MDocument

Release 2020

Yury Sokov

Feb 06, 2021

CONTENTS

CHAPTER

ONE

MDOCUMENT

Modules

mdocument.client
mdocument.document
mdocument.document_array
mdocument.document_dict
mdocument.exceptions
mdocument.model

1.1 mdocument.client

Classes

MDocumentAsyncIOClient(*args, **kwargs)
MDocumentAsyncIOMotorCollection(database, name)
MDocumentAsyncIOMotorDatabase(client, name, . . .)

1.1.1 mdocument.client.MDocumentAsyncIOClient

class mdocument.client.**MDocumentAsyncIOClient**(**args*, ***kwargs*)

Bases: motor.motor_asyncio.AsyncIOMotorClient

__init__(**args*, ***kwargs*)

Create a new connection to a single MongoDB instance at *host:port*.

Takes the same constructor arguments as MongoClient, as well as:

Parameters

- *io_loop* (optional): Special event loop instance to use instead of default

Methods

__init__(*args, **kwargs)	Create a new connection to a single MongoDB instance at *host:port*.
drop_database(name_or_database[, session])	Drop a database.
fsync(**kwargs)	**DEPRECATED**: Flush all pending writes to datafiles.
get_database([name, codec_options, ...])	Get a `MotorDatabase` with the given name and options.
get_default_database([default, ...])	Get the database named in the MongoDB connection URI.
get_io_loop()	
list_database_names([session])	Get a list of the names of all databases on the connected server.
list_databases([session])	Get a cursor over the databases of the connected server.
server_info([session])	Get information about the MongoDB server we're connected to.
start_session([causal_consistency, ...])	Start a logical session.
unlock([session])	**DEPRECATED**: Unlock a previously locked server.
watch([pipeline, full_document, ...])	Watch changes on this cluster.
wrap(obj)	

Attributes

HOST	str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str
PORT	int([x]) -> integer int(x, base=10) -> integer
address	(host, port) of the current standalone, primary, or mongos, or None.
arbiters	Arbiters in the replica set.
close	Cleanup client resources and disconnect from MongoDB.
codec_options	Read only access to the `CodecOptions` of this instance.
event_listeners	The event listeners registered for this client.
is_mongos	If this client is connected to mongos.
is_primary	If this client is connected to a server that can accept writes.
local_threshold_ms	The local threshold for this instance.
max_bson_size	The largest BSON object the connected server accepts in bytes.
max_idle_time_ms	The maximum number of milliseconds that a connection can remain idle in the pool before being removed and replaced.
max_message_size	The largest message the connected server accepts in bytes.
max_pool_size	The maximum allowable number of concurrent connections to each connected server.

continues on next page

Table 4 – continued from previous page

max_write_batch_size	The maxWriteBatchSize reported by the server.
min_pool_size	The minimum required number of concurrent connections that the pool will maintain to each connected server.
nodes	Set of all currently connected servers.
primary	The (host, port) of the current primary of the replica set.
read_concern	Read only access to the `ReadConcern` of this instance.
read_preference	Read only access to the read preference of this instance.
retry_reads	If this instance should retry supported write operations.
retry_writes	If this instance should retry supported write operations.
secondaries	The secondary members known to this client.
server_selection_timeout	The server selection timeout for this instance in seconds.
write_concern	Read only access to the `WriteConcern` of this instance.

property HOST

str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

property PORT

int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

property address

(host, port) of the current standalone, primary, or mongos, or None.

Accessing *address* raises `InvalidOperation` if the client is load-balancing among mongoses, since there is no single address. Use *nodes* instead.

If the client is not connected, this will block until a connection is established or raise ServerSelectionTimeoutError if no server is available.

New in version 3.0.

property arbiters

Arbiters in the replica set.

A sequence of (host, port) pairs. Empty if this client is not connected to a replica set, there are no arbiters, or this client was created without the *replicaSet* option.

property close
Cleanup client resources and disconnect from MongoDB.

On MongoDB >= 3.6, end all server sessions created by this client by sending one or more endSessions commands.

Close all sockets in the connection pools and stop the monitor threads. If this instance is used again it will be automatically re-opened and the threads restarted unless auto encryption is enabled. A client enabled with auto encryption cannot be used again after being closed; any attempt will raise `InvalidOperation`.

Changed in version 3.6: End all server sessions created by this client.

property codec_options
Read only access to the `CodecOptions` of this instance.

drop_database(*name_or_database*, *session=None*)
Drop a database.

Raises `TypeError` if *name_or_database* is not an instance of `basestring` (`str` in python 3) or `Database`.

Parameters

- *name_or_database*: the name of a database to drop, or a `Database` instance representing the database to drop
- *session* (optional): a `ClientSession`.

Changed in version 3.6: Added `session` parameter.

Note: The `write_concern` of this client is automatically applied to this operation when using MongoDB >= 3.4.

Changed in version 3.4: Apply this client's write concern automatically to this operation when connected to MongoDB >= 3.4.

property event_listeners
The event listeners registered for this client.

See `monitoring` for details.

fsync(***kwargs*)
DEPRECATED: Flush all pending writes to datafiles.

Optional parameters can be passed as keyword arguments:

- *lock*: If True lock the server to disallow writes.
- *async*: If True don't block while synchronizing.
- *session* (optional): a `ClientSession`, created with `start_session()`.

Note: Starting with Python 3.7 *async* is a reserved keyword. The async option to the fsync command can be passed using a dictionary instead:

```
options = {'async': True}
await client.fsync(**options)
```

Deprecated. Run the fsync command directly with `command()` instead. For example:

```
await client.admin.command('fsync', lock=True)
```

Changed in version 2.2: Deprecated.

Changed in version 1.2: Added `session` parameter.

Warning: *async* and *lock* can not be used together.

Warning: MongoDB does not support the *async* option on Windows and will raise an exception on that platform.

get_database(*name=None*, *codec_options=None*, *read_preference=None*, *write_concern=None*, *read_concern=None*)

Get a `MotorDatabase` with the given name and options.

Useful for creating a `MotorDatabase` with different codec options, read preference, and/or write concern from this `MotorClient`.

```
>>> from pymongo import ReadPreference
>>> client.read_preference == ReadPreference.PRIMARY
True
>>> db1 = client.test
>>> db1.read_preference == ReadPreference.PRIMARY
True
>>> db2 = client.get_database(
...     'test', read_preference=ReadPreference.SECONDARY)
>>> db2.read_preference == ReadPreference.SECONDARY
True
```

Parameters

- *name*: The name of the database - a string.
- *codec_options* (optional): An instance of `CodecOptions`. If `None` (the default) the *codec_options* of this `MotorClient` is used.
- *read_preference* (optional): The read preference to use. If `None` (the default) the *read_preference* of this `MotorClient` is used. See `read_preferences` for options.
- *write_concern* (optional): An instance of `WriteConcern`. If `None` (the default) the *write_concern* of this `MotorClient` is used.

get_default_database(*default=None*, *codec_options=None*, *read_preference=None*, *write_concern=None*, *read_concern=None*)

Get the database named in the MongoDB connection URI.

```
>>> uri = 'mongodb://host/my_database'
>>> client = MotorClient(uri)
>>> db = client.get_default_database()
>>> assert db.name == 'my_database'
>>> db = client.get_default_database('fallback_db_name')
>>> assert db.name == 'my_database'
>>> uri_without_database = 'mongodb://host/'
```

(continues on next page)

(continued from previous page)

```
>>> client = MotorClient(uri_without_database)
>>> db = client.get_default_database('fallback_db_name')
>>> assert db.name == 'fallback_db_name'
```

Useful in scripts where you want to choose which database to use based only on the URI in a configuration file.

Parameters

- *default* (optional): the database name to use if no database name was provided in the URI.
- *codec_options* (optional): An instance of `CodecOptions`. If `None` (the default) the *codec_options* of this `MotorClient` is used.
- *read_preference* (optional): The read preference to use. If `None` (the default) the *read_preference* of this `MotorClient` is used. See `read_preferences` for options.
- *write_concern* (optional): An instance of `WriteConcern`. If `None` (the default) the *write_concern* of this `MotorClient` is used.
- *read_concern* (optional): An instance of `ReadConcern`. If `None` (the default) the *read_concern* of this `MotorClient` is used.

New in version 2.1: Revived this method. Added the `default`, `codec_options`, `read_preference`, `write_concern` and `read_concern` parameters.

Changed in version 2.0: Removed this method.

property is_mongos

If this client is connected to mongos. If the client is not connected, this will block until a connection is established or raise ServerSelectionTimeoutError if no server is available..

property is_primary

If this client is connected to a server that can accept writes.

True if the current server is a standalone, mongos, or the primary of a replica set. If the client is not connected, this will block until a connection is established or raise ServerSelectionTimeoutError if no server is available.

list_database_names(*session=None*)

Get a list of the names of all databases on the connected server.

Parameters

- *session* (optional): a `ClientSession`.

New in version 3.6.

async list_databases(*session=None*, ***kwargs*)

Get a cursor over the databases of the connected server.

Parameters

- *session* (optional): a `ClientSession`.
- ***kwargs* (optional): Optional parameters of the listDatabases command can be passed as keyword arguments to this method. The supported options differ by server version.

Returns An instance of `CommandCursor`.

New in version 3.6.

property local_threshold_ms
The local threshold for this instance.

property max_bson_size
The largest BSON object the connected server accepts in bytes.

If the client is not connected, this will block until a connection is established or raise ServerSelectionTimeoutError if no server is available.

property max_idle_time_ms
The maximum number of milliseconds that a connection can remain idle in the pool before being removed and replaced. Defaults to *None* (no limit).

property max_message_size
The largest message the connected server accepts in bytes.

If the client is not connected, this will block until a connection is established or raise ServerSelectionTimeoutError if no server is available.

property max_pool_size
The maximum allowable number of concurrent connections to each connected server. Requests to a server will block if there are *maxPoolSize* outstanding connections to the requested server. Defaults to 100. Cannot be 0.

When a server's pool has reached *max_pool_size*, operations for that server block waiting for a socket to be returned to the pool. If `waitQueueTimeoutMS` is set, a blocked operation will raise `ConnectionFailure` after a timeout. By default `waitQueueTimeoutMS` is not set.

property max_write_batch_size
The maxWriteBatchSize reported by the server.

If the client is not connected, this will block until a connection is established or raise ServerSelectionTimeoutError if no server is available.

Returns a default value when connected to server versions prior to MongoDB 2.6.

property min_pool_size
The minimum required number of concurrent connections that the pool will maintain to each connected server. Default is 0.

property nodes
Set of all currently connected servers.

> **Warning:** When connected to a replica set the value of *nodes* can change over time as `MongoClient`'s view of the replica set changes. *nodes* can also be an empty set when `MongoClient` is first instantiated and hasn't yet connected to any servers, or a network partition causes it to lose connection to all servers.

property primary
The (host, port) of the current primary of the replica set.

Returns `None` if this client is not connected to a replica set, there is no primary, or this client was created without the *replicaSet* option.

New in version 3.0: MongoClient gained this property in version 3.0 when MongoReplicaSetClient's functionality was merged in.

property read_concern
Read only access to the `ReadConcern` of this instance.

New in version 3.2.

property read_preference
Read only access to the read preference of this instance.

Changed in version 3.0: The *read_preference* attribute is now read only.

property retry_reads
If this instance should retry supported write operations.

property retry_writes
If this instance should retry supported write operations.

property secondaries
The secondary members known to this client.

A sequence of (host, port) pairs. Empty if this client is not connected to a replica set, there are no visible secondaries, or this client was created without the *replicaSet* option.

New in version 3.0: MongoClient gained this property in version 3.0 when MongoReplicaSetClient's functionality was merged in.

server_info(*session=None*)
Get information about the MongoDB server we're connected to.

Parameters

- *session* (optional): a `ClientSession`.

Changed in version 3.6: Added `session` parameter.

property server_selection_timeout
The server selection timeout for this instance in seconds.

async start_session(*causal_consistency=True*, *default_transaction_options=None*)
Start a logical session.

This method takes the same parameters as PyMongo's `SessionOptions`. See the `client_session` module for details.

This session is created uninitialized, use it in an `await` expression to initialize it, or an `async with` statement.

```
async def coro():
    collection = client.db.collection

    # End the session after using it.
    s = await client.start_session()
    await s.end_session()

    # Or, use an "async with" statement to end the session
    # automatically.
    async with await client.start_session() as s:
        doc = {'_id': ObjectId(), 'x': 1}
        await collection.insert_one(doc, session=s)

        secondary = collection.with_options(
            read_preference=ReadPreference.SECONDARY)

        # Sessions are causally consistent by default, so we can read
        # the doc we just inserted, even reading from a secondary.
        async for doc in secondary.find(session=s):
            print(doc)
```

(continues on next page)

(continued from previous page)

```
    # Run a multi-document transaction:
    async with await client.start_session() as s:
        # Note, start_transaction doesn't require "await".
        async with s.start_transaction():
            await collection.delete_one({'x': 1}, session=s)
            await collection.insert_one({'x': 2}, session=s)

        # Exiting the "with s.start_transaction()" block while throwing an
        # exception automatically aborts the transaction, exiting the block
        # normally automatically commits it.

        # You can run additional transactions in the same session, so long as
        # you run them one at a time.
        async with s.start_transaction():
            await collection.insert_one({'x': 3}, session=s)
            await collection.insert_many({'x': {'$gte': 2}},
                                         {'$inc': {'x': 1}},
                                         session=s)
```

Requires MongoDB 3.6. Do **not** use the same session for multiple operations concurrently. A `MotorClientSession` may only be used with the MotorClient that started it.

Returns An instance of `MotorClientSession`.

Changed in version 2.0: Returns a `MotorClientSession`. Before, this method returned a PyMongo `ClientSession`.

New in version 1.2.

unlock (*session=None*)

DEPRECATED: Unlock a previously locked server.

Parameters

- *session* (optional): a `MotorClientSession`.

Deprecated. Users of MongoDB version 3.2 or newer can run the fsyncUnlock command directly with `command()`:

```
await motor_client.admin.command('fsyncUnlock')
```

Users of MongoDB version 3.0 can query the "unlock" virtual collection:

```
await motor_client.admin["$cmd.sys.unlock"].find_one()
```

Changed in version 2.2: Deprecated.

watch (*pipeline=None*, *full_document=None*, *resume_after=None*, *max_await_time_ms=None*, *batch_size=None*, *collation=None*, *start_at_operation_time=None*, *session=None*, *start_after=None*)

Watch changes on this cluster.

Returns a `MotorChangeStream` cursor which iterates over changes on all databases in this cluster. Introduced in MongoDB 4.0.

See the documentation for `MotorCollection.watch()` for more details and examples.

Parameters

- *pipeline* (optional): A list of aggregation pipeline stages to append to an initial `$changeStream` stage. Not all pipeline stages are valid after a `$changeStream` stage, see the MongoDB documentation on change streams for the supported stages.
- *full_document* (optional): The fullDocument option to pass to the `$changeStream` stage. Allowed values: 'updateLookup'. When set to 'updateLookup', the change notification for partial updates will include both a delta describing the changes to the document, as well as a copy of the entire document that was changed from some time after the change occurred.
- *resume_after* (optional): A resume token. If provided, the change stream will start returning changes that occur directly after the operation specified in the resume token. A resume token is the _id value of a change document.
- *max_await_time_ms* (optional): The maximum time in milliseconds for the server to wait for changes before responding to a getMore operation.
- *batch_size* (optional): The maximum number of documents to return per batch.
- *collation* (optional): The `Collation` to use for the aggregation.
- *start_at_operation_time* (optional): If provided, the resulting change stream will only return changes that occurred at or after the specified `Timestamp`. Requires MongoDB >= 4.0.
- *session* (optional): a `ClientSession`.
- *start_after* (optional): The same as *resume_after* except that *start_after* can resume notifications after an invalidate event. This option and *resume_after* are mutually exclusive.

Returns A `MotorChangeStream`.

Changed in version 2.1: Added the `start_after` parameter.

New in version 2.0.

property write_concern

Read only access to the `WriteConcern` of this instance.

Changed in version 3.0: The *write_concern* attribute is now read only.

1.1.2 mdocument.client.MDocumentAsyncIOMotorCollection

class mdocument.client.**MDocumentAsyncIOMotorCollection**(*database*, *name*, *codec_options=None*, *read_preference=None*, *write_concern=None*, *read_concern=None*, *_delegate=None*)

Bases: `motor.motor_asyncio.AsyncIOMotorCollection`

__init__(*database*, *name*, *codec_options=None*, *read_preference=None*, *write_concern=None*, *read_concern=None*, *_delegate=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(database, name[, codec_options, ...])	Initialize self.
aggregate(pipeline, **kwargs)	Execute an aggregation pipeline on this collection.
aggregate_raw_batches(pipeline, **kwargs)	Perform an aggregation and retrieve batches of raw BSON.
bulk_write(requests[, ordered, ...])	Send a batch of write operations to the server.
count_documents(filter[, session])	Count the number of documents in this collection.
create_index(keys[, session])	Creates an index on this collection.
create_indexes(indexes[, session])	Create one or more indexes on this collection.
delete_many(documents, *args, **kwargs)	Deletes multiple documents in database.
delete_one(document, *args, **kwargs)	Deletes one document in database.
distinct(key[, filter, session])	Get a list of distinct values for *key* among all documents in this collection.
drop([session])	Alias for `drop_collection`.
drop_index(index_or_name[, session])	Drops the specified index on this collection.
drop_indexes([session])	Drops all indexes on this collection.
estimated_document_count(**kwargs)	Get an estimate of the number of documents in this collection using collection metadata.
find(document_query, *args, **kwargs)	Finds multiple documents and returns them with provided type.
find_one(document_query, *args, **kwargs)	Finds one document and returns it with provided type.
find_one_and_delete(filter[, projection, ...])	Finds a single document and deletes it, returning the document.
find_one_and_replace(filter, replacement[, ...])	Finds a single document and replaces it, returning either the original or the replaced document.
find_one_and_update(filter, update[, ...])	Finds a single document and updates it, returning either the original or the updated document. By default *find_one_and_update()* returns the original version of the document before the update was applied::.
find_raw_batches(*args, **kwargs)	Query the database and retrieve batches of raw BSON.
`get_io_loop`()	
index_information([session])	Get information on this collection's indexes.
inline_map_reduce(map, reduce[, ...])	Perform an inline map/reduce operation on this collection.
insert_many(documents, *args, **kwargs)	Inserts multiple documents to database.
insert_one(document, *args, **kwargs)	Inserts one document to database.
list_indexes([session])	Get a cursor over the index documents for this collection. ::.
map_reduce(map, reduce, out[, ...])	Perform a map/reduce operation on this collection.
options([session])	Get the options set on this collection.
reindex([session])	**DEPRECATED**: Rebuild all indexes on this collection.
rename(new_name[, session])	Rename this collection.
replace_one(filter, replacement[, upsert, ...])	Replace a single document matching the filter.
update_many(documents, *args, **kwargs)	Updates multiple documents in database.
update_one(document, *args, **kwargs)	Updates one document in database.
watch([pipeline, full_document, ...])	Watch changes on this collection.

continues on next page

Table 5 – continued from previous page

with_options([codec_options, ...])	Get a clone of this collection changing the specified settings.
wrap(obj)	

Attributes

codec_options	Read only access to the CodecOptions of this instance.
full_name	The full name of this Collection.
name	The name of this Collection.
read_concern	Read only access to the ReadConcern of this instance.
read_preference	Read only access to the read preference of this instance.
write_concern	Read only access to the WriteConcern of this instance.

aggregate(*pipeline*, ***kwargs*)

Execute an aggregation pipeline on this collection.

The aggregation can be run on a secondary if the client is connected to a replica set and its read_preference is not PRIMARY.

Parameters

- *pipeline*: a single command or list of aggregation commands
- *session* (optional): a ClientSession, created with start_session().
- ***kwargs*: send arbitrary parameters to the aggregate command

Returns a MotorCommandCursor that can be iterated like a cursor from *find()*:

```
async def f():
    pipeline = [{'$project': {'name': {'$toUpper': '$name'}}}]
    async for doc in collection.aggregate(pipeline):
        print(doc)
```

MotorCommandCursor does not allow the explain option. To explain MongoDB's query plan for the aggregation, use MotorDatabase.command():

```
async def f():
    plan = await db.command(
        'aggregate', 'COLLECTION-NAME',
        pipeline=[{'$project': {'x': 1}}],
        explain=True)

    print(plan)
```

Changed in version 2.1: This collection's read concern is now applied to pipelines containing the *$out* stage when connected to MongoDB >= 4.2.

Changed in version 1.0: *aggregate()* now **always** returns a cursor.

Changed in version 0.5: *aggregate()* now returns a cursor by default, and the cursor is returned immediately without an await. See aggregation changes in Motor 0.5.

Changed in version 0.2: Added cursor support.

aggregate_raw_batches(*pipeline*, ***kwargs*)
Perform an aggregation and retrieve batches of raw BSON.

Similar to the *aggregate()* method but returns each batch as bytes.

This example demonstrates how to work with raw batches, but in practice raw batches should be passed to an external library that can decode BSON into another data type, rather than used with PyMongo's bson module.

```
async def get_raw():
    cursor = db.test.aggregate_raw_batches()
    async for batch in cursor:
        print(bson.decode_all(batch))
```

Note that aggregate_raw_batches does not support sessions.

New in version 2.0.

bulk_write(*requests*, *ordered=True*, *bypass_document_validation=False*, *session=None*)
Send a batch of write operations to the server.

Requests are passed as a list of write operation instances imported from pymongo: InsertOne, UpdateOne, UpdateMany, ReplaceOne, DeleteOne, or DeleteMany).

For example, say we have these documents:

```
{'x': 1, '_id': ObjectId('54f62e60fba5226811f634ef')}
{'x': 1, '_id': ObjectId('54f62e60fba5226811f634f0')}
```

We can insert a document, delete one, and replace one like so:

```
# DeleteMany, UpdateOne, and UpdateMany are also available.
from pymongo import InsertOne, DeleteOne, ReplaceOne

async def modify_data():
    requests = [InsertOne({'y': 1}), DeleteOne({'x': 1}),
                ReplaceOne({'w': 1}, {'z': 1}, upsert=True)]
    result = await db.test.bulk_write(requests)

    print("inserted %d, deleted %d, modified %d" % (
        result.inserted_count, result.deleted_count, result.modified_count))

    print("upserted_ids: %s" % result.upserted_ids)

    print("collection:")
    async for doc in db.test.find():
        print(doc)
```

This will print something like:

```
inserted 1, deleted 1, modified 0
upserted_ids: {2: ObjectId('54f62ee28891e756a6e1abd5')}

collection:
{'x': 1, '_id': ObjectId('54f62e60fba5226811f634f0')}
{'y': 1, '_id': ObjectId('54f62ee2fba5226811f634f1')}
{'z': 1, '_id': ObjectId('54f62ee28891e756a6e1abd5')}
```

Parameters

- *requests*: A list of write operations (see examples above).
- *ordered* (optional): If `True` (the default) requests will be performed on the server serially, in the order provided. If an error occurs all remaining operations are aborted. If `False` requests will be performed on the server in arbitrary order, possibly in parallel, and all operations will be attempted.
- *bypass_document_validation*: (optional) If `True`, allows the write to opt-out of document level validation. Default is `False`.
- *session* (optional): a `ClientSession`, created with `start_session()`.

Returns An instance of `BulkWriteResult`.

See also:

writes-and-ids

Note: *bypass_document_validation* requires server version **>= 3.2**

Changed in version 1.2: Added session parameter.

property codec_options
Read only access to the `CodecOptions` of this instance.

count_documents (*filter*, *session=None*, ***kwargs*)
Count the number of documents in this collection.

Note: For a fast count of the total documents in a collection see *estimated_document_count()*.

The *count_documents()* method is supported in a transaction.

All optional parameters should be passed as keyword arguments to this method. Valid options include:

- *skip* (int): The number of matching documents to skip before returning results.
- *limit* (int): The maximum number of documents to count. Must be a positive integer. If not provided, no limit is imposed.
- *maxTimeMS* (int): The maximum amount of time to allow this operation to run, in milliseconds.
- *collation* (optional): An instance of `Collation`. This option is only supported on MongoDB 3.4 and above.
- *hint* (string or list of tuples): The index to use. Specify either the index name as a string or the index specification as a list of tuples (e.g. [('a', pymongo.ASCENDING), ('b', pymongo.ASCENDING)]). This option is only supported on MongoDB 3.6 and above.

The *count_documents()* method obeys the *read_preference* of this `Collection`.

Note: When migrating from `count()` to *count_documents()* the following query operators must be replaced:

Operator	Replacement
$where	$expr
$near	$geoWithin with $center
$nearSphere	$geoWithin with $centerSphere

$expr requires MongoDB 3.6+

Parameters

- *filter* (required): A query document that selects which documents to count in the collection. Can be an empty document to count all documents.
- *session* (optional): a `ClientSession`.
- ***kwargs* (optional): See list of options above.

New in version 3.7.

create_index(*keys*, *session=None*, ***kwargs*)

Creates an index on this collection.

Takes either a single key or a list of (key, direction) pairs. The key(s) must be an instance of `basestring` (`str` in python 3), and the direction(s) must be one of (`ASCENDING`, `DESCENDING`, `GEO2D`, `GEOHAYSTACK`, `GEOSPHERE`, `HASHED`, `TEXT`).

To create a single key ascending index on the key `'mike'` we just use a string argument:

```
>>> my_collection.create_index("mike")
```

For a compound index on `'mike'` descending and `'eliot'` ascending we need to use a list of tuples:

```
>>> my_collection.create_index([("mike", pymongo.DESCENDING),
...                             ("eliot", pymongo.ASCENDING)])
```

All optional index creation parameters should be passed as keyword arguments to this method. For example:

```
>>> my_collection.create_index([("mike", pymongo.DESCENDING)],
...                            background=True)
```

Valid options include, but are not limited to:

- *name*: custom name to use for this index - if none is given, a name will be generated.
- *unique*: if `True`, creates a uniqueness constraint on the index.
- *background*: if `True`, this index should be created in the background.
- *sparse*: if `True`, omit from the index any documents that lack the indexed field.
- *bucketSize*: for use with geoHaystack indexes. Number of documents to group together within a certain proximity to a given longitude and latitude.
- *min*: minimum value for keys in a `GEO2D` index.
- *max*: maximum value for keys in a `GEO2D` index.
- *expireAfterSeconds*: <int> Used to create an expiring (TTL) collection. MongoDB will automatically delete documents from this collection after <int> seconds. The indexed field must be a UTC datetime or the data will not expire.
- *partialFilterExpression*: A document that specifies a filter for a partial index. Requires MongoDB >=3.2.
- *collation* (optional): An instance of `Collation`. Requires MongoDB >= 3.4.

- *wildcardProjection*: Allows users to include or exclude specific field paths from a wildcard index using the {"$**" : 1} key pattern. Requires MongoDB >= 4.2.
- *hidden*: if `True`, this index will be hidden from the query planner and will not be evaluated as part of query plan selection. Requires MongoDB >= 4.4.

See the MongoDB documentation for a full list of supported options by server version.

> **Warning:** *dropDups* is not supported by MongoDB 3.0 or newer. The option is silently ignored by the server and unique index builds using the option will fail if a duplicate value is detected.

Note: The `write_concern` of this collection is automatically applied to this operation when using MongoDB >= 3.4.

Parameters

- *keys*: a single key or a list of (key, direction) pairs specifying the index to create
- *session* (optional): a `ClientSession`.
- ***kwargs* (optional): any additional index creation options (see the above list) should be passed as keyword arguments

Changed in version 3.11: Added the `hidden` option.

Changed in version 3.6: Added `session` parameter. Added support for passing maxTimeMS in kwargs.

Changed in version 3.4: Apply this collection's write concern automatically to this operation when connected to MongoDB >= 3.4. Support the *collation* option.

Changed in version 3.2: Added partialFilterExpression to support partial indexes.

Changed in version 3.0: Renamed *key_or_list* to *keys*. Removed the *cache_for* option. *create_index()* no longer caches index names. Removed support for the drop_dups and bucket_size aliases.

create_indexes(*indexes*, *session=None*, ***kwargs*)

Create one or more indexes on this collection:

```
from pymongo import IndexModel, ASCENDING, DESCENDING

async def create_two_indexes():
    index1 = IndexModel([("hello", DESCENDING),
                         ("world", ASCENDING)], name="hello_world")
    index2 = IndexModel([("goodbye", DESCENDING)])
    print(await db.test.create_indexes([index1, index2]))
```

This prints:

```
['hello_world', 'goodbye_-1']
```

Parameters

- *indexes*: A list of `IndexModel` instances.
- *session* (optional): a `ClientSession`, created with `start_session()`.

- ***kwargs* (optional): optional arguments to the createIndexes command (like maxTimeMS) can be passed as keyword arguments.

The `write_concern` of this collection is automatically applied to this operation when using MongoDB >= 3.4.

Changed in version 1.2: Added session parameter.

async delete_many(*documents: List[*mdocument.document.MDocument*]*, **args*, ***kwargs*)
Deletes multiple documents in database. Also updates related documents.

async delete_one(*document:* mdocument.document.MDocument, **args*, ***kwargs*)
Deletes one document in database. Also updates related documents.

distinct(*key*, *filter=None*, *session=None*, ***kwargs*)
Get a list of distinct values for *key* among all documents in this collection.

Raises `TypeError` if *key* is not an instance of `basestring` (`str` in python 3).

All optional distinct parameters should be passed as keyword arguments to this method. Valid options include:

- *maxTimeMS* (int): The maximum amount of time to allow the count command to run, in milliseconds.
- *collation* (optional): An instance of `Collation`. This option is only supported on MongoDB 3.4 and above.

The *distinct()* method obeys the *read_preference* of this `Collection`.

Parameters

- *key*: name of the field for which we want to get the distinct values
- *filter* (optional): A query document that specifies the documents from which to retrieve the distinct values.
- *session* (optional): a `ClientSession`.
- ***kwargs* (optional): See list of options above.

Changed in version 3.6: Added `session` parameter.

Changed in version 3.4: Support the *collation* option.

drop(*session=None*)
Alias for `drop_collection`.

The following two calls are equivalent:

```
await db.foo.drop()
await db.drop_collection("foo")
```

drop_index(*index_or_name*, *session=None*, ***kwargs*)
Drops the specified index on this collection.

Can be used on non-existant collections or collections with no indexes. Raises OperationFailure on an error (e.g. trying to drop an index that does not exist). *index_or_name* can be either an index name (as returned by *create_index*), or an index specifier (as passed to *create_index*). An index specifier should be a list of (key, direction) pairs. Raises TypeError if index is not an instance of (str, unicode, list).

> **Warning:** if a custom name was used on index creation (by passing the *name* parameter to *create_index()* or `ensure_index()`) the index **must** be dropped by name.

Parameters

- *index_or_name*: index (or name of index) to drop
- *session* (optional): a `ClientSession`.
- ***kwargs* (optional): optional arguments to the createIndexes command (like maxTimeMS) can be passed as keyword arguments.

Note: The `write_concern` of this collection is automatically applied to this operation when using MongoDB >= 3.4.

Changed in version 3.6: Added `session` parameter. Added support for arbitrary keyword arguments.

Changed in version 3.4: Apply this collection's write concern automatically to this operation when connected to MongoDB >= 3.4.

drop_indexes(*session=None*, ***kwargs*)
Drops all indexes on this collection.

Can be used on non-existant collections or collections with no indexes. Raises OperationFailure on an error.

Parameters

- *session* (optional): a `ClientSession`.
- ***kwargs* (optional): optional arguments to the createIndexes command (like maxTimeMS) can be passed as keyword arguments.

Note: The `write_concern` of this collection is automatically applied to this operation when using MongoDB >= 3.4.

Changed in version 3.6: Added `session` parameter. Added support for arbitrary keyword arguments.

Changed in version 3.4: Apply this collection's write concern automatically to this operation when connected to MongoDB >= 3.4.

estimated_document_count(***kwargs*)
Get an estimate of the number of documents in this collection using collection metadata.

The *estimated_document_count()* method is **not** supported in a transaction.

All optional parameters should be passed as keyword arguments to this method. Valid options include:

- *maxTimeMS* (int): The maximum amount of time to allow this operation to run, in milliseconds.

Parameters

- ***kwargs* (optional): See list of options above.

New in version 3.7.

async find(*document_query:* mdocument.document.MDocument, **args*, ***kwargs*)
Finds multiple documents and returns them with provided type.

async find_one(*document_query:* mdocument.document.MDocument, **args*, ***kwargs*)
Finds one document and returns it with provided type.

find_one_and_delete(*filter*, *projection=None*, *sort=None*, *hint=None*, *session=None*, ***kwargs*)

Finds a single document and deletes it, returning the document.

If we have a collection with 2 documents like `{'x': 1}`, then this code retrieves and deletes one of them:

```
async def delete_one_document():
    print(await db.test.count_documents({'x': 1}))
    doc = await db.test.find_one_and_delete({'x': 1})
    print(doc)
    print(await db.test.count_documents({'x': 1}))
```

This outputs something like:

```
2
{'x': 1, '_id': ObjectId('54f4e12bfba5220aa4d6dee8')}
1
```

If multiple documents match *filter*, a *sort* can be applied. Say we have 3 documents like:

```
{'x': 1, '_id': 0}
{'x': 1, '_id': 1}
{'x': 1, '_id': 2}
```

This code retrieves and deletes the document with the largest `_id`:

```
async def delete_with_largest_id():
    doc = await db.test.find_one_and_delete(
        {'x': 1}, sort=[('_id', pymongo.DESCENDING)])
```

This deletes one document and prints it:

```
{'x': 1, '_id': 2}
```

The *projection* option can be used to limit the fields returned:

```
async def delete_and_return_x():
    db.test.find_one_and_delete({'x': 1}, projection={'_id': False})
```

This prints:

```
{'x': 1}
```

Parameters

- *filter*: A query that matches the document to delete.
- *projection* (optional): a list of field names that should be returned in the result document or a mapping specifying the fields to include or exclude. If *projection* is a list "_id" will always be returned. Use a mapping to exclude fields from the result (e.g. projection={'_id': False}).
- *sort* (optional): a list of (key, direction) pairs specifying the sort order for the query. If multiple documents match the query, they are sorted and the first is deleted.
- *hint* (optional): An index used to support the query predicate specified either by its string name, or in the same format as passed to `create_index()` (e.g. `[('field', ASCENDING)]`). This option is only supported on MongoDB 4.4 and above.

- *session* (optional): a `ClientSession`, created with `start_session()`.
- ***kwargs* (optional): additional command arguments can be passed as keyword arguments (for example maxTimeMS can be used with recent server versions).

This command uses the `WriteConcern` of this `Collection` when connected to MongoDB >= 3.2. Note that using an elevated write concern with this command may be slower compared to using the default write concern.

Changed in version 2.2: Added `hint` parameter.

Changed in version 1.2: Added `session` parameter.

find_one_and_replace(*filter*, *replacement*, *projection=None*, *sort=None*, *upsert=False*, *return_document=False*, *hint=None*, *session=None*, ***kwargs*)

Finds a single document and replaces it, returning either the original or the replaced document.

The *find_one_and_replace()* method differs from *find_one_and_update()* by replacing the document matched by *filter*, rather than modifying the existing document.

Say we have 3 documents like:

```
{'x': 1, '_id': 0}
{'x': 1, '_id': 1}
{'x': 1, '_id': 2}
```

Replace one of them like so:

```
async def replace_one_doc():
    original_doc = await db.test.find_one_and_replace({'x': 1}, {'y': 1})
    print("original: %s" % original_doc)
    print("collection:")
    async for doc in db.test.find():
        print(doc)
```

This will print:

```
original: {'x': 1, '_id': 0}
collection:
{'y': 1, '_id': 0}
{'x': 1, '_id': 1}
{'x': 1, '_id': 2}
```

Parameters

- *filter*: A query that matches the document to replace.
- *replacement*: The replacement document.
- *projection* (optional): A list of field names that should be returned in the result document or a mapping specifying the fields to include or exclude. If *projection* is a list "_id" will always be returned. Use a mapping to exclude fields from the result (e.g. projection={'_id': False}).
- *sort* (optional): a list of (key, direction) pairs specifying the sort order for the query. If multiple documents match the query, they are sorted and the first is replaced.
- *upsert* (optional): When `True`, inserts a new document if no document matches the query. Defaults to `False`.

- *return_document*: If `ReturnDocument.BEFORE` (the default), returns the original document before it was replaced, or `None` if no document matches. If `ReturnDocument.AFTER`, returns the replaced or inserted document.
- *hint* (optional): An index to use to support the query predicate specified either by its string name, or in the same format as passed to `create_index()` (e.g. `[('field', ASCENDING)]`). This option is only supported on MongoDB 4.4 and above.
- *session* (optional): a `ClientSession`, created with `start_session()`.
- ***kwargs* (optional): additional command arguments can be passed as keyword arguments (for example maxTimeMS can be used with recent server versions).

This command uses the `WriteConcern` of this `Collection` when connected to MongoDB >= 3.2. Note that using an elevated write concern with this command may be slower compared to using the default write concern.

Changed in version 2.2: Added `hint` parameter.

Changed in version 1.2: Added `session` parameter.

find_one_and_update(*filter*, *update*, *projection=None*, *sort=None*, *upsert=False*, *return_document=False*, *array_filters=None*, *hint=None*, *session=None*, ***kwargs*)

Finds a single document and updates it, returning either the original or the updated document. By default *find_one_and_update()* returns the original version of the document before the update was applied:

```
async def set_done():
    print(await db.test.find_one_and_update(
        {'_id': 665}, {'$inc': {'count': 1}, '$set': {'done': True}}))
```

This outputs:

```
{'_id': 665, 'done': False, 'count': 25}}
```

To return the updated version of the document instead, use the *return_document* option.

```
from pymongo import ReturnDocument

async def increment_by_userid():
    print(await db.example.find_one_and_update(
        {'_id': 'userid'},
        {'$inc': {'seq': 1}},
        return_document=ReturnDocument.AFTER))
```

This prints:

```
{'_id': 'userid', 'seq': 1}
```

You can limit the fields returned with the *projection* option.

```
async def increment_by_userid():
    print(await db.example.find_one_and_update(
        {'_id': 'userid'},
        {'$inc': {'seq': 1}},
        projection={'seq': True, '_id': False},
        return_document=ReturnDocument.AFTER))
```

This results in:

```
{'seq': 2}
```

The *upsert* option can be used to create the document if it doesn't already exist.

```
async def increment_by_userid():
    print(await db.example.find_one_and_update(
        {'_id': 'userid'},
        {'$inc': {'seq': 1}},
        projection={'seq': True, '_id': False},
        upsert=True,
        return_document=ReturnDocument.AFTER))
```

The result:

```
{'seq': 1}
```

If multiple documents match *filter*, a *sort* can be applied. Say we have these two documents:

```
{'_id': 665, 'done': True, 'result': {'count': 26}}
{'_id': 701, 'done': True, 'result': {'count': 17}}
```

Then to update the one with the great `_id`:

```
async def set_done():
    print(await db.test.find_one_and_update(
        {'done': True},
        {'$set': {'final': True}},
        sort=[('_id', pymongo.DESCENDING)]))
```

This would print:

```
{'_id': 701, 'done': True, 'result': {'count': 17}}
```

Parameters

- *filter*: A query that matches the document to update.
- *update*: The update operations to apply.
- *projection* (optional): A list of field names that should be returned in the result document or a mapping specifying the fields to include or exclude. If *projection* is a list "_id" will always be returned. Use a dict to exclude fields from the result (e.g. projection={'_id': False}).
- *sort* (optional): a list of (key, direction) pairs specifying the sort order for the query. If multiple documents match the query, they are sorted and the first is updated.
- *upsert* (optional): When `True`, inserts a new document if no document matches the query. Defaults to `False`.
- *return_document*: If `ReturnDocument.BEFORE` (the default), returns the original document before it was updated, or `None` if no document matches. If `ReturnDocument.AFTER`, returns the updated or inserted document.
- *array_filters* (optional): A list of filters specifying which array elements an update should apply. Requires MongoDB 3.6+.
- *hint* (optional): An index to use to support the query predicate specified either by its string name, or in the same format as passed to `create_index()` (e.g. `[('field', ASCENDING)]`). This option is only supported on MongoDB 4.4 and above.

- *session* (optional): a `ClientSession`, created with `start_session()`.
- ***kwargs* (optional): additional command arguments can be passed as keyword arguments (for example maxTimeMS can be used with recent server versions).

This command uses the `WriteConcern` of this `Collection` when connected to MongoDB >= 3.2. Note that using an elevated write concern with this command may be slower compared to using the default write concern.

Changed in version 2.2: Added `hint` parameter.

Changed in version 1.2: Added `array_filters` and `session` parameters.

find_raw_batches(**args*, ***kwargs*)

Query the database and retrieve batches of raw BSON.

Similar to the *find()* method but returns each batch as bytes.

This example demonstrates how to work with raw batches, but in practice raw batches should be passed to an external library that can decode BSON into another data type, rather than used with PyMongo's `bson` module.

```
async def get_raw():
    cursor = db.test.find_raw_batches()
    async for batch in cursor:
        print(bson.decode_all(batch))
```

Note that `find_raw_batches` does not support sessions.

New in version 2.0.

property full_name

The full name of this `Collection`.

The full name is of the form *database_name.collection_name*.

index_information(*session=None*)

Get information on this collection's indexes.

Returns a dictionary where the keys are index names (as returned by create_index()) and the values are dictionaries containing information about each index. The dictionary is guaranteed to contain at least a single key, `"key"` which is a list of (key, direction) pairs specifying the index (as passed to create_index()). It will also contain any other metadata about the indexes, except for the `"ns"` and `"name"` keys, which are cleaned. For example:

```
async def create_x_index():
    print(await db.test.create_index("x", unique=True))
    print(await db.test.index_information())
```

This prints:

```
'x_1'
{'_id_': {'key': [('_id', 1)]},
 'x_1': {'unique': True, 'key': [('x', 1)]}}
```

Changed in version 1.2: Added session parameter.

inline_map_reduce(*map*, *reduce*, *full_response=False*, *session=None*, ***kwargs*)

Perform an inline map/reduce operation on this collection.

Perform the map/reduce operation on the server in RAM. A result collection is not created. The result set is returned as a list of documents.

If *full_response* is `False` (default) returns the result documents in a list. Otherwise, returns the full response from the server to the map reduce command.

The *inline_map_reduce()* method obeys the *read_preference* of this `Collection`.

Parameters

- *map*: map function (as a JavaScript string)
- *reduce*: reduce function (as a JavaScript string)
- *full_response* (optional): if `True`, return full response to this command - otherwise just return the result collection
- *session* (optional): a `ClientSession`.
- ***kwargs* (optional): additional arguments to the map reduce command may be passed as keyword arguments to this helper method, e.g.:

```
>>> db.test.inline_map_reduce(map, reduce, limit=2)
```

Changed in version 3.6: Added `session` parameter.

Changed in version 3.4: Added the *collation* option.

async insert_many(*documents: List[*mdocument.document.MDocument*]*, **args*, ***kwargs*)
Inserts multiple documents to database.

async insert_one(*document*, **args*, ***kwargs*)
Inserts one document to database.

list_indexes(*session=None*)
Get a cursor over the index documents for this collection.

```
async def print_indexes():
    async for index in db.test.list_indexes():
        print(index)
```

If the only index is the default index on `_id`, this might print:

```
SON([('v', 1), ('key', SON([('_id', 1)])), ('name', '_id_')])
```

async map_reduce(*map*, *reduce*, *out*, *full_response=False*, *session=None*, ***kwargs*)
Perform a map/reduce operation on this collection.

If *full_response* is `False` (default) returns a `MotorCollection` instance containing the results of the operation. Otherwise, returns the full response from the server to the map reduce command.

Parameters

- *map*: map function (as a JavaScript string)
- *reduce*: reduce function (as a JavaScript string)
- *out*: output collection name or *out object* (dict). See the map reduce command documentation for available options. Note: *out* options are order sensitive. `SON` can be used to specify multiple options. e.g. SON([('replace', <collection name>), ('db', <database name>)])
- *full_response* (optional): if `True`, return full response to this command - otherwise just return the result collection
- *session* (optional): a `ClientSession`, created with `start_session()`.

- ***kwargs* (optional): additional arguments to the map reduce command may be passed as keyword arguments to this helper method, e.g.:

```
result = await db.test.map_reduce(map, reduce, "myresults",␣
↪limit=2)
```

Returns a Future.

Note: The *map_reduce()* method does **not** obey the *read_preference* of this MotorCollection. To run mapReduce on a secondary use the *inline_map_reduce()* method instead.

Changed in version 1.2: Added session parameter.

property name
The name of this Collection.

options(*session=None*)
Get the options set on this collection.

Returns a dictionary of options and their values - see create_collection() for more information on the possible options. Returns an empty dictionary if the collection has not been created yet.

Parameters

- *session* (optional): a ClientSession.

Changed in version 3.6: Added session parameter.

property read_concern
Read only access to the ReadConcern of this instance.

New in version 3.2.

property read_preference
Read only access to the read preference of this instance.

Changed in version 3.0: The *read_preference* attribute is now read only.

reindex(*session=None*, ***kwargs*)
DEPRECATED: Rebuild all indexes on this collection.

Deprecated. Use command() to run the reIndex command directly instead:

```
await db.command({"reIndex": "<collection_name>"})
```

Note: Starting in MongoDB 4.6, the *reIndex* command can only be run when connected to a standalone mongod.

Parameters

- *session* (optional): a MotorClientSession.
- ***kwargs* (optional): optional arguments to the reIndex command (like maxTimeMS) can be passed as keyword arguments.

> **Warning:** reindex blocks all other operations (indexes are built in the foreground) and will be slow for large collections.

Changed in version 2.2: Deprecated.

rename (*new_name*, *session=None*, ***kwargs*)

Rename this collection.

If operating in auth mode, client must be authorized as an admin to perform this operation. Raises `TypeError` if *new_name* is not an instance of `basestring` (`str` in python 3). Raises `InvalidName` if *new_name* is not a valid collection name.

Parameters

- *new_name*: new name for this collection
- *session* (optional): a `ClientSession`.
- ***kwargs* (optional): additional arguments to the rename command may be passed as keyword arguments to this helper method (i.e. `dropTarget=True`)

Note: The `write_concern` of this collection is automatically applied to this operation when using MongoDB >= 3.4.

Changed in version 3.6: Added `session` parameter.

Changed in version 3.4: Apply this collection's write concern automatically to this operation when connected to MongoDB >= 3.4.

replace_one (*filter*, *replacement*, *upsert=False*, *bypass_document_validation=False*, *collation=None*, *hint=None*, *session=None*)

Replace a single document matching the filter.

Say our collection has one document:

```
{'x': 1, '_id': ObjectId('54f4c5befba5220aa4d6dee7')}
```

Then to replace it with another:

```
async def_replace_x_with_y():
    result = await db.test.replace_one({'x': 1}, {'y': 1})
    print('matched %d, modified %d' %
        (result.matched_count, result.modified_count))

    print('collection:')
    async for doc in db.test.find():
        print(doc)
```

This prints:

```
matched 1, modified 1
collection:
{'y': 1, '_id': ObjectId('54f4c5befba5220aa4d6dee7')}
```

The *upsert* option can be used to insert a new document if a matching document does not exist:

```
async def_replace_or_upsert():
    result = await db.test.replace_one({'x': 1}, {'x': 1}, True)
    print('matched %d, modified %d, upserted_id %r' %
        (result.matched_count, result.modified_count, result.upserted_id))

    print('collection:')
    async for doc in db.test.find():
        print(doc)
```

This prints:

```
matched 1, modified 1, upserted_id ObjectId('54f11e5c8891e756a6e1abd4')
collection:
{'y': 1, '_id': ObjectId('54f4c5befba5220aa4d6dee7')}
```

Parameters

- *filter*: A query that matches the document to replace.
- *replacement*: The new document.
- *upsert* (optional): If `True`, perform an insert if no documents match the filter.
- *bypass_document_validation*: (optional) If `True`, allows the write to opt-out of document level validation. Default is `False`.
- *collation* (optional): An instance of `Collation`. This option is only supported on MongoDB 3.4 and above.
- *hint* (optional): An index to use to support the query predicate specified either by its string name, or in the same format as passed to `create_index()` (e.g. `[('field', ASCENDING)]`). This option is only supported on MongoDB 4.2 and above.
- *session* (optional): a `ClientSession`, created with `start_session()`.

Returns

- An instance of `UpdateResult`.

Note: *bypass_document_validation* requires server version **>= 3.2**

Changed in version 2.2: Added `hint` parameter.

Changed in version 1.2: Added `session` parameter.

async update_many(*documents: List[*mdocument.document.MDocument*]*, **args*, ***kwargs*)
Updates multiple documents in database. Also updates related documents.

async update_one(*document:* mdocument.document.MDocument, **args*, ***kwargs*)
Updates one document in database. Also updates related documents.

watch(*pipeline=None*, *full_document=None*, *resume_after=None*, *max_await_time_ms=None*, *batch_size=None*, *collation=None*, *start_at_operation_time=None*, *session=None*, *start_after=None*)
Watch changes on this collection.

Performs an aggregation with an implicit initial `$changeStream` stage and returns a `MotorChangeStream` cursor which iterates over changes on this collection.

Introduced in MongoDB 3.6.

A change stream continues waiting indefinitely for matching change events. Code like the following allows a program to cancel the change stream and exit.

```
change_stream = None

async def watch_collection():
    global change_stream

    # Using the change stream in an "async with" block
    # ensures it is canceled promptly if your code breaks
    # from the loop or throws an exception.
    async with db.collection.watch() as change_stream:
        async for change in change_stream:
            print(change)

# Tornado
from tornado.ioloop import IOLoop

def main():
    loop = IOLoop.current()
    # Start watching collection for changes.
    loop.add_callback(watch_collection)
    try:
        loop.start()
    except KeyboardInterrupt:
        pass
    finally:
        if change_stream is not None:
            change_stream.close()

# asyncio
from asyncio import get_event_loop

def main():
    loop = get_event_loop()
    task = loop.create_task(watch_collection)

    try:
        loop.run_forever()
    except KeyboardInterrupt:
        pass
    finally:
        if change_stream is not None:
            change_stream.close()

        # Prevent "Task was destroyed but it is pending!"
        loop.run_until_complete(task)
```

The `MotorChangeStream` async iterable blocks until the next change document is returned or an error is raised. If the `next()` method encounters a network error when retrieving a batch from the server, it will automatically attempt to recreate the cursor such that no change events are missed. Any error encountered during the resume attempt indicates there may be an outage and will be raised.

```
try:
    pipeline = [{'$match': {'operationType': 'insert'}}]
    async with db.collection.watch(pipeline) as stream:
        async for change in stream:
            print(change)
```

(continues on next page)

(continued from previous page)

```
except pymongo.errors.PyMongoError:
    # The ChangeStream encountered an unrecoverable error or the
    # resume attempt failed to recreate the cursor.
    logging.error('...')
```

For a precise description of the resume process see the change streams specification.

Parameters

- *pipeline* (optional): A list of aggregation pipeline stages to append to an initial `$changeStream` stage. Not all pipeline stages are valid after a `$changeStream` stage, see the MongoDB documentation on change streams for the supported stages.
- *full_document* (optional): The fullDocument option to pass to the `$changeStream` stage. Allowed values: 'updateLookup'. When set to 'updateLookup', the change notification for partial updates will include both a delta describing the changes to the document, as well as a copy of the entire document that was changed from some time after the change occurred.
- *resume_after* (optional): A resume token. If provided, the change stream will start returning changes that occur directly after the operation specified in the resume token. A resume token is the _id value of a change document.
- *max_await_time_ms* (optional): The maximum time in milliseconds for the server to wait for changes before responding to a getMore operation.
- *batch_size* (optional): The maximum number of documents to return per batch.
- *collation* (optional): The `Collation` to use for the aggregation.
- *session* (optional): a `ClientSession`.
- *start_after* (optional): The same as *resume_after* except that *start_after* can resume notifications after an invalidate event. This option and *resume_after* are mutually exclusive.

Returns A `MotorChangeStream`.

See the tornado_change_stream_example.

Changed in version 2.1: Added the `start_after` parameter.

New in version 1.2.

with_options(*codec_options=None*, *read_preference=None*, *write_concern=None*, *read_concern=None*)

Get a clone of this collection changing the specified settings.

```
>>> coll1.read_preference
Primary()
>>> from pymongo import ReadPreference
>>> coll2 = coll1.with_options(read_preference=ReadPreference.SECONDARY)
>>> coll1.read_preference
Primary()
>>> coll2.read_preference
Secondary(tag_sets=None)
```

Parameters

- *codec_options* (optional): An instance of `CodecOptions`. If `None` (the default) the *codec_options* of this `Collection` is used.

- *read_preference* (optional): The read preference to use. If `None` (the default) the *read_preference* of this `Collection` is used. See `read_preferences` for options.
- *write_concern* (optional): An instance of `WriteConcern`. If `None` (the default) the *write_concern* of this `Collection` is used.
- *read_concern* (optional): An instance of `ReadConcern`. If `None` (the default) the *read_concern* of this `Collection` is used.

property write_concern
Read only access to the `WriteConcern` of this instance.

Changed in version 3.0: The *write_concern* attribute is now read only.

1.1.3 mdocument.client.MDocumentAsyncIOMotorDatabase

class mdocument.client.**MDocumentAsyncIOMotorDatabase**(*client*, *name*, ***kwargs*)
Bases: `motor.motor_asyncio.AsyncIOMotorDatabase`

__init__(*client*, *name*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(client, name, **kwargs)	Initialize self.
aggregate(pipeline, **kwargs)	Execute an aggregation pipeline on this database.
command(command[, value, check, ...])	Issue a MongoDB command.
create_collection(name[, codec_options, ...])	Create a new `Collection` in this database.
current_op([include_all, session])	**DEPRECATED**: Get information on operations currently running.
dereference(dbref[, session])	Dereference a `DBRef`, getting the document it points to.
drop_collection(name_or_collection[, session])	Drop a collection.
get_collection(name[, codec_options, ...])	Get a `Collection` with the given name and options.
get_io_loop()	
list_collection_names([session, filter])	Get a list of all the collection names in this database.
list_collections([session, filter])	Get a cursor over the collectons of this database.
profiling_info([session])	Returns a list containing current profiling information.
profiling_level([session])	Get the database's current profiling level.
set_profiling_level(level[, slow_ms, session])	Set the database's profiling level.
validate_collection(name_or_collection[, ...])	Validate a collection.
watch([pipeline, full_document, ...])	Watch changes on this database.
with_options([codec_options, ...])	Get a clone of this database changing the specified settings.
wrap(obj)	

Attributes

client	This MotorDatabase's MotorClient.
codec_options	Read only access to the CodecOptions of this instance.
incoming_copying_manipulators	All incoming SON copying manipulators.
incoming_manipulators	All incoming SON manipulators.
name	The name of this Database.
outgoing_copying_manipulators	All outgoing SON copying manipulators.
outgoing_manipulators	All outgoing SON manipulators.
read_concern	Read only access to the ReadConcern of this instance.
read_preference	Read only access to the read preference of this instance.
write_concern	Read only access to the WriteConcern of this instance.

aggregate(*pipeline*, ***kwargs*)

Execute an aggregation pipeline on this database.

Introduced in MongoDB 3.6.

The aggregation can be run on a secondary if the client is connected to a replica set and its read_preference is not PRIMARY. The *aggregate()* method obeys the *read_preference* of this MotorDatabase, except when $out or $merge are used, in which case PRIMARY is used.

All optional aggregate command parameters should be passed as keyword arguments to this method. Valid options include, but are not limited to:

- *allowDiskUse* (bool): Enables writing to temporary files. When set to True, aggregation stages can write data to the _tmp subdirectory of the –dbpath directory. The default is False.
- *maxTimeMS* (int): The maximum amount of time to allow the operation to run in milliseconds.
- *batchSize* (int): The maximum number of documents to return per batch. Ignored if the connected mongod or mongos does not support returning aggregate results using a cursor.
- *collation* (optional): An instance of Collation.

Returns a MotorCommandCursor that can be iterated like a cursor from find():

```
async def f():
    # Lists all operations currently running on the server.
    pipeline = [{"$currentOp": {}}]
    async for operation in client.admin.aggregate(pipeline):
        print(operation)
```

Note: This method does not support the 'explain' option. Please use MotorDatabase.command() instead.

Note: The MotorDatabase.write_concern of this database is automatically applied to this operation.

New in version 2.1.

property client
This MotorDatabase's `MotorClient`.

property codec_options
Read only access to the `CodecOptions` of this instance.

command(*command*, *value=1*, *check=True*, *allowable_errors=None*, *read_preference=None*, *codec_options=CodecOptions(document_class=dict, tz_aware=False, uuid_representation=UuidRepresentation.PYTHON_LEGACY, unicode_decode_error_handler='strict', tzinfo=None, type_registry=TypeRegistry(type_codecs=[], fallback_encoder=None))*, *session=None*, ***kwargs*)
Issue a MongoDB command.

Send command `command` to the database and return the response. If `command` is a string then the command `{command: value}` will be sent. Otherwise, `command` must be a `dict` and will be sent as-is.

Additional keyword arguments are added to the final command document before it is sent.

For example, a command like `{buildinfo: 1}` can be sent using:

```
result = await db.command("buildinfo")
```

For a command where the value matters, like `{collstats: collection_name}` we can do:

```
result = await db.command("collstats", collection_name)
```

For commands that take additional arguments we can use kwargs. So `{filemd5: object_id, root: file_root}` becomes:

```
result = await db.command("filemd5", object_id, root=file_root)
```

Parameters

- *command*: document representing the command to be issued, or the name of the command (for simple commands only).

  Note: the order of keys in the *command* document is significant (the "verb" must come first), so commands which require multiple keys (e.g. *findandmodify*) should use an instance of `SON` or a string and kwargs instead of a Python `dict`.

- *value* (optional): value to use for the command verb when *command* is passed as a string
- *check* (optional): check the response for errors, raising `OperationFailure` if there are any
- *allowable_errors*: if *check* is `True`, error messages in this list will be ignored by error-checking
- *read_preference*: The read preference for this operation. See `read_preferences` for options.
- *session* (optional): a `ClientSession`, created with `start_session()`.
- ***kwargs* (optional): additional keyword arguments will be added to the command document before it is sent

Changed in version 1.2: Added session parameter.

async create_collection(*name*, *codec_options=None*, *read_preference=None*, *write_concern=None*, *read_concern=None*, *session=None*, ***kwargs*)

Create a new `Collection` in this database.

Normally collection creation is automatic. This method should only be used to specify options on creation. `CollectionInvalid` will be raised if the collection already exists.

Options should be passed as keyword arguments to this method. Supported options vary with MongoDB release. Some examples include:

- "size": desired initial size for the collection (in bytes). For capped collections this size is the max size of the collection.
- "capped": if True, this is a capped collection
- "max": maximum number of objects if capped (optional)

See the MongoDB documentation for a full list of supported options by server version.

Parameters

- *name*: the name of the collection to create
- *codec_options* (optional): An instance of `CodecOptions`. If `None` (the default) the *codec_options* of this `Database` is used.
- *read_preference* (optional): The read preference to use. If `None` (the default) the *read_preference* of this `Database` is used.
- *write_concern* (optional): An instance of `WriteConcern`. If `None` (the default) the *write_concern* of this `Database` is used.
- *read_concern* (optional): An instance of `ReadConcern`. If `None` (the default) the *read_concern* of this `Database` is used.
- *collation* (optional): An instance of `Collation`.
- *session* (optional): a `ClientSession`.
- ***kwargs* (optional): additional keyword arguments will be passed as options for the create collection command

Changed in version 3.11: This method is now supported inside multi-document transactions with MongoDB 4.4+.

Changed in version 3.6: Added `session` parameter.

Changed in version 3.4: Added the collation option.

Changed in version 3.0: Added the codec_options, read_preference, and write_concern options.

Changed in version 2.2: Removed deprecated argument: options

current_op(*include_all=False*, *session=None*)

DEPRECATED: Get information on operations currently running.

Starting with MongoDB 3.6 this helper is obsolete. The functionality provided by this helper is available in MongoDB 3.6+ using the $currentOp aggregation pipeline stage, which can be used with *aggregate()*. Note that, while this helper can only return a single document limited to a 16MB result, *aggregate()* returns a cursor avoiding that limitation.

Users of MongoDB versions older than 3.6 can use the currentOp command directly:

```
# MongoDB 3.2 and 3.4
await client.admin.command("currentOp")
```

Or query the "inprog" virtual collection:

```
# MongoDB 2.6 and 3.0
await client.admin["$cmd.sys.inprog"].find_one()
```

Parameters

- *include_all* (optional): if `True` also list currently idle operations in the result
- *session* (optional): a `ClientSession`, created with `start_session()`.

Changed in version 2.1: Deprecated, use *`aggregate()`* instead.

Changed in version 1.2: Added session parameter.

`dereference`(*dbref*, *session=None*, ***kwargs*)

Dereference a `DBRef`, getting the document it points to.

Raises `TypeError` if *dbref* is not an instance of `DBRef`. Returns a document, or `None` if the reference does not point to a valid document. Raises `ValueError` if *dbref* has a database specified that is different from the current database.

Parameters

- *dbref*: the reference
- *session* (optional): a `ClientSession`.
- ***kwargs* (optional): any additional keyword arguments are the same as the arguments to `find()`.

Changed in version 3.6: Added `session` parameter.

`drop_collection`(*name_or_collection*, *session=None*)

Drop a collection.

Parameters

- *name_or_collection*: the name of a collection to drop or the collection object itself
- *session* (optional): a `ClientSession`.

Note: The `write_concern` of this database is automatically applied to this operation when using MongoDB >= 3.4.

Changed in version 3.6: Added `session` parameter.

Changed in version 3.4: Apply this database's write concern automatically to this operation when connected to MongoDB >= 3.4.

`get_collection`(*name*, *codec_options=None*, *read_preference=None*, *write_concern=None*, *read_concern=None*)

Get a `Collection` with the given name and options.

Useful for creating a `Collection` with different codec options, read preference, and/or write concern from this `Database`.

```
>>> db.read_preference
Primary()
>>> coll1 = db.test
>>> coll1.read_preference
Primary()
>>> from pymongo import ReadPreference
>>> coll2 = db.get_collection(
...     'test', read_preference=ReadPreference.SECONDARY)
>>> coll2.read_preference
Secondary(tag_sets=None)
```

Parameters

- *name*: The name of the collection - a string.
- *codec_options* (optional): An instance of `CodecOptions`. If `None` (the default) the *codec_options* of this `Database` is used.
- *read_preference* (optional): The read preference to use. If `None` (the default) the *read_preference* of this `Database` is used. See `read_preferences` for options.
- *write_concern* (optional): An instance of `WriteConcern`. If `None` (the default) the *write_concern* of this `Database` is used.
- *read_concern* (optional): An instance of `ReadConcern`. If `None` (the default) the *read_concern* of this `Database` is used.

property incoming_copying_manipulators

All incoming SON copying manipulators.

Changed in version 3.5: Deprecated.

New in version 2.0.

Type DEPRECATED

property incoming_manipulators

All incoming SON manipulators.

Changed in version 3.5: Deprecated.

New in version 2.0.

Type DEPRECATED

list_collection_names(*session=None*, *filter=None*, ***kwargs*)

Get a list of all the collection names in this database.

For example, to list all non-system collections:

```
filter = {"name": {"$regex": r"^(?!system\.)"}}
names = await db.list_collection_names(filter=filter)
```

Parameters

- *session* (optional): a `ClientSession`, created with `start_session()`.
- *filter* (optional): A query document to filter the list of collections returned from the listCollections command.

- ***kwargs* (optional): Optional parameters of the listCollections command can be passed as keyword arguments to this method. The supported options differ by server version.

Changed in version 2.1: Added the `filter` and `**kwargs` parameters.

New in version 1.2.

list_collections(*session=None*, *filter=None*, ***kwargs*)
Get a cursor over the collectons of this database.

Parameters

- *session* (optional): a `ClientSession`.
- *filter* (optional): A query document to filter the list of collections returned from the listCollections command.
- ***kwargs* (optional): Optional parameters of the listCollections command can be passed as keyword arguments to this method. The supported options differ by server version.

Returns An instance of `CommandCursor`.

New in version 3.6.

property name
The name of this `Database`.

property outgoing_copying_manipulators
All outgoing SON copying manipulators.

Changed in version 3.5: Deprecated.

New in version 2.0.

Type DEPRECATED

property outgoing_manipulators
All outgoing SON manipulators.

Changed in version 3.5: Deprecated.

New in version 2.0.

Type DEPRECATED

profiling_info(*session=None*)
Returns a list containing current profiling information.

Parameters

- *session* (optional): a `ClientSession`.

Changed in version 3.6: Added `session` parameter.

profiling_level(*session=None*)
Get the database's current profiling level.

Returns one of (`OFF`, `SLOW_ONLY`, `ALL`).

Parameters

- *session* (optional): a `ClientSession`.

Changed in version 3.6: Added `session` parameter.

property read_concern
Read only access to the `ReadConcern` of this instance.

New in version 3.2.

property read_preference
Read only access to the read preference of this instance.

Changed in version 3.0: The *read_preference* attribute is now read only.

set_profiling_level(*level*, *slow_ms=None*, *session=None*)
Set the database's profiling level.

Parameters

- *level*: Specifies a profiling level, see list of possible values below.
- *slow_ms*: Optionally modify the threshold for the profile to consider a query or operation. Even if the profiler is off queries slower than the *slow_ms* level will get written to the logs.
- *session* (optional): a `ClientSession`.

Possible *level* values:

Level	Setting
`OFF`	Off. No profiling.
`SLOW_ONLY`	On. Only includes slow operations.
`ALL`	On. Includes all operations.

Raises `ValueError` if level is not one of (`OFF`, `SLOW_ONLY`, `ALL`).

Changed in version 3.6: Added `session` parameter.

validate_collection(*name_or_collection*, *scandata=False*, *full=False*, *session=None*, *background=None*)
Validate a collection.

Returns a dict of validation info. Raises CollectionInvalid if validation fails.

See also the MongoDB documentation on the validate command.

Parameters

- *name_or_collection*: A Collection object or the name of a collection to validate.
- *scandata*: Do extra checks beyond checking the overall structure of the collection.
- *full*: Have the server do a more thorough scan of the collection. Use with *scandata* for a thorough scan of the structure of the collection and the individual documents.
- *session* (optional): a `ClientSession`.
- *background* (optional): A boolean flag that determines whether the command runs in the background. Requires MongoDB 4.4+.

Changed in version 3.11: Added `background` parameter.

Changed in version 3.6: Added `session` parameter.

watch(*pipeline=None*, *full_document=None*, *resume_after=None*, *max_await_time_ms=None*, *batch_size=None*, *collation=None*, *start_at_operation_time=None*, *session=None*, *start_after=None*)
Watch changes on this database.

Returns a `MotorChangeStream` cursor which iterates over changes on this database. Introduced in MongoDB 4.0.

See the documentation for `MotorCollection.watch()` for more details and examples.

Parameters

- *pipeline* (optional): A list of aggregation pipeline stages to append to an initial `$changeStream` stage. Not all pipeline stages are valid after a `$changeStream` stage, see the MongoDB documentation on change streams for the supported stages.
- *full_document* (optional): The fullDocument option to pass to the `$changeStream` stage. Allowed values: 'updateLookup'. When set to 'updateLookup', the change notification for partial updates will include both a delta describing the changes to the document, as well as a copy of the entire document that was changed from some time after the change occurred.
- *resume_after* (optional): A resume token. If provided, the change stream will start returning changes that occur directly after the operation specified in the resume token. A resume token is the _id value of a change document.
- *max_await_time_ms* (optional): The maximum time in milliseconds for the server to wait for changes before responding to a getMore operation.
- *batch_size* (optional): The maximum number of documents to return per batch.
- *collation* (optional): The `Collation` to use for the aggregation.
- *start_at_operation_time* (optional): If provided, the resulting change stream will only return changes that occurred at or after the specified `Timestamp`. Requires MongoDB >= 4.0.
- *session* (optional): a `ClientSession`.
- *start_after* (optional): The same as *resume_after* except that *start_after* can resume notifications after an invalidate event. This option and *resume_after* are mutually exclusive.

Returns A `MotorChangeStream`.

Changed in version 2.1: Added the `start_after` parameter.

New in version 2.0.

with_options(*codec_options=None*, *read_preference=None*, *write_concern=None*, *read_concern=None*)

Get a clone of this database changing the specified settings.

```
>>> db1.read_preference
Primary()
>>> from pymongo import ReadPreference
>>> db2 = db1.with_options(read_preference=ReadPreference.SECONDARY)
>>> db1.read_preference
Primary()
>>> db2.read_preference
Secondary(tag_sets=None)
```

Parameters

- *codec_options* (optional): An instance of `CodecOptions`. If `None` (the default) the *codec_options* of this `Collection` is used.

- *read_preference* (optional): The read preference to use. If `None` (the default) the *read_preference* of this `Collection` is used. See `read_preferences` for options.
- *write_concern* (optional): An instance of `WriteConcern`. If `None` (the default) the *write_concern* of this `Collection` is used.
- *read_concern* (optional): An instance of `ReadConcern`. If `None` (the default) the *read_concern* of this `Collection` is used.

New in version 3.8.

property write_concern
Read only access to the `WriteConcern` of this instance.

Changed in version 3.0: The *write_concern* attribute is now read only.

Exceptions

WrongQueryType

1.1.4 mdocument.client.WrongQueryType

exception mdocument.client.**WrongQueryType**

1.2 mdocument.document

Classes

MDocument(doc)
MetaDocument

1.2.1 mdocument.document.MDocument

class mdocument.document.**MDocument**(*doc*)
Bases: `object`

__init__(*doc*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(doc)	Initialize self.
create(*args, **kwargs)	Creates new document.
delete([session])	Deletes document.
find(query, **kwargs)	Finds multiple document.
items()	
many(query[, required, session])	Finds multiple documents.
one(query[, required])	Finds one document.
save([session])	Saves current document to database.

Attributes

—

exception DuplicateError
Bases: *mdocument.exceptions.DocumentException*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception NotFoundError
Bases: *mdocument.exceptions.DocumentException*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

async classmethod create(**args*, ***kwargs*)
Creates new document.

async delete(*session=None*)
Deletes document.

classmethod find(*query*, ***kwargs*)
Finds multiple document. Returns async generator.

async classmethod many(*query: dict*, *required: bool = False*, *session=None*)
Finds multiple documents.

async classmethod one(*query: dict*, *required: bool = True*)
Finds one document.

async save(*session=None*)
Saves current document to database.

1.2.2 mdocument.document.MetaDocument

class mdocument.document.**MetaDocument**
Bases: type

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(*args, **kwargs)	Initialize self.
mro()	Return a type's method resolution order.

Attributes

client
collection
database

mro()
Return a type's method resolution order.

1.3 mdocument.document_array

Classes

DocumentArray([initlist])

1.3.1 mdocument.document_array.DocumentArray

class mdocument.document_array.**DocumentArray**(*initlist=None*)
Bases: collections.UserList

__init__(*initlist=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([initlist])	Initialize self.
append(item)	S.append(value) – append value to the end of the sequence
clear()	
copy()	
count(value)	

continues on next page

Table 16 – continued from previous page

extend(other)	S.extend(iterable) – extend sequence by appending elements from the iterable
index(value, [start, [stop]])	Raises ValueError if the value is not present.
insert(i, item)	S.insert(index, value) – insert value before index
pop([index])	Raise IndexError if list is empty or index is out of range.
remove(item)	S.remove(value) – remove first occurrence of value.
reverse()	S.reverse() – reverse *IN PLACE*
sort(*args, **kwds)	

append(*item*)
S.append(value) – append value to the end of the sequence

clear() → None – remove all items from S

count(*value*) → integer – return number of occurrences of value

extend(*other*)
S.extend(iterable) – extend sequence by appending elements from the iterable

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

insert(*i*, *item*)
S.insert(index, value) – insert value before index

pop([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

remove(*item*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

reverse()
S.reverse() – reverse *IN PLACE*

1.4 mdocument.document_dict

Classes

DocumentDict(**kwargs)	Almost copy-paste of collections.UserDict but without locked .data attribute it was moved to dunder data (__data__) for allowing usage of data.

1.4.1 mdocument.document_dict.DocumentDict

class mdocument.document_dict.**DocumentDict**(***kwargs*)
Bases: collections.abc.MutableMapping

Almost copy-paste of collections.UserDict but without locked .data attribute it was moved to dunder data (__data__) for allowing usage of data. All dunder attributes are not passed to internal mapping.

Init does not create a new dict but links passed mapping.

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(**kwargs)	Initialize self.
clear()	
copy()	
fromkeys(iterable[, value])	
get(k[,d])	
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem()	as a 2-tuple; but raise KeyError if D is empty.
setdefault(k[,d])	
update([E,]**F)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
values()	

clear() → None. Remove all items from D.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

1.5 mdocument.exceptions

Exceptions

DocumentDoesntExist()	
DocumentException(message)	
DuplicateError()	
FieldIsNotJsonEncodable()	
NotFoundError()	
PrimaryKeyNotInSyncedFields()	
RelationNotSet()	
RequiredFieldMissing(field_name)	
UnknownModelField(field_name)	
WrongModelFieldType([extra_message])	
WrongValueType(field_name)	

1.5.1 mdocument.exceptions.DocumentDoesntExist

exception mdocument.exceptions.**DocumentDoesntExist**

1.5.2 mdocument.exceptions.DocumentException

exception mdocument.exceptions.**DocumentException**(*message*)

1.5.3 mdocument.exceptions.DuplicateError

exception mdocument.exceptions.**DuplicateError**

1.5.4 mdocument.exceptions.FieldIsNotJsonEncodable

exception mdocument.exceptions.**FieldIsNotJsonEncodable**

1.5.5 mdocument.exceptions.NotFoundError

exception mdocument.exceptions.**NotFoundError**

1.5.6 mdocument.exceptions.PrimaryKeyNotInSyncedFields

exception mdocument.exceptions.**PrimaryKeyNotInSyncedFields**

1.5.7 mdocument.exceptions.RelationNotSet

exception mdocument.exceptions.**RelationNotSet**

1.5.8 mdocument.exceptions.RequiredFieldMissing

exception mdocument.exceptions.**RequiredFieldMissing**(*field_name*)

1.5.9 mdocument.exceptions.UnknownModelField

exception mdocument.exceptions.**UnknownModelField**(*field_name*)

1.5.10 mdocument.exceptions.WrongModelFieldType

exception mdocument.exceptions.**WrongModelFieldType**(*extra_message=None*)

1.5.11 mdocument.exceptions.WrongValueType

exception mdocument.exceptions.**WrongValueType**(*field_name*)

1.6 mdocument.model

Modules

mdocument.model.field
mdocument.model.index
mdocument.model.model
mdocument.model.relations

1.6.1 mdocument.model.field

Classes

Field(field_type, default, optional, ...)	Basic field.
FieldRelated(document_cls, optional, ...)	Field where its value is a primary key value of a specific document.
FieldSynced(document_cls, optional, ...)	Field where its value is a copy of a specified document.
Synced()	Placeholder for identifying synced fields and models.

mdocument.model.field.Field

class mdocument.model.field.**Field**(*field_type: Type*, *default: Any = None*, *optional: bool = False*, *sensitive: bool = False*, *relation: Type[Relation] = None*, *unique: bool = False*)

Bases: object

Basic field.

__init__(*field_type: Type*, *default: Any = None*, *optional: bool = False*, *sensitive: bool = False*, *relation: Type[Relation] = None*, *unique: bool = False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(field_type[, default, optional, ...])	Initialize self.
connect_to_model(model, name)	Saves field model relation.
to_dict()	
validate(value)	Validates that provided value matches field type.

Attributes

SENSITIVE_PLACEHOLDER
is_primary
is_related

connect_to_model(*model: Type[Model]*, *name: str*)
Saves field model relation.

validate(*value*)
Validates that provided value matches field type.

mdocument.model.field.FieldRelated

class mdocument.model.field.**FieldRelated**(*document_cls: Type[MDocument]*, *optional: bool = False*, *relation: Type[Relation] = None*, *unique: bool = False*)

Bases: *mdocument.model.field.Field*

Field where its value is a primary key value of a specific document.

__init__(*document_cls: Type[MDocument]*, *optional: bool = False*, *relation: Type[Relation] = None*, *unique: bool = False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(document_cls[, optional, relation, ...])	Initialize self.
connect_to_model(model, name)	Saves field model relation.
to_dict()	
validate(value)	Validates that value matches related documents primary key type.

Attributes

SENSITIVE_PLACEHOLDER
is_primary
is_related

connect_to_model(*model: Type[Model]*, *name: str*)
Saves field model relation.

validate(*value*)
Validates that value matches related documents primary key type.

mdocument.model.field.FieldSynced

class mdocument.model.field.**FieldSynced**(*document_cls: Type[MDocument]*, *optional: bool = False*, *relation: Type[Relation] = None*, *unique: bool = False*, *sync_fields: List[str] = None*)

Bases: *mdocument.model.field.Field*

Field where its value is a copy of a specified document. Made for performance.

__init__(*document_cls: Type[MDocument]*, *optional: bool = False*, *relation: Type[Relation] = None*, *unique: bool = False*, *sync_fields: List[str] = None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(document_cls[, optional, relation, ...])	Initialize self.
connect_to_model(model, name)	Saves field model relation.
to_dict()	
validate(value)	Validates that value matches Document model.

Attributes

SENSITIVE_PLACEHOLDER
is_primary
is_related

connect_to_model(*model: Type[Model]*, *name: str*)
Saves field model relation.

validate(*value*)
Validates that value matches Document model.

mdocument.model.field.Synced

class mdocument.model.field.**Synced**
Bases: object

Placeholder for identifying synced fields and models.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
cut_data(document)	Cuts document to only needed fields or single field value.

classmethod cut_data(*document*)
Cuts document to only needed fields or single field value.

1.6.2 mdocument.model.index

Classes

Index(keys, int]], **kwargs)

mdocument.model.index.Index

class mdocument.model.index.**Index**(*keys: Tuple[Tuple[str, int]]*, ***kwargs*)
Bases: object

__init__(*keys: Tuple[Tuple[str, int]]*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(keys, **kwargs)	Initialize self.
connect_to_model(model, name)	Saves field model relation.

connect_to_model (*model: Type[Model]*, *name: str*)
Saves field model relation.

1.6.3 mdocument.model.model

Classes

MetaModel	
Model()	Document model.

mdocument.model.model.MetaModel

class mdocument.model.model.**MetaModel**
Bases: type

__init__ (**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(*args, **kwargs)	Initialize self.
mro()	Return a type's method resolution order.

mro()
Return a type's method resolution order.

mdocument.model.model.Model

class mdocument.model.model.**Model**
Bases: object

Document model. Should be JSON serializable.

All class fields that are not a part of model but are needed for internals should be named with first underscore.

Document model should be set in Document class with *Model* name.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
add_relation(relation)	Adds relation to global set of models relations.
fields()	
fields_dict()	
pop_optional_none(query)	Pops optional fields with None values.
validate(document[, pre_insert])	Validates that fields are present and have correct types.

classmethod add_relation(*relation: Relation*)
Adds relation to global set of models relations.

classmethod pop_optional_none(*query: dict*)
Pops optional fields with None values.

classmethod validate(*document: Union[MDocument, dict]*, *pre_insert=False*)
Validates that fields are present and have correct types.

1.6.4 mdocument.model.relations

Classes

Relation(parent_field, child_field)	
RelationManyToMany(parent_field, child_field)	Multiple parents multiple children.
RelationManyToOne(parent_field, child_field)	Multiple parents one child.
RelationOneToMany(parent_field, child_field)	One parent multiple children.
RelationOneToOne(parent_field, child_field)	One parent one child.

mdocument.model.relations.Relation

class mdocument.model.relations.**Relation**(*parent_field:* mdocument.model.field.Field, *child_field:* mdocument.model.field.Field)

Bases: object

__init__(*parent_field:* mdocument.model.field.Field, *child_field:* mdocument.model.field.Field)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(parent_field, child_field)	Initialize self.
create_index()	Creates index for relation.
delete(client, document[, session])	Deletes related documents.
get_collections(client)	Gets parent and child connection.
register(parent_field, child_field)	Saves relation to Model relations map.
update(client, document)	Updates related documents.

Attributes

child_document_cls
child_model
parent_document_cls
parent_model

abstract async create_index()
Creates index for relation.

abstract async delete(*client*, *document*, *session=None*)
Deletes related documents.

get_collections(*client*) → Tuple[AsyncIOMotorCollection, AsyncIOMotorCollection]
Gets parent and child connection.

classmethod register(*parent_field: Union[*mdocument.model.field.Field*, Type[*mdocument.document.MDocument*], Type[*mdocument.model.model.Model*]]*, *child_field:* mdocument.model.field.Field)
Saves relation to Model relations map.

abstract async update(*client*, *document*)
Updates related documents.

mdocument.model.relations.RelationManyToMany

class mdocument.model.relations.**RelationManyToMany**(*parent_field:* mdocument.model.field.Field, *child_field:* mdocument.model.field.Field)

Bases: *mdocument.model.relations.Relation*

Multiple parents multiple children.

Updates list when parent/child updates. Deletes all children when its all parents deleted.

__init__(*parent_field:* mdocument.model.field.Field, *child_field:* mdocument.model.field.Field)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(parent_field, child_field)	Initialize self.
create_index()	Creates index for relation.
delete(client, document[, session])	Deletes children with only one parent.
get_collections(client)	Gets parent and child connection.
register(parent_field, child_field)	Saves relation to Model relations map.
update(client, document)	Updates all childes fields from parent.

Attributes

child_document_cls
child_model
parent_document_cls
parent_model

abstract async create_index()
Creates index for relation.

async delete(*client: AsyncIOMotorClient*, *document: MDocument*, *session=None*)
Deletes children with only one parent. And removes parent from children.

get_collections(*client*) → Tuple[AsyncIOMotorCollection, AsyncIOMotorCollection]
Gets parent and child connection.

classmethod register(*parent_field: Union[*mdocument.model.field.Field*, Type[*mdocument.document.MDocument*], Type[*mdocument.model.model.Model*]]*, *child_field:* mdocument.model.field.Field)
Saves relation to Model relations map.

async update(*client*, *document*)
Updates all childes fields from parent.

mdocument.model.relations.RelationManyToOne

class mdocument.model.relations.**RelationManyToOne**(*parent_field:* mdocument.model.field.Field, *child_field:* mdocument.model.field.Field)

Bases: *mdocument.model.relations.Relation*

Multiple parents one child.

This means that child field type is a tuple. Updates child field when parent updates. Deletion happens when all parents are deleted.

__init__(*parent_field:* mdocument.model.field.Field, *child_field:* mdocument.model.field.Field)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(parent_field, child_field)	Initialize self.
create_index()	Creates index for relation.
delete(client, document[, session])	Deletes child with only one parent.
get_collections(client)	Gets parent and child connection.
register(parent_field, child_field)	Saves relation to Model relations map.
update(client, document[, session])	Updates child fields from parent.

Attributes

child_document_cls
child_model
parent_document_cls
parent_model

abstract async create_index()
Creates index for relation.

async delete(*client: AsyncIOMotorClient*, *document: MDocument*, *session=None*)
Deletes child with only one parent. And removes parent from children.

get_collections(*client*) → Tuple[AsyncIOMotorCollection, AsyncIOMotorCollection]
Gets parent and child connection.

classmethod register(*parent_field: Union[*mdocument.model.field.Field*, Type[*mdocument.document.MDocument*], Type[*mdocument.model.model.Model*]]*, *child_field:* mdocument.model.field.Field)
Saves relation to Model relations map.

async update(*client*, *document*, *session=None*)
Updates child fields from parent.

mdocument.model.relations.RelationOneToMany

class mdocument.model.relations.**RelationOneToMany**(*parent_field:* mdocument.model.field.Field, *child_field:* mdocument.model.field.Field)

Bases: *mdocument.model.relations.Relation*

One parent multiple children. Updates field when parent/child changes. Deletes children when parent deleted.

__init__(*parent_field:* mdocument.model.field.Field, *child_field:* mdocument.model.field.Field)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(parent_field, child_field)	Initialize self.
create_index()	Creates index for relation.
delete(client, document[, session])	Deletes all children if parent is deleted.
get_collections(client)	Gets parent and child connection.
register(parent_field, child_field)	Saves relation to Model relations map.
update(client, document[, session])	Updates all children field value if parent is updated.

Attributes

child_document_cls
child_model
parent_document_cls
parent_model

abstract async create_index()
Creates index for relation.

async delete(*client*, *document*, *session=None*)
Deletes all children if parent is deleted.

get_collections(*client*) → Tuple[AsyncIOMotorCollection, AsyncIOMotorCollection]
Gets parent and child connection.

classmethod register(*parent_field: Union[mdocument.model.field.Field, Type[mdocument.document.MDocument], Type[mdocument.model.model.Model]]*, *child_field: mdocument.model.field.Field*)
Saves relation to Model relations map.

async update(*client*, *document*, *session=None*)
Updates all children field value if parent is updated.

mdocument.model.relations.RelationOneToOne

class mdocument.model.relations.**RelationOneToOne**(*parent_field: mdocument.model.field.Field*, *child_field: mdocument.model.field.Field*)

Bases: *mdocument.model.relations.Relation*

One parent one child.

Updates field when parent/child changes. Creates unique index for field.

__init__(*parent_field: mdocument.model.field.Field*, *child_field: mdocument.model.field.Field*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(parent_field, child_field)	Initialize self.
create_index()	Creates index for relation.
delete(client, document[, session])	Deletes child if parent is deleted.
get_collections(client)	Gets parent and child connection.
register(parent_field, child_field)	Saves relation to Model relations map.
update(client, document)	Updates children field value.

Attributes

child_document_cls
child_model
parent_document_cls
parent_model

abstract async create_index()
Creates index for relation.

async delete(*client*, *document*, *session=None*)
Deletes child if parent is deleted.

get_collections(*client*) → Tuple[AsyncIOMotorCollection, AsyncIOMotorCollection]
Gets parent and child connection.

classmethod register(*parent_field: Union[*mdocument.model.field.Field*, Type[*mdocument.document.MDocument*], Type[*mdocument.model.model.Model*]]*, *child_field:* mdocument.model.field.Field)
Saves relation to Model relations map.

async update(*client*, *document*)
Updates children field value.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

U

V

W