MongoDB – Query & Data Modeling

Query Selectors - Comparison

```
{field: {operator: value} }
```

- Can be applied on numeric and string field values
- **`$eq`** equal to →

```
{ field: { $eq: value } }
{ field: value }
```

- **`$gt`** greater than
- **`$gte`** greater than or equal to
- **`$lt`** less than
- **`$lte`** less than or equal to
- **`$ne`** not equal to
- **`$in`** matches any of the values specified in an array
- **`$nin`** matches none of the values specified in an array.

https://docs.mongodb.com/manual/reference/operator/query/

Examples - Comparison Query Operators

```
// return all documents that the score property is greater than 85
db.col.find({score: {$gt: 85}})
// return all documents that the name property starts with A, B and C only
db.col.find({name: {$lt: "D"}})
// return all documents that the name property which the first letter starts between B
and D(implicit AND)
db.col.find({name: {$gt: "B", $lt: "D"}})
// return all documents where the qty field value is either 5 or 15
   { _id: 1, qty : 3 }
   { _id: 2, qty : 5 }
db.col.find({ qty: { $in: [ 5, 15 ] } } ) // returns _id: 2
// return all documents where courses field value is either CS472 or CS572
   { _id: 1, courses: [ "CS472", "CS572", "CS477" ] }  (implicit OR)
db.col.find({ courses: { $in: ["CS572", "CS472"] } } )
```

- **Notes:** Because different values types for the same field is possible, MongoDB will do strongly/dynamically typed comparison operations.

Query Selectors - Element

- **`$exists`** Matches documents that have the specified field.
- **`$type`** Selects documents if a field is of the specified type.

```
If <boolean> is true, $exists matches the documents that contain the
field, including documents where the field value is null.
If <boolean> is false, the query returns only the documents that do
not contain the field.
{ field: { $exists: <boolean> } }

$type returns documents where the BSON type of the field matches the
BSON type passed to $type
{ field: { $type: <BSON type number> | <String alias> } }
```

- **BSON types**: https://docs.mongodb.com/manual/reference/operator/query/type/#op._S_type

Examples - $exists $type

```
// return all documents where the qty field exists and its value is not
   equal to 5 nor 15
db.col.find( { qty: { $exists: true, $nin: [ 5, 15 ] } } )

// return all documents where zipCode is the BSON type string
db.col.find( { "zipCode" : { $type : 2 } } );
db.col.find( { "zipCode" : { $type : "string" } } );
```

Examples - $regex

- Provides regular expression capabilities for pattern matching strings in queries.

```
   { field: { $regex: 'pattern', $options: '<options>' } }

// return all documents where the name field has the letter "a"
   anywhere in the value
db.col.find( { name: { $regex: "a" } } )

// return all documents where the name field values end with letter "e"
   upper and lower cases (i for case insensitivity)
db.col.find( { name: { $regex: "e$", $options: "i" } } )
```

Query Selectors - Logical

- Joins query clauses with a logical operation:
- $or returns all documents that match the conditions of either clause.
- $and returns all documents that match the conditions of both clauses.
- $not returns documents that do not match the query expression.
- $nor returns all documents that fail to match both clauses.

```
{ $or:  [ { <expression1> }, { <expression2> }, ... , { <expressionN> } ] }
{ $and: [ { <expression1> }, { <expression2> }, ... , { <expressionN> } ] }

// return all documents where either the quantity field value is
   less than 20 or the price field value equals 10:
db.col.find( { $or: [ { quantity: { $lt: 20 } }, { price: 10 } ] } )
```

Examples - Logical Query Operators

MongoDB provides an implicit AND operation when specifying a comma separated list of expressions.

```
// return all documents where:
//    the price field value is not equal to 1.99
//    "and" the price field exists.
db.col.find( { $and: [ { price: { $ne: 1.99 } }, { price: { $exists: true } } ] } )

// We can reconstruct the query with an implicit AND operation
// by combining the operator expressions for the price field
db.col.find( { price: { $ne: 1.99, $exists: true } } )

// return all documents where:
//    the price field value equals 0.99 or 1.99,
//    "and" the sale field value is equal to true or the qty field value
                                                         is less than 20.

db.col.find( {
        $and : [
                        { $or : [ { price : 0.99 }, { price : 1.99 } ] },// bad
                        { $or : [ { sale : true }, { qty : { $lt : 20 } } ] }
        ] } )
```

Examples - Explanation

- What is the difference between these two queries?

- This query will select all documents where:
  - 1. the price field value is less than or equal to 1.99
  - 2. or the price field does not exist

Has implicit $exist: false

- `db.col.find( { price: { $not: { $gt: 1.99 } } } )`

- This query will select all documents where:
  - 1. the price field value is less than or equal to 1.99
  - 2. the price field does exist

Has implicit $exist: true

- `db.col.find( { price: { $lte: 1.99 } } )`

Searching an Array

```
// { _id: 1, courses: [ "CS472", "CS572", "CS477" ] }

// find all documents where courses value contains "CS472" or "CS572"
db.col.find({ courses: { $in: ["CS572", "CS472"] } })
db.col.find({ $or: [ { courses: "CS572" }, { courses: "CS472" } ] })

// find all documents where courses value contains "CS472" and "CS572"
db.col.find({ courses: { $all: [ "CS572" , "CS472" ] } })
db.col.find({ $and: [ { courses: "CS572" }, { courses: "CS472" } ] })
```

- The **`$in`** operator selects the documents where the value of a field is an array that contains at any order any (at least one) of the specified elements
- The **`$all`** operator selects the documents where the value of a field is an array that contains at any order all the specified elements

Searching an Object

```
{_id: 1, email: { work: "work@mum.edu",
                  personal: "personal@gmail.com"} }

// nothing will be returned
db.col.find({ email: { work: "work@mum.edu" } })
db.col.find({ email: { personal: "personal@gmail.com" } })
db.col.find({ email: { personal: "personal@gmail.com",
                              work: "work@mum.edu"} })

// will work
db.col.find({ email: { work: "work@mum.edu",
                              personal: "personal@gmail.com"} })

// how to search for one key only?
db.col.find({ "email.work": "work@mum.edu" })
db.col.find({ "email.personal": "personal@gmail.com" })
```

Field Update Operators

- By default `update()` will: update a **single document** and **replace** everything but the `_id`

```
// Original document
{ "_id" : "MUM", "students" : 250, "courses" : ["CS572", "CS477"] }

db.col.update({_id: "MUM"}, {"students":500})

// Results
{ "_id" : "MUM", "students" : 500 }

// To target only one field use { $set: { field1: value1, ... } }
// If the field does not exist, $set will add a new field with the
   specified value
{ "_id" : "MUM", "students" : 250, "courses" : ["CS572", "CS477"] }

db.col.update({_id:"MUM"}, {$set: {"students":500, "entry":"Oct"} })

// Results
{ "_id" : "MUM", "students" : 500, "courses" : ["CS572", "CS477"] , "entry":"Oct" }
```

https://www.youtube.com/watch?v=3XUpf04_5nA&list=PLGTrAf5-F1YL49kYm1Bq0islxja4J_QMg&index=19

Field Update Operators

```
// Original document
//If set to true, creates a new document when no document matches the query criteria.
The default value is false.
{ "_id" : "MUM", "students" : 250, "courses" : ["CS572", "CS477"] }
db.col.update({_id:"MUM University"}, {"students":500}, {upsert: true})
// Results
{ "_id" : "MUM", "students" : 250, "courses" : ["CS572", "CS477"] }
{ "_id" : "MUM University", "students" : 500 }

// update all docs to have one more field (city: Fairfield)
{ "_id" : "1", "Dept" : "CompPro" }
{ "_id" : "2", "Dept" : "SustainableLiving" }
db.col.update({}, {$set: {"city":"Fairfield"}} , {multi: true})
// Results
{ "_id" : "1", "Dept" : "CompPro", "city":"Fairfield" }
{ "_id" : "2", "Dept" : "SustainableLiving", "city":"Fairfield" }
```

Field Update Operators

Update Operator	Description	Notes
`$set`	Replaces the value of a field with the specified value	If the field does not exist, `$set` will add a new field with the specified value
`$inc`	Increments a field by a specified value, it accepts positive and negative values	If the field does not exist, `$inc` creates the field and sets the field to the specified value
`$unset`	Deletes a particular field, The specified value in the $unset expression does not impact the operation.	If the field does not exist, then `$unset` does nothing
`$min, $max`	Updates the value of the field to a specified value if the specified value is more/less than the current value of the field	

Array Update Operators

```
//Original Document { "_id" : 1, "a" : [1, 2, 3, 4] }
db.testCol.update({_id:1}, { $set: { "a.2":5 } })
// output: { "_id" : 1, "a" : [1, 2, 5, 4] }
db.col.update({_id:1}, { $push : { "a": 6 } })
// output: { "_id" : 1, "a" : [1, 2, 5, 4, 6] }
db.col.update({_id:1}, { $pop: { "a": 1 } })
// output: { "_id" : 1, "a" : [1, 2, 5, 4] }
db.col.update({_id:1}, { $pop : { "a": -1 } })
// output: { "_id" : 1, "a" : [2, 5, 4] }
db.col.update({_id:1}, {$push: {a: {$each: [7,8,9]}}})
// output: { "_id" : 1, "a" : [2, 5, 4, 7, 8, 9] }
db.col.update({_id:1}, { $pull: { "a": [5] } })
// output: { "_id" : 1, "a" : [2, 4, 7, 8, 9] }
db.col.update({_id:1}, { $pullAll: { "a": [2, 4, 8] } })
// output: { "_id" : 1, "a" : [7, 9] }
db.col.update({_id:1}, { $addToSet: { "a": 5 } })
// output: { "_id" : 1, "a" : [7, 9, 5] }
db.col.update({_id:1}, { $addToSet: { "a": 5 } })
// output: { "_id" : 1, "a" : [7, 9, 5] }
```

Delete documents db.collection.remove()

```
// delete all documents - One by One
db.col.delete({})

// delete all students whose names start with N-Z
db.col.delete({"student": {$gt: "M"}})

// drop the collection - Faster than remove()
db.col.drop()
```

Notes

- When we want to delete large number of documents, it's faster to use drop() but we will need to create the collection again and create all indexes as drop() will take the indexes away (while remove() will keep them)
- Multi-docs remove are not atomic isolated transactions to other R/Ws and it will yield in between.
- Each single document is atomic, no other R/Ws will see a half removed document.

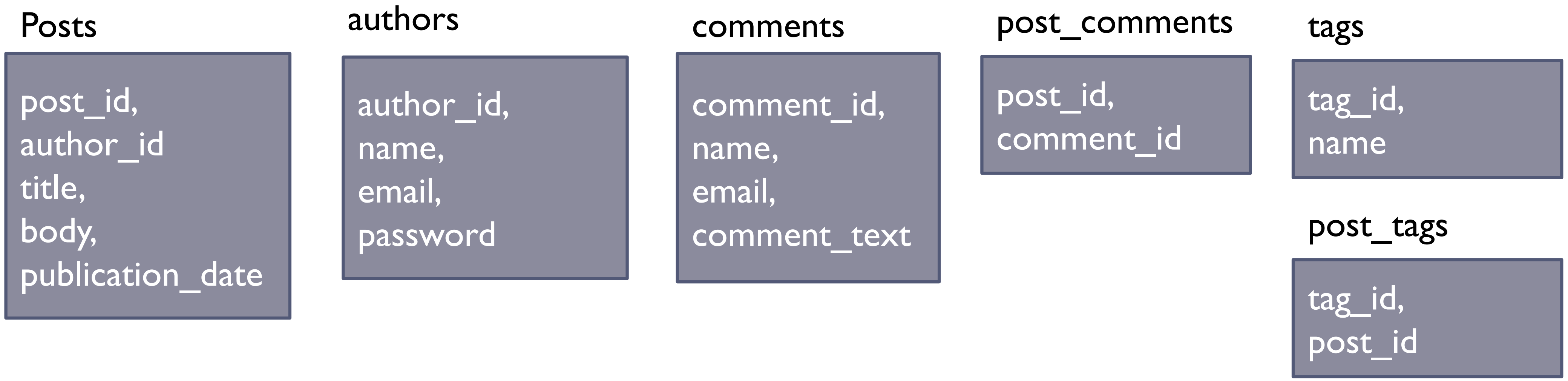
https://www.youtube.com/watch?v=49dcYAjuPws&list=PLGTrAf5-F1YL49kYm1Bq0islxja4J_QMg&index=19

Data Modeling

- Data modeling is generally the analysis of data items in a database and how related they are to other objects within that database.

Modeling Introduction

- Let’s assume that we want to model a blog with these relational tables



Posts

post_id,
author_id
title,
body,
publication_date

authors

author_id,
name,
email,
password

comments

comment_id,
name,
email,
comment_text

post_comments

post_id,
comment_id

tags

tag_id,
name

post_tags

tag_id,
post_id

- In order to display a blog post with its comments and tags, how many tables will need to be accessed?

Modeling Introduction

```
// posts collection
{ title: '',
  body: '',
  user: '', // no need for ID
  date: '',
  comments: [ {user: '', email: '', comment: ''},
              {user: '', email: '', comment: ''}]
  tags: ['', '', ''] }

// authors collection
{ user:'',
  password:‘’
}
```

Why did we embed *tags* or *comments*? Rather than have them in separate collection? Because they need to be accessed at the same time we access the *post*. We don't need to access *comments* or *tags* independently without accessing the *post*.

Given the document schema that we proposed for the blog, how many collections would we need to access to display the blog home page?

MongoDB Schema Design

- In MongoDB we use **Application-Driven Schema**, which means we design our schema based on **how we access the data**.

- Note: The only scenario we cannot embed is when data exceeds 16 MB and we need to put it in separate collection.

Design Considerations

Posts

```
{ _id,
user_id
title,
body,
shares_no
date }
```

users

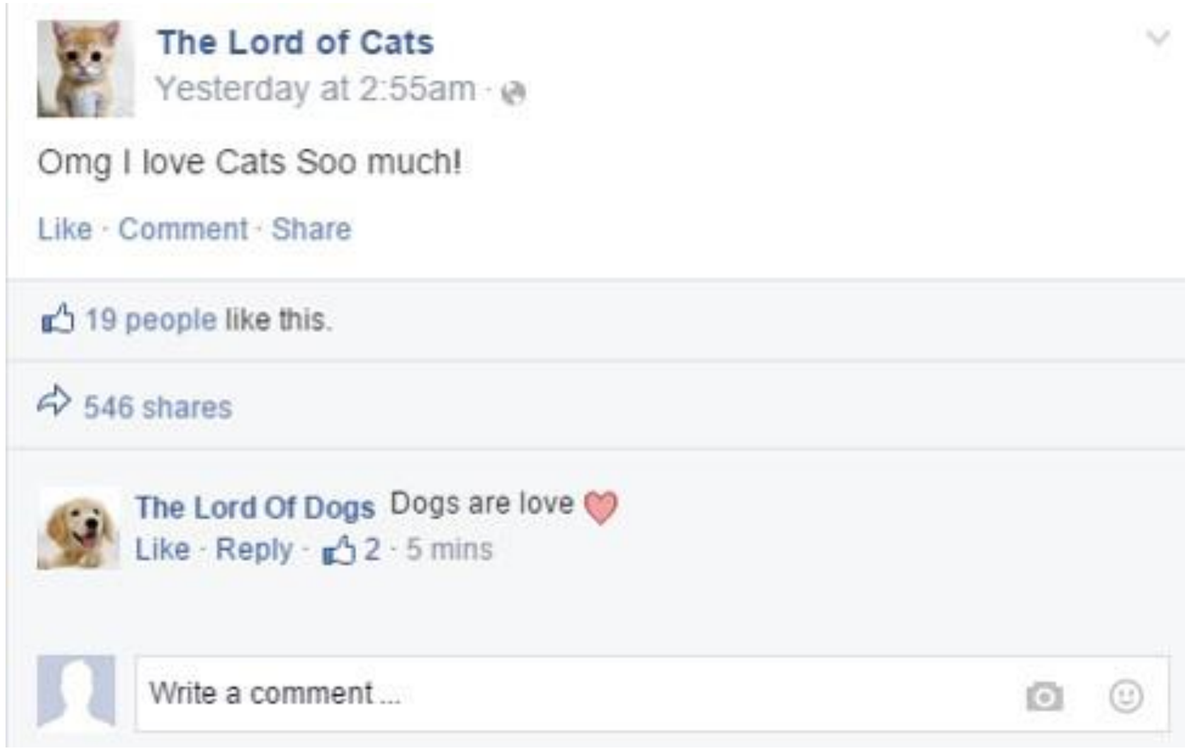
```
{ _id,
name,
email,
password }
```

comments

```
{ _id,
user_id,
post_id
comment_text
order }
```

post_likes

```
{ post_id,
user_id }
```


The Lord of Cats
Yesterday at 2:55am ·

Omg I love Cats Soo much!

Like · Comment · Share

19 people like this.

546 shares

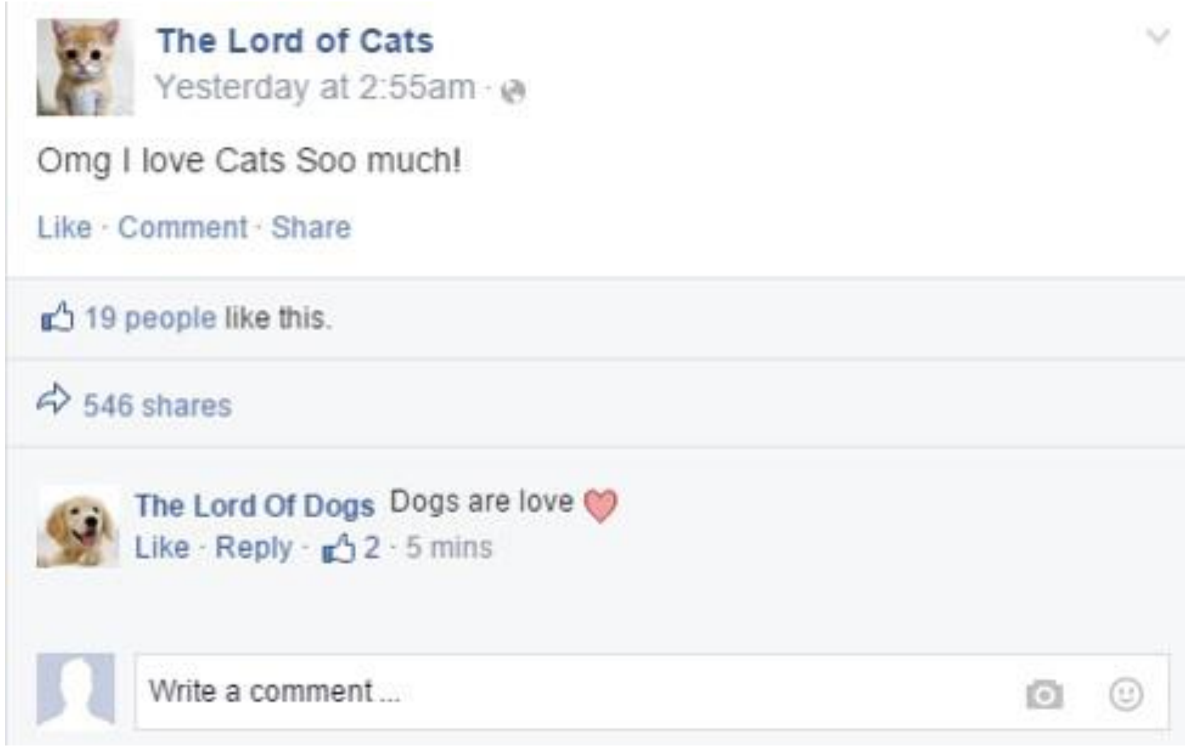
The Lord Of Dogs Dogs are love
Like · Reply · 2 · 5 mins

Write a comment ...

Remember that we don't have constrains, so this design will not work as we need to perform too much work (4 joins) in the code to retrieve our data

Design Considerations

```
{
  _id: objectId(),
  user: 'user1', // use it as ID
  title: '',
  body: '',
  shares_no: 0,
  date: ,
  comments: [
     {user:'user2', comment_text:''},
     {user:'user3', comment_text:''}]
  likes: [ 'user1', 'user2']
}
```


The Lord of Cats
Yesterday at 2:55am ·

Omg I love Cats Soo much!

Like · Comment · Share

19 people like this.

546 shares

The Lord Of Dogs Dogs are love
Like · Reply · 2 · 5 mins

Write a comment ...

This design is optimized for data access pattern so we can access the information much faster with 1 query. Especially that there is no need for data to be updated later.

Transactions vs Atomic Operations

- In the world of **relational DB**, our data is usually located in many tables, so in order to update something, we will need to access all these tables and perform the update on all of them in one **Transaction**.
- In **MongoDB**, our data is usually located in one document, and because most operations are **Atomic**, we accomplish the same thing without the need for transactions.
- Try to restructure your design to use **less collections as possible** (with one document we guarantee Atomic operations)

To embed or not to embed

- One-to-One relation
  - Linking is fine
  - Embed in either sides is fine
  - Embed in two side is fine
- One to Many relation
  - Use linking when large data and have separate collections
  - Use Embed when One-to-Few
- Many to Many relation
  - Embed is better with Few-to-Few
  - Linking is better for performance in large data

employee: resume
building: floor plan
patient: medical history

city: people
post: comments

books: authors
students: teachers

The only scenario we cannot embed is when data exceeds 16 MB and we need to put it in separate collection.

Considerations

- Take into considerations the following
  - Frequently of access and the way you want to access the data
  - Size of items (16M)
  - Atomicity of data

- Benefits of Embedding
  - Improved read performance
  - One round to the DB

Resources

- Data Model
  - https://docs.mongodb.com/manual/core/data-modeling-introduction/
  - https://www.studytonight.com/mongodb/data-modelling-in-mongodb
  - https://severalnines.com/database-blog/how-use-mongodb-data-modeling-improve-throughput-operations
- CRUD Operations
  - https://docs.mongodb.com/manual/crud/
- Other Resources
  - Multiple collections vs Embedded documents

Homework

- Continue working on online shopping project, add features below:
  1. Be able to add a new User using a form. For now we just put the user under request object for now using code below:

```
app.use((req, res, next) => {
    User.findById('5e49bdefce95301ad0fa9870')
        .then(user => {
            req.user = user;
            next();
        })
        .catch(err => console.log(err));
});
```

  2. Add/remove products to shopping cart
  3. Place an order