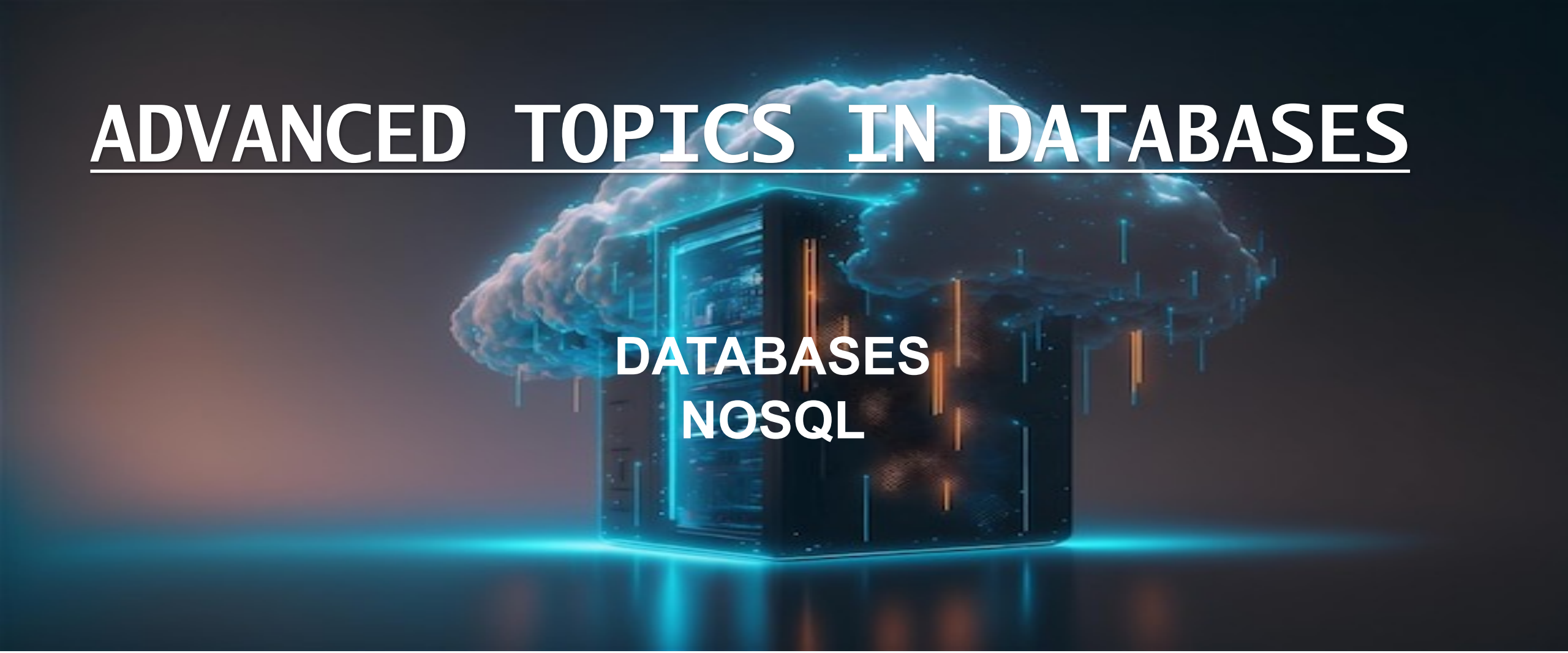
ADVANCED TOPICS IN DATABASES



DATABASES
NOSQL

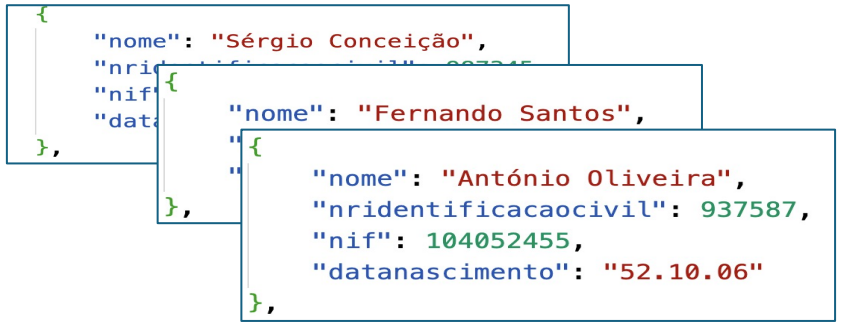
Master in Informatics Engineering
Data Engineering

Informatics Engineering Department

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MongoDB: Terminology

RDBMS	MongoDB
database instance	MongoDB instance
schema	database
table	collection
row	document
rowid	_id



collection

- each JSON document:
  - belongs to a collection
  - has a field _id
    - unique within the collection
- each collection:
  - belongs to a “database”

- http://www.mongodb.org/

MongoDB: Terminology

- The schema is dynamic
  - each document can contain different fields.
  - allows modelling unstructured and polymorphic data
- **Each document** contains **a key element**, on which can get a document univocally.
- **JSON documents**
  - BSON used for storing and accessing documents
- Documents (and their nested data) can be modified through a single sentence.

How to model data in MongoDB?

MongoDB: Terminology

- A good data model can:
  - Make it easier to manger data
  - Make queries more eficiente
  - Use less memory and CPU usage

- MongoDb implement a flexible document data model

Principle: Data is accessed together, should be stored together

Aggregate-Oriented

- An aggregate
  - A data unit with a complex structure
    - Not simply a tuple (a table row) like in RDBMS
  - A collection of related objects treated as a unit
    - unit for data manipulation and management of consistency

- Aggregates give the database information about which bits of data will be manipulated together (What should be stored on the same node)
- Minimize the number of nodes accessed during a search

The aggregates are related to the expected usage of the data

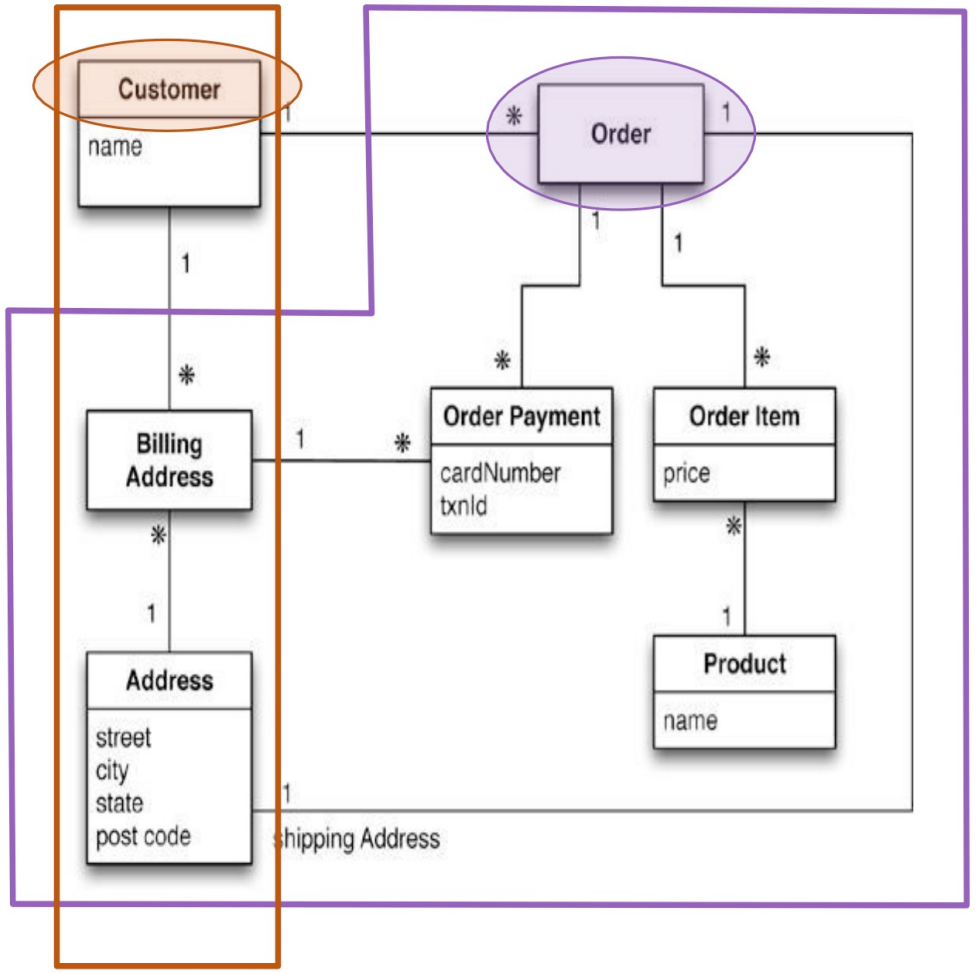
Aggregate Model – Documents

- **Each aggregate has a key** (ID)
- The **aggregate has a structure**
  - **Limited structures and types**
  - More access flexibility
    - Queries based on **aggregate attributes**
    - **Part of the aggregate can be recovered**
    - Indexes can be created by aggregate content

```
# Customer object
{
"customerId": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"payment": [
  {"type": "debit",
  "ccinfo": "1000-1000-1000-1000"}
  ]
}
```

```
# Order object
{
"orderId": 99,
"customerId": 1,
"orderDate":"Nov-20-2011",
"orderItems":[{"productId":27, "price": 32.45}],
"orderPayment":[{"ccinfo":"1000-1000-1000-1000",
        "txnId":"abelif879rft"}],
"shippingAddress":{"city":"Chicago"}
}
```

Aggregate Model – Example 1



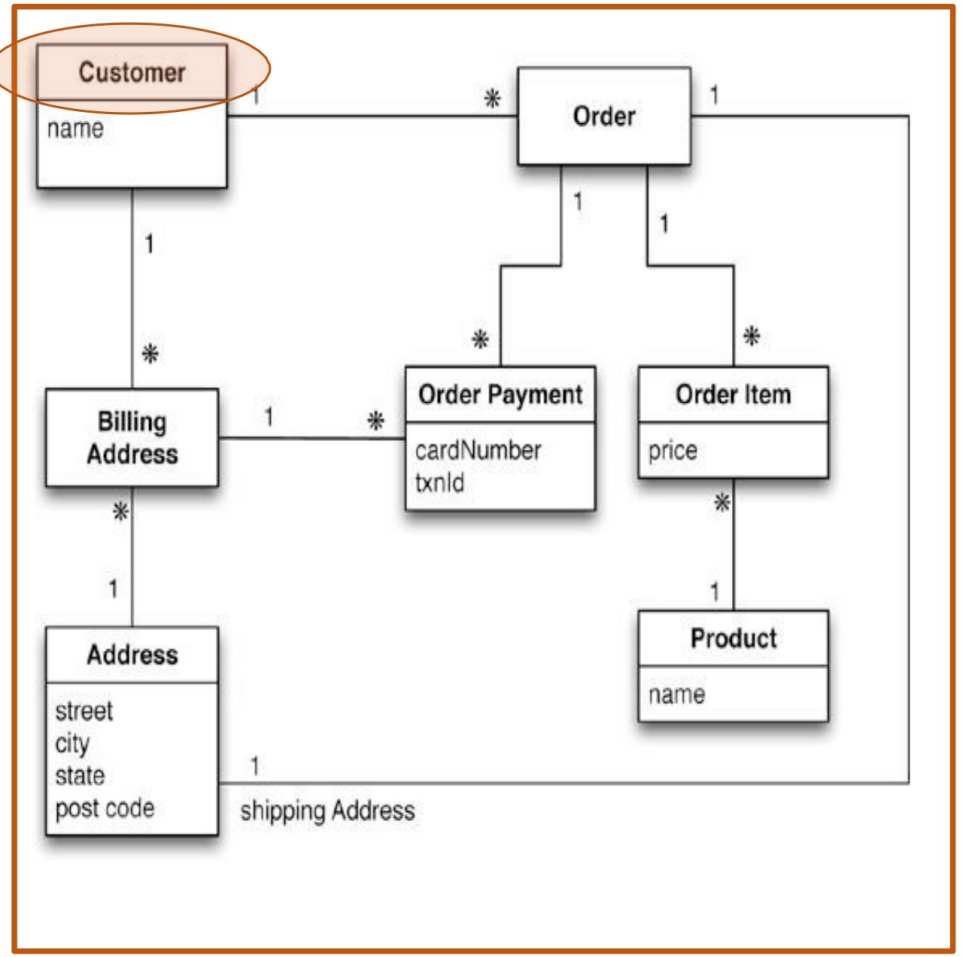

```
// (Single) Customer
{
    "id": 1,
    "name": "Fabio",
    "billingAddress": [
        {
            "street": "via Eco",
            "city": "Bari",
            "state": "IT",
            …
        },

        {
            "street": "via Ugo",
            "city": "Torino",
            "state": "IT",
            …
        },
    ]
}
```

```
//(Single) Order
{
    "id": 99,
    "customerId": 1,
    "orderItems": [
      {"productId": 27,
       "price": 34,
       "productName": "Data Mngm book"
       }, {…} ],
     "shippingAddress": {"city": "Bari", … },
     "orderPayment": [
          { "ccinfo": "100-432423-545-134",
            "txnId": "afdfsdfsd",
            "billingAddress":
                  {"city": "Bari", … }
           }, {…} ]
}
```

Aggregate Model – Example2

When a customer is accessed, all her/his orders are requested

```
// (Single) Customer
{
    "id": 1,
    "name": "Fabio",
    "billingAddresses": [
        {
            "city": "Bari", …
        }
    ]
    "orders": [
      {
      "id": 99,
      "customerId": 1,
      "orderItems": [
        {"productId": 27,
         "price": 34,
         "productName": "Data Mngm book"
           }, {…} ],
       "shippingAddress": {"city": "Bari", …
       "orderPayment": [
            { "ccinfo": "100-432423-545-134",
              "txnId": "afdfsdfsd",
              "billingAddress":
                  {"city": "Bari", … }
          }, {…} ]
       }, {…}, {…}, …]
}
```



Aggregate Model

The focus is on the unit(s) of interaction with the data storage

Pros:

It helps greatly when running on a cluster of nodes

The data of each “complex record” will be manipulated together, and thus should the stored on the same node

Cons:

An aggregate structure may help with some data interactions but be an obstacle for others

Types of Data Relationships

- One-to-one
  - A relationship where a data entity in one set is connect to exactly one data entity in another set
  - In mongoDb just a simple document

One-to-one

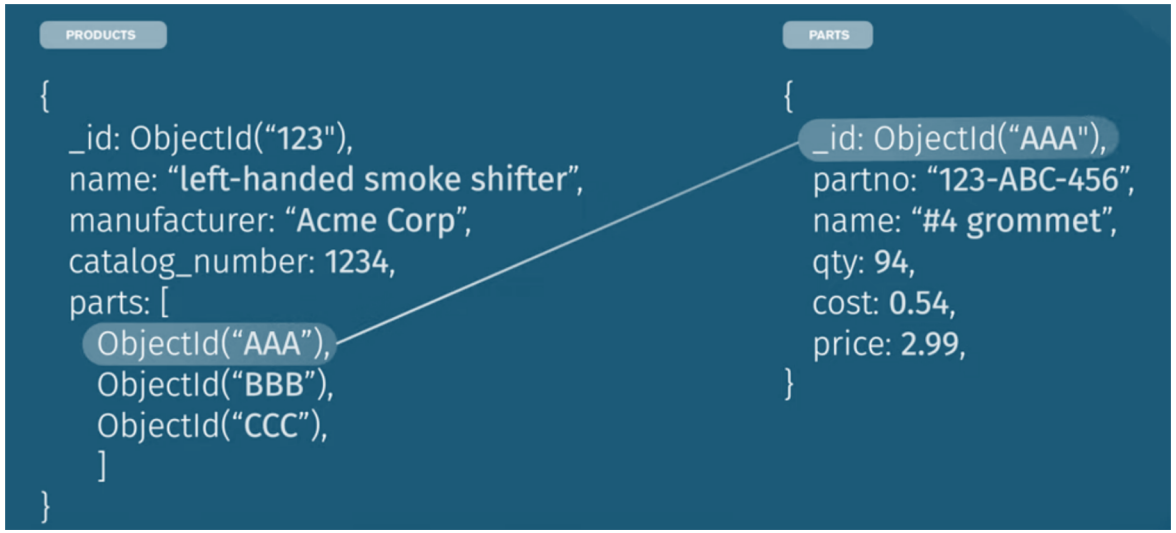
```
{
    "_id": ObjectId("573a1390f29313caabcd413b"),
    "title": "Star Wars: Episode IV - A New Hope",
    "director": "George Lucas",
    "runtime": 121
}
```

Types of Data Relationships

- One-to-many
  - A relationship where a data entity in one set is connect to any number of data entities in another set

One-to-many

```
{
    "_id": ObjectId("573a1390f29313caabcd413b"),
    "title": "Star Wars: Episode IV - A New Hope",
    "cast": [
        {"actor": "Mark Hamill", "character": "Luke Skywalker"},
        {"actor": "Harrison Ford", "character": "Han Solo"},
        {"actor": "Carrie Fisher", "character": "Princess Leia Organa"},
        ...
    ]
```



Types of Data Relationships

- Many-to-many
  - A relationship where any number of data entities in one set are connected to any number of data entities in another set

Embedding – we take related data and insert it into our document

Referencing – we refer to documents in another collection in our document

Embedding

```
{
    "_id": ObjectId("573a1390f29313caabcd413b"),
    "title": "Star Wars: Episode IV - A New Hope",
    "cast": [
        {"actor": "Mark Hamill", "character": "Luke Skywalker"},
        {"actor": "Harrison Ford", "character": "Han Solo"},
        {"actor": "Carrie Fisher", "character": "Princess Leia Organa"},
        ...
    ]
}
```

Referencing

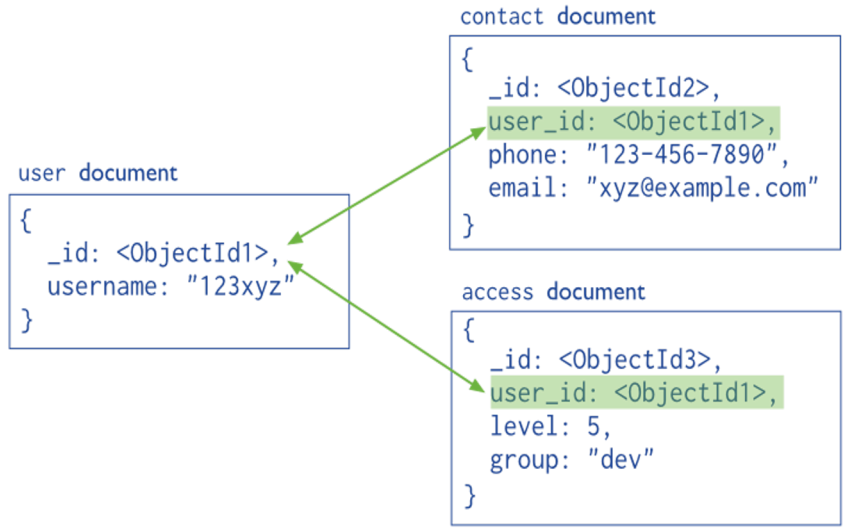
```
{
    "_id": ObjectId("573a1390f29313caabcd413b"),
    "title": "Star Wars: Episode IV - A New Hope",
    "director": "George Lucas",
    "runtime": 121,
    "filming_locations": [
        ObjectID("654a1420f29313fggbcd718"),
        ObjectID("654a1420f29313fggbcd719"),
        ...
```

Modelling Data

Embedded document Model

- Related data in a single document structure
- Documents can have subdocuments (in a field or array)
- Enable us to build complex relationships among data

Referring Model



- Links/references from one document to another
- Normalization of the schema

Embedding Data in Documents

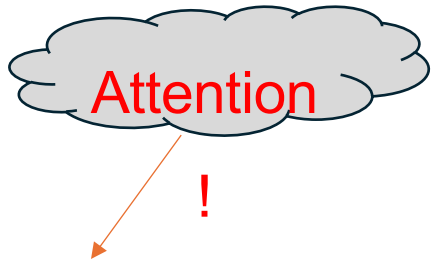
Denormalized schema

Advantages:

- Manipulate related data in a single operation
- Avoiding application joins
- Provides better performance for reads operations
- Allow developers write operation

Warning:

- Embedding data into a single documents **can create large documents**
  - Can result in a slow application performance for end user
- Continuously adding data without **limit creates unbounded documents**
- **Unbounded documents may exceed the BSON document threshold of 16 MB (we should avoid)**



Embedding Data in Documents

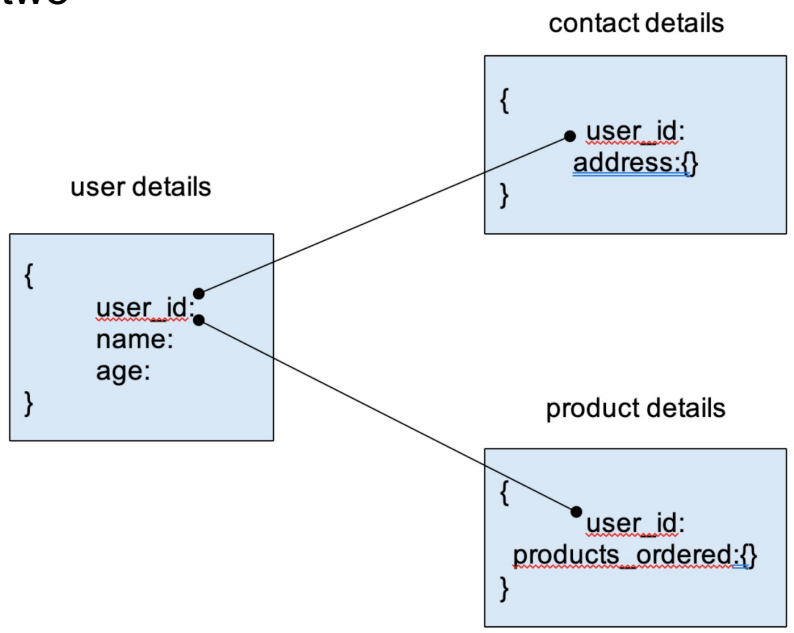
Conclusion:

- Store related data in single document
- Simplify queries' and improves overall a query performance
- Ideal for **one-to-one and one-to-many data**
- Has pitfalls like large documents and unbounded documents

```
{
   _id: POST_ID
   title: TITLE_OF_POST,
   description: POST_DESCRIPTION,
   by: POST_BY,
   url: URL_OF_POST,
   tags: [TAG1, TAG2, TAG3],
   likes: TOTAL_LIKES,
   comments: [
      {
         user:'COMMENT_BY',
         message: TEXT,
         dateCreated: DATE_TIME,
         like: LIKES
      },
      {
         user:'COMMENT_BY',
         message: TEXT,
         dateCreated: DATE_TIME,
         like: LIKES
      }
   ]
}
```

Referencing Data in Documents

- More flexibility than embedding
- save _id field of one document in another document as link between the two
- simple and sufficient form must use cases
- using referring is called linking or data normalization



Disadvantages:

- Can require more roundtrips to the server
  - Documents are accessed one by one
- Queries for multiple documents costs extra resources and impacts re
- Need to join data from multiple documents

RULES

Rule 1 – Embed unless there's a compelling reason not do.

Rule 2- Avoid Joins if they can be, but don’t be afraid if they can provide a better schema design

Rule 3 - Arrays should not grow without bound.

Rule 4 - Needing to access an object on its own is a compelling reason not to embed it.

Rule 5 - **DESIGN A SCHEMA BASED ON THE UNIQUE NEEDS OF YOUR APPLICATION**

Relational Model. Vs MongoDB Model

Users

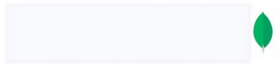
ID	first_name	surname	cell	city	location_x	location_y
1	Paul	Miller	447557505611	London	45.123	47.232

Professions

ID	user_id	profession
10	1	banking
11	1	finance
12	1	trader

Cars

ID	user_id	model	year
20	1	Bentley	1973
21	1	Rolls Royce	1965



```
{
   "first_name": "Paul",
   "surname": "Miller",
   "cell": "447557505611",
   "city": "London",
   "location": [45.123, 47.232],
   "profession": ["banking", "finance", "trader"],
   "cars": [
      {
         "model": "Bentley",
         "year": 1973
      },
      {
         "model": "Rolls Royce",
         "year": 1965
      }
   ]
}
```

Example 1

Which of the following actions can be made to improve this schema? Consider the following requirements:

- Preserve the one-to-one relationship among all the fields.
- Keep the contact and account information separate.
- Store data together that is accessed together

```
{
  "account_id": "MDB653115886",
  "account_holder": "Herminia Mckinney",
  "account_type": "savings",
  "balance": 6617.34,
  "street_num": 123,
  "street": "Main St",
  "city": "Tulsa",
  "state": "OK",
  "zip": 74008,
  "country": "USA",
  "home_phone": 1234567890,
  "cell_phone": 1111111111,
  "transfers": [
    ...
  ],
}
```

Hint: Remember that you can embed subdocuments and create separate collections.

Example 1 – Solution

Two solutions.

1- The first can be creating 2 collections, one for the **accounts** and one for the **client** aligns with the client's requirements.

- **It also ensures that data is accessed together and stored together**

2- Second solution - embedding the **contact numbers as a subdocument**

- can improve the layout and ensure that the data is accessed together and stored together

Example 2

Consider a banking database that **stores information on customers**, such **as their contact infor**mation, **accoun**t information, and **all account activity informati**on.

A **one-to-many** relationship **exists between the customer** and **their transaction**s.

Which of the following are valid ways **to represent this one-to-many relationship between the customer and their transactions?**

a) Create two collections: one collection for the customers' identifying information and one collection for the transactions. In the transactions collection, each document contains a customer's transactions and a field to reference the customer's information document.

b) Create a collection that contains documents for each customer. The document contains embedded subdocuments for the contact information and account information, as well as an array that holds all their transactions.

c) Embed the transactions for each customer as an array of subdocuments in the corresponding customer document.

Example 2

Consider a banking database that **stores information on customers**, such **as their contact infor**mation, **accoun**t information, and **all account activity informati**on.

A **one-to-many** relationship **exists between the customer** and **their transaction**s.

Which of the following are valid ways **to represent this one-to-many relationship between the customer and their transactions?** (Select all that apply)

a) Create two collections: one collection for the customers' identifying information and one collection for the transactions. In the transactions collection, each document contains a customer's transactions and a field to reference the customer's information document.

Correct - is the most common way to model 1:n relationships with referenced documents.

- Embedding the customer's information in a document within each transaction document leads to the repetition of the customer's information.
- To avoid duplication of customer information, use a reference and capture customer information in a separate collection. - **Transaction collection**

Example 2

b) Create a collection that contains documents for each customer. The document contains embedded subdocuments for the contact information and account information, as well as an array that holds all their transactions.

Incorrect - because we don't want to create unlimited arrays in our schema.

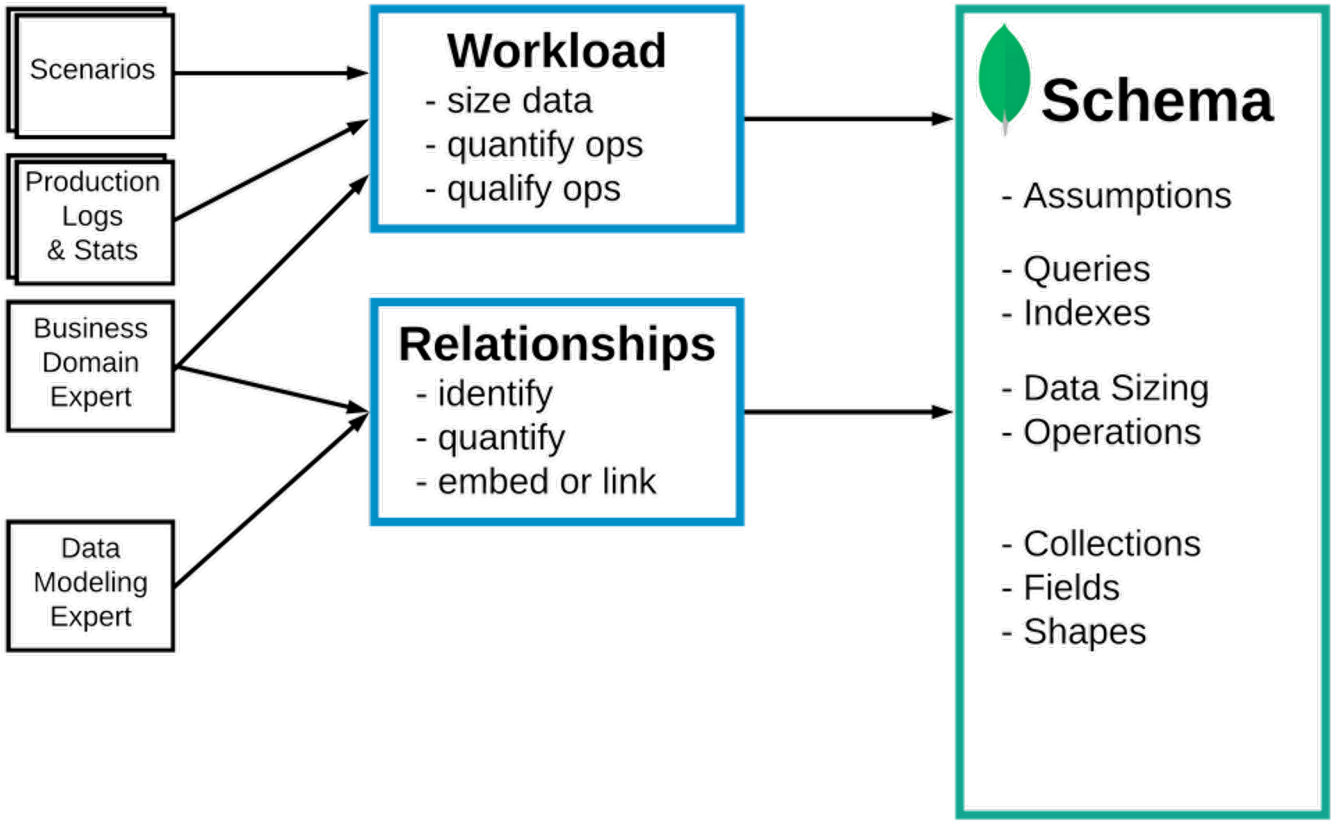
- The array containing all transactions can become an unlimited array

c) Embed the transactions for each customer as an array of subdocuments in the corresponding customer document.

Correct - This is a common way to model a one-to-many relationship with embedded documents

- Embedding connected data in a single document can reduce the number of read operations that are required to obtain data.

Methodology



Our first decision is whether we're going to model for greater schema simplicity or if we want better performance

Methodology

Define workload

- In this first phase it is very important to **understand what operations we are modelling,**
  - for this we must **measure the data** to be **stored**, **quantify and qualify the read and w**rite operations;
  - **list the most important operations,**
    - with metrics such as operations **per second**, **required latency** or **attributes us**ed in queries.
- To do this, we will start **from different usage scenarios of the applicatio**n, **the log**s and **statistics** that we have available or **the knowledge of business analysts**.
  - **Depending on the loa**d, it can cause **different modelling solutions**, since in some the reads may be more important and in others the writes.

Example - workload

Use case:

An organization has 100,000,000 weather sensors

The goal: to build a DB to store the data transmitted by all the sensors to perform statistics

Key data:

Number of devices : 100.000.000

Duration : 10 years

Analytics: 10 years

Assumptions:

In the case of predictions, hourly and per-minute data are equally valid

To drill down deeper into analytics, you need to keep the data for minutes

Detailed Operations

Actor: sensor

Description : Weather data sent to server

Type: write

Data: sensor_id, timestamp, sensor metrics

Frequence: 1.600.000 by seg.

== 100.000.000 by hour /60

Data size : 1000 bytes

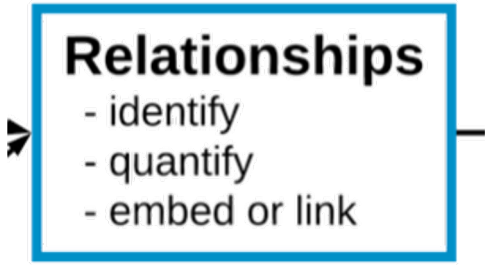
Durability: 10 years;

Example - workload

Actor	CRUD	Operations data	Type of operation	Ratio	Extra information
sensor	sending data Every minute	Sensor_id, metrics	write	1.666.667	One copy is stored (no need for redundancy), 1000 bytes of data read, 10-year lifetime
system	Identify inoperative sensors	Sensor_id Metric times	read by sec.	1 per hour	Latency and query time of 1 hour. Mediante un full scan de dados, The data is renews every hour
system	Aggregate data every hour	Sensor_id, time metrics	write	1 per hour	Redundancy on most nodes, 10-year lifetime
Data Analyst/Scientist	Run 10 analytic queries per hour	Temperatur e Metrics	read	100 per hour (10 per hour per analyst)	Latency and query time of 10 minutes, by means of a full data scan, the data is renewed every hour

Methodology

Relationships

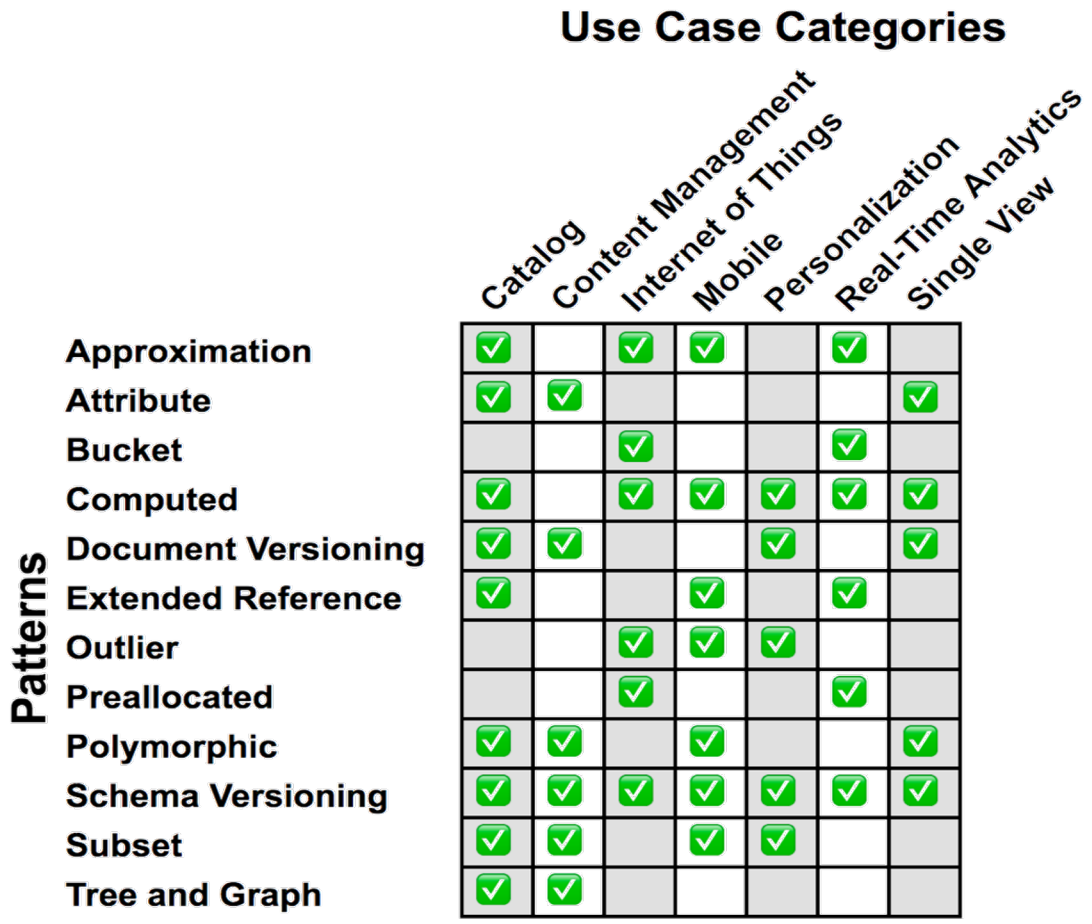


- Applications that use MongoDB use **two techniques to relate documents**:
  - **Create references**
    - $lookup operation to do the join**, or** we will make a second query for the second collection.
  - **Embed documents**
    - if within a document we store the data through sub-documents, either within an attribute or an array, we can obtain all the data through a single access, without the need for foreign keys or referential integrity checks.

Methodology

Patterns

- We can group the patterns into three categories:
  - **Representation:** attribute, document and schema versioning, polymorphic access
  - **Frequency:** subsets, approximation, Cross reference
  - **Grouping:** **c**omputed, bucket, outlier



Patterns

Attribute - used when we have a set of values separated between several fields that are semantically grouped.

```
{
    title: "Star Wars",
    director: "George Lucas",
    ...
    release_US: ISODate("1977-05-20T01:00:00+01:00"),
    release_France: ISODate("1977-10-19T01:00:00+01:00"),
    release_Italy: ISODate("1977-10-20T01:00:00+01:00"),
    release_UK: ISODate("1977-12-27T01:00:00+01:00"),
    ...
}
```

```
{
    title: "Star Wars",
    director: "George Lucas",
    …
    releases: [
        {
        location: "USA",
        date: ISODate("1977-05-20T01:00:00+01:00")
        },
        {
        location: "France",
        date: ISODate("1977-10-19T01:00:00+01:00")
        },
        {
        location: "Italy",
        date: ISODate("1977-10-20T01:00:00+01:00")
        },
        {
        location: "UK",
        date: ISODate("1977-12-27T01:00:00+01:00")
        },
        …
    ],
    …
}
```

Patterns

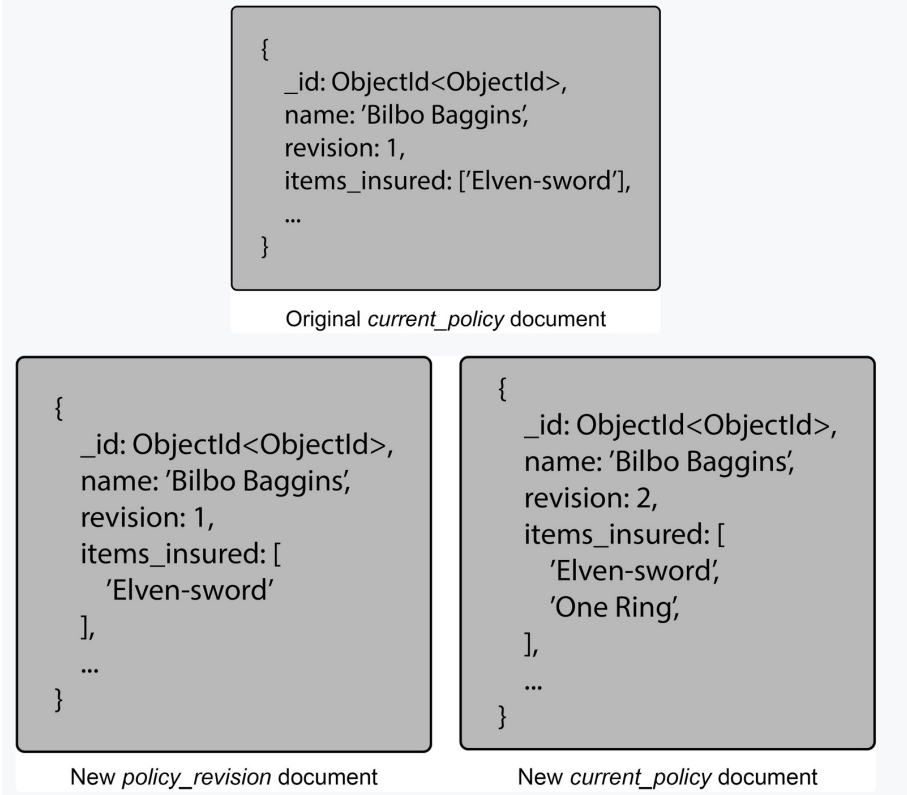
Polymorphic - is used when we have a set of documents that have more similarities than differences and we need them to be in a single collection.

```
{
    “sport”: “tennis”,
    “athlete_name”: “Martina Navratilova”,
    “career_earnings”: {value: NumberDecimal(“216226089”), currency: “USD”},
    “career_tournaments”: 390,
    “career_titles”: 167,
    “event”: [ {
        “type”: “singles”,
        “career_tournaments”: 390,
        “career_titles”: 167
    },
    {
        “type”: “doubles”,
        “career_tournaments”: 233,
        “career_titles”: 177,
        “partner”: [“Tomanova”, “Fernandez”, “Morozova”, “Evert”, ...]
    },
...
}
```

Polymorphic Sub-Documents

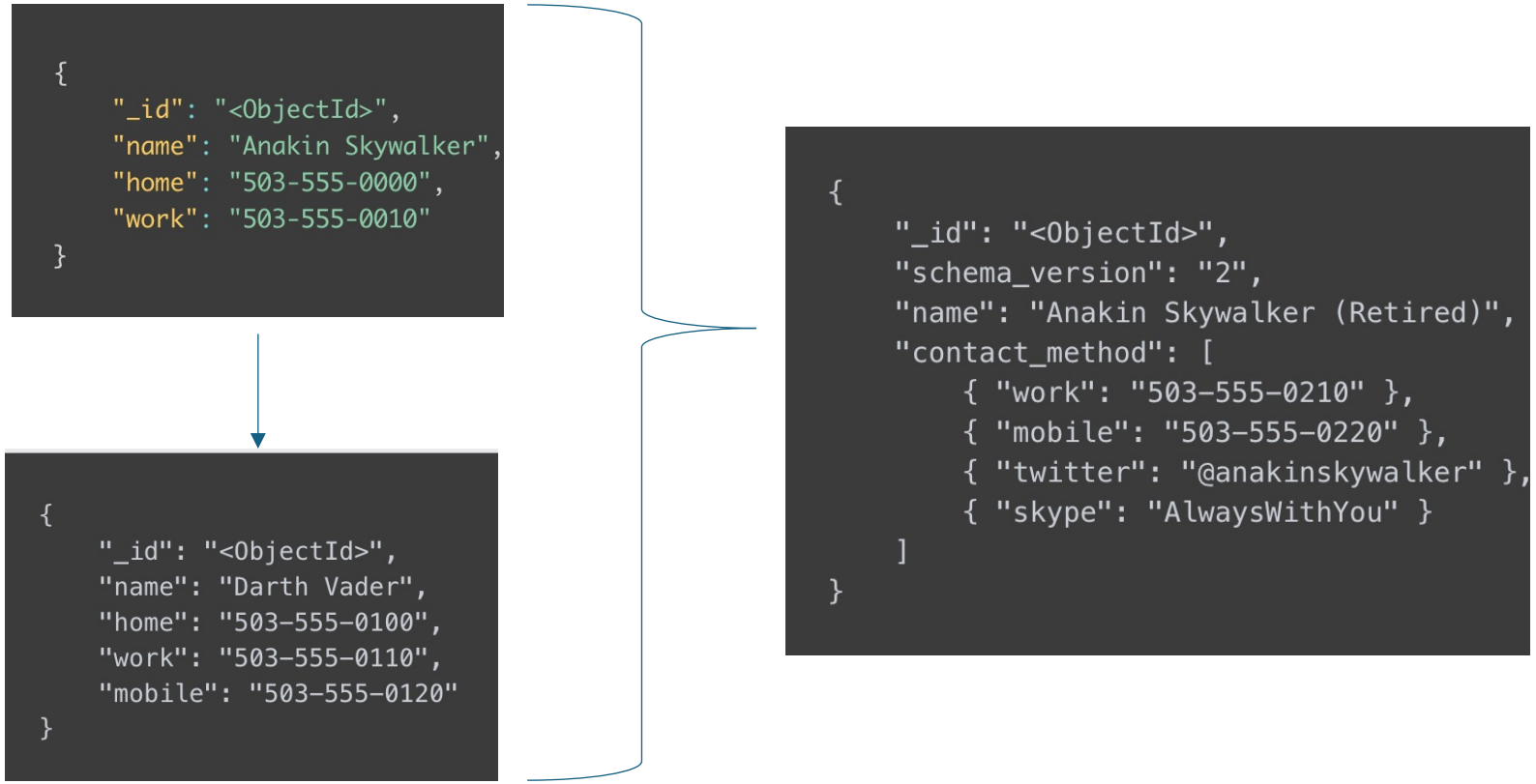
Patterns

Document versioning - used when we need to maintain different versions of a document, for example, to maintain a history.



Patterns

Schema versioning - it is used when we have different schemas for the same document, either due to an evolution of our model, due to the integration of external data, etc...



Patterns

Subset - is used when we have very large documents, with many attributes and containing collections of many documents that are not normally needed when retrieving a document.

```
{
    _id: ObjectId("507f1f77bcf86cd799439011"),
    name: "Super Widget",
    description: "This is the most useful item in your toolbox."
    price: { value: NumberDecimal("119.99"), currency: "USD" },
    reviews: [
          {
            review_id: 786,
            review_author: "Kristina",
            review_text: "This is indeed an amazing widget.",
            published_date: ISODate("2019-02-18")
          },
          {
            review_id: 785,
            review_author: "Trina",
            review_text: "Very nice product, slow shipping.",
            published_date: ISODate("2019-02-17")
          },
   ...
          {
            review_id: 1,
            review_author: "Hans",
            review_text: "Meh, it's okay.",
            published_date: ISODate("2017-12-06")
          }
    ]
}
```

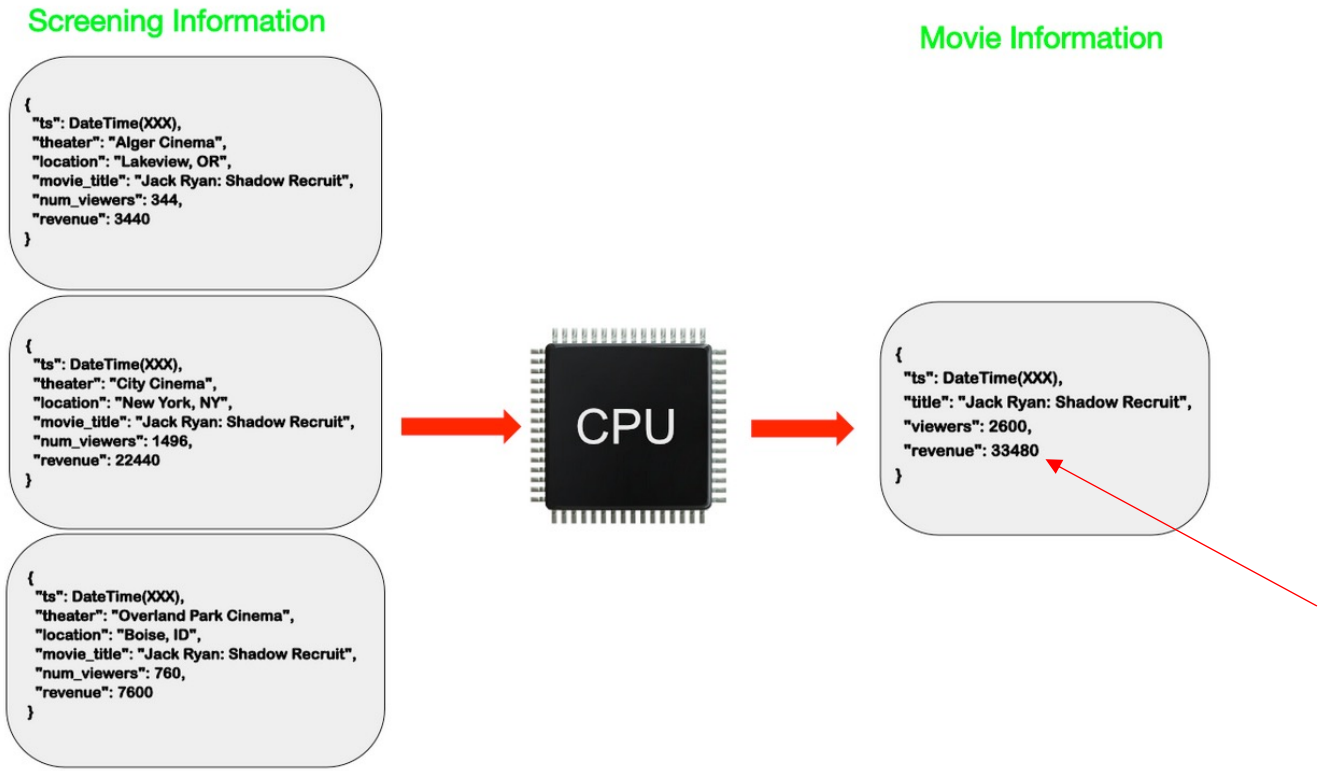
Patterns

Extended reference- is used when we see that we perform many joins to obtain all the information that the application usually accesses.

To do this, we will duplicate the data of the referenced documents.

Patterns

Computed - is used when we want to avoid having to recalculate data on each read. To do this, the calculated data is saved in a separate collection, so that when a new record arrives in the collection with the data, this value is recalculated and the appropriate document in the calculated data collection is modified.



Patterns

Bucket - When we work with IoT, data analytics, or time series it is normal to create a collection with a document for each measurement taken.

```
{
    sensor_id: 12345,
    start_date: ISODate("2019-01-31T10:00:00.000Z"),
    end_date: ISODate("2019-01-31T10:59:59.000Z"),
    measurements: [
       {
       timestamp: ISODate("2019-01-31T10:00:00.000Z"),
       temperature: 40
       },
       {
       timestamp: ISODate("2019-01-31T10:01:00.000Z"),
       temperature: 40
       },
       …
       {
       timestamp: ISODate("2019-01-31T10:42:00.000Z"),
       temperature: 42
       }
    ],
   transaction_count: 42,
   sum_temperature: 2413
}
```

We can group these documents by hours, days, etc... which we can place in collections with quarterly, annual data, etc...

Patterns

Tree - used to represent a hierarchy (tree)

```
{
  _id: <ObjectId1>,
  name: "Samsung 860 EVO 1 TB Internal",
  part_no:  "MZ-76E1T0B ",
  price: {
     value: NumberDecimal( "169.99  "),
     currency:  "USD"
  },
  parent_category:  "Solid State Drives",
  ancestor_categories: [
     "Solid State Drives",
     "Hard Drives",
     "Storage",
     "Computers",
     "Electronics"
  ]
}
```

Preallocation Pattern - simply dictates to create an initial empty structure to be filled later.

Patterns

An example MongoDB data model using various design patterns

items

```
_id : ObjectId,
schema : int,
sku : str,
name : str,
price : decimal,
description : str,
sold_at : [ str ],
tot_rating : int,
num_ratings: int,

top_reviews : [
  { name : str,
    rating : int,
    review : str
  }
]
categories : [ str ]
```

stores

```
_id : ObjectId,
schema : int,
name : str,
address: {
  number : str,
  street : str,
  city : str,
  postal_code : str
},
items_in_stock: [ str ]

staff: [
  {
    role : str,
    name : int,
    id : ObjectId
    contact_info:
     {
       mobile : str,
       email : str
     }
  }
]
```

reviews

```
_id : ObjectId,
schema : int,
start_date : date,
end_date : date,
sku : str,
reviews : [
  {
    timestamp : date,
    username : str,
    rating : int,
    review : str
  }
]
sum_reviews : int,
num_reviews : int
```

Patterns Used:

- Schema Versioning
- Subset
- Computed
- Bucket
- Extended Reference

Exercise

We are required to design the database for the management of an insurance application.

Each customer is characterized by the name, surname, birthdate, the contact methods and the home address.

The contact methods can be, for example, the telephone number, the email, a Skype username, etc. The address is characterized by the street name, street number, city, province and region.

Each insurance policy is signed by a customer and is characterized by its type, the date of signature, and the list of included items. The included items can be modified by the customer over time, by adding new ones or removing undesired ones.

The application should efficiently retrieve the currently active insurance data. However, previous policy versions must be available on request.

In addition to the insurance details, it is necessary to show only the name, surname and the contact methods of the customer.

Indicate for each collection in the database the document structure and the strategies used for modelling.

Exercise

We are required to design the database for the management of an insurance application.

Each **customer** is characterized by the name, surname, birthdate, the contact methods and the home address.

The **contact methods** can be, for example, the **telephone number, the email, a Skype username, etc.** The **address** is characterized by the **street name, street number, city, province and region.**

Each insurance policy is signed by a customer and is characterized by its **type, the date of signature,** and the **list** of included **items**. The included items can be modified by the customer over time, by adding new ones or removing undesired ones.

The application should efficiently retrieve the currently active insurance data. However, previous policy versions must be available on request.

In addition to the **insurance details**, it is necessary to show only the name, surname and the contact methods of the customer.

Indicate for each collection in the database the document structure and the strategies used for modelling.

SOLUTION

Insurance policy (latest version only)

```
{_id: <ObjectId>,
 type: <string>,
 date: <date>,
 customer: {user: <ObjectId>,
            name: <string>,
            surname: <string>,
            contacts: {email: <string>,
                       mobile: <string>,
                       skype: <string>}
           },
 items: [<string>],
 version: <number>
}
```

Insurance_rev (previous policy versions)

```
{_id: < ObjectId >,
 type: <string>,
 date: <date>,
 customer: {user: <ObjectId>,
            name: <string>,
            surname: <string>,
            contacts: {email:<string>,
                       mobile: <string>}
           },
 items: [<string>],
 version: <number>
}
```

Customer

```
{_id: <ObjectId>,
 name: <string>,
 surname: <string>,
 contacts: {email: <string>,
            mobile: <string>,
            skype: <string>},
 birthdate: <date>,
 address: {
          street: <string>,
          number: <string>,
          city: <string>,
          province: <string>,
          region: <string>
}
```

Document versioning for the updates of insurance policies

Extended reference for the customer information

Polymorphic pattern to track only the contact methods that are available

References

- Sadalage, P. J., & Fowler, M. (2012). NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional, 192 p.
- MongoDB Manual: http://docs.mongodb.org/manual/

- uma força motriz na aparência do seu esquema é qual é o padrão de acesso aos dados para esses dados

- http://jackmyers.info/db/videos/Data%20Modeling%20with%20MongoDB.pdf