**ACID Properties in DBMS**

A transaction is a single logical unit of work which accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.  
In order to maintain consistency in a database, before and after transaction, certain properties are followed. These are called **ACID** properties.

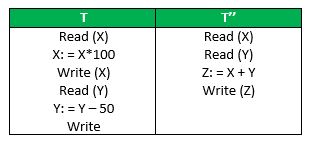
**Atomicity**  
By this, we mean that either the entire transaction takes place at once or doesn’t happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves following two operations.  
—**Abort**: If a transaction aborts, changes made to database are not visible.  
—**Commit**: If a transaction commits, changes made are visible.  
Atomicity is also known as the ‘All or nothing rule’.  
Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.



If the transaction fails after completion of **T1** but before completion of **T2**.( say, after **write(X)** but before **write(Y)**), then amount has been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the transaction must be executed in entirety in order to ensure correctness of database state.

**Consistency**  
This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to correctness of a database. Referring to the example above,  
The total amount before and after the transaction must be maintained.  
Total **before T** occurs = **500 + 200 = 700**.  
Total **after T occurs** = **400 + 300 = 700**.  
Therefore, database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result T is incomplete.

**Isolation**  
This property ensures that multiple transactions can occur concurrently without leading to inconsistency of database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.  
Let **X**= 500, **Y** = 500.  
Consider two transactions **T** and **T”.**



Suppose **T** has been executed till **Read (Y)** and then **T’’** starts. As a result , interleaving of operations takes place due to which **T’’** reads correct value of **X**but incorrect value of **Y** and sum computed by  
**T’’: (X+Y = 50, 000+500=50, 500)**  
is thus not consistent with the sum at end of transaction:   
**T: (X+Y = 50, 000 + 450 = 50, 450)**.  
This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after a they have been made to the main memory.

**Durability:**  
This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even is system failure occurs. These updates now become permanent and are stored in a non-volatile memory. The effects of the transaction, thus, are never lost.

The **ACID** properties, in totality, provide a mechanism to ensure correctness and consistency of a database in a way such that each transaction is a group of operations that acts a single unit, produces consistent results, acts in isolation from other operations and updates that it makes are durably stored.

**MongoDB**

**SQL vs NoSQL**

In case you aren’t familiar with it, MongoDB is a [NoSQL](https://en.wikipedia.org/wiki/NoSQL) database which has become pretty popular throughout the industry in recent years. NoSQL databases provide features of retrieval and storage of data in a much different way than their relational database counterparts.

For decades, SQL databases used to be one of the only choices for developers looking to build large, scalable systems. However, the ever-increasing need for the ability to store complex data structures led to the birth of NoSQL databases, which allow a developer to store heterogeneous and structure-less data.

When it comes to the choices available, most people have to ask themselves the ultimate question, “SQL or NoSQL?” Both SQL and NoSQL have their strengths and weaknesses, and you should choose the one that fits your application requirements the best. Here are a few differences between the two:

**SQL**

* The model is of a relational nature
* Data is stored in tables
* Suitable for solutions where every record is of the same kind and possesses the same properties
* Adding a new property means you have to alter the whole schema
* The schema is very strict
* [ACID](https://en.wikipedia.org/wiki/ACID) transactions are supported
* Scales well vertically

**NoSQL**

* The model is non-relational
* May be stored as JSON, key-value, etc. (depending on type of NoSQL database)
* Not every record has to be of the same nature, making it very flexible
* Add new properties to data without disturbing anything
* No schema requirements to adhere to
* Support for ACID transactions can vary depending on which NoSQL DB is used
* Consistency can vary
* Scales well horizontally

There are many other differences between the two types of databases but those mentioned above are some of the more important differences to know.

Depending on your specific scenario, the use of a SQL database may be preferred, while in other scenarios NoSQL is the more obvious choice to make. When choosing a database you should consider the strengths and weaknesses of each database carefully.

One of the great things about NoSQL is that there are many different types of databases to choose from, and each has its own use-cases:

* Key-Value Store: [DynamoDB](https://aws.amazon.com/dynamodb/)
* Document Store: [CouchDB](http://couchdb.apache.org/), [MongoDB](https://www.mongodb.com/), [RethinkDB](https://www.rethinkdb.com/)
* Column Store: [Cassandra](http://cassandra.apache.org/)
* Data-Structures: [Redis](https://redis.io/)

There are quite a few more, but these are some of the more common types.

In recent years, SQL and NoSQL databases have even begun to merge. For example, PostgreSQL now supports storing and querying JSON data, much like Mongo. With this, you can now achieve much of the same with Postgres as you can with Mongo, but you still don’t get many of the Mongo advantages (like horizontal scaling and the simple interface, etc.). As [this article](https://www.compose.com/articles/is-postgresql-your-next-json-database/) puts it:

If your active data sits in the relational schema comfortably and the JSON content is a cohort to that data then you should be fine with PostgreSQL and it’s much more efficient JSONB representation and indexing capabilities. If though, your data model is that of a collection of mutable documents then you probably want to look at a database engineered primarily around JSON documents like MongoDB or RethinkDB.

**MongoDB**

Let’s now shift our attention to the main focus of this article and shed some light on the specifics of MongoDB.

MongoDB is a document-oriented, open-source database program that is platform-independent. MongoDB, like some other NoSQL databases (but not all!), stores its data in documents using a JSON structure. This is what allows the data to be so flexible and not require a schema.

Some of the more important features are:

* You have support for many of the standard query types, like matching (==), comparison (<, >), or even regex
* You can store virtually any kind of data - be it structured, partially structured, or even polymorphic
* To scale up and handle more queries, just add more machines
* It is highly flexible and agile, allowing you to quickly develop your applications
* Being a document-based database means you can store all the information regarding your model in a single document
* You can change the schema of your database on the fly
* Many relational database functionalities are also available in MongoDB (e.g. indexing)

As for the operations side of things, there are quite a few tools and features for MongoDB that you just can’t find with any other database system:

* Whether you need a standalone server or complete clusters of independent servers, MongoDB is as scalable as you need it to be
* MongoDB also provides load balancing support by automatically moving data across the various shards
* It has automatic failover support - in case your primary server goes down, a new primary will be up and running automatically
* The MongoDB Management Service or MMS is a very nice web tool that provides you with the ability to track your machines
* Thanks to the memory mapped files, you’ll save quite a bit of RAM, unlike relational databases

If you take advantage of the indexing features, much of this data will be kept in memory for quick retrieval. And even without indexing on specific document keys, Mongo caches quite a bit of data using the [least recently used](https://en.wikipedia.org/wiki/Cache_replacement_policies#Least_Recently_Used_.28LRU.29) method.

While at first Mongo may seem like it’s the solution to many of our database problems, it isn’t without its drawbacks. One common drawback you’ll hear about Mongo is its lack of support for ACID transactions. Mongo *does* support ACID transactions in a [limited sense](https://docs.mongodb.com/v3.4/core/write-operations-atomicity/), but not in all cases. At the single-document level, ACID transactions are supported (which is where most transactions take place anyway). However, transactions dealing with multiple documents are not supported due to Mongo’s distributed nature.

Mongo also lacks support for native joins, which must be done manually (and therefore much more slowly). Documents are meant to be all-encompassing, which means, in general, they shouldn’t need to reference other documents. In the real world this doesn’t always work as much of the data we work with is relational by nature. Therefore many will argue that Mongo should be used as a complementary database to a SQL DB, but as you use MongoDB you’ll find that is not necessarily true.

**PyMongo**

Now that we’ve described what MongoDB is exactly, let’s find out how you’d actually use it with Python. The official driver published by the Mongo developers is called [PyMongo](https://pypi.python.org/pypi/pymongo/). This is a good place to start when first firing Python up with MongoDB. We’ll be going through some examples here, but you should also check out the [complete documentation](https://api.mongodb.com/python/current/) since we won’t be able to cover everything.

The first thing you’ll want to do is to install PyMongo in your [virtual environment](https://realpython.com/python-virtual-environments-a-primer/). The easiest way to do this is with pip:

$ pip install pymongo==3.4.0

**NOTE:** For a more comprehensive guide check out the [Installing / Upgrading](https://api.mongodb.com/python/3.4.0/installation.html) page of the docs and follow the steps there to get set up.

Once you are done with the setup, start your Python console and run the following command:

>>> import pymongo

If it runs without raising any exception within the Python shell then your install worked just fine. If not, then carefully perform the steps again. Exit the shell once done.

Next, you have to install the actual MongoDB database.

If you’re on a Mac, we recommend using [Homebrew](http://brew.sh/), but you may have a different preference. If using Homebrew, run this command:

$ brew install mongodb

Once done, follow the directions [here](https://docs.mongodb.com/v3.4/tutorial/install-mongodb-on-os-x/#run-mongodb) to set up the data directory for storing data locally.

If you’re not using a Mac, you can find some great guides on installation from the [Install MongoDB](https://docs.mongodb.com/v3.4/installation/) page of the official docs. There you’ll find tutorials on installing MongoDB for Linux, OS X, and Windows.

Once installed, within a new terminal window, use the following command to start the Mongo daemon:

$ mongod

**NOTE:** Depending on your install method, you may need to run mongod with sudo.

Now let’s get started with the basics of PyMongo.

Establishing a Connection

To establish a connection we’ll use the MongoClient object.

The first thing that we need to do in order to establish a connection is import the MongoClient class. We’ll use this to communicate with the running database instance. Use the following code to do so:

>>> from pymongo import MongoClient

>>> client = MongoClient()

Using the snippet above, the connection will be established to the default host (localhost) and port (27017). You can also specify the host and/or port using:

>>> client = MongoClient('localhost', 27017)

Or just use the Mongo URI format:

>>> client = MongoClient('mongodb://localhost:27017')

All of these calls to MongoClient will do the same thing; it just depends on how explicit you want to be in your code.

Accessing Databases

Once you have a connected instance of MongoClient, you can access any of the databases within that Mongo server. To specify which database you actually want to use, you can access it as an attribute:

db = client.pymongo\_test

Or you can also use the dictionary-style access:

db = client['pymongo\_test']

It doesn’t actually matter if your specified database has been created yet. By specifying this database name and saving data to it, you create the database automatically.

Inserting Documents

Storing data in your database is as easy as calling just two lines of code. The first line specifies which [collection](https://docs.mongodb.com/v3.4/reference/glossary/#term-collection) you’ll be using (posts in the example below). In MongoDB terminology, a collection is a group of documents that are stored together within the database. Collections and documents are [akin](https://docs.mongodb.com/v3.4/reference/sql-comparison/) to SQL tables and rows, respectively. Retrieving a collection is as easy as getting a database.

The second line is where you actually insert the data in to the collection using the insert\_one() method:

posts = db.posts

post\_data = {

'title': 'Python and MongoDB',

'content': 'PyMongo is fun, you guys',

'author': 'Scott'

}

result = posts.insert\_one(post\_data)

print('One post: {0}'.format(result.inserted\_id))

We can even insert many documents at a time, which is much faster than using insert\_one()if you have many documents to add to the database. The method to use here is insert\_many(). This method takes an array of document data:

post\_1 = {

'title': 'Python and MongoDB',

'content': 'PyMongo is fun, you guys',

'author': 'Scott'

}

post\_2 = {

'title': 'Virtual Environments',

'content': 'Use virtual environments, you guys',

'author': 'Scott'

}

post\_3 = {

'title': 'Learning Python',

'content': 'Learn Python, it is easy',

'author': 'Bill'

}

new\_result = posts.insert\_many([post\_1, post\_2, post\_3])

print('Multiple posts: {0}'.format(new\_result.inserted\_ids))

When ran, you should see something like:

One post: 584d947dea542a13e9ec7ae6

Multiple posts: [

ObjectId('584d947dea542a13e9ec7ae7'),

ObjectId('584d947dea542a13e9ec7ae8'),

ObjectId('584d947dea542a13e9ec7ae9')

]

**NOTE:** Don’t worry that your ObjectIds don’t match those shown above. They are dynamically generated when you insert data and consist of a Unix epoch, machine identifier, and other unique data.

Retrieving Documents

To retrieve a document, we’ll use the find\_one() method. The lone argument that we’ll use here (although it supports many more) is a dictionary that contains fields to match. In our example below, we want to retrieve the post that was written by Bill:

bills\_post = posts.find\_one({'author': 'Bill'})

print(bills\_post)

Run this:

{

'author': 'Bill',

'title': 'Learning Python',

'content': 'Learn Python, it is easy',

'\_id': ObjectId('584c4afdea542a766d254241')

}

You may have noticed that the post’s ObjectId is set under the \_id key, which we can later use to uniquely identify it if needed. If we want to find more than one document, we can use the find() method. This time, let’s find all of the posts written by Scott:

scotts\_posts = posts.find({'author': 'Scott'})

print(scotts\_posts)

Run:

<pymongo.cursor.Cursor object at 0x109852f98>

The main difference here is that the document data isn’t returned directly to us as an array. Instead we get an instance of the [Cursor](https://api.mongodb.com/python/3.4.0/api/pymongo/cursor.html#pymongo.cursor.Cursor) object. This Cursor is an iterable object that contains quite a few helper methods to help you work with the data. To get each document, just iterate over the result:

for post in scotts\_posts:

print(post)

**MongoEngine**

While PyMongo is very easy to use and overall a great library, it’s probably a bit too low-level for many projects out there. Put another way, you’ll have to write a lot of your own code to *consistently* save, retrieve, and delete objects.

One library that provides a higher abstraction on top of PyMongo is [MongoEngine](http://mongoengine.org/). MongoEngine is an object document mapper (ODM), which is roughly equivalent to a SQL-based object relational mapper (ORM). The abstraction provided by MongoEngine is class-based, so all of the models you create are classes.

While quite a few Python libraries exist to help you work with MongoDB, MongoEngine is one of the better ones as it has a nice mix of features, flexibility, and community support without becoming too opinionated.

To install, use pip:

$ pip install mongoengine==0.10.7

Once installed, we need to direct the library to connect with our running instance of Mongo. For this we will have to use the connect() function and pass the host and port of the MongoDB database to it. Within the Python shell, type:

from mongoengine import \*

connect('mongoengine\_test', host='localhost', port=27017)

Here we specify the name of our database and location. Since we’re still using the default host and port, you can omit these parameters.

Defining a Document

To set up our document object, we need to define what data we want our document object to have. Similar to many other ORMs, we’ll do this by subclassing the Document class and providing the types of data we want:

import datetime

class Post(Document):

title = StringField(required=True, max\_length=200)

content = StringField(required=True)

author = StringField(required=True, max\_length=50)

published = DateTimeField(default=datetime.datetime.now)

**NOTE:** One of the more difficult tasks with database models is validating data. How do you make sure that the data you’re saving conforms to some format you need? Just because a database is said to be *schema-less* doesn’t mean it is *schema-free*.

In this simple model, we’ve told MongoEngine that we expect a Post instance to have a title, content, an author, and the date it was published. Now the base Document object can use that information to validate the data we provide it.

So, for example, if we try to save a Post without a title then it’ll throw an Exception and let us know. We can take this even further and add more restrictions, like string length. Notice that some of the fields have a max\_length parameter set. This tells the Document, as you probably guessed, to only allow a maximum string length of however many characters we specify. There are quite a few more parameters like this we can set, including:

* db\_field: Specify a different field name
* required: Make sure this field is set
* default: Use the given default value if no other value is given
* unique: Make sure no other document in the collection has the same value for this field
* choices: Make sure the field’s value is equal to one of the values given in an array

Each field type has its own set of parameters, so be sure to [check the documentation](http://docs.mongoengine.org/guide/defining-documents.html#field-arguments) for more info.

Saving Documents

To save a document to our database, we’ll use the save() method. If the document already exists in the database, then all of the changes will be made on the atomic level to the existing document. If it doesn’t exist, however, then it will be created.

Here is an example of creating and saving a document:

post\_1 = Post(

title='Sample Post',

content='Some engaging content',

author='Scott'

)

post\_1.save() # This will perform an insert

print(post\_1.title)

post\_1.title = 'A Better Post Title'

post\_1.save() # This will perform an atomic edit on "title"

print(post\_1.title)

A few things to note about the .save() call:

* PyMongo will perform validation when you call .save(). This means it will check the data you’re saving against the schema you declared in the class. If the schema (or a constraint) is violated, then an exception is thrown and the data is not saved.
* Since Mongo doesn’t support true transactions, there is no way to “roll back” the .save() call like you can in SQL databases. Although you can get close to performing transactions with [two phase commits](https://docs.mongodb.com/v3.4/tutorial/perform-two-phase-commits/), they still don’t support rollbacks.

What happens when you leave off the title?

post\_2 = Post(content='Content goes here', author='Michael')

post\_2.save()

You should see the following exception:

raise ValidationError(message, errors=errors)

mongoengine.errors.ValidationError:

ValidationError (Post:None) (Field is required: ['title'])

Object Oriented Features

With MongoEngine being object oriented, you can also add methods to your subclassed document. Consider the following example where a function is used to modify the default queryset (which returns all objects of the collection). By using this, we can apply a default filter to the class and get only the desired objects:

class Post(Document):

title = StringField()

published = BooleanField()

@queryset\_manager

def live\_posts(clazz, queryset):

return queryset.filter(published=True)

Referencing Other Documents

You can also use the ReferenceField object to create a reference from one document to another. MongoEngine handles the lazy de-referencing automatically upon access, which is more robust and less error-prone than having to remember to do it yourself everywhere in your code. An example:

class Author(Document):

name = StringField()

class Post(Document):

author = ReferenceField(Author)

Post.objects.first().author.name

In the code above, using a document reference, we can easily find the author of the first post.

There are quite a few more field classes (and parameters) than what we introduced here, so be sure to check out the [documentation on Fields](http://docs.mongoengine.org/apireference.html#fields) for more info.

Note: You can also use Mlab website for creating MongoDb Databases

Here is the video URl for that: <https://www.youtube.com/watch?v=ZnOko_y1CfI>

And You can use this script with that: <https://github.com/Satyam-Bhalla/Acadview-Python/blob/master/Database/mongoDB.py>