MongoDB-2

WPI, Mohamed Eltabakh

Query Language in MongoDB

Find() Operator

Means ascending

Collection — Query Criteria — Modifier

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

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

→ Query Criteria →

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

→ Modifier →

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

Find() + Projection

```
db.users.find(
    { age: { $gt: 18 } },
    { name: 1, address: 1 }
).limit(5)
```

← collection
← query criteria
← projection
← cursor modifier

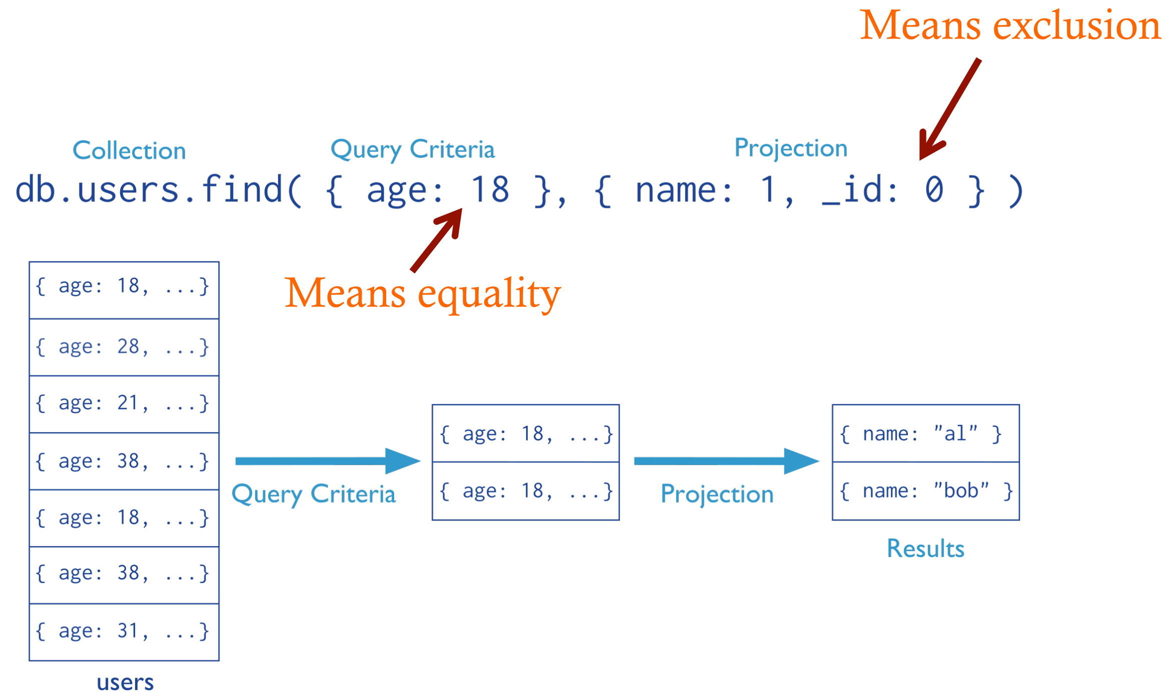
Means inclusion **+**
_id is always automatically included

Equivalent to in SQL:

```
SELECT _id, name, address
FROM   users
WHERE  age > 18
LIMIT  5
```

← projection
← table
← select criteria
← cursor modifier

Find(): Exclude Fields



Cannot mix "inclusion & exclusion" in the same operator except for ***_id***

Find() More Examples

Report all documents in the "inventory" collection

```
db.inventory.find( )
```

```
db.inventory.find( {} )
```

Equivalent to in SQL:

```
Select *
From inventory;
```

Report all documents in the "inventory" collection
Where type = 'food' or 'snacks'

```
db.inventory.find(
      { type: { $in: [ 'food', 'snacks' ] } }
)
```

Equivalent to in SQL:

```
Select *
From inventory
Where type in
      ('food', 'snacks');
```

Find(): AND & OR

AND Semantics

```
db.inventory.find( { type: 'food', price: { $lt: 9.95 } } )
```

OR Semantics

```
db.inventory.find(
    {
      $or: [ { qty: { $gt: 100 } }, { price: { $lt: 9.95 } } ]
    }
)
```

AND + OR Semantics

```
db.inventory.find(
    {
      type: 'food',
      $or: [ { qty: { $gt: 100 } }, { price: { $lt: 9.95 } } ]
    }
)
```

Type = ‘food’ and (qty > 100 or price < 9.95)

$AND

Example	
`{ $and: [ 1, "green" ] }`	**True**
`{ $and: [ ] }`	**True**
`{ $and: [ [ null ], [ false ], [ 0 ] ] }`	**True**
`{ $and: [ null, true ] }`	**False**
`{ $and: [ 0, true ] }`	**False**

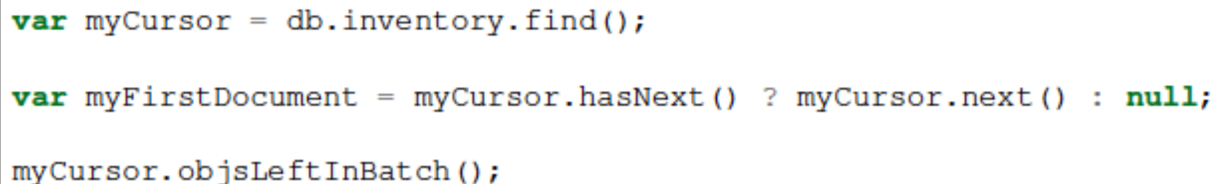
Any thing is true except 0 (for numbers), Null (for objects).

Arrays evaluate to True

Queries Return Cursors

- All queries return a the results in a cursor
- If not assigned to a variable ➔ Printed to screen

  - Results are stored in a cursor
  - Many operators on top of that to manipulate the cursor

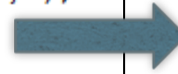


```
var myCursor = db.inventory.find();

var myFirstDocument = myCursor.hasNext() ? myCursor.next() : null;

myCursor.objsLeftInBatch();
```

Cursor's Methods:
http://docs.mongodb.org/manual/reference/method/js-cursor/

Cursor Manipulation

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

myCursor
```



Dumps the content to screen (1st 20 document)

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

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

Explicitly iterate over each document

Shortcuts for iterations

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

myCursor.forEach(printjson);
```

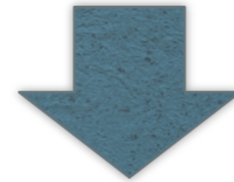
```
var myCursor = db.inventory.find( { type: 'food' } );
var documentArray = myCursor.toArray();
var myDocument = documentArray[3];
```

Querying Complex Types

Querying Complex Types

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "height_cm": 167.6,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

Documents can be complex, E.g., (Arrays, embedded documents, any nesting of these, many levels)



Queries get complex too !!!

Array Manipulation (Exact Match)

```
{ _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] }
{ _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] }
{ _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 8 ] }
```

```
db.inventory.find( { ratings: [ 5, 8, 9 ] } )
```

The operation returns the following document:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
```

Array Manipulation (Search By Element)

```
{ _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] }
{ _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] }
{ _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 8 ] }
```

```
db.inventory.find( { ratings: 5 } )
```

The operation returns the following documents:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 6, "type" : "food", "item" : "bbb", "ratings" : [ 5, 9 ] }
{ "_id" : 7, "type" : "food", "item" : "ccc", "ratings" : [ 9, 5, 8 ] }
```

Notice: if a document has "ratings" as an Integer field = 5, it will be returned

Array Manipulation (Search By Position)

```
{ _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] }
{ _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] }
{ _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 8 ] }
```

```
db.inventory.find( { 'ratings.0': 5 } )
```

The operation returns the following documents:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 6, "type" : "food", "item" : "bbb", "ratings" : [ 5, 9 ] }
```

Notice: if a document has “ratings” as an Integer field = 5, it *will not be* returned

Array Manipulation ($elemMatch)

```
// Document 1
{"foo" : [
    {
      "shape" : "square",
      "color" : "purple",
      "thick" : false
    },
    {
      "shape" : "circle",
      "color" : "red",
      "thick" : true
    }
]}

// Document 2
{"foo" : [
    {
      "shape" : "square",
      "color" : "red",
      "thick" : true
    },
    {
      "shape" : "circle",
      "color" : "purple",
      "thick" : false
    }
]}
```

Who contains purple square?

db.foo.find({"foo.shape": "square", "foo.color": "purple"})

Returns both

db.foo.find({foo: {"shape": "square", "color": "purple"} })

Returns none

db.foo.find({foo: {"$elemMatch": {shape: "square", color: "purple"}}})

Returns Document 1

OK

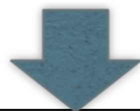
Another Example

```
{ _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] }
{ _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] }
{ _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 8 ] }
```

```
db.inventory.find( { ratings: { $elemMatch: { $gt: 5, $lt: 9 } } } )
```

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 7, "type" : "food", "item" : "ccc", "ratings" : [ 9, 5, 8 ] }
```

```
db.inventory.find( { ratings: { $gt: 5, $lt: 9 } } )
```



```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 6, "type" : "food", "item" : "bbb", "ratings" : [ 5, 9 ] }
{ "_id" : 7, "type" : "food", "item" : "ccc", "ratings" : [ 9, 5, 8 ] }
```

Embedded Object Matching (Exact doc Matching)

```
{
        name: "Joe",
        address: {
                city: "San Francisco",
                state: "CA" } ,
        likes: [ 'scuba', 'math', 'literature' ]
}
```

```
db.persons.find( { "address" : { state: "CA" }} )  //don't match

db.persons.find( { "address" : {city: "San Francisco", state: "CA" }} ) // match

db.persons.find( { "address" : {state: "CA" , city: "San Francisco"}} ) //don't match
```

Exact-match (entire object)

Embedded Object Matching (Field Matching)

```
{
        name: "Joe",
        address: {
                city: "San Francisco",
                state: "CA" } ,
        likes: [ 'scuba', 'math', 'literature' ]
}
```

Find the user documents where the address's state = 'CA'

db.persons.find({"address.state" : "CA"})

Using dot notation

Try This

```
{
        name: "Joe",
        address: {
                city: "San Francisco",
                state: "CA" } ,
        likes: [ 'scuba', 'math', 'literature' ]
}
```

Find the user documents where the address's state = 'CA' and City = "San Francisco"

Find the user documents where the address's state = 'CA' Or likes 'Math'

Matching Arrays of Embedded Documents

```
{
  _id: 100,
  type: "food",
  item: "xyz",
  qty: 25,
  price: 2.5,
  ratings: [ 5, 8, 9 ],
  memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing" } ]
}

{
  _id: 101,
  type: "fruit",
  item: "jkl",
  qty: 10,
  price: 4.25,
  ratings: [ 5, 9 ],
  memos: [ { memo: "on time", by: "payment" }, { memo: "delayed", by: "shipping" } ]
}
```

Select all documents where the memos array contains in the 1st element a document written by 'shipping' department

Matching Arrays of Embedded Documents

```
{
  _id: 100,
  type: "food",
  item: "xyz",
  qty: 25,
  price: 2.5,
  ratings: [ 5, 8, 9 ],
  memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing" } ]
}

{
  _id: 101,
  type: "fruit",
  item: "jkl",
  qty: 10,
  price: 4.25,
  ratings: [ 5, 9 ],
  memos: [ { memo: "on time", by: "payment" }, { memo: "delayed", by: "shipping" } ]
}
```

db.inventory.find({ 'memos.0.by': 'shipping' }) // Returns 1st document

Means the 1st element in the array

Matching Arrays of Embedded Documents

```
{
  _id: 100,
  type: "food",
  item: "xyz",
  qty: 25,
  price: 2.5,
  ratings: [ 5, 8, 9 ],
  memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing" } ]
}

{
  _id: 101,
  type: "fruit",
  item: "jkl",
  qty: 10,
  price: 4.25,
  ratings: [ 5, 9 ],
  memos: [ { memo: "on time", by: "payment" }, { memo: "delayed", by: "shipping" } ]
}
```

Select all documents where the memos array contains a document written by 'shipping' department

Matching Arrays of Embedded Documents

```
{
  _id: 100,
  type: "food",
  item: "xyz",
  qty: 25,
  price: 2.5,
  ratings: [ 5, 8, 9 ],
  memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing" } ]
}

{
  _id: 101,
  type: "fruit",
  item: "jkl",
  qty: 10,
  price: 4.25,
  ratings: [ 5, 9 ],
  memos: [ { memo: "on time", by: "payment" }, { memo: "delayed", by: "shipping" } ]
}
```

```
db.inventory.find( { 'memos.by': 'shipping' } )   // Returns both documents
```

Means any element in the array

Matching Arrays of Embedded Documents: Multiple Conditions

```
{
  _id: 100,
  type: "food",
  item: "xyz",
  qty: 25,
  price: 2.5,
  ratings: [ 5, 8, 9 ],
  memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing" } ]
}

{
  _id: 101,
  type: "fruit",
  item: "jkl",
  qty: 10,
  price: 4.25,
  ratings: [ 5, 9 ],
  memos: [ { memo: "on time", by: "payment" }, { memo: "delayed", by: "shipping" } ]
}
```

Select all documents where the memos array contains a document written by 'shipping' department and the content "on time"

Matching Arrays of Embedded Documents: Multiple Conditions

```
{
  _id: 100,
  type: "food",
  item: "xyz",
  qty: 25,
  price: 2.5,
  ratings: [ 5, 8, 9 ],
  memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing" } ]
}

{
  _id: 101,
  type: "fruit",
  item: "jkl",
  qty: 10,
  price: 4.25,
  ratings: [ 5, 9 ],
  memos: [ { memo: "on time", by: "payment" }, { memo: "delayed", by: "shipping" } ]
}
```

```
db.inventory.find(
   {
     memos:
       {
          $elemMatch:
            {
               memo: 'on time',
               by: 'shipping'
            }
       }
   }
)
```

Query Operators

- http://docs.mongodb.org/manual/reference/operator/query/

- Comparison Operators
- Logical Operators
- Element Operators
- Evaluation Operators
- Array Operators
- …

Query Operators: Comparison Op

Name	Description
$eq	Matches values that are equal to a specified value.
$gt	Matches values that are greater than a specified value.
$gte	Matches values that are greater than or equal to a specified value.
$lt	Matches values that are less than a specified value.
$lte	Matches values that are less than or equal to a specified value.
$ne	Matches all values that are not equal to a specified value.
$in	Matches any of the values specified in an array.
$nin	Matches none of the values specified in an array.

```
db.inventory.find( { qty: { $gte: 20 } } )
```

```
db.inventory.update(
        { "carrier.fee": { $gte: 2 } },
        { $set: { price: 9.99 } }
)
```

Query Operators: Evaluation Op

Name	Description
`$mod`	Performs a modulo operation on the value of a field and selects documents with a specified result.
`$regex`	Selects documents where values match a specified regular expression.
`$text`	Performs text search.
`$where`	Matches documents that satisfy a JavaScript expression.

$Where Operator

- Passes a ***JavaScript expression or function*** to the query system
- Very flexible in expressing complex conditions
- But it is relatively slow as it evaluates for each document (no indexes)
- Similar to using ***UDF*** in the ***Where*** clause in relational databases

```
db.myCollection.find( { $where: "this.credits == this.debits" } );
db.myCollection.find( { $where: "obj.credits == obj.debits" } );

db.myCollection.find( { $where: function() { return (this.credits == this.debits) } });
db.myCollection.find( { $where: function() { return obj.credits == obj.debits; } } );
```

$Where Operator

- Can combine MongoDB operators with $Where

```
db.myCollection.find( { active: true, $where: "this.credits - this.debits < 0" } );


db.myCollection.find( { active: true,
                        $where: function() { return obj.credits - obj.debits < 0; } } );
```

Is this ***And*** semantics or ***Or*** semantics ???

Collection Modeling

Collection Modeling

- Modeling multiple collections that reference each other

- In Relational DBs ➔ FK-PK Relationships

- In MongoDB, two options
  - Referencing
  - Embedding

FK-PK in Relational DBs

- Create "Students" relation

```
CREATE TABLE Students
        (sid CHAR(20),
         name CHAR(20),
         login CHAR(10),
         age INTEGER,
         gpa REAL);
```

- Create "Courses" relation

```
CREATE TABLE Courses
    (cid Varchar2(20),
     name varchar2(50),
     maxCredits integer,
     graduateFlag char(1));
```

Foreign key

Foreign key

- Create "Enrolled" relation

```
CREATE TABLE Enrolled
        (sid CHAR(20),
         cid Varchar2(20),
        enrollDate date,
         grade CHAR(2));
```

- **Each tuple in "Enrolled" reference a specific student and a specific course**

How to Define FK-PK

- Create "Students" relation

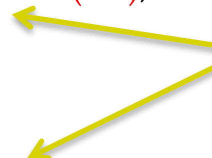
```
CREATE TABLE Students
        (sid CHAR(20) Primary Key,
         name CHAR(20),
         login CHAR(10),
         age INTEGER,
         gpa REAL);
```

- Create "Courses" relation

```
CREATE TABLE Courses
    (cid Varchar2(20) Primary Key,
     name varchar2(50),
     maxCredits  integer,
     graduateFlag char(1));
```

- Create "Enrolled" relation

```
CREATE TABLE Enrolled
        (sid  CHAR(20) Foreign Key References Students (sid),
         X Varchar2(20),
        enrollDate date,
         grade CHAR(2),
 Constraint fk_cid Foreign Key (X) References Courses (cid));
```



Two ways to define the FK constrain while creating a table

FK-PK in Relational DBs

It comes with an enforcement mechanism

- Cannot insert a FK for a non-existing PK
- You cannot delete a PK that has a FK

Enrolled (referencing relation)

sid	cid	grade
53666	Carnatic101	C
53666	Reggae203	B
53650	Topology112	A
53666	History105	B

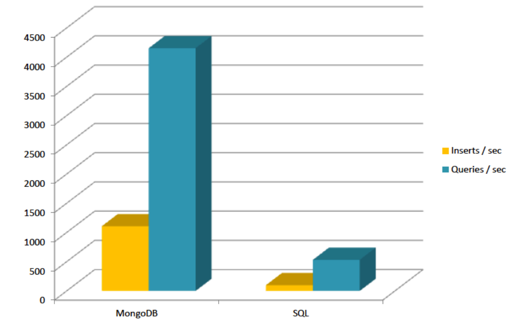
Foreign Key

Students (referenced relation)

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

Primary Key

In MongoDB

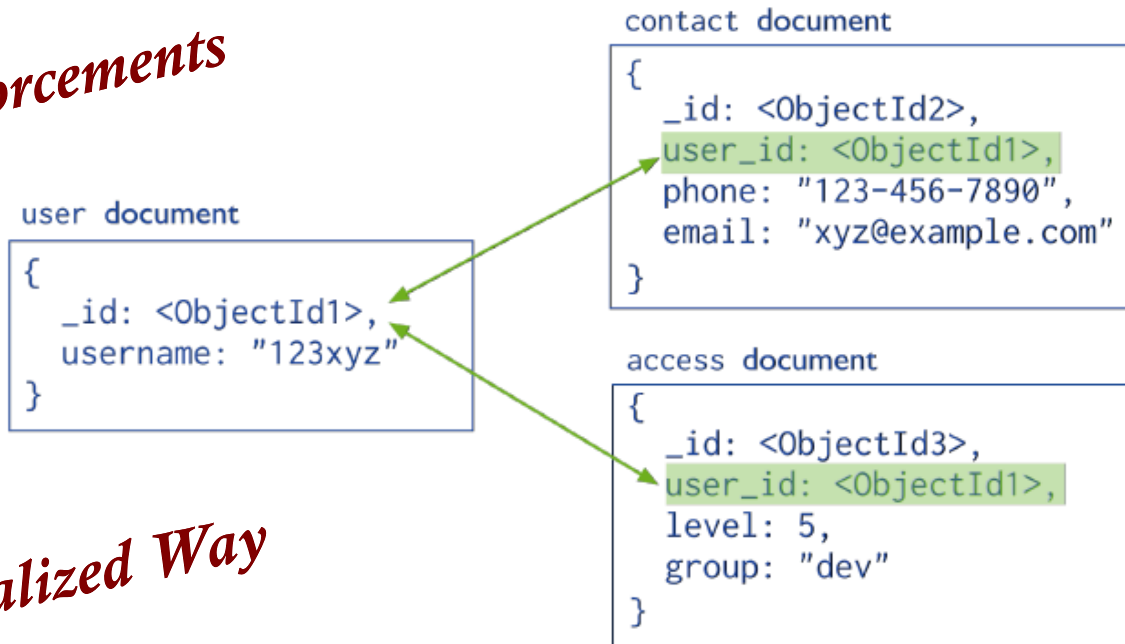


- ***Referencing*** between two collections
  - Use Id of one and put in the other
  - Very similar to FK-PK in Relational DBs
  - **Does not come with enforcement mechanism**

- ***Embedding*** between two collections
  - Put the document from one collection inside the other one

Referencing

No Enforcements



Normalized Way

- Have three collections in the DB: “User”, “Contact”, “Access”
- Link them by _id (or any other field(s))

Embedding

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

Embedded sub-document

Embedded sub-document

De-Normalized Way

- Have one collection in DB: "User"
- The others are embedded inside each user's document

Examples (1)

- "Patron" & "Addresses"

```
{
   _id: "joe",
   name: "Joe Bookreader"
}
```

```
{
   patron_id: "joe",
   street: "123 Fake Street",
   city: "Faketon",
   state: "MA",
   zip: "12345"
}
```

Referencing

- If it is 1-1 relationship
- If usually read the address with the name
- If address document usually does not expand

If most of these hold
➔ better use Embedding

Examples (2)

- "Patron" & "Addresses"

```
{
   _id: "joe",
   name: "Joe Bookreader",
   address: {
               street: "123 Fake Street",
               city: "Faketon",
               state: "MA",
               zip: "12345"
             }
}
```

Embedding

- When you read, you get the entire document at once
- In Referencing ➔ Need to issue multiple queries

Examples (3)

- What if a "Patron" can have many "Addresses"

```
{
   _id: "joe",
   name: "Joe Bookreader"
}
```

```
{
   patron_id: "joe",
   street: "123 Fake Street",
   city: "Faketon",
   state: "MA",
   zip: "12345"
}
```

Referencing

- Do you read them together ➔ Go for Embedding
- Are addresses dynamic (e.g., add new ones frequently)

  ➔ Go for Referencing

Examples (4)

- What if a “Patron” can have many “Addresses”

```
{
   _id: "joe",
   name: "Joe Bookreader",
   addresses: [
                {
                  street: "123 Fake Street",
                  city: "Faketon",
                  state: "MA",
                  zip: "12345"
                },
                {
                  street: "1 Some Other Street",
                  city: "Boston",
                  state: "MA",
                  zip: "12345"
                }
              ]
 }
```

Embedding

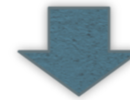
Use array of addresses

Examples (5)

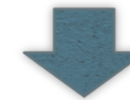
- If addresses are added frequently …

```
{
   _id: "joe",
   name: "Joe Bookreader",
   addresses: [
                {
                  street: "123 Fake Street",
                  city: "Faketon",
                  state: "MA",
                  zip: "12345"
                },
                {
                  street: "1 Some Other Street",
                  city: "Boston",
                  state: "MA",
                  zip: "12345"
                }
              ]
 }
```

This array will expand frequently



Size of "Patron" document increases frequently



May trigger re-locating the document each time *(Bad)*

Document Size and Storage

- Each document needs to be contiguous on disk
- If doc size increases ➔ Document location must change
- If doc location changes ➔ Indexes must be updates ➔ leads to more expensive updates

```
{
   _id: "joe",
   name: "Joe Bookreader",
   addresses: [
                {
                  street: "123 Fake Street",
                  city: "Faketon",
                  state: "MA",
                  zip: "12345"
                },
                {
                  street: "1 Some Other Street",
                  city: "Boston",
                  state: "MA",
                  zip: "12345"
                }
              ]
}
```

- **<u>In a newer version</u>**, each document is allocated a ***power-of-2 bytes*** (the smallest above its size)
- Meaning, the system keeps some space empty for possible expansion

Examples (6)

- **One-to-Many "Book", "Publisher"**
  - A book has one publisher
  - A publisher publishes many books

Referencing is better in this case

- **If embed "Publisher" inside "Book"**
  - Repeating publisher info inside each of its books
  - Very hard to update publisher's info

- **If embed "Book" inside "Publisher"**
  - Book becomes an array (many)
  - Frequently update and increases in size

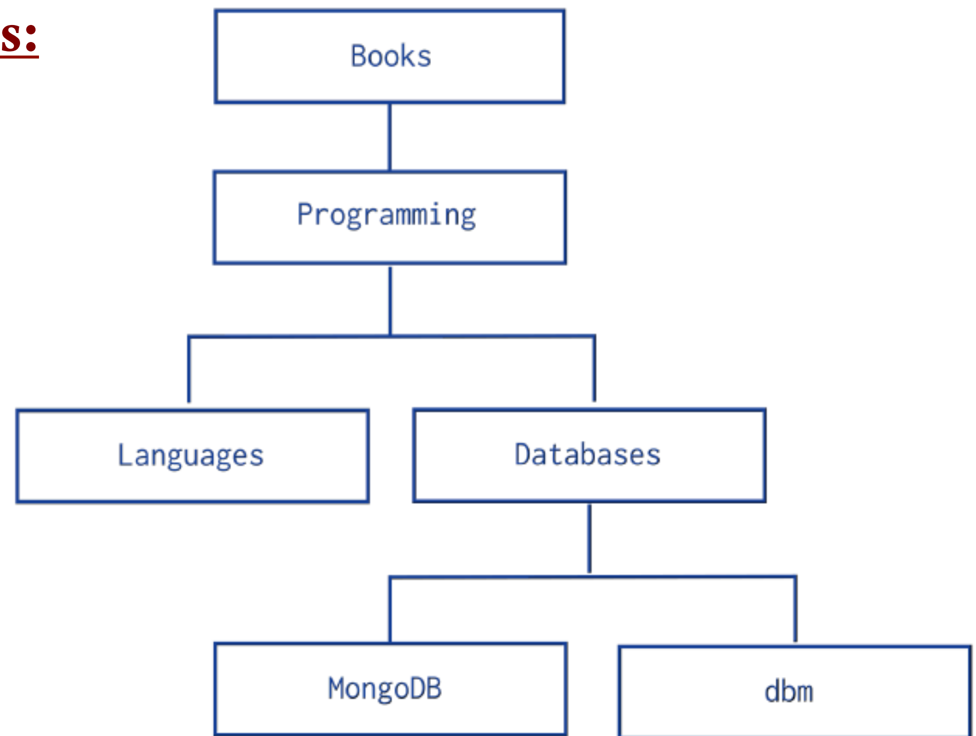
Modeling Tree Structure

Collections with Tree-Like Relationships

- Insert these records while maintaining this tree-like relationship

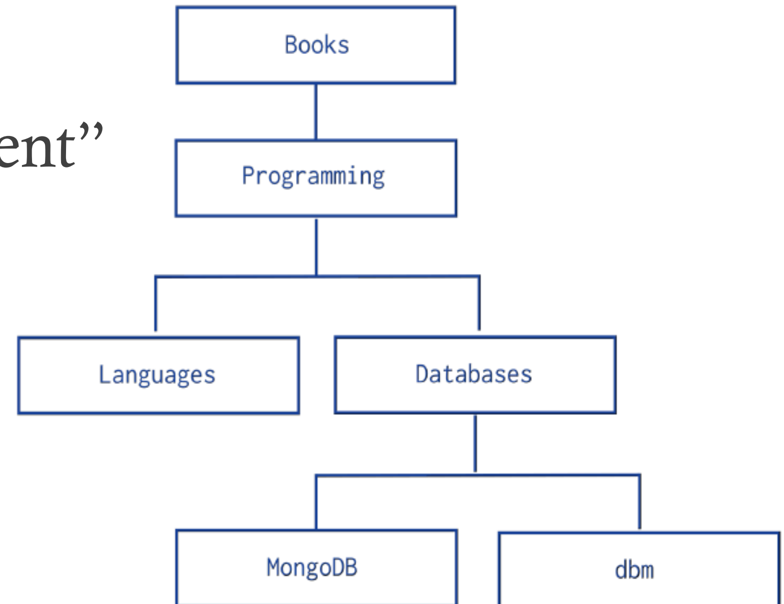
Given one node, answer queries:

- Report the parent node
- Report the children nodes
- Report the ancestors
- Report the descendants
- Report the siblings



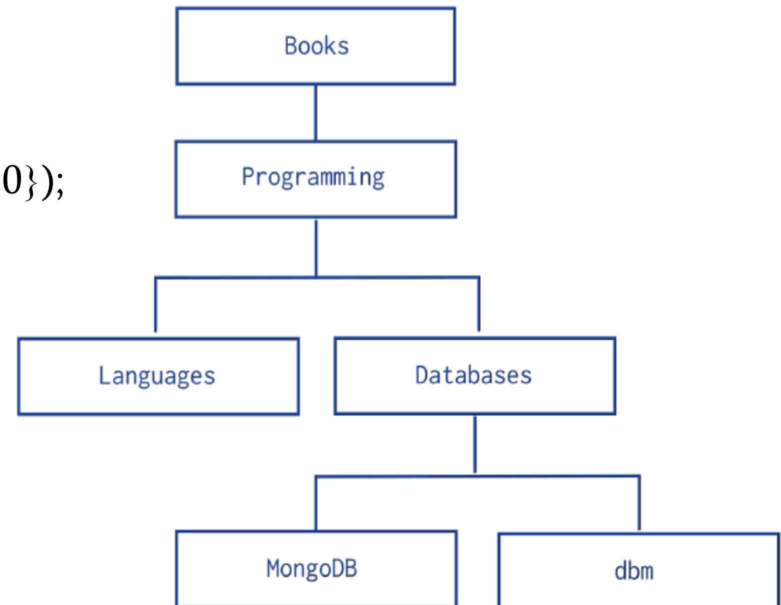
Method 1: Parent References

- Each document has a field "parent"
- Order does not matter




```
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 } )
```

Method 1: Parent References



Q1: Parent of "Programming"

```
db.categories.find( {_id: "Programming"}, {parent: 1, _id: 0});
```

Q2: Siblings of "Databases"

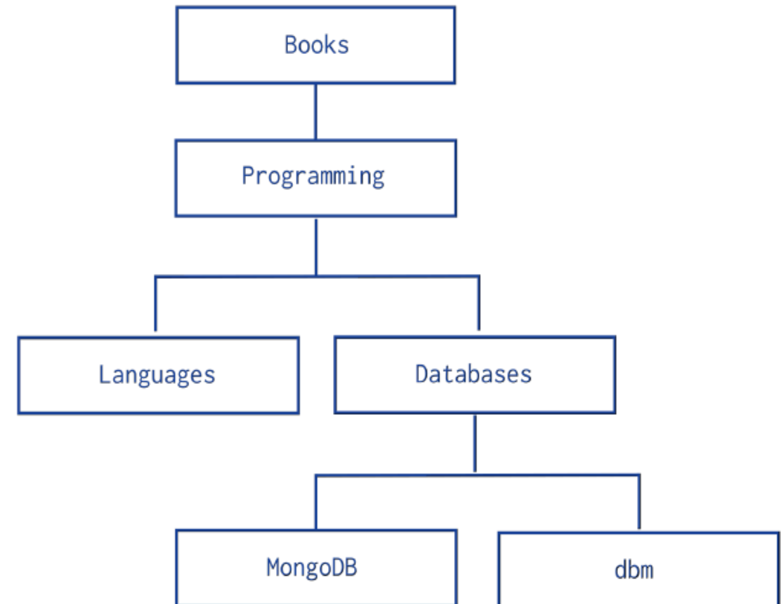
```
var  parentDoc = db.categories.find( {_id: "Databases"});

db.categories.find( {parent:  parentDoc._id,
                     _id: { $ne :"Databases"}     });
```

```
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 } )
```

Method 1: Parent References

Q3: Descendants of "Programming"

Complex…Requires recursive calls




```
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 } )
```

Method 1: Parent References

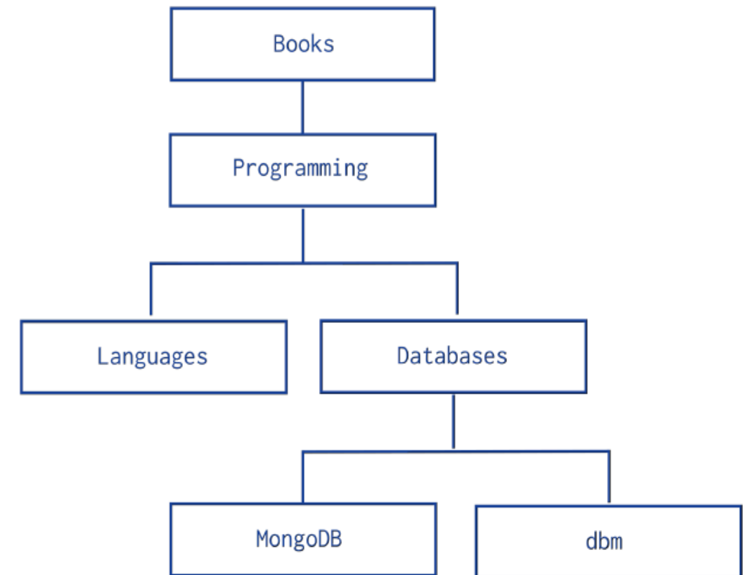
Q3: Descendants of "Programming"

```
var descendants = [];
var stack = [];
var item = db.categories.find({_id: "Programming"});
stack.push(item);
while (stack.length > 0) {
    var current = stack.pop();
    var children =  db.categories.find( {parent: current._id});
    while (children.hasNext() == true) {
        var child = children.next();
        descendants.push(child._id);
        stack.push(child);
    }
}

descendants;
```

Method 1: Parent References

Q4: Ancestors of "MongoDB"

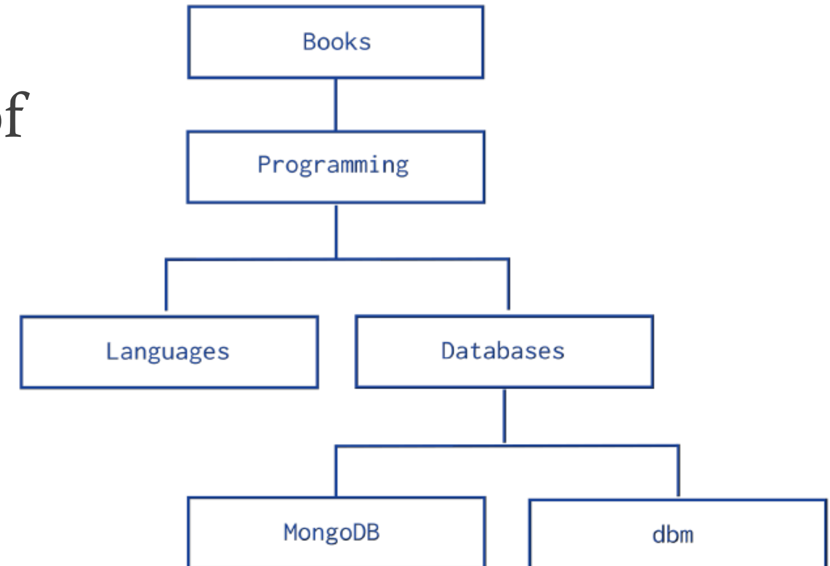
> **Try it yourself….**
> **Should be:**
> "Databases", "Programming", "Books"




```
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 } )
```

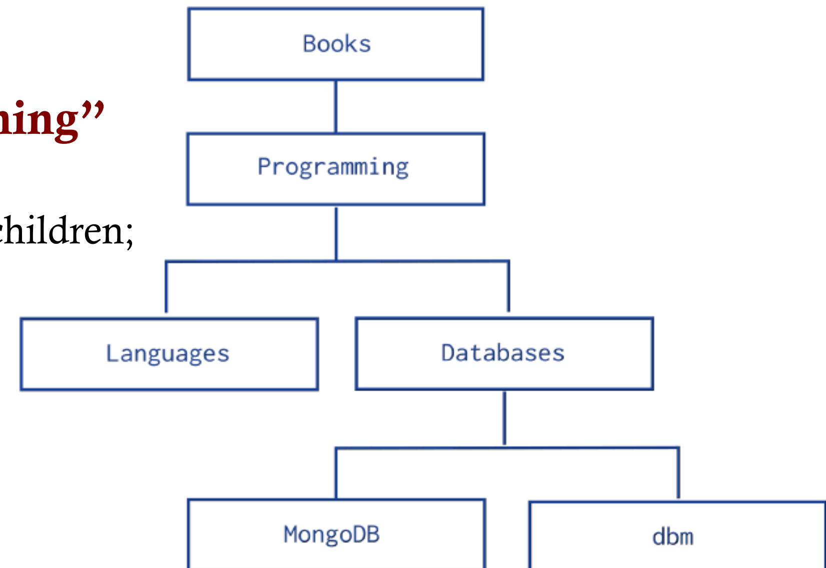
Method 2: Child References

- Each document has an array of immediate children




```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "dbm", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "dbm" ] } )
db.categories.insert( { _id: "Languages", children: [] } )

db.categories.insert( { _id: "Programming", children: [ "Databases", "Languages" ] } )
db.categories.insert( { _id: "Books", children: [ "Programming" ] } )
```

Method 2: Child References



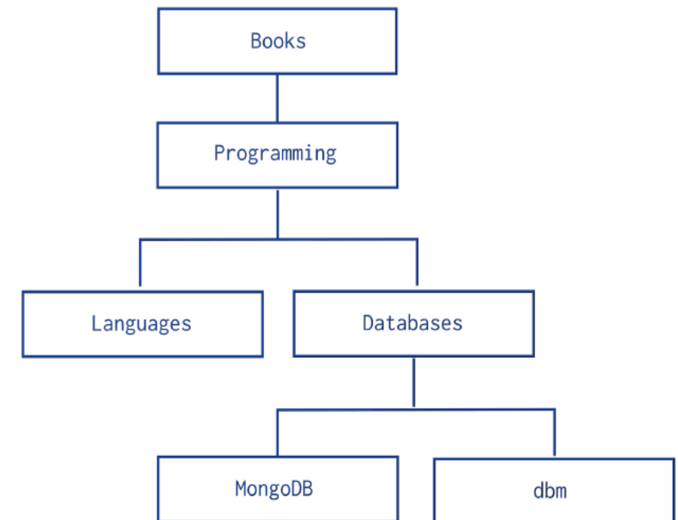
Q1: Get children documents of "Programming"

```
var x = db.categories.findOne({_id: "Programming"}).children;

db.categories.find({_id: {$in: x}});
```

```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "dbm", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "dbm" ] } )
db.categories.insert( { _id: "Languages", children: [] } )

db.categories.insert( { _id: "Programming", children: [ "Databases", "Languages" ] } )
db.categories.insert( { _id: "Books", children: [ "Programming" ] } )
```

Method 2: Child References

Q2: Ancestors of "MongoDB"




```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "dbm", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "dbm" ] } )
db.categories.insert( { _id: "Languages", children: [] } )

db.categories.insert( { _id: "Programming", children: [ "Databases", "Languages" ] } )
db.categories.insert( { _id: "Books", children: [ "Programming" ] } )
```

Method 2: Child References

Q2: Ancestors of “MongoDB”

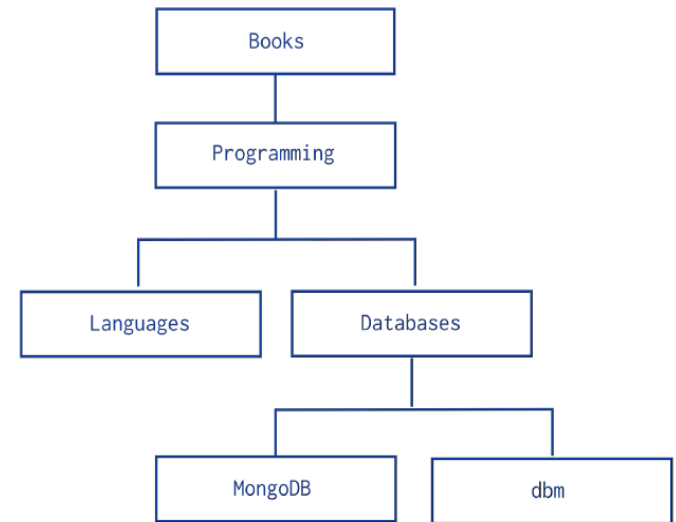
```
var results=[];
var parent = db.categories.findOne({children: "MongoDB"});
while(parent){
	print({Message: "Going up one level…"});
	results.push(parent._id);
	parent = db.categories.findOne({children: parent._id});
}


results;
```

Method 2: Child References

Q3: descendants of “Books”

Try it yourself….

Should be all nodes




```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "dbm", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "dbm" ] } )
db.categories.insert( { _id: "Languages", children: [] } )

db.categories.insert( { _id: "Programming", children: [ "Databases", "Languages" ] } )
db.categories.insert( { _id: "Books", children: [ "Programming" ] } )
```

Other Methods

- **Several other methods:**
  - Include both parent and children
  - Include Ancestors
  - Include root-to-node path