## MongoDB Study Notes

MongoDB: General purpose document database. Structures data as documents which are similar to JSON objects.

### Structure

*Document:* Basic unit of data in MongoDB. They are displayed in JSON format but stored internally in a BSON (Binary JSON) format. BSON supports more data types like Dates. The values in a document can be any data type, including strings, objects, arrays, booleans, nulls, dates, ObjectIds, and more. Documents look like following

{

\_id:< ObjectID >  
<key>:<value>  
<key>:<value>  
<key>:<value>

}

*ObjectID:* Special data type used in MongoDB to create unique identifiers. Every document has a unique ID that acts as a primary key, called “\_id”. ObjectId values are 12 bytes in length, consisting of:

* A 4-byte timestamp, representing the ObjectId's creation, measured in seconds since the Unix epoch (Unix time 0 being UTC midnight on January 1, 1970). We can access creation time of the ObjectID by using ObjectId.getTimestamp()
* A 5-byte random value generated once per process. Random value is unique to machine & process.
* A 3-byte incrementing counter, initialized to a random value.

*Collection:* Grouping of documents. Documents in a collection mostly have the same format but they need not be exactly similar, as MongoDB has a flexible Schema model (polymorphism). Documents in a collection can have different fields and fields can have different data types.

*Database:* A container of many collections.

Optional Schema Validation: Set constraints on the structure of documents. But this is optional. This is for more control on the schema.

### Data

*BSON Date* is a 64-bit integer that represents the number of milliseconds since the Unix epoch (Jan 1, 1970). This results in a date range of about 290 million years into the past and future. It uses UTC date time.

*Timestamps:* internal timestamp type is a 64 bit value. The most significant 32 bits are a time\_t value (seconds since the Unix epoch), the least significant 32 bits are an incrementing ordinal for operations within a given second.

Arrays are an important data type. We need to be careful to avoid bloating of documents due to big arrays.

### Ecosystem

*MongoDB Shell* (mongosh), is a Node.js REPL (Read Evaluate Print Loop) environment that allows users to interact with MongoDB deployments. As it's built on top of Node.js REPL, so it gives users access to the Node.js API and npm packages. So it gives access to JS variables, functions, conditionals, loops and control flow inside the shell.

*Atlas*: A multi cloud database service that helps build deploying, managing applications on data (a developer data platform). Replication and sharding is handled by Atlas.

*Organization*: Allows to group and define users and teams.

*Projects*: Help to organize and define resources. Create separate projects for development, testing and production environments.

*Cluster*: a group of connected MongoDB instances, or nodes, that work together to provide redundancy, high availability, and scalability for handling large volumes of data

Atlas CLI is a unified command line interface to manage MongoDB Atlas, including Atlas Search and Vector Search

*Compass*: is a GUI that allows querying, composing pipelines and analyzing data

*Charts*: a data visualization tool offered by MongoDB

*Data Lake*: to query, transform, and move data across Amazon S3 and Atlas clusters

*MongoDB drivers* are used to connect, query and update MongoDB databases from programming languages such as Python, Node.JS, C#, Java and others. They run database operations on behalf of client applications. Python drivers are

1. Pymongo: For synchronous applications
2. Motor: For asynchronous applications
3. PyMongoArrow is a PyMongo extension for loading MongoDB query result sets as Numpy array or Panda Dataframes.
4. Mongoengine: community maintained Python object document mapper built over Pymongo. An ODM allows you to interact with MongoDB database using JavaScript objects. To create a collection model, define the schema, and use that model to create new instances, fetch them from the database, update them and delete them. Instead of using MongoDB commands to work with the data, you can use the methods provided by the model

MongoDB extension for VS Code is very helpful while coding. It has *Playground*. MongoDB Playgrounds are JavaScript environments where you can prototype queries, aggregations, and MongoDB commands. You can save playgrounds in your workspace and use them to document how your application interacts with MongoDB. MongoDB for VS Code interprets files with the \*.mongodb.js extension as playgrounds.

### Connect to a Database

Connection String: Allows us to connect to a cluster on Atlas from Mongo Shell, Mongo Compass or any other application. We need only one MongoClient instance per Atlas cluster for an application. Having more than one MongoClient instance for a single Atlas cluster will cause unnecessary connections to the database, increasing costs and impacting performance. It comes in 2 types (standard and DNS)

1. Standard format: connect to standard clusters, replica sets or sharded clusters. For localhost servers we use this.
2. DNS seed format: A DNS server list, gives flexibility in deployment and server rotation without reconfiguring clients
3. Format for Atlas deployment is mongodb+srv://<userName>:<password>@<clusterName>.<portNumber>.mongodb.net/
4. In above, mongodb is the required prefix that identifies this as a standard connection string
5. we can specify connection options. Each option is separated by a ?, followed by the <name>=<value> pair option

### Data Operations

#### Cursor

Pointer to the result set of a query

Use cursor.sort() to return query results in a specified order. Within the parentheses of sort(), include an object that specifies the field(s) to sort by and the order of the sort. Use 1 for ascending order, and -1 for descending order.

db.collection.find(<query>, <projection>).sort(<sort>)

#### Projection

Projections are used to specify which fields are returned while querying. Here projection document specifies which are the fields to be included/excluded in the result. We cant combine both the below arguments except for \_id field that can be excluded in the first option below

1. Projection by inclusion: specify the fields to be included by using 1
2. Projection by exclusion: specify the fields to be excluded by using 0

db.collection.find( <query>, { <field> : 1 })

Use cursor.limit() to specify the maximum number of documents the cursor will return.

db.companies.find(<query>).limit(<number>)

#### Counting

We can count total documents in a collection or to count the number of documents that match a query.

db.collection.countDocuments( <query>, <options> )

#### Inserting

db.collection.insertOne(oneDocument)

In case collection is not already created, Mongo will automatically create a collection. In case –id field is not mentioned in document, Mongo will create a unique \_id

The Pymongo method is insert\_one(oneDocument)

db.collection .insertMany([array of comma seperated documents])

The Pymongo method is insert\_many([array of comma seperated documents])

#### Finding

To return a single document that matches a query, append findOne() (Pymongo: find\_one()) to the collection object. The findOne() method can accept a filter argument that specifies the query to be performed.

To return all documents that match a query we use find()

db.collection.find( <query>, <projection> )

To include a field, set its value to 1 in the projection document., to exclude 0

db.collection.find({field:”value”} / { field: {$operator:[value or array of values]} })

$in operator will check if field is equal to any one of the inputs in the array provided.

Comparison operators are $gt, $gte, $lt, $lte, $ne

Nested fields are accessed using dot – “field.nestedField” within quotations

$elemMatch() is used to find documents that have a particular value in a array, that matches all the specified query criteria. { <field>: { $elemMatch: { <query1>, <query2>, ... } } }

db.collection.find({field: {$elemMatch: value }})

Logical operators are $or and $and; they are mentioned as { operator:[ array of {field: value}]} }   
e.g. {$or: [{"item.name":"pens"}, {tag: "writing"}]}

We can also have a $and query implicitly by having comma separated conditions without {}.   
E.g. find({name:"pens", "tag": "writing"})

#### Replacing

Replace an entire document. The replaceOne() method takes the following parameters:

1. filter: A query that matches the document to replace.
2. replacement: The new document to replace the old one with.
3. options: An object that specifies options for the update.

This method returns a document containing an acknowledgement of the operation, a matched count, modified count, and an upserted ID

#### Updating

The updateOne(filter, updates, options) method accepts a filter document, an update document, and an optional options object. findAndModify({query: {}, update: {}, new = true,options) method updates and returns the updated document to us. To update multiple documents, use the updateMany() method

The $set operator replaces the value of a field with the specified value. If the field is not there, it will add this new field to the document. $unset removes a field from document.

The $push operator adds a new value to an array field. If the array is not there it will create a new one. Use the $each modifier with the $push operator to append multiple elements to an array, such as { $push: {field: { $each: ["element1", "element2"]}}

$pop Removes the first or last item from an array. You can specify 1 to remove the last item or -1 to remove the first item. { $pop: { arrayField: 1 } }

$addToSet Adds a value to an array only if the value does not already exist in the array (avoids duplicates).

$rename renames a field

A number can be incremented by the $inc operator { $inc: { counter: 1 }}. $mul multiples,

{upsert: true} is an option that creates a new document if no documents match the filtered criteria.

#### Deleting

deleteOne({filter}, options) and deleteMany({filter}, options).

In Pymongo its delete\_one() amd delete\_many(). They return a result that has a deleted\_count attribute.

#### Transactions

In MongoDB, an operation on a single document is atomic. For situations that require atomicity of reads and writes to multiple documents (in a single or multiple collections), MongoDB supports distributed transactions. If any operation in the transaction fails, the transaction aborts and all data changes made in the transaction are discarded without ever becoming visible.The callback API:

* starts a transaction
* executes the specified operations
* commits the result (or aborts on error)

In Python the steps are

# Step 1: Define the callback function that specifies the sequence of operations to perform inside the transactions. This function has an argument called session that is to be passed on to operations.

* In the callback, include the required parameter, which is the client session. You can also add additional parameters for your particular transaction if needed.
* Within the callback function, get references to the collections that the operations will take place on.
* Write the transaction operations. Note that you must pass the session to each operation.

# Step 2: Start a client session.   
*with client.start\_session() as session:*

# Step 3: Use *session*.*with\_transaction()* to start a transaction, execute the callback, and commit.

* with\_transaction has one required parameter, which is the callback function that specifies the sequence of operations to perform inside the transaction. Best practice for passing arbitrary arguments to the callback is to use a lambda function.

### Aggregation

* **Aggregation**: Collection and summary of data
* **Stage**: One of the built-in methods that can be completed on the data, but does not permanently alter it. It’s a single operation on the data.
* **Aggregation pipeline**: A series of stages completed on the data in order. Data can be filtered, grouped, sorted and transformed
* While using this inside python we need to mention “$operator” and “$field” and all in **“”.** While referring to variables/constants defined in Python code we need not use “”.

Syntax is

db.collection.aggregate([

{$stage1: {

{ expression1 },

{ expression2 }... },

$stage2: {

{ expression1 }... }

}

])

Field References: Field names are referred to with $ to signify a field path. E.g.  
$set: { userName : { $concat: [“$firstName”, “$secondName”] } }

#### Match

$match is used to filter documents that satisfy a query. It works like a find() and should be placed early in a pipeline to reduce the number of documents that will be processed later in the pipeline.  
$match: {query}

#### Group

$group to group results s per a group key. Output is one document for each unique key.  
$group: {\_id:<key>, <$field>:{<accumulator>:<expresssion>}}

E.g. of accumulators are $count, $avg, $sum, $min, $max, $median

#### Sort

Output of previous stage is sorted. 1 for ascending and -1 for descending  
$sort: {<field>:1}

After sorting we can limit the results using $limit:5 etc.

#### Project

Determines what is returned among existing fields and new fields that can be created. Usually the last stage. 1 to include or 0 to exclude. We can specify value for new fields created in this stage. \_id is always returned by default, to avoid this we mention \_id:0.

While projecting we can on the fly create new fields by using operators such as $divide. $divide expects an array [] of 2 fields/numbers.

#### Set

Adds or modifies fields in the pipeline and then outputs the documents with the new fields.  
$set:{”fieldname”:”Value”}

The $count stage creates a new document, with the number of documents at that stage in the aggregation pipeline assigned to the specified field name.

#### Out

For writing the output of a pipeline into a collection. It overwrites if collection already exists with same name.

#### Unwind

To work with array fields

#### Bucket

To categorise documents

### Indexes

Special data structures that store small portions of data that is ordered and can be efficiently searched. They point to \_id of the documents. They improve query performance (we have to use index prefix). Default index is created based on \_id. Adding indexes increase write performance cost.

Use getIndexes() to see all the indexes created in a collection.

db.collection.getIndexes()

#### Single Field

A single field index is an index on a single field of a document. MongoDB creates a single field index on the \_id field by default, but additional indexes may be needed for other fields as well. A single field index can also be a multikey index if it operates on an array field.

Db.collection.createIndex({field:1}, {unique:true}) returns the index name. Here we are asking for ascending order of index based on field and enforcing that this field should be unique.

#### Compound

MongoDB supports compound indexes, where a single index structure holds references to multiple fields within a collection's documents. A compound index is created by specifying the fields that the index should reference, followed by the order in which the fields should be sorted. The order of the fields in the index is important because it determines the order in which the documents are returned when querying the collection. A compound index can also be a multikey index if one of the fields is an array.

#### Multikey indexes

A multikey index is an index on an array field. Each element in the array gets an index key, which supports efficient querying against array fields.

Check if an index is being used on a query

Use explain() in a collection when running a query to see the Execution plan. This plan provides the details of the execution stages (IXSCAN , COLLSCAN, FETCH, SORT, etc.). There is a winning plan that gets executed, other plans are rejected.

1. The IXSCAN stage indicates the query is using an index and what index is being selected.
2. The COLLSCAN stage indicates a collection scan is perform, not using any indexes.
3. The FETCH stage indicates documents are being read from the collection.
4. The SORT stage indicates documents are being sorted in memory.

### Data Modeling

How data is stored and the relationships between different entities in the data. Schema is the organization of data inside a DB. These can be decided on the basis of

1. what the application does: Structure data such that it matches the way the application queries and updates data.
2. what data is to be stored: Data relationships between sets are 1:1, 1:many (modeled as Array) and many:many.
3. how will users access data (access patterns): Data that’s accessed together is to be stored together.
4. what data is most valuable.

Data Relationships are modeled using

1. Embedding: Related data are inserted into a document. Avoids joins and one single query to update and access data into/from a record. This creates larger documents (unbounded documents maybe created as array is a valid data type) which can slow performance as document is fully loaded to memory while accessing. BSON document threshold is 16 MB. Embedding many:many relationships result in data duplication.
2. Referencing (linking or data normalization): Reference to a document in another collection. References store the \_id field of one document in another document creating a unidirectional link between the two. This reduces data duplication and makes documents smaller but reduces read performance and uses more resources as we need to join data from multiple collections.

Common data modeling errors (Schema anti-patterns)

1. Massive arrays
2. Large number of collections
3. Bloated Documents
4. Unnecessary indexes
5. Queries without indexes
6. Data accessed together but kept in different collections

**ObjectId('6286809e2f3fa87b7d86dccd')**

## Resources

### Lesson 01: Introduction to Data Modeling

* [Data Modeling Introduction](https://www.mongodb.com/docs/manual/core/data-modeling-introduction/?_ga=2.60935781.810066485.1665291537-836515500.1666025886)
* [Separating Data That is Accessed Together](https://www.mongodb.com/developer/products/mongodb/schema-design-anti-pattern-separating-data/?_ga=2.60935781.810066485.1665291537-836515500.1666025886)

### Lesson 02: Types of Data Relationships

* [Data Model Design](https://www.mongodb.com/docs/manual/data-modeling/schema-design-process/)
* [Model Relationships Between Documents](https://www.mongodb.com/docs/v4.2/applications/data-models-relationships/?_ga=2.19332209.810066485.1665291537-836515500.1666025886)
* [Embedding MongoDB](https://www.mongodb.com/basics/embedded-mongodb?_ga=2.19332209.810066485.1665291537-836515500.1666025886)
* [MongoDB Schema Design Best Practices](https://www.mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/?_ga=2.19332209.810066485.1665291537-836515500.1666025886)

### Lesson 03: Modeling Data Relationships

* [Data Model Design](https://www.mongodb.com/docs/manual/data-modeling/schema-design-process/)
* [Model Relationships Between Documents](https://www.mongodb.com/docs/v4.2/applications/data-models-relationships/?_ga=2.19332209.810066485.1665291537-836515500.1666025886)

### Lesson 04: Embedding Data in Documents

* [Embedding MongoDB](https://www.mongodb.com/basics/embedded-mongodb?_ga=2.19332209.810066485.1665291537-836515500.1666025886)
* [Model One-to-One Relationships with Embedded Documents](https://www.mongodb.com/docs/manual/tutorial/model-embedded-one-to-one-relationships-between-documents/?_ga=2.19332209.810066485.1665291537-836515500.1666025886)
* [Model One-to-Many Relationships with Embedded Documents](https://www.mongodb.com/docs/manual/tutorial/model-embedded-one-to-many-relationships-between-documents/?_ga=2.19332209.810066485.1665291537-836515500.1666025886)

### Lesson 05: Referencing Data in Documents

* [Normalized Data Models](https://www.mongodb.com/docs/manual/data-modeling/#references)
* [Model One-to-Many Relationships with Document References](https://www.mongodb.com/docs/manual/tutorial/model-referenced-one-to-many-relationships-between-documents/?_ga=2.64006886.810066485.1665291537-836515500.1666025886)

### Lesson 06: Scaling a Data Model

* [Operational Factors and Data Models](https://www.mongodb.com/docs/manual/core/data-model-operations/?_ga=2.64006886.810066485.1665291537-836515500.1666025886)
* [Performance Best Practices: MongoDB Data Modeling and Memory Sizing](https://www.mongodb.com/blog/post/performance-best-practices-mongodb-data-modeling-and-memory-sizing?_ga=2.64006886.810066485.1665291537-836515500.1666025886)

### Lesson 07: Using Atlas Tools for Schema Help

* [A Summary of Schema Design Anti-Patterns and How to Spot Them](https://www.mongodb.com/developer/products/mongodb/schema-design-anti-pattern-summary/?_ga=2.64006886.810066485.1665291537-836515500.1666025886)

### Lesson 01: Using MongoDB Python Client Libraries

* [MongoDB Drivers](https://www.mongodb.com/docs/drivers/)
* [PyMongo Documentation](https://www.mongodb.com/docs/drivers/pymongo/)

### Lesson 02: Connecting to an Atlas Cluster in Python Applications

* [Using PyMongo with MongoDB Atlas](https://pymongo.readthedocs.io/en/stable/atlas.html)
* [Get Connection String](https://www.mongodb.com/docs/guides/atlas/connection-string/)
* [Connection String URI Format](https://www.mongodb.com/docs/manual/reference/connection-string/)
* [Set Up Your Environment](https://www.mongodb.com/developer/languages/python/python-quickstart-crud/?_ga=2.148314039.1652794197.1722098604-695756192.1721270238&_gac=1.183249364.1721754794.CjwKCAjwqf20BhBwEiwAt7dtdRru60NBhLTaHYBd2Jbbg__2WKxt7TEqM6-tpyWrrg-p3L_KsQ_kjRoCcqAQAvD_BwE#set-up-your-environment)

### Lesson 03: Troubleshooting a MongoDB Connection in Python Applications

* [Troubleshoot Connection Issues](https://www.mongodb.com/docs/atlas/troubleshoot-connection/)

#### Lesson 1 - Inserting Documents

* [MongoDB Docs: insertOne()](https://docs.mongodb.com/manual/reference/method/db.collection.insertOne/)
* [MongoDB Docs: insertMany()](https://docs.mongodb.com/manual/reference/method/db.collection.insertMany/)

#### Lesson 2 - Finding Documents

* [MongoDB Docs: find()](https://docs.mongodb.com/manual/reference/method/db.collection.find/)
* [MongoDB Docs: $in](https://docs.mongodb.com/manual/reference/operator/query/in/)

#### Lesson 3 - Finding Documents Using Comparison Operators

* [MongoDB Docs: Comparison Operators](https://docs.mongodb.com/manual/reference/operator/query-comparison/)

#### Lesson 4 - Querying on Array Elements

* [MongoDB Docs: $elemMatch](https://docs.mongodb.com/manual/reference/operator/query/elemMatch/)
* [MongoDB Docs: Querying Arrays](https://docs.mongodb.com/manual/tutorial/query-array-of-documents/#combination-of-elements-satisfies-the-criteria)

#### Lesson 5 - Finding Documents Using Logical Operators

* [MongoDB Docs: Logical Operators](https://docs.mongodb.com/manual/reference/operator/query-logical/)

### Lesson 01: Replacing a Document in MongoDB

* [MongoDB Docs: replaceOne()](https://www.mongodb.com/docs/manual/reference/method/db.collection.replaceOne/?_ga=2.56665699.810066485.1665291537-836515500.1666025886)

### Lesson 02: Updating MongoDB Documents by Using updateOne()

* [MongoDB Docs: Update Operators](https://www.mongodb.com/docs/manual/reference/operator/update/?_ga=2.56665699.810066485.1665291537-836515500.1666025886)
* [MongoDB Docs: $set](https://docs.mongodb.com/manual/reference/operator/update/set/?_ga=2.56665699.810066485.1665291537-836515500.1666025886)
* [MongoDB Docs: $push](https://docs.mongodb.com/manual/reference/operator/update/push/?_ga=2.34644840.810066485.1665291537-836515500.1666025886)
* [MongoDB Docs: upsert](https://www.mongodb.com/docs/drivers/node/current/fundamentals/crud/write-operations/upsert/?_ga=2.123127490.810066485.1665291537-836515500.1666025886)

### Lesson 03: Updating MongoDB Documents by Using findAndModify()

* [MongoDB Docs: findAndModify()](https://www.mongodb.com/docs/manual/reference/method/db.collection.findAndModify/?_ga=2.123127490.810066485.1665291537-836515500.1666025886)

### Lesson 04: Updating MongoDB Documents by Using findAndModify()

* [MongoDB Docs: updateMany()](https://www.mongodb.com/docs/manual/reference/method/db.collection.updateMany/?_ga=2.123127490.810066485.1665291537-836515500.1666025886)

### Lesson 05: Deleting Documents in MongoDB

* [MongoDB Docs: deleteOne()](https://www.mongodb.com/docs/v5.3/reference/method/db.collection.deleteOne/)
* [MongoDB Docs: deleteMany()](https://www.mongodb.com/docs/v5.3/reference/method/db.collection.deleteMany/?_ga=2.23103219.810066485.1665291537-836515500.1666025886)

Use the following resources to learn more about modifying query results in MongoDB:

### Lesson 01: Sorting and Limiting Query Results in MongoDB

* [MongoDB Docs: cursor.sort()](https://www.mongodb.com/docs/manual/reference/method/cursor.sort/?_ga=2.22528882.810066485.1665291537-836515500.1666025886)
* [MongoDB Docs: cursor.limit()](https://www.mongodb.com/docs/manual/reference/method/cursor.limit/?_ga=2.22528882.810066485.1665291537-836515500.1666025886)

### Lesson 02: Returning Specific Data from a Query in MongoDB

* [MongoDB Docs: Project Fields to Return from Query](https://www.mongodb.com/docs/manual/tutorial/project-fields-from-query-results/?_ga=2.22528882.810066485.1665291537-836515500.1666025886)
* [MongoDB Docs: Projection Restrictions](https://www.mongodb.com/docs/manual/reference/limits/?_ga=2.22528882.810066485.1665291537-836515500.1666025886#mongodb-limit-Projection-Restrictions)

### Lesson 03: Counting Documents in a MongoDB Collection

* [MongoDB Docs: db.collection.countDocuments()](https://www.mongodb.com/docs/manual/reference/method/db.collection.countDocuments/?_ga=2.30900342.810066485.1665291537-836515500.1666025886)

### Lesson 01: Working with MongoDB Documents in Python

* [BSON Encoding and Decoding](https://pymongo.readthedocs.io/en/stable/api/bson/index.html)

### Lesson 02: Inserting a Document in Python Applications

* [Inserting a Document](https://pymongo.readthedocs.io/en/stable/tutorial.html#inserting-a-document)
* [Bulk Inserts](https://pymongo.readthedocs.io/en/stable/tutorial.html#bulk-inserts)

### Lesson 03: Querying a MongoDB Collection in Python Applications

* [Getting a Single Document With find\_one()](https://pymongo.readthedocs.io/en/stable/tutorial.html#getting-a-single-document-with-find-one)
* [Querying for More Than One Document](https://pymongo.readthedocs.io/en/stable/tutorial.html#querying-for-more-than-one-document)
* [Querying by ObjectId](https://pymongo.readthedocs.io/en/stable/tutorial.html#querying-by-objectid)

### Lesson 04: Updating Documents in Python Applications

* [update\_one()](https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html?highlight=update#pymongo.collection.Collection.update_one)
* [update\_many()](https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html?highlight=update#pymongo.collection.Collection.update_many)

### Lesson 05: Deleting Documents in Python Applications

* [delete\_one()](https://pymongo.readthedocs.io/en/stable/api/pymongo/operations.html#pymongo.operations.DeleteOne)
* [delete\_many()](https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html?highlight=delete_many#pymongo.collection.Collection.delete_many)

### Lesson 06: Creating MongoDB Transactions in Python Applications

* [Transactions](https://www.mongodb.com/docs/manual/core/transactions/)
* [with\_transaction()](https://pymongo.readthedocs.io/en/stable/api/pymongo/client_session.html?highlight=transactions#pymongo.client_session.ClientSession.with_transaction)

**Lesson 01: Introduction to MongoDB Aggregation**

* [MongoDB Docs: Aggregation Operations](https://www.mongodb.com/docs/manual/aggregation/)
* [MongoDB Docs: Aggregation Pipelines](https://www.mongodb.com/docs/manual/aggregation/#std-label-aggregation-pipeline-intro)

**Lesson 02: Using**$match**and**$group**Stages in a MongoDB Aggregation Pipeline**

* [MongoDB Docs: $match](https://www.mongodb.com/docs/manual/reference/operator/aggregation/match/)
* [MongoDB Docs: $group](https://www.mongodb.com/docs/manual/reference/operator/aggregation/group/)

**Lesson 03: Using**$sort**and**$limit**Stages in a MongoDB Aggregation Pipeline**

* [MongoDB Docs: $sort](https://www.mongodb.com/docs/manual/reference/operator/aggregation/sort/)
* [MongoDB Docs: $limit](https://www.mongodb.com/docs/manual/reference/operator/aggregation/limit)

**Lesson 04: Using**$project**,**$count**, and**$set**Stages in a MongoDB Aggregation Pipeline**

* [MongoDB Docs: $project](https://www.mongodb.com/docs/manual/reference/operator/aggregation/project/)
* [MongoDB Docs: $count](https://www.mongodb.com/docs/manual/reference/operator/aggregation/count/)
* [MongoDB Docs: $set](https://www.mongodb.com/docs/manual/reference/operator/aggregation/set/)

**Lesson 05: Using**$out**Stage in a MongoDB Aggregation Pipeline**

* [MongoDB Docs: $out](https://www.mongodb.com/docs/manual/reference/operator/aggregation/out/)

### Lesson 01: Building a MongoDB Aggregation Pipeline in Python Applications

* [Aggregation Operations](https://www.mongodb.com/docs/manual/aggregation/?_ga=2.59215460.810066485.1665291537-836515500.1666025886)
* [Getting Started with Aggregation Pipelines in Python](https://www.mongodb.com/developer/languages/python/python-quickstart-aggregation/?_ga=2.59215460.810066485.1665291537-836515500.1666025886)
* [Aggregation Examples](https://pymongo.readthedocs.io/en/stable/examples/aggregation.html)
* [Practical MongoDB Aggregations Book](https://www.practical-mongodb-aggregations.com/)

### Lesson 02: Using MongoDB Aggregation Stages with Python: $match and $group

* [$match](https://www.mongodb.com/docs/manual/reference/operator/aggregation/match/?_ga=2.26136564.810066485.1665291537-836515500.1666025886)
* [$group](https://www.mongodb.com/docs/manual/reference/operator/aggregation/group/?_ga=2.26136564.810066485.1665291537-836515500.1666025886)

### Lesson 03: Using MongoDB Aggregation Stages with Python: $sort and $project

* [$sort](https://www.mongodb.com/docs/manual/reference/operator/aggregation/sort/?_ga=2.26136564.810066485.1665291537-836515500.1666025886)
* [$project](https://www.mongodb.com/docs/manual/reference/operator/aggregation/project/?_ga=2.26136564.810066485.1665291537-836515500.1666025886)