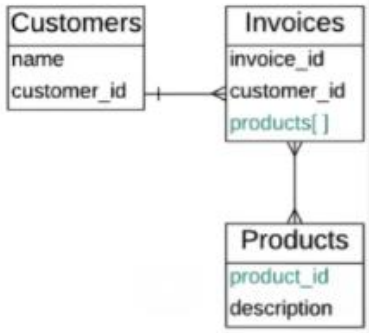
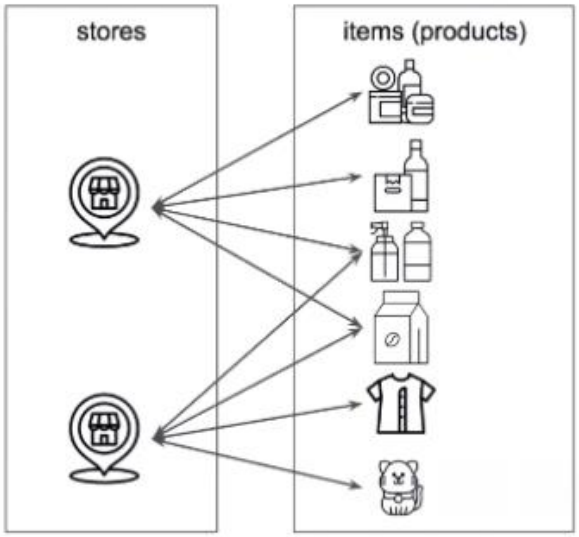
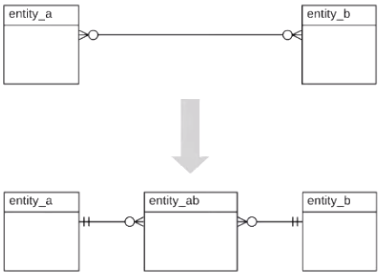
Common Relationships

- one-to-one (1-1)
- one-to-many (1-N)
- many-to-many (N-N)





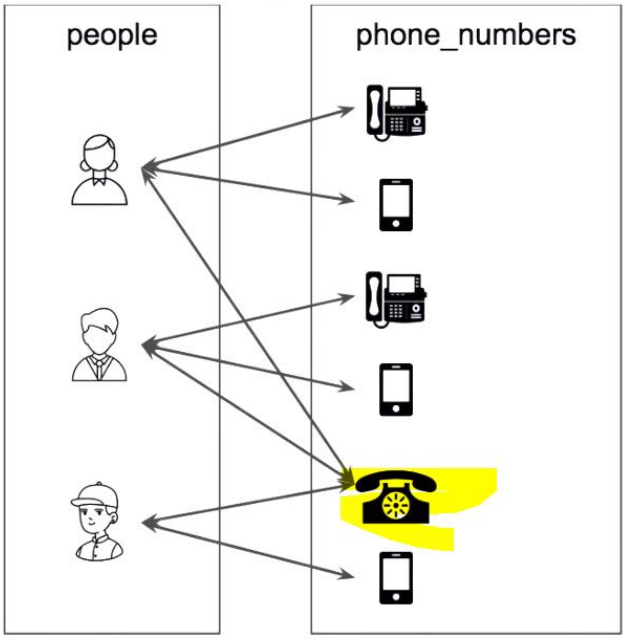
Many-to-Many => 2 x One-to-Many

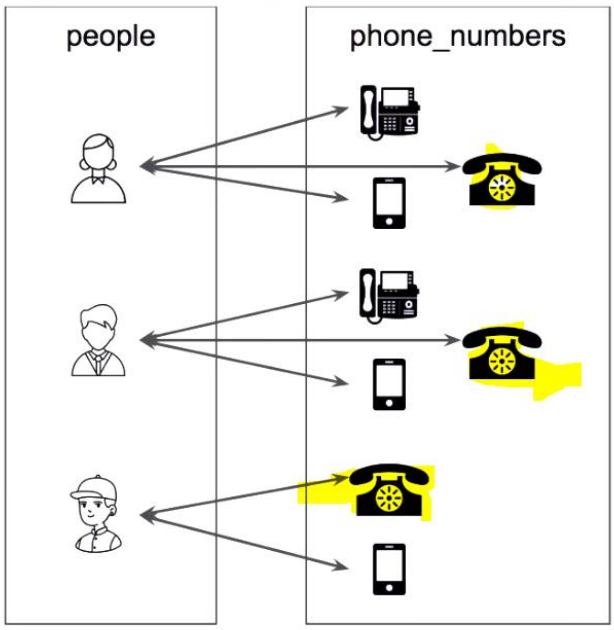


actor_id	actor_name
actor1	Marlon Brando
actor2	Al Pacino
actor3	Robert DeNiro

movie_id	actor_id
movie1	actor1
movie1	actor2
movie2	actor2
movie2	actor3

movie_id	movie_title
movie1	The Godfather
movie2	The Godfather 2


people
phone_numbers


people
phone_numbers

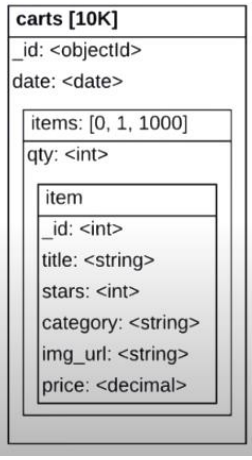
Many-to-Many Representations

1. Embed
   a. array of subdocuments in the "many" side
   b. array of subdocuments in the other "many" side

   Usually, only the most queried side is considered

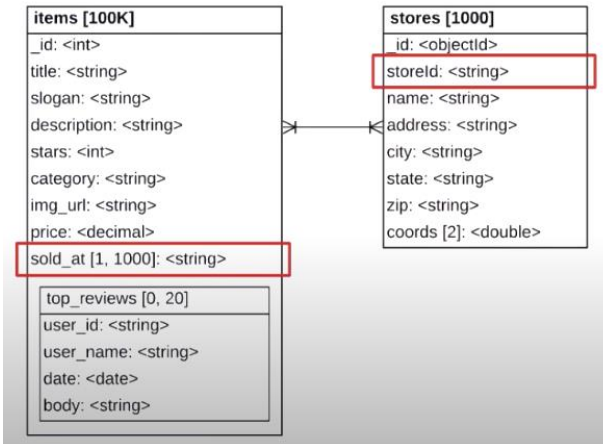
2. Reference
   a. array of references in one "many" side
   b. array of references in the other "many" side

Many-to-Many: embed, in the main side



- the documents from the less queried side are embedded
- results in duplication
- keep "source" for the embedded documents in another collection
- indexing is done on the array

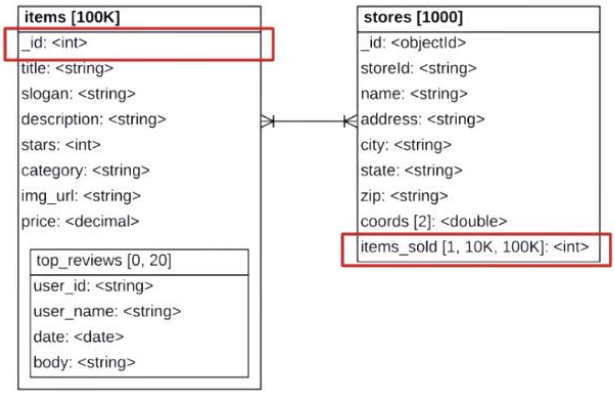
Many-to-Many: reference, in the main side



- array of references to the documents of the other collection
- references readily available upon first query on the "main" collection

```
db.stores.find({_id: {$in: ["store1", "store2", ...]}})
```

Many-to-Many: reference, in the secondary side

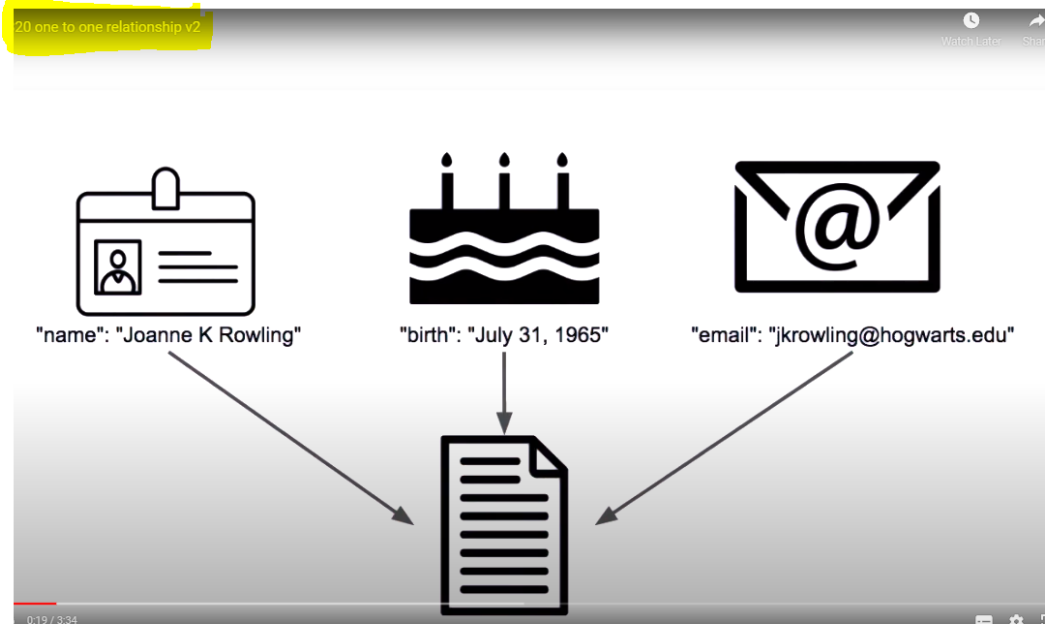


- array of references to the documents of the other collection
- need a second query to get more information

```
db.stores.find({items_sold: 10})
```

Recap for the Many-to-Many Relationships:

- Ensure it is a "many-to-many" relationship that should not be simplified.
- A "many-to-many" relationship can be replaced by two "one-to-many" relationships but does not have to with the document model
- Prefer embedding on the most queried side
- Prefer embedding for information that is primarily static over time and may profit from duplication.
- Prefer referencing over embedding to avoid managing duplication.



One-to-One Representations

1. Embed
   a. fields at same level
   b. grouping in sub-documents

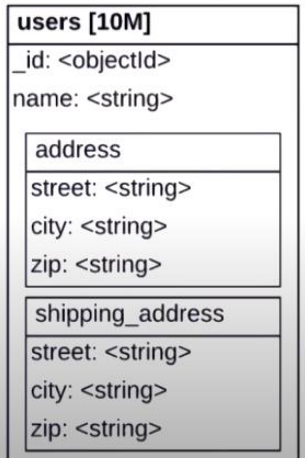
2. Reference
   a. same identifier in both documents
   b. in the main "one" side
   c. in the secondary "one" side

One-to-One: embed, fields at the same level

- very similar to tabular databases

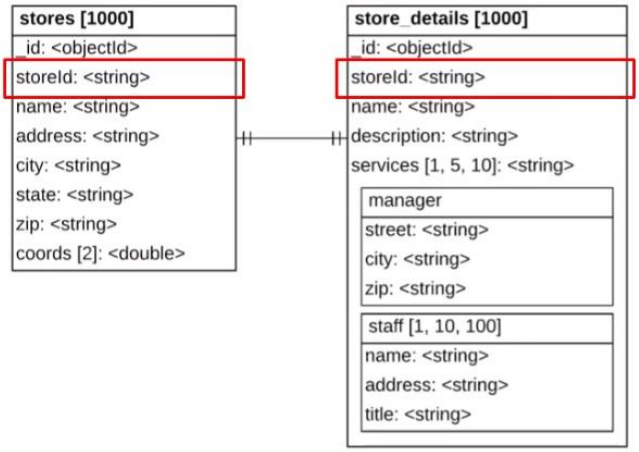
users [10M]
_id: <objectId>
name: <string>
street: <string>
city: <string>
zip: <string>
shipping_street: <string>
shipping_city: <string>
shipping_zip: <string>

One-to-One: embed, using subdocuments



- preferred representation:
  - preserves simplicity
  - documents are clearer

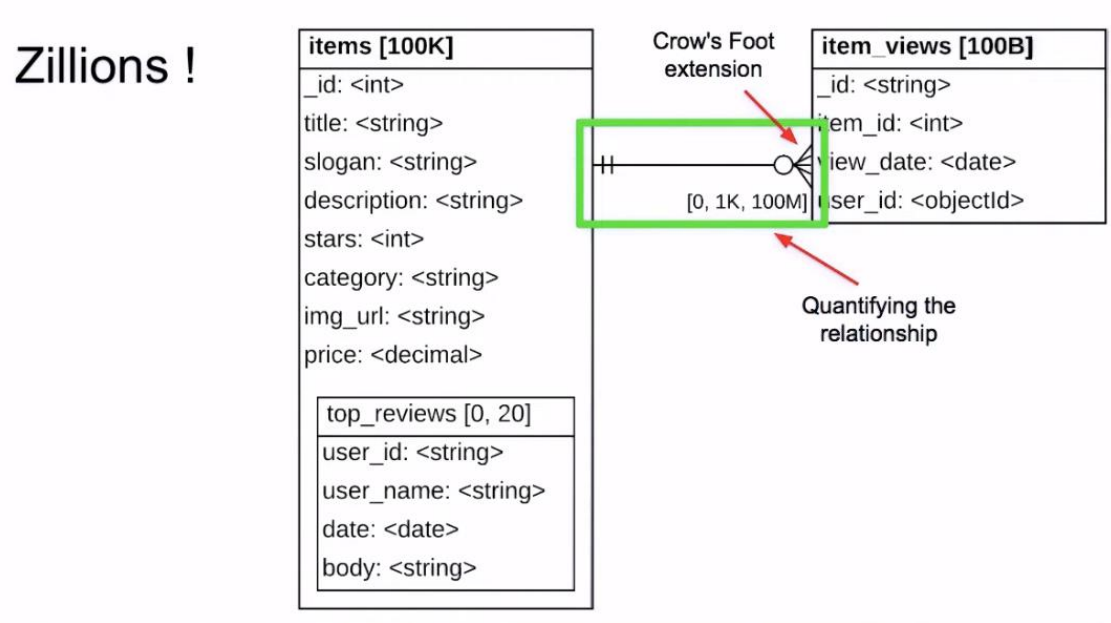
One-to-One: reference

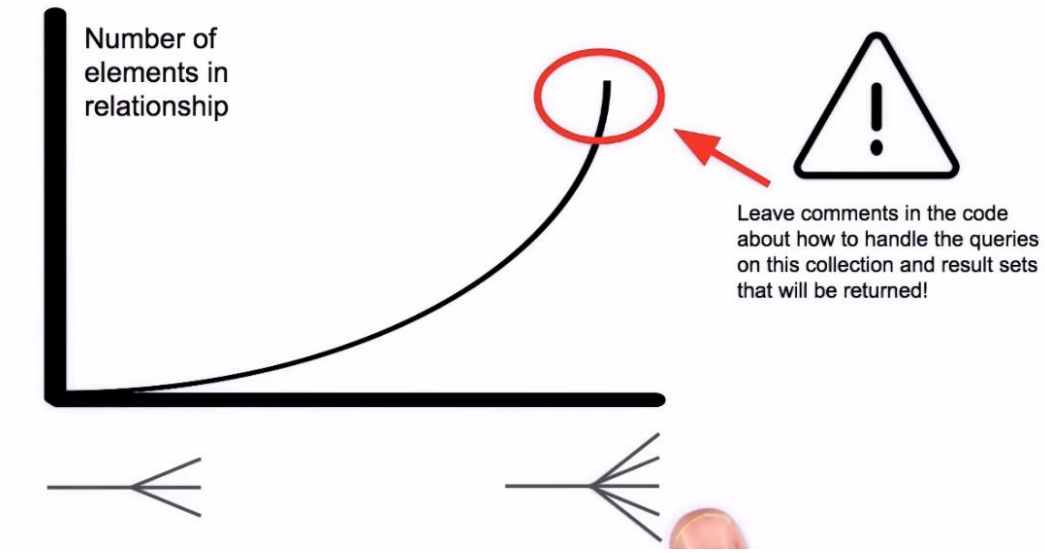


- add complexity
- possible performance improvements with:
  - smaller disk access
  - smaller amount of RAM needed

Recap for the One-to-One Relationships:

- Prefer embedding over referencing for simplicity
- Use subdocuments to organize the fields
- Use a reference for optimization purposes


Zillions !
items [100K]
_id: <int>
title: <string>
slogan: <string>
description: <string>
stars: <int>
category: <string>
img_url: <string>
price: <decimal>
top_reviews [0, 20]
user_id: <string>
user_name: <string>
date: <date>
body: <string>
Crow's Foot extension
[0, 1K, 100M]
item_views [100B]
_id: <string>
em_id: <int>
iew_date: <date>
ser_id: <objectId>
Quantifying the relationship

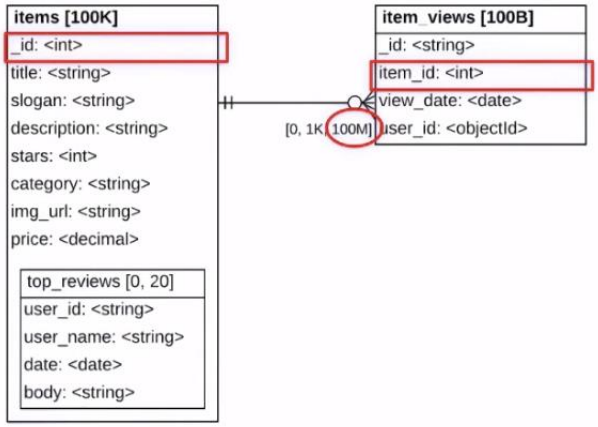

Number of elements in relationship
Leave comments in the code about how to handle the queries on this collection and result sets that will be returned!

One-to-Zillions Representations

1. Embed
   a. ~~in the "one" side~~
   b. ~~in the "many" side~~

2. Reference
   a. ~~in the "one" side~~
   b. in the "many/zillions" side

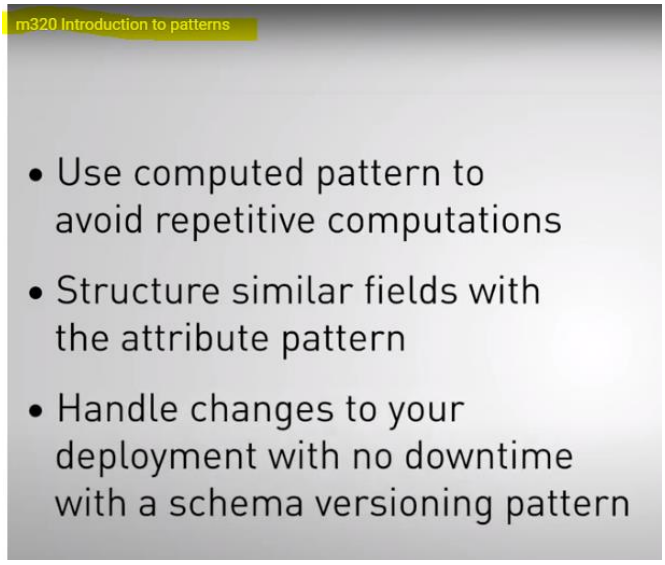
One-to-Zillions: reference, in the "zillions" side



- sole representation for a one-to-zillions relationship
- quantify the relationship to understand the maximum N value

Recap for the One-to-Zillions Relationships:

- It is a particular case of the one-to-many relationship.
- The only available representation is to reference the document on the “one” side of the relationship from the "zillion" side.
- Pay extra attention to queries and code that handle "zillions" of documents.



Applying Patterns may lead to...

1. Duplication

   duplicating data across documents

2. Data Staleness

   accepting staleness in some pieces of data

3. Data Integrity Issues:

   writing extra application side logic to ensure referential integrity

1. Duplication

Why?

- Result of embedding information in a given document for faster access

Concern?

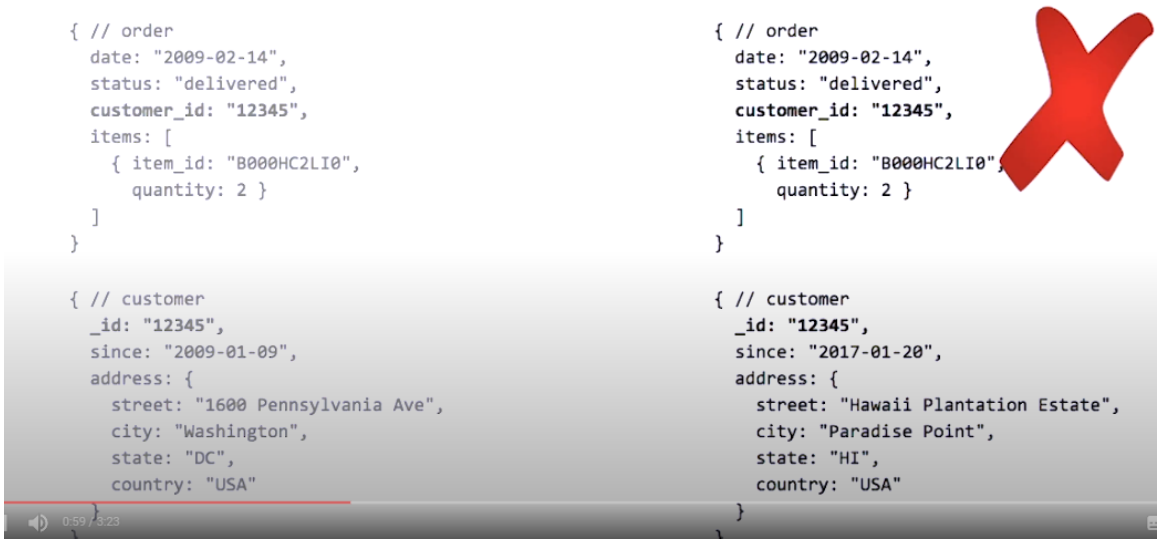
- Challenge for correctness and consistency

1. Duplication - a. is the solution

```
{ // order
  date: "2009-02-14",
  status: "delivered",
  customer_id: "12345",
  items: [
    { item_id: "B000HC2LI0",
      quantity: 2 }
  ]
}

{ // customer
  _id: "12345",
  since: "2009-01-09",
  address: {
    street: "1600 Pennsylvania Ave",
    city: "Washington",
    state: "DC",
    country: "USA"
  }
}
```

1. Duplication - a. is the solution



1. Duplication - a. is the solution

```
{ // order
  date: "2009-02-14",
  status: "delivered",
  customer_id: "12345",
  items: [
    { item_id: "B000HC2LI0",
      quantity: 2 }
  ]
  shipping_address: {
    street: "1600 Pennsylvania Ave",
    city: "Washington",
    state: "DC",
    country: "USA"
  }
}
```

```
{ // order
  date: "2017-01-21",
  status: "delivered",
  customer_id: "12345",
  items: [
    { item_id: "B07CJWLLNX",
      quantity: 1 }
  ]
  shipping_address: {
    street: "Hawaii Plantation Estate",
    city: "Paradise Point",
    state: "HI",
    country: "USA"
  }
}
```

1. Duplication - b. has minimal effect

```
{ // movie
  _id: "tt0076759",
  title: "Star Wars - IV - A New Hope",
  cast: [
    "nm0000434",
    "nm0000148",
    ...
  ]
}

{ // actor
  _id: "nm0000434",
  name: "Mark Hamill",
  filmography: [
    { "tt0076759": "Luke Skywalker" },
    { "tt0080684": "Luke Skywalker" },
    ...
  ]
}
```

```
{ // movie 1
  title: "Star Wars - IV - A New Hope",
  cast: [
    { "Mark Hamill": "Luke Skywalker" },
    { "Harrison Ford": "Han Solo" },
    ...
  ]
}

{ // movie 2
  title: "Star Wars - V - The Empire Strikes Back",
  cast: [
    { "Mark Hamill": "Luke Skywalker" },
    { "Harrison Ford": "Han Solo" },
    ...
  ]
}
```

1. Duplication - c. should be handled

```
{ // movie
  _id: "tt0076759",
  title: "Star Wars - IV - A New Hope",
  gross_revenues: 775000000
}

{ // screenings
  date: "1977-05-30",
  revenues: 15554475,
}
...
{ // screenings
  date: "2019-05-30",
  revenues: 25000,
}
```

1. Duplication - c. should be handled

```
{ // movie
  _id: "tt0076759",
  title: "Star Wars - IV",
  gross_revenues: 775000000
}

{ // screenings
  date: "1977-05-30",
  revenues: 15554475,
}
...
{
  date: "2019-05-30",
  revenues: 5000,
}
```

```
{ // movie
  _id: "tt0076759",
  title: "Star Wars - IV",
  gross_revenues: 775000000
}

{ // screenings
  date: "1977-05-30",
  revenues: 15554475,
}
...
{
  date: "2019-05-30",
  revenues: 25000,
}
...
{
  date: "2019-06-17",
  revenues: 25000,
}
```

```
{ // movie
  _id: "tt0076759",
  title: "Star Wars - IV",
  gross_revenues: 775025000
}

{ // screenings
  date: "1977-05-30",
  revenues: 15554475,
}
...
{
  date: "2019-05-30",
  revenues: 25000,
}
...
{
  date: "2019-06-17",
  revenues: 25000,
}
```

m320 handling duplication part 3

2. Staleness

Why?

- New events come along at such a rate that updating some data constantly causes performance issues.

Concern?

- Data quality and reliability.

Long analytic queries?

Queries on secondary nodes?

2. Resolve Staleness

Batch Updates

Find the changes through a Change Stream

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" }
  ]
}
```

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"}})
```

Attribute Pattern

Problem

- Lots of similar fields
- Want to search across many fields at once
- Fields 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 Examples

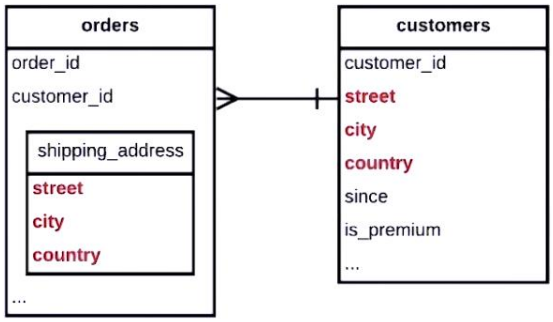
- Characteristics of a product
- Set of fields all having same value type
  - List of dates

Benefits and Trade-Offs

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

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



Extended Reference Pattern

Problem

- Too many repetitive joins

Solution

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

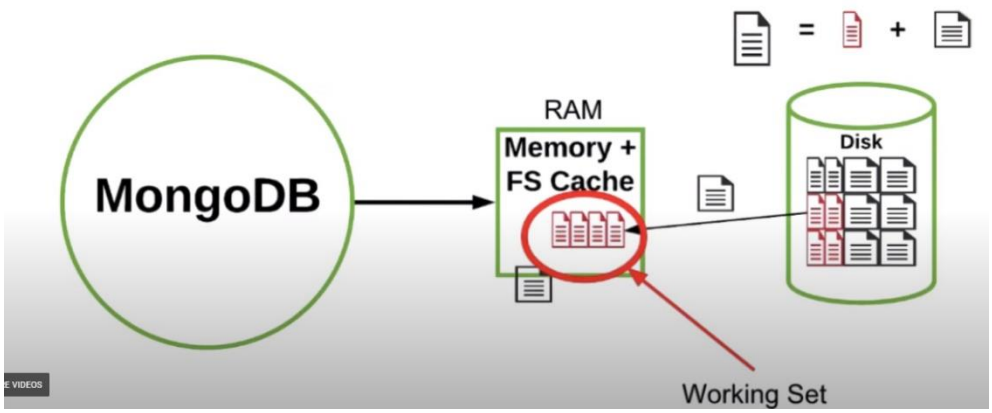
Use Cases Examples

- Catalog
- Mobile Applications
- Real-Time Analytics

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

Reduce the size of the Working Set



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
  - Less used part of documents
- Duplicate part of a 1-N or N-N relationship that is often used in the most used side

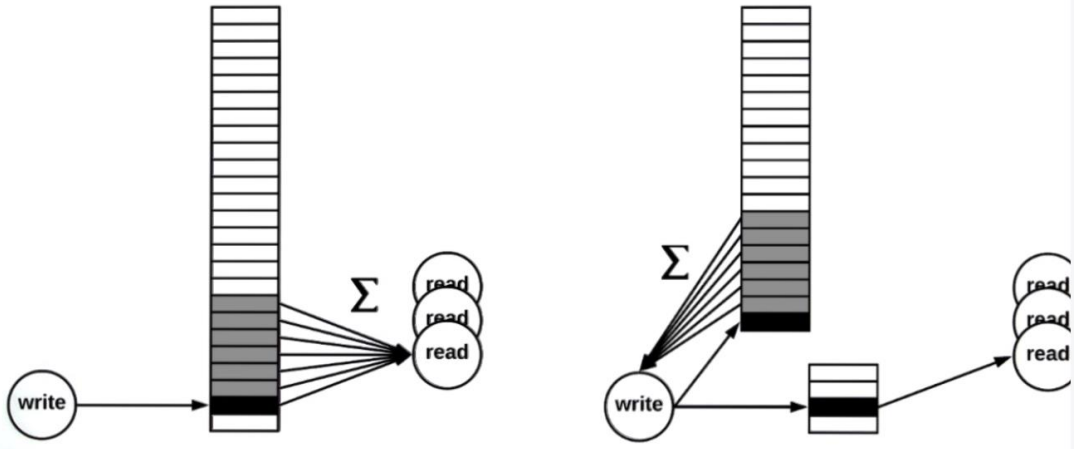
Use Cases Examples

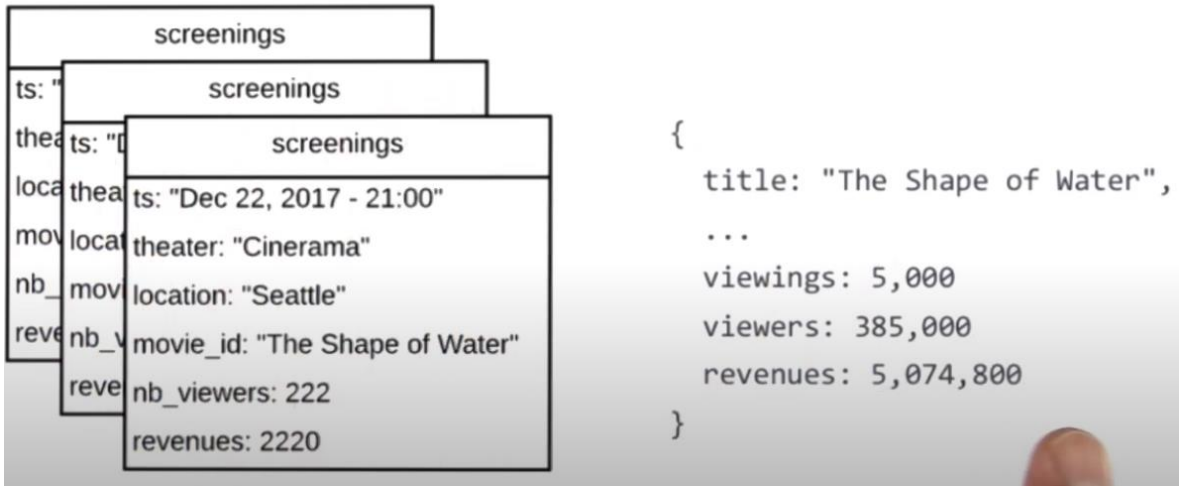
- List of reviews for a product
- List of comments on an article
- List of actors in a movie

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

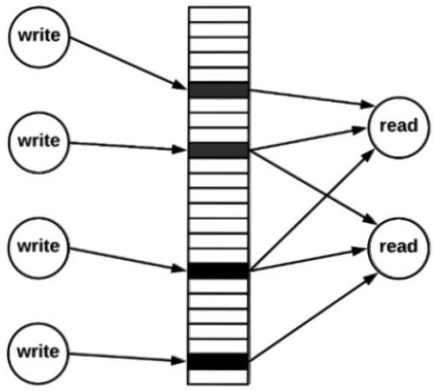
Mathematical Operations



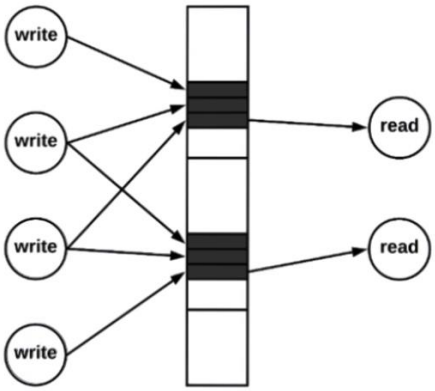


Fan Out Operations

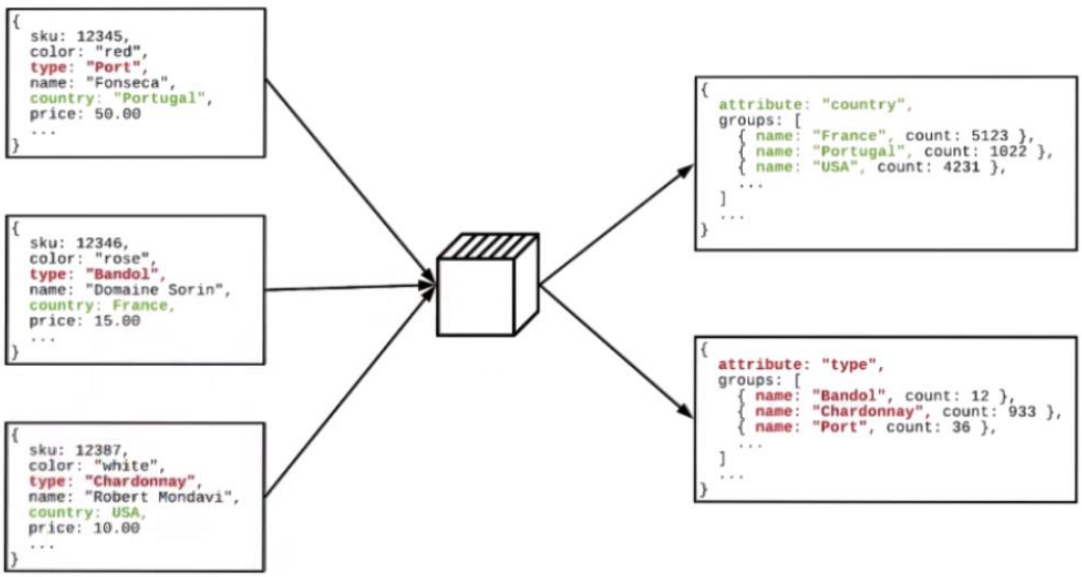
Fan Out on Reads



Fan Out on Writes



Example of Roll Up



Computed Pattern

Problem

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

Solution

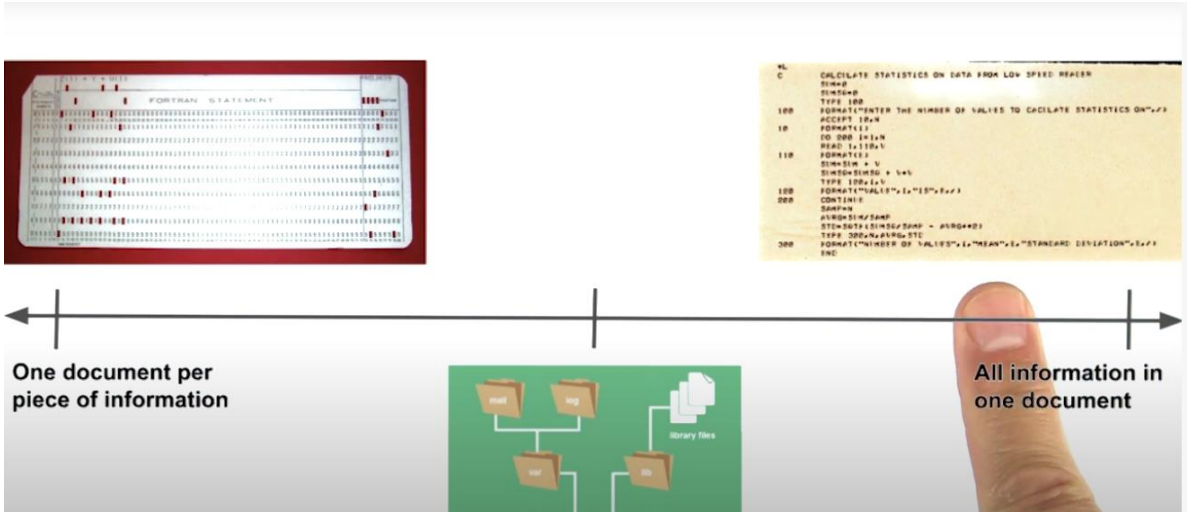
- Perform the operation and store the result in the appropriate document and collection
- If need to redo the operations, keep the source of them

Use Cases 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



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, ... }
}
```

MORE VIDEOS

Collaboration Platform Example

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

```
{
  "channel_id": "13579246",
  "name": "mongodb_tech",
  "date": ISODate("2018-03-02T03:14:16"),
  "message": "<Daniel joined the group>"
}

{
  "channel_id": "13579246",
  "name": "mongodb_tech",
  "messages": [
    "<Channel created>",
    "<Nathan joined the group>",
    "<Daniel joined the group>",
    "Nathan: Welcome Daniel",
    "Daniel: Thanks!"
  ...
  ]
}
```

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"),
  "messages": [
    "<Daniel joined the group>",
    "Nathan: Welcome Daniel",
    "Daniel: Thanks!"
  ...
  ]
}
```

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"}}}])
```

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

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.

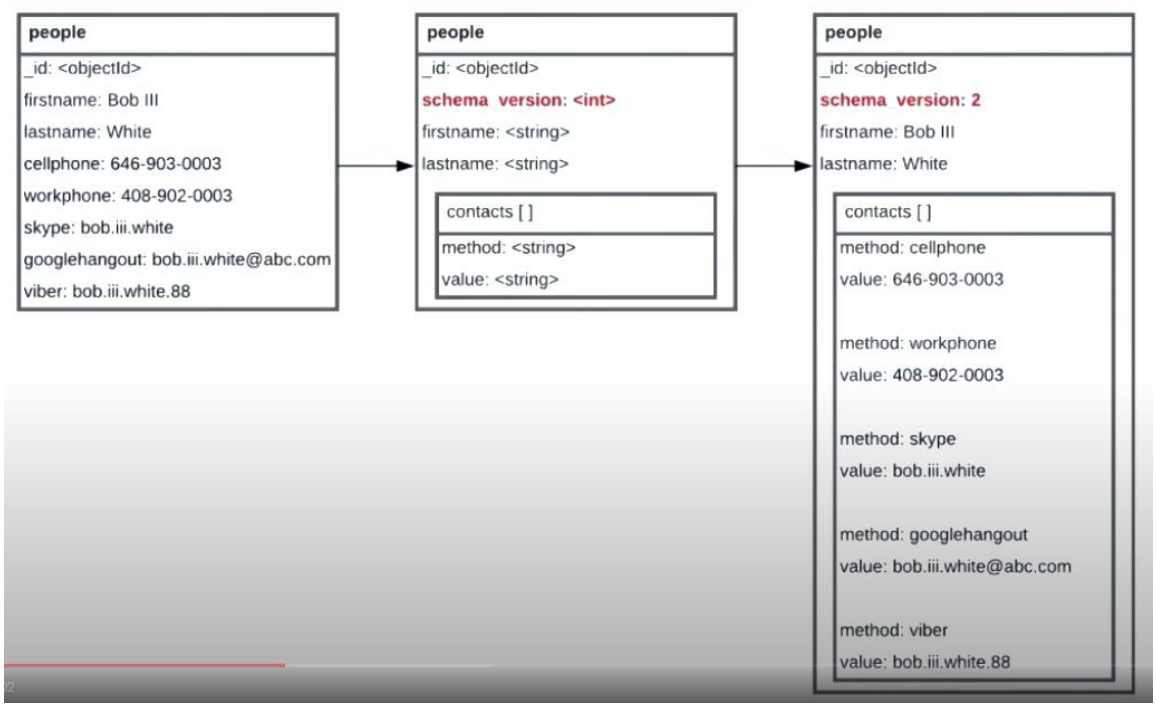
Use Cases Examples

- Internet Of Things
- Data Warehouse
- Lots of information associated to one object

Benefits and Trade-Offs

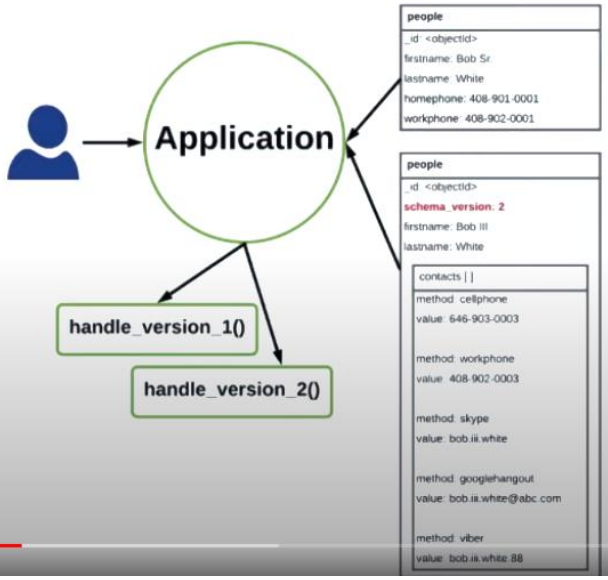
- ✅ Good balance between number of data access and size of data returned
- ✅ Makes data more manageable
- ✅ Easy to prune data
- ❌ Can lead to poor query results in not designed correctly
- ❌ Less friendly to BI Tools


m320 Schema Versioning
people
_id: <objectId>
firstname: Bob Sr.
lastname: White
homephone: 408-901-0001
workphone: 408-902-0001
people
_id: <objectId>
firstname: Bob Jr.
lastname: White
homephone: 408-901-0002
workphone: 408-902-0002
cellphone: 646-903-0002
people
_id: <objectId>
firstname: Bob III
lastname: White
cellphone: 646-903-0003
workphone: 408-902-0003
skype: bob.iii.white
googlehangout: bob.iii.white@abc.com
viber: bob.iii.white.88
people
_id: <objectId>
firstname: <string>
lastname: <string>
homephone: <string>
workphone: <string>
people
_id: <objectId>
firstname: <string>
lastname: <string>
homephone: <string>
workphone: <string>
cellphone: <string>
people
_id: <objectId>
firstname: <string>
lastname: <string>
homephone: <string>
workphone: <string>
cellphone: <string>
skype: <string>
googlehangout: <string>
viber: <string>
VIDEOS
2:01 / 7:02


people
_id: <objectId>
firstname: Bob III
lastname: White
cellphone: 646-903-0003
workphone: 408-902-0003
skype: bob.iii.white
googlehangout: bob.iii.white@abc.com
viber: bob.iii.white.88
people
_id: <objectId>
schema version: <int>
firstname: <string>
lastname: <string>
contacts [ ]
method: <string>
value: <string>
people
_id: <objectId>
schema version: 2
firstname: Bob III
lastname: White
contacts [ ]
method: cellphone
value: 646-903-0003
method: workphone
value: 408-902-0003
method: skype
value: bob.iii.white
method: googlehangout
value: bob.iii.white@abc.com
method: viber
value: bob.iii.white.88

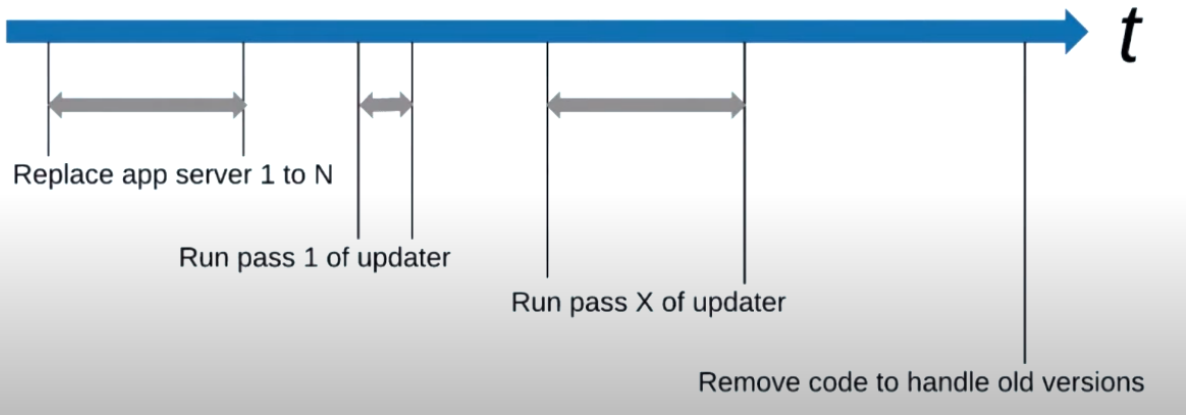
Application Lifecycle

- Modify Application
  - Can read/process all versions of documents
    - 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



VIDEOS 2:56 / 7:02

Timeline of the Migration



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 Cases Examples

- Every application that use a 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
- ✗ May need 2 indexes for same field while in migration period

m320 tree patterns v3

Who are the ancestors of node X?

Who reports to Y ?

Find all nodes that are under Z ?

Change all categories under N to under P

```
{
    "_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"
        },
        …
    ]
}
```

Model Tree Structures

Parent References

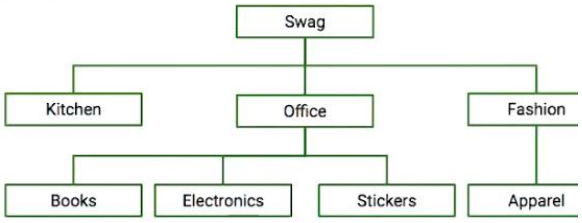
Child References

Array of Ancestors

Materialized Paths

Parent References

```
{
  name: "Office",
  parent: "Swag",
  ...
}
```



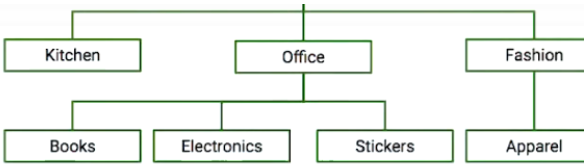
Who are the ancestors of node X?

```
// all ancestors

db.categories.aggregate([
{ $graphLookup: {
      from: 'categories',
      startWith: '$name',
      connectFromField: 'parent',
      connectToField: 'name',
      as: 'ancestors',
  }
}])
```

Parent References

```
{
  name: "Office",
  parent: "Swag",
  ...
}
```

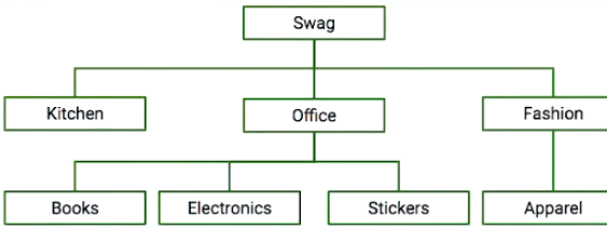


Who reports to Y?

```
// immediate ancestor

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

Parent References




```
{
  name: "Office",
  parent: "Swag",
  ...
}
```

Change all categories under N to under P

```
// all ancestors

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

Parent References

Who are the ancestors of node X? ❗

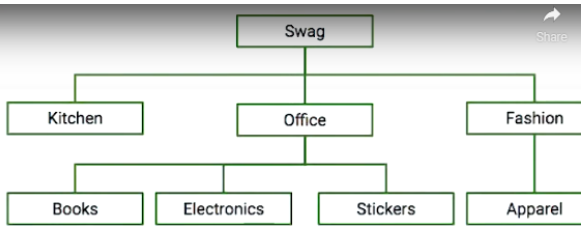
Who reports to Y ? ✔

Find all nodes that are under Z ? ❗

Change all categories under N to under P ✔

n320 tree patterns v3

Child References




```
{
  name: "Office",
  children: ["Books", "Electronics", "Stickers"]
  ...
}
```

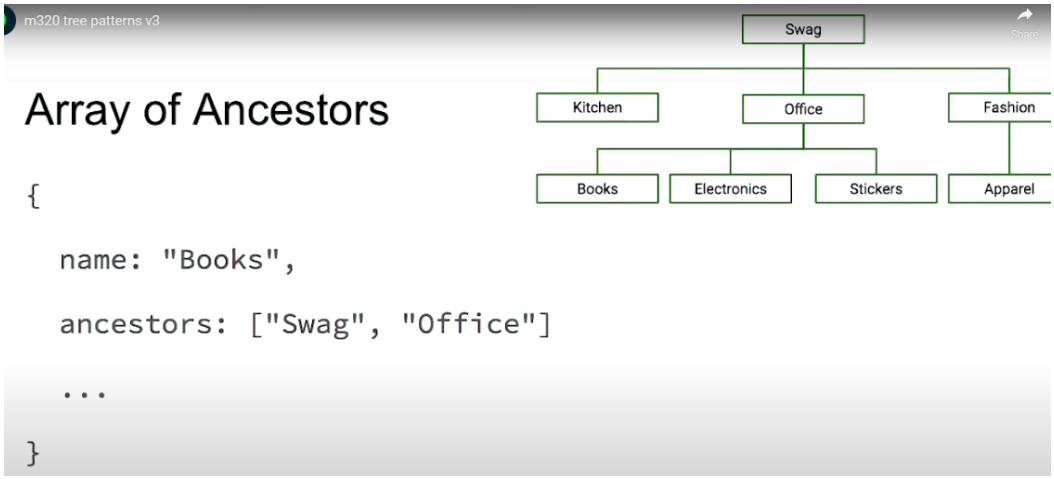
Child References

Who are the ancestors of node X? ❗

Who reports to Y ? ❗

Find all nodes that are under Z ? ✔

Change all categories under N to under P ❗



Array of Ancestors

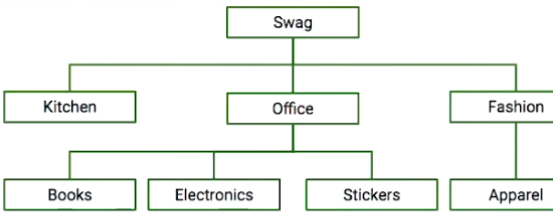
Who are the ancestors of node X? ✔

Who reports to Y ? ✔

Find all nodes that are under Z ? ❗

Change all categories under N to under P ❗

Materialized Paths




```
{
  name: "Books",
  ancestors: ".Swag.Office"
  ...
}
```

```
// immediate ancestor of Y
db.categories.find({ancestors: /\.Y$/})
// if descends from X and Z
db.categories.find({ancestors: /^\.X.*Y/i})
```

Materialized Paths

Who are the ancestors of node X? ✔

Who reports to Y ? ❗

Find all nodes that are under Z ? ❗

Change all categories under N to under P ❗

MongoMart Solution: Ancestor Array + Parent Reference

```
{
  "_id" : 8,
  "name" : "Umbrellas",
  "parent" : "Fashion",
  "ancestors" : [ "Swag", "Fashion" ]
}
```

MongoMart Solution: Ancestor Array + Parent Reference

```
{
  "_id" : 8,
  "name" : "Umbrellas",
  "parent" : "Fashion",
  "ancestors" : [ "Swag", "Fashion" ]
}
```

```
// Navigate up on the tree

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

// Find all nodes under a given category

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

Tree Patterns

Problem

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

Solution

- Different patterns
  - Child Reference
  - Parent Reference
  - Array of Ancestors
  - Materialized Paths

Use Cases Examples

- Org Charts
- Product Categories

Benefits and Trade-Offs

Child Reference

✅ Easy to navigate to children nodes or tree descending access patterns

Parent Reference

✅ Immediate parent node discovery and tree updates

Array of Ancestors

✅ Navigate upwards on the ancestors path

Materialized Path

✅ Makes use of regular expressions to find nodes in the tree

m320 Polymorphic Pattern2

Share

Vehicles

```
{
  "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"
}
```

Products

```
{
  "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"
}
```

Polymorphic Subdocuments

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"
  } ]
}
```

Single View Example

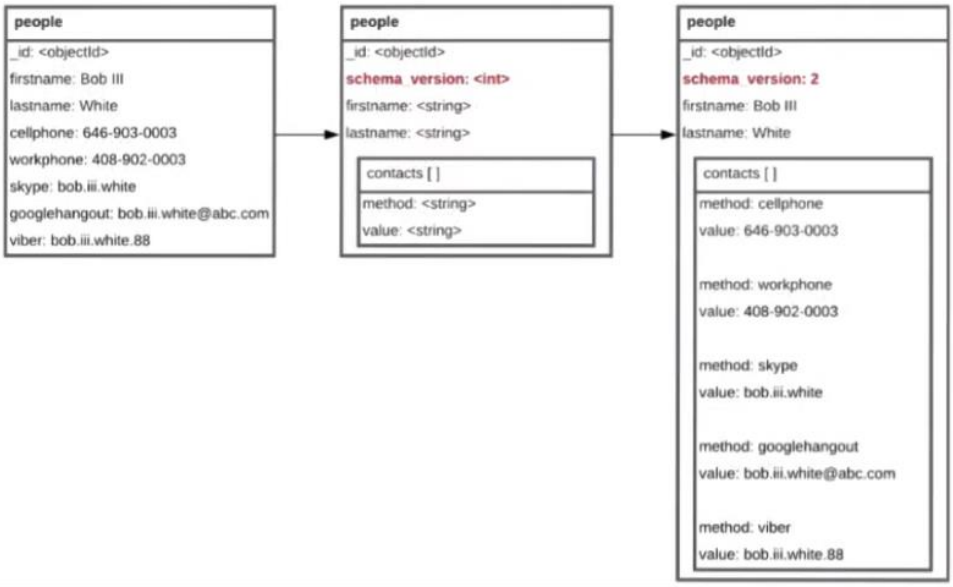
System A
```
{
  id: "203-102-1222",
  name: "Adams",
  first: "Samuel",
  address: "100 Forest",
  city: "Palo Alto",
  state: "California",
}
```

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" }
] }
```

Polymorphism in the Schema Versioning Pattern



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 Cases Examples

- Single View
- Product Catalog
- Content Management

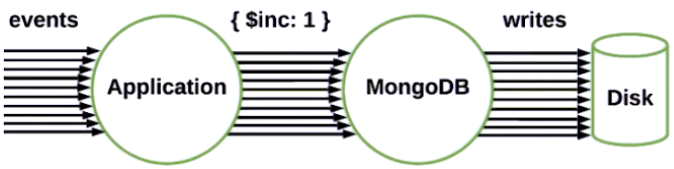
Benefits and Trade-Offs

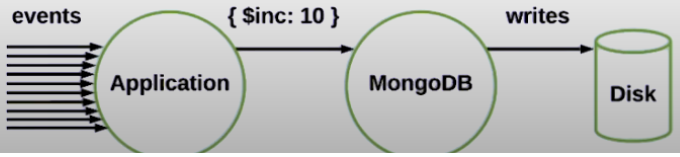
✅ Easier to implement
✅ Allow to query across a single collection

Additional Patterns

- Approximation Pattern
- Outlier Pattern

Approximation 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

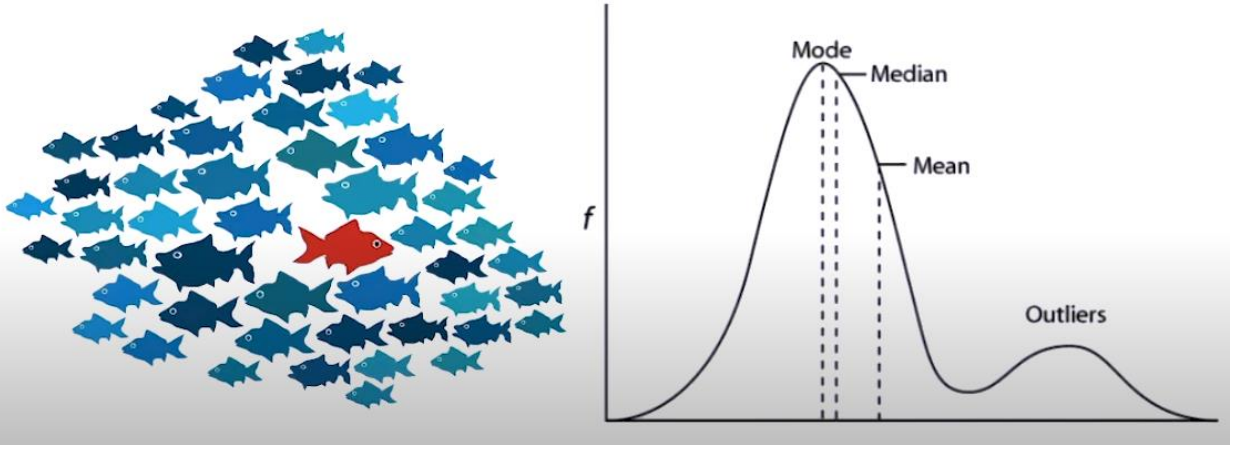
Use Cases Examples

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

Benefits and Trade-Offs

- ✅ Less writes
- ✅ Less contention on documents
- ✅ Statistically valid numbers
- ❌ Not exact numbers
- ❌ Must be implemented in the application

Outlier Pattern



Movies Site Solution

```
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
}
```

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 Cases 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

Recap:

These are some other notable patterns

- Approximation Pattern
  - avoiding performing an operation too often
- Outlier Pattern
  - keeping the focus on the most frequent use cases