# MongoDB
# Basic Schema Design Patterns

COSC 061 - Fall 2022 - Dartmouth College

# RDB vs. MongoDB Design

- For Relational DB’s
  - Get the application requirements
  - Find the data
  - Fit the data into a relational database
  - Give it to the programmer to implement

# RDB vs. MongoDB Design

- For MongoDB
  - Get the application requirements
  - Ask how the user will interact with the application
  - Model the data accordingly
  - Give it to the programmer to implement

# A good data model

- makes it easier to manage the data
- can make queries more efficient in time, memory, and CPU usage

Which contributes to lowering overall costs of the database

# Points to ponder when designing

## Questions …

- What does my application do?
- What data will I store
- How will users access the data
- What data will be most valuable to me? To the users?
- Is any of the data sensitive or regulated?

# Points to ponder when designing

## … the answers will

- help you describe your tasks as well as those of the user
- help you clarify what your data looks like and the relations between data
- identify tools you or the users might need
- predict access patterns that might emerge
- identify any extra care you will need to take with the data

# MongoDB Design Rule

Data that is accessed together
should be stored together

# 1-to-1 vs. 1-to-few

Embed when the child objects never appear outside the context of their parent. Otherwise, store the child objects in a separate collection.

**From *MongoDB in Action, 2e*, Banker et.al.**

# 1-to-few

- Simple embedding is ok for a few, even though some duplicate storage of an address if someone else also lives there.
- Embedding *may* provide a slight performance advantage
- Embedding makes it harder to access the embedded documents

```
db.student.findOne()
{
    fname: 'Lisa',
    lname: 'Simpson',
    sId: 'x0831562',
    addresses: [{
            street: '16 Hilfiger Hall',
            city: 'Cambridge',
            state: 'MA'
        },
        {
            street: '742 Evergreen Terrace',
            city: 'Springfield',
        }
    ]
}
```

# 1-to-many

The “one” side has an array of references (less than about 100) to the items from the “many” side.

```
db.orders.insertOne({
    orderNo: 91269192,
    itemList: [
        ObjectId(“…aaeee0”),
        ObjectId(“…aaeee1”),
        ObjectId(“…aaeee2”)
    ]
})
```

```
db.items.insertMany( [{
            _id: ObjectId(“…aaeee0”),
            itemNo: 'FM191201',
            name: 'Nerdit Gamer Backpack',
            qty: 1,
            area: 'B427',
            rack: 4,
            bin: '702'
        },{
            _id: ObjectId(“…aaeee1”),
            itemNo: 'FM191203',
            name: 'Nerdit Gamer Pack Strap',
            qty: 1,
            area: 'B427',
            rack: 4,
            bin: '702'
        },{
            _id: ObjectId(“…aaeee2”),
            itemNo: 'FM191205',
            name: 'Nerdit Gamer Pack bottle',
            qty: 1,
            area: 'B427',
            rack: 4,
            bin: '704'
        }
    ])
```

# 1-to-many

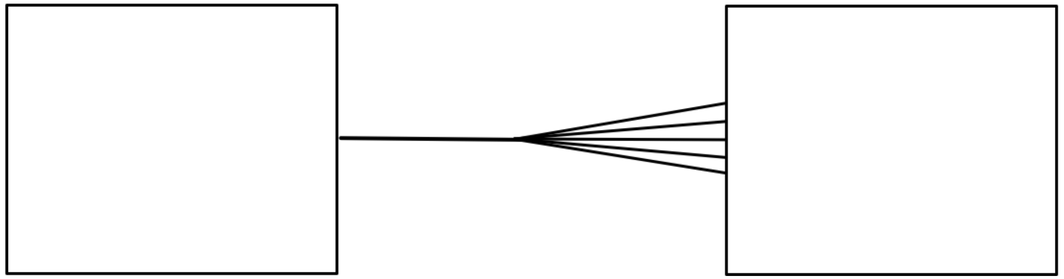
To get a list of the items in the order  we do a MongoDB **join** into an array in program memory.

```
// Fetch the order document
order = db.orders.findOne ( {
    orderNo: 91269192
} );

// Fetch the Parts that are linked to this order

orderItems = db.items.find ( {
    _id: {
        $in: order.itemList
    }
  } ).toArray(); // return all the elements of the
                 // cursor as an array in memory
```

# 1-to-zillions !



Reference the document on the “one” side of the relationship from the “zillion” side

...

# 1-to-zillions !

Turn the model upside down and have the many side reference the one.

Suppose your web server sends its log messages to a MongoDB db.

That’s a lot of log entries really fast!

```
db.webserver.insertOne({
    _id: ObjectId("59242cee60ae8a3ae6aaeee0"),
    name: 'www1.beebleford.com',
    ipAddr: '127.66.67.68'
})

db.logmsg.insertOne({
    server: ObjectId("59242cee60ae8a3ae6aaeee0"),
    time: ISODate("2022-03-13T03:22:41.382Z"),
    ipAddr: '182.77.43.137',
    cmd: "GET /~cs50/programming.css HTTP/1.1",
    codes: "200 8052",
    url: "http://www.cs.dartmouth.edu/~cs50/
programming.html",
    details: "Mozilla/5.0 (Windows NT 6.1; Win64;
x64) AppleWebKit/537.36 (KHTML, like Gecko)
Safari/537.36"
})
```

# 1-to-zillions !

To get the last 1000 log entries we can then do something like this ->

```
// find the parent ‘server’ document (assuming a
unique IP address)
server = db.webserver.findOne({
    ipAddr: '127.66.67.68'
});

// find the most recent 1000 log message
// documents linked to that web server

last1000 = db.logmsg.find({
        webserver: server._id
    })
    .sort({
        time: -1
    })
    .limit(1000)
    .toArray()
```

# Will Zola's 1-to-N guidelines summary

1. Will the entities on the *N* side of the *1-to-N* ever need to stand alone?

2. What is the cardinality of the relationship:

    a. *1-to-few*: Embed the *N* side into the one-side as long as there is no need to access the embedded object outside of the one-side.

    b. *1-to-many*: Use an array of references on the one-side, to the objects on the *N* side if *1-to-many* OR if the N side objects ever need to stand alone.

    c. *1-to-zillions*: Use a reference to the one-side in the objects on the *N* side

# Many-to-many

Consider products in an online catalog.

A product may be found in one or more categories.

A categories may refer to one or more products.

For efficiency, be sure to create an index on the category IDs.

```
{ _id: ObjectId("4d6574baa6b804ea563c132a"),
  title: "Epiphytes" }

{ _id: ObjectId("4d6574baa6b804ea563c459d"),
  title: "Greenhouse flowers" }

// then a product belonging to both
categories will look like this:

{ _id: ObjectId("4d6574baa6b804ea563ca982"),
  name: "Dragon Orchid",
  category_ids: [
        ObjectId(“4d6574baa6b804ea563c132a”),
        ObjectId("4d6574baa6b804ea563c459d")
  ]
}
-- - - - - - - - - - - - - - - - - - - - - - - -
db.products.createIndex({category_ids: 1})
```

Example from MongoDB in Action 2e, Banker et.al.

# Many-to-many

To find all products in the Epiphytes category, match against the category_id field.

To return all category documents related to the Dragon Orchid product, first get the list of that product's category IDs.

Then query the categories collection using the $in operator.

```
db.products.find(
    {
    category_id: ObjectId("4d6574baa6b804ea563c132a
    }
)
-- - - - - - - - - - - - - - - - - - - - - -

product = db.products.findOne(
  { _id: ObjectId("4d6574baa6b804ea563c132a")
  })

db.categories.find(
    {
      _id: {
            $in: product['category_ids']
           }
    }
)
```

Example from MongoDB in Action 2e, Banker et.al.

# Materialized paths

Maintain a path in a document by concatenating _id's (with ':' separators).

Useful for webpage paths, comment or email threads, etc.

```
// materializedPaths.js
{
        _id: ObjectId("4d692b5d59e212384d95003"),
        depth: 2,
        path:
"4d692b5d59e212384d95001:4d692b5d59e212384d951002",
        username: "homer",
        body: "Now where did I put that beer ... DOH!",
        thread_id: ObjectId("4d692b5d59e212384d95223a")
}
// use code or RegEx's to dig around in the path
db.comments.find({
  path: /^4d692b5d59e212384d95001/,
});
```

# Activity

## What would be the best design pattern for … and why?

1. Recipe collection
2. eBay products catalog
3. Library catalog
4. Patient medications record at a Pharmacy
5. License plate lookup database for State Police
6. A Pizza
7. Credit card database
8. Software bill-of-materials list