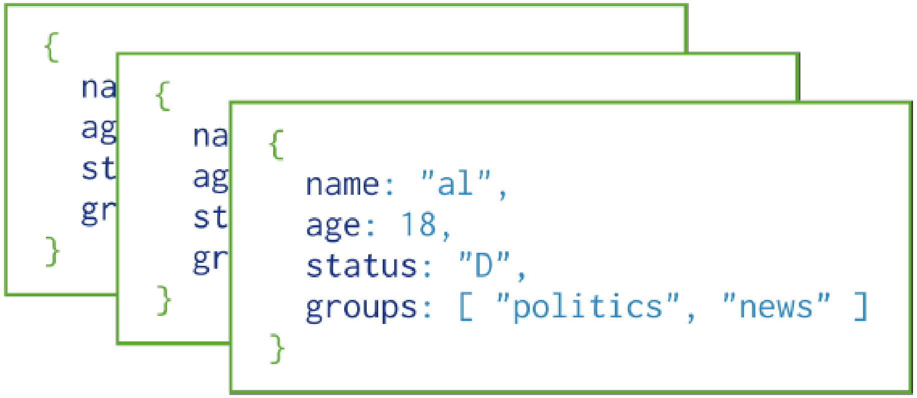
Spatial and text indexes

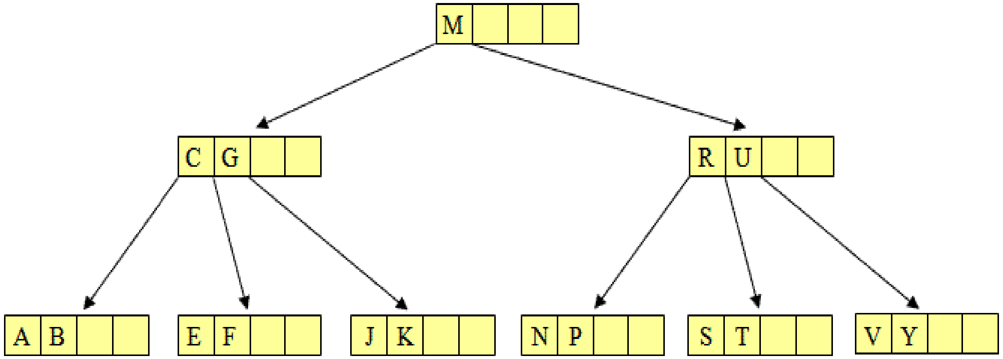
Indexes

- MongoDB uses B-Tree indexes
- Can build the index on any field of the document
- Skips documents that do not have the indexed field (Sparse index)

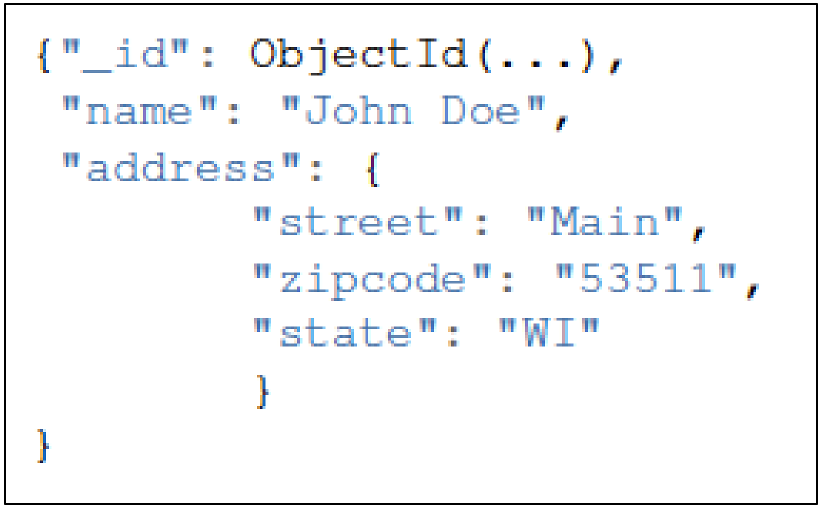


```
{
   name: "al",
   age: 18,
   status: "D",
   groups: [ "politics", "news" ]
}
```

Collection



Examples



```
{"_id": ObjectId(...),
 "name": "John Doe",
 "address": {
            "street": "Main",
            "zipcode": "53511",
            "state": "WI"
            }
}
```

Field Level

db.people.createIndex(“name”: 1)

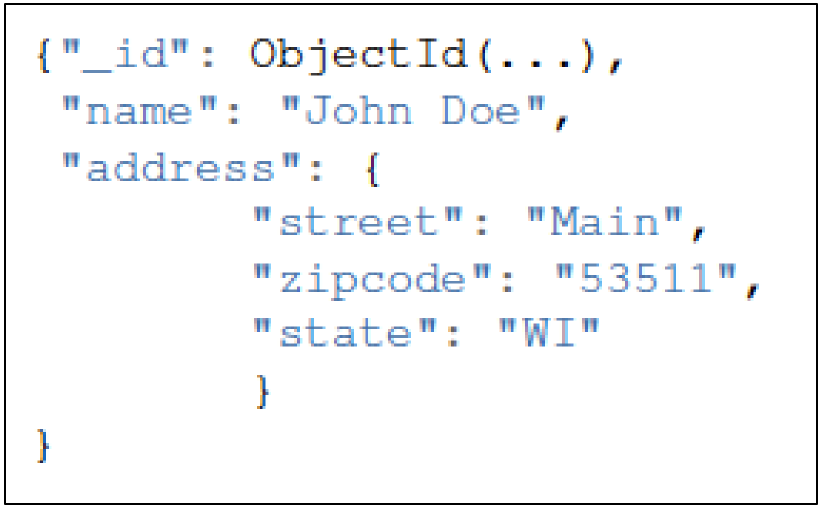
Sub-Field Level

db.people.createIndex(“address.zipcode”: 1)

db.people.createIndex(“address”: 1)

Embedded document Level
(equality search only)

Examples



```
{"_id": ObjectId(...),
 "name": "John Doe",
 "address": {
        "street": "Main",
        "zipcode": "53511",
        "state": "WI"
        }
}
```

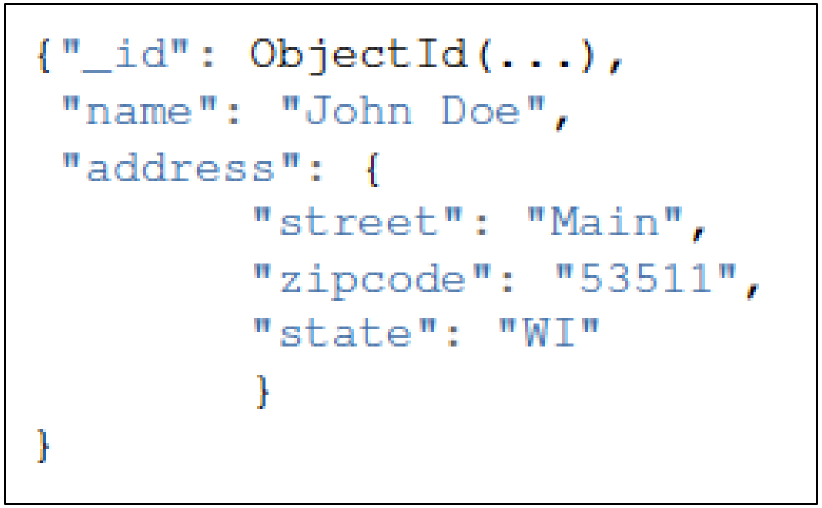
Compound-Field Index

db.people.createIndex({“name”: 1, “_id”: -1})

db.people.find(“_id”: 1000})

Index cannot answer this query (must have a predicate on “name”)

Index Creation Options

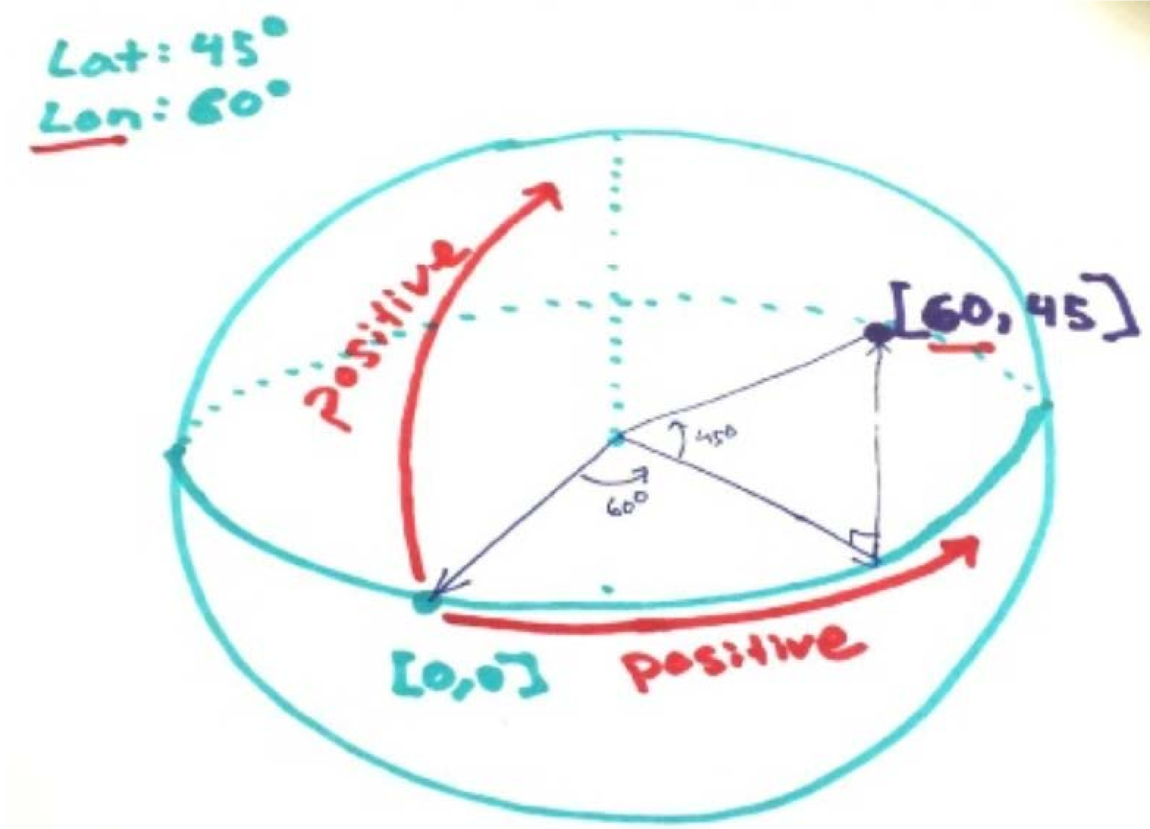


```
{"_id": ObjectId(...),
 "name": "John Doe",
 "address": {
        "street": "Main",
        "zipcode": "53511",
        "state": "WI"
        }
}
```

db.people.createIndex({“name”: 1, “_id”: -1},
{“background: True”, “Sparse”: True,
“unique”: True})

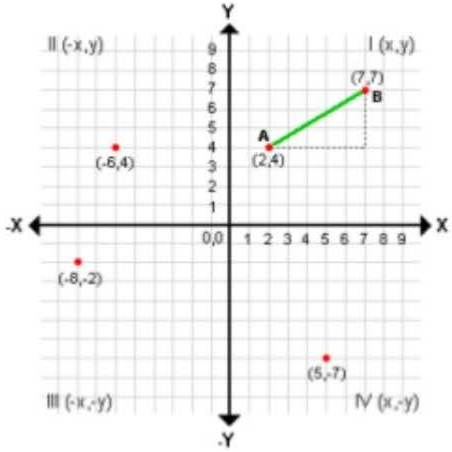
Geospatial indexes

[Longitude, Latitude]



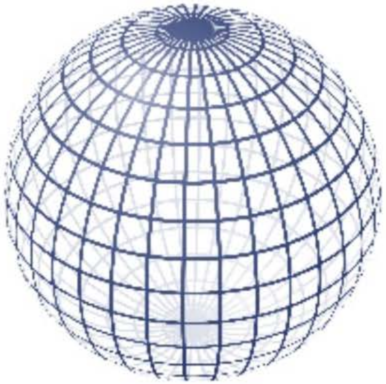
Surface type

Flat



2d Indexes

Spherical

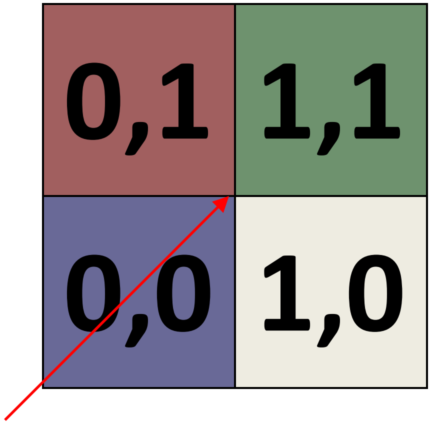


2dsphere Indexes

Quad Trees

- Split on *all* (two) dimensions at each level
- Split key space into equal size partitions (quadrants)
- Add a new node by adding to a leaf, and, if the leaf is already occupied, split until only one node per leaf

quadrant



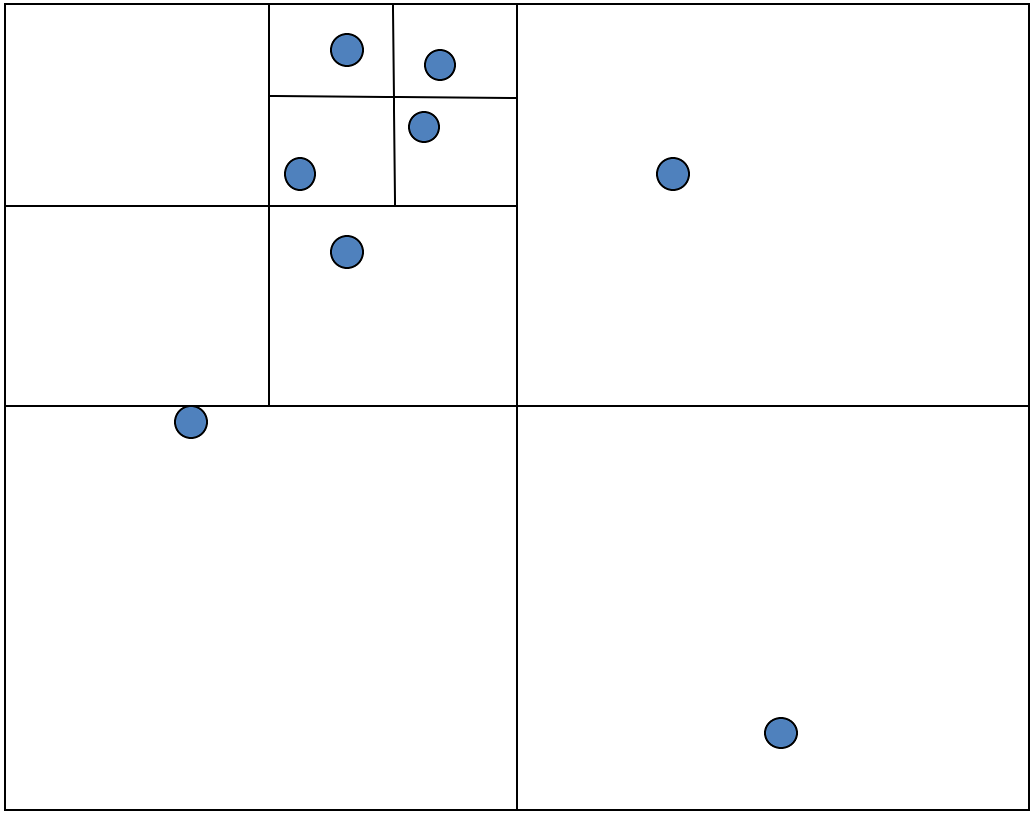
Center

quad tree node

	keys		value	
Center:	x		y	
Quadrants:	0,0	1,0	1,1	0,1

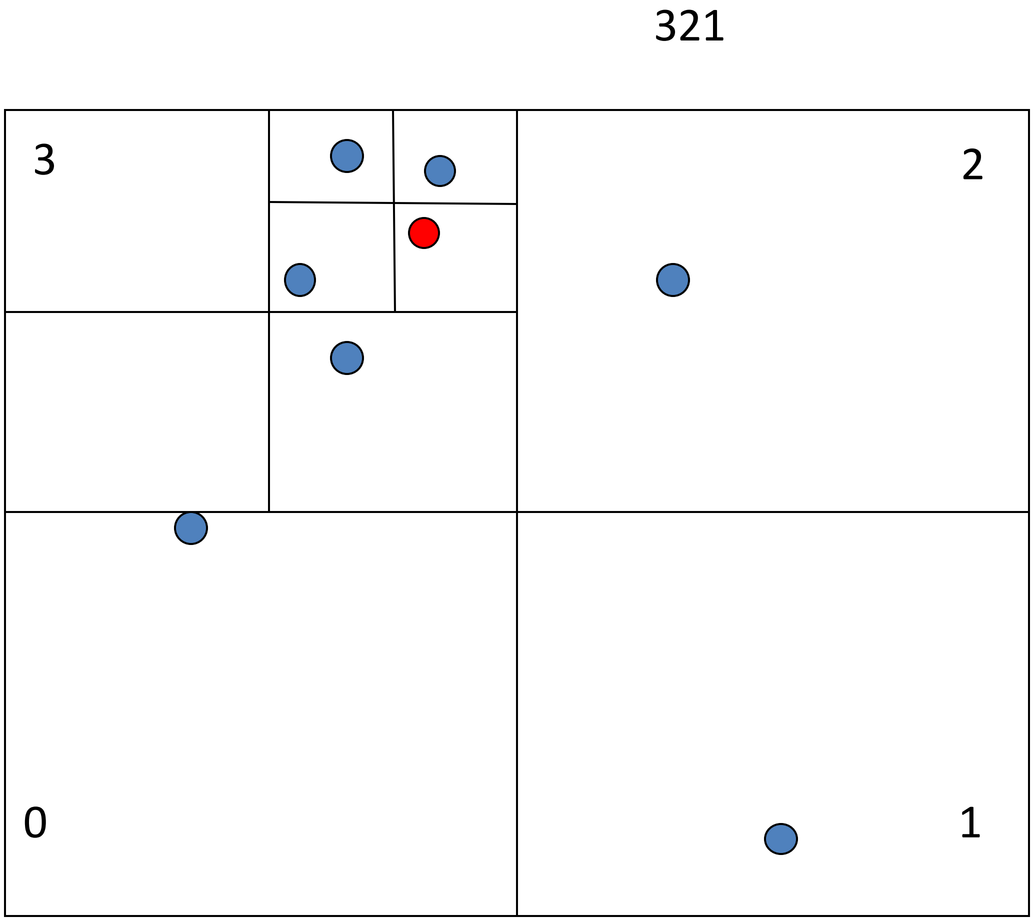
Quadtree

Simplest spatial structure on Earth !



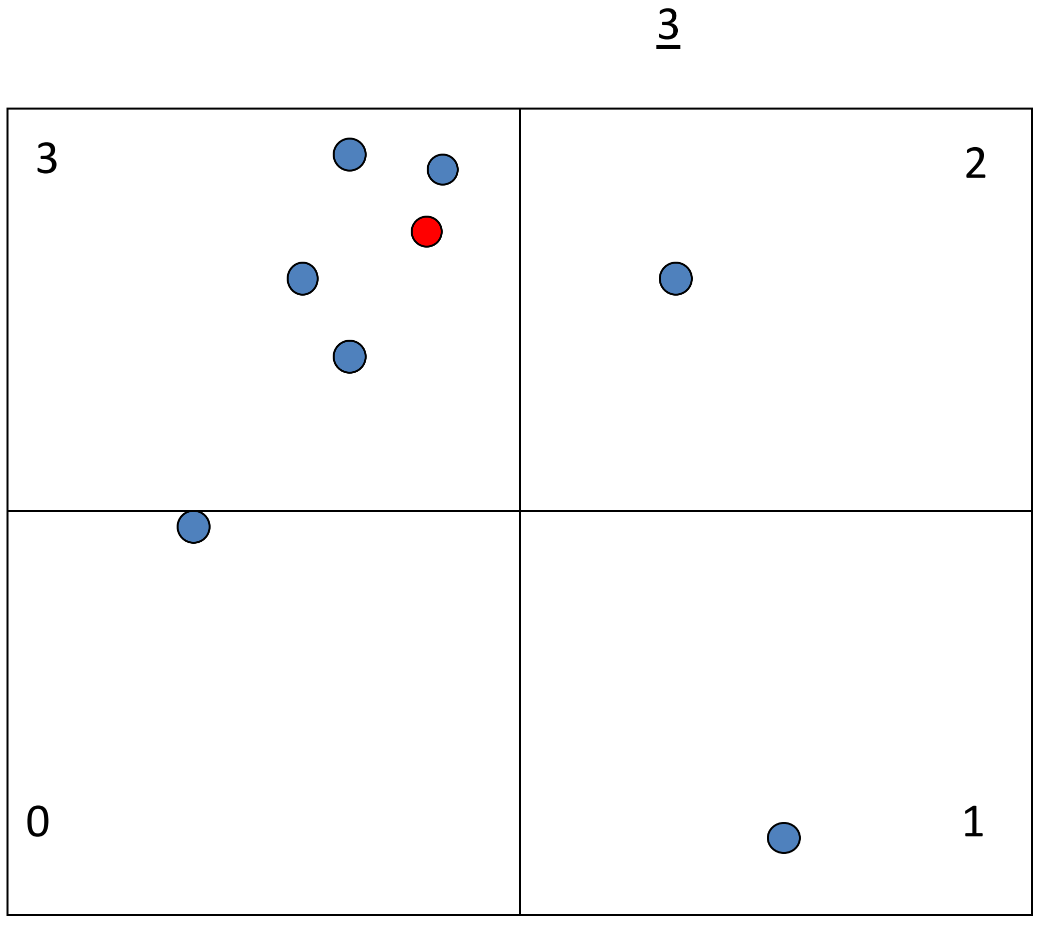
Quadtree

Simplest spatial structure on Earth !



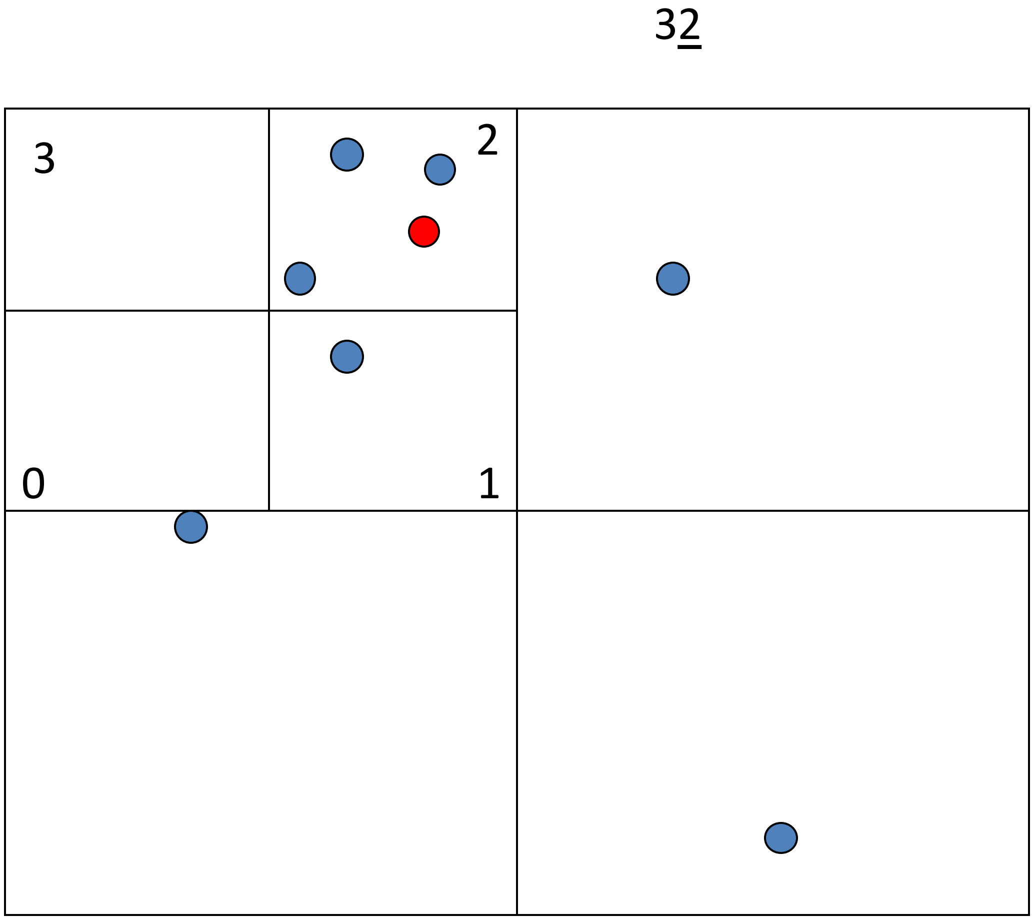
Quadtree

Simplest spatial structure on Earth !



Quadtree

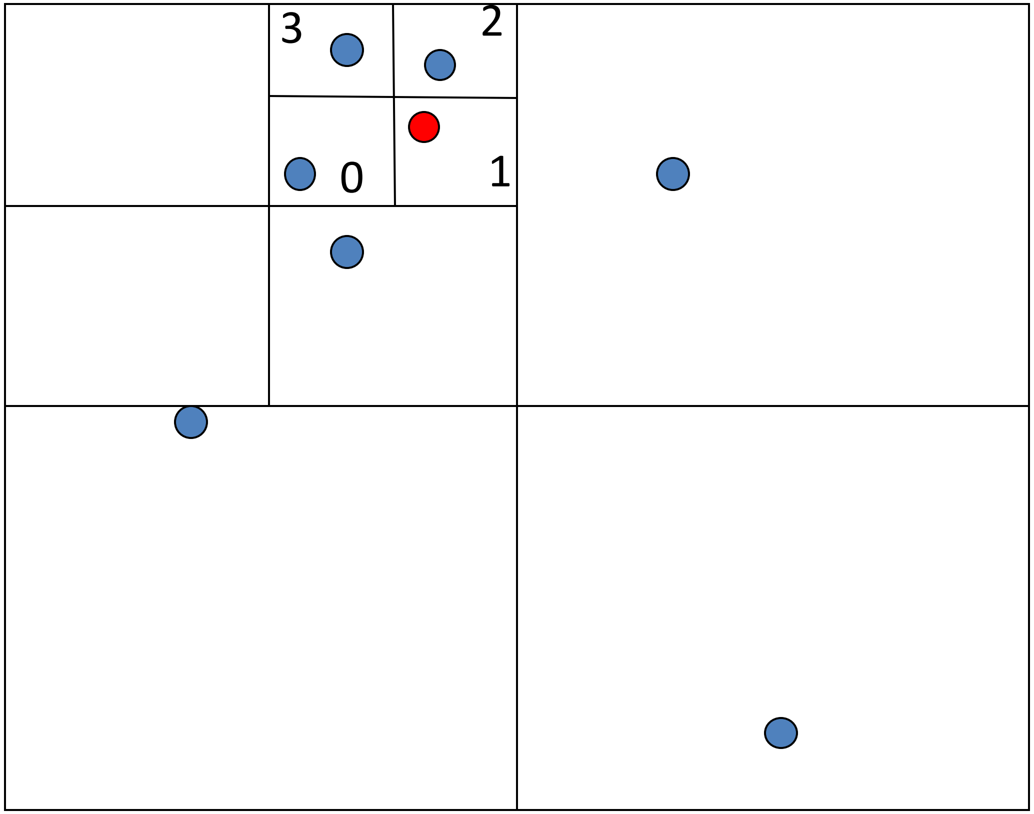
Simplest spatial structure on Earth !



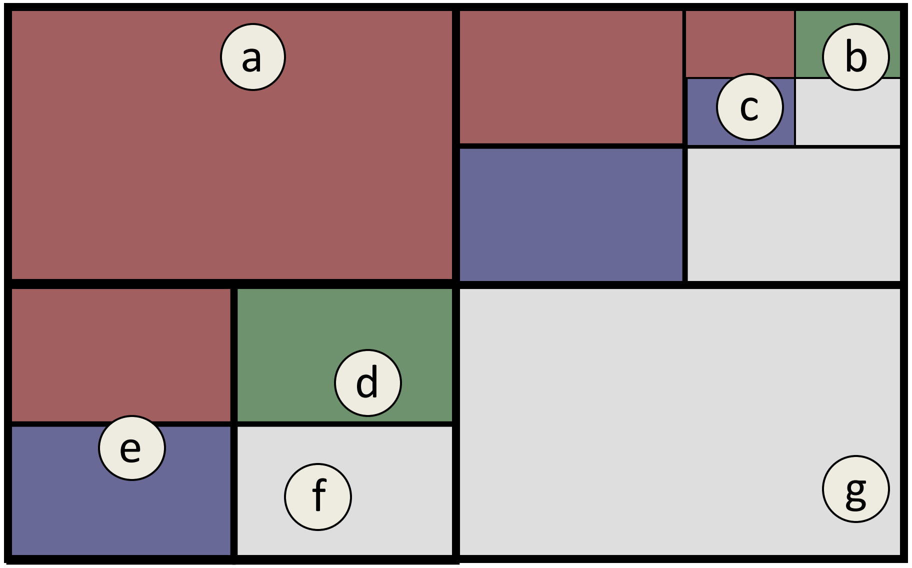
Quadtree

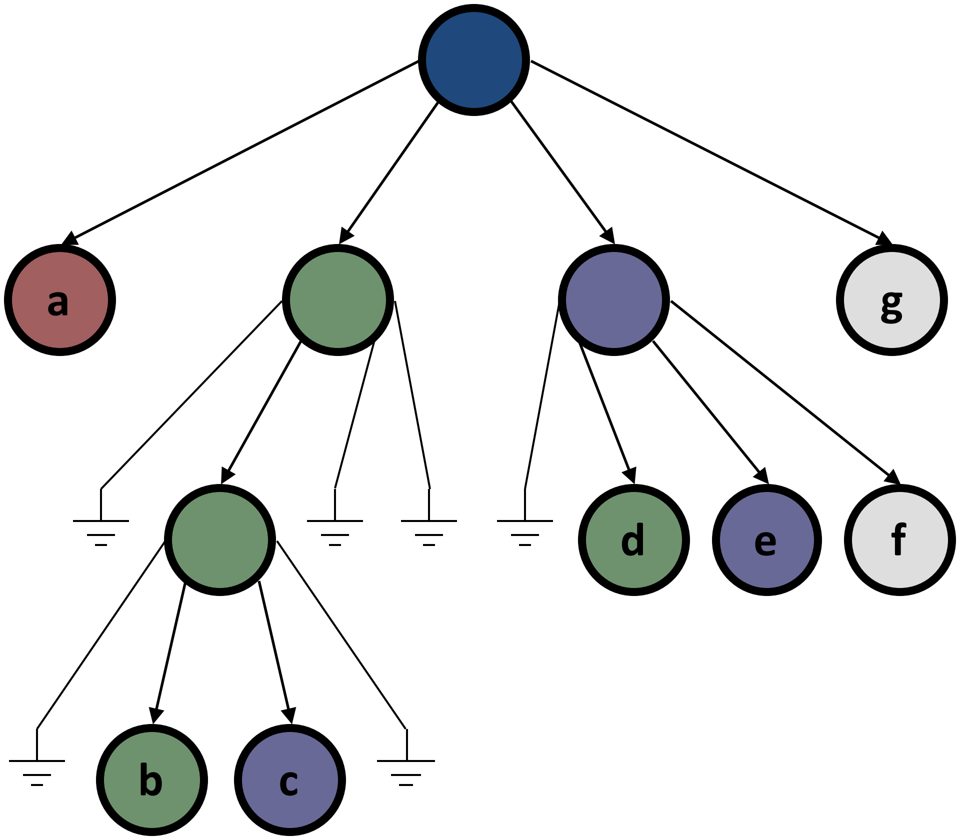
Simplest spatial structure on Earth !

321



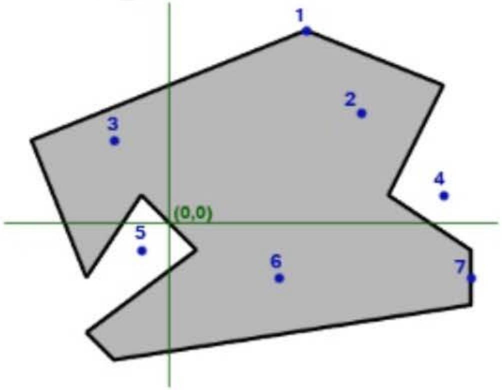
Quadtree Example





Geospatial operators

$geoWithin

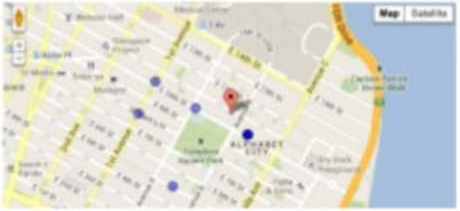


No index, 2d, 2dsphere

$geoIntersects

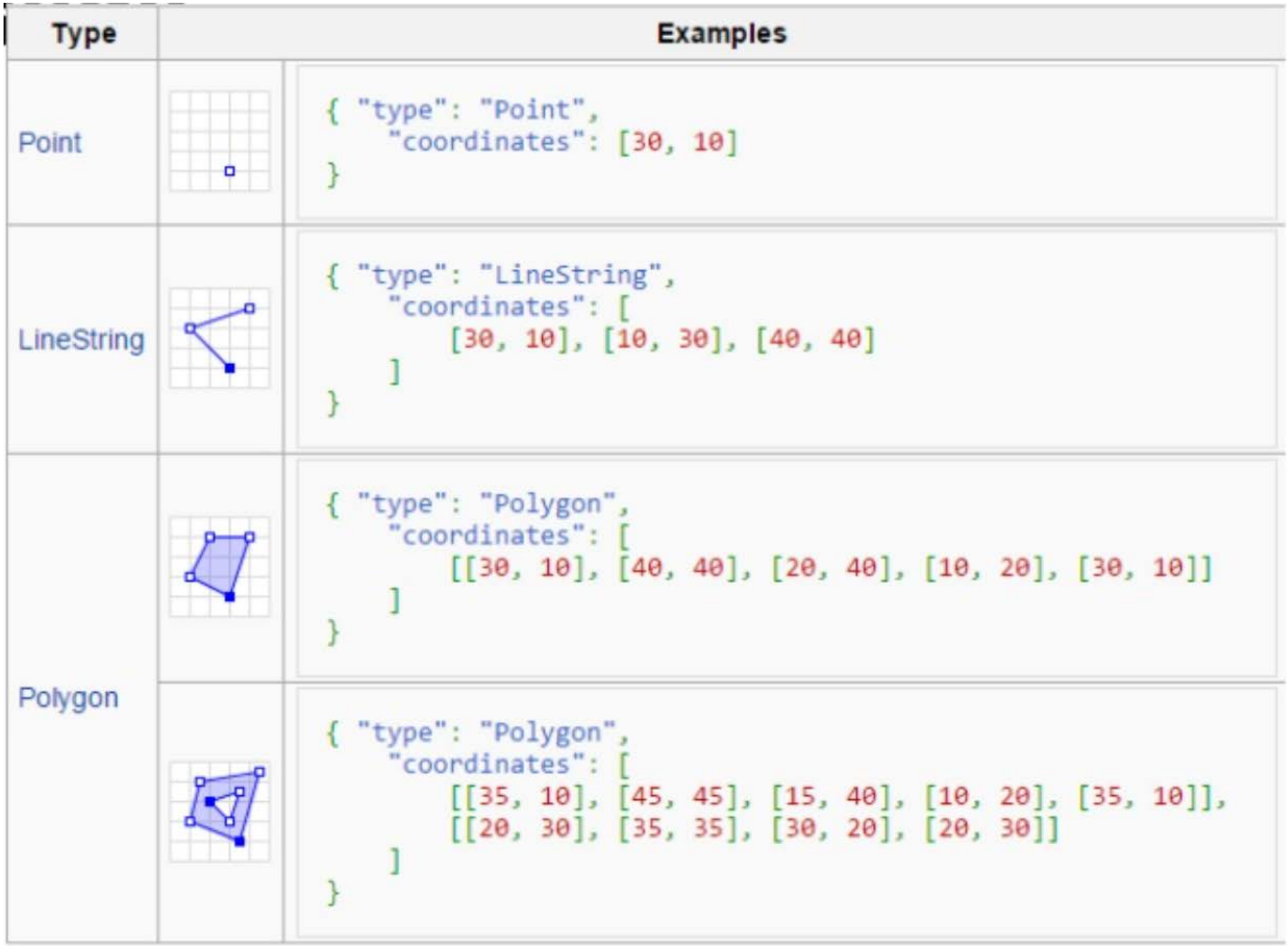
Index required - 2dsphere

$near/$nearSphere



2d, 2dsphere

GeoJSON



Type	Examples	

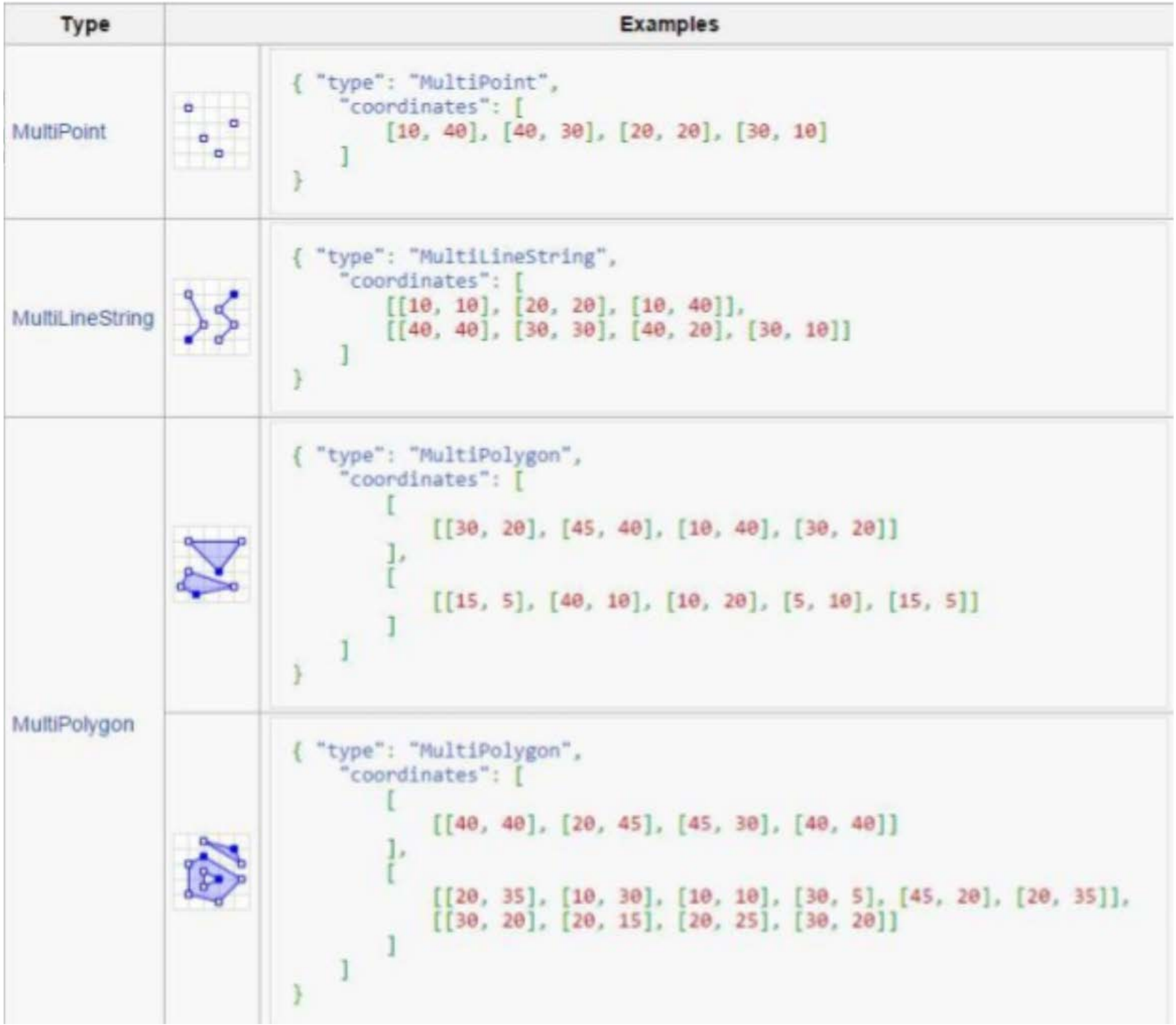
Point		`{ "type": "Point",`
		`    "coordinates": [30, 10]`
		`}`

LineString		`{ "type": "LineString",`
		`    "coordinates": [`
		`        [30, 10], [10, 30], [40, 40]`
		`    ]`
		`}`

Polygon		`{ "type": "Polygon",`
		`    "coordinates": [`
		`        [[30, 10], [40, 40], [20, 40], [10, 20], [30, 10]]`
		`    ]`
		`}`

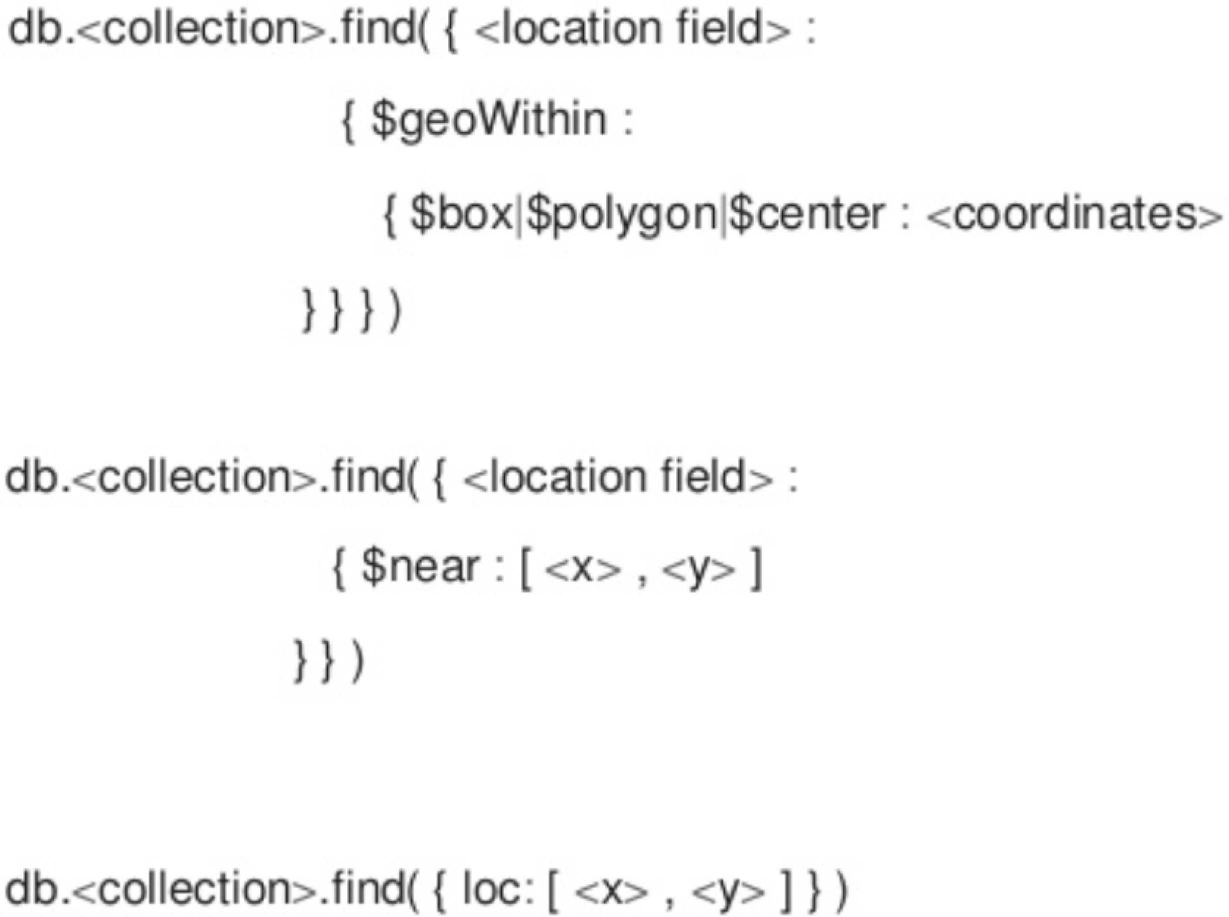
		`{ "type": "Polygon",`
		`    "coordinates": [`
		`        [[35, 10], [45, 45], [15, 40], [10, 20], [35, 10]],`
		`        [[20, 30], [35, 35], [30, 20], [20, 30]]`
		`    ]`
		`}`

GeoJSON



Type	Examples	
MultiPoint		{ "type": "MultiPoint", "coordinates": [[10, 40], [40, 30], [20, 20], [30, 10]] }
MultiLineString		{ "type": "MultiLineString", "coordinates": [[[10, 10], [20, 20], [10, 40]], [[40, 40], [30, 30], [40, 20], [30, 10]]] }
MultiPolygon		{ "type": "MultiPolygon", "coordinates": [[[[30, 20], [45, 40], [10, 40], [30, 20]]], [[[15, 5], [40, 10], [10, 20], [5, 10], [15, 5]]]] }
		{ "type": "MultiPolygon", "coordinates": [[[[40, 40], [20, 45], [45, 30], [40, 40]]], [[[20, 35], [10, 30], [10, 10], [30, 5], [45, 20], [20, 35]], [[30, 20], [20, 15], [20, 25], [30, 20]]]] }

2d Query



```
db.<collection>.find( { <location field> :
                     { $geoWithin :
                        { $box|$polygon|$center : <coordinates>
                   } } } )


db.<collection>.find( { <location field> :
                     { $near : [ <x> , <y> ]
                   } } )



db.<collection>.find( { loc: [ <x> , <y> ] } )
```

Geospatial indexing zips collection

- db.zips.createIndex({loc: “2d”})
- db.zips.find ({loc: { $geoWithin: {$box: [[-73,42.5], [-72, 43]] } } })
- db.zips.find ({loc: { $geoWithin: {$center: [[-73,42.5], 10] } } })
- db.zips.find ({loc: { $near: [-73,42.5] } })

Text Indexes

- Over fields that are strings or array of strings
- Index is used when using ***$text*** search operator
- Only one index on the collection
  - But it can include multiple fields

One field

```
db.collection.createIndex({content: "text”});
```

Two fields

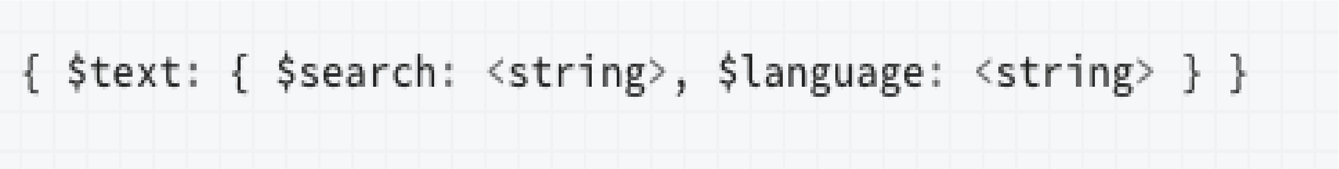
```
db.collection.createIndex({subject: "text”,content: "text”});
```

All text fields

```
db.collection.createIndex({”$**": "text”});
```

$Text

Text search in mongoDB (Exact match)
Uses a text index and searches the indexed fields



```
{ $text: { $search: <string>, $language: <string> } }
```

```
db.articles.find( { $text: { $search: "coffee" } } )
```

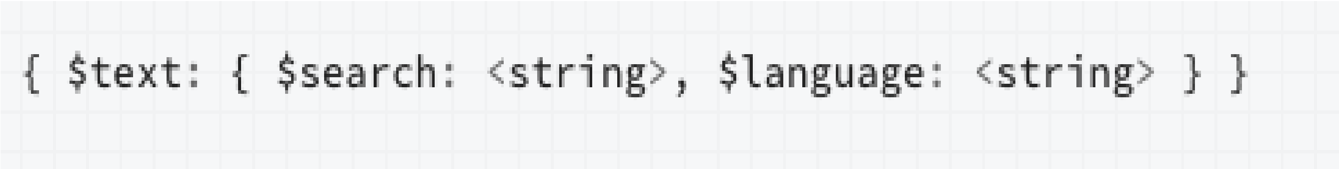
Search for “coffee” in the indexed field(s)

```
db.articles.find( { $text: { $search: "bake coffee cake" } } )
```

Apply “OR” semantics

$Text

Text search in mongoDB
Uses a text index and searches the indexed fields



```
{ $text: { $search: <string>, $language: <string> } }
```

```
db.articles.find( { $text: { $search: "\"coffee cake\"" } } )
```

Treated as one sentence

```
db.articles.find( { $text: { $search: "bake coffee -cake" } } )
```

“bake” or “coffee” but not “cake”

$Text Score

$Text returns a score for each matching document
Score can be used in your query

```
db.articles.find(
   { $text: { $search: "cake" } },
   { score: { $meta: "textScore" } }
).sort( { score: { $meta: "textScore" } } ).limit(3)
```

For regular expression match use $regex operator

City_inspections examples

- db.city_inspections.createIndex({”$**": "text”})

- db.city_inspections.find({ $text: { $search: "food deli" } })

- db.city_inspections.find({ $text: { $search: "\"food deli\"" } })

- db.city_inspections.find({ $text: { $search: "grocery -cigarette" } })

- db.city_inspections.find(
          { $text: { $search: “passed" } },
          { score: { $meta: "textScore" } }
).sort({ score: { $meta: "textScore" } }).limit(3)

Collection Modeling

Collection Modeling

Modeling multiple collections that reference each other

In Relational DBs ➔ FK-PK Relationships

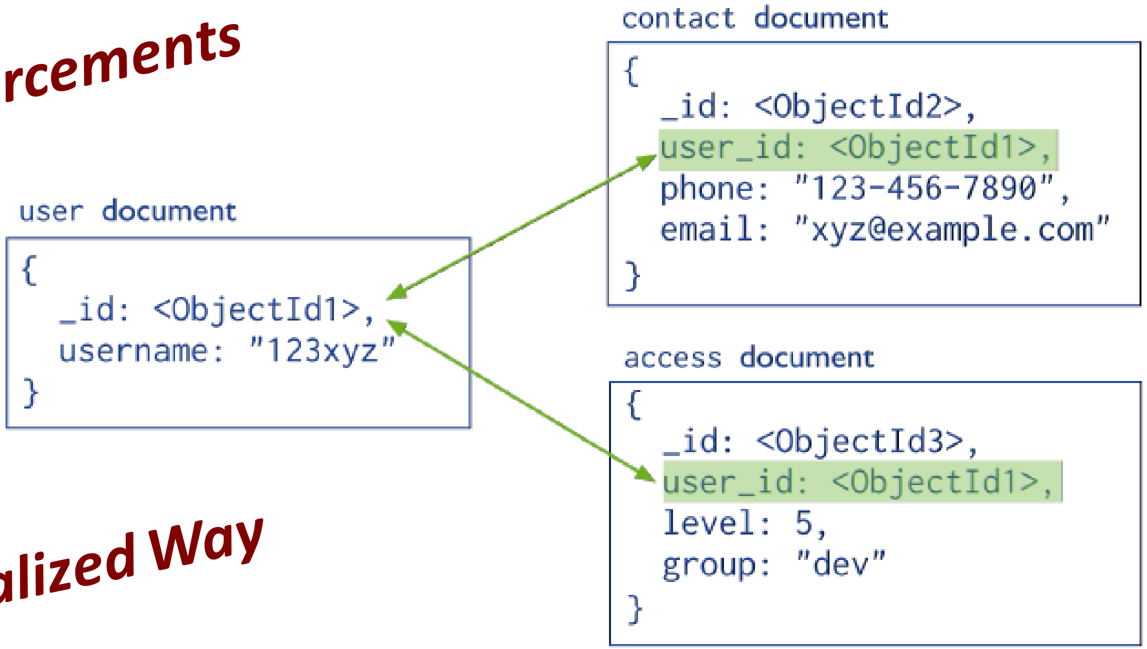
In MongoDB, two options:

- ***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



user document

```
{
  _id: <ObjectId1>,
  username: "123xyz"
}
```

contact document

```
{
  _id: <ObjectId2>,
  user_id: <ObjectId1>,
  phone: "123-456-7890",
  email: "xyz@example.com"
}
```

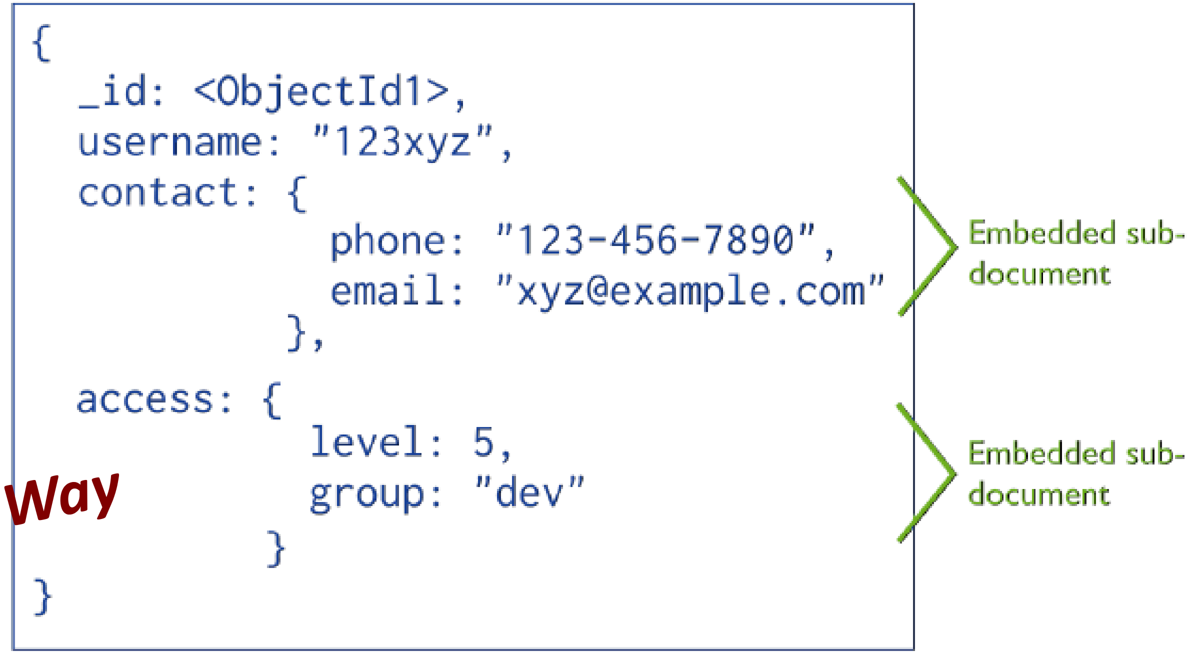
access document

```
{
  _id: <ObjectId3>,
  user_id: <ObjectId1>,
  level: 5,
  group: "dev"
}
```

Normalized Way

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

Embedding



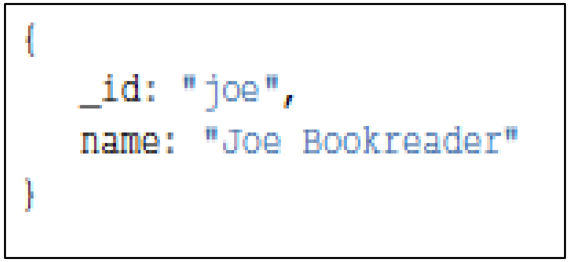
```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
            },
  access: {
             level: 5,
             group: "dev"
           }
}
```

De-Normalized Way

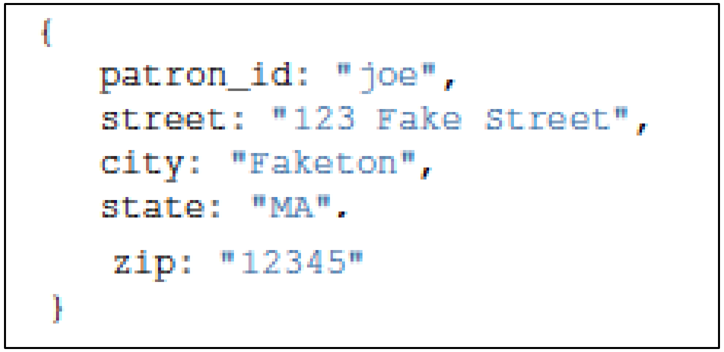
Have one collection in DB: “User”
The others are embedded inside each user’s document

Examples (1)

“Patron” & “Addresses”



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



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

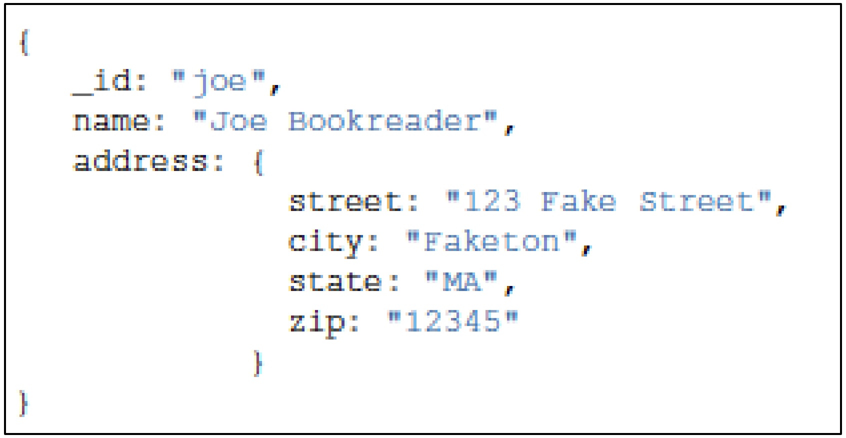
Referencing

- 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”



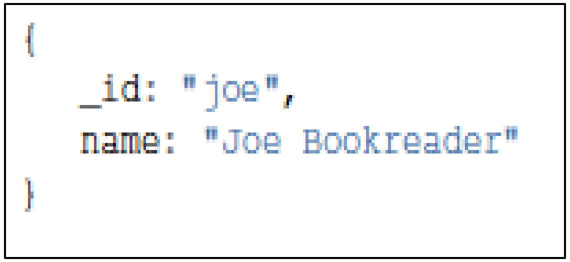
```
{
   _id: "joe",
   name: "Joe Bookreader",
   address: {
               street: "123 Fake Street",
               city: "Faketon",
               state: "MA",
               zip: "12345"
             }
}
```

Embedding

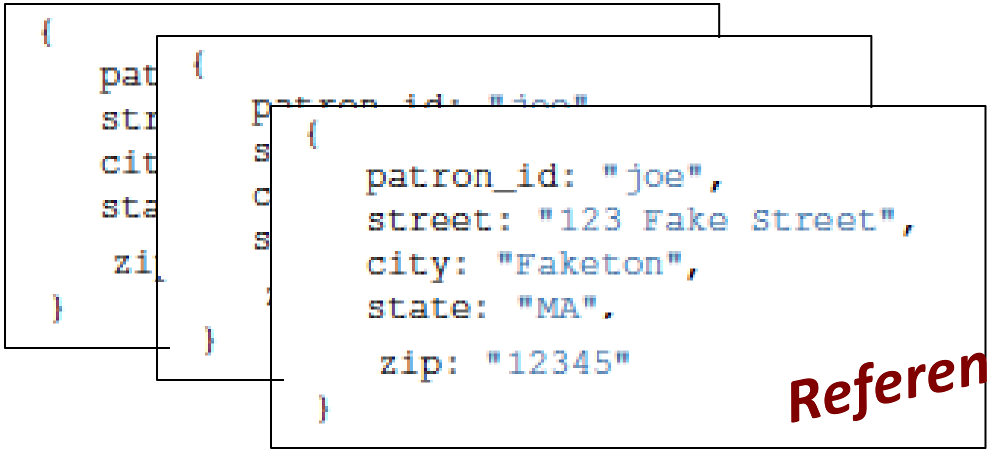
- When you read, you get the entire document at once
- In Referencing ➔ Need to issue multiple queries

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"
}
```

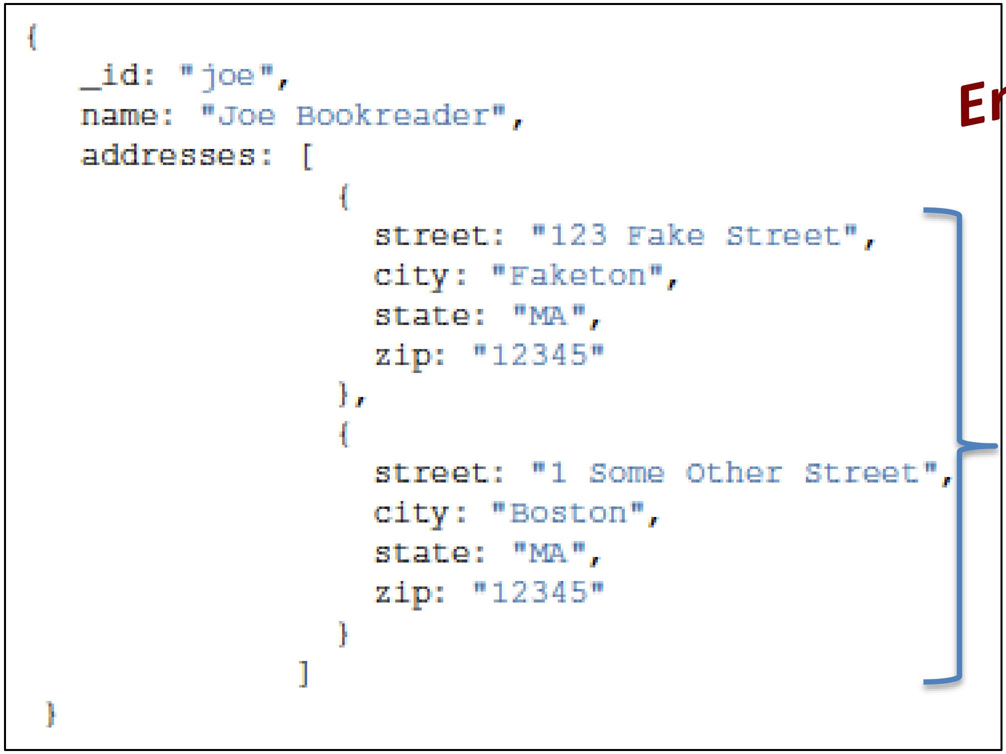
Referencing

- 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”



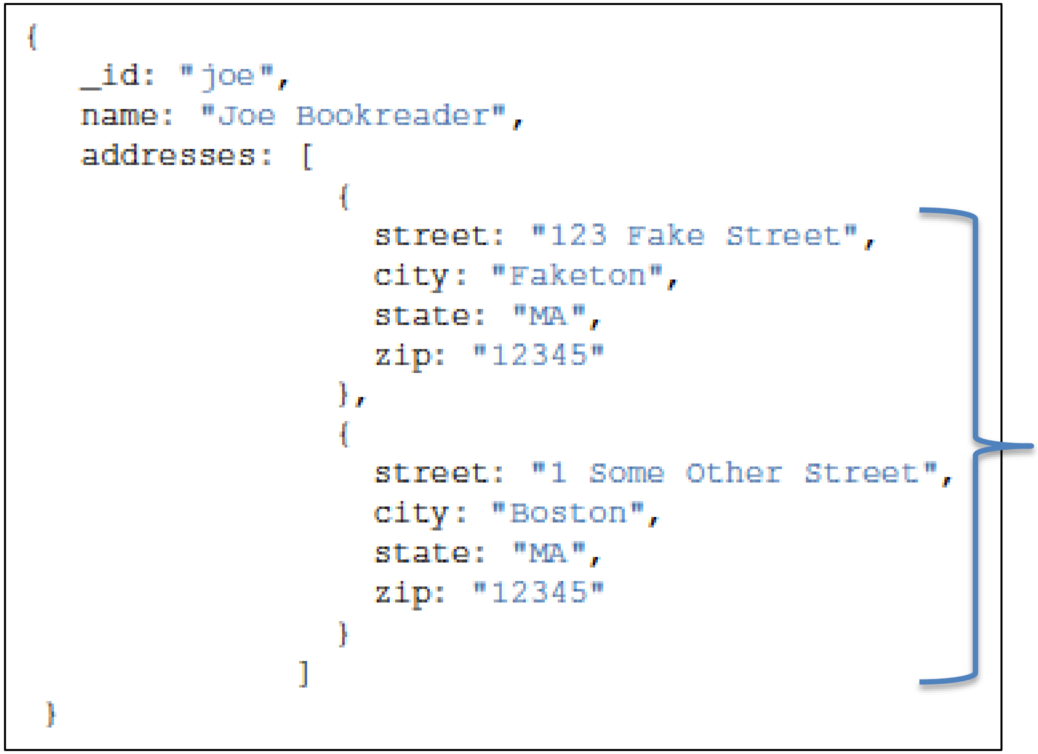
```
{
   _id: "joe",
   name: "Joe Bookreader",
   addresses: [
                {
                  street: "123 Fake Street",
                  city: "Faketon",
                  state: "MA",
                  zip: "12345"
                },
                {
                  street: "1 Some Other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

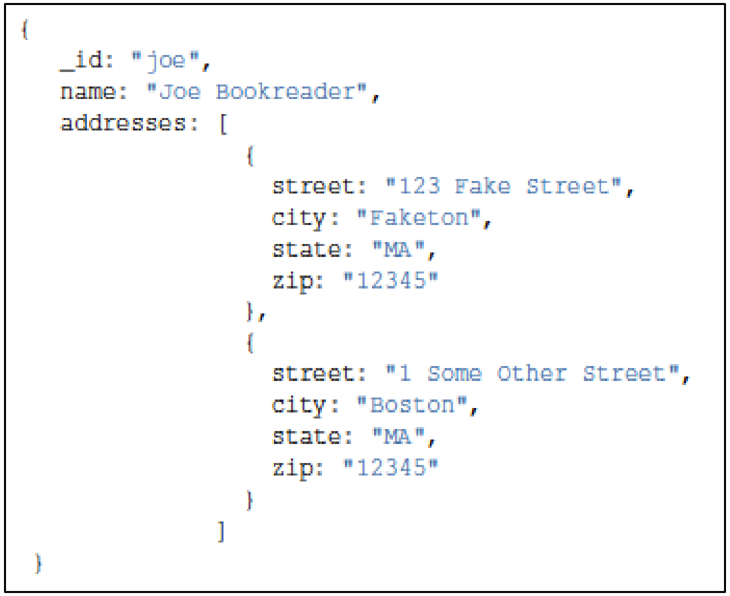
Document Size and Storage

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"
                }
              ]
 }
```

- 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

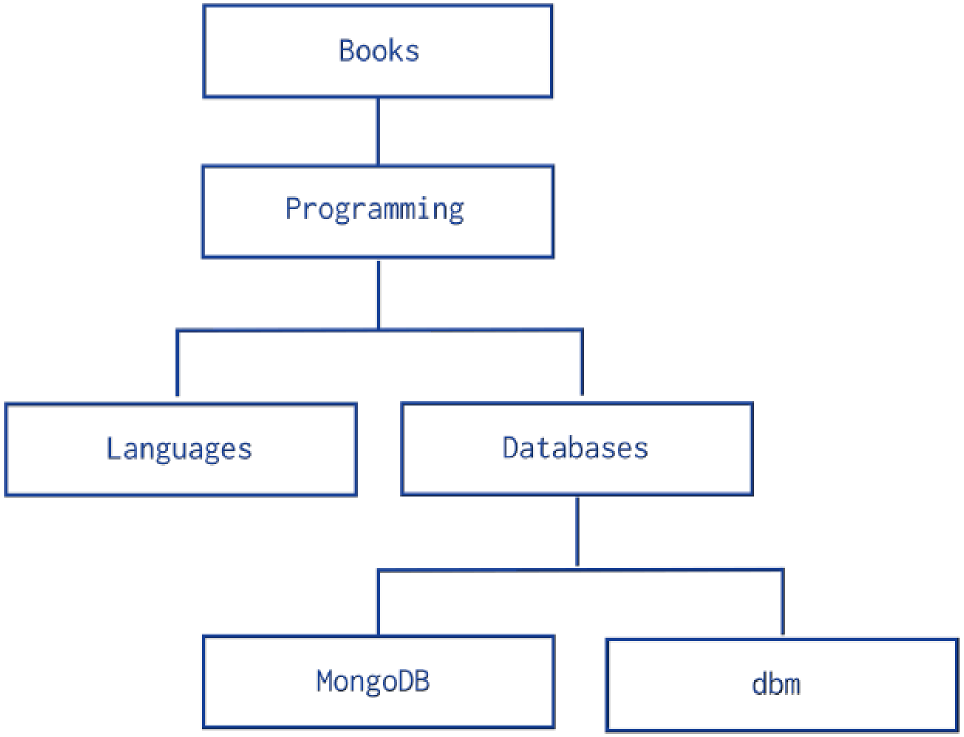
Modeling Tree Structure

Collections with Tree-Like Relationships

- Insert these records while maintaining this tree-like relationship

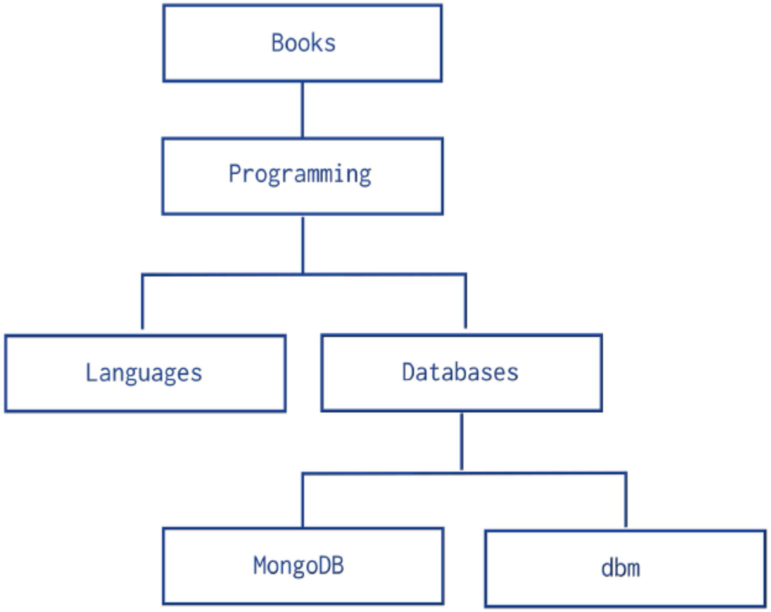
Given one node, answer queries:

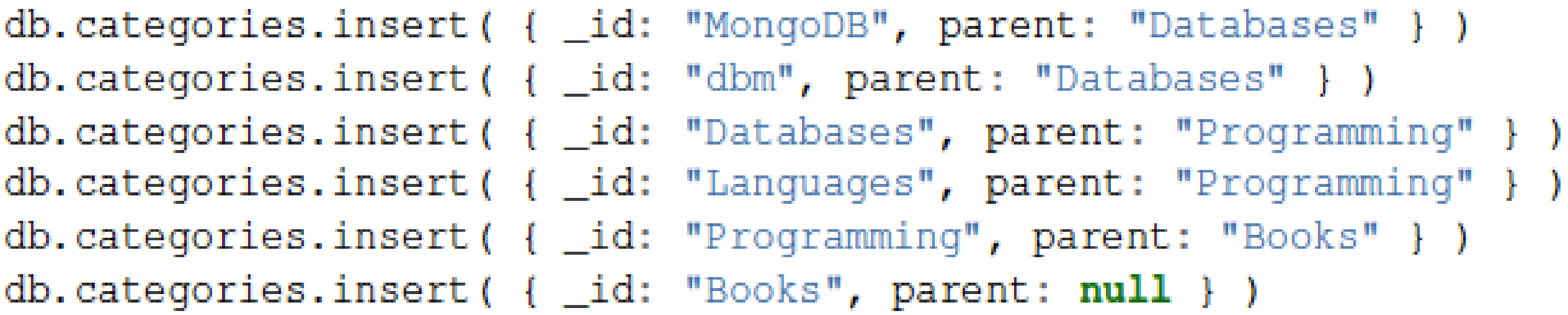
- Report the parent node
- Report the children nodes
- Report the ancestors
- Report the descendants
- Report the siblings



Method 1: Parent References

Each document has a field “parent”
Order does not matter

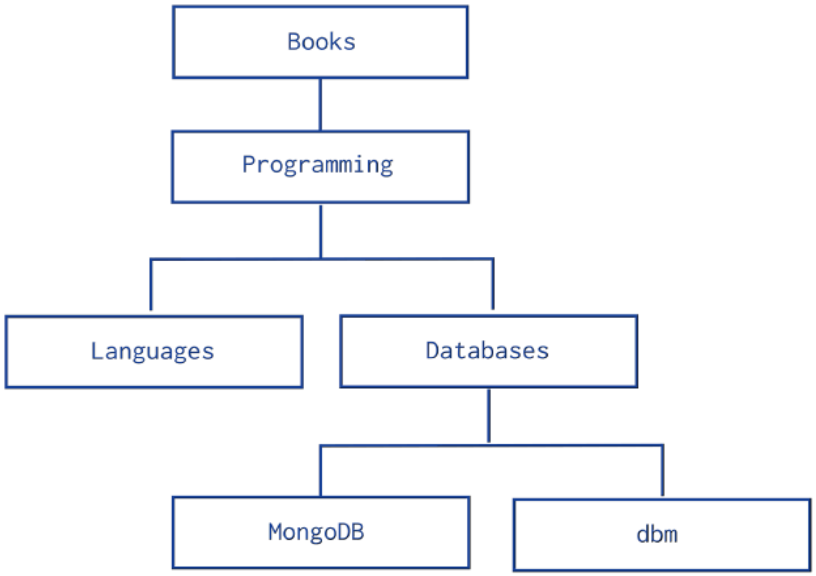


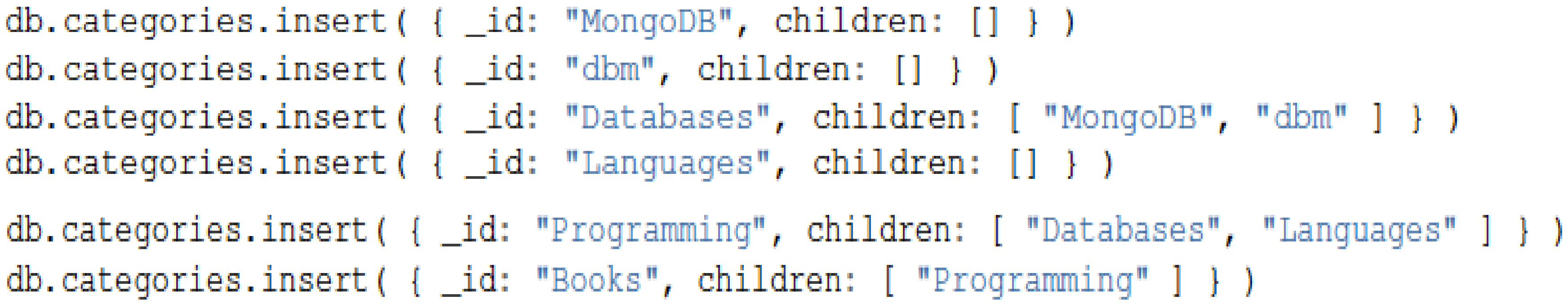


```
db.categories.insert( { _id: "MongoDB", parent: "Databases" } )
db.categories.insert( { _id: "dbm", parent: "Databases" } )
db.categories.insert( { _id: "Databases", parent: "Programming" } )
db.categories.insert( { _id: "Languages", parent: "Programming" } )
db.categories.insert( { _id: "Programming",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

Method 2: Child References

Each document has an array of immediate children



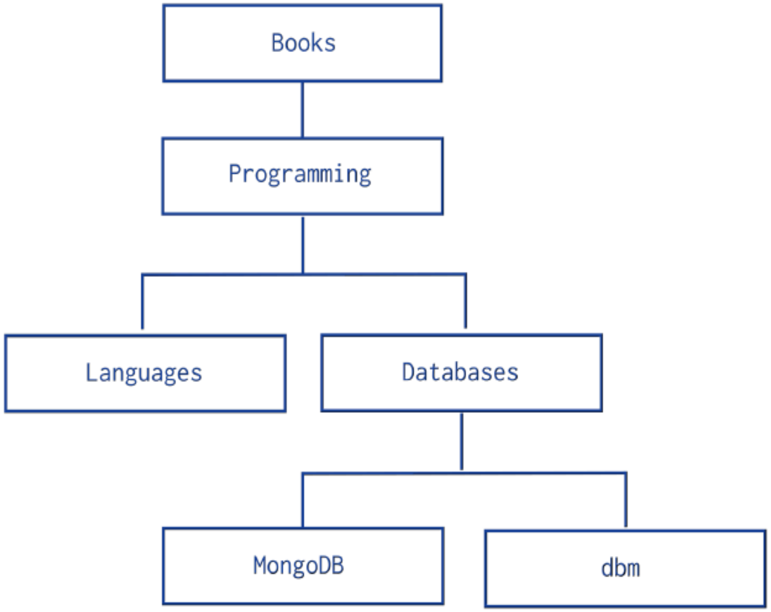


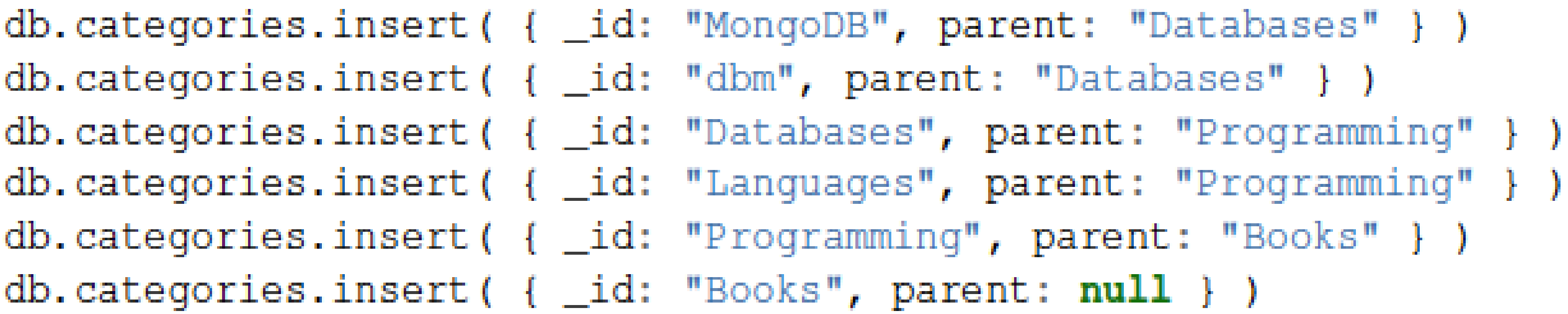
```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "dbm", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "dbm" ] } )
db.categories.insert( { _id: "Languages", children: [] } )

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

Method 1: Parent References

Q1: Parent of “Programming”

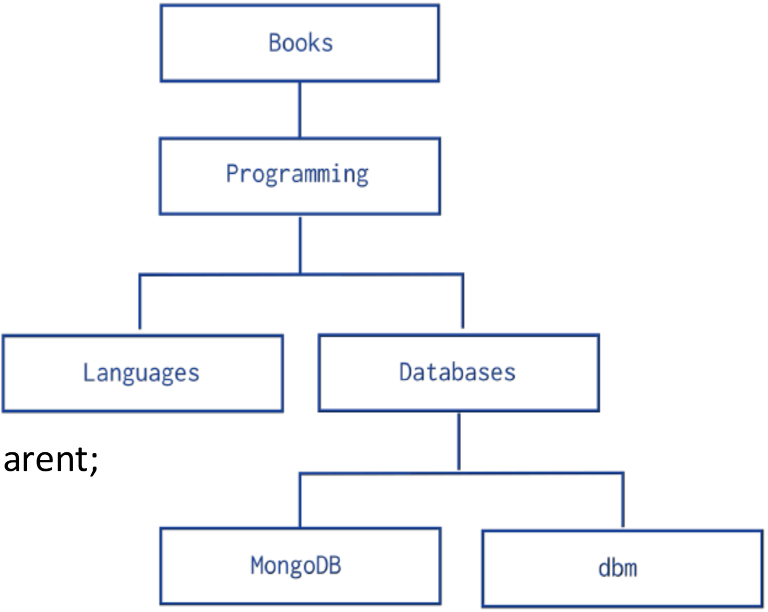
Q2: Siblings of “Databases”





```
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



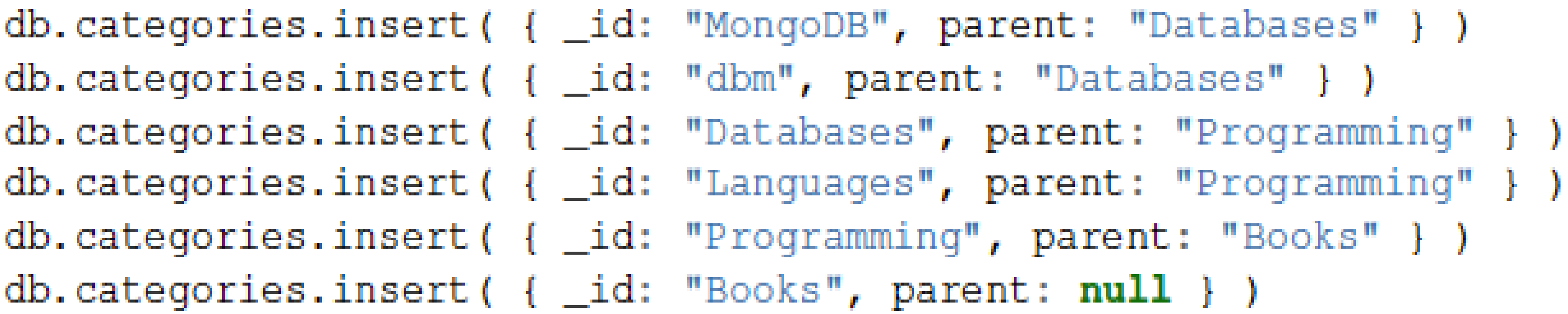
Q1: Parent of “Programming”

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

Q2: Siblings of “Databases”

var parentDoc = db.categories.findOne({_id: "Databases"}).parent;

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

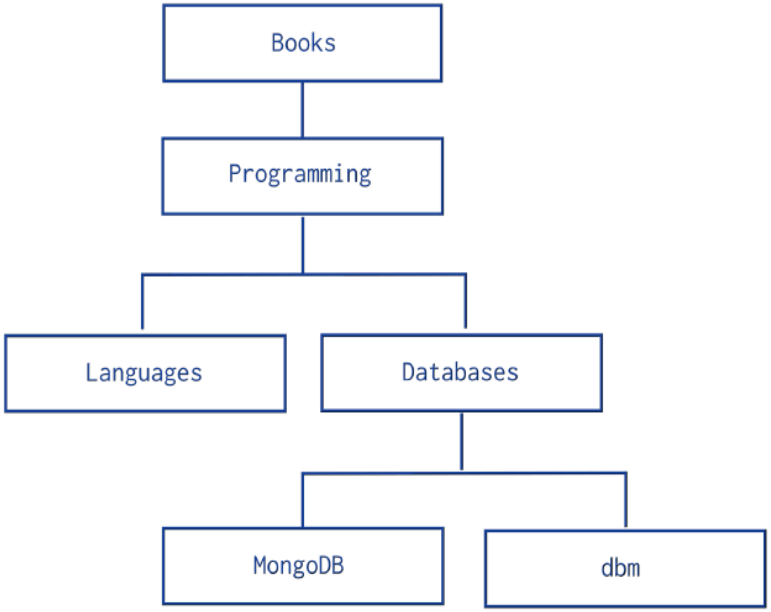


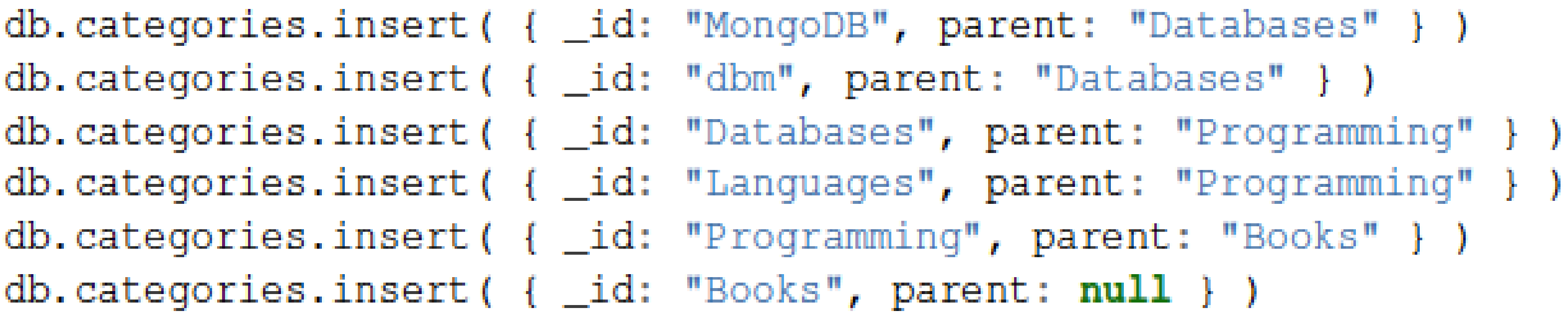
```
db.categories.insert( { _id: "MongoDB", parent: "Databases" } )
db.categories.insert( { _id: "dbm", parent: "Databases" } )
db.categories.insert( { _id: "Databases", parent: "Programming" } )
db.categories.insert( { _id: "Languages", parent: "Programming" } )
db.categories.insert( { _id: "Programming",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

Method 1: Parent References

Q3: Descendants of “Programming”

Complex…Requires recursive calls



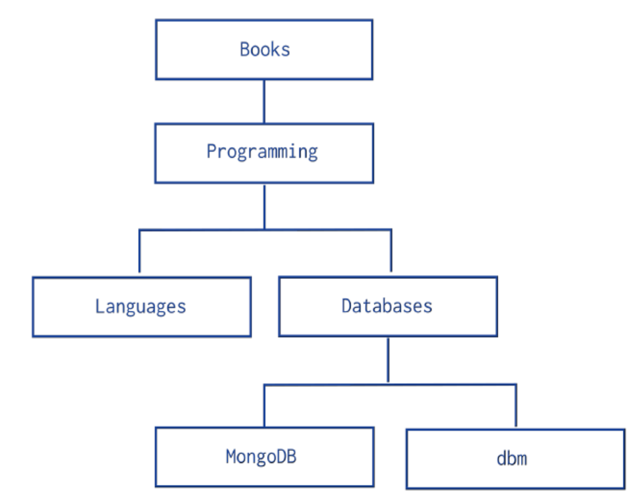


```
db.categories.insert( { _id: "MongoDB", parent: "Databases" } )
db.categories.insert( { _id: "dbm", parent: "Databases" } )
db.categories.insert( { _id: "Databases", parent: "Programming" } )
db.categories.insert( { _id: "Languages", parent: "Programming" } )
db.categories.insert( { _id: "Programming", 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.findOne({_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;
```



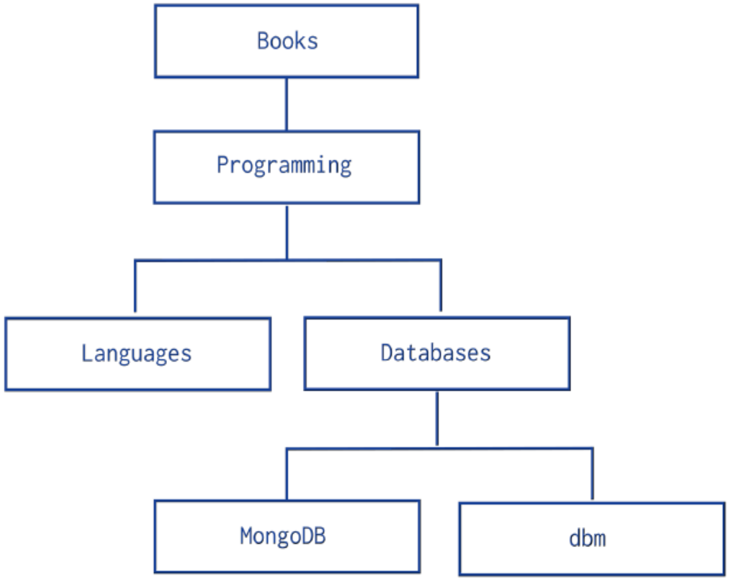
Method 1: Parent References

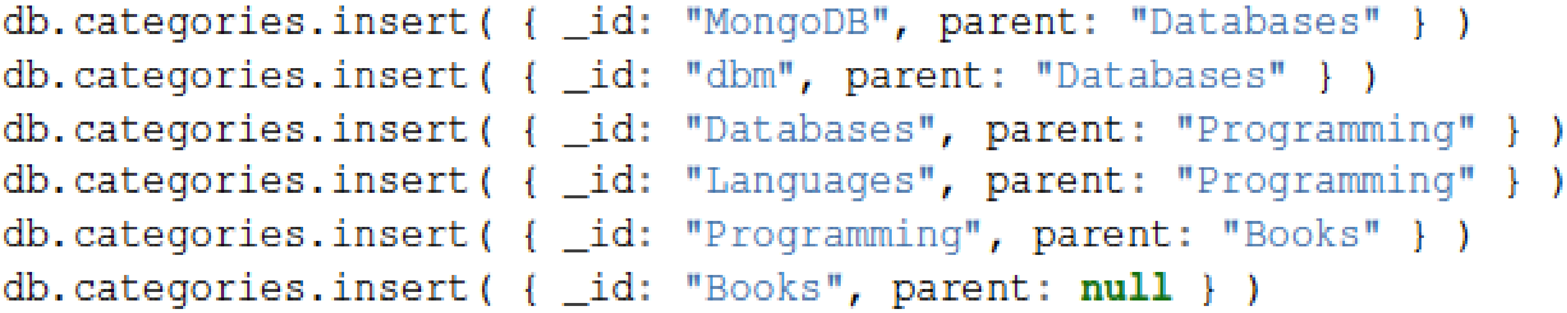
Q4: Ancestors of “MongoDB”

Try it yourself….

Should be:

“Databases”, “Programming”, “Books”

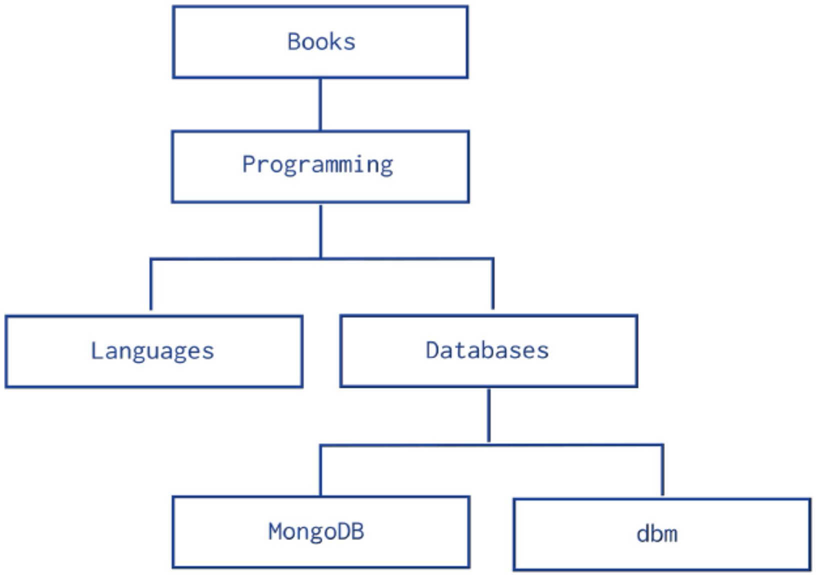


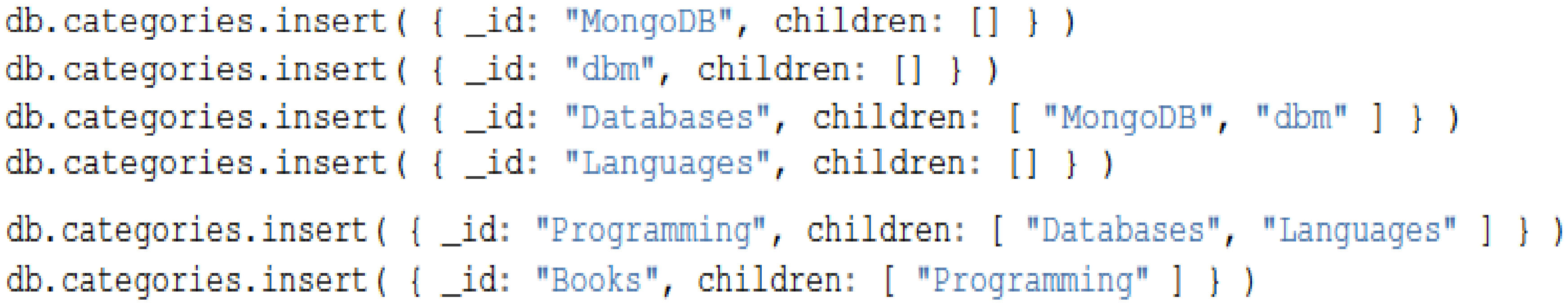


```
db.categories.insert( { _id: "MongoDB", parent: "Databases" } )
db.categories.insert( { _id: "dbm", parent: "Databases" } )
db.categories.insert( { _id: "Databases", parent: "Programming" } )
db.categories.insert( { _id: "Languages", parent: "Programming" } )
db.categories.insert( { _id: "Programming",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

Method 2: Child References

Q1: Get children documents of “Programming”



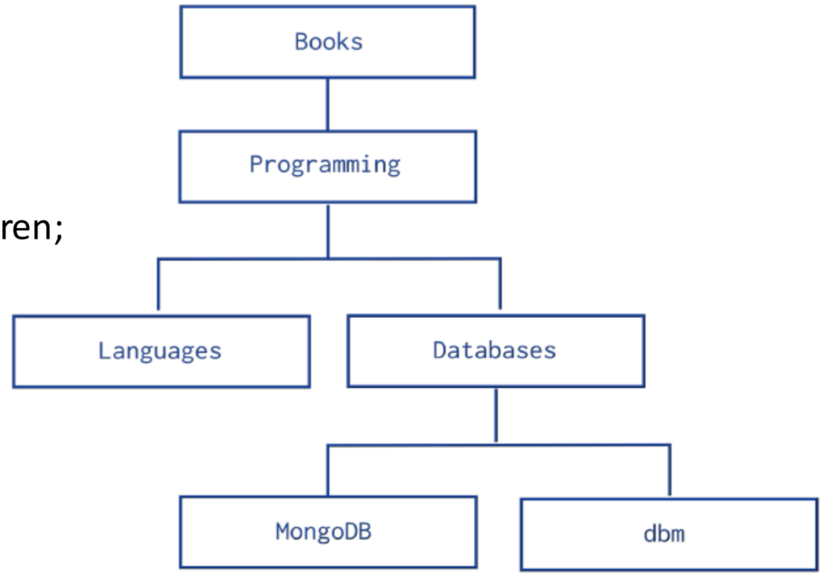


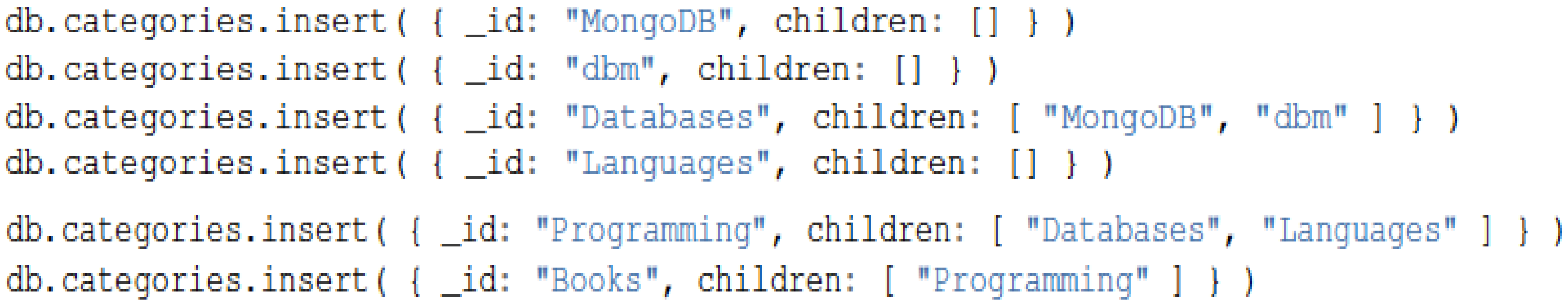
```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "dbm", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "dbm" ] } )
db.categories.insert( { _id: "Languages", children: [] } )

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}});
```

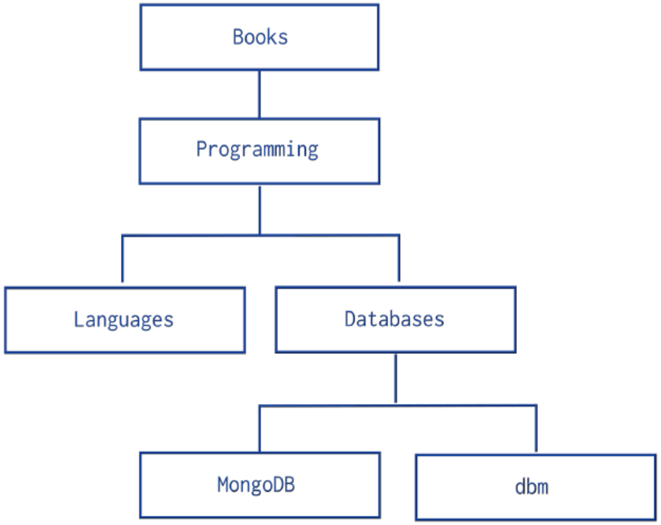


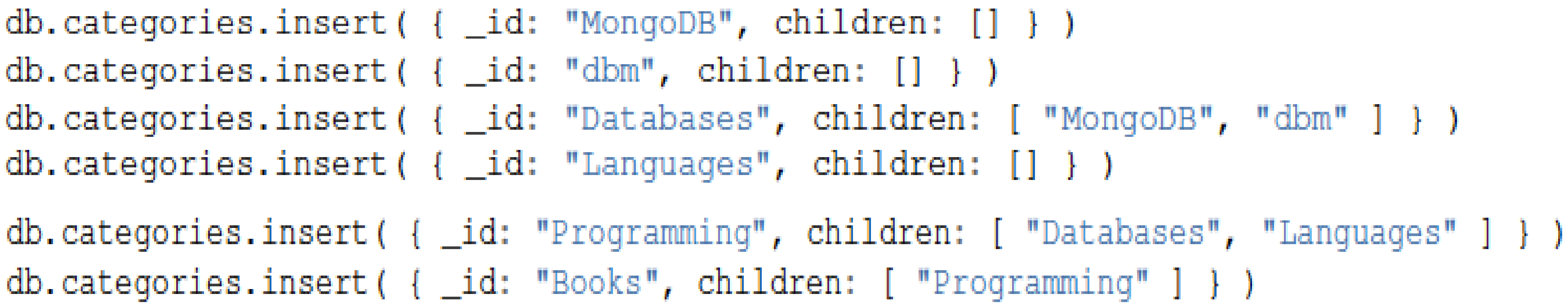


```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "dbm", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "dbm" ] } )
db.categories.insert( { _id: "Languages", children: [] } )

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

Method 2: Child References

Q2: Ancestors of “MongoDB”

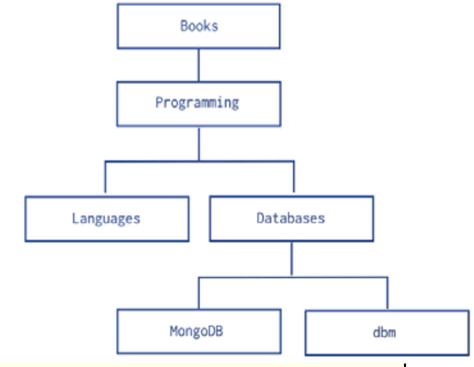




```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "dbm", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "dbm" ] } )
db.categories.insert( { _id: "Languages", children: [] } )

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

Method 2: Child References

Q2: Ancestors of “MongoDB”



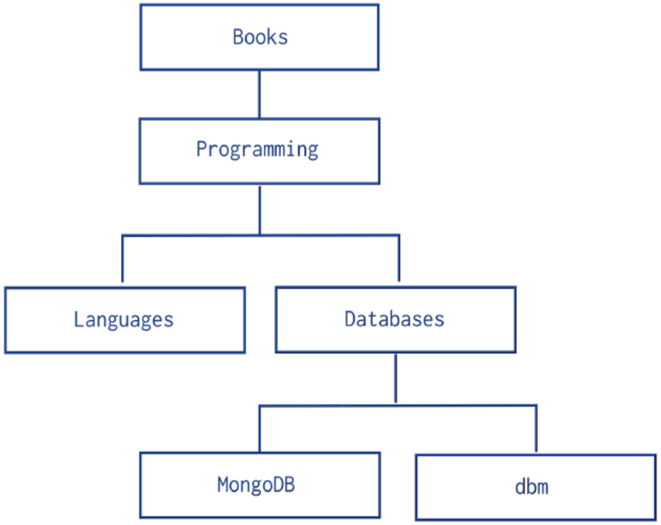
```
var results=[];
var parent = db.categories.findOne({children: "MongoDB"});
while(parent){
        print({Message: "Going up one level…"});
        results.push(parent._id);
        parent = db.categories.findOne({children: parent._id});
}


results;
```

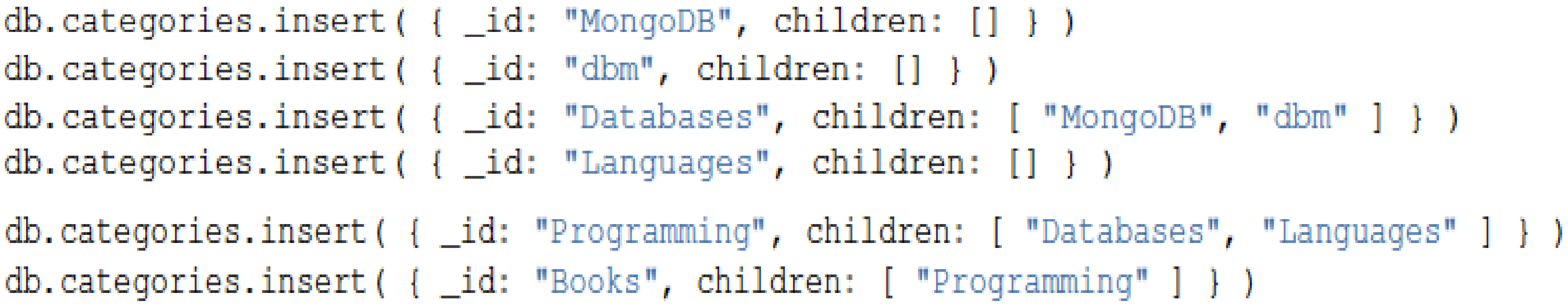
Method 2: Child References

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" ] } )
```

For today

- Download and install Neo4j community edition
- https://neo4j.com/download-center/#community

If you don’t already have it, you will also need to install Oracle JDK or Open JDK - Java Development Kit Standard Edition.

Visit http://localhost:7474 in your web browser
Default username and password: ‘neo4j’

Submit a screenshot to ICON