

Here is when we just make it clear. So, let us run this, this goes clear. So yes, so it got article_id, then it went into commit area, and it committed. Let us assume, if something happens here, after getting the article_id, and before we create and save, of the article authors. And because of that, we have to roll back an article should not get saved.

So, let us just raise a dummy exception. I am just going to raise a dummy exception here. And just throw, I am just going to call dummy error. As soon as the error happens, it goes to the except part and rolls back. Hence, even though we run it again, this should not get inserted.

(Refer Slide Time: 23:57)





```
class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True)
    title = Column(String)
    content = Column(String)
    authors = relationship("User", secondary="article_authors")
    .... get article_id .....

class ArticleAuthors(Base):
    __tablename__ = 'article_authors'
    user_id = Column(Integer, ForeignKey("user.user_id"), primary_key=True, nullable=False)
    article_id = Column(Integer, ForeignKey("article.article_id"), primary_key=True, nullable=False)

engine = create_engine("sqlite:///./testdb.sqlite3")

if __name__ == '__main__':
    with Session(engine, autoflush=False) as session:
        session.begin()
        try:
            article = Article(title="dummy")
            session.add(article)
            session.flush()
            print(".... get article_id ....")
            print(article.article_id)
            raise Exception("Dummy Error")
            article_authors = ArticleAuthors(user_id=1, article_id=article.article_id)
            session.add(article_authors)
        except:
            print("Rolling back")
            session.rollback()
            raise
        else:
            print("Commit")
            session.commit()
```

```
class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True, autoincrement=True)
    title = Column(String)
    content = Column(String)
    authors = relationship("User", secondary="article_authors")

class ArticleAuthors(Base):
    __tablename__ = 'article_authors'
    user_id = Column(Integer, ForeignKey("user.user_id"), primary_key=True, nullable=False)
    article_id = Column(Integer, ForeignKey("article.article_id"), primary_key=True, nullable=False)

engine = create_engine("sqlite:///./testdb.sqlite3")

if __name__ == '__main__':
    with Session(engine, autoflush=False) as session:
        session.begin()
        try:
            article = Article(title="dummy new article", content="my dummy new article content")
            session.add(article)
            session.flush()
            print(".... get article_id ....")
            print(article.article_id)
            raise Exception("Dummy Error")
            article_authors = ArticleAuthors(user_id=1, article_id=article.article_id)
            session.add(article_authors)
        except:
            print("Rolling back")
            session.rollback()
            raise
        else:
            print("Commit")
            session.commit()
```

The image shows two screenshots of a computer interface. The top screenshot is a terminal window titled 'main.py' showing Python code for SQLAlchemy. The bottom screenshot is a 'DB Browser for SQLite' window showing a database table named 'article' with four rows of data.

```

class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True)
    title = Column(String)
    content = Column(String)
    authors = relationship("User", secondary=.....)
    ..... get article_id ......

class ArticleAuthors(Base):
    __tablename__ = 'article_authors'
    user_id = Column(Integer, ForeignKey='User.id')
    article_id = Column(Integer, ForeignKey='article.id')
    ..... get article_id ......

engine = create_engine("sqlite:///./testdb")
if __name__ == '__main__':
    with Session(engine, autocommit=False) as session:
        session.begin()
        try:
            article = Article(title="dummy")
            session.add(article)
            session.flush()
            print(".... get article_id ....")
            print(article.article_id)
            #raise Exception("Dummy Error")
            article.authors = ArticleAuthors()
            session.add(article_authors)
        except:
            print("Rolling back")
            session.rollback()
            raise
        else:
            print("Commit")
            session.commit()

```

article_id	title	content
1	Hello World. This is my first article.	
2	2 hi	hi content
3	3 my new	my new article cont.....
4	4 dummy	my dummy new art.....

Let us just check whether the last time it got inserted. Actually, it has got inserted. So, I am just going to call this my dummy a new article. Let us run this now. It threw an error, you can see a dummy error. And it came here. It is rolling back, it is printed rolling back and rolled back actually, this would not get inserted. So, I can refresh it. It is not inserted. Now, we can just mask this, which means there is no error again, and run it again. We, got committed and inserted, I can see a committed and inserted.

So, this is how we use transactions to handle errors, you if the database transaction has multiple steps, and you want to do many things before you commit everything at once. Ideally, we do not

do this this way we use relationships to add the data into the DB in simple cases, since this is a simple case, we could just use the relationship to insert.

(Refer Slide Time: 25:25)

```
Open ▾ B mali.py - Documents - IIT Madras - BSc Degree
class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True, autoincrement=True)
    title = Column(String)
    content = Column(String)
    authors = relationship("User", secondary="article_authors")

class ArticleAuthors(Base):
    __tablename__ = 'article_authors'
    user_id = Column(Integer, ForeignKey("user.user_id"), primary_key=True, nullable=False)
    article_id = Column(Integer, ForeignKey("article.article_id"), primary_key=True, nullable=False)

engine = create_engine("sqlite:///./testdb.sqlite3")

if __name__ == '__main__':
    with Session(engine, autoflush=False) as session:
        session.begin()
        try:
            author = session.query(User).filter(User.username == "thejeshgn").one()

            article = Article(title="Using relationship", content="Use relationships to insert. It's easy")

            article.authors.append(author)

            session.add(article)
        except:
            print("Exception, rolling back")
            session.rollback()
            raise
        else:
            print("No exceptions, hence commit")
            session.commit()
```

```
Open ▾ B mali.py - Documents - IIT Madras - BSc Degree
from sqlalchemy.orm import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    user_id = Column(Integer, primary_key=True, autoincrement=True)
    username = Column(String, unique=True)
    email = Column(String, unique=True)

class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True, autoincrement=True)
    title = Column(String)
    content = Column(String)
    authors = relationship("User", secondary="article_authors")

class ArticleAuthors(Base):
    __tablename__ = 'article_authors'
    user_id = Column(Integer, ForeignKey("user.user_id"), primary_key=True, nullable=False)
    article_id = Column(Integer, ForeignKey("article.article_id"), primary_key=True, nullable=False)

engine = create_engine("sqlite:///./testdb.sqlite3")

if __name__ == '__main__':
    with Session(engine, autoflush=False) as session:
        session.begin()
        try:
            author = session.query(User).filter(User.username == "thejeshgn").one()

            article = Article(title="Using relationship", content="Use relationships to insert. It's easy")

            article.authors.append(author)

            session.add(article)
        except:
            print("Exception, rolling back")
            session.rollback()
            raise
        else:
            print("No exceptions, hence commit")
            session.commit()
```

```

Open... B
main.py
class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True)
    title = Column(String)
    content = Column(String)
    authors = relationship("User", secondary=.....)
    ..... get article_id .....
class ArticleAuthors(Base):
    __tablename__ = 'article_authors'
    user_id = Column(Integer, ForeignKey("user.id"))
    article_id = Column(Integer, ForeignKey("article.id"))
    ..... get article_id .....
engine = create_engine("sqlite:///./testdb")
if __name__ == '__main__':
    with Session(engine, autoflush=False) as session:
        session.begin()
        try:
            author = session.query(User).filter_by(name='Thejeshgn').one()
            article = Article(title="Using", content="SQLAlchemy's relationship feature")
            article.authors.append(author)
            session.add(article)
        except:
            print("Exception, rolling back")
            session.rollback()
            raise
        else:
            print("No exceptions, hence commit")
            session.commit()

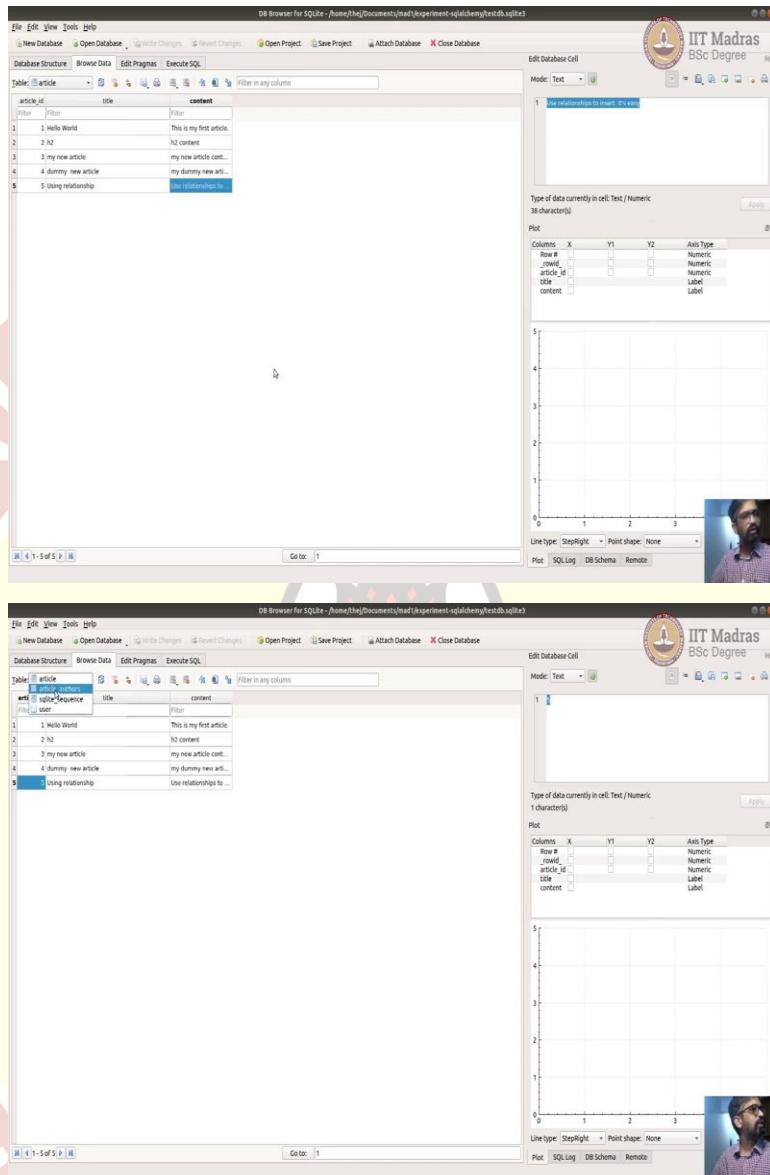
```



I will show one of the examples here, I am still making autoflush is false. And then I am getting an author object, by querying the user table by filtering Thejeshgn, he is going to be my author. And then I am creating an article by passing article title and article content here. Now, I need to insert article authors. Now, instead of like, last time, inserting into article authors directly, we are going to use relationships here, we are going to add an author into, articles author's attribute, and then save the article.

And you know, and then commit, indirectly what it will do is it will actually commit the article, then it will commit article authors based on the relationship, which is, authors here, and based on the relationship, it goes to article authors and inserts respective row and then saves it. This is cleaner, simpler and pythonic way, this is what we meant by it is easier, almost feels like accessing Python things, even though you are accessing database related stuff. Now, let us run this. No exception, and hence commit. So, come here, and then committed, let us see whether it got inserted, using relationship.

(Refer Slide Time: 27:02)



The image shows two screenshots of a computer interface. The top screenshot is titled 'DB Browser for SQLite - /home/kha/Downloads/test/experiment sqlalchemy/testdb.sqlite'. It displays a table named 'user' with columns 'user_id', 'username', and 'email'. Two rows are present: one for 'thejeshgn' with email 'thejeshgn@gmail.com' and another for 'raj' with email 'raj@example.com'. To the right of the table is a 'Plot' tool showing a scatter plot with axes X and Y, both ranging from 0 to 5. The bottom screenshot is titled 'main.py' and shows Python code for creating a database and inserting data into tables 'article' and 'article_authors'. The code uses SQLAlchemy's relationship feature to connect users to articles.

```

class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True, autoincrement=True)
    title = Column(String)
    content = Column(String)
    authors = relationship("User", secondary="article_authors")

class ArticleAuthors(Base):
    __tablename__ = 'article_authors'
    user_id = Column(Integer, ForeignKey("user.user_id"), primary_key=True, nullable=False)
    article_id = Column(Integer, ForeignKey("article.article_id"), primary_key=True, nullable=False)

engine = create_engine("sqlite:///../testdb.sqlite3")

if name == '__main__':
    with Session(engine, autoflush=False) as session:
        session.begin()
        try:
            author1 = session.query(User).filter(User.username == "thejeshgn").one()
            author2 = session.query(User).filter(User.username == "raj").one()

            article = Article(title="2nd Using relationship", content="2nd Use relationships to insert. It's easy")

            article.authors.append(author1)
            article.authors.append(author2)
            session.add(article)
        except:
            print("Exception, rolling back")
            session.rollback()
            raise
        else:
            print("No exceptions, hence commit")
            session.commit()

```

I am going to refresh here and go. It has inserted this. And now actually, the most important thing is whether it has inserted this part, or equivalent of this part, which is connecting this to this. And this title is user relationship, the article_id is 5. And if you go to article 5, it is linked to one, which is if you want to be user table, it takes one.

And that is what we wanted here. You could also done many, like, let us run one more. I would say author 1, author 2, and let us say this was Raj. And second, using relationship, just want to do random thing and do one author, you can add another author two authors. I am just going to print the article_id. We can just compare it using this title. So, here I am doing the same thing I

quiet for two authors. And then, I am creating an article object, I am adding two authors to that article, and just saving it. Let us run it.

(Refer Slide Time: 28:50)

```
main.py
class Article(Base):
    __tablename__ = 'article'
    article_id = Column(Integer, primary_key=True)
    title = Column(String)
    content = Column(String)
    authors = relationship("User", secondary=ArticleAuthors)
engine = create_engine("sqlite:///./testdb")
if __name__ == '__main__':
    with Session(engine, autoflush=False) as session:
        session.begin()
        try:
            author1 = session.query(User).first()
            author2 = session.query(User).first()
            article = Article(title="2nd User")
            article.authors.append(author1)
            article.authors.append(author2)
            session.add(article)
        except:
            print("Exception, rolling back")
            session.rollback()
            raise
        else:
            print("No exceptions, hence commit")
            session.commit()
It's easy!
```

DB Browser for SQLite - /home/thej/Documents/mad1/experiment sqlalchemy/testdb.sqlite

article_id	title	content
1	Hello World	This is my first article.
2	h2	h2 content
3	my new article	my new article cont...
4	dummy new article	my dummy new arti...
5	Using relationship	Use relationship to ...
6	2nd Using relationship	2nd Use relationship...

Plot

Columns: X, Y1, Y2, Axis Type

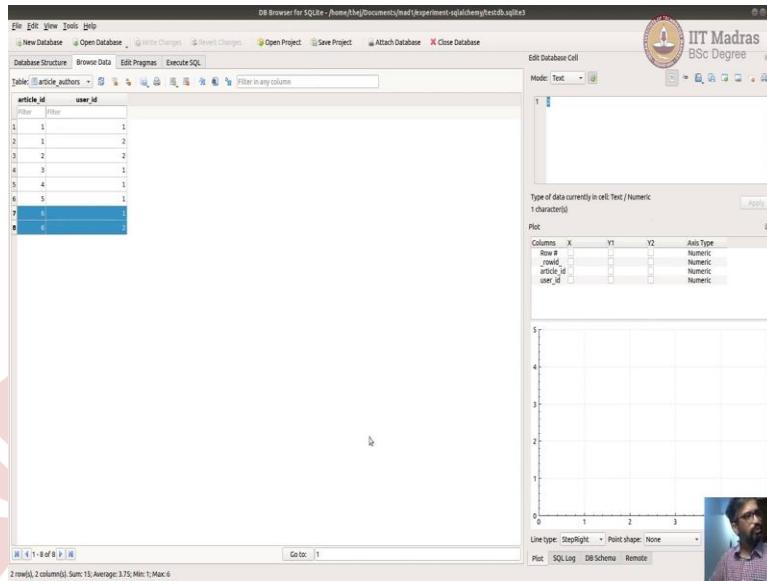
Rows: article_id, title, content

Plot: StepRight, Point shape: None

Plot SQL Log DB Schema Remote

Python Tab Width: 8 Line type: StepRight

IIT Madras BSc Degree



He got no exception hence commit. Let us see what whether this is this article with this title got inserted. Because it got inserted second using relationship and the article id is 6. Let us go, to the article author 6. You can see there are two relationships. It is quite easy to do using the relationship and both acquiring, inserting, even deleting for that matter. So, this is the end of this small introductory session to SQL alchemy, and generally the concept of ORM.

(Refer Slide Time: 29:35)

The screenshot shows the SQLAlchemy documentation page for 'Engine Configuration'. The main content area is titled 'Supported Databases' and discusses how to create an engine object from a URL. It includes examples for PostgreSQL and MySQL, and details about special characters in URLs. Below this, there's a code snippet for generating URL encoding and an example of a URL with special characters.

```

direct://username:password@host:port/database
postgresql://pg8000://dbuser:xx\55\5\7Fg\ghost10/epdb

```

As the URL is like any other URL, **special characters such as those that may be used in the password need to be URL encoded to be passed correctly**. Below is an example of a URL that includes the password "xx\55\5\7Fg", where the percent sign and slash characters are represented as %25 and %2f, respectively.

```

>>> import urllib.parse
>>> urllib.parse.quote_plus("xx\55\5\7Fg")
'xx%25%55%2f%7Fg'

```

Examples for common connection styles follow below. For a full index of detailed information on all included dialects as well as links to third-party dialects, see [Dialects](#).



There are a couple of things that you should remember one how to get the database URL. In our case it is straightforward because we are going to use sqlite, it is just a part of the escalator. Since it is going to use relate to path you have to start with the slash(/) and use this (./)dot slash for the local directory. Usually it will begin with sqlite:/// . We are not giving any driver mean, just using that default driver name.

Then if you already have a database, you have to write the models for the database, it is in the base class. And then, add also models. Now, you are going to use the same name as the column name or the attribute name, give this specific of, column types here, and all the conditions here. And also make sure that you establish the relationship the way you want it to be.

(Refer Slide Time: 30:35)



```
content = Column(String)
authors = relationship("User", secondary="article_authors")

class ArticleAuthors(Base):
    tablename = 'article_authors'
    user_id = Column(Integer, ForeignKey("user.user_id"), primary_key=True, nullable=False)
    article_id = Column(Integer, ForeignKey("article.article_id"), primary_key=True, nullable=False)

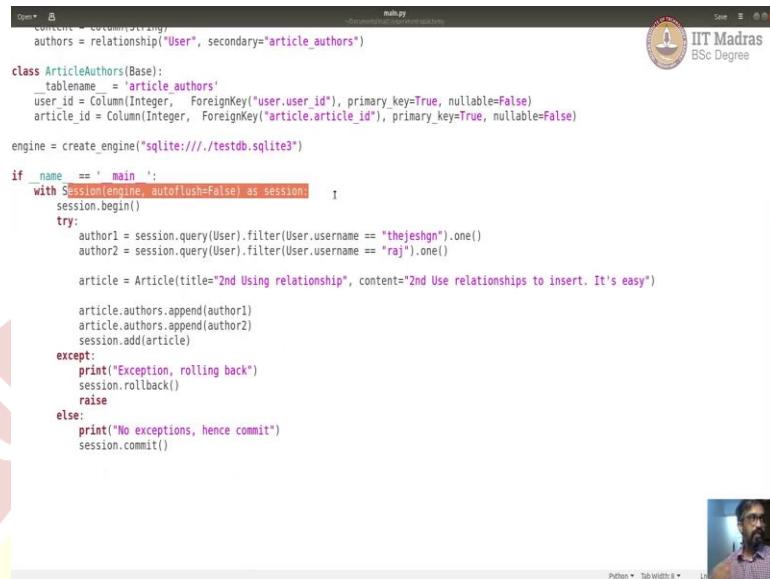
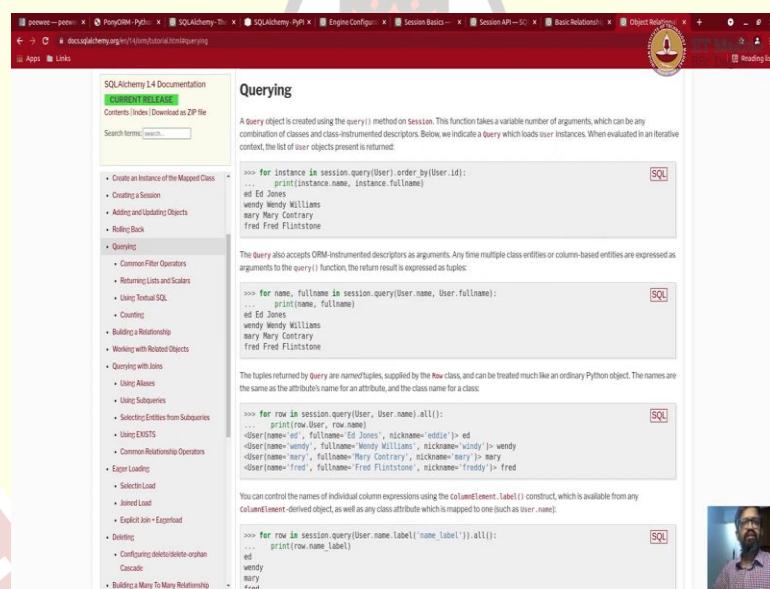
engine = create_engine("sqlite:///./testdb.sqlite3")

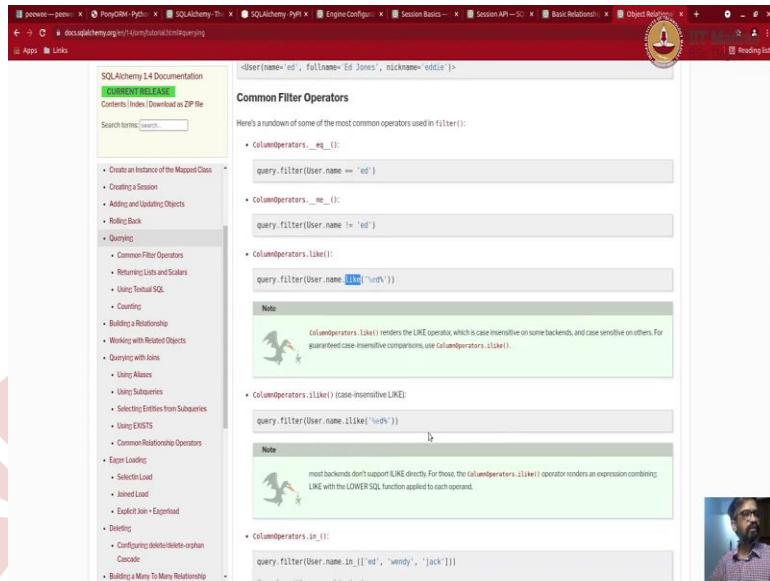
if name == 'main':
    with Session(engine, autoflush=False) as session:
        session.begin()
        try:
            author1 = session.query(User).filter(User.username == "thejeshgn").one()
            author2 = session.query(User).filter(User.username == "raj").one()

            article = Article(title="2nd Using relationship", content="2nd Use relationships to insert. It's easy")

            article.authors.append(author1)
            article.authors.append(author2)
            session.add(article)
        except:
            print("Exception, rolling back")
            session.rollback()
            raise
        else:
            print("No exceptions, hence commit")
            session.commit()


```



If you want more details, you can go to this page and learn a lot about relationship ((0)(30:38), to learn about at least four types of these relationships. So, one to many, many to one, one to one, and many to many, all of them are useful. You will, use them on day to day basis when you are accessing.

The third thing is how to create a session, you create a session by passing the engine. Engine has both dialect and the pool connection pool, you do not usually you will create one only one engine in the application context. And then you pass on that engine to the session and use the session to do all your stuff.

Use the, ORM way to do it use, object relations to create or delete objects. And then use the query on the session to query the database. You should actually go to this page, which has many examples on querying. There is a whole section on querying many ways, including all kind of joint, external and internal, and doing tops and operators like equal not equal, like operators, etcetera, etcetera. You should try most of them is straightforward to try with the given example.

And then you will learn about sessions and transactions, how to pickling, how to commit, how to roll back. This is how we handle errors when you are dealing with databases. That actually if you know, this many concept, I think you are kind of ready to jumpstart into using ORM in your application. This is in a standard way. We are also in the next coming sessions we will learn how to use it with the web application. Thank you for watching. Bye.