Let’s Turn on MongoDB 🎉

Beklager, nogle af jer har set den før.

Men spot on på dagens emner ✨

ERHVERVSAKADEMI AARHUS
BUSINESS ACADEMY AARHUS

Databases, NoSQL & MongoDB

Web Development

RACE

ERHVERVSAKADEMI AARHUS
BUSINESS ACADEMY AARHUS

Agenda

Databases, NoSQL & MongoDB

1. NoSQL og MongoDB
2. MongoDB Atlas
3. Mongo Query Language
4. REST API med MongoDB & React Frontend

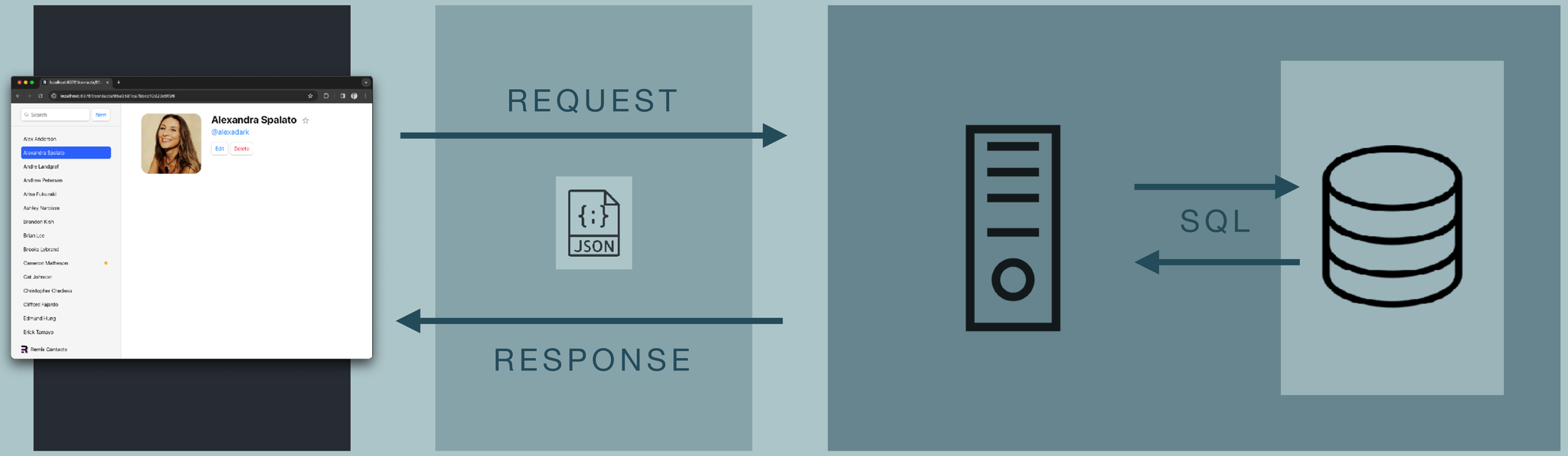
RACE

Web Dev Architecture

FRONTEND

REST API

BACKEND



CLIENT

SERVER

BROWSER/
THUNDERCLIENT

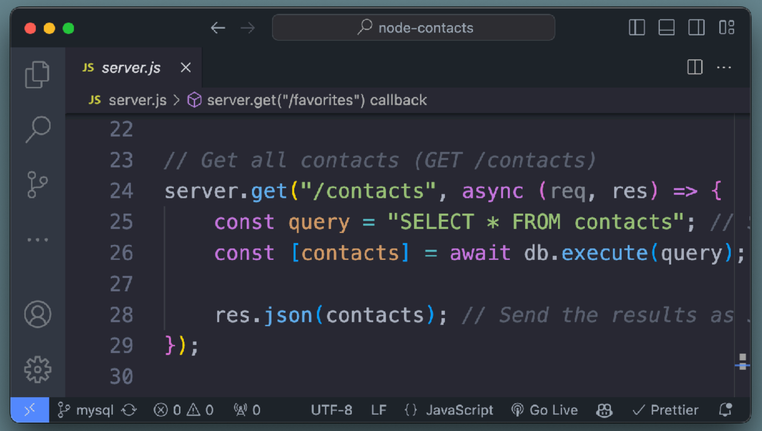
REST API

BACKEND

REQUEST

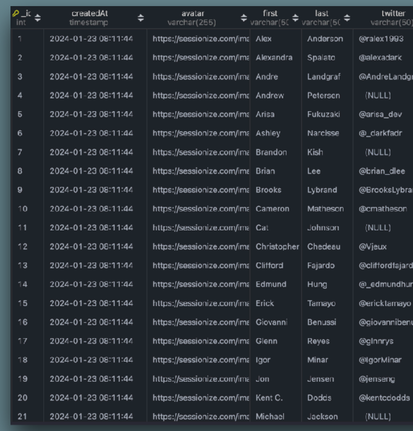
RESPONSE

NODE/EXPRESS



SQL

MYSQL



CLIENT

SERVER

BROWSER/
THUNDERCLIENT

REST API

BACKEND

NODE/EXPRESS

REQUEST

JSON

RESPONSE

MQL

MongoDB
ATLAS

CLIENT

SERVER

NoSQL and MongoDB

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.


MySQL™


mongoDB

Most Popular Databases

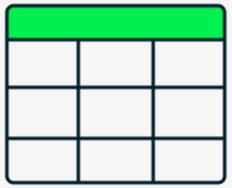
https://survey.stackoverflow.co/2023/#section-most-popular-technologies-databases

https://www.youtube.com/watch?v=-bt_y4Loofg

Why Non-Relational Databases?

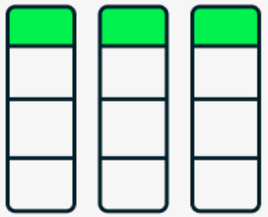
Where it Began: Relational

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

1970s

Relational Databases

SQL was developed by IBM as a way to interact with the new relational databases

1990s

World Wide Web

Need for data storage explodes

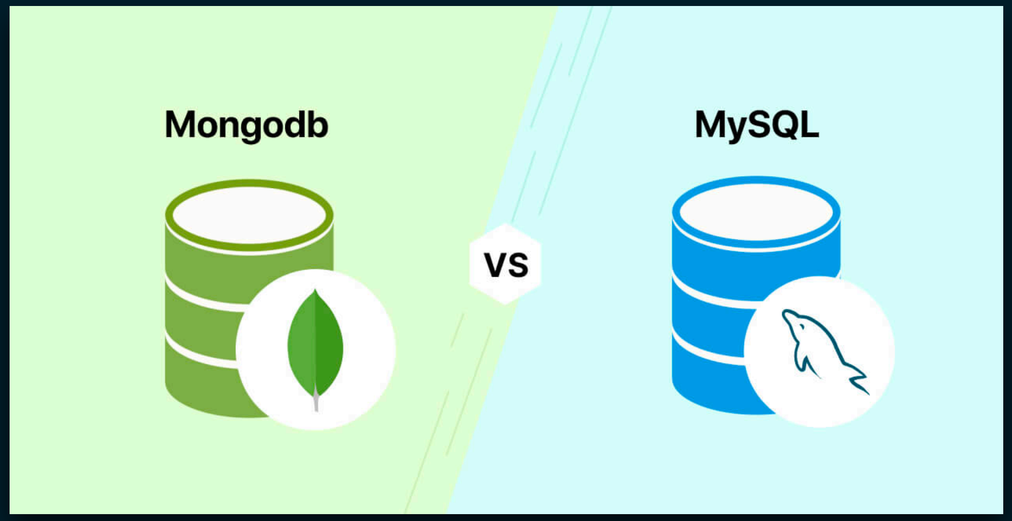
2000s

NoSQL

Unstructured data storage to mitigate costs and increase efficiency

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	https://share.cederdorff.com/i
2	Rasmus Cederdorff	race@dev.dk	Senior Lecturer	https://share.cederdorff.com/i
3	Lars Bogetoft	larb@eaaa.dk	Head of Education	https://kea.dk/slir/w200-c1x1/
4	Edith Terte	edan@kea.dk	Lecturer	https://media.licdn.com/dms/im
5	Frederikke Bender	fbe@kea.dk	Head of Education	https://kea.dk/slir/w200-c1x1/
6	Murat Kilic	mki@eaaa.dk	Senior Lecturer	https://www.eaaa.dk/media/llya
7	Anne Kirketerp	anki@eaaa.dk	Head of Education	https://www.eaaa.dk/media/5buh

From object to database table? WTF!1!

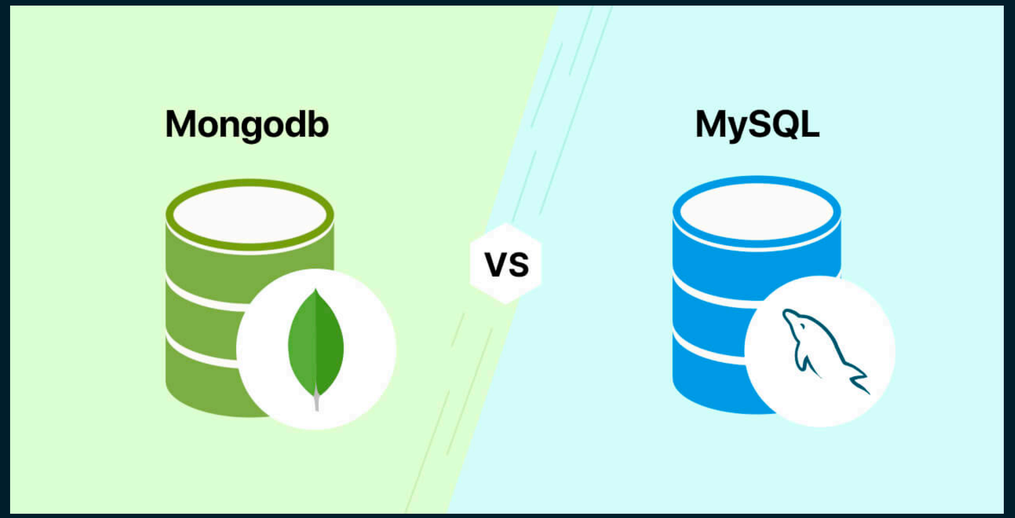
Why NoSQL?

https://www.youtube.com/watch?v=0X43QfCfyk0&t=35s&ab_channel=MongoDB

- **Flexibility:** NoSQL databases do not require a predefined schema, which means that the data structure can be easily changed as the application evolves. This makes them well-suited for applications that handle unstructured or semi-structured data.
- **Scalability:** NoSQL databases are typically horizontally scalable, which means that they can be easily added to more servers to handle increasing data and workloads. This makes them ideal for applications that need to support large amounts of data or that experience sudden spikes in traffic.
- **Performance:** NoSQL databases can often outperform SQL databases for certain types of workloads, such as read-heavy applications or applications that require low latency. This is because they are designed to store and retrieve data efficiently, without the overhead of maintaining a rigid schema.
- **Ease of use:** NoSQL databases can be easier to use than SQL databases, especially for developers who are not familiar with traditional relational databases. They often have simpler APIs and data models, which can make it easier to develop and maintain applications.

But Why MongoDB?

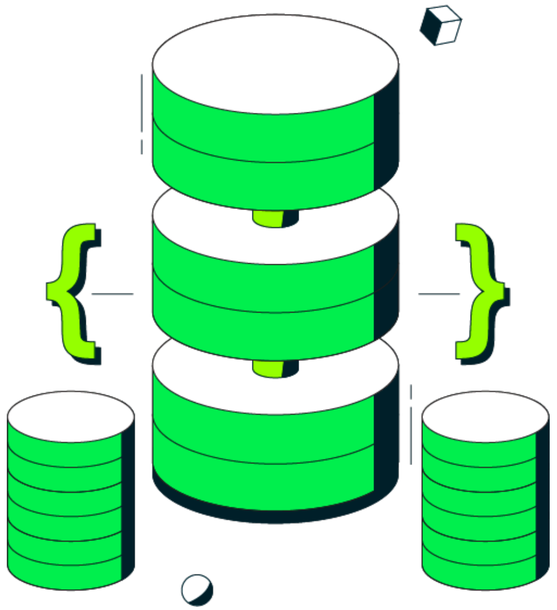
- **Data Flexibility**: MongoDB handles unstructured data well, while MySQL is structured.
- **Schema**: MongoDB is schema-less, allowing for dynamic data, while MySQL requires a predefined schema.
- **Scalability**: MongoDB is horizontally scalable, beneficial for large datasets and high traffic.MySQL 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.

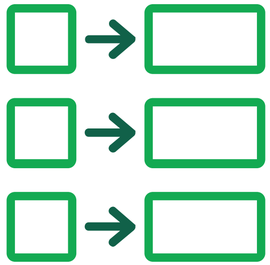
- Polymorphic data structures
- Flexible schemas
- Easy to scale large workloads

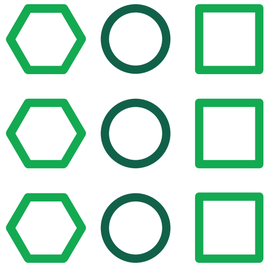
What is a non-relational database?



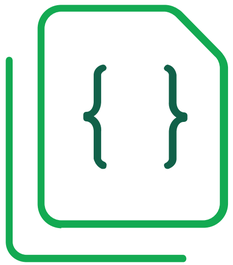
Non-Relational Database Types

Non-Relational Database Types





Column



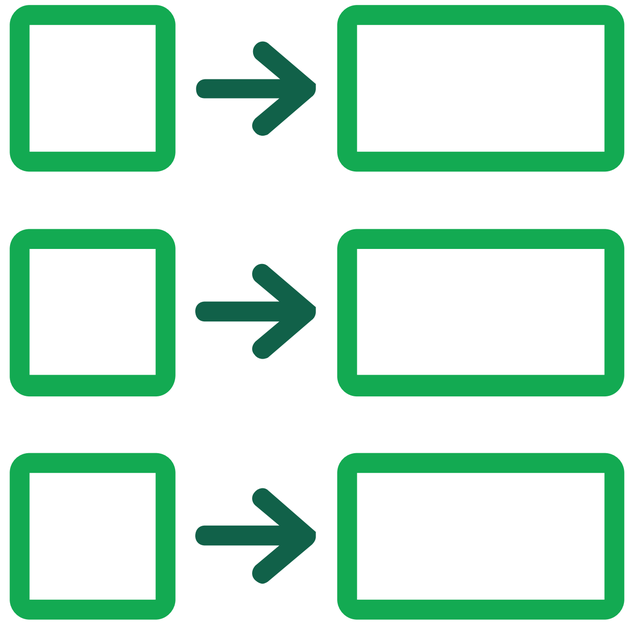
Document

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

Key/Value: Example

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

Structure

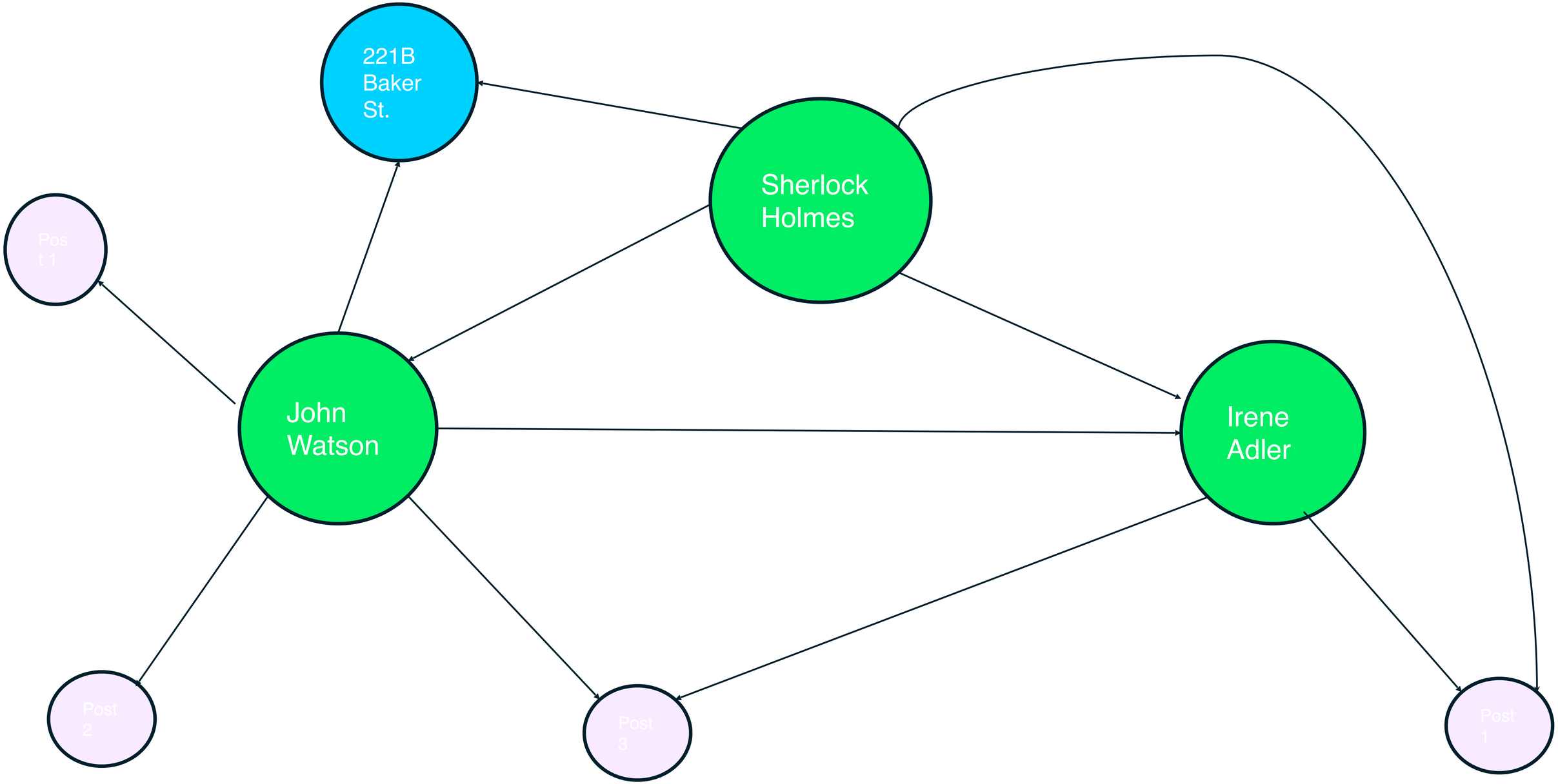
- Captures connected data
- Each element is stored as a node
- Connections between nodes are called links or relationships

Strength

- Traverses the connections between data rapidly

Graph Database

Graph: Example

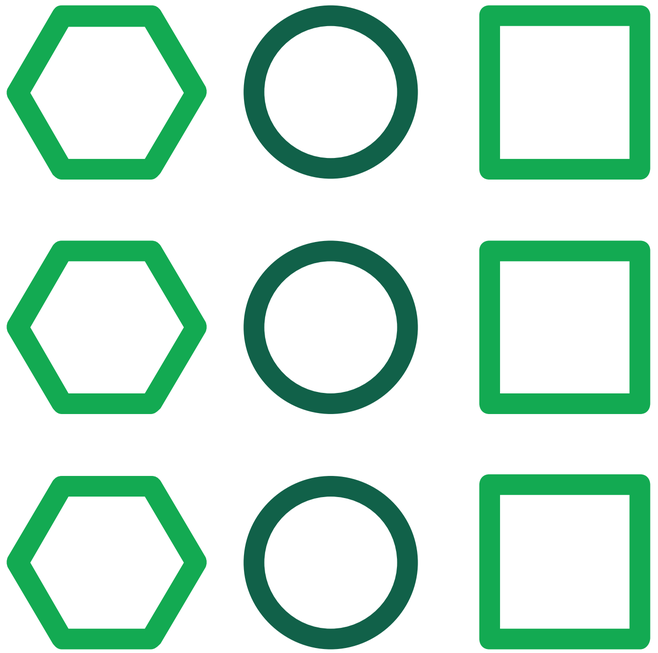


Structure

- 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

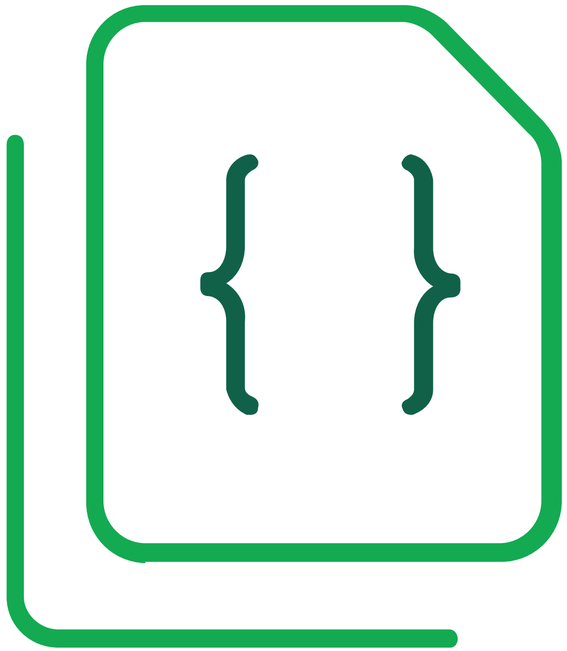
Height	ID
6’2	001
5’9	002
5’7	003

Structure

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

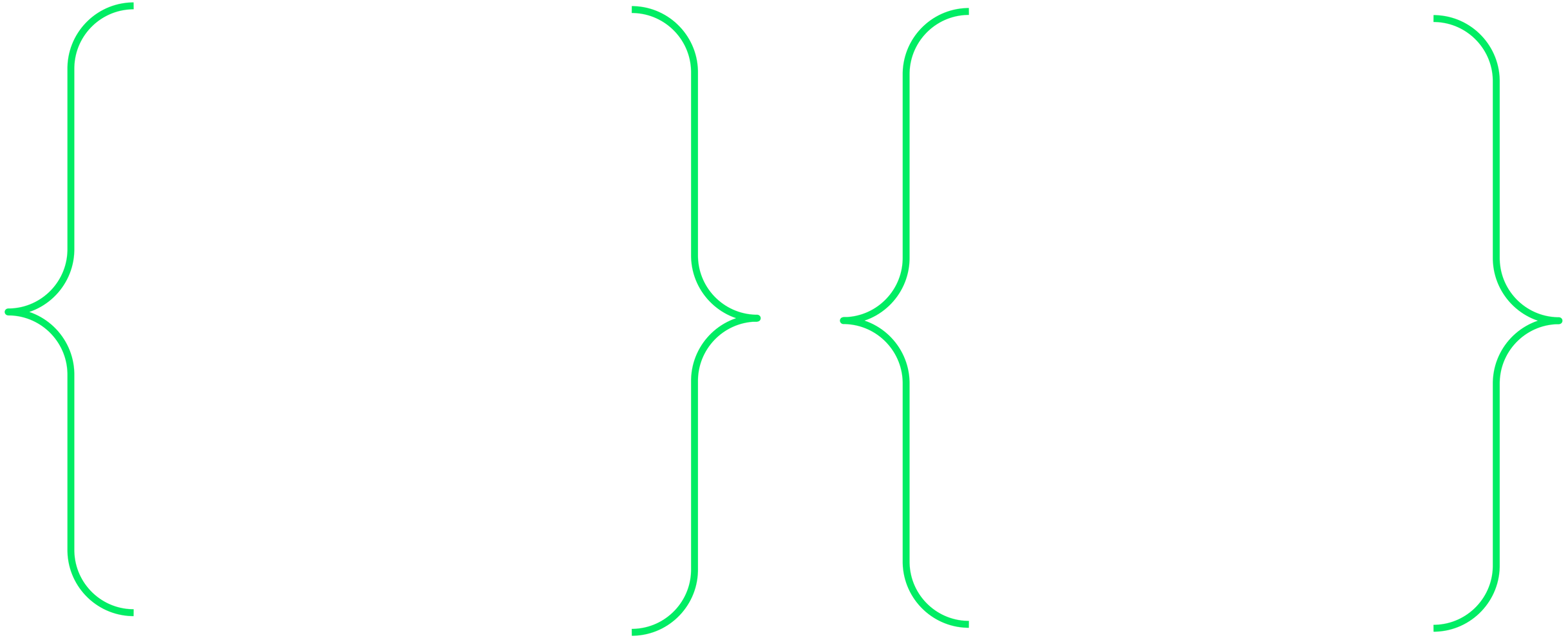
Strengths

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

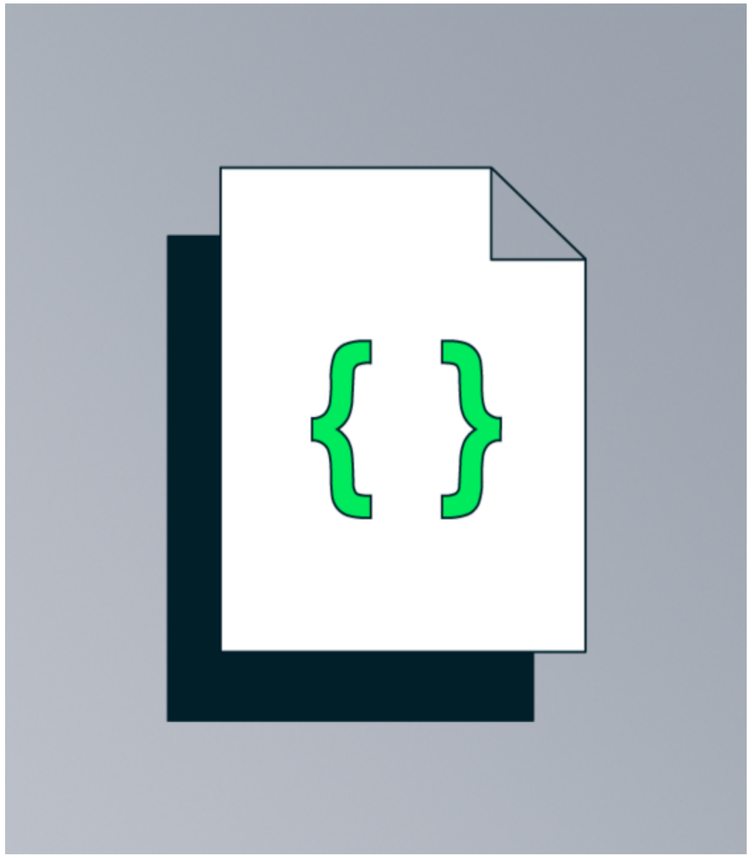


Document Database

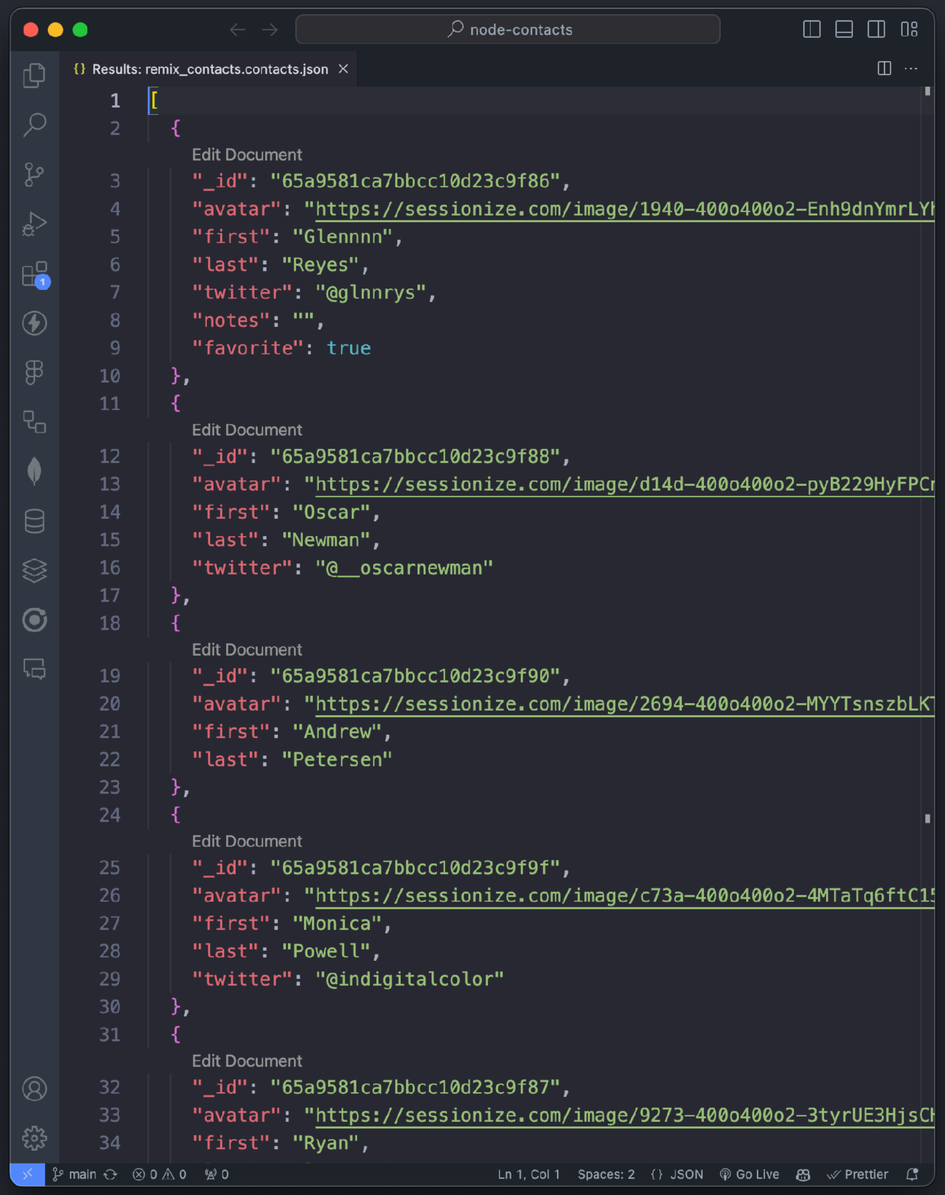
Document Model Example



The Document Model



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

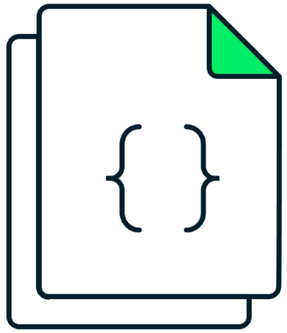


It’s just JSON

- Or in fact, BSON.
- BSON stands for Binary JSON.
- It is a binary-encoded serialization format.
- Designed for efficiency in space and processing compared to plain text JSON.
- Associated with MongoDB as the primary data storage format.
- Includes additional data types like binary data and a richer set of numeric types.
- Suitable for representing complex data structures in a binary form.

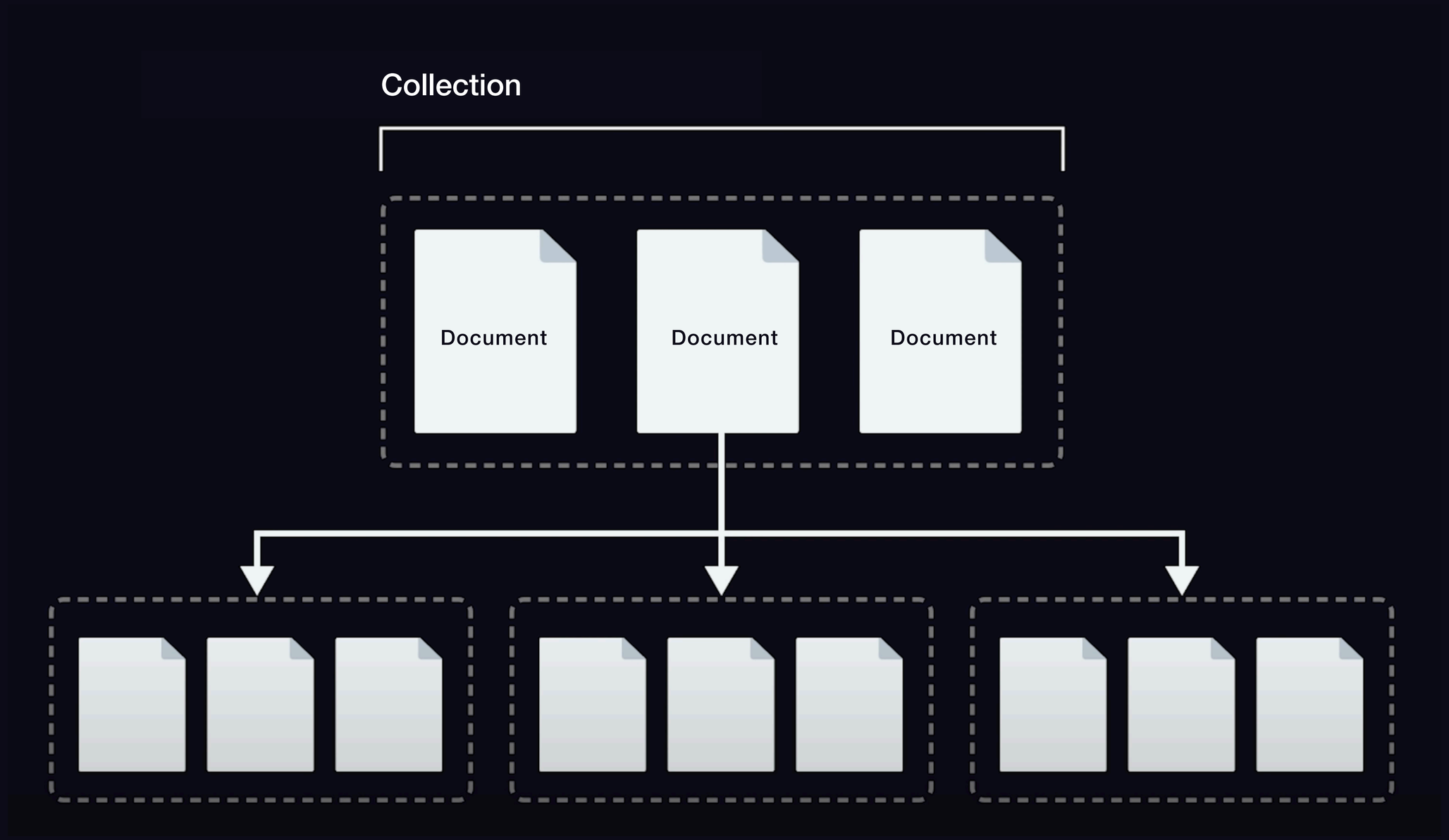
Collections and Documents

Document

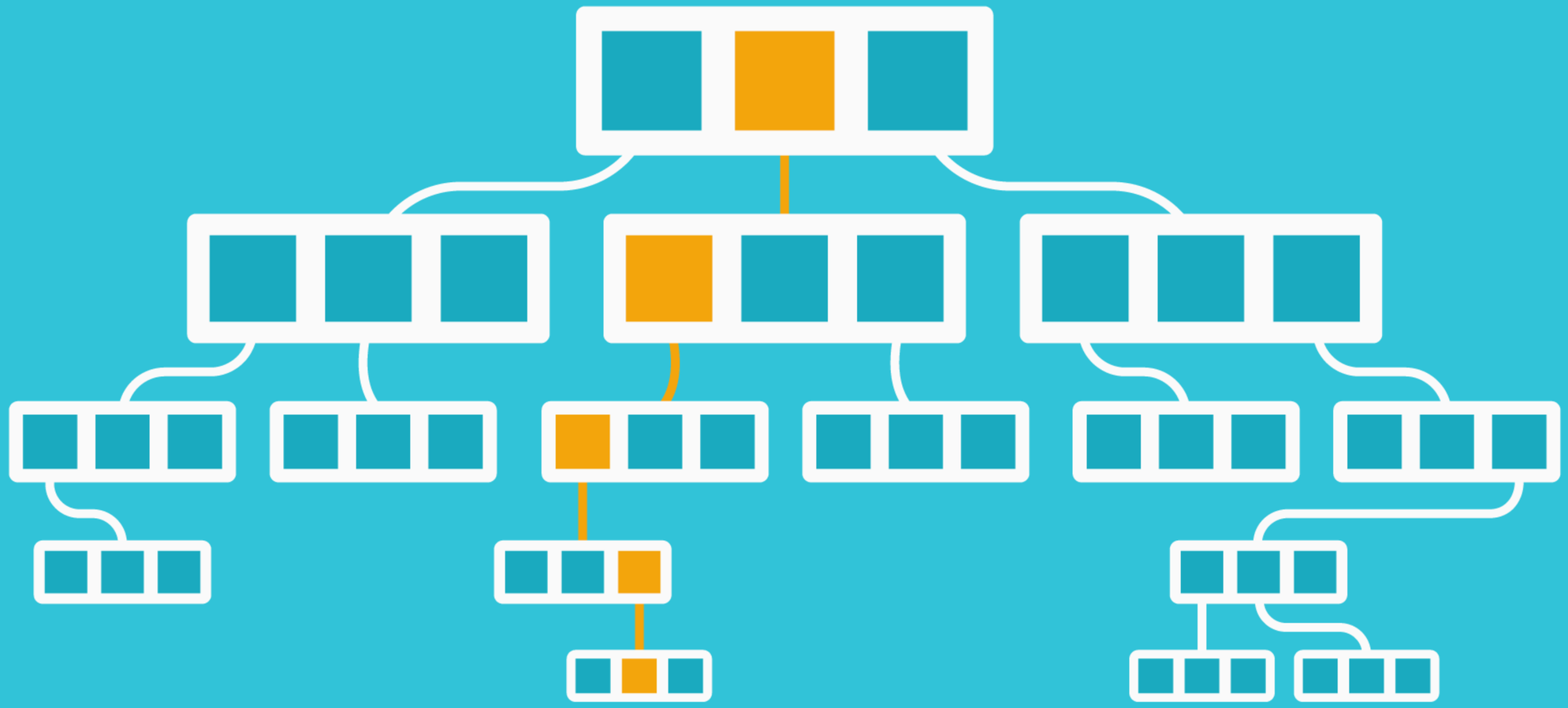


Collection

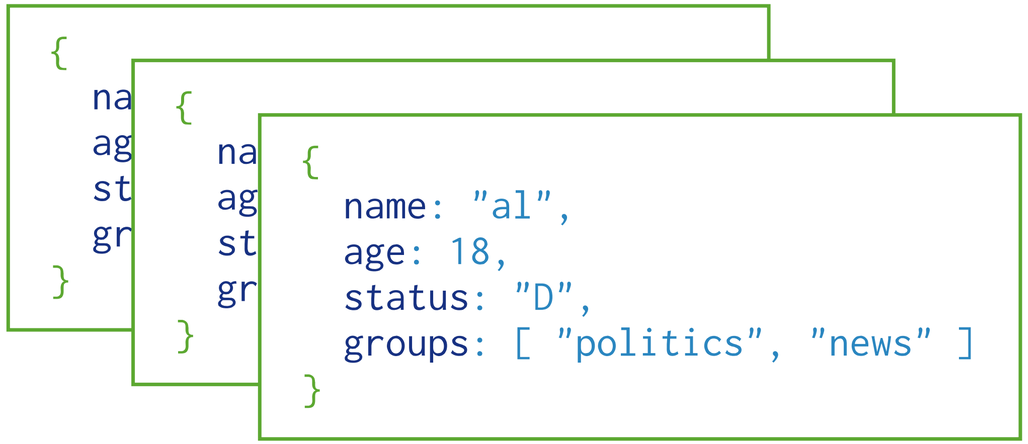
An organized store of documents in MongoDB, usually with common fields between documents


Collection
Document
Document
Document

Collections and Documents



Collections and Documents

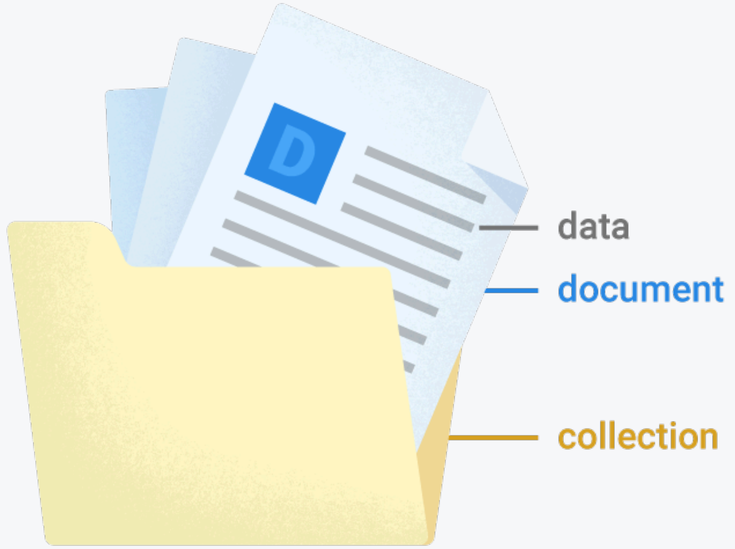


Collection

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

field: value
field: value
field: value
field: value

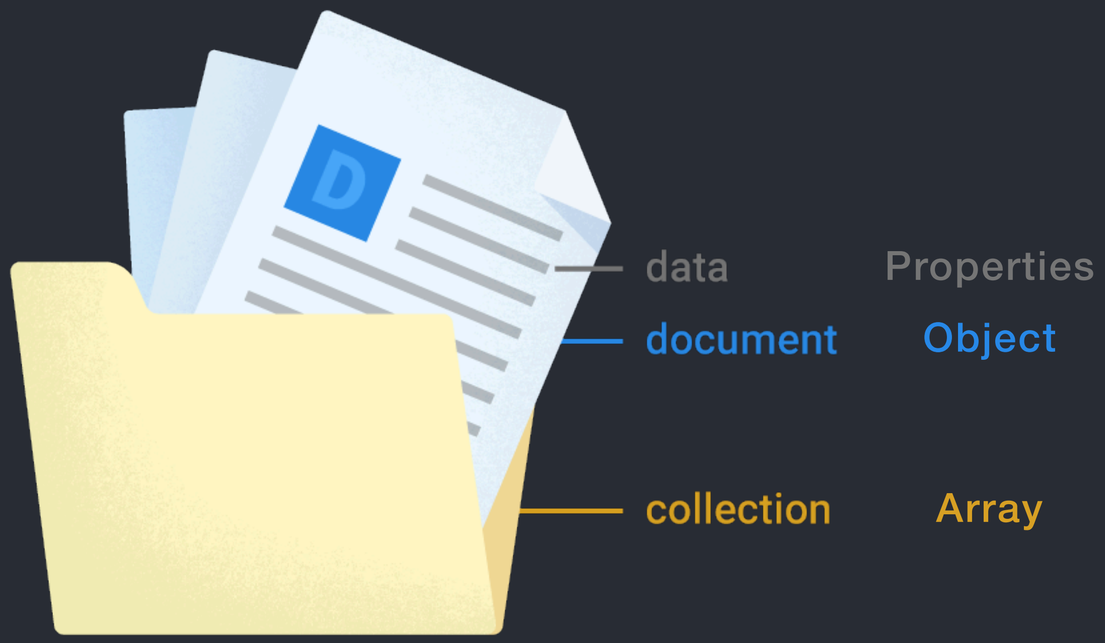
Document



users > seme0qoWIzQh...

mdu-e18front	users	seme0qoWIzQhZDfI1m0L
+ Start collection	+ Add document	+ Start collection
users >	IS35ivFQOBBYrObwZX7k	+ Add field
	cNzvoH8XSRib6XXiUYuX	mail: "bki@eaaa.dk"
	seme0qoWIzQhZDfI1m0L >	name: "Birgitte"

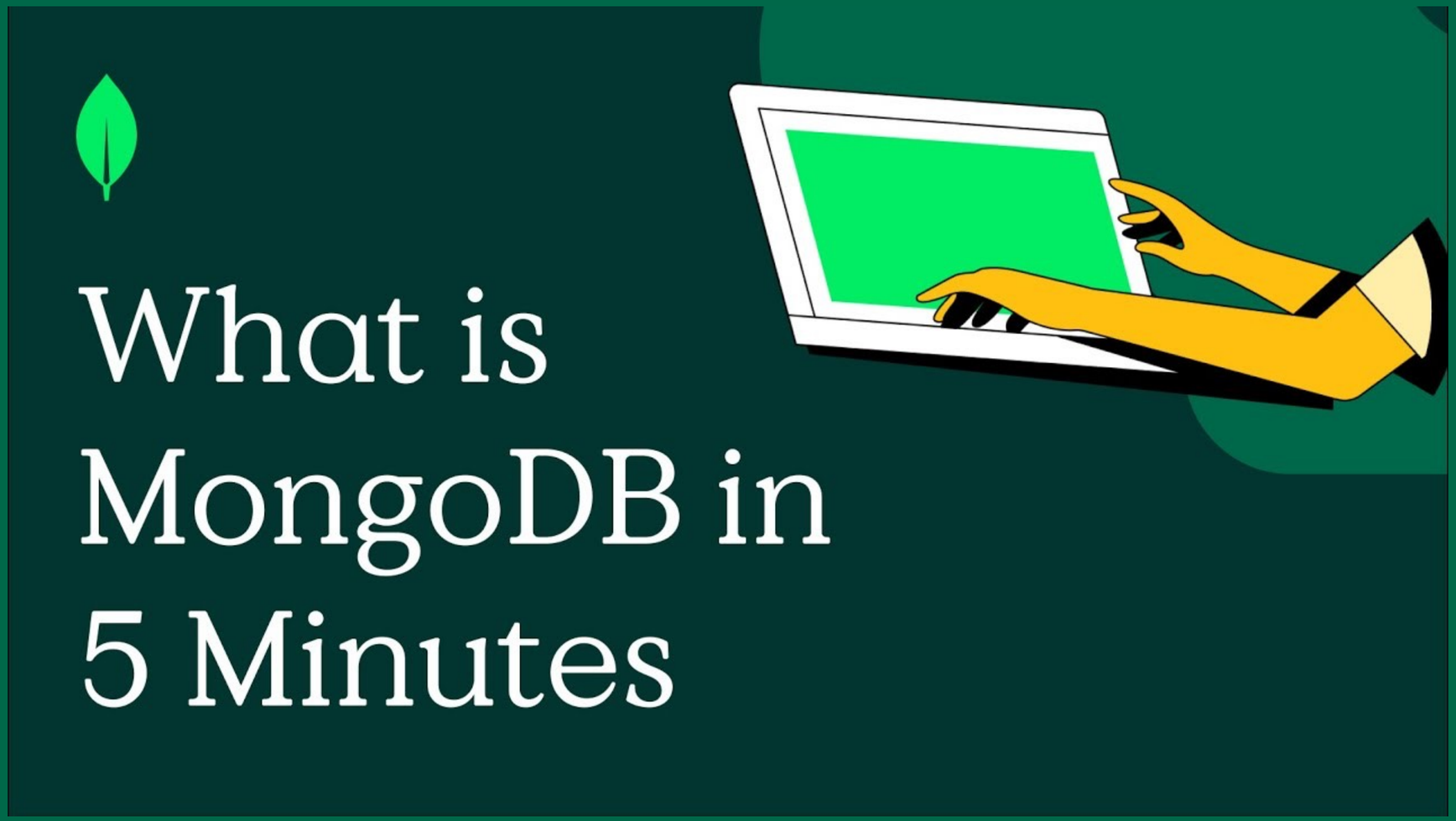
Collection & document vs array & object




```
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
    }
];
```

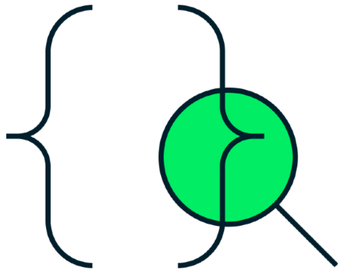
```
{
    "first_name": "Paul",
    "surname": "Miller",
    "cell": "447557505611",
    "city": "London",
    "location": [45.123, 47.232],
    "profession": ["banking", "finance", "trader"],
    "cars": [
        {
            "model": "Bentley",
            "year": 1973
        },
        {
            "model": "Rolls Royce",
            "year": 1965
        }
    ]
}
```

The Document Model and MongoDB

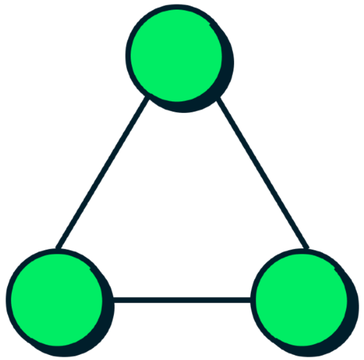


https://www.youtube.com/watch?v=EE8ZTQxa0AM&ab_channel=MongoDB

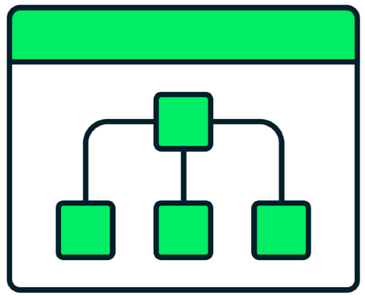
Key Features



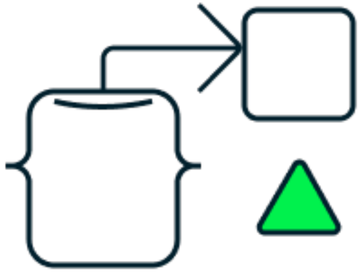
API query or query language



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.

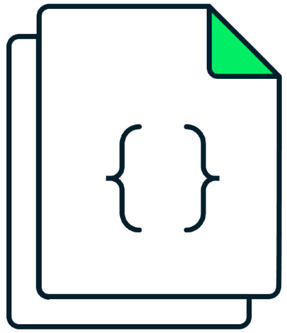
```
{
  "_id": ObjectId(
   "5f4f7fef2d4b45b7f11b6d7a"),
  "user_id": "Sean",
  "age": 29,
  "Status": "A"
}
```

The Document Model: Structure and Syntax

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

Collections in the Document Model

Document



Collection

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"
}
```

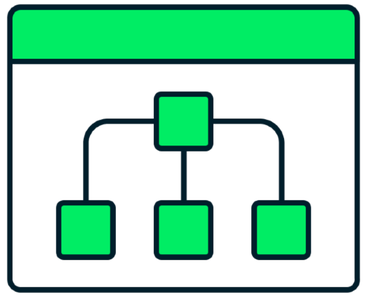
```
{
  "_id": ObjectId(
"5f4f7fef2d4b45b7f11b6d7a"),
  "user_id": "Daniel",
  "age": 25,
  "Status": "A",
  "Country": "USA"
}
```

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.

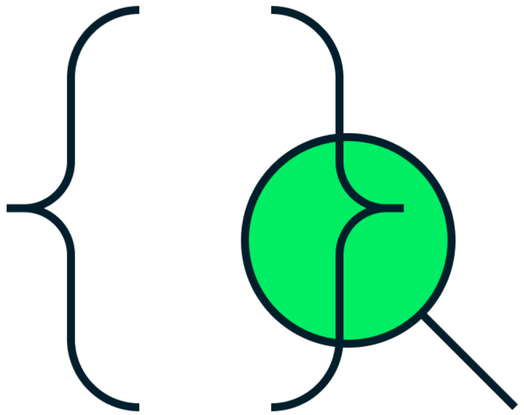
```
db.createCollection("users", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         required: ["image", "mail", "name", "title"],
         properties: {
            _id: {
               bsonType: "objectId",
               description: "must be a unique ObjectId"
         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

MQL
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

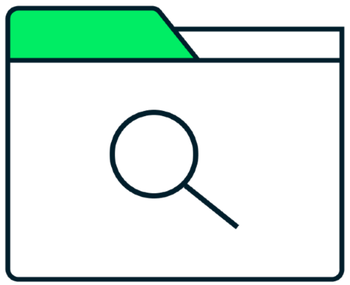
db.collection.find({ <field1>: <value1>, ... })

Specifying query operators

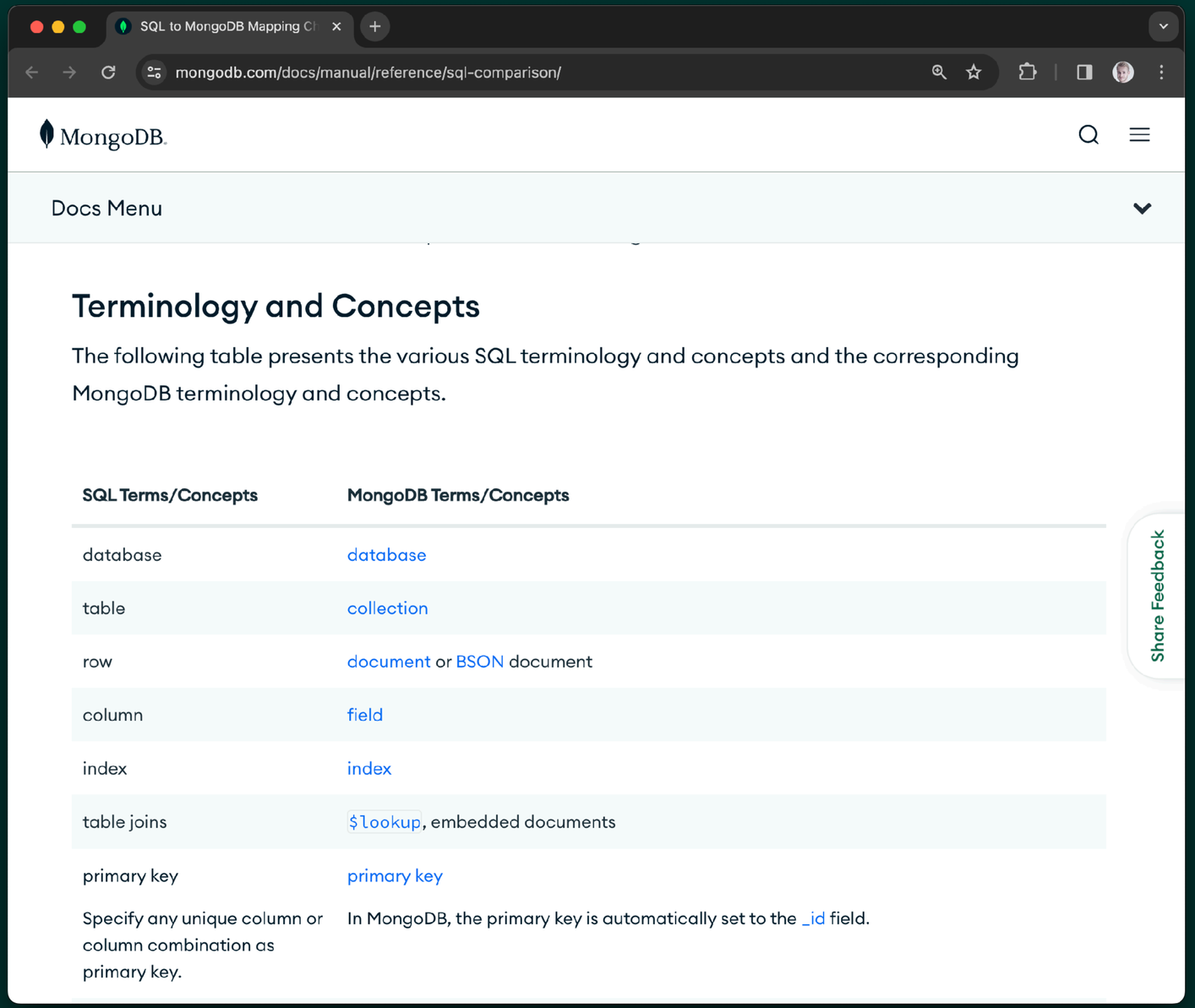
db.<collection>.find({ <field1>: { <operator1>: <value1> }, ... })

MQL Find(): Important Note

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



SQL vs MQL



https://www.mongodb.com/docs/manual/reference/sql-comparison/

SQL Schema Statements

```
CREATE TABLE people (
    id MEDIUMINT NOT NULL
        AUTO_INCREMENT,
    user_id Varchar(30),
    age Number,
    status char(1),
    PRIMARY KEY (id)
)
```

MongoDB Schema Statements

Implicitly created on first `insertOne()` or `insertMany()` operation. The primary key `_id` is automatically added if `_id` field is not specified.

```
db.people.insertOne( {
    user_id: "abc123",
    age: 55,
    status: "A"
} )
```

https://www.mongodb.com/docs/manual/reference/sql-comparison/

SQL INSERT Statements	MongoDB insertOne() Statements

```
INSERT INTO people(user_id,
                   age,
                   status)
VALUES ("bcd001",
        45,
        "A")
```

```
db.people.insertOne(
   { user_id: "bcd001", age: 45, status: "A"
)
```

https://www.mongodb.com/docs/manual/reference/sql-comparison/

SQL SELECT Statements	MongoDB find() Statements
`SELECT *` `FROM people`	`db.people.find()`

`SELECT id,`
`user_id,`
`status`
`FROM people`
	`db.people.find(`
	`{ },`
	`{ user_id: 1, status: 1 }`
	`)`

`SELECT user_id, status`
`FROM people`
	`db.people.find(`
	`{ },`
	`{ user_id: 1, status: 1, _id: 0 }`
	`)`

`SELECT *`
`FROM people`
`WHERE status = "A"`
	`db.people.find(`
	`{ status: "A" }`
	`)`

SQL Update Statements	MongoDB updateMany() Statements

```
UPDATE people
SET status = "C"
WHERE age > 25
```

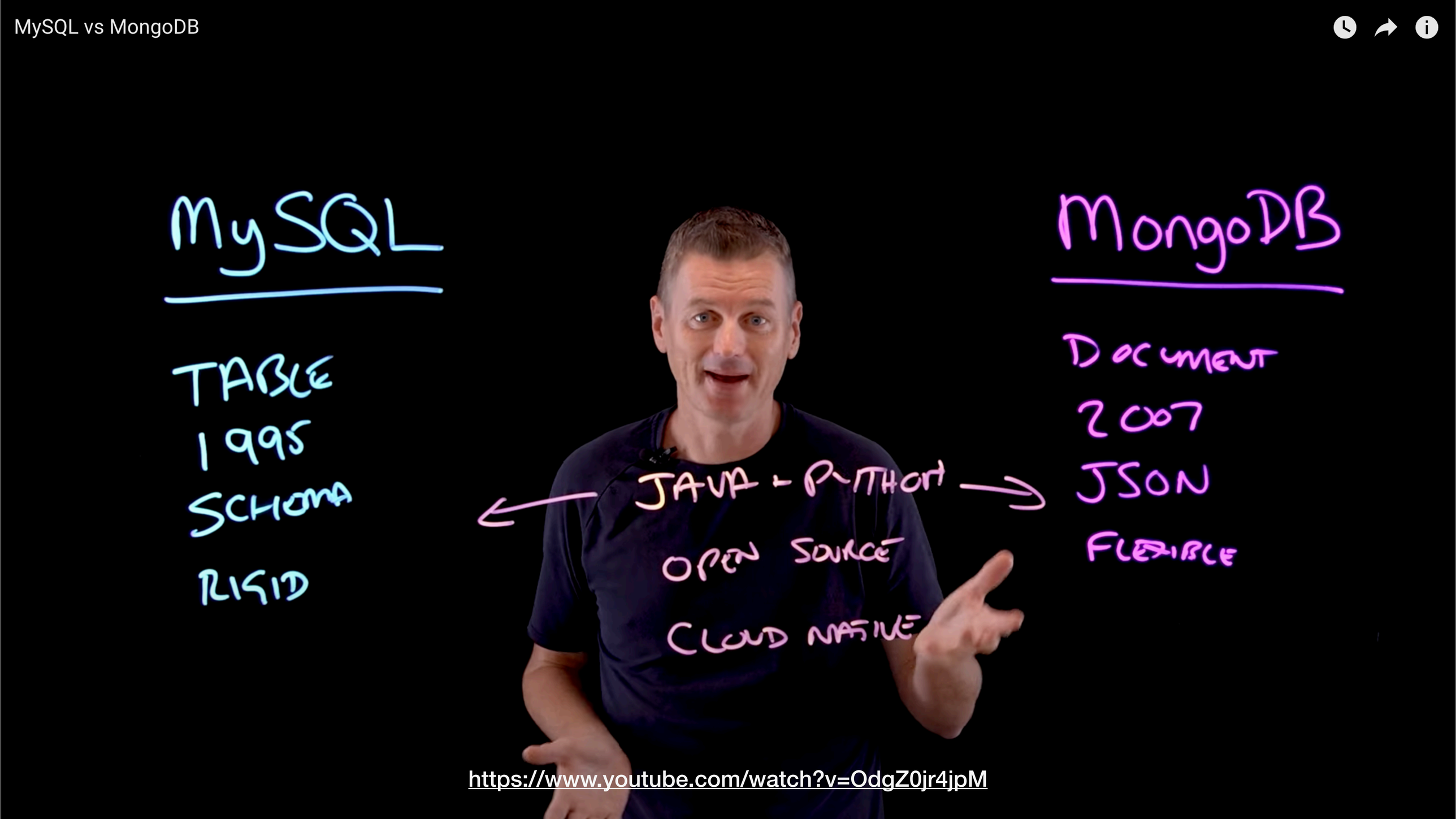
```
db.people.updateMany(
   { age: { $gt: 25 } },
   { $set: { status: "C" } }
)
```

```
UPDATE people
SET age = age + 3
WHERE status = "A"
```

```
db.people.updateMany(
   { status: "A" } ,
   { $inc: { age: 3 } }
)
```

https://www.mongodb.com/docs/manual/reference/sql-comparison/

MySQL vs MongoDB



https://www.youtube.com/watch?v=OdgZ0jr4jpM

Differences between SQL and NoSQL

	SQL Databases	NoSQL Databases
Data Storage Model	Tables with fixed rows and columns	Document: JSON documents, Key-value: key-value pairs, Wide-column: tables with rows and dynamic columns, Graph: nodes and edges
Development History	Developed in the 1970s with a focus on reducing data duplication	Developed in the late 2000s with a focus on scaling and allowing for rapid application change driven by agile and DevOps practices.
Examples	Oracle, MySQL, Microsoft SQL Server, and PostgreSQL	Document: MongoDB and CouchDB, Key-value: Redis and DynamoDB, Wide-column: Cassandra and HBase, Graph: Neo4j and Amazon Neptune
Primary Purpose	General purpose	Document: general purpose, Key-value: large amounts of data with simple lookup queries, Wide-column: large amounts of data with predictable query patterns, Graph: analyzing and traversing relationships between connected data
Schemas	Rigid	Flexible
Scaling	Vertical (scale-up with a larger server)	Horizontal (scale-out across commodity servers)
Multi-Record ACID Transactions	Supported	Most do not support multi-record ACID transactions. However, some – like MongoDB – do.
Joins	Typically required	Typically not required
Data to Object Mapping	Requires ORM (object-relational mapping)	Many do not require ORMs. MongoDB documents map directly to data structures in most popular programming languages.

Features: SQL vs NoSQL

Feature	SQL Databases	NoSQL Databases
Data Structure	Structured	Flexible (Document, Key-Value, Graph, Wide-Column)
Data Types	Structured	Structured, Semi-structured, Unstructured
Scalability	Vertical (increasing resources on a single server)	Horizontal (adding more servers to a cluster)
ACID Compliance	Yes	No (may support eventual consistency)
Query Language	Structured Query Language (SQL)	Less structured query languages or no specific query language
Applications	E-commerce, Financial Systems, Enterprise Applications	Web Applications, Social Media Platforms, Mobile Apps
Strengths	Data organization, Complex queries, ACID compliance	Flexibility, Scalability, Handling unstructured data
Weaknesses	Rigid schema, Slow performance for large data	Less mature technology, Less standardized query syntax

Export to Sheets

https://g.co/bard/share/9a4c7d0aa722

Features: MySQL vs MongoDB

Feature	MySQL	MongoDB
Data Structure	Relational (tables, rows, columns)	Document (JSON-like documents)
Schema	Rigid, requires predefined schema	No schema, flexible data structure
Data Types	Structured	Structured, Semi-structured, Unstructured (JSON, XML, binary data)
Scalability	Vertical (increasing resources on a single server)	Horizontal (adding more servers to a cluster)
ACID Compliance	Yes	Eventual consistency
Query Language	Structured Query Language (SQL)	MongoDB Query Language (MongoDBQL)
Applications	E-commerce, Financial Systems, Enterprise Applications	Web Applications, Social Media Platforms, Mobile Apps, Real-time applications
Strengths	Data organization, Complex queries, ACID compliance, Mature technology	Flexibility, Scalability, Handling unstructured data, Real-time data handling
Weaknesses	Rigid schema, Slow performance for large data	Less mature technology, Less standardized query syntax, May not support complex queries

Export to Sheets

https://g.co/bard/share/1233c9c2da05

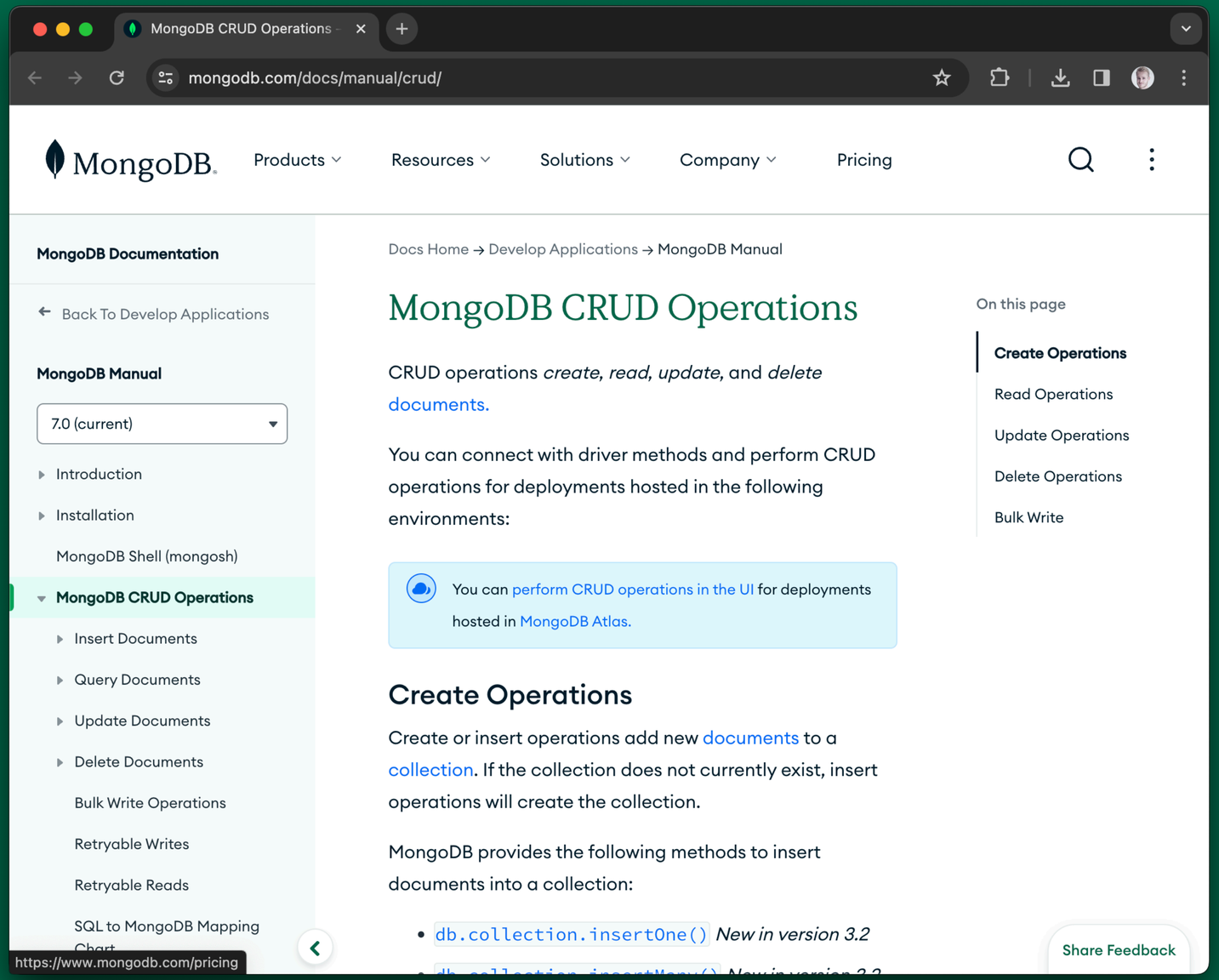
Best For: MySQL vs MongoDB

Feature	MySQL	MongoDB
Best for	Applications with structured data, complex queries, and strict data integrity	Applications with unstructured or evolving data, high scalability, and real-time data handling

Export to Sheets

https://g.co/bard/share/1233c9c2da05

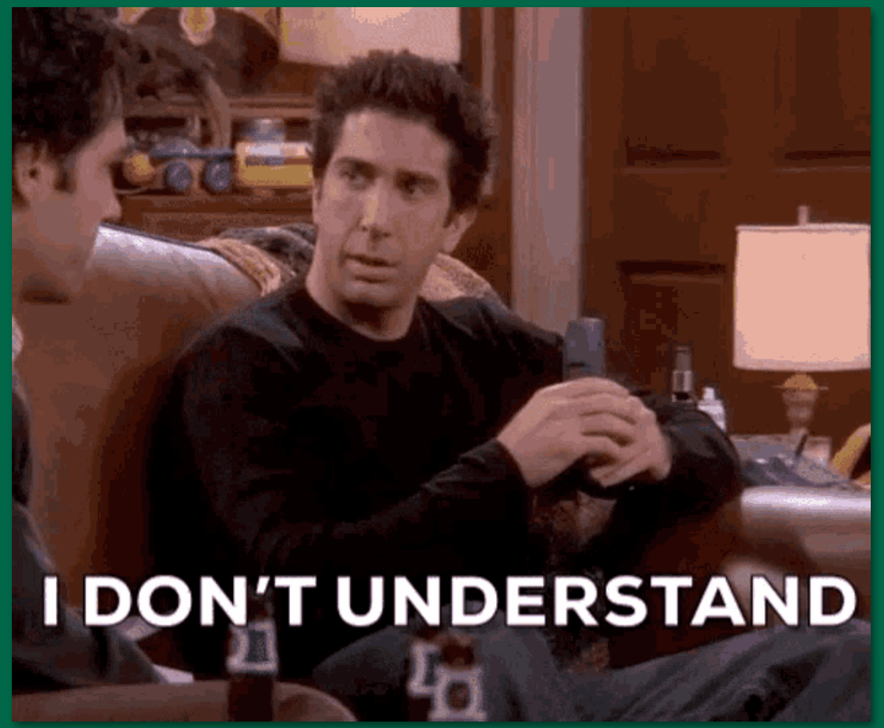
MongoDB CRUD Operations



https://www.mongodb.com/docs/manual/crud/

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(
  {
    name: "sue",
    age: 26,
    status: "pending"
  }
)
```

collection

field: value

field: value

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=r
      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=r
      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
      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
      uid: ObjectId("fjpRTRTjZHwr")
    },
    //...
  ]
)
```

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

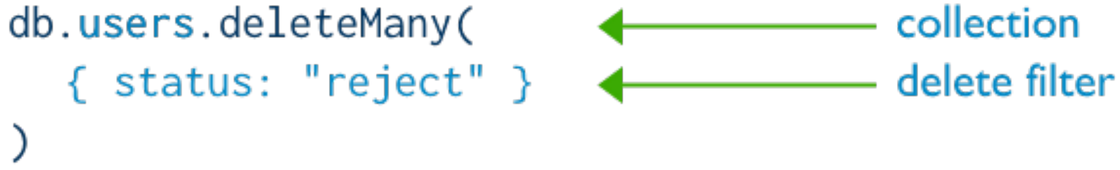
```
db.users.updateOne(
    // Specify the user you want to update
    { _id: ObjectId("ZfPTVEMQKf9v") },
    {
        $set: {
            // Update the name field with the new value
            name: "New Name",
            // Update the title field with the new value
            title: "New Title"
            // 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

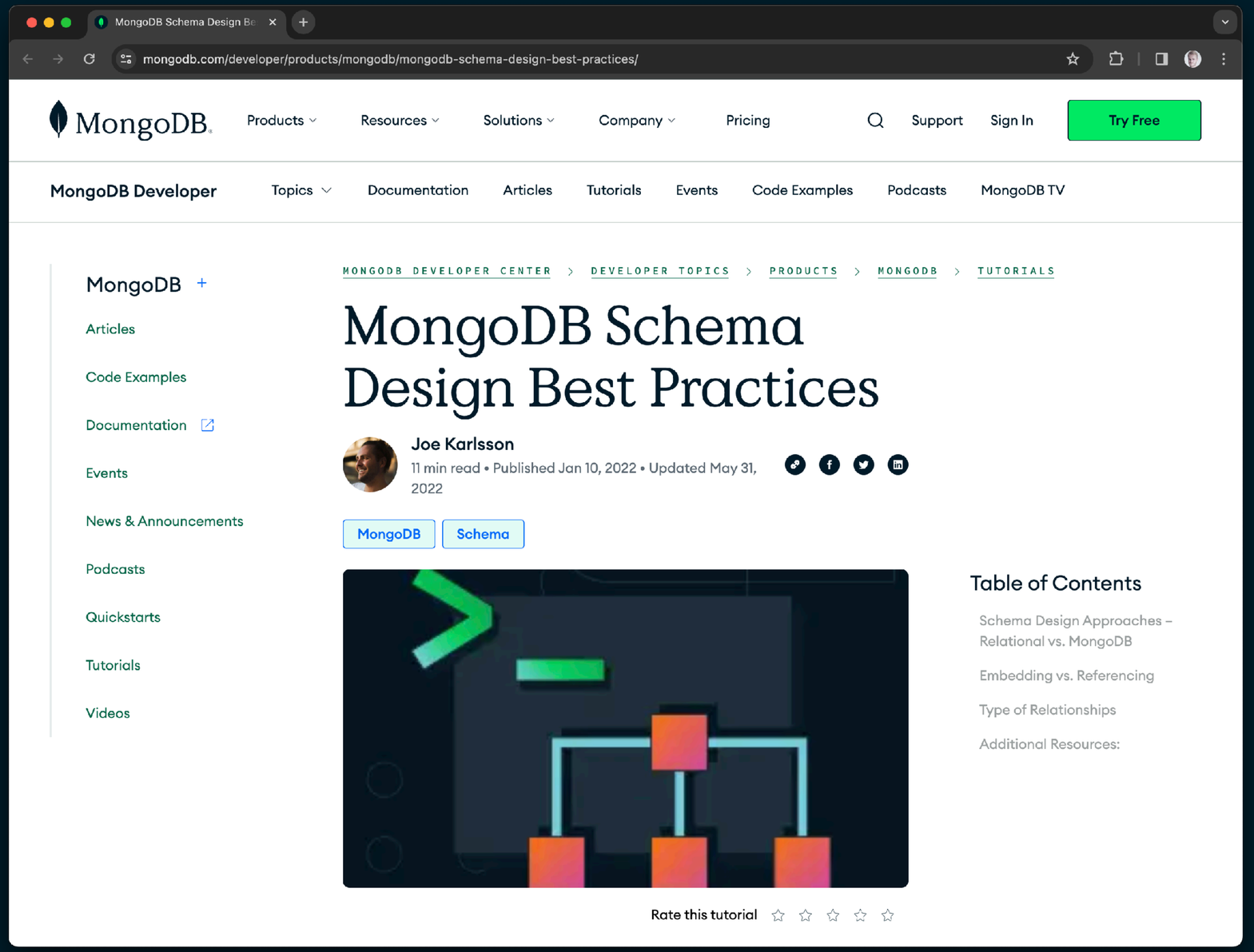
```
db.users.replaceOne(
    // Specify the user you want to replace
    { _id: ObjectId("ZfPTVEMQKf9v") },
    {
        _id: ObjectId("ZfPTVEMQKf9v"),
        image: "https://new-image-url.com",
        mail: "new-email@example.com",
        // Add more fields as needed
    }
);
```

Delete




```
db.users.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



https://www.mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/

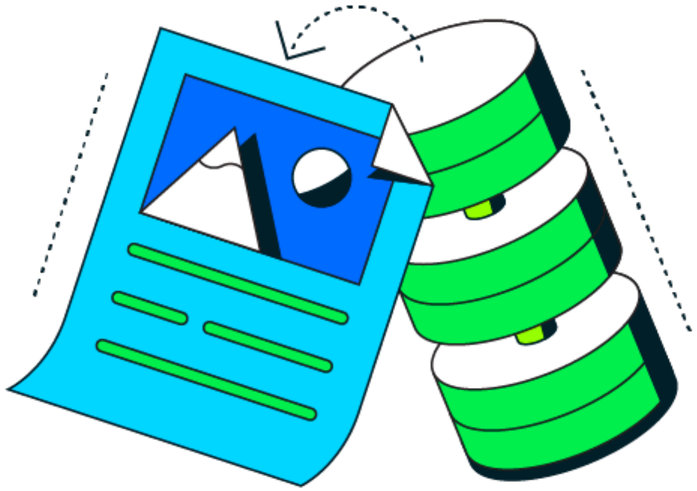
MongoDB Schema Design

- 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.

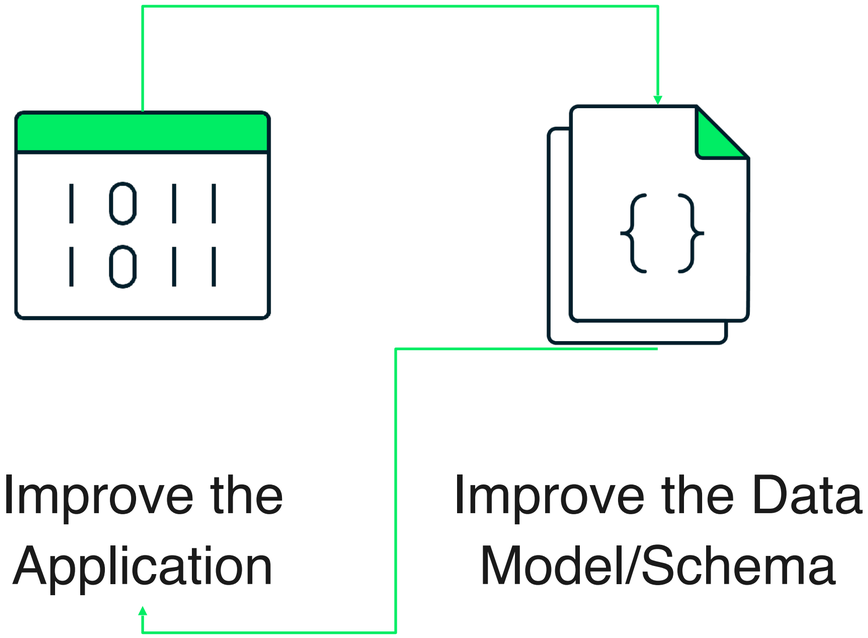


Data Modeling and the Document Model

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

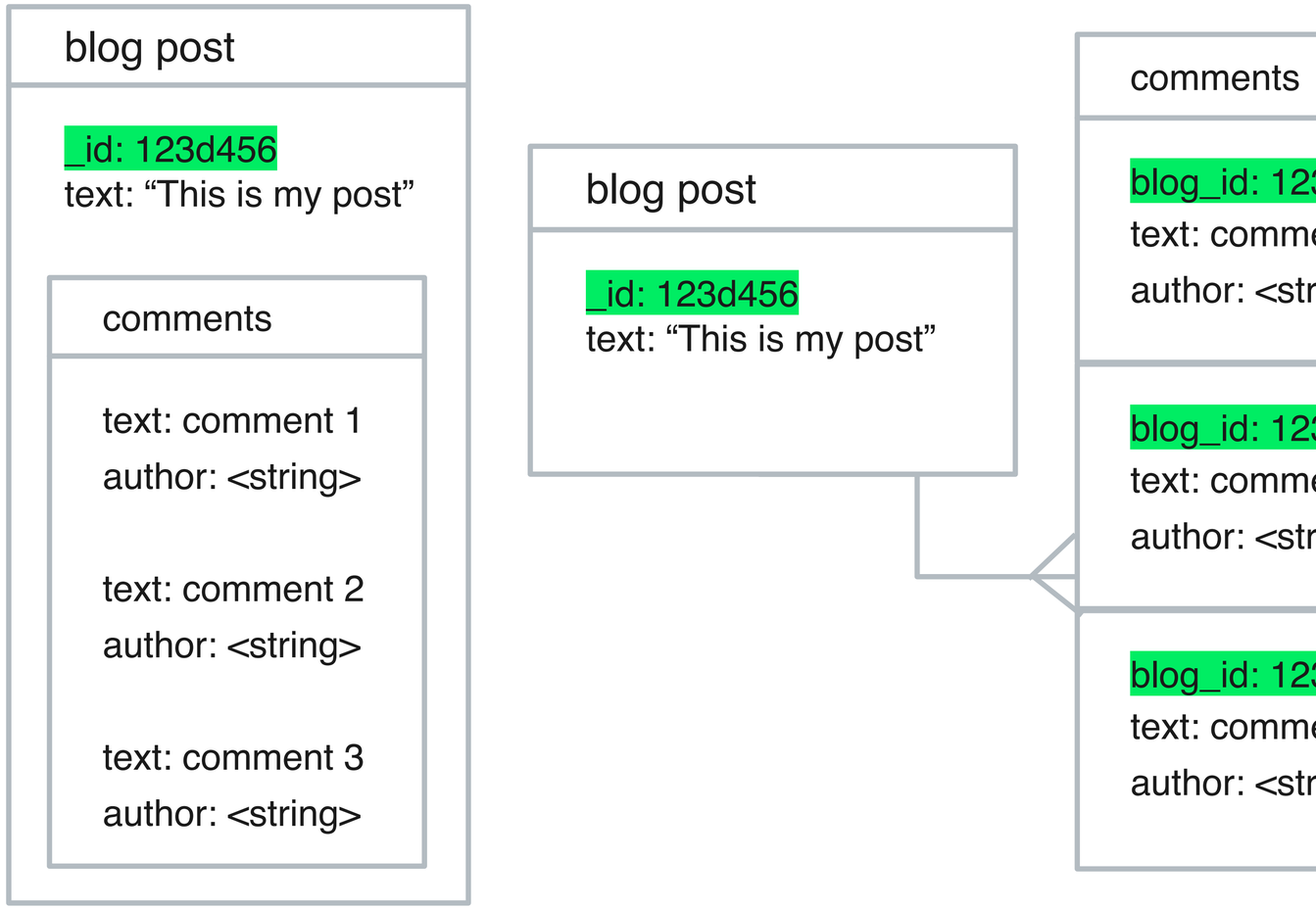
- Your queries and the specific data your application requires.
- How your application reads the data (read patterns).
- How your application writes the data (write patterns).
- What are the relationships between your data (linked or embedded).

Schema Design - Link or Embed?

Embedded vs Linked relationship in the Post-Comment example

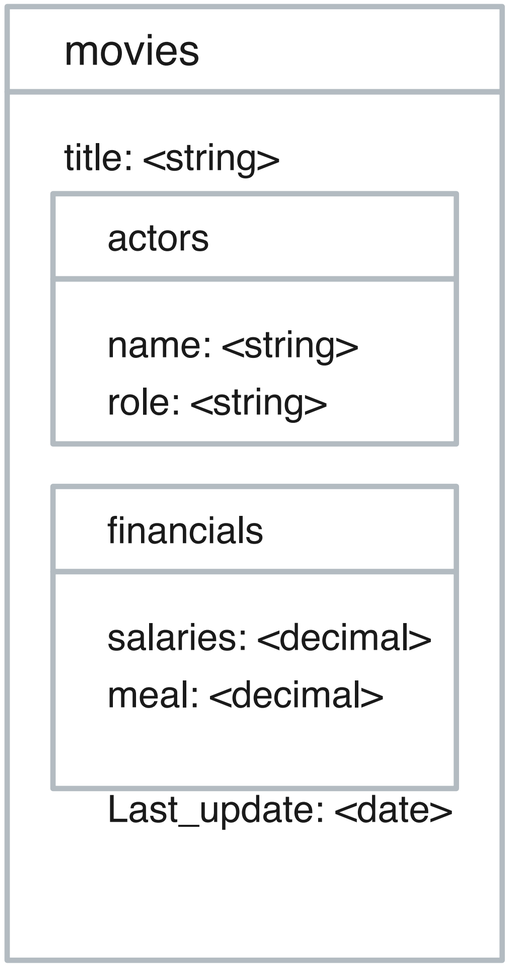
Schema Design - Link or Embed?

- 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?

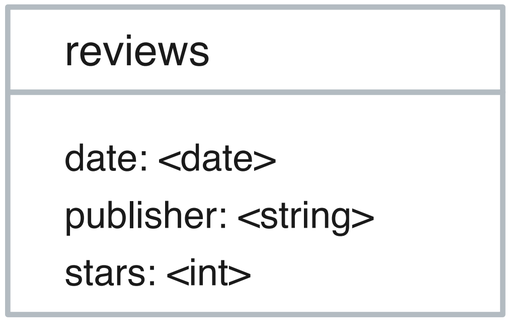


Example: Movies and Reviews

Embedded



Linked



Relationships

One to one (1-1)

One to many (1-N)

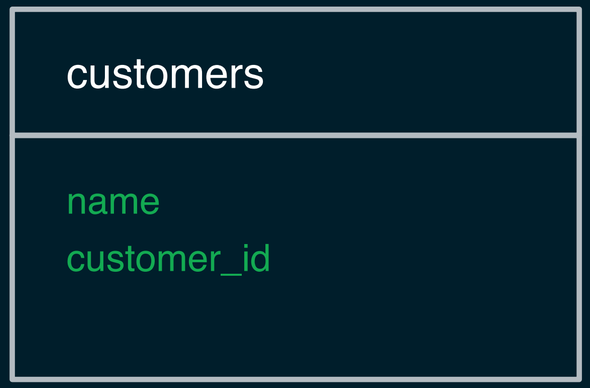
Many to many (N-N)

Relationships and Data Modeling

Relationships

One-to-One (1-1)

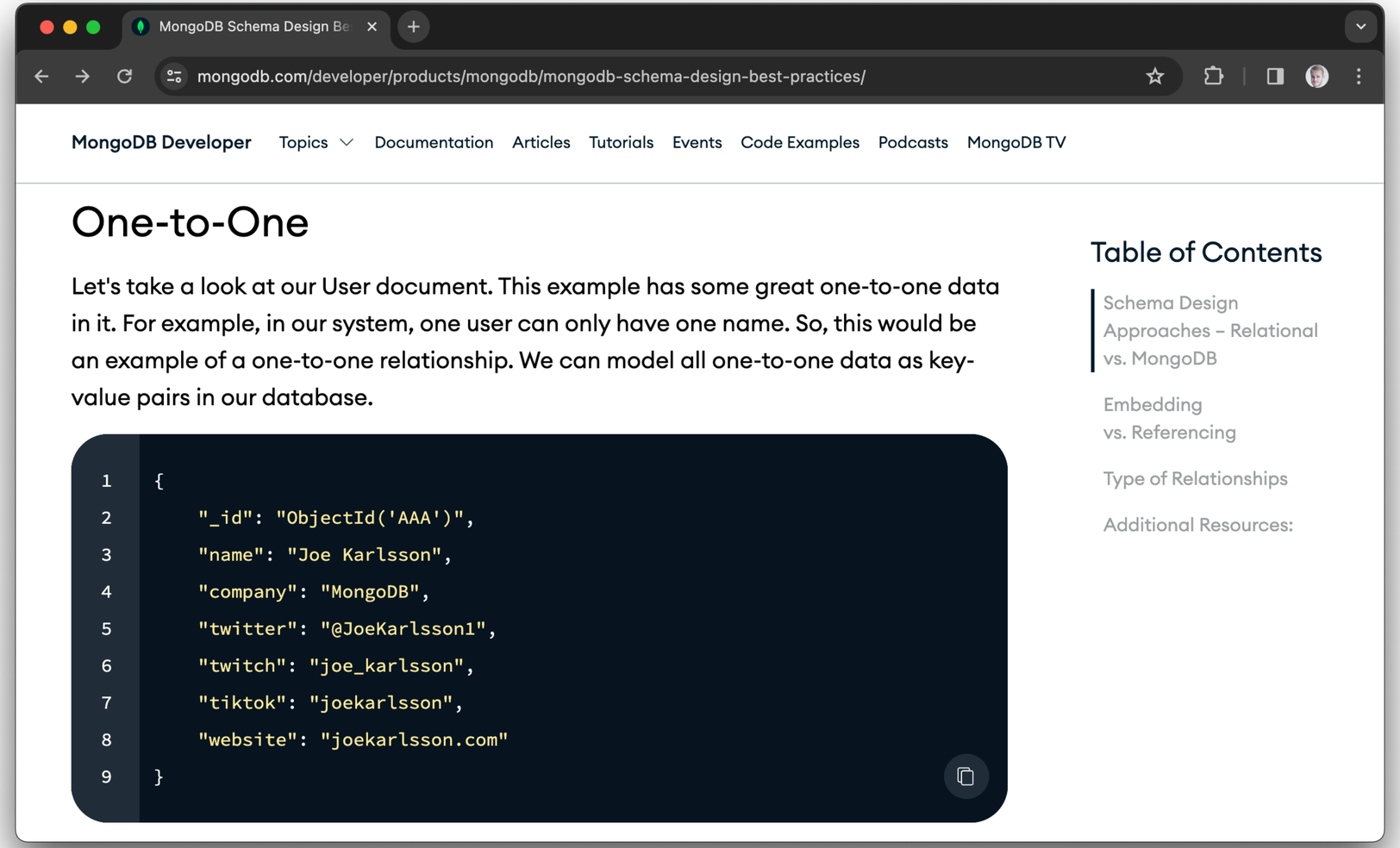
A one-to-one relationship is represented and stored in a single document, this would typically be data like a person's name and the customer id.



One to One (1-1)

Scenario:

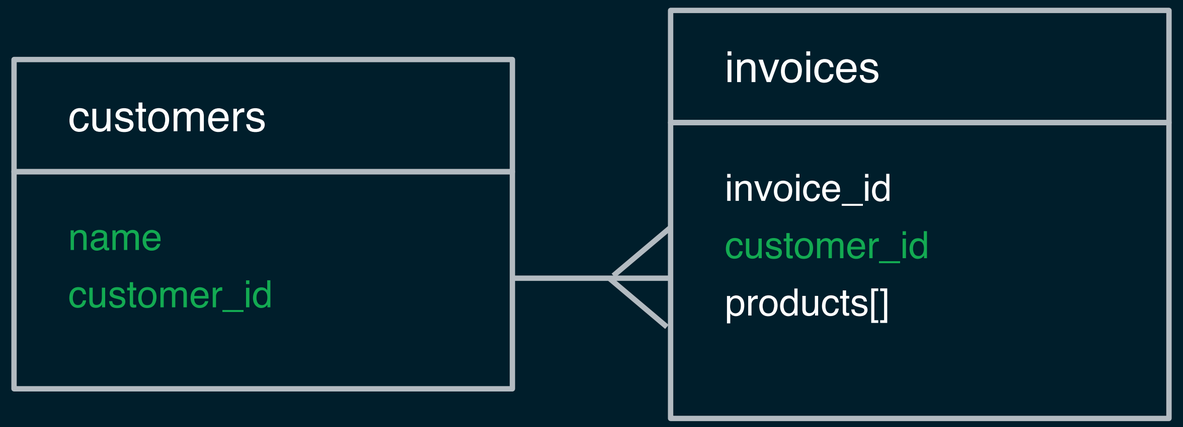
You have to map patron and address relationships. In this example, you'll need to view one data entity in context of the other.


MongoDB Schema Design Be
mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/
MongoDB Developer
Topics
Documentation
Articles
Tutorials
Events
Code Examples
Podcasts
MongoDB TV
One-to-One
Let's take a look at our User document. This example has some great one-to-one data in it. For example, in our system, one user can only have one name. So, this would be an example of a one-to-one relationship. We can model all one-to-one data as key-value pairs in our database.
{
"_id": "ObjectId('AAA')",
"name": "Joe Karlsson",
"company": "MongoDB",
"twitter": "@JoeKarlsson1",
"twitch": "joe_karlsson",
"tiktok": "joekarlsson",
"website": "joekarlsson.com"
}
Table of Contents
Schema Design Approaches – Relational vs. MongoDB
Embedding vs. Referencing
Type of Relationships
Additional Resources:

Relationships

One-to-Many (1-N)

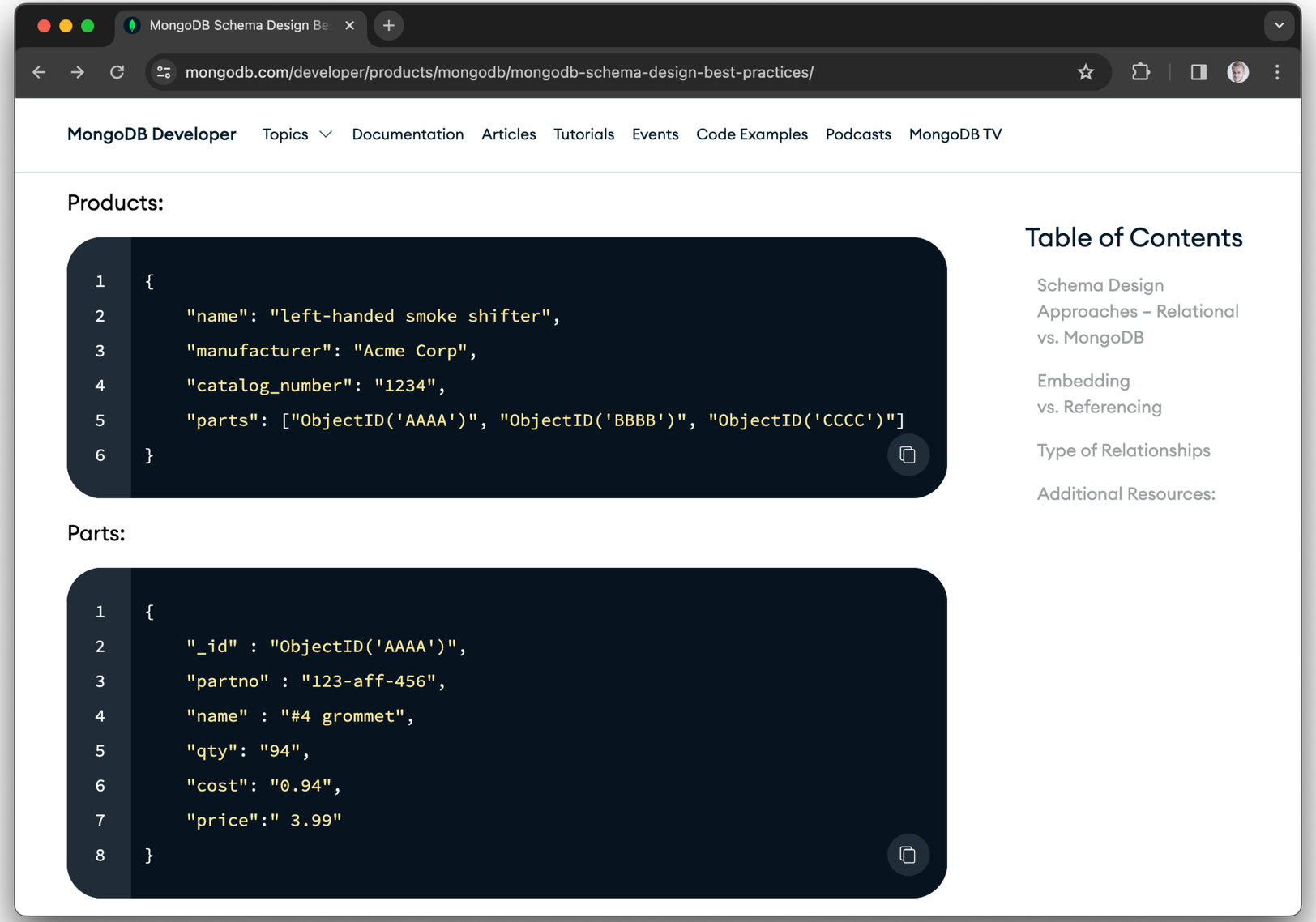
A one-to-many relationship can be considered when an object of a given type is associated with N objects of a second type.

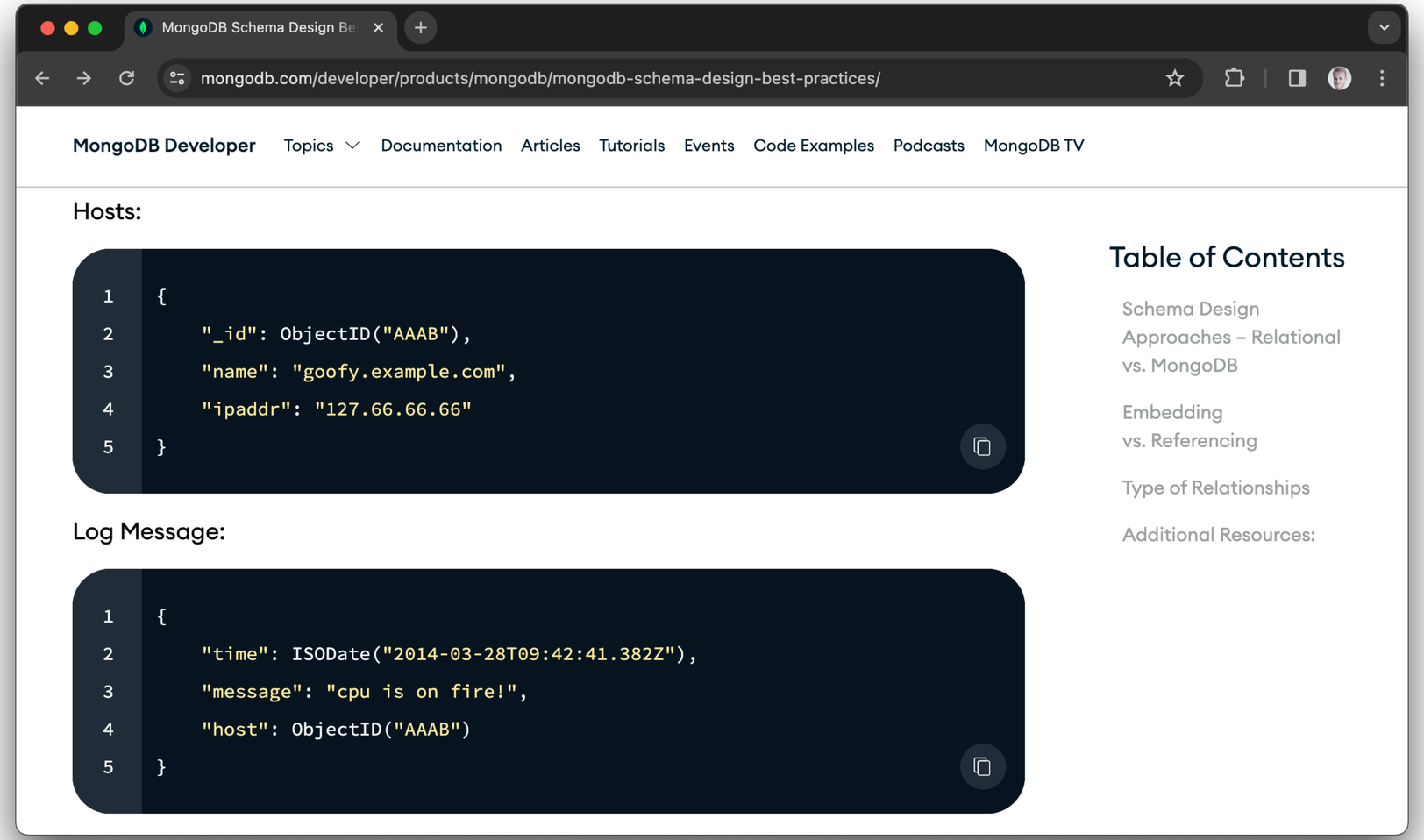


One to Many (1-N)

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.

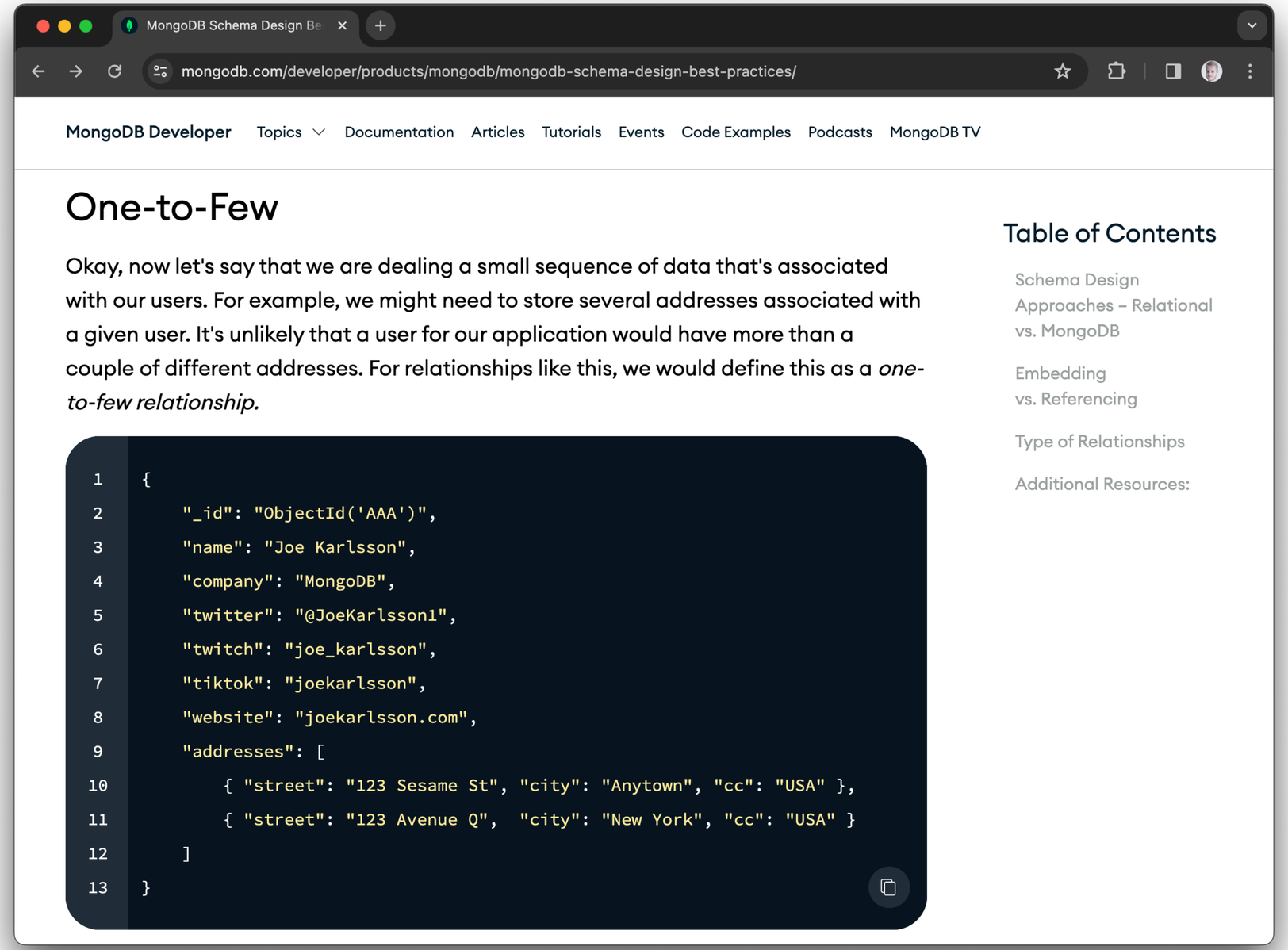

MongoDB Schema Design Be
mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/
MongoDB Developer
Topics
Documentation
Articles
Tutorials
Events
Code Examples
Podcasts
MongoDB TV
Products:
{
"name": "left-handed smoke shifter",
"manufacturer": "Acme Corp",
"catalog_number": "1234",
"parts": ["ObjectID('AAAA')", "ObjectID('BBBB')", "ObjectID('CCCC')"]
}
Parts:
{
"_id" : "ObjectID('AAAA')",
"partno" : "123-aff-456",
"name" : "#4 grommet",
"qty": "94",
"cost": "0.94",
"price":" 3.99"
}
Table of Contents
Schema Design Approaches – Relational vs. MongoDB
Embedding vs. Referencing
Type of Relationships
Additional Resources:


MongoDB Schema Design Be
mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/
MongoDB Developer
Topics
Documentation
Articles
Tutorials
Events
Code Examples
Podcasts
MongoDB TV
Hosts:
{
"_id": ObjectID("AAAB"),
"name": "goofy.example.com",
"ipaddr": "127.66.66.66"
}
Log Message:
{
"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):

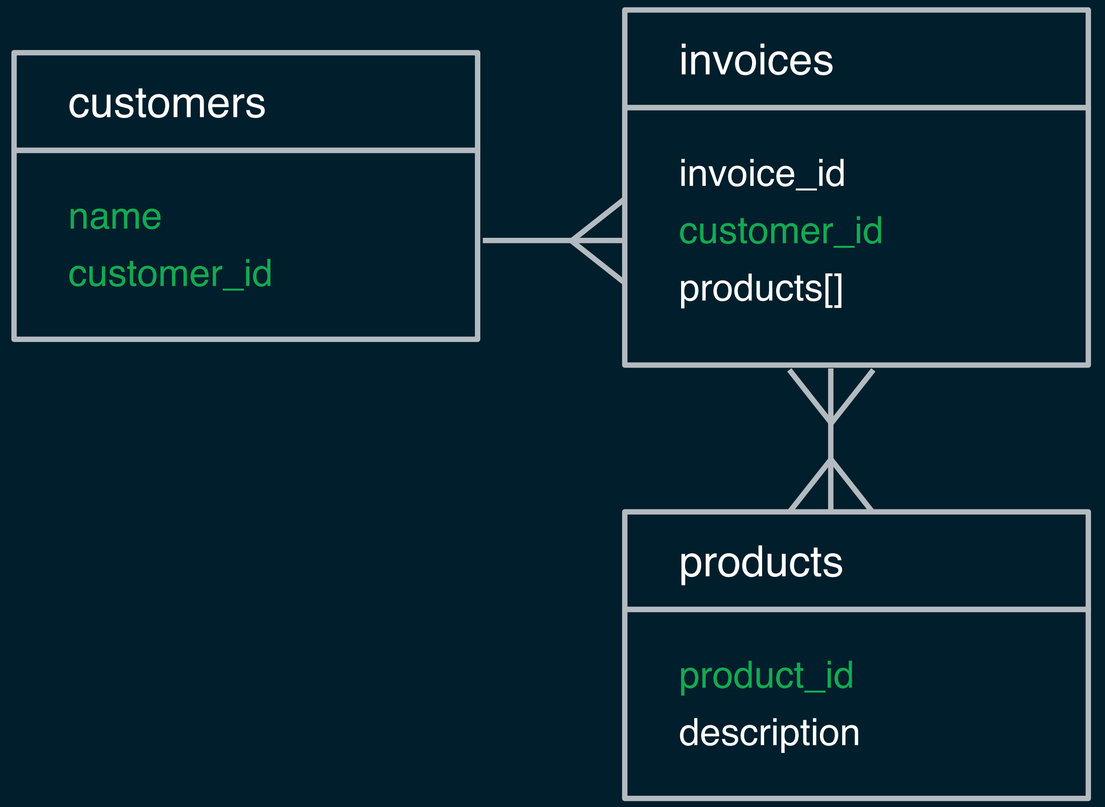
You have to map a patron with multiple address relationships. In this one-to-many relationship between [patron] and [address] data, the [patron] has multiple [address] entities.


MongoDB Schema Design Be
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.*
{
"_id": "ObjectId('AAA')",
"name": "Joe Karlsson",
"company": "MongoDB",
"twitter": "@JoeKarlsson1",
"twitch": "joe_karlsson",
"tiktok": "joekarlsson",
"website": "joekarlsson.com",
"addresses": [
{ "street": "123 Sesame St", "city": "Anytown", "cc": "USA" },
{ "street": "123 Avenue Q", "city": "New York", "cc": "USA" }
]
}
Table of Contents
Schema Design Approaches – Relational vs. MongoDB
Embedding vs. Referencing
Type of Relationships
Additional Resources:

Relationships

Many-to-Many (N-N)

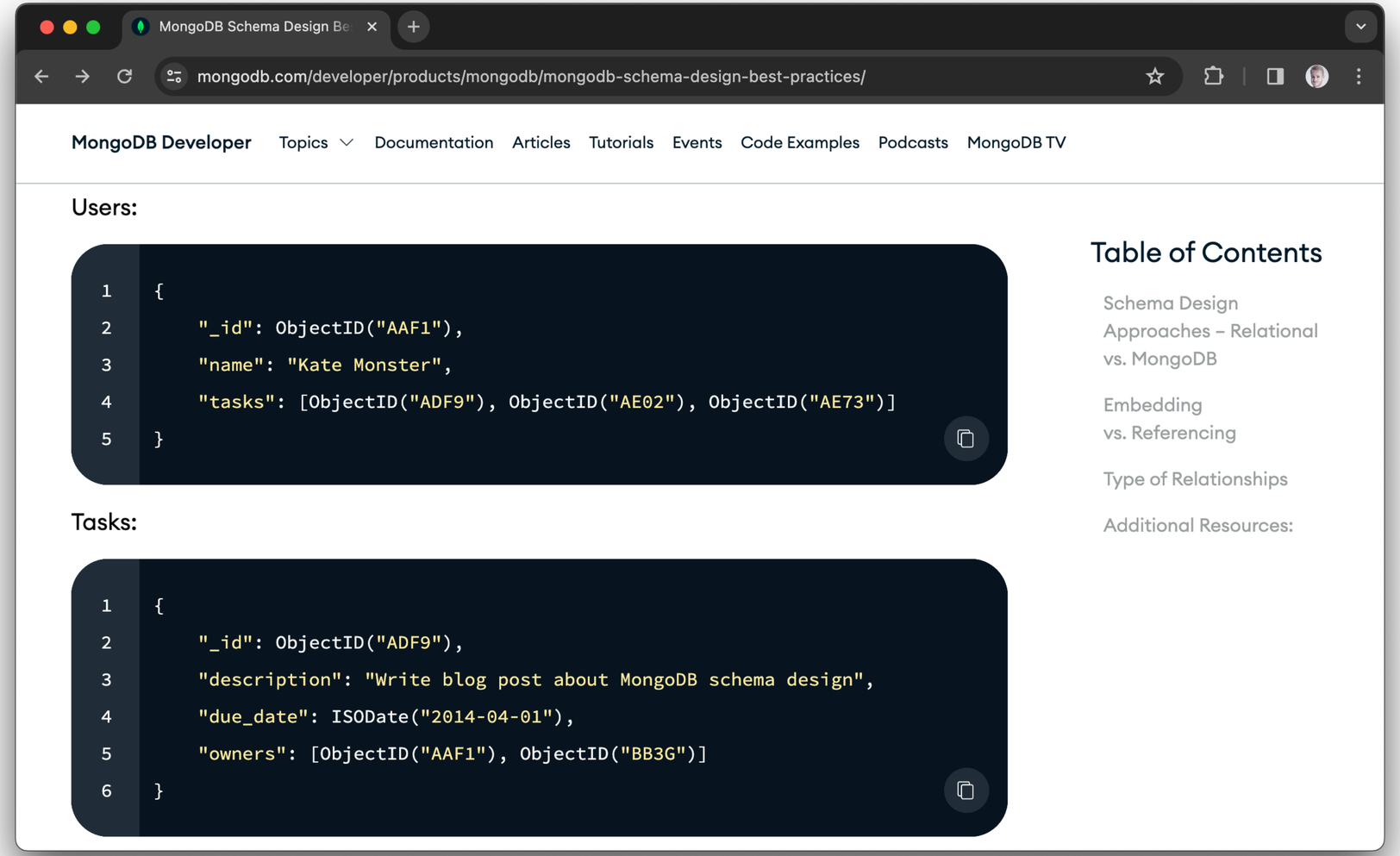
A Many-to-Many relationship between two entities where they both might have many relationships between each other.



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?


MongoDB Schema Design Be
mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/
MongoDB Developer
Topics
Documentation
Articles
Tutorials
Events
Code Examples
Podcasts
MongoDB TV
Users:
{
"_id": ObjectID("AAF1"),
"name": "Kate Monster",
"tasks": [ObjectID("ADF9"), ObjectID("AE02"), ObjectID("AE73")]
}
Tasks:
{
"_id": ObjectID("ADF9"),
"description": "Write blog post about MongoDB schema design",
"due_date": ISODate("2014-04-01"),
"owners": [ObjectID("AAF1"), ObjectID("BB3G")]
}
Table of Contents
Schema Design Approaches – Relational vs. MongoDB
Embedding vs. Referencing
Type of Relationships
Additional Resources:

Embed or Link

- 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

- 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

General Rules for MongoDB Schema Design

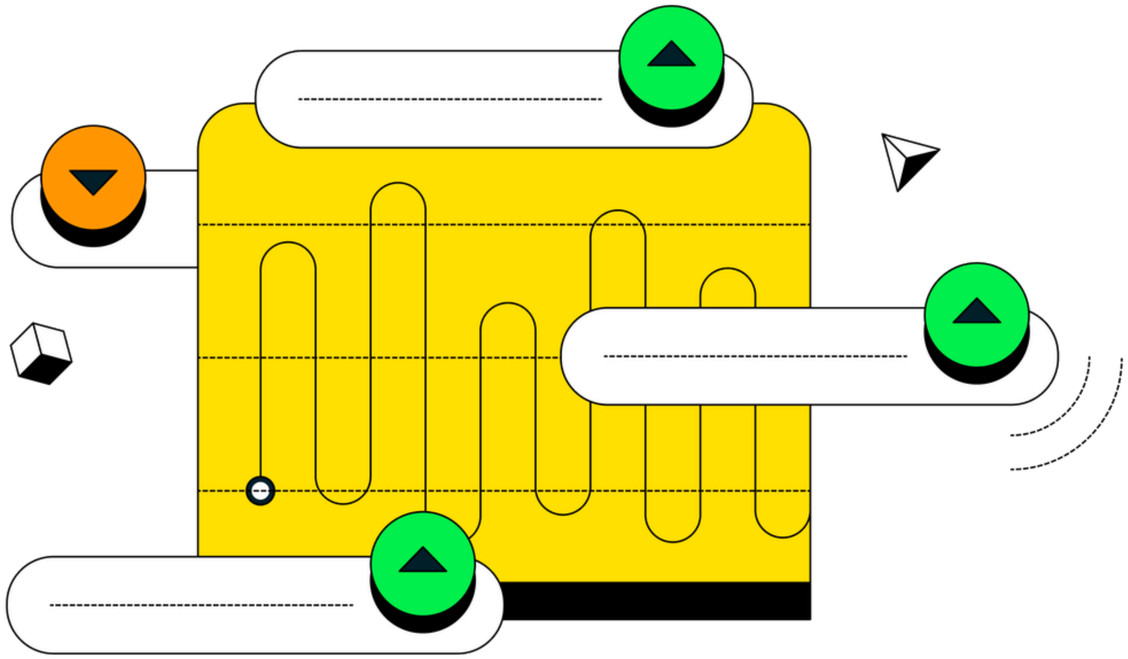
- Rule 1: Favor embedding unless there is a compelling reason not to.
- Rule 2: Needing to access an object on its own is a compelling reason not to embed it.
- Rule 3: Avoid joins and lookups if possible, but don't be afraid if they can provide a better schema design.
- Rule 4: Arrays should not grow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

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



Search Index

Index: example

- This code creates an index on the uid field in the posts collection.
- An index is a data structure that improves the performance of queries that filter or sort based on the indexed field. In this case, the index will improve the performance of queries that filter posts based on the user ID.
- The query `db.posts.find({uid: ObjectId("ZfPTVEMQKf9v")})` will be much faster if there is an index on the uid field, as MongoDB can directly identify the documents associated with the specified user ID. This is in contrast to a scenario where there is no index on the uid field, where MongoDB would have to scan the entire collection to locate the matching documents.

```
db.posts.insertMany(
[
    {
      caption: "A beautiful morning in Aarhus",
      createdAt: new Date("2023-04-06T09:10:54Z"),
      image: "https://images.unsplash.com/photo-1573997953524-ef
      uid: ObjectId("HlvRHr58C05g")
    },
    {
      caption: "Rainbow reflections of the city of Aarhus",
      createdAt: new Date("2023-04-02T20:25:34Z"),
      image: "https://images.unsplash.com/photo-1558443336-dbb3c
      uid: ObjectId("fjpRTRTjZHwr")
    }
  ]
)

 // create index
db.posts.createIndex({ uid: 1 });
// posts with specific user
db.posts.find({uid: ObjectId("ZfPTVEMQKf9v")})
```

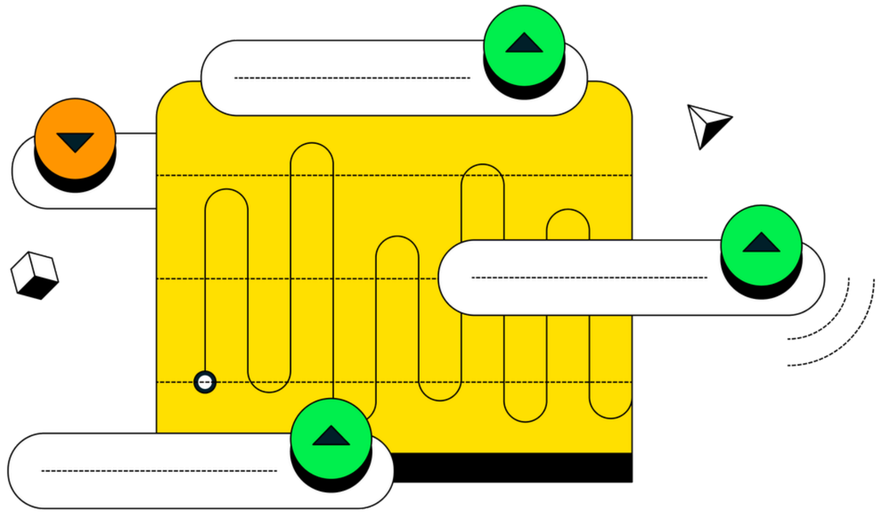
MQL Script: Posts and users

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