***Document Structure***

The values in a document can be any data type, including strings, objects, arrays, booleans, nulls, dates, ObjectIds, and more. Here's the syntax for a MongoDB document, followed by an example:

{

"\_id": 1,

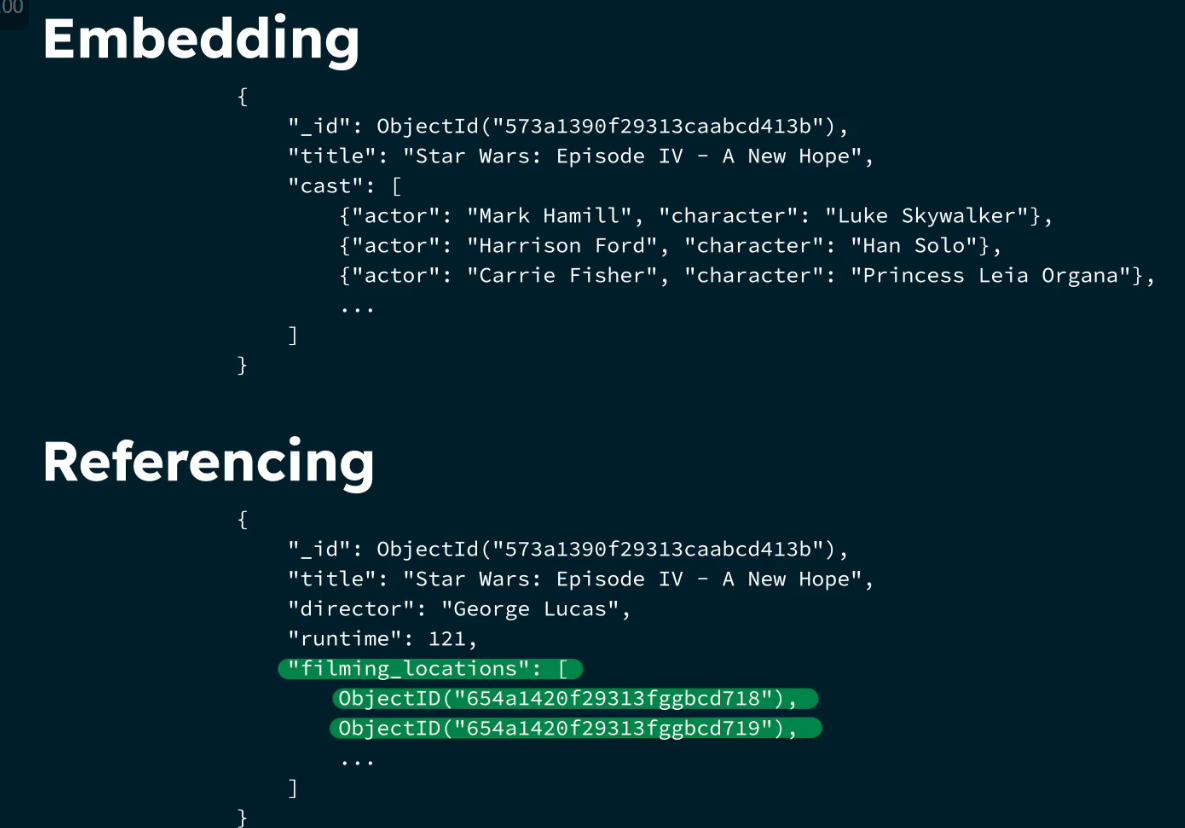
"name": "AC3 Phone",

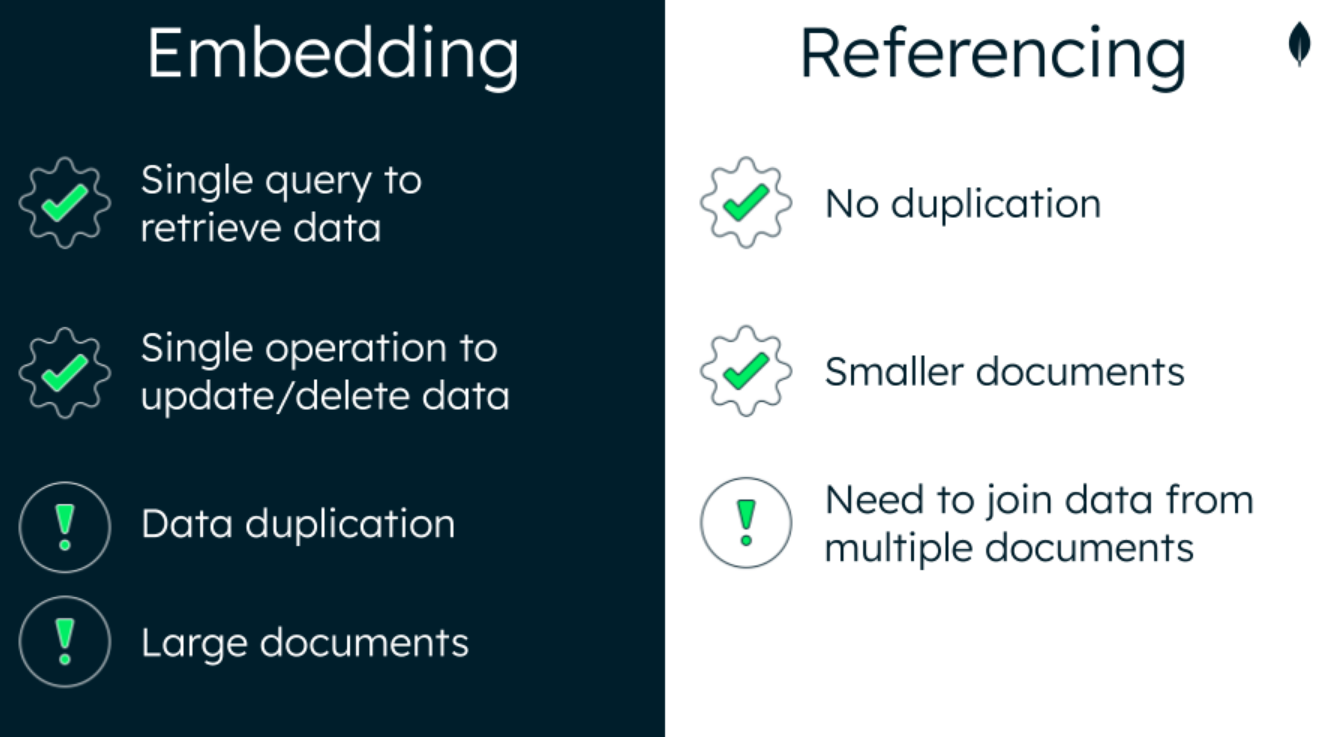
"colors" : ["black", "silver"],

"price" : 200,

"available" : true

}





***Finding Documents in a MongoDB Collection:***

* Find a Document with Equality

When given equality with an \_id field, the find() command will return the specified document that matches the \_id. Here's an example:

db.zips.find({ \_id: ObjectId("5c8eccc1caa187d17ca6ed16") })

* Find a Document by Using the $in Operator

Use the $in operator to select documents where the value of a field equals any value in the specified array. Here's an example:

db.zips.find({ city: { $in: ["PHOENIX", "CHICAGO"] } })

***Finding Documents by Using Comparison Operators:***

* $gt

Use the $gt operator to match documents with a field **greater than** the given value. For example:

db.sales.find({ "items.price": { $gt: 50}})

* $lt

Use the $lt operator to match documents with a field **less than** the given value. For example:

db.sales.find({ "items.price": { $lt: 50}})

* $lte

Use the $lte operator to match documents with a field **less than or equal to** the given value. For example:

db.sales.find({ "customer.age": { $lte: 65}})

* $gte

Use the $gte operator to match documents with a field **greater than or equal to** the given value. For example:

db.sales.find({ "customer.age": { $gte: 65}})

***Querying on Array Elements in MongoDB:***

* Find Documents with an Array That Contains a Specified Value

In the following example, "InvestmentFund" is not enclosed in square brackets, so MongoDB returns all documents within the products array that contain the specified value.

db.accounts.find({ products: "InvestmentFund"})

* Find a Document by Using the $elemMatch Operator

Use the $elemMatch operator to find all documents that contain the specified subdocument. For example:

db.sales.find({

items: {

$elemMatch: { name: "laptop", price: { $gt: 800 }, quantity: { $gte: 1 } },

},

})

***Finding Documents by Using Logical Operators:***

* Find a Document by Using Implicit $and

Use implicit $and to select documents that match multiple expressions. For example:

db.routes.find({ "airline.name": "Southwest Airlines", stops: { $gte: 1 } })

* Find a Document by Using the $or Operator

Use the $or operator to select documents that match at least one of the included expressions. For example:

db.routes.find({

$or: [{ dst\_airport: "SEA" }, { src\_airport: "SEA" }],

})

* Find a Document by Using the $and Operator

Use the $and operator to use multiple $or expressions in your query.

db.routes.find({

$and: [

{ $or: [{ dst\_airport: "SEA" }, { src\_airport: "SEA" }] },

{ $or: [{ "airline.name": "American Airlines" }, { airplane: 320 }] },

]

})

***Replacing a Document in MongoDB:***

To replace documents in MongoDB, we use the replaceOne() method. The replaceOne() method takes the following parameters:

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

db.books.replaceOne(

{

\_id: ObjectId("6282afeb441a74a98dbbec4e"),

},

{

title: "Data Science Fundamentals for Python and MongoDB",

isbn: "1484235967",

publishedDate: new Date("2018-5-10"),

thumbnailUrl:

"https://m.media-amazon.com/images/I/71opmUBc2wL.\_AC\_UY218\_.jpg",

authors: ["David Paper"],

categories: ["Data Science"],

}

)

***Updating MongoDB Documents by Using updateOne():***

The updateOne() method accepts a filter document, an update document, and an optional options object. MongoDB provides update operators and options to help you update documents. In this section, we'll cover three of them: $set, upsert, and $push.

* $set

The $set operator replaces the value of a field with the specified value, as shown in the following code:

db.podcasts.updateOne(

{

\_id: ObjectId("5e8f8f8f8f8f8f8f8f8f8f8"),

},

{

$set: {

subscribers: 98562,

},

}

)

* upsert

The upsert option creates a new document if no documents match the filtered criteria. Here's an example:

db.podcasts.updateOne(

{ title: "The Developer Hub" },

{ $set: { topics: ["databases", "MongoDB"] } },

{ upsert: true }

)

* $push

The $push operator adds a new value to the hosts array field. Here's an example:

db.podcasts.updateOne(

{ \_id: ObjectId("5e8f8f8f8f8f8f8f8f8f8f8") },

{ $push: { hosts: "Nic Raboy" } }

)

***Updating MongoDB Documents by Using findAndModify():***

The findAndModify() method is used to find and replace a single document in MongoDB. It accepts a filter document, a replacement document, and an optional options object. The following code shows an example:

db.podcasts.findAndModify({

query: { \_id: ObjectId("6261a92dfee1ff300dc80bf1") },

update: { $inc: { subscribers: 1 } },

new: true,

})

***Updating MongoDB Documents by Using updateMany():***

To update multiple documents, use the updateMany() method. This method accepts a filter document, an update document, and an optional options object. The following code shows an example:

db.books.updateMany(

{ publishedDate: { $lt: new Date("2019-01-01") } },

{ $set: { status: "LEGACY" } }

)

***Deleting Documents in MongoDB***

To delete documents, use the deleteOne() or deleteMany() methods. Both methods accept a filter document and an options object.

* Delete One Document

db.podcasts.deleteOne({ \_id: Objectid("6282c9862acb966e76bbf20a") })

* Delete Many Documents

db.podcasts.deleteMany({category: “crime”})

***Sorting and Limiting Query Results in MongoDB***

Sorting Results

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

Syntax:

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

Example:

Limiting Results

Use cursor.limit() to return query results in a specified order. Within the parentheses of limit(), specify the maximum number of documents to return.

Syntax:

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

Example:

// Return the three music companies with the highest number of employees. Ensure consistent sort order.

db.companies

.find({ category\_code: "music" })

.sort({ number\_of\_employees: -1, \_id: 1 })

.limit(3);

***Returning Specific Data from a Query in MongoDB***

Add a Projection Document

To specify fields to include or exclude in the result set, add a projection document as the second parameter in the call to db.collection.find().

Syntax:

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

Include a Field

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

Syntax:

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

Exclude a Field

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

Syntax:

db.collection.find(query, { <field> : 0, <field>: 0 })

While the \_id field is included by default, it can be suppressed by setting its value to 0 in any projection.

// Return all restaurant inspections - business name and result fields only

db.inspections.find(

{ sector: "Restaurant - 818" },

{ business\_name: 1, result: 1, \_id: 0 }

)

***Counting Documents in a MongoDB Collection***

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

The query selects the documents to be counted.

Examples:

// Count number of docs in trip collection

db.trips.countDocuments({})

// Count number of trips over 120 minutes by subscribers

db.trips.countDocuments({ tripduration: { $gt: 120 }, usertype: "Subscriber" })

***Introduction to MongoDB Aggregation:***

This section contains key definitions for this lesson, as well as the code for an aggregation pipeline.

Definitions

* **Aggregation**: Collection and summary of data
* **Stage**: One of the built-in methods that can be completed on the data, but does not permanently alter it
* **Aggregation pipeline**: A series of stages completed on the data in order

Structure of an Aggregation Pipeline

db.collection.aggregate([

{

$stage1: {

{ expression1 },

{ expression2 }...

},

$stage2: {

{ expression1 }...

}

}

])

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

* $match

The $match stage filters for documents that match specified conditions. Here's the code for $match:

{

$match: {

"field\_name": "value"

}

}

* $group

The $group stage groups documents by a group key.

{

$group:

{

\_id: <expression>, // Group key

<field>: { <accumulator> : <expression> }

}

}

$match and $group in an Aggregation Pipeline

The following aggregation pipeline finds the documents with a field named "state" that matches a value "CA" and then groups those documents by the group key "$city" and shows the total number of zip codes in the state of California.

db.zips.aggregate([

{

$match: {

state: "CA"

}

},

{

$group: {

\_id: "$city",

totalZips: { $count : { } }

}

}

])

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

* $sort

The $sort stage sorts all input documents and returns them to the pipeline in sorted order. We use 1 to represent ascending order, and -1 to represent descending order.

{

$sort: {

"field\_name": 1

}

}

* $limit

The $limit stage returns only a specified number of records.

{

$limit: 5

}

$sort and $limit in an Aggregation Pipeline

The following aggregation pipeline sorts the documents in descending order, so the documents with the greatest pop value appear first, and limits the output to only the first five documents after sorting.

db.zips.aggregate([

{

$sort: {

pop: -1

}

},

{

$limit: 5

}

])

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

* $project

The $project stage specifies the fields of the output documents. 1 means that the field should be included, and 0 means that the field should be supressed. The field can also be assigned a new value.

{

$project: {

state:1,

zip:1,

population:"$pop",

\_id:0

}

}

* $set

The $set stage creates new fields or changes the value of existing fields, and then outputs the documents with the new fields.

{

$set: {

place: {

$concat:["$city",",","$state"]

},

pop:10000

}

}

* $count

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

{

$count: "total\_zips"

}

***Creating a Single Field Index:***

Use createIndex() to create a new index in a collection. Within the parentheses of createIndex(), include an object that contains the field and sort order.

db.customers.createIndex({

birthdate: 1

})

***Create a Unique Single Field Index:***

Add {unique:true} as a second, optional, parameter in createIndex() to force uniqueness in the index field values. Once the unique index is created, any inserts or updates including duplicated values in the collection for the index field/s will fail.

db.customers.createIndex({

email: 1

},

{

unique:true

})

MongoDB only creates the unique index if there is no duplication in the field values for the index field/s.

***View the Indexes used in a Collection:***

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

db.customers.getIndexes()

***Check if an index is being used on a query:***

Use explain() in a collection when running a query to see the Execution plan. This plan provides the details of the execution stages (IXSCAN , COLLSCAN, FETCH, SORT, etc.).

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

db.customers.explain().find({

birthdate: {

$gt:ISODate("1995-08-01")

}

})

db.customers.explain().find({

birthdate: {

$gt:ISODate("1995-08-01")

}

}).sort({

email:1

})

***Create a Single field Multikey Index:***

Use createIndex() to create a new index in a collection. Include an object as parameter that contains the array field and sort order. In this example accounts is an array field.

db.customers.createIndex({

accounts: 1

})

***Check if an index is being used on a query:***

Use explain() in a collection when running a query to see the Execution plan. This plan provides the details of the execution stages (IXSCAN , COLLSCAN, FETCH, SORT, etc.).

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

db.customers.explain().find({

accounts: 627788

})

***Create a Compound Index:***

Use createIndex() to create a new index in a collection. Within the parentheses of createIndex(), include an object that contains two or more fields and their sort order.

db.customers.createIndex({

active:1,

birthdate:-1,

name:1

})

***Order of Fields in a Compound Index***

The order of the fields matters when creating the index and the sort order. It is recommended to list the fields in the following order: Equality, Sort, and Range.

* Equality: field/s that matches on a single field value in a query
* Sort: field/s that orders the results by in a query
* Range: field/s that the query filter in a range of valid values

The following query includes an equality match on the active field, a sort on birthday (descending) and name (ascending), and a range query on birthday too.

db.customers.find({

birthdate: {

$gte:ISODate("1977-01-01")

},

active:true

}).sort({

birthdate:-1,

name:1

})

Here's an example of an efficient index for this query:

db.customers.createIndex({

active:1,

birthdate:-1,

name:1

})

***Cover a query by the Index***

An Index covers a query when MongoDB does not need to fetch the data from memory since all the required data is already returned by the index.

In most cases, we can use projections to return only the required fields and cover the query. Make sure those fields in the projection are in the index.

By adding the projection {name:1,birthdate:1,\_id:0} in the previous query, we can limit the returned fields to only name and birthdate. These fields are part of the index and when we run the explain() command, the execution plan shows only two stages:

* IXSCAN - Index scan using the compound index
* PROJECTION\_COVERED - All the information needed is returned by the index, no need to fetch from memory

db.customers.explain().find({

birthdate: {

$gte:ISODate("1977-01-01")

},

active:true

},

{name:1,

birthdate:1,

\_id:0

}).sort({

birthdate:-1,

name:1

})