What is MongoDB?

- Not a relational database
- No joins
- No rows or columns
- Not a key-value store
- NoSQL

- Stores structured data
- Rich query interface
- Indexes
- Map-Reduce
- And more ...
**mongoDB Concept**

- **Database** is a physical container for collections.

- **Collection** is a group of MongoDB documents. A collection exists within a single database.

- **A document** is a set of key-value pairs. Documents in the same collection do not need to have the same set of fields or structure.
# RDBMS vs MongoDB

<table>
<thead>
<tr>
<th>mongodb</th>
<th>RDBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection</td>
<td>Table</td>
</tr>
<tr>
<td>Document</td>
<td>Tuple/Row</td>
</tr>
<tr>
<td>Field</td>
<td>Column</td>
</tr>
<tr>
<td>Embedded Documents</td>
<td>Table Join</td>
</tr>
</tbody>
</table>
Documents in MongoDB are JSON-style data structures composed of field-and-value pairs.
Creating a database

```
> use blog
```

“use” will switch to writing to the twitter database. If the database doesn’t exist, mongo will create it right after we insert a document to it.

```
> Db blog
```

By typing “db”, we can find which database we are using.
You can drop it using “db.dropDatabase()”. 


Creating a collection

Collections are sets of documents. Your database can have as many collections as you want.

Since Mongo has no joins, we can pull data from only one collection at a time.

```javascript
> use blog
> db.createCollection( 'posts' )
```

We can create a collection using the “createCollection” command or automatically when we write a document to a collection that doesn’t exist.
Creating a document

Documents in mongoDB can be any valid JSON format. But, they can’t contain functions.

```javascript
> var post = 
{ 
  "author" : "Joe",
  "age" : 45,
  "date" : "Sun Sep 27 2015 14:58:58 GMT+0300 (Jerusalem Daylight Time)",
  "content" : {
    title: "What do Alexander the Great and Kermit the Frog have in common? Their middle names.",
    text: null
  }
}
> db.posts.insert(post)
```
If the collection of ‘posts’ does not exist, a side effect of

```javascript
> db.posts.insert(post)
```

Is the creation of ‘posts’ collection, while inserting a new post.
Finding a document

You can find a document or documents matching a particular pattern using the find function.

To find all the posts in posts collection,

```javascript
> db.posts.find({})
{
  "_id" : ObjectId("5607da0553ac84ac586538ea"),
  "author" : "Joe",
  "age" : 45,
  "date" : "Sun Sep 27 2015 14:58:58 GMT+0300 (Jerusalem Daylight Time)",
  "content" : {
    "title" : "What do Alexander the Great and Kermit the Frog have in common? Their middle names."
    "text" : null
  }
}
```
Finding by partial match

So you have a list of authors and you want to find by name,

```javascript
> db.posts.find(
  {
    author : "Joe",
  }
)
```

You can match on more than one field,

```javascript
> db.posts.find(
  {
    author : "Joe", age: 45
  }
)
```
Finding with expressions

You can also find using expressions,

```javascript
> db.posts.find(
{ 
  age: { $gte: 25 }
}
)
```

More operators like $gte can be found here:
http://docs.mongodb.org/manual/reference/operator/query/
$exists

You can use $exists to filter on the existence of a field.

```> db.posts.find(
{  
    comments: { $exists: true } 
}
)
```

With the last query we can find all the posts that have comments.
 Projection

Find can take a second parameter which allows you to whitelist fields to pass into the output document.

```javascript
> db.posts.find( {},
{  
   _id: 0 , author: 1 , age: 1
} )
```

This will yield,

```javascript
{  
   "author": "Joe" , "age": 45
}
{  
   "author": "Bob" , "age": 40
}
```
Count

Count will convert our result set into a number. We can use it in two ways:

```javascript
> db.posts.find({},
  {
    _id: 0, author: 1, age: 1
  }).count()
```

```javascript
> db.posts.count({},
  {
    _id: 0, author: 1, age: 1
  })
```
Limit and Skip

Limit will allows you to limit the results in the output set.

```javascript
> db.posts.find({}).limit(3)
```

Will give you the first 3 posts. If we want the next 3 after 10 posts we can do the following,

```javascript
> db.posts.find({}).skip(10).limit(3)
```
We can sort the results.

This will sort the posts in ascending order of author.
We can reverse the sort by -1.
And, we can sort by more than one field and even sort by nested fields,
Cursors

Mongo gives us back a cursor object from which we can get the values when we compose a query,

```javascript
> var posts = db.posts.find({});
> while (posts.hasNext()) {
    print(tojson(posts.next()));
}
```

We can iterate over the cursor using a simple while loop.
forEach and map

We can simplify our code using forEach,

```javascript
> db.posts.find().forEach( function(post){
  print( post.author );
}
);
```

Or map, which will return an array of values.

```javascript
> var posts = db.posts.find().map( function(post){
  return post.author;
}
)
```
Updating a document

Below is a comment for a specific post. We will push it to the relevant post.

```javascript
> comment = { author: "Britney", date: new Date(),
             text: "Nice joke" }
```

The update function combines two actions, first find, than update.

```javascript
> var posts = db.posts.update( {
    _id: ObjectId("5607da0553ac84ac586538ea")
} , {
    $push: { comments: comment }
}
)
```
Saving a document

We use save function when we want to pass a specific JSON object that contains the modified object to save.

```javascript
> var post = db.posts.findOne()
> post.age = 46
> db.posts.save(post)
```

The JSON object must include the _id. If the item will be found, it will be updated. If the item won’t be found, it will be created.
Removing a document

We can remove any document by query.

```javascript
> db.posts.remove(
  {
    author : "Joe",
  }
)
```

Each post with author="Joe" will be removed from the collection.
The aggregation framework

The aggregation framework in Mongo gives us a document query pipeline.

With the aggregation framework, we might take a result set, filter it, group by a particular field, then sum values in a particular group.
Aggregation: filtering the pipeline

We want to list only posts that have comments.

```javascript
> db.posts.aggregate(
  {
    $match: {
      comments: { $exists: true }
    }
  }
)
```
Aggregation: modifying the pipeline

We can use $project to modify all the documents in the pipeline.

```javascript
> db.posts.aggregate(
  {
    $match: {
      comments: {
        $exists: true
      }
    }
  },
  {
    $project: {
      author: true,
      age: true,
      _id: false
    }
  }
)
```

We use $project to pull out the author and age from the list.
Aggregation: conditional fields

We can set the value of a field using a Boolean expression using $cond.

```bash
> db.posts.aggregate( {
    $project: {
        author: true, age: true, _id: false,
        has_comments: {
            $cond: { if: '$comments',
              then: true, else: false
            }
        }
    }
})
```
Aggregation: grouping

We can group by a single field,

```javascript
> db.posts.aggregate( {
  $group: {
    _id: '$author'
  }
})
```

Or we can group on more than one field by passing an object to `_id`

```javascript
> db.posts.aggregate( {
  $group: {
    _id: { author: '$author', age: '$age' }
  }
})
```
Aggregation: counting with group

Counting author name popularity,

```
> db.posts.aggregate( {
    $group: {
        _id: '$author',
        count: { $sum: 1 }
    }
} )
```

Aggregating by author name and return the oldest author in each group, and sort the list by age.

```
> db.posts.aggregate( { 
    $group: { 
        _id: '$author', 
        age: { $max: '$age' } ,
        count: { $sum: 1 } 
    },
    $sort: { age: 1 }
} 
)
Aggregation: unwind to count

We use unwind when we have data that contains arrays.

The above example will yield something like this:

```javascript
> db.posts.aggregate({
  $unwind: "$comments"
})
```

The above example will yield something like this:

```javascript
{
  "author": "Joe",
  "comments": "Nice joke"
}
{
  "author": "Bob",
  "comments": "I like your post"
}
{
  "author": "Bob",
  "comments": "Not interesting"
}
```
Map Reduce

We use MapReduce for complex queries. MapReduce is a data processing paradigm for condensing large volumes of data into useful aggregated results.

The map stage:
Accepts a single document and converts it to a usable form.

> var map = function() {
    emit ( this.name,  this.age )
}

> var map = function() {
    emit ( this.name,  this.age )
}
The reduce stage:
Accepts multiple mapped units and aggregate them in the way we define it.

```javascript
var reduce = function(names, age_values) {
  return Array.avg(age_values)
}
```

Finally we issue the query,

```javascript
> db.posts.mapReduce(
  map, reduce, {
    query: {},
    out: { inline: 1}
  }
)
```
Map Reduce cont’2

MapReduce is extremely flexible and easy to take on. It can handle complex operation tasks, and it is better than the aggregation framework when your data is distributed to clusters.

But if not, it might be slower than the aggregation framework pipeline since it runs JS functions.
More info

[1] https://www.mongodb.org/