MongoDB Design

CS 341 Database Systems

What is **JSON**?

What is JSON?

JSON

- Stands for JavaScript Object Notation.

Purpose:

- A lightweight format for storing and transporting data.

Usage:

- Commonly used to send data from a server to a web page.

Language-Independent:

- Not tied to any specific programming language.

Self-Describing:

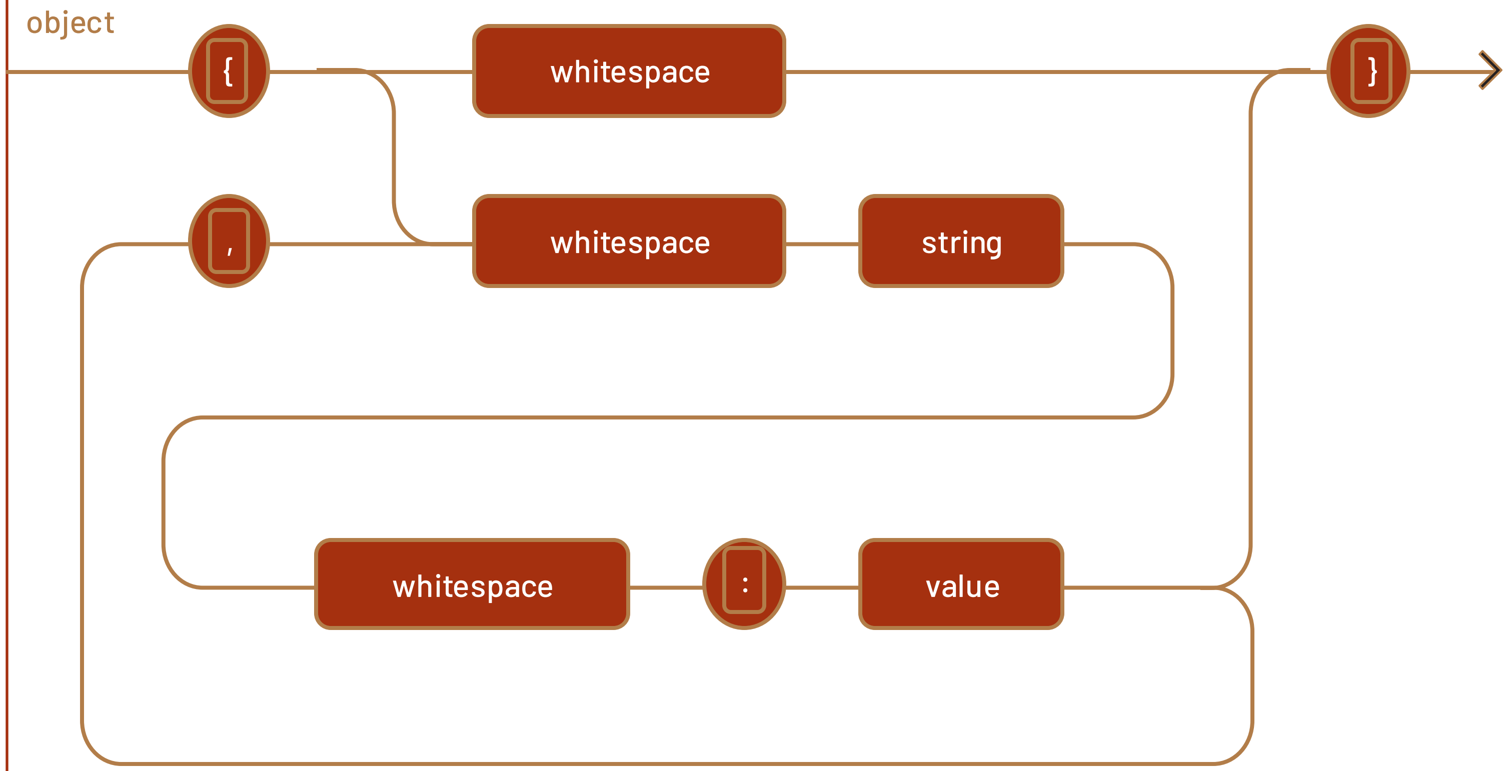
- Easy to read and understand.

JSON Object

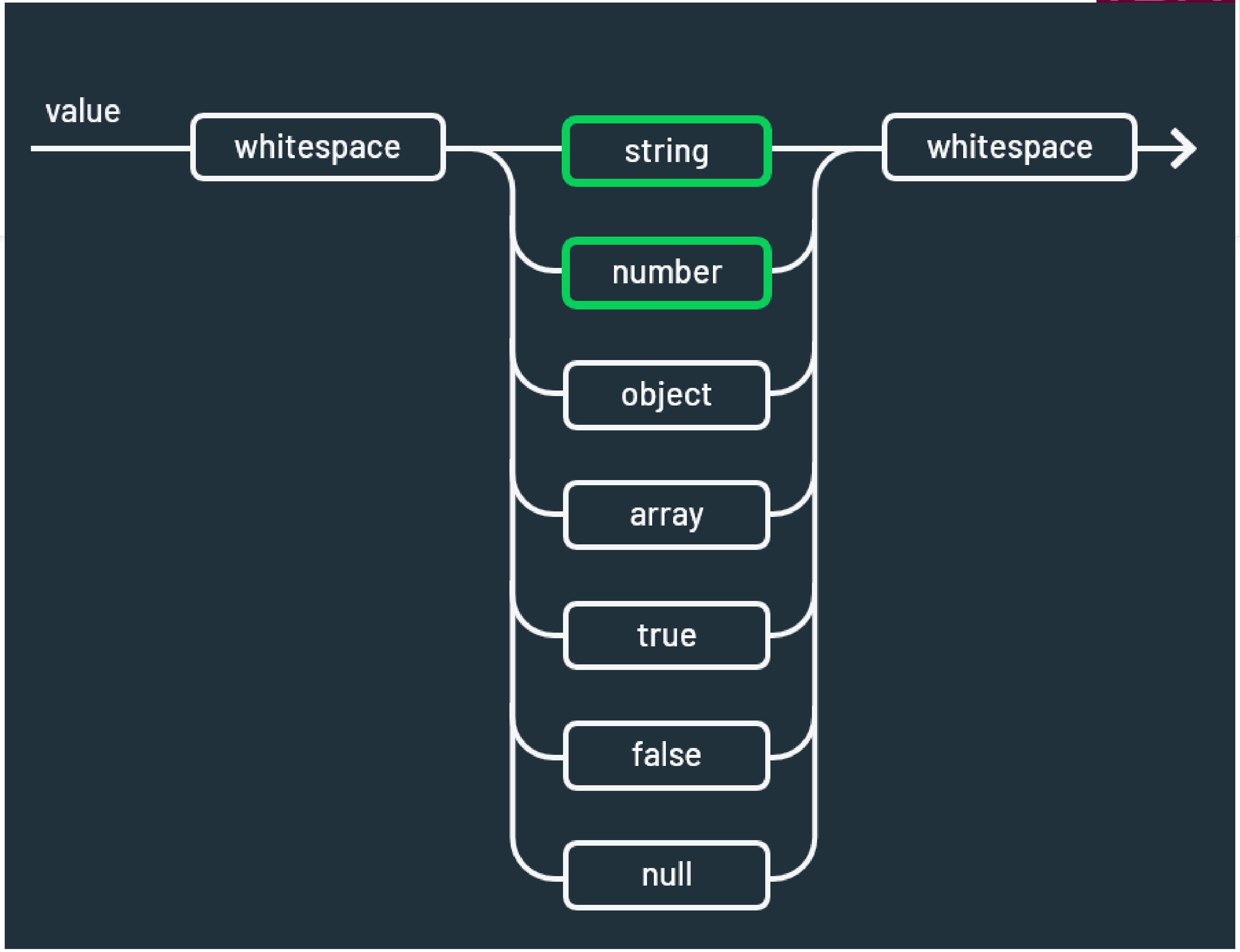
SYNTAX RULES:

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

```
{
   "_id": ObjectId(
        "5f4f7fef2d4b45b7f11b6d7a"),
   "user_id": “Alpha",
   "age": 21,
   "Status": "Active"
}
```



JSON Values


value
whitespace
string
number
object
array
true
false
null
whitespace

JSON Object

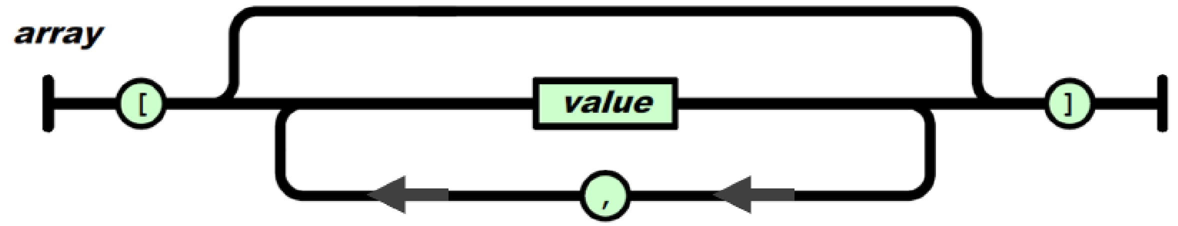
```
{
 "Name": "Alpha",
 "Age": 20,
 “Enrolled": true,
 "Contact": {
  "Phone": 123456789,
  "Address": null
 },
 “Courses": [ “DB", “OS", “SE"]
}
```

Field-Value pairs

Contact is a JSON Object

JSON Array

Array of Objects



```
{
  "name": "John Doe",
  "age": 30,
  "married": true,
  "siblings": [
    {"name": "John", "age": 25},
    true,
    "Hello World"
  ]
}
```

The array contains 3 items. The first item is an object, the second item is a boolean, and the third item is a string.

JSON Example

-
  ```
  {
  "employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
  ]
  }
  ```

Example of JSON - TESLA

SCALAR DOCUMENT

```
{
        "manufacturer" : "Tesla Motors",
        "class" : "full-size",
        "body style" : "5-door liftback"
}
```

ARRAY DOCUMENT

```
{
        "production" : [2012, 2013,2014],
        "model years" : [2013],
        "layout" : ["Rear-motor", "rear-wheel drive"]
}
```

Example – contd.

EMBEDDED DOCUMENT

```
{
        "designer" : {
                "firstname" : "Franz",
                "surname" : "von Holzhausen"
        }
}
```

Example – contd.

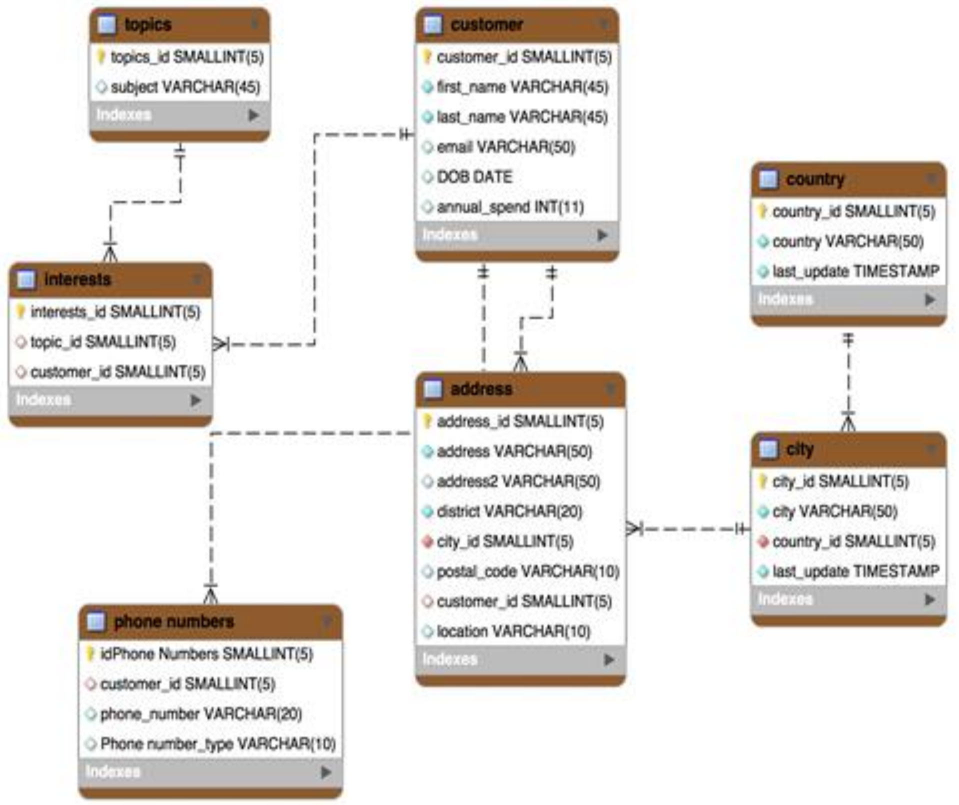
ARRAY WITH EMBEDDED DOCUMENT

```
{
        "assembly" : [
                {
                        "country" : "United States",
                        "city" : "Fremont",
                        "state" : "California"
                },
                {
                        "country" : "The Netherlands",
                        "city" : "Tilburg"
                }
        ]
}
```

MongoDB

- MongoDB is a document database
- Installed locally or hosted on cloud
- Local installation:
  - host your own MongoDB server on your hardware.
  - requires you to manage your server, upgrades, and any other maintenance.
- For the course, we will use MongoDB Atlas, a cloud-based platform, which is simpler to setup and use.

What is MongoDB (the database) ?



```
{
  "_id" : ObjectId("5ad88534e3632e1a35a58d00"),
  "name" : {
    "first" : "John", "last" : "Doe" },
  "address" : [
    { "location" : "work",
      "address" : { "street" : "16 Hatfields",
"city" : "London", "postal_code" : "SE1 8DJ"},
      "geo" : { "type" : "Point", "coord" : [
51.5065752,-0.109081]}},
+    {...}
  ],
  "dob" : ISODate("1977-04-01T05:00:00Z"),
  "retirement_fund" : NumberDecimal("1292815.75")
}
```

Tabular (Relational) Data Model

Related data split across multiple records and tables

Document Data Model

Related data contained in a single, rich document

MongoDB Document

- A way to organize and store data as a set of **field-value pairs.**
- Similar to Dictionaries in Python, Maps in Java, JSON Object in JavaScript
- BSON, or **Binary JSON**, is the data format that MongoDB uses to organize and store data.
- This data format includes all JSON data structure types and adds support for types including dates, different size integers, Object_Ids, and binary data.

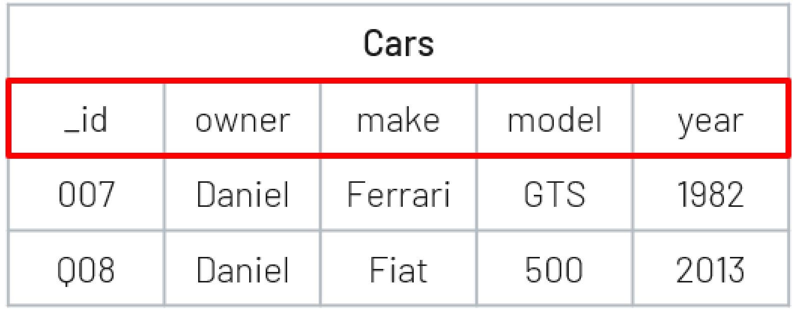
Document model constructs

- Fields (Attributes)
- Sub-documents (Objects)
- Arrays

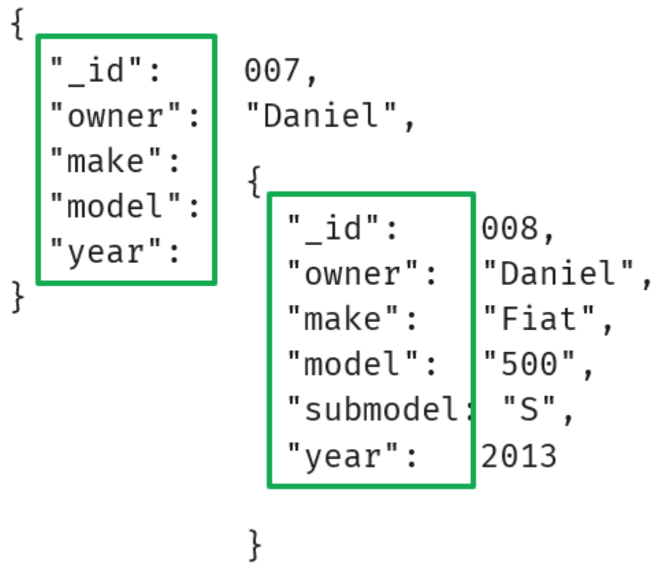
Syntax Rules:

- Curly brackets **{}** : marks the start and end of the document
- Field value pairs are separated by **:**
- Field names are in quotation marks. **" "**
- Field value pairs are separated by commas **(,)**

Fields / attributes



Cars				
_id	owner	make	model	year
007	Daniel	Ferrari	GTS	1982
008	Daniel	Fiat	500	2013



```
{
  "_id":      007,
  "owner":    "Daniel",
  "make":     {
  "model":       "_id":       008,
  "year":        "owner":     "Daniel",
}                "make":      "Fiat",
                 "model":     "500",
                 "submodel:   "S",
                 "year":      2013

               }
```

Tabular (Relational) Data Model

Header with column names
Row with values

Document Data Model

Each document explicitly list the names of the fields and the values.

Object/sub-document: a one-to-one relationship

Cars		
_id	owner	make
007	Daniel	Ferrari
008	Daniel	Fiat

Engines			
_id	car_id	power	consumption
234808	007	660	10
008	008	120	45

OR

Cars				
_id	owner	make	power	consumption
007	Daniel	Ferrari	660	10
008	Daniel	Fiat	120	45

```
{
  "_id": 007,
  "owner": "Daniel",
  "make": "Ferrari",
  "engine": {
              "power":
660hp,
              "consumption"
: 10mpg
  }
  …
}
```

Tabular (Relational) Data Model

A car has one Engine. A one-to-one relationship in a single table or across 2 tables

Document Data Model

The engine information is in its own structure in the parent entity

Array: a one-to-many

Cars		
_id	owner	make
007	Daniel	Ferrari
Q08	Daniel	Fiat

Wheels	
_id	car_id
234819	007
281928	007
392838	007
928038	007
950555	008

Tabular (Relational) Data Model

One-to-Many relationship
from a car to the its wheels

```
{
  "_id": 007,
  "owner": "Daniel",
  "make": "Ferrari",
  wheels: [
             { "partNo": 234819
},
             { "partNo": 281928
},
             { "partNo": 392838
},
             { "partNo": 928038
}
  ],
  ...
}
```

Document Data Model

One-to-Many wheels
expressed as an array

MongoDB Collections

- An organized store of documents in MongoDB, usually with common fields between documents
- MongoDB collections do not by default enforce a single schema on a collection i.e *whilst documents can have common fields, they are not required to have the same fields.*

```
{
  "_id": 11,
  "user_id": "Erin",
  "age": 29,
  "Status": "A"
}
```

```
{
  "_id": 12,
  "user_id": "Daniel",
  "age": 25,
  "Status": "A",
  "Country": "USA"
}
```

Schema Validation

Schema / Document Validation

- Document validation allows restriction to be made when new content is added, it allows for the presence, the type, and the values to be validated as part of this process.
- The **$jsonSchema** operator allows us to define our document structure.

JSON Schema

```
{
   "$schema": http://json-schema.org/draft-04/schema",
   "type": "object",
   "properties": {
      "Book": {
         "type": "object",
         "properties": {
            "Title": {"type": "string"},
            "Authors": {"type": "array", "minItems": 1, "maxItems": 5, "items": { "type": "string" }},
            "Date": {"type": "string", "pattern": "^[0-9]{4}$"},
            "Publisher": {"type": "string", "enum": ["Springer", "MIT Press", "Harvard Press"]}
         },
         "required": ["Title", "Authors", "Date"],
         "additionalProperties": false
      }
   },
   "required": ["Book"],
   "additionalProperties": false
}
```

Validation Rules

- **“type”** : "boolean“ or "number“ or "integer“ or "string“ or "array“ or "object“ or "null“
- **For integer or number:**
  - “minimum” , “maximum”
  - “enum” – allowed items
  - “exclusiveMinimum” , “exclusiveMaximum”– set as true or false if values are to be exclusive.

Validation Rules

- **For string:**
  - “maxLength”, “minLength”
  - “pattern” – compare through regular expression
  - “enum” - allowed items placed in an array
  - “format”
- **For array:**
  - “maxItems”, “minItems”
  - “Items” – value is a JSON schema → sub-schema

Required

- **Required:** Specifies that certain properties must be present in the object for it to be valid.
- "required": ["Title", "Authors", "Date"]
- The object must include all three properties: "Title", "Authors", and "Date".
- If any of these properties are missing, the object will fail validation.

additionalProperties ~ Flexible Schema

- Controls whether the object can have properties that are not explicitly defined in its "properties" section.
- "additionalProperties": false
- The object cannot include any extra properties other than the ones explicitly defined in the schema.
- If an unexpected property is present, validation will fail.

```json
{
  "Title": "Learning JSON Schema",
  "Authors": ["John Doe"],
  "Date": "2023"
}
```

Valid

Missing fields

```json
{
  "Title": "Learning JSON Schema",
  "Date": "2023"
}
```

Extra Fields

```json
{
  "Title": "Learning JSON Schema",
  "Authors": ["John Doe"],
  "Date": "2023",
  "ISBN": "12345"
}
```

additionalProperties: true

- The object can include any other properties beyond those explicitly defined in the schema.
- Use when you want to allow users or systems to add properties freely without strict schema enforcement. - Dynamic or Flexible Objects
- If "additionalProperties" is omitted altogether, the default behavior is the same as "additionalProperties": true.

JSON Schema

```
{
   "$schema": http://json-schema.org/draft-04/schema",
   "type": "object",
   "properties": {
      "Book": {
         "type": "object",
         "properties": {
            "Title": {"type": "string"},
            "Authors": {"type": "array", "minItems": 1, "maxItems": 5, "items": { "type": "string" }},
            "Date": {"type": "string", "pattern": "^[0-9]{4}$"},
            "Publisher": {"type": "string", "enum": ["Springer", "MIT Press", "Harvard Press"]}
         },
         "required": ["Title", "Authors", "Date"],
         "additionalProperties": false
      }
   },
   "required": ["Book"],
   "additionalProperties": false
}
```

Which one is a valid instance?

A or B?

```
{
    "Book": {
        "Title": "Understanding JSON Schema",
        "Authors": ["John Doe", "Jane Smith"],
        "Date": "2023",
        "Publisher": "MIT Press"
    }
}
```

```
{
    "Book": {
        "Title": "Understanding JSON Schema",
        "Authors": [],
        "Date": "2023",
        "Publisher": "Unknown Publisher"
    }
}
```

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "string",
  "allOf": [
          {
            "minLength": 2,
            "oneOf": [
                  {"pattern": "^[0-9]*$"},
                  {"pattern": "^[a-zA-Z]*$"}
                ]
          },
          {
            "maxLength": 5,
            "oneOf": [
                  {"pattern": "^[0-9]*$"},
                  {"pattern": "^[a-zA-Z]*$"}
                ]
          }
  ]
}
```

What is a valid instance of a string in this case?

allOf: Combines multiple schemas into a single validation rule. All schemas listed in allOf must be valid for the JSON instance to pass validation.

oneOf: Combines multiple schemas, but only one schema must be valid for the JSON instance to pass validation.

"$ref": "#/definitions/section" is used to reference another part of the schema.
#: root of schema
Definitions/section : path

```
{
  "book": {
   "title": "Title 1",
   "section": {
    "title": "Title 1.1",
    "section": {
     "title": "Title 1.1.1",
     "section": {
      "title": "Title 1.1.1.1"
     }
    }
   }
  }
}
```

```
{
 "$schema": "http://json-schema.org/draft-04/schema#",
 "definitions": {
  "section": {
   "type": "object",
   "properties": {
    "title": {
     "type": "string"
    },
    "section": {
     "$ref": "#/definitions/section"
    }
   },
   "required": [
    "title"
   ],
   "additionalProperties": false
  }
 },
 "type": "object",
 "properties": {
  "book": {
   "$ref": "#/definitions/section"
  }
 }
}
```

MongoDB Collections

JSON Validation Rules apply

As discussed previously

- **“type”** : "boolean“ or "number“ or "integer“ or "string“ or "array“ or "object“ or "null“
- **For integer or number:**
  - “minimum” , “maximum”
  - “enum” – allowed items placed in an array
  - “exclusiveMinimum” , “exclusiveMaximum”– set as true or false if values are to be exclusive.
- **For string:**
  - “maxLength”, “minLength”
  - “pattern” – compare through regular expression
  - “enum”
  - “format”
- **For array:**
  - “maxItems”, “minItems”
  - “Items” – value is a JSON schema → sub-schema (starts from **bsonType: "object“, properties: { …**

```
db.createCollection("posts", {
 validator: {
  $jsonSchema: {
   bsonType: "object",
   required: [ "title", "body" ],
   properties: {
    title: {
     bsonType: "string",
     description: "Title of post - Required."
    },
    body: {
     bsonType: "string",
     description: "Body of post - Required."
    },
    category: {
     bsonType: "string",
     description: "Category of post - Optional."
    },
    likes: {
     bsonType: "int",
     description: "Post like count. Must be an integer - Optional."
    },
    tags: {
     bsonType: ["string"],
     description: "Must be an array of strings - Optional."
    },
    date: {
     bsonType: "date",
     description: "Must be a date - Optional."
    }
   }
  }
 }
})
```

INSTANCE

```
{
  "title": "My First Post",
  "body": "This is the body of the post.",
  "category": "Technology",
  "likes": 10,
  "tags": ["tech", "mongodb"],
  "date": ISODate("2024-11-24T00:00:00Z")
}
```

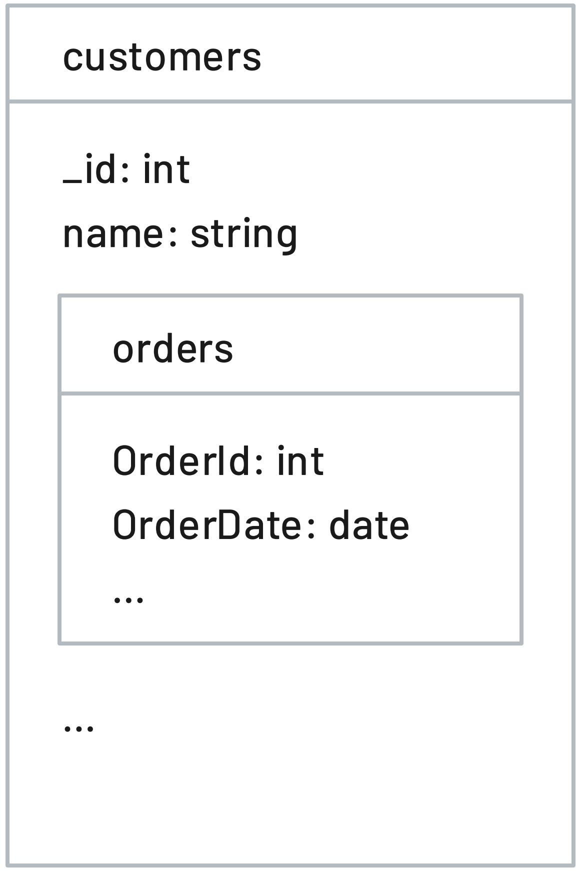
```
db.createCollection("students", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      title: "Student Object Validation",
      required: [ "name", "year" ],
      properties: {
        name: {
          bsonType: "string",
          description: "'name' must be a string and is required"
        },
        year: {
          bsonType: "int",
          minimum: 2017,
          maximum: 3017,
          description: "'year' must be an integer in [ 2017, 3017 ] and is required"
        },
        gpa: {
          bsonType: "double" ,
          description: "'gpa' must be a double if the field exists"
        }
      }
    }
  }
} )
```

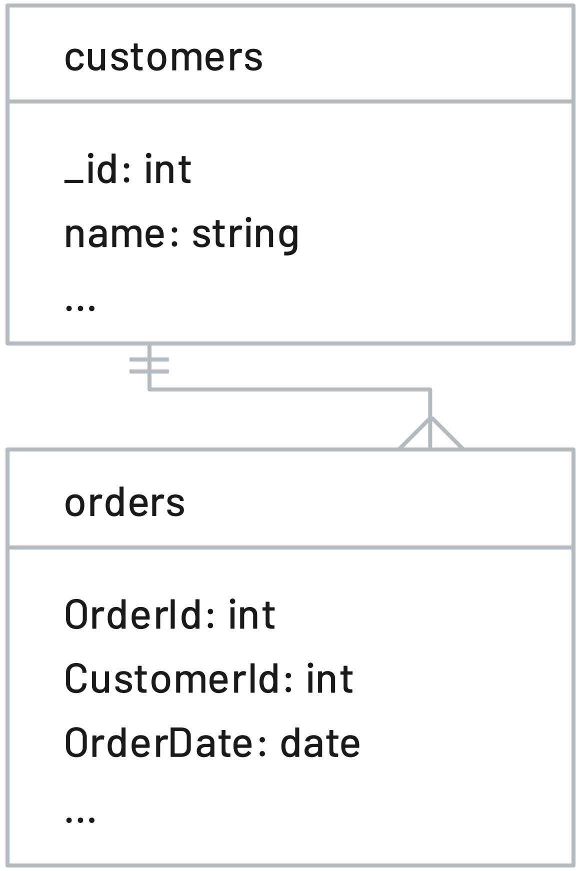
Schema Design - Link or Embed

- Do I want the embedded information mostly?
- Do I need to search with the embedded data?
- How frequently will the embedded data change?
- Do I need the latest version or the same version?
- Is the embedded data shared or private – double check?

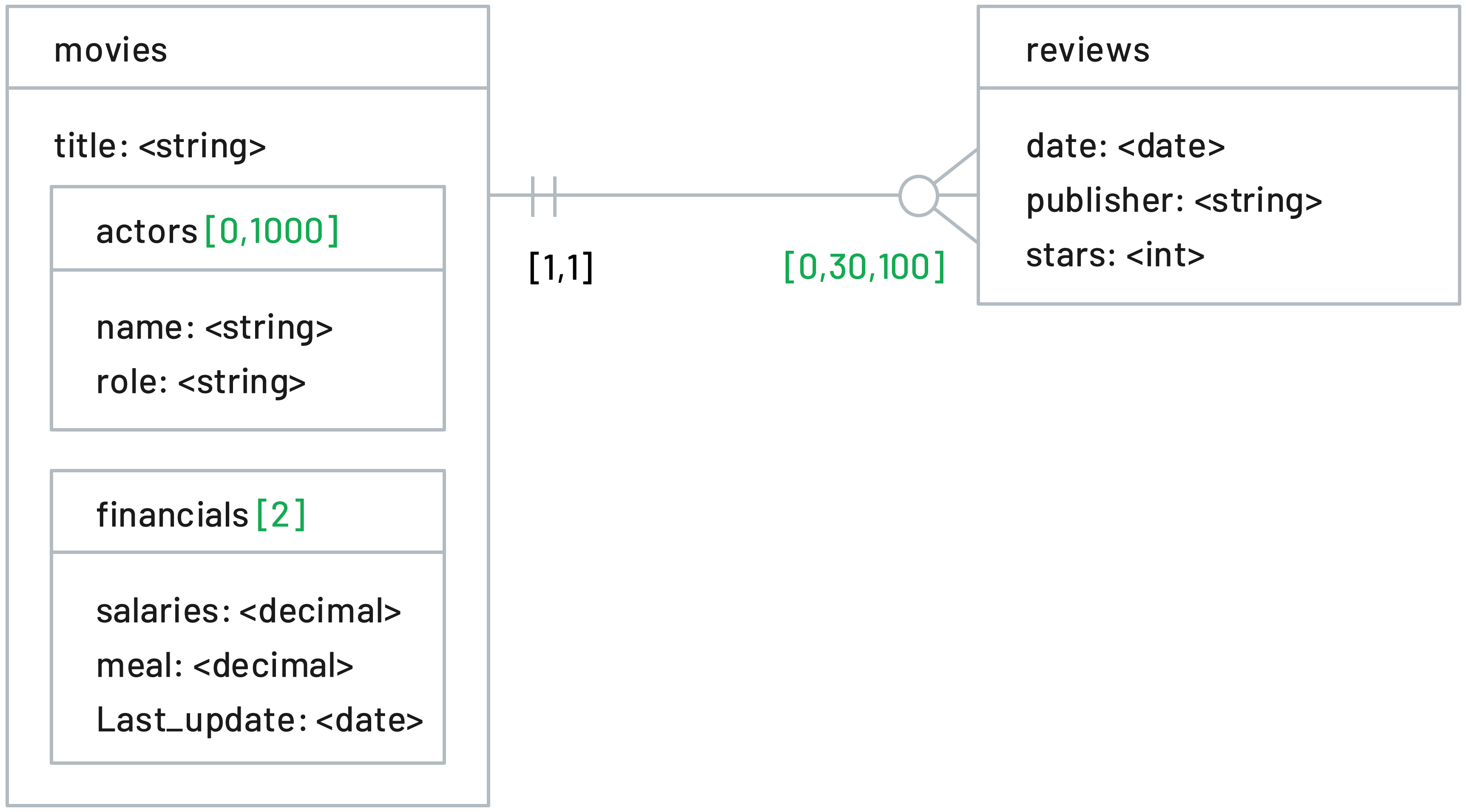
Embedding

Referencing / linking





Embedding and linking



Relationships

One-to-One (1-1)

One-to-Many (1-N)

Many-to-Many (N-N)

Relationships

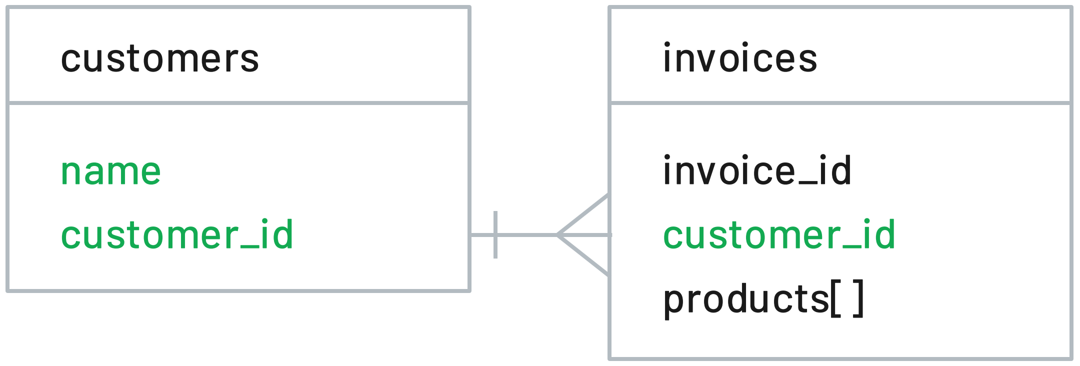
One-to-One (1-1)

One-to-Many (1-N)

Many-to-Many (N-N)

customers
name
customer_id

Embedding is the preferred way to model a 1:1 relationship as it's more efficient to retrieve the document.



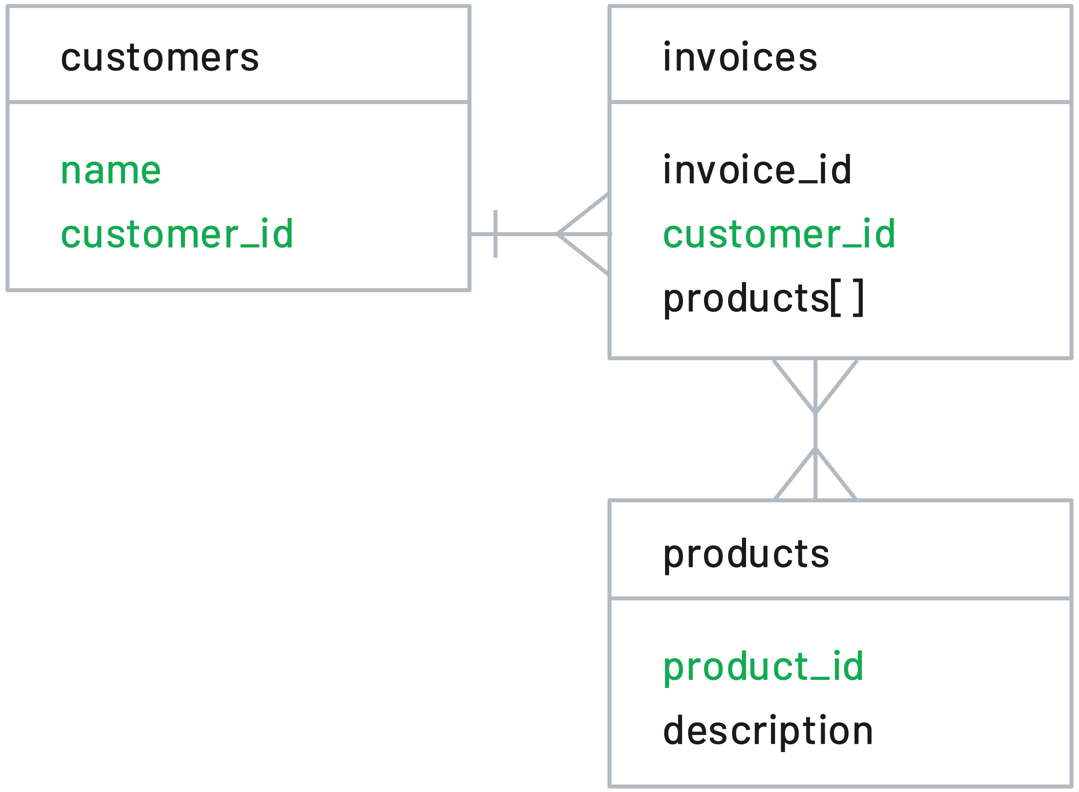
In this type of relationship, the customer can have many invoices. It can be modelled by either linking the data (as shown) or by embedding the data (where the invoices are within the customer documents).

Relationships

One-to-One (1-1)

One-to-Many (1-N)

Many-to-Many (N-N)



Relationships

One-to-One (1-1)

One-to-Many (1-N)

Many-to-Many (N-N)

Many-to-Many Relationship

- Documents on the first side can be associated with many documents on the second side and similarly for the other side.
- **1-way embedding strategy** optimizes the read performance of a N:N relationship by embedding the references in **one** side of the relationship.
- The key step in this strategy is establishing the ***relationship balance*** and *choosing the side which has fewer number of entities*.
- If the relationships are close to an even ratio, then **2-way embedding** is probably a better strategy.

Embed

For integrity for read operations

For integrity for write operations on one-to-one and one-to-many

For data that is deleted together

By default

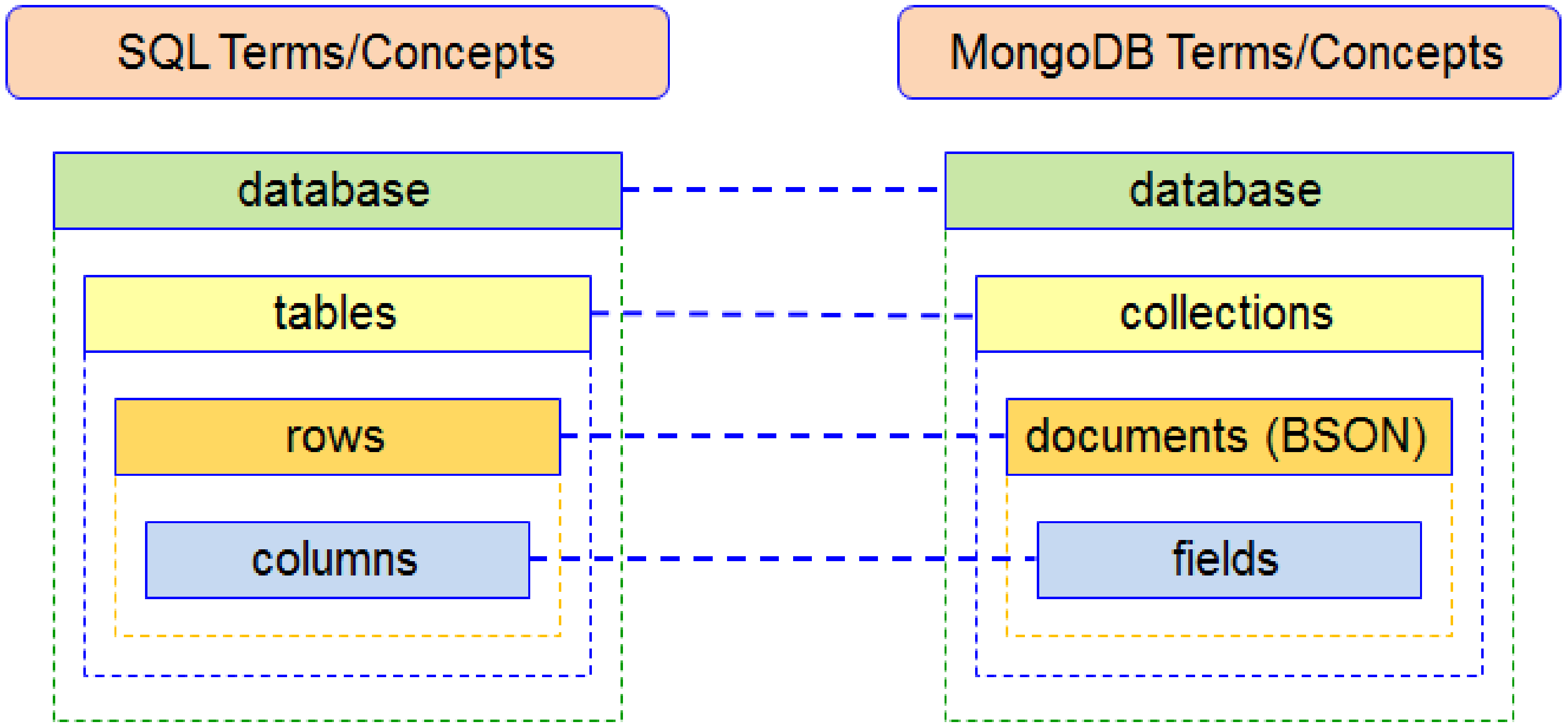
Link

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

Basic Comparison



Schema Design Best Practices

- Usually, flexible
- Major Considerations
  - Storage of data
  - Good query performance
  - Reasonable amount of hardware

Relations

- **One-to-One** - Prefer key value pairs within the document
  - Department and Manager
- **One-to-Few** - Prefer embedding
  - Customer and Addresses
- **One-to-Many** - Prefer embedding
  - Department and Staff
- **Many-to-Many** - Referencing or 1-way embedding or 2-way embedding
  - Courses and Students

General Rules

- **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. (referencing)

General Rules

- **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.*

Consider a Scenario

- Each SalesRep is assigned to one department. Each department has many sales representatives
- Each department is headed by a manager who is also a SalesRep
- Each customer can be dealt by multiple SalesRep and different SalesRep can deal with different customers

Consider a Scenario

- Each **SalesRep** is assigned to one **department**. Each department has many sales representatives
- Each department is headed by a manager who is also a SalesRep
- Each **customer** can be dealt by multiple SalesRep and different SalesRep can deal with different customers

Department – SalesRep
1-M (Assigned)

Department – SalesRep
1-1 (Manager)

Customer – SalesRep
M-M (Deals)

Collections

- **Department**
  - SalesRep as key:value pair or embedded document (Manager)
- **SalesRep**
  - Department as embedded document
  - Embedded array of customers dealt by this salesRep
- **Customer**
  - Embedded array of SalesRep dealing this customer