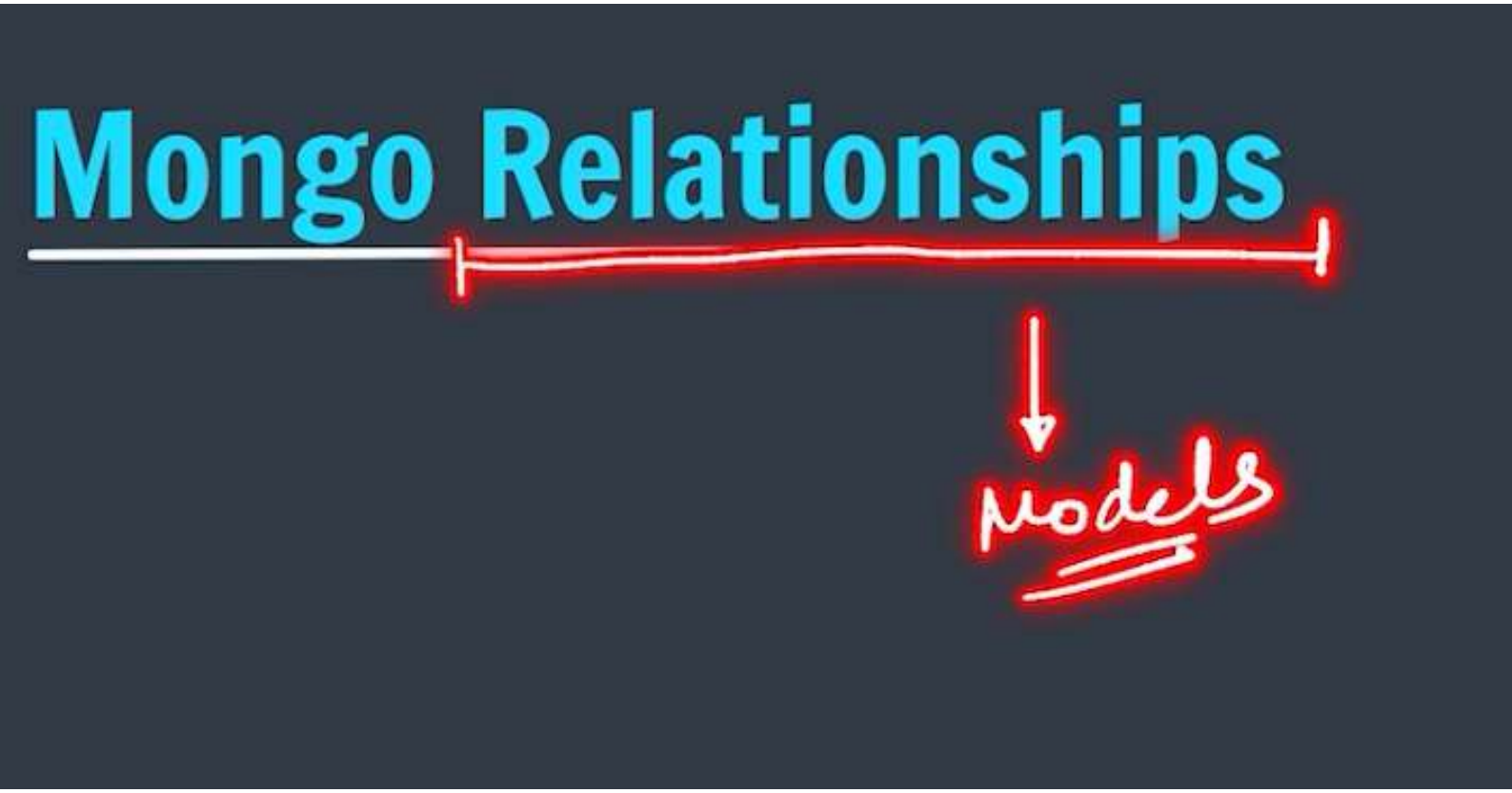

Mongo Relationships
Models

Relationships

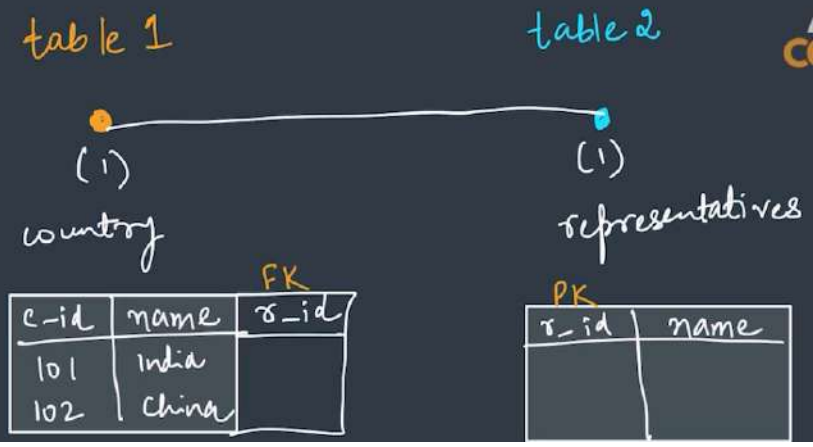
SQL (via Foreign Keys)

- one to one
- one to many
- many to many

Relationships

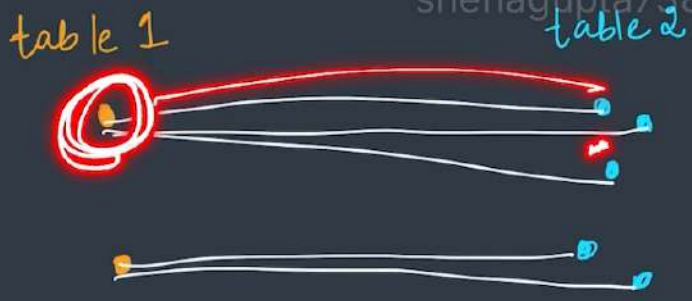
SQL (via Foreign Keys)

- one to one

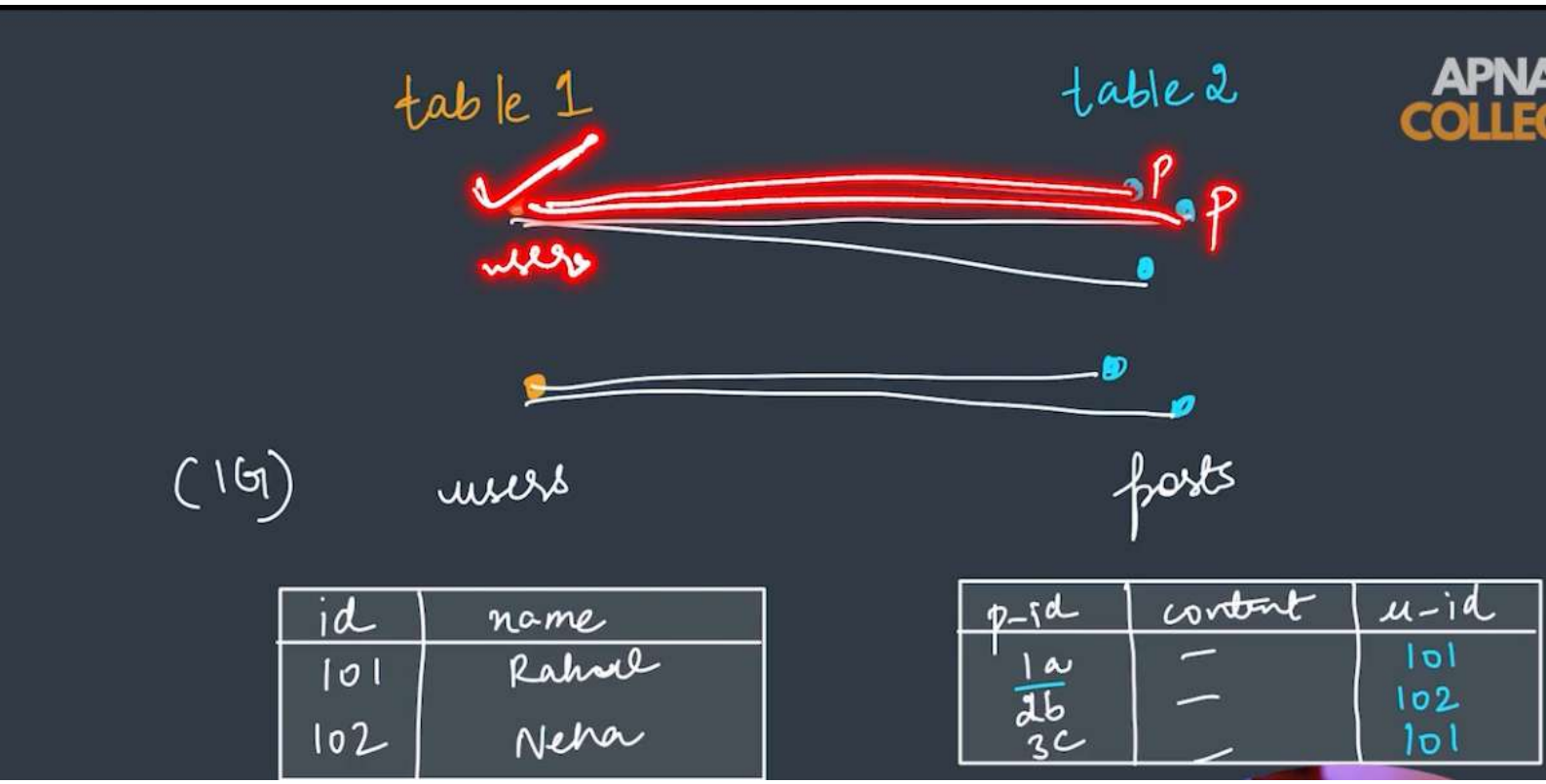


Relationships

SQL (via Foreign Keys)



- one to one
- one to many



(IG)

users

posts

id	name
101	Rahul
102	Neha

p-id	content	u-id
1a	—	101
2b	—	102
3c	—	101

Relationships

SQL (via Foreign Keys)

- one to one
  1 x 1
- one to many
  1 x n
- many to many
  n x n

cardinality →

table 1

student

id	name	s_id
101	—	1a
102	—	1a
103	—	3c

table 2

subject

s_id	name	id
1a	chem	101
2b	phy	—
3c	math	103

Relationships

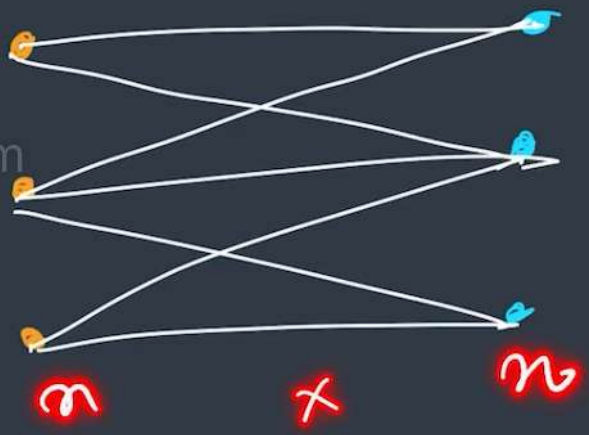
SQL (via Foreign Keys)

table 1

student

id	name	s_id
101	—	1a
102	—	1a
103	—	3c

- one to one
- one to many
- many to many

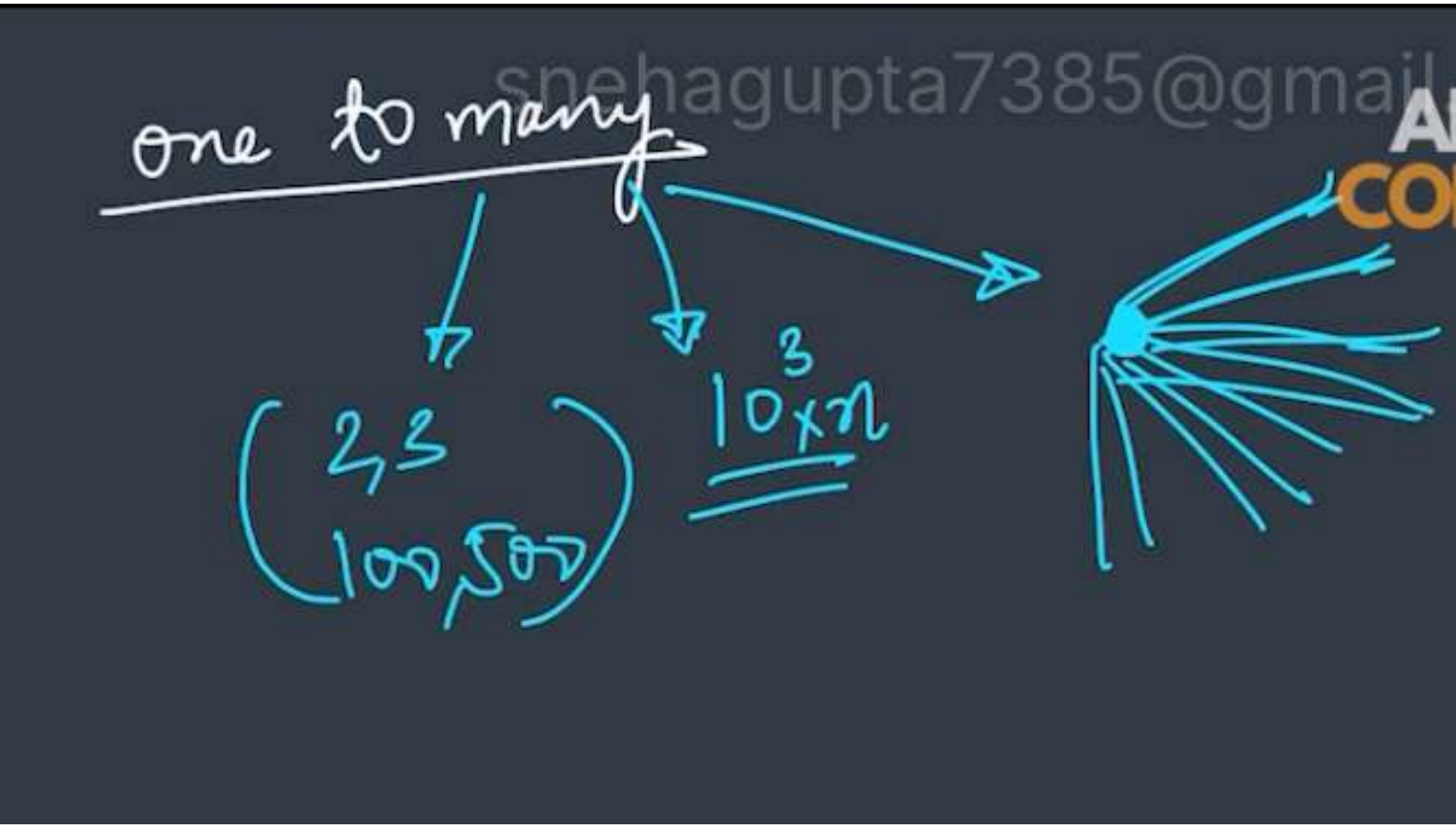


Mongo Relationships

One to Many / Approach 1 (one to few)

Store the child document inside parent

```
{
  _id: ObjectId("651d1b116976164a9cbf5520"),
  username: 'sherlockholmes',
  addresses: [
    { location: '221B Baker Street', city: 'London' },
    { location: 'P36 DownTown', city: 'London' }
  ],
  __v: 1
},
```


one to many
snehagupta7385@gmail
$10^3 \times n$

```
const mongoose = require("mongoose");
const {Schema} = mongoose;
async function main(){
    await mongoose.connect("mongodb://127.0.0.1:27017/relationDemo")
}
main()
 .then(()=> console.log("Connection Successful"))
 .catch((err)=> console.log(err))

const userSchema = new Schema ({
    username: String,
    addresses:[
        {
            _id:false,
            location: String,
            city: String,
        },
    ],
})
const User = mongoose.model("User",userSchema);

const addUsers = async()=>{
    let user1 = new User({
        username:"sherlockholmes",
        addresses:[
            {
                location:"221B Baker Street",
                city:"London",
            },
        ],
    })
    user1.addresses.push({location:"P32 WallStreet",city:"London"});
    let result = await user1.save();
    console.log(result)
}
addUsers();
```

```
ashugoyal@Ashus-MacBook-Air models % node user.js
Connection Successful
{
  username: 'sherlockholmes',
  addresses: [
    { location: '221B Baker Street', city: 'London' },
    { location: 'P32 WallStreet', city: 'London' }
  ],
  _id: new ObjectId('6713467b8ef7c4360083d562'),
  __v: 0
}
```

```
ashugoyal@Ashus-MacBook-Air ~ % mongosh
Current Mongosh Log ID: 67133c257aa617f1ab73f2d3
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.3.2
Using MongoDB:          6.0.15
Using Mongosh:          2.3.2

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

------
   The server generated these startup warnings when booting
   2024-10-18T19:34:21.705+05:30: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
------

test> show dbs
admin          40.00 KiB
config        108.00 KiB
local          96.00 KiB
relationDemo   48.00 KiB
wanderlust     80.00 KiB
test> use relationDemo
switched to db relationDemo
```

```
relationDemo> show collections
users
relationDemo> db.users.find()
[
  {
    _id: ObjectId('67133bdf8e5d49b61a25a50e'),
    username: 'sherlockholmes',
    addresses: [
      {
        location: '221B Baker Street',
        city: 'London',
        _id: ObjectId('67133bdf8e5d49b61a25a50f')
      },
      {
        location: 'P32 WallStreet',
        city: 'London',
        _id: ObjectId('67133bdf8e5d49b61a25a510')
      }
    ],
    __v: 0
  }
]
relationDemo> db.users.deleteMany({})
{ acknowledged: true, deletedCount: 1 }
```

```
relationDemo> db.users.deleteMany({})
{ acknowledged: true, deletedCount: 1 }
relationDemo> db.users.find()

relationDemo> db.users.find()
[
  {
    _id: ObjectId('6713467b8ef7c4360083d562'),
    username: 'sherlockholmes',
    addresses: [
      { location: '221B Baker Street', city: 'London' },
      { location: 'P32 WallStreet', city: 'London' }
    ],
    __v: 0
  }
]
```

Mongo Relationships

One to Many / Approach 2

Store a reference to the child document inside parent

```
{
  _id: ObjectId("651d223314f1e136d6766e14"),
  name: 'Rahul Kumar',
  orders: [
    ObjectId("651d1e5a06e366283d3ae002"),
    ObjectId("651d1e5a06e366283d3ae003")
  ],
  __v: 0
}
```


Customers
order
details

cust1.orders.push(objectId)

objectId

child docs

```
const orderSchema = new Schema ({
    item: String,
    price: Number,
})

const Order = mongoose.model("Order",orderSchema);
```

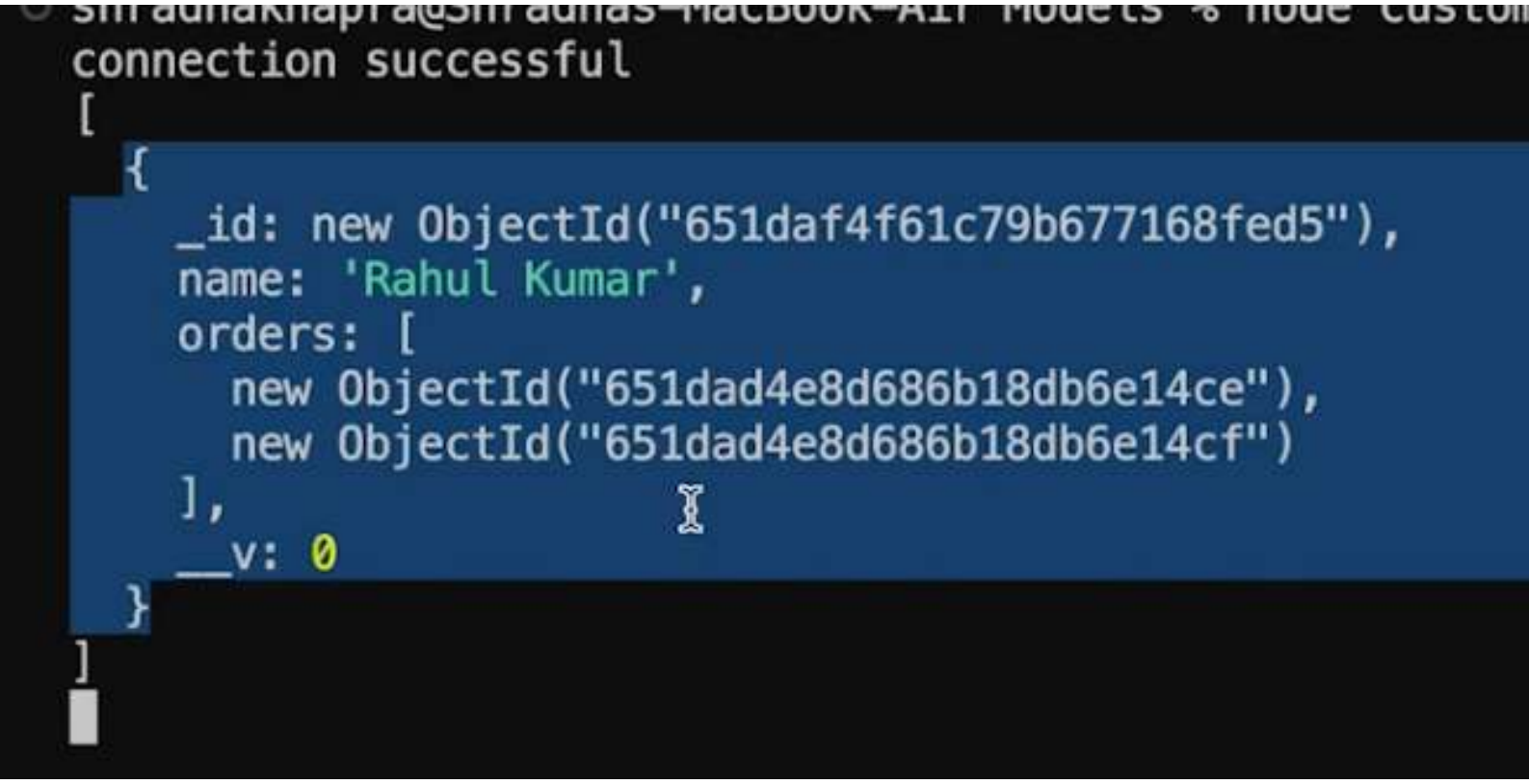
```
const addorders = async()=>{
    let res = await Order.insertMany([
        {
        item: "samosa",
        price:12,
        },
        {
            item: "Chips",
            price: 10,
        },
        {
            item: "Pizza",
            price:40,
        }
    ]);
    console.log(res);
}
addorders();
```

```
ashugoyal@Ashus-MacBook-Air models % node customer.js
Connection Successful
[
  {
    item: 'samosa',
    price: 12,
    _id: new ObjectId('6713570f0a82561d86315e63'),
    __v: 0
  },
  {
    item: 'Chips',
    price: 10,
    _id: new ObjectId('6713570f0a82561d86315e64'),
    __v: 0
  },
  {
    item: 'Pizza',
    price: 40,
    _id: new ObjectId('6713570f0a82561d86315e65'),
    __v: 0
  }
]
```

```
const customerSchema = new Schema ({
    name:String,
    orders: [
        {
            type: Schema.Types.ObjectId,
            ref: "Order"
        }
    ]
})
const Customer = mongoose.model("Customer",customerSchema)

const addcustomer = async ()=>{
    let cust1 = new Customer({
        name: "Rahul Kumar",
    });
    let order1 = await Order.findOne({item:"Chips"});
    let order2 = await Order.findOne({item:"Pizza"});
    cust1.orders.push(order1);
    cust1.orders.push(order2);
    let res = await cust1.save();
    console.log(res);
}

addcustomer();
```

```
ashugoyal@Ashus-MacBook-Air models % node customer.js
Connection Successful
{
  name: 'Rahul Kumar',
  orders: [
    {
      item: 'Chips',
      price: 10,
      _id: new ObjectId('6713570f0a82561d86315e64'),
      __v: 0
    },
    {
      item: 'Pizza',
      price: 40,
      _id: new ObjectId('6713570f0a82561d86315e65'),
      __v: 0
    }
  ],
  _id: new ObjectId('67135abb433f016a17d984a3'),
  __v: 0
}
```

```
let result = await Customer.find({});
console.log(result);
```


connection successful
[
{
_id: new ObjectId("651daf4f61c79b677168fed5"),
name: 'Rahul Kumar',
orders: [
new ObjectId("651dad4e8d686b18db6e14ce"),
new ObjectId("651dad4e8d686b18db6e14cf")
],
__v: 0
}
]

Populate

MongoDB has the join-like $lookup aggregation operator in versions >= 3.2. Mongoose has a more powerful alternative called `populate()`, which lets you reference documents in other collections.

Population is the process of automatically replacing the specified paths in the document with document(s) from other collection(s). We may populate a single document, multiple documents, a plain object, multiple plain objects, or all objects returned from a query. Let's look at some examples.

```
const mongoose = require('mongoose');
const { Schema } = mongoose;

const personSchema = Schema({
  _id: Schema.Types.ObjectId,
  name: String,
  age: Number,
  stories: [{ type: Schema.Types.ObjectId, ref: 'Story' }]
});

const storySchema = Schema({
  author: { type: Schema.Types.ObjectId, ref: 'Person' },
  title: String,
  fans: [{ type: Schema.Types.ObjectId, ref: 'Person' }]
});

const Story = mongoose.model('Story', storySchema);
const Person = mongoose.model('Person', personSchema);
```

So far we've created two Models. Our `Person` model has its `stories` field set to an array of `ObjectId`s. The option is what tells Mongoose which model to use during population, in our case the `Story` model. All `_id`s we store here must be document `_id`s from the `Story` model.

Population

So far we haven't done anything much different. We've merely created a `Person` and a `Story`. Now let's take a look at populating our story's `author` using the query builder:

```
const story = await Story.
  findOne({ title: 'Casino Royale' }).
  populate('author').
  exec();
// prints "The author is Ian Fleming"
console.log('The author is %s', story.author.name);
```

Populated paths are no longer set to their original `_id`, their value is replaced with the mongoose document returned from the database by performing a separate query before returning the results.

Arrays of refs work the same way. Just call the populate method on the query and an array of documents will be returned *in place* of the original `_id`s.

```
const customerSchema = new Schema ({
    name:String,
    orders: [
        {
            type: Schema.Types.ObjectId,
            ref: "Order"
        }
    ]
})
const Customer = mongoose.model("Customer",customerSchema)

const findCustomer = async()=>{
    let result = await Customer.find({}).populate("orders");
    console.log(result)
}
findCustomer();
```

```
ashugoyal@Ashus-MacBook-Air models % node customer.js
Connection Successful
[
  {
    _id: new ObjectId('67135abb433f016a17d984a3'),
    name: 'Rahul Kumar',
    orders: [ [Object], [Object] ],
    __v: 0
  }
]
```

```
const customerSchema = new Schema ({
    name:String,
    orders: [
        {
            type: Schema.Types.ObjectId,
            ref: "Order"
        }
    ]
})
const Customer = mongoose.model("Customer",customerSchema)

const findCustomer = async()=>{
    let result = await Customer.find({}).populate("orders");
    console.log(result[0])
}
findCustomer();
```

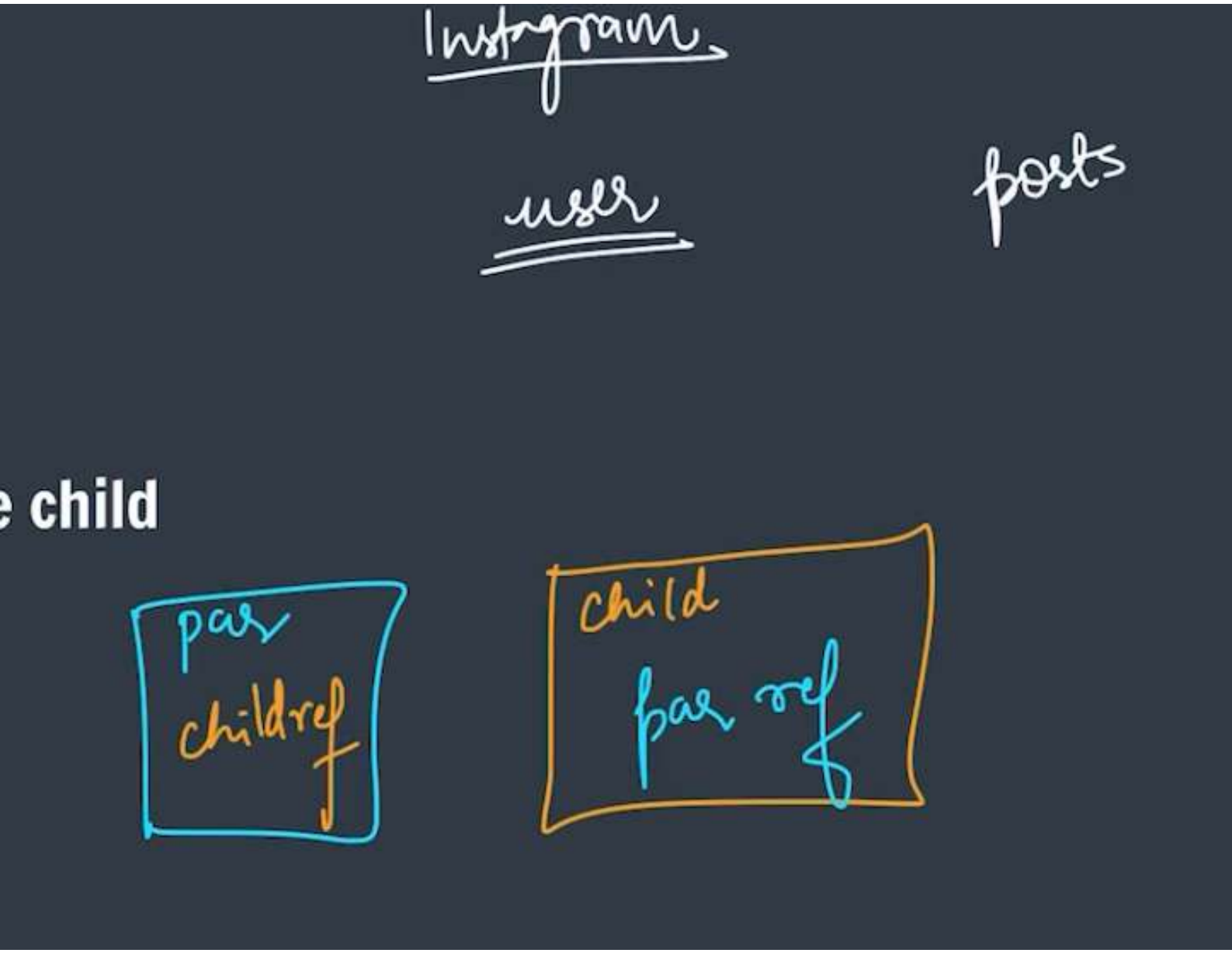
```
ashugoyal@Ashus-MacBook-Air models % node customer.js
Connection Successful
{
  _id: new ObjectId('67135abb433f016a17d984a3'),
  name: 'Rahul Kumar',
  orders: [
    {
      _id: new ObjectId('6713570f0a82561d86315e64'),
      item: 'Chips',
      price: 10,
      __v: 0
    },
    {
      _id: new ObjectId('6713570f0a82561d86315e65'),
      item: 'Pizza',
      price: 40,
      __v: 0
    }
  ],
  __v: 0
}
```

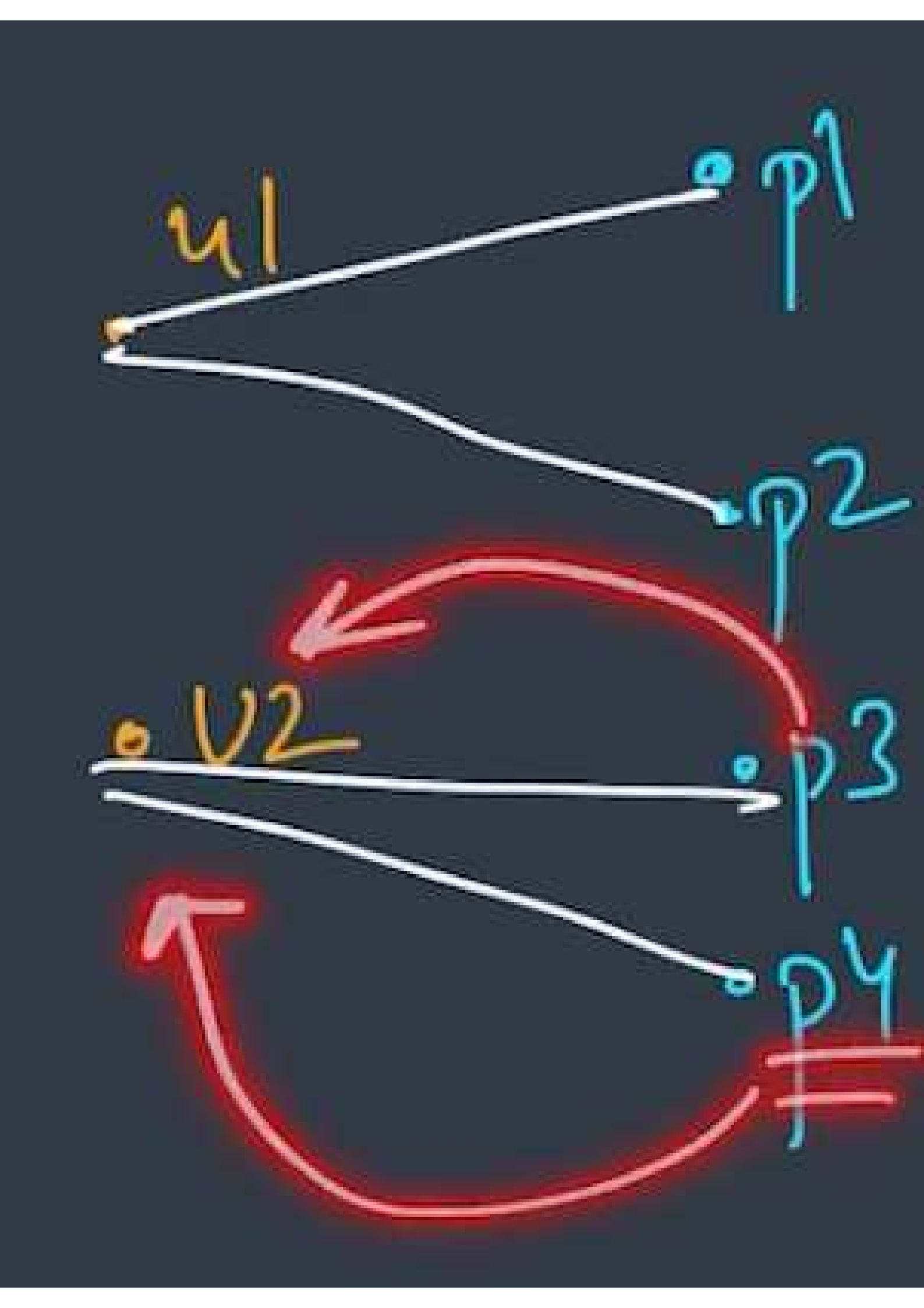
Mongo Relationships

One to Many / Approach 3 (one to squillions)

Store a reference to the parent document inside child

```
{
  _id: ObjectId("651d27deaadf315de08b7fa9"),
  content: 'Hello World!',
  likes: 7,
  user: ObjectId("651d27deaadf315de08b7fa8"),
  __v: 0
},
{
  _id: ObjectId("651d2852f213f39556fddeea"),
  content: 'Bye Bye',
  likes: 23,
  user: ObjectId("651d27deaadf315de08b7fa8"),
  __v: 0
}
```


Instagram
user
posts
e child
par
childref
child
par ref


u1
p1
p2
v2
p3
p4

```
const {Schema} = mongoose;
async function main(){
    await mongoose.connect("mongodb://127.0.0.1:27017/relationDemo")
}
main()
 .then(()=> console.log("Connection Successful"))
 .catch((err)=> console.log(err))

const userSchema = new Schema ({
    username: String,
    email: String,
})

const postSchema = new Schema ({
    content: String,
    likes: Number,
    user: {
        type: Schema.Types.ObjectId,
        ref:"User"
    }
})

const User = mongoose.model("User",userSchema)
const Post = mongoose.model("Post",postSchema)

const addData = async()=>{
    let User1 = new User({
        username: "John",
        email: "john@gmail.com"
    })

    let post1 = new Post({
        content: "Hello World",
        likes: 5,
    })

    post1.user = User1;
    let res = await User1.save();
    console.log(res);
    await post1.save();
}
addData();
```

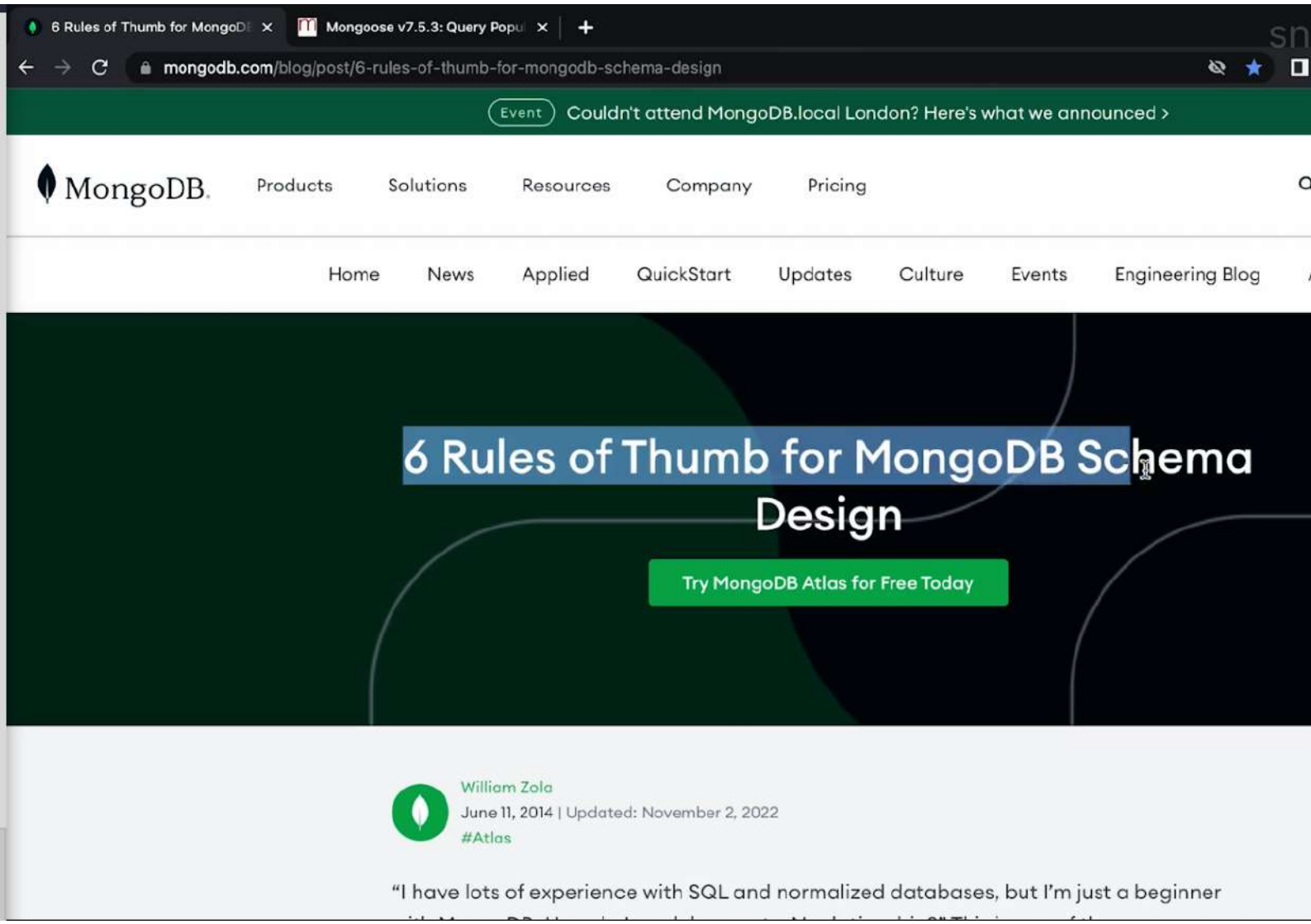
```
ashugoyal@Ashus-MacBook-Air models % node posts.js
Connection Successful
{
  username: 'John',
  email: 'john@gmail.com',
  _id: new ObjectId('67136160c41d987a4b17a61f'),
  __v: 0
}
```

```
const addData = async()=>{
    let user = await User.findOne({username: "John"})

    let post2 = new Post({
        content: "Bye Bye :)",
        likes: 10,
    })

    post2.user = user;
    await post2.save();
}
addData();
```

```
relationDemo> show collections
customers
orders
posts
users
```

```
relationDemo> db.posts.find()
[
  {
    _id: ObjectId('67136160c41d987a4b17a620'),
    content: 'Hello World',
    likes: 5,
    user: ObjectId('67136160c41d987a4b17a61f'),
    __v: 0
  },
  {
    _id: ObjectId('67136310b6359080b99f3159'),
    content: 'Bye Bye :)',
    likes: 10,
    user: ObjectId('67136160c41d987a4b17a61f'),
    __v: 0
  }
]
```

```
relationDemo> db.posts.find()
[
  {
    _id: ObjectId('67136160c41d987a4b17a620'),
    content: 'Hello World',
    likes: 5,
    user: ObjectId('67136160c41d987a4b17a61f'),
    __v: 0
  },
  {
    _id: ObjectId('67136310b6359080b99f3159'),
    content: 'Bye Bye :)',
    likes: 10,
    user: ObjectId('67136160c41d987a4b17a61f'),
    __v: 0
  }
]
```


6 Rules of Thumb for MongoDB
Mongoose v7.5.3: Query Popul
mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design
Event
Couldn't attend MongoDB.local London? Here's what we announced >
MongoDB.
Products
Solutions
Resources
Company
Pricing
Home
News
Applied
QuickStart
Updates
Culture
Events
Engineering Blog
6 Rules of Thumb for MongoDB Schema Design
Try MongoDB Atlas for Free Today
William Zola
June 11, 2014 | Updated: November 2, 2022
#Atlas
"I have lots of experience with SQL and normalized databases, but I'm just a beginner

Basics: Modeling one-to-few

An example of “one-to-few” might be the addresses for a person. This is a good use case for embedding. You’d put the addresses in an array inside of your Person object:

```
> db.person.findOne()
{
  name: 'Kate Monster',
  ssn: '123-456-7890',
  addresses : [
     { street: '123 Sesame St', city: 'Anytown', cc: 'USA' },
     { street: '123 Avenue Q', city: 'New York', cc: 'USA' }
  ]
}
```

This design has all of the advantages and disadvantages of embedding. The main advantage is that you don’t have to perform a separate query to get the embedded details; the main disadvantage is that you have no way of accessing the embedded details as stand-alone entities.

Basics: One-to-squillions

An example of “one-to-squillions” might be an event logging system that collects log messages for different machines. Any given host could generate enough messages to overflow the 16 MB document size, even if all you stored in the array was the ObjectID. This is the classic use case for “parent-referencing.” You’d have a document for the host, and then store the ObjectID of the host in the documents for the log messages.

```
> db.hosts.findOne()
{
    _id : ObjectID('AAAB'),
    name : 'goofy.example.com',
    ipaddr : '127.66.66.66'
}

>db.logmsg.findOne()
{
    time : ISODate("2014-03-28T09:42:41.382Z"),
    message : 'cpu is on fire!',
    host: ObjectID('AAAB')       // Reference to the Host document
}
```

Here are some "rules of thumb" to guide you through these innumerable (but not infinite) choices:

- **One:** Favor embedding unless there is a compelling reason not to.
- **Two:** Needing to access an object on its own is a compelling reason not to embed it.
- **Three:** Arrays should not grow without bound. If there are more than a couple of hundred documents on the "many" side, don't embed them; if there are more than a few thousand documents on the "many" side, don't use an array of ObjectID references. High-cardinality arrays are a compelling reason not to embed.
- **Four:** Don't be afraid of application-level joins: If you index correctly and use the projection specifier, then application-level joins are barely more expensive than server-side joins in a relational database.
- **Five:** Consider the read-to-write ratio with denormalization. A field that will mostly be read and only seldom updated is a good candidate for denormalization. If you denormalize a field that is updated frequently then the extra work of finding and updating all the instances of redundant data is likely to overwhelm the savings that you get from denormalization.
- **Six:** As always with MongoDB, how you model your data depends entirely on your particular application's data access patterns. You want to structure your data to match the ways that your application queries and updates it.

array

APNA COLLEGE

one to many

< 100s : embed

> 100s : array objId references

> 1000s : parent

denormalization

↓

storing copy / duplicate