**Python and MongoDB**

*(Learning from DataCamp’s Introduction to MongoDB in Python)*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Descriptions** | | | | | |
| What is **MongoDB** | * NoSQL database * Dynamic schema, able to hold vastly different data together * Basis of data format: JSON * Allows specification and enforcement of a schema for collection, though not required * Fields do not need to have same type of value across documents in a collection * Field presence * Root-level or any level fields do not need to be present in all documents | |  | | |
| What is an **index** in MongoDB | * Like an index at the back of a book * Used when querying with high specificity, with large documents/collections * Efficient compute across collection * Not needed with a small collection that fits in memory (less than 1000 documents) | |  | | |
|  | * **Databases** * Maps names to collections * Keys are collection names * dict * **Collections** * Accessible by name like in a dictionary * List of documents (i.e. dictionaries) * list * **Documents** * Subdocuments: dictionary as value within document | |  | | |
| What is **JSON** | * JavaScript Object Notation * Common way that web services and client code pass data * 2 collection structures: objects {} and arrays [] | |  | | |
| What is a JSON **object** **{}** | * Maps string 🡪 value * Order of values not important | | Examples:  {‘key1’:value1, ‘key2’:value2, …}  {  ‘id’: 12345,  ‘name’: ‘Donny Winston’,  ‘instructor’: true  } | | |
| What is a JSON **array []** | * Series of values * Orders values important | | Examples:  [value1, value2, …]  [  “instructor\_1”,  “instructor\_2”  ] | | |
| What is a **value** in JSON | Can be   * String * Number * “true”, “false”, “null” * Another object/array | | Examples:  ‘name’:’Donny Winston’ (**string**)  ‘id’: 12345 (**number**)  true / false  null  ‘tags’: [‘Python’, ‘MongoDB’] (**array**)  [{‘id’: 12345, …}, …] (**object**) | | |
| Relationship between JSON, Python and MongoDB | JSON | Python | MongoDB | |  |
| Objects | Dictionaries (dict) | Databases | |
| Arrays | Lists (list) | Collections | |
| strings | str | Value types + datetime, regex | |
| \_numbers\_ | int, float |
| true/false | True / False |
| null | None |
| Other objects/arrays | Other dict / list | Documents, Subdocuments | |
| What is **Pymongo** | * Official Python driver for MongoDB | |  | | |
| What is an **ISO 8601 format** | * Values of form ‘YYYY-MM-DD’ | | Example:  “1937-02-01” | | |
| How to **sort** in MongoDB **shell** | * Use *sort* argument * Input as dictionary * JavaScript objects retain key order as entered (stable sort) | | # Example  .sort({“year”: 1, “category”: -1}) | | |
|  |  | |  |  | |
|  |  | |  |  | |
|  |  | |  |  | |
| Gauging performance in Python | Python console | | python -m timeit “[expression]” | | |
| Jupyter notebook | | %%timeit  #[expression] | | |
|  |  | |  |  | |
|  |  | |  |  | |
|  |  | |  |  | |

|  |  |  |
| --- | --- | --- |
| **pymongo functions** | | |
| Import library |  | from pymongo import MongoClient |
| Connect to DB | Connect to local database server   * By default, localhost | client = MongoClient() |
|  | Connect to non-local database server | client = MongoClient(**host**=”10.9.8.1”, **port**=11823) |
|  | Connect to database server within same Kubernetes cluster | client = MongoClient(“**mongodb**://<service-name>.<namespace>.svc.cluster.local:<port>”) |
| Create database |  | db = client[“nobel”] |
| Create collection |  | db[collection\_name].insert\_many(documents) |
| Access database and collections | Two ways to access:   * Use square bracket notation [] * Use dot notation . | # Use collection name as keys  db = client[“prizes”]  # Use collection name as attribute of database |
| List accessible databases | Use on a client instance  Default databases on every Mongo host:   * admin, local : for internal bookkeeping * system.indexes : stores indexes that make searches faster | .list\_database\_names() |
| List collections of a database | Use on a database instance | .list\_collection\_names() |
| Insert **many** document | Insert many documents in a collection   * *document* : list of dictionaries | .insert\_many(documents) |
| Count documents in a collection | * *filter :* empty list(no filter) | # Count documents  filter = {}  n\_prizes = db.prizes.count\_documents(filter) |
| Select **one** document | Returns a dictionary   * *filter*: dictionary (optional), specifies the pattern document must match * Keys of dictionary are root-level “fields” of document   Like *limit = 1* with automatic fetching from cursor | db.prizes.find\_one()  # Return first document in internal order of collection  doc = db.prizes.find\_one({}) |
| Find documents | Return a *Cursor*   * *filter* * *projection* * *limit:* integer, specified number of documents to return * *skip:* integer, skips results server-side | db.prizes.find(…, limit = 3, skip = 3) |
| Cursor methods | Can do method chaining   * *Sort* * *Skip* * *Limit* | db.prizes.find(…).sort([(“year”, 1)])  .skip(3)  .limit(3)  db.prizes.find(…).limit(3) |
| Find set of distinct values for a field | Collects the distinct values that the specified field across the collection   * An aggregation function * Efficient if there is a collection *index* on the field | # Returns [‘male’, ‘female’, ‘org’]  db.laureates.distinct(“gender”)  db.laureates.distinct(<field>, {<filter>}) |
|  |  |  |
| Sort documents | Option 1: use python in-built library | from operator import itemgetter  docs = list(db.prizes.find(…))  # Sort in descending  docs = sorted(docs, key=itemgetter(“year”), reverse = True) |
| Option 2: ask mongo to do simple sorting by field values first   * Sort ascending, descending order * Sort by multiple fields (need tuples to ensure stable sort) | cursor = db.prizes.find(  {“category”: “physics”},  [“year”],  # Sort by ‘year’ field, ascending order (1)  sort = [(“year”, 1)]  # OR: Sort in descending order (-1)  sort = [(“year”, -1)]  # OR: Sort by multiple fields  sort = [(“year”, 1), (“category”, -1)]  )  print([doc[“year”] for doc in cursor][:5]) |
| Add a single-index field | * Index model: list of *(field, direction)* pairs * Directions: ascending (1 - default), descending (-1) * Improves performance when querying | db.prizes.create\_index([(“year”, 1)]) |
| Add a compound (multiple-field) index | * Mongo does not have to examine collection itself to execute query * Query with projection is “covered” by index * Huge performance gains for common queries * Indexes also take up much less space than collections | db.prizes.create\_index([  (“category”, 1),  (“year”, 1)  ]) |
| Get existing indexes for a collection | * Always an index on “*\_id*” field | db.laureates.index\_information()  # Returns  {“\_id”: {  “v”: 2,  “key”: [(“\_id”, 1)],  ‘ns’: ‘nobel.laureates”  }} |
| Detail how a given query will execute | * *Explain* method of a *Cursor* * Details how a given query will execute * *COLLSCAN*: collection scan * *IXSCAN*: index scan (after creating an index) | db.laureates.find(…).explain()  # Returns  …  ‘winningPlan’: {‘stage’: ‘PROJECTION’,  ‘transformBy’: {‘bornCountry’: 1, ‘\_id’: 0},  ‘inputStage’: {‘stage’: ‘COLLSCAN’,  … |
| Aggregation | * Makes queries explicit (no longer as a input param) * Into aggregation stages | from collections import OrderedDict  db.laureates.aggregate([  {“$match”: {…}},  {“$project”: {…}},  {“$sort”: OrderedDict([“prizes.year”, 1])},  {“$skip”: 1},  {“$limit”: 3},  # Key you specify  {“$count”: “n\_USA-born-laureates”}  ])  # Returns  {“ n\_USA-born-laureates”: 354} |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Parameters, values, structures** | | | | |
| Filters | * Sub-documents * Mirrors structure of documents to match in collection | | | filter\_doc = {  ‘gender’: ‘female’,  ‘diedCountry’: ‘France’,  ‘bornCity’: ‘Warsaw’  } |
| 2-stage pipeline   1. Filter collection for documents that match a filter 2. Then, collect and return distinct values of a field for these documents | | | # Pre-filtering distinct values  db.laureates.distinct(  “prizes.category”,  {“prizes.share”: ‘4’} # filter arg  ) |
| Matching array fields | * *$elemMatch* only considers cases when value of *<field>* is an array is non-empty * Where *<field>* as in the field one layer above *$elemMatch* * Else, it will not consider testing those not matching criteria   \*\* Possible format combinations:  **{ “field”: value }**  **{ “field”: { “$op”: value }, … }**  **{ “field”: { “$op”: { … }, … }}**  where … is the same variation of combinations (nest) | | | db.lauretes.count\_documents({  “prizes”: { “$elemMatch”: {  “category”: “physics”,  “share”: “1”,  “year”: {“lt”: “1945}  }}  }) |
| Query operators | * Place in a filter document * Wrap around a field and its acceptable values * Possible to compose query operators for a field * Alphabetical order for non-numeric values (lexicographically) | | | {  # Match single value exactly  ‘field\_name1’: value1,  # Match any value in dict  ‘field\_name2’: {  $operator1: value1,  $operator2: value2,  … # more operators  },  … # more fields  }  {‘diedCountry’ : {  ‘$in’: [‘France’, ‘USA’],  ‘$ne’: ‘France’,  …  }}  # Value is another operator-value pair  criteria = {‘born’: {‘lt’: ‘1900’}} |
| What | Operator | Value |
| Value in a range | ‘$in’ | <list> |
| Not in list | ‘$nin’ | <list> |
| Not equal | ‘$ne’ | <value> |
| Greater than (>) | ‘$gt’ | <value> |
| Greater than or equal to (>=) | ‘$gte’ | <value> |
| Less than (<) | ‘$lt’ | <value> |
| Less than or equal to (<=) | ‘$lte’ | <value> |
| Field exists or not | ‘$exists’ | True / False |
| Dot notation | * Use to query arrays, subdocuments (their sub-structure) * When value of keys are complex dictionaries/arrays * Can reference using numeric index | | | db.laureates.count\_documents({  “prizes.affiliations.name”: (“University of California”)  })  # Check for existence of array  {“db.laureates.prizes.0” : {‘$exists’: True}}  # Filter for laureates with at least three prizes (i.e. at least 3 values in array/subdocument)  criteria = {‘prizes.affiliation.2’: {‘$exists’: True}} |
| Regular expressions | * pymongo driver includes *bson* package with Regex class * Not recommended to use *re* library, *bson* package more robust for MongoDB * No anchoring values, means can appear anywhere in string | | | # Matches like “\*Poland\*”  db.laureates.distinct(  “bornCountry”,  {“bornCountry”: {“$regex”: “Poland”}}  )  # Ensure case-insensitive matching (i)  {“$regex”: “poland”, “$options”: “i”}  # Compiled regex objects  from bson.regex import Regex  {“bornCountry”: Regex(“poland”, “I”)} |
| Match beginning (^)/end ($) of string | | | {“bornCountry”: Regex(“^Poland”})  {“bornCountry”: Regex(“now Poland\)$”)} |
| Escape special characters (\)   * Parenthesis () | | | {“bornCountry”: Regex(“Poland \(now”)} |
| Projection | * Reducing dimension of data * Specified as **dictionary** * Fields not included in dictionary are not included in projection * *\_id* is included by default * Returns a *cursor* (*pymongo.cursor.Cursor*), an iterable | | | # Returns data only about prizes.affiliations  db.laureates.find(  filter={},  projection={“prizes.affiliations”: 1, “\_id”: 0}  ) |
| * Specified as a **list** * Only projected fields that exist are returned * Documents without field present will just return *\_id* | | | db.laureates.find(  filter={},  projection=[“bornCountry”, “firstname”]  ) |
| List comprehension | * Reduces memory overhead | | | # Filter + projection  docs = list(db.prizes.find(  {“category”: “physics”},  [“year”]  ))  print([doc[“year”] for doc in docs][:5]) |
| Limit, Skip | Return a *Cursor*   * *limit:* integer, specified number of documents to return * *skip:* integer, skips results server-side | | | db.prizes.find(…, limit = 3)  db.prizes.find(…, skip = 3)  # Pagination (limit number of results per page)  db.prizes.find(…, skip = 3, limit = 3) |
|  |  | | |  |
|  |  | | |  |
|  |  | | |  |