

# Turn on your MongoDB



# NoSQL and MongoDB



# Agenda

- Opsamling: Node.js og ORM
- NoSQL og MongoDB
- Collections og Documents
- Implementering af MongoDB CRUD-operationer og andre queries
- Relationer og datamodellering i MongoDB
- Evt. indeksering, søgning og optimering af queries

# Types

There are various types of databases, including relational databases (such as MySQL, PostgreSQL, and Oracle), NoSQL databases (like MongoDB and Cassandra), and other specialized databases tailored to specific use cases.



ORACLE





# Most Popular Databases



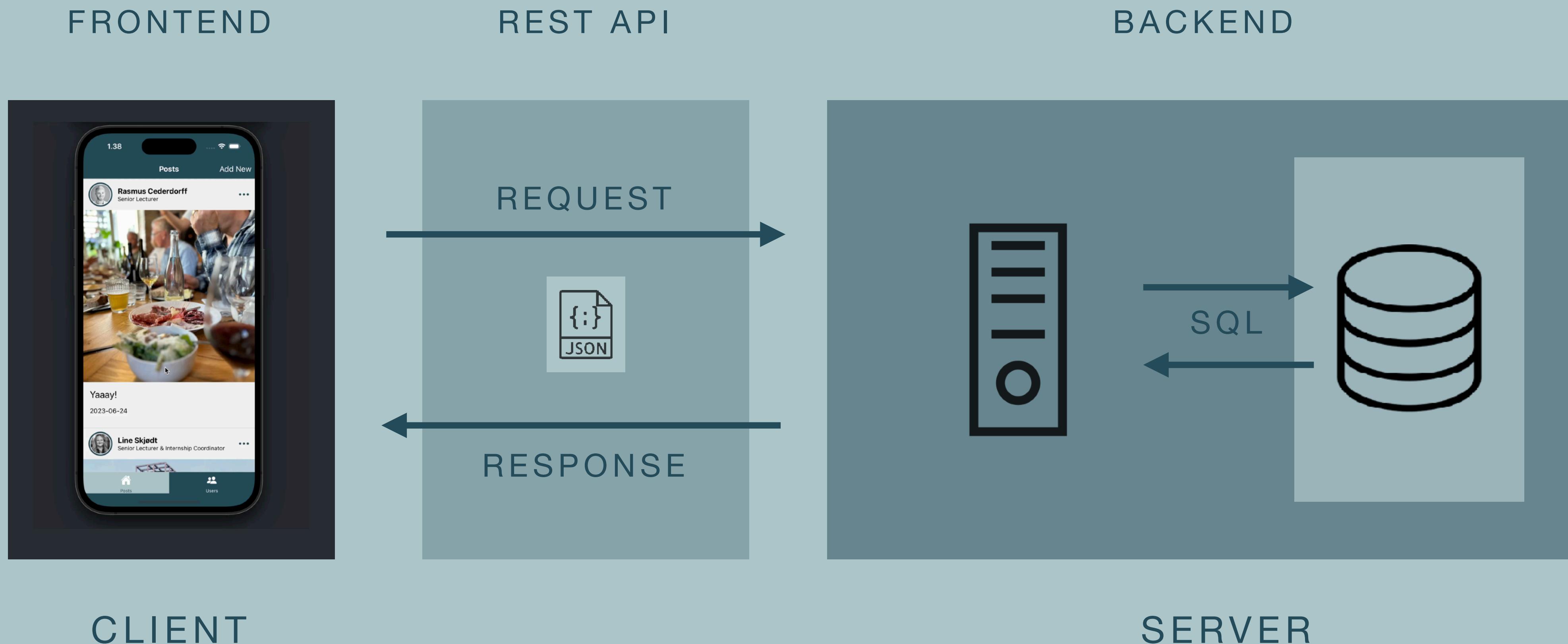
<https://survey.stackoverflow.co/2023/#section-most-popular-technologies-databases>

# Most Popular Databases



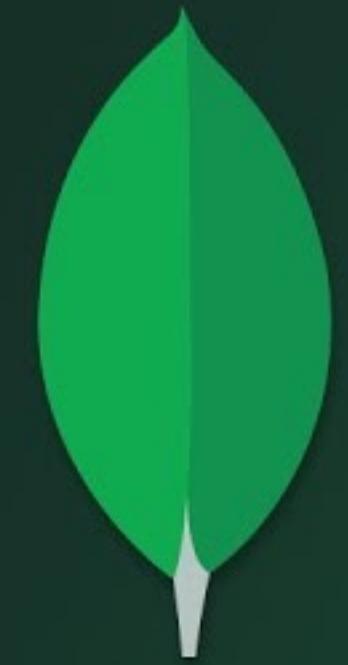
<https://survey.stackoverflow.co/2023/#section-most-popular-technologies-databases>

# Web Dev Architecture



**100 *SECONDS OF***

---



**mongoDB**

[https://www.youtube.com/watch?v=-bt\\_y4Loofg](https://www.youtube.com/watch?v=-bt_y4Loofg)

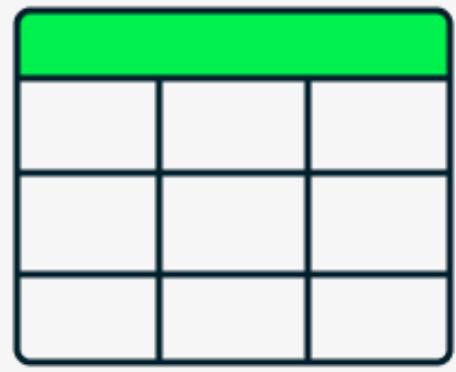
# Non-Relational Databases





# Where it Began: Relational

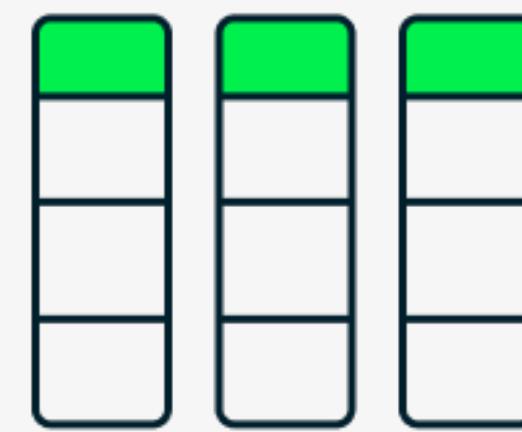
Key features of relational databases



Related data is stored in rows and columns in one table.



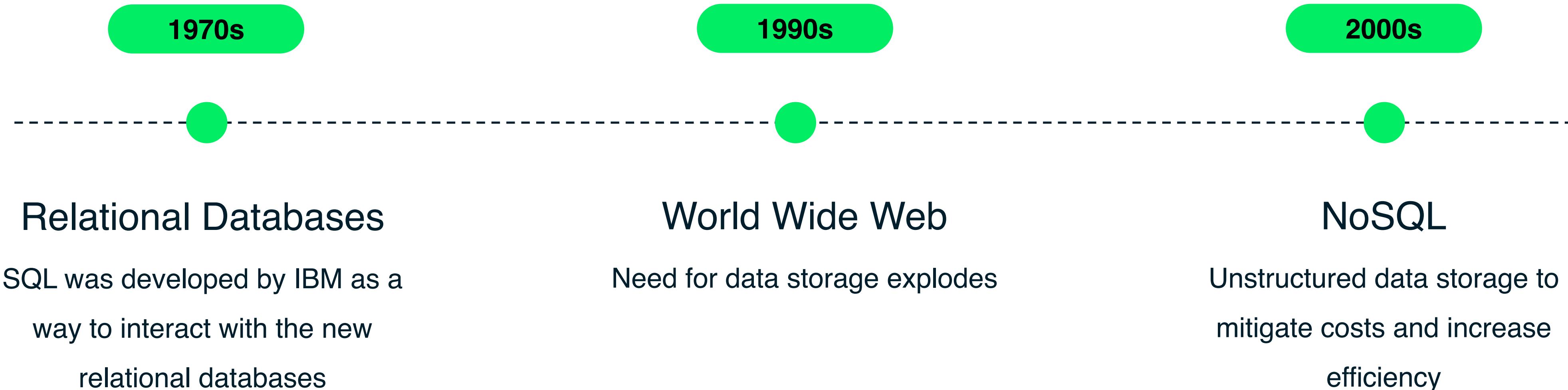
SQL (Structured Query Language)



A table uses columns to define the information being stored and rows for the actual data.



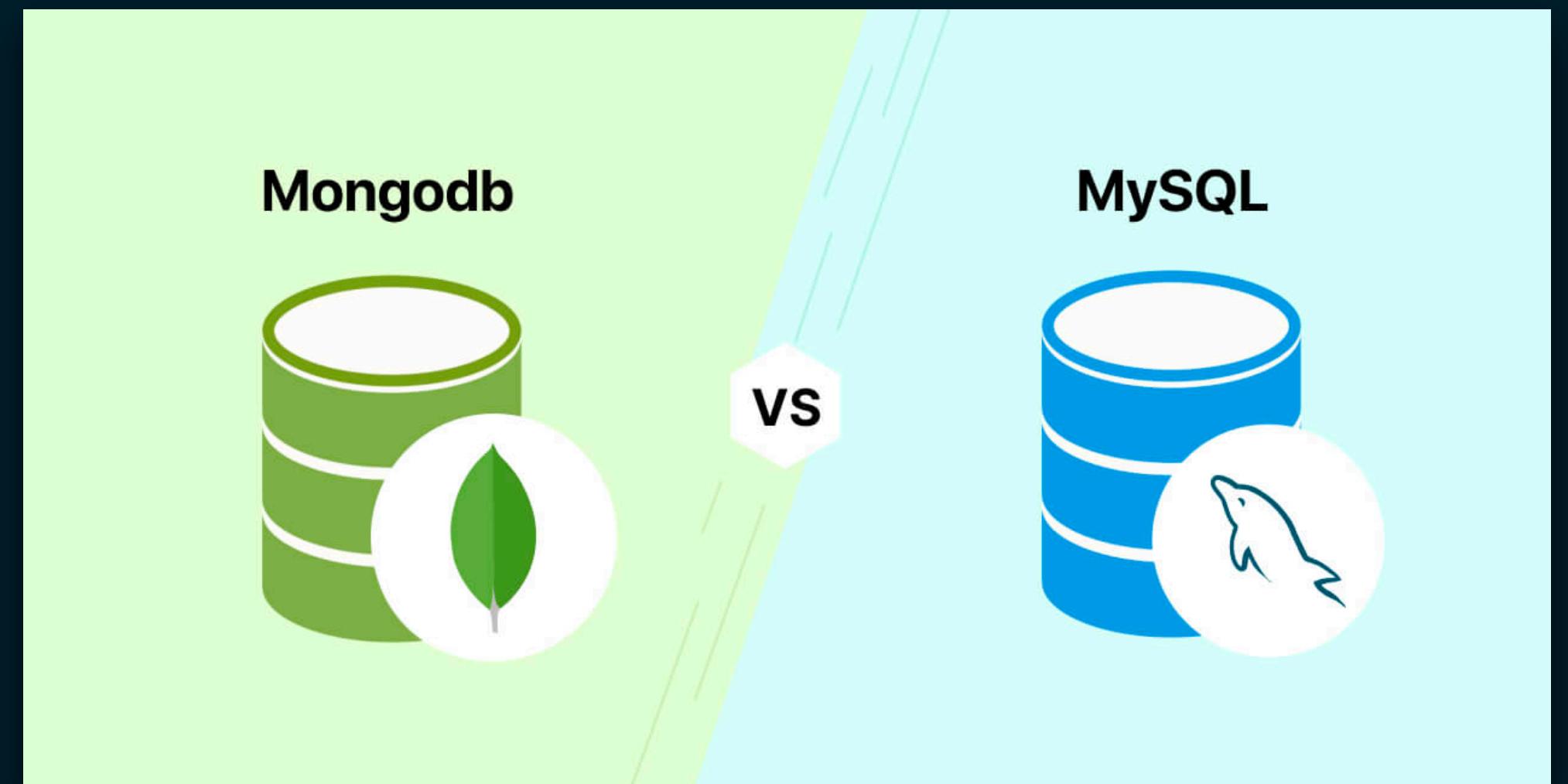
# Filling in the Gap





# Relational Model: Challenges

- Issues with scalability and flexibility
- Not all data is structured
- Object-oriented, tables and columns?





```
const user = {  
  id: 6,  
  name: "Murat Kilic",  
  mail: "mki@eaaa.dk",  
  title: "Senior Lecturer",  
  image: "https://www.eaaa.dk/media/llyavasj/murat-kilic.jpg"  
};
```

id	name	mail	title	image
1	Peter Lind	petl@kea.dk	Senior Lecturer	<a href="https://share.cederdorff.com/i">https://share.cederdorff.com/i</a>
2	Rasmus Cederdorff	race@dev.dk	Senior Lecturer	<a href="https://share.cederdorff.com/i">https://share.cederdorff.com/i</a>
3	Lars Bogetoft	larb@eaaa.dk	Head of Education	<a href="https://kea.dk/slir/w200-c1x1/">https://kea.dk/slir/w200-c1x1/</a>
4	Edith Terte	edan@kea.dk	Lecturer	<a href="https://media.licdn.com/dms/im">https://media.licdn.com/dms/im</a>
5	Frederikke Bender	fbe@kea.dk	Head of Education	<a href="https://kea.dk/slir/w200-c1x1/">https://kea.dk/slir/w200-c1x1/</a>
6	Murat Kilic	mki@eaaa.dk	Senior Lecturer	<a href="https://www.eaaa.dk/media/llyavasj/murat-kilic.jpg">https://www.eaaa.dk/media/llyavasj/murat-kilic.jpg</a>
7	Anne Kirketerp	anki@eaaa.dk	Head of Education	<a href="https://www.eaaa.dk/media/5buhs">https://www.eaaa.dk/media/5buhs</a>



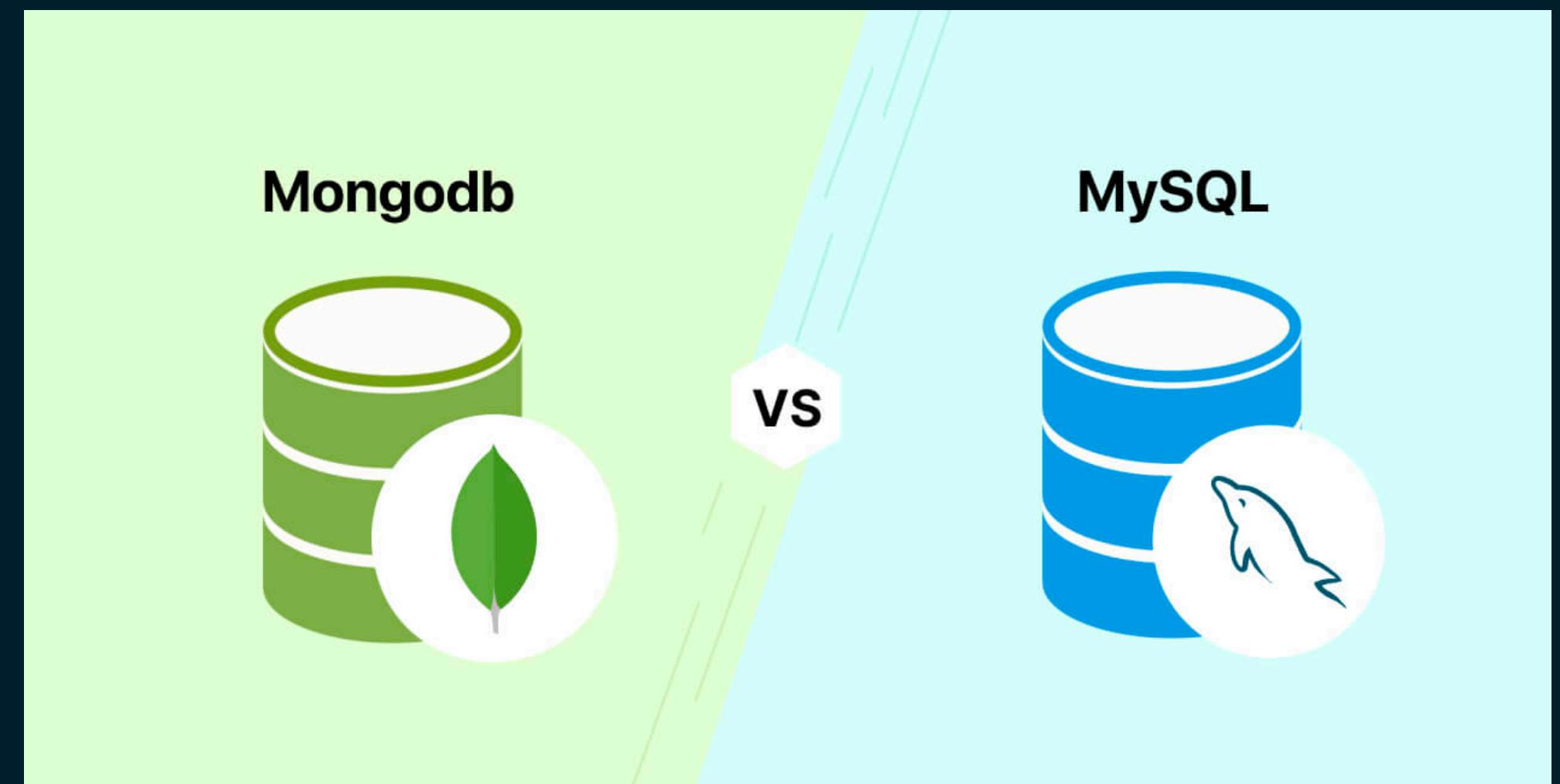
From object to database table? WTF!1!





# But Why?

- **Data Flexibility:** MongoDB handles unstructured data well, while MySQL is structured.
- **Schema:** MongoDB is schema-less, allowing for dynamic data, while MySQL requires a predefined schema.
- **Scalability:** MongoDB is horizontally scalable, beneficial for large datasets and high traffic. MySQL may face challenges with horizontal scalability.
- **Use Cases:** MongoDB suits document-oriented and complex data structures. MySQL is often preferred for traditional relational data.
- **Development Speed:** MongoDB's flexibility can accelerate development for evolving applications.
- **Specific Application Needs:** Choose based on project requirements, considering data complexity and scalability.



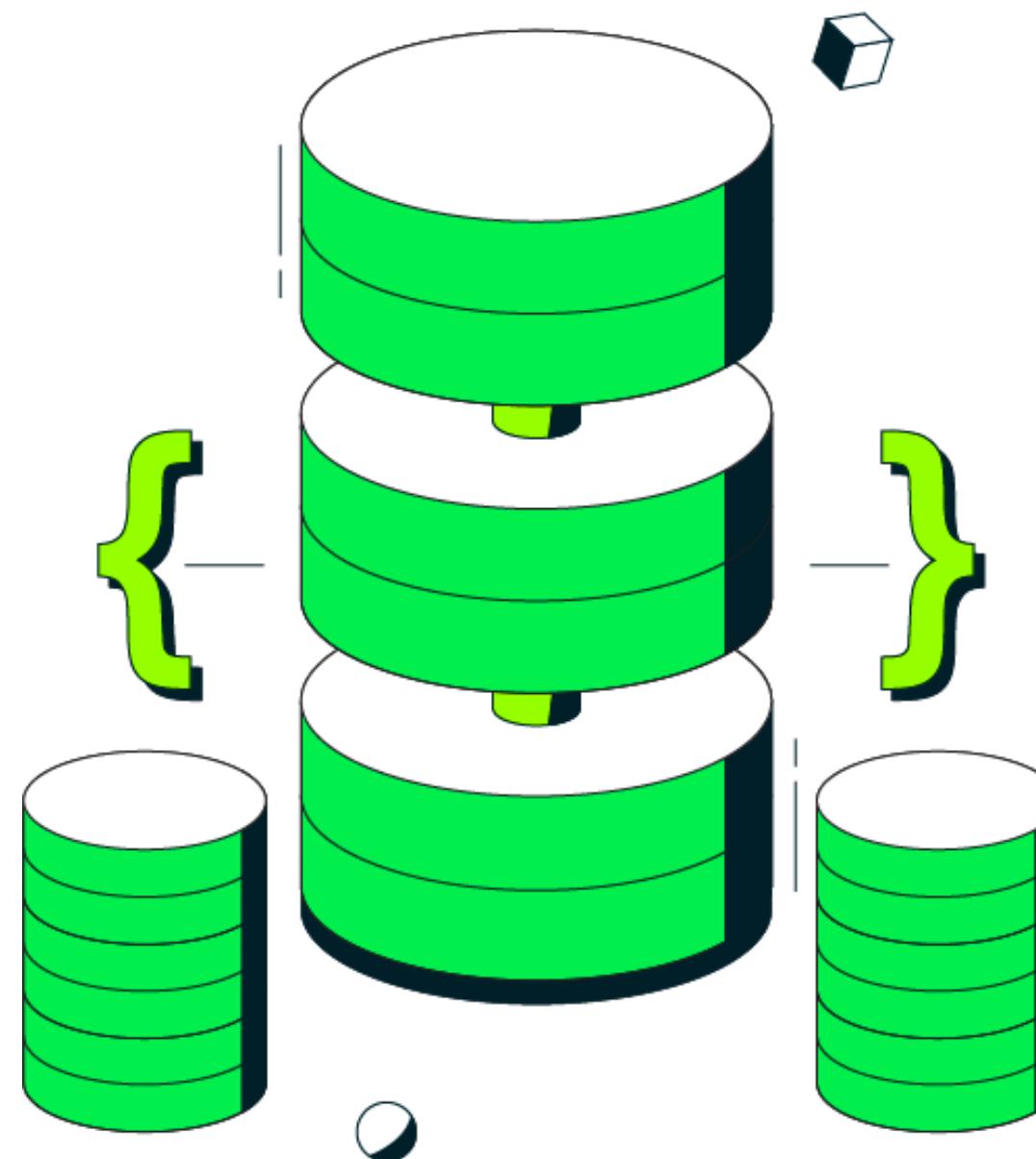


To fix the problem, various technology and software companies introduced new databases referred to as **NoSQL** or **non-relational**.



- Polymorphic data structures
- Flexible schemas
- Easy to scale large workloads

What is a non-relational database?

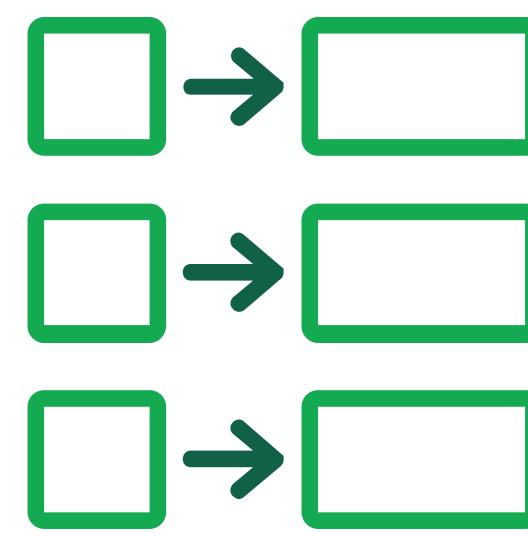


# Non-Relational Database Types





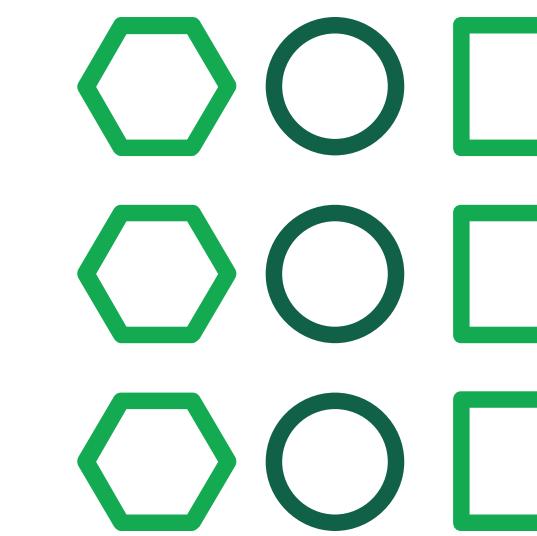
# Non-Relational Database Types



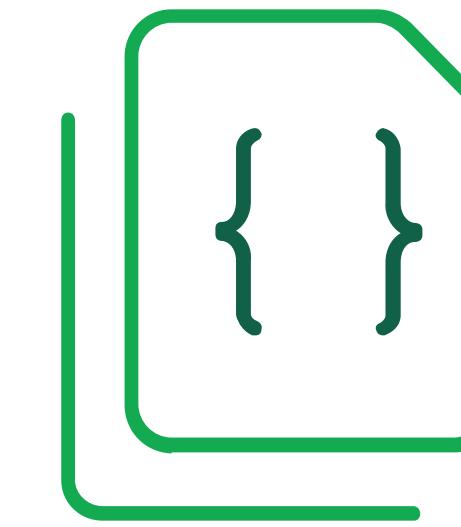
Key/Value



Graph



Column



Document

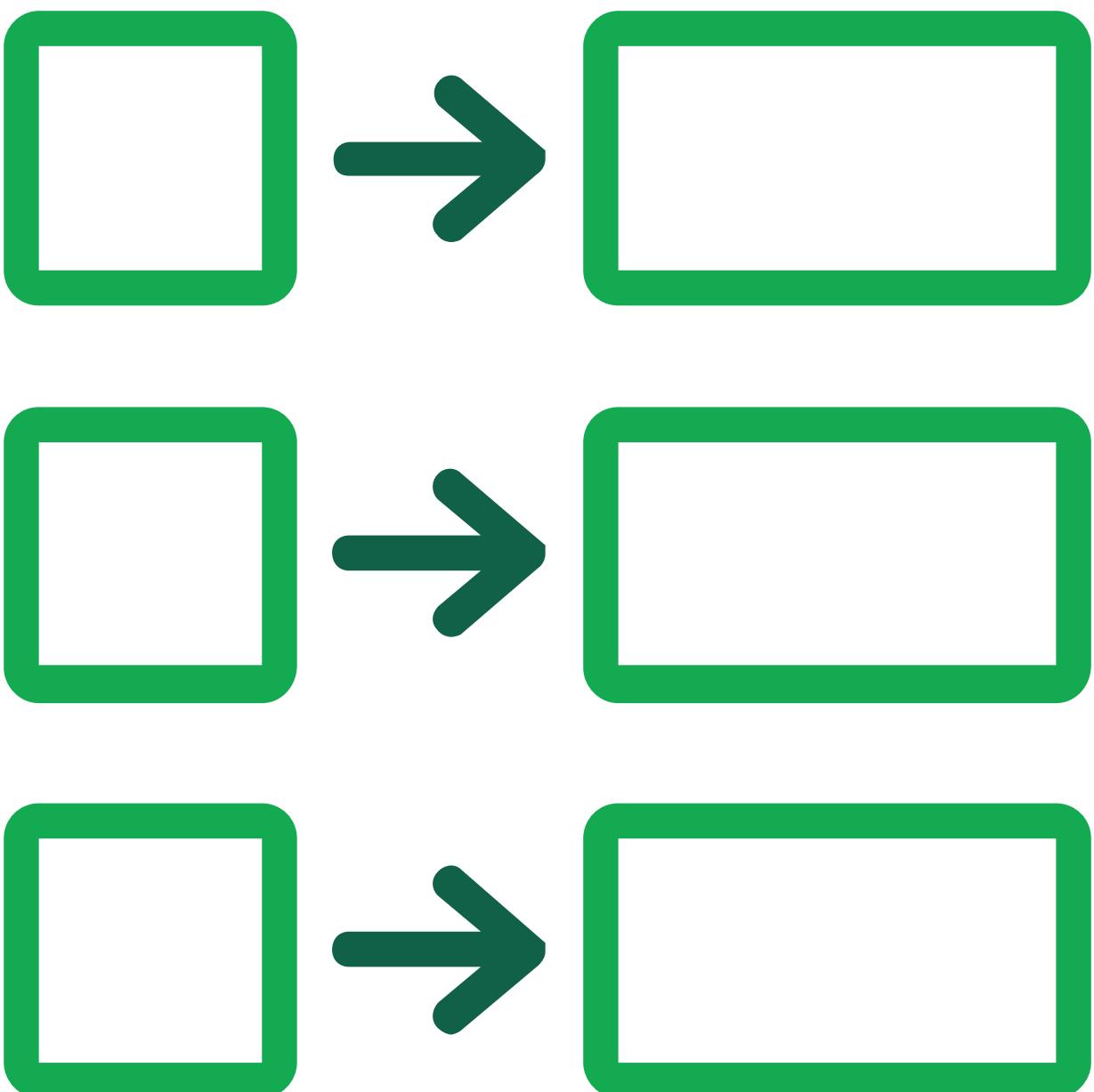


## Structure

- A unique key is paired with a collection of values, where the values can be anything from a string to a large binary object

## Strength

- Simple data model



**Key/Value Database**



# KeyValue: Example

Key	Value
Name	Sherlock Holmes
Age	40
Address	221B Baker Street



## Structure

- Captures connected data
- Each element is stored as a node
- Connections between nodes are called links or relationships

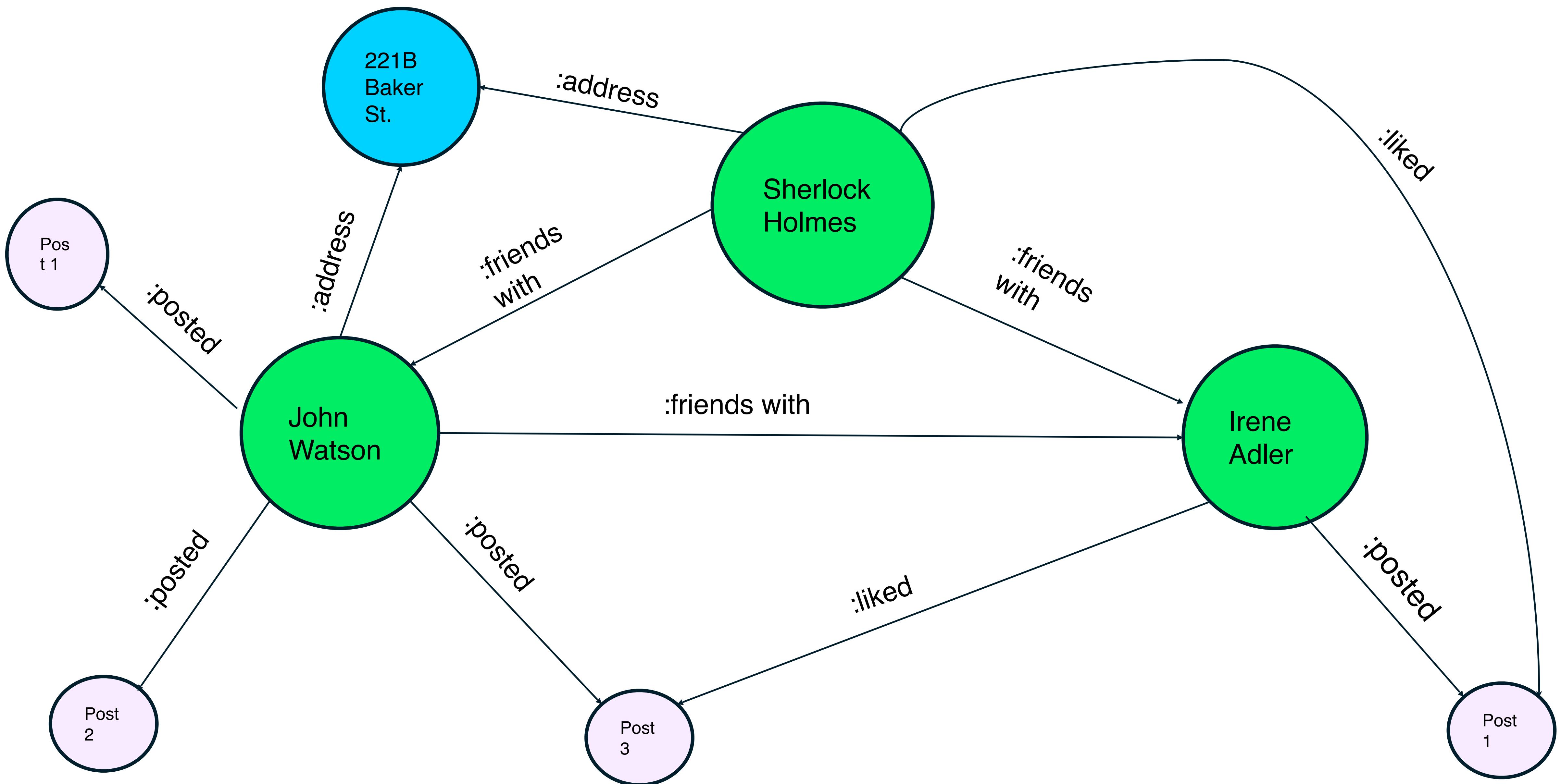
## Strength

- Traverses the connections between data rapidly



Graph Database

# Graph: Example



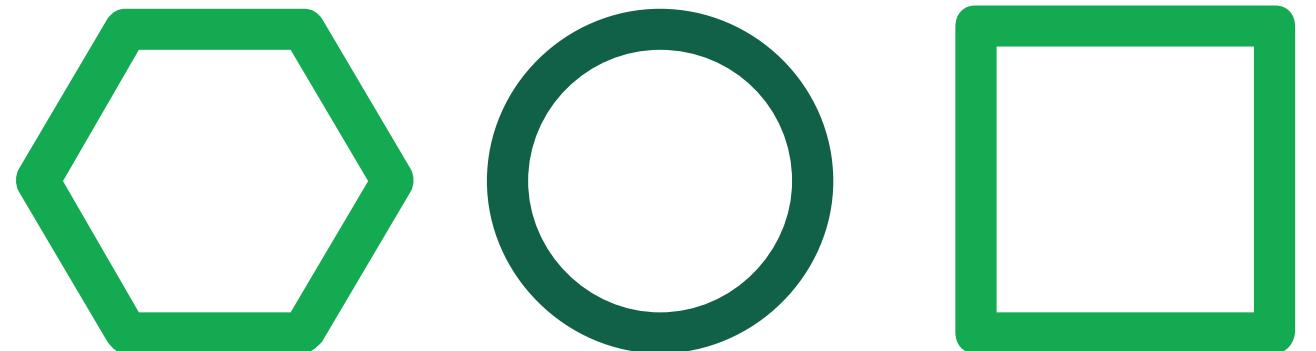
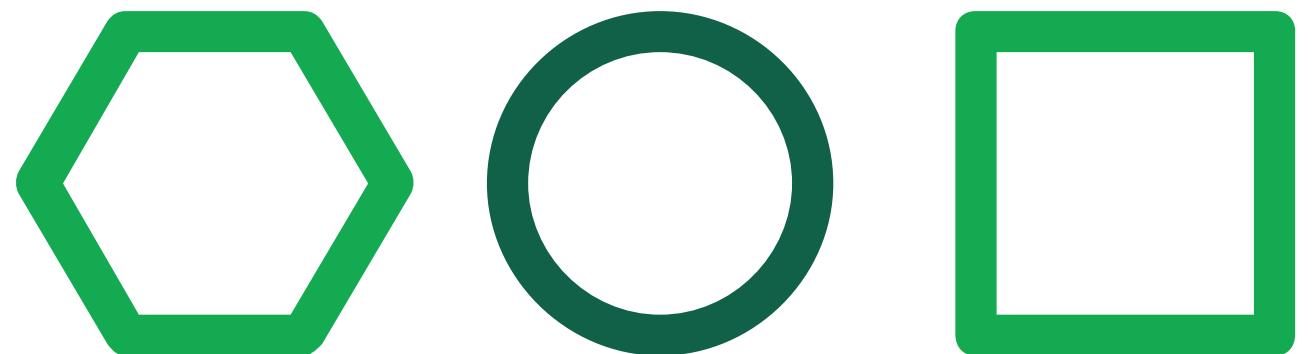
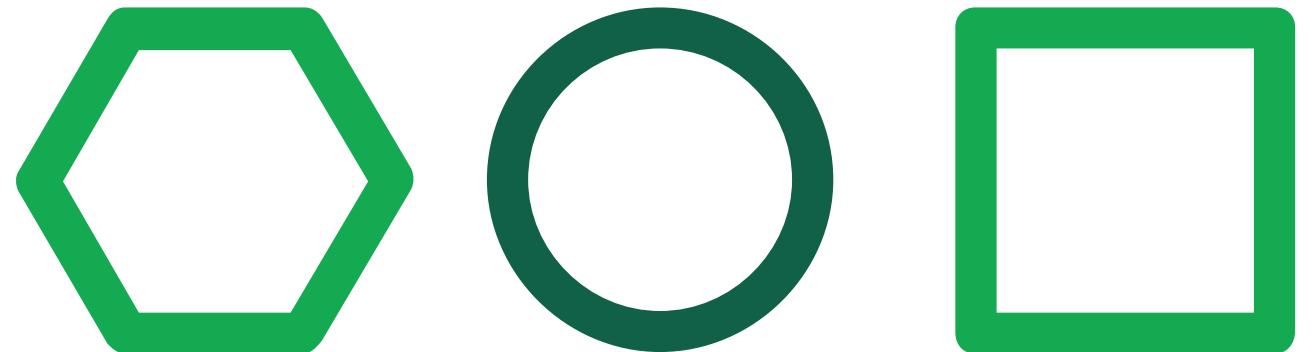


## Structure

- Data is stored using key rows that can be associated with one or more dynamic columns

## Strengths

- Highly performant queries
- Designed for analytics



Column Oriented  
or Wide Column



## Column Oriented Example

Name	ID
Sherlock	001
John	002
Irene	003

Age	ID
40	001
45	002
43	003

Height	ID
6'2	001
5'9	002
5'7	003

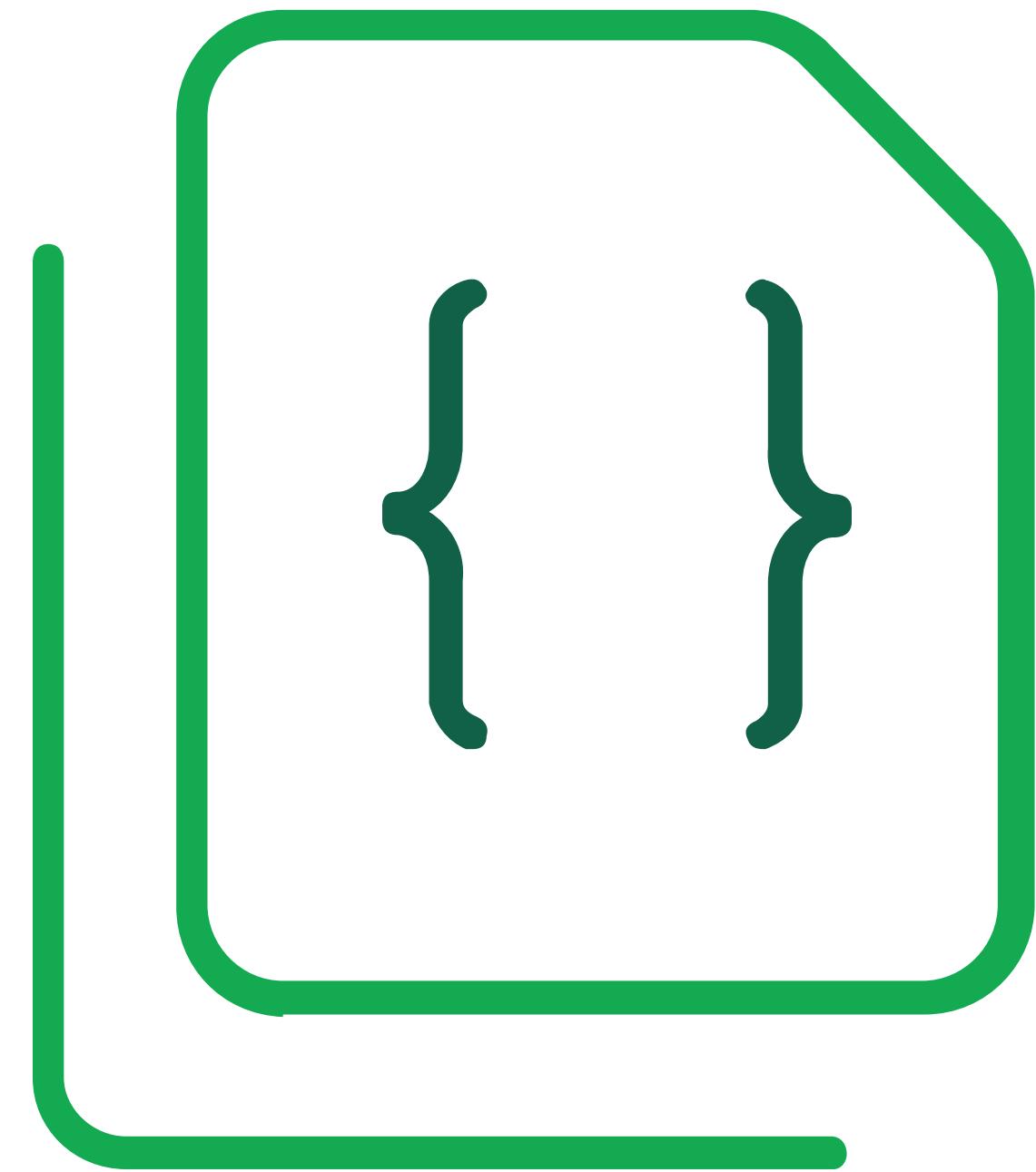


## Structure

- Polymorphic data models
- Each document contains markup that identifies fields and values

## Strengths

- Obvious relationships using embedded arrays and documents
- No complex mapping



Document Database



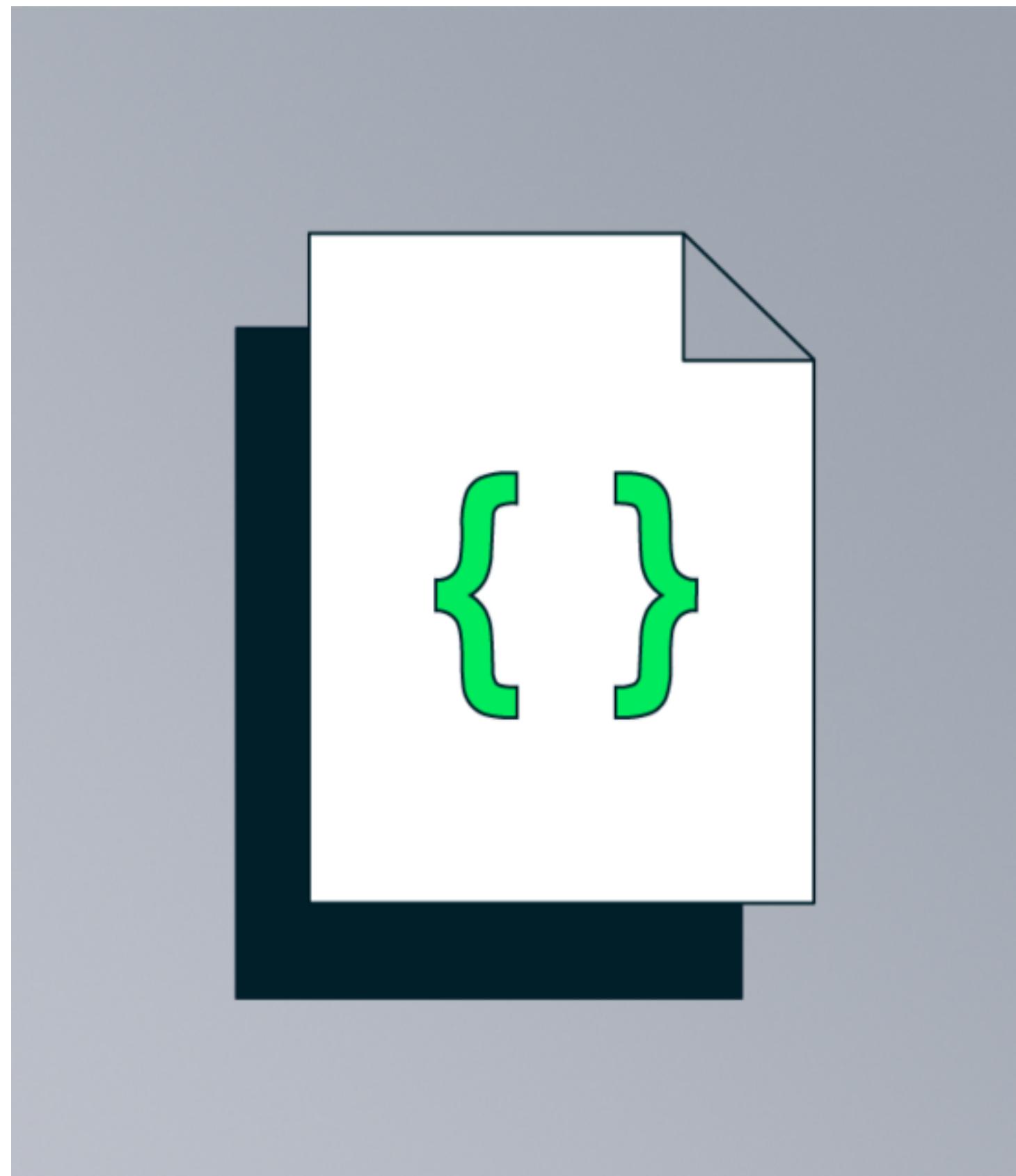
# Document Model Example

```
{  
    "_id":  
        ObjectId("5ef2d4b45b7f11b6d7a"),  
    "user_id": "Sherlock  
Holmes",  
    "age": 40,  
    "address":  
        {  
            "Country": "England"  
            "City": "London",  
            "Street": "221B Baker St."  
        },  
    "Hobbies": [ violin, crime-  
solving ]  
}
```

```
{  
    "_id":  
        ObjectId("6ef8d4b32c9f12b6d4a"),  
    "user_id": "John Watson",  
    "age": 45,  
    "address":  
        {  
            "Country": "England"  
            "City": "London",  
            "Street": "221B Baker St."  
        },  
    "Medical license": "Active"  
}
```



# The Document Model



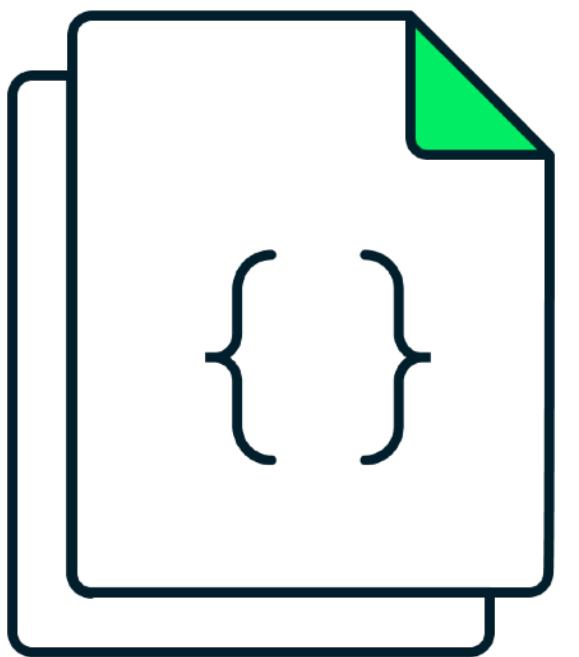
For general purpose use, the document model prevails as the preferred model by developers and database administrators.

# Collections and Documents

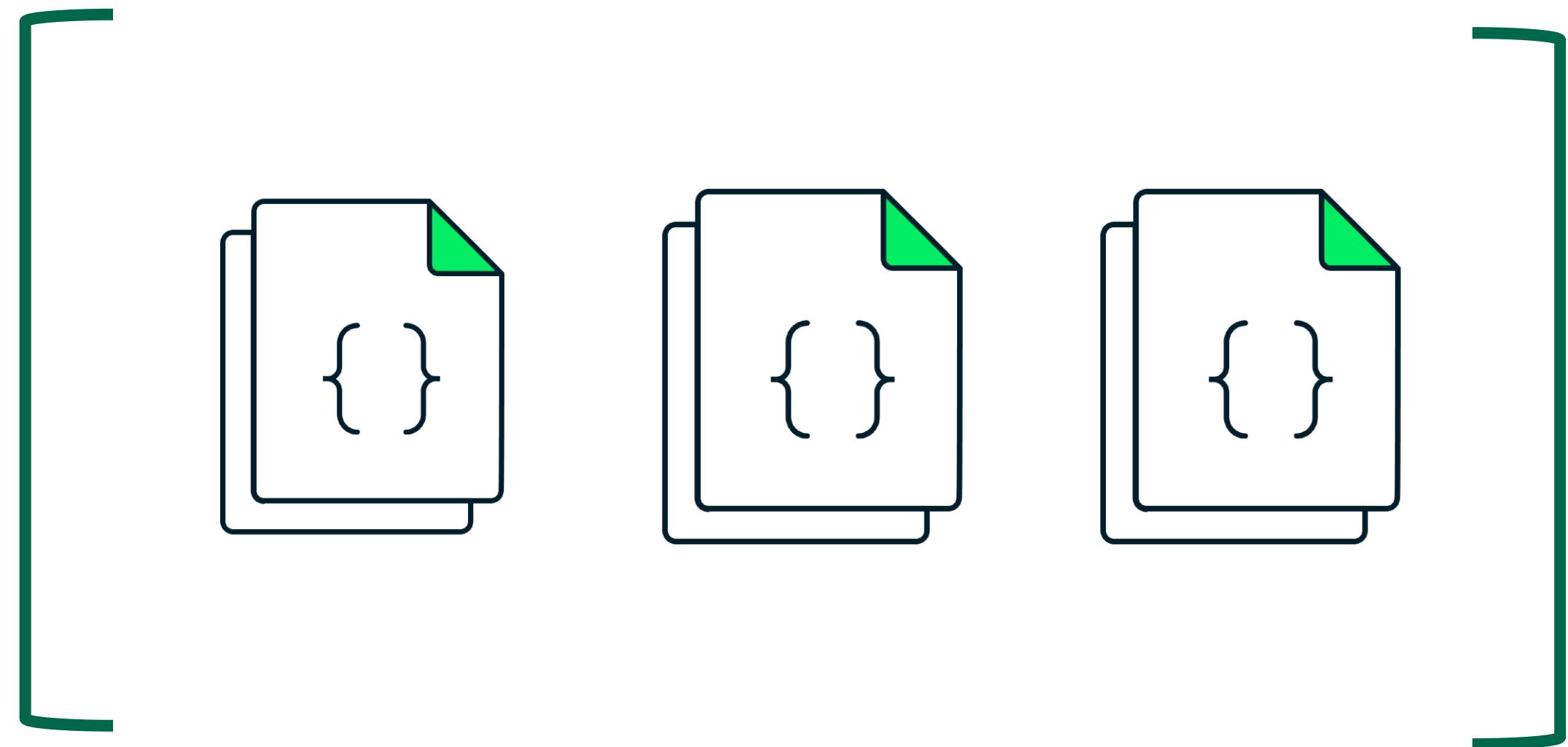




# Document



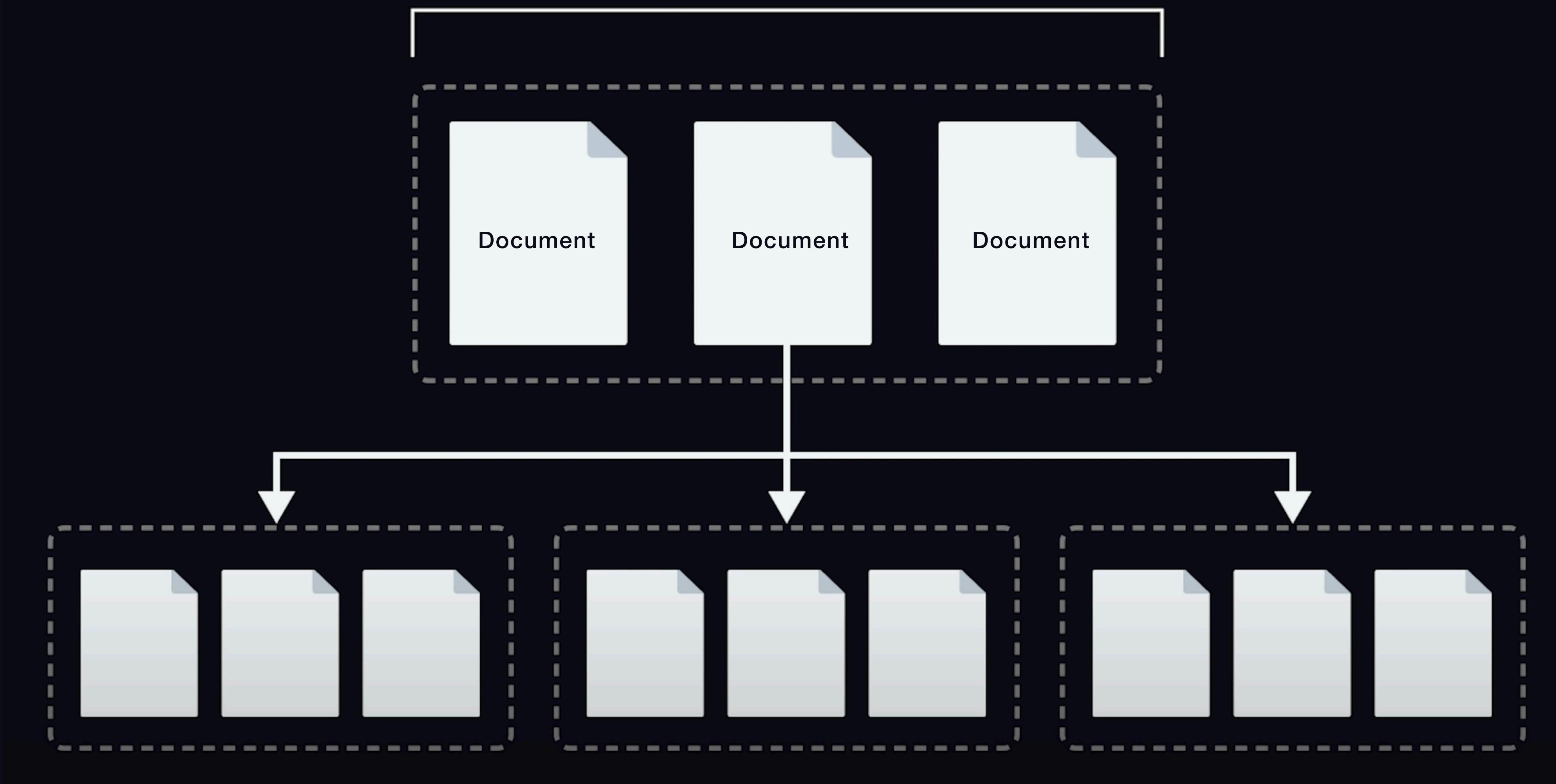
# Collection



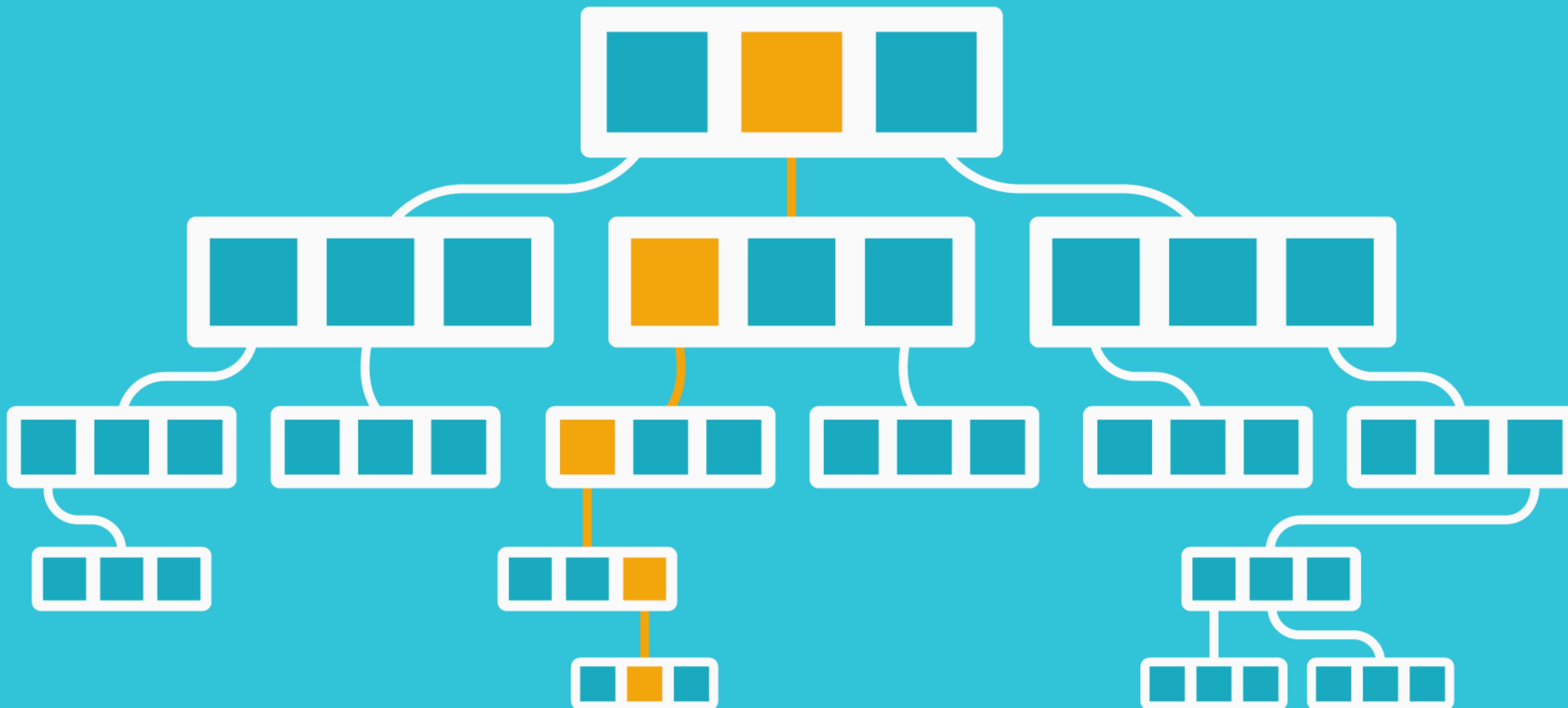
A way to organize and store data as a set of field-value pairs in MongoDB.

An organized store of documents in MongoDB, usually with common fields between documents

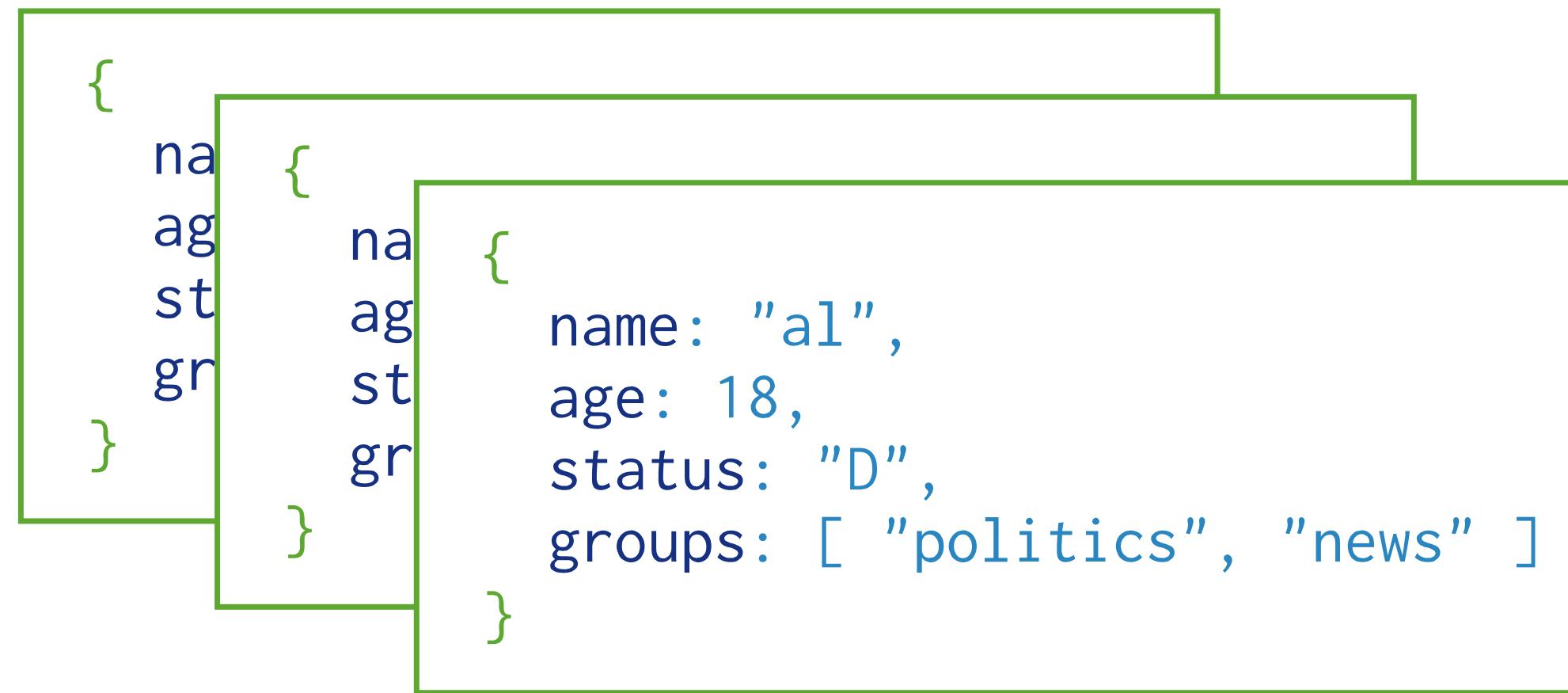
## Collection



# Collections and Documents



# Collections and Documents

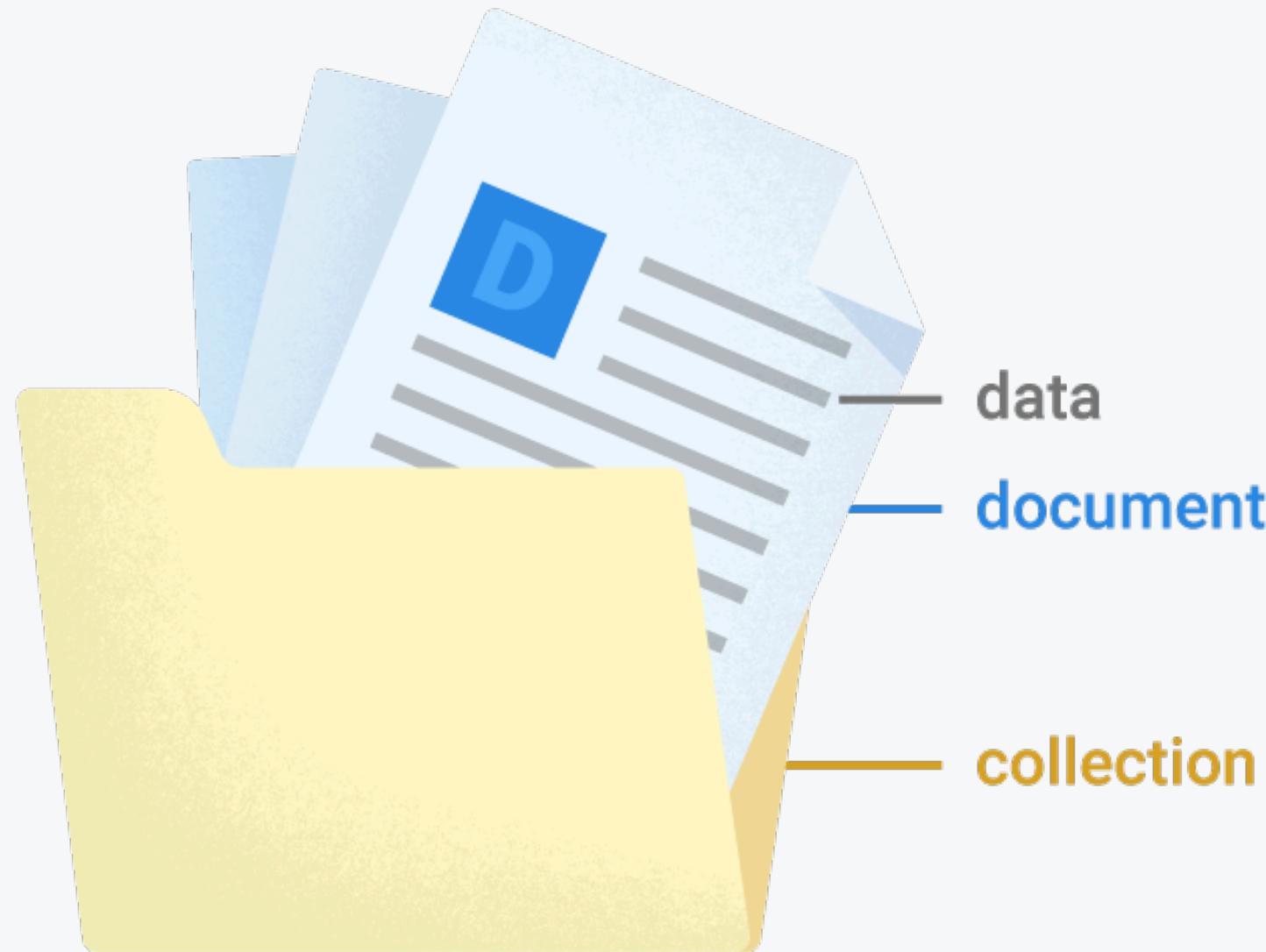


Collection

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

← field: value  
← field: value  
← field: value  
← field: value

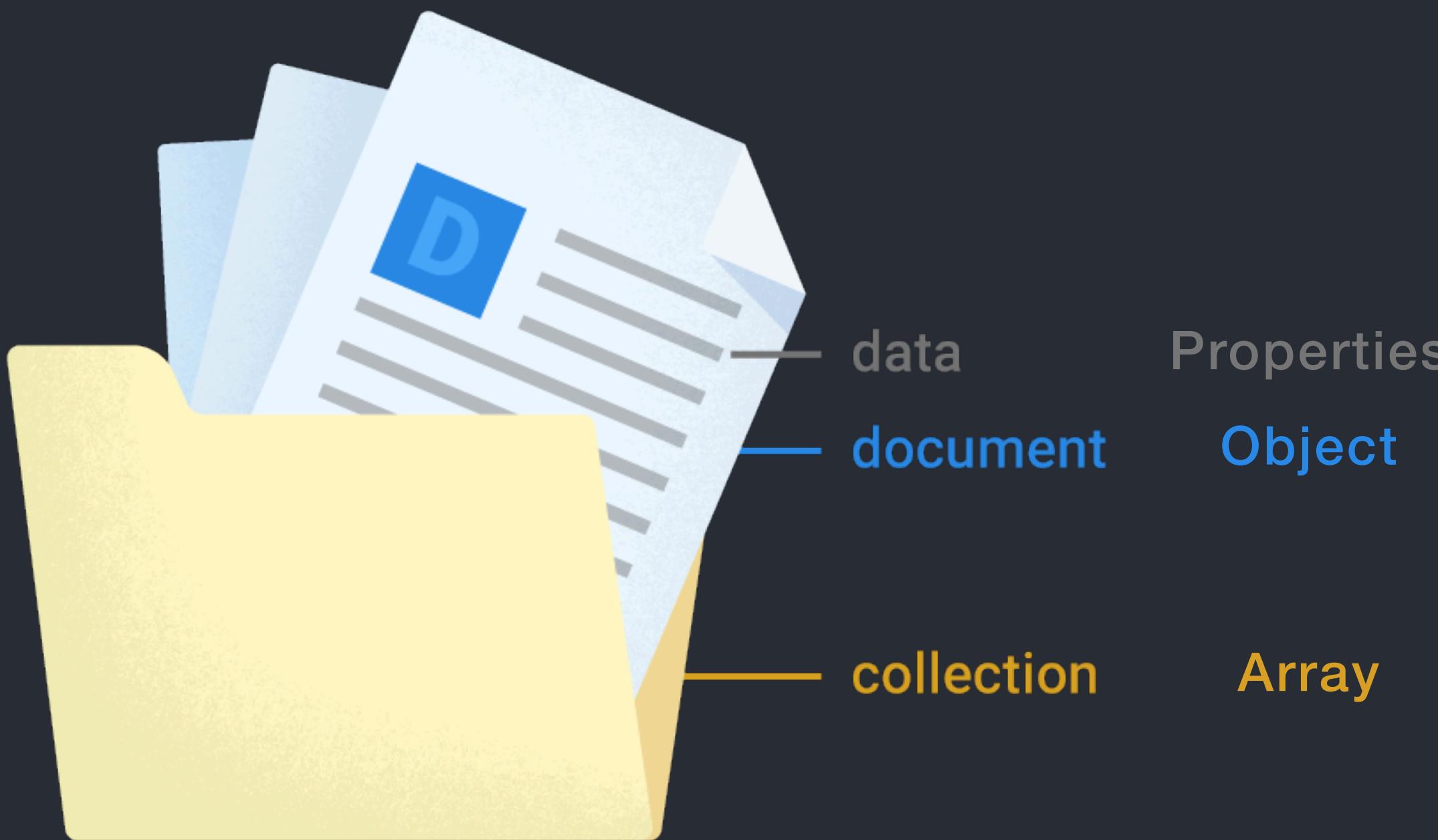
Document



Home > users > seme0qoWIzQh...

mdu-e18front	users	seme0qoWIzQhZDfI1m0L
+ Start collection	+ Add document	+ Start collection
users >	IS35ivFQ0BBYr0bwZX7k cNzvoH8XSRib6XXiUYuX seme0qoWIzQhZDfI1m0L >	+ Add field  mail: "bki@eaaa.dk" name: "Birgitte"

# Collection & document vs array & object



Properties

Object

Array

```
const users = [
  {
    id: 1,
    name: "Rasmus Cederdorff",
    title: "Senior Lecturer",
    mail: "race@eaaa.dk",
    image: "https://share.cederdorff.com/images/race.jpg"
  },
  {
    id: 2,
    name: "Anne Kirketerp",
    title: "Head of Department",
    mail: "anki@eaaa.dk",
    image: "https://www.eaaa.dk/media/5buh1xeo/anne-kirke
  },
  {
    id: 3,
    name: "Murat Kilic",
    title: "Senior Lecturer",
    mail: "mki@eaaa.dk",
    image: "https://www.eaaa.dk/media/llyavasj/murat-kili
  }
];
```

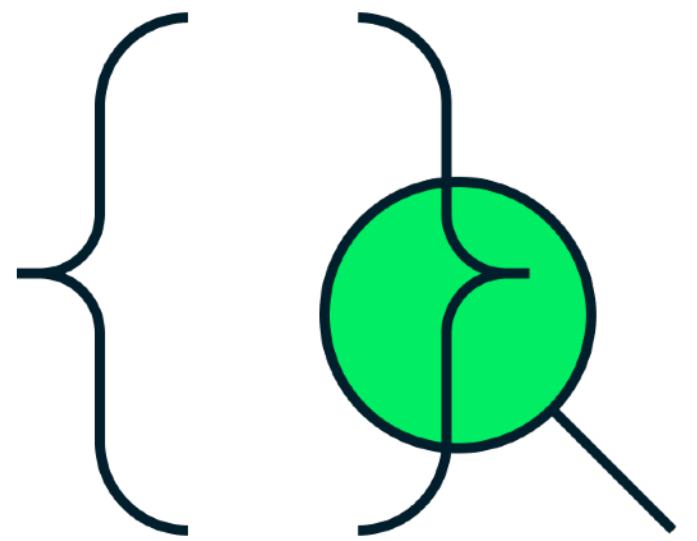
```
1  {
2      "first_name": "Paul",
3      "surname": "Miller",
4      "cell": "447557505611",
5      "city": "London",
6      "location": [45.123, 47.232],
7      "profession": ["banking", "finance", "trader"],
8      "cars": [
9          {
10             "model": "Bentley",
11             "year": 1973
12         },
13         {
14             "model": "Rolls Royce",
15             "year": 1965
16         }
17     ]
18 }
```

# The Document Model and MongoDB

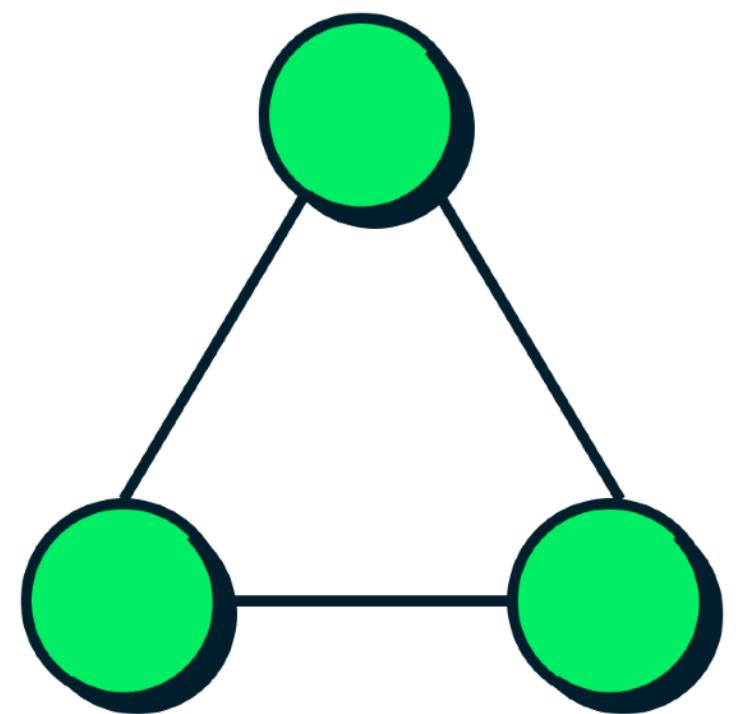




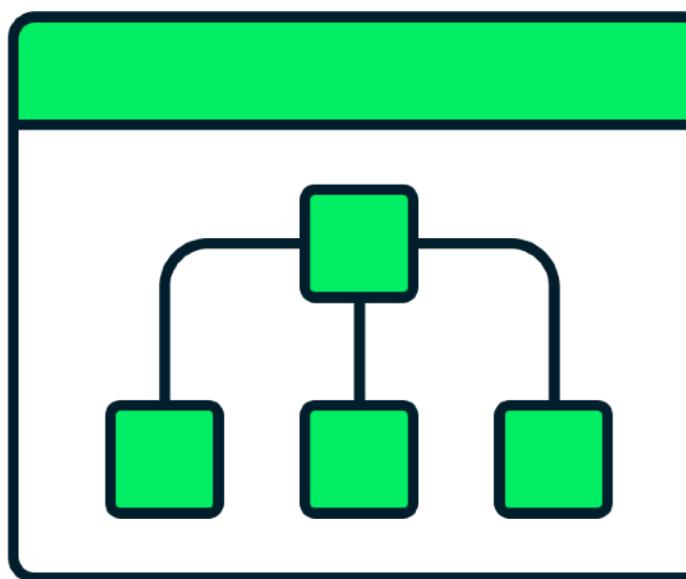
# Key Features



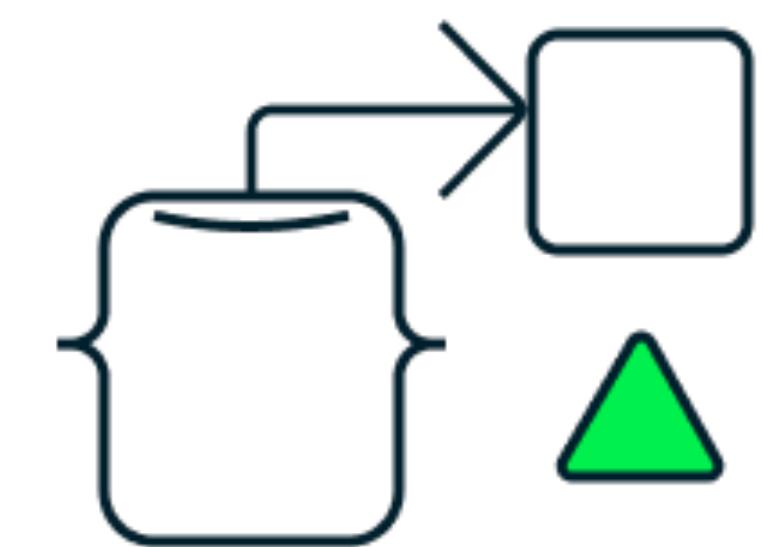
API query or query language



Distributed and resilient



Flexible schema



Object mapping



```
{  
  "_id": ObjectId(  
    "5f4f7fef2d4b45b7f11b6d7a") ,  
  "user_id": "Sean",  
  "age": 29,  
  "Status": "A"  
}
```

## The Document Model: Structure and Syntax

- To the left is an example of a document representing a user details including `user_id`, `age`, and a status `category`.



```
{  
  "_id": ObjectId(  
    "5f4f7fef2d4b45b7f11b6d7a") ,  
  "user_id": "Sean",  
  "age": 29,  
  "Status": "A"  
}
```

## The Document Model: Structure and Syntax

- A document in MongoDB uses the JavaScript Object Notation (JSON) format.
- This format uses curly brackets to mark the start and the end of the document.



```
{  
  "_id": ObjectId(  
    "5f4f7fef2d4b45b7f11b6d7a" ),  
  "user_id": "Sean",  
  "age": 29,  
  "Status": "A"  
}
```

# The Document Model: Structure and Syntax

- MongoDB refers to keys as fields.
- The field-values within a pair in a document are separated by colons (:).



```
{  
  "_id": ObjectId(  
    "5f4f7fef2d4b45b7f11b6d7a" ),  
  "user_id": "Sean",  
  "age": 29,  
  "Status": "A"  
}
```

## The Document Model: Structure and Syntax

- Each field must be enclosed within quotation marks. String values are often quoted as good practice.



```
{  
  "_id": ObjectId(  
    "5f4f7fef2d4b45b7f11b6d7a" ),  
  "user_id": "Sean",  
  "age": 29,  
  "Status": "A"  
}
```

# The Document Model: Structure and Syntax

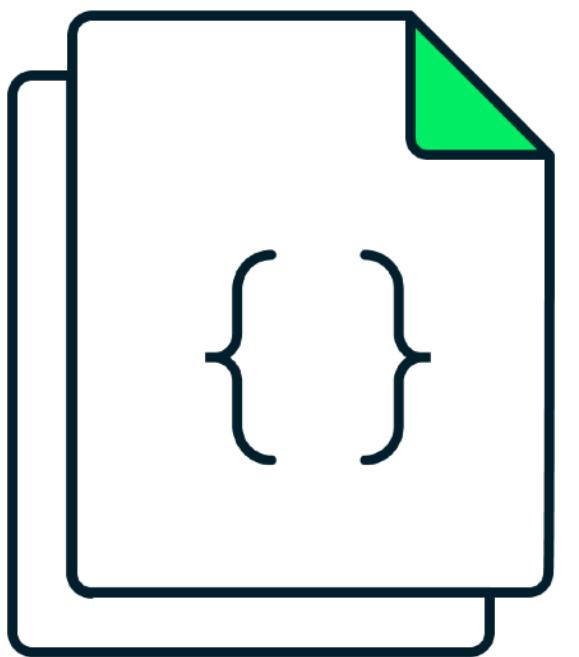
- Each field-value pair is separated within the document by commas.



# Collections in the Document Model



# Document



# Collection



A way to organize and store data as a set of field-value pairs in MongoDB.

An organized store of documents in MongoDB, usually with common fields between documents



# Example

Two documents in the same collection  
but with different fields...

```
{  
  "_id": ObjectId(  
    "5f4f7fef2d4b45b7f11b6d7a") ,  
  "user_id": "Sean",  
  "age": 29,  
  "Status": "A"  
}
```

```
{  
  "_id": ObjectId(  
    "5f4f7fef2d4b45b7f11b6d7a") ,  
  "user_id": "Daniel",  
  "age": 25,  

```

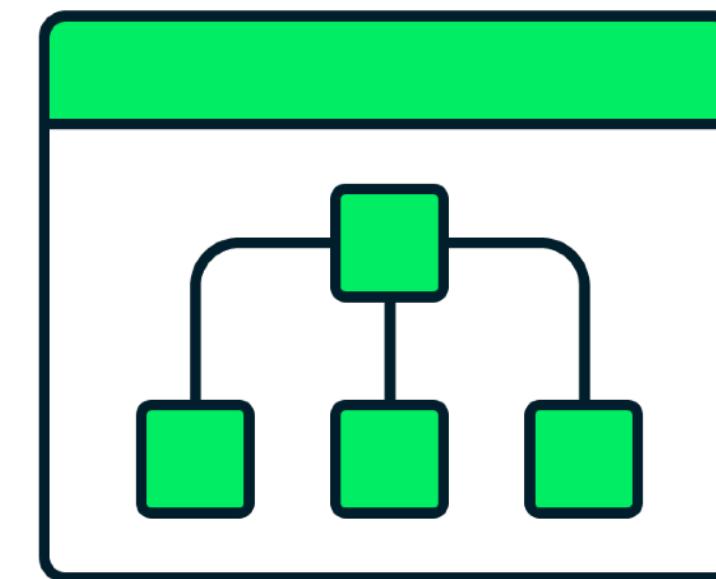


# Collections and Schema Validation

The document model used by MongoDB can enforce a schema if required, the recommended approach is to do so using JSON Schema.

JSON Schema:

- Allows a prescribed document structure to be configured on a per collection basis.
- Can tune schema validation according to use case.
- Can be used by any query to inspect document structure and content.



# Schema Validation

This example creates a users collection with validation rules:

- The document must be an object.
- It must have the required fields: `image`, `mail`, `name`, and `title`.
- The `_id` must be a unique `objectId`.
- The `image` must be a string.
- The `mail` must be a string and follow the pattern of a valid email address.
- The `name` and `title` must be strings.

You can customize the validation rules based on your specific requirements. Adjust the properties, types, and patterns as needed.

```
db.createCollection("users", {  
    validator: {  
        $jsonSchema: {  
            bsonType: "object",  
            required: ["image", "mail", "name", "title"],  
            properties: {  
                _id: {  
                    bsonType: "objectId",  
                    description: "must be a unique ObjectId"  
                },  
                image: {  
                    bsonType: "string",  
                    description: "must be a string and is required"  
                },  
                mail: {  
                    bsonType: "string",  
                    pattern: "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\". [a-zA-Z]{2,}$$",  
                    description: "must be a valid email address and is required"  
                },  
                name: {  
                    bsonType: "string",  
                    description: "must be a string and is required"  
                },  
                title: {  
                    bsonType: "string",  
                    description: "must be a string and is required"  
                }  
            }  
        }  
    }  
});
```

# MongoDB Query Language

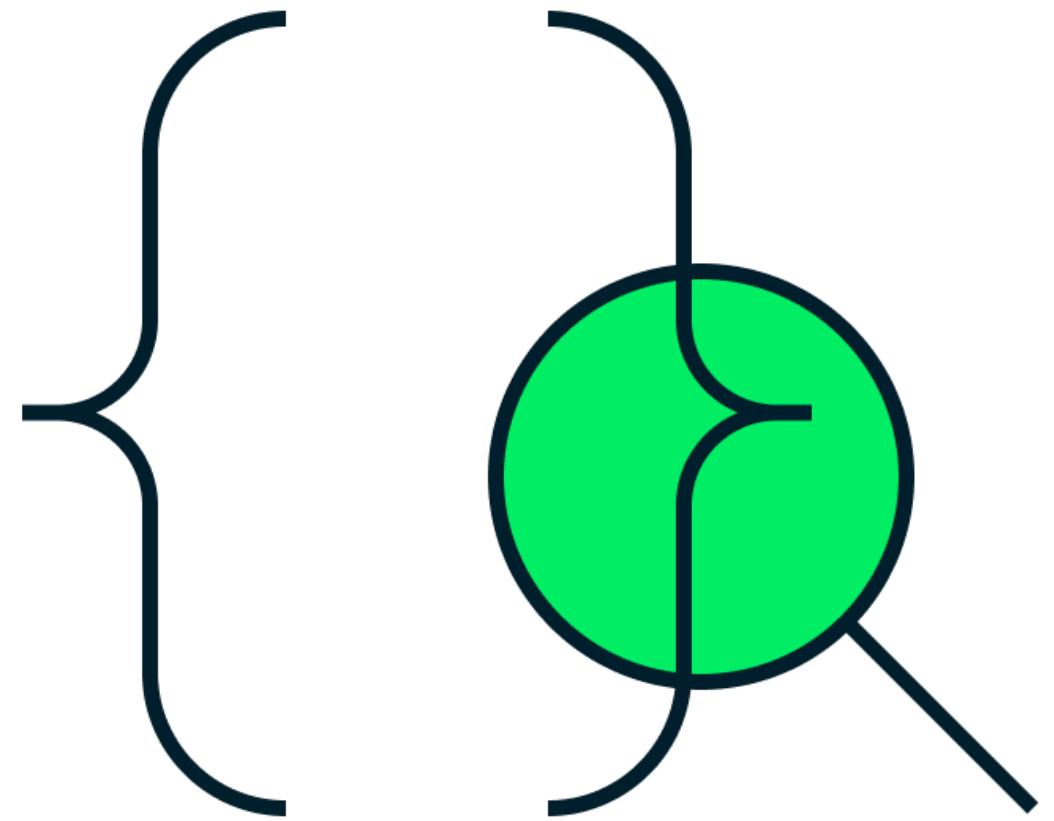




Simple syntax

Designed to query  
documents

Only queries a single  
collection



MongoDB Query Language



# MongoDB Query Language

MQL is designed for single collection queries and it is typically used for creating, reading, updating, or deletion (CRUD) operations.

MQL query operators:

- Comparison
- Logical
- Element
- Array
- Evaluation
- Bitwise
- Comment
- Geospatial
- Projection, Update and Update  
Modifiers



# MQL Find()



# MQL Find()

db.<collection>.find()

Query filter document

db.collection.find({ <field1>: <value1>, ... })

Specifying query operators

db.<collection>.find({ <field1>: { <operator1>: <value1> }, ... })



# MQL Find()

db.<collection>.find()

Query filter document

db.collection.find({ <field1>: <value1>, ... })

Specifying query operators

db.<collection>.find({ <field1>: { <operator1>: <value1> }, ... })



# MQL Find()

db.<collection>.find()

Query filter document

db.collection.find({ <field1>: <value1>, ... })

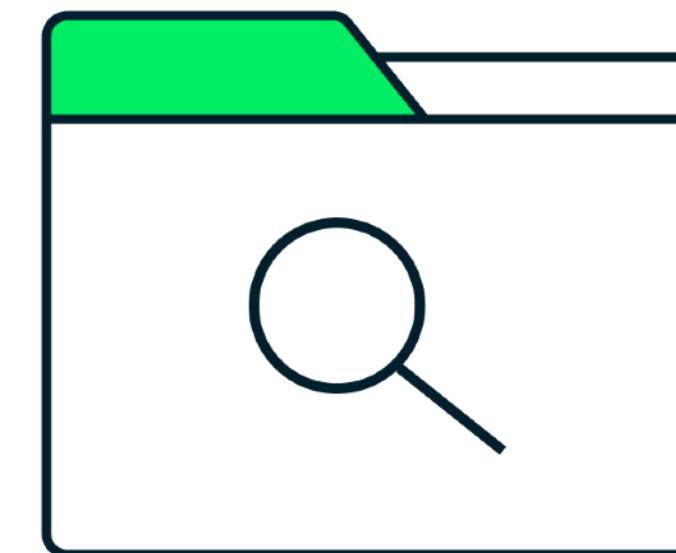
Specifying query operators

db.<collection>.find({ <field1>: { <operator1>: <value1> }, ... })



# MQL Find(): Important Note

The collection is implicit in MQL based on  
the query's criteria.



# MongoDB CRUD Operations



MongoDB CRUD Operations - x +

mongodb.com/docs/manual/crud/

MongoDB Products Resources Solutions Company Pricing

MongoDB Documentation Docs Home → Develop Applications → MongoDB Manual

← Back To Develop Applications

MongoDB Manual

7.0 (current)

- ▶ Introduction
- ▶ Installation
- MongoDB Shell (mongosh)
- ▼ MongoDB CRUD Operations
  - ▶ Insert Documents
  - ▶ Query Documents
  - ▶ Update Documents
  - ▶ Delete Documents
  - Bulk Write Operations
  - Retryable Writes
  - Retryable Reads
  - SQL to MongoDB Mapping Chart

[https://www.mongodb.com/pricing](#)

## MongoDB CRUD Operations

CRUD operations *create, read, update, and delete* documents.

You can connect with driver methods and perform CRUD operations for deployments hosted in the following environments:

 You can perform CRUD operations in the UI for deployments hosted in MongoDB Atlas.

### Create Operations

Create or insert operations add new [documents](#) to a [collection](#). If the collection does not currently exist, insert operations will create the collection.

MongoDB provides the following methods to insert documents into a collection:

- `db.collection.insertOne()` New in version 3.2
- `db.collection.insertMany()` New in version 3.2

Share Feedback

On this page

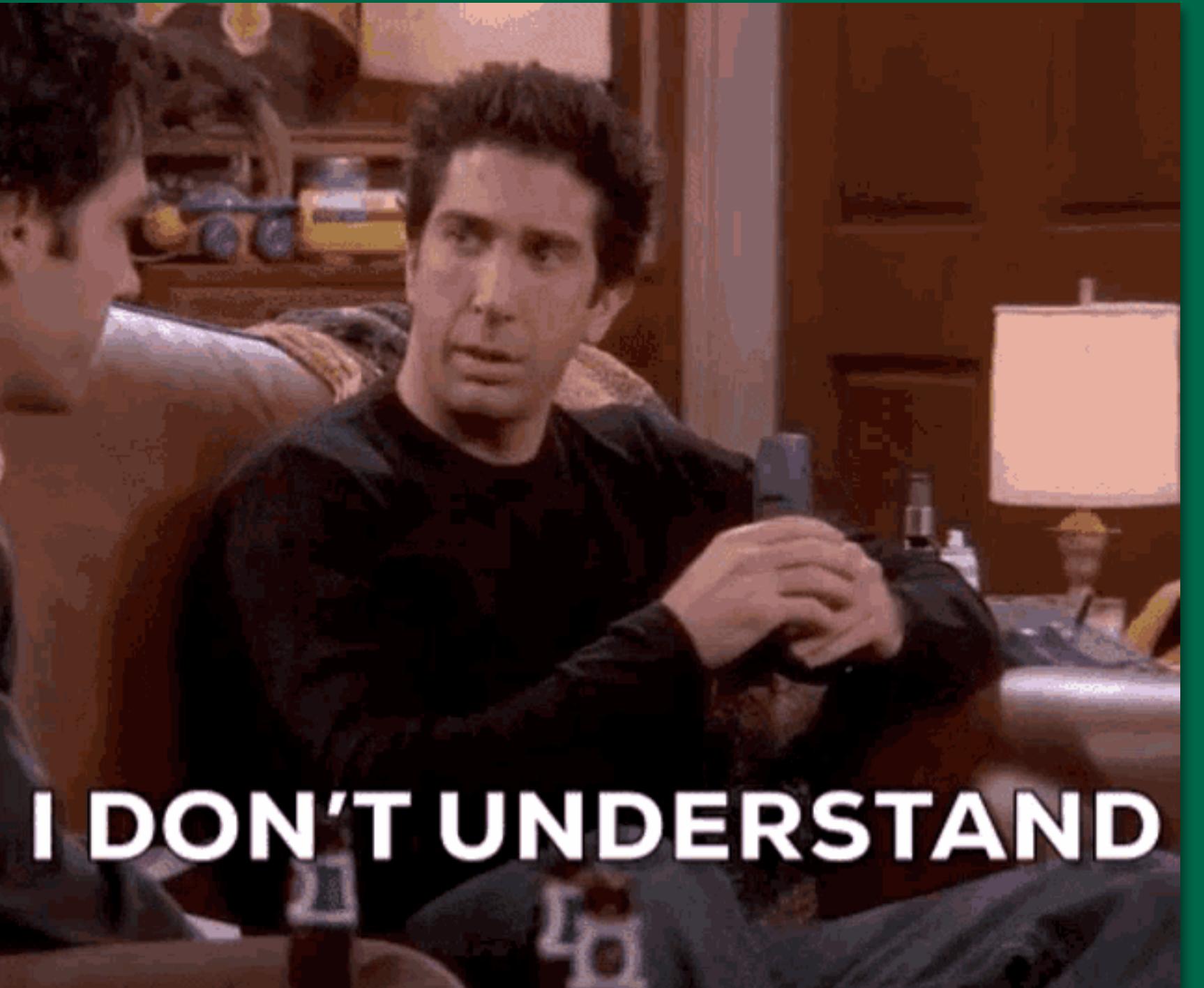
- Create Operations
- Read Operations
- Update Operations
- Delete Operations
- Bulk Write

when you ask Rasmus for help and he says "Read documentation"



# The Docs and Articles

- MongoDB, Inc. 2023. Introduction to MongoDB: <https://www.mongodb.com/docs/manual/introduction/>
- MongoDB, Inc. 2023. Databases and Collections: <https://www.mongodb.com/docs/manual/core/databases-and-collections/>
- MongoDB, Inc. 2023. MongoDB CRUD Operations: <https://www.mongodb.com/docs/manual/crud/>
- MongoDB, Inc. 2023. Data Modeling Introduction: <https://www.mongodb.com/docs/manual/core/data-modeling-introduction/>
- MongoDB, Inc. 2023. Indexes: <https://www.mongodb.com/docs/manual/indexes/>
- Beugnet, M. 2023. MongoDB Cheat Sheet: <https://www.mongodb.com/developer/products/mongodb/cheat-sheet/>
- Karlsson, j. 2023. MongoDB Schema Design Best Practices: <https://www.mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/>



# Create

```
db.users.insertOne(← collection
{
  name: "sue", ← field: value
  age: 26, ← field: value
  status: "pending" ← field: value } document
})
```

```
db.createCollection("posts")

db.posts.insertOne(
{
  caption: "Beautiful sunset at the beach",
  createdAt: new Date("2023-04-05T15:27:14Z"),
  image: "https://images.unsplash.com/photo-1566241832378-917a0f30db2c?ixlib=rb-4.0.3",
  uid: ObjectId("ZfPTVEMQKf9v")
}

db.posts.insertMany([
  {
    caption: "Beautiful sunset at the beach",
    createdAt: new Date("2023-04-05T15:27:14Z"),
    image: "https://images.unsplash.com/photo-1566241832378-917a0f30db2c?ixlib=rb-4.0.3",
    uid: ObjectId("ZfPTVEMQKf9v")
  },
  {
    caption: "Exploring the city streets of Aarhus",
    createdAt: new Date("2023-04-06T10:45:30Z"),
    image: "https://images.unsplash.com/photo-1559070169-a3077159ee16?ixlib=rb-4.0.3",
    uid: ObjectId("fTs84KRoYw5p")
  },
  {
    caption: "Delicious food at the restaurant",
    createdAt: new Date("2023-04-04T20:57:24Z"),
    image: "https://images.unsplash.com/photo-1548940740-204726a19be3?ixlib=rb-4.0.3",
    uid: ObjectId("fjpRTTjZHwr")
  },
  //...
])
```

# Read

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

← collection  
← query criteria  
← projection  
← cursor modifier

```
// find all  
db.posts.find()  
  // find all - to array  
  db.posts.find().toArray()  
  
// posts with specific user  
db.posts.find({uid: ObjectId("ZfPTVEMQKf9v")})  
// all users with title: "Senior Lecturer"  
db.users.find({title:"Senior Lecturer"})  
  
// find and then sort  
// -1 for ascending or -1 for descending  
db.users.find().sort({name:1})  
db.posts.find().sort({createdAt:1})  
  
db.posts.find().count() // count docs
```

# Update

```
db.users.updateMany(  
  { age: { $lt: 18 } },  
  { $set: { status: "reject" } } )
```

← collection  
← update filter  
← update action

```
db.users.updateOne(  
  // Specify the user you want to update  
  { _id: ObjectId("ZfPTVEMQKf9v") },  
  {  
    $set: {  
      // Update the name field with the new value  
      name: "New Name",  
      // Update the title field with the new value  
      title: "New Title"  
      // Add more fields to update as needed  
    }  
  }  
);  
  
db.users.updateMany(  
  // Specify the criteria for the documents you want to update  
  { title: "Senior Lecturer" },  
  {  
    $set: {  
      // Update the title field with the new value  
      title: "Updated Title",  
      // Add more fields to update as needed  
    }  
  }  
);
```

# Update

- If you want to completely replace a document

```
db.users.replaceOne(  
    // Specify the user you want to replace  
    { _id: ObjectId("ZfPTVEMQKf9v") },  
    {  
        _id: ObjectId("ZfPTVEMQKf9v"),  
        image: "https://new-image-url.com",  
        mail: "new-email@example.com",  
        // Add more fields as needed  
    }  
);
```

# Delete

```
db.users.deleteMany(  
  { status: "reject" }  
)
```



collection  
delete filter

```
db.users.deleteOne(  
  // Specify the user you want to delete  
  { _id: ObjectId("ZfPTVEMQKf9v") }  
);  
  
db.users.deleteMany(  
  // Specify the criteria for the  
  // documents you want to delete  
  { title: "Senior Lecturer" }  
);
```

# Data Modeling and Schema Design





# MongoDB Schema Design

- No formal process
- No algorithms
- No rules



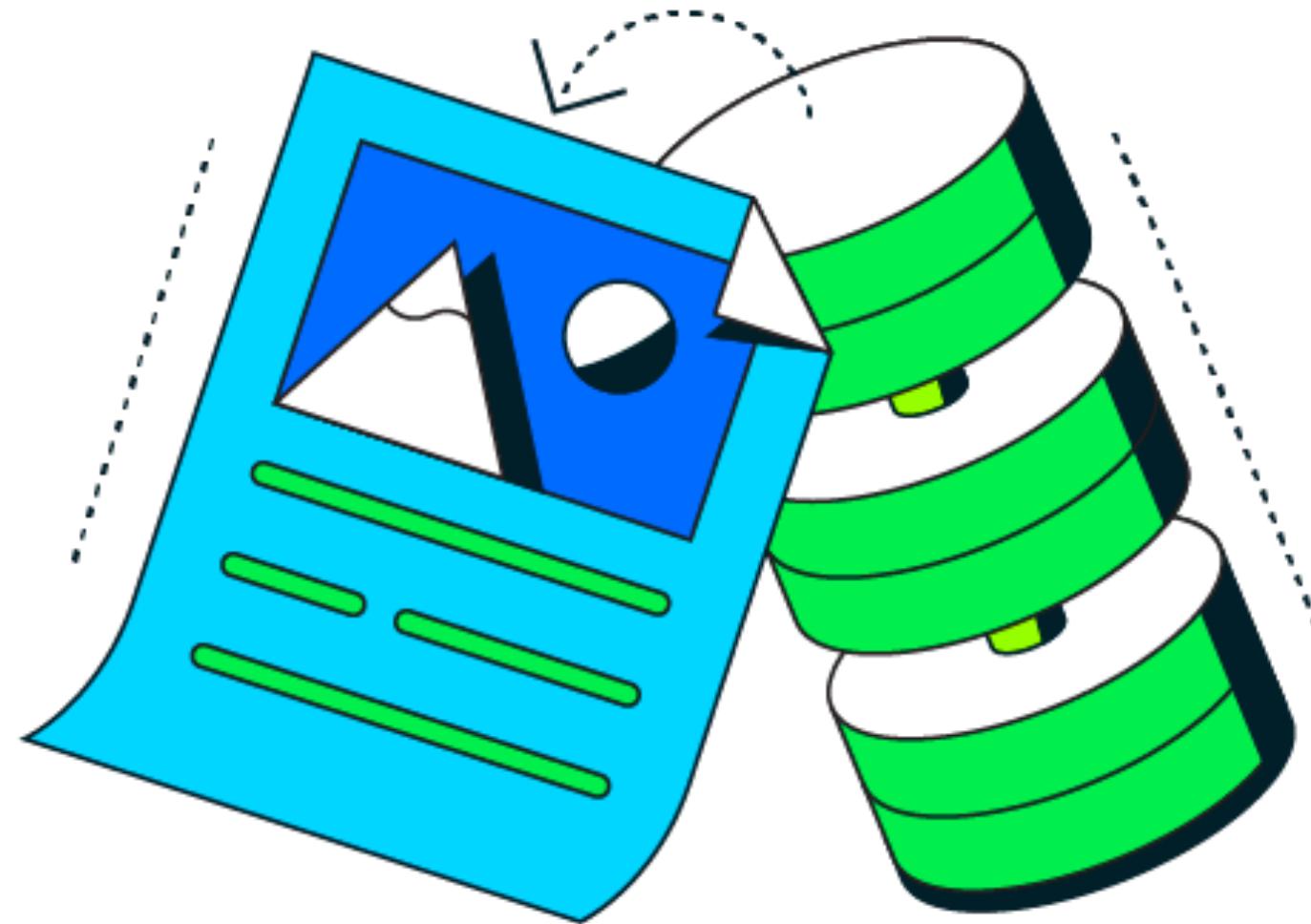


# Schema Design

The design comes from the needs of the application first.

Therefore, the schema should evolve as the application changes.





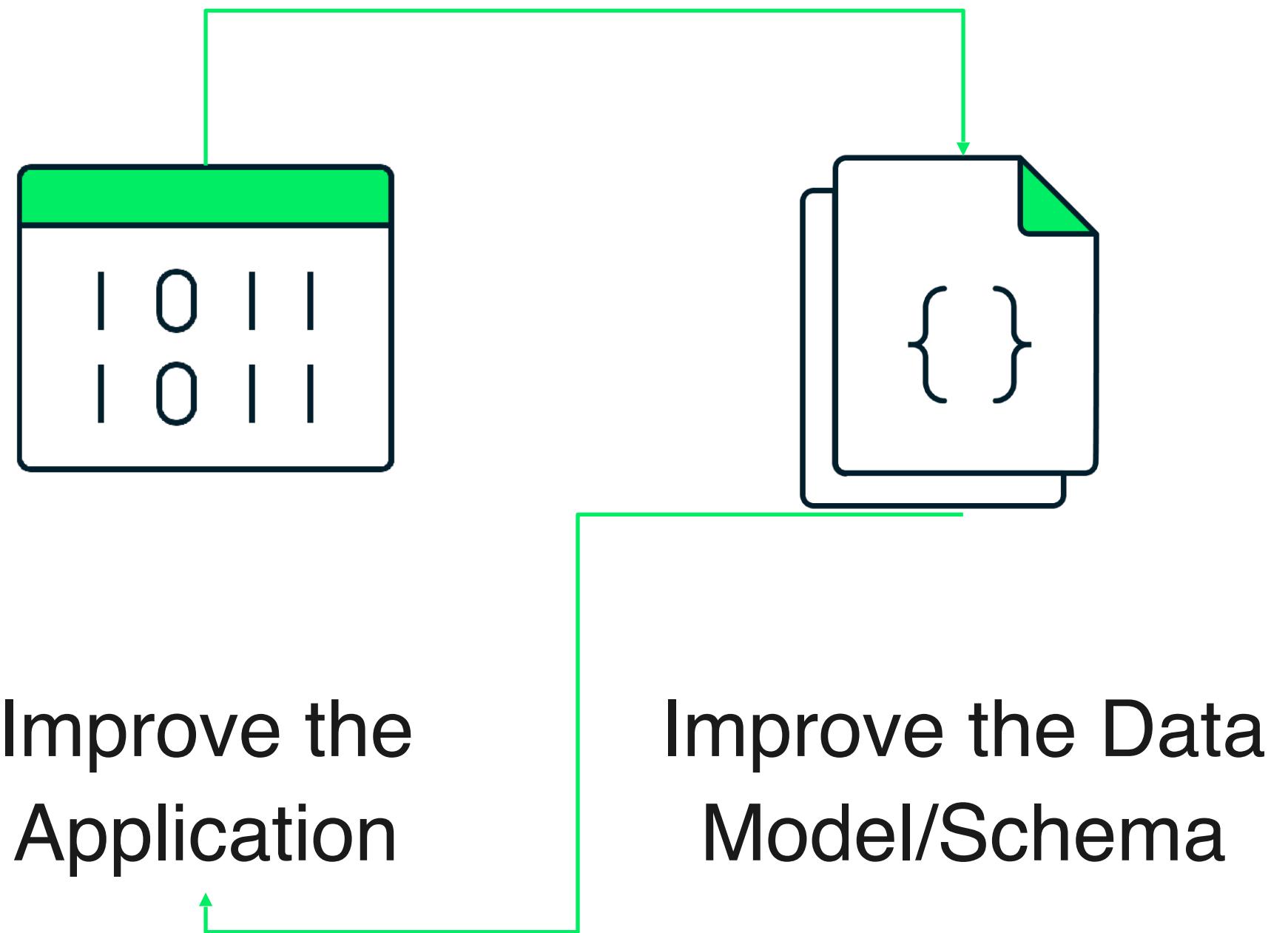
# Data Modeling and the Document Model

The core of data modeling in the document model is to understand what data is needed by your queries. Once that information is known, can you begin designing the schema.



# Data Modeling with MongoDB

- Several design possibilities
- Design for the usage pattern
- Evolving the schema is easy
- No migrations or downtime required for a new version of the schema





# Schema Design: Considerations

- Your queries and the specific data your application requires.
- How your application reads the data (read patterns).
- How your application writes the data (write patterns).
- What are the relationships between your data (linked or embedded).





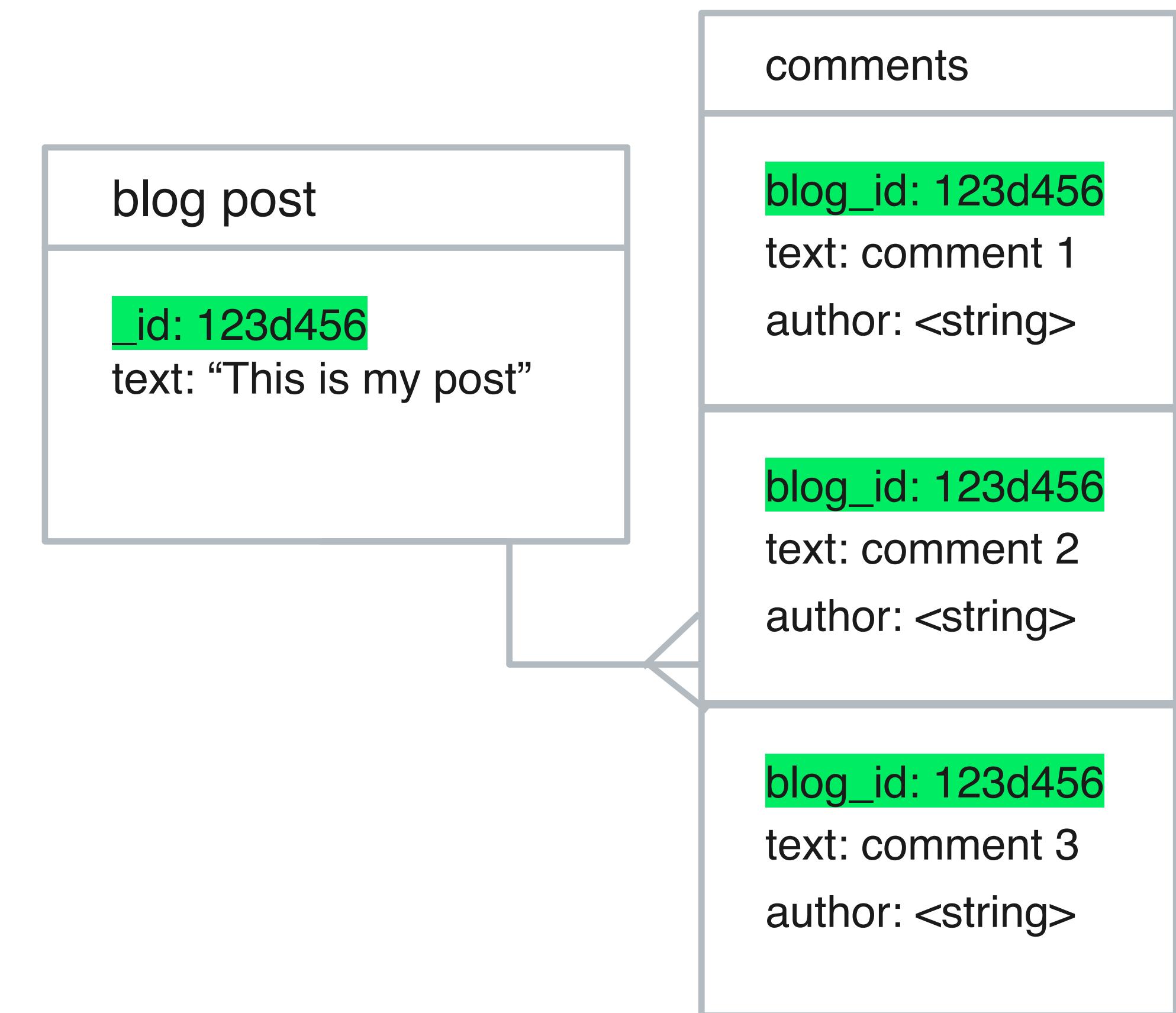
# Schema Design - Link or Embed?

Embedded vs Linked relationship in the Post-Comment example

Embedded



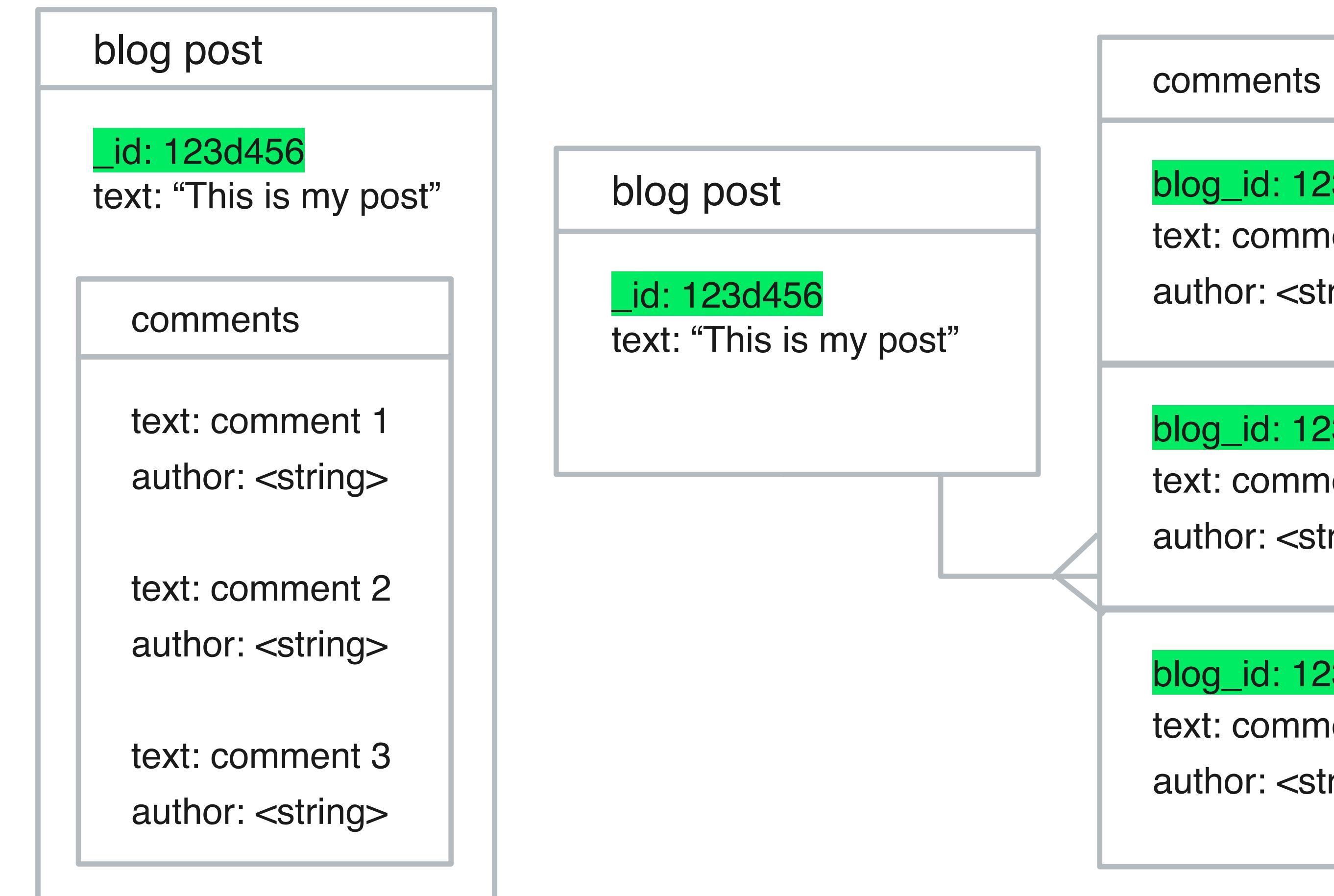
Linked





# Schema Design - Link or Embed?

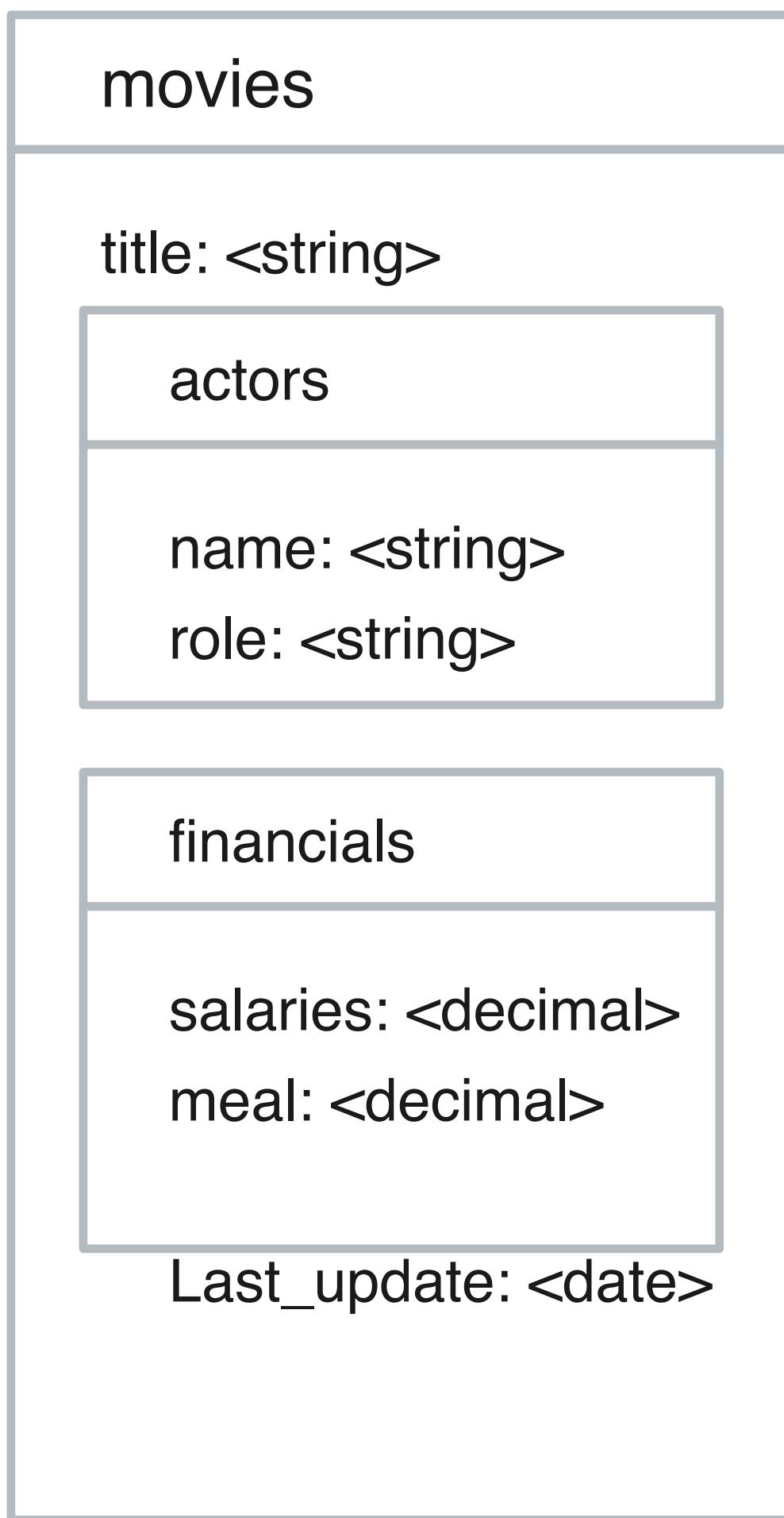
- Do I want most of the data's information embedded?
- Do I need to search within the embedded data?
- How frequently will the embedded data change?
- Is the embedded data shared or private?





# Example: Movies and Reviews

## Embedded



## Linked



# Relationships





One to one (1-1)

One to many (1-N)

Many to many (N-N)

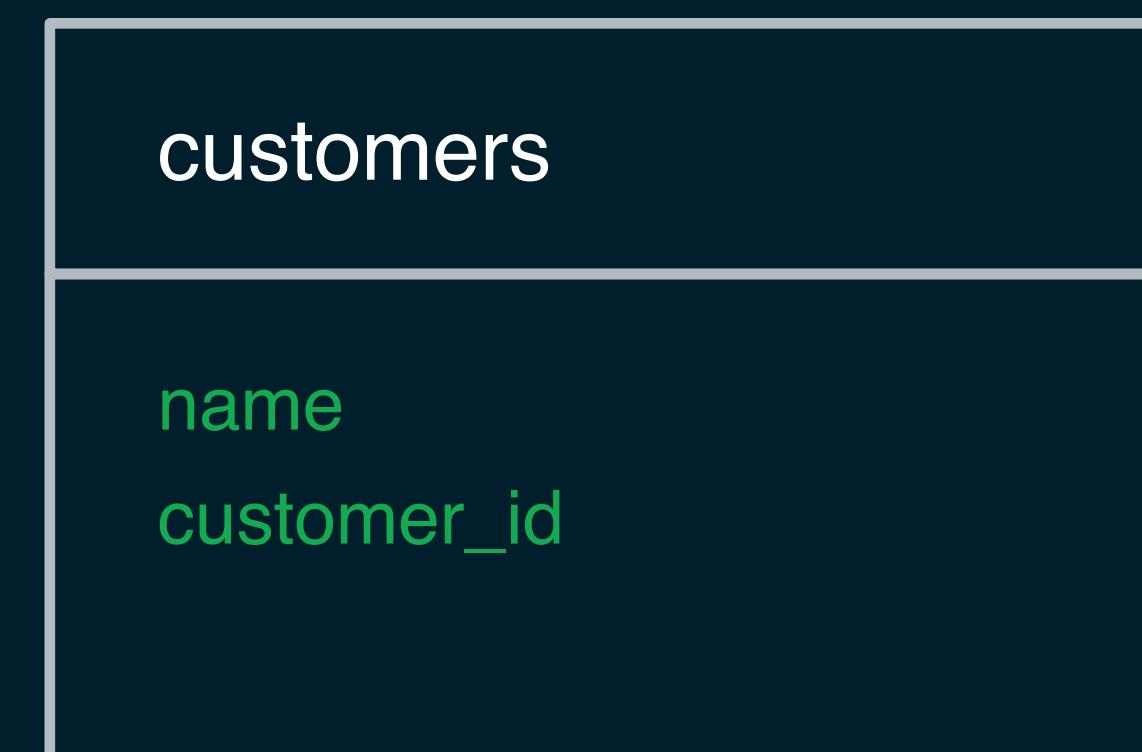
# Relationships and Data Modeling



# Relationships

## One-to-One (1-1)

A one-to-one relationship is represented and stored in a single document, this would typically be data like a person's name and the customer id.





# One to One (1 - 1)

Scenario:

You have to map patron and address relationships. In this example, you'll need to view one data entity in context of the other.

A screenshot of a web browser window displaying the MongoDB Schema Design Best Practices page. The URL in the address bar is [mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/](https://mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/). The page title is "MongoDB Schema Design Best Practices". The main content area has a dark background with white text. It features a section titled "One-to-One" with a sub-section titled "User Document". Below this, there is a code block showing a MongoDB document structure:

```
1  {
2      "_id": "ObjectId('AAA')",
3      "name": "Joe Karlsson",
4      "company": "MongoDB",
5      "twitter": "@JoeKarlsson1",
6      "twitch": "joe_karlsson",
7      "tiktok": "joekarlsson",
8      "website": "joekarlsson.com"
9 }
```

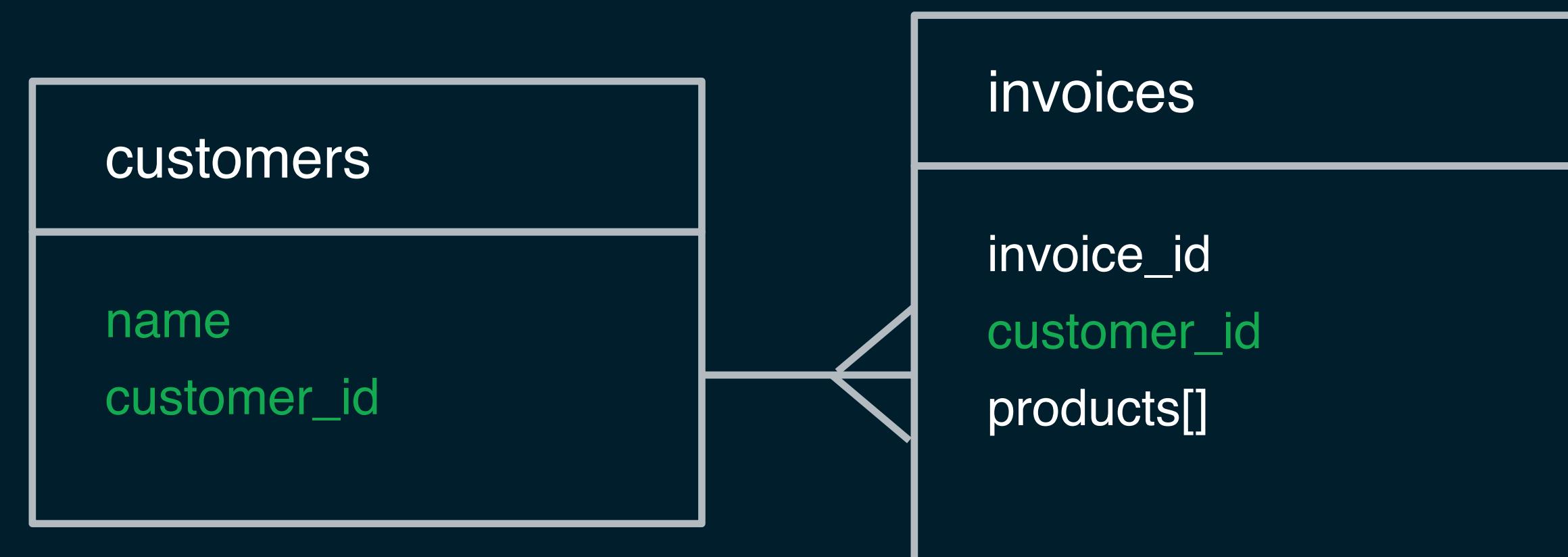
The right side of the page contains a "Table of Contents" sidebar with several sections: "Schema Design Approaches – Relational vs. MongoDB", "Embedding vs. Referencing", and "Type of Relationships". There is also a link to "Additional Resources".



# Relationships

## One-to-Many (1-N)

A one-to-many relationship can be considered when an object of a given type is associated with N objects of a second type.





# One to Many (1 - N)

Scenario (Link):

You have to map publisher and book relationships. Suppose you had the same publisher data for the same book. Embedding the [publisher] document inside the [book] document would lead to repetition of publisher information.

MongoDB Schema Design Best Practices

mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/

MongoDB Developer Topics Documentation Articles Tutorials Events Code Examples Podcasts MongoDB TV

## Products:

```
1  {
2      "name": "left-handed smoke shifter",
3      "manufacturer": "Acme Corp",
4      "catalog_number": "1234",
5      "parts": ["ObjectId('AAAA')", "ObjectId('BBBB')", "ObjectId('CCCC')"]
6 }
```

## Parts:

```
1  {
2      "_id" : "ObjectId('AAAA')",
3      "partno" : "123-aff-456",
4      "name" : "#4 grommet",
5      "qty": "94",
6      "cost": "0.94",
7      "price": " 3.99"
8 }
```

## Table of Contents

- Schema Design
- Approaches – Relational vs. MongoDB
- Embedding vs. Referencing
- Type of Relationships

Additional Resources:

MongoDB Schema Design Best Practices

[mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/](https://mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/)

MongoDB Developer Topics Documentation Articles Tutorials Events Code Examples Podcasts MongoDB TV

**Hosts:**

```
1 {
  "_id": ObjectId("AAAB"),
  "name": "goofy.example.com",
  "ipaddr": "127.66.66.66"
}
```

**Log Message:**

```
1 {
  "time": ISODate("2014-03-28T09:42:41.382Z"),
  "message": "cpu is on fire!",
  "host": ObjectId("AAAB")
}
```

**Table of Contents**

- Schema Design
- Approaches – Relational vs. MongoDB
- Embedding vs. Referencing
- Type of Relationships

Additional Resources:



# One to Many (1 - N)

Scenario (Embed):

You have to map a patron with multiple address relationships. In this one-to-many relationship between [patron] and [address] data, the [patron] has multiple [address] entities.

MongoDB Schema Design Best Practices

mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/

MongoDB Developer Topics Documentation Articles Tutorials Events Code Examples Podcasts MongoDB TV

## One-to-Few

Okay, now let's say that we are dealing a small sequence of data that's associated with our users. For example, we might need to store several addresses associated with a given user. It's unlikely that a user for our application would have more than a couple of different addresses. For relationships like this, we would define this as a *one-to-few relationship*.

```
1 {  
2   "_id": "ObjectId('AAA')",  
3   "name": "Joe Karlsson",  
4   "company": "MongoDB",  
5   "twitter": "@JoeKarlsson1",  
6   "twitch": "joe_karlsson",  
7   "tiktok": "joekarlsson",  
8   "website": "joekarlsson.com",  
9   "addresses": [  
10     { "street": "123 Sesame St", "city": "Anytown", "cc": "USA" },  
11     { "street": "123 Avenue Q", "city": "New York", "cc": "USA" }  
12   ]  
13 }
```

### Table of Contents

- Schema Design
- Approaches – Relational vs. MongoDB
- Embedding vs. Referencing
- Type of Relationships

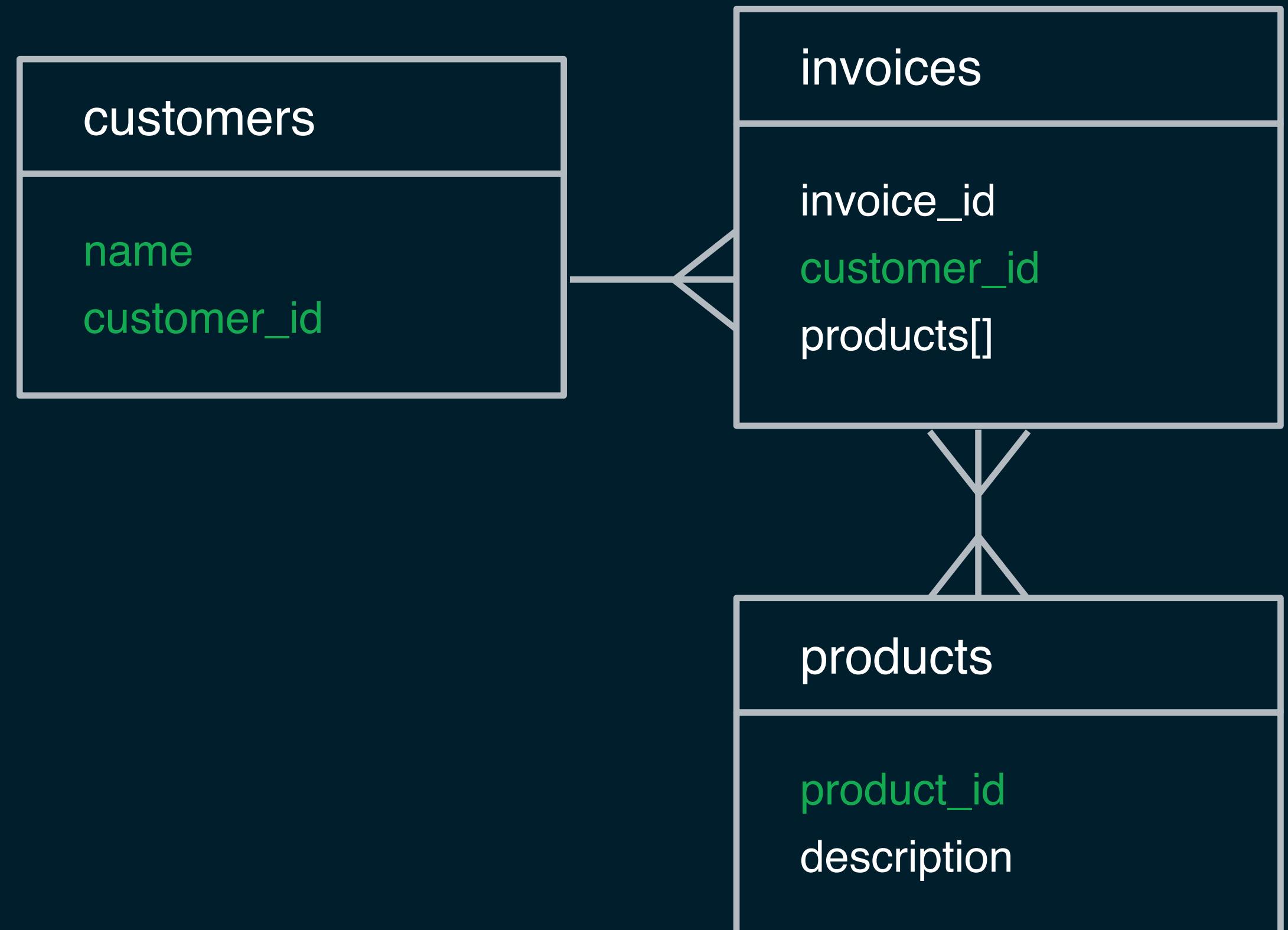
Additional Resources:



# Relationships

## Many-to-Many (N-N)

A Many-to-Many relationship between two entities where they both might have many relationships between each other.





# Many to Many (N-N)

Scenario:

Consider a scenario where a book was written by multiple authors and similarly, one of the authors has written multiple books. How would we go about mapping these relationships?

The screenshot shows a web browser displaying an article titled "MongoDB Schema Design Best Practices" from the MongoDB Developer website. The URL in the address bar is [mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/](https://mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/). The page features a navigation bar with links to "Topics", "Documentation", "Articles", "Tutorials", "Events", "Code Examples", "Podcasts", and "MongoDB TV".

**Users:**

```
1  {
2    "_id": ObjectId("AAF1"),
3    "name": "Kate Monster",
4    "tasks": [ObjectId("ADF9"), ObjectId("AE02"), ObjectId("AE73")]
5 }
```

**Tasks:**

```
1  {
2    "_id": ObjectId("ADF9"),
3    "description": "Write blog post about MongoDB schema design",
4    "due_date": ISODate("2014-04-01"),
5    "owners": [ObjectId("AAF1"), ObjectId("BB3G")]
6 }
```

**Table of Contents**

- Schema Design
- Approaches – Relational vs. MongoDB
- Embedding vs. Referencing
- Type of Relationships

Additional Resources:



- One-to-One - Prefer key value pairs within the document
- One-to-Few - Prefer embedding
- One-to-Many - Prefer embedding
- One-to-Squillions - Prefer Referencing
- Many-to-Many - Prefer Referencing

## Embed or Link



# Embed

- For integrity with read operations
- For integrity with write operations
- On one-to-one and one-to-many
- For data that is deleted together by default

# Link

- When the "many" side is a huge number
- For integrity on write operations on many-to-many
- When a piece is frequently used, but not the other and memory is an issue



- Rule 1: Favor embedding unless there is a compelling reason not to.
- Rule 2: Needing to access an object on its own is a compelling reason not to embed it.
- Rule 3: Avoid joins and lookups if possible, but don't be afraid if they can provide a better schema design.
- Rule 4: 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.
- Rule 5: 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.

# General Rules for MongoDB Schema Design

# Indexes and search

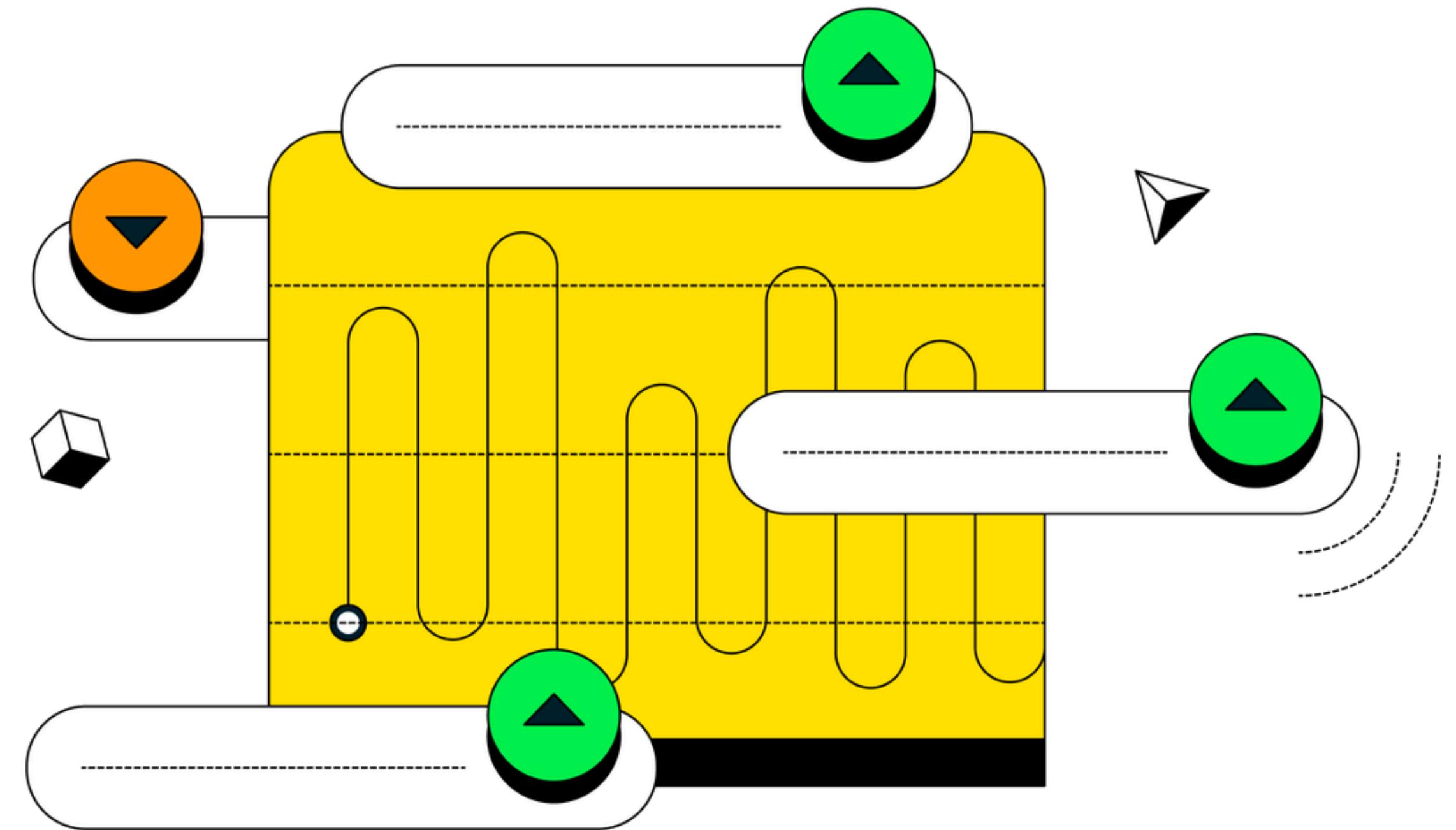




# What is an Index in MongoDB?

Indexes hold a small portion of the collection's data in a form that's easy to traverse. They are used to:

- Speed up queries and updates
- Avoid disk I/O as queries eliminating the need for slow collection scans
- Reduce overall computation



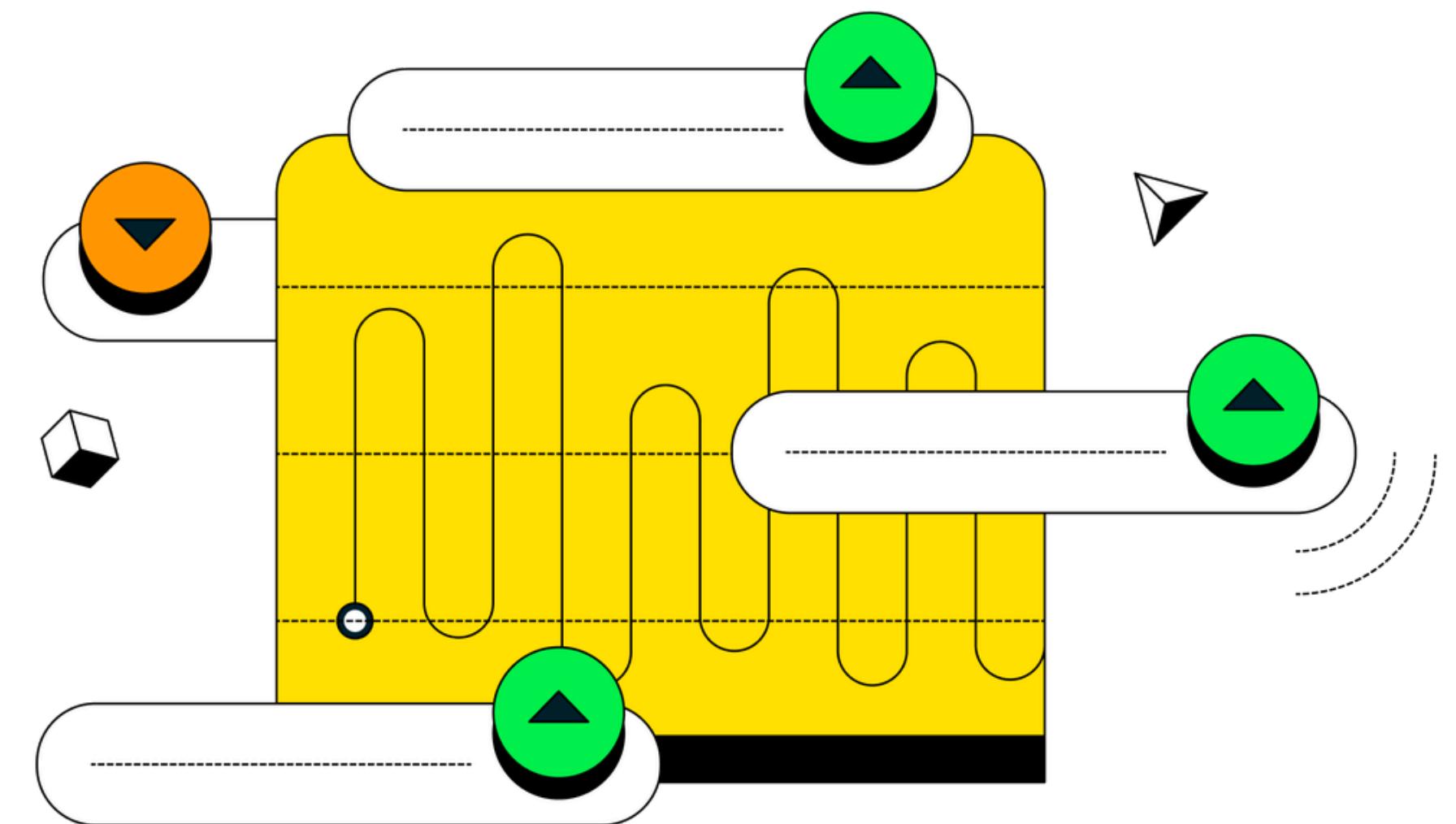


# When to Use Indexes

Developers should use an index when querying data in a collection, especially for frequently run queries.

When determining if an index should be used, consider:

- Four indexes is a good rule of thumb for the ideal number for a given collection.
- Sixty-four indexes are the maximum per collection, however, above 20 and performances renders the system almost unusable for workloads.





# Considerations When Using Indexes

Indexes require RAM.

Avoid unnecessary indexes at all cost, otherwise the write performance will suffer. Each index adds 10% overhead.

When does an index entry get modified?

- Data is inserted (applies to all indexes).
- Data is deleted (applies to all indexes).
- Data is updated in such a way that its indexed field changes.



## Types of Indexes Available

Most common indexes:

- Single Field
- Compound Index

Other types of specialized indexes including:

- Multikey Index
- Geospatial Index
- Text Index
- Hashed Index
- Time-To-Live (TTL) Index
- Hidden Index
- Partial Index
- Wildcard Index