# MongoDB and PyMongo Comprehensive Notes

## PyMongo Fundamentals

- **Official Python Driver**: PyMongo serves as the official Python interface for interacting with MongoDB databases
- **Natural Mapping**: Translates MongoDB's document model directly to Python dictionaries, making data manipulation intuitive
- **Connection Management**:
  ```python
  from pymongo import MongoClient
  client = MongoClient('mongodb://user_name:pw@localhost:27017')
  ```
- **Connection String Components**:
  - Protocol: `mongodb://`
  - Authentication: `username:password@`
  - Host: `localhost` (or IP address/domain)
  - Port: `27017` (MongoDB default)
  - Additional parameters can be added with query string format

## Database and Collection Access

- **Multiple Access Syntaxes**:
  - Dictionary-style access: `db = client['ds4300']`
  - Attribute-style access: `db = client.ds4300`
- **Collection Selection**:
  - Dictionary-style: `collection = db['myCollection']`
  - Attribute-style: `collection = db.myCollection`
- **Lazy Creation**:
  - Databases and collections are created only when documents are first inserted
  - No explicit "create database" or "create collection" commands needed for basic usage

## Document Operations

### Inserting Documents

- **Single Document Insertion**:
  ```python
  post = {
    "author": "Mark",
    "text": "MongoDB is Cool!",
    "tags": ["mongodb", "python"]
  }
  post_id = collection.insert_one(post).inserted_id
  ```

  print(post_id) # Returns the _id of inserted document
  ~~~
- **Automatic ID Generation**:
  - MongoDB automatically assigns an ObjectId if no `_id` field is provided
  - ObjectIds contain timestamp information and are guaranteed unique within a collection
- **Return Values**:
  - `insert_one()` returns `InsertOneResult` object with `inserted_id` property
  - `insert_many()` returns `InsertManyResult` object with list of all IDs

### Querying Documents

- **Basic Find Operations**:
  ```python
  # Find with specific criteria
  movies_2000 = db.movies.find({"year": 2000})

  # Using bson.json_util.dumps for proper serialization of BSON types
  from bson.json_util import dumps
  print(dumps(movies_2000, indent=2))
  ~~~
- **Projection** (Field Selection):
  - Include specific fields with `{field: 1}`
  - Exclude specific fields with `{field: 0}`
  - The `_id` field is included by default unless explicitly excluded
- **Query Modifiers**:
  - `.sort()` - Control result order
  - `.limit()` - Restrict result count
  - `.skip()` - Skip initial results
  - Combinable: `collection.find({}).sort("field", -1).limit(5)`

## MongoDB Aggregation Framework

- **Pipeline Architecture**:
  - Series of data transformation stages
  - Each stage transforms documents and passes to next stage
  - Results only processed when needed (lazy evaluation)

### Common Aggregation Stages

- **$match**: Filters documents (similar to find's query parameter)
  ```python
  {"$match": {"year": {"$lte": 1920}}}
  ```

- **$project**: Reshapes documents (select, rename, compute fields)
  ```python
  {"$project": {"_id": 0, "title": 1, "cast": 1}}
  ```

- **$sort**: Orders documents by specified fields
  ```python
  {"$sort": {"title": 1}}  # 1 ascending, -1 descending
  ```

- **$limit**: Restricts number of documents
  ```python
  {"$limit": 5}
  ```

- **$unwind**: Deconstructs array field to create one document per array element
  ```python
  {"$unwind": "$cast"}  # Creates separate document for each cast member
  ```

- **$group**: Groups documents by key and applies accumulators
  ```python
  {"$group": {"_id": {"release year": "$year"}, "Avg Rating": {"$avg": "$imdb.rating"}}}
  ```

- **$lookup**: Performs left outer join with another collection
  ```python
  {"$lookup": {
      "from": "orders",
      "localField": "custid",
      "foreignField": "custid",
      "as": "orders"
  }}
  ```

### Aggregation Best Practices

- **Pipeline Structure**:
  - Place `$match` stages early to reduce documents processed in later stages
  - Use `$project` to limit fields when possible
  - Order matters: each stage affects what's passed to next stage

- **Readability Improvements**:
  ```python
  ```

  # Define stages separately for complex pipelines
  match = {"$match": {"year": {"$lte": 1920}}}
  limit = {"$limit": 5}
  project = {"$project": {"_id": 0, "title": 1, "cast": 1}}

  # Combine in aggregation call
  agg = mflixdb.movies.aggregate([match, limit, project])

## Query Patterns and Techniques

### Comparison Operators

- **Equality**: `{"field": value}`
- **Greater/Less Than**: `{"field": {"$gt": value}}`
- **In a Set**: `{"field": {"$in": [value1, value2]}}`
- **Multiple Conditions**: `{"$and": [{"field1": value1}, {"field2": value2}]}`

### Text Search and Regex

- **Regular Expression Search**:
  ```python
  # Equivalent to SQL's LIKE 'T%'
  {"name": {"$regex": "^T.*"}}
  ```
- **Text Search**:
  - Requires a text index on the collection
  - `{"$text": {"$search": "keywords"}}`

### Nested Document and Array Queries

- **Dot Notation** for nested fields:
  ```python
  {"address.city": "Boston, MA"}
  ```
- **Array Operations**:
  - Exact match: `{"tags": ["mongodb", "python"]}`
  - Contains element: `{"tags": "mongodb"}`
  - Element matching criteria: `{"tags": {"$elemMatch": {"$regex": "^m"}}}`

## Development Environment Setup

- **Python Environment Isolation**:
  - Conda or virtualenv recommended

  - `pip install pymongo` to add driver
  - `pip install jupyterlab` for interactive development

- **Jupyter Integration**:
  - Excellent for data exploration
  - Visual result examination
  - Cell-by-cell execution for incremental development
  - Magic commands for timing operations

## Performance Considerations

- **Indexing**:
  - Create indexes for frequently queried fields
  - Compound indexes for multi-field queries
  - Text indexes for full-text search
  - Explain plans to verify index usage

- **Query Optimization**:
  - Limit fields returned with projection
  - Filter early with specific `$match` criteria
  - Use proper data types (numbers stored as numbers, not strings)
  - Batch processing for large result sets

- **Connection Management**:
  - Connection pooling built into driver
  - Configure maxPoolSize for high-concurrency applications
  - Consider separate connections for read/write operations

## MongoDB Relationships

- **Embedding vs. Referencing**:
  - Embedding: nested documents within parent document
  - Referencing: storing IDs that point to documents in other collections
  - `$lookup` to perform joins between referenced collections

- **Modeling Approaches**:
  - One-to-few: typically embed
  - One-to-many: depends on growth and access patterns
  - Many-to-many: typically use references

- **Document Size Considerations**:
  - 16MB maximum document size
  - GridFS for larger files
  - Consider splitting very large documents

## Security Practices

- **Authentication**:
  - Use dedicated users with specific permissions
  - Never hardcode credentials in application code
  - Store connection strings in environment variables or config files

- **Network Security**:
  - Enable TLS/SSL for all connections
  - Firewall rules to restrict access
  - Use replica sets with internal authentication

- **Data Validation**:
  - Schema validation for document structure
  - Input sanitization before storing
  - Consider JSON Schema validation rules

These comprehensive notes cover both the basic operations shown in the slides and extend beyond with best practices, optimization strategies, and real-world application considerations for MongoDB and PyMongo development.

Aggregation Fundamentals

The MongoDB Aggregation Framework is a powerful tool for data processing and analysis that goes beyond simple queries. It's designed on the concept of data pipelines where documents flow through multiple stages of transformation.

Core Concepts

- **Pipeline Architecture**: Sequential series of data transformations
- **Document Flow**: Each document passes through all stages in order
- **Transformation Stages**: Each stage modifies the document stream in some way
- **Stage Operations**: Filter, group, sort, reshape, or calculate new values
- **Composability**: Complex operations built from simple building blocks

Key Advantages

- **Server-Side Processing**: Reduces network traffic and client-side computation
- **Optimized Execution**: MongoDB can optimize the pipeline for better performance

- **Expressive Power**: Can perform complex analytics directly in the database
- **Memory Management**: Uses streaming model to handle large datasets efficiently

Essential Aggregation Stages

$match Stage

The `$match` stage filters documents similar to the query in the `find()` method. It's typically placed early in the pipeline to reduce the number of documents processed in subsequent stages.

```
{
   "$match": {
      "year": {"$lte": 1920}
   }
}
```

Best Practices:

- Place `$match` early in the pipeline
- Use indexed fields in `$match` predicates
- Combine multiple conditions with `$and` when needed
- Filter documents as soon as possible to reduce processing load

$project Stage

The `$project` stage reshapes documents by specifying which fields to include, exclude, or modify.

```
{
   "$project": {
      "_id": 0,           # Exclude _id field
      "title": 1,         # Include title field
      "cast": 1,          # Include cast field
      "rating": "$imdb.rating"  # Create new field from existing nested field
   }
}
```

Capabilities:

- Include/exclude existing fields
- Rename fields

- Create computed fields
- Access nested document fields with dot notation
- Perform arithmetic operations
- Apply string transformations
- Manipulate date fields

$sort Stage

The `$sort` stage reorders documents based on specified fields. Value `1` for ascending order, `-1` for descending.

```
{
  "$sort": {
    "title": 1    # Sort by title in ascending order
  }
}
```

Multiple Sort Keys:

```
{
  "$sort": {
    "year": -1,   # Sort by year descending
    "title": 1    # Then by title ascending
  }
}
```

Performance Considerations:

- Sorting large result sets consumes memory
- Using an index for sort criteria improves performance
- When possible, limit documents before sorting with `$match`

$limit Stage

The `$limit` stage restricts the number of documents passed to the next stage.

```
{
  "$limit": 5
}
```

Usage Tips:

- Combine with `$sort` to implement "top N" queries
- Use after `$skip` for pagination
- Place after `$match` and `$sort` but before processing stages

$unwind Stage

The `$unwind` stage deconstructs an array field, creating one output document for each array element.

```
{
   "$unwind": "$cast"
}
```

Before unwinding (single document):

```
{
   "title": "Movie Title",
   "cast": ["Actor1", "Actor2", "Actor3"]
}
```

After unwinding (three documents):

```
{"title": "Movie Title", "cast": "Actor1"}
{"title": "Movie Title", "cast": "Actor2"}
{"title": "Movie Title", "cast": "Actor3"}
```

Advanced Options:

- `preserveNullAndEmptyArrays`: Keep documents with null/empty array fields
- `includeArrayIndex`: Add index field showing element position
- Applications: flattening data, cross-tabulation, analyzing array contents

$group Stage

The `$group` stage groups documents by a specified key and applies aggregation functions to create group-level fields.

```
{
   "$group": {
      "_id": {"release year": "$year"},  # Group by year
      "Avg Rating": {"$avg": "$imdb.rating"},  # Calculate average rating
      "Count": {"$sum": 1},  # Count documents in each group
```

```
        "Min Rating": {"$min": "$imdb.rating"},  # Find minimum rating
        "Max Rating": {"$max": "$imdb.rating"}   # Find maximum rating
    }
}
```

Common Accumulators:

- `$sum`: Calculate sum (or count when using `$sum: 1`)
- `$avg`: Calculate average
- `$min`, `$max`: Find minimum or maximum values
- `$first`, `$last`: Get first or last value when order matters
- `$push`: Create array with all values (can cause memory issues with large groups)
- `$addToSet`: Create array of unique values

$lookup Stage

The `$lookup` stage performs a left outer join with another collection.

```
{
    "$lookup": {
        "from": "orders",          # Join with orders collection
        "localField": "custid",    # Field from input documents
        "foreignField": "custid",  # Field from orders collection
        "as": "orders"             # Array field to add to input documents
    }
}
```

Result Structure:

- Original document fields + new array field containing matching documents
- Empty array if no matches found
- Performance implications for large collections
- Consider denormalization for frequently accessed data

Advanced Techniques

Pipeline Organization

For complex aggregations, organizing stages in variables improves readability and maintenance:

```
match = {"$match": {"year": {"$lte": 1920}}}
limit = {"$limit": 5}
project = {"$project": {"_id": 0, "title": 1, "cast": 1}}

agg = mflixdb.movies.aggregate([match, limit, project])
```

Multi-Stage Grouping

For hierarchical grouping or calculations that depend on previous groupings:

```
# First group by year to get movies per year
stage1 = {"$group": {"_id": "$year", "moviesPerYear": {"$sum": 1}}}

# Then group all to get average movies per year
stage2 = {"$group": {"_id": null, "averageMoviesPerYear": {"$avg": "$moviesPerYear"}}}

agg = db.movies.aggregate([stage1, stage2])
```

Working with Dates

Date operations require special handling:

```
dateGroup = {
   "$group": {
      "_id": {
         "year": {"$year": "$release_date"},
         "month": {"$month": "$release_date"}
      },
      "count": {"$sum": 1}
   }
}
```

Conditional Logic

Use `$cond` for if-then-else logic within aggregations:

```
{
   "$project": {
      "title": 1,
      "ageCategory": {
         "$cond": {
            "if": {"$gte": ["$year", 2000]},
        "then": "Modern",
        "else": "Classic"
      }
    }
  }
}
```

Performance Optimization

Efficient Pipeline Design

- **Filter Early**: Use `$match` as early as possible
- **Project Only Needed Fields**: Reduce memory usage with targeted projection
- **Index Usage**: Ensure operations use indexes when possible
- **Memory Limits**: Aggregation operations have a 100MB memory limit by default
- **Use `allowDiskUse`**: For large datasets that exceed memory limits

Analyzing Pipeline Performance

```
explain = db.movies.aggregate([match, group, sort], {"explain": True})
```

Aggregation vs. Map-Reduce

- Aggregation is generally faster and easier to use than Map-Reduce
- Aggregation leverages MongoDB's indexing and query optimizer
- For extremely complex operations, custom Map-Reduce might still be needed

Real-World Applications

Data Analytics

- **Time-Series Analysis**: Track metrics over time periods
- **Statistical Calculations**: Compute min, max, average, standard deviation
- **Top N Analysis**: Find most frequent values, highest performers

Business Intelligence

- **Sales Reporting**: Group transactions by product, region, time
- **Customer Segmentation**: Group users by behavior patterns
- **Inventory Analysis**: Calculate stock levels, turnover rates

Content Management

- **Content Metrics**: Count posts by category, author
- **Engagement Analysis**: Calculate average interactions per content type
- **Recommendation Preprocessing**: Calculate similarity or popularity scores

Geographical Data

- **Location Clustering**: Group data points by proximity
- **Regional Summaries**: Aggregate metrics by country, state, city
- **Distance Calculations**: Find items within specified distances

---

# MongoDB and PyMongo: Practical Query Examples

## Environment Setup and Connection

### Establishing MongoDB Connection

```python

import pymongo

from bson.json_util import dumps

# Connection string format: mongodb://[username:password@]host[:port]/[database][?options]

uri = "mongodb://username:password@localhost:27017"

client = pymongo.MongoClient(uri)

```

Connection String Components:

- **Protocol**: `mongodb://` (Use `mongodb+srv://` for Atlas clusters)

- **Authentication**: `username:password@` (Optional for unsecured development environments)

- **Host**: `localhost` or IP address/domain name

- **Port**: `27017` (Default MongoDB port)

- **Options**: Additional parameters as query string

Security Best Practices:

- Never hardcode credentials in production code

- Store connection strings in environment variables or secure configuration files

- Use dedicated users with minimal necessary permissions

- Enable TLS/SSL for all production connections

### Database and Collection Selection

```python
# Access a database
mflixdb = client.mflix  # Attribute-style access
demodb = client["demodb"]  # Dictionary-style access


# Access a collection
movies = mflixdb.movies
customers = demodb["customers"]
```

## Sample Data Management

### Creating Test Collections

```python

# Clear existing collections

demodb.customers.drop()

demodb.orders.drop()

# Sample customer data

customers = [

  {"custid": "C13", "name": "T. Cruise", "address": { "street": "201 Main St.", "city": "St. Louis, MO", "zipcode": "63101" }, "rating": 750 },

  {"custid": "C25", "name": "M. Streep", "address": { "street": "690 River St.", "city": "Hanover, MA", "zipcode": "02340" }, "rating": 690 },

  # Additional customers...

]

# Sample order data

orders = [

  { "orderno": 1001, "custid": "C41", "order_date": "2017-04-29", "ship_date": "2017-05-03",

    "items": [ { "itemno": 347, "qty": 5, "price": 19.99 }, { "itemno": 193, "qty": 2, "price": 28.89 } ] },

  # Additional orders...

]
```

```
# Insert data
demodb.customers.insert_many(customers)
demodb.orders.insert_many(orders)

# Verify insertion
numCustomers = demodb.customers.count_documents({})
numOrders = demodb.orders.count_documents({})
print(f'There are {numCustomers} customers and {numOrders} orders')
```

## Basic Query Operations

### Field Selection with Projection

```python
# Including specific fields
data = demodb.customers.find({}, {"name": 1, "rating": 1})
# Result includes _id automatically

# Excluding _id field explicitly
data = demodb.customers.find({}, {"name": 1, "rating": 1, "_id": 0})

# Excluding specific fields
data = demodb.customers.find({}, {"_id": 0, "address": 0})
```

# Result includes all fields EXCEPT _id and address

Projection Rules:

- Cannot mix inclusion and exclusion in same projection (except _id)

- `{"field": 1}` includes specific fields

- `{"field": 0}` excludes specific fields

- `_id` is always included unless explicitly excluded

### Pattern Matching with Regular Expressions

```python
# Find customers whose names start with 'T'
data = demodb.customers.find(
    {"name": {"$regex": "^T.*"}},
    {"_id": 0, "name": 1, "rating": 1}
)
```

Regular Expression Operators:

- `^` - Match beginning of line

- `$` - Match end of line

- `.` - Match any single character

- `.*` - Match any sequence of characters

- `[]` - Match any character in brackets

- `|` - Alternation (OR)

Case Sensitivity Options:

- `{"$regex": pattern, "$options": "i"}` for case-insensitive matching

- `{"$regex": "(?i)pattern"}` alternative syntax for case-insensitive

### Sort, Skip, and Limit

```python
# Sort customers by rating ascending, limit to 2 results
data = demodb.customers.find({}, {"_id": 0, "name": 1, "rating": 1}).sort("rating").limit(2)


# Sort in descending order (-1)
data = demodb.customers.find({}, {"_id": 0, "name": 1, "rating": 1}).sort("rating", -1).limit(2)


# Multiple sort keys
data = demodb.customers.find({}, {"_id": 0, "name": 1, "rating": 1}).sort([("rating", -1), ("name", 1)]).limit(2)
# Alternative syntax
data = demodb.customers.find({}, {"_id": 0, "name": 1, "rating": 1}).sort({"rating": -1, "name": 1}).limit(2)
```

Common Patterns:

- **Pagination**: `.skip(pageSize * (pageNum - 1)).limit(pageSize)`

- **Top N Records**: `.sort(key, -1).limit(N)`

- **Alphabetical Listing**: `.sort(nameField, 1)`

## Advanced Query Techniques

### Complex Filtering with Logical Operators

```python
# Find customers with rating between 600 and 700
data = demodb.customers.find({
    "$and": [
        {"rating": {"$gte": 600}},
        {"rating": {"$lte": 700}}
    ]
}, {"_id": 0, "name": 1, "rating": 1})

# Alternative syntax
data = demodb.customers.find({
    "rating": {"$gte": 600, "$lte": 700}
}, {"_id": 0, "name": 1, "rating": 1})
```

Logical Operators:

- `$and`: All conditions must match
- `$or`: At least one condition must match
- `$nor`: None of the conditions should match
- `$not`: Negates a condition

### Querying Nested Documents

```python
# Find customers from Boston
data = demodb.customers.find({
    "address.city": {"$regex": "^Boston"}
}, {"_id": 0, "name": 1, "address.city": 1})
```

### Array Queries

```python
# Find movies with specific actor in cast
data = mflixdb.movies.find({
    "cast": "Tom Hanks"
}, {"_id": 0, "title": 1, "year": 1})

# Find movies where at least one item in array matches condition
data = mflixdb.movies.find({
    "cast": {"$elemMatch": {"$regex": "^Tom"}}
}, {"_id": 0, "title": 1, "year": 1})
```

### Counting and Existence Checks

```python
# Count movies from 2000
count = mflixdb.movies.count_documents({"year": 2000})

# Find movies that have a director field
data = mflixdb.movies.find({
    "director": {"$exists": True}
}, {"_id": 0, "title": 1, "director": 1})

# Find movies without comments
data = mflixdb.movies.find({
    "comments": {"$exists": False}
}, {"_id": 0, "title": 1}).sort("title", 1)
```

## Practice Exercises

Complete these exercises using the mflix database to reinforce your MongoDB and PyMongo skills:

### Exercise 1: Basic Counting

```python
# How many Users are there in the mflix database? How many movies?
user_count = mflixdb.users.count_documents({})
movie_count = mflixdb.movies.count_documents({})
print(f"Users: {user_count}, Movies: {movie_count}")
```

### Exercise 2: Simple Filtering with Projection

```python
# Which movies have a rating of "TV-G"? Only return the Title and Year.
movies = mflixdb.movies.find(
    {"rated": "TV-G"},
    {"_id": 0, "title": 1, "year": 1}
)
print(dumps(movies, indent=2))
```

### Exercise 3: Numeric Range Queries

```python
# Which movies have a runtime of less than 20 minutes? Return title and runtime.
short_movies = mflixdb.movies.find(
    {"runtime": {"$lt": 20}},
    {"_id": 0, "title": 1, "runtime": 1}
)
print(dumps(short_movies, indent=2))
```

### Exercise 4: OR Conditions

```python
# How many theaters are in MN or MA?
theater_count = mflixdb.theaters.count_documents({
    "$or": [
        {"location.address.state": "MN"},
        {"location.address.state": "MA"}
    ]
})
print(f"Theaters in MN or MA: {theater_count}")
```

### Exercise 5: Existence Checks with Sorting

```python
# Give the names of all movies that have no comments yet, in alphabetical order.
no_comments = mflixdb.movies.find(
    {"comments": {"$exists": False}},
    {"_id": 0, "title": 1}
).sort("title", 1)
```

```
print(dumps(no_comments, indent=2))
```

### Exercise 6: Text Pattern Matching with Array Fields

```python
# Return movie titles and actors from any movie with "Four" in the title, sorted by title.
four_movies = mflixdb.movies.find(
    {"title": {"$regex": "Four", "$options": "i"}},
    {"_id": 0, "title": 1, "cast": 1}
).sort("title", 1)
print(dumps(four_movies, indent=2))
```

## Performance Optimization Tips

1. **Create Indexes** for frequently queried fields:

   ```python
   mflixdb.movies.create_index([("year", 1)])
   mflixdb.movies.create_index([("title", 1)])
   ```

2. **Use Explain Plans** to understand query performance:

   ```python
   explain = mflixdb.movies.find({"year": 2000}).explain()
   ```

3. **Limit Fields** with projection to reduce network transfer:

   ```python
   # Bad: Returns all fields
   mflixdb.movies.find({"year": 2000})

   # Good: Returns only needed fields
   mflixdb.movies.find({"year": 2000}, {"title": 1, "cast": 1, "_id": 0})
   ```

4. **Batch Processing** for large result sets:

   ```python
   cursor = mflixdb.movies.find({})
   batch_size = 100
   results = []

   for doc in cursor:
       results.append(doc)
       if len(results) >= batch_size:
           process_batch(results)
           results = []

   # Process any remaining documents
   ```

```
if results:
    process_batch(results)
```

5. **Use Aggregation** for complex data transformations:

```python
pipeline = [
    {"$match": {"year": {"$gte": 2000}}},
    {"$group": {"_id": "$year", "count": {"$sum": 1}}},
    {"$sort": {"_id": 1}}
]
result = mflixdb.movies.aggregate(pipeline)
```