Q1: What's a good way to get a list of all unique tags for a collection of documents millions of items large when we doesn't have access to mongodb's new "distinct" command ?

The normal way of doing tagging seems to be indexing multikeys. Even if your MongoDB driver doesn't implement distinct, we can implement.

You do a findOne on the "$cmd" collection of whatever database you're using. Pass it the collection name and the key you want to run distinct on.

Q2: How to get the names of all the keys in a MongoDB collection?

db.things.insert({ type : ['dog', 'cat'] });
db.things.insert({ egg : ['cat'] });
db.things.insert({ type : [] });
db.things.insert({ hello: [] });
How to get the unique keys: type, egg, hello We could do this with MapReduce:
mr = db.runCommand({
"mapreduce" : "my_collection",
"map" : function() {
for (var key in this) { emit(key, null); }
b
"reduce" : function(key, stuff) { return null; },
"out": "my_collection" + "_keys" 1)
Then run distinct on the resulting collection so as to find all the keys:
db[mr.result] distinct("_id")
["foo", "bar", "baz", "_id", ...]

Q3: How to use map/reduce to handle more than 10000 unique keys for grouping in MongoDB?

Use Version 2.2. The db.collection.group() method's returned array can contain at most 20,000 elements; i.e. at most 20,000 unique groupings. For group by operations that results in more than 20,000 unique groupings, use mapReduce. Previous versions had a limit of 10,000 elements.

Q4: Can anyone share some insight on the index/RAM relationship and what happens when both an individual index and all of my indexes exceed the size of available RAM?

MongoDB keeps what it can of the indexes in RAM. They'll be swaped out on an LRU basis. You'll often see documentation that suggests you should keep your "working set" in memory: if the portions of index you're actually accessing fit in memory, you'll be fine.
It is the working set size plus MongoDB's indexes which should ideally reside in RAM at all times i.e. the amount of available RAM should ideally be at least the working set size plus the size of indexes plus what the rest of the OS (Operating System) and other software running on the same machine needs. If the available RAM is less than that, LRUing is what happens and we might therefore get significant slowdown.
One thing to keep in mind is that in an index btree buckets are cached, not individual index keys i.e. if we had a uniform distribution of keys in an index including for historical data, we might need more of the index in RAM compared to when we have a compound index on time plus something else. With the latter, keys in the same btree bucket are usually from the same time era, so this caveat does not happen. Also, we should keep in mind that our field names in BSON are stored in the records (but not the index) so if we are under memory pressure they should be kept short

Q5: MongoDB – simulate join or subquery

I'm trying to figure out the best way to structure my data in Mongo to simulate what would be a simple join or subquery in SQL.
Say I have the classic Users and Posts example, with Users in one collection and Posts in another. I want to find all posts by users who's city is "london".
I've simplified things in this question, in my real world scenario storing Posts as an array in the User document won't work as I have 1,000's of "posts" per user constantly inserting.

Q6: Can Mongos $in operator help here? Can $in handle an array of 10,000,000 entries?

Honestly, if you can't fit "Posts" into "Users", then you have two options.
1. Denormalize some User data inside of posts. Then you can search through just the one collection.
2. Do two queries. (one to find users the other find posts)
Based on your question, you're trying to do #2.
Theoretically, you could build a list of User IDs (or refs) and then find all Posts belonging to a User$in that array. But obviously that approach is limited.

Q7: Can $in handle an array of 10,000,000 entries?

Look, if you're planning to "query" your posts for all users in a set of 10,000,000 Users you are well past the stage of "query". You say yourself that each User has 1,000s of posts so you're talking about a query for "Users with Posts who live in London" returning 100Ms of records. 100M records isn't a query, that's a dataset!
If you're worried about breaking the $in command, then I highly suggest that you use map/reduce. The Mongo Map/Reduce will create a new collection for you. You can then trim down or summarize this dataset as you see fit.
$in can handle 100,000 entries. I've never tried 10,000,000 entries but the query (a query is also a document) has to be smaller than 4mb (like every document) so 10,0000,0000 entries isn't possible. Why don't you include the user and its town in the Posts collection? You can index this town because you can index properties of embedded entities. You no longer have to simulate a join because you can query the Posts on the towns of its embedded users.
This means that you have to update the Posts when the town of a user changes but that doesn't happen very often. This update will be fast if you index the UserId in the Posts collection.

Q8: Mongo complex sorting?

Q9: Can I sort with a user-defined function; e.g., supposing a and b are integers, by the difference between a and b (a-b)?

I don't think this is possible directly; the sort documentation certainly doesn't mention any way to provide a custom compare function.
You're probably best off doing the sort in the client, but if you're really determined to do it on the server you might be able to use db.eval() to arrange to run the sort on the server (if your client supports it).

Note that it's also possible to sort via an aggregation pipeline and by the $orderby operator (i.e. in addition to .sort()) however neither of these ways lets you provide a custom sort function either. Why don't create the ield with this operation and sort on it ?

Q10: How to set a primary key in MongoDB?

The _id field is reserved for primary key in mongodb, and that should be an unique value. If you don't set anything to _id it will automatically fill it with "MongoDB Id Object". But you can put any unique info into that field.

Q11: Delete everything in a MongoDB database

use [database];
db.dropDatabase();
Ruby code should be pretty similar.
Also, from the command line:
mongo [database] –eval "db.dropDatabase();"Use
[databaseName]
db.Drop+databaseName();
drop collection
use databaseName
db.collectionName.drop();

Q12: Compare MongoDB and Cassandra

MongoDB:
data mode are document
database scalability is read only
query of the data is multi-index
Cassandra:
data mode are big table like
database scalability is write only
query of the data is using key or scana

Q13: what make magodb best?

it is consider has nosql database because documented oriented (do).

Q14: when an index does not fit into RAM?

index is very huge then index will not fit to RAM.

Q15: If you remove an object attribute, is it deleted from the database?

yes when the object attribute is delete then the object will be drop

Q16: Define MongoDB.

it th DO database which use in high availability and it is dynamic schema loction

Q17: What are the key features of mongodb?

high perform
high availability

automatic scaling,

Q18: What is sharing in MongoDB?

the process of storing the record in multiple Machine is know has sharing.

Q19: How can you see the connection used by Mongo

Mongos use db_adminCommand ("connPoolStats");

Q20: What is replication and why we need?

It is the process of synchronizing data across multiple server. It provides redundancy and increases data availability with multiple copies of data on different database server.WHY: To keep data safe, High availability of data, disaster recovery, No downtime for maintenance.

MongoDBS
MongoDoc
mongoQueries

NoSQL databases feel a bit more like JavaScript than SQL databases.How Mongo works

Most applications have one data- base, like Mongo. These databases are hosted by servers. A Mongo server can have many databases on it, but there is generally one database per application. If you're developing only one application on your computer, you'll likely have only one Mongo database.
To access these databases, you'll run a Mongo server. Clients will talk to these serv- ers, viewing and manipulating the database. There are client libraries for most pro- gramming languages; these libraries are called drivers (mongoose).
Every database will have one or more collections. I like to think of collections as fancy arrays.
Every collection will have any number of documents. Documents aren't technically stored as JSON, but you can think of them that way; they're basically objects with vari- ous properties.
Documents look a lot like JSON, but they're technically Binary JSON, or BSON. You almost never deal with BSON directly; rather, you'll translate to and from JavaScript objects. The specifics of BSON encoding and decoding are a little different from JSON. BSON also supports a few types that JSON does not, like dates, timestamps, and unde- fined values.

Talking to Mongo from Node with Mongoose
Express doesn't dictate how you store your data. So we approach mongoDBA simple Hello BSON document might look like this internally:
\x16\x00\x00\x00\x02hello
Models can serve as simple objects that store database values, but they often have things like data validation, extra methods, and more. As you'll see, Mongoose has a lot of those features.

Username, Password, ime joined, Display name, Biography To specify this in Mongoose, you must define a schema, which contains information about properties, methods, and more. It's pretty easy to translate these English terms into Mongoose code.
Mongo is a database that lets you store arbitrary documents.
■ Mongoose is an official Mongo library for Node. It works well with Express.
■ To securely create user accounts, you need to make sure you never store passwords directly. You'll use the bcrypt module to help us do this. You'll use Passport to authenticate users, making sure they're logged in before
■ they can perform certain operations.

Terminologies:

A MongoDB Database can be called as the container for all the collections.
Collection is a bunch of MongoDB documents. It is similar to tables in RDBMS.
Document is made of fields. It is similar to a tuple in RDBMS, but it has dynamic schema here. Documents of the same collection need not have the same set of fields
the most popular NoSQL database, is an open-source document-oriented database. The term 'NoSQL' means 'non-relational'. It means that MongoDB isn't based on the table-like relational database structure but provides an altogether different mechanism for storage and retrieval of data. This format of storage is called BSON

A simple MongoDB document Structure:

{
title: 'Geeksforgeeks',
by: 'Harshit Gupta',
url: 'https://www.geeksforgeeks.org',
type: 'NoSQL'
}

Collection

[
{
"Name" : "Aman",
Age : 24,
Gender : "Male"

},
{
"Name" : "Suraj",
Age : 32,
Gender : "Male"
},
{
"Name" : "Joyita",
"Age" : 21,
"Gender" : "Female"
},
{
"Name" : "Mahfuj",
"Age" : 24,
"Gender" : "Male"
},
]

Create a New Database :Features of MongoDB

You can create a new Database in MongoDB by using “use Database_Name” command. The command creates a new database if it doesn’t exist, otherwise, it will return the existing database.you can run this command in mongo shell to create a new database. Your newly created database is not present in the list of Database. To display database, you need to insert at least one document into it.
Syntax :use Database_Name
Show list of Databases : You can check currently selected database, using the command **“show dbs“.**
Check current Database : You can check list of databases, using the command **“show dbs”**
Switch to other Database : You can switch to other database using the command **“use Database_Name“.**If Database does not exists then it will create a new Database.

Document Oriented: MongoDB stores the main subject in the minimal number of documents and not by breaking it up into multiple relational structures like RDBMS. For example, it stores all the information of a computer in a single document called Computer and not in distinct relational structures like CPU, RAM, Hard disk, etc.
Indexing: Without indexing, a database would have to scan every document of a collection to select those that match the query which would be inefficient. So, for efficient searching Indexing is a must and MongoDB uses it to process huge volumes of data in very less time.
Scalability: MongoDB scales horizontally using sharding (partitioning data across various servers). Data is partitioned into data chunks using the shard key, and these data chunks are evenly distributed across shards that resides across many physical servers. Also, new machines can be added to a running database.
Replication and High Availability: MongoDB increases the data availability with multiple copies of data on different servers. By providing redundancy, it protects the database from hardware failures. If one server goes down, the data can be retrieved easily from other active servers which also had the data stored on them.
Aggregation: Aggregation operations process data records and return the computed results. It is similar to the GROUPBY clause in SQL. A few aggregation expressions are sum, avg, min, max, etc

MongoDB is preferred over RDBMS in the following scenarios

Big Data: If you have huge amount of data to be stored in tables, think of MongoDB before RDBMS databases. MongoDB has built in solution for partitioning and sharding your database.
Unstable Schema: Adding a new column in RDBMS is hard whereas MongoDB is schema-less. Adding a new field, does not effect old documents and will be very easy.
Distributed data Since multiple copies of data are stored across different servers, recovery of data is instant and safe even if there is a hardware failure.

Introduction

NoSQL databases are databases that store and retrieve the data that is present in a non-tabular format. Depending on the format of the data stored, NoSQL databases are split into 4 major types:
Key-Value
Graph database
Document-oriented
Column family

Key features of MongoDB:

Full index support for high performance
Horizontally scalable and fault tolerant (distributed data storage/sharding)
Rich document based queries for easy readability
Replication and failover for high availability
Map/Reduce for aggregation
Supports Master-Slave replication

No joins nor transactions
No rigid schema, which makes it dynamic
Data represented in JSON / BSON

Connecting to MongoDB using Node.js

```
var client = require('mongodb').MongoClient;
var url = 'mongodb://localhost:27017/admin';
client.connect(url,{ useNewUrlParser: true }, function(err,db)
{
console.log("Connected");
db.close();
});
```

Explanation: To connect to a database/create a database, a MongoClient object needs to be created.
The URL of the MongoDB, followed by the database name should be specified
Using the connect function of the MongoClient object, a connection is established between the server and the MongoDB.**Querying data from MongoDB**

```
var client = require('mongodb').MongoClient;
var url = 'mongodb://localhost:27017/';
client.connect(url,{ useNewUrlParser: true }, function(err,db)
{
var dbo=db.db("admin")
var cursor = dbo.collection('geeks4geeks').find();
cursor.each(function (err,doc)
{
if(doc!=null)
console.log(doc);
});
db.close();
});
```

Explanation:
Using the URL, a connection with the MongoDB server is established.
Using the DB function, a connection to the admin database is created.
All the documents present in the geeks4geeks collection is retrieved and displayed in the console.
Registration Form Using Nodejs and MongoDBapp.js

```
var express=require("express");
var bodyParser=require("body-parser");
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/gfg');
var db=mongoose.connection;
db.on('error', console.log.bind(console, "connection error"));
db.once('open', function(callback){
console.log("connection succeeded");
})
var app=express()
app.use(bodyParser.json());
app.use(express.static('public'));
app.use(bodyParser.urlencoded({
extended: true
}));
app.post('/sign_up', function(req,res){
var name = req.body.name;
var email =req.body.email;
var pass = req.body.password;
var phone =req.body.phone;
var data = {
"name": name,
"email":email,
"password":pass,
"phone":phone
}
db.collection('details').insertOne(data,function(err, collection){
if (err) throw err;
console.log("Record inserted Successfully");
});
return res.redirect('signup_success.html');
})
app.get('/',function(req,res){
```

```
res.set({
'Access-control-Allow-Origin': '*'
});
return res.redirect('index.html');
}).listen(3000)
console.log("server listening at port 3000");
```

MongoDBS
MongoDoc
mongoQueries

Create

```
var MongoClient = require('mongodb').MongoClient;

//Create a database named "mydb":
var url = "mongodb://localhost:27017/mydb";

MongoClient.connect(url,{ useNewUrlParser: true }, function(err, db) {
  if (err) throw err;
  console.log("Database created!");
  db.close();
});
```

collection

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url,{ useNewUrlParser: true }, function(err, db) {
if (err) throw err;
var dbo = db.db("mydb");
dbo.createCollection("customers", function(err, res) {
if (err) throw err;
console.log("Collection created!");
db.close();
});
});
```

To insert a record, or document into a collection, we use the insertOne() method.
The first parameter of the insertOne() method is an object containing the name(s) and value(s) of each field in the document you want to insert.
It also takes a callback function where you can work with any errors, or the result of the insertion:
The first parameter of the insertMany() method is an array of objects, containing the data you want to insert.
It also takes a callback function where you can work with any errors, or the result of the insertion:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url,{ useNewUrlParser: true }, function(err, db) {
if (err) throw err;
var dbo = db.db("mydb");
var myobj = { name: "Company Inc", address: "Highway 37" };
dbo.collection("customers").insertOne(myobj, function(err, res) {
if (err) throw err;
console.log("1 document inserted");
db.close();
});
});
```

find
In MongoDB we use the find and findOne methods to find data in a collection.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url,{ useNewUrlParser: true }, function(err, db) {
if (err) throw err;
var dbo = db.db("mydb");
dbo.collection("customers").findOne({}, function(err, result) {
if (err) throw err;
console.log(result.name);
db.close();
});
});
```

MongoDB Query
When finding documents in a collection, you can filter the result by using a query object.
The first argument of the find() method is a query object, and is used to limit the search.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url,{ useNewUrlParser: true }, function(err, db) {
if (err) throw err;
var dbo = db.db("mydb");
var query = { address: "Park Lane 38" };
dbo.collection("customers").find(query).toArray(function(err, result) {
if (err) throw err;
console.log(result);
db.close();
});
});
```

MongoDB Query find
Find documents where the address starts with the letter "S":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url,{ useNewUrlParser: true }, function(err, db) {
if (err) throw err;
var dbo = db.db("mydb");
var query = { address: /^S/ };
dbo.collection("customers").find(query).toArray(function(err, result) {
if (err) throw err;
console.log(result);
db.close();
});
});
```

Delete

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url,{ useNewUrlParser: true }, function(err, db) {
if (err) throw err;
var dbo = db.db("mydb");
var myquery = { address: 'Mountain 21' };
dbo.collection("customers").deleteOne(myquery, function(err, obj) {
if (err) throw err;
console.log("1 document deleted");
db.close();
});
});
```

Delete Many

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url,{ useNewUrlParser: true }, function(err, db) {
if (err) throw err;
var dbo = db.db("mydb");
var myquery = { address: /^O/ };
dbo.collection("customers").deleteMany(myquery, function(err, obj) {
if (err) throw err;
console.log(obj.result.n + " document(s) deleted");
db.close();
});
});
```

Drop Collections
The drop() method takes a callback function containing the error object and the result parameter which returns true if the collection was dropped successfully, otherwise it returns false.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url,{ useNewUrlParser: true }, function(err, db) {
if (err) throw err;
var dbo = db.db("mydb");
dbo.collection("customers").drop(function(err, delOK) {
if (err) throw err;
if (delOK) console.log("Collection deleted");
db.close();
```

```
});
});
```

Insert Multiples

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url,{ useNewUrlParser: true }, function(err, db) {
if (err) throw err;
var dbo = db.db("mydb");
var myobj = [
{ name: 'John', address: 'Highway 71'},
{ name: 'Peter', address: 'Lowstreet 4'},
{ name: 'Amy', address: 'Apple st 652'},
{ name: 'Hannah', address: 'Mountain 21'},
{ name: 'Michael', address: 'Valley 345'},
{ name: 'Sandy', address: 'Ocean blvd 2'},
{ name: 'Betty', address: 'Green Grass 1'},
{ name: 'Richard', address: 'Sky st 331'},
{ name: 'Susan', address: 'One way 98'},
{ name: 'Vicky', address: 'Yellow Garden 2'},
{ name: 'Ben', address: 'Park Lane 38'},
{ name: 'William', address: 'Central st 954'},
{ name: 'Chuck', address: 'Main Road 989'},
{ name: 'Viola', address: 'Sideway 1633'}
];

dbo.collection("customers").insertMany(myobj, function(err, res) {
if (err) throw err;
console.log("Number of documents inserted: " + res.insertedCount);
db.close();
});
});
```

Joint

MongoDB is not a relational database, but you can perform a left outer join by using the $lookup stage.
The $lookup stage lets you specify which collection you want to join with the current collection, and which fields that should match.
Join the matching "products" document(s) to the "orders" collection:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";

MongoClient.connect(url,{ useNewUrlParser: true }, function(err, db) {
if (err) throw err;
var dbo = db.db("mydb");
dbo.collection('orders').aggregate([
{ $lookup:
{
from: 'products',
localField: 'product_id',
foreignField: '_id',
as: 'orderdetails'
}
}

]).toArray(function(err, res) {
if (err) throw err;
console.log(JSON.stringify(res));
db.close();
});
});
```

limit

The limit() method takes one parameter, a number defining how many documents to return.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url,{ useNewUrlParser: true }, function(err, db) {
if (err) throw err;
var dbo = db.db("mydb");
dbo.collection("customers").find().limit(5).toArray(function(err, result) {
if (err) throw err;
console.log(result);
db.close();
});
});
```

sort
Use the value -1 in the sort object to sort descending.

{ name: 1 } // ascending
{ name: -1 } // descending

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url,{ useNewUrlParser: true }, function(err, db) {
if (err) throw err;
var dbo = db.db("mydb");
var mysort = { name: -1 };
dbo.collection("customers").find().sort(mysort).toArray(function(err, result) {
if (err) throw err;
console.log(result);
db.close();
});
});
```

sort2
Use the sort() method to sort the result in ascending or descending order. Sort the result alphabetically by name:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url,{ useNewUrlParser: true }, function(err, db) {
if (err) throw err;
var dbo = db.db("mydb");
var mysort = { name: 1 };
dbo.collection("customers").find().sort(mysort).toArray(function(err, result) {
if (err) throw err;
console.log(result);
db.close();
});
});
```

Update
The first parameter of the updateOne() method is a query object defining which document to update.
The second parameter is an object defining the new values of the document.
Note: If the query finds more than one record, only the first occurrence is updated.
When using the $set operator, only the specified fields are updated:
updateMany() method.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";

MongoClient.connect(url,{ useNewUrlParser: true }, function(err, db) {
if (err) throw err;
var dbo = db.db("mydb");
var myquery = { address: "Valley 345" };
var newvalues = { $set: {name: "Mickey", address: "Canyon 123" } };
dbo.collection("customers").updateOne(myquery, newvalues, function(err, res) {
if (err) throw err;
console.log("1 document updated");
db.close();
});
});

var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url,{ useNewUrlParser: true }, function(err, db) {
if (err) throw err;
var dbo = db.db("mydb");
var myquery = { address: "Valley 345" };
var newvalues = {$set: {address: "Canyon 123"} };
dbo.collection("customers").updateOne(myquery, newvalues, function(err, res) {
if (err) throw err;
console.log("1 document updated");
db.close();
});
});

var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";
```

```
MongoClient.connect(url,{ useNewUrlParser: true }, function(err, db) {
if (err) throw err;
var dbo = db.db("mydb");
var myquery = { address: /^S/ };
var newvalues = {$set: {name: "Minnie"} };
dbo.collection("customers").updateMany(myquery, newvalues, function(err, res) {
if (err) throw err;
console.log(res.result.nModified + " document(s) updated");
db.close();
});
});
```