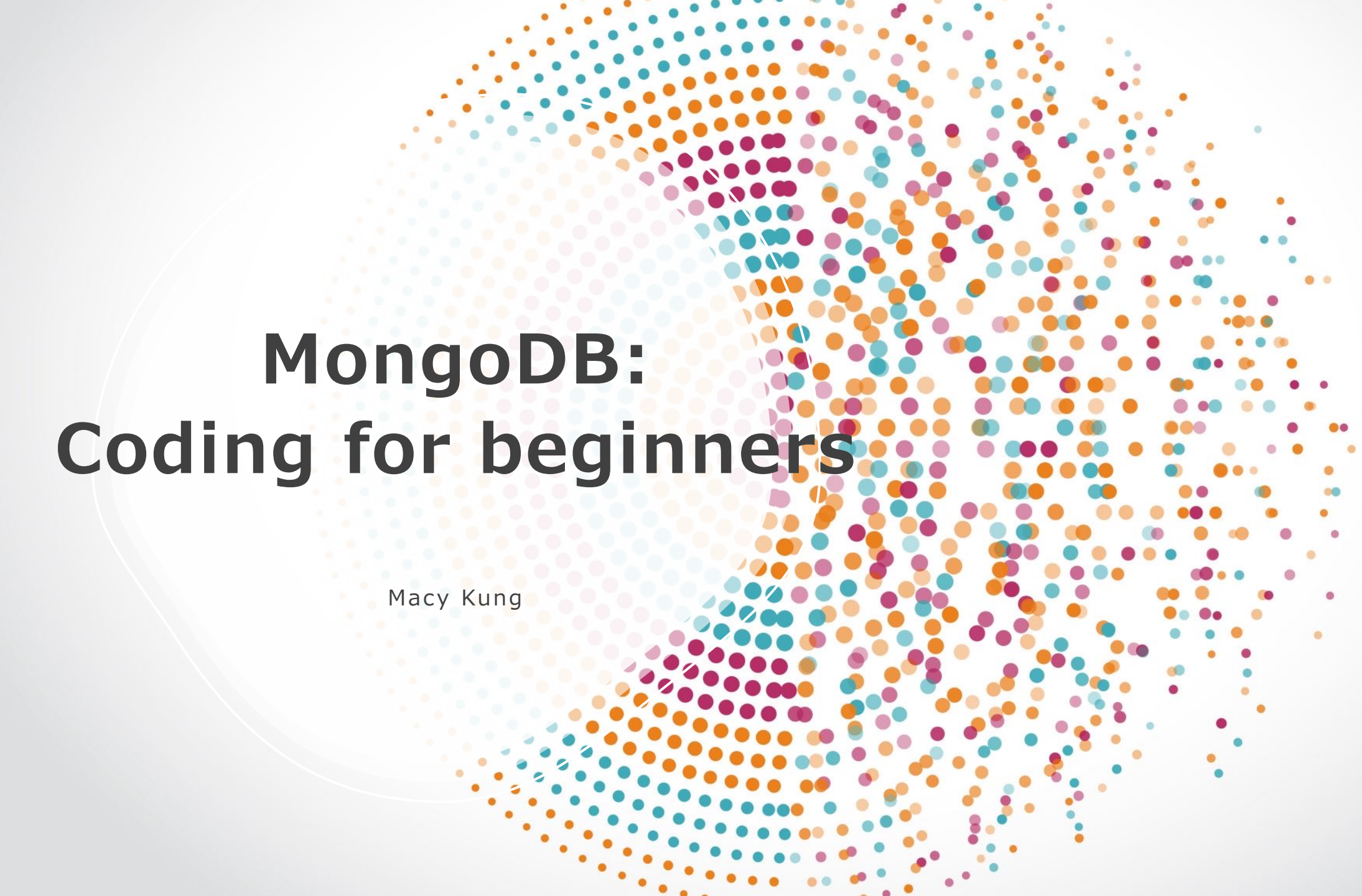


MongoDB: Coding for beginners

Macy Kung

Welcome to the Course

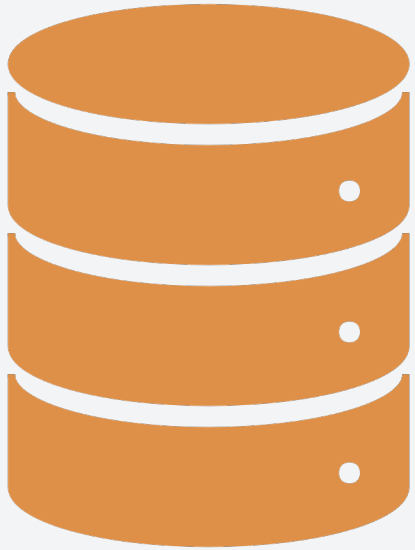
- Introduction to NoSQL and MongoDB
- Knowledge Notes
- Set Up & Installation
- Inserting Documents
- Mongo Query Language
- Deleting Documents
- Updating Documents
- Data Modelling
- Index and Performance
- Aggregation

Introduction to NoSQL Database

What is a Database?

What is an SQL Database?

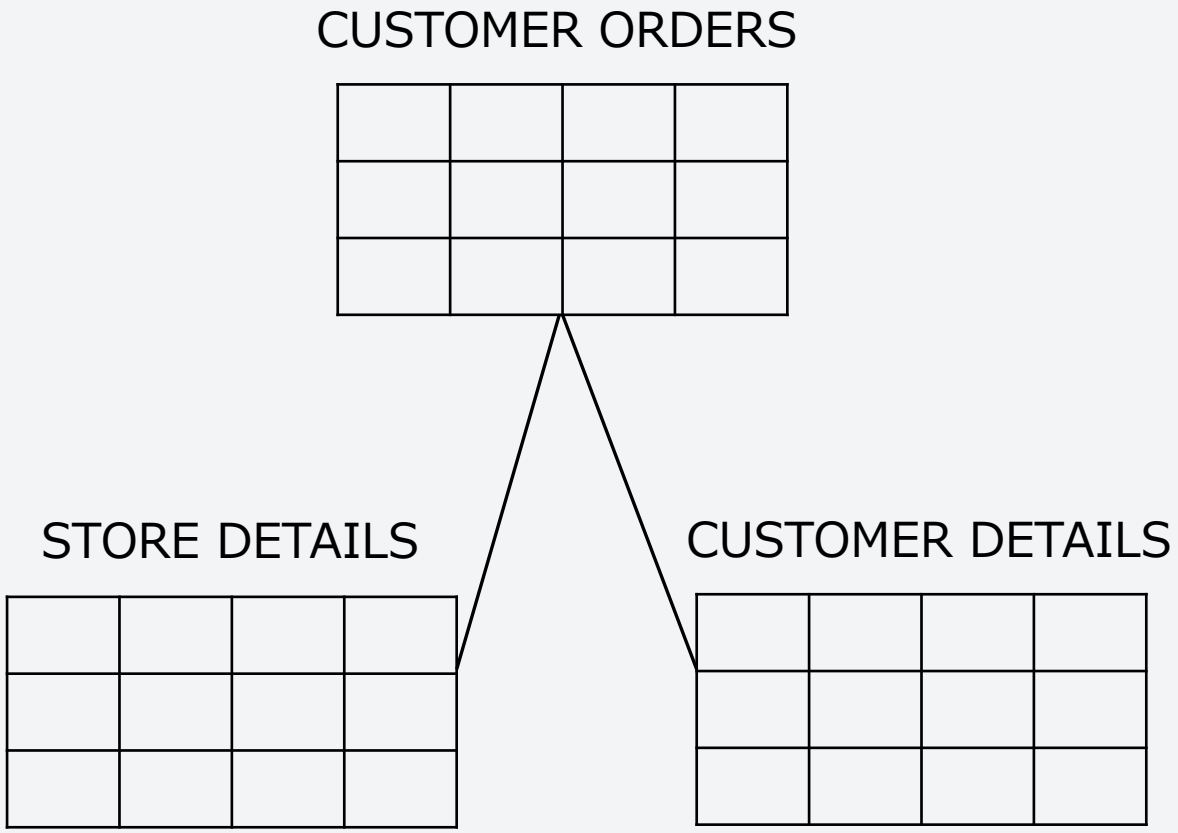
What is a NoSQL Database?



What is a Database

A database is defined as an organized collection of structured information or data typically stored electronically.

What is an SQL Database

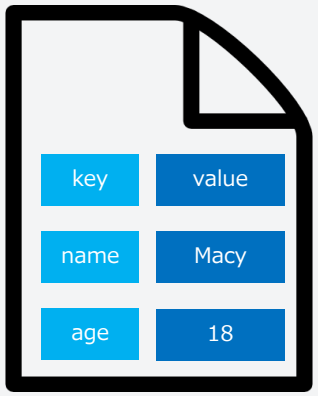


What is a NoSQL Database

NoSQL (also refers to Not only SQL, non-SQL or non-relational) is a database which gives you a way to manage the data which is in a non-relational form i.e. which is not structured in a tabular manner and does not possess tabular relationships.

Types of NoSQL Database

Document



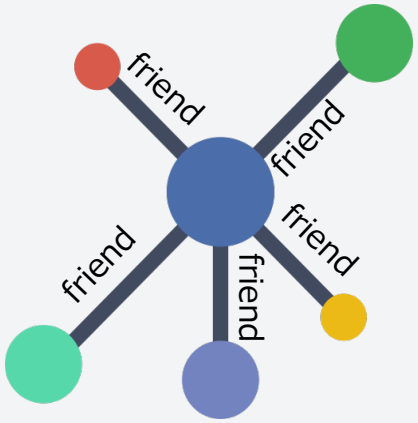
Apache, MongoDB, CouchDB

Key Value

key	value
Student_1	name@Laura @A4- tel@45789
Student_2	name@Chirs @B7- tel@12349

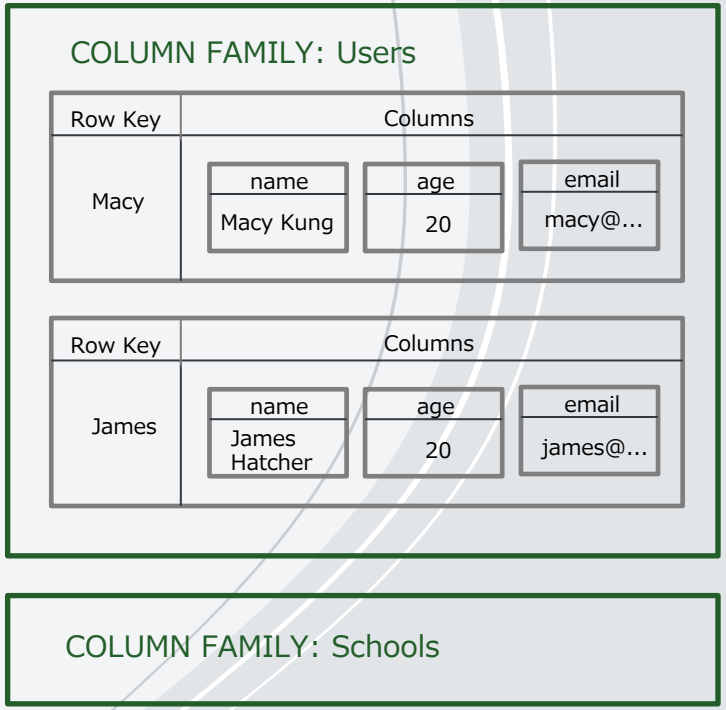
Redis, Dynamo, ZooKeeper

Graph



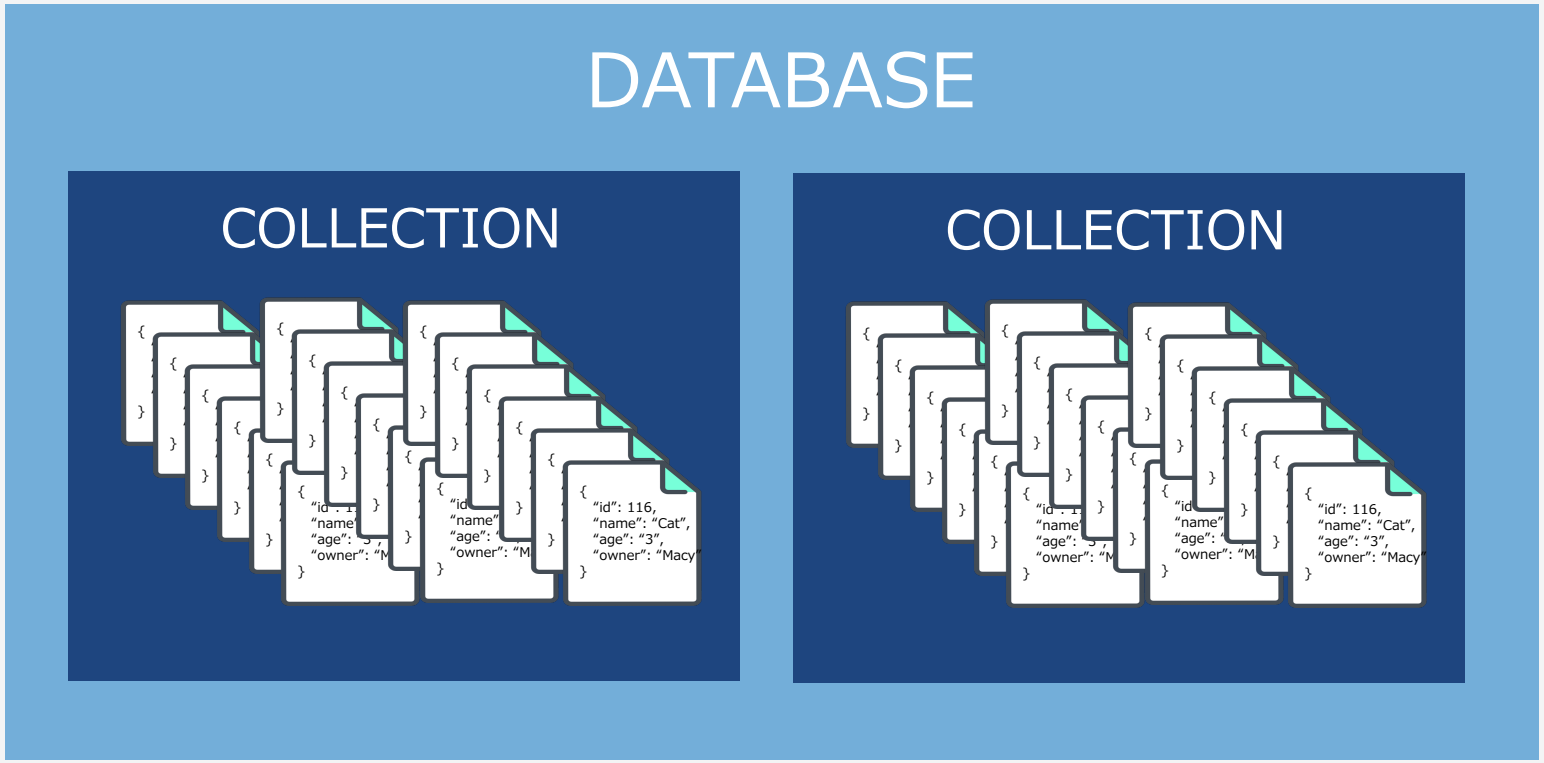
Neo4j, AllegroGraph, InfiniteGraph

Column Family



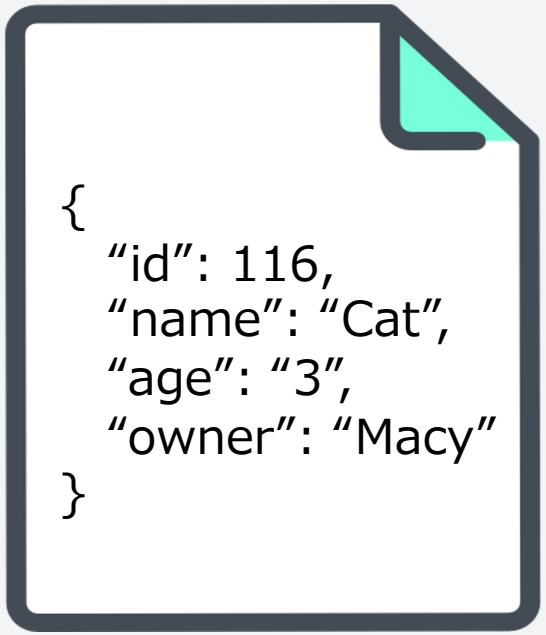
Cassandra, Hbase, Google BigTable

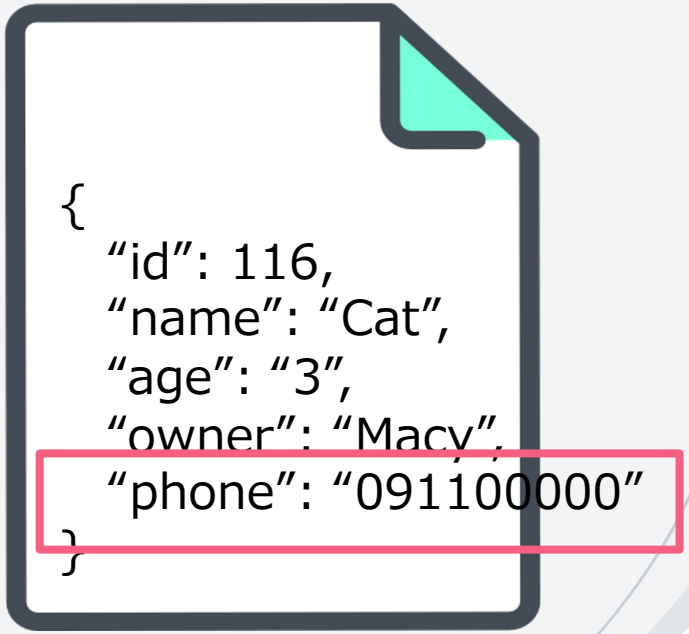
What is MongoDB




```
{
  “field”: “value”,
  “field”: “value”,
  “field”: “value”,
  “field”: “value”
}
```

What is MongoDB



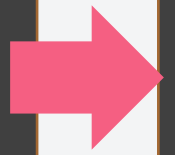


Documents are **polymorphic** i.e. they don't have a fixed structure. Changes can be made easily to an individual document structure e.g. new field value pairs can be added.

Previous example in SQL database

SQL

ID	NAME	AGE	OWNER
1	Cookie	3	Macy
2	Cin cin	7	Cindy
3	Dao	6	Lien
4	Pi	1	Yao



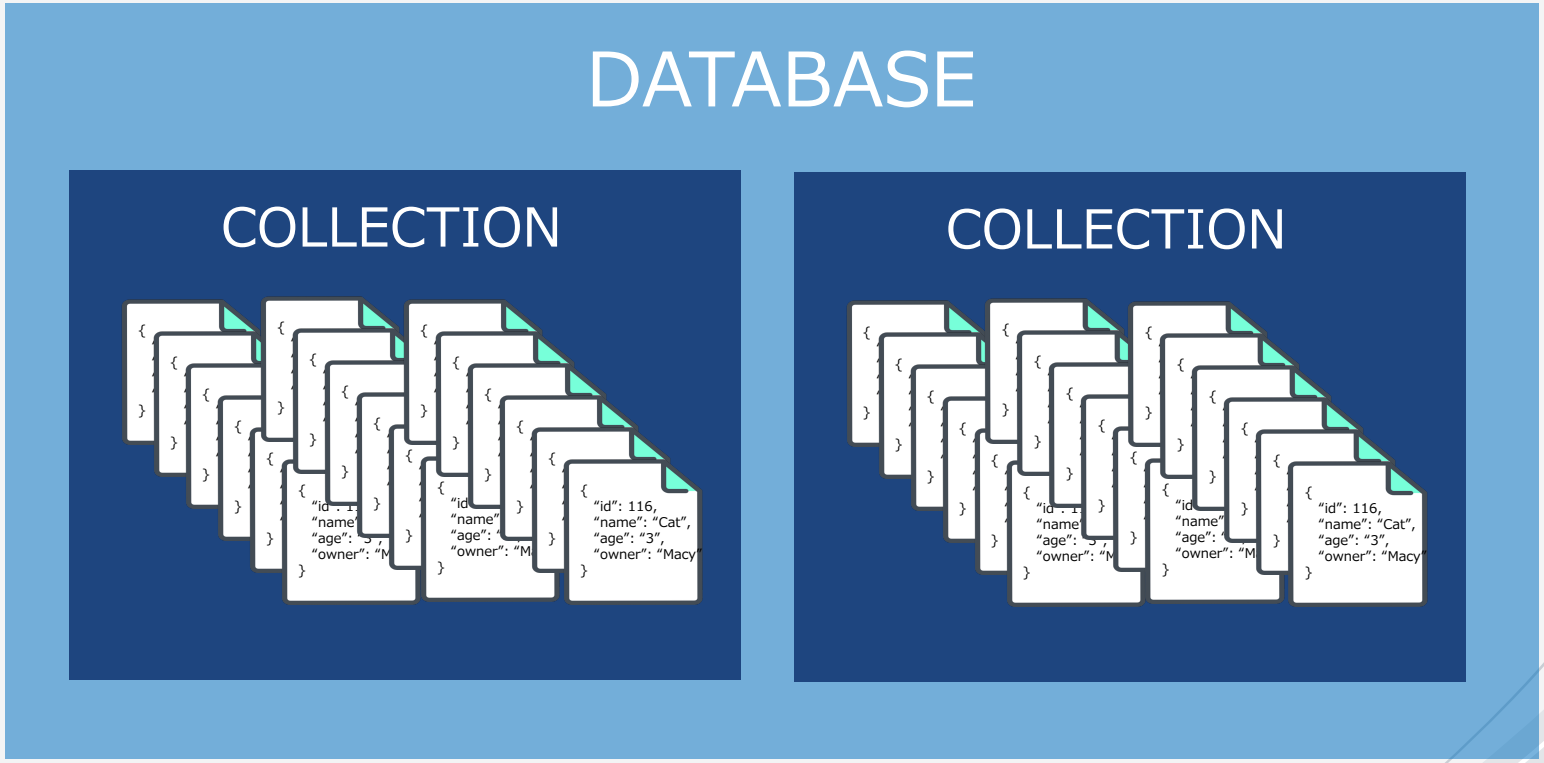
ID	NAME	AGE	OWNER	PHONE
1	Cookie	3	Macy	098100
2	Cin cin	7	Cindy	null
3	Dao	6	Lien	null
4	Pi	1	Yao	091888

What is MongoDB

In general, data is stored more efficiently in MongoDB than in SQL because it is schema-less resulting in less data redundancy and more efficiency

SQL vs NoSQL: comparison table

Documents in MongoDB



Documents in MongoDB

Documents in MongoDB

Documents in MongoDB

```
{
    "_id": ObjectId(11666),
    "name": "Cat",
    "phone": ["02-0034241", "091133442"]
    "address": {
        "street": "No.1 Neihu road",
        "city": "Taipei",
        "postal code": "114"
     }
}
```

An array is a data structure consisting of a collection of elements
With an array data type, we can store multiple values in a single key of the document.

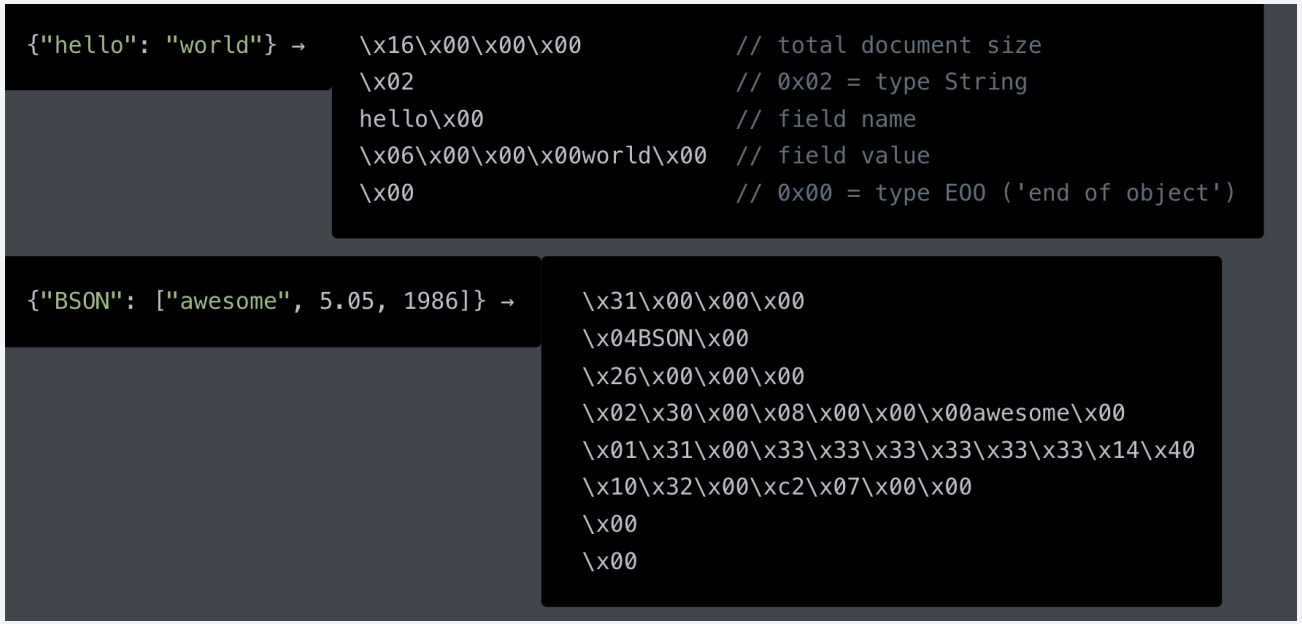
Documents in MongoDB

```
{
    "_id": ObjectId(11666),
    "name": "Cat",
    "age": "3",
    "owner": "Macy"
}
```

```
{
    "_id": ObjectId(11666),
    "name": "Cat",
    "age": "3",
    "owner": "Macy",
    "phone": "091100000"
}
```

```
{
    "_id": ObjectId(11666),
    "name": "Cat",
    "age": "3",
    "owner": "Macy",
    "cell phone": "091100000"
}
```

Not the same field



What is BSON?
Does MongoDB use BSON, or JSON?

MongoDB - Datatypes

NUMERICAL:
Numbers such as integers and decimals

DATE:
Date and time values

STRING:
Charater / text values

👉 Datatypes Reference

Replication

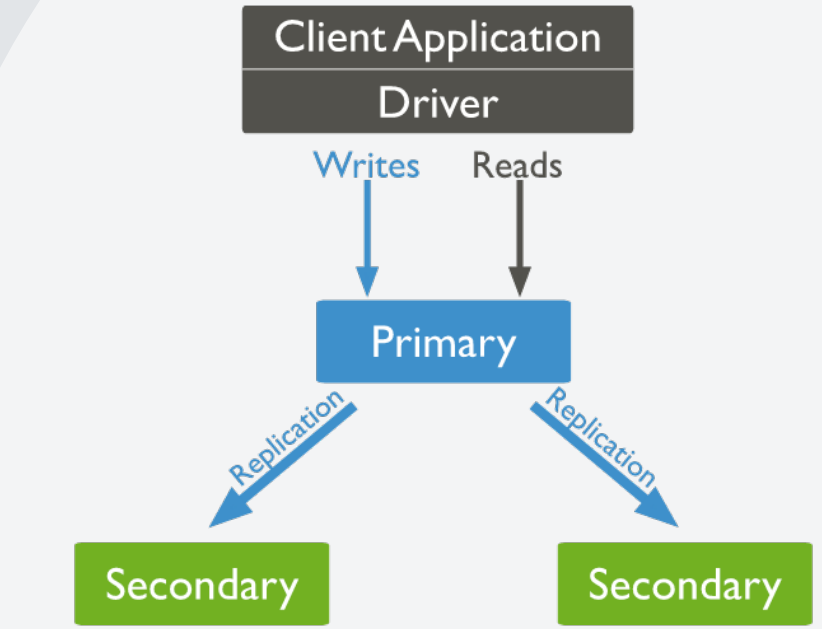
- A replica set in MongoDB is a group of **mongod** processes that maintain the same data set. Replica sets provide redundancy and high availability, and are the basis for all production deployments.

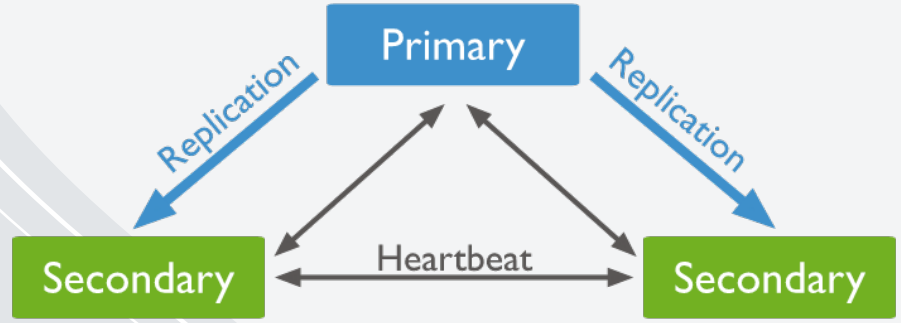
- There are 3 instance in a replica set and each instance contains a full copy of the data in each instance

Replication

- Of the data bearing nodes, one and only one member is deemed the primary node, while the other nodes are deemed secondary nodes. If the primary is unavailable, an eligible secondary will hold an election to elect itself the new primary.

- **mongod** is the primary daemon process for the MongoDB system. It handles data requests, manages data access, and performs background management operations.





Sharded Cluster

- Sharding is a method for distributing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations.

Sharded cluster consists of the following components

SHARD

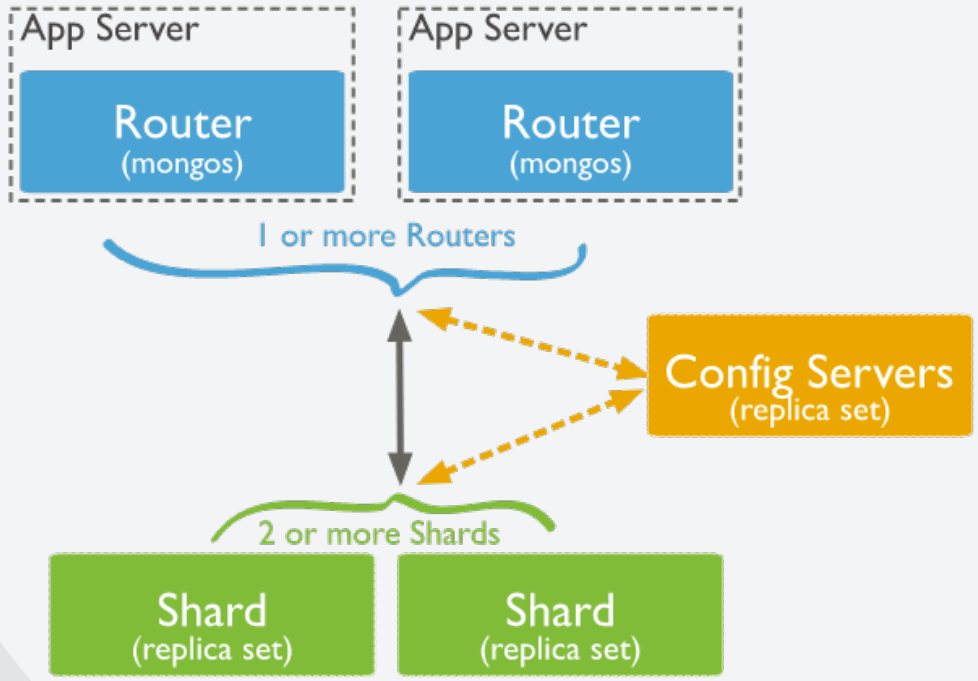
Containing a partition of the data

MONGOS

Interface between any application using MongoDB and the sharded cluster

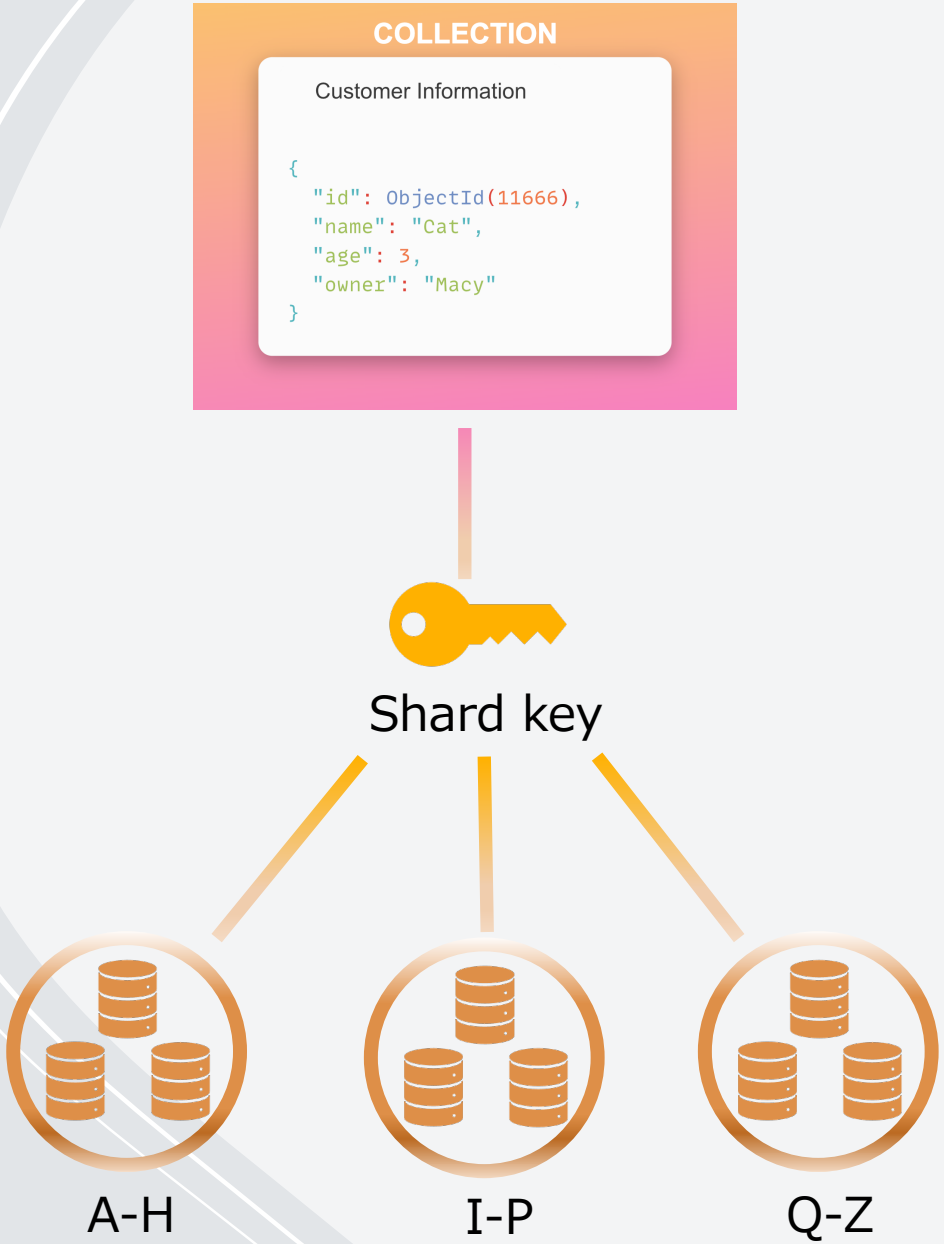
CONFIG SERERS

Store metadata and configuration settings



Natively Distributed System

- MongoDB uses the **shard key** to distribute the collection's documents across shards.
- The shard key consists of a field or multiple fields in the documents.



You get a basic knowledge of MongoDB

- What’s NoSQL database
- MongoDB with format of document
- A replica set typically consists of __ machine(s)?
- A typical Relational Database allows you more flexibility in data structure than MongoDB

Advatages of MongoDB Database

- Less complexity vs relational databases
- Schema-less
- Easier to maintain
- Easy to scale

When to use NoSQL

- To handle a huge volume of structured, semi-structured and unstructured data.
- If your relational database is not capable enough to scale up to your traffic at an acceptable cost.
- If you want to have an efficient, scale-out architecture in place of an expensive and monolithic architecture.
- If you have local data transactions that need not be very durable.
- If you are going with schema-less data and want to include new fields without any ceremony.
- When your priority is easy scalability and availability.

When to Avoid NoSQL

- In such cases like financial transactions, etc., you may go with SQL databases.
- If consistency is a must and if there aren’t going to be any large-scale changes in terms of the data volume, then going with the SQL database is a better option.
- One should also keep in mind that NoSQL databases won’t support structured query language. The querying language may vary from one database to another.

Set Up & Installation

- Click Try Free, SignUp to MongoDB Atlas with Google Account
- Download Mongo Compass
- MongoDB Atlas Quick Tour
- Create database
- Create Users
- Network access
- Connect to MongoDB using Mongo Compass
- Import data

Inserting Documents - insertOne

insertOne

```
db.collection.insertOne(
  <document or array of documents>
)
```

Inserts a single document into a collection.
Returns an object that contains
the status of the operation.

Collection: shoes

```
db.shoes.insertOne({
   brand: "Nike",
   item: "Fragment x Sacai x Nike LDWaffle",
   price: 15000,
   color: "gray",
   size: "UK5"
})
```

Return :

```
{
   "_id": ObjectId("537a41a"),
   "brand": "Nike",
   "item": "Fragment x Sacai x Nike LDWaffle",
   "price": 15000,
   "color": "gray”,
   "size": "UK5"
}
```

Inserting Documents - insertMany

insertMany

```
db.collection.insertMany(
  <document or array of documents>
)

Inserts multiple documents
 into a collection.
```

Collection: shoes

```
db.shoes.insertMany([
{ brand: "Nike",
  item: "Fragment X Sacai X Nike LDWaffle",
  price: 13000,
  color: "blue",
  size: "UK5"
},
{ brand: "Nike",
  item: "Fragment X Sacai X Nike LDWaffle",
  price: 15000,
  color: "blue",
  size: "UK8"
},
])
```

Using the find Methods

find

```
db.collection.find({query},{projection})

Returns all documents in a collection
Query and projection are optional
- Query : Specifies selection filter.
- Projection: Specifies the fields to return
  in the documents that match the query filter.
  1: true, query this field
  0: false, ignore this field
```

Collection: shoes

```
[
  {_id:ObjectId("11ssa2"), brand: "Hoka", size: "UK7"},
  {_id:ObjectId("0a4sde"), brand: "Nike", size: "UK7"},
  {_id:ObjectId("ya85de"), brand: "Hoka", size: "UK5"}
]
```

find

```
MongoQL: db.shoes.find({"brand":"Hoka"},{"size":1})

SQL : select _id, size from shoes where brand='Hoka'

Return :

[
  {_id:ObjectId("11ssa2"), size: "UK7"},
  {_id:ObjectId("ya85de"), size: "UK5"}
]
```

Using the findOne Methods

findOne

```
db.collection.findOne({query},{projection})

Returns one document from the collection
If multiple documents satisfy the query,
the methods returns the first document.
```

Collection: shoes

```
[
  {_id:ObjectId("11ssa2"), brand: "Hoka", size: "UK7"},
  {_id:ObjectId("0a4sde"), brand: "Nike", size: "UK7"},
  {_id:ObjectId("ya85de"), brand: "Hoka", size: "UK5"}
]
```

findOne

```
MongoQL:  db.shoes.findOne({"brand":"Hoka"})

SQL : select * from shoes where brand='Hoka' limit 1

Return:

{_id:ObjectId("11ssa2"), brand: "Hoka", size: "UK7"}
```

Case Sensitivity in MongoDB

Objects in MongoDB are case sensitive.

For example, the findOne method (performed in the sample_training database on the shoes collection) should be as follows:

db.shoes.findOne() – this is the correct syntax

Db.shoes.findOne() – will not work because Db contains an upper case character

db.Shoes.findOne() – will not work because the collection is saved as “trips” not “Trips”, case sensitivity matters

db.shoes.FindOne() – will not work because case sensitivity also matters on methods. It should be “findOne “not “FindOne”

The findOne method performed in the database on the shoes collection

a. db.shoes.findOne()

b. Db.shoes.findOne()

c. db.Shoes.findOne()

d. db.shoes.FindOne()

e. All of the above

Which of the following is correct

Use of $ in Mongo query language

- Precedes operators in Mongo query language
- Precedes field values
- Aggregation Pipeline

Equal to

{“size”:{$eq:”UK8”}}

{field:{operator : value}}

Logical Operators

$and

Return all documents that match all conditions

$or

Return all documents that match any of the conditions

$nor

Return all documents that fail all of the conditions

{operator: [{condition1},{condition2},..]}

$not

Return all documents that do not match the expression

{operator: [{condition1}]}

Comparison Operators

$eq — Equal to
{“size”:”UK8”}
{“size”:{$eq:”UK8”}}

$ne — Not equal to

$gt — Greater than

$gte — Greater than Or Equal to

$lt — Less than

$lte — Less than Or equal to

$in — values specified in an array
{“size”:{$in:[“UK8”, “UK11”]}}

$nin — Not in

Elements Operators

$exists

Return documents that contain the specified field

db.collection.find({field:{$exists:<boolean>}})

$type

Return documents that contain values of a specified data type

db.collection.find({field:{$type:<BSON type>}})

db.collection.find({field:{$type:[<BSON type1>, <BSON type2>,..]}})

Query Embedded Documents

You can perform read operations using the db.collection.find() method

- Math an Embedded/Nested Document
- Specify Equality Match on a Nested Field
- Specify Match using Query Operator
- Specify AND Condition

Math an Embedded/Nested Document

To specify an equality condition on a field that is an embedded/nested document, use the query filter document

{ <field>: <value>}

where <value> is the document to match.

```
db.inventory.find({ size: { h: 14, w: 21, uom: "cm" } })
```

documents

```
[
  { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
  { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "A" },
  { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" },
  { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
  { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" }
]
```

Specify Equality Match on a Nested Field

The following example selects all documents where the field uom nested in the size field equals "in"

```
db.inventory.find({ "size.uom": "in" })
```

documents

```
[
   { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
   { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "A" },
   { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" },
   { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
   { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" }
]
```

Specify Match using Query Operator

A query filter document can use the query operators to specify conditions in the following form:

{ <field1>: { <operator1>: <value1> }, ... }

```
db.inventory.find({ "size.h": { $lt: 15 } })
```

documents

```
[
  { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
  { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "A" },
  { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" },
  { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
  { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" }
]
```

Specify AND Condition

The following query selects all documents where the nested field h is less than 15, the nested field uom equals "in", and the status field equals "D"

```
db.inventory.find({ "size.h": { $lt: 15 }, "size.uom": "in", status: "D" })
```

documents

```
[
   { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
   { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "A" },
   { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" },
   { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
   { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" }
]
```

Query an Array of Embedded Documents

- Query for a Document Nested in an Array
- Specify a Query Condition on a Field Embedded in an Array of Documents
- Use the Array Index to Query for a Field in the Embedded Document
- A Single Nested Document Meets Multiple Query Conditions on Nested Fields
- Combination of Elements Satisfies the Criteria

Query for a Document Nested in an Array

The following example selects all documents where an element in the instock array matches the specified document

```
db.inventory.find( { "instock": { warehouse: "A", qty: 5 } } )
```

documents

```
[
   { item: "journal", instock: [ { warehouse: "A", qty: 5 }, { warehouse: "C", qty: 15 } ] },
   { item: "notebook", instock: [ { warehouse: "C", qty: 5 } ] },
   { item: "paper", instock: [ { warehouse: "A", qty: 60 }, { warehouse: "B", qty: 15 } ] },
   { item: "planner", instock: [ { warehouse: "A", qty: 40 }, { warehouse: "B", qty: 5 } ] },
   { item: "postcard", instock: [ { warehouse: "B", qty: 15 }, { warehouse: "C", qty: 35 } ] }
)
```

Specify a Query Condition on a Field Embedded in an Array of Documents

If you do not know the index position of the document nested in the array, concatenate the name of the array field, with a dot (.) and the name of the field in the nested document.

The following example selects all documents where the instock array has at least one embedded document that contains the field qty whose value is less than or equal to 20

```
db.inventory.find( { 'instock.qty': { $lte: 20 } } )
```

documents

```
[
   { item: "journal", instock: [ { warehouse: "A", qty: 5 }, { warehouse: "C", qty: 15 } ] },
   { item: "notebook", instock: [ { warehouse: "C", qty: 5 } ] },
   { item: "paper", instock: [ { warehouse: "A", qty: 60 }, { warehouse: "B", qty: 15 } ] },
   { item: "planner", instock: [ { warehouse: "A", qty: 40 }, { warehouse: "B", qty: 5 } ] },
   { item: "postcard", instock: [ { warehouse: "B", qty: 15 }, { warehouse: "C", qty: 35 } ] }
)
```

Use the Array Index to Query for a Field in the Embedded Document

Using dot notation, you can specify query conditions for field in a document at a particular index or position of the array. The array uses zero-based indexing.

The following example selects all documents where the instock array has as its first element a document that contains the field qty whose value is less than or equal to 20

```
db.inventory.find( { 'instock.0.qty': { $lte: 20 } } )
```

documents

```
[
   { item: "journal", instock: [ { warehouse: "A", qty: 5 }, { warehouse: "C", qty: 15 } ] },
   { item: "notebook", instock: [ { warehouse: "C", qty: 5 } ] },
   { item: "paper", instock: [ { warehouse: "A", qty: 60 }, { warehouse: "B", qty: 15 } ] },
   { item: "planner", instock: [ { warehouse: "A", qty: 40 }, { warehouse: "B", qty: 5 } ] },
   { item: "postcard", instock: [ { warehouse: "B", qty: 15 }, { warehouse: "C", qty: 35 } ] }
)
```

A Single Nested Document Meets Multiple Query Conditions on Nested Fields

Use $elemMatch operator to specify multiple criteria on an array of embedded documents such that at least one embedded document satisfies all the specified criteria.

The following example queries for documents where the instock array has at least one embedded document that contains both the field qty equal to 5 and the field warehouse equal to A

```
db.inventory.find( { "instock": { $elemMatch: { qty: 5, warehouse: "A" } } } )
```

documents

```
[
   { item: "journal", instock: [ { warehouse: "A", qty: 5 }, { warehouse: "C", qty: 15 } ] },
   { item: "notebook", instock: [ { warehouse: "C", qty: 5 } ] },
   { item: "paper", instock: [ { warehouse: "A", qty: 60 }, { warehouse: "B", qty: 15 } ] },
   { item: "planner", instock: [ { warehouse: "A", qty: 40 }, { warehouse: "B", qty: 5 } ] },
   { item: "postcard", instock: [ { warehouse: "B", qty: 15 }, { warehouse: "C", qty: 35 } ] }
)
```

The following example queries for documents where
the instock array has at least one embedded document that
contains the field qty that is greater than 10 and less than or equal to 20

```
db.inventory.find( { "instock": { $elemMatch: { qty: { $gt: 10, $lte: 20 } } } } )
```

documents

```
[
   { item: "journal", instock: [ { warehouse: "A", qty: 5 }, { warehouse: "C", qty: 15 } ] },
   { item: "notebook", instock: [ { warehouse: "C", qty: 5 } ] },
   { item: "paper", instock: [ { warehouse: "A", qty: 60 }, { warehouse: "B", qty: 15 } ] },
   { item: "planner", instock: [ { warehouse: "A", qty: 40 }, { warehouse: "B", qty: 5 } ] },
   { item: "postcard", instock: [ { warehouse: "B", qty: 15 }, { warehouse: "C", qty: 35 } ] }
)
```

Combination of Elements Satisfies the Criteria

If the compound query conditions on an array field do not use the $elemMatch operator, the query selects those documents whose array contains any combination of elements that satisfies the conditions.

For example, the following query matches documents where any document nested in the instock array has the qty field greater than 10 and any document (but not necessarily the same embedded document) in the array has the qty field less than or equal to 20

```
db.inventory.find( { "instock.qty": { $gt: 10,  $lte: 20 } } )
```

documents

```
[
   { item: "journal", instock: [ { warehouse: "A", qty: 5 }, { warehouse: "C", qty: 15 } ] },
   { item: "notebook", instock: [ { warehouse: "C", qty: 5 } ] },
   { item: "paper", instock: [ { warehouse: "A", qty: 60 }, { warehouse: "B", qty: 15 } ] },
   { item: "planner", instock: [ { warehouse: "A", qty: 40 }, { warehouse: "B", qty: 5 } ] },
   { item: "postcard", instock: [ { warehouse: "B", qty: 15 }, { warehouse: "C", qty: 35 } ] }
)
```

The following example queries for documents where the instock array has at least one embedded document that contains the field qty equal to 5 and at least one embedded document (but not necessarily the same embedded document) that contains the field warehouse equal to A

```
db.inventory.find( { "instock.qty": 5, "instock.warehouse": "A" } )
```

documents

```
[
   { item: "journal", instock: [ { warehouse: "A", qty: 5 }, { warehouse: "C", qty: 15 } ] },
   { item: "notebook", instock: [ { warehouse: "C", qty: 5 } ] },
   { item: "paper", instock: [ { warehouse: "A", qty: 60 }, { warehouse: "B", qty: 15 } ] },
   { item: "planner", instock: [ { warehouse: "A", qty: 40 }, { warehouse: "B", qty: 5 } ] },
   { item: "postcard", instock: [ { warehouse: "B", qty: 15 }, { warehouse: "C", qty: 35 } ] }
)
```

Deleting Documents - deleteOne

deleteOne

```
db.collection.deleteOne(<filter>)

The first document that matches
 the filter condition will be deleted
```

Collection: shoes

```
{
    _id: ObjectId("537a41a"),
    brand: "Nike",
    item: "Fragment x Sacai x Nike LDWaffle",
    price: 15000,
    color: "gray",
    size: "UK5"
}
```

deleteOne

```
MongoQL:

db.shoes.deleteOne({"_id": ObjectId("537a41a")})

Return :

{ "acknowledged" : true, "deletedCount" : 1 }
```

Delete Documents - deleteMany

deleteMany

```
db.collection.deleteMany(<filter>)

All documents that match the condition
will be deleted.
```

Collection: shoes

```
[{
    _id: ObjectId("537a41a"),
    brand: "Nike",
    item: "Fragment x Sacai x Nike LDWaffle",
    price: 15000,
    color: "grey",
    size: "UK5"
}, …]
```

deleteMany

```
MongoQL:

db.shoes.deleteMany({"color": "grey"})

Return :
{ "acknowledged" : true, "deletedCount" : 8 }
```

Updating Documents

updateOne

```
db.collection.updateOne(
 <filter>,<update>,<options>
)
```

Update one will update the first documents that match the filter criteria.

updateMany

```
db.collection.updateMany(
 <filter>,<update>,<options>
)
```

Update many will update all documents that match the filter criteria.

$set

db.collection.updateOne({filter},{$set:{field:value,…},{options}}

db.collection.updateMany({filter},{$set:{field:value,…},{options}}

Update Operators

$unset

Removes the specified field from a document.

db.collection.updateMany({filter},{$unset:{field:value,…}},{options})

$inc

Increments the value of the field by the specified amount.

db.collection.updateMany({filter},{$inc:{field:value,…}},{options})

$rena
me

Renames a field.

db.collection.updateMany({filter},{$rename:{field:value,…}},{options})

$push

Adds an item to an array.

db.collection.updateMany({filter},{$push:{field:value,…}},{options})

Please refer to this link for additional information on MongoDB Update Operators

Upsert

- Upsert is a combination of insert and **update (insert + update = upsert)**.
- We can use the upsert with different update methods.

{ upsert : true }

```
db.shoes.update(
   {
     brand: "Nike",
     item: "Fragment x Sacai x Nike LDWaffle",
     size: "UK8",
     color: "gray"
   },
   { price: 18000 },
   { upsert: true }
)
```

After the Class

Homework 👉 ***SQL vs NoSQL: comparison table***

Practice! Practice! Practice! 😸

Welcome to the Course

- Data Modelling
- Index and Performance
- Aggregation

Data Modeling Considerations

- The right schema design can eanble you to make the most out of your database
- When designing your database it is important to consider:

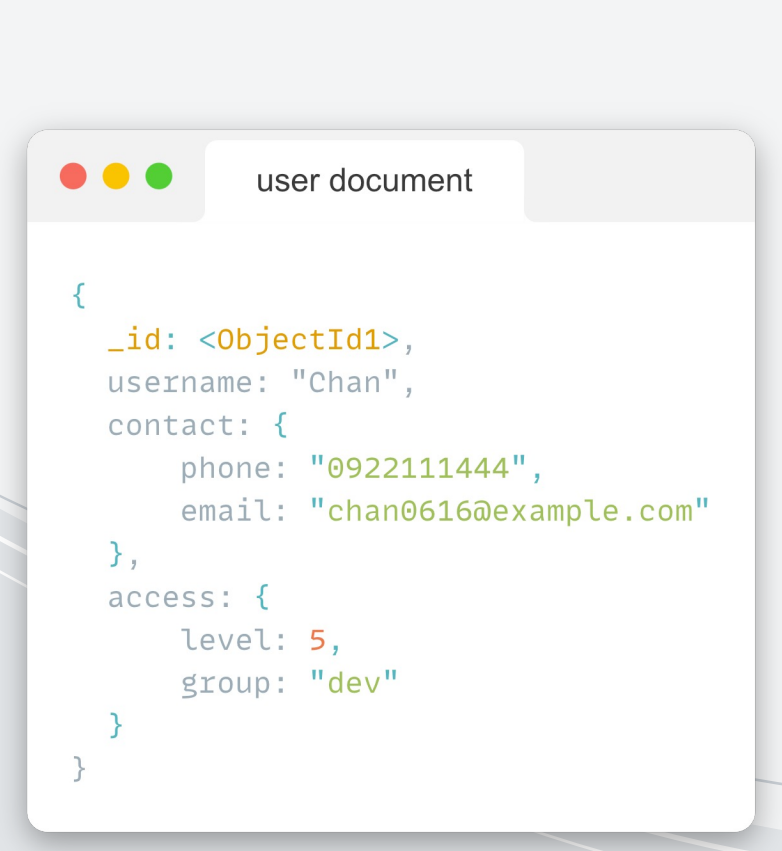
Data Use and Performance

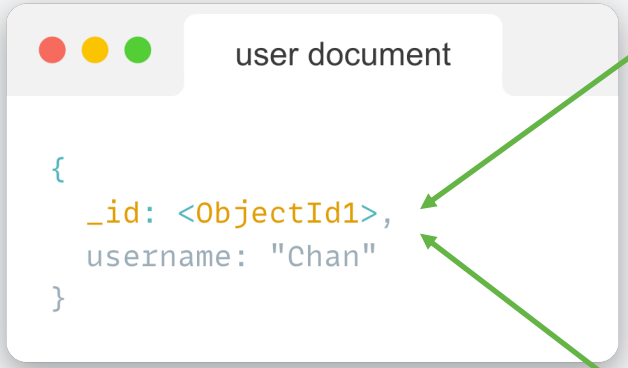
Determine important queries

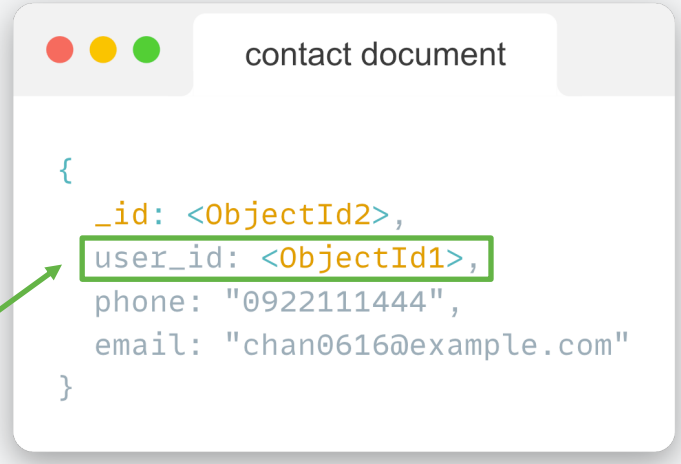
Read vs Write

Data Model Design

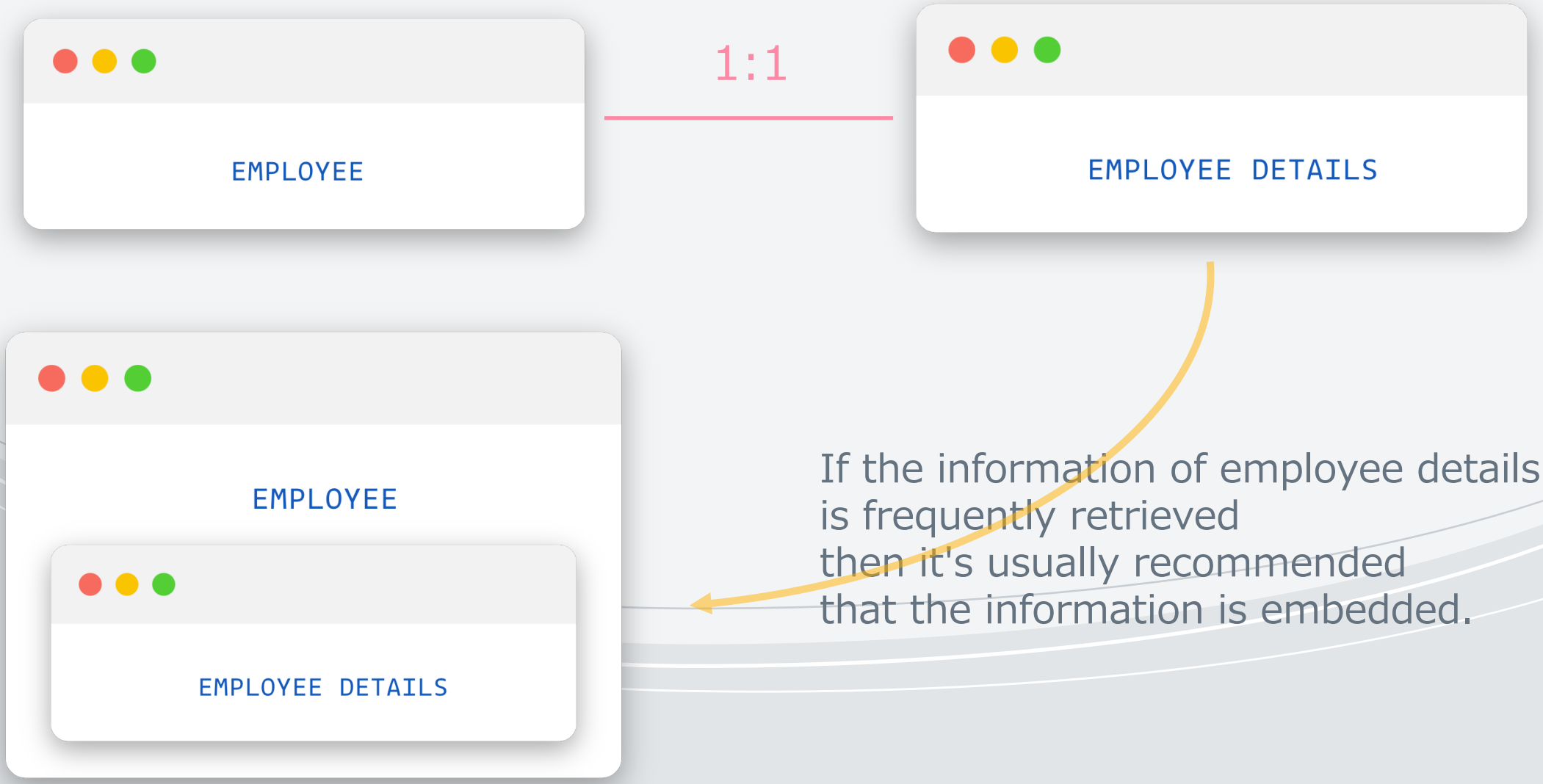
Embedding vs Referencing



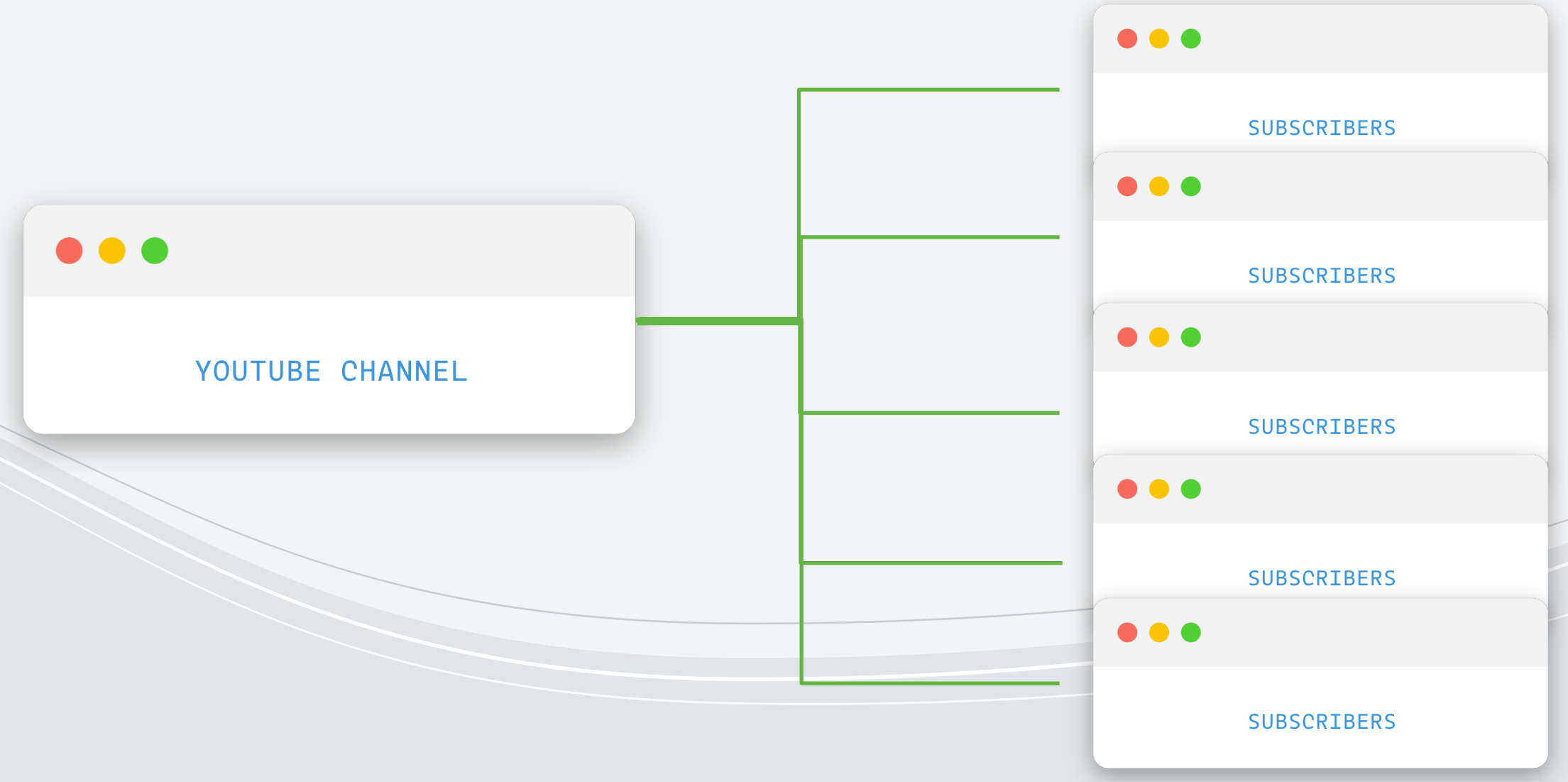




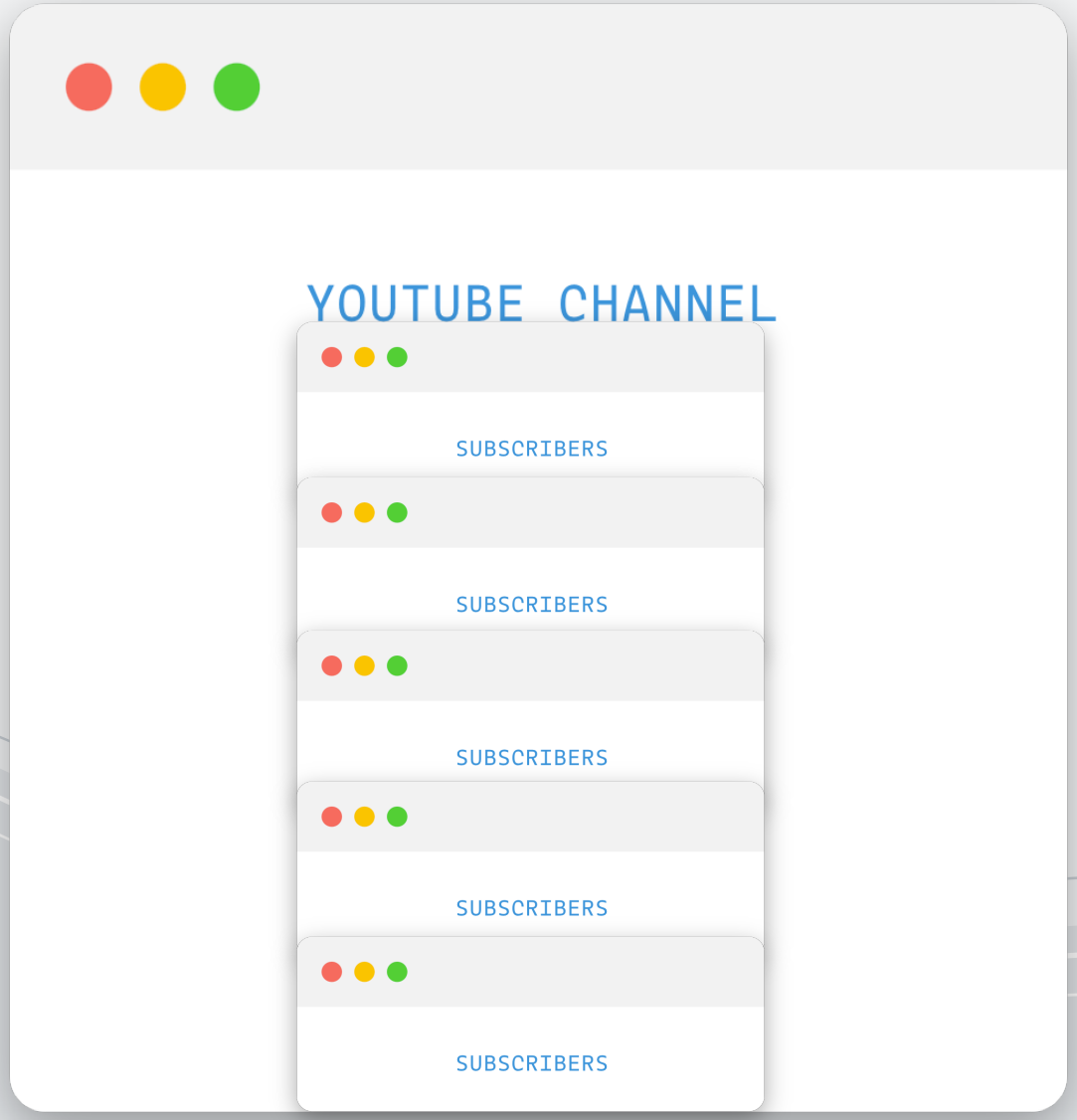
1:1 Relationship



1:Many Relationship

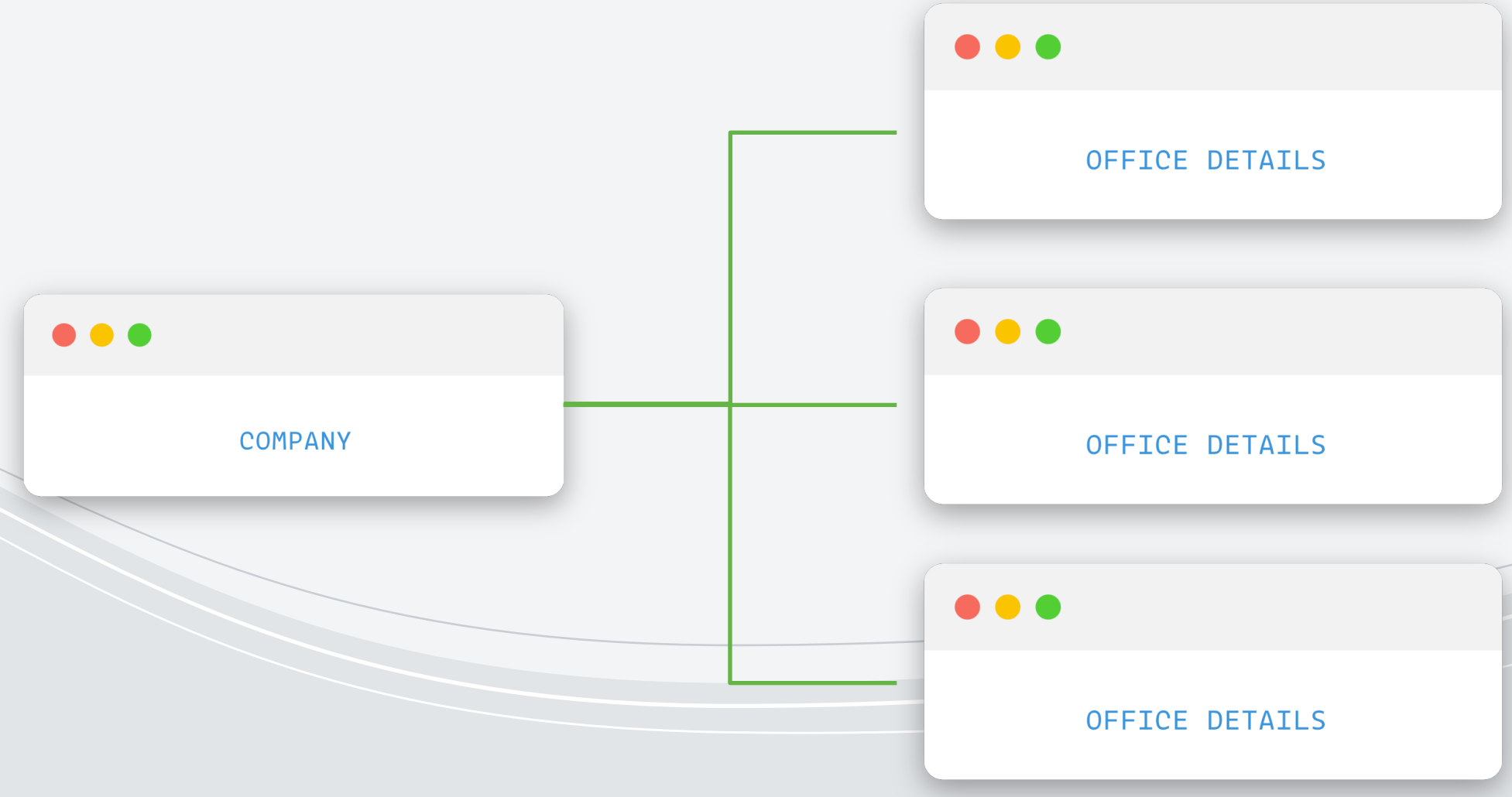


1:Many Relationship

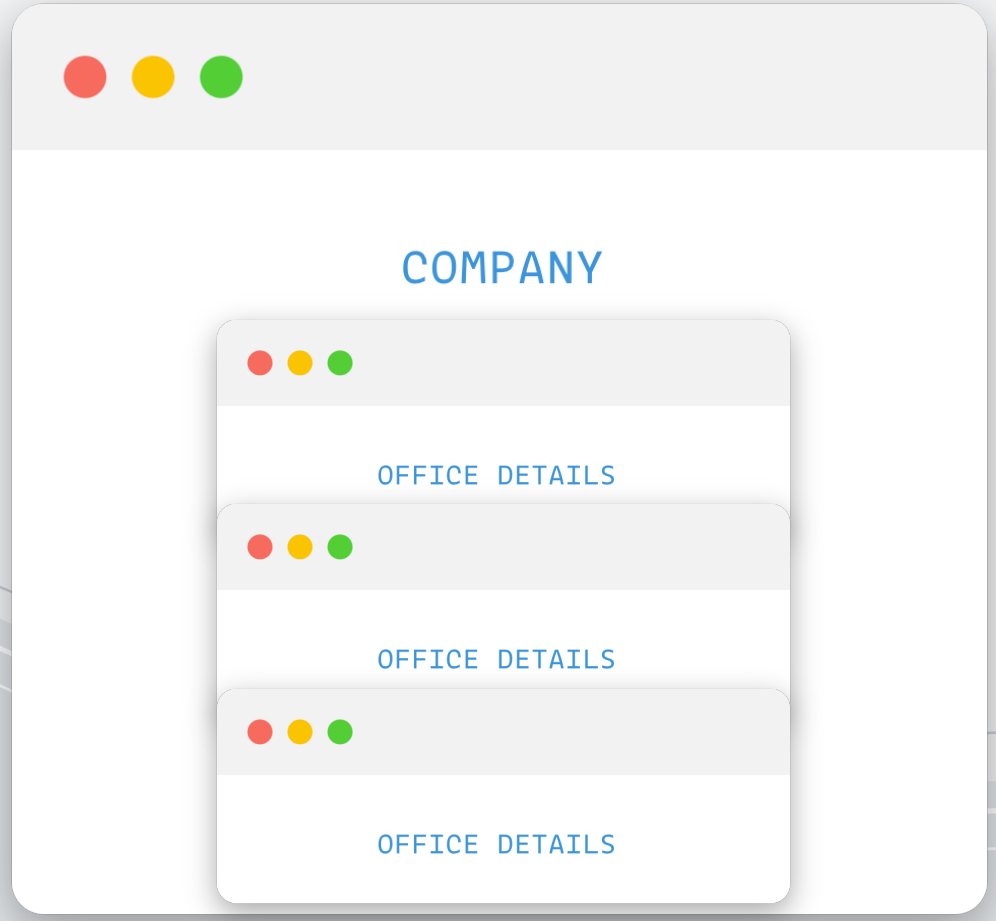


Too many embedded documents

1:Many Relationship

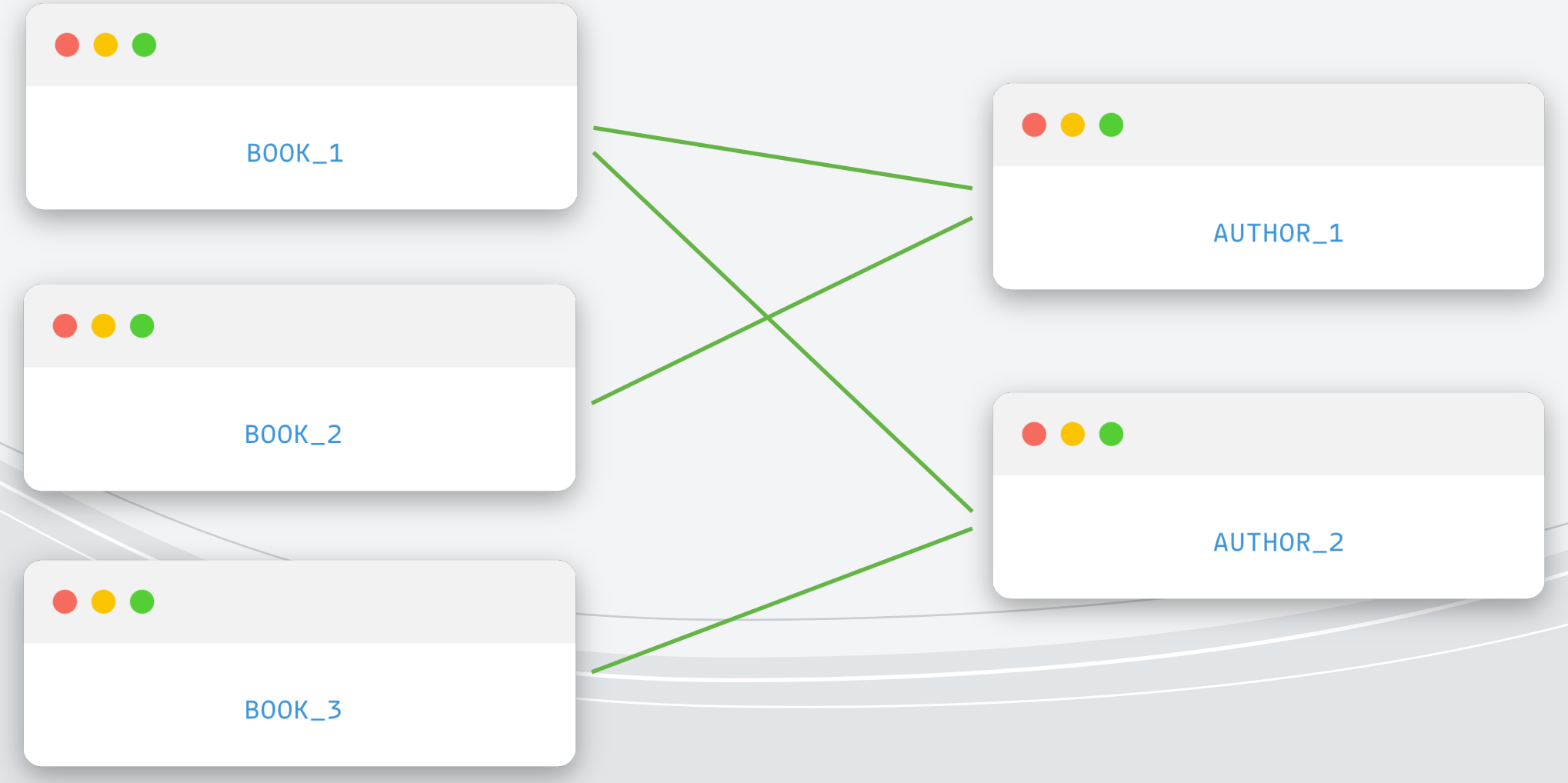


1:Many Relationship

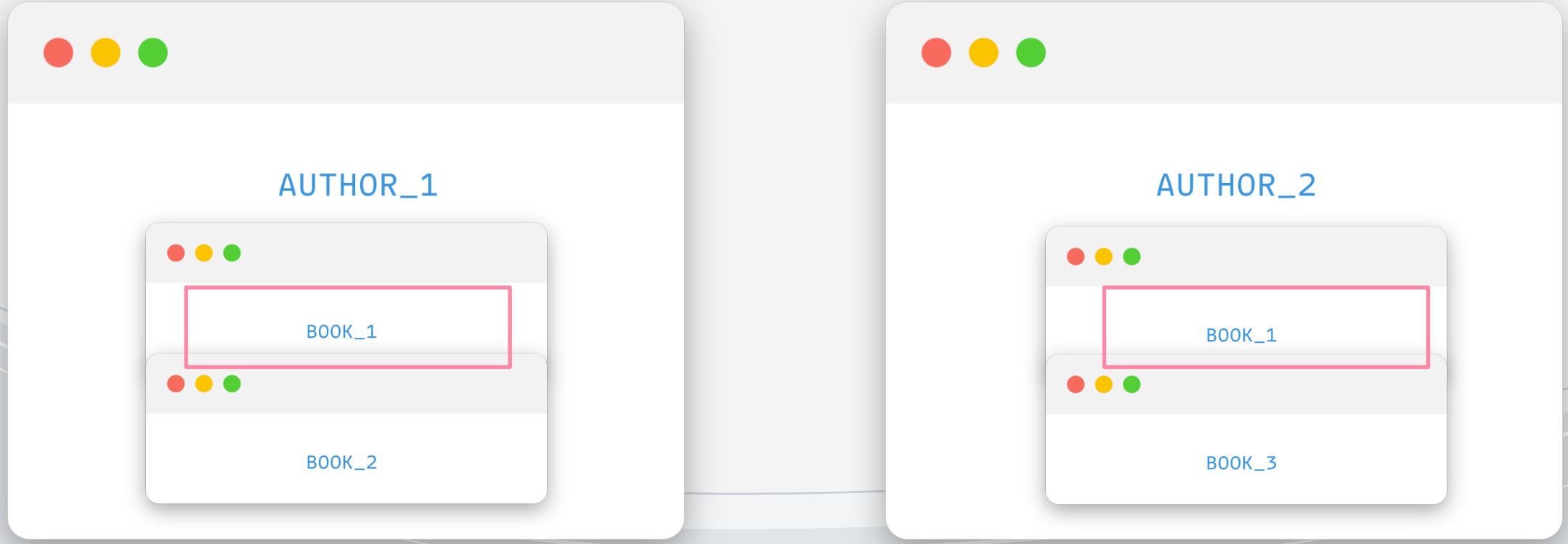


More suited to embedding documents

Many:Many Relationship



Many:Many Relationship



Building with Pattern

Here is a very useful resource on Schema Patterns.

Schema Patterns are very useful when deciding how to model your data in MongoDB so please be sure to check it out.

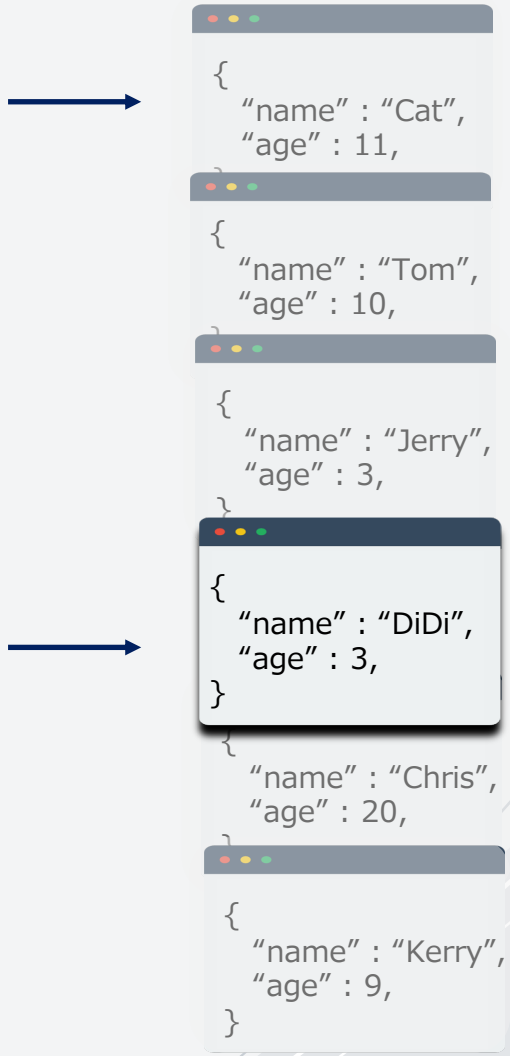
Index and Performance

- Introduction to Index
- Single Field Index
- Compound Indexes

Introduction to Index

db.collection.find({“name”:”DiDi”})

Collection scan(COLLSCAN) is one where every document in the collection is scanned

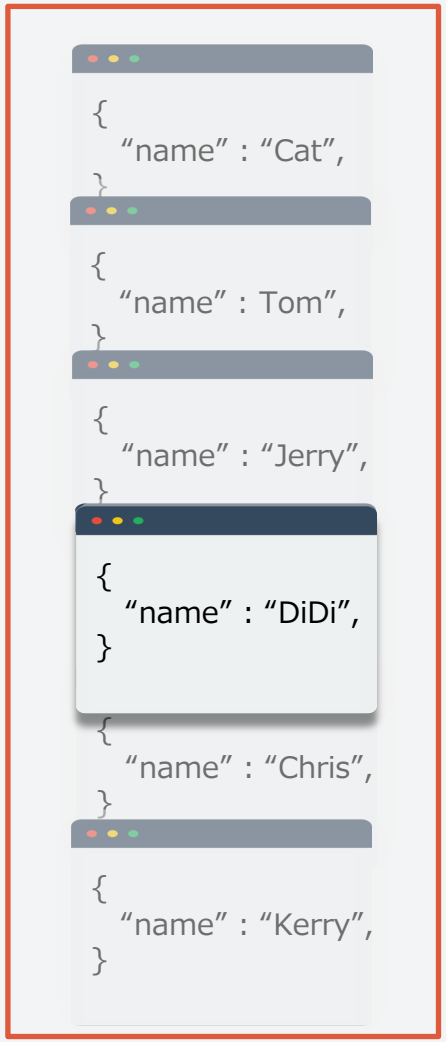


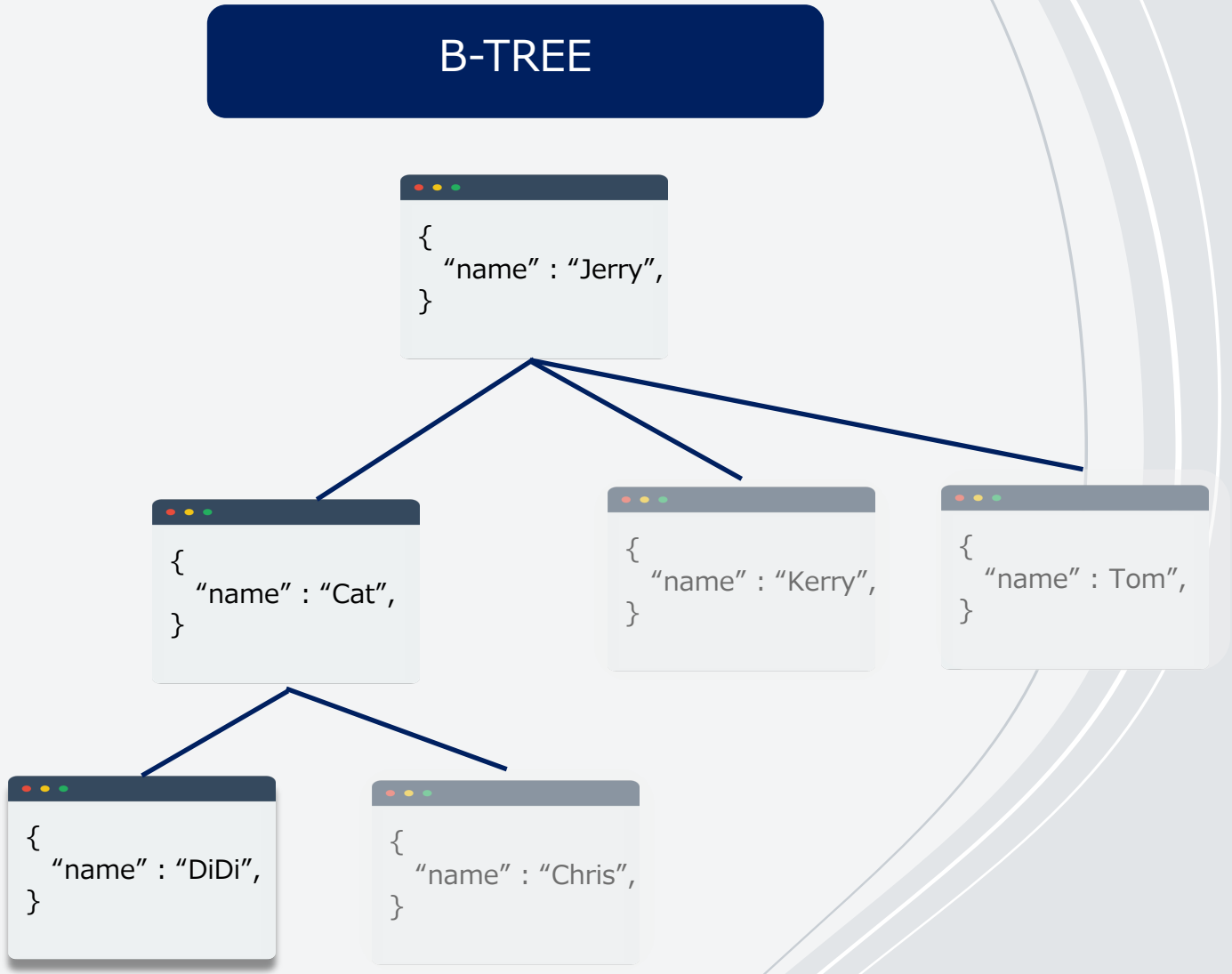
Introduction to Index

db.collection.find({“name”:”DiDi”})

Introduction to Index

Index on name





When to use Index

- If you have a scenario that needs sorting, when the memory sorting exceeds 32 MB.
- Certain fields have unique requirements.
- When the number of Documents is large.
- Create an Index for High-Cardinality Field.

When is it not recommended to use Index

- It is not recommended to use Index when it cannot effectively filter data.
- Don’t set Index on the field that will be updated frequently

Single Field Index

Create an Ascending Index on a Single Field

Collection: records

```
{
  "_id": ObjectId("570c04a4ad233577f97dc459"),
  "score": 1034,
  "location": { state: "NY", city: "New York" }
}
```

The following operation creates an **ascending** index on the **score** field of the **records** collection:

```
db.records.createIndex( { score: 1 } )
```

A value of **-1** specifies an index that orders items in **descending** order.

The created index will support queries that select on the field **score**, such as the following:

```
db.records.find( { score: 2 } )
db.records.find( { score: { $gt: 10 } } )
```

Single Field Index

Create an Index on an Embedded Field

Collection: records

```
{
  "_id": ObjectId("570c04a4ad233577f97dc459"),
  "score": 1034,
  "location": { state: "NY", city: "New York" }
}
```

You can create indexes on fields within embedded documents. Indexes on embedded fields allow you to use a "dot notation," to introspect into embedded documents.

The following operation creates an index on the **location.state** field:

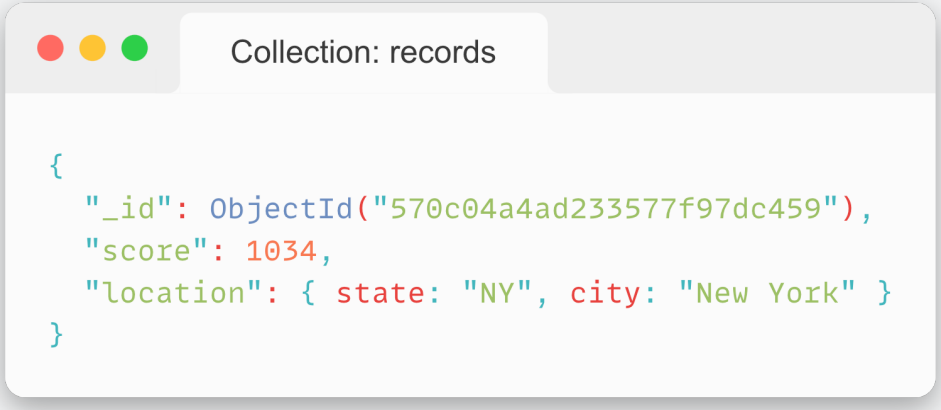
```
db.records.createIndex( { "location.state": 1 } )
```

The created index will support queries that select on the field **location.state**, such as the following:

```
db.records.find( { "location.state": "CA" } )
```

Single Field Index

Create an Index on Embedded Document



The location field is an embedded document, containing the embedded fields city and state. The following command creates an index on the location field as a whole:

```
db.records.createIndex( { location: 1 } )
```

The following query can use the index on the **location** field:

```
db.records.find( { location: { city: "New York", state: "NY" } } )
```

Compound Indexes

Create a Compound Index

Collection: products

```
{
 "_id": ObjectId(...),
 "item": "Banana",
 "category": ["food", "produce", "grocery"],
 "location": "4th Street Store",
 "stock": 4,
 "type": "cases"
}
```

The following operation creates an ascending index on the **item** and **stock** fields:

```
db.products.createIndex( { "item": 1, "stock": 1 } )
```

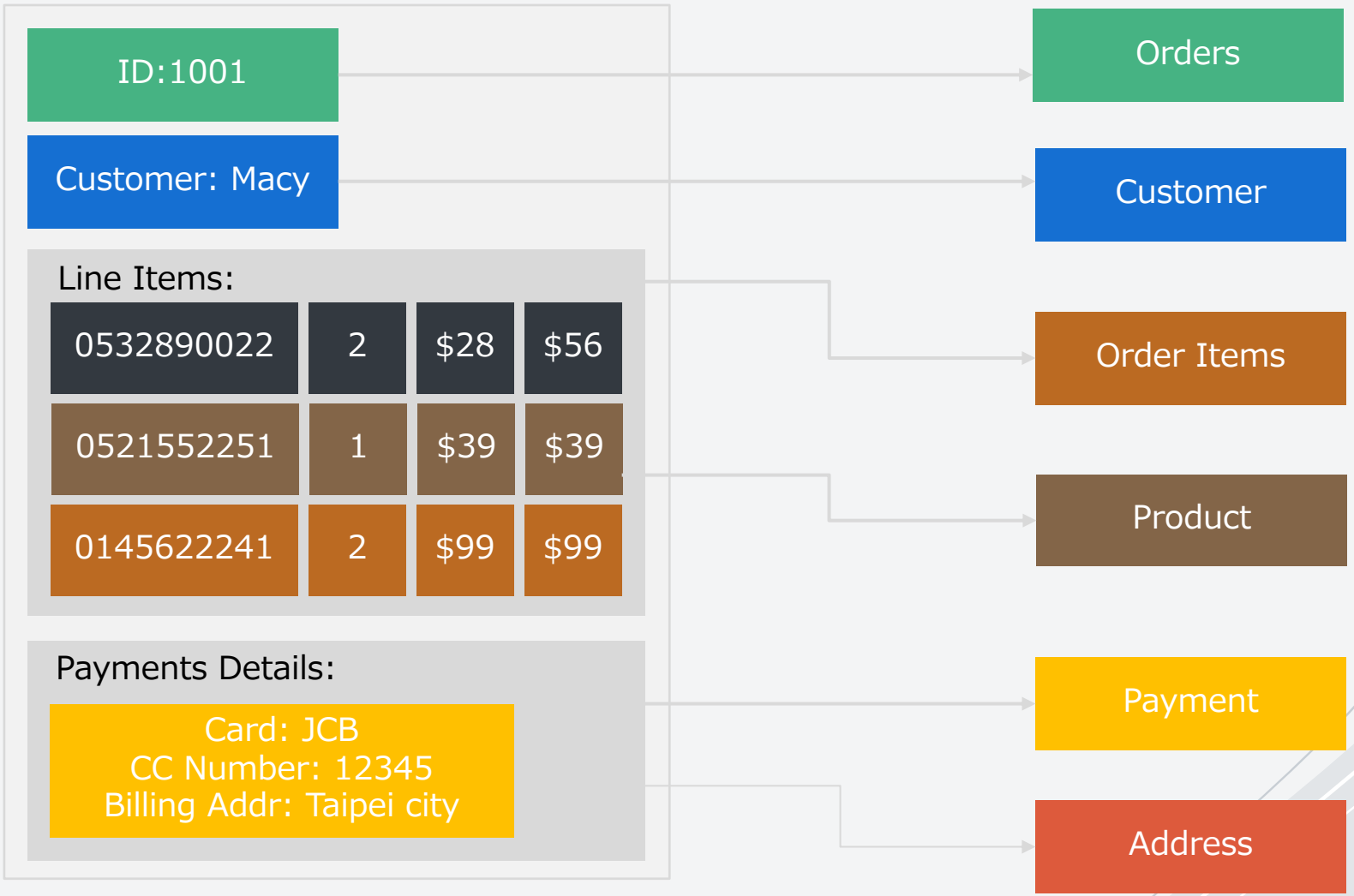
the index supports queries on the item field as well as both item and stock fields:

```
db.products.find( { item: "Banana" } )
db.products.find( { item: "Banana", stock: { $gt: 5 } } )
```

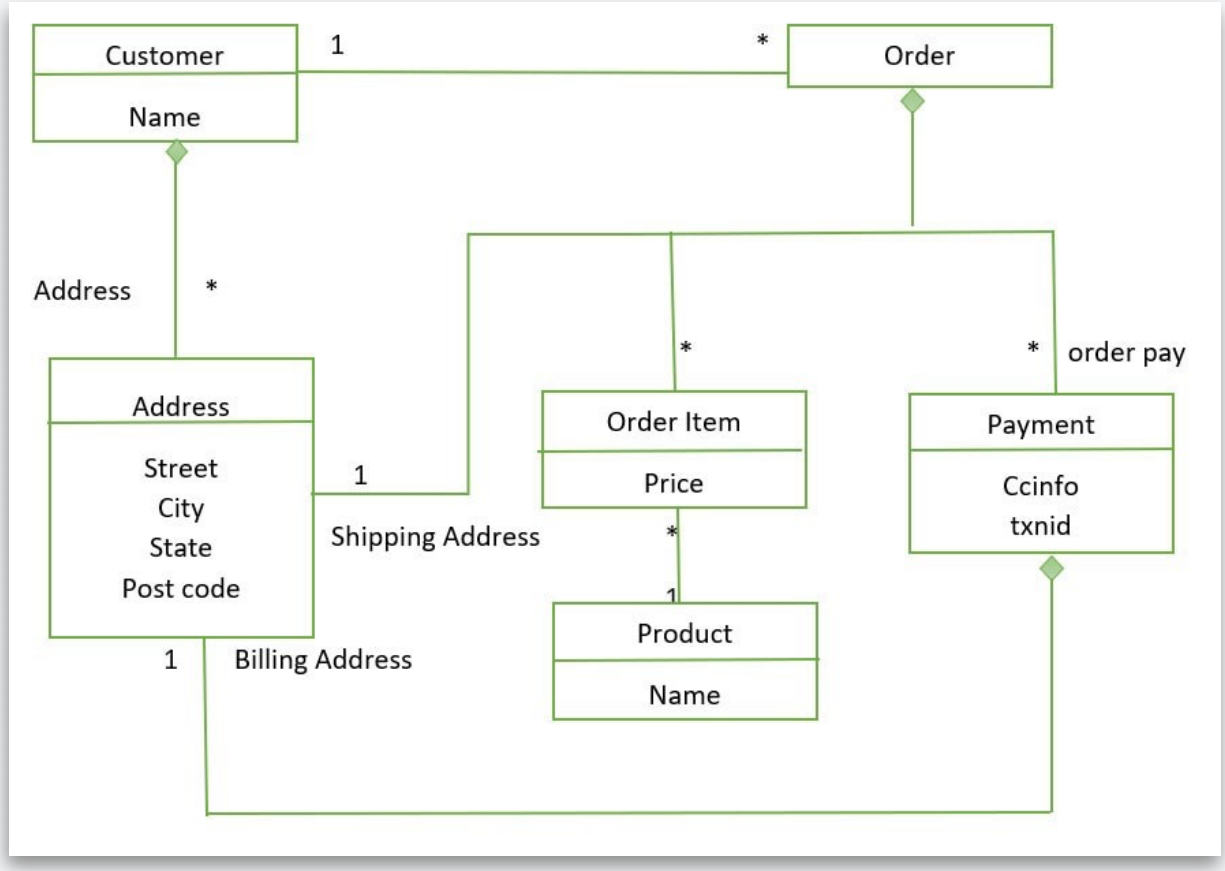
How to design and use Index

- Use Compound Index instead of Single Field Index
- Follow ESR rule
  - Equality : Field for equality query.
  - Sort : Field to sort.
  - Range : Field of the data range to access.

Aggregate Data Models



Aggregate Data Models



Computed Results


```
{
  "customer": {
      "id": 1,
      "name": "Martin",
      "billingAddress": [{
          "city": "Chicago"
      }],
      "orders": [{
          "id": 99,
          "customerId": 1,
          "orderItems": [{
              "productId": 27,
              "price": 32.45,
              "productName": "NoSQL Distilled"
           }],
           "shippingAddress": [{
               "city": "Chicago"
            }]

           "orderPayment": [{
                "ccinfo": "1000-1000-1000-1000",
                "txnId": "abelif879rft",
                "billingAddress": {
                    "city": "Chicago"
                 }
            }],
        }]
    }
}
```

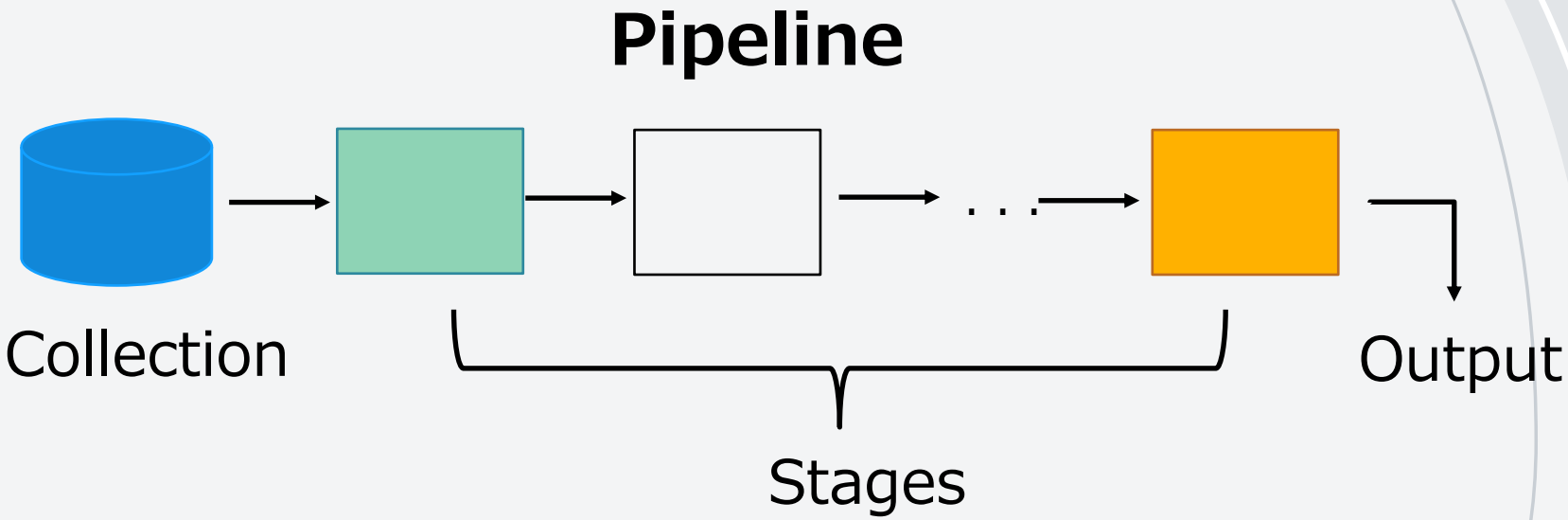
Aggregation Pipeline Example

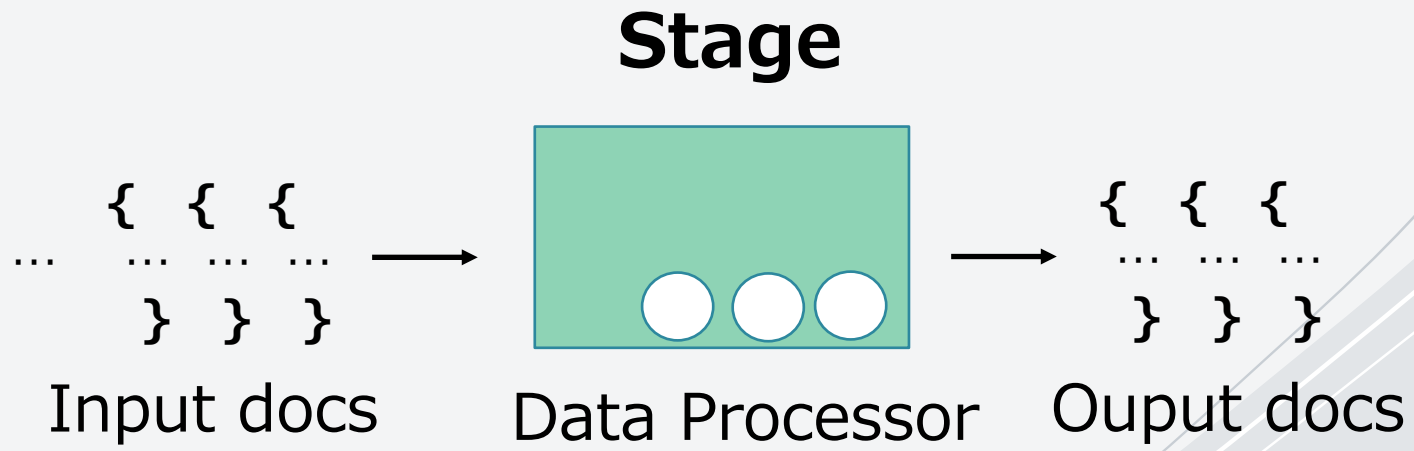
Aggregation operations process data records and return computed results.

aggregate

```
db.orders.aggregate( [

   // Stage 1: Filter pizza order documents by pizza size
   {
      $match: { size: "medium" }
   },

   // Stage 2: Group remaining documents by pizza name and calculate total quantity
   {
      $group: { _id: "$name", totalQuantity: { $sum: "$quantity" } }
   }

] )
```

Aggregation Pipelines





Aggregation Pipeline Stages/Operators

👉 Aggregation Pipeline Stages

- **$lookup**
- $match
- $group
- $set
- $unset

👉 Aggregation Pipeline Operators

- $gt

$lookup

Equality Match with a Single Join Condition

$lookup

```
{
   $lookup:
     {
       from: <collection to join>,
       localField: <field from the input documents>,
       foreignField: <field from the documents of the "from" collection>,
       as: <output array field>
     }
}
```

pseudo-SQL

```
SELECT *, <output array field>
FROM collection
WHERE <output array field> IN (
   SELECT *
   FROM <collection to join>
   WHERE <foreignField> = <collection.localField>
);
```

Perform a Single Equality Join with $lookup

Collection: orders

```
db.orders.insertMany( [
   { "_id" : 1, "item" : "almonds", "price" : 12, "quantity" : 2 },
   { "_id" : 2, "item" : "pecans", "price" : 20, "quantity" : 1 },
   { "_id" : 3  }
] )
```

Collection: inventory

```
db.inventory.insertMany( [
   { "_id" : 1, "sku" : "almonds", "description": "product 1", "instock" : 120 },
   { "_id" : 2, "sku" : "bread", "description": "product 2", "instock" : 80 },
   { "_id" : 3, "sku" : "cashews", "description": "product 3", "instock" : 60 },
   { "_id" : 4, "sku" : "pecans", "description": "product 4", "instock" : 70 },
   { "_id" : 5, "sku": null, "description": "Incomplete" },
   { "_id" : 6 }
] )
```

$lookup

```
db.orders.aggregate( [
   {
     $lookup:
       {
         from: "inventory",
         localField: "item",
         foreignField: "sku",
         as: "inventory_docs"
       }
   }
] )
```

pseudo-SQL

```
SELECT *, inventory_docs
FROM orders
WHERE inventory_docs IN (
   SELECT *
   FROM inventory
   WHERE sku = orders.item
);
```

$lookup

Correlated Subqueries Using Concise Syntax

$lookup

```
{
   $lookup:
      {
         from: <foreign collection>,
         localField: <field from local collection's documents>,
         foreignField: <field from foreign collection's documents>,
         let: { <var_1>: <expression>, …, <var_n>: <expression> },
         pipeline: [ <pipeline to run> ],
         as: <output array field>
      }
}
```

pseudo-SQL

```
SELECT *, <output array field>
FROM localCollection
WHERE <output array field> IN (
   SELECT <documents as determined from the pipeline>
   FROM <foreignCollection>
   WHERE <foreignCollection.foreignField> = <localCollection.localField>
   AND <pipeline match condition>
);
```

👉The $lookup accepts a document with fields

$lookup example: pipeline example with condition

let (optional): To reference variables in pipeline stages, use the "**$$<variable>**" syntax.

pipeline: Specifies the pipeline to run on the joined collection. The pipeline determines the resulting documents from the joined collection.

$expr: $expr can build query expressions that compare fields from the same document in a $match stage. If the $match stage is part of a $lookup stage, $expr can compare fields using let variables.

Aggregation Pipeline

```
db.orders.aggregate( [
   {
      $lookup: {
         from: "restaurants",
         let: { orders_restaurant_name: "$restaurant_name",
                orders_drink: "$drink" },
         pipeline: [ {
            $match: {
               $expr: {
                  $and: [
                     { $eq: [ "$$orders_restaurant_name", "$name" ] },
                     { $in: [ "$$orders_drink", "$beverages" ] }
                  ]
               }
            }
         } ],
         as: "matches"
      }
   }
] )
```

Perform a Concise Correlated Subquery with $lookup

Collection: restaurants

```
db.restaurants.insertMany( [
   {
      _id: 1,
      name: "American Steak House",
      food: [ "filet", "sirloin" ],
      beverages: [ "beer", "wine" ]
   },
   {
      _id: 2,
      name: "Honest John Pizza",
      food: [ "cheese pizza", "pepperoni pizza" ],
      beverages: [ "soda" ]
   }
] )
```

Collection: orders

```
db.orders.insertMany( [
   {
      _id: 1,
      item: "filet",
      restaurant_name: "American Steak House"
   },
   {
      _id: 2,
      item: "cheese pizza",
      restaurant_name: "Honest John Pizza",
      drink: "lemonade"
   },
   {
      _id: 3,
      item: "cheese pizza",
      restaurant_name: "Honest John Pizza",
      drink: "soda"
   }
] )
```

Aggregation Pipeline

```
db.orders.aggregate( [
   {
      $lookup: {
         from: "restaurants",
         let: { orders_restaurant_name: "$restaurant_name",
                orders_drink: "$drink" },
         pipeline: [ {
            $match: {
               $expr: {
                  $and: [
                     { $eq: [ "$$orders_restaurant_name", "$name" ] },
                     { $in: [ "$$orders_drink", "$beverages" ] }
                  ]
               }
            }
         } ],
         as: "matches"
      }
   }
] )
```

pseudo-SQL

```
SELECT *, matches
FROM orders
WHERE matches IN (
   SELECT *
   FROM restaurants
   WHERE restaurants.name = orders.restaurant_name
   AND restaurants.beverages = orders.drink
);
```

Aggregation - Multi-Field Join & One-to-Many

Scenario

You want to generate a report to list all the orders made for each product in 2020. To achieve this, you need to take a shop's **products collection** and join each product record to all its orders stored in an **orders collection**. There is a 1:many relationship between both collections, based on a match of two fields on each side. Rather than joining on a single field like **product_id** (which doesn't exist in this data set), you need to use two common fields to join (**product_name** and **product_variation**). Note that the requirement to **perform a 1:many join** does not mandate the need to join by multiple fields on each side of the join. However, in this example, it has been deemed beneficial to show both of these aspects in one place.

👍🏽 Exercises: 01_aggregation-oneToMany

Sample Data

Collection: products

```
db.products.insertMany([
  {
    "name": "Asus Laptop",
    "variation": "Ultra HD",
    "category": "ELECTRONICS",
    "description": "Great for watching movies",
  },
  {
    "name": "Asus Laptop",
    "variation": "Normal Display",
    "category": "ELECTRONICS",
    "description": "Good value laptop for students",
  },
  {
    "name": "The Day Of The Triffids",
    "variation": "1st Edition",
    "category": "BOOKS",
    "description": "Classic post-apocalyptic novel",
  },
  {
    "name": "The Day Of The Triffids",
    "variation": "2nd Edition",
    "category": "BOOKS",
    "description": "Classic post-apocalyptic novel",
  },
  {
    "name": "Morphy Richards Food Mixer",
    "variation": "Deluxe",
    "category": "KITCHENWARE",
    "description": "Luxury mixer turning good cakes into great",
  },
  {
    "name": "Karcher Hose Set",
    "variation": "Full Monty",
    "category": "GARDEN",
    "description": "Hose + nosels + winder for tidy storage",
  },
]);
```

Collection: orders

```
// Create index for the orders collection
db.orders.createIndex({"product_name": 1, "product_variation": 1});

// Insert 4 records into the orders collection
db.orders.insertMany([
  {
    "customer_id": "elise_smith@myemail.com",
    "orderdate": ISODate("2020-05-30T08:35:52Z"),
    "product_name": "Asus Laptop",
    "product_variation": "Normal Display",
    "value": NumberDecimal("431.43"),
  },
  {
    "customer_id": "tj@wheresmyemail.com",
    "orderdate": ISODate("2019-05-28T19:13:32Z"),
    "product_name": "The Day Of The Triffids",
    "product_variation": "2nd Edition",
    "value": NumberDecimal("5.01"),
  },
  {
    "customer_id": "oranieri@warmmail.com",
    "orderdate": ISODate("2020-01-01T08:25:37Z"),
    "product_name": "Morphy Richards Food Mixer",
    "product_variation": "Deluxe",
    "value": NumberDecimal("63.13"),
  },
  {
    "customer_id": "jjones@tepidmail.com",
    "orderdate": ISODate("2020-12-26T08:55:46Z"),
    "product_name": "Asus Laptop",
    "product_variation": "Normal Display",
    "value": NumberDecimal("429.65"),
  },
]);
```

```
var pipeline = [
  // Join by 2 fields in in products collection to 2 fiel
  // in orders collection
  {"$lookup": {
    "from": "orders",
    "let": {
      "prdname": "$name",
      "prdvartn": "$variation",
    },
    // Embedded pipeline to control how the join is matched
    "pipeline": [
      // Join by two fields in each side
      {"$match":
        {"$expr":
          {"$and": [
            {"$eq": ["$product_name",  "$$prdname"]},
            {"$eq": ["$product_variation",  "$$prdvartn"]},
          ]},
        },
      },

      // Match only orders made in 2020
      {"$match": {
        "orderdate": {
          "$gte": ISODate("2020-01-01T00:00:00Z"),
          "$lt": ISODate("2021-01-01T00:00:00Z"),
        }
      }},

      // Exclude some unwanted fields from the right side of the join
      {"$unset": [
        "_id",
        "product_name",
        "product_variation",
      ]},
    ],
    as: "orders",
  }},

  // Only show products that have at least one order
  {"$match": {
    "orders": {"$ne": []},
  }},

  // Omit unwanted fields
  {"$unset": [
    "_id",
```

Execution

db.products.aggregate(pipeline);

🤙 MongoDB Exercises