Advanced Databases

MongoDB
Part 1

Giulia Vilone

giulia.vilone@tudublin.ie

MongoDB profile

- Document-oriented NoSQL database.
- Schema-free.
- Based on Binary JSON; BSON.
- Organized in Group of Documents collections
  - Informal namespacing
- Auto-sharding in order to scale horizontally.
- Simple query language. Rich, document-based queries.
- Map/Reduce support
- Open Source (GNU AGPL v4.4)

Motivations

- Problems with SQL
  - Rigid schema
  - Not easily scalable (designed for 90's technology or worse)
  - Requires unintuitive joins

- Benefits of MongoDB
  - Easy interface with common languages (Java, Javascript, PHP, etc.)
  - DB tech should run anywhere (VM's, cloud, etc.)
  - Keeps essential features of RDBMS's while learning from key-value NoSQL systems

Data model

- Document-Based (max 16 MB)

- Documents are in BSON format, consisting of field-value pairs

- Each document stored in a collection
- Collections:
  - Have index set in common
  - Like tables of relational databases.
  - Documents do not have to have uniform structure

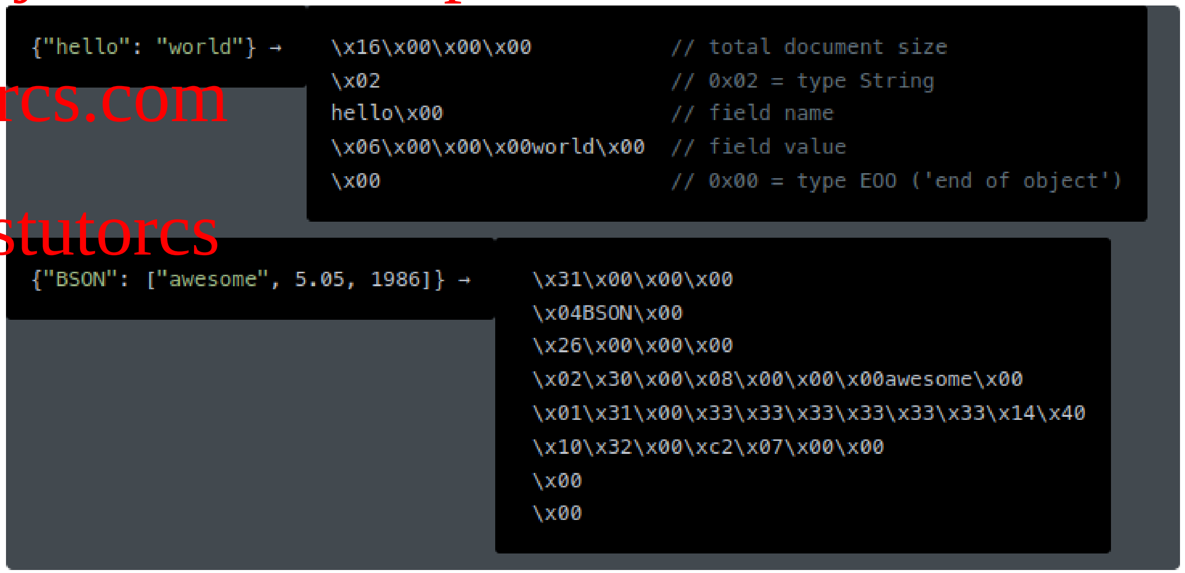
JSON

- “JavaScript Object Notation”
- Easy for humans to write/read, easy for computers to parse/generate
- Objects can be nested
- Built on
  - name/value pairs
  - ordered list of values

https://json.org/

BSON

- “Binary JSON”
- Binary-encoded serialization of JSON-like docs
- Also allows “referencing”
- Embedded structure reduces need for joins
- Goals:
  - Lightweight
  - Traversable
  - Efficient (decoding and encoding)



```
{"hello": "world"} →   \x16\x00\x00\x00              // total document size
                       \x02                          // 0x02 = type String
                       hello\x00                     // field name
                       \x06\x00\x00\x00world\x00     // field value
                       \x00                          // 0x00 = type EOO ('end of object')
```

```
{"BSON": ["awesome", 5.05, 1986]} →   \x31\x00\x00\x00
                                      \x04BSON\x00
                                      \x26\x00\x00\x00
                                      \x02\x30\x00\x08\x00\x00\x00awesome\x00
                                      \x01\x31\x00\x33\x33\x33\x33\x33\x33\x14\x40
                                      \x10\x32\x00\xc2\x07\x00\x00
                                      \x00
                                      \x00
```

https://bsonspec.org/

BSON example

```
{
"_id" :  "37010"
"city" :            "ADAMS",
"pop" :             2660,
"state" :           "TN",
“councilman” : {
                  name: “John Smith”
                  address: “13 Scenic Way”
                }
}
```

The _id field

- By default, each document contains an _id field. This field has several special characteristics:
  - Value serves as primary key for collection.
  - Value is unique, immutable, and may be any non-array type.
  - Default data type is ObjectId, which is “small, likely unique, fast to generate, and ordered.” Sorting on an ObjectId value is roughly equivalent to sorting on creation time.

https://docs.mongodb.com/manual/reference/bson-types/

The _id field

- **_id** is a 12 bytes hexadecimal number which assures the uniqueness of every document:
  - First 4 bytes -> current timestamp
  - Next 3 bytes -> machine id
  - Next 2 bytes -> process id of MongoDB server
  - Last 3 bytes -> simple incremental VALUE
- You can provide _id while inserting the document. If you don’t provide it, then MongoDB provides a unique id for every document.

MongoDB vs. SQL

MongoDB	SQL
Document	Tuple
Collection	Table/View
PK: _id Field	PK: Any Attribute(s)
Uniformity not Required	Uniform Relation Schema
Index	Index
Embedded Structure	Joins
Shard	Partition

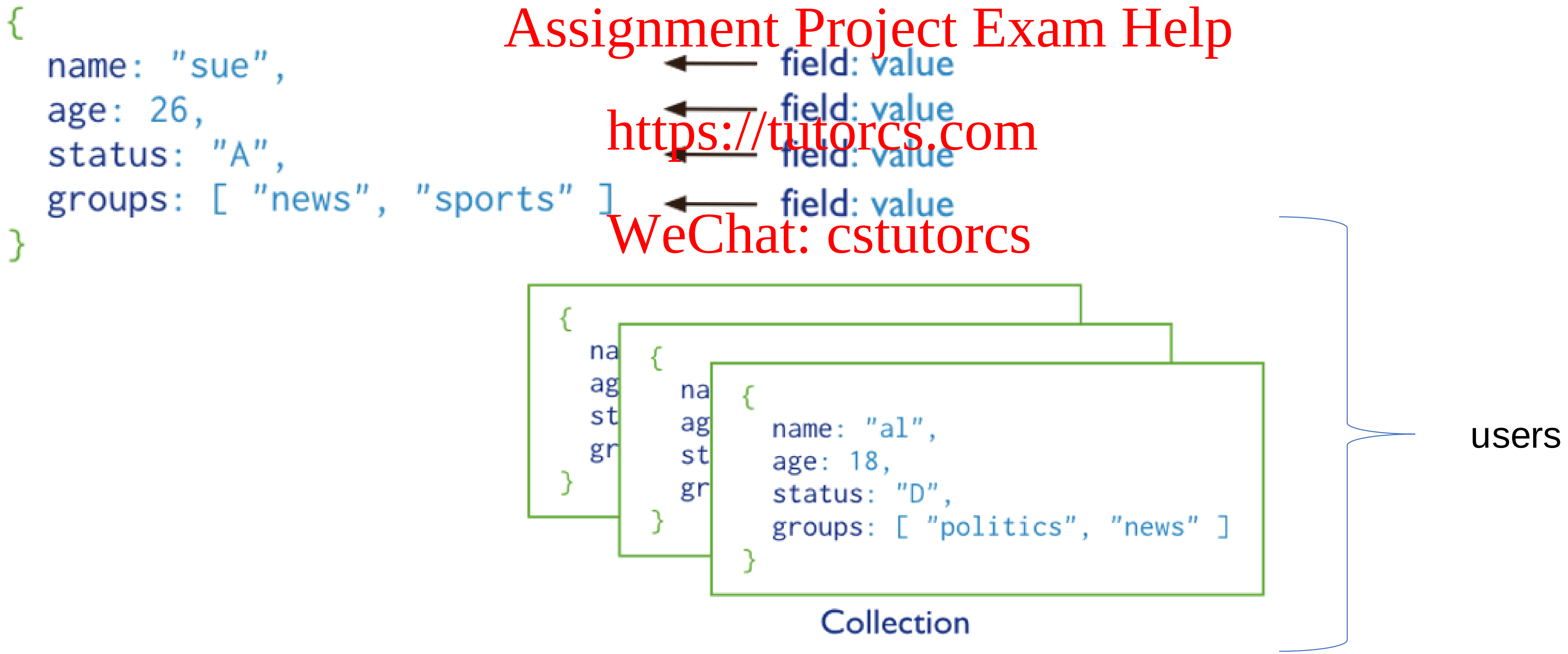
Data type

Data type	Description
String	The most used datatype to store the data. String in MongoDB must be UTF-8 valid
Integer	Numerical values of 32 bit or 64 bit, depending upon your server.
Boolean	Boolean (True/False, 0/1) values
Double	Floating point values
Min/Max keys	Used to compare a value against the lowest and highest BSON elements
Arrays	Store arrays or list or multiple values into one key.
Timestamp	ctimestamp. Handy for recording when a document has been modified or added
Object	Store embedded documents

Data type

Data type	Description
Null	Null values
Symbol	Identical usage of string type, but generally reserved for languages utilizing specific symbol types
Date	Current date or time in UNIX time format. You can specify your own date/time by creating a Date object and passing day, month, year into it.
Object ID	Document's ID
Binary data	Binary data
Code	Store JavaScript code into a document
Regular expression	Store regular expressions

Basic operations



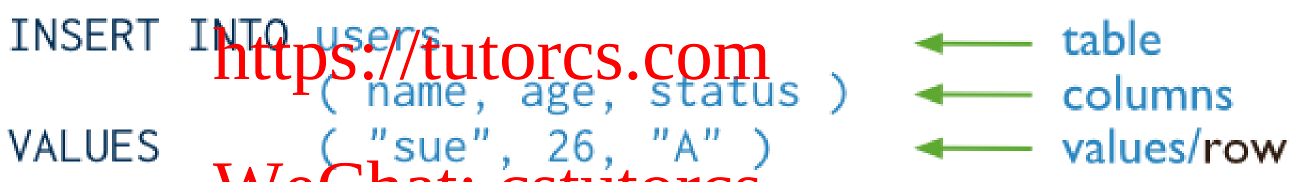
```
{
   name: "sue",
   age: 26,
   status: "A",
   groups: [ "news", "sports" ]
}
```

```
{
   name: "al",
   age: 18,
   status: "D",
   groups: [ "politics", "news" ]
}
```

CRUD operations - Create

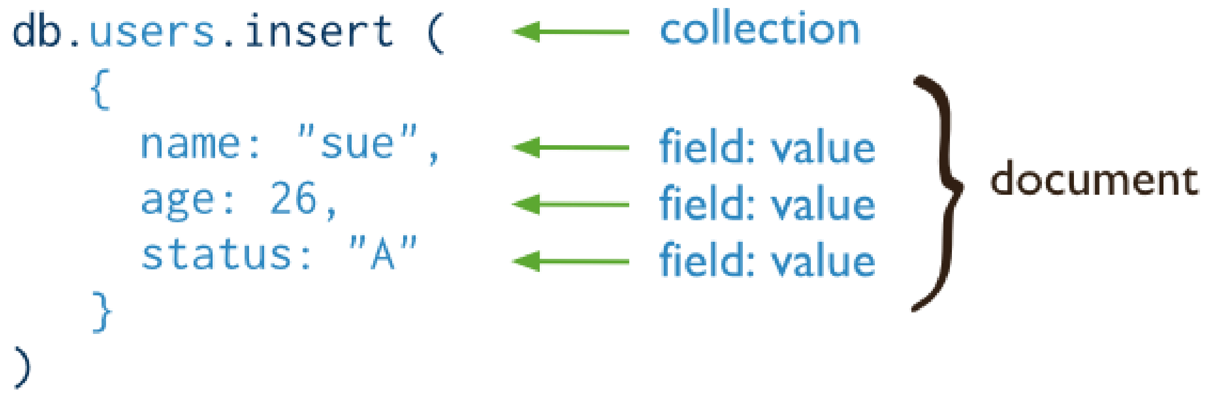
Insert a new user.

SQL



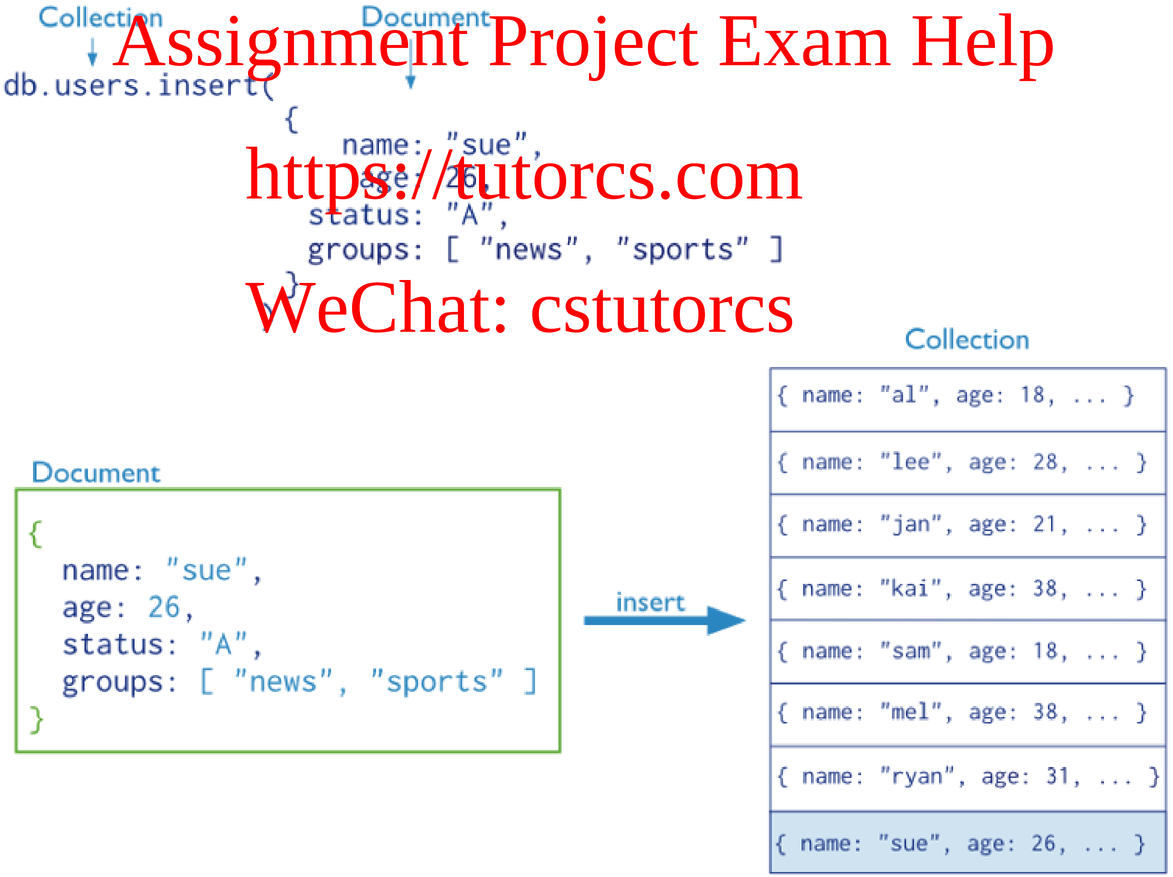
```
INSERT INTO users
            ( name, age, status )
VALUES      ( "sue", 26, "A" )
```

MongoDB



```
db.users.insert (
   {
     name: "sue",
     age: 26,
     status: "A"
   }
)
```

CRUD operations - Create



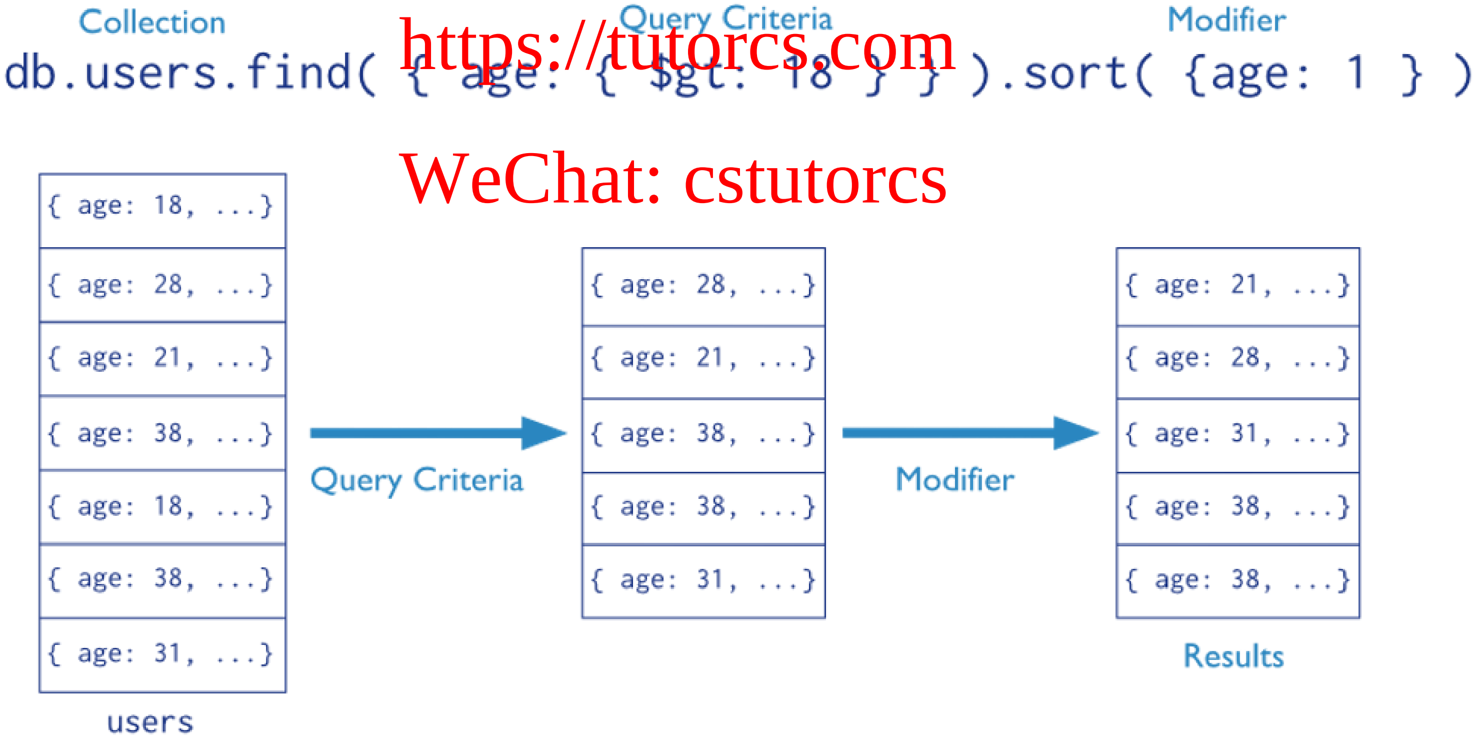
```
db.users.insert(
   {
      name: "sue",
      age: 26,
      status: "A",
      groups: [ "news", "sports" ]
   }
)
```

```
{
  name: "sue",
  age: 26,
  status: "A",
  groups: [ "news", "sports" ]
}
```

```
{ name: "al", age: 18, ... }
{ name: "lee", age: 28, ... }
{ name: "jan", age: 21, ... }
{ name: "kai", age: 38, ... }
{ name: "sam", age: 18, ... }
{ name: "mel", age: 38, ... }
{ name: "ryan", age: 31, ... }
{ name: "sue", age: 26, ... }
```

CRUD operations - Read

Find the users of age greater than 18 and sort by age.



```
db.users.find( { age: { $gt: 18 } } ).sort( {age: 1 } )
```

{ age: 18, ...}
{ age: 28, ...}
{ age: 21, ...}
{ age: 38, ...}
{ age: 18, ...}
{ age: 38, ...}
{ age: 31, ...}

{ age: 28, ...}
{ age: 21, ...}
{ age: 38, ...}
{ age: 38, ...}
{ age: 31, ...}

{ age: 21, ...}
{ age: 28, ...}
{ age: 31, ...}
{ age: 38, ...}
{ age: 38, ...}

Logical tests

Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>:<value>}	db.mycol.find({"by":"tutorials point"}).pretty()	where by = 'tutorials point'
Less Than	{<key>:{$lt:<value>}}	db.mycol.find({"likes":{$lt:50}}).pretty()	where likes < 50
Less Than Equals	{<key>:{$lte:<value>}}	db.mycol.find({"likes":{$lte:50}}).pretty()	where likes <= 50
Greater Than	{<key>:{$gt:<value>}}	db.mycol.find({"likes":{$gt:50}}).pretty()	where likes > 50
Greater Than Equals	{<key>:{$gte:<value>}}	db.mycol.find({"likes":{$gte:50}}).pretty()	where likes >= 50
Not Equals	{<key>:{$ne:<value>}}	db.mycol.find({"likes":{$ne:50}}).pretty()	where likes != 50

Querying

SQL

```
SELECT *
FROM <table>
WHERE <field> = <value1> OR <field> = <value2>;
```

MongoDB

OR

```
db.<collection>.find({ $or: [<field>:<value1>
                             <field>:<value2>]
})
```

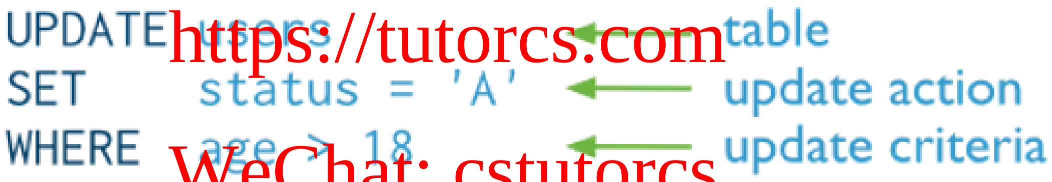
Checking for multiple values of same field

```
db.<collection>.find({<field>: {$in [<value>, <value>]}})
```

CRUD operations - Update

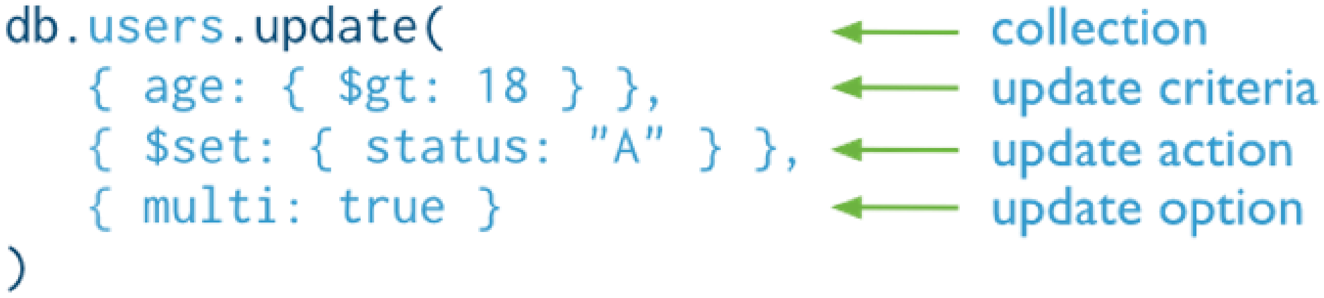
Update the users of age greater than 18 by setting the status field to A.

SQL



```
UPDATE users
SET    status = 'A'
WHERE  age > 18
```

MongoDB

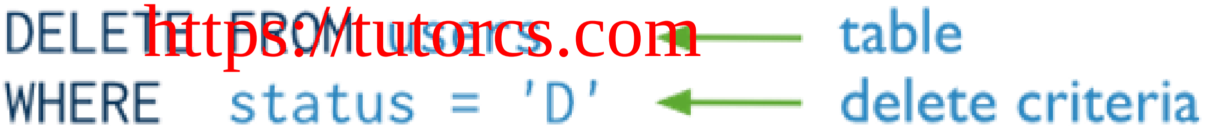


```
db.users.update(
   { age: { $gt: 18 } },
   { $set: { status: "A" } },
   { multi: true }
)
```

CRUD operations – Delete

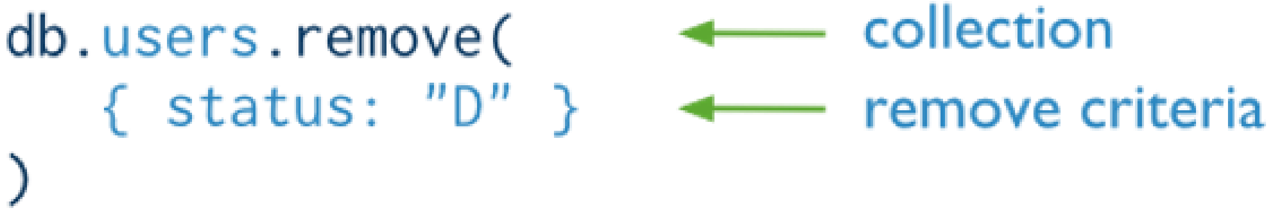
Delete the users with status equal to D

SQL



```
DELETE FROM users
WHERE  status = 'D'
```

MongoDB



```
db.users.remove(
   { status: "D" }
)
```

Schema design

SQL vs. MongoDB concepts

RDBMS		MongoDB
Database	→	Database
Table	→	Collection
Row	→	Document
Index	→	Index
Join	→	Embedded document
Foreign key	→	Reference

MongoDB is basically schema free

- The purpose of schema in SQL is for meeting the requirements of tables and SQL implementation.

- Every "row" in a database "table" is a data structure, much like a "struct" in C, or a "class" in Java. A table is then an array (or list) of such data structures.

- So, we what we design in mongoDB is basically same way how we design a compound data type binding in JSON.

There are some patterns

- **Embedding** - Embed the document into the other document
  - Similar to denormalized joins

- **Linking** (also known as reference)
  - Use the id of a document as a field in another document
  - Similar to a FK in SQL

One-to-one relationship - Embedding

```
zip = {
        _id: 35004,
        city: "ACMAR",
        location: [-86, 33],
        population: 6065,
        state: "AL"
}
council_person = {
        zip_id = 35004,
        name: "John Doe",
        address: "123 Fake St.",
        phone: 123456
}
```

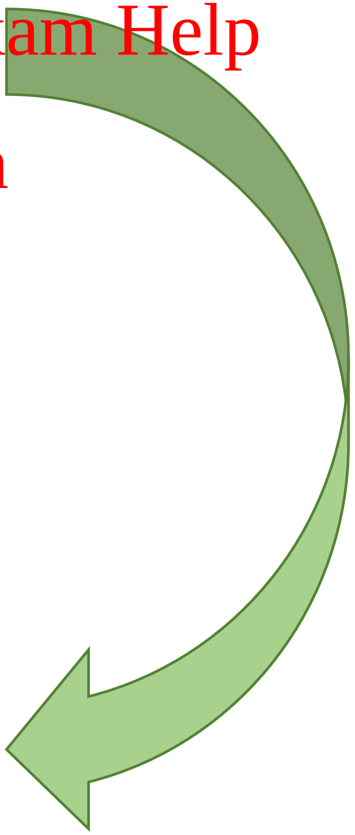
```
zip = {
        _id: 35004 ,
        city: "ACMAR"
        location: [-86, 33],
        population: 6065,
        state: "AL",

        council_person: {
        name: "John Doe",
        address: "123 Fake St.",
        phone: 123456
       }
}
```

One-to-one relationship - Embedding

```
book = {
   title: "MongoDB: The Definitive Guide",
   authors: [ "Kristina Chodorow", "Mike Dirolf" ]
   published_date: ISODate("2010-09-24"),
   pages: 216,
   language: "English",
         publisher: {
            name: "O’Reilly Media",
            founded: "1980",
            location: "CA” }
}
```

One-to-one relationship - Linking

```
publisher = {
    _id: "oreilly",
    name: "O'Reilly Media",
    founded: "1980",
    location: "CA"}
book = {
    title: "MongoDB: The Definitive Guide",
    authors: [ "Kristina Chodorow", "Mike Dirolf" ]
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English",
    publisher_id: "oreilly"}
```



Linking vs. Embedding

- Embedding is a bit like pre-joining data.
- Document level operations are easy for the server to handle.
- Embed when the “many” objects always appear with (viewed in the context of) their parents.
- Linking when you need more flexibility, less redundancy.

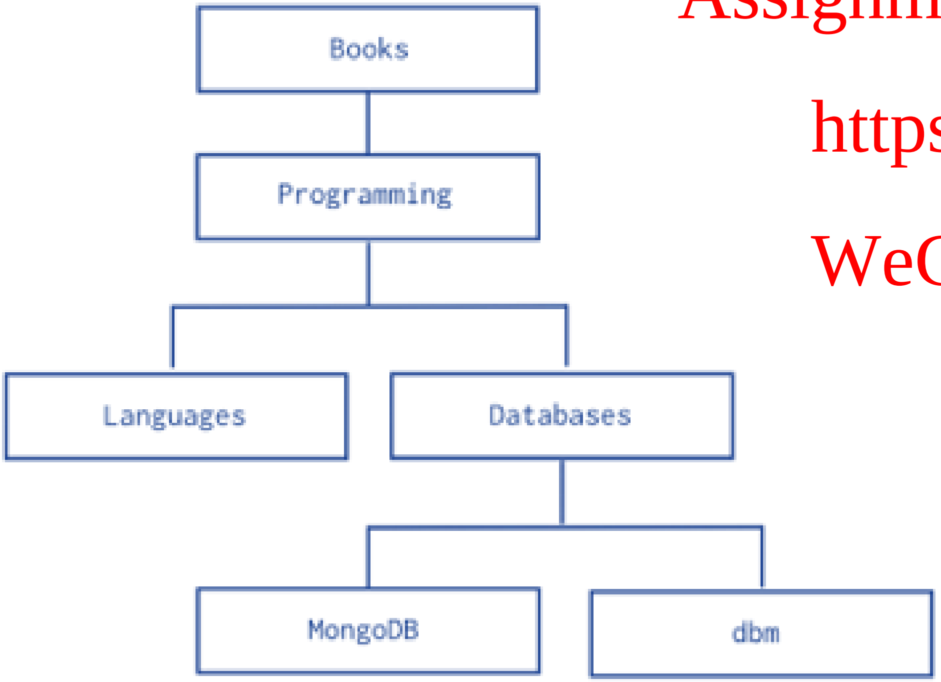
Modelling checkouts

```
student = {
    _id: "joe"
    name: "Joe Bookreader"
    join_date: ISODate("2011-10-15"),
    address: { ... }
}

book = {
    _id: "123456789"
    title: "MongoDB: The Definitive Guide",
    authors: [ "Kristina Chodorow", "Mike Dirolf" ],
    ...
}
```

Modelling checkouts

```
student = {
    _id: "joe"
    name: "Joe Bookreader",
    join_date: ISODate("2011-10-15"),
    address: { ... },
    checked_out: [
        { _id: "123456789", checked_out: "2012-10-15" },
        { _id: "987654321", checked_out: "2012-09-12" },
        ...
    ]
}
```

Model tree structure



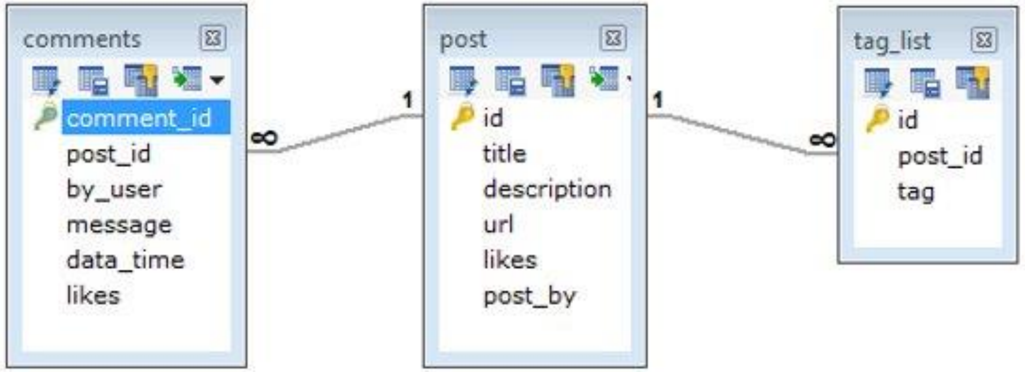
```
»   db.categories.insert( { _id: "MongoDB", parent: "Databases" } )
    db.categories.insert( { _id: "dbm", parent: "Databases" } )
    db.categories.insert( { _id: "Databases", parent: "Programming" } )
    db.categories.insert( { _id: "Languages", parent: "Programming" } )
    db.categories.insert( { _id: "Programming", parent: "Books" } )
    db.categories.insert( { _id: "Books", parent: null } )
```

```
»   db.categories.findOne( { _id: "MongoDB" } ).parent
»   db.categories.ensureIndex( { parent: 1 } )
»   db.categories.find( { parent: "Databases" } )
```

Another example

Suppose a client needs a database design for his blog/website and knows the differences between RDBMS and MongoDB schema design. His website has the following requirements:

- Every post has the unique title, description and url.
- Every post can have one or more tags.
- Every post has the name of its publisher and total number of likes.
- Every post has comments given by users along with their name, message, data-time and likes.
- On each post, there can be zero or more comments.



MongoDB document

```
{
  _id: POST_ID
  title: TITLE_OF_POST,
  description: POST_DESCRIPTION,
  by: POST_BY,
  url: URL_OF_POST,
  tags: [TAG1, TAG2, TAG3],
  likes: TOTAL_LIKES,
  comments: [ {
      user:'COMMENT_BY',
      message: TEXT,
      dateCreated: DATE_TIME,
      like: LIKES
    },
    {
      user:'COMMENT_BY',
      message: TEXT,
      dateCreated: DATE_TIME,
      like: LIKES
    }]
}
```

Some consideration while designing a schema in MongoDB

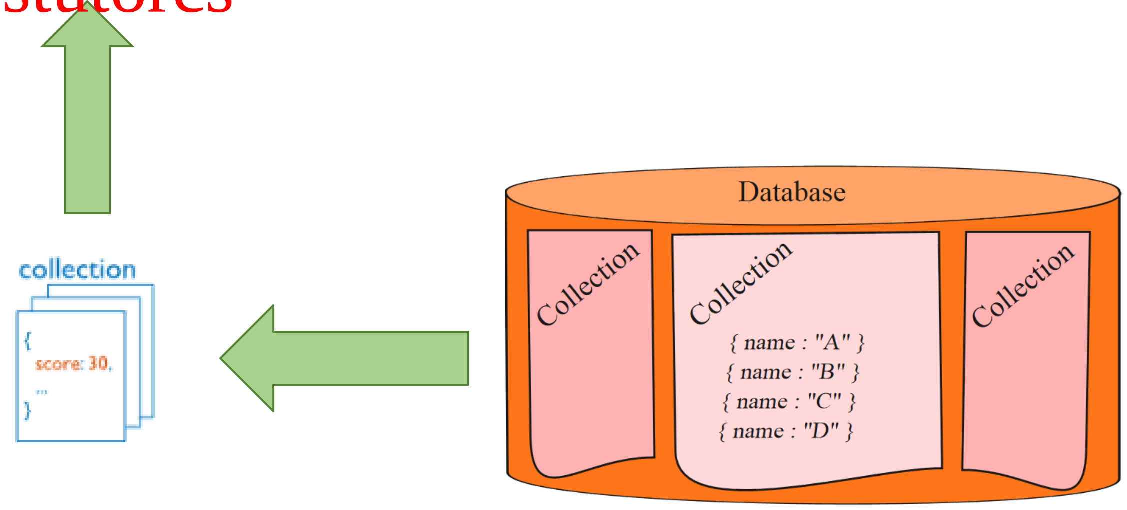
- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
- Duplicate the data (but limited) because disk space is cheap as compared to compute time.
- Do joins while write, not on read.
- Optimize your schema for most frequent use cases.
- Do complex aggregation in the schema.

Index in MongoDB

Before index

- What does database normally do when we query?
  - MongoDB must scan every document.
  - Inefficient because process large volume of data

```
db.users.find( { score: { “$lt” : 30} } )
```



Definition of index

Definition:

Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form.

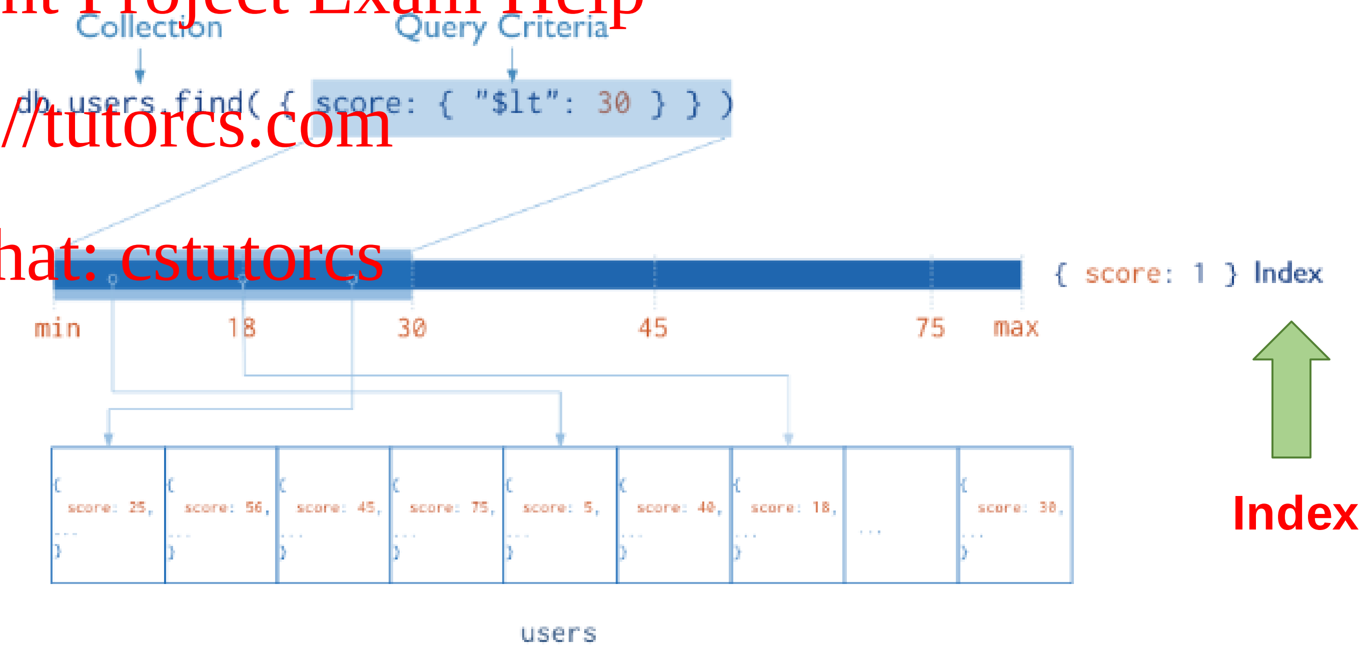


Diagram of a query that uses an index to select

Index in MongoDB

- Creation index
  - db.users. createIndex({ score: 1 })
- Show existing indexes
  - db.users.getIndexes()
- Drop index
  - db.users.dropIndex({score: 1})

Index in MongoDB

- Types of index:
  - **Single Field Indexes**
  - Compound Field Indexes
  - Multikey Indexes

Single Field Indexes

```
db.users. createIndex( { score: 1 } )
```

Ascending index; for descending index, specify a value of -1

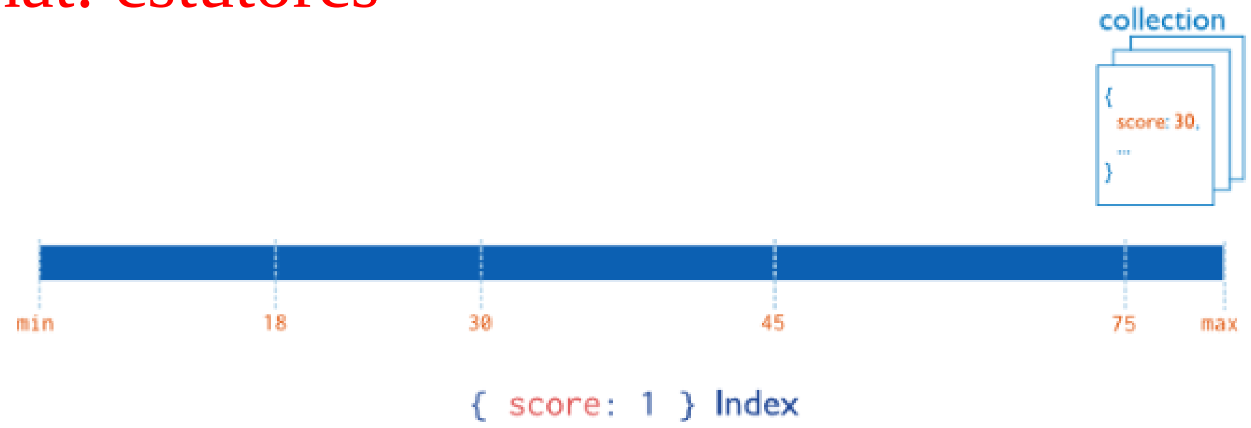


Diagram of an index on the score field (ascending).

Index in MongoDB

- Types of index:
  - Single Field Indexes
  - **Compound Field Indexes**
  - Multikey Indexes

Compound Field Indexes

```
db.users. createIndex( { userid:1, score: -1 } )
```

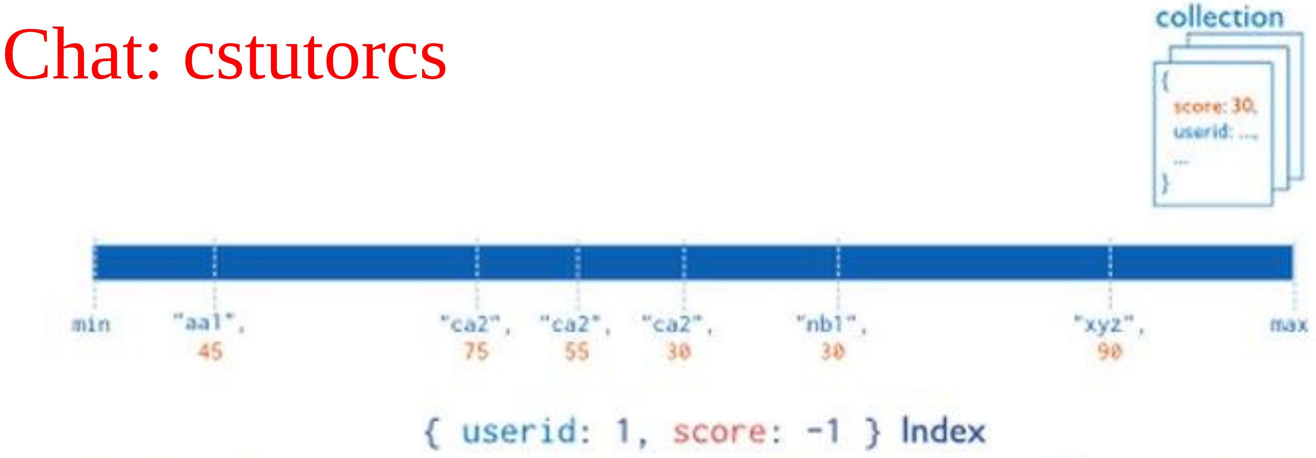


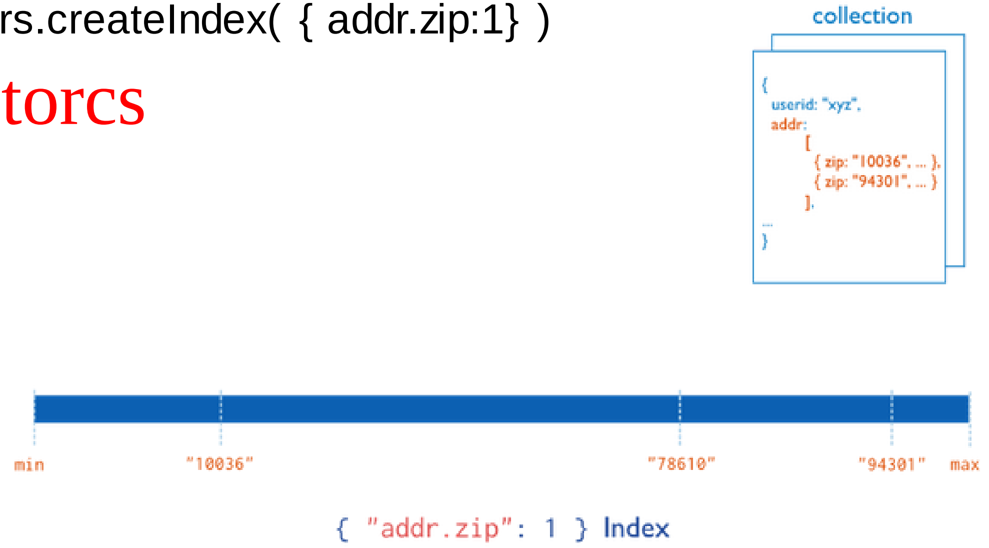
Diagram of a compound index on the userid field (ascending) and the score field (descending). The index sorts first by the userid field and then by the score field.

Index in MongoDB

- Types of index:
  - Single Field Indexes
  - Compound Field Indexes
  - **Multikey Indexes**

Multikey Indexes

```
db.users.createIndex( { addr.zip:1} )
```



```
{
  userid: "xyz",
  addr:
       [
         { zip: "10036", ... },
         { zip: "94301", ... }
       ],
...
}
```

Diagram of a multikey index on the `addr.zip` field. The `addr` field contains an array of address documents. The address documents contain the `zip` field.

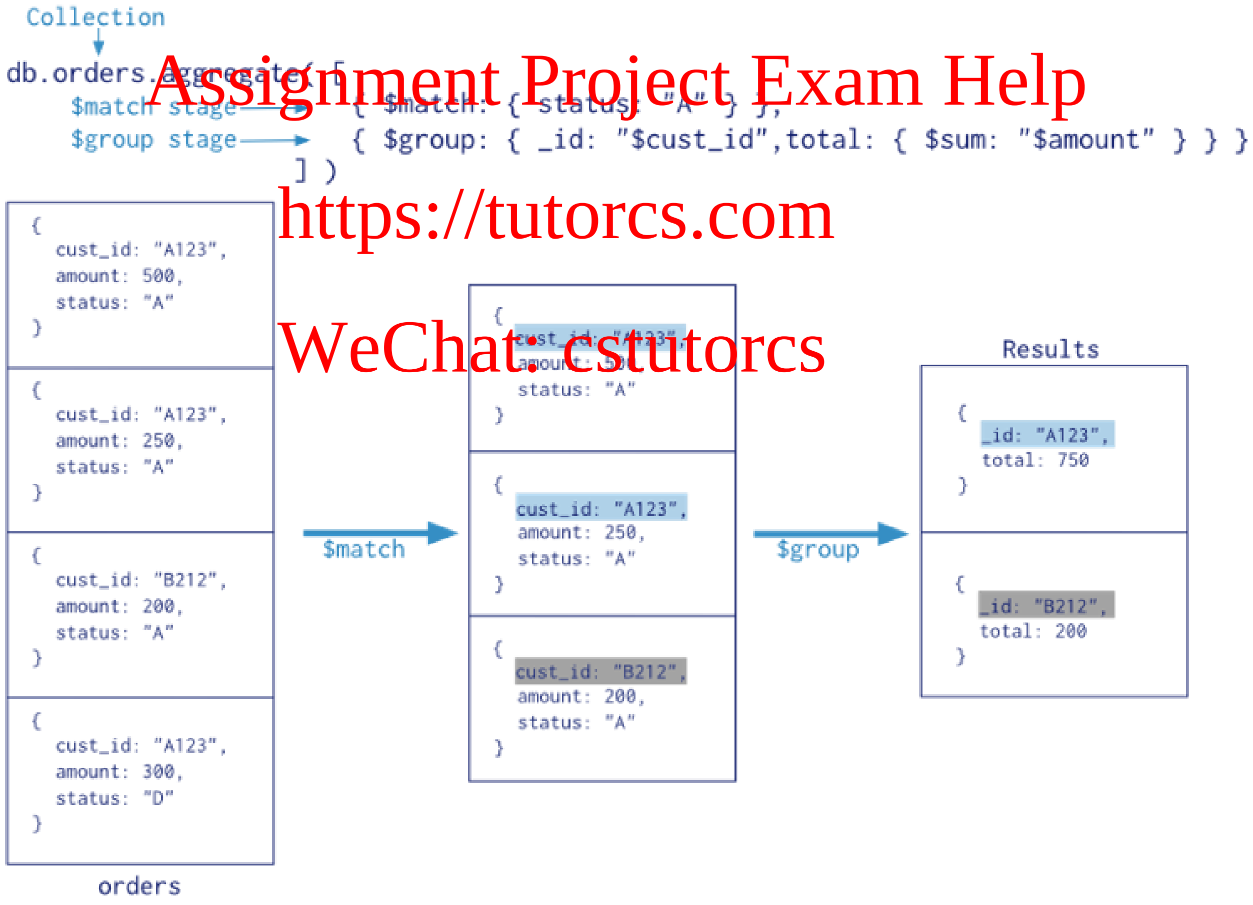
Aggregation

- Operations that process data records and return computed results.
- MongoDB provides aggregation operations.
- Running data aggregation on the mongod instance simplifies application code and limits resource requirements.
- Aggregation can be done with
  - Papeline ($group operator)
  - Map_reduce

Pipelines

- MongoDB’s aggregation framework is modelled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into an aggregated result.
- The most basic pipeline stages provide *filters* that operate like queries and *document transformations* that modify the form of the output document.
- Other pipeline operations provide tools for *grouping* and *sorting* documents by specific field or fields as well as tools for aggregating the contents of arrays
- Pipeline stages can use operators for tasks such as calculating the average or concatenating a string.
- Pipeline is the preferred method for data aggregation in MongoDB.

Aggregation using a pipeline



```
db.orders.aggregate( [
   { $match: { status: "A" } },
   { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
] )
```

```
{
   cust_id: "A123",
   amount: 500,
   status: "A"
}
{
   cust_id: "A123",
   amount: 250,
   status: "A"
}
{
   cust_id: "B212",
   amount: 200,
   status: "A"
}
{
   cust_id: "A123",
   amount: 300,
   status: "D"
}
```

```
{
   cust_id: "A123",
   amount: 500,
   status: "A"
}
{
   cust_id: "A123",
   amount: 250,
   status: "A"
}
{
   cust_id: "B212",
   amount: 200,
   status: "A"
}
```

```
{
   _id: "A123",
   total: 750
}
{
   _id: "B212",
   total: 200
}
```

Aggregator operators

Expression	Description	Example
$sum	Sums up the defined value from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : "$likes"}}}])
$avg	Calculates the average of all given values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$avg : "$likes"}}}])
$min. $min	Gets the minimum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$min : "$likes"}}}])
$push	Inserts the values to an array in the resulting document.	db.mycol.aggregate([{$group : {_id : "$by_user", url : {$push: "$url"}}}])
$addToSet	Inserts the value to an array in the resulting document but does not create duplicates.	db.mycol.aggregate([{$group : {_id : "$by_user", url : {$addToSet : "$url"}}}])
$first	Gets the first document from the source documents according to the grouping. Typically this makes only sense together with some previously applied “$sort”-stage.	db.mycol.aggregate([{$group : {_id : "$by_user", first_url : {$first : "$url"}}}])
$last	Gets the last document from the source documents according to the grouping. Typically this makes only sense together with some previously applied “$sort”-stage.	db.mycol.aggregate([{$group : {_id : "$by_user", last_url : {$last : "$url"}}}])

Some examples

```
db.test_db.find({gender:  'f'});
db.test_db.find({gender:  'm'});
db.test_db.find({gender:  'm', $or: [{nationality: 'english'}, {nationality:'american'}]});
db.test_db.find({gender:  'm', $or: [{nationality: 'english'},{nationality: 'american'}]}).sort({nationality:  -1});

db.test_db.find({gender:  'm', $or: [{nationality: 'english'},{nationality: 'american'}]}).sort({nationality:  -1, first: 1});
db.test_db.find({gender:  'm', $or: [{nationality: 'english'},{nationality: 'american'}]}).limit(2);

db.test_db.update({first:  'james', last: 'caan'}, {$set:{hair_colour:  'brown'}});
db.test_db.update({  nationality: "american"  },{ $inc: { age: 2} })

db.test_db.aggregate(  [ { $match: { 'age' : { '$gte' : 37 }}}, {$group: { _id: '$nationality', total : { $sum : 1} }}] );
db.test_db.aggregate(  [ { $match: { 'age' : { '$gte' : 37 }}}, {$group: { _id: '$gender', total : { $sum : 1} }}] );
db.test_db.aggregate(  [ {$group: { _id: '$gender', avg_age  : { $avg : '$age'} }}] );
```

Output from a shell

- Some practical tips
- When running from a script:
  - Output of a query is not displayed by default, use the following function to display it.

```
function get_results (result)
   { print(tojson(result)); }

db.col.find(…).forEach(get_results…)
```

Document update

- MongoDB does not allow to update a field by using an expression containing other fields of the collections

- Therefore, you cannot write $field = $field + 1 or something similar (as we did for SQL)

- For numeric update, use the $inc operator with the update function
- Or... JavaScript always an option!

- Example:

```
{ _id: 1, item: "abc123", quantity: 10, metrics: { orders: 2,
ratings: 3.5 } }
db.products.update( { item: "abc123" }, { $inc: { quantity: -2,
"metrics.orders": 1 } } )
```

Multiple updates

- Add {multi: true). It controls the ability of MongoDB to update more than one field in a single query
- Try:

```
> db.julia.update({first: { $ne: "aa"} },{ $inc: {age: 2}})
WriteResult({"nMatched": 1, "nUpserted": 0, "nModified": 1})

> db.julia.update({first: {$ne: "aa"} },{$inc: {age: 2}},
{multi: true})
WriteResult({ "nMatched": 7, "nUpserted": 0, "nModified": 7})
```

MongoDB & JavaScript

You can store your commands in a JavaScript script and execute them with

```
> mongo js_file.js
```

Shell – Script commands

SHELL	**JS SCRIPT**
show dbs, show databases	db.adminCommand('listDatabases')
use <db>	db = db.getSiblingDB('<db>')
show collections	db.getCollectionNames()

Cursors

```
var myCursor = db.inventory.find( { type: 'food' } );

while (myCursor.hasNext()) {
      print(tojson(myCursor.next())); }


myCursor.forEach(printjson);


var documentArray = myCursor.toArray();
var myDocument = documentArray[3];


var myDocument = myCursor[3];
```

Map-Reduce

- Algorithm ("template") to perform distributed parallel computation

- Used in MongoDB for performing distributed queries, for instance aggregated queries

- MongoDB provides the function map-reduce
- Map reduce is a concept from functional programming

```
map even [3,4,5,6,7,9] = [4,6]
```

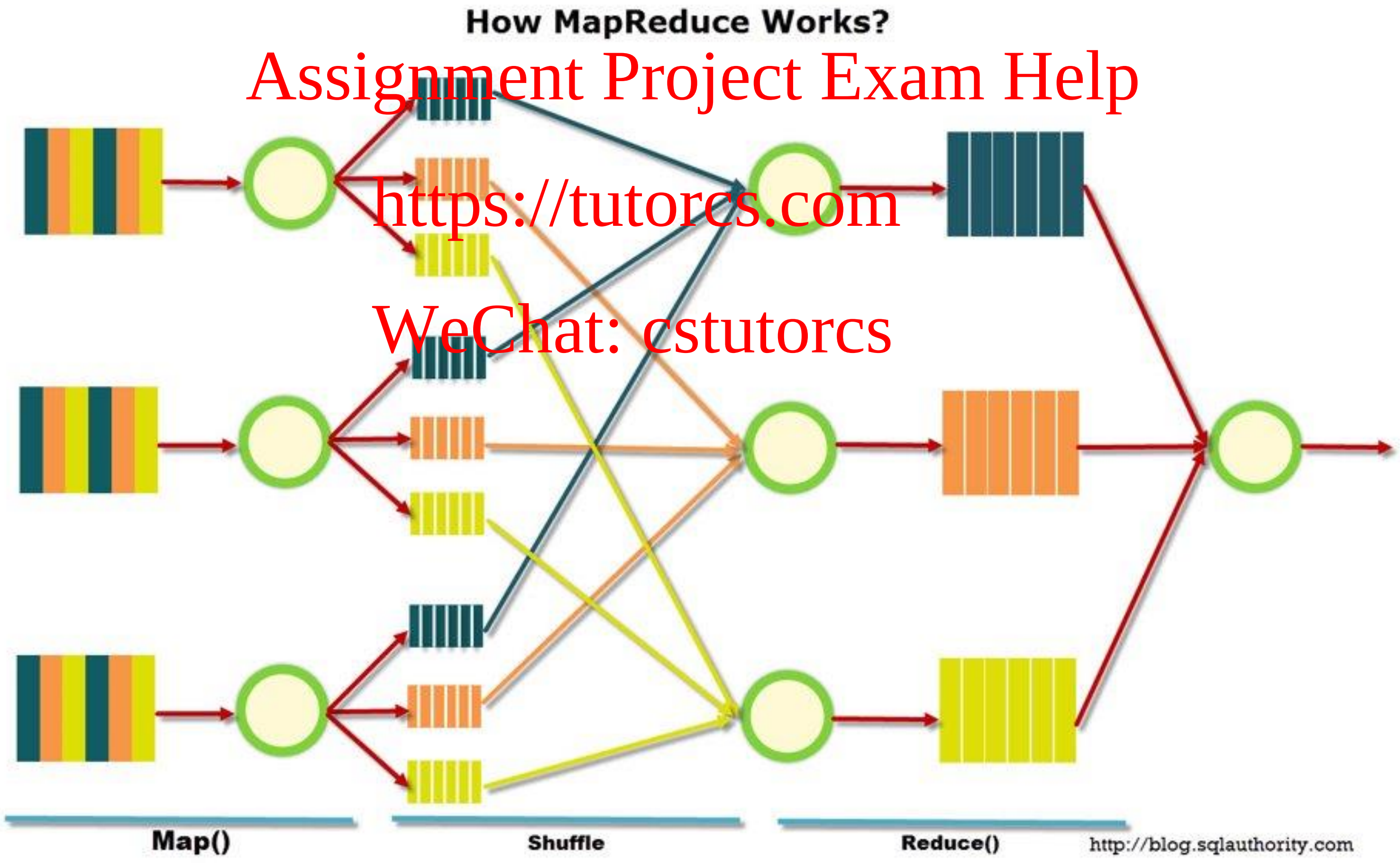
Map-Reduce

- Has two phases:
  - A map stage that processes each document and emits one or more objects for each input document.
  - A reduce phase that combines the output of the map operation.
  - An optional finalize stage for final modifications to the result.
- Uses Custom JavaScript functions
  - Provides greater flexibility but is less efficient and more complex than the aggregation pipeline.
- Can have output sets that exceed the 16-megabyte output limitation of the aggregation pipeline.
- As of MongoDB 5.0 the map-reduce operation is deprecated; use an aggregation pipeline instead.

Map-Reduce in MongoDB

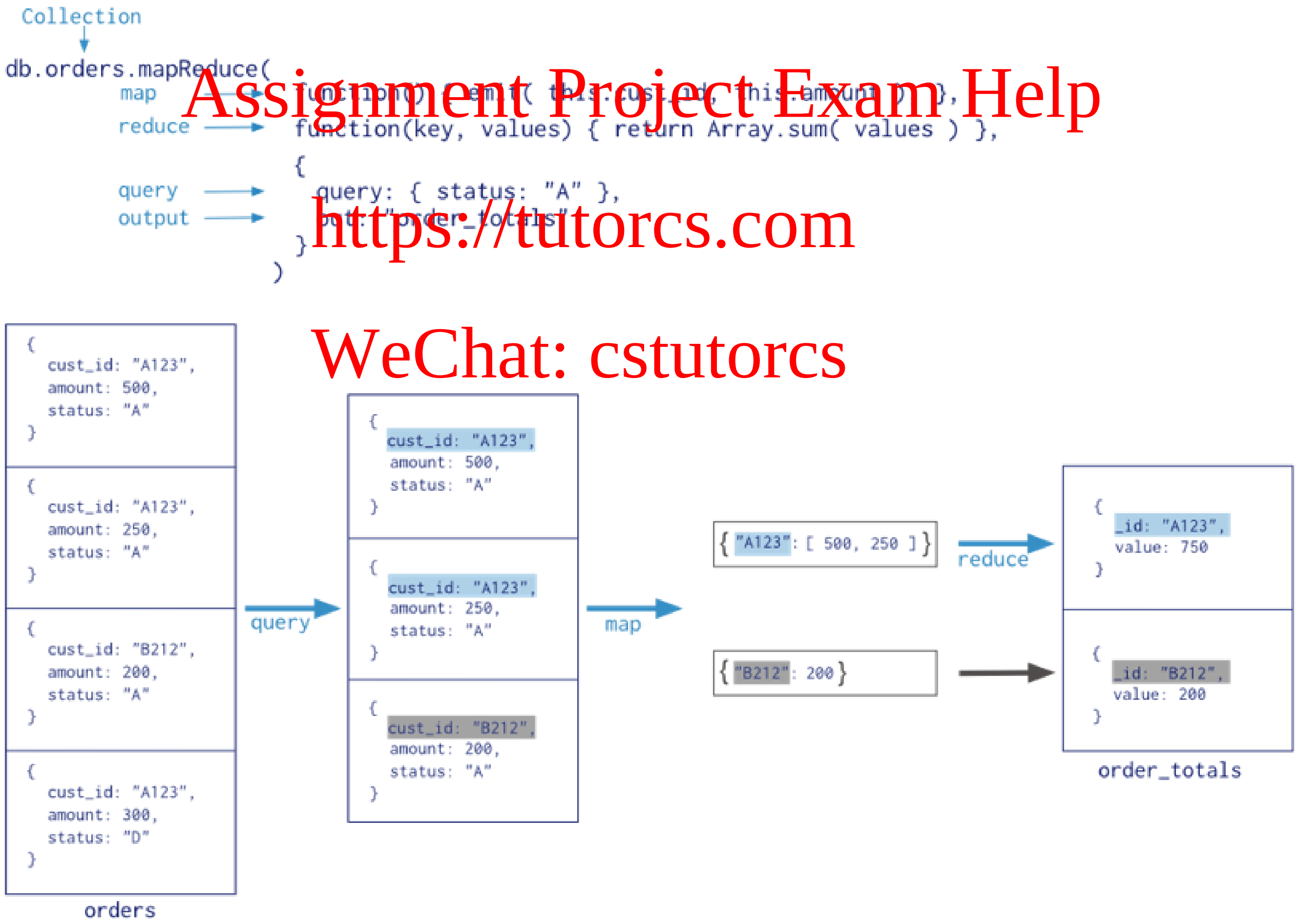
```
db.runCommand( {
    mapReduce: <collection>,
    map: <function>,
    reduce: <function>,
    {
        out: <output>,
        query: <document>,
        sort: <document>,
        limit: <number>,
    }
    finalize: <function>,
    verbose: <boolean> } )
```

Map-Reduce



http://blog.sqlauthority.com

Map-Reduce example



```
db.orders.mapReduce(
                    function() { emit( this.cust_id, this.amount ); },
                    function(key, values) { return Array.sum( values ) },
                    {
                      query: { status: "A" },
                      out: "order_totals"
                    }
                  )
```

```
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
{
  cust_id: "A123",
  amount: 300,
  status: "D"
}
```

orders

```
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
```

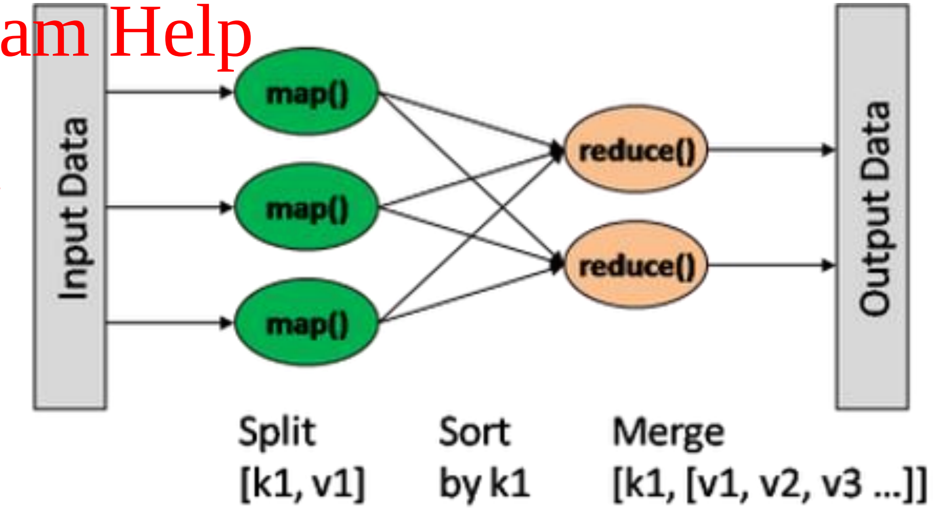
```
{ "A123": [ 500, 250 ] }
{ "B212": 200 }
```

```
{
  _id: "A123",
  value: 750
}
{
  _id: "B212",
  value: 200
}
```

order_totals

Map-Reduce

```
db.collection.mapReduce(
        <mapfunction>,
        <reducefunction>,
        {
                out: <collection>,
                query: <>,
                sort: <>,
                limit: <number>,
                finalize: <function>,
                verbose: <boolean>
        } )
```



```
var mapFunction1 = function() { emit(this.cust_id, this.price); };

var reduceFunction1 = function(keyCustId, valuesPrices)
{ return sum(valuesPrices); };
```

Map-Reduce as JavaScript

In MongoDB, map-reduce operations use custom JavaScript functions to *map*, or associate, values to a key. If a key has multiple values mapped to it, the operation *reduces* the values for the key to a single object.

Map function

function() { ... emit(key, value); }

The map function has the following requirements:

- In the map function, reference the current document as **this** within the function.
- The map function should *not* access the database for any reason.
- The map function should be pure or have *no* impact outside of the function (i.e., side effects.)
- The map function may optionally call **emit(key, value)** any number of times to create an output document associating key with value.