**«Books and publishers»**

• Each publisher is characterized by the name, the year of foundation and the country. A list of contact methods is available. The possible contact methods are: email, telephone, website. At most one contact for each method can be recorded. Additional methods might be added in the future.

• Each book is identified by the ISBN code and characterized by the title, the subtitle, the number of pages, the language and the publication date. Each book is published by only one publisher. A publisher distributes several books.

• The design should be optimized to display for each book all its information and the publisher's name

**Publisher**

{\_id: <objectId>,

name: <str>,

year:<date>,

country: <str> ,

contacts: {email: <str> , tel: <str> , website: <str> }

}

**Polymorphic pattern** to track only the contact methods that are available

**book**

{\_id: <objectId>, // ISBN

title: <str> ,

subtitle: <str> ,

n: <int>,

language: <str> ,

pub\_date:<date>

publisher:

{\_id: <objectId>,

name: <str>}

}

**Extended reference pattern** to display the name of the publisher. The application can access the book collection to show both the book information and the name of the publisher. In this way we avoid a lookup to the publisher collection.

**«Blog website»:**

• Each user subscribed to the blog is identified by a username. We want to keep track of user contact methods, such as email, phone, Twitter account, etc. The contact methods tracked by the platform may change over the time. A user might have more contacts of the same method. Each contact method can have a status, e.g., “active”, “inactive”, “confirmed”, etc.

• Blog posts are characterized by the title, the description and the author (i.e., username). The posts receive comments from users. Each comment is characterized by the text, the user who published it and its timestamp.

• The application should display for each post only the top 10 most recent comments. The user profile should also include the total number of posts and comments written by the user.

**user**

{\_id: , // username

contacts:[ {k: , // contact method

v: , // contact value

s: , // status ],

nPosts: , nComments: }

**Attribute pattern** to track the contact methods. The embedded documents in the contact fields consists of the k field which represents the contact method, the v field providing the corresponding value, and the s field for the status

**Computed pattern** to store the number of posts and comments published by the user.

**post**

{\_id: , title: , description: ,

author: , // username

comments: [ // most recent comments

{\_id: user: , text: , timestamp: } ] }

**Subset pattern** to track only the most recent comments.

**Comment**

{\_id: user: , text: , timestamp: }

**«Breadcrumbs»**

• Each page of the webisite is characterized by the relative path, the title, the body, the creation timestamp, the last update timestamp and a list of tags.

• The pages are organized in a tree strcture. The home page is the root. All other pages are in sub paths from the root.

• The application should build the breadcrumbs code for each page. The breadcrumbs are an ordered list of links containing all the ancestors of the current page back to the home page, i.e., the root. o Indicate for each collection in the database the document structure and the strategies used for modelling.

**Page**

{\_id: < ObjectId >,

path: , title: ,

body: , creation: , update: ,

tags: [ ],

parent: {\_id: , path: }

ancestors: [ {\_id: , // parent path: },

{\_id: , // grandpa path: }, … ] }

**Tree pattern** to track the ancestors. The parent field is added to use, if necessary, the $lookup and $graphlookup functions.

For each ancestor, the **extended reference pattern** is exploited to include the path in the embedded document and avoid join operations.

**«Museum exhibitions»**

•We want to design the database to manage museum exhibitions according to the following requirements. •Museums are characterized by their name, address, a telephone number, and a website (if available). The address consists of geographical coordinates, street name and number, postal code, and city. •The items exhibited in the museums are identified by a progressive number and characterized by a title, a description and the list of author names. The items are categorized as either archaeological finds, or paintings, or sculptures. •The database must record all the main features of each item, such as its dimensions (i.e., width, height, weight, etc.). Each feature has at least a name and a value, and possibly a unit of measure. For instance, the main material is a feature of an archaeological find, the geometrical sizes are features of a painting. For each item, the museum to which it belongs must be recorded, with the museum name frequently accessed together with the item itself.

•Several exhibitions are hosted in each museum. The exhibition is characterized by a title, a description, the list of curator names. You must record all the items associated with each exhibition; they can be in the order of hundreds. An item can be part of different exhibitions. Moreover, each exhibition can be hosted by several museums in different periods. You must record the start and end dates of each exhibition in each museum. •Given an item, the database must be designed to efficiently provide the name of the museum that owns it. •Given an exhibition, the database must be designed to efficiently provide the name of the museum and the geographical coordinates where it has been hosted. •Furthermore, given an exhibition, the list of items included in the exhibition and their number must be efficiently returned.

**Museums**

{\_id: ObjectId(), name: ,

address: { street: , number: ,

postal\_code: , city: , province: ,

geo\_ref: { type: , coordinates: [ ]

} },

phone: , website: // optional }

**Items**

{ \_id: ,

title: , description: ,

authors: [ ],

category: ,

features: [ {k: , v: , u: } ],

museum:

{ \_id: ObjectId(), name: } }

**Polymorphic pattern** to track the museum information in the museum collection.

**Attribute pattern (with polymorphic pattern)** to track the features of each item.

**Extended reference pattern** to track the museum information associated with each item

**Exhibitions**

{\_id: ObjectId(), title: , description:,

curators: [ ],

events: [

{start: ,

end: ,

museum:{ \_id: itemId(),

name: ,

geo\_ref: { type: , coordinates: [ ] }

} ],

items: [ ] // \_id of items } }

**Bucket pattern** with extended reference pattern to track when an exhibition is hosted in a museum.

**Parcel delivery**

•Design a MongoDB database to manage the parcel delivery according to the following requirements. •Customers of the parcel delivery service are citizens identified by their social security number. They can be senders or recipients of delivered parcels. They are characterized by their name, surname, email address, a telephone number, and by different addresses, one for each type (e.g., one billing address, one home address, one work address, etc.). Each address consists of street name, street number, postal code, city, province, and country. •Parcels are characterized by a unique barcode and their physical dimensions (specifically: width, height, depth, and weight). All widths, heights, and depths are always expressed in meters. All weights are always expressed in kilograms. •The recipient and the sender information required to deliver each parcel must be always available when accessing the data of a parcel. Recipient and sender information required to deliver a parcel consists of full name, street name, street number, postal code, city, province, and country.

•The parcel warehouse is divided into different areas. Each area is identified by a unique code, e.g., 'area\_51' and consists of different lines. Each line is identified by unique code, e.g., 'line\_12', and hosts several racks. Each rack is identified by unique code, e.g., 'rack\_33', and is made up of shelves. Each parcel is placed on a specific shelf of the warehouse, identified by a unique code, e.g., 'shelf\_99'. The database is required to track the location of each parcel within the warehouse.

•Given a parcel, the database must be designed to efficiently provide its full location, from the shelf, up to the area, through rack and line.

•Given a customer, the database must be designed to efficiently provide all her/his parcels as a sender, and all his/her parcels as a recipient. oIndicate for each collection in the database the document structure and the strategies used for modelling

**Parcel**

{\_id: , // barcode

dimensions: { // **also 1st-level attribute is fine**

width: , height: , depth: , weight: },

recipient: { \_id: , // e.g., SSN, id of the customer street: ,

number: , zip\_code: , city: , province: },

sender:{ \_id: ,// e.g., SSN, \_id of the customer street: ,

number: , zip\_code: , city: , province: },

father\_pos: , // code of area/lane/rack/shelf

locations: [ // list of codes of area/lane/rack/shelf ] }

**Extended reference** pattern for recipient and sender address information. The recipient and sender \_ids are required to look up all parcels of a given customer

**Tree pattern** for the position. The full list of treepattern ancestors is required. The parent ancestor of the tree-pattern is optional

**Customers**

{\_id: , // fiscal code name: , surname: , email: , tel: ,

addresses:

{ ‘home’: { street\_name: , street\_num: , postal\_code: , city: , province: , country: },

‘billing’:{ street\_name: , street\_num: , postal\_code: , city: , province: , country: },

‘work’: {...} }

}

**Attribute pattern** (optional) for the addresses attribute.

**BOOK**

**book**

{\_id: ‘1234 -5678’, // ISBN title: ‘Book Title’,

subtitle: ‘Subtitle of the book’, pages: 123,

lang: ‘EN’, published: 2021 -01 -31

prices: [ { format: ‘ebook’,

country: ‘UK’,

amount: 12.34,

currency: ‘£’ },

{ format: ‘paperback’,

country: ‘US’,

amount: 23.45,

currency: ‘$’ } ],

categories: [‘fantasy’,‘classical’]

ancestors: [‘fiction’, ‘children’] }

**Attribute pattern** to track the price according to the book format and its country

**Tree pattern** to track all the ancestors in the category hierarchy

**Property Rental Reviews**

We are required to design a MongoDB database for the management of a website similar to AirBnb where property rental reviews from users are collected. Each property is characterized by a name, the price per night, and its location (postal code, city, and country). The price per night is characterized by its amount and its currency. Each review is characterized by the timestamp, the text of the review, the grade and the author information, and it is related to only one property. When accessing a review, only the name and the email of the author who wrote the review are displayed. When accessing a property, only the top 10 most recent reviews are displayed, each presenting only its text, grade and timestamp. Each property page reports the number of total reviews received, and the average grade

**Properties**

{\_id: , name: ,

price: { amount: , currency: },

location: { postal\_code: , city: , country: , },

reviews:[ ***// top 10 most recent***

{review\_id: text: , timestamp: , grade: } ],

nReviews: , avgGrade: }

**Reviews**

{\_id: , property\_id: , text: , timestamp: ,

grade: , author: { name: , email: } }

**Subset pattern** to track only the most recent reviews of each property.

**Extended reference pattern** for the properties, to display the relevant information of the reviews without joins.

**Computed pattern** to store the overall number of reviews and their average grade.

**Insurance**

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.

**Insurance policy (latest version only)**

{\_id: , type: , date: ,

customer: {user: , name: , surname: ,

contacts: {email: , mobile: , skype: } },

items: [],

version: }

**Insurance\_rev (previous policy versions)**

{\_id: < ObjectId >, type: , date: ,

customer: {user: , name: , surname: ,

contacts: {email:, mobile: } },

items: [],

version: }

**Customer**

{\_id: , name: , surname: , birthdate: ,

contacts: {email: , mobile: , skype: },

address: { street: , number: , city: , province: , region: } }

**Document versioning** for the updates of insurance policies

**Polymorphic pattern** to track only the contact methods that are available

**Extended reference** for the customer information

We are required to design a MongoDB database to store the power consumption data sent by sensors. Each sensor is identified by a code and it is characterized by its brand and the release date. The system is required to store the sensor specs (e.g., battery life, weight). For each spec the system should store, if available, the unity of measure.Each sensor send a measure each minute. The measures are characterized by the timestamp and the power consumption measured.

**Sensor collection** {“\_id”: 10001,  
“brand”: “SensIT”,  
“release\_date”: ISODate( “2018-04-26”),   
“specs”:[  
              { “k”: “battery\_life”, “v”: 24, "u": "hours"},  
              { “k”: “height”, “v”: 9, “u”: “mm”},  
              { “k”: “width”, “v”: 12, “u”: “mm”},             …]}

**Attribute pattern** to track all the specs of each sensor with the corresponding unity of measure.

**Measures collection**{“\_id”: ObjectID(…),

“id\_sensor”: 10001,

“start”: ISODate("2019-12-14T10:00:00.000Z"),

“end”: ISODate("2019-12-14T10:59:59.000Z"),

“measures”: [ { “timestamp”: ISODate("2019-12-14T10:00:00.000Z"), “power”: 100 }, …

{“timestamp”: ISODate("2019-12-14T10:39:00.000Z"), “power”: 30 } ],

“n\_measures”: 39}

**Bucket pattern** to track all the measures of each sensor. Each document collect the measures sent by one sensor in one hour (fixed bucket).

**BOOK**

Each book is identified by the ISBN code and it is characterized by its title, subtitle, the number of pages, the language (‘IT’, ‘EN’, ‘FR’, etc.) and the publication date.

The system is required to store the book prices (e.g., 12.34$), for each country and for each format (e.g., ‘ebook’, ‘paperback’). Please note that book prices are searched based on their currency (e.g., all book prices in dollars $, or pounds £), and separately also on their amount (e.g., higher than 10.0).

Each book belongs to one or more categories (e.g., ‘fantasy’,‘classical’). Categories are organized in a hierarchical structure (e.g., ‘children’ category includes ‘fiction’, which includes ‘fantasy’,‘classical’, etc.).

**BOOK**

{\_id: ‘1234-5678’,  // ISBN  
 title: ‘Book Title’,  
 subtitle: ‘Subtitle of the book’,  
 pages: 123,  
 lang: ‘EN’,  
 published: 2021-01-31  
 prices: [ // attibute pattern  
 { format: ‘ebook’, country: ‘UK’,  amount: 12.34, currency: ‘£’  },  
          {format: ‘paperback’, country: ‘US’,  amount: 23.45, currency: ‘$’ }  
         ],  
categories: [‘fantasy’,‘classical’]  
ancestors:  [‘fiction’, ‘children’] // tree pattern }

**Attribute pattern** to track the price according to the book format and its country

Tree pattern to track all the ancestors in the category hierarchy

The data to be displayed on the review website for each hotel include the hotel name, the number of stars, and the list of provided services, e.g., free wifi, baby parking, pet allowed, etc. For each hotel, the venue information and its top 10 reviews must be always shown. The venue information consists of the address, the city, and the country. Furthermore, the official website address might be included in the venue information. Each review consists of a timestamp, a score (e.g., 4.5), the nickname of its author, the number of “likes”, and a textual description. Each review is related to one specific hotel. Given a hotel, the database must be designed to efficiently provide all the data describing the hotel, its top 10 reviews (those having the highest numbers of “likes”), its total number of reviews, and their average score. Instead, given a review, the database must efficiently provide the hotel name, its number of stars and its city

**Hotel**

{ \_id: ObjectId(), name: , stars: , services: [],

venue: { address: , city: , country: , website: <url>},

top\_reviews: [ {\_id: ObjectId(), timestamp: , score: , nickname: , likes: , description: } ],

tot\_reviews: ,

avg\_score: }

**Review**

{\_id: ObjectId(), timestamp: , score: , nickname: , likes: , description: ,

hotel: { \_id: ObjectId(), name: , stars: , city: } }

**Polymorphic pattern** to track the venue information in the hotel collection (due to the optional website info).

**Subset** **pattern** to track the top 10 reviews for each hotel.

**Computed pattern** for the average score and total review count of each hotel.

**Extended reference** for the review collection to show the hotel info

Design a MongoDB database to store offers of merchants for a mobile app according to the following requirements. Data to display for each merchant includes the merchant's name, type of merchant (e.g., restaurant, hotel, supermarket), a textual description, the venue, the average rating given by customers, and the contact information. The venue consists of the full address, the city, ZIP code, and country. Contact information includes the phone number and email address. The official website address and the Facebook page might be included in the contact information. Each offer consists of a title, a textual description, a list of categories (e.g., wellness, food, wine), a price in euros, and the validity period (i.e., start and end dates). Each offer is related to a specific merchant. A merchant can have many offers. Given a merchant, the database must be designed to efficiently provide all the data describing the merchant, the number of available offers, and their price range (i.e., min and max prices). Instead, given an offer, the database must efficiently provide the merchant name, the merchant type and its venue information.

**MERCHANT**

{ \_id: ObjectId(), name: , type: , description: , rating: , venue: { address: , city: zipcode: , country: },

contact: { phone: , email: , website: facebook: },

available\_offers: , price\_range:{ min: , max: } }

**OFFER**

{ \_id: ObjectId(), title: , description: , category: [],

price: , start: , end: ,

merchant: { \_id: ObjectId(), name: , type: ,

venue: { address: , city: , zipcode: , country: } } }

**Polymorphic pattern** to track the contact information in the merchant collection (due to the optional website and facebook info).

**Computed pattern** for the total number of offers and the price range available for each merchant. Extended reference for the offers collection to show the merchant info.

Design a MongoDB database to manage online courses according to the following requirements. Teachers are characterized by their name, surname, email, and list of subjects they can teach (e.g., maths, electronics, etc.). Each teacher can have one or more online profiles on different platforms (e.g., Facebook, LinkedIn, Wikipedia, etc.). For each online profile, if available, the database tracks the corresponding URL of the profile (e.g., https://en.wikipedia.org/wiki/Ranjitsinh\_Disale). Note that for each teacher and each platform, at most one profile can exist. A teacher can teach different courses. The courses are characterized by a name, a syllabus, a list of keywords, and the teacher. Each course has several editions. For each edition, the start date, the end date, and the number of enrolled students are known. Given a course, the database must be designed to efficiently provide the name, the surname and the email of its teacher. Furthermore, given a course, the number of editions and the average number of enrolled students in each edition must be efficiently returned. Teachers are typically retrieved by subject (e.g., all those teaching maths), and by online profile platform (e.g., all those having a wikipedia page).

**Teacher**

{ \_id: ObjectId(), name: , surname: , email: ,

profiles:

{ facebook: , linkedin: , …. }

subjects: [ ] }

**Course**

{ \_id: ObjectId(), name: , syllabus:, keywords: [ ],

teacher: { \_id: ObjectId(), name: , surname: , email: , }

editions: [ {start: , end: , n\_students: ]

n\_editions: , tot\_students: }

**Polymorphic pattern** to track the online profile information in the Teacher collection**.**

**Extended reference pattern** to track the teacher information associated with each course.

**Bucket pattern** to track when a course is provided. Computed pattern for average students on each edition.

Design a MongoDB database to manage museum exhibitions according to the following requirements. Museums are characterized by their name, address, a telephone number, and a website (if available). The address consists of geographical coordinates, street name and number, postal code, and city. The items exhibited in the museums are identified by a progressive number and characterized by a title, a description and the list of author names. The items are categorized as either archaeological finds, or paintings, or sculptures. The database must record all the main features of each item, such as its dimensions (i.e., width, height, weight, etc.). Each feature has at least a name and a value, and possibly a unit of measure. For instance, the main material is a feature of an archaeological find, the geometrical sizes are features of a painting. For each item, the museum to which it belongs must be recorded, with the museum name frequently accessed together with the item itself. Several exhibitions are hosted in each museum. The exhibition is characterized by a title, a description, the list of curator names. You must record all the items associated with each exhibition, they can be in the order of hundreds. An item can be part of different exhibitions. Moreover, each exhibition can be hosted by several museums in different periods. You must record the start and end dates of each exhibition in each museum. Given an item, the database must be designed to efficiently provide the name of the museum that owns it. Given an exhibition, the database must be designed to efficiently provide the name of the museum and the geographical coordinates where it has been hosted. Furthermore, given an exhibition, the list of items included in the exhibition and their number must be efficiently returned.

**Museum**

{ \_id: ObjectId(), name: ,

address:

{ street: , number: , postal\_code: , city:, province: ,

geo\_ref: {type: , coordinates: [ ] } }

tel: , website: // optional }

**Items**

{ \_id: , title: , description:,

authors: [ ], category: ,

features: [ {k: , v: , u: } ],

museum: { \_id: itemId(), name: } }

**Exhibition**

{ \_id: ObjectId(), title: , description:,

curators: [ ],

events: [ {start: , end: ,

museum:{ \_id: itemId(), name: ,

geo\_ref: {type: , coordinates: [ ] } } ],

items: [ ] // \_id of items }

**Polymorphic pattern** to track the website information in the museum collection.

**Attribute pattern** (with polymorphic pattern) to track the features of each item