

NoSQL with Python



teach me no sql db with python

Sure! Let's dive into **NoSQL databases with Python**. We'll go through the basics and get hands-on with one of the most popular NoSQL databases: **MongoDB**. After that, I can show you other NoSQL types like Redis (key-value), Cassandra (wide-column), or Neo4j (graph), if you want.

♦ What is NoSQL?

NoSQL databases are non-relational and flexible in storing data. There are 4 common types:

1. **Document-based** (e.g., MongoDB)
2. **Key-value** (e.g., Redis)
3. **Column-based** (e.g., Cassandra)
4. **Graph-based** (e.g., Neo4j)

For now, let's start with **MongoDB + Python**, the most beginner-friendly combo.

🔧 Step-by-step: MongoDB with Python

Step 1: Install Required Packages

```
bash

`pip install pymongo`
```

Step 2: Run MongoDB

You can use a local MongoDB server or a cloud-hosted one like [MongoDB Atlas](#).

For now, let's assume you're running it locally on ``mongodb://localhost:27017/``.

Step 3: Basic Python Code Example

```
python

`from pymongo import MongoClient

# Connect to MongoDB
client = MongoClient("mongodb://localhost:27017/")

# Create or connect to a database
db = client["mydatabase"]`
```

```
# Create or connect to a collection (like a table in SQL)
collection = db["users"]

# Insert a document (like a row)
user = {"name": "Alice", "age": 25, "city": "NYC"}
collection.insert_one(user)

# Find one
print(collection.find_one({"name": "Alice"}))

# Update
collection.update_one({"name": "Alice"}, {"$set": {"age": 26}})

# Delete
collection.delete_one({"name": "Alice"})
```

How data looks in MongoDB:

Documents are stored as **BSON** (binary JSON), like this:

```
json

{
  "name": "Alice",
  "age": 25,
  "city": "NYC"
}
```

Why Use NoSQL?

- Schema-less (flexible structure)
- Scales easily (especially for big data)
- Fast for reads/writes in many cases
- Great for nested/JSON-like data

Want to Try It Yourself?

If you want, I can give you a hands-on project idea, like:

- A user profile store
- A simple to-do app
- Chat history storage

Let me know what you're most interested in, or if you'd rather try a different type of NoSQL database like Redis or Neo4j!



how to run it locally

Perfect! Let's get you set up to run **MongoDB locally** on your machine and connect to it with Python.

Step-by-step: Run MongoDB Locally (and use it with Python)

✓ 1. Install MongoDB Locally

For Windows / macOS / Linux:

Go to the official MongoDB Community Server download page: 

<https://www.mongodb.com/try/download/community>

- Choose your OS
 - Install it with default options
 - On Windows, **also install MongoDB Compass** (GUI tool – optional but helpful)
-

✓ 2. Run MongoDB Server

Once installed, start the server.

On macOS/Linux (with Homebrew):

```
bash

brew services start mongodb-community@6.0
```

On Windows:

- Use **Services** to start `MongoDB Server`
- Or run it from the command line:

```
bash

"C:\Program Files\MongoDB\Server\6.0\bin\mongod.exe"
```

By default, it runs on: `mongodb://localhost:27017/``

✓ 3. (Optional) Verify it's running

You can open a terminal and type:

```
bash

mongo
```

This opens the MongoDB shell.

You can try:

```
bash

`show dbs`
```

✓ 4. Write Python Code to Connect

Install the MongoDB Python driver:

```
bash

`pip install pymongo`
```

Then, run this in a Python script:

```
python

`from pymongo import MongoClient

# Connect to the local MongoDB server
client = MongoClient("mongodb://localhost:27017/")

# Access a database and collection
db = client["test_db"]
collection = db["people"]

# Insert a document
collection.insert_one({"name": "Alice", "age": 30})

# Read it back
result = collection.find_one({"name": "Alice"})
print(result)`
```

✓ 5. (Optional) GUI View with MongoDB Compass

If you installed MongoDB Compass:

- Open it
- Connect to: `mongodb://localhost:27017/`
- Browse your `test_db` and `people` collection

Let me know your OS and if you hit any issues—I can walk you through it step-by-step!



cant we run it without doing installation and directly using vscode

Yes! You **absolutely can** run MongoDB without installing anything **locally** by using **Docker**, **MongoDB Atlas (cloud)**, or even **embedded document DBs** like TinyDB for quick testing.

Let me show you the **easiest no-install path** to get started inside VS Code depending on what you're comfortable with:

Option 1: Use MongoDB with Docker (no full install)

If you have **Docker installed**, this is a 2-minute setup.

1. Run MongoDB in Docker:

```
bash

`docker run -d -p 27017:27017 --name mongodb mongo`
```

This will:

- Download the MongoDB image
- Run it locally on port `27017`

2. Now connect from Python (VS Code):

```
python

`from pymongo import MongoClient

client = MongoClient("mongodb://localhost:27017/")
db = client["mydb"]
collection = db["users"]

collection.insert_one({"name": "DockerUser", "age": 22})
print(collection.find_one({"name": "DockerUser"}))`
```

Boom. MongoDB is now running without installing anything permanently.

Option 2: Use MongoDB Atlas (Cloud – no install, no Docker)

1. Go to <https://www.mongodb.com/atlas/database>

- Sign up and create a **free cluster**
- Choose a username and password
- Whitelist your IP (0.0.0.0/0 = allow all)

2. Connect string looks like this:

```
perl

`mongodb+srv://<username>:<password>@cluster0.mongodb.net/test`
```

3. Use this in Python:

```
python

`from pymongo import MongoClient

uri = "mongodb+srv://<username>:<password>@cluster0.mongodb.net/?retryWrites=true&w=majority"
client = MongoClient(uri)

db = client["cloud_db"]
collection = db["users"]

collection.insert_one({"name": "CloudUser", "age": 28})
print(collection.find_one({"name": "CloudUser"}))`
```

✓ No local setup, just online.

Option 3: Use TinyDB (Python-only, embedded NoSQL)

Want a super light, file-based NoSQL DB without any server at all?

```
bash

`pip install tinydb`
```

```
python

`from tinydb import TinyDB, Query

db = TinyDB("db.json")
User = Query()

db.insert({"name": "TinyUser", "age": 18})
print(db.search(User.name == "TinyUser"))`
```

This is perfect for VS Code experiments or small apps. It stores everything in a local JSON file.

Which one do you prefer?

- Docker MongoDB (quick + real Mongo)
- MongoDB Atlas (cloud-based, no setup)
- TinyDB (zero dependencies, for learning/play)

Let me know and I'll tailor the next steps to that choice!



I am using docker here. is there any vs extension to see the data inside this db

Nice! Since you're using **Docker** and running MongoDB locally, you have a few great ways to view and manage your MongoDB data **right inside VS Code** using extensions.

Best VS Code Extension: MongoDB for VS Code

Official Extension by MongoDB

Name: **MongoDB for VS Code**

Publisher: MongoDB, Inc

Install from: [MongoDB for VS Code – Marketplace](#)

Features:

- View databases & collections
- Run MongoDB queries
- See documents in a table or JSON view
- Edit documents directly
- Connect to local or cloud MongoDB

How to Use It (with Docker MongoDB)

1. Install the Extension

- Go to Extensions tab (`Ctrl+Shift+X`)
- Search for `MongoDB` and install the one from **MongoDB, Inc.**

2. Open the MongoDB Sidebar

- Click the **Mongo leaf icon** on the sidebar

3. Connect to Your Local MongoDB

- Click **"Add Connection"**
- Paste your connection string:

```
arduino
`mongodb://localhost:27017`
```

- Give it a name (e.g., "Local Mongo Docker")

4. Browse and Manage

- Expand your connection
- Browse databases, collections
- View documents
- Run queries like:

```
javascript

db.users.find({ age: { $gt: 18 } })
```

Bonus: Try a Live Playground

- Right-click a collection → **Open Playground**
- Write Mongo shell queries directly in VS Code
- You can even run aggregation pipelines and see results instantly

TL;DR

Task	Tool
View MongoDB in VS Code	✅ MongoDB for VS Code extension
Run queries	✅ Playground inside the extension
Works with Docker	✅ Yes, just use <code>`localhost:27017`</code>

Let me know if you want a demo query or help using the extension!



i have installed extension now

Awesome! You're almost all set 🎉 Here's how to hook it up and start browsing your MongoDB running in Docker.

Connect MongoDB for VS Code to Your Local Docker MongoDB

✅ Step-by-Step

1. Open the MongoDB Extension

- Click the  MongoDB leaf icon in the VS Code sidebar.

2. Click "Connect" or "Add Connection"

- If this is your first time, there should be a **"Connect"** or **"Add Connection"** button.

3. Enter Connection String

```
arduino






mongodb://localhost:27017
```

4. Name Your Connection (Optional)

- For example: ``Local Docker Mongo``

5. Click Connect

Once Connected, You Can:

-  Browse databases (like ``test``, ``mydb``, etc.)
-  See collections (e.g. ``users``, ``products``)
-  View/edit documents
-  Insert new documents
-  Open a **Playground** (right-click a DB or collection → “Open Playground”) to run queries like:

```
javascript

`use mydb

db.users.find()
```

If You Don't See Anything

1. Make sure Docker Mongo is running:

```
bash

`docker ps
```

You should see the ``mongo`` container active.

2. Try inserting a test document via Python or Playground:

```
python

`# run in Python
collection.insert_one({"name": "VSCodeUser", "age": 29})
```

3. Then refresh the collection in the extension.

Want a quick sample script to populate test data so you can see it in VS Code?



how to create python playgroud

Great question! VS Code doesn't have a built-in **Python playground** like it does for MongoDB, but you can absolutely simulate one with a couple of easy tricks. Here's how to set up a mini **Python MongoDB playground** experience inside VS Code:

Option 1: Create a Python Script Playground

1. Create a New File

- In VS Code, open a new file
- Save it as something like: `mongo_playground.py`

2. Add Your MongoDB Code

Example:

```
python

`from pymongo import MongoClient

# Connect to MongoDB
client = MongoClient("mongodb://localhost:27017/")
db = client["mydb"]
collection = db["users"]

# Playground: write your test queries here
print("--- All Users ---")
for user in collection.find():
    print(user)

# Example Insert
# collection.insert_one({"name": "TestUser", "age": 99})
`
```

3. Run It

- Press `Ctrl+Alt+N` (if you have the **Code Runner** extension installed)
- OR open the terminal and run:

```
bash

`python mongo_playground.py
```

Option 2: Use Jupyter Notebooks in VS Code

If you like interactive cells:

1. Install Required Extensions

- Install: **Jupyter** + **Python** extension from VS Code marketplace

2. Create a `.ipynb` File

- File → New File → Save as `mongo_playground.ipynb`

3. Run MongoDB Code in Cells:

```
python

`from pymongo import MongoClient

client = MongoClient("mongodb://localhost:27017/")
db = client["mydb"]
collection = db["users"]

collection.find_one()
```

You can run each cell independently and see results right there 🔥

Quick Switch Tips

Want to...	Do this
Run one-off queries	Use <code>.py`</code> file or notebook
View/edit data visually	Use MongoDB VS Code extension
Run Mongo shell-style commands	Use MongoDB Playground (JS style)
Mix Python + Mongo interactivity	Use Jupyter notebooks

Want me to generate a full starter playground script or notebook for you?



using ipynb, now teach me mongo intermediate

Awesome! Let's level you up to **intermediate MongoDB with Python** using your Jupyter notebook setup 🚀

Intermediate MongoDB Topics with Python

We'll go through:

1. 🔍 Advanced queries
2. 🛠️ Updating documents
3. 📄 Query operators
4. 🧱 Indexing basics
5. 📊 Aggregation pipeline
6. 🎯 Projections and filters

Setup (you likely already have this)

```
python

`from pymongo import MongoClient

client = MongoClient("mongodb://localhost:27017/")
db = client["mydb"]
collection = db["users"]
```

1. Advanced Queries

Find all users older than 25:

```
python

`list(collection.find({"age": {"$gt": 25}}))
```

Combine multiple conditions:

```
python

`list(collection.find({"age": {"$gte": 25}, "city": "New York"}))
```

Use `\$or`, `\$in`, `\$regex`:

```
python

`list(collection.find({
    "$or": [
        {"city": {"$in": ["New York", "LA"]}},
        {"name": {"$regex": "^A", "$options": "i"}} # name starts with A, case-insensitive
    ]
}))
```

2. Update Documents

Update one field:

```
python

`collection.update_one({"name": "Alice"}, {"$set": {"age": 28}})
```

Increment a value:

```
python

`collection.update_one({"name": "Alice"}, {"$inc": {"age": 1}})
```

Update many documents:

```
python

collection.update_many({"city": "NYC"}, {"$set": {"region": "East Coast"}})
```

3. Query Operators

Operator	Meaning	Example
`\$gt`	Greater than	`{"age": {"\$gt": 30}}`
`\$lt`	Less than	`{"age": {"\$lt": 25}}`
`\$in`	In a list	`{"city": {"\$in": ["NYC", "LA"]}}`
`\$or`	OR conditions	`{"\$or": [{"name": "A"}, {"name": "B"}]}`
`\$exists`	Field exists or not	`{"region": {"\$exists": False}}`

4. Indexing (Speed Up Queries)

Create an index on the `age` field:

```
python

collection.create_index("age")
```

Compound index:

```
python

collection.create_index([("city", 1), ("age", -1)]) # 1=asc, -1=desc
```

View indexes:

```
python

collection.index_information()
```

5. Aggregation Pipeline

Count users by city:

python

```
`pipeline = [
    {"$group": {"_id": "$city", "count": {"$sum": 1}}},
    {"$sort": {"count": -1}}
]
list(collection.aggregate(pipeline))`
```

Average age per city:

python

```
`pipeline = [
    {"$group": {"_id": "$city", "avg_age": {"$avg": "$age"}}}
]
list(collection.aggregate(pipeline))`
```

6. Projections (Return only specific fields)

Only return name and age (exclude `_id`):

python

```
`list(collection.find({}, {"_id": 0, "name": 1, "age": 1}))`
```

Want to try a few exercises together? Like:

- Insert sample data
- Run queries
- Build an aggregation report

I can walk you through one step at a time.



give complete crud

Sure! Here's a **complete CRUD** (Create, Read, Update, Delete) example using **MongoDB with Python**.

We'll cover:

1. **Create** – Insert documents into the database
2. **Read** – Retrieve documents based on queries
3. **Update** – Modify documents
4. **Delete** – Remove documents from the collection

Setup

First, let's ensure your MongoDB client is set up:

```
python

`from pymongo import MongoClient

# Connect to MongoDB
client = MongoClient("mongodb://localhost:27017/")
db = client["mydb"]
collection = db["users"]
```

1. Create – Insert Data

Insert One Document

```
python

`user = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}

# Insert the document into the 'users' collection
collection.insert_one(user)
```

Insert Many Documents

```
python

`users = [
    {"name": "Bob", "age": 25, "city": "Los Angeles"},
    {"name": "Charlie", "age": 35, "city": "Chicago"},
    {"name": "Dave", "age": 28, "city": "Miami"}
]

# Insert multiple documents
collection.insert_many(users)
```

2. Read – Retrieve Data

Find All Documents

```
python

`# Get all users in the 'users' collection
all_users = list(collection.find())
print(all_users)
```

Find One Document (First Match)

```
python

`# Find a user by name
alice = collection.find_one({"name": "Alice"})
print(alice)
```

Query with Conditions

```
python

`# Find users older than 30
older_than_30 = list(collection.find({"age": {"$gt": 30}}))
print(older_than_30)
```

Find with Projections (Specific Fields)

```
python

`# Only get 'name' and 'city' fields, excluding '_id'
users_in_nyc = list(collection.find({"city": "New York"}, {"_id": 0, "name": 1, "city": 1}))
print(users_in_nyc)
```

3. Update – Modify Data

Update One Document

```
python

`# Increase Alice's age by 1
collection.update_one({"name": "Alice"}, {"$inc": {"age": 1}})
```

Update Many Documents

```
python

`# Update all users in New York to add a 'region' field
collection.update_many({"city": "New York"}, {"$set": {"region": "East Coast"}})
```

Replace a Document

```
python

`# Replace Bob's document entirely
collection.replace_one({"name": "Bob"}, {"name": "Bob", "age": 26, "city": "Los Angeles", "region":
```


4. Delete – Remove Data

Delete One Document

```
python

`# Delete a user named Alice
collection.delete_one({"name": "Alice"})`
```

Delete Many Documents

```
python

`# Delete all users older than 30
collection.delete_many({"age": {"$gt": 30}})`
```

Full CRUD Example:

```
python

`from pymongo import MongoClient

# Step 1: Setup connection
client = MongoClient("mongodb://localhost:27017/")
db = client["mydb"]
collection = db["users"]

# Step 2: Create (Insert data)
collection.insert_one({"name": "Alice", "age": 30, "city": "New York"})
collection.insert_many([
    {"name": "Bob", "age": 25, "city": "Los Angeles"},
    {"name": "Charlie", "age": 35, "city": "Chicago"}
])

# Step 3: Read (Find data)
print(collection.find_one({"name": "Alice"}))
print(list(collection.find({"age": {"$gt": 30}})))

# Step 4: Update (Modify data)
collection.update_one({"name": "Alice"}, {"$inc": {"age": 1}})
collection.update_many({"city": "New York"}, {"$set": {"region": "East Coast"}})

# Step 5: Delete (Remove data)
collection.delete_one({"name": "Alice"})
collection.delete_many({"age": {"$gt": 30}})`
```

Summary of MongoDB CRUD Operations:

Operation	Command	Example
Create	<code>`insert_one()` / `insert_many()`</code>	<code>`collection.insert_one({...})`</code>
Read	<code>`find()` / `find_one()`</code>	<code>`collection.find({"age": {"\$gt": 30}})`</code>

Operation	Command	Example
Update	<code>`update_one()` / `update_many()`</code>	<code>`collection.update_one({"name": "Alice"}, {"\$set": {"age": 31}})`</code>
Delete	<code>`delete_one()` / `delete_many()`</code>	<code>`collection.delete_one({"name": "Alice"})`</code>

? Want to try it with your data?

I can help with customizing the queries for your collection. Let me know what you'd like to explore more!



do we need orm for this

No, you **do not need an ORM (Object-Relational Mapping)** tool for MongoDB when working with it using Python. MongoDB is a **NoSQL** database, and it doesn't follow the relational model used by SQL databases, so an ORM is not necessary.

However, if you'd like to **simplify** your interactions with MongoDB and make things more **Pythonic** (and easier to manage with objects and classes), you can use **ODM (Object-Document Mapper)** libraries. These tools work similarly to ORMs but are designed for document-based databases like MongoDB.

Popular MongoDB ODMs for Python:

1. MongoEngine

- An **ODM** for MongoDB that allows you to define **Python classes** that map to MongoDB documents.

Example:

```
bash

`pip install mongoengine`
```

Example Python code using MongoEngine:

```
python

`from mongoengine import Document, StringField, IntField, connect

# Connect to MongoDB
connect("mydb", host="mongodb://localhost:27017/")

class User(Document):
    name = StringField(required=True)
    age = IntField()
    city = StringField()

# Create a new user
user = User(name="Alice", age=30, city="New York")
user.save()

# Query for users
users = User.objects(age__gt=25)
```

```
for user in users:
    print(user.name, user.age, user.city)
```

MongoEngine lets you interact with MongoDB using Python classes, similar to how you'd work with SQLAlchemy for SQL databases.

2. Pymodm (Another ODM for MongoDB)

- Another lightweight ODM for MongoDB that works with MongoDB in an object-oriented way.

Example:

```
bash

`pip install pymodm`
```

Example code:

```
python

`from pymodm import connect, fields, MongoModel

# Connect to MongoDB
connect("mongodb://localhost:27017/mydb")

class User(MongoModel):
    name = fields.CharField()
    age = fields.IntegerField()
    city = fields.CharField()

# Create and save a user
user = User(name="Bob", age=25, city="Los Angeles")
user.save()

# Query for users
for user in User.objects.raw({"age": {"$gt": 20}}):
    print(user.name, user.age, user.city)
```

Pymodm helps you manage MongoDB documents as Python objects and query them easily.

Why Use an ODM?

- **Cleaner code:** It abstracts away raw queries, allowing you to interact with documents like objects.
- **Validation:** ODMs often allow for field validation, which ensures the data being stored follows your defined schema.
- **Ease of use:** ODMs like MongoEngine or Pymodm can make querying, inserting, updating, and deleting data much easier and more readable.

Do You Need One?

You **don't need an ODM** if you're comfortable working with **raw queries** and just want a quick and direct approach to interact with MongoDB using Python's native drivers.

If you prefer a more structured approach (with schema, model definitions, and object manipulation), then using an ODM might be beneficial.

Let me know if you'd like to see more examples or dive deeper into an ODM!



in industry level do we use odm

Yes, **ODM (Object-Document Mappers)** are commonly used at the **industry level** when working with MongoDB and other NoSQL databases, especially in larger applications and systems that require more structure and abstraction.

However, whether to use an **ODM** or stick with the native MongoDB driver depends on several factors like the complexity of the application, the team's preferences, and the project's requirements.

Why ODMs Are Used in Industry:

1. Schema Definition and Validation:

- While MongoDB is schema-less, ODMs allow developers to define a schema for documents. This helps with data consistency and validation.
- For example, MongoEngine or Pymodm let you define required fields, data types, and even validation rules for your documents (e.g., ensuring an `age` field is always an integer).

2. Cleaner and More Pythonic Code:

- ODMs map MongoDB documents directly to Python objects (e.g., `User` class), which makes interacting with the data feel more natural and object-oriented.
- This is especially useful when your application grows in complexity and you have many models to manage.

3. Faster Development:

- ODMs abstract away some of the complexity of working with MongoDB's low-level operations. For instance, querying MongoDB using an object-oriented interface (e.g., `User.objects.filter(age__gt=30)`) is faster to write and read compared to raw queries.
- This can save time when developing and maintaining large applications.

4. Integration with Other Tools:

- Many ODMs integrate well with other Python frameworks (e.g., Flask, Django) and offer built-in support for features like migrations, which can make it easier to manage your database schema over time.

5. Reduced Boilerplate:

- An ODM reduces repetitive code, especially when dealing with object creation, updating, and deletion.
- Instead of manually mapping documents to dictionaries and dealing with low-level query syntax, ODMs allow you to work directly with Python objects.

6. Easier Debugging and Testing:

- Having a structured class-based approach often makes it easier to debug issues and test specific parts of your application.

When Might You NOT Use an ODM in Industry?

While ODMs offer a lot of benefits, they might not be necessary in all cases. Here are a few reasons why teams might opt to avoid ODMs:

1. Performance Considerations:

- In very high-performance applications, ODMs can introduce overhead due to the abstraction layer. Raw queries can be more efficient and allow for more control over performance optimizations.
- For example, if you're dealing with a large volume of reads/writes and need precise control over the queries, working directly with the MongoDB driver might be preferred.

2. Simple Use Cases:

- If your project is small or the data model is simple (e.g., limited fields and relationships), using the native MongoDB driver might be sufficient, and adding an ODM might be overkill.

3. Flexibility:

- MongoDB's strength lies in its flexibility, and ODMs sometimes introduce rigid structures that might not align well with projects that require dynamic or highly flexible data models.

4. Learning Curve:

- ODMs require developers to learn a new API and way of thinking about MongoDB interactions. In some teams, especially when working with a NoSQL database like MongoDB for the first time, there may be a preference for using the native driver to avoid unnecessary abstraction.

Examples of Where ODMs Are Used in the Industry

1. Web Applications:

- Web apps that need to handle user data, session information, or other dynamic content often use ODMs to manage MongoDB documents efficiently.
- Frameworks like **Flask** and **Django** (with extensions like ``mongoengine`` for Flask) often use ODMs in production environments.

2. Data-Driven Applications:

- Applications that manage large datasets with varied data types and require flexibility in their schema (e.g., e-commerce, real-time analytics) often rely on ODMs for better structure.

3. Microservices:

- In microservice architectures, services often use MongoDB as a data store. ODMs help organize and manage interactions with MongoDB, reducing the boilerplate code needed to interact with the database.

4. Real-Time Systems:

- ODMs can be used in systems where data is continuously updated (like messaging apps or social media platforms), and the application needs to quickly and easily update and retrieve documents.

Conclusion

In the industry, **ODMs are widely used**, particularly in projects where:

- You need structured models and data validation.
- You want to abstract away low-level query operations.
- You are working with large or complex systems that benefit from the clean, object-oriented approach.

However, whether to use an ODM or not depends on your project's complexity, performance needs, and how much abstraction you're comfortable with. For smaller projects, or when raw performance is a priority, you might prefer using MongoDB's native driver.

Would you like to dive deeper into how to implement MongoDB with an ODM in a real-world project, or do you need examples of both ODM and native driver approaches in an industry-level context?



which nosql db should I learn next

Great question! Since you're already working with MongoDB, you're off to a solid start with NoSQL databases. When choosing what to learn next, it depends on what kinds of systems or applications you want to build, and what features you're interested in.

Here are some popular NoSQL databases you could consider, each with its strengths and unique features:

1. Cassandra – For Highly Scalable, Distributed Systems

- **When to Learn:** If you're working on systems that need **high availability**, **horizontal scalability**, and are optimized for **write-heavy** workloads (like social media platforms, IoT systems, or recommendation engines).
- **Why Cassandra:**
 - **Wide-column store** that is great for distributed data with high scalability.
 - **Eventual consistency** and **partitioned architecture** make it ideal for large-scale systems that must remain available even when parts of the system fail.
 - Supports **massive amounts of data** across many servers (horizontally scalable).
- **Example Use Cases:**
 - Online shopping cart systems
 - Event logging or time-series data

- Social media data storage

2. Redis – For In-Memory, Low-Latency Data

- **When to Learn:** If you're building applications that require **extreme speed** and **low-latency data** access, like caching systems, message queues, or real-time analytics.
- **Why Redis:**
 - **In-memory data store** that supports extremely fast read and write operations.
 - Great for caching, session management, and real-time applications.
 - Supports **pub/sub messaging** patterns, **key-value storage**, and advanced data structures like lists, sets, and hashes.
- **Example Use Cases:**
 - Caching (e.g., caching API responses)
 - Session storage for web applications
 - Real-time leaderboards and counting systems

3. CouchDB – For Document-Oriented with ACID Compliance

- **When to Learn:** If you want a document-oriented database with features like **easy replication** and **offline-first capabilities** (great for mobile or distributed applications).
- **Why CouchDB:**
 - **Document-based NoSQL** similar to MongoDB, but with an emphasis on **multi-master replication** and offline support.
 - Uses **MapReduce** for views and supports **ACID transactions**.
 - Offers powerful **eventual consistency** and **replication**.
- **Example Use Cases:**
 - Mobile applications with offline functionality
 - Distributed systems that need easy replication and conflict resolution

4. Neo4j – For Graph Databases

- **When to Learn:** If you need to work with highly **connected data** or graph-based models (e.g., social networks, recommendation systems, fraud detection).
- **Why Neo4j:**
 - **Graph database** designed for handling highly connected data.
 - Stores data in **nodes** (entities) and **edges** (relationships), which is perfect for applications where relationships between data are as important as the data itself.
 - Uses **Cypher** query language, which is designed to work well with graph structures.
- **Example Use Cases:**

- Social networks
- Recommendation systems
- Fraud detection
- Network and IT operations monitoring

5. DynamoDB – For Managed, Scalable Key-Value and Document Store

- **When to Learn:** If you are looking for a **fully managed NoSQL** solution, especially in AWS environments, with **scalability** and high **availability**.
- **Why DynamoDB:**
 - **Fully managed by AWS**, so you don't need to worry about hardware or infrastructure.
 - Supports both **key-value** and **document data models**.
 - **Scalable and fast**, with built-in **auto-scaling** and **high availability**.
 - Integrates well with other AWS services like **Lambda** and **Kinesis**.
- **Example Use Cases:**
 - E-commerce applications (high read/write traffic)
 - Mobile apps with high scalability needs
 - Serverless applications (works well with AWS Lambda)

6. Elasticsearch – For Full-Text Search and Analytics

- **When to Learn:** If your application requires **advanced search** and **real-time analytics** on large volumes of text-based data (e.g., logs, user-generated content).
- **Why Elasticsearch:**
 - A **search engine** built on top of **Lucene**, perfect for full-text search, log analysis, and even real-time analytics.
 - Offers **distributed search** and allows for complex querying and aggregation.
- **Example Use Cases:**
 - Search engines (site-wide search)
 - Log aggregation (e.g., integrating with the **ELK stack**: Elasticsearch, Logstash, Kibana)
 - Analytics and monitoring systems

7. Firebase Firestore – For Real-Time, Serverless Apps

- **When to Learn:** If you're building mobile or web apps that need **real-time data synchronization** and **serverless architecture**, particularly in the **Google Cloud** ecosystem.
- **Why Firebase Firestore:**
 - **Real-time data synchronization** out of the box.

- Built for **serverless apps**, so it scales automatically.
- Integrated with **Firebase** SDK, making it easy to use in mobile and web development.
- **Example Use Cases:**
 - Real-time chat apps
 - Collaborative apps (like Google Docs)
 - Mobile apps with real-time data needs

How to Choose the Next NoSQL Database to Learn?

1. Data Model:

- If you like **document-oriented** databases (like MongoDB), you might enjoy learning **CouchDB** or **DynamoDB**.
- If your data is highly **connected**, a **graph database** like **Neo4j** would be ideal.

2. Scalability Needs:

- For **massive scalability**, consider learning **Cassandra** or **DynamoDB**.
- If you need **real-time, low-latency** operations, **Redis** or **Firestore** could be more suitable.

3. Use Cases:

- For **search and analytics**, Elasticsearch is a powerful choice.
- For **real-time synchronization** (especially in mobile apps), **Firestore** is a great option.

Conclusion:

If you're comfortable with MongoDB and want to expand your skillset, **learning Cassandra** (for scalable, distributed systems) or **Redis** (for fast, in-memory data operations) would be excellent choices.

However, if you're interested in exploring other types of NoSQL models (like graph databases or search engines), **Neo4j** (for graphs) or **Elasticsearch** (for full-text search) could be the next step.

Feel free to pick one based on your personal project goals or the specific features that interest you the most!

Would you like help diving deeper into one of these NoSQL databases?



lets learn cassandra

Awesome choice! **Cassandra** is a powerful distributed NoSQL database that is well-suited for applications requiring **high availability**, **horizontal scalability**, and **fast write-heavy workloads**. It's particularly useful when you need to handle large amounts of data across many nodes and ensure the system remains operational even if part of the database goes down.

Let's dive in!

1. Introduction to Cassandra

Cassandra is a **wide-column store** NoSQL database that organizes data into **tables**, similar to relational databases. However, unlike relational databases, Cassandra is designed to handle a massive volume of data distributed across multiple nodes in a cluster.

Key Features of Cassandra:

- **Decentralized:** Every node in the Cassandra cluster is the same; there's no master-slave hierarchy.
- **Fault-Tolerant:** It's designed to remain operational even when individual nodes fail.
- **Scalable:** It can scale horizontally by adding more nodes to the cluster.
- **Eventual Consistency:** It follows the **AP** part of the **CAP theorem**, offering high availability and partition tolerance, with eventual consistency.

2. Setting Up Cassandra Locally

To start with Cassandra, you can either set it up on your local machine or use Docker. Here's a simple guide to getting it up and running using Docker.

Step 1: Run Cassandra with Docker

```
bash

`docker run --name cassandra -d -p 9042:9042 cassandra:latest`
```

This will start a Cassandra container, and it will expose the default Cassandra port `9042` on your local machine.

Step 2: Connect to Cassandra

To connect to Cassandra, you'll need a Cassandra client. One common client is **CQLSH** (Cassandra Query Language Shell), which allows you to run queries interactively.

```
bash

`docker exec -it cassandra cqlsh`
```

You should now be in the Cassandra shell, where you can start interacting with the database.

3. Basic Cassandra Concepts

Before diving into code, let's understand a few key concepts in Cassandra:

- **Cluster:** A collection of nodes. Each node holds a portion of the data.
- **Node:** A single machine in the Cassandra cluster.
- **Keyspace:** Similar to a database in relational systems. It contains tables.
- **Table:** Holds the data. It's defined by a schema, but the schema can be flexible, and you can alter it over time.

- **Partition Key:** Determines how the data is distributed across nodes. It's important to choose a good partition key to avoid hotspots.
- **Clustering Columns:** Determines how rows are sorted within a partition.

4. Create Keyspace and Table in Cassandra

Once you're connected to `cqlsh`, you can start creating a **keyspace** and **table**.

Create a Keyspace:

```
cql

`CREATE KEYSPACE IF NOT EXISTS my_keyspace
WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1};`
```

This creates a keyspace named `my_keyspace` with **SimpleStrategy** replication and a replication factor of 1 (for local development).

Use the Keyspace:

```
cql

`USE my_keyspace;`
```

Create a Table:

```
cql

`CREATE TABLE IF NOT EXISTS users (
    user_id UUID PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    email TEXT
);`
```

This creates a table `users` with a **primary key** of `user_id` and additional fields for the user's name and email.

5. Basic CRUD Operations in Cassandra

Now, let's run through some **basic CRUD** operations:

Insert Data:

```
cql

`INSERT INTO users (user_id, first_name, last_name, email)
VALUES (uuid(), 'John', 'Doe', 'john.doe@example.com');`
```

Select Data:

```
cql
\SELECT * FROM users;
```

Update Data:

```
cql
\UPDATE users SET email = 'new.email@example.com' WHERE user_id = <uuid_value>;
```

Note that you **must** specify the **primary key** (in this case, `user_id`) when updating records.

Delete Data:

```
cql
\DELETE FROM users WHERE user_id = <uuid_value>;
```

6. Cassandra Data Modeling

Cassandra uses a **denormalized** approach to data modeling, where data is often duplicated to improve query performance. Here are some key considerations:

- **Query-first design:** When designing your data model in Cassandra, you typically start with the queries you need to support and design the schema around those queries.
- **Denormalization:** Instead of using joins like in relational databases, you often store redundant data to avoid performance bottlenecks.
- **Choose good partition keys:** Data is distributed across the nodes based on the partition key. A bad partition key can lead to **hotspots** and uneven data distribution.

Example of **wide-row design** (store similar data together for efficient querying):

```
cql
\CREATE TABLE IF NOT EXISTS user_posts (
    user_id UUID,
    post_id UUID,
    post_text TEXT,
    PRIMARY KEY (user_id, post_id)
);
```

This design is optimized for quickly retrieving all posts by a user.

7. Advanced Cassandra Topics

Once you're comfortable with the basics, you can explore more advanced features like:

- **Cassandra Indexes:** Used for performing queries on non-primary key columns.
- **Data Consistency:** Cassandra allows configuring consistency levels for read and write operations, balancing between availability and consistency.
- **Tuning and Scaling:** Learn how to scale Cassandra clusters by adding more nodes, configure data replication, and adjust consistency levels for different use cases.

Next Steps:

- **Cassandra Clients:** Explore Python client libraries like `cassandra-driver` to connect to Cassandra from your Python code.
- **Advanced Data Modeling:** Learn about more advanced data modeling techniques such as using **composite keys** and **collections** (lists, sets, maps).
- **Performance Tuning:** Learn about how to optimize your queries and data model for better performance at scale.

Let me know if you want to dive deeper into any of these topics or set up Cassandra with Python code!



how to use Cassandra in vs code

To use **Cassandra** in **VS Code** effectively, you'll need to connect to your Cassandra instance, run queries, and interact with the database. Here's a guide on how to set up your environment and use Cassandra in VS Code:

1. Install Necessary Extensions in VS Code

Before working with Cassandra in VS Code, you'll want to install a few helpful extensions.

Cassandra Extension (Optional)

There isn't an official Cassandra extension for VS Code, but you can use **CQL** (Cassandra Query Language) syntax highlighting and basic support for interacting with databases using these extensions:

- **CQL for Visual Studio Code:** This extension provides syntax highlighting for CQL files.
 - Install it from the Extensions marketplace by searching for "CQL".
 - It will help you write CQL commands with better syntax highlighting and formatting.

Docker Extension (For managing Docker containers directly from VS Code)

If you are using Docker to run Cassandra, the **Docker extension** allows you to manage your containers directly from VS Code.

- Install the **Docker extension** by searching for "Docker" in the VS Code marketplace. This will enable you to interact with Docker containers without leaving VS Code.

2. Connect to Cassandra from VS Code

Method 1: Using CQLSH (Cassandra Query Language Shell)

If you're running Cassandra in a Docker container, you can open a terminal in VS Code and run ``cqlsh`` to interact with Cassandra:

1. Open the **Terminal** in VS Code (using ``ctrl + ~``).
2. If Cassandra is running in Docker, run the following command to open the **CQL Shell**:

```
bash

`docker exec -it cassandra cqlsh`
```

3. This will bring you to the CQLSH interactive shell, where you can run CQL queries directly from the terminal in VS Code.

Method 2: Using Python and ``cassandra-driver``

To interact with Cassandra directly from your Python code within VS Code, you can use the ``cassandra-driver`` library. Here's how to do it:

1. Install the Cassandra Driver for Python:

- In the terminal in VS Code, install the ``cassandra-driver`` via pip:

```
bash

`pip install cassandra-driver`
```

2. Create a Python Script to Connect to Cassandra:

```
python

`from cassandra.cluster import Cluster

# Connect to Cassandra cluster
cluster = Cluster(['127.0.0.1']) # replace with your Cassandra node IPs
session = cluster.connect()

# Use a keyspace (replace with your keyspace name)
session.set_keyspace('my_keyspace')

# Run a query
rows = session.execute('SELECT * FROM users;')

# Print the results
for row in rows:
    print(row)

# Close the connection
cluster.shutdown()
`
```

- Replace ``127.0.0.1`` with the IP address of your Cassandra instance (if using Docker, it's typically ``localhost``).
- Replace ``my_keyspace`` with your keyspace name.
- This script will connect to Cassandra, query data from the ``users`` table, and print the results.

3. Run the Python Script:

- Open the terminal in VS Code and run your Python script:

```
bash

python your_script_name.py
```

- This will execute the query and output the results to the terminal.

3. Interacting with Cassandra Using VS Code's Docker Extension

If you're running Cassandra via Docker, you can interact with it directly through the **Docker extension** in VS Code.

1. Start the Cassandra Container:

- If you haven't already, start your Cassandra container using Docker (you can do this from the terminal or the VS Code Docker extension interface).

```
bash

docker run --name cassandra -d -p 9042:9042 cassandra:latest
```

2. Use Docker Extension to Manage Containers:

- Open the **Docker panel** in VS Code (on the left sidebar, click on the Docker icon).
- You'll see a list of running containers, including your Cassandra container.
- You can start/stop containers, view logs, and more.

3. Access Logs:

- You can view the Cassandra container's logs by right-clicking the container in the **Docker panel** and selecting **View Logs**. This will give you insight into the container's activity and help troubleshoot any issues.

4. Cassandra Database Management

While VS Code itself doesn't have a built-in GUI for managing Cassandra, you can use it for editing and running queries, and for managing Docker containers. For a more GUI-based approach to managing Cassandra databases, you can use **DataStax Studio** or **Cassandra Reaper**, which are third-party tools for working with Cassandra visually.

- **DataStax Studio:** A web-based tool for querying, visualizing, and analyzing Cassandra data. It's an excellent alternative if you prefer a GUI over working directly in VS Code.

5. Useful Resources for Cassandra in VS Code

- **CQL (Cassandra Query Language)** documentation: [Apache Cassandra CQL](#)
- **Docker and Cassandra:** [Docker Cassandra](#)

Conclusion

With VS Code, you can manage your Cassandra container with the Docker extension, write and run CQL queries, and connect to Cassandra via Python for more automated interactions. While VS Code doesn't have a fully integrated GUI for managing Cassandra databases, it provides a powerful, extensible environment for development.

If you want to explore further into working with Cassandra and Python, let me know! I can guide you through more advanced queries, data modeling, or performance optimizations.



using ipynb

Using **Jupyter Notebooks (IPython Notebooks)** is an excellent way to interact with Cassandra, especially if you're doing data analysis or querying databases within a notebook environment.

Here's a step-by-step guide on how to interact with **Cassandra** from a Jupyter notebook (`.ipynb` file) using Python:

1. Install Required Libraries

You'll need to install the following Python libraries:

- `cassandra-driver`: To connect to and interact with the Cassandra cluster.
- `ipython-sql` (optional): To execute SQL-like queries in Jupyter Notebooks.

Install the necessary libraries by running the following command in your terminal or Jupyter notebook:

```
bash
`pip install cassandra-driver ipython-sql`
```

2. Start Jupyter Notebook

Once the libraries are installed, you can start a Jupyter Notebook session. Run the following command in your terminal:

```
bash
`jupyter notebook`
```


This will open a new tab in your browser where you can create and interact with Jupyter Notebooks.

3. Connect to Cassandra from Jupyter Notebook

Now that you're inside Jupyter, you can start interacting with Cassandra. Here's an example notebook workflow:

Step 1: Import Required Libraries

In your Jupyter notebook cell, import the necessary libraries to connect to Cassandra:

```
python

`from cassandra.cluster import Cluster
from cassandra.auth import PlainTextAuthProvider
`
```

Step 2: Create a Cassandra Cluster and Connect

Assuming you're running Cassandra locally (e.g., using Docker), connect to your Cassandra instance:

```
python

`# Connect to the Cassandra cluster
cluster = Cluster(['127.0.0.1']) # replace with your Cassandra node IPs
session = cluster.connect()
`
```

This establishes a connection to Cassandra. If you're using Docker and running Cassandra on your local machine, `127.0.0.1` (localhost) should work.

Step 3: Set the Keyspace

Set the **keyspace** you want to work with. For example, if you have a keyspace named `my_keyspace`:

```
python

`session.set_keyspace('my_keyspace')
`
```

Step 4: Run a Query (CRUD Operations)

Insert Data

```
python

`# Insert data into the 'users' table
session.execute("""
    INSERT INTO users (user_id, first_name, last_name, email)
    VALUES (uuid(), 'Alice', 'Smith', 'alice.smith@example.com')
""")
`
```

Select Data

```
python

`# Query data from the 'users' table
rows = session.execute('SELECT * FROM users;')

# Print the results
for row in rows:
    print(row)
```

Update Data

```
python

`# Update a user's email address
session.execute("""
    UPDATE users
    SET email = 'alice.newemail@example.com'
    WHERE user_id = <uuid_value>;
""")
```

Delete Data

```
python

`# Delete a user from the 'users' table
session.execute("""
    DELETE FROM users
    WHERE user_id = <uuid_value>;
""")
```

Make sure to replace ``<uuid_value>`` with an actual ``UUID`` from your database.

4. Use `ipython-sql` to Run SQL-Like Queries in Jupyter

If you prefer working with SQL-like syntax in Jupyter Notebooks, you can use `ipython-sql` to execute SQL-like queries directly in notebook cells.

Step 1: Load `ipython-sql` Extension

In a notebook cell, load the SQL extension:

```
python

`%load_ext sql
```

Step 2: Set Up the Connection String

Now, connect to your Cassandra instance using a connection string (note that this part doesn't natively work with Cassandra directly, but can be used with other SQL databases):

```
python
```

```
`%sql postgresql://username:password@127.0.0.1:5432/my_database`
```

However, if you're working with **Cassandra**, you should still use the ``cassandra-driver`` as shown in earlier steps, since Cassandra doesn't natively support SQL queries in the way relational databases like PostgreSQL do.

5. Example of a Complete Jupyter Notebook Workflow with Cassandra

```
python
```

```
`from cassandra.cluster import Cluster
from cassandra.auth import PlainTextAuthProvider

# Step 1: Connect to Cassandra
cluster = Cluster(['127.0.0.1']) # Replace with your Cassandra node IP
session = cluster.connect()

# Step 2: Set the keyspace
session.set_keyspace('my_keyspace')

# Step 3: Create a new table
session.execute("""
    CREATE TABLE IF NOT EXISTS users (
        user_id UUID PRIMARY KEY,
        first_name TEXT,
        last_name TEXT,
        email TEXT
    );
""")

# Step 4: Insert data into the 'users' table
session.execute("""
    INSERT INTO users (user_id, first_name, last_name, email)
    VALUES (uuid(), 'John', 'Doe', 'john.doe@example.com');
""")

# Step 5: Query data from the 'users' table
rows = session.execute('SELECT * FROM users;')

# Step 6: Print the results
for row in rows:
    print(row)

# Step 7: Update data in the table
session.execute("""
    UPDATE users SET email = 'john.doe@newdomain.com' WHERE user_id = <uuid_value>;
""")

# Step 8: Delete data from the table
session.execute("""
    DELETE FROM users WHERE user_id = <uuid_value>;
""")

# Step 9: Close the connection
cluster.shutdown()
`
```

6. Handling Large Datasets with Cassandra in Jupyter

Cassandra is designed for fast writes and distributed data, but it might not be ideal for performing heavy data analysis like relational databases (e.g., PostgreSQL or MySQL). When working with large

datasets in Cassandra, it's important to:

- **Use partition keys effectively** to ensure that queries can be distributed across multiple nodes.
- **Limit the number of rows** retrieved in a query (use paging or filtering to handle large datasets).
- **Use batch operations** to insert or update multiple records efficiently.

Conclusion

By using **Jupyter notebooks**, you can easily integrate **Cassandra** into your data analysis workflow, performing CRUD operations, running queries, and visualizing the data. The `cassandra-driver` in Python allows you to interact with the database programmatically, while `ipython-sql` can provide a more SQL-like interface.

Would you like to see more examples of advanced queries or how to model your data in Cassandra for better performance? Let me know!



```
-----
InvalidRequest                                Traceback (most recent call last)
Cell In[10], line 1
----> 1 session.set_keyspace('my_keyspace')

File /opt/anaconda3/envs/prahlad-env/lib/python3.12/site-packages/cassandra/cluster.py:3340, in
Session.set_keyspace(self, keyspace)
    3335 def set_keyspace(self, keyspace):
    3336     """
    3337     Set the default keyspace for all queries made through this Session.
    3338     This operation blocks until complete.
    3339     """
-> 3340     self.execute('USE %s' % (protect_name(keyspace),))

File /opt/anaconda3/envs/prahlad-env/lib/python3.12/site-packages/cassandra/cluster.py:2677, in
Session.execute(self, query, parameters, timeout, trace, custom_payload, execution_profile,
paging_state, host, execute_as)
    2634 def execute(self, query, parameters=None, timeout=_NOT_SET, trace=False,
    2635               custom_payload=None, execution_profile=EXEC_PROFILE_DEFAULT,
    2636               paging_state=None, host=None, execute_as=None):
    2637     """
    2638     Execute the given query and synchronously wait for the response.
    2639     (...)
    2674     on a DSE cluster.
    2675     """
-> 2677     return self.execute_async(query, parameters, trace, custom_payload, timeout,
execution_profile, paging_state, host, execute_as).result()

File /opt/anaconda3/envs/prahlad-env/lib/python3.12/site-packages/cassandra/cluster.py:4956, in
ResponseFuture.result(self)
```

```
4954 return ResultSet(self, self._final_result)
4955 else:
-> 4956 raise self._final_exception
```

InvalidRequest: Error from server: code=2200 [Invalid query] message="Keyspace 'my_keyspace' does not exist"

The error message you're encountering indicates that the keyspace `my_keyspace` does not exist in your Cassandra database. Before setting the keyspace, you need to ensure that the keyspace is created.

Here's what you need to do:

1. Check if the Keyspace Exists

To check if the keyspace exists, you can run a query to list all keyspaces:

```
python

`# List all keyspaces
rows = session.execute('DESCRIBE KEYSPACES')

for row in rows:
    print(row)
```

If the `my_keyspace` doesn't appear in the output, it means that you need to create it.

2. Create the Keyspace

If the keyspace does not exist, you can create it with the following query:

```
python

`# Create the keyspace (replace 'my_keyspace' with your desired keyspace name)
session.execute("""
    CREATE KEYSPACE IF NOT EXISTS my_keyspace
    WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1}
""")
```

- `SimpleStrategy` is typically used for a single-node or small clusters.
- `replication_factor: 1` specifies that one copy of the data will be stored.

3. Set the Keyspace

After creating the keyspace, you can set it as the active keyspace in your session:

```
python

`session.set_keyspace('my_keyspace')
```

This should work without any issues now, assuming the keyspace is successfully created.

Full Example to Create Keyspace and Set It

```
python

`from cassandra.cluster import Cluster

# Connect to the Cassandra cluster
cluster = Cluster(['127.0.0.1']) # replace with your Cassandra node IP
session = cluster.connect()

# Create the keyspace if it doesn't exist
session.execute("""
    CREATE KEYSPACE IF NOT EXISTS my_keyspace
    WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1}
""")

# Set the keyspace
session.set_keyspace('my_keyspace')

# Create a table in the new keyspace
session.execute("""
    CREATE TABLE IF NOT EXISTS users (
        user_id UUID PRIMARY KEY,
        first_name TEXT,
        last_name TEXT,
        email TEXT
    );
""")

# Insert data into the 'users' table
session.execute("""
    INSERT INTO users (user_id, first_name, last_name, email)
    VALUES (uuid(), 'John', 'Doe', 'john.doe@example.com')
""")

# Query data from the 'users' table
rows = session.execute('SELECT * FROM users;')

# Print the results
for row in rows:
    print(row)

# Close the connection
cluster.shutdown()
`
```

Troubleshooting

- If you are using **Docker**, make sure your Cassandra container is running correctly and that the IP address used for `Cluster(['127.0.0.1'])` matches the container's network settings (often `localhost` or `127.0.0.1` for local setups).
- If you're using **Cloud-managed Cassandra**, ensure that your connection settings (e.g., authentication, IP whitelist) are correct.

Let me know if you need further assistance!



is there any orm or odm for this

Yes, for Cassandra, there are Object-Relational Mappers (ORMs) and Object-Document Mappers (ODMs) available, although they are not as widely used or feature-rich as those for relational databases.

Here are some popular Python libraries you can use to interact with **Cassandra** using an ORM/ODM pattern:

1. `Cassandra-ORM` (cassandra-orm)

`cassandra-orm` is an ORM-like library for Cassandra that provides a higher-level API to interact with the database, similar to what you get with SQLAlchemy for relational databases. It offers some of the features you'd expect from an ORM, like automatic mapping between Python objects and Cassandra rows.

- **Installation:**

```
bash

`pip install cassandra-orm`
```

- **Usage Example:**

```
python

`from cassandra.orm import model, columns
from cassandra.cluster import Cluster

# Define your model
class User(model.Model):
    user_id = columns.UUID(primary_key=True)
    first_name = columns.Text()
    last_name = columns.Text()
    email = columns.Text()

# Connect to Cassandra cluster
cluster = Cluster(['127.0.0.1'])
session = cluster.connect()

# Create the table in Cassandra (if it doesn't exist)
session.set_keyspace('my_keyspace')
User.create_table()

# Create a new user
user = User.create(user_id='some-uuid', first_name='Alice', last_name='Smith', email='alice@example.com')

# Query the users table
users = User.objects.first()
print(users.first_name, users.last_name)

# Close the connection
cluster.shutdown()
`
```

2. `Cassandra-Driver` + `CQLAlchemy`

`CQLAlchemy` is a library that integrates **Cassandra** with **SQLAlchemy** (which is typically an ORM for relational databases). It allows you to use SQLAlchemy's ORM features, such as declarative models, but with Cassandra's CQL syntax.

- **Installation:**

```
bash

`pip install cqlalchemy`
```

- **Usage Example:**

```
python

`from cqlalchemy import columns, Model
from cassandra.cluster import Cluster

# Define a model
class User(Model):
    __keyspace__ = 'my_keyspace'
    __table__ = 'users'

    user_id = columns.UUID(primary_key=True)
    first_name = columns.Text()
    last_name = columns.Text()
    email = columns.Text()

# Connect to Cassandra
cluster = Cluster(['127.0.0.1'])
session = cluster.connect('my_keyspace')

# Query and print data
users = User.objects.all()
for user in users:
    print(user.first_name, user.last_name)

# Close the connection
cluster.shutdown()
`
```

3. `Pycassa` (Deprecated but still used)

`Pycassa` is another ORM-style library for interacting with Cassandra, although it's considered outdated. It is sometimes still used in older applications. It provides an easy interface for defining models and performing CRUD operations on Cassandra tables.

- **Installation:**

```
bash

`pip install pycassa`
```

- **Usage Example** (for older versions of Python):

```
python

`from pycassa import system, ColumnFamily
from pycassa.pool import ConnectionPool

# Connect to Cassandra
pool = ConnectionPool('my_keyspace', ['127.0.0.1'])
column_family = ColumnFamily(pool, 'users')

# Insert data into 'users' table
column_family.insert('some-uuid', {'first_name': 'Alice', 'last_name': 'Smith', 'email': 'alice@smith.com'})

# Retrieve a user by ID
user = column_family.get('some-uuid')
```



```
print(user)
```

4. `Cassandra-ODM` (cassandra-odm)

While there isn't a very widely accepted ODM for Cassandra like there is for MongoDB (e.g., `mongoengine`), there are smaller projects that aim to provide ODM-like features for Cassandra. These are still in early stages compared to the more established libraries for other databases.

For example, you can look into:

- `cassandra-odm`: A basic ODM-style library for Cassandra (not widely used).
- `datastax-driver`: While this is not strictly an ODM, it provides a higher-level interface to interact with Cassandra, and you can build your own abstraction on top of it.

Considerations When Using ORM/ODM with Cassandra

While using an ORM or ODM for **Cassandra** can be convenient, you should keep in mind the following:

- **Cassandra is a distributed NoSQL database**, and ORMs/ODMs may introduce some limitations or inefficiencies because Cassandra's design focuses on horizontal scalability, high write throughput, and data partitioning.
- **Avoid complex joins**: Cassandra is optimized for denormalized data, so you should avoid patterns that require multiple joins (which are common in relational databases). Many ORM/ODM libraries may not be efficient for this.
- **Consistency and availability**: Cassandra offers different consistency levels. You may need to customize your ORM/ODM queries to handle the eventual consistency model effectively.

Conclusion

While there are a few ORMs and ODMs for **Cassandra** like `cassandra-orm` and `cqlalchemy`, it's important to carefully consider your use case before adopting one. For performance-critical applications or large-scale systems, using the **Cassandra Python driver** directly might be a better choice, as it provides more control over your queries and data modeling.

Let me know if you'd like help with a more in-depth example of using these libraries, or if you need assistance setting up or configuring them!



```
-----
ImportError                                Traceback (most recent call last)
Cell In[18], line 1
----> 1 from cqlalchemy import columns, Model
      2 from cassandra.cluster import Cluster
      4 # Define a model
```

```
ImportError: cannot import name 'columns' from 'cqlalchemy' (/opt/anaconda3/envs/prahlad-env/lib/python3.12/site-packages/cqlalchemy/__init__.py)
```

It seems that you're encountering an error when trying to use the `cqlalchemy` package, likely due to the incorrect import statements or potentially missing modules. It looks like `cqlalchemy` has a different API from what I initially suggested. Let's fix this and go through a working example.

Fixing the ImportError

1. **Ensure `cqlalchemy` is installed:** First, make sure that `cqlalchemy` is installed in your environment. You can verify this by running:

```
bash

`pip install cqlalchemy`
```

2. **Correct Usage:** The correct way to use `cqlalchemy` involves importing `Model` and `columns` in a slightly different manner. Here's the updated version of how you can define and use the models.

Working Example with `cqlalchemy`

```
python

`from cassandra.cluster import Cluster
from cqlalchemy import columns, Model

# Connect to Cassandra
cluster = Cluster(['127.0.0.1']) # Replace with your Cassandra node IP
session = cluster.connect()

# Create keyspace if not exists
session.execute("""
    CREATE KEYSPACE IF NOT EXISTS my_keyspace
    WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1}
""")

session.set_keyspace('my_keyspace')

# Define the Model for the 'users' table
class User(Model):
    __keyspace__ = 'my_keyspace' # Define keyspace
    __table__ = 'users' # Define table name

    user_id = columns.UUID(primary_key=True) # Define the primary key column
    first_name = columns.Text() # Define other columns
    last_name = columns.Text()
    email = columns.Text()

# Create the 'users' table (if not exists)
User.create_table()

# Insert a new user into the 'users' table
user = User.create(user_id='some-uuid', first_name='Alice', last_name='Smith', email='alice@example')

# Query the data (retrieving all users)
users = User.objects.all()

# Print user data
for u in users:
    print(f"{u.first_name} {u.last_name} - {u.email}")

# Close the connection
```

```
cluster.shutdown()
```

Notes:

1. **Model Definition:** In `sqlalchemy`, the `columns` module provides the types for each column, and `Model` is used to define the Cassandra table schema.
2. `create_table()`: The method `create_table()` creates the table for you if it doesn't already exist. It's similar to what you'd do with an ORM in a relational database.
3. **Inserting Data:** You can use `User.create()` to insert data into the table.
4. **Querying Data:** `User.objects.all()` will retrieve all the rows in the table (you can apply other filters as needed).

Troubleshooting

- If you encounter further issues related to `sqlalchemy`, make sure that your Python environment is correctly configured and that there are no conflicting packages.
- Double-check that Cassandra is running and accessible at the correct IP (`127.0.0.1` or any other relevant IP for your environment).

Let me know if you still face issues or need more details!



on industry do people use Cassandra orm

In the industry, **Cassandra ORM** (Object-Relational Mapping) libraries are **not commonly used** compared to traditional relational databases. Here's why:

1. Cassandra's Design Philosophy

Cassandra is designed as a **distributed NoSQL** database optimized for high availability, scalability, and performance in write-heavy applications. It is generally used in scenarios where horizontal scaling and fault tolerance are crucial, such as large-scale real-time analytics, recommendation systems, and logging. Cassandra works best with **denormalized data** (i.e., data is often duplicated across different tables to optimize for query performance).

- **ORMs are often inefficient:** ORMs like SQLAlchemy are typically designed to work with relational databases that support **joins** and **normalized schemas**. In contrast, **Cassandra** operates with a **denormalized schema** in mind and does not support **joins** or complex relational queries. Using an ORM on top of Cassandra could lead to inefficient queries that don't take advantage of Cassandra's strengths, such as its partitioning and replication models.

2. Cassandra is Schema-less, but Schema-Driven

While Cassandra is **schema-driven**, it's much more flexible than relational databases, and its focus is on **high availability and performance** rather than making data modeling as abstract and flexible as

possible like in traditional SQL-based systems. This makes **using ORM tools** somewhat impractical, as it would enforce unnecessary abstractions on top of the database's simple, straightforward data model.

3. Use Cases in Industry

In practice, **Cassandra** is often used **directly** via its native driver (e.g., ``cassandra-driver`` in Python) or through query languages like **CQL (Cassandra Query Language)**, which is **very similar to SQL**.

Industry teams usually opt to work directly with **CQL** for the following reasons:

- **High Performance:** Working with CQL directly gives developers more control over the queries and ensures they are optimized for Cassandra's internal architecture.
- **Scalability and Partitioning:** Cassandra's partitioning model is designed to spread data across a distributed cluster, and ORM tools can complicate the process of understanding and controlling how data is distributed.
- **Eventual Consistency:** Cassandra uses eventual consistency instead of strong consistency, and using ORM-like abstractions may obscure the understanding and control needed to manage consistency levels and read/write paths effectively.

4. Industry Approaches

In industry, when using **Cassandra**, developers typically follow these patterns:

- **Direct interaction with CQL:** Using the **native Cassandra drivers** (like ``cassandra-driver`` for Python) to communicate with the database and construct queries directly.
- **Data modeling for queries:** Denormalizing data and modeling it according to **access patterns** instead of relying on relational schema rules like **normalization**.
- **Use of microservices:** In microservice architectures, each service might have its own database, and Cassandra is often paired with other databases like MongoDB, PostgreSQL, or Elasticsearch, depending on the use case.

5. Alternative Tools and Libraries

While ORMs are rare for Cassandra, there are other higher-level tools that can simplify working with Cassandra, but they are not typically full-fledged ORM systems. These include:

- ``CQLAlchemy``: A library that allows SQLAlchemy-like functionality, though it's less commonly used in production.
- ``Datastax Driver``: The official Datastax driver for Cassandra provides higher-level abstractions for interacting with the database.
- **Custom Abstractions:** Some companies create their own abstraction layers, but they are usually **lightweight** and focused on **query generation** or **data mapping** rather than a full-fledged ORM system.

6. When ORM Might Be Used with Cassandra

In some edge cases, smaller projects or legacy systems might use ORMs, but it's important to understand the limitations and challenges involved:

- **Small-scale or Proof-of-Concept Projects:** In cases where developers are prototyping or working on smaller-scale applications, an ORM may be used for simplicity and ease of use.
- **Data Access Layers in Larger Systems:** Some teams create lightweight **data access layers** that abstract interactions with Cassandra, but they are custom-built solutions and not full-fledged ORM systems.

Conclusion: Industry Trends

- In the **industry**, **ORMs** are **not typically used with Cassandra** due to its design and intended use cases. Instead, developers interact with Cassandra through its native **CQL** and driver libraries, taking full advantage of its distributed nature and high performance.
- The **focus is on query optimization and partitioning strategies**, not abstracting away the underlying database like ORMs do for relational databases.

If you're planning to work with Cassandra in the industry, it's best to familiarize yourself with **CQL** and **best practices** for data modeling in a distributed database. Let me know if you'd like more insights or guidance on this!



create a small project with Cassandra and python

Let's create a small project using **Cassandra** and **Python**. The project will include basic **CRUD operations** (Create, Read, Update, Delete) for a ``users`` table in Cassandra. This will give you a good hands-on experience with Cassandra using Python.

Project Overview:

- **Database:** Cassandra
- **Table:** ``users``
 - Columns: ``user_id`` (UUID, Primary Key), ``first_name`` (Text), ``last_name`` (Text), ``email`` (Text)
- **Operations:**
 - Create a keyspace (``my_keyspace``).
 - Create a ``users`` table.
 - Insert a user into the table.
 - Retrieve and update user data.
 - Delete a user from the table.

Step 1: Install the Cassandra Python Driver

First, you need to install the official Cassandra Python driver:

```
bash

`pip install cassandra-driver`
```

Step 2: Set Up the Project

Create a Python script for the project. Let's call it `cassandra_crud.py`.

`cassandra_crud.py`

```
python

`from cassandra.cluster import Cluster
from cassandra.query import SimpleStatement
import uuid

# Step 1: Connect to Cassandra cluster
cluster = Cluster(['127.0.0.1']) # Replace with your Cassandra node IP
session = cluster.connect()

# Step 2: Create Keyspace if it doesn't exist
session.execute("""
    CREATE KEYSPACE IF NOT EXISTS my_keyspace
    WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1}
""")
session.set_keyspace('my_keyspace')

# Step 3: Create the Users table if it doesn't exist
session.execute("""
    CREATE TABLE IF NOT EXISTS users (
        user_id UUID PRIMARY KEY,
        first_name TEXT,
        last_name TEXT,
        email TEXT
    )
""")

# Step 4: Define CRUD operations

# Create a new user (INSERT operation)
def create_user(first_name, last_name, email):
    user_id = uuid.uuid4()
    session.execute("""
        INSERT INTO users (user_id, first_name, last_name, email)
        VALUES (%s, %s, %s, %s)
    """, (user_id, first_name, last_name, email))
    print(f"User {first_name} {last_name} created with ID: {user_id}")

# Read a user by user_id (SELECT operation)
def get_user(user_id):
    rows = session.execute("""
        SELECT * FROM users WHERE user_id = %s
    """, (user_id,))
    for row in rows:
        print(f"User ID: {row.user_id}, First Name: {row.first_name}, Last Name: {row.last_name}, E")

# Update a user's email (UPDATE operation)
def update_user_email(user_id, new_email):
    session.execute("""
        UPDATE users SET email = %s WHERE user_id = %s
    """, (new_email, user_id))
    print(f"User {user_id} email updated to {new_email}")

# Delete a user (DELETE operation)
def delete_user(user_id):
    session.execute("""
        DELETE FROM users WHERE user_id = %s
    """, (user_id,))
```

```

"""', (user_id,))
print(f"User {user_id} deleted from the database")

# Step 5: Example Usage of CRUD operations

# Create a new user
create_user("John", "Doe", "john.doe@example.com")

# Assuming we know the user ID, let's retrieve the user's information
user_id_to_lookup = uuid.UUID('insert-a-valid-uuid-here') # Replace with an actual UUID
get_user(user_id_to_lookup)

# Update the user's email
update_user_email(user_id_to_lookup, "new.email@example.com")

# Delete the user
delete_user(user_id_to_lookup)

# Step 6: Clean up and close the connection
cluster.shutdown()
`

```

Step 3: How It Works

1. **Connect to Cassandra:** The script connects to a **Cassandra cluster** running locally at ``127.0.0.1``. You can change the IP if you are running Cassandra on a different machine or container.
2. **Create Keyspace:** It creates a **keyspace** ``my_keyspace`` if it doesn't exist, and sets it as the active keyspace.
3. **Create Table:** The script defines a simple ``users`` table with columns for ``user_id`` (UUID), ``first_name`` (TEXT), ``last_name`` (TEXT), and ``email`` (TEXT). It will create the table if it doesn't already exist.
4. **CRUD Functions:**
 - **Create:** It generates a unique ``user_id`` using ``uuid.uuid4()`` and inserts the user into the table.
 - **Read:** It retrieves a user by their ``user_id`` using a ``SELECT`` query.
 - **Update:** It updates the email of a user given their ``user_id``.
 - **Delete:** It deletes a user by their ``user_id``.

Step 4: Run the Project

1. **Start Cassandra:** Make sure your **Cassandra** instance is running. If you're using Docker, make sure your container is up and running.
2. **Execute the Script:** Run the script with Python:

```

bash

`python cassandra_crud.py
`

```

This will execute the CRUD operations in sequence:

- Create a user.
- Retrieve the user (based on ``user_id``).

- Update the user's email.
- Delete the user.

Step 5: Verify

To verify that the operations have worked:

1. Open **Cassandra's CQL shell** (`cqlsh`) and connect to the `my_keyspace`.
2. Query the `users` table:

```
bash

USE my_keyspace;
SELECT * FROM users;
```

You should see the user data you inserted, and after the delete operation, the user will be removed from the table.

Conclusion

This small project demonstrates how to interact with **Cassandra** using Python and perform basic **CRUD** operations. You learned how to:

- Set up a Cassandra connection.
- Create a keyspace and table.
- Perform insert, select, update, and delete operations.
- Use **UUIDs** for unique user identification.

This project can be expanded further to add more complex data models, query optimization, or even integration with web frameworks such as **Flask** or **Django**.

Let me know if you'd like any additional features or improvements on this project!



is there vs code extension to view Cassandra db

Yes, there **is no official VS Code extension** specifically for viewing Cassandra databases like there is for MongoDB or PostgreSQL, but there are a few **alternatives** you can use to **view and manage Cassandra data** effectively while working in VS Code:

■ Option 1: DataStax Studio (external but powerful)

While not a VS Code extension, [DataStax Studio](#) is the most feature-rich, GUI-based tool for working with Cassandra. You can:

- Run CQL queries.

- Visualize data and schemas.
- Explore tables, keyspaces, and clusters.

👉 **Recommended for full-featured DB exploration.**

■ Option 2: Cassandra using VS Code Terminal + Jupyter/IPython

You're already using `.ipynb` files in VS Code. Combine that with `cassandra-driver` and run queries directly in Jupyter notebooks:

```
python

`from cassandra.cluster import Cluster
session = Cluster(["127.0.0.1"]).connect("your_keyspace")
rows = session.execute("SELECT * FROM your_table")
for row in rows:
    print(row)
`
```

✓ **Great for inline experimentation and script-based workflows.**

■ Option 3: Use Docker + CQLSH in VS Code terminal

If you're using Cassandra via Docker, open the VS Code terminal and connect to Cassandra using:

```
bash

`docker exec -it <container_name> cqlsh
`
```

From there, run:

```
sql

`USE my_keyspace;
SELECT * FROM users;
`
```

📦 This gives full access to the DB via CLI.

🔍 Option 4: Community Extensions

There are **some community-made extensions** (not officially endorsed or widely maintained) you might find in the Marketplace. For example:

- "SQLTools" + a generic CQL connector (limited support).
- Some try to support Cassandra via plugins, but features are very limited compared to tools for relational DBs.

● **Not recommended for long-term or professional usage.**

✓ **Best Combo for Dev Experience:**

- Use **VS Code + Jupyter Notebooks** for scripting and testing.
- Use **DataStax Studio** or **CQLSH** for schema exploration and manual queries.

Would you like help setting up DataStax Studio or a better Jupyter workflow for Cassandra?



what data does ansys mechanical simulation has

Great question! When you run a simulation in **ANSYS Mechanical**, it generates a **rich set of data** depending on the type of analysis you perform (e.g., static structural, modal, thermal, etc.).

Here's a breakdown of the **common data types** that ANSYS Mechanical simulations produce:

1. Geometry and Mesh Data

- **Nodes:** Coordinates of points in space.
 - **Elements:** Mesh connectivity (triangles, tetrahedra, hexahedra, etc.).
 - **Element types:** Info about element formulation (SOLID187, etc.).
 - **Element volumes, areas.**
-

2. Simulation Setup Data

- **Material properties:** Young's modulus, Poisson's ratio, thermal conductivity, etc.
 - **Boundary conditions:** Constraints, loads (forces, pressures, displacements, etc.).
 - **Contacts:** Definitions between surfaces, with types like frictional, bonded, etc.
 - **Analysis settings:** Solver control, time stepping (for transient analysis), etc.
-

3. Results Data (Primary + Derived)

The actual **output** of simulations—these can vary based on the type of analysis:

A. Structural Analysis:

- **Nodal displacements:** U_x , U_y , U_z
- **Element stresses:**
 - Von Mises stress
 - Principal stresses
 - Shear stresses
- **Strains:**
 - Elastic, total, plastic
- **Reaction forces** at supports

- **Contact pressures and status**

B. Modal Analysis:

- **Natural frequencies**
- **Mode shapes**

C. Thermal Analysis:

- **Nodal temperatures**
- **Heat flux**
- **Temperature gradients**

D. Transient Analysis:

- Time-varying data for all of the above (displacement, stress, temperature, etc.)
-

4. File Outputs

ANSYS outputs a number of files:

- `.rst`` – Result file (binary, contains most of the simulation results)
 - `.db`` – Database file (model data)
 - `.inp`` – Input file (used for solver input)
 - `.out`` – Text output file (solver log and convergence info)
 - `.cdb`` – Full model data file (used for external tools sometimes)
-

5. Post-Processed Fields

You can also extract:

- **Path plots** (stress/strain along a path)
 - **Volume/surface integrals**
 - **Deformation plots**
 - **Fatigue life predictions**
 - **Buckling load factors**
 - **Stress intensity factors (for fracture analysis)**
-

Pro Tip:

You can **access this data programmatically** using:

- **ANSYS ACT extensions** (Python APIs)

- **pyansys** (Python library to read `.rst` files)
- **APDL commands** (scripted automation)

Would you like to see how to extract or visualize this data using Python or how to post-process results efficiently within ANSYS Workbench or scripting?



example dat

Sure! Here's a quick example of **realistic data** that you might get from an **ANSYS Mechanical structural simulation**, like a simple static stress test on a steel bracket.



Simulation Context:

- Static structural analysis
- Steel bracket fixed at one end
- Load of 1000 N applied on the free end
- Material: Structural Steel ($E = 200 \text{ GPa}$, $\nu = 0.3$)



Geometry and Mesh (simplified)

plaintext

```
`Nodes:
ID      X (mm)  Y (mm)  Z (mm)
1        0.0    0.0    0.0
2       10.0    0.0    0.0
...

Elements:
ID      Node1   Node2   Node3   Node4
1         1      2       3       4
...
```



Material Properties

python

```
`Material: Structural Steel
Young's Modulus (E): 2.0e11 Pa
Poisson's Ratio (ν): 0.3
Density: 7850 kg/m³`
```

□ Boundary Conditions

```
plaintext
```

```
`Fixed Support:
  - Face A: Ux = Uy = Uz = 0

Force:
  - Face B: Fz = -1000 N (downward force)
```

Simulation Results

1. Nodal Displacement (mm)

```
plaintext
```

```
`Node    Ux      Uy      Uz
1        0.0000  0.0000  0.0000
42       0.0023 -0.0001 -1.2042
```

2. Von Mises Stress (MPa)

```
plaintext
```

```
`Element  Stress_VM
1         45.6
2         88.2
15        187.4 ← Max
```

3. Strain

```
plaintext
```

```
`Element  Equivalent Strain
1         0.00023
15        0.00095
```

4. Reaction Forces

```
plaintext
```

```
`Support Face:
Fx = 0 N
Fy = 0 N
Fz = +1000 N ← Balances the applied load
```

Post-Processing Visuals (usually inside ANSYS)

- **Total deformation plot:** shows deflected shape (scaled)
- **Stress contour:** color-coded Von Mises stress on the geometry

- **Probe at point:** "Stress = 187.4 MPa at Node 105"

Bonus: Python Access with `pyansys` (optional)

You can load and inspect result data from `.rst` using:

```
python

`import pyansys

result = pyansys.read_binary('file.rst')
stress = result.stress(0) # get stress for result set 0
displacement = result.nodal_displacement(0)
```

Would you like a sample file, or to walk through visualizing this in a Jupyter notebook or script?



give a python script to generate 1000 dataset for all these.

Sure! Here's a complete Python script that **generates a synthetic dataset** of 1000 samples simulating the kind of data you'd get from an ANSYS structural simulation. It includes:

- Node coordinates
- Nodal displacements
- Von Mises stress
- Equivalent strain
- Material info (same for all samples)
- Reaction force

You can run this in a `.ipynb` notebook in VS Code:

Python Code to Generate ANSYS-like Synthetic Dataset

```
python

`import numpy as np
import pandas as pd

# Constants (simulate Structural Steel properties)
E = 2.0e11 # Young's Modulus (Pa)
nu = 0.3 # Poisson's Ratio
density = 7850 # kg/m³

# Generate synthetic data
np.random.seed(42)
n_samples = 1000

# Random node positions in a 3D space (in mm)
X = np.random.uniform(0, 1000, n_samples)
Y = np.random.uniform(0, 1000, n_samples)
Z = np.random.uniform(0, 1000, n_samples)
```

```
# Simulated displacements (in mm)
Ux = np.random.normal(0, 0.05, n_samples)
Uy = np.random.normal(0, 0.05, n_samples)
Uz = np.random.normal(0, 1.0, n_samples) # Uz is more affected by vertical load

# Von Mises stress (in MPa)
stress_vm = np.random.normal(150, 50, n_samples).clip(min=0)

# Equivalent strain (unitless)
strain_eq = stress_vm * 1e6 / E #  $\sigma = E\varepsilon \rightarrow \varepsilon = \sigma/E$ 

# Reaction forces (just for illustration, balancing a 1000 N load)
Fz_reaction = np.random.normal(1000, 5, n_samples)

# Create dataframe
df = pd.DataFrame({
    'Node_ID': np.arange(1, n_samples+1),
    'X_mm': X,
    'Y_mm': Y,
    'Z_mm': Z,
    'Ux_mm': Ux,
    'Uy_mm': Uy,
    'Uz_mm': Uz,
    'Stress_VM_MPa': stress_vm,
    'Strain_Equivalent': strain_eq,
    'Reaction_Force_Fz_N': Fz_reaction
})

# Display first few rows
df.head()
```

📁 Save to CSV (optional)

```
python

df.to_csv("ansys_simulation_data.csv", index=False)
print("Data saved to ansys_simulation_data.csv")
```

This gives you a realistic mock-up dataset that mimics what you'd extract from ANSYS postprocessing—perfect for building dashboards, ML models, or just exploring simulation data patterns.

Let me know if you want this visualized or integrated into a GUI or database!



lets use this data in physics informed neural netwoek

Awesome! Let's build a **Physics-Informed Neural Network (PINN)** using the synthetic ANSYS-like data. A PINN doesn't just learn from data—it also respects the physics equations (like PDEs) governing the system.

🧠 Step-by-Step Plan for Using PINNs with Simulation Data

We'll focus on **linear elasticity** for structural problems (Hooke's law + equilibrium equations).

1. Physics Setup (Linear Elasticity PDE)

The governing equations in 3D are:

- **Equilibrium equations:**

$$\nabla \cdot \sigma + f = 0$$

where σ is the stress tensor, and f is body force.

- **Stress-strain relation:**

$$\sigma = \lambda(\nabla \cdot u)I + 2\mu\varepsilon$$

with $\varepsilon = \frac{1}{2}(\nabla u + \nabla u^T)$, and Lamé constants λ, μ from E, ν .

2. Data (Already Generated)

From your dataset:

- Inputs: (x, y, z)
- Outputs: (u_x, u_y, u_z) — displacements
- Material: $E = 2.0 \times 10^{11}, \nu = 0.3$

3. PINN Model (TensorFlow or PyTorch)

Here's the basic layout:

```
python

`import torch
import torch.nn as nn

class PINN(nn.Module):
    def __init__(self, layers):
        super().__init__()
        self.model = nn.Sequential()
        for i in range(len(layers) - 1):
            self.model.add_module(f"layer_{i}", nn.Linear(layers[i], layers[i+1]))
            if i < len(layers) - 2:
                self.model.add_module(f"tanh_{i}", nn.Tanh())

    def forward(self, x):
        return self.model(x)
```

4. Loss Function = Data Loss + Physics Loss

```
python

`def loss_fn(model, coords, disp_data, E, nu):
    lam = (E * nu) / ((1 + nu) * (1 - 2 * nu))
    mu = E / (2 * (1 + nu))

    coords.requires_grad_(True)
```



```

u_pred = model(coords)

# Data loss
loss_data = torch.mean((u_pred - disp_data) ** 2)

# Physics loss (simplified example:  $\nabla \cdot \sigma = 0$ )
u_x = torch.autograd.grad(u_pred[:, 0], coords, grad_outputs=torch.ones_like(u_pred[:, 0]), create_graph=True)
u_y = torch.autograd.grad(u_pred[:, 1], coords, grad_outputs=torch.ones_like(u_pred[:, 1]), create_graph=True)
u_z = torch.autograd.grad(u_pred[:, 2], coords, grad_outputs=torch.ones_like(u_pred[:, 2]), create_graph=True)

div_u = u_x[:, 0] + u_y[:, 1] + u_z[:, 2]
loss_phys = torch.mean(div_u**2)

return loss_data + loss_phys

```

🚩 5. Training Loop

Train with both data and physics terms in the loss. Use `Adam` or `LBFGS` optimizers.

Would you like me to:

- Build the full working notebook with training code?
- Add visualizations?
- Or integrate it with the synthetic data you generated?

Let's make it click step by step 🖱️



yes

Perfect! Let's build a **complete Physics-Informed Neural Network (PINN)** for your ANSYS-style synthetic data.

⚙️ Plan

We'll build this in stages, all inside your `.ipynb`:

1. Load the synthetic data
2. Build the PINN model (PyTorch)
3. Define physics-informed loss function
4. Train the model
5. Visualize predictions vs true data

✅ Step 1: Load the Synthetic Data

```

python

`import pandas as pd
import torch

```

```
from sklearn.preprocessing import MinMaxScaler

# Load the synthetic data
df = pd.read_csv("ansys_simulation_data.csv")

# Select inputs and outputs
X = df[['X_mm', 'Y_mm', 'Z_mm']].values
y = df[['Ux_mm', 'Uy_mm', 'Uz_mm']].values

# Normalize (important for training)
x_scaler = MinMaxScaler()
y_scaler = MinMaxScaler()
X_scaled = x_scaler.fit_transform(X)
y_scaled = y_scaler.fit_transform(y)

# Convert to PyTorch tensors
X_tensor = torch.tensor(X_scaled, dtype=torch.float32)
y_tensor = torch.tensor(y_scaled, dtype=torch.float32)
```

✓ Step 2: Define the PINN Model

python

```
`import torch.nn as nn

class PINN(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3, 64),
            nn.Tanh(),
            nn.Linear(64, 64),
            nn.Tanh(),
            nn.Linear(64, 64),
            nn.Tanh(),
            nn.Linear(64, 3)
        )

    def forward(self, x):
        return self.net(x)
```

✓ Step 3: Loss Function (Data + Physics)

python

```
`def physics_loss(model, x, E=2e11, nu=0.3):
    x.requires_grad_(True)
    u = model(x)

    lam = (E * nu) / ((1 + nu) * (1 - 2 * nu))
    mu = E / (2 * (1 + nu))

    grads = [torch.autograd.grad(u[:, i], x, grad_outputs=torch.ones_like(u[:, i]), create_graph=True)
              for i in range(3)]

    # Divergence of displacement ( $\nabla \cdot u$ )
    div_u = grads[0][:, 0] + grads[1][:, 1] + grads[2][:, 2]
    loss_pde = torch.mean(div_u ** 2)

    return loss_pde
```

✓ Step 4: Train the Model

```
python

`model = PINN()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

for epoch in range(1000):
    optimizer.zero_grad()
    u_pred = model(X_tensor)

    data_loss = torch.mean((u_pred - y_tensor)**2)
    phys_loss = physics_loss(model, X_tensor)

    total_loss = data_loss + phys_loss
    total_loss.backward()
    optimizer.step()

    if epoch % 100 == 0:
        print(f"Epoch {epoch}: Total Loss = {total_loss.item():.6f}")
`
```

✓ Step 5: Visualize Results

```
python

`import matplotlib.pyplot as plt

u_pred = model(X_tensor).detach().numpy()
u_pred_real = y_scaler.inverse_transform(u_pred)

# Compare predicted vs actual Uz
plt.scatter(df['Z_mm'], df['Uz_mm'], label='True', alpha=0.5)
plt.scatter(df['Z_mm'], u_pred_real[:, 2], label='Predicted', alpha=0.5)
plt.title("Predicted vs True Uz")
plt.legend()
plt.show()
`
```

Would you like me to package all this into a notebook for you? Or want to extend this to stress prediction or time-dependent problems next?



is there good vs code extension for csv

Yes! Here are some great **VS Code extensions for working with CSV files** efficiently:




🧩 Top CSV Extensions for VS Code

1. Excel Viewer by GrapeCity

- View CSV/TSV files in a **spreadsheet-like UI** inside VS Code.
- Features sorting, filtering, freezing headers.
- Live preview while editing.



 [Excel Viewer Extension](#)

2. Rainbow CSV

-  Syntax highlighting for CSV/TSV/DSV files.
-  Column alignment, delimiter detection, and SQL-like query support.
-  Smart auto-detection of separators.

 [Rainbow CSV Extension](#)

3. Edit csv by janisdd

-  Allows editing CSV files **in a table/grid mode**.
-  Two-way sync between table view and raw text.

 [Edit CSV Extension](#)

Recommendation

Install **Rainbow CSV** for syntax highlighting + **Edit CSV** for interactive editing. They work really well together for both development and data cleaning.

Want help setting them up or using them with your ANSYS dataset?



give example of inheritance vs composition in python

Absolutely! Here's a simple, clear example comparing **inheritance** vs **composition** in Python, using a real-world analogy: cars and engines.

Inheritance

In inheritance, a **child class** extends the **parent class**.

Example: `ElectricCar` inherits from `Car`

```
python

`class Car:
    def drive(self):
        print("The car is driving")

class ElectricCar(Car):
    def charge(self):
        print("Charging the battery")

# Usage
tesla = ElectricCar()
tesla.drive()  # Inherited from Car
```

```
tesla.charge() # Defined in ElectricCar
```

✓ Use When:

- There is a **"is-a"** relationship (ElectricCar *is a* Car).
- You want to reuse or extend the parent's behavior.

Composition

In composition, a class is **composed of** other objects.

Example: `Car` has an `Engine`

```
python

`class Engine:
    def start(self):
        print("Engine started")

class Car:
    def __init__(self):
        self.engine = Engine() # Composition

    def drive(self):
        self.engine.start()
        print("Car is moving")

# Usage
car = Car()
car.drive()
```

✓ Use When:

- There is a **"has-a"** relationship (Car *has an* Engine).
- You want more flexibility and loose coupling.

TL;DR

Concept	Inheritance	Composition
Relationship	"Is-a"	"Has-a"
Coupling	Tightly coupled	Loosely coupled
Flexibility	Less (fixed hierarchy)	More (can swap components)
Example	`ElectricCar` → `Car`	`Car` → uses `Engine`

Let me know if you want examples with abstract classes, interfaces, or real-world system design (like Uber's drivers or rides).



how to save the model

Great question! In PyTorch (since we used it for the PINN), you can save and load models easily.

Saving a Model

◆ Save just the model weights:

```
python

torch.save(model.state_dict(), "pinn_model_weights.pth")
```

This is the most common approach.

◆ Save the full model (architecture + weights):

```
python

torch.save(model, "pinn_full_model.pth")
```

Not always recommended—less flexible and version-sensitive.

Loading the Model

1. When using `state_dict` (recommended):

```
python

model = PINN() # You must define the model class before loading
model.load_state_dict(torch.load("pinn_model_weights.pth"))
model.eval() # Set to inference mode
```

2. When loading full model (if saved that way):

```
python

model = torch.load("pinn_full_model.pth")
model.eval()
```

✓ Tip:

Always call `model.eval()` before making predictions after loading.

Would you like to also:

- Export the model to ONNX?
- Visualize the model architecture?
- Integrate the save/load steps into your training notebook?

Let me know how you'd like to use the model next!



```
-----
NameError                                Traceback (most recent call last)
Cell In[3], line 1
----> 1 torch.save(model.state_dict(), "pinn_model_weights.pth")

NameError: name 'torch' is not defined
```

Ah, yep — that error means you haven't imported `torch` in the current notebook cell (or kernel session).

Just run this at the top of your notebook:

```
python

`import torch`
```

Then try saving again:

```
python

`torch.save(model.state_dict(), "pinn_model_weights.pth")`
```

Let me know if you'd like to:

- Automatically save the model every N epochs
- Save the training history alongside the model
- Load and test it on new data

Happy to walk you through any of that!



Export the model to ONNX?

Visualize the model architecture?

Integrate the save/load steps into your training notebook?

Let's tackle all three tasks! Here's how you can **export the model to ONNX**, **visualize the model architecture**, and **integrate save/load steps** into the training notebook:

1. Export the Model to ONNX

ONNX (Open Neural Network Exchange) allows you to export your PyTorch model to a cross-platform format. Here's how you can do it:

Exporting the Model:

```
python

`import torch.onnx

# Example: Export the model to ONNX format
dummy_input = torch.randn(1, 3) # Example input for the model, matching the input shape
torch.onnx.export(model, dummy_input, "pinn_model.onnx", input_names=['input'], output_names=['output'])
```

This will generate an `onnx` file, which can be used in frameworks like **TensorFlow**, **ONNX Runtime**, and **Caffe2**.

Tip:

Make sure the `dummy_input` matches the shape of your actual input data!

2. Visualize the Model Architecture

To visualize the model architecture, we can use the `torchviz` library.

Install `torchviz` (if not installed):

```
bash

`pip install torchviz`
```

Visualize the model:

```
python

`from torchviz import make_dot

# Create a visualization of the model
# Pass a sample input to generate the graph
sample_input = torch.randn(1, 3)
output = model(sample_input)
make_dot(output, params=dict(model.named_parameters())).render("model_architecture", format="png")
```

This will generate a **graph image** of your model's architecture as `model_architecture.png`.

3. Integrate Save/Load Steps into Training Notebook

You can integrate the **saving** and **loading** logic into your training process like this:

Saving the Model (after each epoch or at the end):

```
python

`# Save model weights every N epochs
if epoch % 100 == 0: # For example, save every 100 epochs
    torch.save(model.state_dict(), f"pinn_model_epoch_{epoch}.pth")
    print(f"Model saved at epoch {epoch}")
`
```

Loading the Model (at the start of the notebook or after training):

```
python

`# Load the model weights
model = PINN()
model.load_state_dict(torch.load("pinn_model_epoch_100.pth"))
model.eval() # Set the model to evaluation mode
`
```

Full Example: Save after training

```
python

`# After training loop
torch.save(model.state_dict(), "pinn_model_final.pth")
print("Model saved!")
`
```

And then later:

```
python

`model = PINN()
model.load_state_dict(torch.load("pinn_model_final.pth"))
model.eval() # Don't forget to set to evaluation mode
`
```

Full Training Notebook Integration Example:

```
python

`import torch
import torch.nn as nn
import torch.optim as optim

# Define your model
class PINN(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3, 64),
            nn.Tanh(),
            nn.Linear(64, 64),
            nn.Tanh(),
            nn.Linear(64, 64),
        )
`
```

```

        nn.Tanh(),
        nn.Linear(64, 3)
    )

    def forward(self, x):
        return self.net(x)

# Initialize model, optimizer, loss
model = PINN()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

# Training loop
for epoch in range(1000):
    optimizer.zero_grad()
    # Your training code here...

    # Save model every 100 epochs
    if epoch % 100 == 0:
        torch.save(model.state_dict(), f"pinn_model_epoch_{epoch}.pth")
        print(f"Model saved at epoch {epoch}")

```

Next Steps

- You can now export, visualize, and save/load models in your training workflow.
- Need help with model testing or performance evaluation after saving/loading?

Let me know how you'd like to continue!



```

-----
ModuleNotFoundError                                Traceback (most recent call last)
File /opt/anaconda3/envs/prahlad-env/lib/python3.12/site-
packages/torch/onnx/_internal/onnx_proto_utils.py:220, in _add_onnxscript_fn(model_bytes,
custom_opsets)
    219 try:
--> 220     import onnx
    221 except ImportError as e:

```

ModuleNotFoundError: No module named 'onnx'

The above exception was the direct cause of the following exception:

```

OnnxExporterError                                Traceback (most recent call last)
Cell In[11], line 5
      3 # Example: Export the model to ONNX format
      4 dummy_input = torch.randn(1, 3) # Example input for the model, matching the input shape
----> 5 torch.onnx.export(model, dummy_input, "pinn_model.onnx", input_names=['input'],
output_names=['output'])

```

```

File /opt/anaconda3/envs/prahlad-env/lib/python3.12/site-packages/torch/onnx/utils.py:551, in
export(model, args, f, export_params, verbose, training, input_names, output_names,
operator_export_type, opset_version, do_constant_folding, dynamic_axes,
keep_initializers_as_inputs, custom_opsets, export_modules_as_functions, autograd_inlining,
dynamo)

```

```

546 if f is None:
547     raise ValueError(
548         "Export destination must be specified for torchscript-onnx export."
549     )
--> 551 _export(
552     model,
553     args,
554     f,
555     export_params,
556     verbose,
557     training,
558     input_names,
559     output_names,
560     operator_export_type=operator_export_type,
561     opset_version=opset_version,
562     do_constant_folding=do_constant_folding,
563     dynamic_axes=dynamic_axes,
564     keep_initializers_as_inputs=keep_initializers_as_inputs,
565     custom_opsets=custom_opsets,
566     export_modules_as_functions=export_modules_as_functions,
567     autograd_inlining=autograd_inlining,
568 )
570 return None

```

File /opt/anaconda3/envs/prahlad-env/lib/python3.12/site-packages/torch/onnx/utils.py:1722, in _export(model, args, f, export_params, verbose, training, input_names, output_names, operator_export_type, export_type, opset_version, do_constant_folding, dynamic_axes, keep_initializers_as_inputs, fixed_batch_size, custom_opsets, add_node_names, onnx_shape_inference, export_modules_as_functions, autograd_inlining)

```

1703 (
1704     proto,
1705     export_map,
1706     ...)
1719     node_attr_to_name,
1720 )
1721 # insert function_proto into model_proto.
-> 1722 proto = onnx_proto_utils._add_onnxscript_fn(
1723     proto,
1724     custom_opsets,
1725 )
1726 if verbose:
1727     torch.onnx.log("Exported graph: ", graph)

```

File /opt/anaconda3/envs/prahlad-env/lib/python3.12/site-packages/torch/onnx/_internal/onnx_proto_utils.py:222, in _add_onnxscript_fn(model_bytes, custom_opsets)

```

220 import onnx
221 except ImportError as e:
--> 222     raise errors.OnnxExporterError("Module onnx is not installed!") from e
224 # For > 2GB model, onnx.load_fromstring would fail. However, because
225 # in _export_onnx, the tensors should be saved separately if the proto

```

```
226 # size > 2GB, and if it for some reason did not, the model would fail on
227 # serialization anyway in terms of the protobuf limitation. So we don't
228 # need to worry about > 2GB model getting here.
229 model_proto = onnx.load_model_from_string(model_bytes) # type: ignore[attr-defined]
```

OnnxExporterError: Module onnx is not installed!

The error you're encountering occurs because the **ONNX** library isn't installed in your current environment.

To fix this, install the ONNX library:

1. Open a terminal in your environment and run:

```
bash

`pip install onnx`
```

2. Once it's installed, try exporting your model to ONNX again:

```
python

`import torch.onnx

# Example: Export the model to ONNX format
dummy_input = torch.randn(1, 3) # Example input for the model, matching the input shape
torch.onnx.export(model, dummy_input, "pinn_model.onnx", input_names=['input'], output_names=['output'])`
```

This should work once ONNX is installed. Let me know if it works or if you encounter any other issues!



how to host that to TensorFlow, ONNX Runtime, and Caffe2.

Great! Once you've exported your PyTorch model to the **ONNX** format, you can use several frameworks like **TensorFlow**, **ONNX Runtime**, and **Caffe2** to load and serve the model.

Here's how you can host the exported ONNX model in each of these frameworks:

1. ONNX Runtime

ONNX Runtime is a high-performance inference engine for running ONNX models across various platforms.

Install ONNX Runtime:

```
bash

`pip install onnxruntime`
```

🧠 Load and Run the Model:

```
python

`import onnxruntime as ort
import numpy as np

# Load the ONNX model
onnx_model = ort.InferenceSession("pinn_model.onnx")

# Prepare input data (make sure it matches the input shape of your model)
input_data = np.random.randn(1, 3).astype(np.float32)

# Get model input names
input_name = onnx_model.get_inputs()[0].name

# Run the model
output = onnx_model.run(None, {input_name: input_data})

# Print the output
print(output)
```

2. TensorFlow (Using ONNX-TensorFlow)

TensorFlow does not natively support ONNX, but you can use **onnx-tf** to convert and run the ONNX model.

📦 Install `onnx-tf`:

```
bash

`pip install onnx-tf
```

🧠 Convert ONNX Model to TensorFlow:

```
python

`import onnx
from onnx_tf.backend import prepare

# Load the ONNX model
onnx_model = onnx.load("pinn_model.onnx")

# Convert to TensorFlow
tf_rep = prepare(onnx_model)

# Run the model in TensorFlow
input_data = np.random.randn(1, 3).astype(np.float32)
output = tf_rep.run(input_data)

# Print output
print(output)
```

3. Caffe2

Caffe2, which is now part of **PyTorch**, can also work with ONNX models directly.

Install Caffe2:

```
bash

`pip install torch==1.7.1 # Caffe2 is part of PyTorch 1.7.x`
```

Load and Run the Model:

```
python

`import torch
import onnx
from torch import onnx

# Load ONNX model
onnx_model = onnx.load("pinn_model.onnx")

# Import into PyTorch (since Caffe2 is merged with PyTorch)
# Define the model as in PyTorch (make sure to match input/output names)
model = torch.onnx._import_onnx(onnx_model)

# Run the model
input_data = torch.randn(1, 3).float()
output = model(input_data)

# Print output
print(output)`
```

4. Deployment in a Web Service (Example: FastAPI)

If you're looking to **serve** the model through a web service, you can use **FastAPI** to create an API for your model.

Install Dependencies:

```
bash

`pip install fastapi uvicorn onnxruntime numpy`
```

Create a Simple FastAPI Endpoint:

```
python

`from fastapi import FastAPI
import onnxruntime as ort
import numpy as np
from pydantic import BaseModel

# FastAPI app
app = FastAPI()`
```

```
# Load ONNX model
onnx_model = ort.InferenceSession("pinn_model.onnx")

# Define input data structure
class InputData(BaseModel):
    input: list

@app.post("/predict/")
async def predict(data: InputData):
    input_data = np.array(data.input, dtype=np.float32).reshape(1, -1)
    input_name = onnx_model.get_inputs()[0].name
    output = onnx_model.run(None, {input_name: input_data})
    return {"output": output[0].tolist()}

# Run with `uvicorn main:app --reload`
```

Run the FastAPI server with:

```
bash

`uvicorn main:app --reload`
```

This will start a web server where you can send POST requests to `/predict/` with input data in JSON format.

Summary of Frameworks:

1. **ONNX Runtime:** Directly run the ONNX model with high performance.
2. **TensorFlow:** Convert ONNX to TensorFlow and run it using `onnx-tf`.
3. **Caffe2:** Import the ONNX model into PyTorch (since Caffe2 is now part of it).
4. **Web Service:** Host the model with FastAPI and serve it to the web.

Let me know which framework you're interested in, and I can guide you through a deeper integration or deployment process.



```
-----
ModuleNotFoundError          Traceback (most recent call last)
Cell In[21], line 2
      1 import onnx
----> 2 from onnx_tf.backend import prepare
      4 # Load the ONNX model
      5 onnx_model = onnx.load("pinn_model.onnx")

File /opt/anaconda3/envs/prahlad-env/lib/python3.12/site-packages/onnx_tf/__init__.py:1
----> 1 from . import backend

File /opt/anaconda3/envs/prahlad-env/lib/python3.12/site-packages/onnx_tf/backend.py:29
      27 from onnx_tf.common import get_unique_suffix
      28 from onnx_tf.common import supports_device as common_supports_device
```

```

---> 29 from onnx_tf.common.handler_helper import get_all_backend_handlers
    30 from onnx_tf.pb_wrapper import OnnxNode
    31 import onnx_tf.common as common

```

```

File /opt/anaconda3/envs/prahlad-env/lib/python3.12/site-
packages/onnx_tf/common/handler_helper.py:3

```

```

    1 from onnx import defs
----> 3 from onnx_tf.handlers.backend import * # noqa
    4 from onnx_tf.handlers.backend_handler import BackendHandler
    5 import onnx_tf.common as common

```

```

File /opt/anaconda3/envs/prahlad-env/lib/python3.12/site-
packages/onnx_tf/handlers/backend/hardmax.py:3

```

```

    1 import numpy as np
    2 import tensorflow as tf
----> 3 import tensorflow_addons as tfa
    5 from onnx_tf.handlers.backend_handler import BackendHandler
    6 from onnx_tf.handlers.handler import onnx_op

```

```

ModuleNotFoundError: No module named 'tensorflow_addons'

```

The error you're encountering is due to a missing dependency called `tensorflow_addons`, which is required by **onnx-tf** to handle certain operations in the conversion from ONNX to TensorFlow.

To fix this:

1. **Install TensorFlow Addons:** You can install it using pip:

```

bash

`pip install tensorflow-addons

```

2. After installing **TensorFlow Addons**, try importing **onnx-tf** again and running your model conversion.

If you encounter further issues, feel free to let me know!



```

[33mWARNING: Skipping /opt/anaconda3/envs/prahlad-env/lib/python3.12/site-
packages/protobuf-4.25.3.dist-info due to invalid metadata entry 'name'
[0m[33mWARNING: Skipping /opt/anaconda3/envs/prahlad-env/lib/python3.12/site-
packages/protobuf-4.25.3.dist-info due to invalid metadata entry 'name'
[0m[31mERROR: Could not find a version that satisfies the requirement tensorflow-addons
(from versions: none)
[0m[31mERROR: No matching distribution found for tensorflow-addons
[0m[33mWARNING: Skipping /opt/anaconda3/envs/prahlad-env/lib/python3.12/site-
packages/protobuf-4.25.3.dist-info due to invalid metadata entry 'name'
[0m[33mWARNING: Skipping /opt/anaconda3/envs/prahlad-env/lib/python3.12/site-

```



```
packages/protobuf-4.25.3.dist-info due to invalid metadata entry 'name'
WARNING: Skipping /opt/anaconda3/envs/prahlad-env/lib/python3.12/site-
packages/protobuf-4.25.3.dist-info due to invalid metadata entry 'name'
```

It seems that **TensorFlow Addons** is not compatible with Python 3.12 yet, which might be causing the installation issue.

You can try the following solutions:

1. Use a Compatible Python Version:

TensorFlow Addons might work better with Python 3.8 or 3.9. Consider creating a new virtual environment with a lower version of Python:

Create a new environment with Python 3.9:

```
bash

conda create -n tf-env python=3.9
conda activate tf-env
pip install tensorflow tensorflow-addons
```

Then, you can install the necessary dependencies like **onnx-tf** and **TensorFlow Addons**.

2. Use `onnxruntime` Instead:

If you don't specifically need to use **TensorFlow**, **ONNX Runtime** can run ONNX models efficiently across platforms and is simpler to set up.

To install **ONNX Runtime** and use it to run your model:

```
bash

pip install onnxruntime
```

Then, load and run your ONNX model as previously shown:

```
python

import onnxruntime as ort
import numpy as np

# Load the ONNX model
onnx_model = ort.InferenceSession("pinn_model.onnx")

# Prepare input data (make sure it matches the input shape of your model)
input_data = np.random.randn(1, 3).astype(np.float32)

# Get model input names
input_name = onnx_model.get_inputs()[0].name

# Run the model
output = onnx_model.run(None, {input_name: input_data})

# Print the output
```

```
print(output)
```

Let me know if you'd prefer using **ONNX Runtime**, or if you'd like to proceed with the other methods!



for predict give curl

To send a prediction request using **cURL** for a model hosted in a web service (such as using **FastAPI**), you'll need to make a POST request to the appropriate endpoint.

Here's how you can use **cURL** to send a prediction request for the **FastAPI** example we discussed earlier:

1. Make Sure FastAPI is Running

Ensure your FastAPI app is running locally. You can run it using:

```
bash

uvicorn main:app --reload
```

This will start the FastAPI server at `http://127.0.0.1:8000`.

2. Send Prediction Request using cURL

Assume the API is expecting JSON input like this:

```
json

{
  "input": [0.5, -0.2, 0.1]
```

Here's the **cURL** command to send a POST request to the `/predict/` endpoint:

```
bash

curl -X 'POST' \
  'http://127.0.0.1:8000/predict/' \
  -H 'Content-Type: application/json' \
  -d '{
    "input": [0.5, -0.2, 0.1]
```

Breakdown of the cURL command:

- `-X 'POST'`: Specifies that the request method is POST.
- `'http://127.0.0.1:8000/predict/'`: The URL of the FastAPI endpoint.

- `-H 'Content-Type: application/json'`: Sets the header to indicate that the data being sent is in JSON format.
- `-d '...'`: Provides the data to be sent in the body of the request.

3. Receive the Response

After sending the request, you'll receive the prediction in response, similar to:

```
json

{
  "output": [[1.23, 0.45, 0.67]]
}
```

The output depends on how your model is defined and the input data.

Let me know if you need help setting up the FastAPI app or handling other aspects of the prediction process!



give code for fastapi

Here's an example of how to create a **FastAPI** app to serve your model and make predictions. This example assumes you're using an ONNX model, but you can adjust it for a PyTorch model or other frameworks.

1. Install Required Libraries

First, ensure you have **FastAPI**, **uvicorn**, and **ONNX Runtime** installed:

```
bash

pip install fastapi uvicorn onnxruntime
```

2. Create FastAPI App

Create a Python file (e.g., `main.py`) with the following content:

```
python

import onnxruntime as ort
import numpy as np
from fastapi import FastAPI
from pydantic import BaseModel
from typing import List

# Create FastAPI instance
app = FastAPI()

# Define the input data structure
class InputData(BaseModel):
    input: List[float]
```

```
# Load the ONNX model
onnx_model = ort.InferenceSession("pinn_model.onnx")

@app.post("/predict/")
async def predict(data: InputData):
    # Convert input data to numpy array
    input_data = np.array(data.input, dtype=np.float32).reshape(1, -1)

    # Get the input name for the model (assuming it's the first input)
    input_name = onnx_model.get_inputs()[0].name

    # Run the model to get the output
    output = onnx_model.run(None, {input_name: input_data})

    # Return the output
    return {"output": output[0].tolist()}
```

Explanation:

- `InputData` class: Uses **Pydantic** to define the expected input structure. Here, we expect an input list of floats.
- `onnx_model`: Loads the ONNX model using **ONNX Runtime**.
- `/predict/` endpoint: Accepts POST requests with the input data, runs it through the model, and returns the output.

3. Run FastAPI with Uvicorn

Once your FastAPI app is ready, run the following command to start the server:

```
bash

uvicorn main:app --reload
```

This will start the FastAPI server locally at `http://127.0.0.1:8000`.

4. Test the API with cURL

You can now send a prediction request using **cURL** as shown earlier:

```
bash

curl -X 'POST' \
  'http://127.0.0.1:8000/predict/' \
  -H 'Content-Type: application/json' \
  -d '{
    "input": [0.5, -0.2, 0.1]
  }'
```

5. Alternative: Test with Swagger UI

FastAPI automatically generates **Swagger UI** for your API. You can visit the following URL in your browser to interact with the API:

```
arduino
```

```
`http://127.0.0.1:8000/docs`
```

This will give you a nice web interface to try out the API and see the documentation automatically generated by FastAPI.

This setup should enable you to serve your model using FastAPI, allowing you to send and receive predictions via HTTP requests. Let me know if you run into any issues!



is there any vs code extension to view.pth files

While there is no direct VS Code extension to view `.pth` files (which are typically PyTorch model weight files), there are some ways to interact with them effectively within VS Code:

1. Use Python Interactive Window / Jupyter Notebooks in VS Code

You can load and inspect `.pth` files using Python code in the interactive environment (such as Jupyter notebooks or the Python Interactive window in VS Code). This gives you flexibility in inspecting the contents of the model file. Here's a quick guide:

Steps to load and inspect `.pth` in Python:

1. Open a Jupyter notebook or use the Python interactive window in VS Code.
2. Load the model using PyTorch and inspect its weights:

```
python
```

```
`import torch
```

```
# Load your model weights
model = torch.load("your_model.pth")
```

```
# Inspect model state_dict (weights)
print(model.state_dict())
```

```
# Or inspect the entire model
print(model)
```

You can also run specific queries on the model weights, like checking layer names, dimensions, etc.

2. VS Code Extensions

Here are a couple of helpful extensions that can aid in working with `.pth` files and machine learning models:

- **Python:** This is the most essential extension for working with Python files in VS Code, including `.pth` files. It provides IntelliSense, linting, and debugging support.

- Install via: [Python Extension for VS Code](#)
- **Jupyter:** If you're working in a Jupyter notebook, this extension is invaluable. It lets you work directly with `.ipynb` files and interact with model weights easily in a notebook interface.`
 - Install via: [Jupyter Extension for VS Code](#)
- **PyTorch Snippets:** This extension provides handy snippets for PyTorch-related code, which may help you quickly load or inspect `.pth` files.`
 - Install via: [PyTorch Snippets for VS Code](#)

3. Visualizing Model Structure

If you want to visualize the architecture of a model saved in a `.pth` file, you can use a few tools to convert it into a more readable format (such as a graph):`

- **TensorBoard:** You can visualize model graphs and other metrics. PyTorch has built-in support for TensorBoard.

```
python

`from torch.utils.tensorboard import SummaryWriter

writer = SummaryWriter('runs/model_visualization')
writer.add_graph(model, input_to_model)
writer.close()
```

You can then view the graph in TensorBoard by running:

```
bash

`tensorboard --logdir=runs
```

Then open the TensorBoard interface in your browser at `http://localhost:6006`.`

Let me know if you'd like more help setting up any of these solutions!