Topic 09: Mongita Example - Introduction to MongoDB Concepts

Overview

This topic provides a **hands-on introduction to MongoDB concepts** through interactive exploration with **Mongita**. Unlike previous topics that built structured applications, this is an exploratory session where you experiment with document-oriented database operations directly in the Python REPL. This prepares you for working with MongoDB in subsequent topics by establishing core concepts in a simple, local environment.

What is Mongita?

Mongita is a lightweight, pure-Python implementation of MongoDB's API that:

- **Requires no server**: Runs entirely in Python process
- **Uses MongoDB syntax**: Commands work identically in MongoDB
- **Stores data locally**: Files persist on disk
- **Perfect for learning**: Explore concepts without infrastructure setup
- **Zero configuration**: Just pip install and start coding

Think of it as: SQLite for MongoDB - a learning and prototyping tool.

MongoDB Paradigm Shift

From Relational to Document-Oriented

SQL/Relational (Topics 1-7):

- Fixed schemas with tables and columns
- Relationships via foreign keys
- Rows must match table structure
- Schema changes require ALTER TABLE migrations

MongoDB/Document (Topics 8+):

- Flexible schemas with collections and documents
- Relationships via embedding or references
- Documents can have different fields
- Schema evolves naturally with application

Key Terminology

SQL Term	MongoDB Term	Description
Database	Database	Container for collections
Table	Collection	Container for documents
Row	Document	Individual data record

SQL Term	MongoDB Term	Description
Column	Field	Attribute of a document
Primary Key	_id	Unique identifier (auto-generated)
JOIN	Embedding/$lookup	Combining related data

Interactive Session Walkthrough

1. Installation and Setup

```
pip3 install mongita
```

Mongita has no external dependencies beyond Python - installs in seconds.

2. Connecting and Creating Database/Collection

```
from mongita import MongitaClientDisk
client = MongitaClientDisk()

# Access database (created automatically if doesn't exist)
hello_world_db = client.hello_world_db

# Access collection (created automatically if doesn't exist)
mongoose_collection = hello_world_db.mongoose_collection
```

Key observations:

- No schema definition needed
- No CREATE DATABASE or CREATE TABLE commands
- Database and collection created on first use
- Dot notation for access: `client.database.collection`

3. Inserting Documents

```
mongoose_collection.insert_many([
    {'name': 'Meercat', 'does_not_eat': 'Snakes'},
    {'name': 'Yellow mongoose', 'eats': 'Termites'}
])
```

MongoDB concepts demonstrated:

Flexible schema:

- First document has `does_not_eat` field

- Second document has `eats` field
- Both are valid - documents in same collection can have different fields
- No pre-defined schema required

Document structure:

- Documents are JSON-like dictionaries
- Fields can be strings, numbers, arrays, nested objects
- Each document automatically gets `_id` field (like auto-increment primary key)

Contrast with SQL:

```
-- SQL requires consistent columns
INSERT INTO animals (name, does_not_eat, eats)
VALUES ('Meercat', 'Snakes', NULL);  -- Must specify all columns
```

4. Counting Documents

```
mongoose_collection.count_documents({})
# Returns: 2
```

Query syntax:

- `{}` - Empty query matches all documents
- Like `SELECT COUNT(*) FROM table` in SQL
- Returns integer count

5. Updating Documents

```
mongoose_collection.update_one(
    {'name': 'Meercat'},           # Query: which document to update
    {'$set': {"weight": 2}}        # Update: what to change
)
```

MongoDB update operators:

- `$set` - Sets field value (adds field if doesn't exist)
- Query selects document to update
- Update specifies modifications

Result:

```
{'name': 'Meercat', 'does_not_eat': 'Snakes', '_id': ObjectId('...'),
'weight': 2}
```

Key insight: Added `weight` field without modifying schema - document evolved naturally!

Other update operators (not shown in session):

- `$unset` - Remove field
- `$inc` - Increment numeric value
- `$push` - Add to array
- `$pull` - Remove from array

6. Querying Documents

Query with condition:

```
cursor = mongoose_collection.find({'weight': {'$gt': 1}})
mongoose_list = list(cursor)
len(mongoose_list)
# Returns: 1
```

MongoDB query operators:

- `$gt` - Greater than
- `$lt` - Less than
- `$gte` - Greater than or equal
- `$lte` - Less than or equal
- `$ne` - Not equal
- `$in` - In array of values

Cursor pattern:

- `find()` returns a cursor (iterator)
- Must convert to list to see results
- Cursors are lazy - query executes when you iterate
- **Important**: Cursors are consumed - can only iterate once!

Why cursor consumed?

```
cursor = mongoose_collection.find({'weight': {'$gt': 1}})
list(cursor)  # [{'name': 'Meercat', ...}]
list(cursor)  # []  <- Empty! Cursor already consumed
```

Must re-query to iterate again:

```
cursor = mongoose_collection.find({'weight': {'$gt': 1}})
list(cursor)  # Fresh results
```

Find all documents:

```
list(mongoose_collection.find())
# Returns all documents in collection
```

Like `SELECT * FROM table` in SQL.

7. Deleting Documents

```
mongoose_collection.delete_one({'name': 'Meercat'})
```

Delete operations:

- `delete_one()` - Removes first matching document
- `delete_many()` - Removes all matching documents
- Query selects which documents to delete

Verify deletion:

```
list(mongoose_collection.find({'weight': {'$gt': 1}}))
# Returns: []
```

Document with `weight` field is gone.

8. Re-inserting

```
mongoose_collection.insert_many([
    {'name': 'Meercat', 'does_not_eat': 'Snakes'}
])
```

Observation: Re-inserted without `weight` field - demonstrates schema flexibility.

Final state:

```
list(mongoose_collection.find())
# [
#   {'name': 'Yellow mongoose', 'eats': 'Termites', '_id':
ObjectId('...')},
#   {'name': 'Meercat', 'does_not_eat': 'Snakes', '_id': ObjectId('...')}
# ]
```

Two documents with completely different fields - valid in MongoDB!

Key Concepts Explored

1. Schema-less Design

Documents in the same collection can have different fields. The "schema" is determined by what you insert, not pre-defined structure.

2. Automatic ID Generation

Every document gets a unique `_id` field with an ObjectId - no need to define primary keys.

3. Query Language

MongoDB uses a rich query language with operators like `$gt`, `$set`, `$in` instead of SQL strings.

4. Cursors

Queries return iterators (cursors) that are consumed when read - different from SQL result sets.

5. Collection Auto-creation

Collections and databases are created automatically when first accessed - no DDL commands needed.

6. Update Operators

Updates use operators like `$set` to modify documents - more flexible than SQL UPDATE.

7. Flexible Documents

Documents are JSON-like structures that can contain nested objects, arrays, and varied fields.

Comparing with SQL Operations

Operation	SQL	MongoDB
Insert	`INSERT INTO table VALUES (...)`	`collection.insert_one({...})`
Query all	`SELECT * FROM table`	`collection.find()`
Query filtered	`SELECT * WHERE col > 1`	`collection.find({'field': {'$gt': 1}})`
Update	`UPDATE table SET col=val WHERE id=1`	`collection.update_one({'id': 1}, {'$set': {'col': val}})`
Delete	`DELETE FROM table WHERE id=1`	`collection.delete_one({'id': 1})`
Count	`SELECT COUNT(*) FROM table`	`collection.count_documents({})`

Data Storage

Mongita stores data in local files:

```
# After running the session
ls
# hello_world_db/
#   mongoose_collection/
#     <data files>
```

Files persist between sessions - just like SQLite's .db file.

Experimenting Further

Try these variations:

```
# Insert document with nested structure
mongoose_collection.insert_one({
    'name': 'Dwarf mongoose',
    'diet': {
        'primary': 'Insects',
        'secondary': 'Small reptiles'
    },
    'habitat': ['Africa', 'Grasslands']
})

# Query nested field
list(mongoose_collection.find({'diet.primary': 'Insects'}))

# Query array
list(mongoose_collection.find({'habitat': 'Africa'}))

# Update with $inc (increment)
mongoose_collection.update_one(
    {'name': 'Meercat'},
    {'$inc': {'sightings': 1}}
)

# Delete multiple
mongoose_collection.delete_many({'weight': {'$lt': 3}})
```

What's Next

This hands-on session introduces core MongoDB concepts. Upcoming topics build on these foundations:

- **Topic 10**: MongoDB with Flask applications (using Mongita)
- **Topic 11**: MongoDB Atlas (cloud-hosted MongoDB)
- **Topic 12**: Local MongoDB server setup
- **Topic 13**: Advanced MongoDB queries and aggregation
- **Topic 14**: Modeling complex relationships
- **Topic 15**: MapReduce and data processing
- **Topic 16**: Geospatial queries

- **Topic 17**: MongoDB security

Key Takeaways

1. **No schema required** - documents define their own structure
2. **Automatic creation** - databases and collections created on first use
3. **Flexible documents** - different fields in same collection are valid
4. **Query operators** - `$gt`, `$set`, etc. instead of SQL syntax
5. **Cursors are consumed** - must re-query to iterate again
6. **ObjectId auto-generated** - like auto-increment primary keys
7. **Update operators** - `$set`, `$inc`, etc. for flexible modifications
8. **Mongita = MongoDB syntax** - everything learned applies to real MongoDB
9. **JSON-like structure** - documents are dictionaries with nested data
10. **Exploration encouraged** - interactive session reveals how MongoDB thinks

Running the Session

Option 1: Interactive Python REPL

```
pip3 install mongita
python
# Copy and paste commands from mongita-session-clean.py
```

Option 2: Script

```
python mongita-session-clean.py
```

Option 3: Jupyter Notebook

```
# Create notebook cells with each command
# Execute cells interactively to see results
```

Quick Reference

Essential commands:

```
# Connect
from mongita import MongitaClientDisk
client = MongitaClientDisk()
db = client.database_name
collection = db.collection_name

# CRUD Operations
```

```
collection.insert_one({'field': 'value'})
collection.insert_many([{...}, {...}])
collection.find({'field': 'value'})
collection.find_one({'field': 'value'})
collection.update_one({'query': 1}, {'$set': {'field': 'new'}})
collection.delete_one({'field': 'value'})
collection.count_documents({})

# Query operators
{'field': {'$gt': 5}}   # Greater than
{'field': {'$lt': 5}}   # Less than
{'field': {'$in': [1, 2, 3]}}  # In list
```

This interactive exploration establishes the mental model for document-oriented databases, preparing you for the MongoDB topics that follow. The hands-on approach makes abstract concepts concrete through immediate experimentation.