Marker Tools

CS572 Modern Web Applications Programming
Maharishi University of Management
Department of Computer Science
Assistant Professor Asaad Saad

OUTLINE

Search...

What is MongoDB?

- MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling.
- Non relational DB, stores BSON documents.
- Schemaless: Two documents don't have the same schema.

PREV NEXT

NoSQL Revolution

NoSQL (Not Only SQL) databases were created for "**Big Data**" and **Real-Time Web Applications**, it provides new data architectures that can handle the ever-growing velocity and volume of data.

Name	Year	Type	Developer
MongoDB	**2008**	**Document**	**10Gen**
CouchDB	2005	Document	Apache
Cassandra	2008	Column Store	Apache
CouchBase	2011	Document	Couchbase
Riak	2009	Key-Value	Basho Technologies
SimpleDB	2007	Document	Amazon
BigTable	2015	Column Store	Google
Azure Cosmos DB	2017	Multi-Model	Microsoft

PREV NEXT

NOSQL Database Types

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

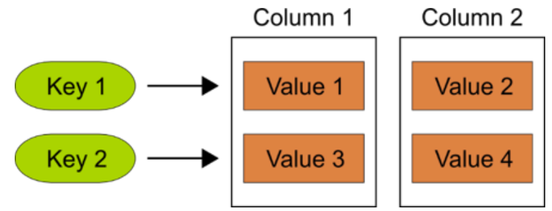
Key-Value Stores

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
             phone: "123-456-7890",
             email: "xyz@example.com"
           },
  access: {
            level: 5,
            group: "dev"
          }
}
```

Embedded sub-document

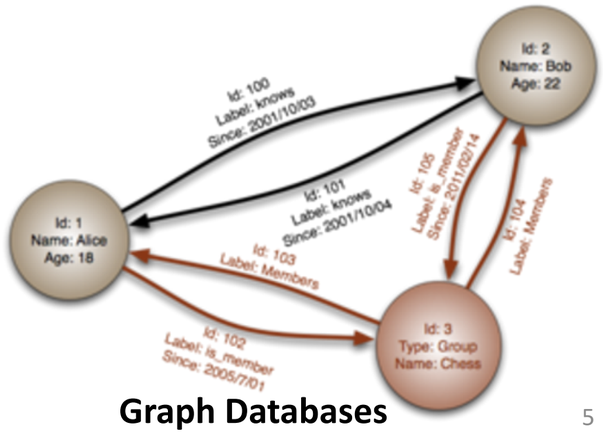
Embedded sub-document

Document Databases



Column Family Stores

Key-Value pairs in hash table, always unique key. Logical group of keys are called: buckets
Document Databases uses Key-Value pairs in a document (JSON, BSON)
Column Stores data is stored in cells that are grouped in columns of data rather than rows (unlimited columns)
Graph Databases, uses flexible graphical representation (edges and nodes) instead of k/v pairs. Index free. Very fast for associative data sets and maps.



Graph Databases

PREV NEXT

OUTLINE

Search...

Choosing NoSQL For Performance

When the number of concurrent clients using your application/db is reasonably small (let's say 500 users) RDBMS can work great. But what if that number grows? Especially if you are writing a Web app. At 50,000 users, can you still run on a single instance of SQL Server or MySQL? How powerful does your hardware have to be to handle that? What about at 500,000 or 5,000,000 users, still good?

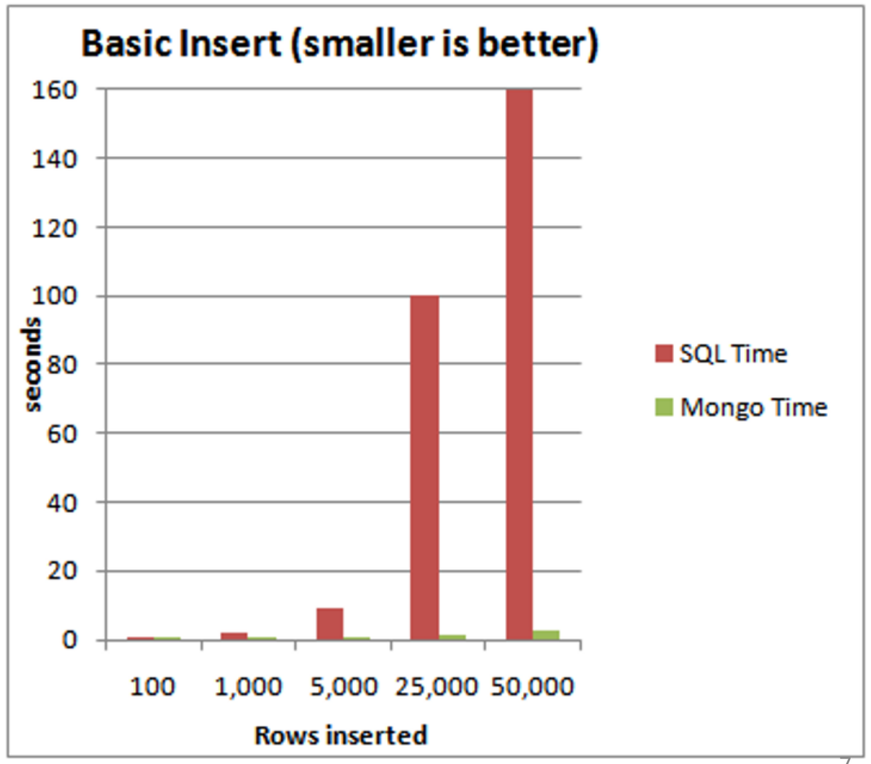
Wait a minute now! There are plenty of systems with tons of users built upon relational databases!

Yes but how much expensive hardware and software (licenses) do these require? A basic SQL Server cluster might cost you around $100,000 just to get it up and running on decent hardware.

PREV NEXT

How Much Faster?

Here's a simple comparison of running a bunch of concurrent inserts into SQL Server 2008 and MongoDB on the same computer.
These inserts were performed by inserting 50,000 independent objects using NoRM for MongoDB and LINQ to SQL for SQL Server 2008.



Number of Parallel Clients 5 Time in seconds

Basic Insert	Total Rows	Rows / client	SQL Time	Mongo Time	Sql Ops/sec	Mongo Ops/sec
several columns	100	20	0.19	0.011	526	9,091
600 bytes per row	1,000	200	1.8	0.02	556	50,000
	5,000	1,000	9	0.25	556	20,000
	25,000	5,000	100	1.5	250	16,667
	50,000	10,000	270	2.5	185	20,000

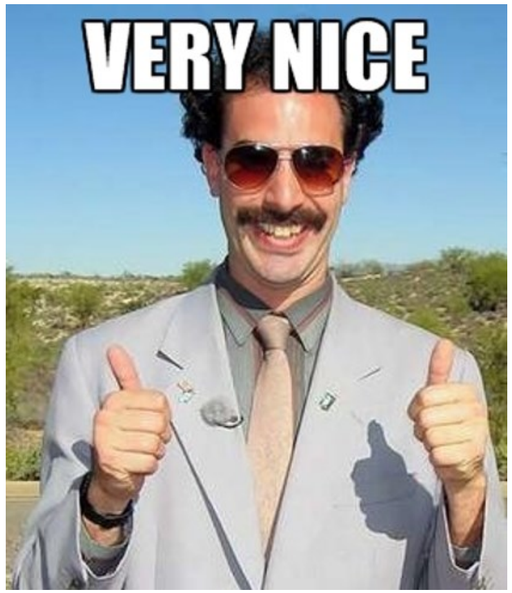
PREV NEXT

The Results

Looks like under heavy load, MongoDB is about 100 times faster.

```
MongoDB Client
Warming up ...
Building insert data...
Waiting on mutex
Running!
Finished with 10000 MongoDB inserts in 2.032 sec.
Done
```

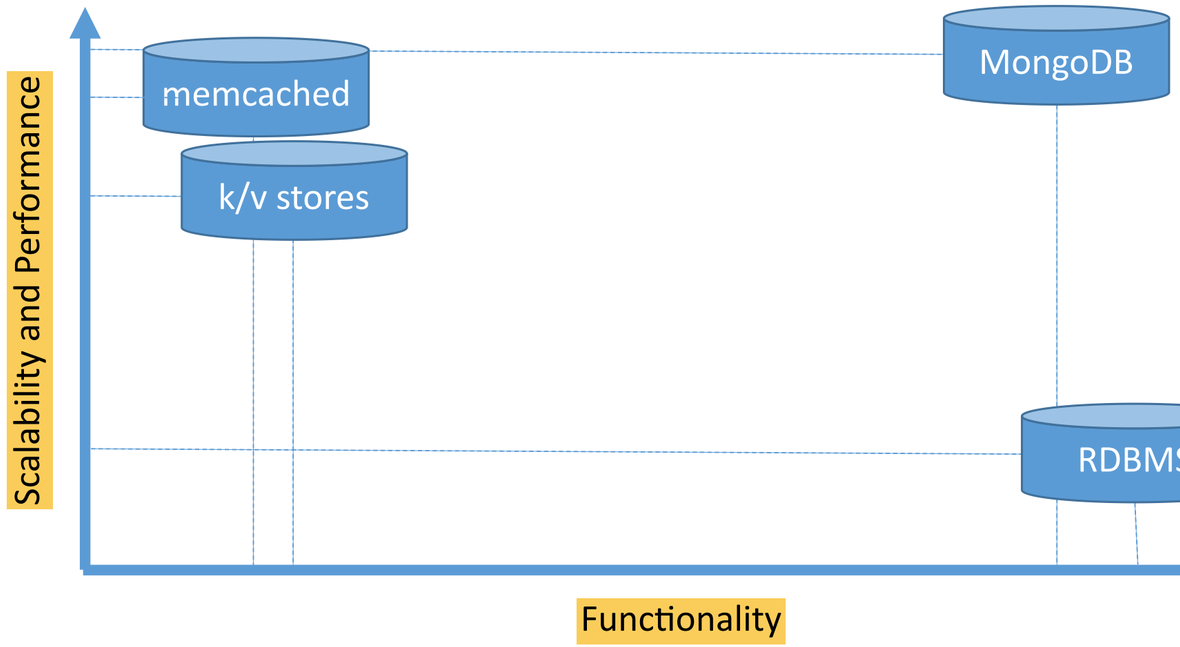
```
SQL Server Client
Warming up ...
Building insert data...
Waiting on mutex
Running!
Finished with 10000 SQL inserts in 204.215 sec.
Done
```



That’s right. It’s 2 seconds verses 3.5 minutes!

PREV NEXT

Databases comparison



9

Marker Tools

Advantages of NoSQL

- **Transparent Scaling**
- **Significantly cheaper** to scale with commodity hardware
- **Less Management** than RDBMS
- **Unlimited Space** with cloud solutions
- **Performance** as embedded data models reduces I/O activity on DB
- **Depth of Functionality** Aggregation framework, Text Search, Geospatial Queries
- **Support for Multiple Storage Engines** such as WiredTiger Storage Engine (default) and MMAPv1 Storage Engine

High Availability and Scalability

- **Replica Sets**

  replica set (group of MongoDB servers), provides automatic failover and data redundancy.

- **Shards**

  Sharding distributes data across a cluster of machines.

PREV NEXT

Document Data Model

- A record in MongoDB is a **Document**
- Structure of key/value pairs
- Values may contain other documents (embedded documents), arrays and arrays of documents (rich document).

```
{
    _id: 1,
    name: "Asaad",
    email: "asaad@mum.edu",
    courses: ["CS472", "CS572"]
}
```

Document Implications

- Documents (objects) correspond to native data types in many programming languages.
- Embedded documents & arrays reduce the need for expensive joins.
- Dynamic schema supports fluent polymorphism: A **polymorphic type** is one whose operations can also be applied to values of some other type(s)
- Atomic transaction on the document level.

Atomic transaction: is either all occur or nothing occur.

PREV NEXT

Marker Tools

Data Types

The value of a field can be any of the **BSON data types**, including other documents, arrays, and arrays of documents.

```
var mydoc = {
        _id: ObjectId("5099803df3f4948bd2f98391"),
        name: { first: "Asaad", last: "Saad" },
        birth: new Date('Oct 31, 1979'),
        courses: [ "CS472", "CS572" ],
        students : NumberLong(1250000)
    }
```

15

14 / 57 00:00 / 00:00

PREV NEXT

OUTLINE

Search...

Marker Tools

BSON

- BSON, short for **Binary JSON**, is a binary-encoded serialization of JSON-like documents.
- Both JSON and BSON support **Rich Documents** *(embedding documents and arrays within other documents and arrays)*.
- BSON also contains extensions that allow representation of **data types** that are not part of the JSON spec. *(For example, BSON has a **BinData** **ObjectId**, **64 bits Integers** and **Date** type…etc)*
- **Encoding** data to BSON and **decoding** from BSON can be performed very quickly in most languages. *For example, integers are stored as 32 (or 64) bit integers and they don't need to be parsed to and from text.*

PREV NEXT

Schema and Agile Structure

By default, a collection does not require its documents to have the same schema, the documents in a single collection do not need to have the same set of fields and the data type for a field can differ across documents within a collection.

To add more information to a table in relational DBMS, we need to add more columns to the table (which might leave many null values) or add new table.

MongoDB is agile, there will be no need to have same structure in documents. Every document can have its own structure.

(Starting of MongoDB 3.2, you can enforce document validation rules for a collection during update and insert operations)

Building an application

- **Client** *(JavaScript)* ask for data or want to store data
- **Server** *(JavaScript)* - Driver to communicate with MongoDB
- **MongoDB Local Server** *(JavaScript)*
  - MongoShell - Browse to the bin folder (or add bin folder to PATH)
    - `mongo.exe` (Shell)
- **MongoDB As a Service DaaS** *(JavaScript)*
  - Mongo Atlas

Note: An easy way to manage your data visually is to use a GUI for MongoDB like: MongoDB Compass

Collections

MongoDB stores documents in **collections**. *(Collections are similar to tables in relational databases)*

If a database/collection does not exist, MongoDB creates the db/collection when you first store data for that collection

```
use myNewDB

db.myNewCollection.insert( { x: 1 } )
```

The insert() operation creates both the database myNewDB and the collection myNewCollection if they do not already exist.

Exploring the shell - Demo

```
show dbs
use testDB // switch or create
show collections
db.testCol.insert({"name": "Saad"})// db var refers to the current database
db.testCol.find() // notice _id
// passing a parameter to find a document that has a property "name" and value
"Asaad"
db.testCol.find({"name":"Asaad"})
// save() = upsert if _id provided
db.testCol.save({"name":"Mike"})
// insert 10 documents – Shell is C++ app that uses V8
for (var i=0; i<10; i++){ db.testCol.insert({"x": i}) }
```


PREV NEXT

Exploring the shell - Demo

```
db.testCol.save({a:1, b:2})
db.testCol.save({a:3, b:4, fruit: ["apple", "orange"] })
db.testCol.save({name: "Asaad", address: {city: "Fairfield",
                                          zip: 52557,
                                          street: "1000 N 4th street"} })
// show documents in a nice way, it will only work when you have nested or larger
documents:
db.testCol.find().pretty()
```


PREV NEXT

General Rules

- Field names are strings.
- The field name `_id` is reserved for use as a primary key. It is immutable and always the first field in the document. It may contain values of any BSON data type, other than an array.
- The field names cannot start with the dollar sign ($) character and cannot contain the dot (.) character or `null`. Field names cannot be duplicated.
- The maximum BSON document size is 16 megabytes. *(To store documents larger than the maximum size, MongoDB provides the GridFS API)*

PREV NEXT

Marker Tools

Connect to MongoDB

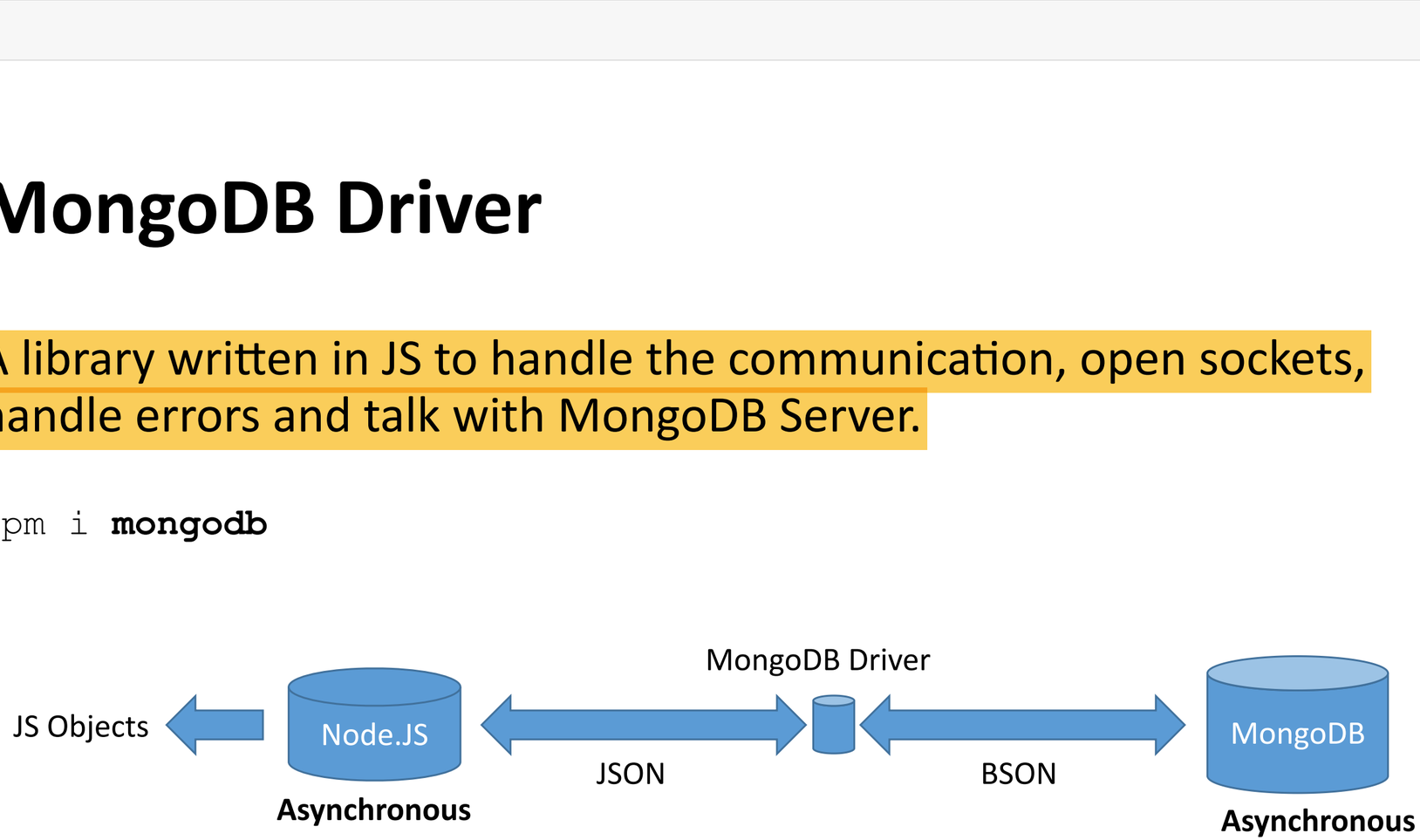
```
const MongoClient = require('mongodb').MongoClient;
const client = new MongoClient('mongodb://localhost:27017');

client.connect(function(err) {

   const db = client.db('myDB');
   const collection = db.collection('myCollection');

   collection.findOne({}, function (err, doc) {
       console.dir(doc);
       client.close();
   });

   console.dir("Done");
});
```

What's the output of this code?

MongoDB Driver

A library written in JS to handle the communication, open sockets, handle errors and talk with MongoDB Server.

```
npm i mongodb
```



Marker Tools

`db.collection.findOne({query}, {projection: {} })`

Returns **one document** that satisfies the specified **query** criteria. If multiple documents satisfy the query, this method returns the first document according to the **natural order** which reflects the order of documents on the disk. If no document satisfies the query, the method returns null.

- The **`query`** is equivalent to **`where`** in SQL, it takes the form of JSON object.
- The **`project`** method accepts JSON of the following form:

```
{ field1: <boolean>, field2: <boolean> ... }
```

Notes:

- The **`findOne()`** method always includes the **`_id`** field even if the field is not explicitly specified in the projection parameter, unless you explicitly exclude it.
- The projection argument cannot mix include and exclude specifications, with the exception of excluding the **`_id`** field.

25

PREV NEXT

Examples- findOne()

```
// return one document with all fields
db.collection.findOne({})

// return one document with two fields "_id" and "name"
db.collection.findOne({}, { projection: {name: 1} )

// return one document that has "name" property with value "Asaad",
   this document will have all fields but "_id" and "birth"
db.collection.findOne({name: 'Asaad'},{ projection: { _id: 0, birth: 0 } })
```

`db.collection.find({query}).project({projection})`

Selects documents in a collection and returns a **`cursor`** to the selected documents.

`cursor`: A pointer to the result set of a query. Clients can iterate through a cursor to retrieve results. By default, cursors timeout after 10 minutes of inactivity.

Notes:

- Executing **`find()`** in the mongo shell automatically iterates the cursor to display the first 20 documents. Type **`it`** to continue iteration.

Examples- find()

```
// returns all documents in a collection
db.collection.find({})


// It works also for Array type fields:
   return all documents where the tags field value is CS572
   { _id: 1, tags: [ "CS472", "CS572", "CS435" ] }
db.collection.find({ tags: "CS572" })
```


PREV NEXT

count()

We can use **count()** method exactly like **find()** to get the count of all the documents that match a certain criteria.

```
// returns number of all documents in the collection
db.collection.count()

// returns number of students who received A
db.collection.count({ "grade": "A" })
```

Example- Using `findOne()`

```
const query = { 'grade' : 100 };

db.collection.findOne(query, function(err, doc) {
        console.dir(doc);
});
```

`console.dir vs console.log`
`console.log()` only prints out a string, whereas `console.dir()` prints out a navigable object tree

Example- Using find()

```
const query = { 'grade' : 100 };

db.collection.find(query).toArray(function(err, docsArr) {
        console.dir(docsArr);
});
```

`toArray()` will buffer all data in memory as array before processing the callback function.

PREV NEXT

OUTLINE

Example- Using `find()` with cursors

```
const query = { 'grade' : 100 };

const cursor = db.collection.find(query);

cursor.forEach(function(err, doc) {
        console.dir(doc.student);
});
```

Behind the scene, MongoDB sends the data in batches (stream) is doesn’t send everything at once. The cursor will send a new request every time it finishes processing the batch.

Marker Tools

Example- Using `find()` with projection

```
var query = { 'grade' : 100 };
var projection = { 'student' : 1, '_id' : 0 };

db.collection('grades').find(query).project(projection).toArray(function(err, docsArr){
        if(err) throw err;
        docsArr.forEach(function (doc) {
                console.dir(doc.student);
        });
        db.close();
});
```

Note: Projection is a good practice to save bandwidth and retrieve only the data we need.

sort() limit() skip()

Similar to SQL language, MongoDB provides certain methods on the collection object, they work as instructions sent to DB to affect the retrieval of data, all these methods will return a cursor back (chain):

SQL	MongoDB Method
Order by	sort()
Limit	limit()
Skip	skip()

Note: These will set instructions to DB server to process the information before its being sent to client.
No processing will ever happen at the client side.

PREV NEXT

Example- Skip, Limit and Sort

```
const cursor = db.collection.find({});
cursor.skip(10);
cursor.limit(5);
cursor.sort('grade', 1); //cursor.sort([['grade', 1], ['student', -1]]);

cursor.forEach(function(err, doc) {
        console.dir(doc);
});
```

Note: These will be implemented in the DB in a very specific order: **1. sort, 2. skip, 3. limit** no matter how we put them in the code

PREV NEXT

OUTLINE

Search...

Marker Tools

Example- Skip, Limit and Sort

```
const MongoClient = require('mongodb').MongoClient;
const client = new MongoClient('mongodb://localhost:27017');

client.connect(function(err) {
   const db = client.db('myDB');
   const collection = db.collection('myCollection');

   const options = { 'skip' : 10, 'limit' : 5, 'sort' : ['grade', 1] };
   const cursor = collection.find({}, options);

   cursor.forEach(function(err, doc) {
       console.dir(doc);
   });
});
```

Example- Using insert()

```
var doc = { 'student' : 'Asaad', 'grade' : 100 };

db.collection.insert(doc, function(err, docInserted) {
        console.dir(`Success: ${docInserted}`);
});
```

```
var docs = [ { 'student' : 'Kevin', 'grade' : 90 },
             { 'student' : 'Susie', 'grade' : 95 } ];

db.collection.insert(docs, function(err, docsInserted) {
        console.dir(`Success: ${docInserted}`);
});
```


PREV NEXT

Delete documents db.collection.remove()

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

// delete all students whose names start with N-Z
db.col.remove({"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.

PREV NEXT

Example- Using remove()

```
var query = { 'assignment' : 'hw3' };

// remove all documents that have 'hw3' value in 'assignment'
db.collection.remove(query, function(err, removed) {
        console.dir( removed + " documents removed!");
});
```


PREV NEXT

`{field: {operator: value} }`

Comparison Query Operators

Can be applied on **numeric** and **string** field values

- **$eq** equal to
- **$gt** greater than
- **$gte** greater than or equal to
- **$lt** less than
- **$lte** less than or equal to
- **$ne** not equal to
- **$in** matches any of the values specified in an array (implicit OR)
- **$nin** matches none of the values specified in an array.
- **Comma** between operators works as (implicit AND)

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


PREV NEXT

Examples- Comparison Query Operators

```
// return all documents that the score property is greater than 85
db.col.find({score: {$gt: 85}})

// 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", "CS435" ] }  (implicit OR)
db.col.find( { courses: { $in: ["CS572", "CS472"] } } )
```

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

PREV NEXT

Element Query Operators

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

$type returns documents where the BSON type of the field matches the BSON type
passed to $type, return all documents where zipCode is the BSON type string
db.col.find( { "zipCode" : { $type : 2 } } );
db.col.find( { "zipCode" : { $type : "string" } } );
```

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

Examples- $regex

Provides regular expression capabilities for pattern matching strings in queries. *(consume a lot of CPU time are extremely slow)*

```
{ 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" } } )
```

Logical Query Operators

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 } ] } )
```


PREV NEXT

Examples- Logical Query Operators

```
// 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 } } ] }
       ] } )
```

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

45

Example – Using Operators in Node

```
var query = { 'student':'Saad', 'grade' : { '$gt':92, '$lt':98 } };

db.collection.find(query).forEach(function(err, doc) {
        console.dir(doc);
});
```

Notes:

- Add quotation for all MongoDB operators.
- Create one DB connection globally to be shared by your application.
- Save all application configurations in a JSON file or `.env` file using `dotenv` module.

Examples – Be careful!

What is the output of this query?

```
db.col.find( { price : { $gt : 50 }, price : { $lt : 60 } } );
```

What is the difference between these two queries?

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

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


PREV NEXT

Examples- Explanation

This query will select all documents where:

1. the price field value is not greater than 1.99
2. or the price field does not exist

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

Has implicit $exist: false

This query will select all documents where:

1. the price field value is less than or equal to 1.99
2. the price field must exist

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

Has implicit $exist: true

Modeling

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?

PREV NEXT

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.

PREV NEXT

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 }
```

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']
}
```

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.

PREV NEXT

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 documents as possible** (with one document we guarantee Atomic operations)

PREV NEXT

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

employee: resume
building: floor plan
patient: medical history

One to Many relation

- Use linking when large data and have separate collections
- Use Embed when **One-to-Few**

city: people
post: comments

Many to Many relation

- Embed is better with **Few-to-Few**
- Linking is better for performance in large data

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.

PREV NEXT

Marker Tools

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

Import/Export JSON in MongoDB

mongoimport `-d dbName` `-c colName` `file.json`

Database Name — Collection Name — JSON File

```
{ "student" : "Saad", "assignment" : "hw1", "grade" : 90 }
{ "student" : "Saad", "assignment" : "hw2", "grade" : 80 }
{ "student" : "Saad", "assignment" : "hw3", "grade" : 85 }
 ...
```

mongoexport `-d dbName` `-c colName` `-o file.json`

Database Name — Collection Name — JSON Output File

Note: `mongoimport` and `mongoexport` might not work very well with rich documents.
Instead you may use `mongodump –db dbName` and `mongorestore` for full support of rich documents.

PREV NEXT

MongoDB in the Cloud

Atlas is a fully managed cloud database service featuring automated provisioning and scaling of MongoDB databases.

Atlas's Database-as-a-Service platform powers databases across AWS, Azure, and Google Cloud Platform.

Other options:

- Use Mongo Docker container

PREV NEXT