INTRODUCTION TO PATTERNS

Lecturer: Trần Thế Trung

1. Attribute Pattern;
2. Extended Reference Pattern;
3. Subset Pattern;
4. Computed Pattern;
5. Bucket Pattern;
6. Schema Versioning Pattern;
7. Tree Patterns;
8. Polymorphic Pattern;
9. Other Patterns.

1. Attribute Pattern

Polymorphic, is one of the most frequent schema design patterns used in MongoDB.

Polymorphic is when you put different products, like these three examples, in one collection without going through relational acrobatics.

1. Attribute Pattern

Our products should have an identification like manufacturer, brand, sub-brand, enterprise that are common across the majority of products.

Products' additional fields that are common across many products, like color and size either these values may have different units and means different things for the different products.

For example the size of a beverage made in the US maybe measured as ounces, while the same drink in Europe will be measured in milliliters. As for charger, well, the size is measured according to its three dimensions. For the size of a Cherry Coke six-pack, we would say 12 ounces for a single can, six times 12 ounces, or 72 ounces to count the full six-pack.

Then there is the third list of fields, the set of fields that are not going to exist in all the products. You may not even know where they are in advance. They may exist in the new description that your supplier is providing you.

1. Attribute Pattern

```
{
 "description": "Cherry Coke 6-pack",
 "manufacturer": "Coca-Cola",
 "brand": "Coke",
 "sub_brand": "Cherry Coke",
 "price": 5.99,
 ...
 "color": "red",
 "size": "12 ounces",
 ...
 "container": "can",
 "sweetener": "sugar"
}
```


```
{
 "description": "Evian 500ml",
 "manufacturer": "Danone",
 "brand": "Evian",
 "price": 1.99,
 ...
 "size": "500 ml",
 ...
 "container": "plastic bottle"
}
```


```
{
 "description": "MongoDB charger",
 "manufacturer": "China",
 "brand": "MongoDB",
 "sub_brand": "University",
 "price": 0.00,
 ...
 "color": "black",
 "size": "100 x 70 x 10 mm"
 ...
 "input": "5V/1300 mA"
 "output": "5V/1A"
 "capacity": "4200 mAh"
}
```

1. Attribute Pattern

Optimized queries

```
db.products.find({"capacity": { $gt: 4000 }})

    => index on "capacity"

db.products.find({"output": "5V"})

    => index on "output"

=> May need a lot of indexes
```

1. Attribute Pattern

For this case you want to use the attribute pattern.

To use the attribute pattern you start by identifying the list of fields you want to transpose. Here we transpose the fields input, output, and capacity.

Then for each field in associated value, we create that pair. The name of the keys for those pairs do not matter. Only for consistency, let's use K for key and V for value, as some of our aggregation functions do. Under the field name K, we put the name of the original field as the value.

For the first one, the field was named "input," so that became the value for K. Then the value for input was five volts or 1,300 milliamps, so this is the value for the field V.

Repeating the same thing for the original field's output and capacity, we get three documents, each adding a K and a V in them.

1. Attribute Pattern

Using the Attribute pattern

```
{
  "manufacturer": "China",
  "brand": "MongoDB",
  "sub_brand": "University",
  "price": 0.00,
  ...
  "color": "black",
  "size": "100 x 70 x 10 mm"
  ...
  "input": "5V/1300 mA"
  "output": "5V/1A"
  "capacity": "4200 mAh"
}
```

```
{
  "manufacturer": "China",
  "brand": "MongoDB",
  "sub_brand": "University",
  "price": 0.00,
  ...
  "color": "black",
  "size": "100 x 70 x 10 mm"
  ...
  "add_specs": [
    { "k": "input",    "v": "5V/1300 mA" },
    { "k": "output"    "v": "5V/1A" },
    { "k": "capacity", "v": 4200, "u": "mAh" }
  ]
}
```

1. Attribute Pattern

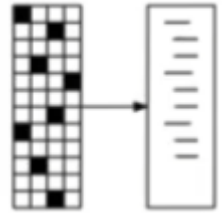
Fields that share Common Characteristics

```
{
  "title": "Dunkirk",
  ...
  "release_USA": "2017/07/23",
  "release_Mexico": "2017/08/01",
  "release_France": "2017/08/01",
  "release_Festival_San_Jose":
   "2017/07/22"
}
```

```
{
  "title:" "Dunkirk",
  ...
  "releases": [ {
    { "k": "release_USA",
      "v": "2017/07/23" },
    { "k": "release_Mexico",
      "v": "2017/08/01" },
    { "k": "release_France",
      "v": "2017/08/01" },
    { "k": "release_Festival_San_Jose",
      "v": "2017/07/22" }
  ]
}
```

db.movies.find({ “releases.v” : { $gte: “2017/07”, $lt: “2017/08 } })

1. Attribute Pattern



Problem

- Lots of similar fields
- Want to search across many fields at once
- Field present in only a small subset of documents

Solution

- Break the field/value into a sub-document with:
  fieldA : *field*
  fieldB : *value*
- Example:
  { color: “blue”; size: “large” }
  { [{ k : “color”, v : “blue” },
  { k : “size”, v : “large” }] }

Use Cases Example

- Characteristics of a product;
- Set of fields all having same value type
  List of dates

Benefits and Trade – Offs

- ✓ Easier to index;
- ✓ Allow for non-deterministic field names;
- ✓ Ability to qualify the relationship of the original field and value

1. Attribute Pattern

Summary

- Orthogonal Pattern to Polymorphism;

- Add organization for:
  - common characteristics;
  - rare/unpredictable fields.

- Reduces number of indexes;

- Transpose keys/values as:
  - Array of sub-documents of form:
  - { k : “key”, v : “value” }

1. Attribute Pattern - Lab

User Story: The museum we work at has grown from a local attraction to one that is seen as saving very popular items. For this reason, other museums in the World have started exchanging pieces of art with our museum.

Our database was tracking if our pieces are on display and where they are in the museum.

To track the pieces we started exchanging with other museum, we added an array called events, in which we created an entry for each date a piece was loaned and the museum it was loaned to.

1. Attribute Pattern - Lab

Problem: This schema has a problem where each time we start a exchange with a new museum, we must then add a new index. For example, when we started working with The Prado in Madrid, we added this index: { "events.prado" : 1 }.

Please review the Problem Schema and the Problem Document to help understand the old schema and how we need to change the schema to apply the pattern.

1. Attribute Pattern - Lab

```
Problem-Schema
    {
      "_id": "<objectId>",
      "title": "<string>",
      "artist": "<string>",
      "date_acquisition": "<date>",
      "location": "<string>",
      "on_display": "<bool>",
      "in_house": "<bool>",
      "events": [{
        "moma": "<date>",
        "louvres": "<date>"
      }]
    }
```

1. Attribute Pattern - Lab

```
Problem-document
    {
      "_id": ObjectId("5c5348f5be09bedd4f196f18"),
      "title": "Cookies in the sky",
      "artist": "Michelle Vinci",
      "date_acquisition": ISODate("2017-12-25T00:00:00.000Z"),
      "location": "Blue Room, 20A",
      "on_display": false,
      "in_house": false,
      "events": [{
        "moma": ISODate("2019-01-31T00:00:00.000Z"),
        "louvres": ISODate("2020-01-01T00:00:00.000Z")
      }]
    }
```

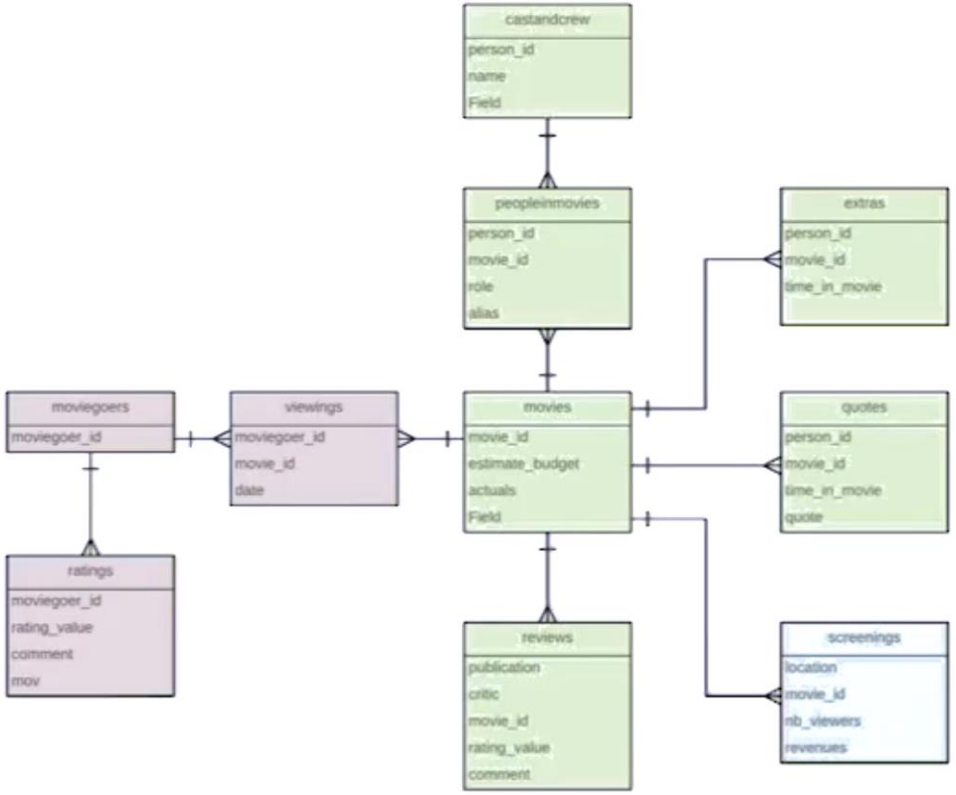
1. Attribute Pattern - Lab

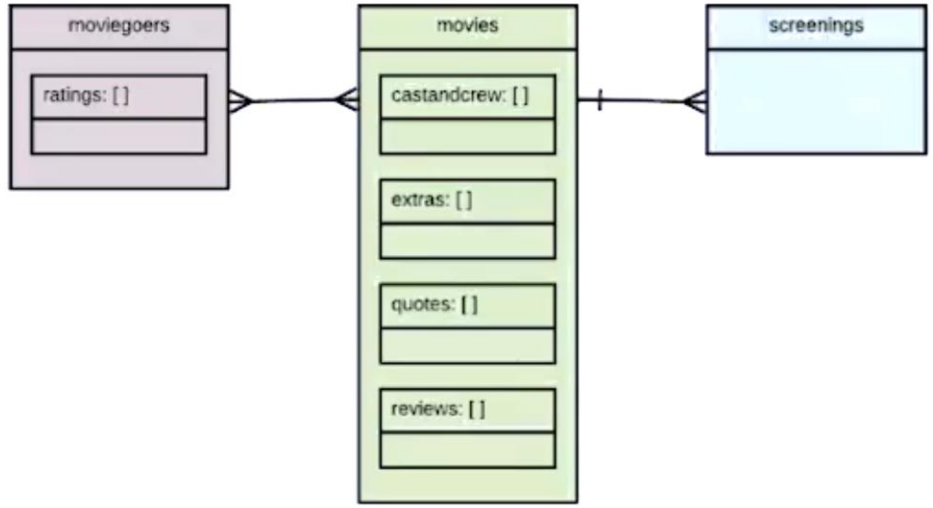
```
Pattern-Attribute
   {
     "_id": "<objectId>",
     "title": "<string>",
     "artist": "<string>",
     "date_acquisition": "<date>",
     "location": "<string>",
     "on_display": "<bool>",
     "in_house": "<bool>",
     "events": [{
       "k": "<string>",
       "v": "<date>"
     }]
   }
```

1. Attribute Pattern - Lab

```
Answer-document
   {
     "_id": ObjectId("5c5348f5be09bedd4f196f18"),
     "title": "Cookies in the sky",
     "artist": "Michelle Vinci",
     "location": "Blue Room, 20A",
     "on_display": false,
     "in_house": false,
     "events": [
      {"k": "date_acquisition", "v": ISODate("2017-12-25T00:00:00.000Z")},
      {"k": "moma", "v": ISODate("2019-01-31T00:00:00.000Z")},
      {"k": "louvres", "v": ISODate("2020-01-01T00:00:00.000Z")}
       ]
   }
```

2. Extended Reference Pattern





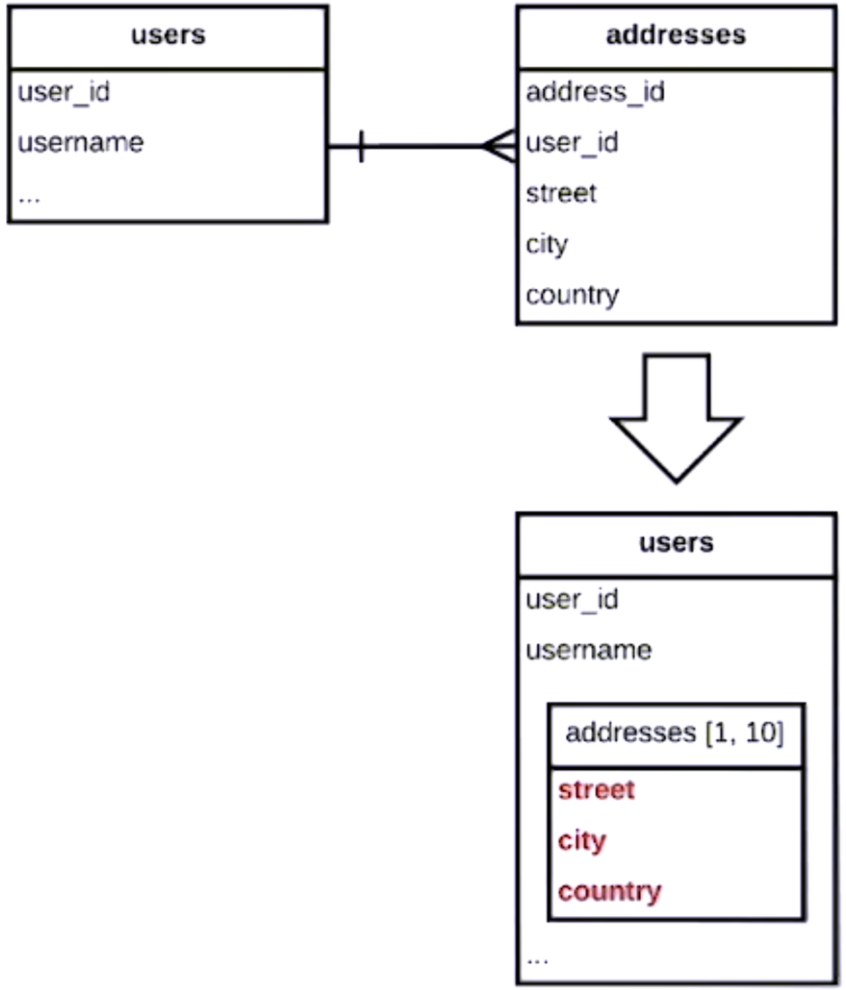
2. Extended Reference Pattern

How joins are performed in MongoDB

A. Application side;

B. Lookups:
   - $lookup;
   - $graphLookup.

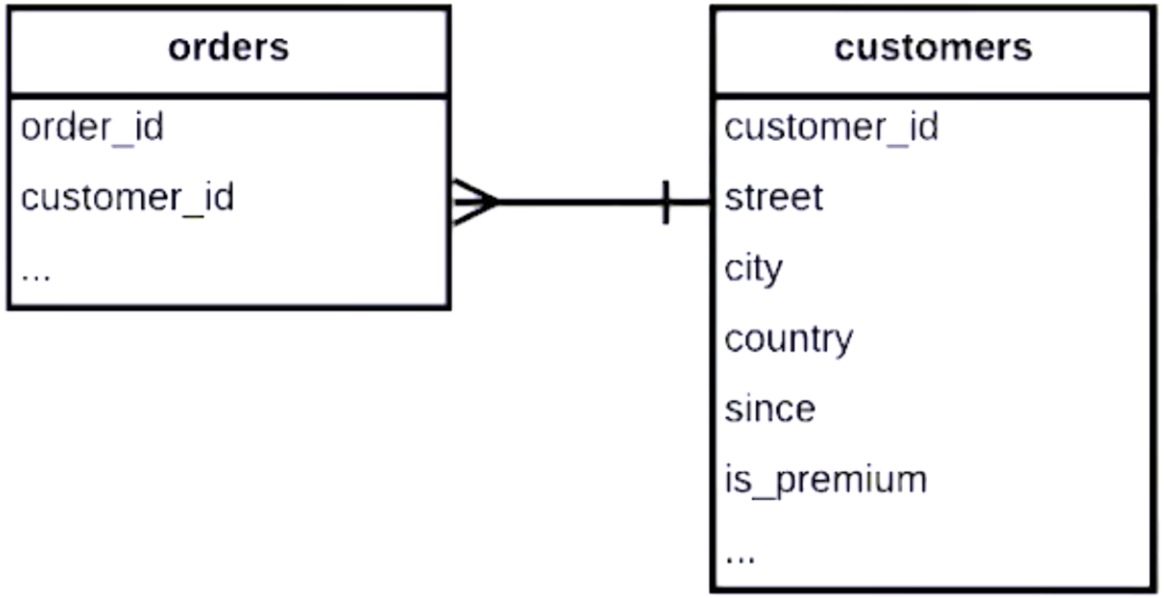
C. Avoid a join by embedding the joined table.



2. Extended Reference Pattern

Extended Reference for Many-to-One Relationships

- focus of queries is on the many-side



2. Extended Reference Pattern

Extended Reference for Many-to-One Relationships

Let's say that our applications focus is on order management and fulfillment.

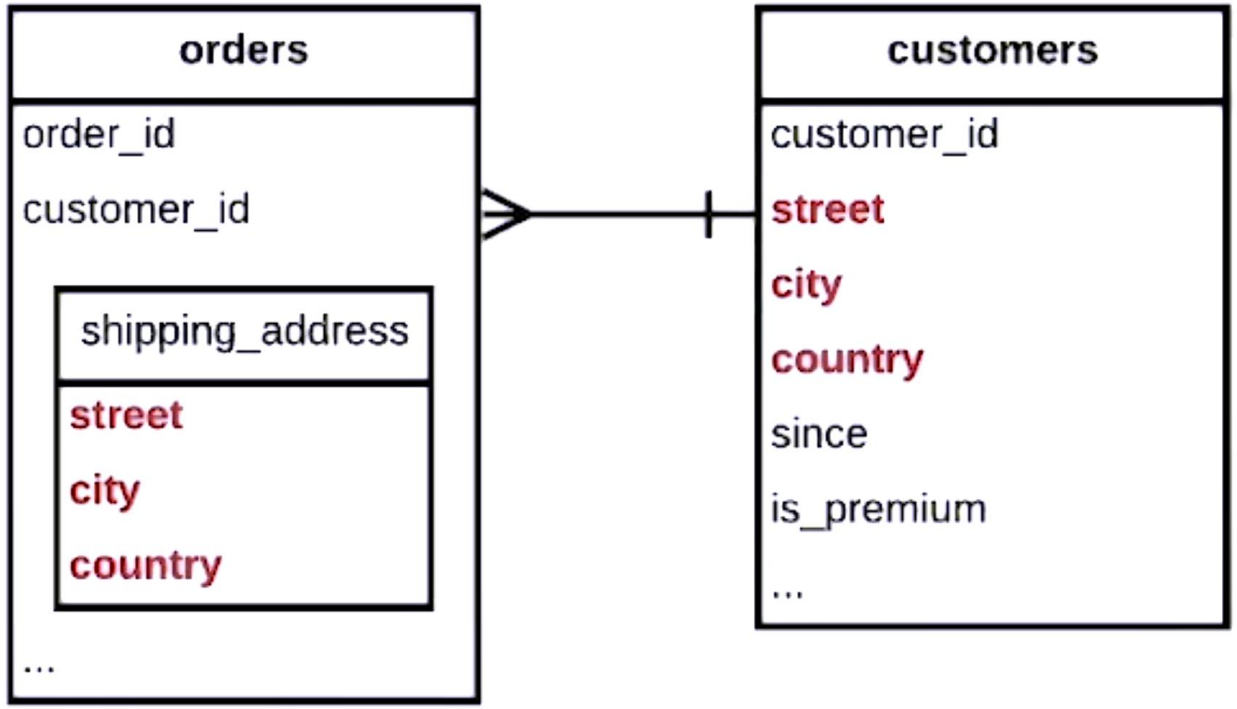
And we will query for specific orders way more often than query all the orders for a given customer. In other words, the center of attention is more often the order for this type of query.

If we want to embed the information from one side, for example, the address of the customer, into the order, we are duplicating this info in all orders.

2. Extended Reference Pattern

Extended Reference for Many-to-One Relationships

- Embed the “One” side, of a “One-to-Many” relationship. Into the “many side”;
- Only the part that we need to join often



2. Extended Reference Pattern

Extended Reference for Many-to-One Relationships

The preferred way to do this duplication, which we identify as our extended reference pattern, is only to copy the fields you need to access frequently, leaving the rest of the information in the source collection.

Basically, instead of adding a simple reference from the document in the invoice collection to this one in the customer collection, we built an extended reference, meaning the reference is rich enough, that we will not need to perform the join most of the time.

Every time we talk about duplicating data, it is worth understanding the consequences.

2. Extended Reference Pattern

Managing Duplication

1. Minimize it
   a. Select fields that do not change often;
   b. Bring only the fields you need to avoid joins.

2. After a source is update:
   a. What are the extended references to changed;
   b. When should the extended references be updated.

3. Duplication may be better than a unique reference

2. Extended Reference Pattern

Managing Duplication: the first thing to understand is how to minimize duplication.

The extended reference pattern will work best if you select fields that do not change often. For example, the user ID of a person can be complemented by his name, as people rarely change their name. Also, only bring the fields you need.

When we look up an orders, we may want to see the list of products with their supplier name. But there is little reason to show the phone number of the supplier at this point. If we want additional information, we can solve the supplier collection.

Then, when the source field is updated, identify what should be change, meaning the list of extended references, and the when should they be changed. For example, it may be necessary to update them right away. However, sometimes it is OK to leave the data alone and update it later in a batch, when you have available resources.

For example, changing the ranking of my best-selling products does not require me to instantly update all the products that spell out the rank of the product on their page.

2. Extended Reference Pattern

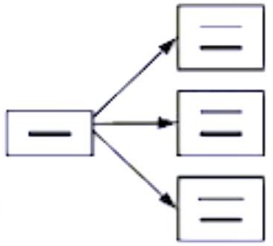
Interestingly enough, the example we use with invoices and addresses is a case where duplication is not a problem, but rather the right solution.

When the invoice was created, the customer may have lived in one location, and they have moved since. If we were to keep the last address of the customer in a reference in the invoice, we would be pointing to the new address, not where the products of this invoice were shipped.

In this case, we want the invoice to keep pointing to the old address. The characteristics of the invoice that make it work well are the fact that the invoice is created at a given time in the timeline, and the invoice remains static over time.

Any other entity sharing those characters is likely to work the same way, in the sense that duplication is not an issue, but a good thing.

2. Extended Reference Pattern



1. Problem

- Too many repetitive joins

2. Solution

- Identify fields on the lookup side;
- Bring those fields into the main object.

3. Use Cases Examples

- Catalog;
- Mobile Applications;
- Real-Time Analytics.

4. Benefits and Trade-Offs

✓ Faster reads;

✓ Reduce number of joins and lookups;

☒ May introduce lots of duplication if extended reference contains fields that mutate a lot.

2. Extended Reference Pattern

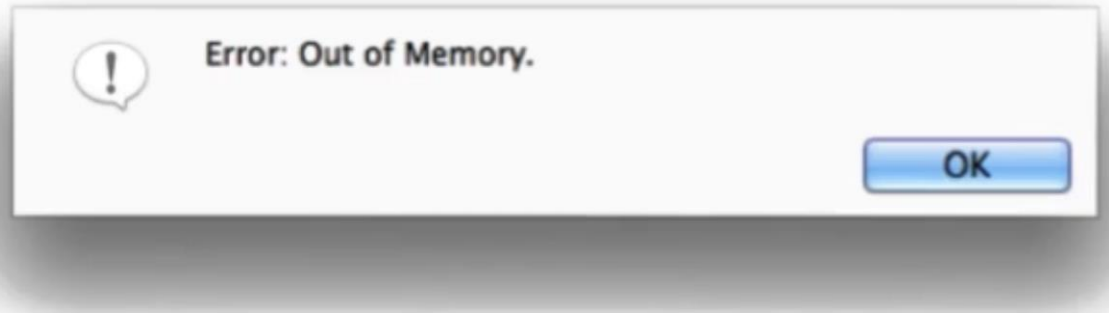
The problem the extended reference pattern addresses is avoiding joining too many pieces of data at query time. If the query is frequent enough and generates a considerable amount of lookups, pre-joining the data can be done by applying this pattern.

The solution is to identify the fields you are interested in on the looked-up side and make a copy of those fields in the main object.

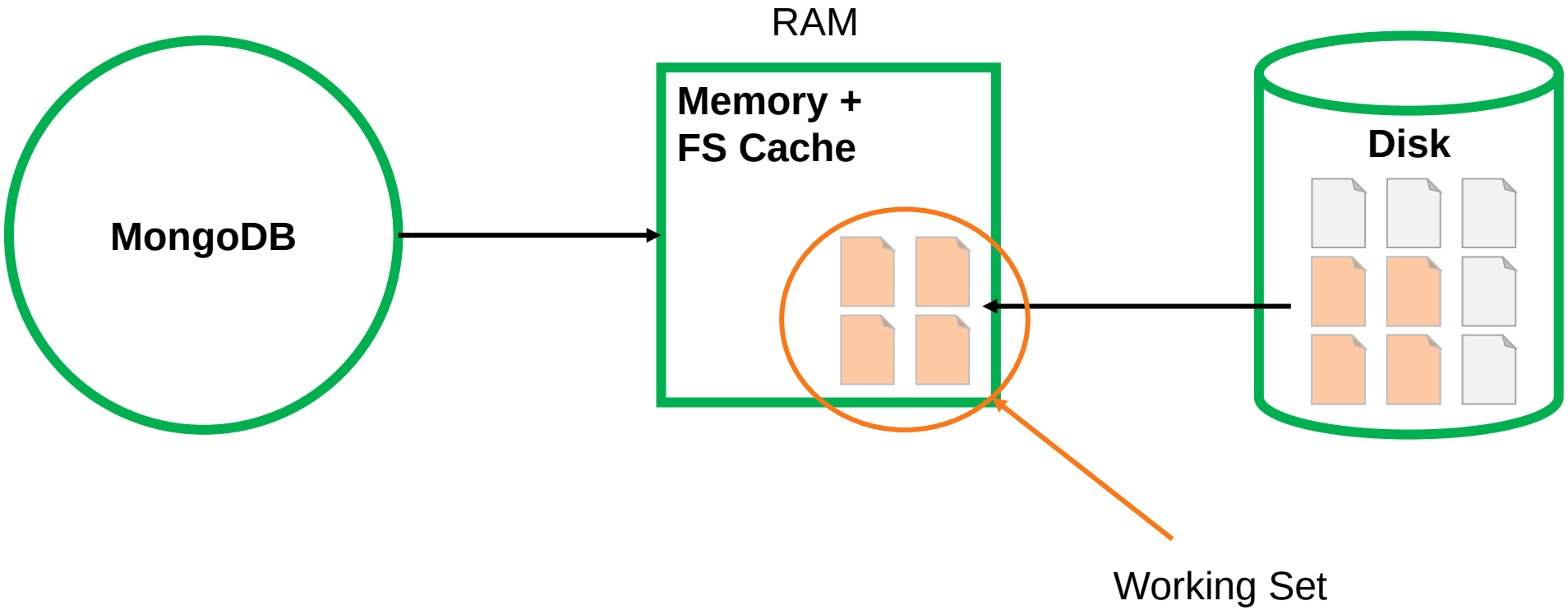
You will see this pattern often used in catalogs, mobile applications, and real-time analytics.

The common thread here is to reduce the latency of your read operations avoid round trips or avoid touching too many pieces of data. You will get faster reads, due to the reduced number of joints and lookups. The price you will pay for the improvement in performance is the fact that you may have to manage a fair amount of duplication, especially if you embed Many-to-One relationships, where the fields change or mutate a lot.

3. Subset Pattern



3. Subset Pattern



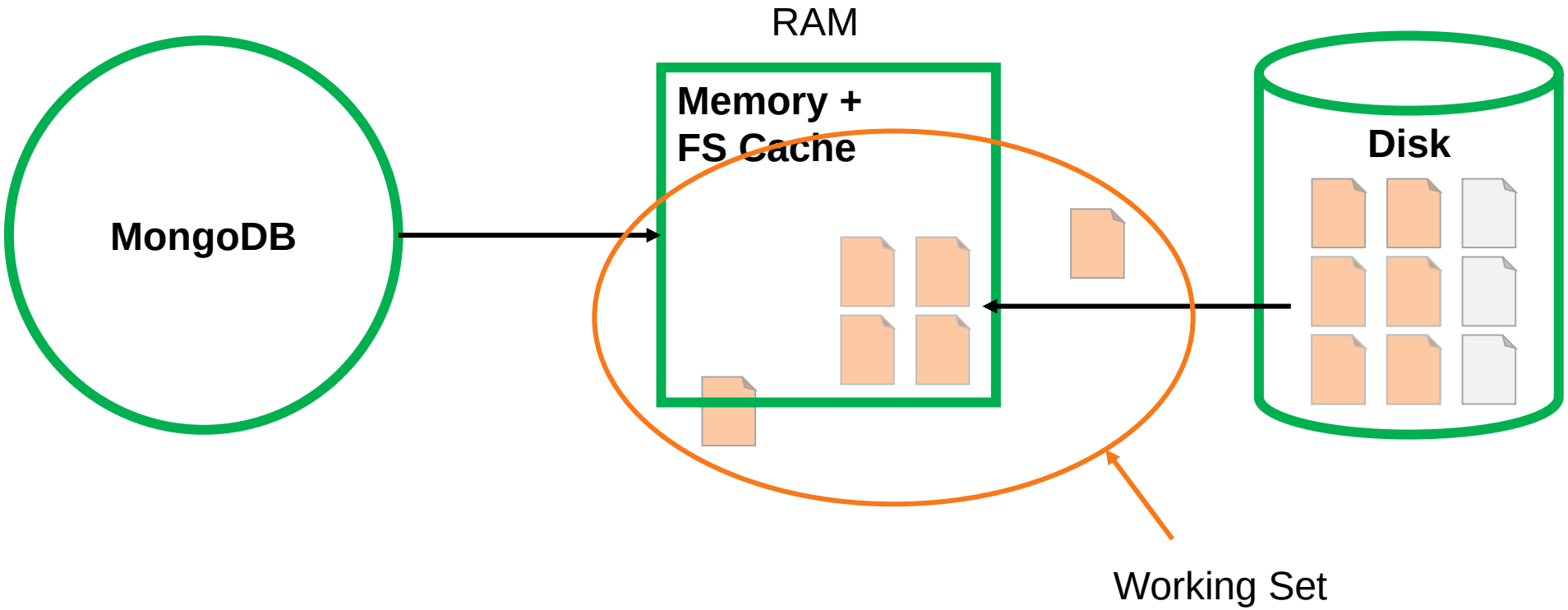
3. Subset Pattern

MongoDB tries to optimize the use of RAM by pulling in memory only the document that it needs from the disk through the RAM. When there is no more memory available, it evicts pages that contains the document it doesn't need anymore to make room for more document it needs to process at the moment.

This mechanism allows to have the hardware configuration that does less RAM than the total size of the data on disk. As long as the size of your working set fits in RAM, you get good performance. The working set refers to the amount of space taken by the documents and the portions of indexes that are frequently accessed.

Either if you get into a situation where the working set is larger than RAM in this picture, the four red documents that are needed all the time, may not fit in memory. So the server finds itself often dropping documents. It will soon have to fetch back from the disk. This process, of constantly ejecting documents that should stay in memory, is pretty bad.

3. Subset Pattern



3. Subset Pattern

Either if you get into a situation where the working set is larger than RAM in this picture, the four red documents that are needed all the time, may not fit in memory.

So the server finds itself often dropping documents. It will soon have to fetch back from the disk. This process, of constantly ejecting documents that should stay in memory, is pretty bad.

3. Subset Pattern

Working set is too big

A. Add RAM;

B. Scale with Sharding;

C. Reduce the size of the **working set**.

3. Subset Pattern

```
1   _id: ObjectId("573a1396f29313caabce557f")                                   ObjectId
2   title : "The Godfather: Part II "                                           String
3   year : 1974                                                                 Int32
4   runtime : 200                                                               Int32
5   released : 1974-12-19 16:00:00.000                                          Date
6 › cast : Array                                                                Array
7   metacritic : 80                                                             Int32
8   poster : "http://ia.media-imdb.com/images/M/MV5BNDc2NTM3MzU1Nl5BMl5BanBnXkFtZTcwMTA5Mzg3OA@@._V1_SX300.jpg "   String
9   : "The early life and career of Vito Corleone in 1920s New York is portrayed while his son, Michael, expands and tightens his g…"  String
10  : "The continuing saga of the Corleone crime family tells the story of a young Vito Corleone growing up in Sicily and in 1910s …"  String
11  awards : "Won 6 Oscars. Another 13 wins & 16 nominations. "                 String
12  lastupdated : "2015-09-02 00:15:27.537000000 "                              String
13  type : "movie "                                                             String
14 › languages : Array                                                          Array
15 › directors : Array                                                          Array
16 › writers : Array                                                            Array
17 › imdb : Object                                                              Object
18 › countries : Array                                                          Array
19  rated : "R "                                                                String
20 › genres : Array                                                             Array
21 › tomatoes : Object                                                          Object
22  num_mflix_comments : 2                                                      Int32
23 › comments : Array                                                           Array
24 › quotes : Array                                                             Array
25 › releases : Array                                                           Array
26 › reviews : Array                                                            Array
```

3. Subset Pattern

The key to that is breaking up huge documents, which we only need a fraction of it.

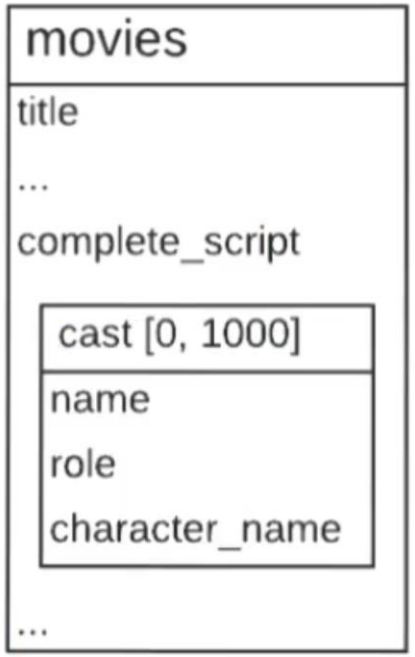
For Example, let's say we have a system that keeps a lot of movies in memory, and each of these movies is taking a fair amount of memory. Maybe there is some information that we don't need to use that often in those documents. For example, most of the time users want to see the top actors and the top reviews, rather than all of them.

As for the fields here at the bottom comments, quote, and release it's also unlikely that you need all of them, most of the time. We could keep only 20 of the cast members, the main actors, and also 20 of each of those comments, quotes, releases, and reviews.

The rest of the information can go into a separate collection.

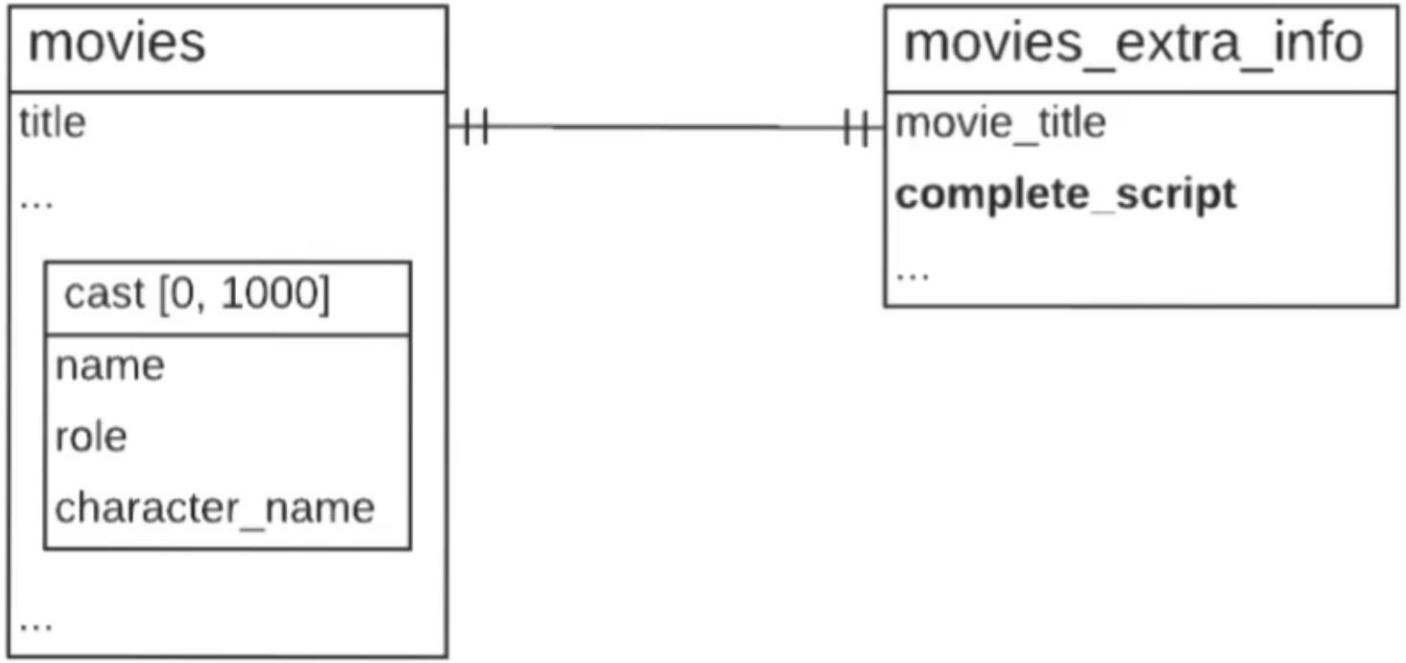
3. Subset Pattern

One document



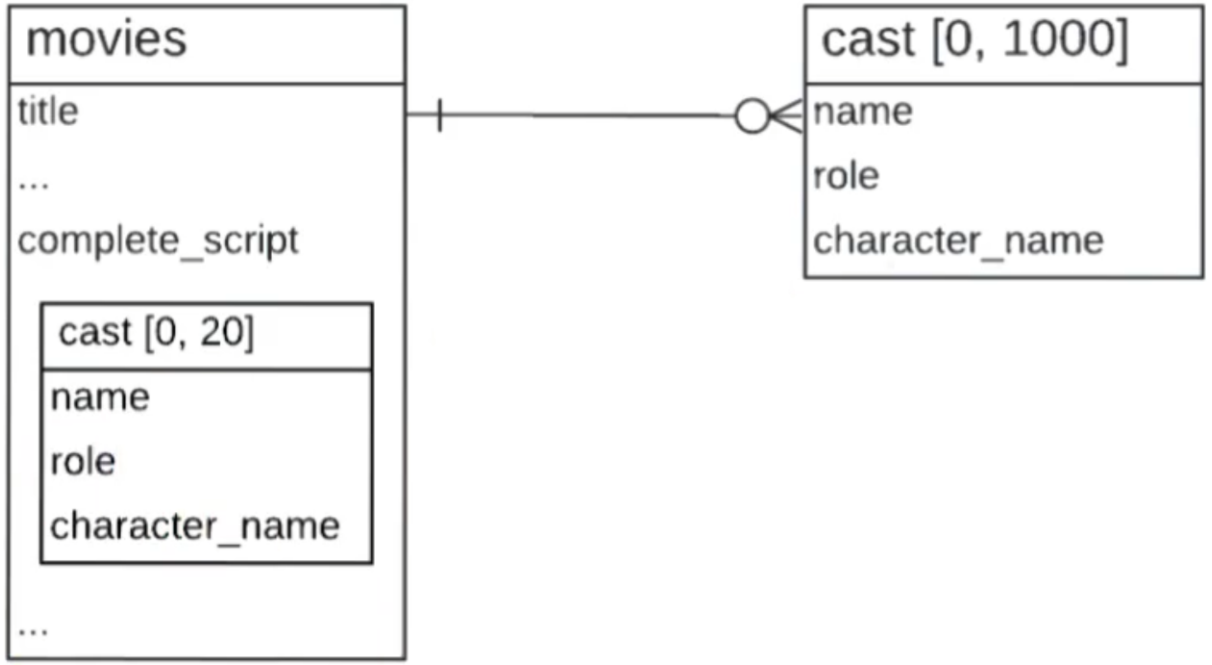
3. Subset Pattern

Moving some Fields with One-to-One Relationship



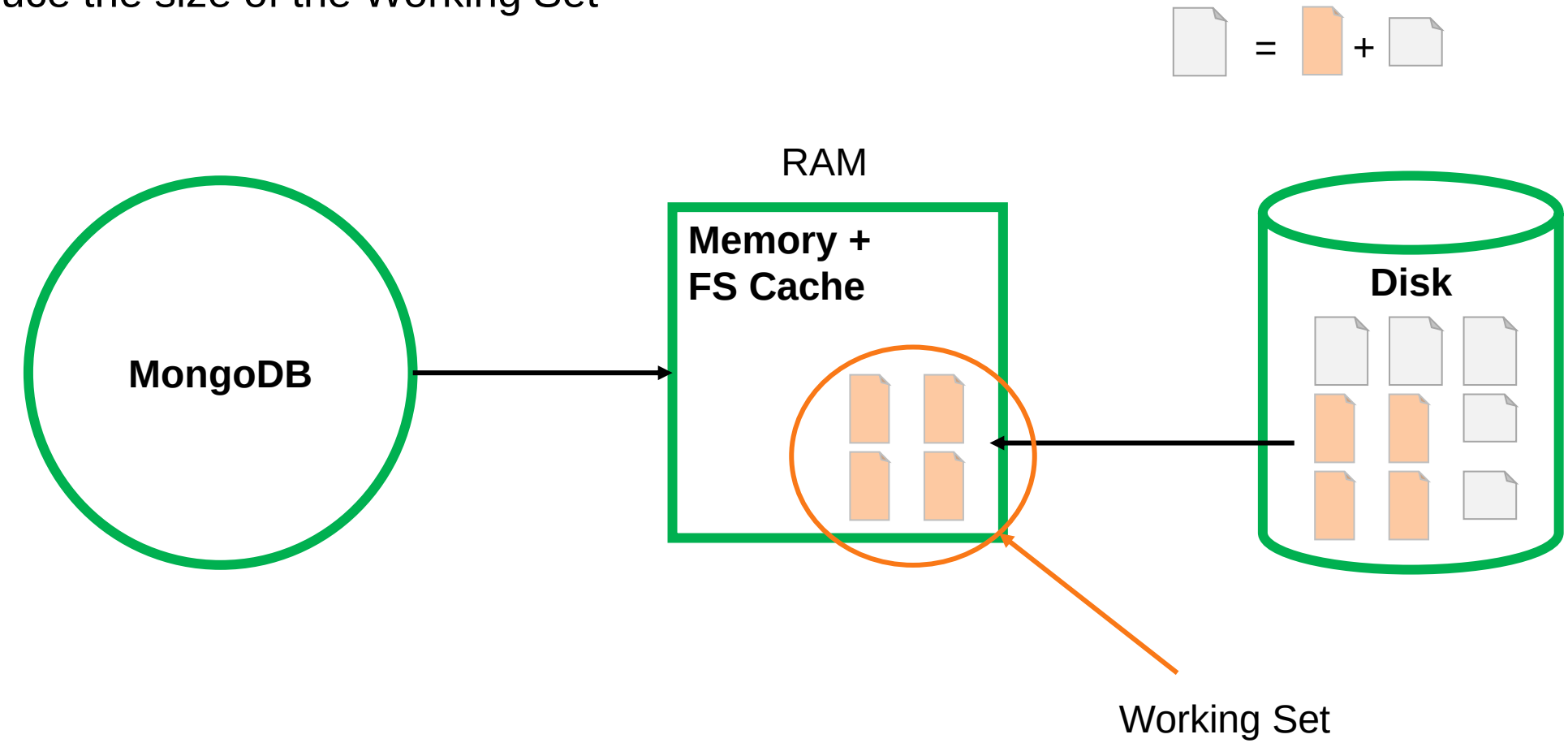
3. Subset Pattern

Moving some Fields with One-to-Many Relationship

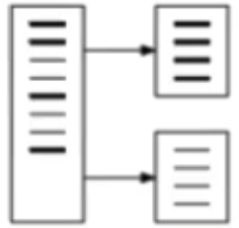


3. Subset Pattern

Reduce the size of the Working Set



3. Subset Pattern



Problem

- Working set is too big
- Lot of pages are evicted from memory
- A large part of documents is rarely needed

Solution

- Split the collection in 2 collections
  - Most used part of documents
  - Least used part of documents
- Duplicate part of a 1-N or N-N relationship that is often used in the most used side

Use case Examples

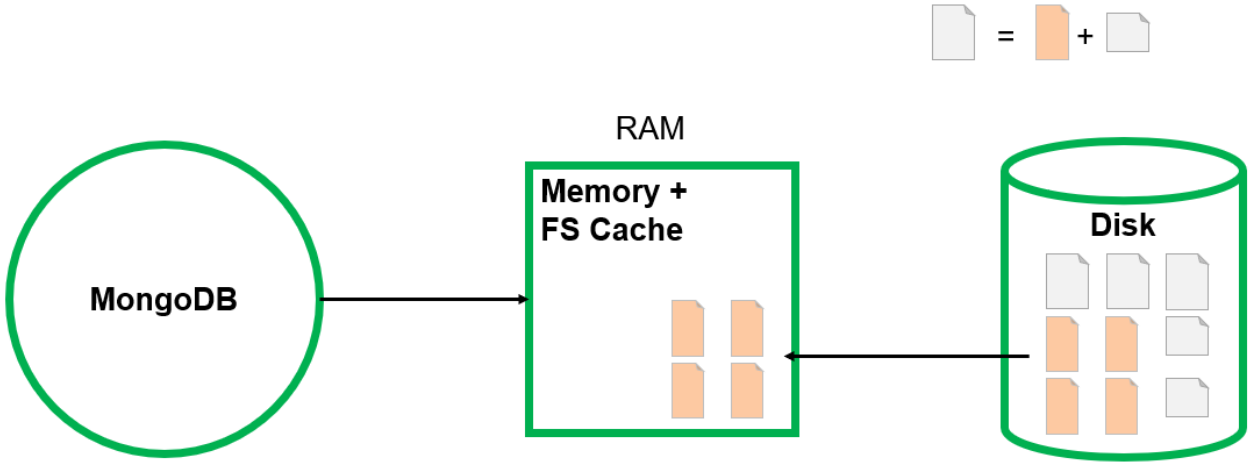
- List of reviews for a product
- List of comments on an article
- List of actors in a movie

Benefits and Trade-Offs

✓ Smaller working set, as often used documents are smaller;
✓ Shorter disk access for bringing in additional documents from the most used collection;
☒ More round trips to the server;
☒ A little more space used on disk.

3. Subset Pattern

- Reduces working set size;
- Split information as:
  - frequently needed;
  - rarely needed.



1. Subset Pattern - Lab

- **User Story**:
  - You are the lead developer for an online organic recycled clothing store.
  - Due to the growing number of environmentally-conscious consumers, our store's inventory has increased exponentially. We now also have an increasingly large pool of makers and suppliers.

- **Problem**:
  - We recently found that our shopping app is getting slower due to the fact that the frequently-used documents can no longer all fit in RAM. This is happening largely due to having all product reviews, questions, and specs stored in the same document, which grows in size as reviews and ratings keep coming in.
  - To resolve this issue, we want to reduce the amount of data immediately available to the user in the app and only load additional data when the user asks for it.

3. Subset Pattern - Lab

Problem-Schema

```
{
  "_id": "<objectId>",
  "item_code": "<string>",
  "name": "<string>",
  "maker_brand": "<string>",
  "price": "<decimal>",
  "description": "<string>",
  "materials": ["<string>"],
  "country": "<string>",
  "image": "<string>",
  "available_sizes": {
    "mens": ["<string>"],
    "womens": ["<string>"]
  },
  "package_weight_kg": "<decimal>",
  "average_rating": "<decimal>",
  "reviews": [{
    "author": "<string>",
    "text": "<string>",
    "rating": "<int>"
  }],
  "questions": [{
    "author": "<string>",
    "text": "<string>",
    "likes": "<int>"
  }],
  "stock_amount": "<int>",
  "maker_address": {
    "building_number": "<string>",
    "street_name": "<string>",
    "city": "<string>",
    "country": "<string>",
    "postal_code": "<string>"
  },
  "makers": ["<string>"]
}
```

3. Subset Pattern - Lab

Problem-document

```
{
  "_id": ObjectId("5c9be463f752ec6b191c3c7e"),
  "item_code": "AS45OPD",
  "name": "Recycled Kicks",
  "maker_brand": "Shoes From The Gutter",
  "price": 100.00,
  "description": "These amazing Kicks are made from recycled plastics and
  fabrics.They come in a variety of sizes and are completely unisex in design.
  If your feet don't like them within the first 30 days, we'll return your
  money no questions asked.
  ",
  "materials": [
    "recycled cotton",
    "recycled plastic",
    "recycled food waste",
  ],
  "country": "Russia",
  "image": "https:///www.shoesfromthegutter.com/kicks/AS45OPD.img",
  "available_sizes": {
      "mens": ["5", "6", "8", "8W", "10", "10W", "11", "11W", "12", "12W"],
      "womens": ["5", "6", "7", "8", "9", "10", "11", "12"]
  },
  "package_weight_kg": 2.00,
  "average_rating": 4.8,
  "reviews": [{
      "author": "i_love_kicks",
      "text": "best shoes ever! comfortable, awesome colors and design!",
      "rating": 5
    },
    {
      "author": "i_know_everything",
      "text": "These shoes are no good because I ordered the wrong size.",
      "rating": 1
    },
    "..."
  ],
  "questions": [{
      "author": "i_love_kicks",
      "text": "Do you guys make baby shoes?",
      "likes": 1223
    },
    {
      "author": "i_know_everything",
      "text": "Why do you make shoes out of garbage?",
      "likes": 0
    },
    "..."
  ],
  "stock_amount": 10000,
  "maker_address": {
    "building_number": 7,
    "street_name": "Turku",
    "city": "Saint-Petersburg",
    "country": "RU",
    "postal_code": 172091
  },
  "makers": ["Ilya Muromets", "Alyosha Popovich", "Ivan Grozniy", "Chelovek Molekula"],
}
```

3. Subset Pattern - Lab

Pattern-Subset

```
{
  "_id": "<objectId>",
  "item_code": "<string>",
  "name": "<string>",
  "maker_brand": "<string>",
  "price": "<decimal>",
  "description": "<string>",
  "materials": ["<string>"],
  "country": "<string>",
  "image": "<string>",
  "available_sizes": {
    "mens": ["<string>"],
    "womens": ["<string>"]
  },
  "package_weight_kg": "<decimal>",
  "average_rating": "<decimal>",
  "reviews": [{
    "author": "<string>",
    "text": "<string>",
    "rating": "<int>"
  }],
  "questions": [{
    "author": "<string>",
    "text": "<string>",
    "likes": "<int>"
  }],
  "stock_amount": "<int>",
  "maker_address": {
    "building_number": "<string>",
    "street_name": "<string>",
    "city": "<string>",
    "country": "<string>",
    "postal_code": "<string>"
  },
  "makers": ["<string>"]
}
```

3. Subset Pattern - Lab

Answer-document


```
{
  "_id": ObjectId("5c9be463f752ec6b191c3c7e"),
  "item_code": "AS45OPD",
  "name": "Recycled Kicks",
  "maker_brand": "Shoes From The Gutter",
  "price": 100.00,
  "description": "These amazing Kicks are made from recycled plastics and
  fabrics.They come in a variety of sizes and are completely unisex in design.
  If your feet don't like them within the first 30 days, we'll return your
  money no questions asked.
  ",
  "materials": [
    "recycled cotton",
    "recycled plastic",
    "recycled food waste",
  ],
  "country": "Russia",
  "image": "https:///www.shoesfromthegutter.com/kicks/AS45OPD.img",
  "available_sizes": {
      "mens": ["5", "6", "8", "8W", "10", "10W", "11", "11W", "12", "12W"],
      "womens": ["5", "6", "7", "8", "9", "10", "11", "12"]
  },
  "package_weight_kg": 2.00,
  "average_rating": 4.8,
  "top_five_reviews": [{
      "author": "i_love_kicks",
      "text": "best shoes ever! comfortable, awesome colors and design!",
      "rating": 5
    },
    {
      "author": "i_know_everything",
      "text": "These shoes are no good because I ordered the wrong size.",
      "rating": 1
    },
    "..."
  ],
  "top_five_questions": [{
      "author": "i_love_kicks",
      "text": "Do you guys make baby shoes?",
      "likes": 1223
    },
    {
      "author": "i_want_to_know_everything",
      "text": "How are these shoes made?",
      "likes": 1120
    },
    "..."
  ],
  "stock_amount": 10000,
}
```

04. Computed Pattern

Some computations are very expensive to do. If you store your information as base units in your database, you may find yourself redoing the same computations, manipulations, or transformations over and over.

In big data systems, these kinds of repeated computations can lead to very poor performance. What kind of transformations are usually applied to data?

The list is long. However, they usually fall into one of those three categories:

- Mathematical operations;
- Fan-out operations;
- Roll-up operations.

We will group those under a new pattern, which we will call the computed pattern.

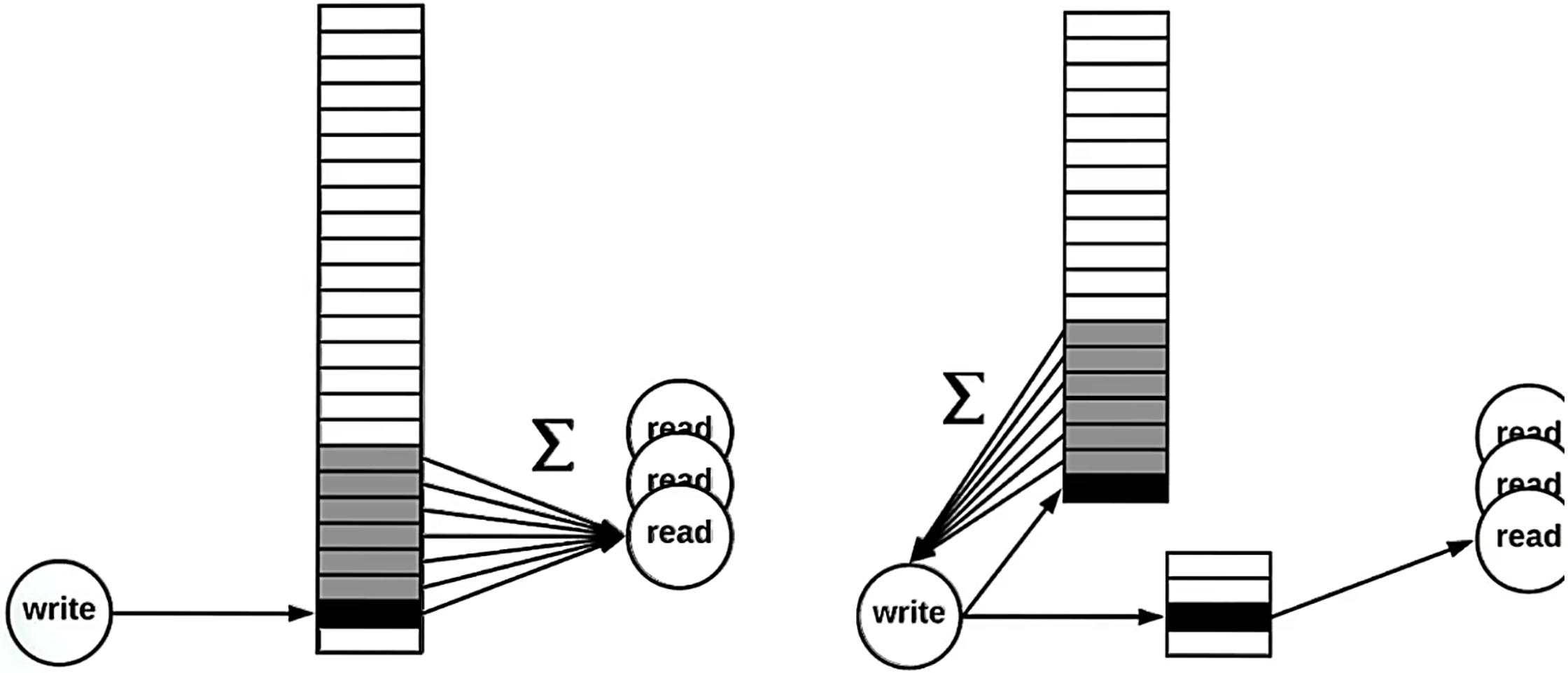
04. Computed Pattern

Kind of Computations/Transformations

- Mathematical Operations;
- Fan-Out Operations;
- Roll-up Operations.

04. Computed Pattern

Mathematical Operations



04. Computed Pattern

Mathematical Operations

The mathematical operations are easy to identify. These are the ones where we compute a sum or an average, find a median, et cetera. These are often associated with calling a built-in function in the server.

Why does it matter that we want to apply a pattern here?

Let's say we have a write operation that comes in. This piece of data is added as a document to a given collection. Another part of the application reads this collection and does, let's say, a sum on the numbers.

If we are doing 1,000 times more reads than writes, the sum operation we do with those reads is identical and very often does the exact same calculation for each of those read operations.

04. Computed Pattern

Mathematical Operations

We can save ourselves from all those identical operations by calculating the results when we get a new piece of data.

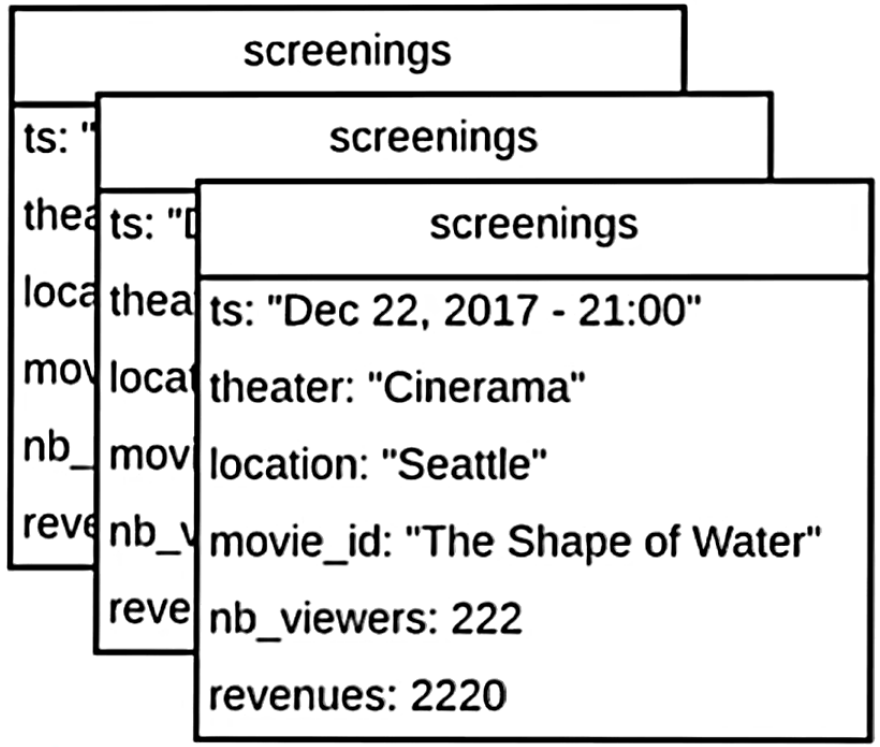
Once we have a new piece of data, we read the other element for the sum and store the result in another collection with documents more appropriate to keep the sum for that element.

This results in much fewer computations in the system, and we also reduce the amount of data being read.

Since we don't have to do the computation that at read time, we also save on those read operations, too. In this example, that will be 1,000 fewer computation and 1,000 fewer reads.

04. Computed Pattern

Mathematical Operations




```
{
  title: "The Shape of Water",
  ...
  viewings: 5,000
  viewers: 385,000
  revenues: 5,074,800
}
```

04. Computed Pattern

Mathematical Operations

A good example of this will be keeping track of ticket sales and then reporting the sale numbers let's say for a specific movie on a movie website.

There are likely fewer screenings happening per hour, which will be all right, than page views for the given movie where we want to display the sums.

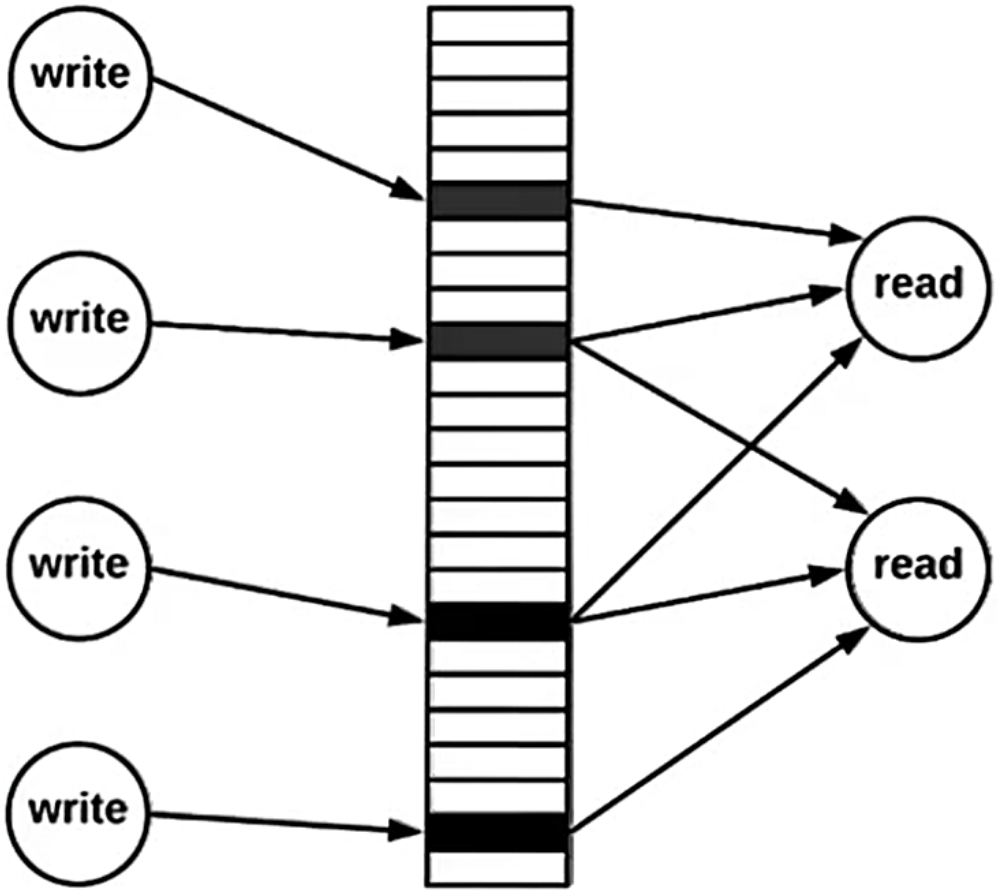
So instead of summing those screening documents to display the total viewers and sales every time access the movie, we are better off keeping the information in the document and updating it every time we get a new screening document.

We don't have to keep the screenings once the calculation is done. However, the idea was to regenerate the sums if needed or give us the ability to do more analytics.

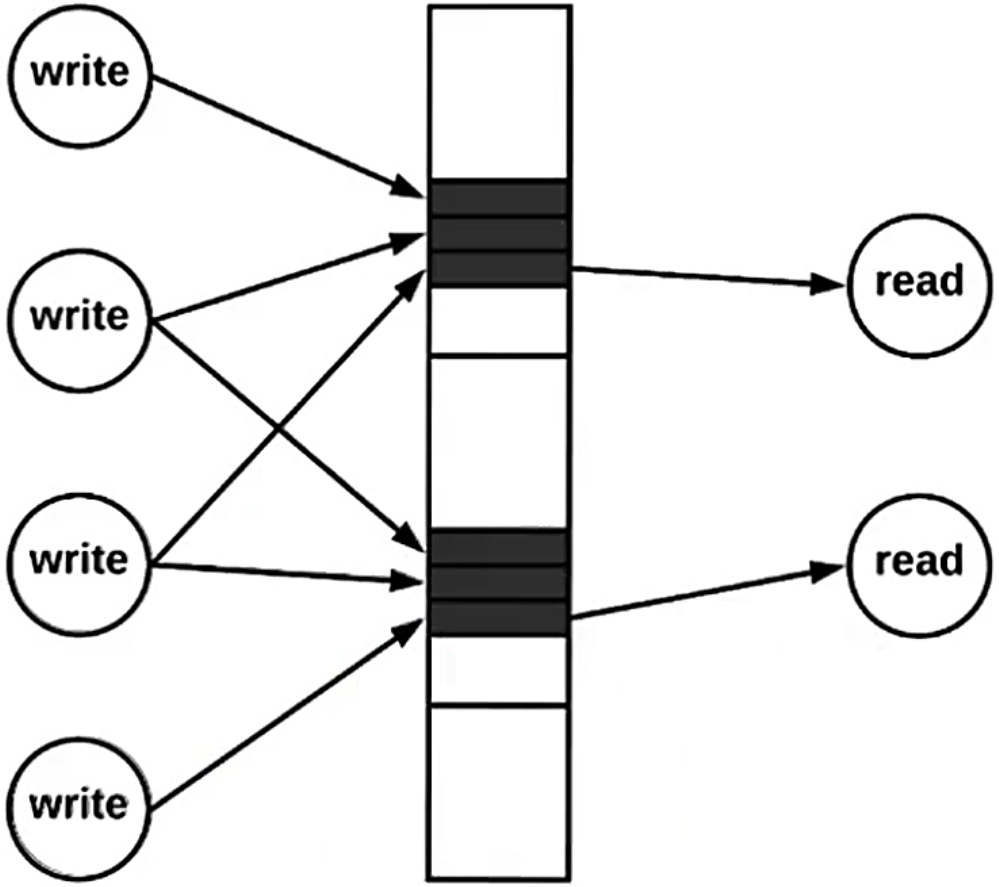
04. Computed Pattern

Fan-Out Operations

Fan-Out on Reads



Fan-Out on Writes



04. Computed Pattern

Fan-Out Operations

First, what does fan out mean? It means to do many tasks to represent one logical task. There are two basic schemes.

Either you fan out on reads, which means in order to return the appropriate data, the query must fetch data from different locations, or you fan out on writes, which means every logical write operation translates into several writes to different documents.

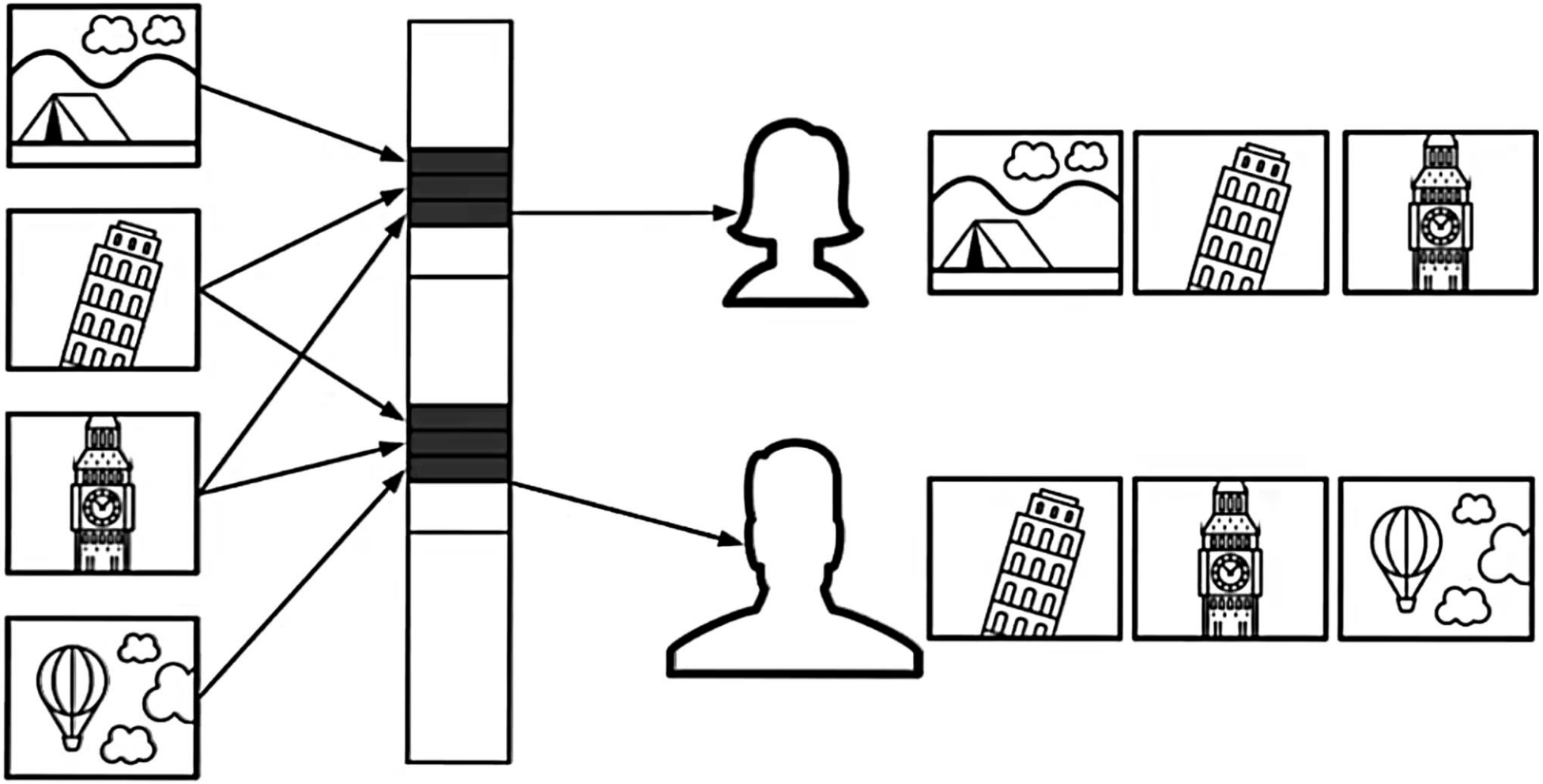
In doing so, the read does not have to fan out anymore, as the data is pre-organized at write time. If you don't pay attention to this pattern, you are likely doing fan out on reads. Why would you use fan out on writes?

If the system has plenty of time when the information arrives compared to the acceptable latency of returning data on a read operation, then preparing the data at write time makes a lot of sense.

Note that if you are doing more writes than reads so the system becomes bound by writes, this may not be a good pattern to apply.

04. Computed Pattern

Example of Fan-Out on Writes



04. Computed Pattern

Example of Fan-Out on Writes

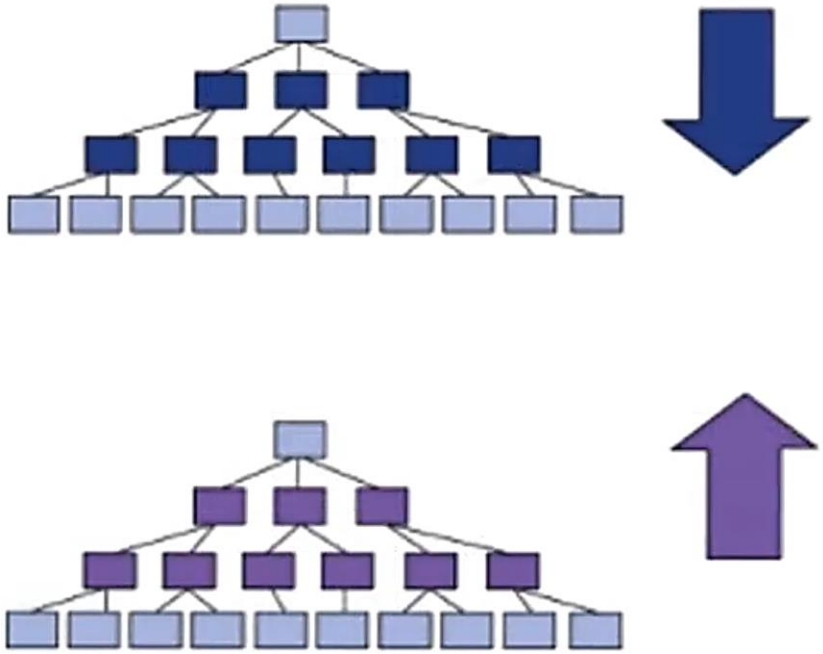
A good example for this pattern will be a social networking system for sharing photos. The system may copy the new photo on each follower's home page as it gets the new photo.

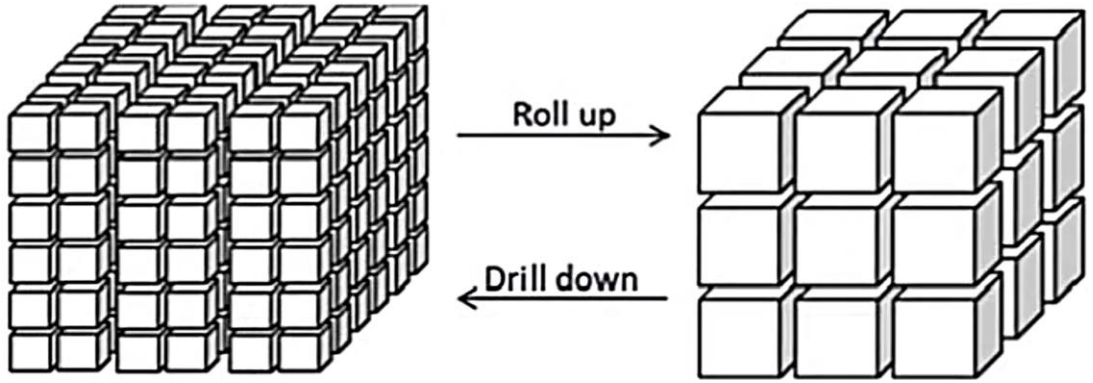
This way, when a user loads their own page, the system does not have to spend resources assembling all the information from all the people this user follows.

Everything needed for the user's home page would be available via a single document, which led to a much better experience with the site.

04. Computed Pattern

Roll Up Operations





04. Computed Pattern

Roll Up Operations.

We use this terminology in a position to drill down operations. In the roll-up operation, we merge data together.

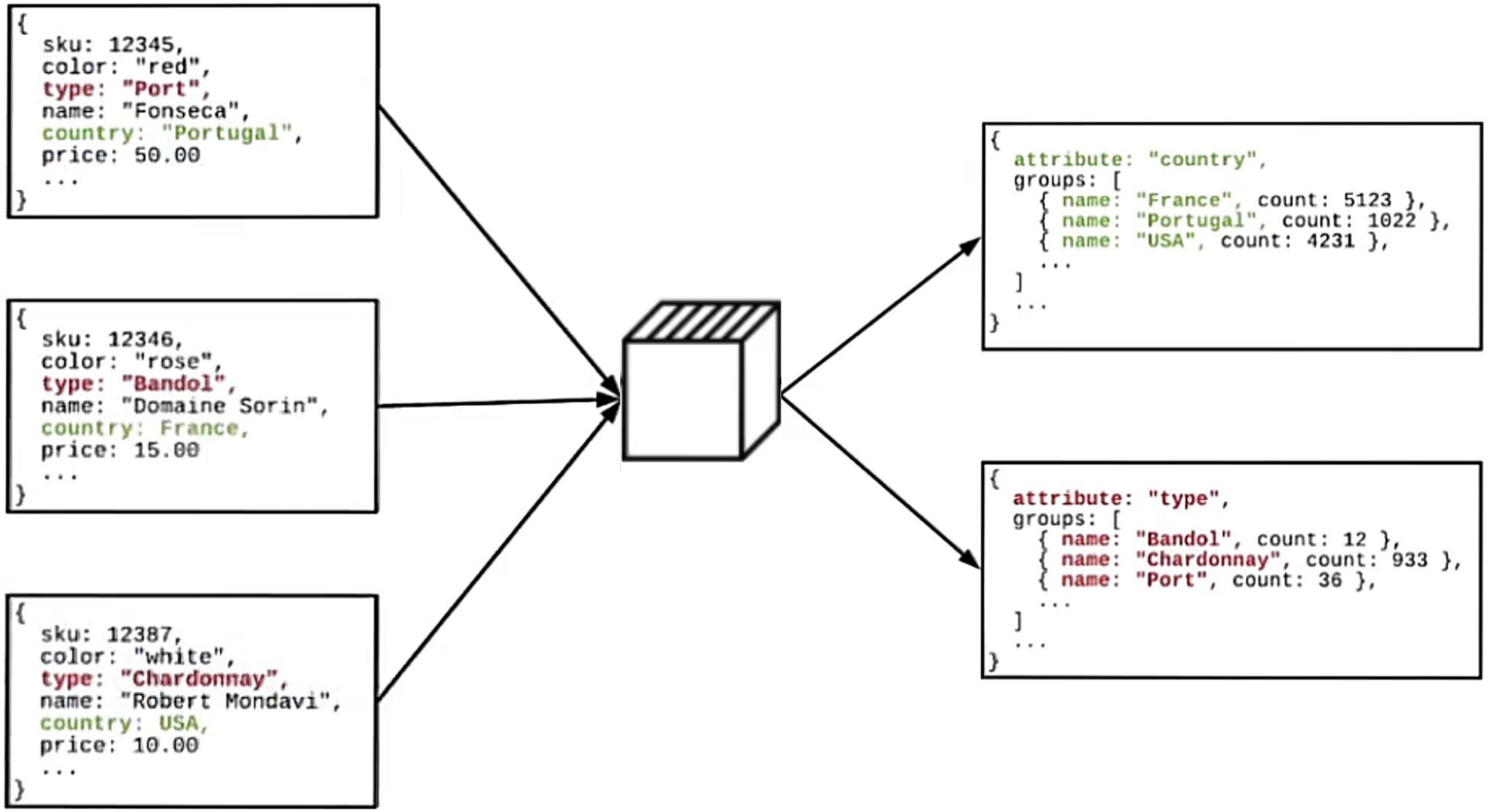
For example, grouping categories together in a parent category will be a roll-up. Grouping time-based data from small intervals to large ones will be another good example of a roll-up. This type of roll-up is often seen in reporting for hourly, daily, monthly, or yearly summaries.

Any operation that wants to see data at a high level is basically looking at rolling up data. Mathematical computations are roll-ups. However, roll-ups are more generic.

You can often think of roll-up data as running a group operation. So similarly to mathematical computation, doing those roll-ups at the right time may make more sense than having our read operation pay the costs of processing the transformations

04. Computed Pattern

Example of Roll Up.



04. Computed Pattern

Example of Roll Up.

Let's look at a concrete example. An inventory has different wine types.

The inventory change once in a while, however, not frequently. What is more frequent is looking at the wine organized by various categories.

For example, I may want to see the count of wine types per country of origin or per type so I can buy what is missing in my inventory to cover all my customer needs.

If I'm looking more often at the information on the right, the non-aggregated data than the changes that are happening on my collection on the left, it makes more sense to generate this data and cache it in the appropriate documents.

04. Computed Pattern

When to apply the Computed Pattern?

- Oversure of resources (CPU)
  - *If you see you are using a lot of CPU, this may be a sign that you're doing much more than transferring data from and to the disk. Keep in mind that some tasks like compressing and decompressing data between the disk and the cache require a lot of CPU, too.*

- Reduce latency for read operations
  - *If you have a long-read operation that depends on complex aggregation queries, you might want to make them run faster.*

04. Computed Pattern

Problem

- Costly computation or manipulation of data;
- Executed frequently on the same data, producing the same result;

Solution

- Perform the operation and store the result in the appropriate document and collection;
- If need to redo the operations, keep the source of them.

Use case Examples

- Internet Of Things (IOT);
- Event Sourcing;
- Time Series Data;
- Frequent Aggregation Framework queries.

Benefits and Trade-Offs

✓ Read queries are faster;

✓ Saving on resources like CPU and Disk;

☒ May be difficult to identify the need;

☒ Avoid applying or overusing it unless needed.

04. Lab - Computed Pattern

- **User Story**:
  - Our city is going green, and we're reassessing our power plant. A lot of residents are switching to solar panels for their source of electricity and the power plant needs to be both ecologically friendly and flexible in its distribution of electricity throughout the city. A number of residents are installing solar panels on their homes and selling the excess energy back to the power plant.
  - Our database is tracking the following data about each building in the city: how much energy in kilowatts (kW) per hour it produces (if any), how much energy it consumes, and how much energy needs to be supplemented daily.
  - In order to make our plant more flexible in adapting to the growing number of solar-powered homes, our computer algorithms are analyzing consumption data by city zones one day at a time.
  - To resolve this issue, we want to reduce the amount of data immediately available to the user in the app and only load additional data when the user asks for it.

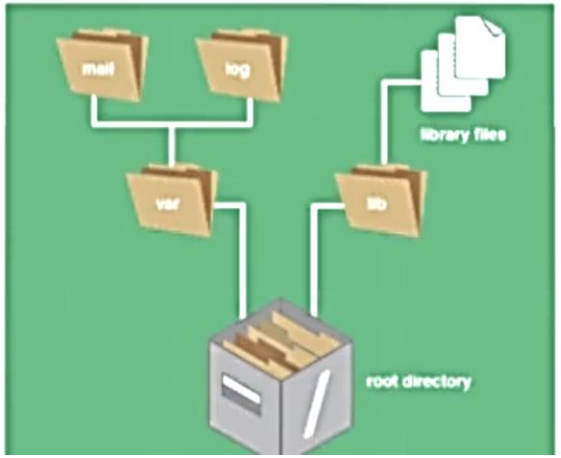
04. Lab - Computed Pattern

- **Problem**:
  - The current design requires the application to calculate zone totals from each building's report every day, which is a lot of unnecessary calculation during reads. Every zone has anywhere from 100 to 900 buildings and we will create a zone collection using the consumption data that comes in from every unit in each zone daily.
  - The new design you will implement will be based on the Computed Pattern. You will need to create a separate collection that stores zone-based summaries on zone consumption, production, and city-supplemented energy metrics, which we will calculate and overwrite whenever new data comes into the system.

05. Bucket Pattern


```
C     CALCULATE STATISTICS ON DATA FROM LOW SPEED READER
      SUM=0
      SUMSQ=0
      TYPE 100
100   FORMAT("ENTER THE NUMBER OF VALUES TO CALCULATE STATISTICS ON"/)
      ACCEPT 10,N
10    FORMAT(I)
      DO 200 I=1,N
      READ 1,110,V
110   FORMAT(F)
      SUM=SUM + V
      SUMSQ=SUMSQ + V**2
      TYPE 120,I,V
120   FORMAT("VALUE",I5,"IS",F8.2/)
200   CONTINUE
      SAMP=N
      AVRG=SUM/SAMP
      STD=SQRT(SUMSQ/SAMP - AVRG**2)
      TYPE 300,N,AVRG,STD
300   FORMAT("NUMBER OF VALUES",I5,"MEAN",F8.2,"STANDARD DEVIATION",F8.2/)
      END
```

One document per piece of information



All information in one document

05. Bucket Pattern

In the early days of programming, you would feed the giant computers with punch cards, like this one on the left. Each instruction would be stored on a separate card. And yes, it was preferable to not draw the ordered stack of cards, especially if you did not use some kind of label to reorder them.

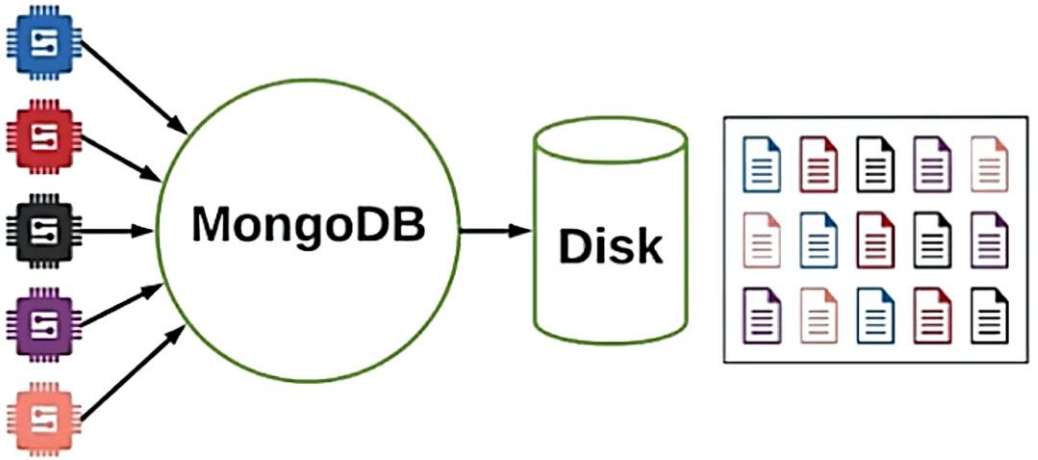
Just a little later, the advent of terminals permitted the developers to put all these instructions in a file, like the one we have on the right. This was better. However, as programs got bigger, the files got larger, and it was not a viable solution anymore. The solution at that time was to group a set of instructions together. Call it files, classes, or libraries this is still the system in use today.

The point I want to make is that sometimes you need a middle-of-the-road solution because both extremes are not working well and are far from being optimal. The punch card is too granular, while the single file solution is too broad and does not provide enough granularity.

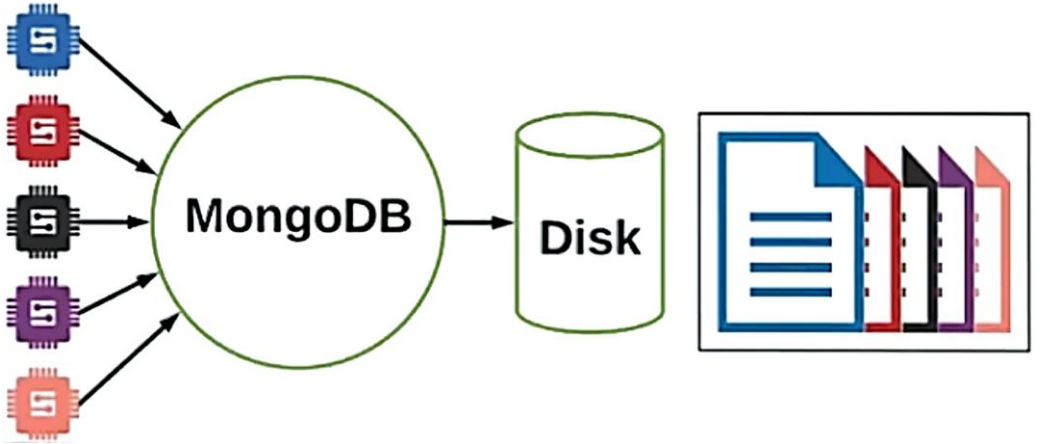
05. Bucket Pattern

Too Many or Too Big

- One document per piece of information, per device;



- One document will all the information for a device



05. Bucket Pattern

Just the Right Amount

One document per device per day

```
{
  "device_id": 000123456,
  "type": "2A",
  "date": ISODate("2018-03-02"),
  "temp": [ [ 20.0, 20.1, 20.2, ... ],
            [ 22.1, 22.1, 22.0, ... ],
            ...
          ]
}

{
  "device_id": 000123456,
  "type": "2A",
  "date": ISODate("2018-03-03"),
  "temp": [ [ 20.1, 20.2, 20.3, ... ],
            [ 22.4, 22.4, 22.3, ... ],
            ...
          ]
}
```

One document per device per hour

```
{
  "device_id": 000123456,
  "type": "2A",
  "date": ISODate("2018-03-02T13"),
  "temp": { 1: 20.0, 2: 20.1, 3: 20.2, ... }
}

{
  "device_id": 000123456,
  "type": "2A",
  "date": ISODate("2018-03-02T14"),
  "temp": { 1: 22.1, 2: 22.1, 3: 22.0, ... }
}
```

05. Bucket Pattern

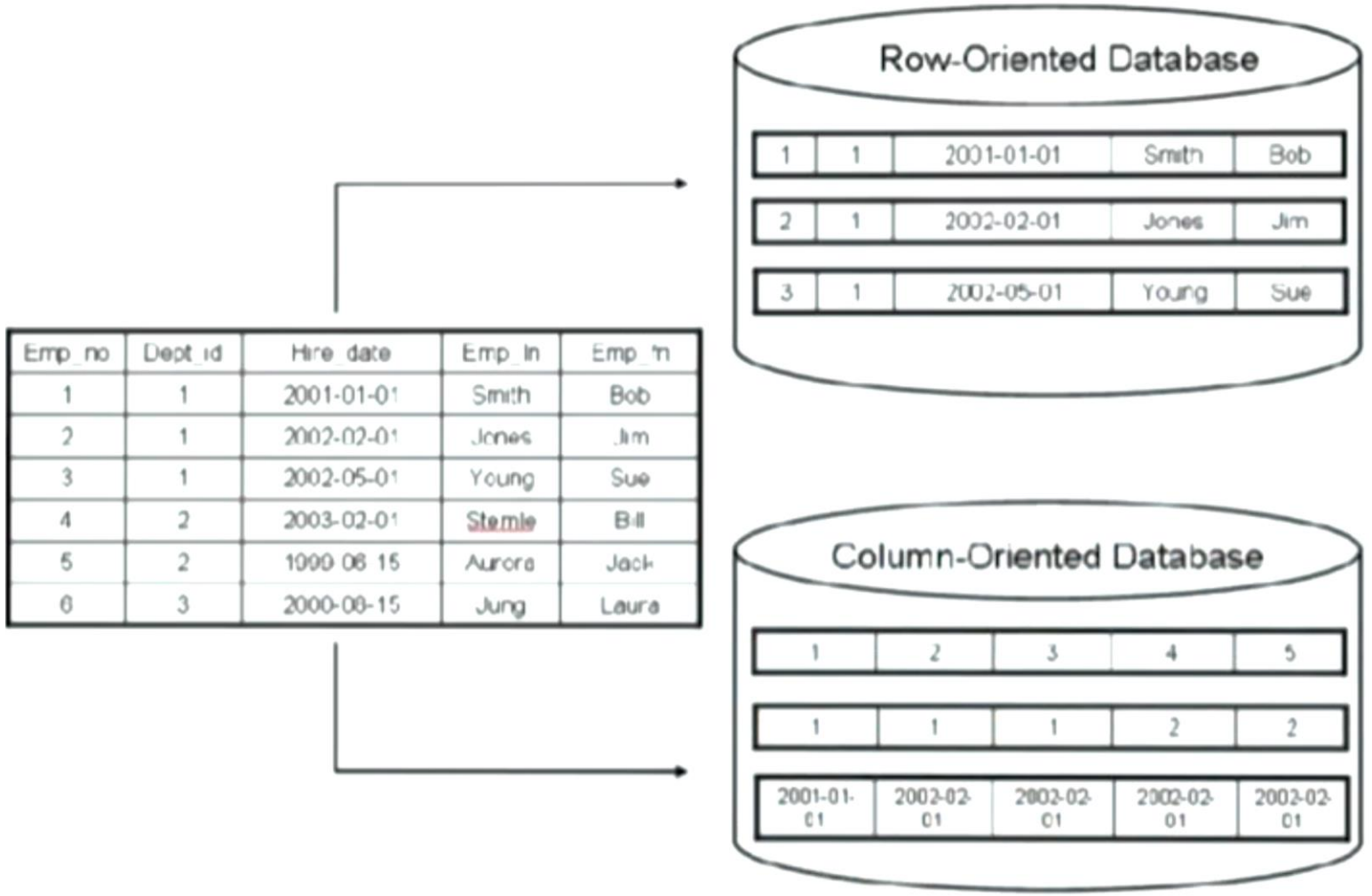
Collaboration Platform Example

- Many channels
- Design alternatives
  - One document per message
  - One document for all messages in a channel
  - One document per channel, per day

```
{ {
    "channel_id": "13579246",
    "name": "mongodb_tech",
    "date": ISODate("2018-03-02"),3:14:16"),
    "messages": [                    e group>" ,
      "<Daniel joined the group>",
}     "Nathan: Welcome Daniel",
      "Daniel: Thanks!"
    ...
    ]
  }
      "<Channel created>",
      "<Nathan joined the group>",
      "<Daniel joined the group>",
      "Nathan: Welcome Daniel",
      "Daniel: Thanks!"
    ...
    ]
  }
```

05. Bucket Pattern

Row-based vs Column-based approach



05. Bucket Pattern

Column Oriented Data

```
{
  "device_id": 000123456,
  "location": [ 85.1, 17.2 ]
  "date": ISODate("2018-03-02T13"),
  "temp": [ 20.0, 20.1, 20.2, ... ],
  "pressure": [ 1.01, 1.02, 1.01, ... ],
  "light": [ 8500, 8520, 8525, ... ],
  ...
}
```

```
db.iot.aggregate([{$project:{avg_temp:{$avg:
  "$temp"}}}])
```

```
{
  "device_id": 000123456,
  "date": ISODate("2018-03-02T13"),
  "temp": [ 20.0, 20.1, 20.2, ... ]
}
```

```
{
  "device_id": 000123456,
  "date": ISODate("2018-03-02T13"),
  "pressure": [ 1.01, 1.02, 1.01, ... ]
}
```

```
{
  "device_id": 000123456,
  "date": ISODate("2018-03-02T13"),
  "light": [ 8500, 8520, 8525, ... ]
}
```

```
db.iot.aggregate([{$match:{temp:{$exists:1}}},
  {$project:{avg_temp:{$avg:"$temp"}}}])
```

05. Bucket Pattern

Gotchas with Buckets

- Random insertions or deletions in buckets;
- Difficult to sort across buckets;
- Ad-hoc queries may be more complex, again across buckets;
- Works best when the “complexity” is hidden through the application code.

05. Bucket Pattern

Problem

- Avoiding too many documents or too big documents;
- A 1-to-many relationship that can't be embedded.

Solution

- Define the optimal amount of information to group together;
- Create arrays to store the information in the main object;
- It is basically an embedded 1-to-many relationship, where you get N documents, each having an average of Many/N sub documents.

Use case Examples

- Internet Of Things (IOT);
- Data warehouse;
- Lots of information associated to one object.

Benefits and Trade-Offs

✓ Good balance between number of data access and size of data returned;
✓ Makes data more manageable;
✓ Easy to prune data;
☒ Can lead to poor query results in not designed correctly;
☒ Less friendly to BI tools.

05. Lab - Bucket Pattern

1. Scenario

- You've been called in to help improve the performance profile of an application that provides a dashboard and reporting information of cell tower quality service metrics.
- This application collects a set of metric information sent directly from cell towers.
- The information sent in each message corresponds to:
  - the current longitude and latitude of the cell tower – coordinates;
  - a unique identifier of the cell tower - celltower_id;
  - the date of the measurement – date;
  - the following accumulated counters, accumulated since last reboot:
    - the number of established cell phone calls - established_calls;
    - the number of dropped cell phone calls - dropped_calls;
    - the amount of inbound data traffic - data_in_gb;
    - the amount of outbound data traffic - data_out_gb.

05. Lab - Bucket Pattern

1. Scenario

- This system needs to support the following operational requirements:
- Be capable of storing measurements for at least 500 cell towers;
- Be able to produce cell tower cumulative reports on one or all four of the accumulated counters, with the following specifications:
  - Each of these reports consists of a plotted graph of the last 24 hours for each metric, with a granularity of 5 minutes;
  - The 95 percentile request latency expected for this report is 100ms.

05. Lab - Bucket Pattern

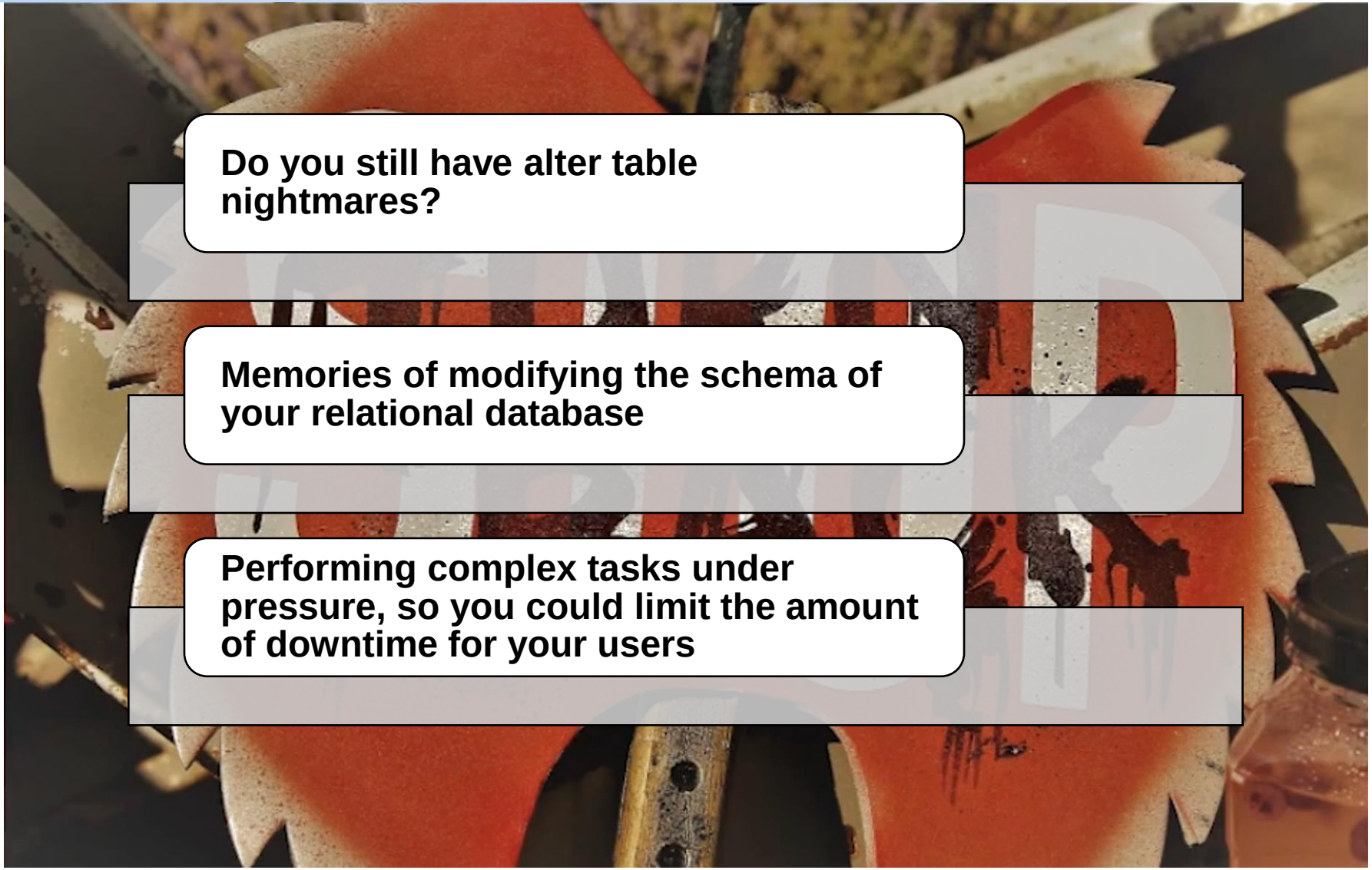
2. Current Implementation

- This implementation is expected to fill out the database servers disk space in one month and there is no budget left for hardware improvements in the next 12 months.
- The current implementation is unable to produce the reports within the required 100ms.

3. Your Solution

- In order to resolve the issues that the current approach is facing, you need to come up with a schema design alternative that allows for:
  - an increase in the number of IoT devices and associated workload growth;
  - all report generation to comply with the expected SLA of less than 100ms;
  - allow for the application to report status for the last 24 hours, with a granularity of 5 minutes.
- Using your pattern knowledge, consider the following three choices for the implementation

06. Schema Versioning Pattern



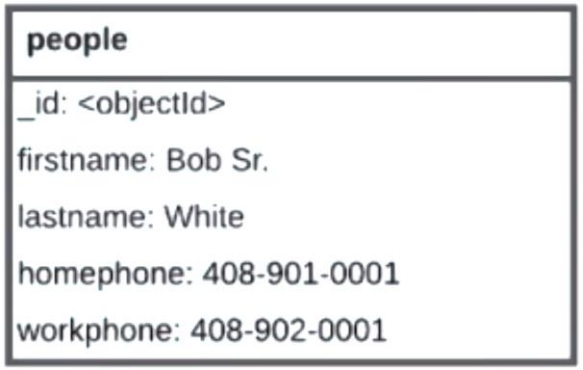
06. Schema Versioning Pattern

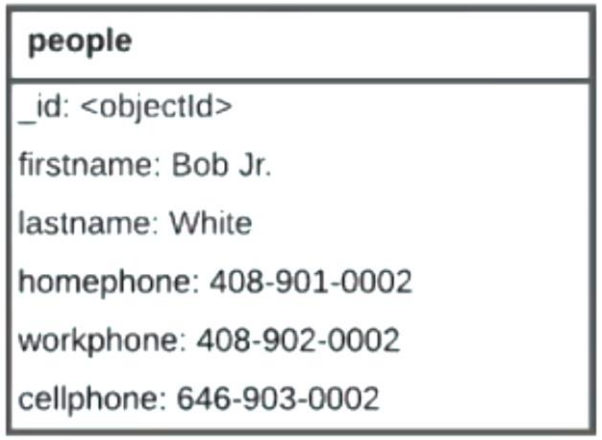
06. Schema Versioning Pattern

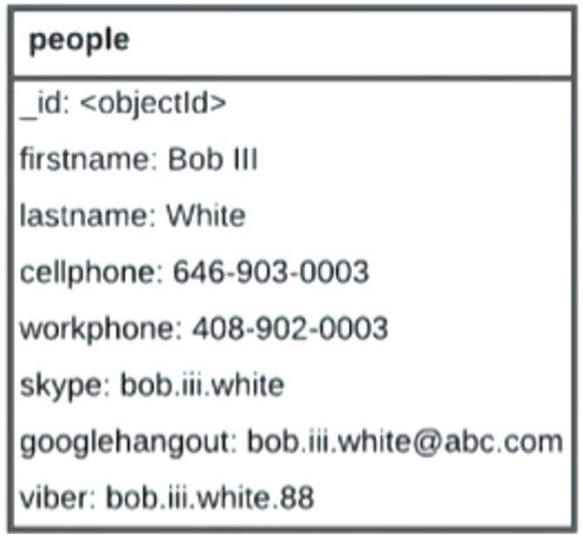
Updating a Relational Database Schema

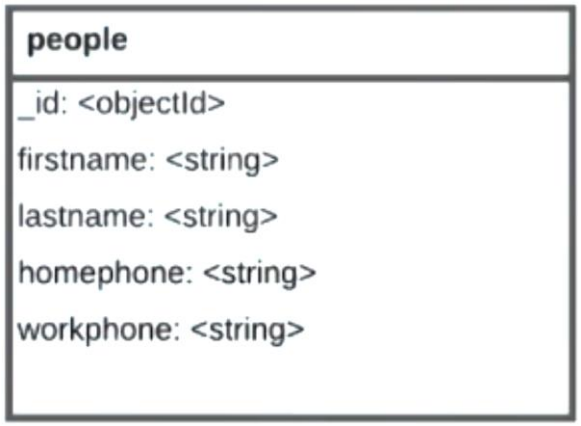
- Need time to update the Data;
- Usually done by stopping the Application;
- Hard to revert if something goes wrong.

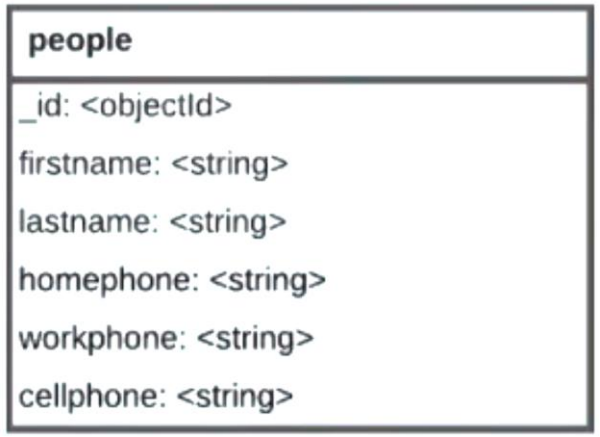
06. Schema Versioning Pattern











06. Schema Versioning Pattern

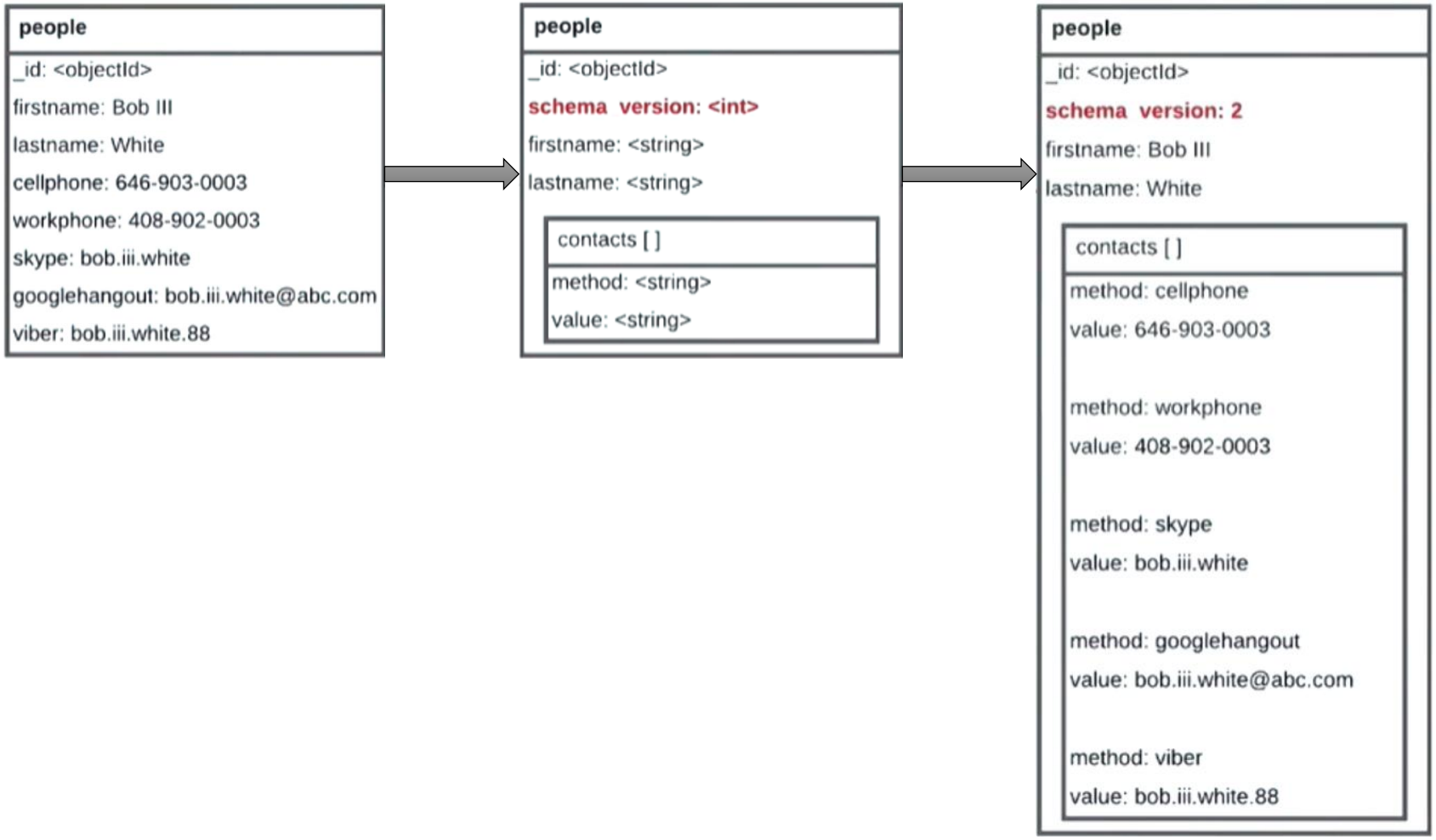
Because documents can have different shapes. By that, we refer to having different fields or different types for a given field. We construct our applications to deal with those variants.

Let's say we have Bob White as a person in our people collection. Bob was born a while ago and can be reached at home or at work through these phone numbers.

Then we have Junior, Bob White's son. Bob is from a generation that got used to cell phones. And we need to add this information to his profile, and have our application deal with this additional information. I suspect making this change was not too painful. It would even be pretty smooth with a relational database, as adding a column is not a big deal.

Now, as the next generation of White Bob III does not have a phone at home. However, has an account with Skype, Google Hangouts, Viber, and who knows what next week. Well, this is becoming difficult to handle in the application. However because the developers took the wonderful class Entry 20 on Data Modeling, they realized that the attribute pattern would be a good fit here.

06. Schema Versioning Pattern



06. Schema Versioning Pattern

So instead of growing the documents with unpredictable contact methods, the attribute pattern is applied. The application could easily handle the differences in the Bob White documents.

Here, instead of dealing with an additional field, you get to see an array of contacts method when you process it accordingly.

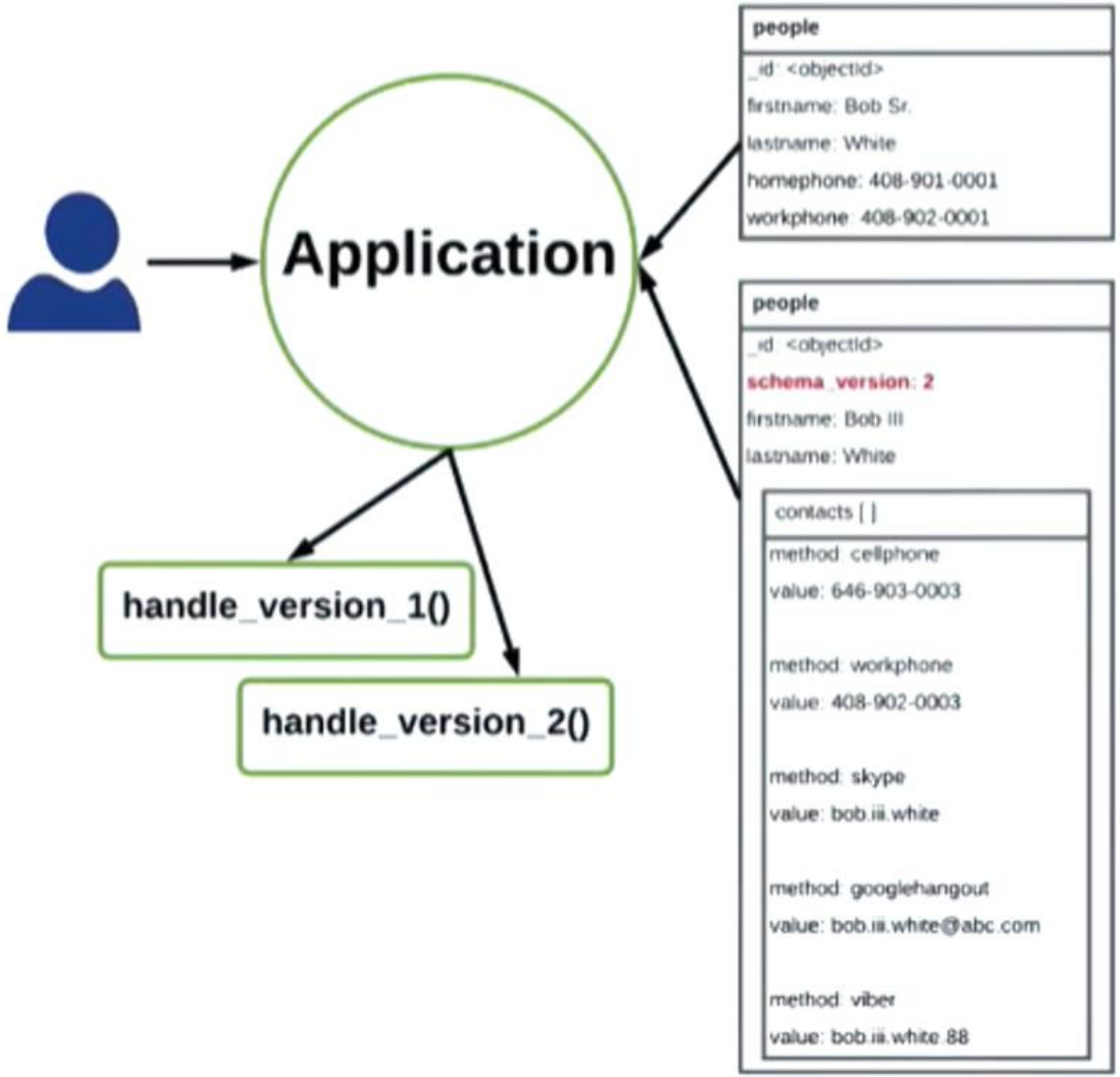
To help the application identify the shape, we will have a version field and set it to 2, the non-presence of this field being the implicit version 1. Note that this field is present in each document.

This is in contrast to a relational database, where you only have one global number to track the version of the schema. Here we have each document telling us which version of the schema to adhere to.

06. Schema Versioning Pattern

Application Lifecycle

- Modify Application
  - Can read/process all versions of document
    - Have different handler per version;
    - Reshape the document before processing it.
- Update all Application servers
  - Install updated application;
  - Remove old processes.
- Once migration completed
  - Remove the code to process old versions.



06. Schema Versioning Pattern

Application Lifecycle

The first thing you will need to do is to change the application so it can read all the different versions of the documents. You can use separate handlers or functions to handle the different versions.

The field schema version makes it easy to decide where to send the document. Alternatively, reshape the document when you see an old version, and then process it.

Testing that functionality is relatively easy. As you start adding documents with the upcoming shape to your test environment, and then observe how the application deals with those changes. Update your application server so they use the modified application.

Once the migration of all documents is completed, it's up to you to decide whether to remove or not the obsolete code.

06. Schema Versioning Pattern

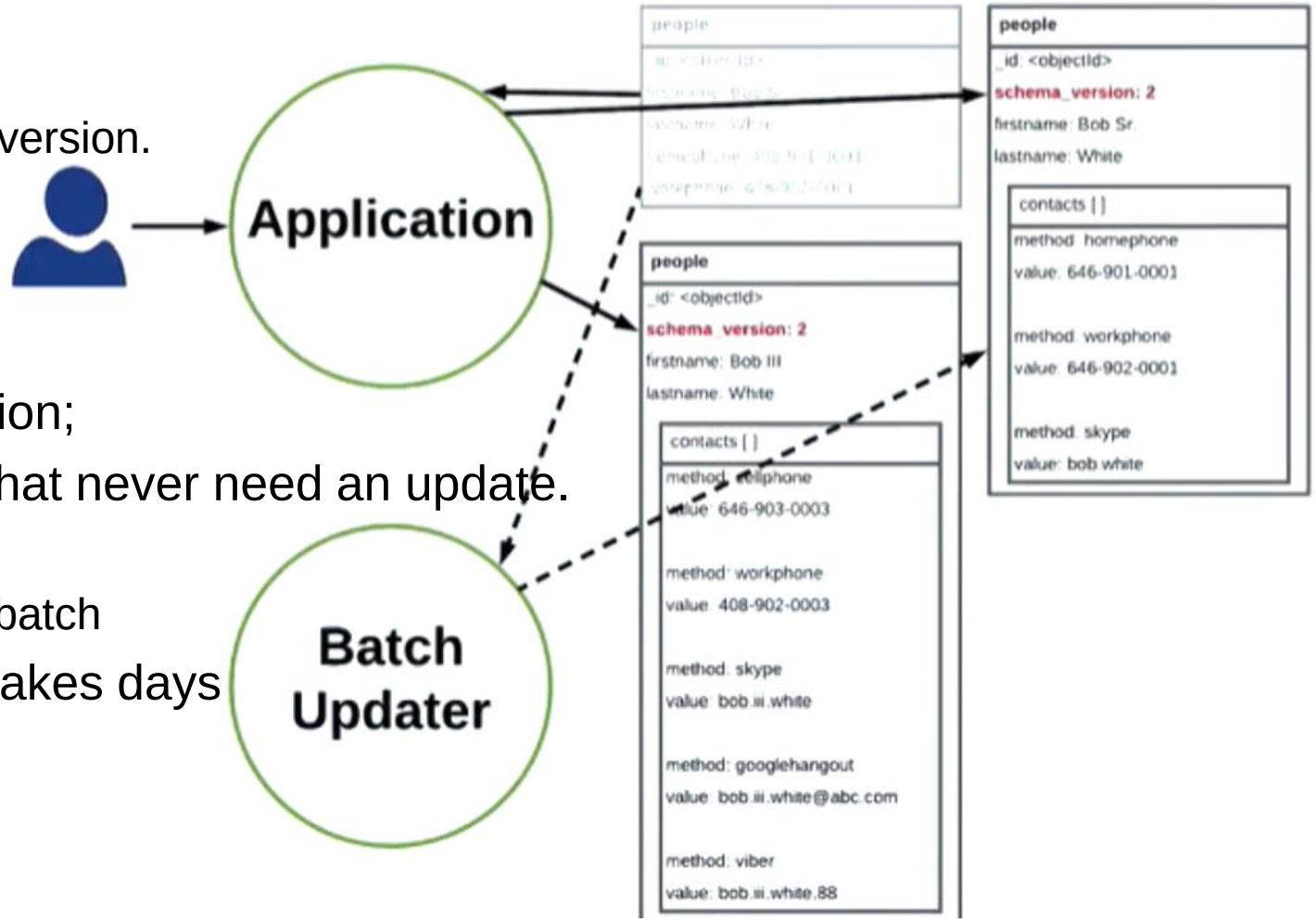
Document Lifecycle

- New Documents:
  - Application writes them in lastest version.

- Existing Documents

  A. Use updates to documents
     - To transform to latest version;
     - Keep forever documents that never need an update.

  B. … or transform all documents in batch
     - No worry even if process takes days



06. Schema Versioning Pattern

Document Lifecycle

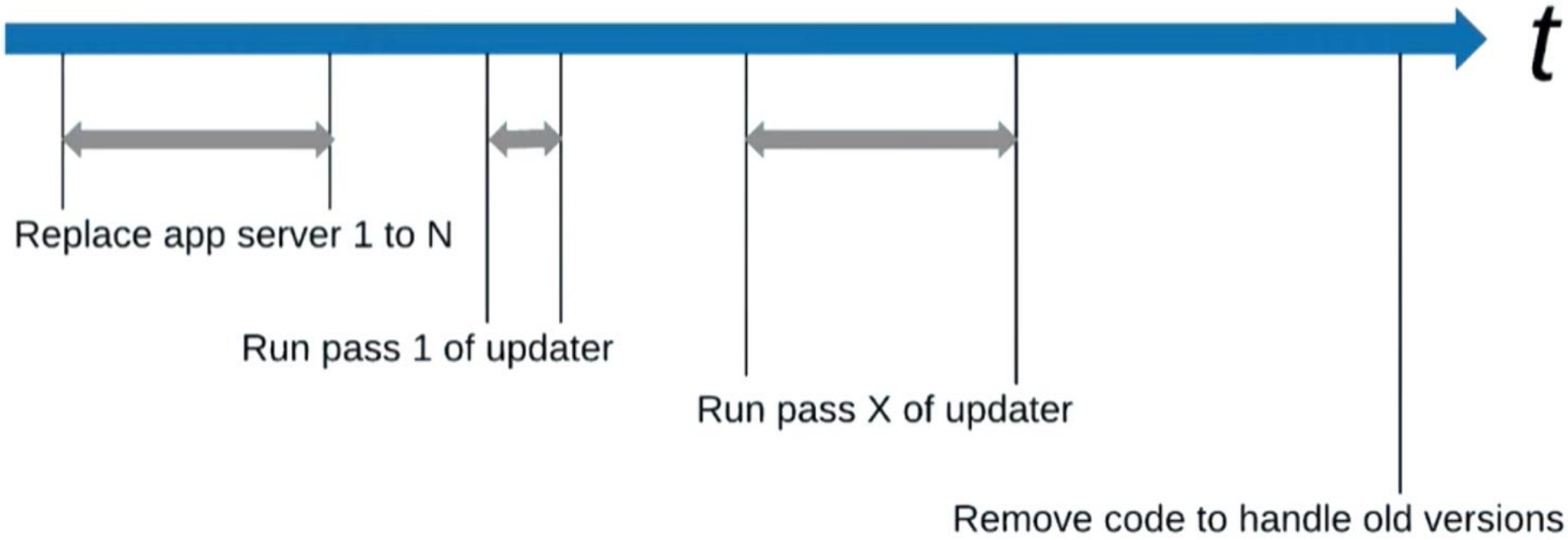
New documents get written in the latest schema version.

For existing documents, you can piggyback on the fact that you're going to make an update to change the shape and write it in the new schema. As a matter of fact, I remember a customer that had billions of documents, a lot being inactive. And that was their only migration strategy.

That means that the old obsolete documents remained unchanged because they did not want to pay the price to update them with a job running for a few weeks.

Alternatively, to this selective update strategy, the most common one is to have a background task to do the updates.

06. Schema Versioning Pattern



06. Schema Versioning Pattern

And that can be done in a few passes.

While the documents are getting updated, your application servers can handle both schema version work well with all documents, old and new shapes.

At the end, if and once all documents are updated, you can remove the code to handle the old version in your application.

Some users keep it there for protection or for keeping the ability to reimport documents in the old shape.

Anyway, there's a lot of possible variations of migration. Just ensure you get a plan.

06. Schema Versioning Pattern

Problem

- Avoid downtime while doing schema upgrades;
- Upgrading all documents can take hours, days or even weeks when dealing with big data;
- Don’t want to update all documents

Solution

- Each document gets a “schema_version” field;
- Application can handle all versions;
- Choose your strategy to migrate the documents

Use case Examples

- Every application that use database deployed in production and heavily used;
- System with a lot of legacy data.

Benefits and Trade-Offs

✓ No downtime needed;
✓ Feel in control of the migration;
✓ Less future technical debt;

☒ Can lead to poor query results in not designed correctly.

06. Lab - Schema Versioning Pattern

Problem:

Which of the following scenarios are best suited for applying the Schema Versioning Pattern?

- Scenario A:

Your team was assigned to upgrade the current schema with additional fields and transforming the type of different fields without bringing the system down for this upgrade. However, all documents need to be updated to the new shape quickly.

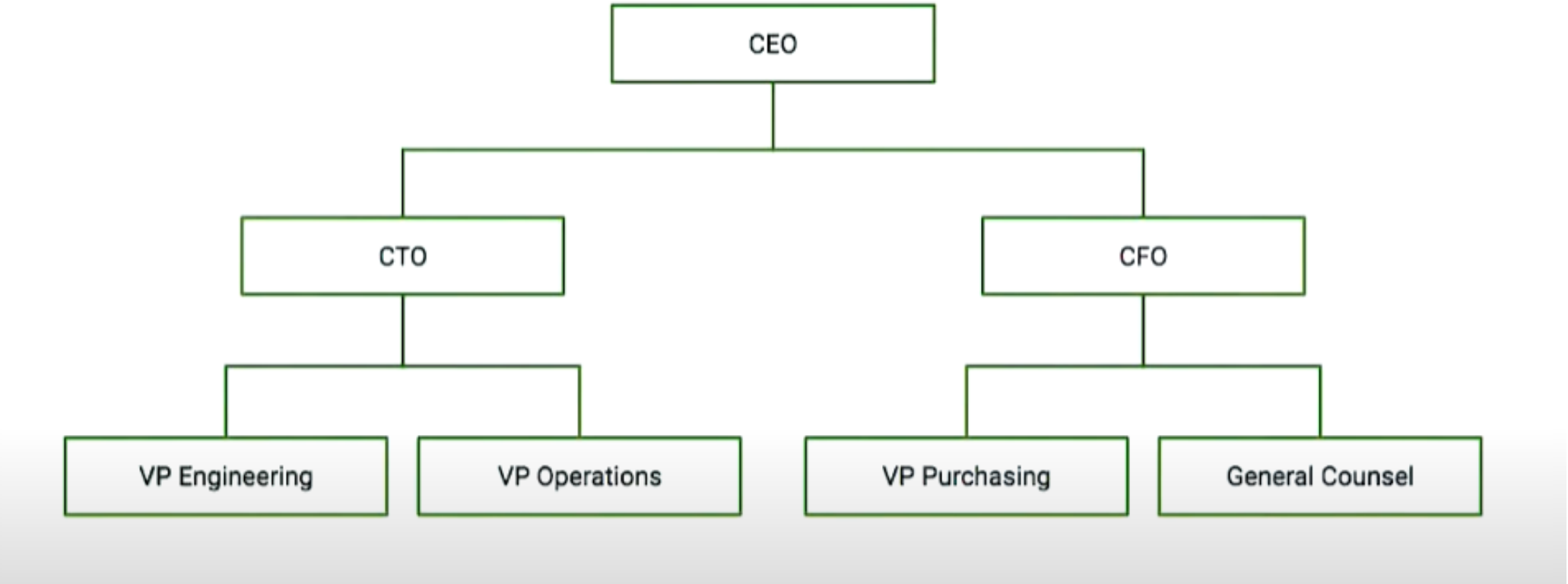
- Scenario B:

The performance of your application became suboptimal over time. Your team has identified that the most commonly used collection could profit from embedding additional information from other collections using the Subset and Computed Patterns. All documents in the commonly-used collection will need to undergo this modification. If possible, you would like the transition to be done without downtime.

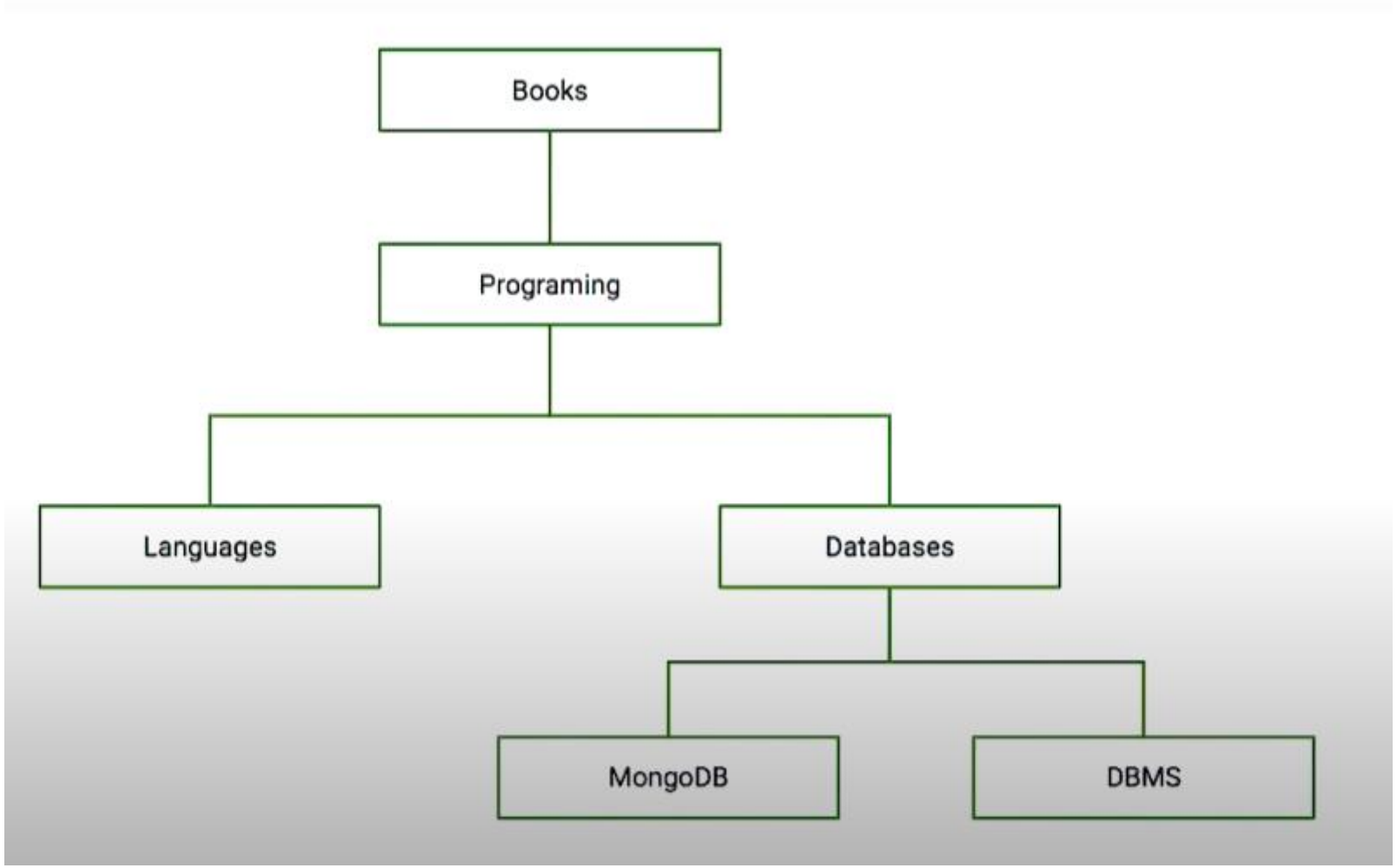
- Scenario C:

Your company was bought by its slightly more successful competitor. Thankfully both your and your new owner's applications are flexible enough to handle both document shapes well. You do not have to modify the application or your document shape, but due to the merger, you have to keep documents with different structures in the same collection.

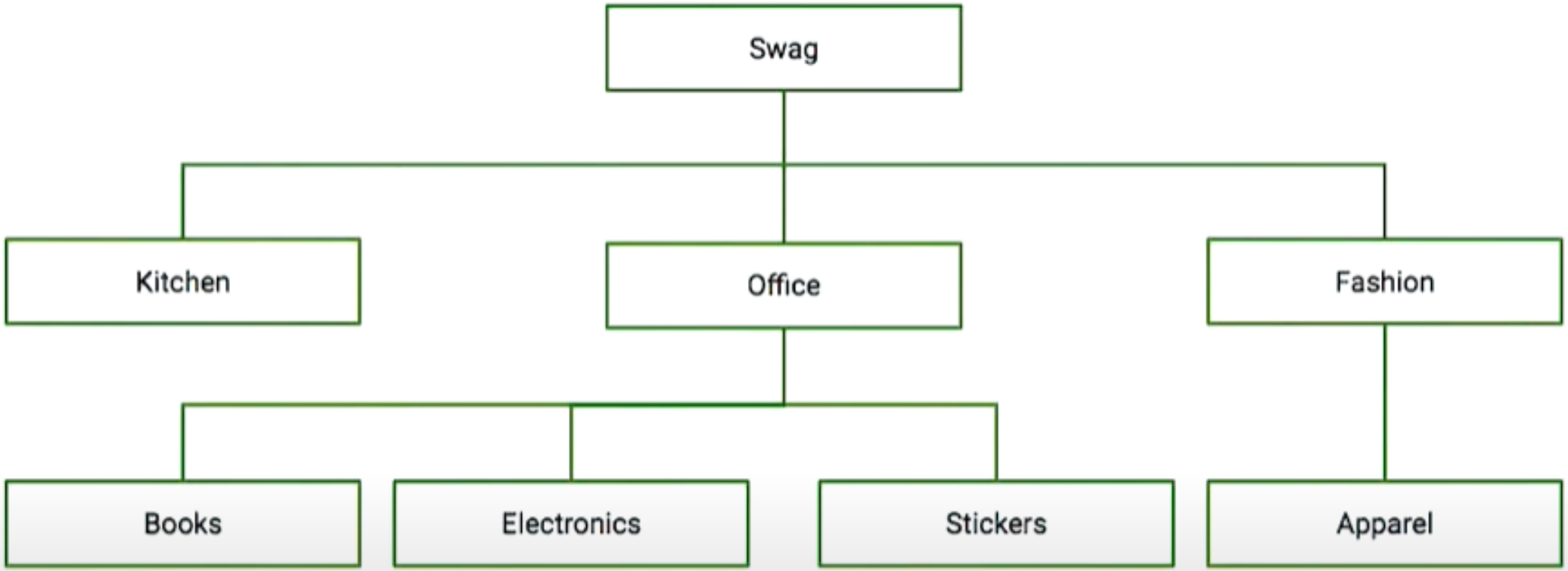
07. Tree Patterns



07. Tree Patterns



07. Tree Patterns



07. Tree Patterns

In this lesson, we are going to learn about how to model hierarchical information using tree patterns and examine common situations where tree patterns can be used.

There are several types of use cases that find hierarchies between objects or nodes and the relationship between them, use cases like company organization charts, subject areas structures in a given domain like books in a bookstore, or even a more typical example, the categories of products for a given e-commerce site or shop.

All of these examples are hierarchical in nature where there is a direct relationship between the parent nodes and the child nodes.

07. Tree Patterns

- Who are the ancestors of node X?
- Who report to Y?
- Find all nodes that are under Z?
- Change all categories under N to under P

07. Tree Patterns

```
    "_id" : 7,
    "title" : "Brown Tumbler",
    "slogan" : "Bring your coffee to go",
    "description" : "The MongoDB Insulated Travel Tumbler is smartly designed to maintain temperatures ",
    "stars" : 0,
    "category" : "Kitchen",
    "img_url" : "/img/products/brown-tumbler.jpg",
    "price" : NumberDecimal("9.00"),
    "sold_at" : [
        ObjectId("55aea9699f876cd5bcb77fcc"),
        ObjectId("55aea9699f876cd5bcb77fdb"),
        ...
    ],
    "top_reviews" : [
        {
            "body" : "Customer finish player that.",
            "user_id" : ObjectId("5cc8beb8b2c78b148aededf6"),
            "user_name" : "Cody Allen",
            "date" : "1983-04-18"
        },
        …
    ]
}
```

07. Tree Patterns

Each hierarchical node relationship comes associated with common operations that will be useful to us in many situations regardless of the actual content of the database, things like who are the ancestors of node X or reports to Y.

Find all nodes that are under Z, or change all categories of N to under P. These are common operations applicable to any information that is hierarchical in nature.

Documents, in themselves, are hierarchical structures. They contain multidimensional information within a given node, such as the relationships between a product and the stores that the product's sold at or the reviews of the same product.

However, when representing the dependency and hierarchy between nodes of the same entity, like categories, it is better to use tree patterns.

07. Tree Patterns

Model Tree Structures

Parent References

Child References

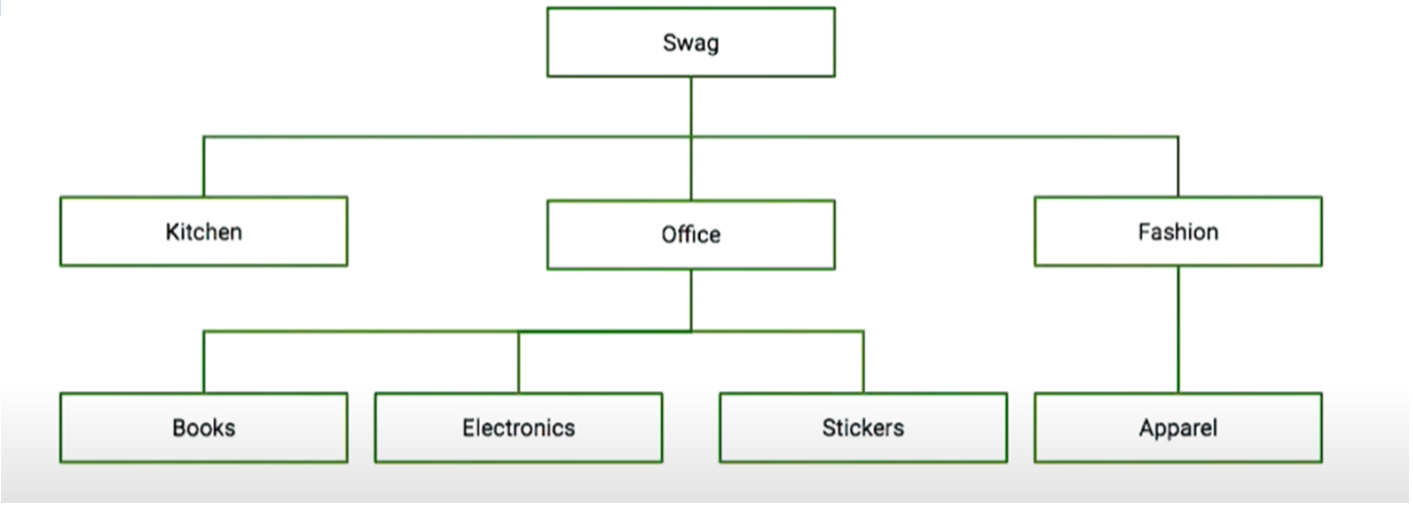
Array of Ancestors

Materialized Paths

07. Tree Patterns

Parent References

```
{
        Name: “Office”,
        Parent: “Swag”,
        …
}
```



Change all categories under N to under P

```
// all ancestors

db.categories.updateMany(
    {parent: N},
    {$set: { parent: P}}
)
```

07. Tree Patterns

Parent References

The document model offers a few patterns for modeling tree structures. We have the parent references, child references, arrays of ancestors, and materialized baths.

In the case of parent preferences, the document holds a reference to the parent node in the tree hierarchy. We can collect all ancestors by running an aggregation pipeline with a graph lookup stage to retrieve all subsequent parents of the immediate parents traversing the full tree.

To find all reports of a given parent, we can run a find command matching for the parent and then retrieving all children nodes.

In order to change all nodes that report, or are children of one parent, in other words, change all categories under N to children of P, we can use an update operation.

07. Tree Patterns

Parent References

! Who are the ancestors of node X?

✓ Who report to Y?

! Find all nodes that are under Z?

✓ Change all categories under N to under P

07. Tree Patterns

Parent References

For parent references, we can perform operations like who reports to Y and change all categories under N to under P using very little code and changing only one single document.

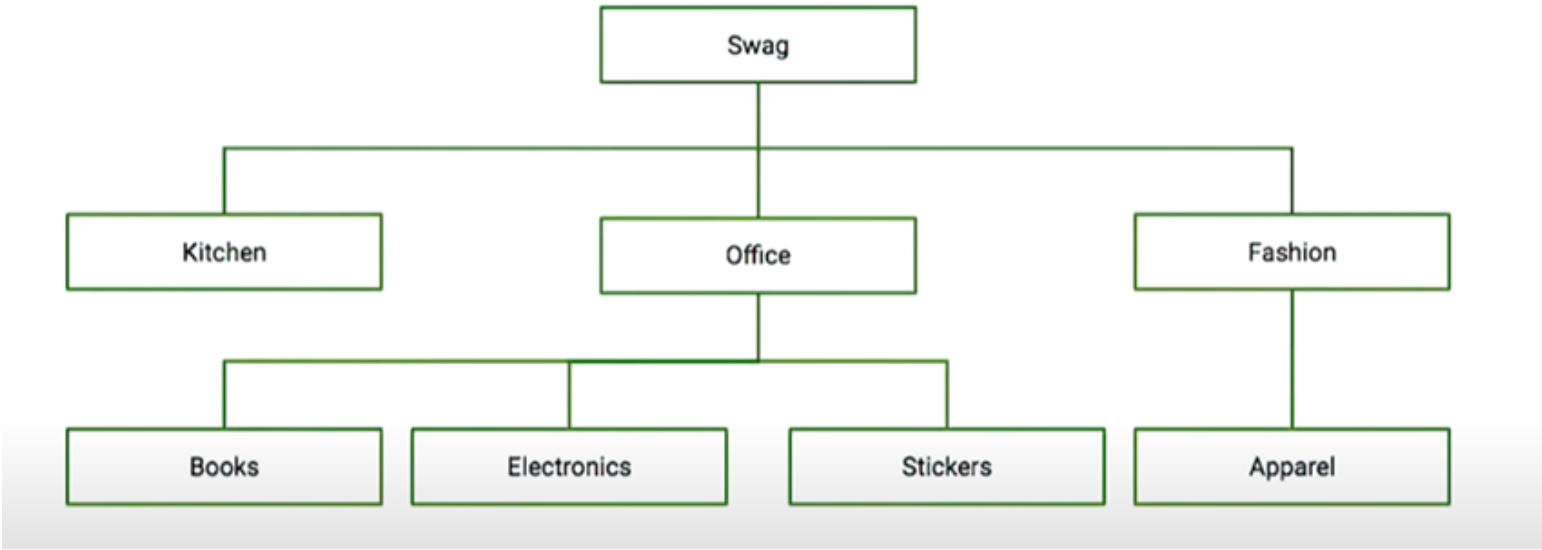
To find all nodes that are under Z, that is also pretty straightforward.

However, we have to iterate over a set of documents.

We can also respond to questions, Who are the ancestors of node X, by resorting to a lookup, which is essentially as joined, which may not be ideal to accomplish this operation.

07. Tree Patterns

Child References




```
{
    name : “Office”,
    Children : [“Books”, “Electronics”, “Stickers”],
    …
}
```

07. Tree Patterns

Child References

! Who are the ancestors of node X?

! Who report to Y?

✓ Find all nodes that are under Z?

! Change all categories under N to under P

07. Tree Patterns

Child References

In the child reference model, the parent node contains a single array of all the immediate node children.

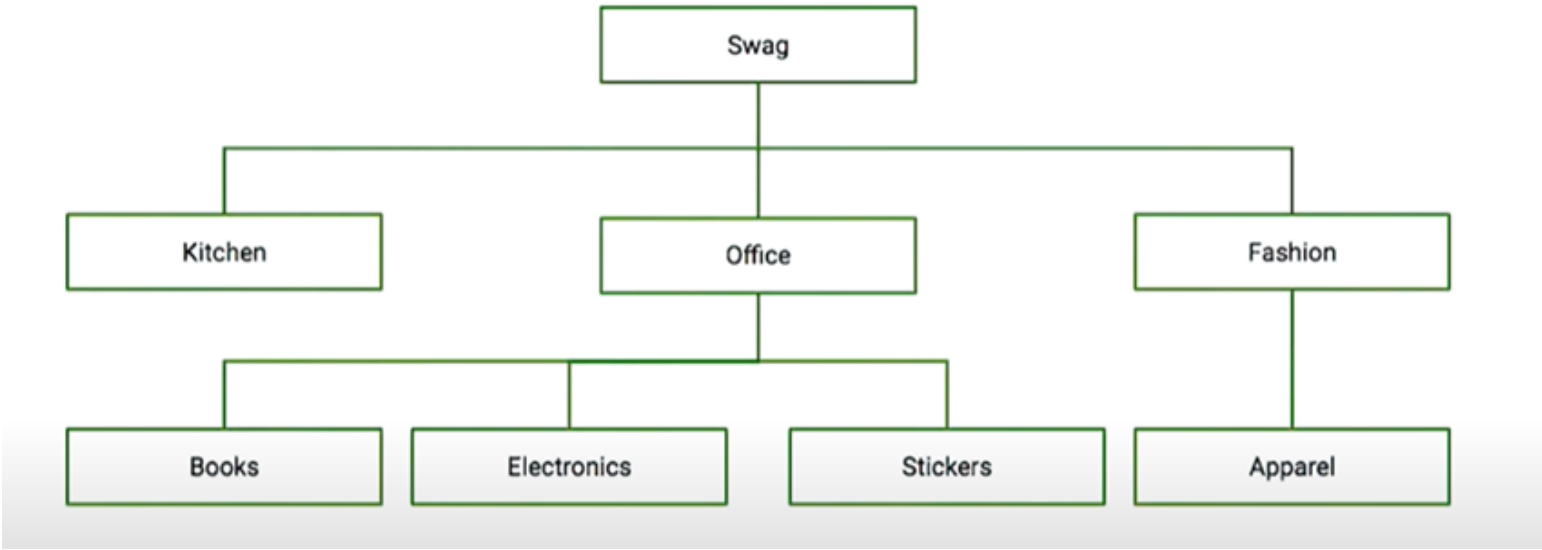
In this example, the office node has three children-- books, electronics, and stickers. So to perform the operation finding all nodes that are under Z, in this case, under office becomes a less computational demanding operation for the database.

A single request retrieves all of that information.

However, other questions like, who are the ancestors of node X, become a bit more complicated. Finding all nodes that report to Y or even changing all categories under N to under P are not ideal for this pattern.

07. Tree Patterns

Array of Ancestors




```
{
    Name: “Books”,
    Parent: [“Swag”, “Office”]
    …
}
```

07. Tree Patterns

Array of Ancestors

- ✓ Who are the ancestors of node X?
- ✓ Who report to Y?
- ✓ Find all nodes that are under Z?
- ! Change all categories under N to under P

07. Tree Patterns

Array of Ancestors

Another model to represent the hierarchy of a tree is the array of ancestors model. This model uses an ordered array to store a list of all of a node's ancestors on that node.

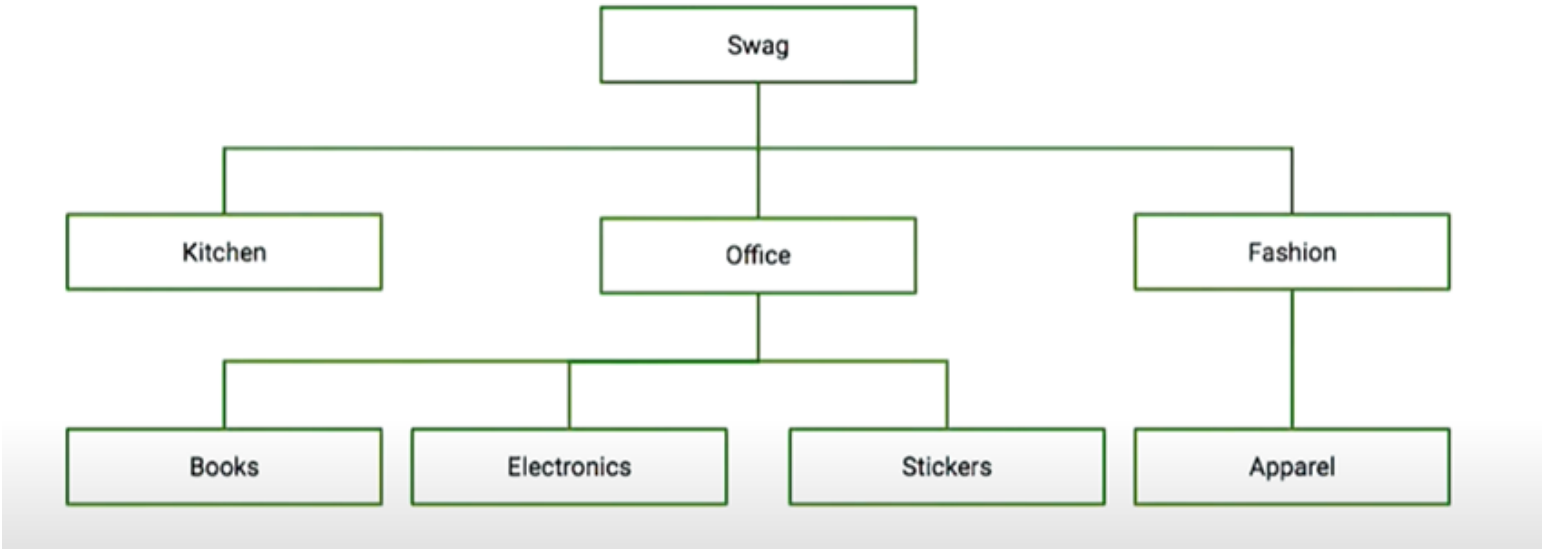
This model is very efficient for finding what are the ancestors of a given node. They are all stored in the ancestor array.

We can also find out which is the immediate reporting or parent node using array position operators if we know the level that the node occupies in the tree.

The next operation, which is still possible, a bit more tricky to do, is to change all categories under N to under P, since this operation requires a few more trips to the database to apply all the necessary changes.

07. Tree Patterns

Materialized Paths




```
{
    Name: “Books”,
    Parent: “.Swag.Office”
    …
}
```

```
// immediate ancestor of Y

db.categories.find({ancestors: /\.Y$/})

// if descends from X and Z

db.categories.find({ancestors: /^\.X.*Y/i})
```

07. Tree Patterns

Materialized Paths

A variation of the array ancestors is the materialized paths pattern. In this approach, we use a string value to describe the node's ancestors with some value separator.

In this example, we have a string ancestors with a value .Swag.Office. That describes the direct branch of the books node in the tree.

Using the materialized path's approach comes with a great benefit.

We can use a single regular expression over a single index field value for all queries prepended on the root tree node, which means that we can use a simple index in order to find a particular path of a branch in our hierarchy.

07. Tree Patterns

Materialized Paths

✓ Who are the ancestors of node X?

! Who report to Y?

! Find all nodes that are under Z?

! Change all categories under N to under P

07. Tree Patterns

Materialized Paths

In this approach, we can access our ancestor's fields to quickly answer the question, who are the ancestors of X?

For the question, who reports to Y, this is a simple query but potentially not very efficient, even relaying on a single field. The index will not be used if we do not ask for the full branch matching path from root.

Finding all nodes that are under Z suffers from the same problem as reports to Y.

And finally, moving nodes around might be quite challenging depending on the node that we are moving and the reports that subsequent node in the hierarchy chain.

07. Tree Patterns

MongoMark Solution = **Ancestor** Array + **Parent** Reference

```
{
        “_id” : 8,
        “name” : “Umbrellas”,
        “parent” : [ “Swag”, “Fashion”]
}
```

```
// Navigate up on the tree

db.categories.find({parent: X})

// Find all nodes under a given category

db.categories.find({ancestors: Y })
```

07. Tree Patterns

Problem

- Representation of hierarchical structured data;
- Different access patterns to navigate the tree;
- Provide optimized model for common operations

Solution

- Child Reference;
- Parent Reference;
- Array of Ancestors;
- Materialized Paths.

Use case Examples

- Org charts;
- Product Categories

Benefits and Trade-Offs

- Child Reference: easy to navigate to children nodes or tree descending.
- Parent Reference: immediate parent node discovery and tree updates.
- Array of Ancestors: Navigate upwards on the ancestors path.
- Materialized Paths: makes use of regular expressions to find nodes in the tree.

08. Polymorphic Pattern

08. Polymorphic Pattern

Products

Customers

08. Polymorphic Pattern

When we want to organize objects, we can either group them by what they have in common or by their differences. This will result in either keeping the documents in the same collection or in different collections.

For example, if we have a product from Canada, a product from Mexico, a customer from Canada, and a customer from Mexico, it is likely that we will group the two customers together and the two food items together. We usually group objects based on queries we want to perform on the system. A major restriction with most other base systems is that it is difficult to query objects across different tables or a collection.

If there is one collection for the customers and the one collection for the products it becomes more difficult to find all things Canadians, for example. If the main query of this system is that they're mining objects and people by country, then we will probably put together everything in one collection. So we should group things together because we want to query them together.

08. Polymorphic Pattern

```
{
  "vehicle_type": "car",
  ...
  "owner": "Yulia",
  "taxes": "200",
  "wheels": 4
}
```

```
{
  "vehicle_type": "truck",
  ...
  "owner": "Daniel",
  "taxes": "800",
  "wheels": 10,
  "axles": 3,
}
```

```
{
  "vehicle_type": "boat",
  ...
  "owner": "Norberto",
  "taxes": "2000"
}
```

08. Polymorphic Pattern

When grouping objects together that may have substantial differences, for example, using a vehicle collection to store cars, trucks, and boats, and other types of vehicles would lead us to query the vehicle for registration, ownership, vehicle taxes.

A specific vehicle type may have specific aspects that are not shared with the other types of vehicles. A car typically has four wheels, which we can omit. Other trucks have a variable number of wheels and axles. While the boat usually has none. These differences make a car or truck, and a boat polymorphic entities for a vehicle object.

The usual implementation represents polymorphic objects as a field that describes the name of this shape for a given document or sub-document. In our current example, a vehicle type identifies the document type and lets the application and all the expected fields for a particular type or shape of the document.

08. Polymorphic Pattern

The document of type car will imply that the vehicle has four wheels, while the truck should have a field to identify the number of wheels in the field to identify the number of axles. Due to the nature of the document model, there is usually polymorphous in our database.

We refer to the polymorphic pattern in our schema designs when we can clearly name the different entities that are represented in the same collection.

08. Polymorphic Pattern

```
{
  "product_type": "shirt",
  "color": "blue",
  "size": "large",
  "price: "100.00"
}
```

```
{
  "product_type": "book",
  "title": "Le Petit Prince",
  "size": "20cm x 15cm x 1cm",
  "price": "10.00"
}
```

08. Polymorphic Pattern

```
People
{
  "first_name": "Norberto",
  ...
  "addresses": [ {
    "street": "725 5th Ave"
    "city": "New York",
    "state": "NY",
    "zip_code": "10022",
    "country": "USA"
  },

  ...

 ]
}
```

```
People
{
  "first_name": "Norberto",
  ...
  "addresses": [ {
    "street": "725 5th Ave"
    "city": "New York",
    "state": "NY",
    "zip_code": "10022",
    "country": "USA"
  }, {
    "street": "Palacio de Belem",
    "city": "Lisbon",
    "postal_code": "1300-004",
    "country": "Portugal"
  } ]
}
```

08. Polymorphic Pattern

So far, we discussed a polymorphic document type in the context of the shape of a document. However, we can also apply this model to subentities, such as sub-documents.

For example, if we take a person, let's say, Norberto, Norberto is Portuguese and lives in New York. So he has an address in the USA. It is a typical urban address in the US, with a building number, street name, city name, and zip code.

Norberto has also an address in Portugal. This structure is similar to the address in the USA. However, a well-known building in Portugal may not have a building number. Portugal does not use zip codes. However, they have a critical post style, which translates to a postal code.

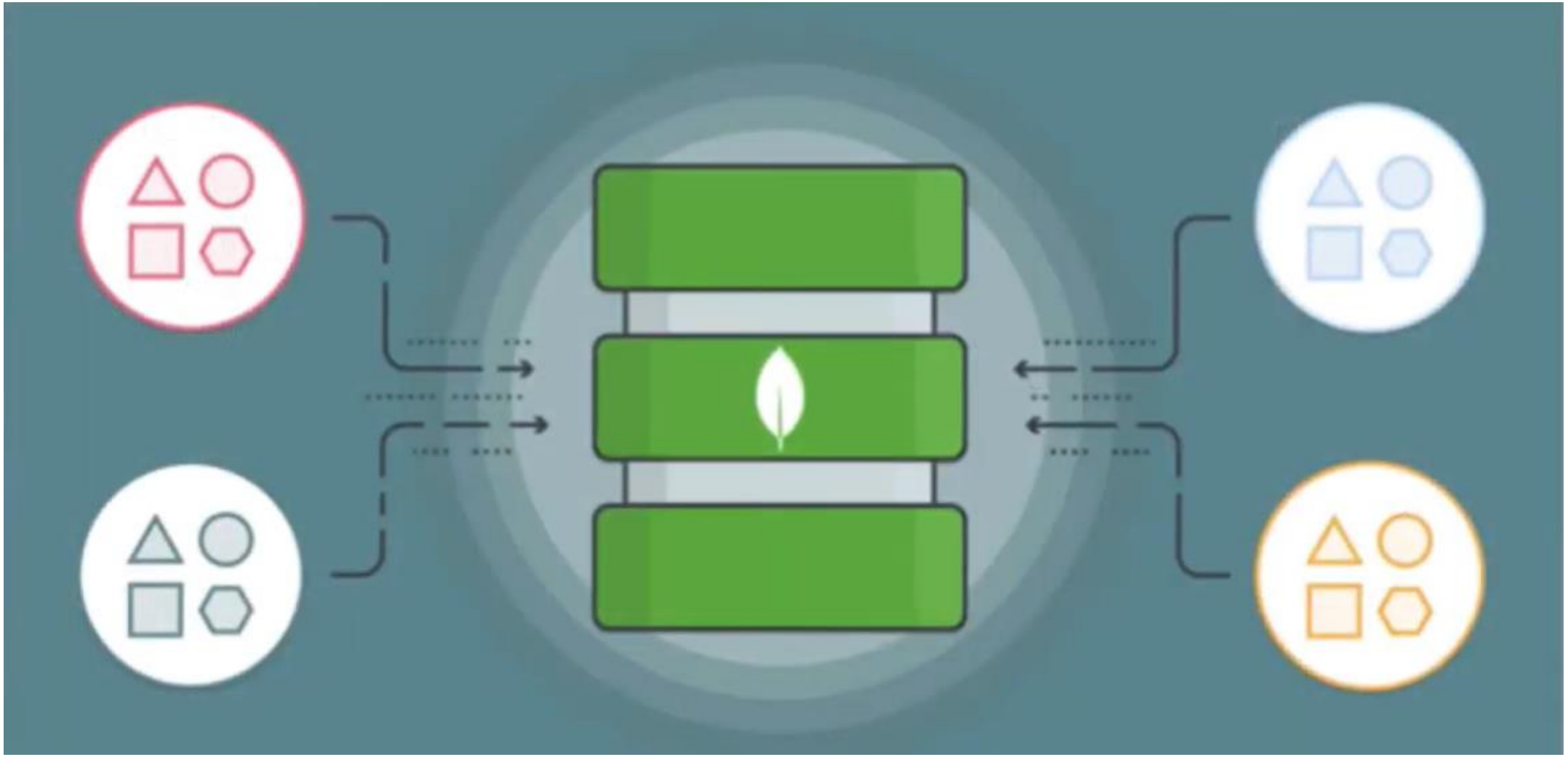
The critical post style is 7 digits long. So if we want to have valid data on the postal code or a zip code, they will have to be a little bit different, as they have to handle 7 and 5 digits, respectively.

08. Polymorphic Pattern

Because our addresses are the documents to which we want to apply the polymorphic pattern, each of these documents needs a field to specify its shape.

We could use an additional field, however, in this case, the country name should do it. Based on the country for a given address, we can determine the shape of that sub-document.

08. Polymorphic Pattern



08. Polymorphic Pattern

The polymorphic pattern is commonly used when implementing a single-view solution. A single-view solution allows for the aggregation of data from different sources into one central location. It is common that different sources exist because systems tend to be designed to deal with one single problem.

An integrated view is key to gaining meaningful insight across all this data, but it's often difficult to merge these data sets together. A typical tabular or relational database approach to solving this problem tends to be very complex, resource-intensive, and incomplete.

We've seen several organizations over the years failing to create a relational schema that could accommodate data from different sources, different relational database schema, or different data sets.

The complexity of this problem becomes greatly reduced when users decide to use MongoDB to create this unified view of the data.

08. Polymorphic Pattern

System A

```
{
  id: "203-102-1222",
  name: "Adams",
  first: "Samuel",
  address: "100 Forest",
  city: "Palo Alto",
  state: "California",
}
```

System B

```
{
  ss: "203-102-1222",
  last_name: "Adams",
  first_name: "Samuel",
  address: "100 Forest",
  city: "Palo Alto",
  state: "CA",
}
```

System C

```
{
  pkey: "123456",
  soc_sec: "203-102-1222",
  name: "Samuel Adams",
  street: "222 University",
  city: "Palo Alto",
  state: "CA",
}
```

Single View

```
{
  _id: "203-102-1222",
  insurance_types: ["life", "home", "car"],
  last_name: "Adams",
  first_name: "Samuel",
  addresses: [
    { address: "100 Forest",
      city: "Palo Alto", state: "California" },
    { address: "100 Forest",
      city: "Palo Alto", state: "CA" },
    { street: "222 University",
      city: "Palo Alto", state: "CA" }
  ] }
```

08. Polymorphic Pattern

Imagine an insurance company that has acquired many other insurance companies over the years. Each acquired organization represents its insurance policies in a different way, using a different schema in its tabular or relational database.

When a customer calls the insurance company, they need to specify which system the specific insurance policy is stored in and access the given system. It is possible that some customers had policies with several of these insurance companies, which means that they would have multiple policies across the different systems. For example, some home policy may be in System A, while his car policy may be in System B.

Imagine the mess it creates and the impact it has on the customer, who is not able to receive a single unified policy. By creating a single view that merges all sources of data into one MongoDB database, the organization is able to see all the policies and information about the given customer on their one single system.

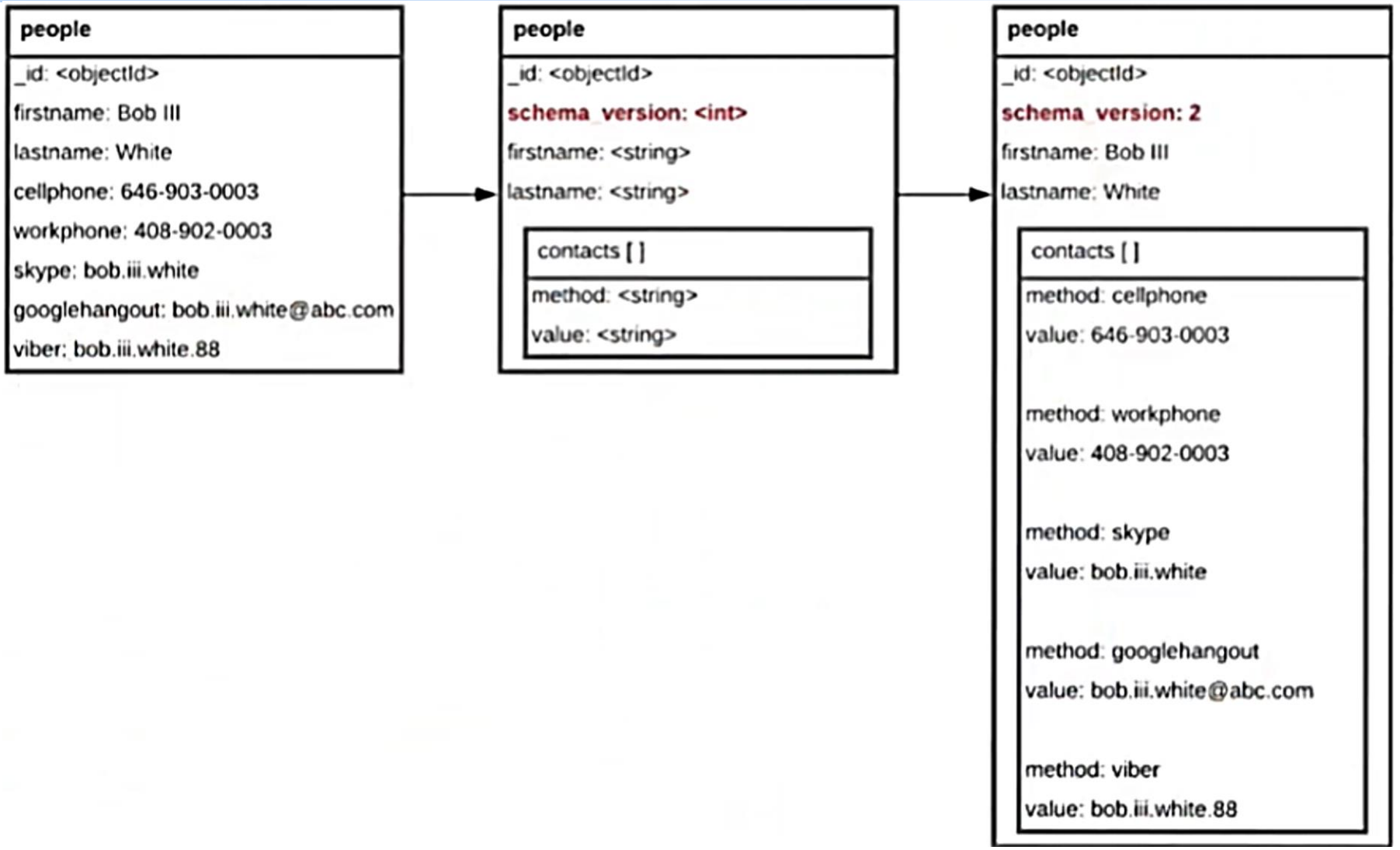
08. Polymorphic Pattern

In this example, we are bringing data from different sources into one common collection.

- First, we select the field that we want to use to identify the customer ID. Let's say we want to use the Social Security number.
- Second, maybe some fields are easy to merge. Let's say that the first and the last names of the person can be easily identified in the different sources, and they carry the same values. That will result in keeping only this shape for this field. Pick the field name you want, however, the values are the same.
- Finally, some fields may be more messy and difficult to merge. For example, let's say that addresses have proven to be difficult to clean and represent uniquely. For that, we can keep a list of variants found in the different databases.

We can confirm the right value with the customer later or write a separate intelligent process a few months down the road to unify these three sub-documents. This approach helps us be operational very quickly, and then we can improve the information we extract in an incremental fashion.

08. Polymorphic Pattern



08. Polymorphic Pattern

Problem

- Objects more similar than different;
- Want to keep objects in same collection

Solution

- Field tracks the type of document or sub-document;
- Application has different code paths per document type or has subclasses.

Use case Examples

- Single view;
- Product Catalog;
- Content Management.

Benefits and Trade-Offs

✓ Easier to implement;

✓ Allow to query across a single collection.

08. Polymorphic Pattern

Recap

- In summary, the polymorphic pattern is a basic pattern.
- A lot of other patterns are, in fact, a specialization of this pattern.
- We can also use the polymorphic pattern in a more generic way to store different shapes of documents within the same collection.

08. Lab - Polymorphic Pattern

1. User Story:

- Our company has been selling books online for many years. Recently, we decided to expand and acquired the rights to sell the books as e-books and as audiobooks.

- Currently, the books in our catalog looks like the example in the **4. Problem-Document 1**. E-books and audiobooks in our catalog look like the examples in the **5. Problem-Document 2**.

- We decided to put all the documents, one per media type, in the same collection to minimize the impact on the current applications. As we retrieve a document, we can forward the information about the product to the right Web page to render the information about the book in a given format.

- Please review the **3. Problem-Schema** and the two different Problem Document to help understand the old schema and how we need to change the schema to apply the pattern.

08. Lab - Polymorphic Pattern

2. Task:

To address the schema changes, we will use the Polymorphic Pattern to identify the shape of each document depending on its format.

This will require a modification to the current schema so that all documents will share certain common elements:

- All documents will have a new field called format that will have a value of book, ebook, or audiobook;
- Across all documents, we will have a field called product_id that will accept an integer as its value;
- The field description should be unified across all documents;
- The field authors should be unified across all documents.

08. Lab - Polymorphic Pattern

2. Task:

Please review the **6. Problem-Schema** and the **7. Pattern-Polymorphic** to help understand the layout for the new schema.

Start by creating a field format which is limited to a string value of book, or ebook, or audiobook. Next, add a field product_id to the ebook and audiobook documents. In order to unify the description across documents, you can either rename the field desc to description for the ebook and audiobook or you make a copy of the contents of desc into description to keep backward compatibility, either approach will work to unify the description.

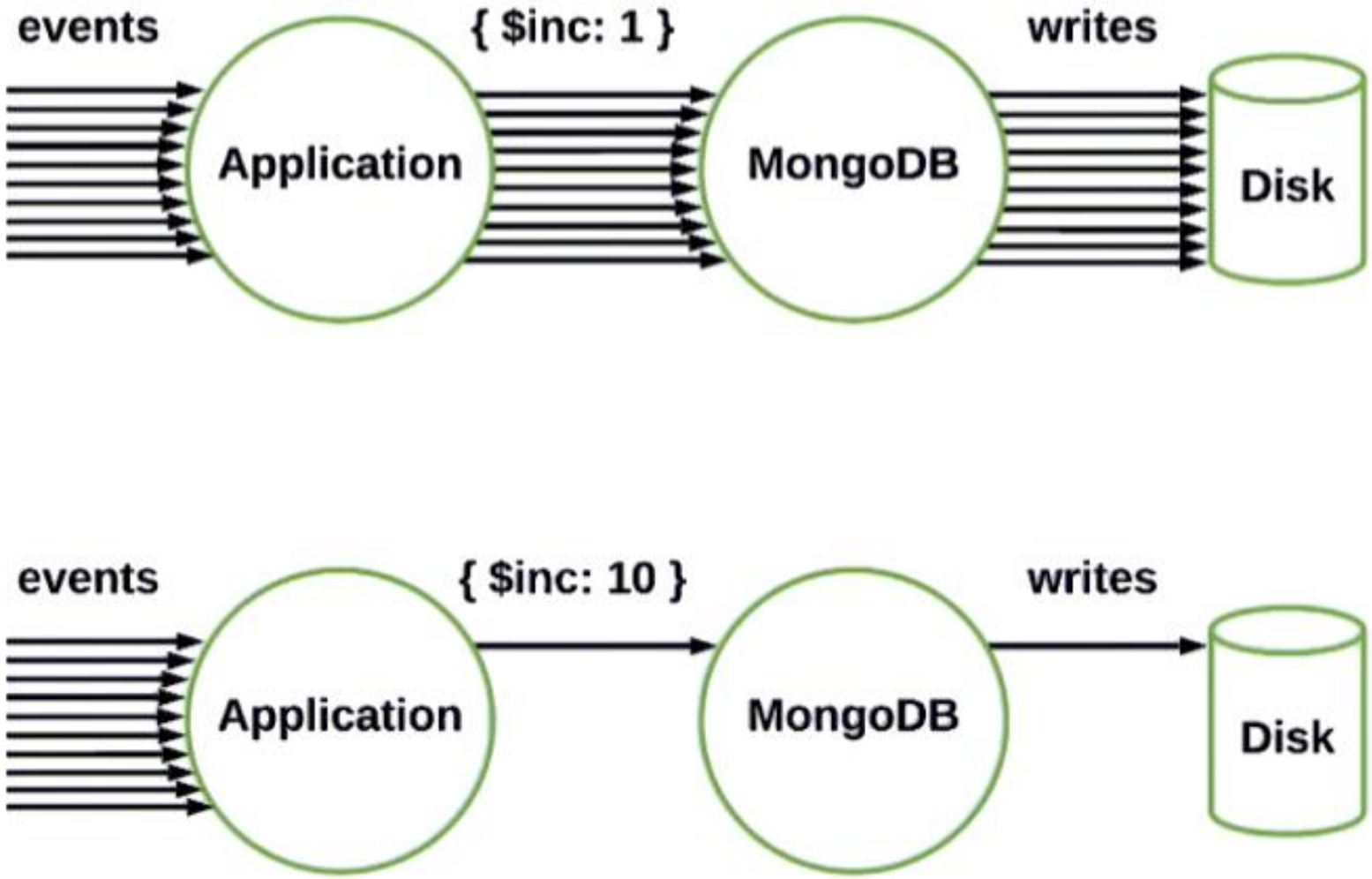
Finally, for the authors, we need to turn the author of the audiobook documents from a string to an array of strings.

09. Other Pattern

- **Approximation** Pattern
- **Outlier** Pattern

09. Other Pattern

Approximation Pattern



09. Other Pattern

Approximation Pattern

Problem

- Data is expensive to calculate;
- It does not matter if the number is not precise.

Solution

- Fewer, but writes with higher payload

Use case Examples

- Web page counters;
- Any counters with tolerance to imprecision;
- Metric statistics.

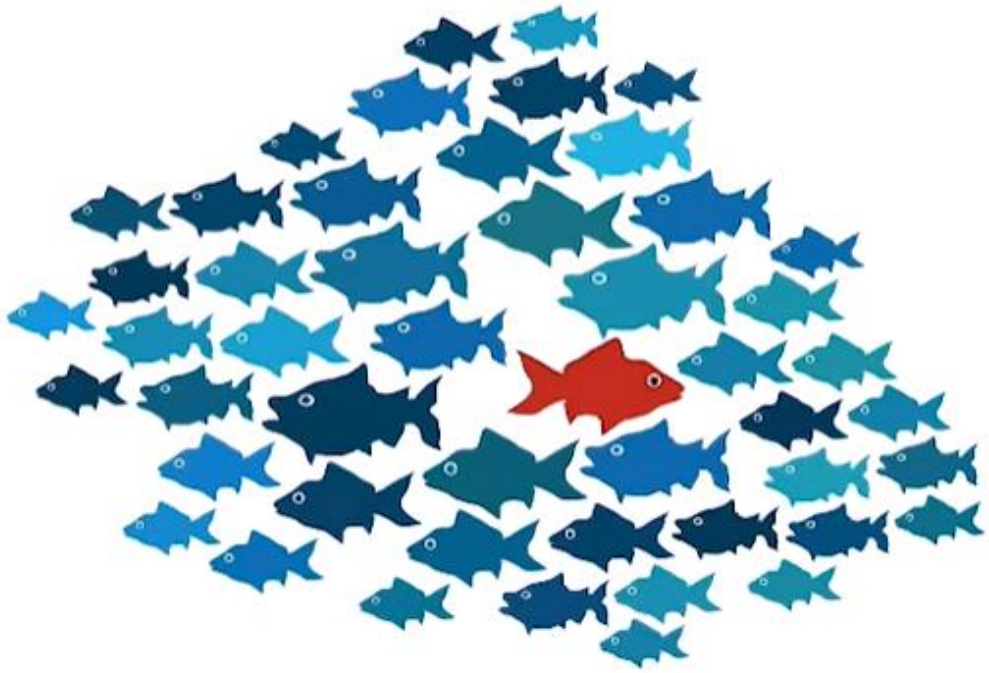
Benefits and Trade-Offs

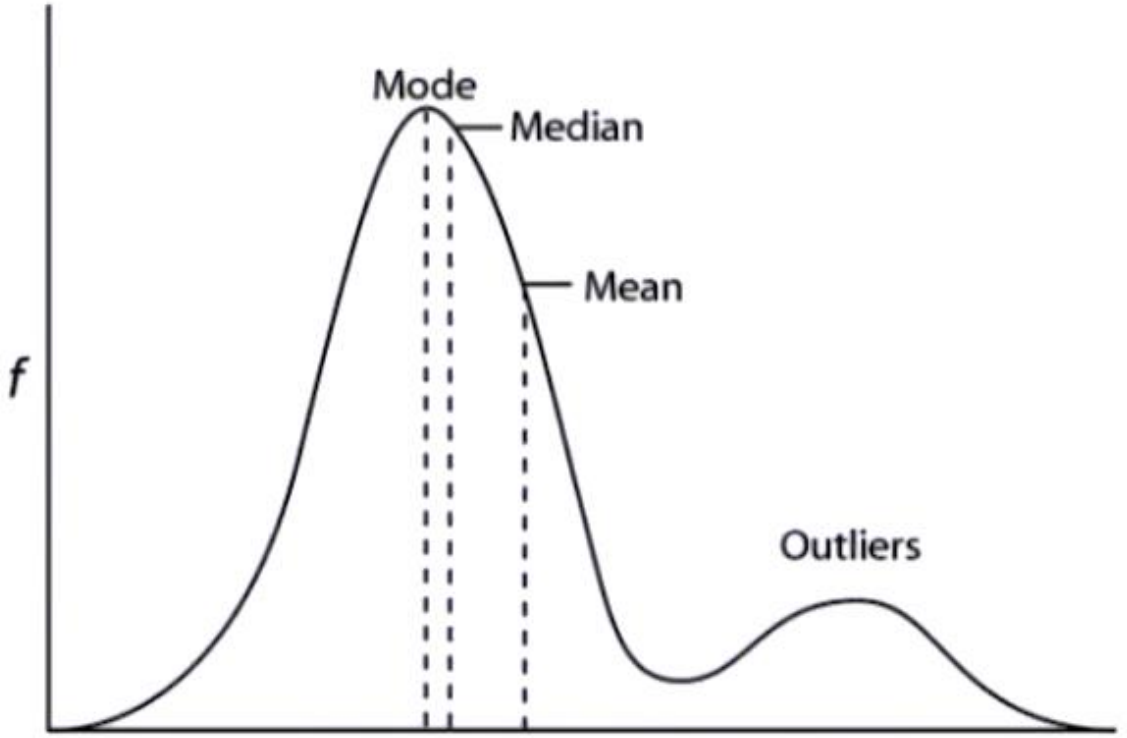
✓ Less writes;
✓ Less contention on documents;
✓ Statistics valid numbers;

☒ Not exact numbers;
☒ Must be implemented in the application.

09. Other Pattern

Outlier Pattern





09. Other Pattern

Outlier Pattern

```
Collection: movie_extras
{
  "_id": 1980001234,
  "title": "Gandhi",
  "has_overflow_extras": true,
  "extras": [
    { "first_name": "Angshuman",
      "Last_name": "Bagchi" },
    { "first_name": "Rupa",
      "last_name": "Narayanan" },
    ...
  ],
  number_extras: 1000
}
```

```
{
  "_id": 1980001234001,
  "movie_id": 1980001234,
  "title": "Gandhi",
  "is_overflow_extra": true,
  "extras": [
    { "first_name": "Sachin",
      "Last_name": "Tendulkar" },
    { "first_name": "Sourav",
      "last_name": "Ganguly" },
    ...
  ],
  number_extras: 820
}
```

09. Other Pattern

Outlier Pattern

Problem

- Few documents would drive the solution;
- Impact would be negative on the majority of queries.

Solution

- Implementation that works for majority;
- Field identifies outliers as exception;
- Outliers are handled differently on the application side.

Use case Examples

- Social Networks;
- Popularity.

Benefits and Trade-Offs

✓ Optimized solution for most use cases;

☒ Differences handled application side;

☒ Difficult for aggregation of ad-hoc queries.

09. Other Pattern

Recap

- **Approximation** Pattern
  - Avoiding performing an operation too often
- **Outlier** Pattern
  - Keeping the focus on the most frequent use cases