7. Python Database Interaction

7.1. Introduction to NoSQL database

What is NoSQL?

As per the official Wiki definition: "A **NoSQL** (originally referring to "**non SQL**" or "**non relational**") database provides a mechanism for storage and retrieval of data that is modeled in means other than **the tabular relations** used in relation databases (**RDBMS**). It encompasses a wide variety of different database technologies that were developed in response to a rise in the volume of data stored about users, objects and products, the frequency in which this data is accessed, and performance and processing needs. Generally, NoSQL databases are structured in a key-value pair, graph database, document-oriented or column-oriented structure.

Over decades and decades of software development, we have been using databases in form of SQL (Structured Query Language) where we store our data in relational tables. However, in recent years with the tremendous rise in use of internet and Web 2.0 applications, the databases have grown into thousands and thousands of terabytes. Applications such as Facebook, Google, Amazon, Watsapp, etc. gave rise to an entire new era of database management which follows approach of simple design, speed and faster scaling than the traditional databases. Such databases are used in big data, massive real time applications and analytics.

As an example, consider that you have a blogging application that stores user blogs. Now suppose that you have to incorporate some new features in your application such as users liking these blog posts or commenting on them or liking these comments. With a typical RDBMS implementation, this will need a complete overhaul to your existing database design. However, if you use NoSQL in such scenarios, you can easily modify your data structure to match these agile requirements. With NoSQL you can directly start inserting this new data in your existing structure without creating any new pre-defined columns or pre-defined structure.

Challenges of RDBMS

- RDBMS assumes a well-defined structure of data and assumes that the data is largely uniform.
- It needs the schema of your application and its properties (columns, types, etc.) to be defined up-front before building the application. This does not match well with the agile development approaches for highly dynamic applications.

- As the data starts to grow larger, you have to scale your database vertically, i.e. adding more capacity to the existing servers.

7.2. Advantages of NoSQL database

Benefits of NoSQL over RDBMS

Schema Less:

NoSQL databases being schema-less do not define any strict data structure.

Dynamic and Agile:

NoSQL databases have good tendency to grow dynamically with changing requirements. It can handle structured, semi-structured and unstructured data.

Scales Horizontally:

In contrast to SQL databases which scale vertically, NoSQL scales horizontally by adding more servers and using concepts of sharing and replication. This behavior of NoSQL fits with the cloud computing services such as Amazon Web Services (AWS) which allows you to handle virtual servers which can be expanded horizontally on demand.

Better Performance:

All the NoSQL databases claim to deliver better and faster performance as compared to traditional RDBMS implementations.

Talking about the limitations, since NoSQL is an entire set of databases (and not a single database), the limitations differ from database to database. Some of these databases do not support ACID transactions while some of them might be lacking in reliability. But each one of them has their own strengths due to which they are well suited for specific requirements.

Types of NoSQL Databases

Document Oriented Databases:

Document oriented databases treat a document as a whole and avoid splitting a document in its constituent name/value pairs. At a collection level, this allows for putting together a diverse set of documents into a single collection. Document databases allow indexing of documents on the basis of not only its primary identifier but also its properties. Different open-source document databases are available today but the most prominent among the available options are MongoDB and CouchDB. In fact, **MongoDB** has become one of the most popular NoSQL databases.

Graph Based Databases:

A graph database uses graph structures with nodes, edges, and properties to represent and store data. By definition, a graph database is any storage system that provides index-free adjacency. This means that every element contains a direct pointer to its adjacent element and no index lookups are necessary. General graph databases that can store any graph are distinct from specialized graph databases such as triple-stores and network databases. Indexes are used for traversing the graph.

Column Based Databases:

The column-oriented storage allows data to be stored effectively. It avoids consuming space when storing nulls by simply not storing a column when a value doesn't exist for that column. Each unit of data can be thought of as a set of key/value pairs, where the unit itself is identified with the help of a primary identifier, often referred to as the primary key. **Bigtable** and its clones tend to call this primary key the row-key.

Key Value Databases:

The key of a key/value pair is a unique value in the set and can be easily looked up to access the data. Key/value pairs are of varied types: some keep the data in memory and some provide the capability to persist the data to disk. A simple, yet powerful, key/value store is Oracle's Berkeley DB.

Popular NoSQL Databases

Let us summarize some popular NoSQL databases that falls in the above categories respectively.

- **Document Oriented Databases** − **MongoDB**, HBase, Cassandra, Amazon SimpleDB, Hypertable, etc.
- **Graph Based Databases** − Neo4j, OrientDB, Facebook Open Graph, FlockDB, etc.
- **Column Based Databases** − CouchDB, OrientDB, etc.
- **Key Value Databases** − Membase, Redis, MemcacheDB, etc.

Conclusion

In this article, we learnt about what NoSQL database technology is and how it primarily differs from a RDBMS implementation. We then explored various types of NoSQL databases, their applications and some of the most popular databases of each type.

A lot of organizations today are adapting to such databases for their huge datasets and high-scale applications. This shows that NoSQL is definitely going to be the next big thing in web and database technologies which has the potential to break the years long legacy of RDBMS.

7.3. SQL Vs NoSQL

As we know both SQL and NoSQL are the types of databases and on the basis of their implementation and nature, both are categorized as of two types.

The following are the important differences between SQL and NoSQL.

Sr. No.	Key	SQL	NoSQL
1	Type	SQL database is generally classified as a Relational database i.e. RDBMS.	While NOSQL database is known as non-relational or distributed database.
2	Language	As we already know SQL uses structured query language for its CRUD operation which is defined as SQL. This makes SQL database to store data in more structured form and also preferred for more complex operations which could get completed with complex SQL queries.	NoSQL database on other hand has dynamic schema for unstructured data. Data stored in this type of database is not structured and could be stored in either of forms such as document-oriented, column-oriented, graph-based or organized as a KeyValue store. This syntax can be varied from DB to DB.
3	Scalability	SQL database can extends its capacity on single server by increasing things like RAM, CPU or SSD i.e we can say that SQL dbs could be scalable in vertical as their storage could be increase for the same server by enhancing	In order to increase the capacity of NOSQL dbs we required to install new servers parallel to the parent server i.e NOSQL dbs could be scalable in horizontal and this made them more preferable choice for large

Sr. No.	Key	SQL	NoSQL
		its storage components.	or ever-changing data sets.
4	Internal implementation	SQL follows ACID properties for its operations which is abbreviation of Atomicity, Consistency, Isolation and Durability.	On other hand NOSQL is based on Brewers CAP theorem which maily focus on Consistency, Availability and Partition tolerance.
5	Performance and suited for	SQL databases are best suited for complex queries but are not preferred for hierarchical large data storage.	NoSQL databases are not so good for complex queries because these are not as powerful as SQL queries but are best suited for hierarchical large data storage.
6	**Examples**	SQL dbs is implemented in both open source and commercial Database such as like Postgres & MySQL as open source and Oracle and Sqlite as commercial.	On other hand NOSQL is purely open source and **MongoDB**, BigTable, Redis, RavenDB, Cassandra, Hbase, Neo4j, CouchDB are the main implementation of it.

7.4. Introduction to MongoDB with python

Python MongoDB

Python can be used in database applications.

One of the most popular NoSQL database is MongoDB.

MongoDB

MongoDB stores data in JSON-like documents, which makes the database very flexible and scalable.

To be able to experiment with the code examples in this tutorial, you will need access to a MongoDB database.

You can download a free MongoDB database at https://www.mongodb.com. Or get started right away with a MongoDB cloud service at https://www.mongodb.com/cloud/atlas.

PyMongo

Python needs a MongoDB driver to access the MongoDB database.

In this tutorial we will use the MongoDB driver "PyMongo".

We recommend that you use PIP to install "PyMongo".

PIP is most likely already installed in your Python environment.

Navigate your command line to the location of PIP, and type the following:

Download and install "PyMongo":

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>python -m pip install pymongo
```

Now you have downloaded and installed a mongoDB driver.

Test PyMongo

To test if the installation was successful, or if you already have "pymongo" installed, create a Python page with the following content:

demo_mongodb_test.py:

```
import pymongo
```

Python MongoDB Create Database

Creating a Database

To create a database in MongoDB, start by creating a MongoClient object, then specify a connection URL with the correct ip address and the name of the database you want to create.

MongoDB will create the database if it does not exist, and make a connection to it.

Example

Create a database called "mydatabase":

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
```

```
mydb = myclient["mydatabase"]
```

Important: In MongoDB, a database is not created until it gets content!

MongoDB waits until you have created a collection (table), with at least one document (record) before it actually creates the database (and collection).

Check if Database Exists

Remember: In MongoDB, a database is not created until it gets content, so if this is your first time creating a database, you should complete the next two chapters (create collection and create document) before you check if the database exists!

You can check if a database exist by listing all databases in you system:

Example

Return a list of your system's databases:

```
print(myclient.list_database_names())
```

Or you can check a specific database by name:

Example

Check if "mydatabase" exists:

```
dblist = myclient.list_database_names()
if "mydatabase" in dblist:
  print("The database exists.")
```

7.5. Exploring Collections and Documents &

7.6. Performing basic CRUD operations with MongoDB and python

Python MongoDB Create Collection

A **collection** in MongoDB is the same as a **table** in SQL databases.

Creating a Collection

To create a collection in MongoDB, use database object and specify the name of the collection you want to create.

MongoDB will create the collection if it does not exist.

Example

Create a collection called "customers":

```
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
```

Important: In MongoDB, a collection is not created until it gets content!

MongoDB waits until you have inserted a document before it actually creates the collection.

Check if Collection Exists

Remember: In MongoDB, a collection is not created until it gets content, so if this is your first time creating a collection, you should complete the next chapter (create document) before you check if the collection exists!

You can check if a collection exist in a database by listing all collections:

Example

Return a list of all collections in your database:

```
print(mydb.list_collection_names())
```

Or you can check a specific collection by name:

Example

Check if the "customers" collection exists:

```
collist = mydb.list_collection_names()
if "customers" in collist:
  print("The collection exists.")
```

Insert Into Collection

To insert a record, or *document* as it is called in MongoDB, into a collection, we use the insert_one() method.

The first parameter of the insert_one() method is a dictionary containing the name(s) and value(s) of each field in the document you want to insert.

Example

Insert a record in the "customers" collection:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
mydict = { "name": "John", "address": "Highway 37" }
x = mycol.insert_one(mydict)
```

Return the _id Field

The insert_one() method returns a InsertOneResult object, which has a property, inserted_id, that holds the id of the inserted document.

Example

Insert another record in the "customers" collection, and return the value of the _id field:

```
mydict = { "name": "Peter", "address": "Lowstreet 27" }
x = mycol.insert_one(mydict)
print(x.inserted_id)
```

If you do not specify an _id field, then MongoDB will add one for you and assign a unique id for each document.

In the example above no _id field was specified, so MongoDB assigned a unique _id for the record (document).

Insert Multiple Documents

To insert multiple documents into a collection in MongoDB, we use the insert_many() method.

The first parameter of the insert_many() method is a list containing dictionaries with the data you want to insert:

Example:

```
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
mylist = [
  { "name": "Amy", "address": "Apple st 652"},
  { "name": "Hannah", "address": "Mountain 21"},
  { "name": "Michael", "address": "Valley 345"},
  { "name": "Sandy", "address": "Ocean blvd 2"},
  { "name": "Betty", "address": "Green Grass 1"},
  { "name": "Richard", "address": "Sky st 331"},
  { "name": "Susan", "address": "One way 98"},
  { "name": "Vicky", "address": "Yellow Garden 2"},
  { "name": "Ben", "address": "Park Lane 38"},
  { "name": "William", "address": "Central st 954"},
  { "name": "Chuck", "address": "Main Road 989"},
  { "name": "Viola", "address": "Sideway 1633"}
]
x = mycol.insert_many(mylist)
#print list of the _id values of the inserted documents:
print(x.inserted_ids)
```

The insert_many() method returns a InsertManyResult object, which has a property, inserted_ids, that holds the ids of the inserted documents.

Insert Multiple Documents, with Specified IDs

If you do not want MongoDB to assign unique ids for you document, you can specify the _id field when you insert the document(s).

Remember that the values has to be unique. Two documents cannot have the same _id.

Example

```
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
mylist = [
  { "_id": 1, "name": "John", "address": "Highway 37"},
  { "_id": 2, "name": "Peter", "address": "Lowstreet 27"},
  { "_id": 3, "name": "Amy", "address": "Apple st 652"},
  { "_id": 4, "name": "Hannah", "address": "Mountain 21"},
  { "_id": 5, "name": "Michael", "address": "Valley 345"},
  { "_id": 6, "name": "Sandy", "address": "Ocean blvd 2"},
  { "_id": 7, "name": "Betty", "address": "Green Grass 1"},
  { "_id": 8, "name": "Richard", "address": "Sky st 331"},
  { "_id": 9, "name": "Susan", "address": "One way 98"},
  { "_id": 10, "name": "Vicky", "address": "Yellow Garden 2"},
  { "_id": 11, "name": "Ben", "address": "Park Lane 38"},
  { "_id": 12, "name": "William", "address": "Central st 954"},
  { "_id": 13, "name": "Chuck", "address": "Main Road 989"},
  { "_id": 14, "name": "Viola", "address": "Sideway 1633"}
]
x = mycol.insert_many(mylist)
#print list of the _id values of the inserted documents:
print(x.inserted_ids)
```

Python MongoDB Find

In MongoDB we use the **find** and **find_one** methods to find data in a collection.

Just like the **SELECT** statement is used to find data in a table in a MySQL database.

Find One

To select data from a collection in MongoDB, we can use the find_one() method.

The find_one() method returns the first occurrence in the selection.

Example

Find the first document in the customers collection:

```
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
x = mycol.find_one()
print(x)
```

Find All

To select data from a table in MongoDB, we can also use the find() method.

The find() method returns all occurrences in the selection.

The first parameter of the find() method is a query object. In this example we use an empty query object, which selects all documents in the collection.

No parameters in the find() method gives you the same result as **SELECT *** in MySQL.

Example

Return all documents in the "customers" collection, and print each document:

```
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
for x in mycol.find():
  print(x)
```

Return Only Some Fields

The second parameter of the find() method is an object describing which fields to include in the result.

This parameter is optional, and if omitted, all fields will be included in the result.

Example

Return only the names and addresses, not the _ids:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
for x in mycol.find({},{ "_id": 0, "name": 1, "address": 1 }):
  print(x)
```

You are not allowed to specify both 0 and 1 values in the same object (except if one of the fields is the _id field). If you specify a field with the value 0, all other fields get the value 1, and vice versa:

Example

This example will exclude "address" from the result:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

for x in mycol.find({},{ "address": 0 }):
  print(x)
```

Example

You get an error if you specify both 0 and 1 values in the same object (except if one of the fields is the _id field):

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

for x in mycol.find({},{ "name": 1, "address": 0 }):
  print(x)
```

Python MongoDB Query

Filter the Result

When finding documents in a collection, you can filter the result by using a query object.

The first argument of the find() method is a query object, and is used to limit the search.

Example

Find document(s) with the address "Park Lane 38":

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": "Park Lane 38" }

mydoc = mycol.find(myquery)

for x in mydoc:
  print(x)
```

Advanced Query

To make advanced queries you can use modifiers as values in the query object.

E.g. to find the documents where the "address" field starts with the letter "S" or higher (alphabetically), use the greater than modifier: {"$gt": "S"}:

Example

Find documents where the address starts with the letter "S" or higher:

```
import pymongo


myclient = pymongo.MongoClient("mongodb://localhost:27017/")

mydb = myclient["mydatabase"]

mycol = mydb["customers"]
```

```
myquery = { "address": { "$gt": "S" } }
mydoc = mycol.find(myquery)

for x in mydoc:
  print(x)
```

Filter with Regular Expressions

You can also use regular expressions as a modifier.

Regular expressions can only be used to query *strings*.

To find only the documents where the "address" field starts with the letter "S", use the regular expression {"$regex": "^S"}:

Example

Find documents where the address starts with the letter "S":

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": { "$regex": "^S" } }

mydoc = mycol.find(myquery)

for x in mydoc:
  print(x)
```

Python MongoDB Sort

Sort the Result

Use the sort() method to sort the result in ascending or descending order.

The sort() method takes one parameter for "fieldname" and one parameter for "direction" (ascending is the default direction).

Example

Sort the result alphabetically by name:

```
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
mydoc = mycol.find().sort("name")
for x in mydoc:
  print(x)
```

Sort Descending

Use the value -1 as the second parameter to sort descending.

```
sort("name", 1) #ascending
sort("name", -1) #descending
```

Example

Sort the result reverse alphabetically by name:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
mydoc = mycol.find().sort("name", -1)
for x in mydoc:
  print(x)
```

Python MongoDB Update

Update Collection

You can update a record, or document as it is called in MongoDB, by using the update_one() method.

The first parameter of the update_one() method is a query object defining which document to update.

Note: If the query finds more than one record, only the first occurrence is updated.

The second parameter is an object defining the new values of the document.

Example

Change the address from "Valley 345" to "Canyon 123":

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
myquery = { "address": "Valley 345" }
newvalues = { "$set": { "address": "Canyon 123" } }

mycol.update_one(myquery, newvalues)

#print "customers" after the update:
for x in mycol.find():
  print(x)
```

Update Many

To update *all* documents that meets the criteria of the query, use the update_many() method.

Example

Update all documents where the address starts with the letter "S":

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": { "$regex": "^S" } }
newvalues = { "$set": { "name": "Minnie" } }

x = mycol.update_many(myquery, newvalues)

print(x.modified_count, "documents updated.")
```

Python MongoDB Limit

Limit the Result

To limit the result in MongoDB, we use the limit() method.

The limit() method takes one parameter, a number defining how many documents to return.

Consider you have a "customers" collection:

Customers

```
{'_id': 1, 'name': 'John', 'address': 'Highway37'}
{'_id': 2, 'name': 'Peter', 'address': 'Lowstreet 27'}
{'_id': 3, 'name': 'Amy', 'address': 'Apple st 652'}
{'_id': 4, 'name': 'Hannah', 'address': 'Mountain 21'}
{'_id': 5, 'name': 'Michael', 'address': 'Valley 345'}
{'_id': 6, 'name': 'Sandy', 'address': 'Ocean blvd 2'}
{'_id': 7, 'name': 'Betty', 'address': 'Green Grass 1'}
{'_id': 8, 'name': 'Richard', 'address': 'Sky st 331'}
{'_id': 9, 'name': 'Susan', 'address': 'One way 98'}
{'_id': 10, 'name': 'Vicky', 'address': 'Yellow Garden 2'}
{'_id': 11, 'name': 'Ben', 'address': 'Park Lane 38'}
{'_id': 12, 'name': 'William', 'address': 'Central st 954'}
{'_id': 13, 'name': 'Chuck', 'address': 'Main Road 989'}
{'_id': 14, 'name': 'Viola', 'address': 'Sideway 1633'}
```

Example

Limit the result to only return 5 documents:

```
import pymongo


myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
myresult = mycol.find().limit(5)
#print the result:
for x in myresult:
  print(x)
```

Python MongoDB Delete Document OR Record

Delete Document

To delete one document, we use the delete_one() method.

The first parameter of the delete_one() method is a query object defining which document to delete.

Note: If the query finds more than one document, only the first occurrence is deleted.

Example

Delete the document with the address "Mountain 21":

```
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
myquery = { "address": "Mountain 21" }
mycol.delete_one(myquery)
```

Delete Many Documents

To delete more than one document, use the delete_many() method.

The first parameter of the delete_many() method is a query object defining which documents to delete.

Example

Delete all documents were the address starts with the letter S:

```
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
myquery = { "address": {"$regex": "^S"} }
x = mycol.delete_many(myquery)
print(x.deleted_count, " documents deleted.")
```

Delete All Documents in a Collection

To delete all documents in a collection, pass an empty query object to the delete_many() method:

Example

Delete all documents in the "customers" collection:

```
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
x = mycol.delete_many({})
print(x.deleted_count, " documents deleted.")
```

Python MongoDB Drop Collection

Delete Collection

You can delete a table, or collection as it is called in MongoDB, by using the drop() method.

Example

Delete the "customers" collection:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mycol.drop()
```

MongoDB: importing .json and .csv collection files

First you must navigate in to MongoDB installed directory

C:\Program Files\MongoDB\Server\3.2\bin>

Then with the following command you can import these files

```
mongoimport --host localhost:27017 --db mydatabase --collection docs < c:\hotels.json
```