# The Fine Art of Schema Design: Dos and Don'ts

Matias Cascallares
Senior Solutions Architect, MongoDB Inc.
matias@mongodb.com

# Who am I?

- Originally from Buenos Aires, Argentina
- Solutions Architect at MongoDB Inc based in Singapore
- Software Engineer, most of my experience in web environments
- In my toolbox I have Java, Python and Node.js

mongoDB

# RDBMs

- Relational databases are made up of **tables**
- Tables are made up of **rows:**
  - All rows have **identical structure**
  - Each row has the same number of **columns**
  - Every cell in a column stores the **same type of data**

mongoDB

MONGODB IS A

# DOCUMENT ORIENTED

DATABASE

# Show me a document

```
{
    "name"       :  "Matias Cascallares",
    "title"      :  "Senior Solutions Architect",
    "email"      :  "matias@mongodb.com",
    "birth_year" :  1981,
    "location"   :  [ "Singapore", "Asia"],
    "phone"      :  {
                        "type"   : "mobile",
                        "number" : "+65 8591 3870"
                    }
}
```

# Document Model

- MongoDB is made up of **collections**
- Collections are composed of **documents**
  - Each document is a set of key-value pairs
  - No predefined schema
  - Keys are always strings
  - Values can be any (supported) data type
  - Values can **also** be an array
  - Values can **also** be a document

mongoDB

# Benefits of document model ..?

# Flexibility

- Each document can have different fields
- No need of long migrations, easier to be agile
- Common structure enforced at application level

# Arrays

- Documents can have field with array values
- **Ability to query and index array elements**
- We can model relationships with no need of different tables or collections

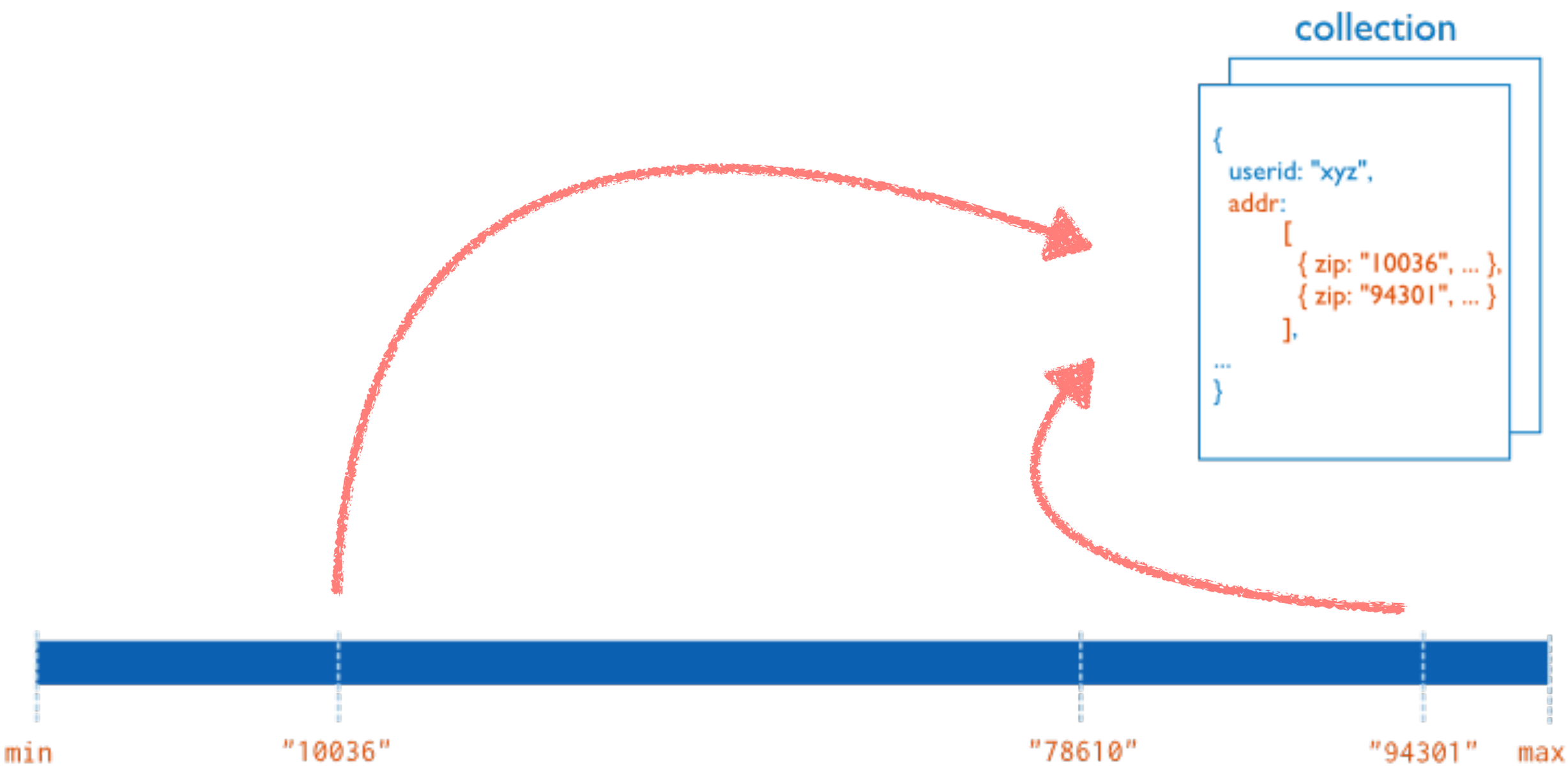
mongoDB

# Embedded documents

- Documents can have field with document values
- **Ability to query and index nested documents**
- Semantic closer to Object Oriented Programming

mongoDB

# Indexing an array of documents




{ "addr.zip": 1 } Index

mongoDB

Relational Schema Design

Focus on data storage

Document Schema Design

Focus on data usage

# SCHEMA DESIGN IS AN ART

https://www.flickr.com/photos/76377775@N05/11098637655/

# Implementing Relations

https://www.flickr.com/photos/ravages/2831688538

# A task tracking app

# Requirement #1

*"We need to store user information like name, email and their addresses... yes they can have more than one."*

— Bill, a *project manager, contemporary*

mongoDB

# Relational

| id | name | email | title |
|---|---|---|---|
| 1 | Kate | kate.powell@somedomain.c | Regional Manager |

| id | street | city | user_id |
|---|---|---|---|
| 1 | 123 Sesame Street | Boston | 1 |
| 2 | 123 Evergreen Street | New York | 1 |

mongoDB

# Let's use the document model

```
> db.user.findOne( { email: "kate.powell@somedomain.com"} )
{
    _id: 1,
    name: "Kate Powell",
    email: "kate.powell@somedomain.com",
    title: "Regional Manager",
    addresses: [
        { street: "123 Sesame St", city: "Boston" },
        { street: "123 Evergreen St", city: "New York" }
    ]
}
```

# Requirement #2

*"We have to be able to store tasks, assign them to users and track their progress..."*

— Bill, a *project manager, contemporary*

mongoDB

# Embedding tasks

```
> db.user.findOne( { email: "kate.powell@somedomain.com"} )
{
    name: "Kate Powell",
    // ... previous fields
    tasks: [
        {
            summary: "Contact sellers",
            description: "Contact agents to specify our needs
                and time constraints",
            due_date: ISODate("2014-08-25T08:37:50.465Z"),
            status: "NOT_STARTED"
        },
        { // another task }
    ]
}
```

# Embedding tasks

- Tasks are unbounded items: initially we do not know how many tasks we are going to have
- A user along time can end with thousands of tasks
- **Maximum document size in MongoDB: 16 MB !**
- It is harder to access task information without a user context

mongoDB

# Referencing tasks

```
> db.user.findOne({_id: 1})
{
    _id: 1,
    name: "Kate Powell",
    email: "kate.powell@...",
    title: "Regional Manager",
    addresses: [
        { // address 1 },
        { // address 2 }
    ]
}
```

```
> db.task.findOne({user_id: 1})
{
    _id: 5,
    summary: "Contact sellers",
    description: "Contact agents
        to specify our ...",
    due_date: ISODate(),
    status: "NOT_STARTED",
    user_id: 1
}
```

# Referencing tasks

- Tasks are unbounded items and our schema supports that
- Application level joins
- Remember to create proper indexes (e.g. user_id)

mongoDB

# Embedding

# vs

# Referencing

# One-to-many relations

- Embed when you have a *few* number of items on ‘many' side
- Embed when you have some level of control on the number of items on ‘many' side
- Reference when you cannot control the number of items on the 'many' side
- Reference when you need to access to ‘many' side items without parent entity scope

mongoDB

# Many-to-many relations

- These can be implemented with two one-to-many relations with the same considerations

# RECIPE #1

# USE EMBEDDING FOR ONE-TO-FEW RELATIONS

# RECIPE #2

# USE REFERENCING FOR ONE-TO-MANY RELATIONS

# Working with arrays

https://www.flickr.com/photos/kishjar/10747531785

# Arrays are great!

# List of sorted elements

```
> db.numbers.insert({
    _id: "even",
    values: [0, 2, 4, 6, 8]
});

> db.numbers.insert({
    _id: "odd",
    values: [1, 3, 5, 7, 9]
});
```

# Access based on position

```
db.numbers.find({_id: "even"}, {values: {$slice: [2, 3]}})
{
    _id: "even",
    values: [4, 6, 8]
}
```

```
db.numbers.find({_id: "odd"}, {values: {$slice: -2}})
{
    _id: "odd",
    values: [7, 9]
}
```

# Access based on values

```
// is number 2 even or odd?
> db.numbers.find( { values : 2 } )
{
    _id: "even",
    values: [0, 2, 4, 6, 8]
}
```

# Like sorted sets

```
> db.numbers.find( { _id: "even" } )
{
    _id: "even",
    values: [0, 2, 4, 6, 8]
}

> db.numbers.update(
    { _id: "even"},
    { $addToSet: { values: 10  } }
);
```

Several times...!

```
> db.numbers.find( { _id: "even" } )
{
    _id: "even",
    values: [0, 2, 4, 6, 8, 10]
}
```

# Array update operators

- `pop`
- `push`
- `pull`
- `pullAll`

mongoDB

# But...

# Storage

DocA

DocB

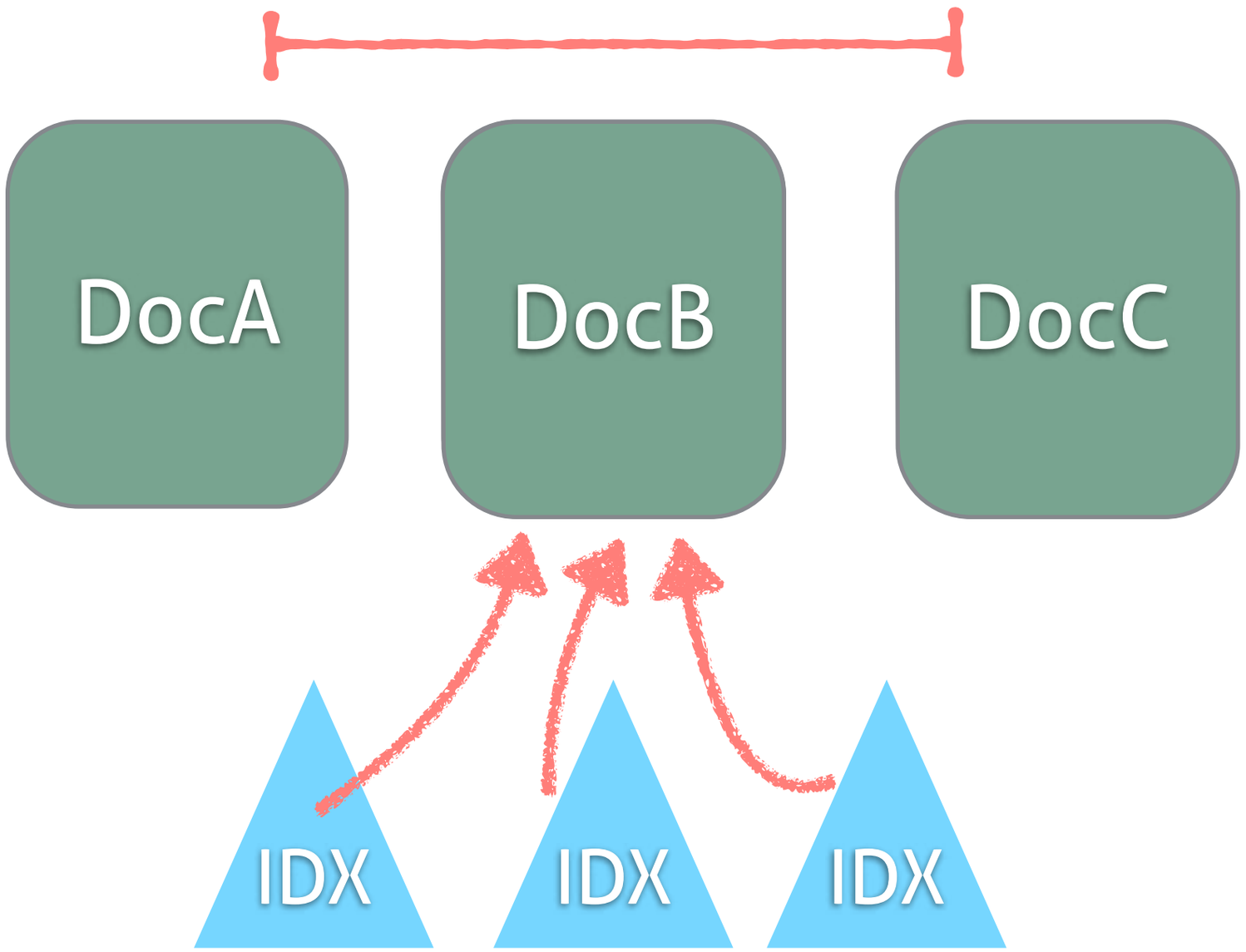
```
{
 _id: 1,
 name: "Nike Pump Air 180",
 tags: ["sports", "running"]
}
```

DocC

```
db.inventory.update(
  { _id: 1},
  { $push: { tags: "shoes" } }
)
```

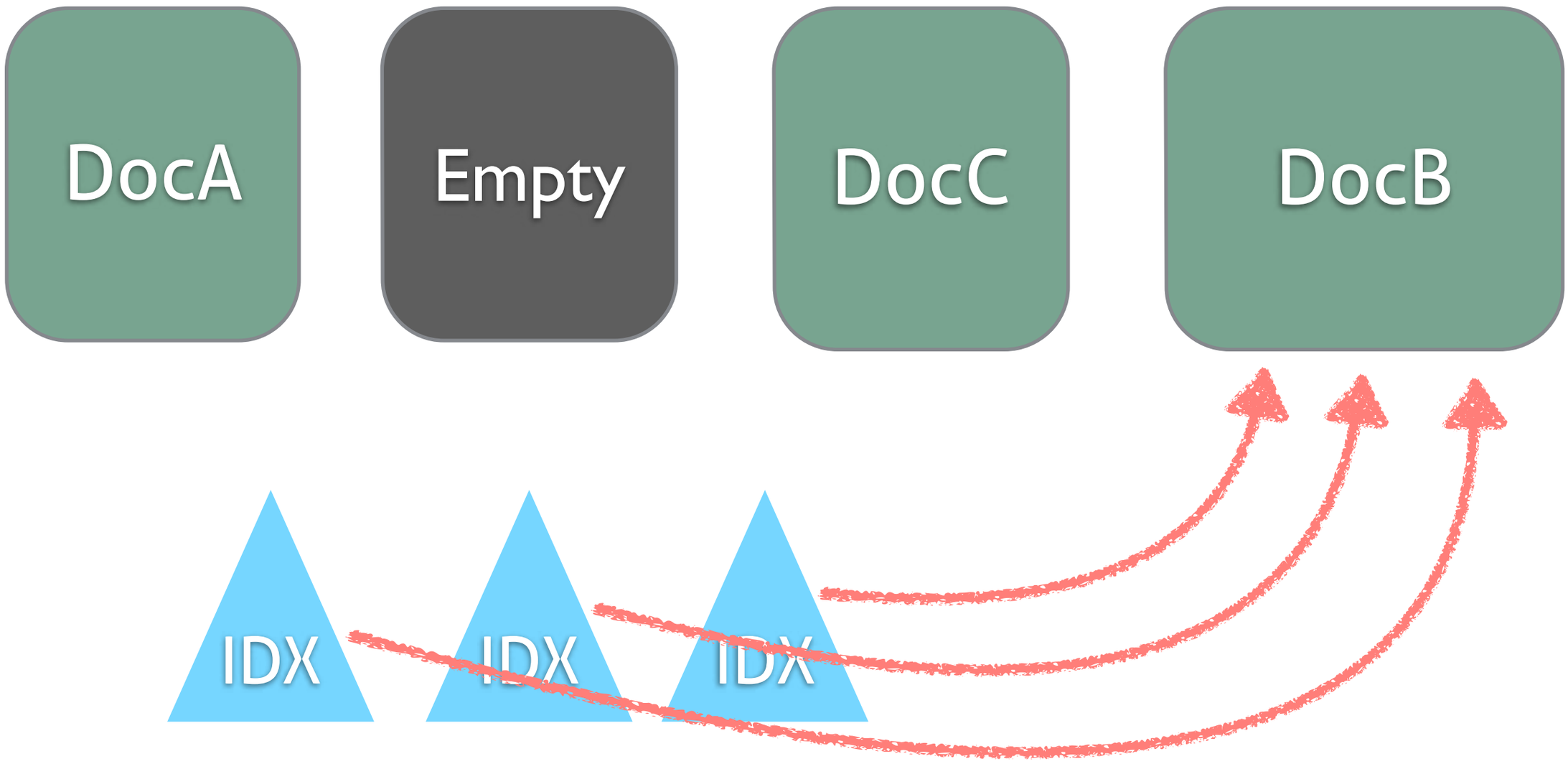
mongoDB

# Storage




mongoDB

# Storage




mongoDB

# Why is expensive to move a doc?

1. We need to write the document in another location (**$$**)
2. We need to mark the original position as free for new documents (**$**)
3. We need to update all those index entries pointing to the moved document to the new location (**$$$**)

mongoDB

# Considerations with arrays

- Limited number of items
- Avoid document movements
  - Document movements can be delayed with padding factor
  - Document movements can be mitigated with pre-allocation

mongoDB

# RECIPE #3

# AVOID EMBEDDING LARGE ARRAYS

# RECIPE #4

# USE DATA MODELS THAT MINIMIZE THE NEED FOR DOCUMENT GROWTH

# Denormalization

https://www.flickr.com/photos/ross_strachan/5146307757

# Denormalization

*"...is the process of attempting to optimise the read performance of a database by adding redundant data ..."*

– Wikipedia

mongoDB

# Products and comments

```
> db.product.find( { _id: 1 } )
{
    _id: 1,
    name: "Nike Pump Air Force 180",
    tags: ["sports", "running"]
}
```

```
> db.comment.find( { product_id: 1 } )
{ score: 5, user: "user1", text: "Awesome shoes"  }
{ score: 2, user: "user2", text: "Not for me.." }
```

# Denormalizing

```
> db.product.find({_id: 1})
{
    _id: 1,
    name: "Nike Pump Air Force 180",
    tags: ["sports", “running"],
    comments: [
        { user: "user1", text: "Awesome shoes" },
        { user: "user2", text: "Not for me.." }
    ]
}

> db.comment.find({product_id: 1})
{ score: 5, user: "user1", text: "Awesome shoes"  }
{ score: 2, user: "user2", text: "Not for me.."}
```

# RECIPE #5

# DENORMALIZE TO AVOID APP-LEVEL JOINS

# RECIPE #6

# DENORMALIZE ONLY WHEN YOU HAVE A HIGH READ TO WRITE RATIO


# Bucketing

https://www.flickr.com/photos/97608671@N02/13558864555/

# What's the idea?

- Reduce number of documents to be retrieved
- Less documents to retrieve means less disk seeks
- Using arrays we can store more than one entity per document
- We group things that are accessed together

mongoDB

# An example

Comments are showed in buckets of 2 comments

A 'read more' button loads next 2 comments


mongoDB

# Bucketing comments

```
> db.comments.find({post_id: 123})
            .sort({sequence: -1})
            .limit(1)
{
   _id: 1,
   post_id: 123,
   sequence: 8, // this acts as a page number
   comments: [
      {user: user1@somedomain.com, text: "Awesome shoes.."},
      {user: user2@somedomain.com, text: "Not for me..”}
   ] // we store two comments per doc, fixed size bucket
}
```

# RECIPE #7

# USE BUCKETING TO STORE THINGS THAT ARE GOING TO BE ACCESSED AS A GROUP

# 谢谢

mongoDB